TCS

Notes Compiled By Prof.: GANESH SIR

Notes SET-I

Sem – V

(COMP)

CHOPRA ACADEMY

Degree & Diploma

<table>
<thead>
<tr>
<th>Bandra (W)</th>
<th>Thane (W)</th>
<th>Vashi</th>
</tr>
</thead>
<tbody>
<tr>
<td>201 &amp; 305, Hill Road, Rizvi Chambers, Mumbai :400050</td>
<td>Off No 1 &amp; 2Gnd Flr, Gangadhar Apt, RS Rd,Data Mandir Thane - 400601</td>
<td>350 C- Wing Sector – 17 Vashi Plaza, Navi-Mumbai</td>
</tr>
<tr>
<td>Call : 2655 9366. 9967 843 854</td>
<td>Call : 6579 4001. 9967 843 853</td>
<td>Call : 26421927 9967 843 852</td>
</tr>
</tbody>
</table>

Office Timing: 10.30 a.m to 7.30 p.m (Monday to Saturday) 10.30 a.m to 3.30 pm (Sunday)
About GANESH SIR:

Sir has completed his B.E(Comp) from Mumbai university, M.E (Comp-IT) from Pune University. He was working for Infosys and Oracle for 3 years. Currently he is pursuing his Ph.D in Theory of Computer Science (TCS). He has expertise in Model checking and Formal Method, which are the advance part of TCS.

Dear Students,

Welcome to the world of Theory in computation. For any doubts you can contact me at 83 84 82 05 75 between 9:00 pm to 1:00 am. I will also be posting important question and exam solution on my blog before the examination.
**Syllabus:**

<table>
<thead>
<tr>
<th>Chapter No.</th>
<th>Chapter Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Regular expressions (RE)</td>
<td>Definition, FA and RE, RE to FA, FA to RE, algebraic laws for RE, applications of REs, Regular grammars and FA, FA for regular grammar, Regular grammar for FA.</td>
</tr>
<tr>
<td>3</td>
<td>Proving languages to be non–regular</td>
<td>Pumping Lemma, and its applications. Some closure properties of Regular languages – Closure under Boolean operations, reversal homomorphism, inverse homomorphism, etc. Mhill–Nerode Theorem.</td>
</tr>
<tr>
<td>4</td>
<td>DFA Minimization</td>
<td>Some decision properties of Regular languages – emptiness, finiteness, membership, equivalence of two DFAs or REs, Finite automata with output.</td>
</tr>
</tbody>
</table>
| **5** | **Context-free Grammars (CFGs)** | Formal definition, sentential forms, leftmost and rightmost derivations, the language of a CFG. Derivation tree or Parse tree—Definition, Relationship between parse trees and derivations.

Parsing and ambiguity, Applications of CFGs, Ambiguity in grammars and Languages. Simplification of CFGs – Removing useless symbols, epsilon–Productions, and unit productions, Normal forms –CNF and GNF. Proving that some languages are not context free –Pumping lemma for CFLs, applications. Some closure Properties of CFLs – Closure under union, concatenation, Kleene closure, substitution, Inverse homomorphism, reversal, intersection with regular set, etc. Some more decision properties of CFLs, Review of some undecidable CFL problems. |
| **6** | **Pushdown Automata (PDA)** | Formal definition, behaviour and graphical notation, Instantaneous descriptions (1ds), The language of PDA (acceptance by final state and empty stack). Equivalence of acceptance by final state and empty stack, Equivalence of PDAs and CFGs, CFG to PDA, PDA to CFG. DPDAs – Definition, DPDAs and Regular Languages, DPDAs, Multistack DPDAs & NPDAs and CFLs. Languages of DPDAs, NPDAs, and ambiguous grammars. |
| **7** | **Turing Machines** | Formal definition and behaviour, Transition diagrams, Language of a TM, TM as accepters deciding and generators. TM as a computer of integer functions, Design of TMs, Programming techniques for TMs – Storage in state, multiple tracks, subroutines, etc. Universal TMs, Variants of TMs – Multitape TMs, |
| **Nondeterministic TMs.** TMs with semi–infinite tapes, Multistack machines, Simulating TM by computer, Simulating a Computer by a TM, Equivalence of the various variants with the basic model. Recursive and recursively enumerable languages, Properties of recursive and recursively enumerable languages, A language that is non-recursively enumerable (the diagonalization language). The universal language, Undecidability of the universal language, The Halting problem, Rice’s Theorem, Greibach Theorem, Post’s Correspondence Problem (PCP) – Definition, Undecidability of PCP. Context sensitive language and linear bounded automata. Chomsky hierarchy. |
| **8** | **Intractable Problems** | The classes P and NP, an NP–complete problem, A Restricted Satisfiability problem, Additional NP–complete problems, Complements of languages in NP, Problems Solvable in polynomial space, A problem that is complete for PS, Language Classes based on randomization, The complexity of primality testing. |
Subject Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 1

Finite Automata/Finite State Machine

History of Automata:

- Automata theory is the study of abstract computing devices, or "machines."

- Abstract machine - A procedure for executing a set of instructions in some formal language, possibly also taking in input data and producing output. Such abstract machines are not intended to be constructed as hardware but are used in thought experiments about computability.

- Before there were computers, in the 1930's, A. Turing studied an abstract machine that had all the capabilities of today's computers, at least as far as in what they could compute.

- Turing's goal as to describe precisely the boundary between what a computing machine could do and what it could not do.

- In the 1940's and 1950's, simpler kinds of machines, which we today call "finite automata," were studied by a number of researchers.

- These automata, originally proposed to model brain function, turned out to be extremely useful for a variety of other purposes.

- Also in the late 1950’s, the linguist N. Chomsky began the study of formal "grammars."

- While not strictly machines, these grammars have close relationships to abstract automata and serve today as the basis of some important software components, including parts of compilers.

- In 1969, S. Cook extended Turing's study of what could and what could not be computed.
• Cook was able to separate those problems that cannot be solved efficiently by computer from those problems that can in principle be solved, but in practice take so much time that computers are useless for all but very small instances of the problem.

• The latter class of problems is called "intractable," or "NP-hard." It is highly unlikely that even the exponential improvement in computing speed that computer hardware has been following ("Moore's Law") will have significant impact on our ability to solve large instances of intractable problems.

• Any of these theoretical developments bear directly on what computer scientists do today.

• Some of the concepts, like finite automata and certain kinds of formal grammars, are used in the design and construction of important kinds of software.

• Other concepts, like the Turing machine, help us understand what we can expect from our software.

• Especially, the theory of intractable problems lets us deduce whether we are likely to be able to meet a problem "head-on" and write a program to solve it (because it is not in the intractable class), or whether we have to find some way to work around the intractable problem: find an approximation, use a heuristic, or use some other method to limit the amount of time the program will spend solving the problem.

Why Study Automata Theory?

There are several reasons why the study of automata and complexity is an important part of the core of Computer Science.

Introduction to Finite Automata:

Definition: A finite automaton has a set of states, and its "control" moves from state to state in response to external "inputs".
 Finite automata are a useful model for many important kinds of hardware and software. We shall see some examples of how the concepts are used. For the moment, let us just list some of the most important kinds:

1. Software for designing and checking the behaviour of digital circuits.

2. The "lexical analyzer" of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuation.

3. Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases, or other patterns.

4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

There are many systems or components, such as those enumerated above, that may be viewed as being at all times in one of a finite number of "states."

The purpose of a state is to remember the relevant portion of the system's history.

Since there are only a finite number of states, the entire history generally cannot be remembered, so the system must be designed carefully, to remember what is important and forget what is not.

The advantage of having only a finite number of states is that we can implement the system with a fixed set of resources.

For example, we could implement it in hardware as a circuit, or as a simple form of program that can make decisions looking only at a limited amount of data or using the position in the code itself to make the decision.

Example 1: Perhaps the simplest nontrivial finite automaton is an on/off switch. The device remembers whether it is in the "on" state or the "off" state, and it allows the user to press a button whose effect is different, depending on the state of the switch. That is, if the switch is in the off state, then pressing the button changes it to the on state.
state, and if the switch is in the on state, then pressing the same button turns it to the off state.

![Diagram of a finite automaton modeling an on/off switch](image)

- The finite-automaton model for the switch is shown in Fig.

- As for all finite automata, the states are represented by circles; in this example, we have named the states on and off. Arcs between states are labeled by "inputs," which represent external influences on the system. Here, both arcs are labeled by the input Push, which represents a user pushing the button. The intent of the two arcs is that whichever state the system is in, when the Push input is received it goes to the other state.

- One of the states is designated the "start state," the state in which the system is placed initially. In our example, the start state is off, and we conventionally indicate the start state by the word Start and an arrow leading to that state.

- It is often necessary to indicate one or more states as "final" or "accepting" states. Entering one of these states after a sequence of inputs indicates that the input sequence is good in some way. For instance, we could have regarded the state on in Fig. as accepting, because in that state, the device being controlled by the switch will operate. It is conventional to designate accepting states by a double circle, although we have not made any such designation in Fig.

- Example 2: Sometimes, what is remembered by a state can be much more complex than an on/off choice. Figure, shows another finite automaton that could be part of a lexical analyzer.
The job of this automaton is to recognize the keyword then. It thus needs five states, each of which represents a different position in the word then that has been reached so far.

These positions correspond to the prefixes of the word, ranging from the empty string (i.e., nothing of the word has been seen so far) to the complete word.

![Finite automaton diagram]

**Figure 1.2: A finite automaton modeling recognition of then**

In Fig, the five states are named by the prefix of then seen so far. Inputs correspond to letters. We may imagine that the lexical analyzer examines one character of the program that it is compiling at a time, and the next character to be examined is the input to the automaton.

The start state corresponds to the empty string, and each state has a transition on the next letter of then to the state that corresponds to the next-larger prefix. The state named then is entered when the input has spelled the word then.

Since it is the job of this automaton to recognize when then has been seen, we could consider that state the lone accepting state.

### Automata and Complexity:

- Automata are essential for the study of the limits of computation. As we mentioned in the introduction to the chapter, there are two important issues:

1. **What can a computer do at all?** This study is called "decidability," and the problems that can be solved by computer are called "decidable."
2. **What can a computer do efficiently?** This study is called "intractability," and the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input are called "tractable." Often, we take all polynomial functions to be "slowly growing," while functions that grow faster than any polynomial are deemed to grow too fast.

### The Central Concepts of Automata Theory

In this section we shall introduce the most important definitions of terms that pervade the theory of automata. These concepts include the "alphabet" (a set of symbols), "strings" (a list of symbols from an alphabet), and "language" (a set of strings from the same alphabet).

#### 1. Alphabets

An *alphabet* is a finite, nonempty set of symbols. Conventionally, we use the symbol $\Sigma$ for an alphabet. Common alphabets include:

- $\Sigma = \{0, 1\}$, the *binary* alphabet.
- $\Sigma = \{a, b, ..., z\}$, the set of all lower-case letters.
- The set of all ASCII characters or the set of all printable ASCII characters.

#### 2. Strings

A string (or sometimes word) is a finite sequence of symbols chosen from some alphabet. For example, 01101 is a string from the binary alphabet $\Sigma = \{0, 1\}$.

The string 111 is another string chosen from this alphabet.

#### 3. The Empty String:
The empty string is the string with zero occurrences of symbols. This string, denoted \( \varepsilon \) is a string that may be chosen from any alphabet whatsoever.

### 4. Length of a String
- It is often useful to classify strings by their length, that is, the number of positions for symbols in the string.
- For instance, 01101 has length 5. It is common to say that the length of a string is "the number of symbols" in the string; this statement is colloquially accepted but not strictly correct.
- Thus, there are only two symbols, 0 and 1, in the string 01101, but there are five positions for symbols, and its length is 5.
- However, you should generally expect that "the number of symbols" can be used when "number of positions" is meant.
- The standard notation for the length of a string \( w \) is \( |w| \). For example,

\[ |011| = 3 \text{ and } |\varepsilon| = 0. \]

### 5. Powers of an Alphabet
- If \( \Sigma \) is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define \( \Sigma^k \) to be the set of strings of length \( k \), each of whose symbols is in \( E \).

- Example: Note that \( \Sigma^0 = \{ \varepsilon \} \), regardless of what alphabet \( \Sigma \) is. That is, \( \varepsilon \) is the only string whose length is 0.

\[ \Sigma = \{ 0, 1 \}, \text{ then } \Sigma^1 = \{ 0, 1 \}, \Sigma^2 = \{ 00, 01, 10, 11 \}, \Sigma^3 = \{ 000, 001, 010, 011, 100, 101, 110, 111 \} \text{ and so on.} \]

- Note that there is a slight confusion between \( \Sigma \) and \( \Sigma^1 \). The former is an alphabet; its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1.
The set of all strings over an alphabet $\Sigma$ is conventionally denoted $\Sigma^*$. For instance, $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11,000, \ldots\}$. Put another way, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$.

Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty strings from alphabet $\Sigma$ is denoted $\Sigma^+$. Thus, two appropriate equivalences are:

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots$

$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

### 6. Concatenation of Strings

- Let $x$ and $y$ be strings. Then $xy$ denotes the concatenation of $x$ and $y$, that is, the string formed by making a copy of $x$ and following it by a copy of $y$.
- More precisely, if $x$ is the string composed of $i$ symbols $x = a_1a_2\ldots a_i$ and $y$ is the string composed of $j$ symbols $y = b_1b_2\ldots b_j$, then $xy$ is the string of length $i+j$ and $xy = a_1a_2\ldots a_ib_1b_2\ldots b_j$.
- Example: Let $x = 01101$ and $y = 110$. Then $xy = 01101110$ and $yx = 11001101$.
- For any string $w$, the equations $\varepsilon w = w = w \varepsilon$ hold. That is, $\varepsilon$ is the identity for concatenation, since when concatenated with any string it yields the other string as a result (analogously to the way $0$, the identity for addition, can be added to any number $x$ and yields $x$ as a result).

### Languages:

- A set of strings all of which are chosen from some $\Sigma^*$, where $\Sigma$ is a particular alphabet, is called a language.
If $\Sigma$ is an alphabet, and $L \subseteq \Sigma^*$, then $L$ is a language over $\Sigma$.

Notice that a language over $\Sigma$ need not include strings with all the symbols of $\Sigma$, so once we have established that $L$ is a language over $\Sigma$, we also know it is a language over any alphabet that is a superset of $\Sigma$.

The choice of the term "language" may seem strange. However, common languages can be viewed as sets of strings.

An example is English, where the collection of legal English words is a set of strings over the alphabet that consists of all the letters.

Another example is C, or any other programming language, where the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters. The exact alphabet may differ slightly among different programming languages, but generally includes the upper- and lower-case letters, the digits, punctuation, and mathematical symbols.

However, there are also many other languages that appear when we study automata. Some are abstract examples, such as:

1. The language of all strings consisting of $n$ 0's followed by $n$ 1's, for some $n \geq 0$
   
   $L = \{ \varepsilon, 01, 0011, 000111, \ldots \}$

2. The set of strings of 0's and 1's with an equal number of each:
   
   $L = \{ \varepsilon, 01, 10, 0011, 0101, 1001, \ldots \}$

3. The set of binary numbers whose value is a prime:
   
   $L = \{ 10, 11, 101, 111, 1011, \ldots \}$

4. $L^*$ is a language for any alphabet $\Sigma$.

5. $\emptyset$, the empty language, is a language over any alphabet.

6. $\{ \varepsilon \}$, the language consisting of only the empty string, is also a language over any alphabet. Notice that $\emptyset \neq \{ \varepsilon \}$; the former has no strings and the latter has one string.

The only important constraint on what can be a language is that all alphabets are finite. Thus languages, although they can have an infinite number of strings, are restricted to consist of strings drawn from one fixed, finite alphabet.
Set-Formers as a Way to Define Languages

- It is common to describe a language using a "set-former":
  \[ \{ \omega \mid \text{something about } \omega \} \]

- This expression is read "the set of words \( \omega \) such that (whatever is said about \( \omega \) to the right of the vertical bar)."

Examples are:

1. \( \{ \omega \mid \omega \text{ consists of an equal number of 0's and 1's} \} \).

2. \( \{ \omega \mid \omega \text{ is a binary integer that is prime} \} \).

3. \( \{ \omega \mid \omega \text{ is a syntactically correct C program} \} \).

- It is also common to replace \( \omega \) by some expression with parameters and describe the strings in the language by stating conditions on the parameters. Here are some examples; the first with parameter \( n \), the second with parameters \( i \) and \( j \):

1. \( \{ 0^n 1^n \mid n \geq 1 \} \). Read "the set of 0 to then 1 to then such that \( n \) is greater than or equal to 1," this language consists of the strings \( \{01, 0011, 000111, \ldots \} \).

2. \( \{ 0^i 1^j \mid 0 \leq i \leq j \} \). This language consists of strings with some 0's (possibly none) followed by at least as many 1's.

Problems:

- In automata theory, a problem is the question of deciding whether a given string is a member of some particular language.

- It turns out, as we shall see, that anything we more colloquially call a "problem" can be expressed as membership in a language. More precisely, if \( \Sigma \) is an alphabet, and \( L \) is a language over \( \Sigma \), then the problem \( L \) is:

  Given a string \( \omega \) in \( \Sigma^* \), decide whether or not \( \omega \) is in \( L \).
Design steps for Finite Automata: Refer class notes.

Model of a Finite Automata/ Finite State Machine (FSM):

The Model consists of three main parts, as follows:

1. Input Tape
2. Reading Head
3. Finite Control

1. **Input Tape:**
   a. The input tape is divided into squares, each square containing the input symbol from the input alphabet $\Sigma$.
   
   b. The end square of the tape contain the endmarker $\$$ at the left end and the endmarker $\$$ at the right end.
c. The absence of endmarkers indicates that the tape is of infinite length. The left to right sequence of symbol between the two endmarkers is the input string to be processed.

2. **Reading Head:**
   a. The head examines only one square at a time and can move one square either to left or to the right.
   
   b. We restrict the movement of the Reading head only to the right side. The head read the input data from the input tape and give it to the Finite Control.

3. **Finite Control:**
   a. Finite Control has the actual logic of the model.
   
   b. The input to the finite control will usually be the symbol under the Reading Head, say ‘∑i’ and the present state of the machine say ‘Qi’ then the next state of the machine will be given by δ(Qi, ∑i) → Qi.

**Operation of Finite Automata/ Finite State Machine is given below:**

1. Input string is fed to the machine through a tape. Tape is divided into squares and each square contains an input symbol.
2. The main machine is shown as a box called as the Finite Control. The finite control has the mathematical logic of the model.
3. Initially the machine is in the starting state (q0). Reading head is placed at the leftmost square of the tape.
4. At regular intervals, the machine reads one symbol from the tape and then enters a new state. Transition to a state depends only on the current state and the input symbol.
$\delta(Q_i, \Sigma) \rightarrow Q_j$

Machine transits from $Q_i$ to $Q_j$ on input $\Sigma_i$.

5. After reading an input symbol, the reading head moves right to the next square.

6. This process is repeated again and again, i.e.
   a. A symbol is read.
   b. State of the machine (finite control) changes.
   c. Reading head moves to the right.

7. Eventually, the reading head reaches the end of the input string. The last cell in the input tape is represented by $\$. \$

8. Now, the automaton has to say ‘yes’ or ‘no’. If the machine ends up in one of a set of final states($q_1$) then the answer is ‘yes’ otherwise the answer is ‘no’.

Types of Finite Automata:

- As was mentioned earlier, a finite automaton has a set of states, and its "control" moves from state to state in response to external "inputs."

- One of the crucial distinctions among classes of finite automata is whether that control is "deterministic," meaning that the automaton cannot be in more than one state at any one time, or "nondeterministic," meaning that it may be in several states at once.

- Types of Finite Automata:
  1. Deterministic Finite Automata (DFA)
  2. Non-Deterministic Finite Automata (NFA)

Deterministic Finite Automata

- The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state.
The term"finite automaton or finite state machine" will refer to the deterministic variety, although we shall use"deterministic" or the abbreviation DFA normally.

**Note:** If you ask to design FSM or FA, then you should design DFA.

**Definition of a Deterministic Finite Automaton**

A deterministic finite automaton consists of:

1. A finite set of states, often denoted \( Q \).
2. A finite set of input symbols, often denoted \( \Sigma \).
3. A transition function that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted \( \delta \). In our informal graph representation of automata, \( \delta \) was represented by arcs between states and the labels on the arcs. If \( q \) is a state, and \( a \) is an input symbol, then \( \delta(q,a) \) is that state \( p \) such that there is an arc labelled \( a \) from \( q \) to \( p \).
4. A start state, one of the states in \( Q \).
5. A set of final or accepting states \( F \). The set \( F \) is a subset of \( Q \).

The most brief representation of a DFA is a listing of the five components above. In proofs we often talk about a DFA in "five-tuple" notation:

\[ A = (Q, \Sigma, \delta, q_0, F) \]

where \( A \) is the name of the DFA, \( Q \) is its set of states, \( \Sigma \) its input symbols, \( \delta \) its transition function, \( q_0 \) its start state, and \( F \) its set of accepting states.

**Simpler Notations for DFA's:**

There are two preferred notations for representing transition function for automata:

1. A transition diagram, which is a graph.
2. A transition table, which is a tabular listing of the $\delta$ function, which by implication tells us the set of states and the input alphabet.

1. **Transition Diagrams**
A transition diagram for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

1. For each state in $Q$ there is a node.

2. For each state $q$ in $Q$ and each input symbol $a$ in $\Sigma$, let $\delta(q, a) = p$, then the transition diagram has an arc from node $q$ to node $p$, labelled $a$. If there are several input symbols that cause transitions from $q$ to $p$, then the transition diagram can have one arc, labelled by the list of these symbols.

3. There is an arrow into the start state $q_0$, labelled Start. This arrow does not originate at any node.

4. Nodes corresponding to accepting states (those in $F$) are marked by a double circle. States not in $F$ have a single circle.

![Transition Diagram Example](image)

**Figure:** The transition diagram for the DFA accepting all strings with a substring 01

2. **Transition Tables**

- A transition table is a conventional, tabular representation of a function like $\delta$ that takes two arguments and returns a value.

- The rows of the table correspond to the states, and the columns correspond to the inputs.
The entry for the row corresponding to state $q$ and the column corresponding to input $a$ is the state $\delta(q, a)$.

![Transition Table]

Figure: Transition table for the DFA accepting all strings with a substring 01

Extending the Transition Function to Strings: Refer class notes

The Language of a DFA:

- Now, we can define the language of a DFA $A = (Q, \Sigma, \delta, q_0, F)$. This language is denoted $L(A)$, and is defined by

$$L(A) = \{w \mid \delta(q_0, w) \text{ is in } F\}$$

- That is, the language of $A$ is the set of strings $\omega$ that take the start state $q_0$ to one of the accepting states.

Nondeterministic Finite Automata

- A "nondeterministic" finite automaton (NFA) has the power to be in several states at once. This ability is often expressed as an ability to "guess" something about its input.
Like the DFA, an NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states.

It also has a transition function, which we shall commonly call $\delta$. The difference between the DFA and the NFA is in the type of $\delta$.

For the NFA, $\delta$ is a function that takes a state and input symbol as arguments (like the DFA 's transition function), but returns a set of zero, one, or more states (rather than returning exactly one state, as the DFA must).

Example: Figure, shows a nondeterministic finite automaton, whose job is to accept all and only the strings of 0's and 1's that end in 01. State $q_0$ is the start state, and we can think of the automaton as being in state $q_0$ (perhaps among other states) whenever it has not yet "guessed" that the final 01 has begun. It is always possible that the next symbol does not begin the final 01, even if that symbol is 0. Thus, state $q_0$ may transition to itself on both 0 and 1.

![Figure: An NFA accepting all strings that end in 01](image)

However, if the next symbol is 0, this NFA also guesses that the final 01 has begun. An arc labelled 0 thus leads from $q_0$ to state $q_1$. Notice that there are two arcs labelled 0 out of $q_0$.

The NFA has the option of going either to $q_0$ or to $q_1$, and in fact it does both, as we shall see when we make the definitions precise. In state $Q_1$, the NFA checks that the next symbol is 1, and if so, it goes to state $q_2$ and accepts.

Notice that there is no arc out of $q_1$ labeled 0, and there are no arcs at all out of $q_2$. In these situations, the thread of the NFA's existence corresponding to those states simply "dies," although other threads may continue to exist.
**CHOPRA ACADEMY**

@Bandra:9867 843 854 @Thane:9867 843 853 @Vashi :9867 843 852
For Doubts Contact Ganesh Sir@83 84 82 05 75

- While a DFA has exactly one arc out of each state for each input symbol, an NFA has no such constraint; we have seen in Fig. cases where the number of arcs is zero, one, and two, for example.

- Below figure suggests how an NFA processes inputs. We have shown what happens when the automaton of above Fig. receives the input sequence 00101. It starts in only its start state, q₀. When the first 0 is read, the NFA may go to either state q₀ or state q₁, so it does both. These two threads are suggested by the second column in below Fig.

Then, the second 0 is read. State q₀ may again go to both q₀ and q₁. However, state q₁ has no transition on 0, so it "dies." When the third input, a 1, occurs, we must consider transitions from both q₀ and q₁. We find that q₀ goes only to q₀ on 1, while q₁ goes only to q₂.

![Diagram](image)

**Figure:** The states an NFA is in during the processing of input sequence 00101

- Thus, after reading 001, the NFA is in states q₀ and q₂. Since q₂ is an accepting state, the NFA accepts 001.
• However, the input is not finished. The fourth input, a 0, causes q2’s thread to die, while q0 goes to both q0 and q1. The last input, a 1, sends q0 to q0 and q1, to q2. Since we are again in an accepting state, 00101 is accepted.

Definition of Nondeterministic Finite Automata

• Now, let us introduce the formal notions associated with nondeterministic finite automata. The differences between DFA’s and NFA’s will be pointed out as we do. An NFA is represented essentially like a DFA:

\[ A = (Q, \Sigma, \delta, q_0, F) \]

where:

1. \( Q \) is a finite set of states.
2. \( \Sigma \) is a finite set of input symbols.
3. \( q_0 \), a member of \( Q \), is the start state.
4. \( F \), a subset of \( Q \), is the set of final (or accepting) states.
5. \( \delta \), the transition function that takes a state in \( Q \) and an input symbol in \( \Sigma \) as arguments and returns a subset of \( Q \). Notice that the only difference between an NFA and a DFA is in the type of value that \( \delta \) returns: a set of states in the case of an NFA and a single state in the case of a DFA.

The Extended Transition Function: Refer Class Notes.

The Language of an NFA:

• As we have suggested, an NFA accepts a string \( \omega \) if it is possible to make any sequence of choices of next state, while reading the characters of \( \omega \), and go from the start state to any accepting state.

CHOPRA ACADEMY
@Bandra:9867 843 854 @Thane:9867 843 853 @Vashi :9867 843 852
For Doubts Contact Ganesh Sir@83 84 82 05 75
The fact that other choices using the input symbols of \( \omega \) lead to a non-accepting state, or do not lead to any state at all (i.e., the sequence of states "dies"), does not prevent \( \omega \) from being accepted by the NFA as a whole.

Formally, if \( A = (Q, \Sigma, \delta, q_0, F) \) is an NFA, then

\[
L(A) = \{ w | \delta(q_0, w) \cap F \neq \emptyset \}
\]

That is, \( L(A) \) is the set of strings \( \omega \) in \( \Sigma^* \) such that \( \delta(q_0, \omega) \) contains at least one accepting state.

Refer Class Notes for Examples.

Equivalence of Deterministic and Nondeterministic: Finite Automata: Refer Class Notes

An Application: Text Search

1. Finding Strings in Text

- A common problem in the age of the Web and other on-line text repositories is the following.
  1. Given a set of words, find all documents that contain one (or all) of those words. A search engine is a popular example of this process. The search engine uses a particular technology, called inverted indexes, where for each word appearing on the Web (there are 100,000,000 different words), a list of all the places where that word occurs is stored. Machines with very large amounts of main memory keep the most common of these lists available, allowing many people to search for documents at once.
2. Inverted-index techniques do not make use of finite automata, but they also take very large amounts of time for crawlers to copy the Web and set up the indexes. There are a number of related applications that are unsuited for in-vetted indexes, but are good applications for automaton-based techniques.

- The characteristics that make an application suitable for searches that use automata are:

  1. The repository on which the search is conducted is rapidly changing. For example:
     - Every day, news analysts want to search the day's on-line news articles for relevant topics. For example, a financial analyst might search for certain stock ticker symbols or names of companies.
     - A "shopping robot" wants to search for the current prices charged for the items that its clients request. The robot will retrieve current catalog pages from the Web and then search those pages for words that suggest a price for a particular item.

  2. The documents to be searched cannot be cataloged. For example, Amazon.com does not make it easy for crawlers to find all the pages for all the books that the company sells. Rather, these pages are generated "on the fly" in response to queries. However, we could send a query for books on a certain topic, say "finite automata," and then search the pages retrieved for certain words, e.g., "excellent" in a review portion.

2. Nondeterministic Finite Automata for Text Search

- Suppose we are given a set of words, which we shall call the keywords, and we want to find occurrences of any of these words. In applications such as these, a useful way to proceed is to design a nondeterministic finite automaton, which signals, by entering an accepting state, that it has seen one of the keywords.

- The text of a document is fed, one character at a time to this NFA, which then recognizes occurrences of the keywords in this text. There is a simple form to an NFA that recognizes a set of key words.

  1. There is a start state with a transition to itself on every input symbol, e.g. every printable ASCII character if we are examining text. Intuitively, the start
state represents a "guess" that we have not yet begun to see one of the keywords, even if we have seen some letters of one of these words.

2. For each keyword $a_1a_2...a_k$ there are $k$ states, say $q_1$, $q_2$, ... , $q_k$. There is a transition from the start state to $q_1$ on symbol $a_1$, a transition from $q_1$ to $q_2$ on symbol $a_2$, and so on. The state $q_k$ is an accepting state and indicates that the keyword $a_1a_2...a_k$ has been found.

- Example: Suppose we want to design an NFA to recognize occurrences of the words web and ebay. The transition diagram for the NFA designed using the rules above is in below Fig. State 1 is the start state, and we use $\Sigma$ to stand for the set of all printable ASCII characters. States 2 through 4 have the job of recognizing web, while states 5 through 8 recognize ebay.

- We have two major choices for an implementation of this NFA.
  1. Write a program that simulates this NFA by computing the set of states it is in after reading each input symbol. The simulation was suggested in below Fig.
  2. Convert the NFA to an equivalent DFA using the subset construction. Then simulate the DFA directly.

![Diagram of an NFA]

Figure: An NFA that searches for the words web and ebay

- Some text-processing programs, such as advanced forms of the UNIX grep command (egrep and fgrep) actually use a mixture of these two approaches. However, for our purposes, conversion to a DFA is easy and is guaranteed not to increase the number of states.
Finite Automata With Epsilon-Transitions: Refer Class Notes

FSM properties:

1. Periodicity:
   a. The limitations of FSM is that it does not have the capacity to remember
      arbitrarily large amount of information, because it has only a fixed number of
      states and this set a limit to the length of the sequence it can remember.

   b. Also, we have seen a finite control representation of FSM, where read head
      moves always one position to the right after reading an input symbol.

   c. Head can never move in reverse direction, therefore, FSM cannot retrieve
      what it read previously, before coming to current position of tape.

   d. As it cannot retrieve we cannot say, it can remember something. This also
      means that FSM eventually will always repeat a state produce a periodic
      sequence of states.

2. State Determination:
   a. Since, the initial states of an FSM and the input sequence given to it,
      determines the output sequence.
b. It is always possible to discover the unknown state, in which the FSM resides at a particular instance.

3. **Impossibility of Multiplication:**
   a. An FSM cannot remember arbitrarily long sequences.
   
   b. Hence for multiplication operation it is required to remember two full sequences corresponding to multiplier and multiplicand, while multiplying, it is also required to store the partial sums that we obtain normally at intermediate stages of multiplication.
   
   c. Therefore, no FSM can multiply two given arbitrarily large numbers.

4. **Impossibility of palindrome recognition:** FSM can recognizing a palindrome, because it does not have that capability to remember all the symbols it reads until half the way point of input sequence, in order to match them in reverse order, with the symbols in second half of the sequence.

5. **Impossibility to check well-formedness of parenthesis:**

As FSM has no capability to remember all the earlier inputs to it, cannot compare with the remaining to check well-formedness. It is an impossible task for any FSM.

**Problems on Finite Automata / Finite State Machine:**

Refer class Notes for below problems:

**Divisibility:**
1. Design a machine which checks whether a given decimal number is divisible by 3.

2. Design a machine which checks whether a given decimal number is divisible by 4.

3. Design a machine which checks whether the given binary number is divisible by 3. *(May-01, May-02, Dec-02, June-07)*
4. Design a FSM to check whether a given unary number is divisible by 3. (May-03)

5. Design a FSM to check whether a given unary number is divisible by 4. (Dec-05)

6. Design divisibility by four testers FSM for binary numbers. (Dec-05)

7. Design a machine which checks whether the given ternary number is divisible by 4. (Dec-02)

8. Design a machine which checks whether the given ternary number is divisible by 5 or not. (May-03)

Ending with:

9. Design FSM that accepts set of all strings ending with 101.

10. Design FSM that accepts set of all strings ending with 'aab'.

11. Design FSM that accepts set of all strings ending with '110' or '101'.

12. Design FSM that accepts set of all strings ending with 'abb' or 'bba'.

13. Design FSM that accepts set of all strings with second last symbol is 'a' over {a, b}.

Contains:

14. Design FSM in which input is valid if it contains '1101' over {0, 1}.

15. Design FSM that accepts set of all strings, if it contains at least one occurrence of substring 'bba' over {a, b}.

16. Design an FSM in which input is valid if it does not contain any occurrence of 3 consecutive b's over {a, b}.

Adder:
17. Design FSM to implement Binary Adder. *(Jun-08)*

**Extra Problems:**

18. Design FSM, which has odd number of 0's and any number of 1's.

19. Design FSM which accepts a string if it contains even number of 0's and odd number of 1's.

20. Design FSM which accepts all string which end with 00.

21. Design FSM that accepts set of all strings containing with 1001.

22. Design FSM which accepts the string if it is ending with 'aa'.

23. Design FSM for the language in which the string is acceptable if the second last symbol is 'a' over the $\Sigma=\{a, b\}$

24. Design a machine which checks whether a given decimal number is even.

25. Design FSM for the language over the $\Sigma=\{a, b\}$ and the string is acceptable if it contains
   a. At most 3 a's
   b. At least 3 a's
   c. Exactly 3 a's

26. Design FSM for the language over the $\Sigma=\{a, b\}$ and the string is acceptable if it does not contain 'bbb'.

27. Design FSM to recognize sub string CAT from the set $\Sigma=\{C, H, A, R, I, O, T\}$. *(May-00, Dec-01)*

28. Design FSM to add 2 binary numbers of equal lengths. *(May-00, Dec-01)*

29. Design FSM which generates the remainder when the given decimal number is divided by 3. *(Dec-01)*
30. Design FSM which generates the remainder when the given binary number is divided by 4.

31. Design FSM which generates the remainder when the given ternary number is divided by 3.

Problems:

Refer class Notes for below problems:

1. Give DFA accepting the following language over $\Sigma = \{0, 1\}$
   a. Number of 1's is multiple of 3.
   b. Number of 1's is not multiple of 3.

2. Give DFA accepting the following language over $\Sigma = \{0, 1\}$
   a. Number of 1's is even and number of 0's is even.
   b. Number of 1's is odd and number of 0's is odd.

3. Design a DFA for a set of strings over $\Sigma = \{0, 1\}$ such that the number of 0's is divisible by 5, and number of 1's divisible by 3.

4. Draw DFA for the following language over $\Sigma = \{0, 1\}$
   a. All strings of length at most 5.
   b. All strings with exactly two 1's.
   c. All strings containing at least two 0's.
   d. All strings containing at most two 0's.
   e. All strings starting with 1 and length of the string is divisible by 3.

5. Draw DFA for the following language over $\Sigma = \{0, 1\}$
   a. All strings starting with 'abb'.
   b. All strings with 'abb' as a substring i.e., 'abb' anywhere in the string.
   c. All strings ending in 'abb'.

6. Design DFA for a language of string 0 and 1, such that:
   a. Ending with '10'.
   b. Ending with '11'.
   c. Ending with '1'.

---

CHOPRA ACADEMY

@Bandra:9867 843 854 @Thane:9867 843 853 @Vashi:9867 843 852
For Doubts Contact Ganesh Sir@83 84 82 05 75
7. Design the DFA which accepts set of strings such that every string containing '00' as a substring but not '000' as a substring.  

8. Design the DFA for the language, containing strings in which leftmost symbol differ from rightmost symbol, $\Sigma = \{0, 1\}$.  

9. Design a DFA for set of strings over $\Sigma = \{0, 1\}$ in which there are at least two occurrences of 'b' between any two occurrences of 'a'.

10. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ ending in either 'ab' or 'ba'.

11. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ containing both 'ab' and 'ba' as substrings.

12. Design a FA that reads string defined over $\Sigma = \{0, 1\}$ and accepts only those strings which end up in either 'aa' or 'bb'.

13. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ containing neither 'aa' nor 'bb' as a substring.

14. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ such that each 'a' in $\omega$ is immediately preceded and immediately followed by a 'b'.

15. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ such that every block of five consecutive symbols contains at least two 0's.

16. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ such that strings either begin or end with '01'.

17. Design a DFA for set of all strings over $\Sigma = \{0, 1\}$ such that the third symbol from the right end is '1'.

18. Construct a DFA for set of strings containing either the substring 'aaa' or 'bbb'.

---

**CHOPRA ACADEMY**

@Bandra:9867 843 854 @Thane:9867 843 853 @Vashi :9867 843 852

For Doubts Contact Ganesh Sir@83 84 82 05 75
19. Construct a DFA for accepting a set of strings over $\Sigma = \{0, 1\}$ not ending in '010'.

20. Design a DFA that reads strings made up of letters in the word 'CHARIOT' and recognizes these strings that contain the word 'CAT' as a substring.

Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>(Computer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 2

Regular Expressions

- We begin this chapter by introducing the notation called "regular expressions." These expressions are another type of language-defining notation.

- Regular expressions also may be thought of as a "programming language," in which we express some important applications, such as text-search applications or compiler components.

Regular Expressions

- Now, we switch our attention from machine-like descriptions of languages — deterministic and nondeterministic finite automata — to an algebraic description: the "regular expression."

- However, regular expressions offer something that automata do not: a declarative way to express the strings we want to accept.

1. Search commands such as the UNIX grep or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems. These systems use a regular-expression-like notation for describing patterns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA, and simulate that automaton on the file being searched.

2. Lexical-analyzer generators, such as Lex or Flex. Recall that a lexical analyzer is the component of a compiler that breaks the source program into logical units (called tokens) of one or more characters that have a stated significance. Examples of tokens include keywords (e.g., while), identifiers (e.g., any letter followed by zero or more letters and/or digits), and signs, such as + or <=. A lexical-analyzer generator accepts descriptions of the forms of tokens, which are essentially regular expressions, and produces a DFA that recognizes which token appears next on the input.
Operators of the Regular Expression:

1. **Union:**
   Say \( L_1 \) and \( L_2 \) are the two Language, then \( L_1 \cup L_2 = \{a, b \mid a \in L_1 \text{ and } b \in L_2\} \)
   Example: \( L_1 = \{a, b\} \) over \( \Sigma = \{a, b\} \) and \( L_2 = \{aa, bb\} \) over \( \Sigma = \{a, b\} \)
   Then \( L_1 \cup L_2 = \{a, b, aa, bb\} \)

2. **Concatenation:**
   Say \( L_1 \) and \( L_2 \) are the two Language, then \( L_1.L_2 = \{ab \mid a \in L_1 \text{ and } b \in L_2\} \)
   Example: \( L_1 = \{a, b\} \) over \( \Sigma = \{a, b\} \) and \( L_2 = \{aa, bb\} \) over \( \Sigma = \{a, b\} \)
   Then \( L_1.L_2 = \{aaa, abb, baa, bbb\} \)

3. **Closure (zero or more):**
   Say \( L \) is a Language, then language closure \((L^*)\) is denoted as:
   \[ L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots \]
   Example:
   If \( \Sigma = \{0, 1\} \)
   Then \( L^* \) over \( \Sigma = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001 \ldots\} \)

4. **Positive Closure (one or more):**
   Say \( L \) is a Language, then Language positive closure \((L^+)\) is denoted as:
   \[ L^+ = L^1 \cup L^2 \cup L^3 \ldots \]
   Example:
   If \( \Sigma = \{0, 1\} \)
   Then \( L^+ \) over \( \Sigma = \{0, 1, 00, 01, 10, 11, 000, 001 \ldots\} \)
   \[ L^+ = L.L^* \]

Building Regular Expressions:
- Algebras of all kinds start with some elementary expressions, usually constants and/or variables.
• Algebras then allow us to construct more expressions by applying a certain set of operators to these elementary expressions and to previously constructed expressions.

• Usually, some method of grouping operators with their operands, such as parentheses, is required as well.

• For instance, the familiar arithmetic algebra starts with constants such as integers and real numbers, plus variables, and builds more complex expressions with arithmetic operators such as + and ×.

• The algebra of regular expressions follows this pattern, using constants and variables that denote languages, and operators for the three operations union, dot, and star.

• We can describe the regular expressions recursively, as follows. In this definition, we not only describe what the legal regular expressions are, but for each regular expression E, we describe the language it represents, which we denote L(E).

The basis consists of three parts:

1. The constants ε and ϕ are regular expressions, denoting the languages {ε} and ϕ, respectively. That is, L(ε) = {ε}, and L(ϕ) = ϕ.

2. If a is any symbol, then a is a regular expression. This expression denotes the language {a}. That is, L(a) = {a}. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that a refers to cl, should be obvious.

3. A variable, usually capitalized and italic such as L, is a variable, representing any language.
Expressions and Their Languages

Strictly speaking, a regular expression $E$ is just an expression, not a language. We should use $L(E)$ when we want to refer to the language that $E$ denotes. However, it is common usage to refer to say “$E$” when we really mean “$L(E)$.” We shall use this convention as long as it is clear we are talking about a language and not about a regular expression.

Precedence of Regular-Expression Operators

- Like other algebras, the regular-expression operators have an assumed order of "precedence," which means that operators are associated with their operands in a particular order.

- We are familiar with the notion of precedence from ordinary arithmetic expressions. For instance, we know that $xy + z$ groups the product $xy$ before the sum, so it is equivalent to the parenthesized expression $(xy) + z$ and not to the expression $x(y + z)$. Similarly, we group two of the same operators from the left in arithmetic, so $x - y - z$ is equivalent to $(x - y) - z$, and not to $x - (y - z)$.

- For regular expressions, the following is the order of precedence for the operators:

  1. The star operator is of highest precedence. That is, it applies only to the smallest sequence of symbols to its left that is a well-formed regular expression.

  2. Next in precedence comes the concatenation or "dot" operator. After grouping all stars to their operands, we group concatenation operators to their operands. That is, all expressions that are juxtaposed (adjacent, with no intervening operator) are grouped together. Since concatenation is an associative operator it does not matter in what order we group consecutive concatenations; although if there is a choice to be made, you should group them from the left. For instance, $012$ is grouped $(01)2$. 
3. Finally, all unions ( + operators) are grouped with their operands. Since union is also associative, it again matters little in which order consecutive unions are grouped, but we shall assume grouping from the left.

### Finite Automata and Regular Expressions

- While the regular-expression approach to describing languages is fundamentally different from the finite-automaton approach, these two notations turn out to represent exactly the same set of languages, which we have termed the "regular languages."

- We have already shown that deterministic finite automata, and the two kinds of nondeterministic finite automata — with and without ε-transitions — accept the same class of languages. In order to show that the regular expressions define the same class, we must show that:

  1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we can assume the language is accepted by some DFA.

  2. Every language defined by a regular expression is defined by one of these automata. For this part of the proof, the easiest is to show that there is an NFA with ε-transitions accepting the same language.

Figure: shows all the equivalences we have proved or will prove. An arc from class X to class Y means that we prove every language defined by class X is also defined by class Y. Since the graph is strongly connected (i.e., we can get from each of the four nodes to any other node) we see that all four classes are really the same.
Figure: Plan for showing the equivalence of four different notations for regular languages

From DFA's to Regular Expressions:

- The construction of a regular expression to define the language of any DFA is surprisingly tricky. Roughly, we build expressions that describe sets of strings that label certain paths in the DFA’s transition diagram.

- However, the paths are allowed to pass through only a limited subset of the states. In an inductive definition of these expressions, we start with the simplest expressions that describe paths that are allowed to pass through any states (i.e., they are single nodes or single arcs), and inductively build the expressions that let the paths go through progressively larger sets of states.

- Finally, the paths are allowed to go through any state; i.e., the expressions we generate at the end represent all possible paths. These ideas appear in the proof of the following theorem.

Theorem: If $L = L(A)$ for some DFA $A$, then there is a regular expression $R$ such that $L = L(R)$.

Refer Class Notes for Proof.
Converting DFA’s to Regular Expressions by Eliminating States: Refer Class Notes.

Converting Regular Expressions to Automata:

**Theorem:** Every language defined by a regular expression is also defined by a finite automaton.

**PROOF:** Suppose $L = L(R)$ for a regular expression $R$. We show that $L = L(E)$ for some $\varepsilon$-NFA $E$ with:

1. Exactly one accepting state.
2. No arcs into the initial state.
3. No arcs out of the accepting state.

The proof is by structural induction on $R$, following the recursive definition of regular expressions that we had before.
In part (a) we see how to handle the expression $E$. The language of the automaton is easily seen to be $\{\varepsilon\}$, since the only path from the start state to an accepting state is labeled $E$.

Part (b) shows the construction for $\phi$. Clearly there are no paths from start state to accepting state, so $\phi$ is the language of this automaton.

Finally, part (c) gives the automaton for a regular expression $a$. The language of this automaton evidently consists of the one string $a$, which is also $L(a)$. It is easy to check that these automata all satisfy conditions (1), (2), and (3).
The four cases are:

1. The expression is $R + S$ for some smaller expressions $R$ and $S$. Then the automaton of Fig. (a) serves. That is, starting at the new start state, we can go to the start state of either the automaton for $R$ or the automaton for $S$. We then reach the accepting state of one of these automata, following a path labeled by some string in $L(R)$ or $L(S)$.

Figure: The inductive step in the regular-expression-to-$\varepsilon$-FA construction
respectively. Once we reach the accepting state of the automaton for $R$ or $S$, we can follow one of the $\varepsilon$-arcs to the accepting state of the new automaton.

Thus, the language of the automaton in Fig. (a) is $L(R) \cup L(S)$.

2. The expression is $RS$ for some smaller expressions $R$ and $S$. The automaton for the concatenation is shown in Fig. (b). Note that the start state of the first automaton becomes the start state of the whole, and the accepting state of the second automaton becomes the accepting state of the whole. The idea is that the only paths from start to accepting state go first through the automaton for $R$, where it must follow a path labeled by a string in $L(R)$, and then through the automaton for $S$, where it follows a path labeled by a string in $L(S)$. Thus, the paths in the automaton of Fig. (b) are all and only those labeled by strings in $L(R)L(S)$.

3. The expression is $R^*$ for some smaller expression $R$. Then we use the automaton of Fig. (c). That automaton allows us to go either:

   (a) Directly from the start state to the accepting state along a path labeled $\varepsilon$. That path lets us accept $\varepsilon$, which is in $L(R^*)$ no matter what expression $R$ is.

   (b) To the start state of the automaton for $R$, through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept strings in $L(R)$, $L(R)L(R)$, $L()L(R)L(R)$, and so on, thus covering all strings in $L(R^*)$ except perhaps $\varepsilon$, which was covered by the direct arc to the accepting state mentioned in (1a).

4. The expression is $(R)$ for some smaller expression $R$. The automaton for $R$ also serves as the automaton for $(R)$, since the parentheses do not change the language defined by the expression.
Figure: Automata constructed for $(0 + 1)^*1(0 + 1)$
3.3 Applications of Regular Expressions

A regular expression that gives a "picture" of the pattern we want to recognize is the medium of choice for applications that search for patterns in text. The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text. In this section, we shall consider two important classes of regular-expression-based applications: lexical analyzers and text search.

3.3.1 Regular Expressions in UNIX

Before seeing the applications, we shall introduce the UNIX notation for extended regular expressions. This notation gives us a number of additional capabilities. In fact, the UNIX extensions include certain features, especially the ability to name and refer to previous strings that have matched a pattern, that actually allow nonregular languages to be recognized. We shall not consider these features here; rather we shall only introduce the shorthands that allow complex regular expressions to be written succinctly.

The first enhancement to the regular-expression notation concerns the fact that most real applications deal with the ASCII character set. Our examples have typically used a small alphabet, such as \{0, 1\}. The existence of only two symbols allowed us to write succinct expressions like \(0 + 1\) for "any character." However, if there were 128 characters, say, the same expression would involve listing them all, and would be highly inconvenient to write. Thus, UNIX regular expressions allow us to write character classes to represent large sets of characters as succinctly as possible. The rules for character classes are:

- The symbol . (dot) stands for "any character."
- The sequence \([a_1a_2\cdots a_k]\) stands for the regular expression

\[a_1 + a_2 + \cdots + a_k\]

This notation saves about half the characters, since we don't have to write the \(+\) signs. For example, we could express the four characters used in C comparison operators by \([<>=!]\).

- Between the square braces we can put a range of the form \(x-y\) to mean all the characters from \(x\) to \(y\) in the ASCII sequence. Since the digits have codes in order, as do the upper-case letters and the lower-case letters, we can express many of the classes of characters that we really care about with just a few keystrokes. For example, the digits can be expressed \([0-9]\), the upper-case letters can be expressed \([A-Z]\), and the set of all letters and digits can be expressed \([A-Za-z0-9]\). If we want to include a minus sign among a list of characters, we can place it first or last, so it is not confused with its use to form a character range. For example, the set
of digits, plus the dot, plus, and minus signs that are used to form signed decimal numbers may be expressed \([-+.0-9]\). Square brackets, or other characters that have special meanings in UNIX regular expressions can be represented as characters by preceding them with a backslash (\).

- There are special notations for several of the most common classes of characters. For instance:
  
a) \[[:digit:]\] is the set of ten digits, the same as \([0-9]\).\(^3\)
b) \[[:alpha:]\] stands for any alphabetic character, as does \([A-Za-z]\).
c) \[[:alnum:]\] stands for the digits and letters (alphabetic and numeric characters), as does \([A-Za-z0-9]\).

In addition, there are several operators that are used in UNIX regular expressions that we have not encountered previously. None of these operators extend what languages can be expressed, but they sometimes make it easier to express what we want.

1. The operator \(|\) is used in place of + to denote union.
2. The operator \(?\) means “zero or one of.” Thus, \(R?\) in UNIX is the same as \(\epsilon + R\) in this book’s regular-expression notation.
3. The operator \(+\) means “one or more of.” Thus, \(RR^*\) in UNIX is shorthand for \(RRR\) in our notation.
4. The operator \({n}\) means “\(n\) copies of.” Thus, \(R\{5\}\) in UNIX is shorthand for \(RRRRR\).

Note that UNIX regular expressions allow parentheses to group subexpressions, just as for the regular expressions described in Section 3.1.2, and the same operator precedence is used (with ?, + and \{n\} treated like \* as far as precedence is concerned). The star operator \* is used in UNIX (without being a superscript, of course) with the same meaning as we have used.

### 3.3.2 Lexical Analysis

One of the oldest applications of regular expressions was in specifying the component of a compiler called a “lexical analyzer.” This component scans the source program and recognizes all tokens, those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens, but there are many others.

\(^3\)The notation \[[:digit:]\] has the advantage that should some code other than ASCII be used, including a code where the digits did not have consecutive codes, \[[:digit:]\] would still represent \([0123456789]\), while \([0-9]\) would represent whatever characters had codes between the codes for 0 and 9, inclusive.
The Complete Story for UNIX Regular Expressions

The reader who wants to get the complete list of operators and shorthands available in the UNIX regular-expression notation can find them in the manual pages for various commands. There are some differences among the various versions of UNIX, but a command like `man grep` will get you the notation used for the `grep` command, which is fundamental. “Grep” stands for “Global (search for) Regular Expression and Print,” incidentally.

The UNIX command `lex` and its GNU version `flex`, accept as input a list of regular expressions, in the UNIX style, each followed by a bracketed section of code that indicates what the lexical analyzer is to do when it finds an instance of that token. Such a facility is called a *lexical-analyzer generator*, because it takes as input a high-level description of a lexical analyzer and produces from it a function that is a working lexical analyzer.

Commands such as `lex` and `flex` have been found extremely useful because the regular-expression notation is exactly as powerful as we need to describe tokens. These commands are able to use the regular-expression-to-DFA conversion process to generate an efficient function that breaks source programs into tokens. They make the implementation of a lexical analyzer an afternoon’s work, while before the development of these regular-expression-based tools, the hand-generation of the lexical analyzer could take months. Further, if we need to modify the lexical analyzer for any reason, it is often a simple matter to change a regular expression or two, instead of having to go into mysterious code to fix a bug.

Laws for Regular Expression:

Following are the algebraic laws for Regular expressions:

1. **Associativity**: Associativity is the property of an operator that allows to regroup the operands when the operator is applied twice.
   
   Example:
2. **Commutative**: Commutative is the property of an operator that says we can switch the order of operands and get the same result.
   Example:
   Commutative law of union:
   \( A + B = B + A \)

3. **Identity**: An identity for an operator is a value such that when the operator is applied to the identity and some other value, the result is the other value.
   Example:
   0 is the identity for addition:
   \( 0 + X = X + 0 = X \), here 0 is the identity.

   1 is the identity for Multiplication:
   \( 1 \times X = X \times 1 = X \), here 1 is the identity.

   \( \phi \) is the identity for union
   \( \Phi \cup L = L \cup \Phi = L \), here \( \Phi \) is the identity.

   \( \varepsilon \) is the identity for concatenation
   \( \varepsilon . L = L . \varepsilon = L \), here \( \varepsilon \) is the identity.

4. **Annihilators**: An annihilator for an operator is a value such that when the operator is applied to the annihilator and some other value, the result is the annihilator.
   Example:
   0 is the annihilator for multiplication.
   \( 0 \times X = X \times 0 = 0 \), here 0 is the annihilator.

   \( \Phi \) is the annihilator for concatenation.
   \( \Phi . L = L . \Phi = L \), here \( \Phi \) is the annihilator.

5. **Distributive Law**: A distributive law involves two operators and asserts that one operator can be pushed down to be applied to each argument of other operator individually.
   Example:
   \( X \times (Y + Z) = X \times Y + X \times Z \)
(X+Y)*Z = X*Z + Y*Z

6. **Idempotent Law**: An operator is said to be idempotent if the result of applying it to two of the same values as arguments is that value.
Example:
Law of idempotent for union:
L U L = L

7. **Laws involving closures**:
   \((L^*)^* = L^*
   \phi^* = \epsilon
   \epsilon^* = \epsilon
   L^* = LL*
   L^* = L^* + \epsilon

**Problems:**

Refer class Notes for below problems:

1. Describe the following sets by Regular expression.
   a. \{abb\}
   b. \{1010\}
   c. \{ab, ba\}
   d. \{\epsilon, aa\}
   e. \{011, 0, 1, 110\}

2. Write the regular expression for the following:
   a. Set of all strings on \{a, b\} terminated by either an ‘a’ or a ‘bb’.
   b. Set of all strings on \{0, 1\} starting with 10 and ending with ‘111’.
   c. Set of all strings on \{a, b\} with an even number of a’s followed by an odd number of b’s.

3. Write regular expressions for the following languages:
   a. \(L = \{a^nb^m| n>=4, m<=3 \}\)
   b. \(L = \{w| (|w|mod 3 = 0), w \in \{a, b\} \}\)

4. Write regular expressions for the following languages:
   a. \(L = \{a^nb^m| (n+m) \) is even \}
b. \( L \in \{a, b\}^* \mid (|\text{number of } a's \text{ in } w| \mod 3 = 0) \) 

(Dec-07)

5. Write the regular expression to generate strings of length 6 or less over \( \{0, 1\} \)

6. Find a regular expression corresponding to each of the following subset of \( \{0, 1\} \):
   a. The language of all strings containing exactly two 0's
   b. The language of all strings that begin or end with 00 or 11.
   c. The language of all strings containing both 11 and 010 as substrings.

(Dec-06)

7. What is the language represented by the regular expression \( L((a \cup b)^* a) \).

8. Express in words the language represented by the following regular expression \((a^*bc)^* a\).

9. Write a regular expression for the set of all strings of 0's and 1's containing no more than 2 consecutive 1's.

10. Write regular expression for the following:
   a. Set of all strings on \( \{a, b\} \) which end in a 'a' or 'bb'
   b. Set of all strings on \( \{0, 1\} \) starting with 10 and ending with 111.
   c. Set of all strings on \( \{0, 1\} \) containing no more than 2 consecutive 1's.

11. Write a regular expression corresponding to each of the following subset of \( \{a, b\}^* \):
   a. Set of all strings having even number of a's and no b's.
   b. Set of all strings that contain even number of a's and b's.
   c. Set of all strings that contain odd number of a's and b's.

12. Write regular expression for the following:
   a. Set of all strings 0's and 1's such that 10th symbol from the right end is 1.
   b. Set of all string in \((0+1)^* \) such that some of two 0's are separated by a string whose length is 4i, for some \( i \geq 0 \).

13. Define the language such that all words begin and end with 'a' and in between any word using 'b'.

14. Define language such that it could contain at least one double letter.

15. Define language such that it could contain no occurrence of a double letter.
16. Prove the following identities for regular expressions $r$, $s$ and $t$.
   a. $r + s = s + r$
   b. $(r + s) + t = r + (s + t)$
   c. $(rs)t = r(st)$ (Dec-03)
   d. $r(s + t) = rt + st$ (Dec-03)
   e. $(r*)* = r*$
   f. $(e + r)^* = r^*$
   g. $(r^*s^*)^* = (r + s)^*$

17. Verify the following identities involving regular expressions:
   a. $(r + s) + t = r + (s + t)$
   b. $(rs)t = r(st)$ (May-06)

18. Prove or disprove the following regular expressions $r$, $s$ and $t$:
   a. $(rs + r)^*r = r(sr + r)^*$
   b. $s(rs + s)^r = rr^*s(r^*s)^*$
   c. $(r + s)^* = r^* + s^*$

19. Prove the formula: $(111^*)^* = (11 + 111)^*$

20. Show that $(1 + 00^*1) + (1 + 00^*1)(0 + 10^*1)^*(0 + 10^*1) = 0^*1(0 + 10^*1)^*$. (May-06)

21. Show that $0(0 + 1)^* + (0 + 1)^*00(0 + 1)^* = (1^*0^*)(01^*)^*$. (Dec-07)
Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 3

Proving languages to be non-regular

- We have established that the class of languages known as the regular languages has at least four different descriptions. They are the languages accepted by DFA's, by NFA's, and by ε-NFA's; they are also the languages defined by regular expressions.

- Not every language is a regular language. We shall introduce a powerful technique, known as the "pumping lemma," for showing certain languages not to be regular. We then give several examples of non-regular languages.

The Pumping Lemma for Regular Languages

- Let us consider the language L_{01} = \{0^n1^n | n \geq 1 \}. This language contains all strings 01, 0011, 000111, and so on, that consist of one or more 0's followed by an equal number of 1's. We claim that L_{01} is not a regular language.

- The intuitive argument is that if L_{01} were regular, then L_{01} would be the language of some DFA A. This automaton has some particular number of states, say k states. Imagine this automaton receiving k 0's as input. It is in some state after consuming each of the k + 1 prefixes of the input: 0, 0, 00, ..., 0^k.

- Since there are only k different states, the pigeonhole principle tells us that after reading two different prefixes, say 0^i and 0^j, A must be in the same state, say state q.

- However, suppose instead that after reading i or j 0's, the automaton A starts receiving 1's as input. After receiving i 1's, it must accept if it previously received i 0's, but not if it received j 0's. Since it was in state
q when the 1's started, it cannot "remember" whether it received i or j 0's, so we can "fool" A and make it do the wrong thing - accept if it should not, or fail to accept when it should.

- The above argument is informal, but can be made precise. However, the same conclusion, that the language $L_{01}$ is not regular, can be reached using a general result, as follows.

- **Theorem**: (The pumping lemma for regular languages) Let $L$ be a regular language. Then there exists a constant $n$ (which depends on $L$) such that for every string $\omega$ in $L$ such that $|\omega| \geq n$, we can break $\omega$ into three strings, $\omega = xyz$, such that:
  1. $y \neq \varepsilon$
  2. $|xy| \leq n$.
  3. For all $k \geq 0$, the string $xy^kz$ is also in $L$.

- That is, we can always find a nonempty string $y$ not too far from the beginning of $w$ that can be "pumped"; that is, repeating $y$ any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language $L$. 
**PROOF:** Suppose $L$ is regular. Then $L = L(A)$ for some DFA $A$. Suppose $A$ has $n$ states. Now, consider any string $w$ of length $n$ or more, say $w = a_1a_2 \cdots a_m$, where $m \geq n$ and each $a_i$ is an input symbol. For $i = 0, 1, \ldots, n$ define state $p_i$ to be $\delta(q_0, a_1a_2 \cdots a_i)$, where $\delta$ is the transition function of $A$, and $q_0$ is the start state of $A$. That is, $p_i$ is the state $A$ is in after reading the first $i$ symbols of $w$. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n + 1$ different $p_i$’s for $i = 0, 1, \ldots, n$ to be distinct, since there are only $n$ different states. Thus, we can find two different integers $i$ and $j$, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1a_2 \cdots a_i$.
2. $y = a_{i+1}a_{i+2} \cdots a_j$.
3. $z = a_{j+1}a_{j+2} \cdots a_m$.

That is, $x$ takes us to $p_i$ once; $y$ takes us from $p_i$ back to $p_i$ (since $p_i$ is also $p_j$), and $z$ is the balance of $w$. The relationships among the strings and states are suggested by Fig. 4.1. Note that $z$ may be empty, in the case that $i = 0$. Also, $z$ may be empty if $j = n = m$. However, $y$ can not be empty, since $i$ is strictly less than $j$.

![Diagram](image)

Figure 4.1: Every string longer than the number of states must cause state to repeat.
Now, consider what happens if the automaton $A$ receives the input $xy^kz$ for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state $q_0$ (which is also $p_0$) to $p_i$ on input $x$. Since $p_i$ is also $p_j$, it must be that $A$ goes from $p_i$ to the accepting state shown in Fig. 4.1 on input $z$. Thus, $A$ accepts $xz$.

If $k > 0$, then $A$ goes from $q_0$ to $p_i$ on input $x$, circles from $p_i$ to $p_i$ $k$ times on input $y^k$, and then goes to the accepting state on input $z$. Thus, for any $k \geq 0$, $xy^kz$ is also accepted by $A$; that is, $xy^kz$ is in $L$. □

Applications of the Pumping Lemma:

- Let us See some examples of how the pumping lemma is used. In each case, we shall propose a language and use the pumping lemma to prove that the language is not regular.
The Pumping Lemma as an Adversarial Game

Recall our discussion from Section 1.2.3 where we pointed out that a theorem whose statement involves several alternations of “for-all” and “there-exists” quantifiers can be thought of as a game between two players. The pumping lemma is an important example of this type of theorem, since it in effect involves four different quantifiers: “for all regular languages \( L \) there exists \( n \) such that for all \( w \) in \( L \) with \( |w| \geq n \) there exists \( xyz \) equal to \( w \) such that \( \ldots \).” We can see the application of the pumping lemma as a game, in which:

1. Player 1 picks the language \( L \) to be proved nonregular.
2. Player 2 picks \( n \), but doesn’t reveal to player 1 what \( n \) is; player 1 must devise a play for all possible \( n \)’s.
3. Player 1 picks \( w \), which may depend on \( n \) and which must be of length at least \( n \).
4. Player 2 divides \( w \) into \( x, y, \) and \( z \), obeying the constraints that are stipulated in the pumping lemma; \( y \neq \epsilon \) and \( |xy| \leq n \). Again, player 2 does not have to tell player 1 what \( x, y, \) and \( z \) are, although they must obey the constraints.
5. Player 1 “wins” by picking \( k \), which may be a function of \( n, x, y, \) and \( z \), such that \( xy^k z \) is not in \( L \).
Example 4.2: Let us show that the language $L_{eq}$ consisting of all strings with an equal number of 0's and 1's (not in any particular order) is not a regular language. In terms of the “two-player game” described in the box on “The Pumping Lemma as an Adversarial Game,” we shall be player 1 and we must deal with whatever choices player 2 makes. Suppose $n$ is the constant that must exist if $L_{eq}$ is regular, according to the pumping lemma; i.e., “player 2” picks $n$. We shall pick $w = 0^n1^n$, that is, $n$ 0's followed by $n$ 1's, a string that surely is in $L_{eq}$.

Now, “player 2” breaks our $w$ up into $xyz$. All we know is that $y \neq \epsilon$, and $|xy| \leq n$. However, that information is very useful, and we “win” as follows. Since $|xy| \leq n$, and $xy$ comes at the front of $w$, we know that $x$ and $y$ consist only of 0's. The pumping lemma tells us that $xz$ is in $L_{eq}$, if $L_{eq}$ is regular. This conclusion is the case $k = 0$ in the pumping lemma. However, $xz$ has $n$ 1's, since all the 1's of $w$ are in $z$. But $xz$ also has fewer than $n$ 0's, because we
lost the 0's of \( y \). Since \( y \neq \varepsilon \) we know that there can be no more than \( n - 1 \) 0's among \( x \) and \( z \). Thus, after assuming \( L_{eq} \) is a regular language, we have proved a fact known to be false, that \( \varepsilon z \varepsilon \) is in \( L_{eq} \). We have a proof by contradiction of the fact that \( L_{eq} \) is not regular. □

**Example 4.3**: Let us show that the language \( L_{pr} \) consisting of all strings of 1's whose length is a prime is not a regular language. Suppose it were. Then there would be a constant \( n \) satisfying the conditions of the pumping lemma. Consider some prime \( p \geq n + 2 \); there must be such a \( p \), since there are an infinity of primes. Let \( w = 1^p \).

By the pumping lemma, we can break \( w = xyz \) such that \( y \neq \varepsilon \) and \( |xy| \leq n \). Let \( |y| = m \). Then \( |xz| = p - m \). Now consider the string \( xy^{p - m}z \), which must be in \( L_{pr} \) by the pumping lemma, if \( L_{pr} \) really is regular. However,

\[
|xy^{p - m}z| = |xz| + (p - m)|y| = p - m + (p - m)m = (m + 1)(p - m)
\]

It looks like \( |xy^{p - m}z| \) is not a prime, since it has two factors \( m + 1 \) and \( p - m \). However, we must check that neither of these factors are 1, since then \( (m + 1)(p - m) \) might be a prime after all. But \( m + 1 > 1 \), since \( y \neq \varepsilon \) tells us \( m \geq 1 \). Also, \( p - m > 1 \), since \( p \geq n + 2 \) was chosen, and \( m \leq n \) since

\[
m = |y| \leq |xy| \leq n
\]

Thus, \( p - m > 2 \).

Again we have started by assuming the language in question was regular, and we derived a contradiction by showing that some string not in the language was required by the pumping lemma to be in the language. Thus, we conclude that \( L_{pr} \) is not a regular language. □

**Closure Properties of Regular Languages:**

- In this section, we shall prove several theorems of the form "if certain languages are regular, and a language \( L \) is formed from them by certain operations (e.g., \( L \) is the union of two regular languages), then \( L \) is also regular." These theorems are often called closure properties of the regular
languages, since they show that the class of regular languages is closed under the operation mentioned.

- Closure properties express the idea that when one (or several) languages are regular, then certain related languages are also regular. They also serve as an interesting illustration of how the equivalent representations of the regular languages (automata and regular expressions) reinforce each other in our understanding of the class of languages, since often one representation is far better than the others in supporting a proof of a closure property. Here is a summary of the principal closure properties for regular languages:

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

Closure of Regular Languages Under Boolean Operations

Our first closure properties are the three boolean operations: union, intersection, and complementation:

1. Let $L$ and $M$ be languages over alphabet. Then $L \cup M$ is the language that contains all strings that are in either or both of $L$ and $M$.

2. Let $L$ and $M$ be languages over alphabet. Then $L \cap M$ is the language that contains all strings that are in both $L$ and $M$.

3. Let $L$ be a language over alphabet. Then $\overline{L}$, the complement of $L$, is the set of strings in that are not in $L$. 
It turns out that the regular languages are closed under all three of the boolean operations.

**Homomorphism:**

A string homomorphism is a function on strings that works by substituting a particular string for each symbol.

**Example 4.13**: The function $h$ defined by $h(0) = ab$ and $h(1) = \epsilon$ is a homomorphism. Given any string of 0's and 1's, it replaces all 0's by the string $ab$ and replaces all 1's by the empty string. For example, $h$ applied to the string 0011 is $abab$.  

Formally, if $h$ is a homomorphism on alphabet $\Sigma$, and $w = a_1a_2\cdots a_n$ is a string of symbols in $\Sigma$, then $h(w) = h(a_1)h(a_2)\cdots h(a_n)$. That is, we apply $h$ to each symbol of $w$ and concatenate the results, in order. For instance, if $h$ is the homomorphism in Example 4.12, and $w = 0011$, then $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\epsilon)(\epsilon) = abab$, as we claimed in that example.

Further, we can apply a homomorphism to a language by applying it to each of the strings in the language. That is, if $L$ is a language over alphabet $\Sigma$, and $h$ is a homomorphism on $\Sigma$, then $h(L) = \{h(w) \mid w \text{ is in } L\}$. For instance, if $L$ is the language of regular expression $10^*1$, i.e., any number of 0's surrounded by single 1's, then $h(L)$ is the language $(ab)^*$. The reason is that $h$ of Example 4.13 effectively drops the 1's, since they are replaced by $\epsilon$, and turns each 0 into $ab$. The same idea, applying the homomorphism directly to the regular expression, can be used to prove that the regular languages are closed under homomorphisms.

For Examples Refer class Notes.
Inverse Homomorphism:

Homomorphisms may also be applied "backwards," and in this mode they also preserve regular languages. That is, suppose $h$ is a homomorphism from some alphabet $\Sigma$ to strings in another (possibly the same) alphabet $T$. Let $L$ be a language over alphabet $T$. Then $h^{-1}(L)$, read "$h$ inverse of $L$," is the set of strings $w$ in $\Sigma^*$ such that $h(w)$ is in $L$. Figure 4.5 suggests the effect of a homomorphism on a language $L$ in part (a), and the effect of an inverse homomorphism in part (b).

![Diagram](image)

(a)

![Diagram](image)

(b)

Figure 4.5: A homomorphism applied in the forward and inverse direction

For Examples Refer class Notes.
Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 4

DFA Minimization

Decision Properties of Regular Languages:

- In this section we consider how one answers important questions about regular languages.

- First, we must consider what it means to ask a question about a language.

- The typical language is infinite, so you cannot present the strings of the language to someone and ask a question that requires them to inspect the infinite set of strings.

- Rather, we present a language by giving one of the finite representations for it that we have developed: a DFA, an NFA, an €-NFA, or a regular expression.

- Of course the language so described will be regular, and in fact there is no way at all to represent completely arbitrary languages.

- However, for many of the questions we ask, algorithms exist only for the class of regular languages. The same questions become "undecidable" (no algorithm to answer them exists) when posed using more "expressive" notations (i.e., notations that can be used to express a larger set of languages) than the representations we have developed for the regular languages.

- We begin our study of algorithms for questions about regular languages by reviewing the ways we can convert one representation into another for the same language.

- In particular, we want to observe the time complexity of the algorithms that perform the conversions. We then consider some of the fundamental questions about languages:
1. Is the language described empty?

2. Is a particular string $w$ in the described language?

3. Do two descriptions of a language actually describe the same language?

This question is often called "equivalence" of languages.

**Testing Emptiness of Regular Languages**

- At first glance the answer to the question "is regular language $L$ empty?" is obvious: $\phi$ is empty, and all other regular languages are not.

- The problem is not stated with an explicit list of the strings in $L$. Rather, we are given some representation for $L$ and need to decide whether that representation denotes the language $\phi$.

- If our representation is any kind of finite automaton, the emptiness question is whether there is any path whatsoever from the start state to some accepting state.

- If so, the language is nonempty, while if the accepting states are all separated from the start state, then the language is empty.

- Deciding whether we can reach an accepting state from the start state is a simple instance of graph-reachability, similar in spirit to the calculation of the $\varepsilon$-closure.
1. \( R = R_1 + R_2 \). Then \( L(R) \) is empty if and only if both \( L(R_1) \) and \( L(R_2) \) are empty.

2. \( R = R_1 R_2 \). Then \( L(R) \) is empty if and only if either \( L(R_1) \) or \( L(R_2) \) is empty.

3. \( R = R_1^* \). Then \( L(R) \) is not empty; it always includes at least \( \epsilon \).

4. \( R = (R_1) \). Then \( L(R) \) is empty if and only if \( L(R_1) \) is empty, since they are the same language.

Testing Membership in a Regular Language

- The next question of importance is, given a string \( w \) and a regular language \( L \), is \( w \) in \( L \). While \( w \) is represented explicitly, \( L \) is represented by an automaton or regular expression.

- If \( L \) is represented by a DFA, the algorithm is simple. Simulate the DFA processing the string of input symbols \( w \), beginning in the start state.

- If the DFA ends in an accepting state, the answer is "yes"; otherwise the answer is "no." This algorithm is extremely fast. If \( |w| = n \), and the DFA is represented by a suitable data structure, such as a two-dimensional array that is the transition table, then each transition requires constant time, and the entire test takes \( O(n) \) time.

- If \( L \) has any other representation besides a DFA, we could convert to a DFA and run the test above. That approach could take time that is exponential in the size of the representation, although it is linear in \( |w| \).

- However, if the representation is an NFA or \( \epsilon \)-NFA, it is simpler and more efficient to simulate the NFA directly. That is, we process
symbols of \( w \) one at a time, maintaining the set of states the NFA can be in after following any path labeled with that prefix of \( w \).

- If \( w \) is of length \( n \), and the NFA has \( s \) states, then the running time of this algorithm is \( O(ns^2) \). Each input symbol can be processed by taking the previous set of states, which numbers at most \( s \) states, and looking at the successors of each of these states.

- We take the union of at most \( s \) sets of at most \( s \) states each, which requires \( O(s^2) \) time.

- If the NFA has \( \varepsilon \)-transitions, then we must compute the \( \varepsilon \)-closure before starting the simulation. Then the processing of each input symbol \( a \) has two stages, each of which requires \( O(s^2) \) time.

- First, we take the previous set of states and find their successors on input symbol \( a \).

- Next, we compute the \( \varepsilon \)-closure of this set of states. The initial set of states for the simulation is the \( \varepsilon \)-closure of the initial state of the NFA.

- Lastly, if the representation of \( L \) is a regular expression of size. We can convert to an NFA with at most \( 2s \) states, in \( O(s) \) time. We then perform the simulation above, taking \( O(ns^2) \) time on an input \( w \) of length \( n \).

**Minimization of DFA's**

- Another important consequence of the test for equivalence of states is that we can "minimize" DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language.

- Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

  1. First, eliminate any state that cannot be reached from the start state.
2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent. Theorem, below, shows that we can always make such a partition.

Refer Class Notes for Examples.

Moore and Mealy Machine

- FA is the mathematical model of a machine and is defined by a 5-tuple \( (Q, \Sigma, \delta, q_0, F) \) which does not include the information about output.

- After reading a string if FA resides in final state, it says that the string is “accepted” by FA else it says that, the string is “rejected”.

- But if we need to produce some more precise information and not only ‘accept’ or ‘reject’; there are two different types of machines, which can be formulated as FA with output, viz.


1. **Moore Machine:**

   - It is the machine with finite number of states and for which, the output symbol at a given time depends only upon the present state of the machine.

   - In Moore Machine, an output symbol is associated with each state. When the machine is in a particular state, it produces the output, irrespective of what the input on which the transition is made.
• For Moore Machine, If the length of the input sequence is n, then the length of the output sequence is n+1.

**Moore Machine is a six-tuple:**

\[ M=(Q, \Sigma, \Delta, \delta, \lambda, q_0) \]

Where,

- Q - Finite set of states
- \( \Sigma \) - Finite input alphabet
- \( \Delta \) - Finite output alphabet
- \( \delta \) - State function, \( \delta: Q \times \Sigma \rightarrow Q \)
- \( \lambda \) - Machine function, \( \lambda: Q \rightarrow \Delta \)
- \( q_0 \) - initial state of the machine. \( q_0 \in Q \)

2. **Mealy Machine:**

• It is the machine with finite number of states and for which, the output symbol at a given time is a function of the present input symbol as well as the present state of the machine.

• Thus, for this type of machine output depends on both current state and the current input symbol.

• For Mealy Machine, if the length of the input sequence is n, then the length of the output sequence is n.

**Mealy machine is denoted by a six-tuple:**

\[ M=(Q, \Sigma, \Delta, \delta, \lambda, q_0) \]

Where,

- Q - finite set of states
- \( \Sigma \) - finite input alphabet
- \( \Delta \) - finite output alphabet
- \( \delta \) - state function, \( \delta: Q \times \Sigma \rightarrow Q \)
- \( \lambda \) - machine function, \( \lambda: Q \rightarrow \Delta \)
- \( q_0 \) - initial state of the machine. \( q_0 \in Q \)
Procedure for transforming Moore Machine to Mealy Machine:
If \( M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0) \) is Moore Machine, then equivalent Mealy Machine is 

\( M_2 = (Q, \Sigma, \Delta, \delta, \lambda', q_0) \)

where,

\[ \lambda'(q,a) = \lambda(\delta(q,a)) \]

for all states \( Q \) and input symbols \( a \).

Procedure for transforming Mealy Machine to Moore Machine:  
If given Mealy Machine is, \( M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0) \) then, an equivalent Moore machine is:

\( M_2 = ([Q \times \Delta], \Sigma, \Delta, \delta', \Delta', [q_0, b_0]) \),

Where, 'b₀' is an arbitrarily selected member of \( \Delta \) and \( \delta'([q,b], a) = [\delta(q,a), \lambda(q,a)] \) and \( ([q,b]) = b \)

Problems:

University questions will be solved in Class:

1. Give Mealy and Moore machine for the following process:
   For input from \((0 + 1)^*\),
   - If input ends in 101 output = A
   - If input ends in 110 output = B
   - Otherwise output = C


2. Design Moore and Mealy machine for binary input sequence which produces an output A if 101 are recognized otherwise output B.

3. Construct a Mealy Machine which can output EVEN and ODD according as the total number of 1’s encountered is even or odd. The input symbols are 0 and 1.

   (Nov-2004)
4. Design Moore and Mealy Machine to find 1's complement of given binary number.  
   (May-2003)

5. Design Moore and Mealy Machine to increment binary number by 1

6. Design Moore and Mealy Machine to find 2's complement of given binary number.  

7. Design Moore and Mealy machine for a binary input string giving output as the remainder when divided by 3.

8. Design Moore Machine for the following process:  
   For input from binary (0+1)* print the residue modulo 3 of the input.  
   (Dec-1999)

9. Give Mealy and Moore machine for the following:  
   For input from ∑*, where ∑, print the residue modulo 5 of the input treated as a ternary  
   (base 3, with digits 0,1 and 2 number.  
   (May-2006)

10. Design a Moore Machine that will read sequences made up of letters a, e, l, o, u and  
    will give as output, same characters except when an 'i' is followed by 'e', it will be  
    changed to 'u'.

11. Design the Mealy machine for the above.

12. Design Moore Machine to convert each occurrence of 100 to 101.  
    (Dec-2000)

13. Design Mealy Machine to convert each occurrence of substring add by aba over  
    ∑={a,B}.  
    (Dec-2006, Dec-2000)

14. Design Moore Machine to convert each occurrence of 101 to 100.
15. Design Moore Machine to convert each occurrence of 1000 to 1001.  
   (Dec-2002, Jun-2008)

16. Design Moore Machine to convert each occurrence of 121 to 120.

17. Design Moore Machine to convert each occurrence of 121 to 021.

18. Design Moore and Mealy machines to convert substring 121 to 122 for strings of languages having $\Sigma = \{0, 1, 2\}$  
   (Dec-2002)

19. Design Mealy Machine to convert HEX numbers to OCTAL numbers.  
   (Dec-2005)

20. Design Moore Machine to convert HEX numbers to OCTAL numbers.  
   (Dec-2005)

21. Construct the Mealy Machine to accept the language $(0+1)*(00+11)$.  
   (Dec-2003)

   (Dec-2003)

23. Explain the equivalence of Moore and Mealy Machine. Design a Mealy Machine for the language $(0+1)*(00+11)$ and convert this Mealy machine to Moore Machine.  
   (May-2005)

24. Design a Moore Machine which counts the occurrence of substring aab in a long input string over $\{a, b\}$  
   (Nov-2004)
Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 5

Context–free Grammars

- We have seen that every finite automata $M$ accepts a language $L$, which is represented by $L(M)$.

- We have seen that a regular language can be described by a regular expression.

- We have seen that there are several languages which are not regular.
  
  1. $L_1 = \{a^p \mid p \text{ is a prime}\}$ is not regular.
  2. $L_2 = \{a^nb^n \mid n \geq 0\}$ is not regular.

- We have seen two ways of representing a language.
  
  1. Using finite automata.
  2. Using a regular expression.

- If a language is not-regular, it cannot be represented either using a FA or using a regular expression. Hence there was a need for representing such languages.

- Grammar is another approach for representing a language:
  
  1. In this approach, a language is represented using a set of equations.
  2. Equations are recursive in nature.
  3. Finite Automata have a set of states; a grammar has set of variables.
  4. Finite Automata are defined over an alphabet; a grammar is defined over a set of terminals.
  5. Finite Automata have a set of transitions; a grammar has a set of equations (Productions).

- For building any model, we should consider two aspects of the given grammar:
  
  1. The generative capacity of the grammar i.e., the grammar used should generate all and only the sentence of the language for which it is written.
  2. Grammatical constituents like terminals and non-terminals.

**Definition:** Grammar is used for specifying the syntax of a language and for representing a non-regular language.

  1. A parse structure grammar is denoted by a quadruple of the form $G = (V, T, P, S)$
Where,

\[ V \] – is a set of Variables.
\[ T \] – is a set of Terminals.
\[ P \] – is a set of Productions.
\[ S \] – is a special variable called that start symbol \( S \in V \).

**Notations:**

1. Terminals are denoted by lower case letters \( a, b, c \ldots \) or digits \( 0, 1, 2 \ldots \) etc.
2. Variables (Non-Terminals) are denoted by capital letters \( A, B \ldots V, W, X \ldots \).
3. A string of terminals or a word \( \omega \in L \) is represented using \( u, v, w, x, y, z \).
4. A sentential form is a string of terminals and variables and it is denoted by \( \alpha, \beta, \gamma \ldots \) etc.

**Variables:** Variables are those symbols that take part in the derivation of a sentence, but are not the part of derived sentence.

**Terminals:** Terminals are those symbols that are the part of derived sentence.

**The Language of A Grammar:**

- Every grammar generates a language. A word of a language is generated by applying productions a finite number of times.
- Derivation of a string should start from the start symbol and the final string should consist of terminals.
- If \( G \) is a grammar with start symbol \( S \) and set of terminals \( T \), then the language of \( G \) is the set:

\[
L(G) = \{ \omega \mid \omega \in T^* \text{ and } S \Rightarrow \omega \}
\]

The * represent that, production can be applied multiple time.

- A string can be derived from start symbol of the grammar, using the productions of the grammar.

Derivations are represented either in the:

1. Sentential Form OR
2. Parse Tree Form

**Sentential Form:**

Let us consider a grammar given below:

\[
S \rightarrow A1B \\
A \rightarrow 0A | \epsilon \\
B \rightarrow 0B | 1B | \epsilon
\]

Where Grammar \( G \) is given by \((V, T, P, S)\)

\[
V = \{S, A, B\} \\
T = \{0, 1\} \\
P = \{S \rightarrow A1B, A \rightarrow 0A | \epsilon, B \rightarrow 0B | 1B | \epsilon\} \\
S = \{S\}
\]

Let us try to generate the string ‘00101’ with above grammar.

There are two different derivations possible:

1. **Leftmost Derivation.**
2. **Rightmost Derivation.**

**Leftmost Derivation:** If at each step in a derivation, a production is applied to the leftmost variable (non-terminal), then the derivation is called as leftmost derivation.

Example:

\[
\begin{align*}
S & \rightarrow A1B & \text{(Start Variable)} \\
S & \rightarrow 0A1B & \text{(using } A \rightarrow 0A) \\
S & \rightarrow 00A1B & \text{(using } A \rightarrow 0A) \\
S & \rightarrow 001B & \text{(using } A \rightarrow \epsilon) \\
S & \rightarrow 0010B & \text{(using } B \rightarrow 0B) \\
S & \rightarrow 00101B & \text{(using } B \rightarrow 1B) \\
S & \rightarrow 00101 & \text{(using } B \rightarrow \epsilon)
\end{align*}
\]

**Rightmost Derivation:** If at each step in a derivation, a production is applied to the rightmost variable (non-terminal), then the derivation is called as rightmost derivation.

Example:
Parse Tree Form:

A set of derivations applied to generate a word can be represented using a tree. Such a tree is known as a parse tree. A parse tree is constructed with the following condition:

1. Root of the tree is represented by start symbol.
2. Each interior node is represented by a variable belonging to V.
3. Each leaf node is represented by a terminal or $\varepsilon$.

The parse tree is also created in two ways:

1. Using leftmost derivations.
2. Using rightmost derivations.

Problems:

University questions will be solved in Class:

1. For the grammar given below:
   \[ S \rightarrow A1B \]
   \[ A \rightarrow 0A \mid \varepsilon \]
   \[ B \rightarrow 0B \mid 1B \mid \varepsilon \]

   Give parse tree for leftmost and rightmost derivation of the string ‘1001’ and ‘00101’.

2. For the grammar given below:
   \[ S \rightarrow 0S1 \mid 01 \]

   Give derivation of ‘000111’.

3. Consider the grammar given as:
   \[ G = \{S, A\}, \{a, b\}, P, S\]
   Where P consists of –
   \[ S \rightarrow aAS \mid a \]
A $\rightarrow$ SbA | SS | ba  
Derive ‘aabaaabbaaa’ using the leftmost derivation and rightmost derivation. 

Derive ‘aabbaa’ using the leftmost derivation and rightmost derivation.

4. Consider the following grammar:
   \[ S \rightarrow aB | bA \]
   \[ A \rightarrow a | aS | bAA \]
   \[ B \rightarrow b | bS | aBB \]
Find the leftmost and rightmost derivation for the string: ‘bbaaba’, ‘aaabbabbbba’, ‘aaabbb’ and ‘aaaba’.

5. Consider the following grammar:
   \[ S \rightarrow XbbaaX | aX \]
   \[ X \rightarrow Xa | Xb | \varepsilon \]
Construct leftmost derivation and rightmost derivation for the string ‘abaabb’.

**Context Free Grammar (CFG):**

A context free grammar $G$ is a quadruple ($V$, $T$, $P$, $S$)  
Where,
   \[ V \] – is a set of variable.  
   \[ T \] – is a set of terminals.  
   \[ P \] – is a set of productions.  
   \[ S \] – is a start symbol $S \in V$.  

A production is of the form  
\[ V_i \rightarrow \alpha \] where $V_i \in V$ and $\alpha$ is a string of terminals and variables.

OR

**Definition:** A Grammar is said to be CFG if all the production are of the form:  
\[ A \rightarrow \alpha \]
Where,  
$A$ - is a variable and  
$\alpha$ – is sentential form. (Sentential forms means, the combination of Variables and Terminals).
Problems:

University questions will be solved in Class:

1. Let \( G = (V, T, P, S) \) be the CFG having following set of productions. Derive the string ‘aabbaa’ using leftmost derivation and rightmost derivation.
   
   \[
   S \rightarrow aAS | a \\
   A \rightarrow SbA | SS | ba
   \]

2. For the grammar given below:
   
   \[
   E \rightarrow E + T | T \\
   T \rightarrow T * F | F \\
   F \rightarrow (E) | a | b
   \]

Give the derivation of \((a + b)^* a + b\).

3. Write CFG to generate:
   
   i. Set of all strings that start with ‘a’ over \( \Sigma = \{a, b\} \)
   ii. Set of all strings that start and end with different symbol over \( \{0, 1\} \)
   iii. Set of all strings that start and end with same symbol over \( \{0, 1\} \)
   iv. Set of all strings that contain over \( \Sigma = \{a, b\} \):
      
      i. Atleast 3 a’s
      ii. Exactly 2 a’s
      iii. Atmost 1 a

4. Drive the grammar for the given languages:
   
   i. \( L = \{\epsilon, a, aa, aaa \ldots\} \)
   ii. \( L = \{a, aa, aaa,aaaa \ldots\} \)
   iii. \( L = \{b, ab, aab, aaab \ldots\} \)

5. Drive the grammar for the given languages:
   
   i. \( L = \{\omega \in \{a, b\}^*\} \)
   ii. \( L = \{\epsilon, ab, aabb, \ldots, a^nb^n\} \)
   iii. \( L = \{ab, aabb, \ldots, a^nb^n\} \)
   iv. \( L = \{\omega \in \{a, b\}^* | \omega \text{ is a palindrome of odd length}\} \)
   v. \( L = \{\omega \in \{a, b\}^* | \omega \text{ is a palindrome of even length with } |\omega|>0\} \)
   vi. \( L = \{\omega \in \{a, b\}^* | \omega \text{ is a palindrome of even and odd length with } |\omega|>0\} \)

Rules for Grammar:
1. **Union rule for Grammar:**
   If a language $L_1$ is generated by a grammar with start symbol $S_1$ and $L_2$ is generated by a grammar with start symbol $S_2$ then the union of the languages $L_1 \cup L_2$ can be generated with start symbol $S$, where
   
   $$ S \rightarrow S_1 \mid S_2 $$

   **Example:**
   Let the language $L_1$ and $L_2$ are given as below:
   
   $$ L_1 = \{a^n \mid n>0\} $$
   $$ L_2 = \{b^n \mid n>0\} $$

   Productions for $L_1$ are:
   
   $$ S_1 \rightarrow aS \mid a $$

   Productions for $L_2$ are:
   
   $$ S_2 \rightarrow bS \mid b $$

   Then the productions for $L = L_1 \cup L_2$ can be written as:
   
   $$ S \rightarrow S_1 \mid S_2 $$
   $$ S_1 \rightarrow aS \mid a $$
   $$ S_2 \rightarrow bS \mid b $$

2. **Concatenation rule for Grammar:**
   If a language $L_1$ is generated by a grammar with start symbol $S_1$ and $L_2$ is generated by a grammar with start symbol $S_2$ then the concatenation (product) of the languages $L_1.L_2$ can be generated with start symbol $S$, where

   $$ S \rightarrow S_1S_2 $$

   **Example:**
   Let the language $L_1$ and $L_2$ are given as below:
   
   $$ L_1 = \{a^n \mid n>0\} $$
   $$ L_2 = \{b^n \mid n>0\} $$
Productions for $L_1$ are:
   \[ S \rightarrow aS | a \]

Productions for $L_2$ are:
   \[ S \rightarrow bS | b \]

Then the productions for $L = L_1 \cdot L_2$ can be written as:
   \[ S \rightarrow S_1 \mid S_2 \\
   S_1 \rightarrow aS \mid a \\
   S_2 \rightarrow bS \mid b \]

**Problems:**

**University questions will be solved in Class:**

1. Give a context free grammar for the following language:
   \[ 0(0+1)^*01(0+1)^*1 \]

2. Construct the context free grammar corresponding to the regular expression:
   \[ R = (0+1)^1*(1+(01)^*) \]

3. Give the CFG for $L = \{ a^ib^j \mid i \leq j \leq 2i, i \geq 1 \}$

4. Give the CFG for $L = \{ a^ib^jc^q \mid i+j=q ; (i,j) \geq 1 \}$

5. Find context free grammars generating each of these languages:
   a. $L = \{ a^ib^ck \mid i=j+k \}$
   b. $L = \{ a^ib^ck \mid j=i+k \}$
   c. $L = \{ a^ib^ck \mid i=j \text{ or } j=k \}$

6. Give the context free grammar for the following languages:
   a. $(011+1)^*(01)^*$
   b. $0^i1^i+0^k \text{ where } i,k \geq 0$
7. Show that the languages:
   a. \( L = \{ a^i b^i c^i \mid i \geq 1 \} \) and
   b. \( L = \{ a^i b^i c^j \mid i, j \geq 1 \} \) are context free languages.


9. Give CFG for all strings with at least two 0's, \( \sum = \{0, 1\} \)

10. Give CFG for set of odd length strings in \( \{0, 1\}^* \) with middle symbol '1'.

11. Give CFG for set of even length strings in \( \{a, b, c, d\}^* \) with two middle symbol equal.

12. Give CFG for \( L = \{ x \mid x \text{ contains equal number of a's and b's} \} \)

13. Give CFG for strings in \( ab^* \).

14. Give CFG for strings in \( a^* b^* \).

15. Find CFG for generating: (Dec-06, May-09, Dec-09)
   a. String containing alternate sequence of 0's and 1's, \( \sum = \{0, 1\} \)
   b. The string containing no consecutive b's but a's can be consecutive
   c. The set of all string over alphabet \( \{a, b\} \) with exactly twice as many a's as b's.
   d. Language having number of a's greater then number of b's

16. Write CFG for the language
   \( \sum = \{a, b\} \) number of a's is a multiple of 3.

Ambiguous Grammar:
**Definition:** A grammar is said to be ambiguous if the language generated by the grammar contains some string that has two different parse trees.

Example: Let us consider the grammar given below:

\[ E \rightarrow E + E \mid a \mid b \]

A string \((a + b + a)\) is generated by the given grammar.

![Parse Trees](image)

Fig: Parse Trees considered for ambiguity
• The grammar generates \((a + b + a)\) in two different ways. The two derivations are shown in Fig (a) and (b).

• The first derivation Fig (a) says that \((b + a)\) is evaluated first and then the evaluated value is added to \(a\). Thus the right side + operator gets a precedence over the left side + operator. The expression \((a + b + a)\) is treated as \((a + (b + a))\).

• The second derivation Fig (b) says that \((a + b)\) is evaluated first and then the evaluated value is added to \(a\). Thus the left side + operator gets a precedence over the right side + operator. The expression \((a + b + a)\) is treated as \(((a + b) + a)\).

Removing ambiguity:

• There is no general rule for removing ambiguity from CFG. Removing ambiguity from a grammar involves rewriting of grammar so that there is only one derivation tree for every string belonging to \(L(G)\) i.e., language generated by grammar \(G\).

  Ambiguity from the grammar

  \[
  E \to E + E \mid a \mid b
  \]

  Can be removed by strictly assigning higher precedence to left side + operator over right side + operator. This will mean evaluation of an expression of the form \((a + b + a)\), from left to right.

  • This property can be incorporated in the grammar itself by suitably modifying the grammar.

  • Parse tree of Fig (b) is based on left to right evaluation.

  • Left to right evaluation in Grammar can be enforced by introducing one more variable \(T\). Variable \(T\) cannot be broken by + operator.

  • An unambiguous grammar for the grammar is given as:

  \[
  E \to E + T \mid T \\
  T \to a \mid b
  \]
A production of the form $E \rightarrow E + T$ provides a binding. $E + T$, implies that $E$ must be evaluated first before an atomic $T$ can be added to it. $E$ can be broken down in $E + T$ but $T$ cannot be broken further. This ensures higher precedence to left side $+$ operator over right side $+$ operator.

- Parse tree in Fig. given below is based on unambiguous grammar for the string $(a + b + a)$.

Fig: A Parse tree for $(a + b + c)$ using an unambiguous grammar

Problems:

University questions will be solved in Class:

1. Consider the grammar:
   
   $E \rightarrow E + E \mid E * E \mid (E) \mid I$
   
   $I \rightarrow a \mid b$

   Show that the grammar is ambiguous.
   Remove ambiguity.
2. Show that the CFG given below. Which generates all strings of balanced parentheses is ambiguous. Give an equivalent unambiguous grammar.
   \[ S \rightarrow SS \mid (S) \mid \varepsilon \]

3. Write an unambiguous CFG for arithmetic expressions with operators:
   +, *, /, ^, unary minus and operand a, b, c, d, e and f.
   Also, if should be possible to generate brackets with your grammar.
   Derive \((a+b)^c/d\) from your grammar. (Dec 2005)

4. Is the following CFG ambiguous?
   \[ S \rightarrow aB \mid ab \]
   \[ A \rightarrow aAB \mid a \]
   \[ B \rightarrow ABb \mid b \]
   If so, show multiple derivation trees for the same string.

5. Is the following CFG ambiguous?
   \[ G = (\{ S, A \}, \{ a, b \}, P, S) \]
   Where, P consists of
   \[ S \rightarrow aAS \mid a \]
   \[ A \rightarrow SbA \mid SS \mid ba \]

6. Consider the grammar having productions:
   \[ S \rightarrow aS \mid \varepsilon \]
   \[ S \rightarrow aSbS \]
   The grammar is ambiguous.
   i. Show in particular that the string ‘aab’ has two parse trees.
   ii. Find an unambiguous grammar for the same.

7. Let \(G\) be the grammar
   \[ S \rightarrow aB \mid bA \]
   \[ A \rightarrow a \mid aS \mid bAA \]
   \[ B \rightarrow b \mid bS \mid aBB \]
   For string ‘aaabbabba’ find
   i. Left most derivation
   ii. Right most derivation
   iii. Parse Tree

CHOPRA ACADEMY
@Bandra:9867 843 854 @Thane:9867 843 853 @Vashi :9867 843 852
For Doubts Contact Ganesh Sir@83 84 82 05 75
iv. Is the grammar unambiguous? (Dec-2009)

8. In each case, show that the grammar is ambiguous, and find the equivalent unambiguous grammar:
   i. \[ S \rightarrow SS \mid a \mid b \]
   ii. \[ S \rightarrow ABA, A \rightarrow aA \mid \varepsilon, B \rightarrow bB \mid \varepsilon \]
   iii. \[ S \rightarrow aSb \mid aaSb \mid \varepsilon \]

9. Consider the grammar:
   \[ G = (\{V = \{E, F\}, \{T = \{a, b, \_\}\}, E, P) \]
   Where P consists of rules:
   \[ E \rightarrow F - E \]
   \[ F \rightarrow a \]
   \[ E \rightarrow E - F \]
   \[ F \rightarrow b \]
   \[ E \rightarrow F \]
   i. Show that G is ambiguous
   ii. Remove the ambiguity.

10. Test whether the following grammars are ambiguous:
    i. \[ S \rightarrow 0S1S \mid 1S0S \mid \varepsilon \]
    ii. \[ S \rightarrow AA, A \rightarrow aAb \mid bAa \mid \varepsilon \]

Simplification of CFG:

- A grammar written in a simple form is easy to analyse. Certain restrictions are imposed on simplified grammar.

- Simplification of CFG involves transforming CFG into an equivalent form that satisfies certain restrictions on its form. A CFG can be simplified by eliminating:
  1. Useless symbols.
  2. \( \varepsilon \) - Productions.
  3. Unit productions.
1. **Elimination of useless symbols:** A grammar may contain symbols and productions which are not useful for derivation of strings. Two types of symbols are useless.
   a. **Non – generating symbols**
   b. **Non reachable symbol.**

   **Non – generating symbols:** A symbol $X \in V$ (Set of variables) is a generating symbol if :
   \[
   X \xrightarrow{G}^{*} w
   \]
   Where, $w \in T^*$ i.e. every variable must generate a string of terminals.

   Example: Consider the following Grammar:
   \[
   S \rightarrow Aa \mid Bb \mid a \mid b \\
   A \rightarrow Aa \mid a \\
   B \rightarrow bB
   \]
   Requires simplification as the symbol $B$ is non-generating. Only production for $B$ is
   \[
   B \rightarrow bB
   \]
   and it cannot generate a string of terminals. The Grammar can be simplified by deleting every production containing the useless symbol $B$. A simplified grammar is given as:
   \[
   S \rightarrow Aa \mid a \mid b \\
   A \rightarrow Aa \mid a
   \]
   A grammar containing a non-generating symbol $V_i$ should be simplified by deleting every production containing the non-generating symbol $V_i$.

   **Finding non-generating symbols:**

   There are two rules for finding a set of generating symbols for the given grammar.
   
   1. Every symbol in $T$ (terminal) is generating.
   2. If there is a production $A \rightarrow \alpha$ and every symbol in $\alpha$ is generating, then $A$ is generating. Where, $\alpha \in (V+T)^*$
A symbol not in a set of generating symbols is said to be non-generating.

**Problems on Non-generating symbols:**

**University questions will be solved in Class:**

1. Find non-generating symbols in the grammar given below
   
   \[ \begin{align*}
   S & \rightarrow AB \mid CA \\
   B & \rightarrow BC \mid AB \\
   A & \rightarrow a \\
   C & \rightarrow aB \mid b
   \end{align*} \]

2. Find non-generating symbols in the grammar given below
   
   \[ \begin{align*}
   S & \rightarrow aAa \\
   A & \rightarrow Sb \mid bCC \\
   C & \rightarrow abb \\
   E & \rightarrow aC
   \end{align*} \]

3. Find non-generating symbols in the grammar given below
   
   \[ \begin{align*}
   S & \rightarrow aAa \\
   A & \rightarrow Sb \mid bCC \mid DaA \\
   C & \rightarrow abb \mid DD \\
   E & \rightarrow aC \\
   D & \rightarrow aDa
   \end{align*} \]

**Non-reachable Symbols:**

A symbol X is reachable if it can be reached from the start symbol S, i.e. if:

\[ S \xrightarrow[G]{*} \alpha \]

and \( \alpha \) contains a variable X then X is reachable.

A grammar containing a non-reachable symbol \( V_i \) should be simplified by deleting every production containing the non-reachable symbol \( V_i \).
Finding non-reachable symbols:

Non-reachable symbols can be located with the help of a dependently graph. A variable \( X \) is said to be dependent on \( S \) if there is a production:

\[
S \rightarrow \alpha_1 X \alpha_2
\]

We must draw a dependency graph for all productions. If there is no path from the start symbol \( S \) to a variable \( X \), then \( X \) is non-reachable.

Problems on Non-reachable symbols:

University questions will be solved in Class:

1. Eliminate non-reachable symbols from the given grammar:
   \[
   S \rightarrow aAa \\
   A \rightarrow Sb \mid bCC \\
   C \rightarrow abb \\
   E \rightarrow aC
   \]

2. Eliminate non-reachable symbols from the given grammar:
   \[
   S \rightarrow aBa \mid BC \\
   A \rightarrow aC \mid BCC \\
   C \rightarrow a \\
   B \rightarrow bCC \\
   D \rightarrow E \\
   E \rightarrow d
   \]

3. Eliminate non-reachable symbols from the given grammar:
   \[
   S \rightarrow aAa \\
   A \rightarrow bBB \\
   B \rightarrow ab \\
   C \rightarrow aB
   \]

4. Eliminate non-reachable symbols from the given grammar:
2. **Elimination of \( \varepsilon \)-productions:**

A production of the form \( A \rightarrow \varepsilon \), is called a null productions or \( \varepsilon \)-production. For every context free grammar \( G \) with \( \varepsilon \)-productions, we can find a context-free grammar \( G_1 \) having no \( \varepsilon \)-productions such that

\[
L(G_1) = L(G) - \{ \varepsilon \}
\]

The procedure for finding \( G_1 \) is as follows:

**Step 1:** Find nullable variables.

**Step 2:** Addition of productions with nullable variables removed.

**Step 3:** Remove \( \varepsilon \)-productions.

---

**Problems on elimination of \( \varepsilon \)-productions:**

**University questions will be solved in Class:**

1. Eliminate the \( \varepsilon \)-production from the grammar given below:
   - \( S \rightarrow aS \)
   - \( S \rightarrow \varepsilon \)

2. Eliminate the \( \varepsilon \)-production from the grammar given below:
   - \( S \rightarrow ABA \)
   - \( A \rightarrow \varepsilon \)
   - \( B \rightarrow \varepsilon \)

3. Eliminate the \( \varepsilon \)-production from the grammar given below:
   - \( S \rightarrow aS | AB \)
   - \( A \rightarrow \varepsilon \)
3. Elimination of Unit Productions:

A production of the form $A \rightarrow B$ is known as the unit production where $A$ and $B$ are variables.

For every context free grammar $G$ with unit productions, we can find a context free grammar $G_1$ having no unit productions such that

$$L(G_1) = L(G)$$

The procedure for finding $G_1$ is as follows:

- The technique is based on expansion of unit production until it disappears. This technique works in most of the cases. This technique does not work if there is a cycle of unit productions such as
  
  $A_1 \rightarrow A_2$
  $A_2 \rightarrow A_3$
  $A_3 \rightarrow A_4$
  $A_4 \rightarrow A_1$

- The steps for elimination of unit productions are as follows:

  
  STEP 1: Add all non-unit production of $G$ to $G_1$.

  STEP 2: Locate every pair of variables $(A_i, A_j)$ such that

  $$A_i \xrightarrow{G} A_j$$
STEP 3: From pairs constructed in step 2, we can construct a chain like \( A_1 \rightarrow A_2 \ldots \rightarrow A_j \rightarrow \alpha \) is a non-unit production.

Each variable \( A_i \) to \( A_j \) will derive \( \alpha \).

**Problems on Elimination of Unit Productions:**

**University questions will be solved in Class:**

1. Eliminate unit productions form:
   
   \[
   S \rightarrow ABA \mid BA \mid AA \mid AB \mid A \mid B \\
   A \rightarrow aA \mid a \\
   B \rightarrow bB \mid b
   \]

2. Eliminate unit productions from the grammar:
   
   \[
   E \rightarrow E + T \mid T \\
   T \rightarrow T \ast F \mid F \\
   F \rightarrow (E) \mid I \\
   I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1
   \]

3. Simplify the following grammar:
   
   \[
   S \rightarrow ASB \mid \varepsilon \\
   A \rightarrow aAS \mid a \\
   B \rightarrow SbS \mid A \mid bb
   \]

4. Simplify the following grammar:
   
   \[
   S \rightarrow 0A0 \mid 1B1 \mid BB \\
   A \rightarrow C \\
   B \rightarrow S \mid A \\
   C \rightarrow S \mid \varepsilon
   \]

5. Simplify the following grammar:
   
   \[
   S \rightarrow Ab \\
   A \rightarrow a \\
   B \rightarrow C \mid b \\
   C \rightarrow D
   \]
D $\rightarrow$ E
E $\rightarrow$ a

6. Find a reduced grammar equivalent to:
   \[ S \rightarrow aC \mid SB \]
   \[ A \rightarrow bSCa \]
   \[ B \rightarrow aSB \mid bBC \]
   \[ C \rightarrow aBC \mid ad \]

Normal forms for CFG:

Productions in G, satisfying certain restrictions are said to be in normal form. There are two normal forms for CFG.
   1. Chomsky Normal Form (CNF)
   2. Greibach Normal Form (GNF)

1. Chomsky Normal Form (CNF): (Dec-2009)

A context free grammar (CFG) without $\varepsilon$-production is said to be in CNF if every production is of the form:

1. \[ A \rightarrow BC, \text{ where } A, B, C \in V. \]
2. \[ A \rightarrow a, \text{ where } A \in V \text{ and } a \in T. \]

The grammar should have no useless symbols. Every CFG without $\varepsilon$-productions can be converted into an equivalent CNF form.

Algorithm for CFG to CNF Conversion:

1. Eliminate $\varepsilon$-productions, unit productions and useless symbols from the grammar.

2. Every variable deriving a string of length 2 or more should consist only of variables. i.e. every production of the form \[ A \rightarrow \alpha \] with $|\alpha| \geq 2$, $\alpha$ should consist only of variables.
Examples: Consider a production \( A \rightarrow V_1 V_2 a V_3 b V_4 \).

Terminal symbols ‘a’ and ‘b’ can be removed by rewriting the production
\[
\begin{align*}
A & \rightarrow V_1 V_2 a V_3 b V_4 \\
A & \rightarrow V_1 V_2 C_a V_3 C_b V_4
\end{align*}
\]

And adding two productions
\[
\begin{align*}
C_a & \rightarrow a \\
C_b & \rightarrow b
\end{align*}
\]

3. Every production deriving 2 or more variables (\( A \rightarrow \alpha \) with \( |\alpha| \geq 3 \)) can be broken down into a cascade of productions with each deriving a string of two variables.

Examples: Consider a production \( A \rightarrow X_1 X_2 \ldots X_n \) where \( n \geq 3 \) and as \( X_i \)'s are variables. The production \( A \rightarrow X_1 X_2 \ldots X_n \) should be broken down as given below:

\[
\begin{align*}
A & \rightarrow X_1 C_1 \\
C_1 & \rightarrow X_2 C_2 \\
C_2 & \rightarrow X_3 C_3 \\
& \vdots \\
C_{n-2} & \rightarrow X_{n-1} X_n
\end{align*}
\]
each with two variables on the right.

Problems on CFG to CNF conversion:

University questions will be solved in Class:

1. Find the CNF equivalent to:
   \[
   \begin{align*}
   S & \rightarrow a AbB \\
   A & \rightarrow a A \\
   B & \rightarrow b B | b
   \end{align*}
   \]

2. Convert the grammar given below to its equivalent CNF:
   \[
   \begin{align*}
   S & \rightarrow P Q P \\
P & \rightarrow 0 P | \varepsilon \\
Q & \rightarrow 1 Q | \varepsilon
   \end{align*}
   \]

3. Check whether the given grammar is in CNF:
S \rightarrow bA \mid aB \\
A \rightarrow bAA \mid aS \mid a \\
B \rightarrow aBB \mid bS \mid b \\

If it is not in CNF, find the equivalent CNF.

4. Design a CNF grammar for the set of strings of balanced parenthesis.

5. Convert the following grammar to CNF:

\begin{align*}
S & \rightarrow \text{Aba} \\
S & \rightarrow \text{aab} \\
B & \rightarrow \text{Ac} \\
\end{align*}

6. Convert the following grammar to CNF:

\begin{align*}
S & \rightarrow \text{AACD} \\
A & \rightarrow \text{aAb} \mid \varepsilon \\
C & \rightarrow \text{aC} \mid a \\
D & \rightarrow \text{aDa} \mid \text{bDb} \mid \varepsilon \\
\end{align*}

7. Given a CFG G, find G’ in CNF generating \(L(G) - \varepsilon\)  
(May-2006, May-2009)

\begin{align*}
S & \rightarrow \text{ASB} \mid \varepsilon \\
A & \rightarrow \text{AaS} \mid a \\
B & \rightarrow \text{SbS} \mid A \mid bb \\
\end{align*}

8. Convert the given grammar to CNF

\begin{align*}
S & \rightarrow \text{aSB} \mid \text{aA} \\
A & \rightarrow \text{Aa} \mid \text{Sa} \mid a \\
\end{align*}


A Context free grammar \(G = (V, T, P, S)\) is said to be in GNF if every production is of the form:

\[ A \rightarrow a\alpha \]
Where, a ∈ T is a terminal and α is a string of zero or more variables. The language L(G) should be without \( \varepsilon \). Right hand side of each production should start with a terminal followed by a string of non-terminals of length zero or more.

**Removing Left Recursion:**

Elimination of left recursion is an important step in algorithm used in conversion of a CFG into GNF form.

**Left recursive grammar:** A production of the form \( A \rightarrow A\alpha \) is called left recursive as the left hand side variable appears as the first symbol on the right hand side.

**Language generated by left recursive grammar:** Let us consider a CFG containing productions of the form

\[
A \rightarrow A\alpha \\
\text{[Left recursive ]}
\]

\[
A \rightarrow \beta \\
\text{[For termination of recursion ]}
\]

The language generated by above productions is:

\[
A \rightarrow A\alpha \]

\[
A \rightarrow A\alpha\alpha \\
\text{... using } A \rightarrow A\alpha
\]

\[
A \rightarrow A\alpha\alpha\alpha \\
\text{... using } A \rightarrow A\alpha
\]

\[
A \rightarrow A\alpha^n \\
\text{... using } A \rightarrow A\alpha
\]

\[
A \rightarrow \beta\alpha^n \\
\text{... using } A \rightarrow \beta
\]

**Right recursive grammar for \( \beta\alpha^n \):** A right recursive grammar for \( \beta\alpha^n \) can be written as:

\[
A \rightarrow \beta B | \beta
\]

\[
B \rightarrow \alpha B | \alpha
\]

Thus a left recursive grammar

\[
A \rightarrow A\alpha | \beta
\]
can be written using a right recursive grammar as:

\[
A \rightarrow \beta B | \beta \\
B \rightarrow \alpha B | \alpha
\]

**Problems on conversion of Left Recursive grammar to Right Recursive grammar:**

1. \( A \rightarrow Aa | b \)
2. \( A \rightarrow Aa | b | c \)
3. \( A \rightarrow ABC | BC \)
4. \( A \rightarrow ABC | DA | EC \)
5. \( S \rightarrow S10 | 0 \)

**Algorithm for conversion from CFG to GNF:**

1. Eliminate \( \varepsilon \)-productions, unit productions and useless symbols from the grammar.

2. In production of the form \( A \rightarrow X_1X_2...X_i...X_n \), other than \( X_1 \), every other symbol should be a variable. \( X_1 \) could be a terminal.
   
   Example: consider a production
   
   \[
   A \rightarrow V_1V_2aV_3bV_4
   \]
   
   as
   
   \[
   A \rightarrow V_1V_2V_cV_3V_bV_4
   \]
   
   and adding two productions
   
   \[
   C_a \rightarrow a
   \]
   
   and
   
   \[
   C_b \rightarrow b
   \]
   
   Thus, at the end of step 2 all productions must be of the forms:
   
   \[
   A \rightarrow \alpha \\
   A \rightarrow a \\
   A \rightarrow a\alpha
   \]
   
   Where, ‘\( a \)’ is a terminal and \( \alpha \) is a string of non-terminals.

3. Rename variables as \( A_1, A_2, A_3 ... A_n \) to create \( A \)-productions.
   
   Example: Consider a grammar given below
S → aXSY | YSX | b

The variables S, X, and Y can be renamed as A1, A2, and A3 respectively. Then the becomes

A1 → aA1A2A3 | A1A2A3 | b

4. Modify the productions to ensure that if there is a production Ai > Ajα then i should be ≤ j. If there is a production Ai → Ajα with i > j, then we must generate productions substituting for Aj.

5. Repeating step 4, several times will guarantee that for every production Ai → Ajα, i ≤ j.

6. Remove left recursion from every production of the form Ak → Akα. B-productions should be added to remove left recursion.

7. Modify Ai-production to the form Ai → aα, where 'a' is a terminal and α is a string of non-terminals.

8. Modify Bi-productions to the form Bi → aα, where 'a' is a terminal and α is a string of non-terminals.

Problems on CFG to GNF conversion:

University questions will be solved in Class:

1. Construct a grammar in GNF which is equivalent to the grammar

   S → AA | a
   A → SS | b

2. Find the grammar in GNF for the given CFG

   E → E + T | T
   T → T * F | F
   F → (E) | a

3. Give the GNF for following CFG
4. Reduce the following grammar to GNF
   \[ S \rightarrow AB \]
   \[ A \rightarrow BS \mid b \]
   \[ B \rightarrow SA \mid a \]

5. Convert the following grammar to Greibach Normal Form (GNF)
   \[ S \rightarrow BS \]
   \[ S \rightarrow Aa \]
   \[ A \rightarrow bc \]
   \[ B \rightarrow Ac \]

6. Find GNF of the grammar given below
   \[ S \rightarrow ABAb \mid ab \]
   \[ B \rightarrow ABA \mid a \]
   \[ A \rightarrow a \mid b \]

7. Find the GNF equivalent to the CFG
   \[ S \rightarrow AB \]
   \[ A \rightarrow aA \mid bB \mid b \]
   \[ B \rightarrow b \]

8. Find a GNF grammar equivalent to the following CFG
   \[ S \rightarrow BA \mid ab \]
   \[ B \rightarrow AB \mid a \]
   \[ A \rightarrow Bb \mid BB \]

9. Convert the given grammar to GNF
   \[ S \rightarrow SS \mid aSb \mid ab \]

10. Convert the following grammar into GNF
    \[ S \rightarrow XY1 \mid 0 \]
    \[ X \rightarrow 00X \mid Y \]
11. Convert the following CFG to GNF (Dec-2005)

\[ S \rightarrow aSa \mid bSb \mid c \]

**Regular Grammar**

**Definition:** The language accepted by finite automata can be described using a set of productions known as regular grammar. The productions of a regular grammar are of the following form:

- \( A \rightarrow a \)
- \( A \rightarrow aB \)
- \( A \rightarrow Ba \)
- \( A \rightarrow \epsilon \)

Where, \( a \in T \) and \( A, B \in V \).

A language generated by a regular grammar is known as regular language. A regular grammar could be written in two forms:

1. Right-linear form
2. Left-linear form

**Right-Linear Form:** A right linear regular grammar will have production of the given form.

- \( A \rightarrow a \)
- \( A \rightarrow aB \)
A → \epsilon

Note: Variable B in A → aB is the second symbol on the right.

**Left-Linear Form:** A left linear regular grammar will have productions of the following form:

\[ A → a \]
\[ A → Ba \]
\[ A → \epsilon \]

Note: Variable B in A → Ba is the first symbol on the left.

**DFA to Right Linear Regular Grammar:**

Every DFA can be described using a set of production using the following steps:

1. Let the DFA, \( M = (Q, \Sigma, \delta, q_0, F) \)
   
   Let the corresponding right linear grammar be \( G = (V, T, P, S) \).

2. Rename \( q_0 \in Q \) as \( S \in V \), relating start state of \( M \) with starting symbol of \( G \).

3. Rename states of \( Q \) as \( A, B, C, D \ldots \) where, \( A, B, C, D \ldots \in V \).

4. Creating a set of production \( P \).
   
   a. If \( q_0 \in F \) then add a production
      
      \[ S → \epsilon \]
      
      to \( P \).
   
   b. For every transition of the form,
      
      \[ q \xrightarrow{a} C \]

      Add a production \( B → aC \), where \( C \) is a non-accepting state.

   c. For every transition of the form,
      
      \[ q \xrightarrow{a} \]

      Add a production \( B → aC \), where \( C \) is a non-accepting state.
Add two productions $B \to aC$, $B \to a$, where $C$ is an accepting state.

**DFA to Left Linear Regular Grammar:**

Following steps are required to write a left linear grammar corresponding to a DFA.

1. Interchange starting state and the final state.
2. Reverse the direction of all the transitions.
3. Write the grammar from the transition graph in left-linear form.

**Right Linear Grammar to DFA:**

Every right linear grammar can be represented using a DFA:

1. A production of the form $A \to aB$ will generate a transition

   ![Diagram of A → aB]

   for the DFA.

2. A production of the form $A \to aB \mid a$ will generate a transition

   ![Diagram of A → aB | a]

   provided every transition entering $B$ terminates in $B$.

3. A Production of the form $A \to \varepsilon$ will make $A$ a final state.

   ![Diagram of A]
4. An independent production of the form $A \rightarrow b$, will generate a transition

![Transition Diagram]

Where, $F$ is a new state and it should be a final state.

**Left Linear Grammar to DFA:**

Every left linear grammar can be represented using an equivalent DFA. Following steps are required to draw a DFA for a given left linear grammar.

1. Draw a transition graph from the given left linear grammar.
2. Reverse the direction of all the transitions.
3. Interchange starting state and the final state.
4. Carry out conversion from FA to DFA.

**Right Linear Grammar to Left Linear Grammar:**

![Conversion Diagram]

Fig: From Right Linear Grammar to Left Linear Grammar.

Every right linear grammar can be represented by an equivalent left linear grammar. The conversion process involves drawing of an intermediate transition graph. Following steps are required:
1. Represent the right grammar using a transition graph. Mark the final state as \( \varepsilon \).
2. Interchange the start and the final state.
3. Reverse the direction of all transitions.
4. Write left – linear grammar from the transition graph.

**Left Linear Grammar to Right Linear Grammar:**

![Transition Graph Diagram]

**Fig: From Left Linear Grammar to Right Linear Grammar.**

Every left linear grammar can be represented by an equivalent right linear grammar. The conversion process involves drawing of an intermediate transition graph. Following steps are required:

1. Represent the left grammar using a transition graph. Mark the final state as \( \varepsilon \).
2. Interchange the start and the final state.
3. Reverse the direction of all transitions.
4. Write right – linear grammar from the transition graph.

**Problems:**

**University questions will be solved in Class:**

1. Construct right linear grammar and left linear grammar for the language \((ba^*)\).

2. Convert the following right-linear grammar to an equivalent DFA:
   
   \[
   S \rightarrow bB \\
   B \rightarrow bC
   \]
3. Convert following RG to DFA:

\[
\begin{align*}
S & \rightarrow 0A \mid 1B \\
A & \rightarrow 0C \mid 1A \mid 0 \\
B & \rightarrow 1B \mid 1A \mid 1 \\
C & \rightarrow 0 \mid 0A
\end{align*}
\]

4. Final the equivalent DFA accepting the regular language defined by the right linear grammar given as:

\[
\begin{align*}
S & \rightarrow aA \mid bB \\
A & \rightarrow aA \mid bc \mid a \\
B & \rightarrow aB \mid b \\
C & \rightarrow bB
\end{align*}
\]

5. Construct DFA accepting the regular language generated by the left linear grammar given below:

\[
\begin{align*}
S & \rightarrow Ca \mid Bb \\
C & \rightarrow Bb \\
B & \rightarrow Ba \mid b
\end{align*}
\]

6. Construct DFA accepting the language generated by the left linear grammar given below:

\[
\begin{align*}
S & \rightarrow B1 \mid A0 \mid C0 \\
B & \rightarrow B1 \mid 1 \\
A & \rightarrow A1 \mid B1 \mid C0 \mid 0 \\
C & \rightarrow A0
\end{align*}
\]

7. Convert the following right linear grammar to an equivalent left-linear grammar:

\[
\begin{align*}
S & \rightarrow bB \mid b \\
B & \rightarrow bC \\
B & \rightarrow aB \\
C & \rightarrow a \\
B & \rightarrow b
\end{align*}
\]
8. Write an equivalent left linear grammar from the given right linear grammar:
   \[ S \rightarrow 0A \mid 1B \]
   \[ A \rightarrow 0C \mid 1A \mid 0 \]
   \[ B \rightarrow 1B \mid 1A \mid 1 \]
   \[ C \rightarrow 0 \mid 0A \]

9. For right linear grammar given below, obtain an equivalent left linear grammar:
   \[ S \rightarrow 10A \mid 01 \]
   \[ A \rightarrow 00A \mid 1 \]

10. Write an equivalent right linear grammar from the given left linear grammar:
    \[ S \rightarrow C0 \mid A0 \mid B1 \]
    \[ A \rightarrow A1 \mid C0 \mid B1 \mid 0 \]
    \[ B \rightarrow B1 \mid 1 \]
    \[ C \rightarrow A0 \]

11. Construct the right linear grammar corresponding to the regular expression:
    \[ R = (0+1)1^*(1+(01)^*) \]

12. Write an equivalent right recursive grammar for the given left recursive grammar:
    \[ S \rightarrow S10 \mid 0 \]

13. Construct the right linear grammar corresponding to the regular expression:
    \[ R = (1+(01)^*)1^*(0+1) \]

14. Draw NFA accepting the language generated by grammar with productions:
    \[ S \rightarrow abA \mid bB \mid aba \]
    \[ A \rightarrow b \mid aB \mid bA \]
    \[ B \rightarrow aB \mid aA \]

15. Construct right linear and left linear grammar for the language:
    \[ L = \{a^n b^m, n \geq 2, m \geq 3\} \]

16. Construct left linear and right linear grammar for the language:
    \[ 0^*(1(0+1))^* \]
17. Construct left linear and right linear grammar for the language:
   \((0+1)^*00(0+1)^*\)

18. Construct left linear and right linear grammar for the language:
   \(((01+10)^*11)^*00)^*\)

19. Describe the language generated by the following grammar:
   
   \[
   
   \begin{align*}
   S & \rightarrow bS \mid aA \mid \epsilon \\
   A & \rightarrow aA \mid bB \mid b \\
   B & \rightarrow bS
   \end{align*}
   
   \]

20. Find the CFL associated with CFG:
   
   \[
   
   \begin{align*}
   S & \rightarrow 0Q \mid 1P \\
   P & \rightarrow 0 \mid 0S \mid 1PP \\
   Q & \rightarrow 1 \mid 1S \mid 0QQ
   \end{align*}
   
   \]
# Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 6

Pushdown Automata (PDA)

Introduction to Pushdown Automata (PDA): (Dec-2005)

- The context-free languages have a type of automaton that defines them. This automaton, called a "pushdown automaton," is an extension of the nondeterministic finite automaton with ε-transitions, which is one of the ways to define the regular languages.

- The pushdown automaton is essentially an ε-NFA with the addition of a stack. The stack can be read, pushed, and popped only at the top, just like the "stack" data structure.

- In this chapter, we define two different versions of the pushdown automaton:
  1. one that accepts by entering an accepting state, like finite automata do, and
  2. another version that accepts by emptying its stack, regardless of the state it is in.

- Informally, pushdown automata can be viewed as finite automata with stack. An added stack provides memory and increases the capability of the machine.

- A pushdown automata can do the followings:
  1. Read input symbol [as in case of FA].
  2. Perform stack operations:
     - Push operation
     - Pop operation
     - Check empty condition of a stack through an initial stack symbol.
     - Read top symbol of stack without a pop.
  3. Make state changes.
- PDA is more powerful than FA. A context-free language (CFL) can be recognized by a PDA. Only a subset of CFL that are regular can be recognized by finite automata.

1. A context free language can be recognized by PDA.
2. For every context-free language, there exists a PDA.
3. The language of PDA is a context-free language.

**Example:** A string of the form $a^nb^n$ cannot be handled by a finite automaton. But the same can be handled by a PDA.

1. Any machine recognizing a string of the form $a^nb^n$, must keep track of a’s as number of b’s must be equal to the number of a’s.
2. First half of the string can be remembered through a stack.

3. As the machine reads the first half of $a^nb^n$, it remembers it by pushing it on top of the stack. As shown in Fig. after reading first 5 a’s, the stack contains 5 b’s.

4. While reading the second half of the input string consisting of b’s, the machine pops out an ‘a’ from the stack for every ‘b’ as input.

![Fig: Structure of PDA](image-url)
5. After reading 5 b's, input will finish and the stack will become empty. This will indicate that the input string is of the form $a^nb^n$.

6. The machine will have two states $q_0$ and $q_1$:
   a. State $q_0$ – while the machine is reading a’s.
   b. State $q_1$ – while the machine is reading b’s

7. While in state $q_0$ an input ‘a’ is not allowed and hence there is a need for two states.

8. A transition in PDA depends on:
   a. Current state
   b. Current input
   c. Top symbol of the stack

9. A transition in PDA can be shown as a directed edge from the state $q_i$ to $q_j$. While moving to state $q_j$, the machine can also perform stack operation. A transition edge from $q_i$ to $q_j$ should be marked with current input, current stack symbol and the stack operation. It is shown in below fig:

10. A PDA uses three stack operations:
    a. Pop operation, it removes the top symbol from the stack.
    b. Push operation, it inserts a symbol onto the top of the stack.
    c. Nop operation, it does nothing to stack.

11. The language \{ $a^nb^n$ | n $\geq$ 1 \} can be accepted by the PDA of Fig:
12. The state $q_0$ will keep track of the number of a’s in an input string, by pushing symbol ‘a’ onto the stack for each input ‘a’. A second state $q_1$ is used to pop an ‘a’ from the stack for each input symbol ‘b’. Finally, after consuming the entire input the will become empty.

The Formal Definition of PDA:

Push Down Automata:

- PDA consists of read-only input tape. In addition it has a stack called Push down Store (PDS). It is a read-write push down store as we add elements from PDS.

- PDA consists of finite set of states, one initial and one or more final states.

- After the input symbol is read the machine can remain in the same state or change the state, at the same time it can push a symbol on the stack or pop the top most symbol or perform neither push nor pop.
A PDA is a 7-tuple
\[ P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \]
where,

\[ \Sigma: \text{A finite set of input symbols, also analogous to the corresponding component of a finite automaton.} \]

\[ \Gamma: \text{A finite stack alphabet. This component, which has no finite-automaton analog, is the set of symbols that we are allowed to push onto the stack.} \]

\[ \delta: \text{The transition function. As for a finite automaton, } \delta \text{ governs the behavior of the automaton. Formally, } \delta \text{ takes as argument a triple } \delta (q, a, X), \text{ where:} \]

1. \( q \) is a state in \( Q \).
2. \( a \) is either an input symbol in \( \Sigma \) or \( a = \varepsilon \), the empty string, which is assumed not to be an input symbol.
3. \( X \) is a stack symbol, that is, a member of \( \Gamma \).

\[ q_0: \text{The start state. The PDA is in this state before making any transitions.} \]

\[ Z_0: \text{The start symbol. Initially, the PDA's stack consists of one instance of this symbol, and nothing else.} \]

\[ F: \text{The set of accepting states, or final states.} \]

A Graphical Notation for PDA's

- Sometime, a diagram, generalizing the transition diagram of a finite automaton, will make aspects of the behavior of a given PDA clearer.
We shall therefore introduce and subsequently use a transition diagram for PDA's in which:

1. The nodes correspond to the states of the PDA.

2. An arrow labeled Start indicates the start state, and doubly circled states are accepting, as for finite automata.

3. The arcs correspond to transitions of the PDA in the following sense. An arc labeled \( a, X/\alpha \) from state \( q \) to state \( p \) means that \( \delta(q, a, X) \) contains the pair \( (p, \alpha) \), perhaps among other pairs. That is, the arc label tells what input is used, and also gives the old and new tops of the stack.

![Transition Diagram](image)

**Figure:** Representing a PDA as a generalized transition diagram

- The only thing that the diagram does not tell us is which stack symbol is the start symbol. Conventionally, it is \( Z_0 \), unless we indicate otherwise.
Instantaneous Description of a PDA:

Let \( P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \) be a PDA.

An ID is \((q, x, \alpha)\)
\[
q \in Q, x \in \Sigma^*, \alpha \in \Gamma^* 
\]
say \((q, a_1a_2...a_n, z_1z_2...z_m)\) is an ID

This describes the PDA when the current state is \(q\), the input string to be processed is \(a_1a_2...a_n\) and the PDS has \(z_1z_2...z_m\) with \(z_1\) at the top and \(z_m\) lowest.

Definition:

Let \( P \) be a PDA

A move relation (denoted by \( \rightarrow \)) between ID's is defined as
\[
(q, a_1a_2...a_n, z_1z_2...z_m) \rightarrow (q', a_2, ... a_n \beta z_2...z_m)
\]

If \( \delta(q, a_1, z_1) \) contains \((q', \beta)\)

i.e. PDA in state of with \(z_1z_2...z_m\) in PDS \((z_1\) is at the top) reads the input symbol \(a_1\), the PDA moves to state \(q'\) and write \(\beta\) on the top of \(z_2, ...z_m\). After this transition, the input string to be processed \(a_2a_3,...a_n\).

Example:

\[
P = \{(q_0, q_1, q_2), \{a, b, c\}, \{a, b, z_0\}, \delta, q_0, z_0, \{q_2\}\}
\]

\[
\delta:
\delta(q_0, a, z_0) = \{(q_0, az_0)\}
\delta(q_0, a, q) = \{(q_0, aa)\}
\delta(q_0, a, b) = \{(q_0, ab)\}
\delta(q_0, b, z_0) = \{(q_0, bz_0)\}
\delta(q_0, b, a) = \{(q_0, ba)\}
\delta(q_0, b, b) = \{(q_0, bb)\}
\delta(q_0, c, a) = \{(q_1, a)\}
\delta(q_0, c, b) = \{(q_1, b)\}
\delta(q_0, c, z_0) = \{(q_1, z_0)\}
\delta(q_1, a, a) = \{(q_1, \epsilon)\}
\delta(q_1, b, b) = \{(q_1, \epsilon)\}
\delta(q_1, \epsilon, z_0) = \{(q_2, z_0)\}
\]
Input sequence “bacab”
(q₀, bacab, z₀)
(q₀, acab, bz₀)
(q₀, cab, ab z₀)
(q₁, ab, abz₀)
(q₁, b, bz₀)
(q₁, Є, z₀)
(q₁, Є, z₀)
Thus (q₀, bacab, z₀) |-* (q₂, z₀)

Acceptance by PDA:

PDA by Final State Method:
Definition:
Let P = (Q, Σ, Γ, δ, q₀, Z₀, F) be PDA.
The language accepted by final state is defined as

\[ W ∈ Σ^* | - (q₀,w,z₀) | - (q_f, Є, α) \]
\[ q_f ∈ F \quad α ∈ Γ^* \]

PDA by Null Store Method:
Definition:
Let P = (Q, Σ, Γ, δ, q₀, Z₀, F) be a PDA.
The set accepted by null store (or empty store) is defined as

\[ W ∈ Σ^* | - (q₀,w,z₀) | - (q_f, Є, Є) \] for some q ∈ Q.

Problems:
University questions will be solved in Class:

1. Design a PDA to accept strings of type $a^n b^n$.  
   (May-2004)

2. Design a PDA to accept strings of type $0^n 1^n$.  
   (Dec-2006)

3. Design a PDA to accept strings of type $a^n b^{2n}$.  

4. Design a PDA to accept strings of type $a^{2n} b^n$.  

5. Design a PDA to accept strings of type $0^n 1^{2n+1}$.  
   (Dec-2002)

6. Design a PDA to accept $(bdb)^n$.  
   (Dec-2005)

7. Design a PDA to accept $(bdb)^a c^n$.  
   (Dec-2005)

8. Design a PDA which accepts the strings containing equal no. of a’s and b’s.

9. Design a PDA to accept strings of type $(ab)^n c^n$.  

10. Design a PDA to accept $(ab)^n (cd)^n$.  
    (Jun-2007)

11. Construct a PDA that accepts the language – \{a^n b^m a^{n+m} | m, n \geq 1\}

12. Construct a PDA $M$ accepting \{a^n b^m a^n | m, n \geq 1\} by null store.  
    (Dec-2002, Dec-2006)

13. Design a PDA to check for well-formed ness of parenthesis.  
    (Dec-2006)

14. Design DPDA accepting balanced string of brackets.  
    (Nov-2004)

15. Design DPDA to accept strings with more a’s than b’s.

16. Construct a PDA that will recognize the language

   \[L=\{WCWR | W \in \{a, b\}^* WR = \text{reverse of W}\}\]
17. Construct a PDA that will recognize the language

\[ L = \{ WW^R \mid W \in \{a, b\}^* \text{ where } W^R \text{ is the reverse of } W \} \text{ (even palindrome)} \]


18. Design the PDA to accept the language containing all odd length palindromes over \( \Sigma = \{0, 1\} \)

(Dec-2007)

Pushdown Automata and Context – free Languages:

- There is a general relation between Context – free Languages and NPDA. In the following section we see that for every context – free language there is a NPDA that accepts it and conversely, that the language accepted by any NPDA is context – free.

**Theorem:**

For any context – free language \( L \), there exists an NPDA \( M \) such that

\[ L = L(M) \]

**Proof:**

If \( L \) is \( \mathcal{L} \) – free context free language, there exists a CFG in Greibach Normal Form for it. Let \( G = (V, T, S, P) \) be such a grammar. We then construct an NPDA which simulates leftmost derivations in this grammar. As suggested, the simulation will be done so that the unprocessed part of the sentential form is in the stack, while the terminal prefix of any sentential form matches the corresponding prefix of the input string. Specifically, the NPDA will be

\[ M = (\{q_0, q_1, q_f\}, T, V \cup \{z\}, \delta, q_0, z, \{q_f\}) \]

where \( z \in V \). Note that the input alphabet of \( M \) is identical with the set of terminals of \( G \) and that the stack alphabet contains the set of variables of the grammar.

The transition function will include

\[ \delta(q_0, \epsilon, z) = \{(q_1, Sz)\} \]
so that after the first move of M, the stack contains the start symbol S of the derivation.
(the stack start symbol z is a marker to allow us to detect the end of the derivation.). In
addition, the set of transition rules is such that

\[(q_1, u) \in (q_1, a, A)\]
Whenever
\[A \rightarrow au\]
is in \(p\). This reads input a and remaining variable A from the stack replacing it with u. In
this way it generates the transitions that allow the PDA to simulate all derivations.
Finally, we have
\[\delta(q_1, \epsilon, z) = \{(q, z)\}\]
to get M into a final state.
To show that M accepts any \(w \in L(G)\), consider the partial leftmost derivation
\[S \Rightarrow a_1a_2...a_nA_1A_2...A_m\]
\[\Rightarrow a_1a_2...a_nbB_1B_2...B_kA_2...A_m.\]

If M is to simulate this derivation, then after reading \(a_1a_2...a_n\), the stack must contain
\(A_1A_2...A_m\). To take the next step in the derivation, \(G\) must have a production
\[A_1 \rightarrow bB_1B_2...B_k\]

But the construction is such that then M has a transition rule in which
\[\delta(q_1, B_1...B_k) \in \delta(q_1, b, A_1),\]
so that the stack now contains \(B_1...B_kA_2...A_m\) after having read \(a_1a_2...a_n\).
A simple induction argument on the number of steps in the derivation shows that if
\[S \Rightarrow w,\]
Then
\[(q_1, w, Sz) \vdash (q_1, \epsilon, z),\]

Now we have,
\[(q_0, w, z) \vdash (q_1, w, Sz) \vdash (q_1, \epsilon, z) \vdash (q_0, \epsilon, z),\]
So that \(L(G) \subseteq L(M)\).

To prove that \(L(M) \subseteq L(G)\), let \(w \in L(M)\). then by definition.
\[(q_0, w, z) \vdash (q_0, \epsilon, u).\]
But there is only one way to get from $q_0$ to $q_1$ and only one way from $q_1$ to $q_f$. Therefore must have

$$(q_1, w, Sz) \vdash (q_1, \epsilon, z),$$

Now let us write $w = a_1a_2a_3...a_n$. Then the first step in

$$(q_1, a_1a_2a_3...a_n, Sz) \vdash (q_1, a_2a_3...a_n, u_1z).$$

But then the grammar has a rule of the form $S \rightarrow a_1u_1$, so that

$$S \Rightarrow a_1u_1.$$  
Repeating this, writing $u_1 = A_1u_2$, then

$$(q_1, a_2a_3...a_n, A_1u_2z) \vdash (q_1, a_3...a_n, u_1u_2z).$$

Implying that $A \rightarrow a_2u_3$ is in the grammar and that

$$S \Rightarrow a_1a_2a_3...a_n.$$  
In the consequence, $L(M)$ subset of $L(G)$, completing the proof if the language does not contain $\epsilon$.

Problems:

**University questions will be solved in Class:**

1. Construct a PDA equivalent to the following grammar: (Nov-2004, May-2005)
   
   $$S \rightarrow aAA$$
   
   $$A \rightarrow aS \mid bS \mid a$$

2. Construct the PDA equivalent to the following CFG: (May-2006, Dec-2007)
   
   $$S \rightarrow 0BB$$
B → 0S | 1S | 0
Test whether 010000 is in the language.

3. Construct a PDA equivalent to the following grammar
   S → aA
   A → aABC | bB | a
   B → b
   C → c

4. Construct a PDA equivalent to the following grammar:
   S → aSa | bSb | c

5. Construct a PDA equivalent the following grammar:
   E → E + E | E * E | (E) | id

6. Design PDA for the following CFG:
   S → (S) | SS | ^

7. Write CFG for language having number of a's greater than number of b's and Design
   a PDA for the same.
   (Dec-2009)
Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 7

Turing machine

Turing Machine:

- Turing Machine is a simple mathematical model of a general purpose computer.

- Turing machine models the computing power of a computer i.e. the Turing machine is capable of performing any calculation which can be performed by any computing machine.

- The Turing Machine can be thought of as a finite automata connected to read/write head.

- It has one tape which is divided into number of cells. Each cell can store one symbol.

- The input to and the output from the Finite Automata are affected by the read/write head which can examine one cell at a time.

- In one move, the machine examines the present symbol under the read/write head on the tape and the present state of an automation to determine:
  - A new symbol to be written on the tape in the cell under the read/write head.
  - A motion of the read/write head along the tape i.e. either the head moves one cell left (L), one cell right (R) or stay at the same cell (S)
  - The next state of machine.

Turing Machine is a 7-tuple

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \]

Where,

- \( Q \) - finite nonempty set of states
Σ- set of input symbols

Ѓ- finite nonempty set of tape symbols

δ- transition function mapping the state of finite automaton and tape symbols to states, tape symbols and movement of head

δ : Q × ″ → Q × ″ × { L, R, S}

q₀- initial state of

B- special tape symbol representing blank

F- set of final states F

**Instantaneous Descriptions:**

- Snapshots of a Turing machine in action can be used to describe a Turing Machine. These give instantaneous descriptions of a Turing Machine.

- An ID of Turing Machine is defined in terms entire input string and the current state.

**Definition:**

An ID of Turing Machine M is a string γβα

Where,

- β is the present state of M.

- The input string is split as γα.

- The first symbol of γ is the current symbol a under read/write head and γ has all the subsequent symbols of the input string.

- The substring α of the input string formed by all the symbols to the left of a.
**Moves in a Turing Machine**

Say \( \delta(q, x_i) = (p, y, L) \)

The input string to be processed is \( x_1 x_2 \ldots x_n \) and the present symbol under read/write head is \( x_i \).

So the ID before processing \( x_i \) is \( x_1 x_2 \ldots x_{i-1} q x_i \ldots x_n \)

After processing of \( x_i \), the resulting ID is

\[ x_1 \ldots x_{i-1} p x_i y x_i+1 \ldots x_n \]

This is represented by,

\[ x_1 x_2 \ldots x_{i-1} q x_i \ldots x_n \mid x_i \ldots x_{i-2} p x_i y x_i+1 \ldots x_n \]

Say \( \delta(q, x_i) = (p, y, R) \)

\[ x_1 x_2 \ldots x_{i-1} q x_i \ldots x_n \mid \neg x_1 x_2 \ldots x_{i-1} x_i y x_i+1 \ldots x_n \]

Note: The description of moves by IDs is very much useful to represent the processing of input strings.

**Language Acceptability by Turing Machine:**

Let us consider the Turing Machine, \( M = (Q, \Sigma, \Gamma, \delta, q_0, \beta, F) \)

A string is said to be accepted by \( M \)

If \( q_0 w \mid - \alpha_1 p \alpha_2 \)

Where

\( P \) is the final state and
\( \alpha_1 \) and \( \alpha_2 \) are tape symbols.

The TM \( M \) does not accept \( w \)

If the machine \( M \) either halts in a non-accepting state or does not halts.
Problems:

University questions will be solved in Class:

Turing Machine as language recognizer:

1. Design a Turing machine which recognizes the language
   \[ \{a^n b^n \mid n \geq 1\} \]

2. Design a Turing machine which recognizes the language
   \[ \{0^n 1^n \mid n \geq 1\} \]

3. Design a Turing machine which recognizes the language
   \[ \{0^n 1^n 2^n \mid n \geq 1\} \]

4. Design a Turing machine which recognizes the language
   \[ \{0^{2n} 1^n \mid n \geq 1\} \] (Dec-2002)

5. Design a Turing machine which recognizes the language
   \[ \{0^n 1^n 0^n \mid n \geq 1\} \] (Dec-2002)
6. Design a Turing machine which recognizes the language \( \{a^n b^n a^n \mid n \geq 1\} \)

7. Design a Turing machine which recognizes the language having equal number of a’s and b’s

   OR

   Design TM to accept the language –
   
   \[ L = \{ x \in \{0, 1\}^* \mid x \text{ contains equal number of 0's and 1's} \} \]

   Simulate the operation for the string 110100.

   (Dec-2006, Jun-2008)

8. Design a TM to accept a language over \{a, b\} such that the number of a’s > number of b’s.

   (May-2007)

9. Construct Turing Machine that will accept the language \( L \) over \( \Sigma = \{a, b\} \) where \( L = \{ w : \mid w \mid \text{is even} \} \)

10. Construct Turing Machine that will accept the language \( L \) over \( \Sigma = \{a, b\} \) where \( L = \{ w : \mid w \mid \text{is a multiple of 3} \} \)

11. Design a Turing machine to check for well-formed ness of parenthesis

   OR

   Design a Turing Machine that checks whether a string of left and right parenthesis is well formed or not.

   (May-2006)
12. Design TM to recognized string containing even number of a’s and odd number of b’s over $\Sigma = \{a, b\}$

13. Design TM to recognize palindromes over $\Sigma = \{a, b\}$

14. Design TM that can accept set of all even palindromes over alphabet (0, 1).

15. Design a Turing machine which recognizes the language
   \[ L = \{WCWR \mid W \in \{a, b\}* \ W^R \text{ – reverse of } W \}. \]

16. Design a Turing machine which recognizes the language
   \[ L = \{WWR \mid W \in \{0, 1\}* \ W^R \text{ – reverse of } W \}. \]

University questions will be solved in Class:

Turing Machine to compute functions:

1. Design a T.M. to perform addition of two numbers.
   (Dec-2003)

2. Design a M. to perform multiplication of two numbers.
3. Design a T.M. to perform subtraction \( m - n \)
   Which is defined as
   
   \[
   \begin{align*}
   m - n & \quad m > n \\
   m \leq n & \quad 0
   \end{align*}
   \]

4. Construct Turing Machine to subtract two numbers assume \( m > n \)
   (May-2006, Dec-2009)

5. Design T.M. to perform division and find the quotient and remainder.

6. Design T.M. to find \( n^2 \)  
   (Nov-2004)

7. Design T.M. to find \( n! \)  

8. Design T.M. to find \( \log_2 4 \)  
   (Dec-2003)

9. Design T.M. to find \( \log_2 n \)  

10. Design TM to perform \( C = A - B \) given \( \#A\#B\# \).  
    (May-2005)

11. Design T.M. to find 2’c of given binary number.
12. Design TM to detect whether a unary number is divisible by 3. 
   (May-2005)

13. Design T.M. to increment binary number by 1.

14. Design TM to covert Unary number to Binary Number

15. Design TM to convert Binary Number to Unary Number

16. Design TM to perform $C = A + B$
   
   \[
   \text{i/p} \quad \text{"A + B"} \\
   \text{o/p} \quad \text{"C"}
   \]

17. Design TM to decrement binary number by 1.

18. Design TM to add Binary numbers

19. Construct a Turing Machine that compares 2 numbers $m$ and $n$ and leaves at the end $x$ on the tape where $x = g/l/e$ depending on whether $m>n$ / $m<n$ / $m=n$ respectively.

20. Design TM to perform $C = A \times B$
   
   \[
   \text{i / p} \quad \text{"A \times B"} \\
   \text{0 / p} \quad \text{"C"}
   \]

21. Design TM to recognize palindromes over the input $\Sigma = \{a, b\}^*$
22. Design a TM for generating $2^n$ where $n$ is binary. The result should also be binary.


Universal Turing Machine:

- A universal Turing Machine (UTM) is a Turing Machine ™ which is al powerful or universal in the sense that it is capable of doing anything that any other TM can do.

- In other words, the UTM should have the capability of imitating any Turing machine 'T' given the following information in its tape:
  
  - The description of 'T' in terms its operation or program area of the tape (i.e. the transaction table).
  
  - The initial configuration of the TM i.e. starting state or the current state and the symbol scanned. The processing data to be fed to 'T' (data area of the tape).
  
  - This obviously means that the UTM should have an algorithm to interpret correctly the rules of operation given about the TM 'T'.

- The behavior of the UTM is simple, namely, simulating 'T' one step at a time as follows:
  
  - A marker to indicate the point at which the description of 'T' begins, and it keeps a complete account of how the tape of 'T' looks likes at every instant guides it.
  
  - Also, it remembers the state 'T' is in, and the symbol 'T' is reading. Then it simply looks at the description of 'T'; to carry out what 'T' is supposed to do.
  
  - In order to exhibit this behavior, the UTM should have a lookup facility and should perform the following steps:

  **Step 1:** Scan the square on the state area of the tape and read the symbol that 'T' reads and initial state of 'T'.

  **Step 2:** Find the triplet which corresponds to the initial state and the input symbol read in step 1.
(Triplet: new state, new symbol to be replaced and direction of move).

**Step 3:** Move the tape to reach the appropriate square in the data area, replace the symbol, move the tape in the required direction, read the next symbol and finally reach state area and replace the state and scanned symbols. Goto step 1.

The UTM laid the foundation for:

- Stored program computers and
- Interpretative implementation of programming languages.

**Variations of Turing Machine:**

1. **Turing Machine with Two Way Infinite Tape:**

A Turing Machine with a two infinite tape is denoted by

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \]

as in the original model.

- As it name implies, the tape is infinite to the left as well as to the right.
- We imagine that there is infinity of blank cells to the left and the right of the current nonblank portion of the tape.
- \( L \) is recognized by a Turing Machine with a two-way infinite tape if and only if it is recognized by a Turing Machine with a one-way infinite tape.

2. **Multitape Turing Machine:**

- A multitape Turing machine consists of a finite control with \( k \) tape heads and \( k \) tapes; each tape is infinite in both directions.
- On a single move, depending on the state of the finite control and the symbol by each of the tape heads, the machine can:
  - Change State
- Print a new symbol on each of the cells scanned by its tape heads.

- Move each of its tape head, independently, one cell to the left right, or keep it stationary.

3. Nondeterministic Turing Machine:

- A nondeterministic Turing Machine is a device with a finite control and a single one-way infinite tape.

- For a given state and tape symbol scanned by the tape head, the machine has a finite number of choices for the next move. Each choice consists of a
  - New state
  - Tape Symbol to print
  - Direction of head motion

4. Multidimensional Turing Machine:

- It is a device having a finite control and the tape consists of a k-dimensional array of cells infinite in all 2k directions for some fixed k. Depending on the state and the symbol scanned, the device

  - Changes state
  - Prints a new symbol
  - Move the tape head in one of 2k directions either positively or negatively along with one of the k axes.

5. Multihead Turing Machine:

- A k-head Turing Machine has fixed number, k, of heads. The heads are numbered 1 through k, and a move of the TM depends on the state and on the symbol scanned by each head.

- In one move, the heads may each move independently left, right or remain stationary.
6. Composite T.M.:

- Two or more Turing Machine can be combined to solve a collection of simpler problem, so that the output of one TM forms the input to the next TM and so on. This is called as Composition.

- The idea of composite TM give rise to the concept of breaking the complicated job into number of jobs implementing each separately and then combining them together to get answer for the job required to be done.

7. Iterated T.M.:

In Iterated TM the output it applied to the input repetitively.

8. TM with Semi-infinite tapes:

- Till now we have allowed the tape head of TM to move either left or right from its initial position.

- It is only necessary that the TM’s head be allowed to move within the positions at and to the right of the initial head position.

- In TM with Semi-infinite tapes there are no cells to the left of the initial position.

The Halting Problem:

For a give configuration of a TM case can arise:

- The machine starting at this configuration will halt after a finite number of steps.
- The machine starting at this configuration never no matter how long it runs.
Given any TM, problem of determining whether it halts ever or not, is called as halting problem.

To solve the halting problem, we should have some mechanism to which given any functional matrix, input data type and initial configuration of the TM for which we want to detect, determines whether the process will ever halt or not.

Note:  In reality, one cannot solve the halting problem. The halting problem is unsolvable. That means there exists no TM, which can determine whether a given program including itself, will ever halt, or not.

Proof:-

1. Let us prove the halting problem by contradiction. Suppose that there exists a TM ‘A’ which decides whether or not any computation by a TM ‘T’ will ever halt, given the description ‘dT’ of ‘T’ and the tape ‘t’ of ‘T’. Then for every input (t, dT) to ‘A’, if ‘T’ halts for the input ‘t’, ‘A’ reaches an “accept halt”;

2. If ‘T’ does not halt for the input ‘t’, then ‘A’ reaches an “reject halt”. We can now construct another TM ‘B’ which takes ‘dT’ as the input and proceed as follows:
   - First it copies the input ‘dT’ and duplicates ‘dT’ on its input tape and then takes this duplicated information tape as the input to ‘A’ with one modification namely, whenever ‘A’ is supposed to reach an “accept halt”, ‘B’ will loop forever.

3. Considering the original behavior of ‘A’, we find that ‘B’ acts as follows. It loops if ‘T’ halts for input t = dT and halts if ‘T’ does not halt for the input t = dT.

4. Since ‘B’ itself is a TM, let us set T=B. Thus replacing ‘T’ by ‘B’ we get that, ‘B’ halts for the input ‘dB’ if and only if ‘B’ halts for the input ‘dB’. This is a contradiction.

5. Hence we conclude that machine ‘A’ which can decide whether any other TM will ever halt, does not exist. Therefore, halting problem is unsolvable.
Consequences of Halting Problem:

1. We cannot decide whether a TM ever prints a given symbol of its alphabet. This is also unsolvable.

2. Two TM’s with the same alphabet cannot be checked for equivalence or inequivalence by an algorithm; i.e. there is no effective general way to decide whether a given computational process will ever terminate or whether two given processes are equivalent. This is also another unsolvable problem.

3. Blank-tape theorem: There exists a TM which when started on a Blank tape, can write its own description. This is of interest in constructing self-reproducing machine.

Turing Machine and Computers:

Simulating a TM by Computer:

1. Given a particular TM M, we must write a program that acts like M.

2. Finite Control: since there are only a finite number of states as character strings and use a table of transition, which it looks up to determine each move.

3. Machine Tape:
   - Zip disks or removable hard disks can simulate infinite machine tape.
   - We can arrange the disks placed in two stacks: One stack holds the data in cells of the left of the tape head, and other stack holds the data significantly to the right of the tape head.

Simulating a Computer by a TM:

The following diagram shows how the TM would be designed to simulate a computer.

1. The first tape represents memory of the computer.

2. The second tape holds the memory locations on tape1. The value stored in this location will be interpreted as the next computer instruction to be executed.
3. The third tape holds the memory address or the contents of that address after that address has been located on tape1. To execute an instruction, the TM must find the contents of memory address that hold data involved in the computation.

4. The fourth tape holds the simulated input to the computer, since the computer must read its input from a file.

5. The scratch tape would be used to compute mathematical operations efficiently.

Refer Class Notes for Examples.

Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>
Chapter 8

Intractable Problems

- We now bring our discussion of what can or cannot be computed down to the level of efficient versus inefficient computation.

- We focus on problems that are decidable, and ask which of them can be computed by Turing machines that run in an amount of time that is polynomial in the size of the input.

  1. The problems solvable in polynomial time on a typical computer are exactly the same as the problems solvable in polynomial time on a Turing machine.

  2. Experience has shown that the dividing line between problems that can be solved in polynomial time and those that require exponential time or more is quite fundamental.

  3. Practical problems requiring polynomial time are almost always solvable in an amount of time that we can tolerate, while those that require exponential time generally cannot be solved except for small instances.

- In this chapter we introduce the theory of "intractability," that is, techniques for showing problems not to be solvable in polynomial time.

- We start with a particular problem - the question of whether a boolean expression can be satisfied, that is, made true for some assignment of the truth values TRUE and FALSE to its variables.

- Since we are dealing with whether problems can be solved in polynomial time, our notion of a reduction must change.

- It is no longer sufficient that there be an algorithm to transform instances of one problem to instances of another.

- The algorithm itself must take at most polynomial time, or the reduction does not let us conclude that the target problem is intractable, even if the source problem is.
• There is another important distinction between the kinds of conclusions we drew in the theory of undecidability and those that intractability theory lets us draw.

• We assume the class of problems that can be solved by nondeterministic TM’s operating in polynomial time includes at least some problems that cannot be solved by deterministic TM’s operating in polynomial time (even if we allow a higher degree polynomial for the deterministic TM).

**The Classes P and NP:**

1. P denotes the class of problems, for each of which, there is at least one known polynomial time deterministic TM solving it.

2. NP denotes the class of all problems, for each of which, there is at least one known non-deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time deterministic TM.

3. Time complexity of an algorithm is defined as a function of the size of the problem.

4. For comparative study of algorithms, growth rate is considered to be very important.

5. Size of a problem is often measured in terms of the size of the input.

6. An algorithm with time complexity which can be expressed as a polynomial of the size of the problem is considered to have an efficient solution.

7. A problem which does not have any polynomial time algorithm is called an intractable problem, otherwise it is called tractable.

8. A solution by deterministic TM is called an algorithm. A solution by a non-deterministic TM may not be an algorithm.

9. For every non-deterministic TM solution, there is a deterministic TM solution of a problem. But there is no computation equivalence between deterministic TM and non-deterministic TM.
10. If a problem is solvable in polynomial time by non-deterministic TM then there is no guarantee that there exists a deterministic TM that can solve it in polynomial time.
   a. If P is set of tractable problem then P is subset of NP. It follows from the fact that every deterministic TM is a special case of non-deterministic TM.
   b. It is still not known whether P = NP.

**NP-Complete Problems:**

1. A problem is NP-complete if it is in NP and for which no polynomial time deterministic TM solution is known so far.

2. An interesting aspect of NP-complete problem is that for each of these problems:
   - It has not been possible to design a deterministic TM, so far.
   - It has not been possible to establish that a deterministic TM does not exist.

3. There is a large number of problems, for which it is not known whether it is in P or not in P. However, for each of these problems, it is known that it is in NP.

4. Each of these problems can be solved by at least on Non-deterministic TM, the time complexity of which is a polynomial function of the size of the problem.

5. A problem from the class NP, can be defined as one for which a potential solution, if given, can be verified in polynomial time whether the potential solution is actually a solution or not.

6. Some of the NP-complete problems include:
   - Satisfiability Problem (SAT)
   - Travelling Salesman Problem (TSP)
   - Hamiltonian circuit Problem (HCP)
   - The Vertex cover Problem (VCP)
   - K-Colourability Problem
   - The complete subgraph problem

**A Restricted Satisfiability Problem:**

1. The satisfiability problem states: Given a Boolean expression, is it satisfiable?
2. A Boolean expression said to be satisfiable if at least one truth assignment makes the Boolean expression ‘true’. Example, the Boolean expression \((X1 \land X2) \lor X3\) is true for \(X1 = 1, X2 = 1\) and \(X3 = 0\). Hence it is satisfiable.

3. A Boolean expression involves:
   a. Boolean variables \(X1, X2... Xn\), each of these can assume a value either TRUE or FALSE.
   b. Boolean operators:

4. The truth value of a Boolean expression depends on the truth values of its variables.

5. Satisfiability problem is NP-Complete.

6. SAT is NP-Complete and it is also known as Cook’s theorem.

7. A Boolean expression in conjunctive normal form is NP-complete. A Boolean expression is said to be in CNF, if it is expressed as \(C1 \land C2 \land C3 \land ... \land Ck\) where each \(Ci\) is a disjunction of the form \(X_{i1} \lor X_{i2} \lor ... \lor X_{im}\).

\[X_{ij}\] is a literal. A literal is either a variable \(Xi\) or negation \(Xi\).

**NP-Completeness:**

1. Polynomial-time reduction plays an important role in defining NP-completeness. A polynomial-time reduction is a polynomial-time algorithm which constructs instances of a problem \(P2\) from the instances of some other problem \(P1\).

2. If \(P1\) be a problem which is already known to be NP-complete. We want to check whether a problem \(P2\) is NP-complete or not. If we can design an algorithm which transforms or constructs an instance of \(P2\) for each instance of \(P2\), then \(P2\) is also NP-complete.

3. A method of establishing NP-completeness of a problem \(P2\) requires designing a polynomial time reduction algorithm that constructs an instance of \(P2\) for each instance of \(P1\), where \(P1\) is already known to be NP-complete.

4. **NP-Hard Problem:**
   a. A problem \(L\) is said to be NP-Hard if for any problem \(L1\) in NP, there is a polynomial-time reduction of \(L1\) to \(L\).
   b. In other words, a problem is NP-Hard if:
      i. Establishing \(L\) as an NP-class problem is so far not possible.
ii. For any problem L1 in NP, there is polynomial time reduction of L1 to L.
c. Every NP-Complete problem must be NP-Hard problem.

5. **Complements of Languages in NP:**
   a. It is not known whether NP is closed under complementation. The class of languages 
P is closed under complementation. It is believed that whenever language L is NP-complete, its complement is not in NP.
b. There is a set of languages whose complements are in NP. Such languages are called Co-NP.
c. We believe that complement of NP-complete problem is not in NP.
d. No NP-complete problem is in Co-NP.
e. We believe that the complements of NP-complete problems, which are in Co-NP are 
not in NP.

**Language Classes Based on Randomization:**

1. A randomized algorithm uses a random number generator. Decisions made in such 
   algorithms depend on the output of random number generator. The output of a randomized 
algorithm is unpredictable and it may differ from run to run for the same input.

2. A Turing machine can use random numbers in its calculation. A Turing machine with such a 
capability is known as randomized Turing machine.

3. There are some similarities between a randomized TM and non-deterministic TM. The non-
deterministic choice of a NDTM could be based on random number.

4. The class of languages accepted by a randomized TM can be divided into two categories. 
   a. The class RP 
   b. The class ZPP

5. RP stands for randomized polynomial-time algorithm. ZPP stands for zero-error 
   probabilistic probabilistic polynomial algorithm. The class ZPP is based on a randomized TM 
   that always halts in polynomial time.

**Complexity of Primality Testing:**

1. An integer number is prime if it is divisible only by 1 and itself. An algorithm can be written 
to test whether a number is prime or not. These algorithms are found to be in following 
classes:
   a. Np 
   b. Co-NP 
   c. RP
2. Given in integer n, the problem of deciding whether n is prime is known as primality testing.

3. Algorithms for primality testing are based on the following two theorems:
   a. If ‘n’ is a prime, then a(n-1) = 1 modulo P ---------------- Fermat’s theorem
   b. The equation X^2 = 1 (modulo n) has exactly two solutions namely 1 and n-1, if n is a prime.

4. An algorithm based on Fermat’s theorem can be written with time complexity of O(n^3).

5. A randomized algorithm based on Fermat’s theorem will have a time complexity of O(log2n).

Questions:

1. Explain classes of complexity with example. (Dec-2009)
2. Write notes on complements of languages in P. (Dec-2009)

Subjects Taken by Ganesh Sir:

<table>
<thead>
<tr>
<th>Semester</th>
<th>Subject</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM-VI (Computer)</td>
<td>SPCC (System Programming and Compiler Construction)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCAJP 1.7 (Oracle Certified Associate JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
<tr>
<td>ALL</td>
<td>OCPJP 1.7 (Oracle Certified Professional JAVA Programmer)</td>
<td>Vacation + Regular</td>
</tr>
</tbody>
</table>