PSUs

(PUBLIC SECTOR UNDERTAKINGS)

ISRO, DRDO, BARC, BEL, HAL, NTPC, ONGC, BHEL, SAIL, GAIL, MTNL,
FCI, ECL, ATC, DMRC, HLL, UPRVNL, CSPEB, OPTCL....

Practice Book

COMPUTER SCIENCE & IT

Topicwise Objective Solved Questions

Topicwise Objective Solved Questions

Practice Set-I (Basic Level)

Practice Set-II (Advance Level)

3000 Solved Questions

Answers with Explanations

MADE EASY
Publications

MADE EASY Publications

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016
E-mail: infomep@madeeasy.in
Contact: 011-45124660, 8860378007

Visit us at: www.madeeasypublications.org

Practice Book for PSUs : Computer Science & IT

© Copyright, by MADE EASY Publications.
All rights are reserved. No part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photo-copying, recording or otherwise), without the prior written permission of the above mentioned publisher of this book.

First Edition: 2010
Second Edition: 2012
Third Edition: 2013
IBSN: 978-93-81069-32-5

MADE EASY PUBLICATIONS has taken due care in collecting the data and providing the solutions, before publishing this book. Inspite of this, if any inaccuracy or printing error occurs then MADE EASY PUBLICATIONS owes no responsibility. MADE EASY PUBLICATIONS will be grateful if you could point out any such error. Your suggestions will be appreciated.

© **All rights reserved by MADE EASY PUBLICATIONS.** No part of this book may be reproduced or utilized in any form without the written permission from the publisher.

PREFACE

"Theory & Practice Book on Computer Science" contains theory including basic concepts and formulae and nearly 3000 questions with answers and explanations. The authors (MADE EASY Team) are very well aware of the requirements of the Public Sector Examinations like ISRO, DRDO, BARC, BEL, HAL, NTPC, ONGC, BHEL, SAIL, GAIL, MTNL, ECL, ATC, DMRC, HLL, UPRVNL, CSPEB, OPTCL....etc. Therefore content of this book includes such questions similar to which questions are normally asked in the Public Sector Examinations.

Since last 10 years authors have closely studied the pattern & standard of the various examinations and it is found that the standard of questions vary greatly from one PSU to the other PSU. MADE EASY Team has divided the questions of each subject in two levels i.e. **'Basic Level'** & **'Advance Level'.**

The MADE EASY Team feels that this book will be highly useful for all objective type competitive examinations conducted for engineering graduates. The authors have incorporated the memory based previous PSUs questions.

Any suggestions from the readers for the improvement of this book are most welcome.

MADE EASY Team

CONTENTS

1. **Theory of Computer Science** **7-96**
   Theory : 009-043 | Practice Set I : 045-070 | Practice Set II : 071-096

2. **Programming with 'C'** **97-156**
   Theory : 079-091 | Practice Set I : 093-112 | Practice Set II : 113-134

3. **Data Structure & Algorithms** **157-217**
   Theory : 159-171 | Practice Set I : 173-195 | Practice Set II : 197-217

4. **Database Management Systems** **219-286**
   Theory : 221-231 | Practice Set I : 233-258 | Practice Set II : 259-286

5. **Operating Systems** **287-362**
   Theory : 289-305 | Practice Set I : 307-330 | Practice Set II : 331-362

6. **Digital Logic** **363-431**
   Theory : 365-378 | Practice Set I : 379-404 | Practice Set II : 405-431

7. **Computer Organization** **433-500**
   Theory : 435-443 | Practice Set I : 445-468 | Practice Set II : 469-500

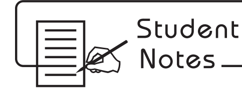
8. **Compiler Design** **501-569**
   Theory : 503-511 | Practice Set I : 513-533 | Practice Set II : 535-569

9. **Computer Network** **571-637**
   Theory : 573-595 | Practice Set I : 596-614 | Practice Set II : 615-637

10. **Software Engineering** **639-672**
    Theory : 641-648 | Practice Set I : 649-659 | Practice Set II : 661-672

❍❍❍❍

Theory of Computer Science



FINITE AUTOMATON

Introduction

We have several things to be done in designing and implementation of an automatic machine. One important and initial thing is: what is the behavior of machine? The subject that solves this purpose is known as "Automata Theory" or "Theory of Computation".

"Automata theory is a subject which describes the behavior of automatic machines mathematically."

In other words, we can say that:

"A computational model or a recognition model can be described by automata theory."

It is a philosophy of automatic machines described mathematically.

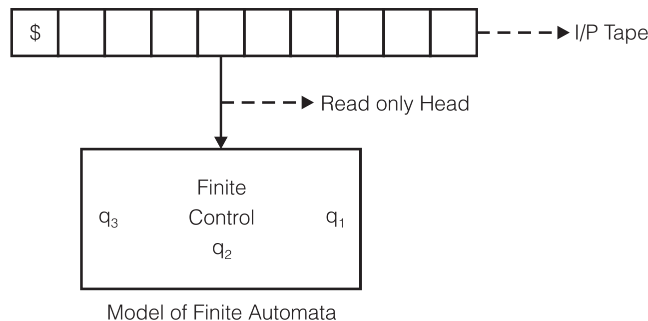
In Computer Science field, the word **"AUTOMATA"** is used for **recognizer** or **acceptor** or **transducer**.

An automata understands a set of words or strings and rejects others. One application of automata theory in Computer Science is in the design of lexical analyzer, which is a part of compiler.

Model of Finite Automaton (FA)

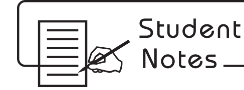
A finite automaton consists of an input tape, a finite-nonempty set of states, an input alphabet, a read-only head, a transition function which defines the change of configuration, an initial state, and a finite-nonempty set of final states.

A model of finite automata is shown in figure.



Model of Finite Automata

The head reads one symbol on the input tape and finite control controls the next configuration. The head can read either from left-to-right or right-to-left one cell at a time. The **head can't write and can't move backward** so, finite automata can't remember its previous read symbols. This is a drawback of FA.



- **Deterministic Finite Automata (DFA):** A deterministic finite automata M can be described by 5-tuple **($Q, \Sigma, \delta, q_0, F$)** where
  1. Q is a non-empty and finite set of states. It contains all the states of the machine,
  2. Σ is an input alphabet,
  3. δ is the transition function which maps $Q \times \Sigma \rightarrow Q$, that is, the head reads a symbol in its present state and moves into a next state as given by the δ function,
  4. $q_0 \in Q$, known as initial state or starting state, and
  5. $F \subseteq Q$, known as a set of final states.
- **Non-deterministic Finite Automata (NDFA or NFA):** A non-deterministic finite automata M can be described by 5-tuple ($Q, \Sigma, \delta, q_0, F$), where
  1. Q is a non-empty and finite set of states. It contains all the states.
  2. Σ is an input alphabet,
  3. δ is transition function which maps $Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$, that is, the head reads a symbol in its present state and moves into the set of next state(s). 2^Q is the power set of Q,
  4. $q_0 \in Q$, known as initial state or starting state, and
  5. $F \subseteq Q$, known as set of final states.

  It is also known as **non-deterministic finite state** machine.

  The difference between a DFA and a NFA is only in the transition function.

  The transition function for a DFA maps $Q \times \Sigma \rightarrow Q$ whereas the same for NFA maps $Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$.

Transducer or Finite State Machines (FSM)

A finite state machine is similar to finite automata (FA) except that it has the additional capability of producing output.

FSM ≡ FA + Output capability

Description of a finite state machine (FSM):

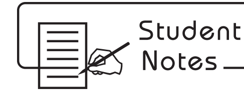
A finite state machine is described by 6-tuple ($Q, \Sigma, \Delta, \delta, \lambda, S$), where

1. Q is finite and non-empty set of states,
2. Σ is input alphabet,
3. Δ is output alphabet,
4. δ is a transition function which maps present state and input symbol on to the next state or $Q \times \Sigma \rightarrow Q$,
5. λ is an output function, and
6. $S \in Q$, is the initial state or starting state.

Types of Finite State Machines (FSM)

There are two types of FSMs. The example gives above belongs to a class of FSM called Mealy Machines. However, there is also, another class of FSM called Moore Machine.

1. **Mealy Machines:** If the output of finite state machine is dependent on present state and present input, then this model of finite state machine is known as Mealy machine.



Mealy Machine Details: A Mealy machine can be described by six-tuple (Q, Σ, Δ, δ, λ, S), where

1. Q is finite and non-empty set of states,
2. Σ is input alphabet,
3. Δ is output alphabet,
4. δ is transition function which maps present state and input symbol on to the next state or $Q \times \Sigma \rightarrow Q$,
5. λ is the output function which maps $Q \times \Sigma \rightarrow \Delta$, ((Present state, present input symbol) → Output), and
6. $S \in Q$, is the initial state or starting state.

2. **Moore Machines**: If the output of finite state machine is dependent on present state only, then this model of finite state machine is known as Moore machine.

Moore Machine Details: A Moore machine can be described by six-tuple (Q, Σ, Δ, δ, λ, S), where

1. Q is a finite and non-empty set of states,
2. Σ is the input alphabet,
3. Δ is output alphabet,
4. δ is transition function which maps present state and input symbol on to the next state or $Q \times \Sigma \rightarrow Q$,
5. λ is the output function which maps $Q \rightarrow \Delta$, (Present state → Output), and
6. $S \in Q$, is the initial state or starting state.

Limitations of FA

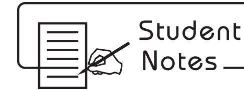
FA is the most restricted model of automatic machines. It is an abstract model of a computer system. An FA has the following limitations:

1. Input tape is read only and the only memory it can have is by moving from state to state and since there are finite number of states, an FA's memory is strictly finite.
2. The FA has only string pattern (regular expressions) recognizing power.
3. The head movement is restricted in one direction, either from left-to-right or from right-to-left.

We can modify or enhance the model of finite automata by removing one or more limitations like movement in both direction (Two-way FA), but these enhancements do not increase the recognizing power of FA.

Comparison of DFA and NFA

Title	NFA	DFA
1. Power	Same	Same
2. Supremacy	All DFA are NFA, but not all NFA are DFA.	All DFA are NFA.
3. Transition Function	Maps $Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$, the number of next states is zero or one or more.	$Q \times \Sigma \rightarrow Q$, the number of next states is exactly one.
4. Time complexity	The time needed for executing an input string is more as compare to DFA.	The time needed for executing an input string is less as compare to NFA.



Applications of FA & FSMs

We have several applications based on FA and FSMs. Some are given below:

1. Lexical Analyzers
2. Text Editors
3. Spell Checkers
4. Sequential Circuits Design (Transducer)

REGULAR LANGUAGES SETS AND FINITE AUTOMATION

Grammar

A phrase structure grammar or simply a grammar is described by four terms (V, T, S, P)

1. V is a finite, nonempty set of variables,
2. T is a finite, nonempty set of terminals,
3. $S \in V$ and known as start symbol, and
4. P is a finite, nonempty set of production rules.

Here, $V \cap T = \phi$ and P includes the production rules like $\alpha \to \beta$ (read as α derives β), where $\alpha, \beta \in (V \cup T)^*$ and α must have a symbol from V.

Language Generated by a Grammar G

The language generated by grammar G is denoted by L(G) and defined as the set of all terminal strings derived by G. Depending upon the number of strings in L(G), it may be finite, infinite or empty language.

Example: The productions of a grammar are given below. Find the language generated.

$S \to SS$, $S \to a$

Solution: Let grammar G = ({S}, {a}, {P_1, P_2}, S), where

P_1: $S \to SS$, and

P_2: $S \to a$

Using P_2: $S \Rightarrow a$

It means, $a \in L(G)$

Using P_1: $S \Rightarrow SS$ (One step derivation)

$\Rightarrow SSS$ (Replacing leftmost S by SS)

…

$\Rightarrow$ SSSSS ... n-times (Replacing leftmost S by SS (n – 1) times)

$\Rightarrow$ aSSSSS ... (Using P_2 and replacing leftmost S by a)

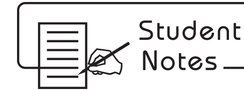
…

$\Rightarrow$ aaaaa ... n-times (Using P_2 n-times and replacing S by a)

It means, a^n for n = 2, 3, 4, ... is in L(G).

So, a and a^n for $n \geq 2$ are in L(G).

Therefore, L(G) = {a, aa, aaa, ...} = $\{a^n : \text{for } n \geq 1\}$



Chomsky Classification of Grammars/Languages

Noam Chomsky has classified the grammars in four categories (type 0 to type 3) based on the right hand side forms of the productions.

1. **Type 0:** These type of grammars are also known as phrase structured grammars, and RHS of these are free from any restriction. All grammars are type 0 grammar.
2. **Type 1:** We apply some restrictions on type 0 grammars and these restricted grammars are known as type 1 or context-sensitive grammars (CSGs). Suppose a type 0 production $\gamma\alpha\delta \to \gamma\beta\delta$ and the production $\alpha \to \beta$ is restricted such that $|\alpha| \leq |\beta|$ and $\beta \neq \varepsilon$, then this type of productions are known as type 1 production. If all productions of a grammar are of type 1 productions, then grammar is known as type 1 grammar. The language generated by a context-sensitive grammar is called context-sensitive language (CSL).
3. **Type 2:** We apply some more restrictions on RHS of type 1 productions and these productions are known as type 2 or context-free productions. A production of the form $\alpha \to \beta$, where $\alpha \in V$ and $\beta \in (V \cup T)^*$ is known as type 2 production.
4. **Type 3:** This is the most restricted type. Productions of types {$A \to a$, $A \to aB$} or {$A \to a$, $A \to Ba$}, where $A, B \in V$ and $a \in T^*$ are known as type 3 or regular grammar productions. In other words, we can say $\alpha \to \beta$ is type 3 grammer if and only if

   $\beta = VT^*$ OR
   T^*V OR
   T^*

   A production of type $S \to \varepsilon$ is also allowed if ε is in generated language.

   Example: Productions $S \to aS$, $S \to a$ are type 3 production.

   Life-linear production: A production of type $A \to Ba$ is called left-linear production.

   Right-linear production: A production of type $A \to aB$ is called right-linear production.

   Type-3 grammar cannot have mixture of left linear and right linear productions. This means all productions must be left linear (left linear grammar) or all productions must be right linear (right linear grammar).

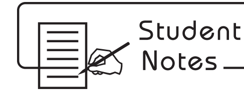
   A left-linear or right-linear grammar is called regular grammar. The language generated by a regular grammar is known as regular language.

   A production of type $A \to w$ or $A \to wB$ or $A \to Bw$, where $w \in T^*$ can be converted into the forms $A \to a$ or $A \to aB$ or $A \to Ba$, where $A, B \in V$ and $a \in T$.

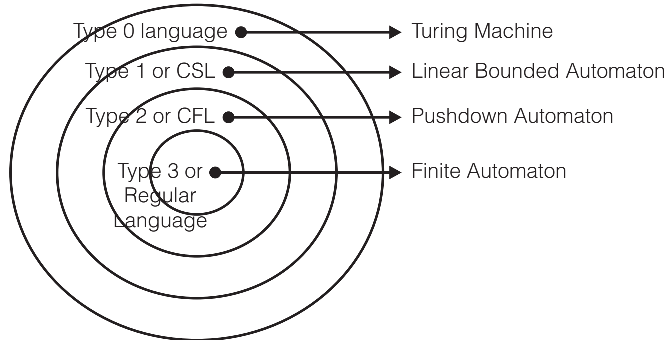
Relation Among Grammars and Languages

Type 3 $\subseteq$ Type 2 $\subseteq$ Type 1 $\subseteq$ Type 0

Type 0 is the super set and type 1 is contained in type 0, type 2 is contained in type 1, and type 3 is contained in type 2.



Language and Their Related Automaton



Languages and their related Automaton

Regular Expressions

The languages accepted by FA are regular languages and these languages are easily described by simple expressions called regular expressions.

Regular expressions are means to represent certain sets of strings in some algebraic fashion or regular expressions describe the language accepted by FA.

If Σ is an alphabet then regular expression(s) over this can be described by following rules

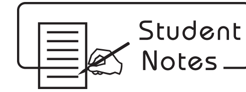
1. Any symbol from Σ, ε and ϕ are regular expressions.
2. If r1 and r2 are two regular expressions, then union of these represented as $r_1 \cup r_2$ or $r_1 + r_2$ is also a regular expression.
3. If r_1 and r_2 are two regular expressions, then concatenation of these represented as r_1r_2 is also a regular expression.
4. The Kleene closure (or iteration or reflexive-transitive closure) of a regular expression r is denoted by r* is also a regular expression.
5. If r is a regular expression then (r) is also a regular expression.
6. The regular expressions obtained by applying rules 1 to 5 once or more are also regular expressions.

Regular Sets:

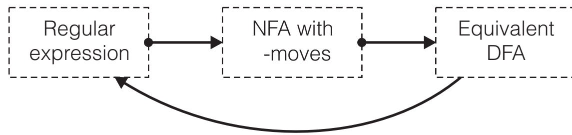
A set represented by a regular expression is called regular set.

Identities For Regular Expressions

1. $\phi + R = R + \phi = R$ (Where R is a regular expression)
2. $\phi R = R\phi = \phi$
3. $\varepsilon R = R\varepsilon = R$
4. $\varepsilon^* = \varepsilon$ and $\phi^* = \varepsilon$
5. R + R = R, where R is a regular expression
6. $RR^* = R^*R = R^+$
7. $(R^*)^* = R^*$
8. $R^*R^* = R^*$
9. $(P + Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$, where P and Q are regular expressions
10. R(P + Q) = RP + RQ and (P + Q)R = PR + QR
11. $P(QP)^* = (PQ)^*P$



Equivalence of Regular Expressions and FA



If r is regular expression, then L(r) is the language denoted by this. Following are some identities for regular languages and expressions:

1. If r, r1, and r2 are regular expressions such that $r = r_1 + r_2$, then $L(r) = L(r_1 + r_2) = L(r_1) + L(r_2)$
2. If r, r_1 and r_2 are regular expressions such that $r = r_1r_2$, then $L(r) = L(r_1r_2) = L(r_1)L(r_2)$
3. If r and r_1 are regular expressions such that $r = r_1{}^*$ then $L(r) = L(r_1{}^*)$
4. If r, r_1 and r_2 are regular expressions such that $r = r_1{}^* \cup r_2{}^*$ then $L(r) = L(r_1{}^* \cup r_2{}^*) = L(r_1{}^*) + L(r_2{}^*)$

Theorem: If r is a regular expression, then there exists an NFA with ε-moves, which accepts L(r).

Properties of Regular Sets/Languages

Regular sets/languages are closed under following properties:

1. Union
2. Concatenation
3. Kleene Closure
4. Complementation
5. Transpose
6. Intersection

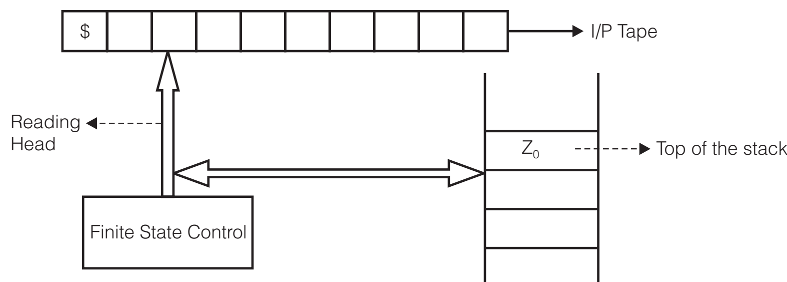
PUSHDOWN AUTOMATON (PDA)

Introduction

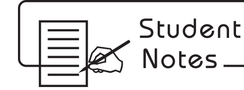
Pushdown automaton (PDA) are abstract devices that recognize context-free languages, Informally, a pushdown automaton is a finite automata outfitted with access to a potentially unlimited amount of memory in the form of single stack. A pushdown automaton is a nondeterministic device. The deterministic version of pushdown automata accepts only a subset of CFL, known as deterministic context-free languages (DCFLs), which is able to describe most of the programming languages.

Model of Pushdown Automata (PDA)

A model of pushdown automata is shown in figure. It consists of a finite tape, a reading head, which reads from the tape, a stack memory operates in **last-in-first-out** (LIFO) fashion.



Model of Pushdown Automata



CONTEXT-FREE GRAMMARS AND LANGUAGES

Mathematical Description of PDA

A pushdown automata is described by 7 tuple $(Q, \Sigma, \Gamma, \delta, s\ Z_0, F)$ where

1. Q is finite and nonempty set of states,
2. Σ is input alphabet,
3. Γ is finite and nonempty set of pushdown symbols,
4. δ is the transition function which maps
   From $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to (finite subsets of) $Q \times \Gamma$
5. $s \in Q$, is the starting state,
6. $Z_0 \in \Gamma$, is the starting (top most or initial) stack symbol, and
7. $F \subseteq Q$, is the set of final states.

Context Free Grammars

A grammar $G = (V_n, T, S, P)$ is said to be a CFG if the productions of G are of the form $A \to \alpha$, where $\alpha \in (V_n \cup S)^*$ and $A \in V_n$.

As we know that a CFG has no context; neither left nor right. This is why, it is known as CONTEXT-FREE.

Many programming languages have recursive structures that can be defined by CFGs.

Example: Consider a grammar $G = (V_n, T, S, P)$ having productions: $S \to aSa \mid bSb \mid \varepsilon$. Check the productions and find the language generated.

Solution: Let $P_1 : S \to aSa$ (RHS is terminal variable terminal)
$P_2 : S \to bSb$ (RHS is terminal variable terminal)
$P_3 : S \to \varepsilon$ (RHS is null string)

Since, all productions are of the form $A \to \alpha$, where $\alpha \in (V_n \cup \Sigma)^*$, hence G is a CFG.

Language generated: Here, $L(G) = \{ww^R : w \in (a + b)^*\}$.

Left Most and Right Most Derivations

Leftmost Derivation: If $G = (V_n, \Sigma, P, S)$ is a CFG and $w \in L(G)$, then a derivation $S \underset{L}{\Rightarrow} w$ is called leftmost derivation if and only if all steps involved in derviation have leftmost variable replacement only.

Rightmost Derivation: If $G = (V_n, \Sigma, P, S)$ is a CFG and $w \in L(G)$, then a derivation $S \underset{R}{\Rightarrow} w$ is called rightmost derivation if and only if all steps involved in derivation have rightmost variable replacement only.

Example: Consider a CFG $S \to bA \mid aB$, $A \to aS \mid aAA \mid a$, $B \to bS \mid aBB \mid b$ and find leftmost and rightmost derivations for w = aaabbabbba.

Solution:

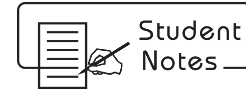
Leftmost derivation for w = aaabbabbba:

$S \Rightarrow a\underline{B}$ (Using $S \to aB$ to generate first symbol of w)

$\Rightarrow aa\underline{B}B$ (Since, second symbol is a, so we use $B \to aBB$)

$\Rightarrow aaa\underline{B}BB$ (Since, third symbol is a, so we use $B \to aBB$)

$\Rightarrow aaab\underline{B}B$ (Since, fourth symbol is b, so we use $B \to b$)



$\Rightarrow aaabb\underline{B}$ (Since, fifth symbol is b, so we use $B \to b$)

$\Rightarrow aaabba\underline{B}B$ (Since, sixth symbol is a, so we use $B \to aBB$)

$\Rightarrow aaabbab\underline{B}$ (Since, seventh symbol is b, so we use $B \to b$)

$\Rightarrow aaabbabb\underline{S}$ (Since, eight symbol is b, so we use $B \to bS$)

$\Rightarrow aaabbabb\underline{A}$ (Since, ninth symbol is b, so we use $S \to bA$)

$\Rightarrow aaabbabbba$ (Since, the tenth last symbol is a, so using $A \to a$)

Rightmost derivation for w = aaabbabbba:

$S \Rightarrow a\underline{B}$ (Using $S \to aB$ to generate first symbol of w)

$\Rightarrow aaB\underline{B}$

(We need a as the rightmost symbol and this second symbol from the left side, so we use $B \to aBB$)

$\Rightarrow aaBb\underline{S}$

(We need a as a rightmost symbol and this can be obtained from A only, we use $B \to bS$)

$\Rightarrow aaBbb\underline{A}$ (Using $S \to bA$)

$\Rightarrow aa\underline{B}bba$ (Using, $A \to a$)

$\Rightarrow aaaB\underline{B}bba$ (We need b as the fourth symbol from the right)

$\Rightarrow aaa\underline{B}bbba$ (Using $B \to b$)

$\Rightarrow aaab\underline{S}bbba$ (Using $B \to Sb$)

$\Rightarrow aaabb\underline{A}bbba$ (Using $S \to bA$)

$\Rightarrow aaabbabbba$ (Using $A \to a$)

Left Recursion

A production of the grammar G = (V_n, T, S, P) is said to be left recursive if it is of the form $A \to A\alpha$, where A is a variable and $\alpha \in (V_n \cup S)^*$.

Elimination of left recursion: Let the variable A has left recursive productions as following:

$A \to A\alpha_1 | A\alpha_2 | A\alpha_3 ... | A\alpha_n | \beta_1 | \beta_2 | \beta_3 | ... | \beta_m$, where $\beta_1, \beta_2, \beta_3, ...\beta_m$ do not begin with A, then we replace A-productions by:

$A \to \beta_1 A' | \beta_2 A' | ... | \beta_m A'$, where $A' \to \alpha_1 A' | \alpha_2 A' | a_3 A' | ... | \alpha_n A' | \varepsilon$

Example: Consider the following grammar and remove left recursion (if any) $S \to Aa | b$, $A \to Ac | Sd | \varepsilon$.

Solution: One thing noticeable here, the given grammar has not immediate left recursion. So, replacing occurrence of S in A-productions by RHS of S-productions, we have $A \to Ac | Aad | bd | \varepsilon$.

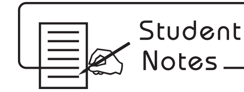
Now, the-productions have immediate left recursion. So, removing left recursion from the A-productions, we get

$A \to bdA' | A'$, $A' \to cA' | adA' | \varepsilon$

(**Note:** $\varepsilon A' \equiv A'$)

Now, the grammar without left recursion is following:

$S \to Aa | b$, $A \to bdA' | A'$, $A' \to cA' | adA' | \varepsilon$



Left Factoring

Two or more productions of a variable A of the grammar G = (V_n, T, S, P) are said to have left factoring If A-productions are of the form $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n$, where $\beta_1 \in (V_n \cup \Sigma)^*$ and does not start (prefix) with α. All these A-productions have common left factor α.

Elimination of left factoring: Let the variable A has (left factoring) productions as following:

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \beta_3 | \ldots | \alpha\beta_n | \gamma_1 | \gamma_2 | \ldots | \gamma_m$, where β_1, β_2, β_3 ... β_n and γ_1, γ_2,...γ_m do not contain α as a prefix, then we replace A-productions by:

$A \rightarrow \alpha A' | \gamma_1 | \gamma_2 | \ldots | \gamma_m$, where $A' \rightarrow \beta_1 | \beta_2 | \beta_3 | \ldots | \beta_n$

Example: Consider the grammar $S \rightarrow aSa | aa$ and remove the left factoring (if any).

Solution: $S \rightarrow aSa$ and $S \rightarrow aa$ have α = a as a left factor, so removing the left factoring, we get the productions: $S \rightarrow aS'$, $S' \rightarrow Sa | a$.

Normal Forms

If G is a CFG and productions satisfy certain restrictions, their G is said to in a normal form. There are several normal forms depending on restrictions applied. Two of these, which are popular and important, are given below:

1. Chomsky Normal From (CNF), and
2. Greibach Normal Form (GNF)

Decision Algorithms For CFLs

Theorem 2: There are algorithms to determine following for a CFL L:

1. **Empty:** In there any word in L? This is the question of emptiness.
2. **Finite:** Is L finite? This is the question of finitness.
3. **Infinite:** Is L infinite? This is the question of whether L is not finite.
4. **Membership:** For a particular string w, is w$\in$L? This is the question of membership.

   Membership algorithm for CFLs is also known as **CYK algorithm**. CYK algorithm is suggested by John Cocke and subsequently published by Daniel H. Younger and Tadao Kasami in 1965. The letters C, Y and K are taken from the names of the inventors.

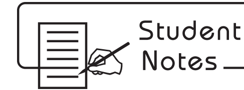
Closure Properties of CFLs

Context-Free language are closed under following properties

1. Union
2. Concatenation and
3. Kleen Closure
4. Substitution
5. Homomorphism
6. Reverse homomorphism

Context-free language are not closed under following properties:

1. Intersection
2. Complementation



Properties of Context-Free Languages:

- An alternative and equivalent definition of context-free language employs non-deterministic pushdown automata: a language is context-free if and only if such an automation can accept it.
- The union and concatenation of two context-free languages is context-free; the intersection need not be.
- The reverse of a context-free language is context free, but the complement need not be.
- Every regular language is context-free because it can be described by a regular grammar
- The intersection of a context-free language and a regular is always context-free.
- There exist context-sensitive languages, which are not context free
- To prove that a given language is not context-free, one employee the pumping lemma for context-free languages.

TURING MACHINE

Introduction

We are interested in designing of an automation (machine) that can solve our two objectives.

1. **Recognizing** and
2. **Computation**

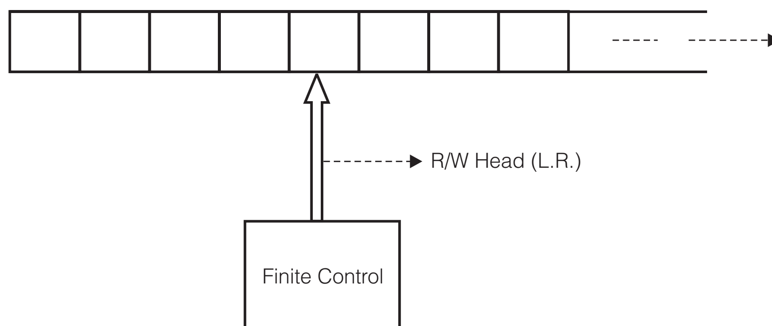
A Turing machine (TM) is a generalization of a PDA, which uses a tape instead of a tape and stack. The potential (length) of the tape is assumed to be infinite and divided into cells, and one cell can hold one input symbol. The head of TM is capable to read as well as write on the tape and can move left or right or remain static.

We say a language is decidable, if there is a TM, which will always stop in halting state. There are some type 0 languages, which are not decidable; the one most famous problem is the halting problem (HP).

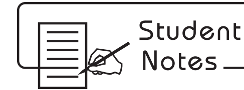
There is not a single type of grammar, which captures all decidable languages. However, there is a subset of decidable languages, which are called-sensitive languages, are generated by context-sensitive grammars.

Model of Turning Machine

A Turing machine consists of a tape, which is finite at the left end and infinite at the right end, a finite control, a read/write head as shown in figure.

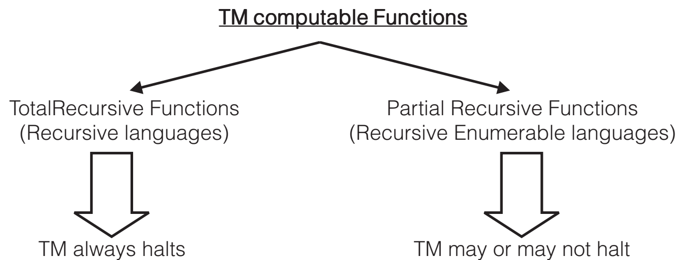


Model Turing machine



Mathematical description of Turing Machine: A TM can be described by-7 tuples (Q, Σ, Γ, δ, q_0, □, F), where

1. Q is the finite set of states, not including the halt state (h).
2. Σ is the input alphabet which is a subset of tape alphabet not including the blank symbol □.
3. Γ is a finite set of symbols called the tape alphabet.
4. δ is the transition function which maps from $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
5. □ ∈ Γ is a special symbol called blank.
6. $q_0 \in Q$ is the initial state.
7. $F \subseteq Q$ is the set of final states.



Variation of Turing Machines

Two-Stack PDA: Pushdown automaton has only one stack and can recognise only the context-free languages. We have one question here. Would the addition of a second stack to a PDA alter its power? A PDA with two stacks is known as 2PDA.

Theorem: Some Single tape TM simulates every 2PDA.

Theorem: Every single tape TM is simulated by some 2PDA.

Read-only TM and Two-way FA: We will now consider what can happen if we restrict the TM to read only. We consider the read-only TM that can read only , but can't write. This simply means that, in every transition are or edge label the read and writes fields are identical. Clearly, such a TM can't produce any output, and therefore can't compute any function on its input, which requires writing. The read-only TM may therefore be considered as a FA has the additional property of being able to move its head in both directions (left and right), which is known as two-way FA.

Now we have some questions here.

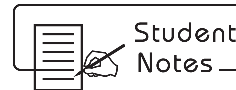
1. What class of languages is recognised by these machines?
2. Does the restriction to the TM decreases its power, or does the enhancement to the FA increase its power, or both?

By the kleene's Theorem that is any read TM can be converted to regular expression. So we conclude that the Read-only TM accepts only the regular languages.

Multiple Tracks TM: A K-track (for some fixed K >0) TM (known as KTM) has K tracks and one read/ write head that reads and writes all of them one by one.

Theorem: A KTM for some fixed K ≤ 2 can be simulated by a single track TM.

Two-way Infinite Tape TM: Our TM model has a one-way-infinite tape. That is, the tape has an end at the left beyond which the machine



should not move. But it is unbounded (infinite) to the right. Would there be any increase in power if we allowed the tape to be infinite to the left as well?

Multi-Tape TM: The multi-tape TM differs from the KTM in that it has a separate tape head for each of its multiple tapes.

Theorem: A multi-tape Turing machine is equivalent to a single tape Turing machine.

Non-deterministic TM: A non deterministic TM or NTM has a single, one way infinite tape. For a given state and input symbol has at least one choice to move (finite number of choices for the next move), each choice select several choices of path that it might follow for a given input string.

Theorem: A non-deterministic TM is equivalent to a deterministic TM.

The Off-line TM: This TM is similar to multi-tape TM, but its input is read-only and has two end markers for input; $ for left end and ψ for right end. The remaining tapes are two-way infinite tapes. This TM does not allow its head to move across the marked region on the input tape. This TM is a special type multi-tape TM, so this arrangement does not increase the power of the off-line TM. It is obvious that a off-line TM is equivalent to a one-way infinite TM.

Church's Thesis

"The principle that TMs and formal version of algorithms and no computational procedure will be considered on algorithm unless it can be implemented as a TM is known as Church's Thesis."

Universal Turing machine (UTM)

A Universal Turing machine is a specified Turing machine that can simulate the behaviour of any TM.

Recursively Enumerable (R.E.) and Recursive Languages

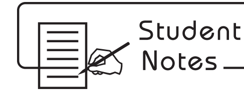
When a TM executes an input there are three possible outcomes of execution. The TM

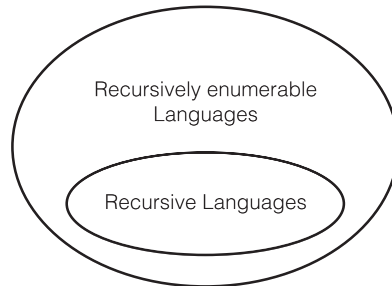
1. Halts and accepts the input,
2. Halts and rejects the input,
3. Never halts (fall into loop)

We also know that the language accepted by TM is known as recursively enumerable (R.E.) and the set of recursive languages is subset of recursively enumerable set.

The set of recursively enumerable (R.E.) languages are precisely those language that can be listed (enumerated) by a TM.

A language is recursive if there is a Turing machine that accepts every string of the language and rejects every string that is not in the language. So, we are sure about the rejection of the strings which are not in the given recursive language.





Theorem: If a language L is recursive, then it is recursively enumerable language.

Enumerating a language: To enumerate a language is to place the elements of the language in some sequence like 1,2,3,...,n. In other words, we can say, we have one-to-one correspondance with strings of the language and the natural numbers.

Theorem: All recursive enumerable languages are TM enumerable.

Theorem: The power set of an infinite set is not countable.

Theorem: Not all languages are recursively enumerable.

Theorem: Any language generated by an unrestricted grammar is recursively enumerable (R.E.) language.

Theorem: RE languages are closed under $\cup$, $\cap$, concatenation and Kleene closure.

Theorem: RE languages are not closed under complementation.

Theorem: REC languages are closed under $\cup$, $\cap$, complementation, concatenation and Kleene closure.

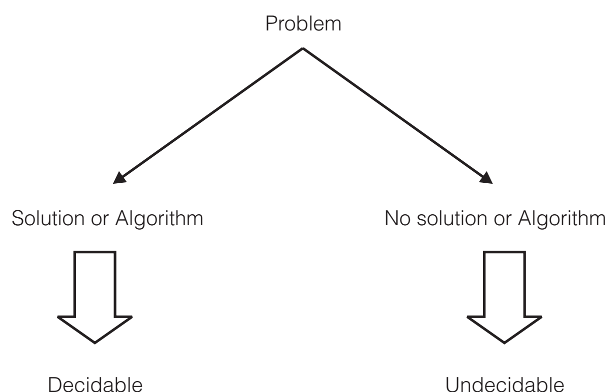
DECIDABILITY AND UNDECIDABILITY

Decidable and Undecidable Problems:

A problem is said to be decidable if

1. Its language is recursive, or
2. It has solution or answer (algorithm).

Now, we can say. “If for a problem, there an algorithm which tells that the answer is either “YES” or “NO” then problem is decidable.



If for a problem both the answer are possible; sometime “ YES” and sometime “No”, then problem is undecidable.

Decidable Problems for Finite state Automation, Regular Grammars and Regular Languages

Some decidable problems are given below:

1. Does FA accept regular language?
2. Is the power of NFA and DFA same?
3. L_1 and L_2 are two regular languages. Are these closed under following
   (i) Union (ii) Concatenation
   (iii) Intersection (iv) Complement
   (v) Transpose (vi) Kleene closure (positive transitive closure)
4. For given FA M and string w over alphabet Σ, is $w \in L$ (M)? This is decidable problem.
5. For a given FA M, is $L(M) = \phi$? This is decidable problem.
6. For a given FA M and alphabet Σ, is $L(M) = \Sigma^*$?, This is a decidable problem.
7. For a given FA M_1 and M_2 $L(M_1), L(M_2) \in \Sigma^*$, is $L(M_1) = L(M_2)$? This is a decidable problem.
8. For given two regular languages L_1 and L_2 over same alphabet Σ, is $L_1 \subset L_2$? This is a decidable problem.

Decidable and Undecidable Problems About CFLs, & CFGs

Decidable Problems: Some decidable problems about CFLs and CFG are given below:

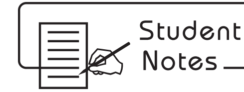
1. If L_1 and L_2 are two CFLs over some alphabets Σ, then $L_1 \cup L_2$ is CFL.
2. If L_1 and L_2 are two CFLs over some alphabets S, then L_1L_2 is CFL.
3. If L is a CFL over some alphabet Σ, then L* is a CFL.
4. If L_1 is a regular language, L_2 is a CFL then $L_1 \cup L_2$ is CFL..
5. If L_1 is a regular language, L_2 is a CFL over some alphabet S, then $L_1 \cap L_2$ is CFL
6. For a given CFG G is $L(G) = \phi$ or not?
7. For a given CFG G, finding whether L(G) is finite or not, is decidable.
8. For a given CFG G and a string w over Σ, checking whether $w \in L(G)$ or not is decidable.

Undecidable Problems: Following are some undecidable problem about CFGs and CFLs:

1. For two given CFLs L_1 and L_2, whether $L_1 \cap L_2$ is CFL or not, is undecidable. (For proof, see the theorem 6,7 of chapter 6.)
2. For a given CFL L over some alphabets Σ, whether complement of L i.e. S*-L is CFL or not, is undecidable.
3. For a given CFG G is ambiguous? This is undecidable problem
4. For two arbitrary CFGs G_1 and G_2 deciding $L(G_1) \cap L(G_2) = \phi$ or not, is undecidable.
5. For two arbitrary CFGs G_1 and G_2, deciding $L(G_1) \subseteq L(G_2)$ or not,is undecidable

Decidability and Undecidability About Turing Machines

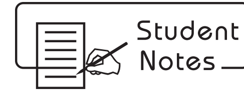
According to Church-Turing thesis, we have considered TM as an algorithm and an algorithm as a TM. So, for a problem, if there is an algorithm (solution or procedure to find answer) then problem is decidable



and TM can solve that problem. We have several problems related to computation and recognization that have no solution and these problems are undecidable.

- **Partial decidable/solvable and decidable/solvable problems:** A TM M is said to partially solve a given problem O if it provides the answer for each instance of the problem and the problem is said to be partially solvable. If all the computations of the TM are halting computations for P, then the problem P is said to **solvable**.
  A TM is said to partially decide a problem if the following two conditions are satisfied.
  1. The problem is a decision problem, and
  2. The TM accepts a given input if and only if the problem has an answer "YES" for the input, that is the TM accepts the language L = {x:x is an instance of the problem, and the problem has the answer "YES" for x}.

  A TM is said to decide a problem if it partially decides the problem and all its computations are halting computations.
  The main difference between TM M_1 that partially solves (partially decides) a problem and a TM M_2 that solves (decides) the same problem is that M_1 might reject an input by non-halting computation, whereas M_2 can reject the input only by a halting computation.
  A problem is said to be unsolvable if no algorithm can solve it, and a problem is said to be undecidable if it is a decision problem and no algorithm can decide it.
- **Decidable problems about recursive and recursive enumerable languages:**
  1. The complement of a recursive language L over some alphabet Σ is recursive.
  2. The union of two recursive languages is recursive.
     In general, recursive languages are closed under union operation
  3. A language is recursive if and only if its complement is recursive
  4. The union of two recursively enumerable languages is recursive enumerable. In general, recursive enumerable languages are closed under union operation.
  5. If a language L over some alphabet Σ and its complement $\overline{L} = \Sigma^* - L$ is recursive enumerable, then L and $\overline{L}$ are recursive languages.
  6. We have following co-theorem based on above discussion for recursive enumerable and recursive languages.
     Let L and $\overline{L}$ be two languages, where $\overline{L}$ is the complement of L, then one of the following is true:
     (i) Both L and $\overline{L}$ are recursive languages.
     (ii) Neither L nor $\overline{L}$ is recursive languages.
     (iii) If L is recursive enumerable but not recursive, then $\overline{L}$ is not recursive enumerable and vice versa.
- **Undecidable problems about turing machines:**
  Halting Problem (HP): The halting problem is a problem which can be informally stated as following:



"Given a description of an algorithm and a description of its initial arguments, determine whether the algorithm, when executed with these arguments, ever halts. The alternative is that the given algorithm runs forever without halting".

Other Undecidable Problems:

1. Does a given TM M halt on all inputs?
2. Does TM M halt for any input i.e. is $L(M) = \phi$?
3. Does TM M halt when given a blank input tape?
4. Do two Turing machines M1 and M2 accept the same language i.e. $N(M_1) = N(M_2)$?
5. Is the language L(M) finite?
6. Does L(M) contain any two strings of the same length?
7. Does L(M) contain a string of length k, for some given $k \geq 1$?
8. Rice's theorem: Any non-trivial property (a property which is true for some but not all RE languages) of RE languages is undecidable.

❍❍❍❍