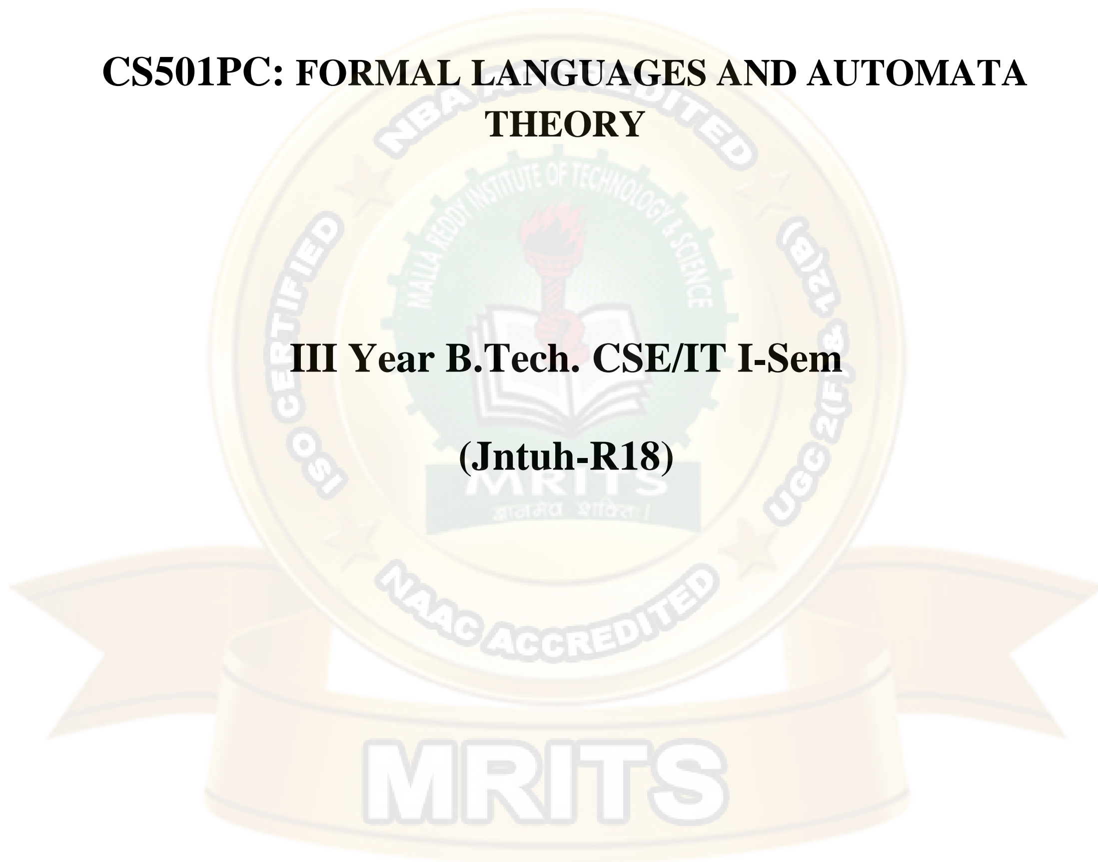
MALL REDDY INSTITUTE OF TECHNOLOGY & SCIENCE AND SCIENCE

LECTURE NOTES

On

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY



III Year B.Tech. CSE/IT I-Sem

(Jntuh-R18)

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE I-Sem

L	T	P	C
3	0	0	3

Course Objectives

1. To provide introduction to some of the central ideas of theoretical computer science from the perspective of formal languages.
2. To introduce the fundamental concepts of formal languages, grammars and automata theory.
3. Classify machines by their power to recognize languages.
4. Employ finite state machines to solve problems in computing.
5. To understand deterministic and non-deterministic machines.
6. To understand the differences between decidability and undecidability.

Course Outcomes

1. Able to understand the concept of abstract machines and their power to recognize the languages.
2. Able to employ finite state machines for modeling and solving computing problems.
3. Able to design context free grammars for formal languages.
4. Able to distinguish between decidability and undecidability.

UNIT - I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory – Alphabets, Strings, Languages, Problems.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines

UNIT - II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages, Statement of the pumping lemma, Applications of the Pumping Lemma.

Closure Properties of Regular Languages: Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

UNIT - III

Context-Free Grammars: Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages. **Push Down Automata**: Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG

UNIT - IV

Normal Forms for Context- Free Grammars: Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form Griebech Normal form.

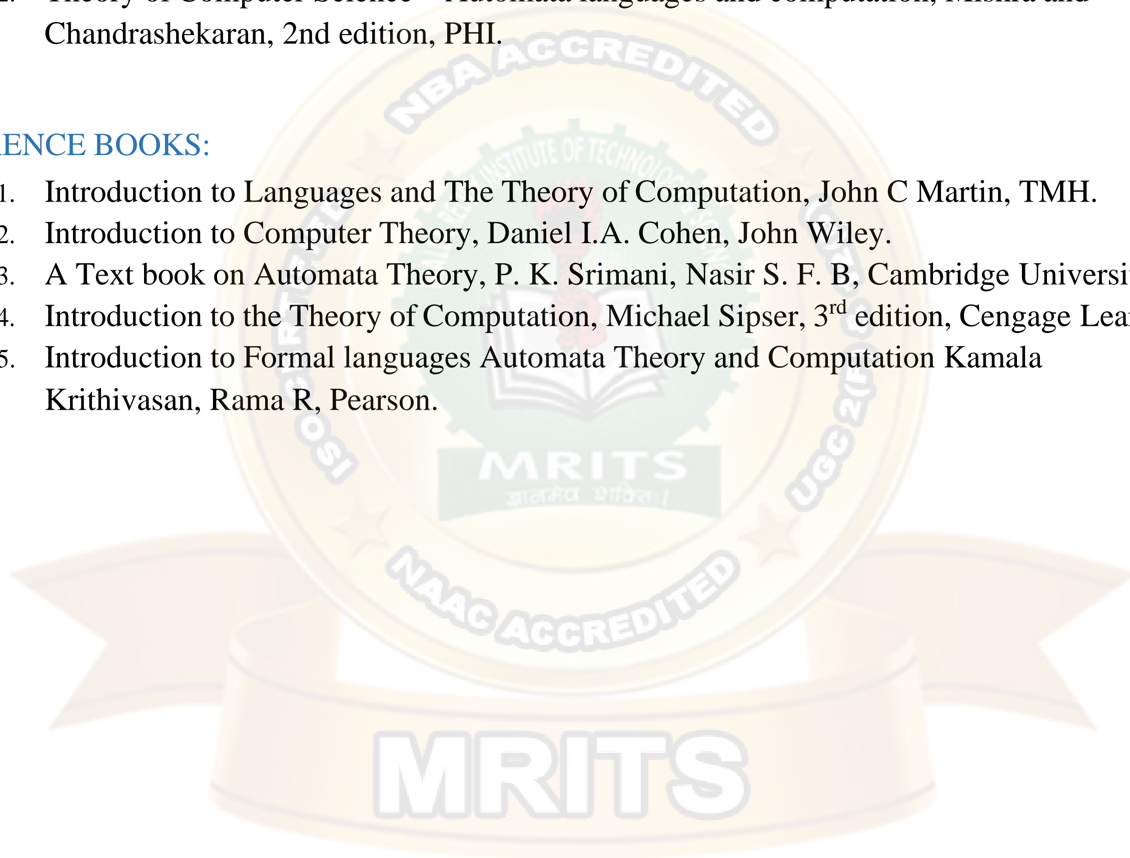
Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications
Closure Properties of Context-Free Languages: Closure properties of CFL's, Decision Properties of CFL's

Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

UNIT - V

Types of Turing machine: Turing machines and halting

Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem, Other Undecidable Problems, Counter machines.



TEXT BOOKS:

1. Introduction to Automata Theory, Languages, and Computation, 3nd Edition, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science – Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCE BOOKS:

1. Introduction to Languages and The Theory of Computation, John C Martin, TMH.
2. Introduction to Computer Theory, Daniel I.A. Cohen, John Wiley.
3. A Text book on Automata Theory, P. K. Srimani, Nasir S. F. B, Cambridge University Press.
4. Introduction to the Theory of Computation, Michael Sipser, 3rd edition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and Computation Kamala Krithivasan, Rama R, Pearson.

Formal Languages and Automata Theory

UNIT-I

CONTENTS

INTRODUCTION TO FINITE AUTOMATA

Structural Representations

Automata & Complexity

The Central Concepts Of Automata Theory-

Alphabets , Strings , Languages, Problems

NON DETERMINISTIC FINITE AUTOMATA:

Formal Definition

An Application

Text Search

Finite automata With Epsilon- Transition

DETERMINISTIC FINITE AUTOMATA :

Definition of DFA

How A DFA Processing Strings

The Language of DFA

Convertion of NFA with Epsilon Transition to NFA without

Epsilon Transition

Convertion of NFA to DFA

Moore and Melay machins

INTRODUCTION TO FINITE AUTOMATA

Automata – What is it?

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a **Finite Automaton** (FA) or **Finite State Machine** (FSM).

Formal definition of a Finite Automaton

An automaton can be represented by a 5-tuple (Q, $\sum$, δ, q_0, F), where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols, called the **alphabet** of the automaton.
- **δ** is the transition function.
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Structural Representations

The purpose of a logical framework such as LF is to provide a language for defining logical systems suitable for use in a logic-independent proof development environment. All inferential activity in an object logic (in particular, proof search) is to be conducted in the logical framework via the representation of that logic in the framework. An important tool for controlling search in an object logic, the need for which is motivated by the difficulty of reasoning about large and complex systems, is the use of structured theory presentations. In this paper a rudimentary language of structured theory presentations is presented, and the use of this structure in proof search for an arbitrary object logic is explored. The behaviour of structured theory presentations under representation in a logical framework is studied, focusing on the problem of "lifting" presentations from the object logic to the metalogic of the framework. The topic of imposing structure on logic presentations, so that logical systems may themselves be defined in a modular fashion, is also briefly consider.

Automata & Complexity

Automata Theory

● Finite automata: Computers with no memory

Motivation: Numbers, names, in Prog. Languages

Example: $x = -0.0565$

● Context-free grammars: Memory = stack

Motivation: Syntax, grammar of Prog. Languages

Example: *if (...) then (...) else (...)*

Computability Theory

● Turing Machines: "Compute until the sun dies"

● Motivation: What problems can be solved at all?

● Example: Given a program, does it have a bug?

We will prove impossible to determine!

Complexity Theory

● P, NP: Model your laptop

● Motivation: What problems can be solved fast?

● Example: Given a formula with 1000 variables like

(X OR Y) AND (X OR NOT Z) AND (Z OR Y) …

Is it satisfiable or not?

Does it take 1 thousand years or 1 second to know?

Recap

● Automata theory: Finite automata, grammars

● Computability Theory: Turing Machines

● Complexity Theory: P, NP

● Theme: Computation has many guises:

Automata, grammar, Turing Machine, formula ...

The Central Concepts Of Automata Theory-

Alphabets , Strings , Languages, Problems

Alphabet

- **Definition** − An **alphabet** is any finite set of symbols.
- **Example** − ∑ = {a, b, c, d} is an **alphabet set** where 'a', 'b', 'c', and 'd' are **symbols**.
- An alphabet is a finite, non-empty set of symbols We use the symbol ∑ (sigma) to denote an alphabet Examples: Binary: ∑ = {0,1}¬ All lower case letters: ∑ = {a,b,c,..z}¬
- Alphanumeric: ∑ = {a-z, A-Z, 0-9}

String

- **Definition** − A **string** is a finite sequence of symbols taken from ∑.
- **Example** − 'cabcad' is a valid string on the alphabet set ∑ = {a, b, c, d}

A string or word is a finite sequence of symbols chosen from ∑ (epsilon)εEmpty string is donated by λ (lambda).

Length of a String

It is the number of meaningful) present in aεsymbols/alphabets (non- string.

Length of a string w, denoted by "|w|".

E.g., if x = 010100 then |x| = 6 then|x| = ?ε 00 ε 1 ε 0 ε– If x = 01 If |W|= 0, it is called an empty string (Denoted by λ or ε) xy = concatenation of two strings x and y

Powers of an alphabet

Let ∑ be an alphabet. ∑k = the set of all strings of length k

∑* = ∑0 U ∑1 U ∑2 U …□ ∑+ = ∑1 U ∑2 U ∑3 U …

- **Definition** − It is the number of symbols present in a string. (Denoted by **|S|**).
- **Examples** −
  - If S = 'cabcad', |S|= 6
  - If |S|= 0, it is called an **empty string** (Denoted by **λ** or **ε**)

Kleene Closure / Star

- **Definition** − The Kleene star, **∑***, is a unary operator on a set of symbols or strings, ∑, that gives the infinite set of all possible strings of all possible lengths over ∑ including **λ**.
- **Representation** − $\sum^* = \sum_0 \cup \sum_1 \cup \sum_2 \cup \ldots\ldots$ where $\sum_p$ is the set of all possible strings of length p.
- **Example** − If ∑ = {a, b}, ∑* = {λ, a, b, aa, ab, ba, bb,… .... }

Kleene Closure Plus

- **Definition** − The set $\sum^+$ is the infinite set of all possible strings of all possible lengths over ∑ excluding λ.
- **Representation** − $\sum^+ = \sum_1 \cup \sum_2 \cup \sum_3 \cup \ldots\ldots$

  $\sum^+ = \sum^* - \{ \lambda \}$

- **Example** − If ∑ = { a, b } , $\sum^+$ = { a, b, aa, ab, ba, bb,… ....}

Language

- **Definition** − A language is a subset of ∑* for some alphabet ∑. It can be finite or infinite.

Example –

1. If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, bb, ba, bb}

2. Let L be the language of all strings consisting of n 0's followed by n 1's: ,01,0011,000111,…}εL = {
2. Let L be the language of all strings of with equal number of 0's and 1's: ,01,10,0011,1100,0101,1010,1001,…}εL = { 3. If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = { ab, bb, ba, bb} Ø denotes the Empty language }; Is L=Ø?ε•

The Membership Problem ∑*and a language L over ∑,∈Given a string w L.∈decide whether or not w Example: Let w = 100011 the language of strings with equal∈Q) Is w number of 0s and 1s?

Operations on languages: –Union: LUM= {w| w ∈ Lor w∈M}.
–Intersection: L∩M={w |w ∈ L and w∈ M}.
–Subtraction:L−M={w|w∈Landw /∈M}.
–Complementation:L={w|w∈Σ∗−L}.
–Concatenation:LM={w|w∈Landx/∈M}.
Note: if L and M are over different alphabets, say Σ1 and Σ2, then the resulting
language can be taken to be over Σ1UΣ2. In this course, it will not happen.
Notation: we will write LM instead of L ·M .
Also, concatenation is associative (i.e.L(MN) = (LM)N), so we drop the brackets and write LMN.
Note: for any language L ,∅ L=L∅=∅.
And also {ε}L=L{ε}=L.

– Lk = LL..... L| {z }k times for k > 0; L0 = {"}.
Exercise: Prove that for any language L, and any m _ 0, n _ 0, LmLn = Lm+n.
– Kleene closure (or star closure): L_ = L0 [L1 [L2 [. . . .

– Reverse: LR = {wR|w 2 L}.

NON DETERMINISTIC FINITE AUTOMATA:

Finite Automaton can be classified into two types −

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

Non deterministic finite automata(NFA)

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

An NDFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the alphabets.
- **δ** is the transition function where δ: $Q \times \sum \rightarrow 2^Q$

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)

- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).

- **F** is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of an NDFA: (same as DFA)

- An NFA is a five-tuple: M = (Q, Σ, δ, q_0, F)

Q A <u>finite</u> set of states
Σ A <u>finite</u> input alphabet
q_0 The initial/starting state, q_0 is in Q
F A set of final/accepting states, which is a subset of Q
δ A transition function, which is a total function from Q x Σ to 2^Q

δ: (Q x Σ) -> $\mathbf{2^Q}$ -2^Q is the power set of Q, the set of all subsets of Q δ(q,s) -The set of all states p such that there is a transition l

abeled s from q to p δ(q,s) is a

function from Q x S to 2^Q (but

not to Q)

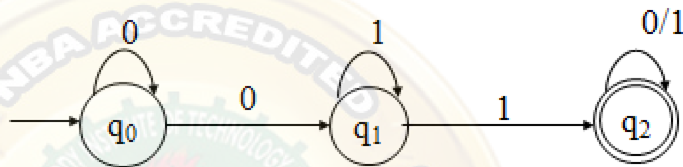
- Let M = (Q, Σ, δ,q_0,F) be an NFA and let w be in Σ*. Then w is *accepted* by M iff δ({q_0}, w) contains at least one state in F.

- Let M = (Q, Σ, δ, q_0, F) be an NFA. Then the *language accepted* by M is the set: L(M) = {w | w is in Σ* and δ({q_0},w) contains at least one state in F}

- Another equivalent definition:
  L(M) = {w | w is in Σ* and w is accepted by M}

- Example: some 0's followed by some 1's

Q = {q_0, q_1, q_2}
Σ = {0, 1}
Start state is q_0
F = {q_2}



δ:

	0	1
q_0	{q_0, q_1}	{}
q_1	{}	{q_1, q_2}
q_2	{q_2}	{q_2}

Example

An Application - Text Search

Automata theory is the study of abstract machines and automata. It also includes the computational problems that can be solved using them . In the theory of computation,the simpler abstract machine is finite automata. The otherimportant abstract machines are 1. Pushdown Automata2.Turing Machine. The finite automata proposed to model brain function of the human. The simplest example for finite automata is the switch with two states "on" and "off" [1].The Finite Automata is the five tuples combination focusing on states and transition through input characters. Infigure 1 the ending state is ON, starting state is OFF and collection of states are OFF and ON. It is having only a single input PUSH for making the transition from the state OFF to ON, then ON to OFF. The switch is the simplest practical application of finite automata.

Acceptors, Classifiers, and Transducers

Acceptor (Recognizer)

An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier

A **classifier** has more than two final states and it gives a single output when it terminates.

Transducer

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types −

- **Mealy Machine** − The output depends both on the current state and the current input.
- **Moore Machine** − The output depends only on the current state.

Finite automata With Epsilon- Transition

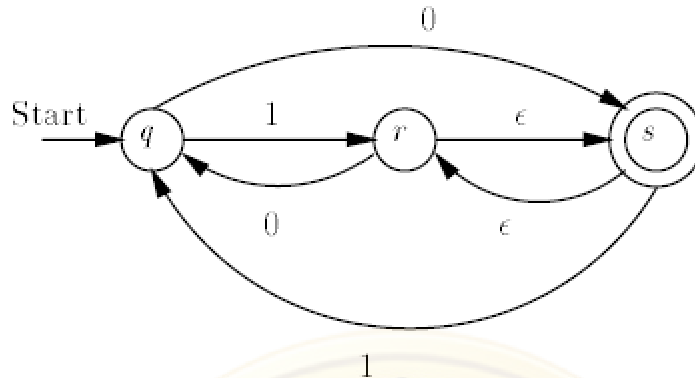
Allow to be a label on arcs.

Nothing else changes: acceptance of w is still the existence of a path from the start state to an accepting state with label w.

✦ But can appear on arcs, and means the

empty string (i.e., no visible contribution to w).

Example



001 is accepted by the path q; s; r; q; r; s, with label 0 01 = 001.

DETERMINISTIC FINITE AUTOMATA :

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where −

- **Q** is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- **δ** is the transition function where δ: Q × ∑ → Q
- **q_0** is the initial state from where any input is processed (q_0 ∈ Q).
- **F** is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram**.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Let a deterministic finite automaton be →

- Q = {a, b, c},
- ∑ = {0, 1},
- q_0 = {a},
- F = {c}, and

Transition function δ as shown by the following table –

Present state	next state for input 0	next state for input 1
A	A	B
B	C	A
C	B	C

DFA vs NDFA

The following table lists the differences between DFA and NDFA.

DFA	**NDFA**
The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*.	The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

How A DFA Processing Strings

Acceptors, Classifiers, and Transducers

Acceptor (Recognizer)

An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier

A **classifier** has more than two final states and it gives a single output when it terminates.

Transducer

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types −

- **Mealy Machine** − The output depends both on the current state and the current input.
- **Moore Machine** − The output depends only on the current state.

Acceptability by DFA and NDFA

A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a DFA/NDFA (Q, ∑, δ, q_0, F), iff

$\boldsymbol{\delta^*(q_0, S) \in F}$

The language **L** accepted by DFA/NDFA is

$\boldsymbol{\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \in F\}}$

A string S′ is not accepted by a DFA/NDFA (Q, ∑, δ, q_0, F), iff

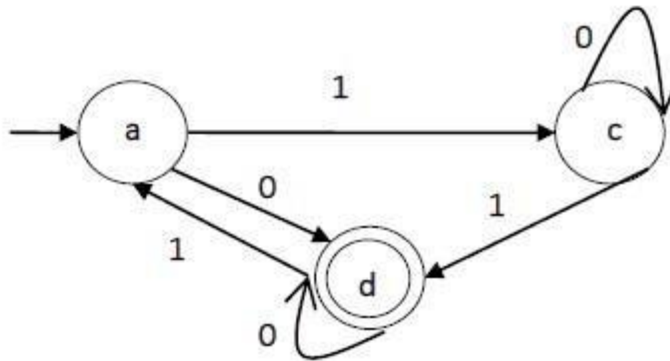
$\boldsymbol{\delta^*(q_0, S') \notin F}$

The language L′ not accepted by DFA/NDFA (Complement of accepted language L) is

$\boldsymbol{\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \notin F\}}$

Example

Let us consider the DFA shown in Figure 1.3. From the DFA, the acceptable strings can be derived.



Strings accepted by the above DFA: {0, 00, 11, 010, 101,........... }

Strings not accepted by the above DFA: {1, 011, 111,........ }

The Language of DFA

- A DFA is a five-tuple: $M = (Q, \Sigma, \delta, q_0, F)$

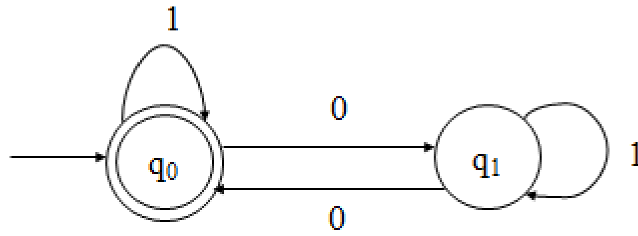
Q	A finite set of states
Σ	A finite input alphabet
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to Q

$\delta: (Q \times \Sigma) \rightarrow Q$ δ is defined for any q in Q and s in Σ, and $\delta(q,s) = q'$ is equal to another state q' in Q.

Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

- For Example #1:

$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let w be in Σ^*. Then w is *accepted* by M iff

  $\delta(q_0,w) = p$ for some state p in F.

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then the *language accepted* by M is the set:

  $$L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } \delta(q_0,w) \text{ is in } F\}$$

- Another equivalent definition:

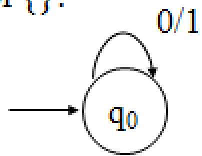
  $$L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } w \text{ is accepted by } M\}$$

- Let L be a language. Then L is a *regular language* iff there exists a DFA M such that L = L(M).

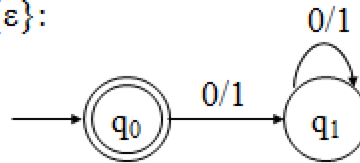
- Notes:
  - A DFA $M = (Q, \Sigma, \delta, q_0, F)$ partitions the set Σ^* into two sets: L(M) and Σ^* - L(M).
  - If L = L(M) then L is a subset of L(M) and L(M) is a subset of L.
  - Similarly, if $L(M_1) = L(M_2)$ then $L(M_1)$ is a subset of $L(M_2)$ and $L(M_2)$ is a subset of $L(M_1)$.

- Let $\Sigma = \{0, 1\}$. Give DFAs for $\{\}$, $\{\varepsilon\}$, Σ^*, and Σ^+.

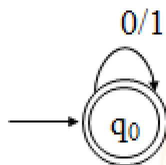
For $\{\}$:



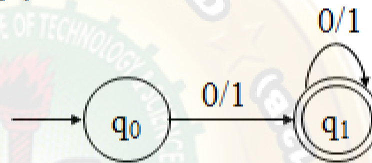
For $\{\varepsilon\}$:

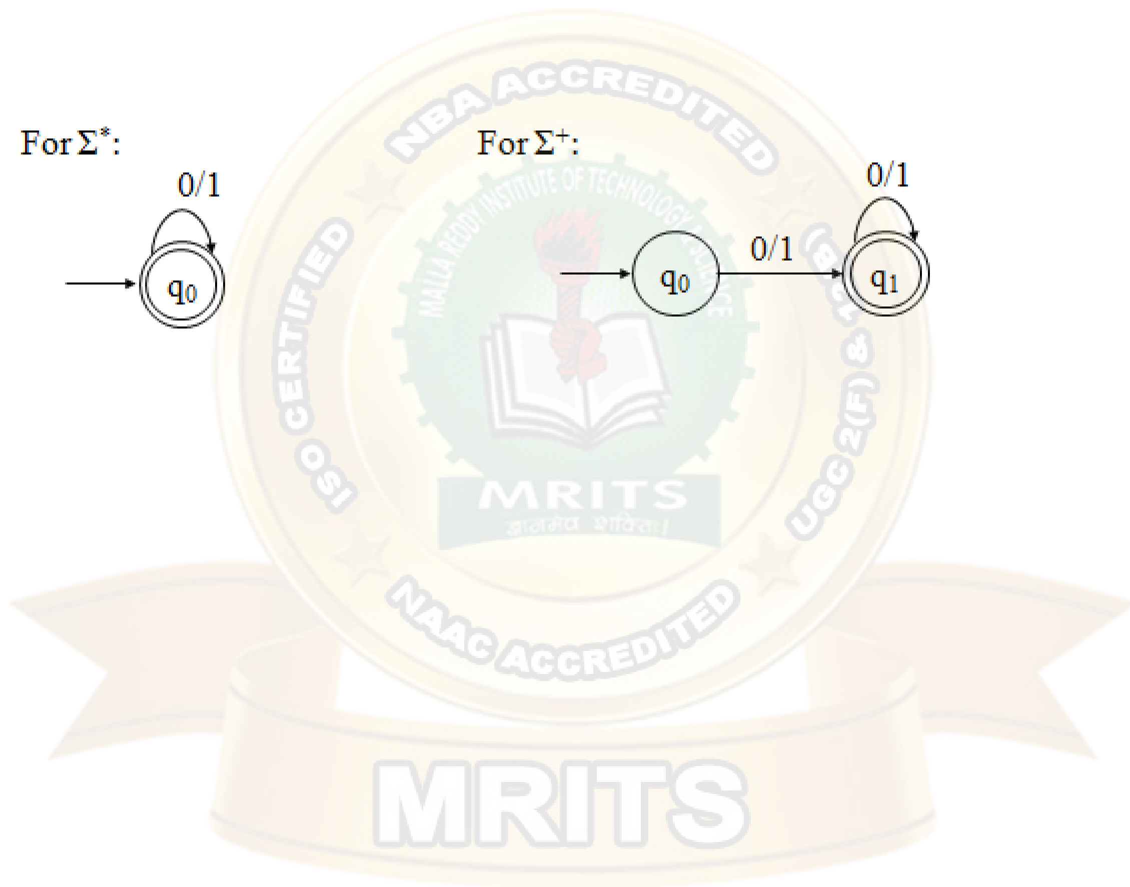


For Σ^*:



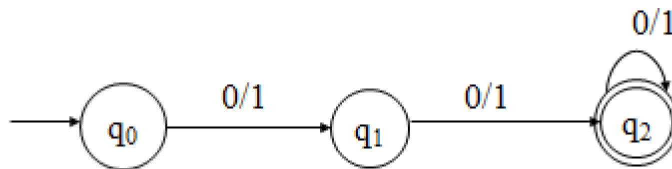
For Σ^+:



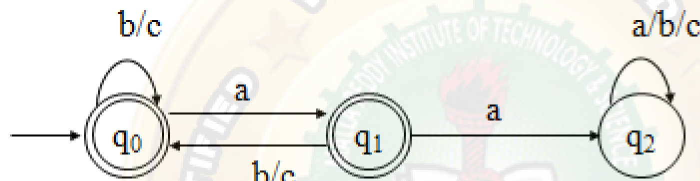


- Give a DFA M such that:

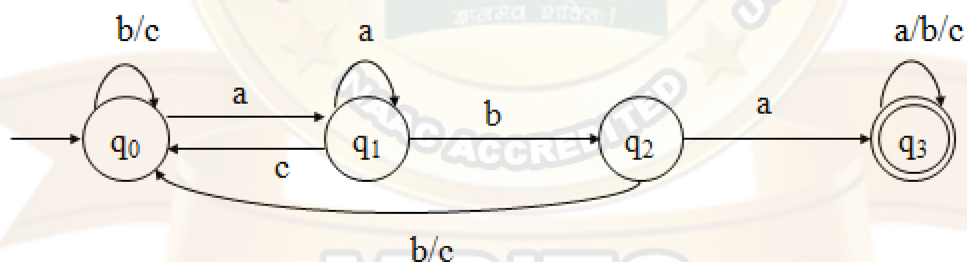
L(M) = {x | x is a string of 0's and 1's and |x| >= 2}



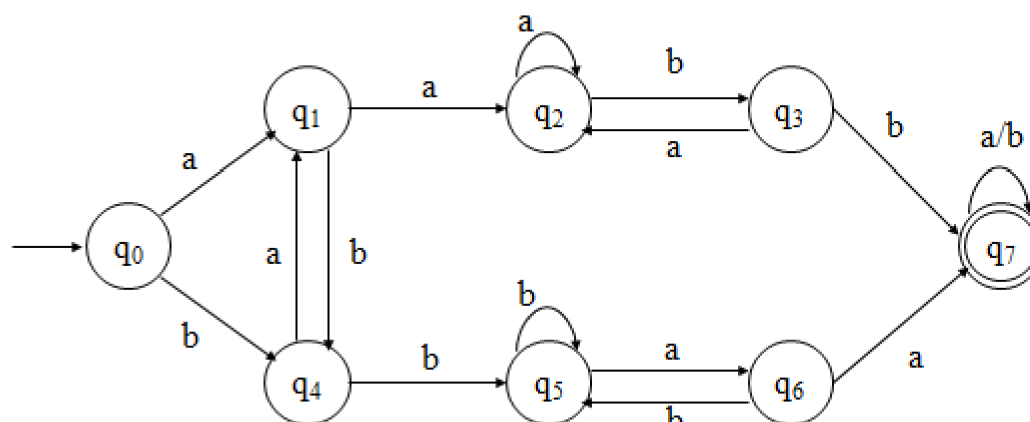
L(M) = {x | x is a string of (zero or more) a's, b's and c's such that x does not contain the substring *aa*}



L(M) = {x | x is a string of a's, b's and c's such that x contains the substring *aba*}



L(M) = {x | x is a string of a's and b's such that x contains both *aa* and *bb*}



Convertion of NFA with Epsilon Transition to NFA without

Epsilon Transition

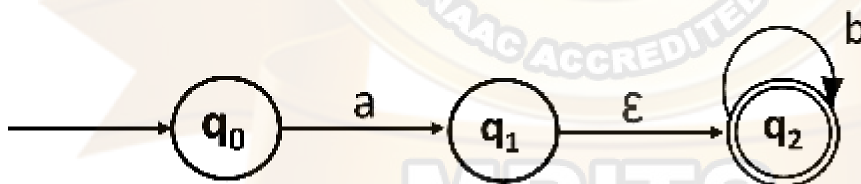
Eliminating ε Transitions

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure{q1} where qi ∈ Q.
2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

Example:

Convert the following NFA with ε to NFA without ε.



Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

ε-closure(q0) = {q0}
ε-closure(q1) = {q1, q2}
ε-closure(q2) = {q2}

Now the δ' transition on each input symbol is obtained as:

δ'(q0, a) = ε-closure(δ(δ^(q0, ε),a))
= ε-closure(δ(ε-closure(q0),a))
= ε-closure(δ(q0, a))

$= \varepsilon\text{-closure}(q1) = \{q1, q2\}$

$\delta'(q0, b) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q0, \varepsilon),b))$
$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q0),b))$
$= \varepsilon\text{-closure}(\delta(q0, b))$
$= \Phi$

Now the δ' transition on q1 is obtained as:

$\delta'(q1, a) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q1, \varepsilon),a))$
$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q1),a))$
$= \varepsilon\text{-closure}(\delta(q1, q2), a)$
$= \varepsilon\text{-closure}(\delta(q1, a) \cup \delta(q2, a))$
$= \varepsilon\text{-closure}(\Phi \cup \Phi)$
$= \Phi$

$\delta'(q1, b) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q1, \varepsilon),b))$
$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q1),b))$
$= \varepsilon\text{-closure}(\delta(q1, q2), b)$
$= \varepsilon\text{-closure}(\delta(q1, b) \cup \delta(q2, b))$
$= \varepsilon\text{-closure}(\Phi \cup q2)$
$= \{q2\}$

The δ' transition on q2 is obtained as:

$\delta'(q2, a) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q2, \varepsilon),a))$
$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q2),a))$
$= \varepsilon\text{-closure}(\delta(q2, a))$
$= \varepsilon\text{-closure}(\Phi)$
$= \Phi$

$\delta'(q2, b) = \varepsilon\text{-closure}(\delta(\delta\hat{}(q2, \varepsilon),b))$
$= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q2),b))$
$= \varepsilon\text{-closure}(\delta(q2, b))$
$= \varepsilon\text{-closure}(q2)$
$= \{q2\}$

Now we will summarize all the computed δ' transitions:

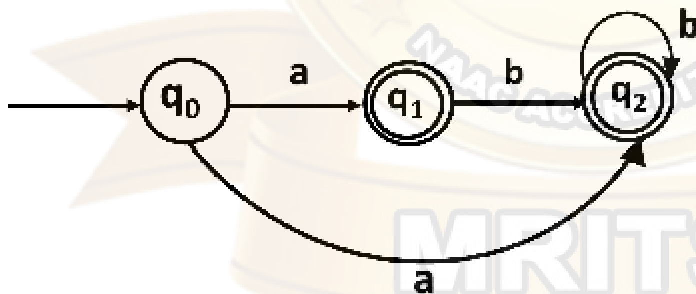
$\delta'(q0, a) = \{q0, q1\}$
$\delta'(q0, b) = \Phi$
$\delta'(q1, a) = \Phi$

δ'(q1, b) = {q2}
δ'(q2, a) = Φ
δ'(q2, b) = {q2}

The transition table can be:

States	a	b
→q0	{q1, q2}	Φ
*q1	Φ	{q2}
*q2	Φ	{q2}

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:



Conversion from NFA to DFA

In this section, we will discuss the method of converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

Steps for converting NFA to DFA:

Step 1: Initially Q' = ϕ

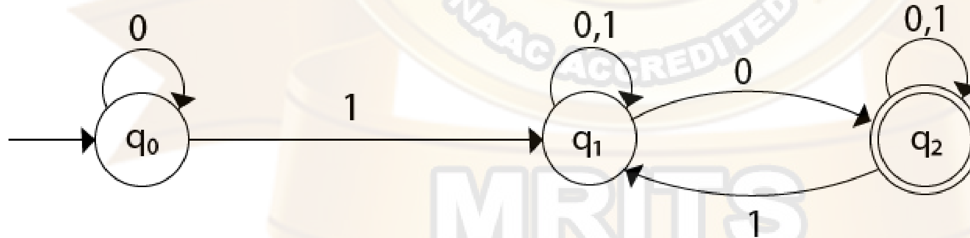
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)

Example 1:

Convert the NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1

q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

δ'([q0], 0) = [q0]
δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

δ'([q1], 0) = [q1, q2] (**new** state generated)
δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

δ'([q2], 0) = [q2]
δ'([q2], 1) = [q1, q2]

Now we will obtain δ' transition on [q1, q2].

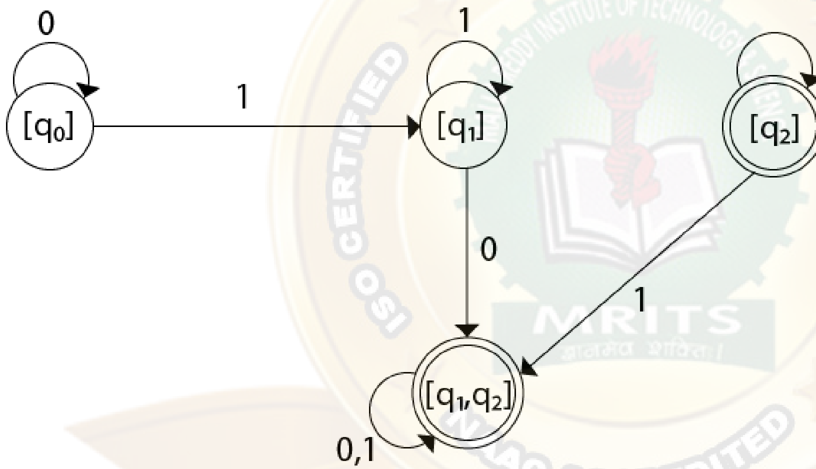
δ'([q1, q2], 0) = δ(q1, 0) ∪ δ(q2, 0)
= {q1, q2} ∪ {q2}
= [q1, q2]
δ'([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)
= {q1} ∪ {q1, q2}
= {q1, q2}
= [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]

[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

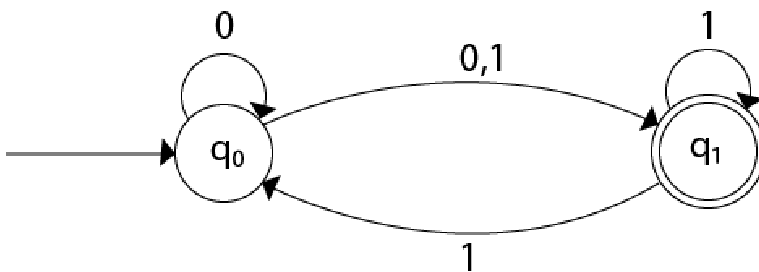
The Transition diagram will be:



The state q2 can be eliminated because q2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

Now we will obtain δ' transition for state q0.

δ'([q0], 0) = {q0, q1}
= [q0, q1] (**new** state generated)
δ'([q0], 1) = {q1} = [q1]

The δ' transition for state q1 is obtained as:

δ'([q1], 0) = ϕ
δ'([q1], 1) = [q0, q1]

Now we will obtain δ' transition on [q0, q1].

δ'([q0, q1], 0) = δ(q0, 0) ∪ δ(q1, 0)
= {q0, q1} ∪ ϕ
= {q0, q1}
= [q0, q1]

Similarly,

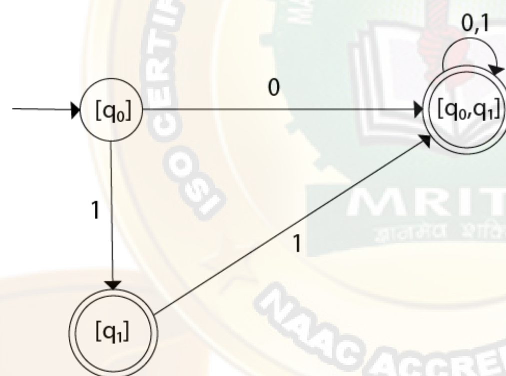
δ'([q0, q1], 1) = δ(q0, 1) ∪ δ(q1, 1)
= {q1} ∪ {q0, q1}
= {q0, q1}
= [q0, q1]

As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

The Transition diagram will be:

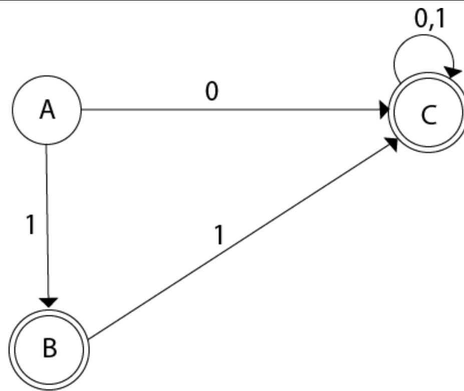


Even we can change the name of the states of DFA.

Suppose

A = [q0]
B = [q1]
C = [q0, q1]

With these new names the DFA will be as follows:



Conversion from NFA with ε to DFA

Non-deterministic finite automata(NFA) is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 states. It can contain ε move. It can be represented as M = { Q, ∑, δ, q0, F}.

Where

Q: finite set of states
∑: finite set of the input symbol
q0: initial state
F: **final** state
δ: Transition function

NFA with ∈ move: If any FA contains ε transaction or move, the finite automata is called NFA with ∈ move.

ε-closure: ε-closure for a given state A means a set of states which can be reached from the state A with only ε(null) move including the state A itself.

Steps for converting NFA with ε to DFA:

Step 1: We will take the ε-closure for the starting state of NFA as a starting state of DFA.

Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

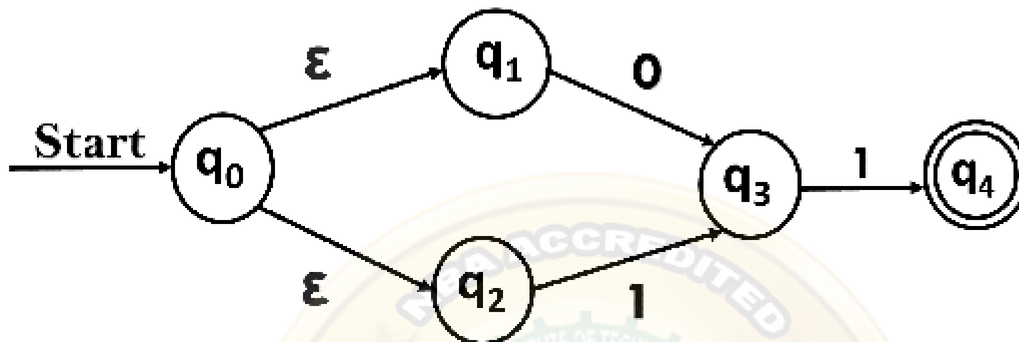
Step 3: If we found a new state, take it as current state and repeat step 2.

Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

Example 1:

Convert the NFA with ε into its equivalent DFA.



Solution:

Let us obtain ε-closure of each state.

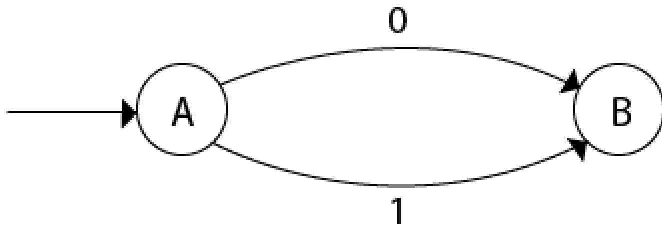
ε-closure {q0} = {q0, q1, q2}
ε-closure {q1} = {q1}
ε-closure {q2} = {q2}
ε-closure {q3} = {q3}
ε-closure {q4} = {q4}

Now, let ε-closure {q0} = {q0, q1, q2} be state A.

Hence

```
δ'(A, 0) = ε-closure {δ((q0, q1, q2), 0) }
             = ε-closure {δ(q0, 0) U δ(q1, 0) U δ(q2, 0) }
             = ε-closure {q3}
             = {q3}           call it as state B.

δ'(A, 1) = ε-closure {δ((q0, q1, q2), 1) }
             = ε-closure {δ((q0, 1) U δ(q1, 1) U δ(q2, 1) }
             = ε-closure {q3}
             = {q3} = B.
```

The partial DFA will be



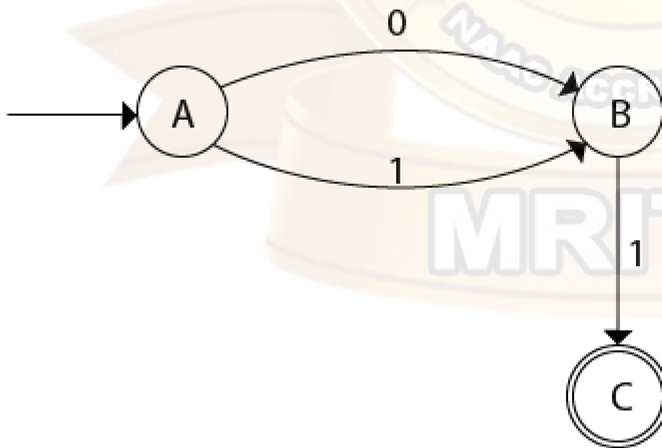
Now,

$\delta'(B, 0) = \varepsilon\text{-closure } \{\delta(q3, 0)\}$
$= \phi$
$\delta'(B, 1) = \varepsilon\text{-closure } \{\delta(q3, 1)\}$
$= \varepsilon\text{-closure } \{q4\}$
$= \{q4\}$ **i.e. state C**

For state C:

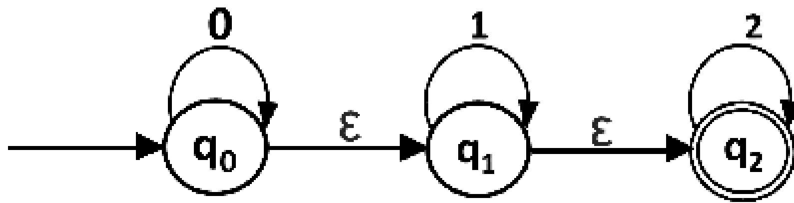
$\delta'(C, 0) = \varepsilon\text{-closure } \{\delta(q4, 0)\}$
$= \phi$
$\delta'(C, 1) = \varepsilon\text{-closure } \{\delta(q4, 1)\}$
$= \phi$

The DFA will be,



Example 2:

Convert the given NFA into its equivalent DFA.



Solution: Let us obtain the ε-closure of each state.

ε-closure(q0) = {q0, q1, q2}
ε-closure(q1) = {q1, q2}
ε-closure(q2) = {q2}

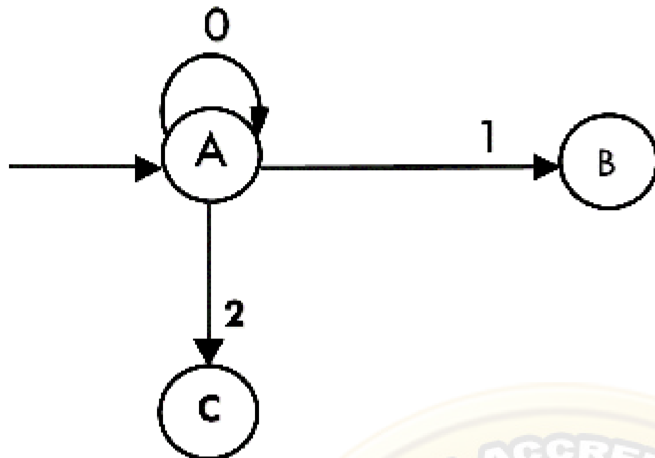
Now we will obtain δ' transition. Let ε-closure(q0) = {q0, q1, q2} call it as **state A**.

```
δ'(A, 0) = ε-closure{δ((q0, q1, q2), 0)}
               = ε-closure{δ(q0, 0) U δ(q1, 0) U δ(q2, 0)}
               = ε-closure{q0}
               = {q0, q1, q2}

δ'(A, 1) = ε-closure{δ((q0, q1, q2), 1)}
              = ε-closure{δ(q0, 1) U δ(q1, 1) U δ(q2, 1)}
               = ε-closure{q1}
               = {q1, q2}         call it as state B

δ'(A, 2) = ε-closure{δ((q0, q1, q2), 2)}
               = ε-closure{δ(q0, 2) U δ(q1, 2) U δ(q2, 2)}
               = ε-closure{q2}
               = {q2}         call it state C
```

Thus we have obtained

δ'(A, 0) = A
δ'(A, 1) = B
δ'(A, 2) = C

The partial DFA will be:



Now we will find the transitions on states B and C for each input.

Hence

δ'(B, 0) = ε-closure{δ((q1, q2), 0)}
= ε-closure{δ(q1, 0) U δ(q2, 0)}
= ε-closure{ϕ}
= ϕ

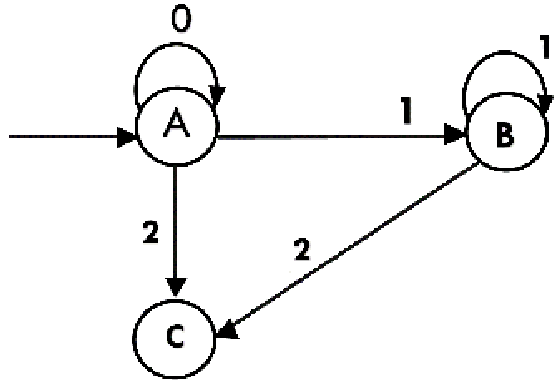
δ'(B, 1) = ε-closure{δ((q1, q2), 1)}
= ε-closure{δ(q1, 1) U δ(q2, 1)}
= ε-closure{q1}
= {q1, q2} **i.e. state B itself**

δ'(B, 2) = ε-closure{δ((q1, q2), 2)}
= ε-closure{δ(q1, 2) U δ(q2, 2)}
= ε-closure{q2}
= {q2} **i.e. state C itself**

Thus we have obtained

δ'(B, 0) = ϕ
δ'(B, 1) = B
δ'(B, 2) = C

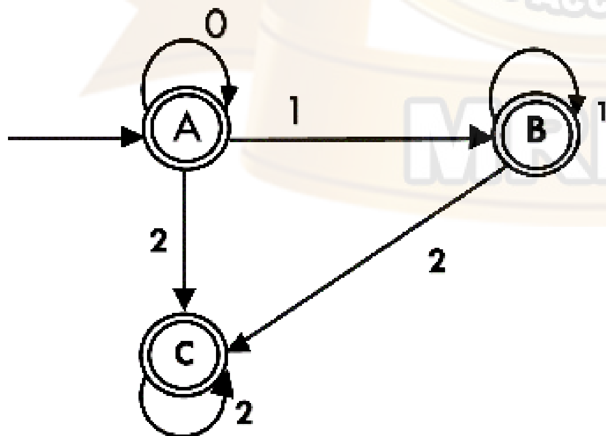
The partial transition diagram will be



Now we will obtain transitions for C:

```
δ'(C, 0) = ε-closure{δ(q2, 0)}
             = ε-closure{ϕ}
             = ϕ

δ'(C, 1) = ε-closure{δ(q2, 1)}
             = ε-closure{ϕ}
             = ϕ

δ'(C, 2) = ε-closure{δ(q2, 2)}
             = {q2}
```

Hence the DFA is



As A = {q0, q1, q2} in which final state q2 lies hence A is final state. B = {q1, q2} in which the state q2 lies hence B is also final state. C = {q2}, the state q2 lies hence C is also a final state.

FINITE AUTOMATA WITH OUTPUT

Moore and Melay machins

Moore Machine

Moore machine is a finite state machine in which the next state is decided by the current state and current input symbol. The output symbol at a given time depends only on the present state of the machine. Moore machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ) where,

Q: finite set of states

q0: initial state of machine

∑: finite set of input symbols

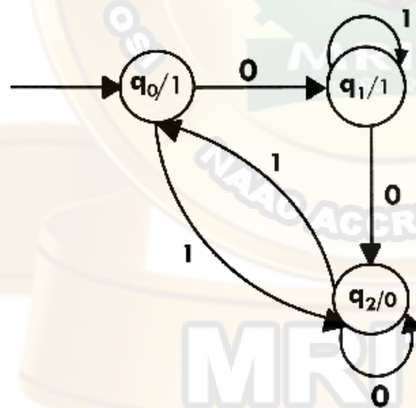
O: output alphabet

δ: transition function where Q × ∑ → Q

λ: output function where Q → O

Example 1:

The state diagram for Moore Machine is



Transition table for Moore Machine is:

NBA ACCREDITED NAAC ACCREDITED MRITS

Current State	Next State (δ)		Output(λ)
	0	1	
q_0	q_1	q_2	1
q_1	q_2	q_1	1
q_2	q_2	q_0	0

In the above Moore machine, the output is represented with each input state separated by /. The output length for a Moore machine is greater than input by 1.

Input: 010

Transition: δ (q0,0) => δ(q1,1) => δ(q1,0) => q2

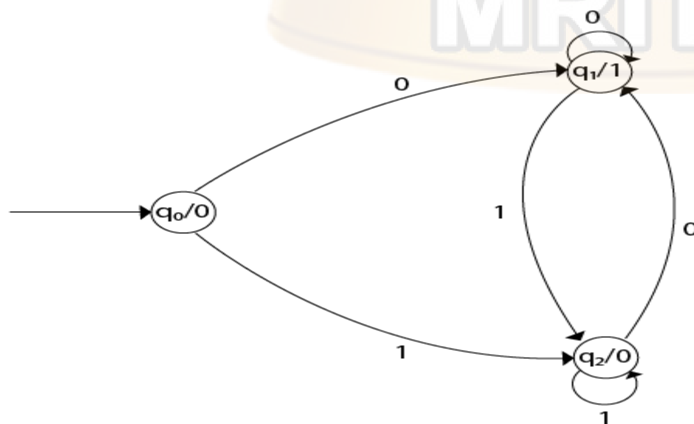
Output: 1110(1 for q0, 1 for q1, again 1 for q1, 0 for q2)

Example 2:

Design a Moore machine to generate 1's complement of a given binary number.

Solution: To generate 1's complement of a given binary number the simple logic is that if the input is 0 then the output will be 1 and if the input is 1 then the output will be 0. That means there are three states. One state is start state. The second state is for taking 0's as input and produces output as 1. The third state is for taking 1's as input and producing output as 0.

Hence the Moore machine will be,



For instance, take one binary number 1011 then

Input		1	0	1	1
State	q0	q2	q1	q2	q2
Output	0	0	1	0	0

Thus we get 00100 as 1's complement of 1011, we can neglect the initial 0 and the output which we get is 0100 which is 1's complement of 1011. The transaction table is as follows:

Current State	Next State (δ)		Output (λ)
	0	1	
→ q_0	q_1	q_2	0
q_1	q_1	q_2	1
q_2	q_1	q_2	0

Thus Moore machine M = (Q, q0, ∑, O, δ, λ); where Q = {q0, q1, q2}, ∑ = {0, 1}, O = {0, 1}. the transition table shows the δ and λ functions.

Mealy Machine

A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. In the Mealy machine, the output is represented with each input symbol for each state separated by /. The Mealy machine can be described by 6 tuples (Q, q0, ∑, O, δ, λ') where

Q: finite set of states

q0: initial state of machine

∑: finite set of input alphabet

O: output alphabet

δ: transition function where Q × ∑ → Q

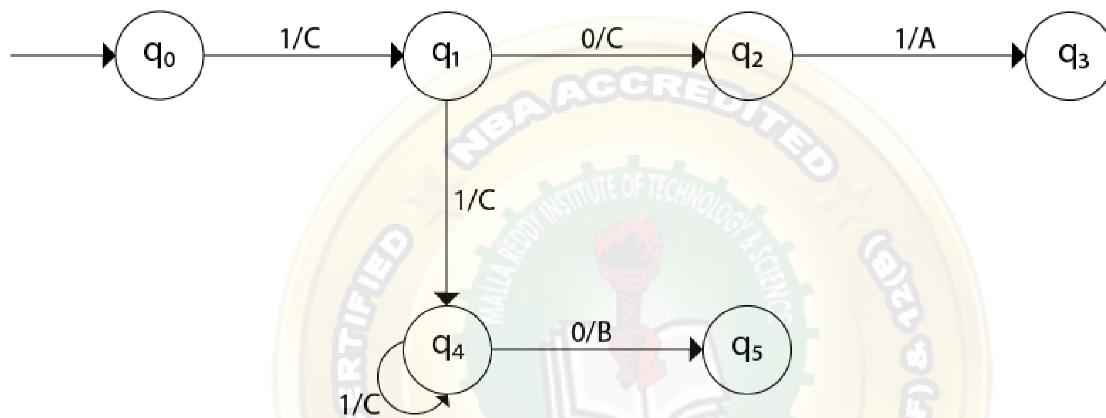
λ': output function where Q × ∑ →O

Example 1:

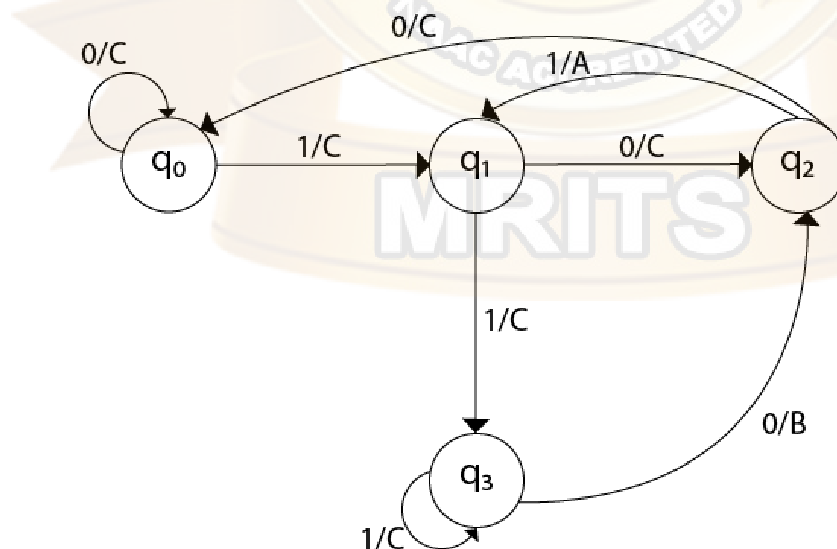
Design a Mealy machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

Solution: For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A. If we recognize 110, the output will be B. For other strings the output will be C.

The partial diagram will be:



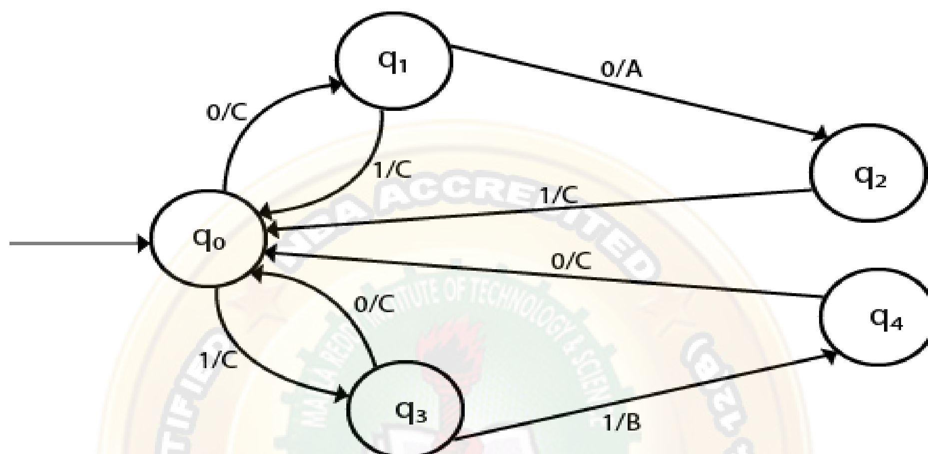
Now we will insert the possibilities of 0's and 1's for each state. Thus the Mealy machine becomes:



Example 2:

Design a mealy machine that scans sequence of input of 0 and 1 and generates output 'A' if the input string terminates in 00, output 'B' if the string terminates in 11, and output 'C' otherwise.

Solution: The mealy machine will be:



Conversion from Mealy machine to Moore Machine

In Moore machine, the output is associated with every state, and in Mealy machine, the output is given along the edge with input symbol. To convert Moore machine to Mealy machine, state output symbols are distributed to input symbol paths. But while converting the Mealy machine to Moore machine, we will create a separate state for every new output symbol and according to incoming and outgoing edges are distributed.

The following steps are used for converting Mealy machine to the Moore machine:

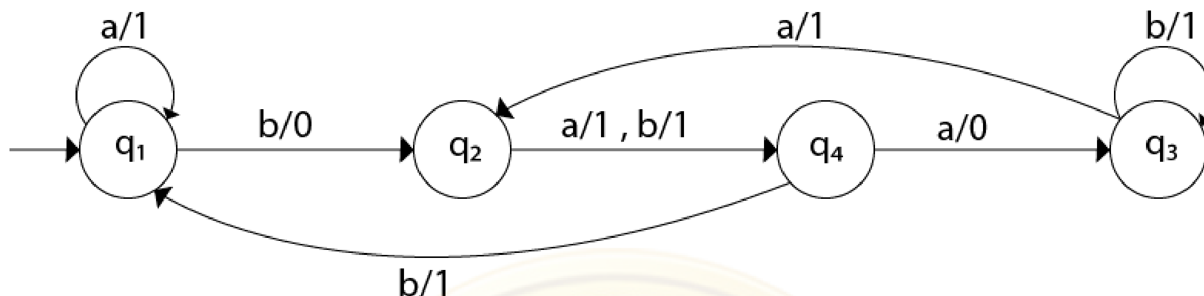
Step 1: For each state(Qi), calculate the number of different outputs that are available in the transition table of the Mealy machine.

Step 2: Copy state Qi, if all the outputs of Qi are the same. Break qi into n states as Qin, if it has n distinct outputs where n = 0, 1, 2....

Step 3: If the output of initial state is 0, insert a new initial state at the starting which gives 1 output.

Example 1:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

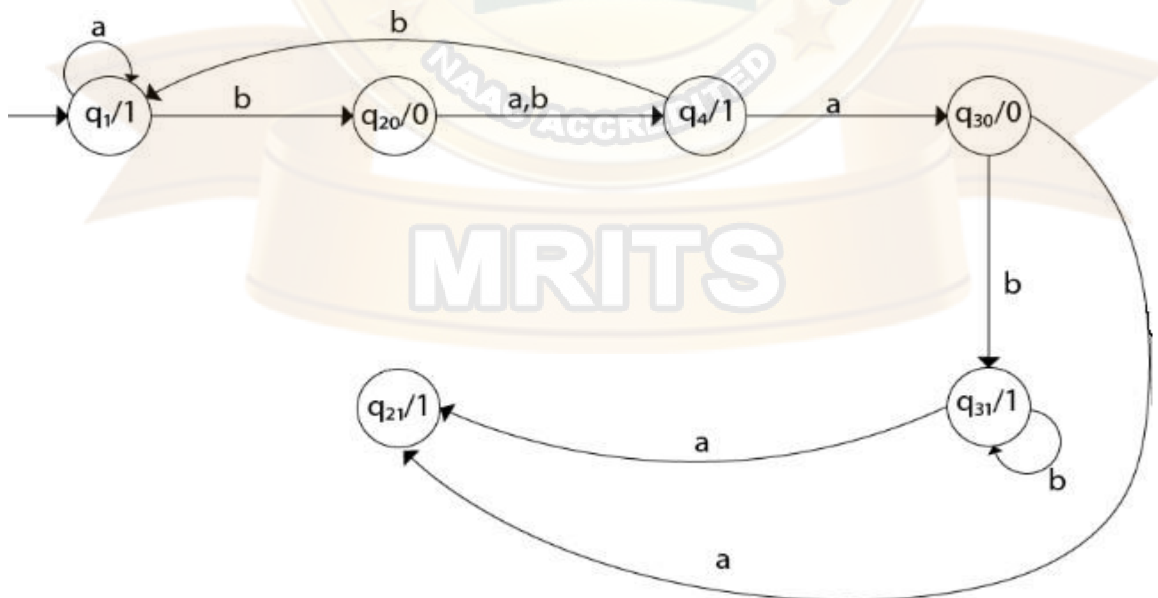
Present State	Next State			
	a		b	
	State	O/P	State	O/P
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

- For state q1, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.
- For state q2, there is 2 incident edge with output 0 and 1. So, we will split this state into two states q20(state with output 0) and q21(with output 1).
- For state q3, there is 2 incident edge with output 0 and 1. So, we will split this state into two states q30(state with output 0) and q31(state with output 1).
- For state q4, there is only one incident edge with output 0. So, we don't need to split this state in Moore machine.

Transition table for Moore machine will be:

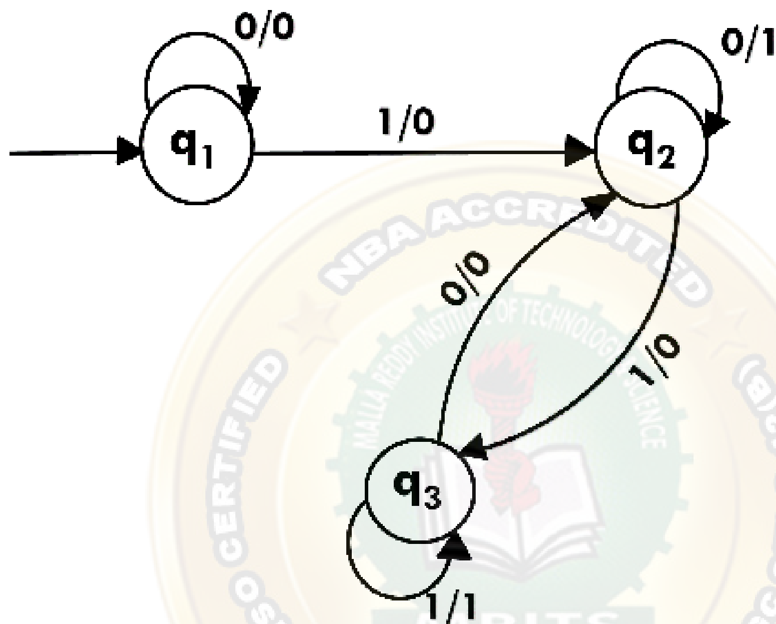
Present State	Next State		Output
	a=0	a=1	
q_1	q_1	q_2	1
q_{20}	q_4	q_4	0
q_{21}	∅	∅	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_4	1

Transition diagram for Moore machine will be:



Example 2:

Convert the following Mealy machine into equivalent Moore machine.



Solution:

Transition table for above Mealy machine is as follows:

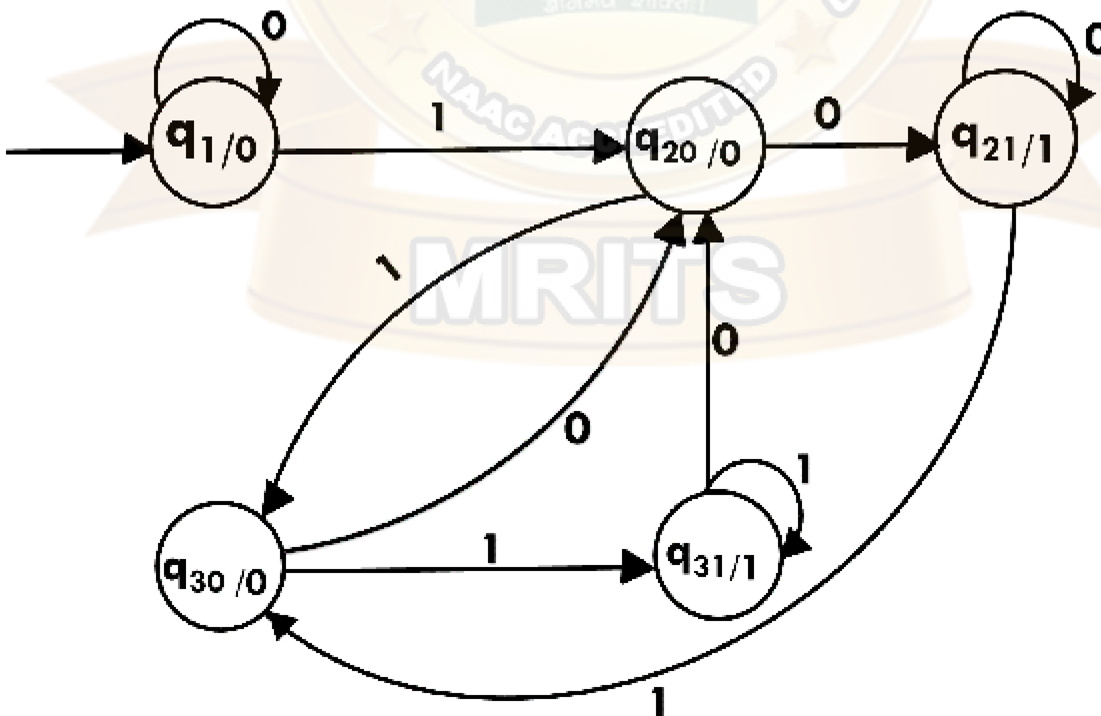
Present State	Next State 0		Next State 1	
	State	O/P	State	O/P
q_1	q_1	0	q_2	0
q_2	q_2	1	q_3	0
q_3	q_2	0	q_3	1

The state q1 has only one output. The state q2 and q3 have both output 0 and 1. So we will create two states for these states. For q2, two states will be q20(with output 0) and q21(with output 1). Similarly, for q3 two states will be q30(with output 0) and q31(with output 1).

Transition table for Moore machine will be:

Present State	Next State 0	Next State 1	0/P
q_1	q_1	q_{20}	0
q_{20}	q_{21}	q_{30}	0
q_{21}	q_{21}	q_{30}	1
q_{30}	q_{20}	q_{31}	0
q_{31}	q_{20}	q_{31}	1

Transition diagram for Moore machine will be:



Conversion from Moore machine to Mealy Machine

In the Moore machine, the output is associated with every state, and in the mealy machine, the output is given along the edge with input symbol. The equivalence of the Moore machine and Mealy machine means both the machines generate the same output string for same input string.

We cannot directly convert Moore machine to its equivalent Mealy machine because the length of the Moore machine is one longer than the Mealy machine for the given input. To convert Moore machine to Mealy machine, state output symbols are distributed into input symbol paths. We are going to use the following method to convert the Moore machine to Mealy machine.

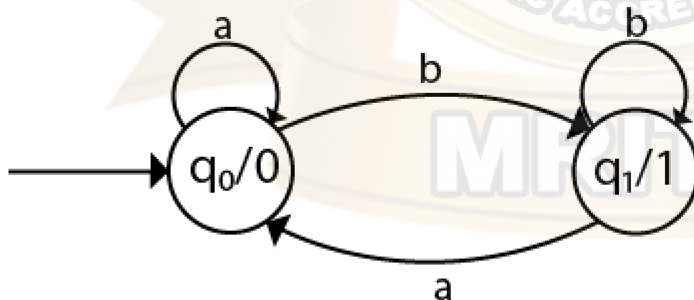
Method for conversion of Moore machine to Mealy machine

Let M = (Q, ∑, δ, λ, q0) be a Moore machine. The equivalent Mealy machine can be represented by M' = (Q, ∑, δ, λ', q0). The output function λ' can be obtained as:

λ' (q, a) = λ(δ(q, a))

Example 1:

Convert the following Moore machine into its equivalent Mealy machine.



Solution:

The transition table of given Moore machine is as follows:

Q	A	b	Output(λ)
q0	q0	q1	0
q1	q0	q1	1

The equivalent Mealy machine can be obtained as follows:

λ' (q0, a) = λ(δ(q0, a))
= λ(q0)
= 0

λ' (q0, b) = λ(δ(q0, b))
= λ(q1)
= 1

The λ for state q1 is as follows:

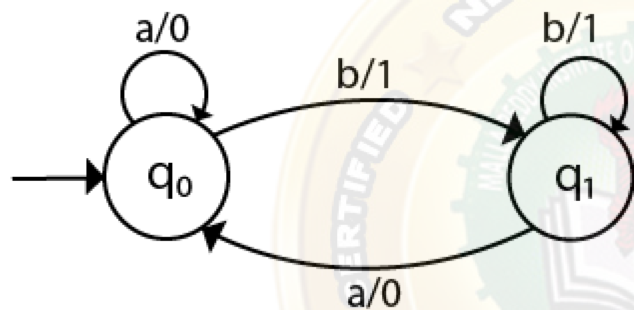
λ' (q1, a) = λ(δ(q1, a))
= λ(q0)
= 0

λ' (q1, b) = λ(δ(q1, b))
= λ(q1)
= 1

Hence the transition table for the Mealy machine can be drawn as follows:

Q \ Σ	Input 0		Input 1	
	State	O/P	State	O/P
q_0	q_0	0	q_1	1
q_1	q_0	0	q_1	1

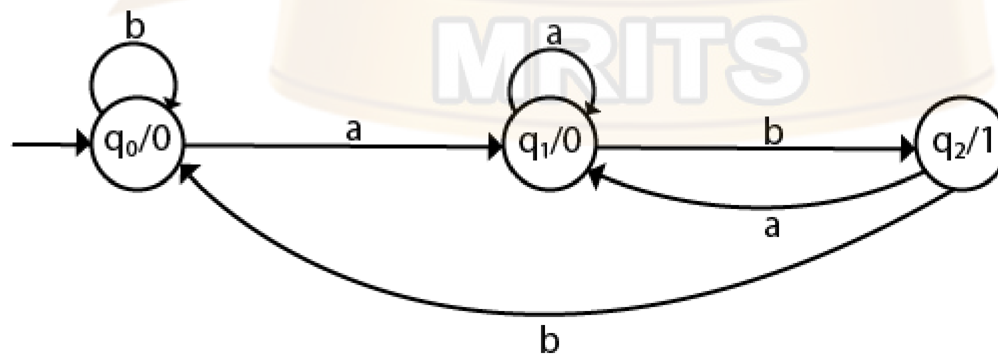
The equivalent Mealy machine will be,



Note: The length of output sequence is 'n+1' in Moore machine and is 'n' in the Mealy machine.

Example 2:

Convert the given Moore machine into its equivalent Mealy machine.



Solution:

The transition table of given Moore machine is as follows:

Q	a	b	Output(λ)
q0	q1	q0	0
q1	q1	q2	0
q2	q1	q0	1

The equivalent Mealy machine can be obtained as follows:

λ' (q0, a) = λ(δ(q0, a))
= λ(q1)
= 0

λ' (q0, b) = λ(δ(q0, b))
= λ(q0)
= 0

The λ for state q1 is as follows:

λ' (q1, a) = λ(δ(q1, a))
= λ(q1)
= 0

λ' (q1, b) = λ(δ(q1, b))
= λ(q2)
= 1

The λ for state q2 is as follows:

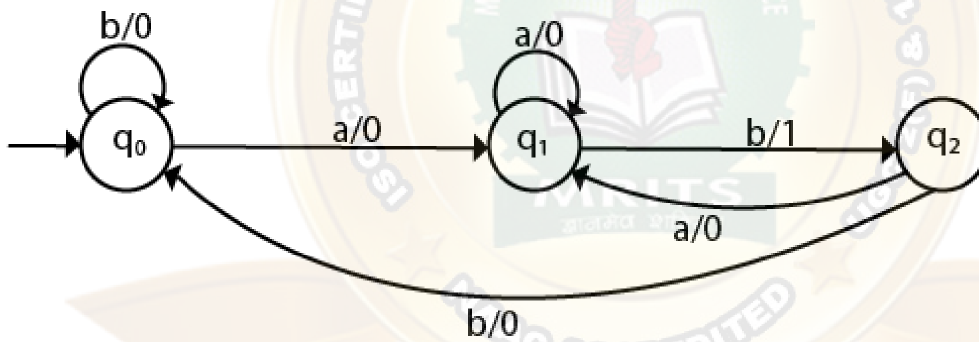
λ' (q2, a) = λ(δ(q2, a))
= λ(q1)
= 0

λ' (q2, b) = λ(δ(q2, b))
= λ(q0)
= 0

Hence the transition table for the Mealy machine can be drawn as follows:

Q \ Σ	Input a		Input b	
	State	Output	State	Output
q_0	q_1	0	q_0	0
q_1	q_1	0	q_2	1
q_2	q_1	0	q_0	0

The equivalent Mealy machine will be,



MALL REDDY INSTITUTE OF TECHNOLOGY & SCIENCE AND SCIENCE

LECTURE NOTES

On

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY (UNIT-II)

III Year B.Tech. CSE/IT I-Sem

(Jntuh-R18)

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE I-Sem **L T P C**

3 0 0

Course Objectives

1. To provide introduction to some of the central ideas of theoretical computer science from the perspective of formal languages.
2. To introduce the fundamental concepts of formal languages, grammars and automata theory.
3. Classify machines by their power to recognize languages.
4. Employ finite state machines to solve problems in computing.
5. To understand deterministic and non-deterministic machines.
6. To understand the differences between decidability and undecidability.

Course Outcomes

1. Able to understand the concept of abstract machines and their power to recognize the languages.
2. Able to employ finite state machines for modeling and solving computing problems.
3. Able to design context free grammars for formal languages.
4. Able to distinguish between decidability and undecidability.

UNIT - I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory – Alphabets, Strings, Languages, Problems.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines

UNIT - II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages, Statement of the pumping lemma, Applications of the Pumping Lemma.

Closure Properties of Regular Languages: Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

UNIT - III

Context-Free Grammars: Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages.

Push Down Automata: Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG

UNIT - IV

Normal Forms for Context- Free Grammars: Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form Griebech Normal form.

Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications

Closure Properties of Context-Free Languages: Closure properties of CFL's, Decision

Properties of CFL's

Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

UNIT - V

Types of Turing machine: Turing machines and halting

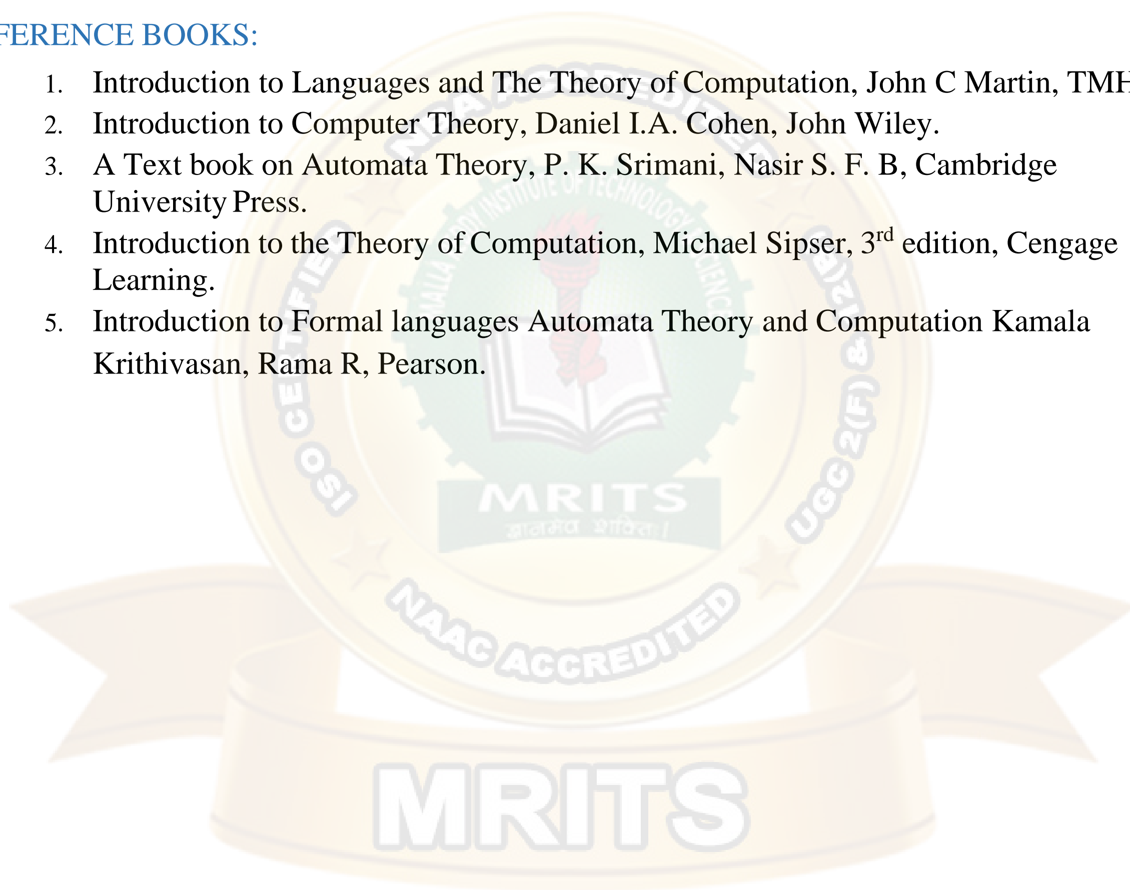
Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem, Other Undecidable Problems, Counter machines.

TEXT BOOKS:

1. Introduction to Automata Theory, Languages, and Computation, 3rd Edition, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science – Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCE BOOKS:

1. Introduction to Languages and The Theory of Computation, John C Martin, TMH.
2. Introduction to Computer Theory, Daniel I.A. Cohen, John Wiley.
3. A Text book on Automata Theory, P. K. Srimani, Nasir S. F. B, Cambridge University Press.
4. Introduction to the Theory of Computation, Michael Sipser, 3rd edition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and Computation Kamala Krithivasan, Rama R, Pearson.



UNIT - II

CONTENTS

Regular Expressions:

Finite Automata and Regular Expressions

Applications of Regular Expressions

Algebraic Laws for Regular Expressions,\

Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages,

Statement of the pumping lemma,

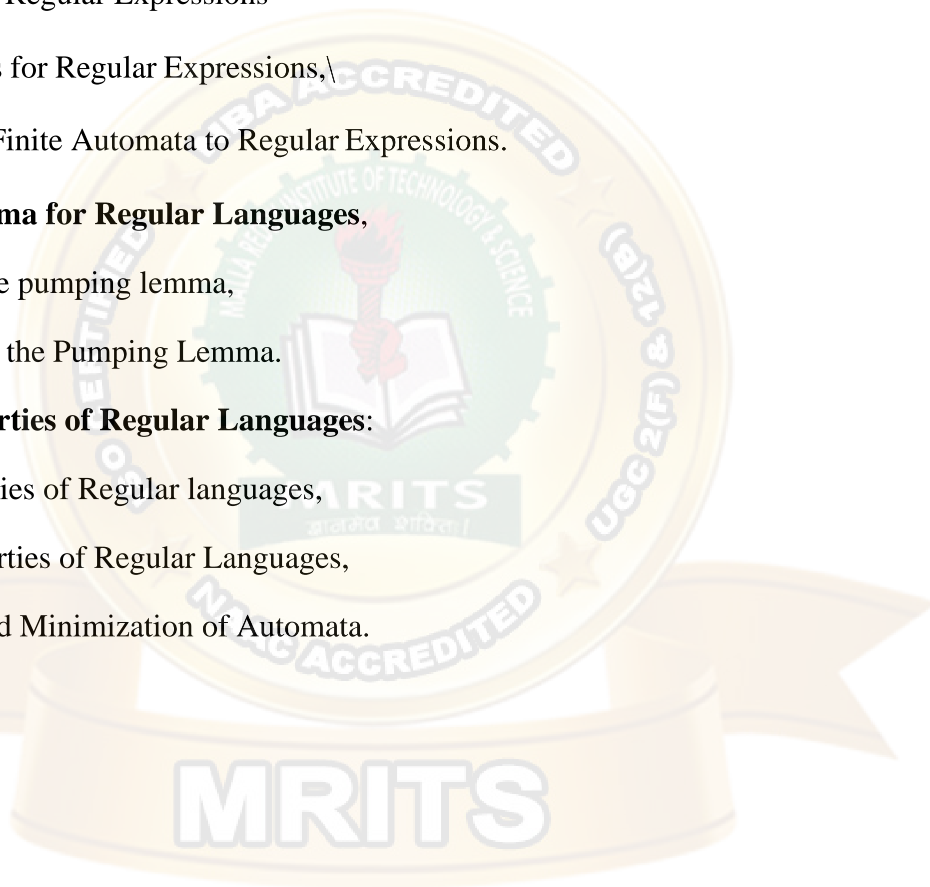
Applications of the Pumping Lemma.

Closure Properties of Regular Languages:

Closure properties of Regular languages,

Decision Properties of Regular Languages,

Equivalence and Minimization of Automata.



Regular Expressions:

Just as finite automata are used to recognize patterns of strings, regular expressions are used to generate patterns of strings. A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of the expression.

1.A Regular Language is used to specify a Language and it does so preciously
2. Regular expressions are very intuitive.
3.Regular expressions are very useful in a variety of contexts.
4.Given a regular expression, an NFA-ε can be constructed from it automatically.
5.Thus, so can an NFA, a DFA, and a corresponding program, all automatically!

Operands in a regular expression can be:

- characters from the alphabet over which the regular expression is defined.
- variables whose values are any pattern defined by a regular expression.
- epsilon which denotes the empty string containing no characters.
- null which denotes the empty set of strings.

- Let Σ be an alphabet. The regular expressions over Σ are:
  - Ø Represents the empty set { }
  - ε Represents the set {ε}
  - a Represents the set {a}, for any symbol a in ΣOperators used in regular expressions include:

- Union: If R1 and R2 are regular expressions, then R1 | R2 (also written as R1 U R2 or R1 + R2) is also a regular expression.

  L(R1|R2) = L(R1) U L(R2).

- Concatenation: If R1 and R2 are regular expressions, then R1R2 (also written as R1.R2) is also a regular expression.

  L(R1R2) = L(R1) concatenated with L(R2).

- Kleene closure: If R1 is a regular expression, then R1* (the Kleene closure of R1) is also a regular expression.

  L(R1*) = epsilon U L(R1) U L(R1R1) U L(R1R1R1) U ...

Closure has the highest precedence, followed by concatenation, followed by union.

Let r and s be regular expressions that represent the sets R and S, respectively.

- r+s Represents the set R U S (precedence 3)
- rs Represents the set RS (precedence 2)
- r^* Represents the set R* (highest precedence)
- (r) Represents the set R (not an op, provides precedence)

If r is a regular expression, then L(r) is used to denote the corresponding language.

Examples: Let Σ = {0, 1}

(0 + 1)*	All strings of 0's and 1's
0(0 + 1)*	All strings of 0's and 1's, beginning with a 0
(0 + 1)*1	All strings of 0's and 1's, ending with a 1
(0 + 1)*0(0 + 1)*	All strings of 0's and 1's containing at least one 0
(0 + 1)*0(0 + 1)*0(0 + 1)*	All strings of 0's and 1's containing at least two 0's
(0 + 1)*01*01*	All strings of 0's and 1's containing at least two 0's
(1 + 01*0)*	All strings of 0's and 1's containing an even number of 0's
1*(01*01*)*	All strings of 0's and 1's containing an even number of 0's
(1*01*0)*1*	All strings of 0's and 1's containing an even number of 0's

Identities:

1. Øu = uØ = Ø Multiply by 0

2. εu = uε = u Multiply by 1

3. Ø* = ε

4. ε* = ε

5. u+v = v+u

6. u + Ø = u

6. u + u = u 8.

7. u* = (u*)*

9. u(v+w) = uv+uw

10. (u+v)w = uw+vw

11. (uv)*u = u(vu)

12. (u+v)* = (u*+v)*

= u*(u+v)*

= (u+vu*)*

= (u*v*)*

= u*(vu*)*

= (u*v)*u*

Finite Automata and Regular Expressions

If L=L(A) for some DFA, then there is a regular expression R such that L=L(R)
We are going to construct regular expressions from a DFA by eliminating states.
When we eliminate a state s, all the paths that went through s no longer exist in the automaton.
If the language of the automaton is not to change, we must include, on an arc that goes directly from q to p ,
the labels of paths that went from some state q to state p , through s.
The label of this arc can now involve strings, rather than single symbols (may be an infinite number of strings).
We use a regular expression to represent all such strings.
Thus, we consider automata that have regular expressions as labels.

Conversion of Finite Automata to Regular Expressions.

Regular expressions and finite automata have equivalent expressive power:

- For every regular expression R, there is a corresponding FA that accepts the set of strings generated by R.
- For every FA A there is a corresponding regular expression that generates the set of strings accepted by A.

The proof is in two parts:

1. an algorithm that, given a regular expression R, produces an FA A such that L(A) == L(R).
2. an algorithm that, given an FA A, produces a regular expression R such that L(R) == L(A).

Our construction of FA from regular expressions will allow "epsilon transitions" (a transition from one state to another with epsilon as the label). Such a transition is always possible, since epsilon (or the empty string) can be said to exist between any two input symbols. We can show that such epsilon transitions are a notational convenience; for every FA with epsilon transitions there is a corresponding FA without them.

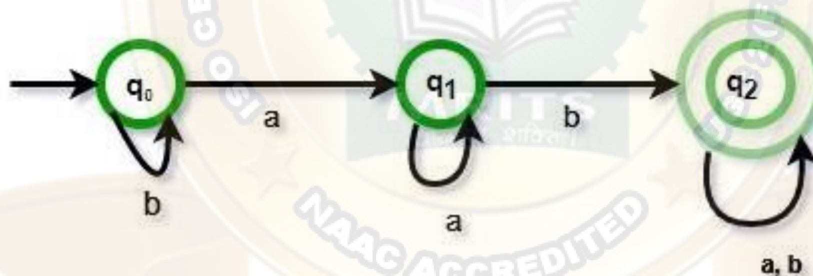
Designing Finite Automata from Regular Expression

In this article, we will see some popular regular expressions and how we can convert them to finite automata.

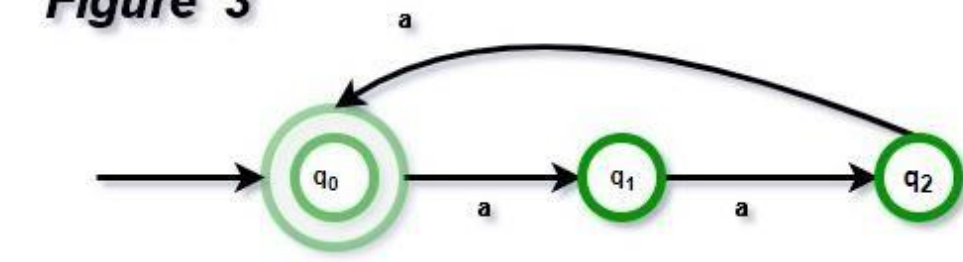
- **Even number of a's :** The regular expression for even number of a's is **(b|ab*ab*)***. We can construct a finite automata as shown in Figure 1.

  The above automata will accept all strings which have even number of a's. For zero a's, it will be in q0 which is final state. For one 'a', it will go from q0 to q1 and the string will not be accepted. For two a's at any positions, it will go from q0 to q1 for 1st 'a' and q1 to q0 for second 'a'. So, it will accept all strings with even number of a's.

- **String with 'ab' as substring :** The regular expression for strings with 'ab' as substring is **(a|b)*ab(a|b)***. We can construct finite automata as shown in Figure 2.



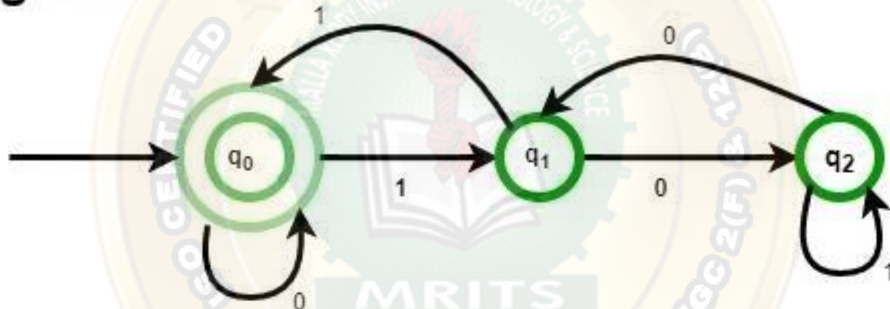
- The above automata will accept all string which have 'ab' as substring. The automata will remain in initial state q0 for b's. It will move to q1 after reading 'a' and remain in same state for all 'a' afterwards. Then it will move to q2 if 'b' is read. That means, the string has read 'ab' as substring if it reaches q2.
- **String with count of 'a' divisible by 3 :** The regular expression for strings with count of a divisible by 3 is $\{a^{3n} \mid n >= 0\}$. We can construct automata as shown in Figure 3.



The above automata will accept all string of form a^{3n}. The automata will remain in initial state q0 for ε and it will be accepted. For string 'aaa', it will move from q0 to q1 then q1 to q2 and then q2 to q0. For every set of three a's, it will come to q0, hence accepted. Otherwise, it will be in q1 or q2, hence rejected.

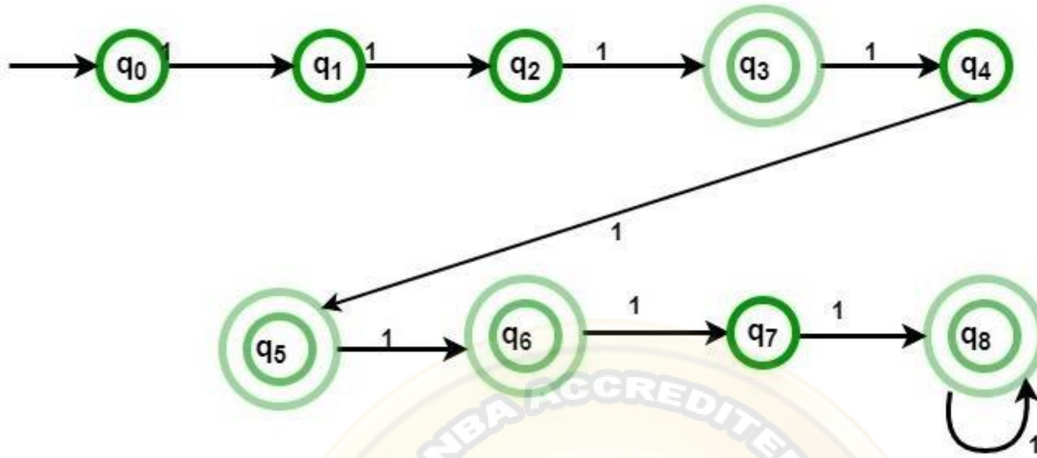
Note : If we want to design a finite automata with number of a's as 3n+1, same automata can be used with final state as q1 instead of q0.
If we want to design a finite automata with language $\{a^{kn} \mid n >= 0\}$, k states are required. We have used k = 3 in our example.

- **Binary numbers divisible by 3 :** The regular expression for binary numbers which are divisible by three is **(0|1(01*0)*1)***. The examples of binary number divisible by 3 are 0, 011, 110, 1001, 1100, 1111, 10010 etc. The DFA corresponding to binary number divisible by 3 can be shown in Figure 4.



The above automata will accept all binary numbers divisible by 3. For 1001, the automata will go from q0 to q1, then q1 to q2, then q2 to q1 and finally q2 to q0, hence accepted. For 0111, the automata will go from q0 to q0, then q0 to q1, then q1 to q0 and finally q0 to q1, hence rejected.

- **String with regular expression (111 + 11111)* :** The string accepted using this regular expression will have 3, 5, 6(111 twice), 8 (11111 once and 111 once), 9 (111 thrice), 10 (11111 twice) and all other counts of 1 afterwards. The DFA corresponding to given regular expression is given in Figure 5.



Equivalence of Regular Expressions and NFA-ε

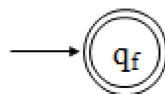
- Note: Throughout the following, keep in mind that a string is accepted by an NFA-ε if there exists a path from the start state to a final state.
- Lemma 1: Let r be a regular expression. Then there exists an NFA-ε M such that L(M) = L(r). Furthermore, M has exactly one final state with no transitions out of it.
- Proof: (by induction on the number of operators, denoted by OP(r), in r).

- Basis: OP(r) = 0

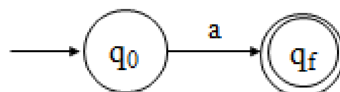
Then r is either Ø, ε, or a, for some symbol a in Σ

For Ø:

For ε:



For **a**:



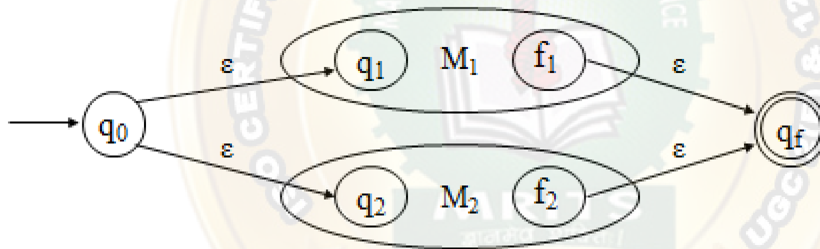
- Inductive Hypothesis: Suppose there exists a k□0 such that for any regular expression r where 0 □OP(r) k□, there exists an NFA-ε such that L(M) = L(r). Furthermore, suppose that M has exactly one final state.

- Inductive Step: Let r be a regular expression with k + 1 operators (OP(r) = k + 1), where k + 1 >= 1.

Case 1) r = r1 + r2

Since OP(r) = k +1, it follows that 0<= OP(r1), OP(r2) <= k. By the inductive hypothesis there exist NFA-ε machines M1 and M2 such that L(M1) = L(r1) and L(M2) = L(r2). Furthermore, both M1 and M2 have exactly one final state.

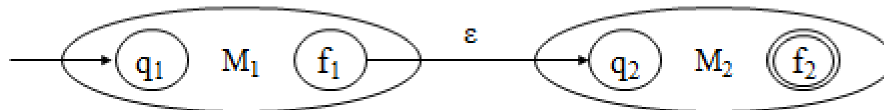
Construct M as:



Case 2) r = r1r2

Since OP(r) = k+1, it follows that 0<= OP(r1), OP(r2) <= k. By the inductive hypothesis there exist NFA-ε machines M1 and M2 such that L(M1) = L(r1) and L(M2) = L(r2). Furthermore, both M1 and M2 have exactly one final state.

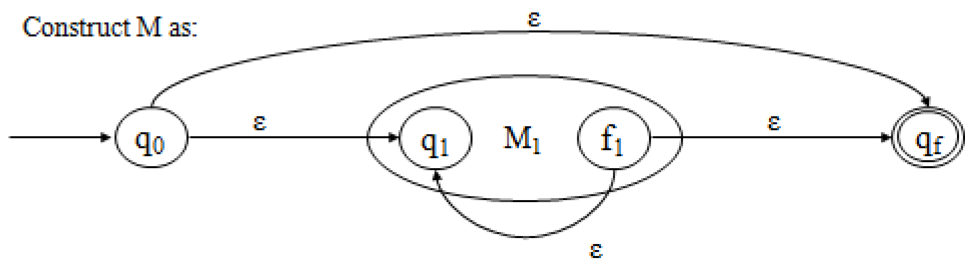
Construct M as:



Case 3) r = r1*

Since OP(r) = k+1, it follows that 0<= OP(r1) <= k. By the inductive hypothesis there exists an NFA-ε machine M1 such that L(M1) = L(r1). Furthermore, M1 has exactly one final state.

NAAC ACCREDITED
MRITS



Examples:

Problem: Construct FA equivalent to RE, r = 0(0+1)*

Solution: r = r1r2 r1 = 0

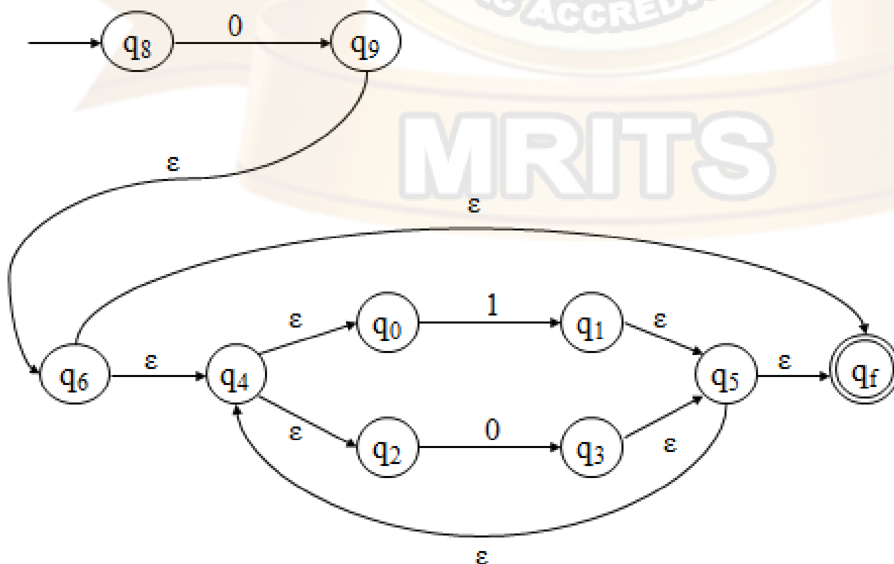
r2 = (0+1)*

r2 = r3* r3 = 0+1

r3 = r4 + r5 r4 = 0

r5 = 1

Transition graph:

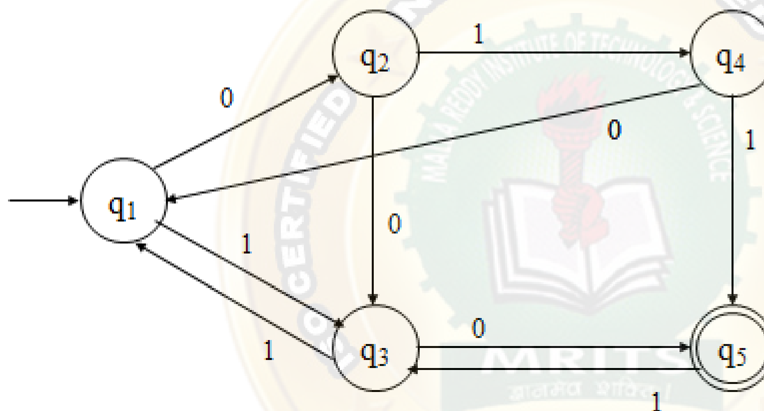


Definitions Required to Convert a DFA to a Regular Expression

1.Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA with state set $Q = \{q_1, q_2, \ldots, q_n\}$,
and define: $R_{i,j} = \{ x \mid x \text{ is in } \Sigma^* \text{ and } \delta(q_i,x) = q_j$
$R_{i,j}$ is the set of all strings that define a path in M from q_i to q_j.

2. Note that states have been numbered starting at 1!

- Example:



$R_{2,3} = \{0, 001, 00101, 011, \ldots\}$
$R_{1,4} = \{01, 00101, \ldots\}$
$R_{3,3} = \{11, 100, \ldots\}$

Observations;

1) $R^{n}_{i,j} = R_{i,j}$

2) $R^{k-1}_{i,j}$ is a subset of $R^{k}_{i,j}$

3) $L(M) = \bigcup_{q \in F} R^{n}_{1,q} = \bigcup_{q \in F} R_{1,q}$

4) $R^{0}_{i,j} = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & i \neq j \\ \{a \mid \delta(q_i, a) = q_j\} \bigcup \{\varepsilon\} & i = j \end{cases}$ Easily computed from the DFA!

5) $R^{k}_{i,j} = R^{k-1}_{i,k} (R^{k-1}_{k,k})^{*} R^{k-1}_{k,j} \cup R^{k-1}_{i,j}$

Lemma 2: Let M = (Q, Σ, δ, q1, F) be a DFA. Then there exists a regular expression r such that L(M) = L(r).

Proof:

First we will show (by induction on k) that for all i,j, and k, where 1 □ i,j □ n And 0 □ k □ n, that there exists a regular expression r such that L(r) = Rki,j .

Basis: k=0

R0i,j contains single symbols, one for each transition from qi to qj, and possibly ε if i=j.

Case 1) No transitions from qi to qj and i != j r0i,j = Ø

$$r^{0}_{i,j} = \varnothing$$

Case 2) At least one (m ≥ 1) transition from q_i to q_j and i != jCase 2) At least one (m ≥ 1) transition from q_i to q_j and i != j

$$r^{0}_{i,j} = a_1 + a_2 + a_3 + \ldots + a_m \qquad \text{where } \delta(q_i, a_p) = q_j,$$

for all $1 \le p \le m$

Case 3) No transitions from q_i to q_j and i = j

$$r^{0}_{i,j} = \varepsilon$$

Case 4) At least one (m ≥ 1) transition from q_i to q_j and i = j

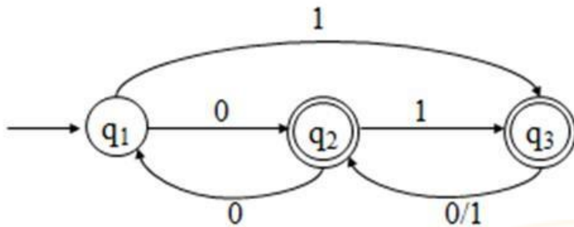
$$r^{0}_{i,j} = a_1 + a_2 + a_3 + \ldots + a_m + \varepsilon \qquad \text{where } \delta(q_i, a_p) = q_j$$

for all $1 \le p \le m$

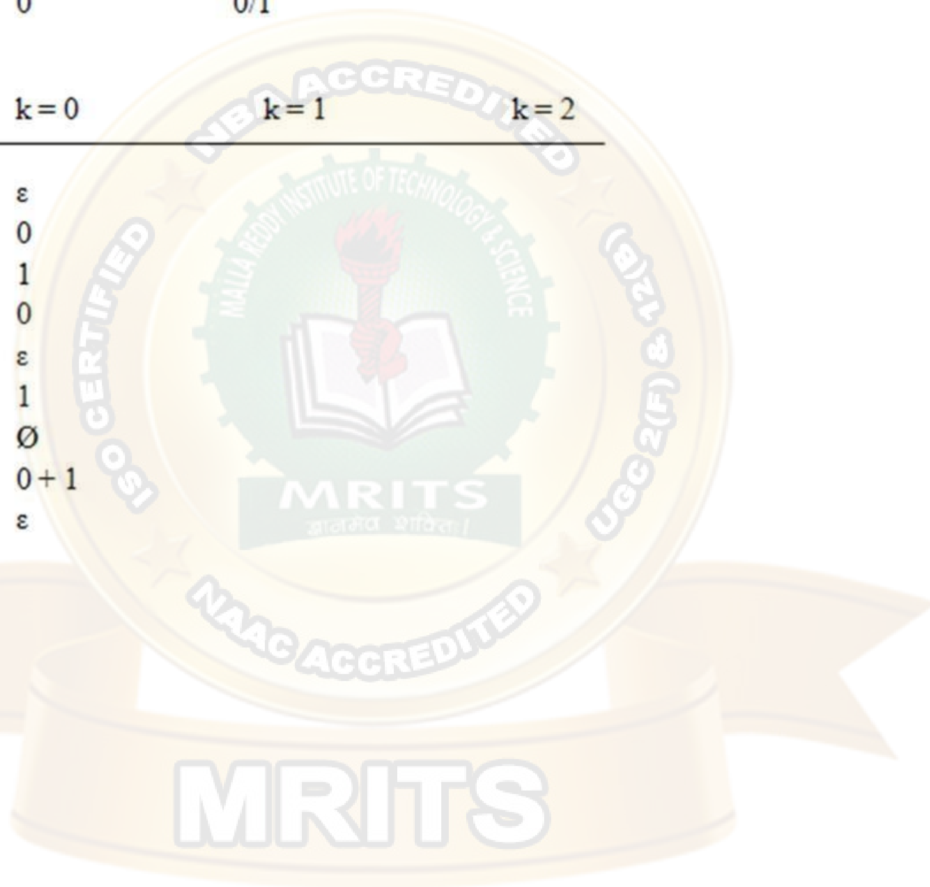
Inductive Hypothesis:

Suppose that $R^{k-1}_{i,j}$ can be represented by the regular expression $r^{k-1}_{i,j}$ for all
$1 \le i,j \le n$, and some $k \ge 1$.

- **Example:**



First table column is computed from the DFA.



	k = 0	k = 1	k = 2
$r^k_{1,1}$	ε		
$r^k_{1,2}$	0		
$r^k_{1,3}$	1		
$r^k_{2,1}$	0		
$r^k_{2,2}$	ε		
$r^k_{2,3}$	1		
$r^k_{3,1}$	Ø		
$r^k_{3,2}$	0 + 1		
$r^k_{3,3}$	ε		

- All remaining columns are computed from the previous column using the formula.

$r^1_{2,3} = r^0_{2,1} (r^0_{1,1})^* r^0_{1,3} + r^0_{2,3}$
$= 0 (\varepsilon)^* 1 + 1$
$= 01 + 1$

	k = 0	k = 1	k = 2
$r^k_{1,1}$	ε	ε	
$r^k_{1,2}$	0	0	
$r^k_{1,3}$	1	1	
$r^k_{2,1}$	0	0	
$r^k_{2,2}$	ε	ε + 00	
$r^k_{2,3}$	1	**1 + 01**	
$r^k_{3,1}$	Ø	Ø	
$r^k_{3,2}$	0 + 1	0 + 1	
$r^k_{3,3}$	ε	ε	

$r^2_{1,3} = r^1_{1,2} (r^1_{2,2})^* r^1_{2,3} + r^1_{1,3}$
$= 0 (\varepsilon + 00)^* (1 + 01) + 1$
$= 0^*1$

	k = 0	k = 1	k = 2
$r^k_{1,1}$	ε	ε	(00)*
$r^k_{1,2}$	0	0	0(00)*
$r^k_{1,3}$	1	1	**0*1**
$r^k_{2,1}$	0	0	0(00)*
$r^k_{2,2}$	ε	ε + 00	(00)*
$r^k_{2,3}$	1	1 + 01	0*1
$r^k_{3,1}$	Ø	Ø	(0 + 1)(00)*0
$r^k_{3,2}$	0 + 1	0 + 1	(0 + 1)(00)*
$r^k_{3,3}$	ε	ε	ε + (0 + 1)0*1

Applications of Regular Expressions

Applications:

- Regular expressions are useful in a wide variety of text processing tasks, and more generally string processing, where the data need not be textual. Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems, and many other tasks.
- While regexps would be useful on Internet search engines, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex.

Algebraic Laws for Regular Expressions

Definition :Two regular expressions with variables are equivalent if whatever languageswe substitute for the variables, the results of the two expressions are the same language.

$$L^+ \stackrel{\text{def}}{=} LL^*,$$

$$L? \stackrel{\text{def}}{=} \varepsilon + L.$$

Some laws for union

$$(L+M)+N = L+(M+N) \quad \text{(associativity)}$$
$$L+\emptyset = \emptyset+L = L \quad \text{(identity)}$$
$$L+M = M+L \quad \text{(commutativity)}$$
$$L+L = L \quad \text{(idempotence)}$$

Remark

There is no inverse for union.

Some laws for concatenation

$$(LM)N = L(MN) \quad \text{(associativity)}$$
$$L\varepsilon = \varepsilon L = L \quad \text{(identity)}$$
$$LM \neq ML \quad \text{(non-commutativity)}$$
$$L\emptyset = \emptyset L = \emptyset \quad \text{(}\emptyset\text{ is the annihilator for concatenation)}$$

Remark

There is no inverse for concatenation.

Some laws for union and concatenation

L(M+N)=LM+LN (distributive)

(L+M)N=LN+MN (distributive)

Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages:

In the theory of formal languages, the **pumping lemma for regular languages** is a lemma that describes an essential property of all regular languages. Informally, it says that all sufficiently long words in a regular language may be *pumped*—that is, have a middle section of the word repeated an arbitrary number of times—to produce a new word that also lies within the same language.

Specifically, the pumping lemma says that for any regular language L there exists a constant p such that any word w in L with length at least p can be split into three substrings, $w = xyz$, where the middle portion y must not be empty, such that the words xz, xyz, $xyyz$, $xyyyz$, … constructed by repeating y zero or more times are still in L. This process of repetition is known as "pumping". Moreover, the pumping lemma guarantees that the length of xy will be at most p, imposing a limit on the ways in which w may be split. Finite languages vacuously satisfy the pumping lemma by having p equal to the maximum string length in L plus one.

The pumping lemma is useful for disproving the regularity of a specific language in question. It was first proven by Michael Rabin and Dana Scott in 1959,[1] and rediscovered shortly after by Yehoshua Bar-Hillel, Micha A. Perles, and Eli Shamir in 1961, as a simplification of their pumping lemma for context-free languages.

Statement of the pumping lemma

Let L be a regular language. Then there exists an integer $p \geq 1$ depending only on L such that every string w in L of length at least p (p is called the "pumping length"[4]) can be written as $w = xyz$ (i.e., w can be divided into three substrings), satisfying the following conditions:

1. $|y| \geq 1$,
2. $|xy| \leq p$, and
3. $xy^nz \in L$ for all $n \geq 0$.

y is the substring that can be pumped (removed or repeated any number of times, and the resulting string is always in L). (1) means the loop y to be pumped must be of length at least one; (2) means the loop must occur within the first p characters. $|x|$ must be smaller than p (conclusion of (1) and (2)), but apart from that, there is no restriction on x and z.

In simple words, for any regular language L, any sufficiently long word w (in L) can be split into 3 parts. i.e. $w = xyz$, such that all the strings xy^nz for $n \geq 0$ are also in L.

Below is a formal expression of the Pumping Lemma

Use of the lemma

The pumping lemma is often used to prove that a particular language is non-regular: a proof by contradiction (of the language's regularity) may consist of exhibiting a word (of the required length) in the language that lacks the property outlined in the pumping lemma.

For example, the language $L = \{a^n b^n : n \geq 0\}$ over the alphabet $\Sigma = \{a, b\}$ can be shown to be non-regular as follows. Let w, x, y, z, p, and i be as used in the formal statement for the pumping lemma above. Let w in L be given by $w = a^p b^p$. By the pumping lemma, there must be some decomposition $w = xyz$ with $|xy| \leq p$ and $|y| \geq 1$ such that xy^iz in L for every $i \geq 0$. Using $|xy| \leq p$, we know y only consists of instances of a. Moreover, because $|y| \geq 1$, it contains at least one instance of the letter a. We now pump y up: xy^2z has more instances of the letter a than the letter b, since we have added some instances of a without adding instances of b. Therefore, xy^2z is not in L. We have reached a contradiction. Therefore, the assumption that L is regular must be incorrect. Hence L is not regular.

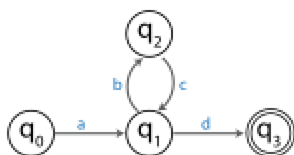
The proof that the language of balanced (i.e., properly nested) parentheses is not regular follows the same idea. Given p, there is a string of balanced parentheses that begins with more than p left parentheses, so that y will consist entirely of left parentheses. By repeating y, we can produce a string that does not contain the same number of left and right parentheses, and so they cannot be balanced.

Proof of the pumping lemma

Proof idea: Whenever a sufficiently long string xyz is recognized by a finite automaton, it must have reached some state (q_s=q_t) twice. Hence, after repeating ("pumping") the middle part y arbitrarily often ($xyyz$, $xyyyz$, ...) the word will still be recognized.

For every regular language there is a finite state automaton (FSA) that accepts the language. The number of states in such an FSA are counted and that count is used as the pumping length p. For a string of length at least p, let q_0 be the start state and let $q_1, ..., q_p$ be the sequence of the next p states visited as the string is emitted. Because the FSA has only p states, within this sequence of $p + 1$ visited states there must be at least one state that is repeated. Write q_s for such a state. The transitions that take the machine from the first encounter of state q_s to the second encounter of state q_s match some string. This string is called y in the lemma, and since the machine will match a string without the y portion, or with the string y repeated any number of times, the conditions of the lemma are satisfied.

For example, the following image shows an FSA.



The FSA accepts the string: **abcd**. Since this string has a length at least as large as the number of states, which is four, the pigeonhole principle indicates that there must be at least one repeated state among the start state and the next four visited states. In this example, only q_1 is a repeated state. Since the substring **bc** takes the machine through transitions that start at state q_1 and end at state q_1, that portion could be repeated and the FSA would still accept, giving the string **abcbcd**. Alternatively, the **bc** portion could be removed and the FSA would still accept giving the string **ad**. In terms of the pumping lemma, the string **abcd** is broken into an x portion **a**, a y portion **bc** and a z portion **d**.

General version of pumping lemma for regular languages

If a language L is regular, then there exists a number $p \geq 1$ (the pumping length) such that every string uwv in L with $|w| \geq p$ can be written in the form

$$uwv = uxyzv$$

with strings x, y and z such that $|xy| \leq p$, $|y| \geq 1$ and

uxy^izv is in L for every integer $i \geq 0$.[5]

From this, the above standard version follows a special case, with both u and v being the empty string.

Since the general version imposes stricter requirements on the language, it can be used to prove the non-regularity of many more languages, such as $\{ a^m b^n c^n : m \geq 1 \text{ and } n \geq 1 \}$.[6]

Converse of lemma not true

While the pumping lemma states that all regular languages satisfy the conditions described above, the converse of this statement is not true: a language that satisfies these conditions may still be non-regular. In other words, both the original and the general version of the pumping lemma give a *necessary* but not *sufficient* condition for a language to be regular.

For example, consider the following language L:

In other words, L contains all strings over the alphabet {0,1,2,3} with a substring of length 3 including a duplicate character, as well as all strings over this alphabet where precisely 1/7 of the string's characters are 3's. This language is not regular but can still be "pumped" with $p = 5$. Suppose some string s has length at least 5. Then, since the alphabet has only four characters, at least two of the first five characters in the string must be duplicates. They are separated by at most three characters.

- If the duplicate characters are separated by 0 characters, or 1, pump one of the other two characters in the string, which will not affect the substring containing the duplicates.

- If the duplicate characters are separated by 2 or 3 characters, pump 2 of the characters separating them. Pumping either down or up results in the creation of a substring of size 3 that contains 2 duplicate characters.
- The second condition of *L* ensures that *L* is not regular: Consider the string . This string is in *L* exactly when and thus *L* is not regular by the Myhill-Nerode theorem.

The Myhill-Nerode theorem provides a test that exactly characterizes regular languages. The typical method for proving that a language is regular is to construct either a finite state machine or a regular expression for the language.

Applications of the Pumping Lemma

Not all languages are regular. For example, the language $L = \{a^n b^n : n \geq 0\}$ is not regular. Similarly, the language $\{a^p : p \text{ is a prime number}\}$ is not regular. A pertinent question therefore is how do we know if a language is not regular.

Question: Can we conclude that a language is not regular if no one could come up with a DFA, NFA, ε-NFA, regular expression or regular grammar so far?
- No. Since, someone may very well come up with any of these in future.

We need a property that just holds for regular languages and so we can prove that any language without that property is not regular. Let's recall some of the properties.

- We have seen that a regular language can be expressed by a finite state automaton. Be it deterministic or non-deterministic, the automaton consists of a finite set of states.

- Since the states are finite, if the automaton has no loop, the language would be finite.

  - Any finite language is indeed a regular language since we can express the language using the regular
  expression: S1 + S2 + ... + SN, where N is the total number of strings accepted by the automaton.

- However, if the automaton has a loop, it is capable of accepting infinite number of strings.

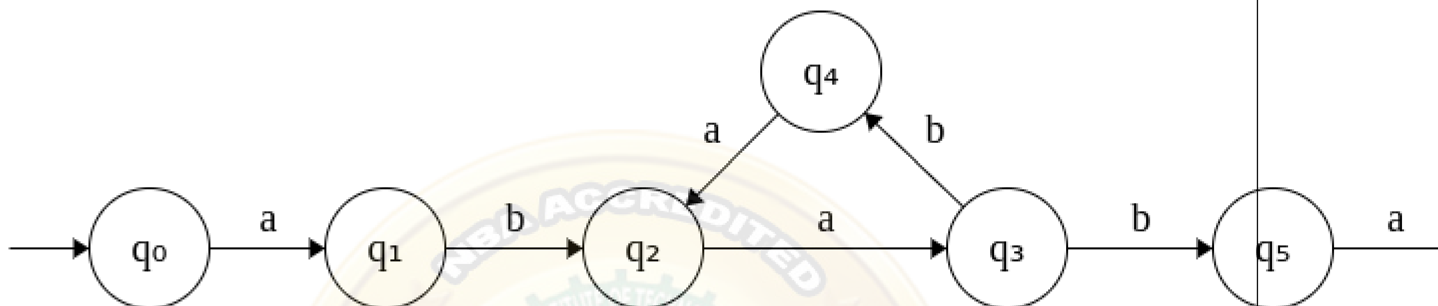
  - Because, we can loop around any number of times and keep producing more and more strings.
  - This property is called the pumping property (elaborated below).

The pumping property of regular languages

Any finite automaton with a loop can be divided into parts three.

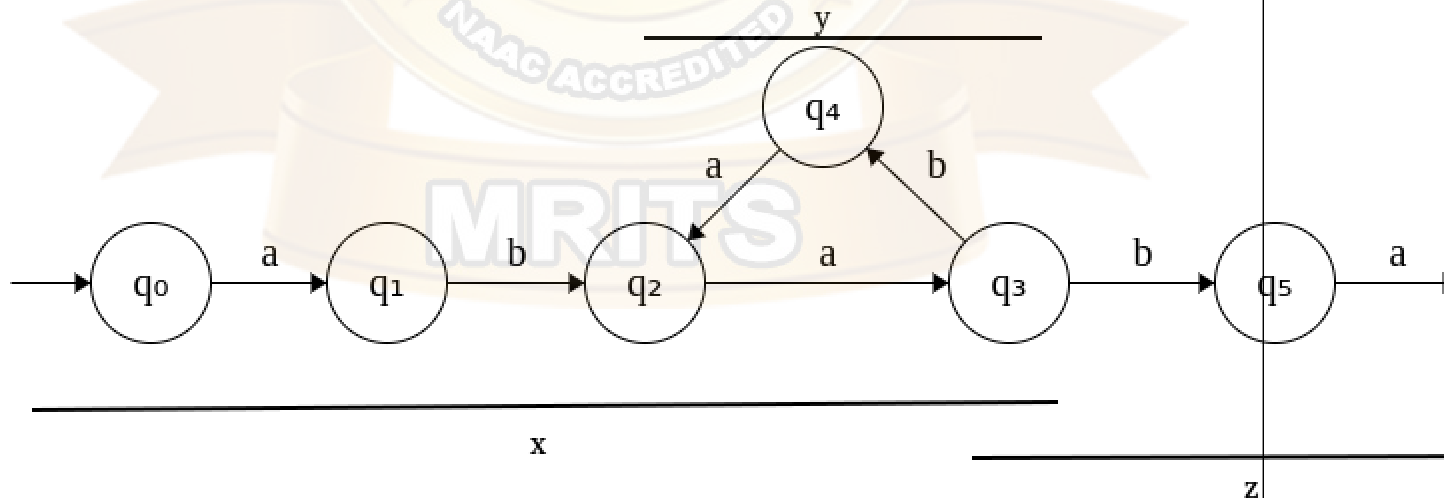
- Part 1: The transitions it takes before the loop.
- Part 2: The transitions it takes during the loop.
- Part 3: The transitions it takes after the loop.

For example consider the following DFA. It accepts all strings that start with aba followed by any number of baa's and finally ending with ba.



1. What strings are accepted by this DFA?
 aba**baa**ba, aba**baabaa**ba, aba**baabaabaa**ba, ...so on and so forth. Thus the strings accepted by the above DFA can be divided into three parts: **aba,** $\mathbf{(baa)^i}$ **and ba**. Here, i > 0.

Investigating this further, we can say that any string w accepted by this DFA can be written as $\mathbf{w = x\,y^i\,z}$
where y represents the part that can be pumped again and again to generate more and more valid strings. This is shown below for the given example.



Before we generalize further, let's investigate this example a little more.

2. What if the loop was at the beginning? Say a self-loop at q_0 instead of at q_2.
 Then x = ε or |x| = 0. In such a special case, w = yz.

3. What is the loop was at the end. Say a self loop at q_6 instead of at q_2.
Then z = ε or |z| = 0. In such a special case, w = xy.

4. Can y be equal to ε ever?
No. It is impossible. If y = ε, it implies there is no loop which implies the language is finite. We have already seen that a finite language is always regular. So, we are now concerned only with infinite regular language. Hence, y can never be ε. Or |y| > 0.

5. What is the shortest string that is accepted by the DFA?
ababa. Obviously, a string obtained without going through the loop.
There is a catch however. See the next question.

6. What is the shortest string accepted if there are more final states? Say q_2 is final.
ab of length 2.

7. What is the longest string accepted by the DFA without going through the loop even once?
ababa (= xz). So, any string of length > 5 accepted by DFA must go through the loop at least once.

8. What is the longest string accepted by the DFA by going through the loop exactly once?
ababaaba (= xyz) of length 8. We call this pumping length.

More precisely, pumping length is an integer p denoting the length of the string w such that w is obtained by going through the loop exactly once. In other words, |w| = |xyz| = p.

9. Of what use is this pumping length p?
We can be sure that |xy| ≤ p. This can be used to prove a language non-regular.

Now, let's define a regular language based on the pumping property.

Pumping Lemma: If L is a regular language, then there exists a constant p such that every string w ∈ L, of length p or more can be written as w = xyz, where

1. |y| > 0
2. |xy| ≤ p
3. $xy^iz \in L$ for all i

Proving languages non-regular

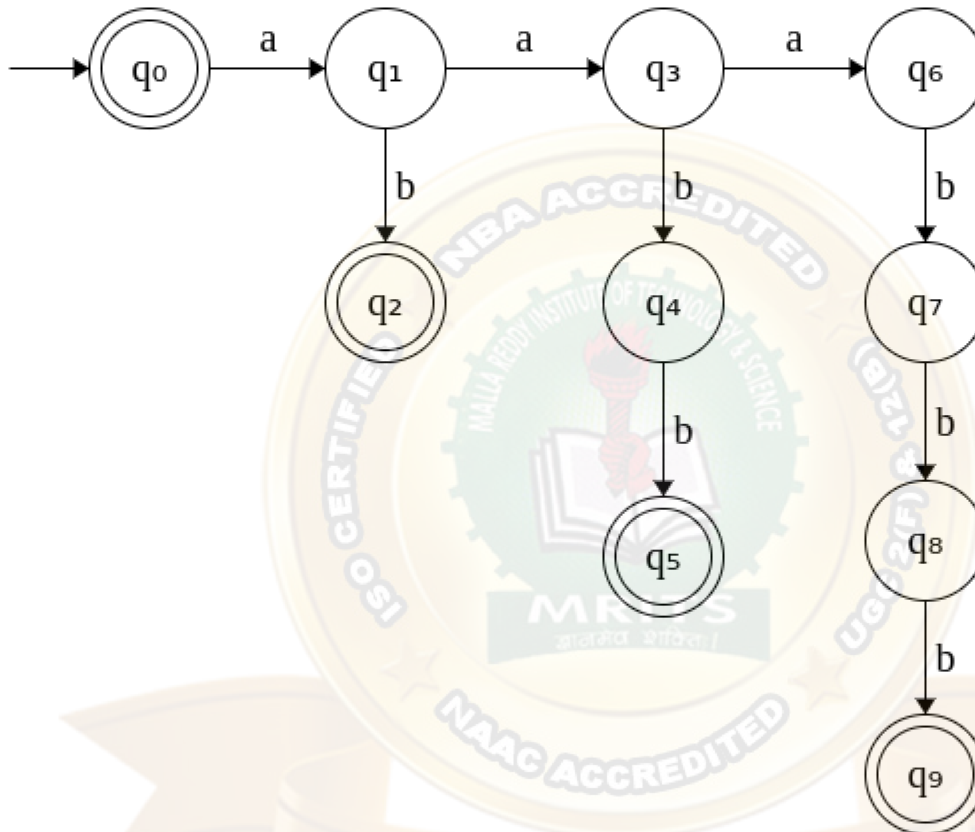
1. The language L = { a^nb^n : n ≠ 0 } is not regular.

Before proving L is not regular using pumping property, let's see why we can't come up with a DFA or regular expression for L.

L = { ε, ab, aabb, aaabbb,..... }

It may be tempting to use the regular expression a*b* to describe L. No doubt, a*b* generates these strings. However, it is not appropriate since it generates other strings not in L such as a, b, aa, ab, aaa, aab, abb, ...

Let's try to come up with a DFA. Since it has to accept ε, start state has to be final. The following DFA can accept $a^n b^n$ for $n \leq 3$. i.e. {ε, a, b, ab, aabb, aaabbb}



The basic problem is DFA does not have any memory. A transition just depends on the current state. So it cannot keep count of how many a's it has seen. So, it has no way to match the number of a's and b's. So, only way to accept all the strings of L is to keep adding newer and newer states which makes automaton to infinite states since n is unbounded.

Now, let's prove that L does not have the pumping property.

Lets assume L is regular. Let p be the pumping length.

Consider a string w = aa....abb. b such that |w| = p.

⇒ $w = a^{p/2} b^{p/2}$

We know that w can be broken into three terms xyz such that $y \neq \varepsilon$ and $xy^iz \in L$.

There are three cases to consider.

- Case 1: y is made up of only a's

  Then xy^2z has more a's than b's and does not belong to L.

- Case 2: y is made up of only b's

  Then xy^2z has more b's than a's and does not belong to L.

- Case 3: y is made up of a's and b's

  Then xy^2z has a's and b's out of order and does not belong to L.

Since none of the 3 cases hold, the pumping property does not hold for L. And therefore L is not regular.

2. The language $L = \{ uu^R : u \in \{a,b\}^* \}$ is not regular.

Lets assume L is regular. Let p be the pumping length.

Consider a string $w = a^pbba^p$.

$|w| = 2p + 2 \geq p$

Since, $xy \leq p$, xy will consist of only a's.

$\Rightarrow$ y is made of only a's

$\Rightarrow$ y^2 is made of more number of a's than y since $|y| > 0$
(Let's say y^2 has m a's more than y where $m > 1$)

$\Rightarrow$ $xy^2z = a^{p+m}bba^p$ where $m \geq 1$

$\Rightarrow$ $xy^2z = a^{p+m}bba^p$ cannot belong to L.

Therefore, pumping property does not hold for L. Hence, L is not regular.

3. The language $L = \{ a^n : n \text{ is prime} \}$ is not regular.

Lets assume L is regular. Let p be the pumping length. Let $q \geq p$ be a prime number (since we cannot assume that pumping length p will be prime).

Consider the string w = aa....a such that $|w| = q \geq p$.

We know that w can be broken into three terms xyz such that $y \neq \varepsilon$ and $xy^iz \in L$

$\Rightarrow xy^{q+1}z$ must belong to L
$\Rightarrow |xy^{q+1}z|$ must be prime

$|xy^{q+1}z| = |xyzy^q|$
$= |xyz| + |y^q|$
$= q + q.|y|$
$= q(1 + |y|)$ which is a composite number.

Therefore, $xy^{q+1}z$ cannot belong to L. Hence, L is not regular.

Exercises

Show that the following languages are not regular.

4. $L = \{ a^nb^m : n \neq m \}$
5. $L = \{ a^nb^m : n > m \}$
6. $L = \{ w : n_a(w) = n_b(w) \}$
7. $L = \{ ww : w \in \{a,b\}^* \}$
8. $L = \{ a^{n^2} : n > 0 \}$

IMPORTANT NOTE

Never use pumping lemma to prove a language regular. Pumping property is necessary but not sufficient for regularity.

Closure Properties of Regular Languages:

For natural language that is regulated, see List of language regulators.

"Kleene's theorem" redirects here. For his theorems for recursive functions, see Kleene's recursion theorem.

In theoretical computer science and formal language theory, a **regular language** (also called a **rational language**[1][2]) is a formal language that can be expressed using a regular expression, in the strict sense of the latter notion used in theoretical computer science (as opposed to many regular expressions engines provided by modern programming languages, which are augmented with features that allow recognition of languages that cannot be expressed by a classic regular expression).

Alternatively, a regular language can be defined as a language recognized by a finite automaton. The equivalence of regular expressions and finite automata is known as **Kleene's theorem**[3] (after American mathematician Stephen Cole Kleene). In the Chomsky hierarchy, regular languages are defined to be the languages that are generated by Type-3 grammars (regular grammars).

Regular languages are very useful in input parsing and programming language design

Closure properties of Regular languages

Formal definition

The collection of regular languages over an alphabet Σ is defined recursively as follows:

- The empty language Ø, and the empty string language {ε} are regular languages.
- For each $a \in \Sigma$ (a belongs to Σ), the singleton language $\{a\}$ is a regular language.
- If A and B are regular languages, then $A \cup B$ (union), $A \bullet B$ (concatenation), and A^* (Kleene star) are regular languages.
- No other languages over Σ are regular.

See regular expression for its syntax and semantics. Note that the above cases are in effect the defining rules of regular expression.

Examples

All finite languages are regular; in particular the empty string language {ε} = Ø* is regular. Other typical examples include the language consisting of all strings over the alphabet $\{a, b\}$ which contain an even number of as, or the language consisting of all strings of the form: several as followed by several bs.

A simple example of a language that is not regular is the set of strings $\{ a^n b^n \mid n \geq 0 \}$.[4] Intuitively, it cannot be recognized with a finite automaton, since a finite automaton has finite memory and it cannot remember the exact number of a's. Techniques to prove this fact rigorously are given below.

Equivalent formalisms

A regular language satisfies the following equivalent properties:

1. it is the language of a regular expression (by the above definition)
2. it is the language accepted by a nondeterministic finite automaton (NFA)[note 1][note 2]
3. it is the language accepted by a deterministic finite automaton (DFA)[note 3][note 4]
4. it can be generated by a regular grammar[note 5][note 6]
5. it is the language accepted by an alternating finite automaton

6. it can be generated by a prefix grammar
7. it can be accepted by a read-only Turing machine
8. it can be defined in monadic second-order logic (Büchi-Elgot-Trakhtenbrot theorem[5])
9. it is recognized by some finite monoid M, meaning it is the preimage $\{ w \in \Sigma^* \mid f(w) \in S \}$ of a subset S of a finite monoid M under a monoid homomorphism $f: \Sigma^* \to M$ from the free monoid on its alphabet[note 7]
10. the number of equivalence classes of its "syntactic relation" ~ is finite[note 8][note 9] (this number equals the number of states of the minimal deterministic finite automaton accepting L.)

Properties 9. and 10. are purely algebraic approaches to define regular languages; a similar set of statements can be formulated for a monoid $M \subset \Sigma^*$. In this case, equivalence over M leads to the concept of a recognizable language.

Some authors use one of the above properties different from "1." as alternative definition of regular languages.

Some of the equivalences above, particularly those among the first four formalisms, are called *Kleene's theorem* in textbooks. Precisely which one (or which subset) is called such varies between authors. One textbook calls the equivalence of regular expressions and NFAs ("1." and "2." above) "Kleene's theorem".[6] Another textbook calls the equivalence of regular expressions and DFAs ("1." and "3." above) "Kleene's theorem".[7] Two other textbooks first prove the expressive equivalence of NFAs and DFAs ("2." and "3.") and then state "Kleene's theorem" as the equivalence between regular expressions and finite automata (the latter said to describe "recognizable languages").[2][8] A linguistically oriented text first equates regular grammars ("4." above) with DFAs and NFAs, calls the languages generated by (any of) these "regular", after which it introduces regular expressions which it terms to describe "rational languages", and finally states "Kleene's theorem" as the coincidence of regular and rational languages.[9] Other authors simply *define* "rational expression" and "regular expressions" as synonymous and do the same with "rational languages" and "regular languages".[1][2]

Closure properties

The regular languages are closed under the various operations, that is, if the languages K and L are regular, so is the result of the following operations:

- the set theoretic Boolean operations: union $K \cup L$, intersection $K \cap L$, and complement L, hence also relative complement $K-L$.
- the regular operations: $K \cup L$, concatenation $K \circ L$, and Kleene star L^*.
- the trio operations: string homomorphism, inverse string homomorphism, and intersection with regular languages. As a consequence they are closed under arbitrary finite state transductions, like quotient K / L with a regular language. Even more, regular languages are closed under quotients with *arbitrary* languages: If L is regular then L/K is regular for any K.[*citation needed*]
- the reverse (or mirror image) L^R.[*citation needed*]

Decidability properties

Given two deterministic finite automata A and B, it is decidable whether they accept the same language.[12] As a consequence, using the above closure properties, the following problems are also decidable for arbitrarily given deterministic finite automata A and B, with accepted languages L_A and L_B, respectively:

- Containment: is $L_A \subseteq L_B$?[note 10]
- Disjointness: is $L_A \cap L_B = \{\}$?
- Emptiness: is $L_A = \{\}$?
- Universality: is $L_A = \Sigma^*$?
- Membership: given $a \in \Sigma^*$, is $a \in L_B$?

For regular expressions, the universality problem is NP-complete already for a singleton alphabet.[13] For larger alphabets, that problem is PSPACE-complete. If regular expressions are extended to allow also a *squaring operator*, with "A^2" denoting the same as "AA", still just regular languages can be described, but the universality problem has an exponential space lower bound,[16][17][18] and is in fact complete for exponential space with respect to polynomial-time reduction.[19]

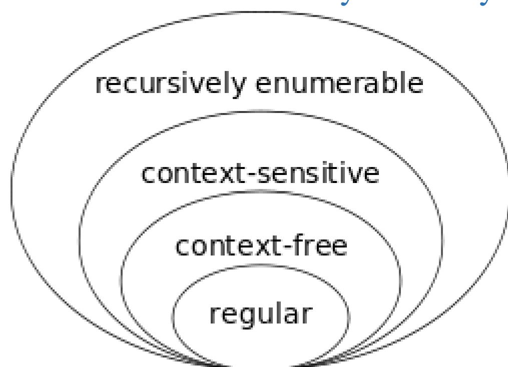
Complexity results

In computational complexity theory, the complexity class of all regular languages is sometimes referred to as **REGULAR** or **REG** and equals DSPACE(O(1)), the decision problems that can be solved in constant space (the space used is independent of the input size). **REGULAR** $\neq$ **AC**^{0}, since it (trivially) contains the parity problem of determining whether the number of 1 bits in the input is even or odd and this problem is not in **AC**^{0}. On the other hand, **REGULAR** does not

contain **AC**^{0}, because the nonregular language of palindromes, or the nonregular language can both be recognized in **AC**^{0}.

If a language is *not* regular, it requires a machine with at least $\Omega(\log \log n)$ space to recognize (where n is the input size).[22] In other words, DSPACE($o(\log \log n)$) equals the class of regular languages. In practice, most nonregular problems are solved by machines taking at least logarithmic space.

Location in the Chomsky hierarchy



Regular language in classes of Chomsky hierarchy.

To locate the regular languages in the Chomsky hierarchy, one notices that every regular language is context-free. The converse is not true: for example the language consisting of all strings having the same number of *a*'s as *b*'s is context-free but not regular. To prove that a language such as this is not regular, one often uses the Myhill–Nerode theorem or the pumping lemma among other methods.[23]

Important subclasses of regular languages include

- Finite languages - those containing only a finite number of words.[24] These are regular languages, as one can create a regular expression that is the union of every word in the language.
- Star-free languages, those that can be described by a regular expression constructed from the empty symbol, letters, concatenation and all boolean operators including complementation but not the Kleene star: this class includes all finite languages.[25]
- **Cyclic languages**, satisfying the conditions $uv \in L \Leftrightarrow vu \in L$ and $w \in L \Leftrightarrow w^n \in L$. The number of words in a regular language

Let denote the number of words of length in . The ordinary generating function for *L* is the formal power series

> The generating function of a language *L* is a rational function if *L* is regular. Hence for any regular language there exist an integer constant , complex constants and complex polynomials such that for every the number of words of length in is .Thus, non-regularity of certain languages can be proved by counting the words of a given length in . Consider, for example, the Dyck language of strings of balanced parentheses. The number of words of length in the Dyck language is equal to the Catalan number , which is not of the form , witnessing the non-regularity of the Dyck language. Care must be taken since some of the eigenvalues could have the same magnitude. For example, the number of words of length in the language of all even binary words is not of the form , but the number of words of even or odd length are of this form; the corresponding eigenvalues are . In general, for every regular language there exists a constant such that for all , the number of words of length is asymptotically .The *zeta function* of a language *L* is The zeta function of a regular language is not in general rational, but that of a cyclic language is.

Generalizations

The notion of a regular language has been generalized to infinite words (see ω-automata) and to trees (see tree automaton).

Rational set generalizes the notion (of regular/rational language) to monoids that are not necessarily free. Likewise, the notion of a recognizable language (by a finite automaton) has namesake as recognizable set over a monoid that is not necessarily free. Howard Straubing notes in relation to these facts that "The term "regular language" is a bit unfortunate. Papers influenced by Eilenberg's monograph[35] often use either the term "recognizable language", which refers to the behavior of automata, or "rational language", which refers to important analogies between regular expressions and rational power series. (In fact, Eilenberg defines rational and recognizable subsets of arbitrary monoids; the two notions do not, in general, coincide.) This terminology, while better motivated, never really caught on, and "regular language" is used almost universally."[36]

Rational series is another generalization, this time in the context of a formal power series over a semiring. This approach gives rise to weighted rational expressions and weighted automata. In this algebraic context, the regular languages (corresponding to Boolean-weighted rational expressions) are usually called *rational languages*. Also in this context, Kleene's theorem finds a generalization called the Kleene-Schützenberger theorem.

Decision Properties of Regular Languages

• Some of these questions we can answer already with what we know
– Is a given string in the language?
– Given 2 languages are there strings that are in both?
• Others require additional tools:
– Is the language the same as another regular language?
– Is there a language L that is not
regular?

Equivalence and Minimization of Automata

DFA Minimization using Myphill-Nerode Theorem

Algorithm

Input − DFA

Output − Minimized DFA

Step 1 − Draw a table for all pairs of states (Q_i, Q_j) not necessarily connected directly [All are unmarked initially]

Step 2 − Consider every state pair (Q_i, Q_j) in the DFA where $Q_i \in F$ and $Q_j \notin F$ or vice versa and mark them. [Here F is the set of final states]

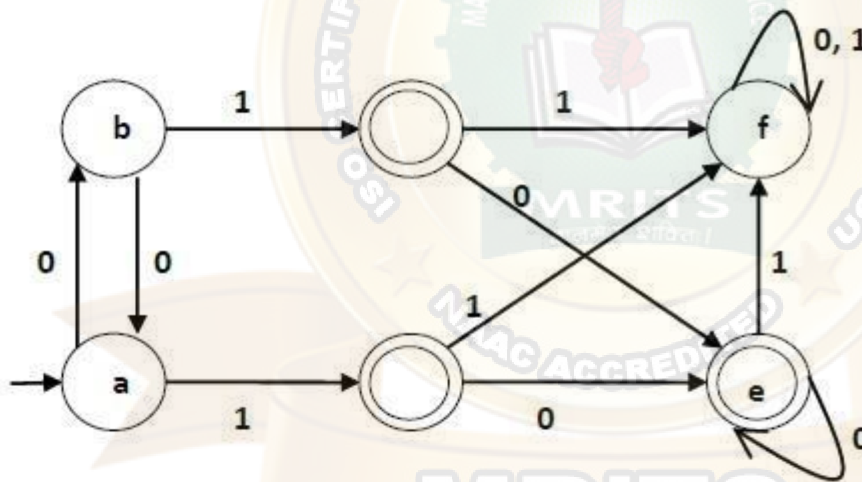
Step 3 − Repeat this step until we cannot mark anymore states −

If there is an unmarked pair (Q_i, Q_j), mark it if the pair {δ (Q_i, A), δ (Q_i, A)} is marked for some input alphabet.

Step 4 − Combine all the unmarked pair (Q_i, Q_j) and make them a single state in the reduced DFA.

Example

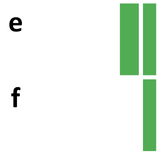
Let us use Algorithm 2 to minimize the DFA shown below.



Step 1 − We draw a table for all pair of states.

	a	b	c	d	e	f
a						
b						
c						
d						

NBA ACCREDITED MRITS



Step 2 − We mark the state pairs.

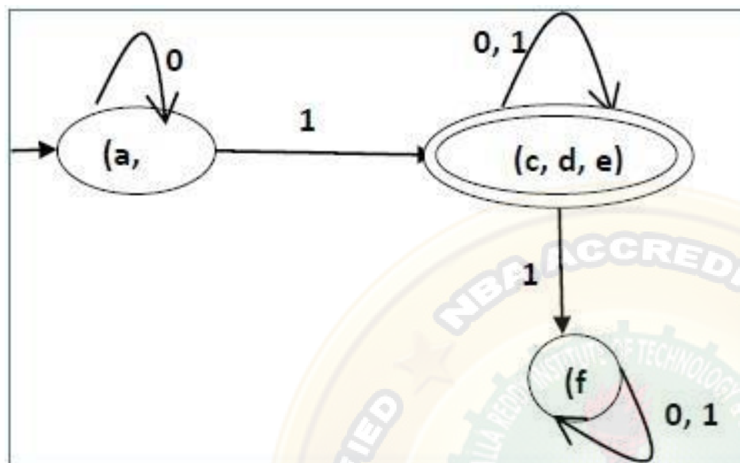
Step 3 − We will try to mark the state pairs, with green colored check mark, transitively. If we input 1 to state ‘a’ and ‘f’, it will go to state ‘c’ and ‘f’ respectively. (c, f) is already marked, hence we will mark pair (a, f). Now, we input 1 to state ‘b’ and ‘f’; it will go to state ‘d’ and ‘f’ respectively. (d, f) is already marked, hence we will mark pair (b, f).

After step 3, we have got state combinations {a, b} {c, d} {c, e} {d, e} that are unmarked.

We can recombine {c, d} {c, e} {d, e} into {c, d, e}

Hence we got two combined states as − {a, b} and {c, d, e}

So the final minimized DFA will contain three states {f}, {a, b} and {c, d, e}



DFA Minimization using Equivalence Theorem

If X and Y are two states in a DFA, we can combine these two states into {X, Y} if they are not distinguishable. Two states are distinguishable, if there is at least one string S, such that one of δ (X, S) and δ (Y, S) is accepting and another is not accepting. Hence, a DFA is minimal if and only if all the states are distinguishable.

Algorithm 3

Step 1 − All the states **Q** are divided in two partitions − **final states** and **non-final states** and are denoted by $\mathbf{P_0}$. All the states in a partition are 0th equivalent. Take a counter **k** and initialize it with 0.

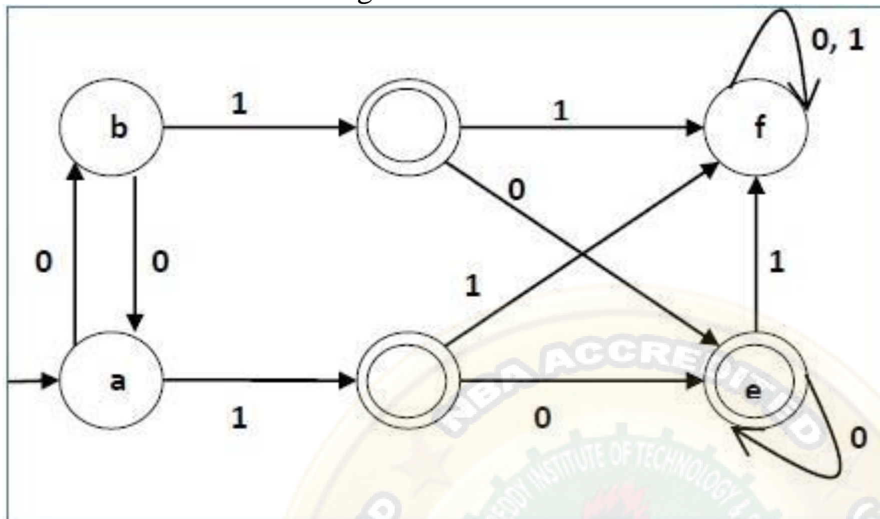
Step 2 − Increment k by 1. For each partition in P_k, divide the states in P_k into two partitions if they are k-distinguishable. Two states within this partition X and Y are k-distinguishable if there is an input **S** such that **δ(X, S)** and **δ(Y, S)** are (k-1)-distinguishable.

Step 3 − If $P_k \neq P_{k-1}$, repeat Step 2, otherwise go to Step 4.

Step 4 − Combine k^{th} equivalent sets and make them the new states of the reduced DFA.

Example

Let us consider the following DFA −



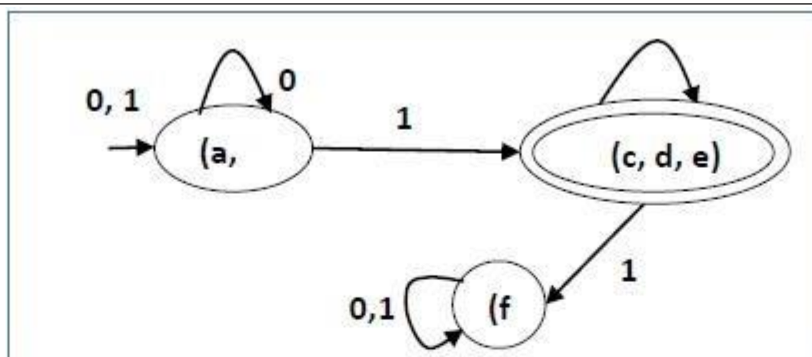
q	δ(q,0)	δ(q,1)
a	b	c
b	a	d
c	e	f
d	e	f
e	e	f
f	f	f

Let us apply the above algorithm to the above DFA −

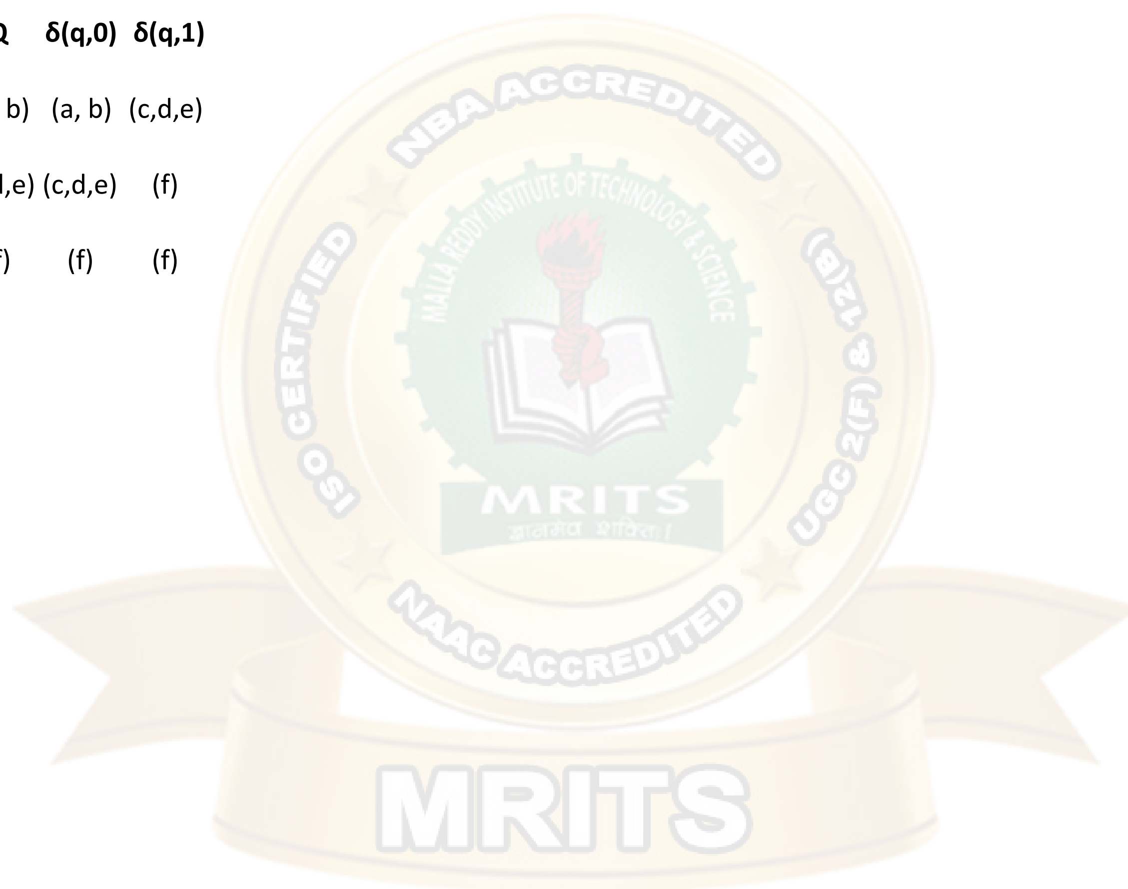
- P_0 = {(c,d,e), (a,b,f)}
- P_1 = {(c,d,e), (a,b),(f)}
- P_2 = {(c,d,e), (a,b),(f)}

Hence, $P_1 = P_2$.

There are three states in the reduced DFA. The reduced DFA is as follows −



Q	δ(q,0)	δ(q,1)
(a, b)	(a, b)	(c,d,e)
(c,d,e)	(c,d,e)	(f)
(f)	(f)	(f)



MALL REDDY INSTITUTE OF TECHNOLOGY & SCIENCE AND SCIENCE

LECTURE NOTES

On

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE/IT I-Sem

(Jntuh-R18)

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE I-Sem **L T P C**

3 0 0 3

Course Objectives

1. To provide introduction to some of the central ideas of theoretical computer science from the perspective of formal languages.
2. To introduce the fundamental concepts of formal languages, grammars and automata theory.
3. Classify machines by their power to recognize languages.
4. Employ finite state machines to solve problems in computing.
5. To understand deterministic and non-deterministic machines.
6. To understand the differences between decidability and undecidability.

Course Outcomes

1. Able to understand the concept of abstract machines and their power to recognize the languages.
2. Able to employ finite state machines for modeling and solving computing problems.
3. Able to design context free grammars for formal languages.
4. Able to distinguish between decidability and undecidability.

UNIT - I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory – Alphabets, Strings, Languages, Problems.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines

UNIT - II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages, Statement of the pumping lemma, Applications of the Pumping Lemma.

Closure Properties of Regular Languages: Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

UNIT - III

Context-Free Grammars: Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages. **Push Down Automata**: Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG

UNIT - IV

Normal Forms for Context- Free Grammars: Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form Griebech Normal form.

Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications **Closure Properties of Context-Free Languages**: Closure properties of CFL’s, Decision Properties of CFL's

Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

UNIT - V

Types of Turing machine: Turing machines and halting

Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem, Other Undecidable Problems, Counter machines.

TEXT BOOKS:

1. Introduction to Automata Theory, Languages, and Computation, 3nd Edition, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science – Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCE BOOKS:

1. Introduction to Languages and The Theory of Computation, John C Martin, TMH.
2. Introduction to Computer Theory, Daniel I.A. Cohen, John Wiley.
3. A Text book on Automata Theory, P. K. Srimani, Nasir S. F. B, Cambridge University Press.

4. Introduction to the Theory of Computation, Michael Sipser, 3rd edition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and Computation Kamala Krithivasan, Rama R, Pearson.

UNIT-III

Definition − A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

- **N** is a set of non-terminal symbols.
- **T** is a set of terminals where **N ∩ T = NULL.**
- **P** is a set of rules, **P: N → (N ∪ T)***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
- **S** is the start symbol.

Example

- The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
- The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
- The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

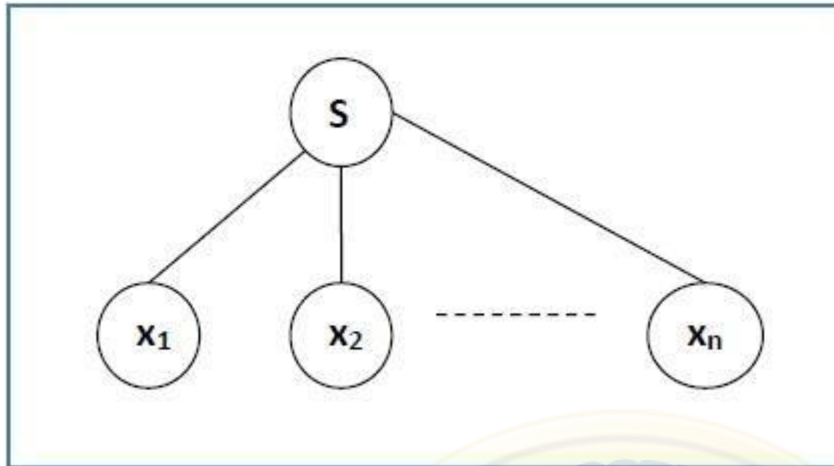
Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

- **Root vertex** − Must be labeled by the start symbol.
- **Vertex** − Labeled by a non-terminal symbol.
- **Leaves** − Labeled by a terminal symbol or ε.

If S → x_1x_2 …… x_n is a production rule in a CFG, then the parse tree / derivation tree will be as follows −



There are two different approaches to draw a derivation tree −

Top-down Approach −

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

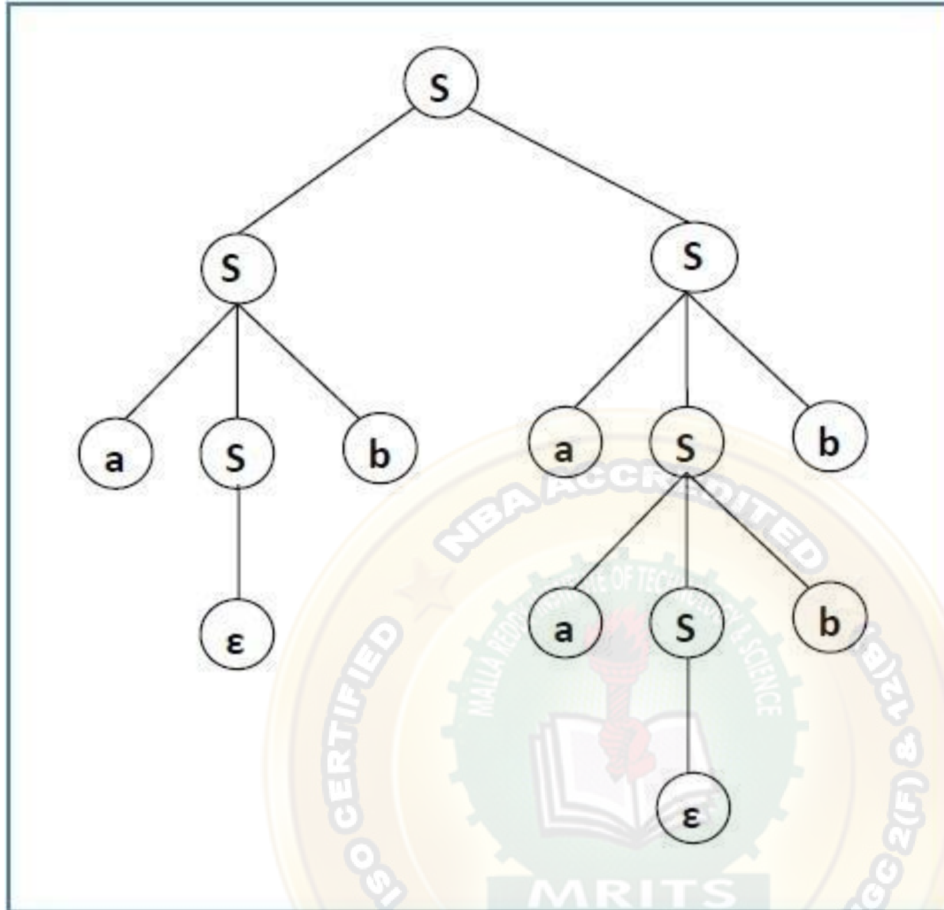
Example

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is "abaabb"

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



Sentential Form and Partial Derivation Tree

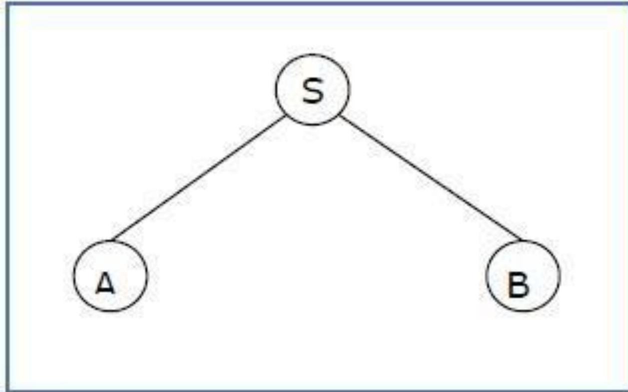
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are −

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following −



If a partial derivation tree contains the root S, it is called a **sentential form**. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String

- **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

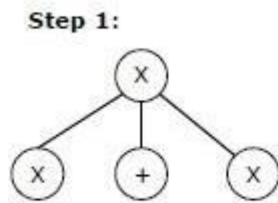
X → X+X | X*X |X| a

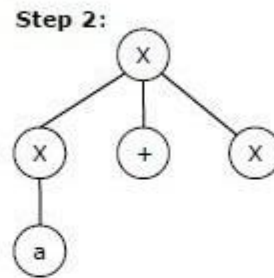
over an alphabet {a}.

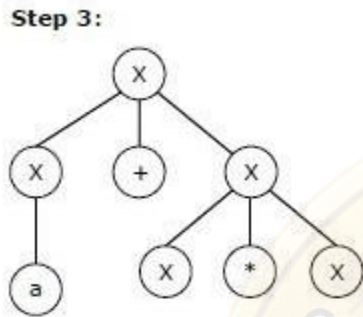
The leftmost derivation for the string **"a+a*a"** may be −

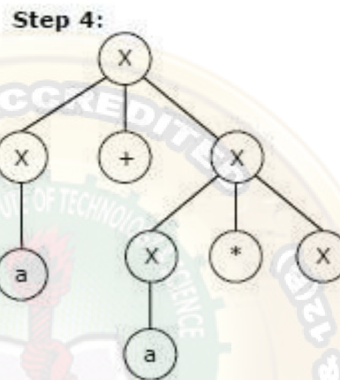
X → X+X → a+X → a + X*X → a+a*X → a+a*a

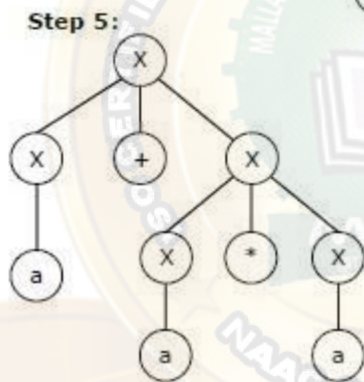
The stepwise derivation of the above string is shown as below −









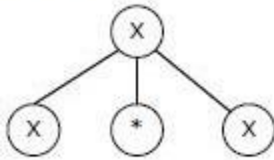


The rightmost derivation for the above string **"a+a*a"** may be −

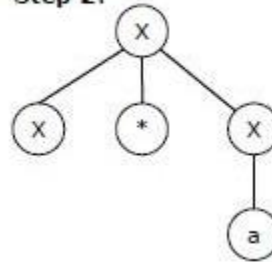
X → X*X → X*a → X+X*a → X+a*a → a+a*a

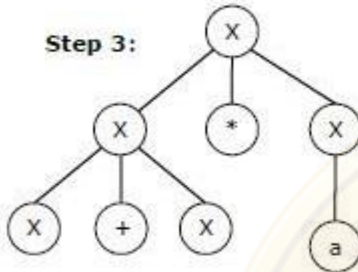
The stepwise derivation of the above string is shown as below −

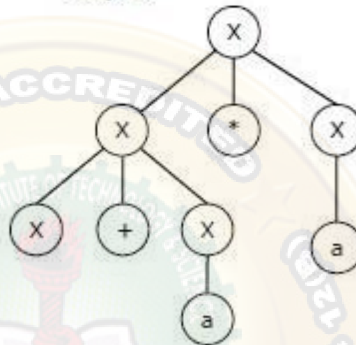
Step 1:



Step 2:







Left and Right Recursive Grammars

In a context-free grammar **G**, if there is a production in the form **X → Xa** where **X** is a non-terminal and **'a'** is a string of terminals, it is called a **left recursive production**. The grammar having a left recursive production is called a **left recursive grammar**.

And if in a context-free grammar **G**, if there is a production is in the form **X → aX** where **X** is a non-terminal and **'a'** is a string of terminals, it is called a **right recursive production**. The grammar having a right recursive production is called a **right recursive grammar**.

Context-Free Grammars

A context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to generate patterns of strings.

A CFG consists of the following components:

- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.

A CFG for Arithmetic Expressions

An example grammar that generates strings representing arithmetic expressions with the four operators +, -, *, /, and numbers as operands is:

1. <expression> --> number
2. <expression> --> (<expression>)
3. <expression> --> <expression> + <expression>
4. <expression> --> <expression> - <expression>
5. <expression> --> <expression> * <expression>
6. <expression> --> <expression> / <expression>

The only nonterminal symbol in this grammar is <expression>, which is also the start symbol. The terminal symbols are {+,-,*,/,(,),number}. (We will interpret "number" to represent any valid number.)

The first rule (or production) states that an <expression> can be rewritten as (or replaced by) a number. In other words, a number is a valid expression.

The second rule says that an <expression> enclosed in parentheses is also an <expression>. Note that this rule defines an expression in terms of expressions, an example of the use of recursion in the definition of context-free grammars.

The remaining rules say that the sum, difference, product, or division of two <expression>s is also an expression.

Generating Strings from a CFG

In our grammar for arithmetic expressions, the start symbol is <expression>, so our initial string is:

<expression>

Using rule 5 we can choose to replace this nonterminal, producing the string:

<expression> * <expression>

We now have two nonterminals to replace. We can apply rule 3 to the first nonterminal, producing the string:

<expression> + <expression> * <expression>

We can apply rule two to the first nonterminal in this string to produce:

(<expression>) + <expression> * <expression>

If we apply rule 1 to the remaining nonterminals (the recursion must end somewhere!), we get:

(number) + number * number

This is a valid arithmetic expression, as generated by the grammar.

When applying the rules above, we often face a choice as to which production to choose. Different choices will typically result in different strings being generated.

Given a grammar G with start symbol S, if there is some sequence of productions that, when applied to the initial string S, result in the string s, then s is in L(G), the language of the grammar.

CFGs with Epsilon Productions

A CFG may have a production for a nonterminal in which the right hand side is the empty string (which we denote by *epsilon*). The effect of this production is to remove the nonterminal from the string being generated.

Here is a grammar for balanced parentheses that uses epsilon productions.

P --> (P)
P --> P P
P --> epsilon

We begin with the string P. We can replace P with epsilon, in which case we have generated the empty string (which does have balanced parentheses). Alternatively, we can generate a string of balanced parentheses within a pair of balanced parentheses, which must result in a string of balanced parentheses. Alternatively, we can concatenate two strings of balanced parentheses, which again must result in a string of balanced parentheses.

This grammar is equivalent to:

P --> (P) | P P | epsilon

We use the notational shorthand '|', which can be read as "or", to represent multiple rewriting rules within a single line.

CFG Examples

A CFG describing strings of letters with the word "main" somewhere in the string:

<program> --> <letter*> m a i n <letter*>
<letter*> --> <letter> <letter*> | epsilon
<letter> --> A | B | ... | Z | a | b ... | z

A CFG for the set of identifiers in Pascal:

<id> --> <L> <LorD*>
<LorD*> --> <L> <LorD*> | <D> <LorD*> | epsilon
<L> --> A | B | ... | Z | a | b ... | z
<D> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A CFG describing real numbers in Pascal:

<real> --> <digit> <digit*> <decimal part> <exp>
<digit*> --> <digit> <digit*> | epsilon
<decimal part> --> '.' <digit> <digit*> | epsilon

<exp> --> 'E' <sign> <digit> <digit*> | epsilon
<sign> --> + | - | epsilon
<digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A CFG for C++ compound statements:

<compound stmt> --> { <stmt list> }
<stmt list> --> <stmt> <stmt list> | epsilon
<stmt> --> <compound stmt>
<stmt> --> if (<expr>) <stmt>
<stmt> --> if (<expr>) <stmt> else <stmt>
<stmt> --> while (<expr>) <stmt>
<stmt> --> do <stmt> while (<expr>) ;
<stmt> --> for (<stmt> <expr> ; <expr>) <stmt>
<stmt> --> case <expr> : <stmt>
<stmt> --> switch (<expr>) <stmt>
<stmt> --> break ; | continue ;
<stmt> --> return <expr> ; | goto <id> ;

Finding all the Strings Generated by a CFG

There are several ways to generate the (possibly infinite) set of strings generated by a grammar. We will show a technique based on the number of productions used to generate the string.

Find the strings generated by the following CFG:

<S> --> w c d <S> | b <L> e | s
<L> --> <L> ; <S> | <S>

0. Applying at most zero productions, we cannot generate any strings.

1. Applying at most one production (starting with the start symbol) we can generate {wcd<S>, b<L>e, s}. Only one of these strings consists entirely of terminal symbols, so the set of terminal strings we can generate using at most one production is {s}.

2. Applying at most two productions, we can generate all the strings we can generate with one production, plus any additional strings we can generate with an additional production.

{wcdwcd<S>, wcdb<L>e, wcds, b<S>e, b<L>;<S>e,s}

The set of terminal strings we can generate with at most two productions is therefore {s, wcds}.

3. Applying at most three productions, we can generate:

{wcdwcdwcd<S>, wcdwcdb<L>e, wcdwcds, wcdb<L>;<S>e,
wcdb<S>e, bwcd<S>e, bb<L>ee, bse, b<L>;<S>Se,

b<S><S>e, b<L>wcd<S>e, b<L>b<L>ee, b<L>se }

The set of terminal strings we can generate with at most three productions is therefore {s, wcds, wcdwcds, bse}.

We can repeat this process for an arbitrary number of steps N, and find all the strings the grammar can generate by applying N productions.

CFGs vs Regular Expressions

Context-free grammars are strictly more powerful than regular expressions.

- Any language that can be generated using regular expressions can be generated by a context-free grammar.
- There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.

As a corollary, CFGs are strictly more powerful than DFAs and NDFAs.

The proof is in two parts:

- Given a regular expression R , we can generate a CFG G such that L(R) == L(G).
- We can define a grammar G for which there there is no FA F such that L(F) == L(G)

In formal language theory, a **context-free grammar** (**CFG**) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the rule replaces with 5. There can be

multiple replacement rules for any given value. For example,

means that can be replaced with either or .In context-free grammars, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is always a nonterminal symbol. This means that the symbol does not appear in the resulting formal language. So in our case, our language contains the letters and but not

Rules can also be applied in reverse to check if a string is grammatically correct according to the grammar.

Here is an example context-free grammar that describes all two-letter strings containing the letters and .

If we start with the nonterminal symbol then we can use the rule to turn into . We can then apply one of the two later rules. For example, if we apply to the first we get . If we then apply to the

second we get . Since both and are terminal symbols, and in context-free grammars terminal

symbols never appear on the left hand side of a production rule, there are no more rules that can be applied. This same process can be used, applying the second two rules in different orders in order to get all possible strings within our simple context-free grammar.

Languages generated by context-free grammars are known as context-free languages (CFL). Different context-free grammars can generate the same context-free language. It is important to distinguish the properties of the language (intrinsic properties) from the properties of a particular grammar (extrinsic properties). The language equality question (do two given context-free grammars generate the same language?) is undecidable.

Context-free grammars arise in linguistics where they are used to describe the structure of sentences and words in a natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose, but have not really lived up to their original expectation. By contrast, in computer science, as the use of recursively-defined concepts increased, they were used more and more. In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the *Document Type Definition*.[2]

In linguistics, some authors use the term **phrase structure grammar** to refer to context-free grammars, whereby phrase-structure grammars are distinct from dependency grammars. In computer science, a popular notation for context-free grammars is Backus–Naur form, or *BNF*.

Since the time of Pāṇini, at least, linguists have described the grammars of languages in terms of their block structure, and described how sentences are recursively built up from smaller phrases, and eventually individual words or word elements. An essential property of these block structures is that logical units never overlap. For example, the sentence:John, whose blue car was in the garage, walked to the grocery store.can be logically parenthesized as follows:(John, ((whose blue car) (was (in the garage))), (walked (to (the grocery store)))).

A context-free grammar provides a simple and mathematically precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the "block structure" of sentences in a natural way. Its simplicity makes the formalism amenable to rigorous mathematical study. Important features of natural language syntax such as agreement and reference are not part of the context-free grammar, but the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of adjectives and adverbs are swallowed by nouns and verbs, is described exactly.The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky,[3] and also their classification as a special type of formal grammar (which he called phrase-structure grammars).[4] What Chomsky called a phrase structure grammar is also known now as a constituency grammar, whereby constituency grammars stand in contrast to dependency grammars. In Chomsky's generative grammar framework, the syntax of natural language was described by context-free rules combined with transformation rules.

Block structure was introduced into computer programming languages by the Algol project (1957–1960), which, as a consequence, also featured a context-free grammar to describe the resulting Algol syntax. This became a standard feature of computer languages, and the notation

for grammars used in concrete descriptions of computer languages came to be known as Backus–Naur form, after two members of the Algol language design committee.[3] The "block structure" aspect that context-free grammars capture is so fundamental to grammar that the terms syntax and grammar are often identified with context-free grammar rules, especially in computer science. Formal constraints not captured by the grammar are then considered to be part of the "semantics" of the language.

Context-free grammars are simple enough to allow the construction of efficient parsing algorithms that, for a given string, determine whether and how it can be generated from the grammar. An Earley parser is an example of such an algorithm, while the widely used LR and LL parsers are simpler algorithms that deal only with more restrictive subsets of context-free grammars.

Formal definitions

A context-free grammar *G* is defined by the 4-tuple:where

1. *V* is a finite set; each element is called *a nonterminal character* or a *variable*. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by *G*.
2. Σ is a finite set of *terminal*s, disjoint from *V*, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar *G*.
3. *R* is a finite relation from *V* to , where the asterisk represents the Kleene star operation. The members of *R* are called the *(rewrite) rule*s or *production*s of the grammar. (also commonly symbolized by a *P*)
4. *S* is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of *V*.

Production rule notation

A production rule in *R* is formalized mathematically as a pair , where is a nonterminal and is a string of ariables and/or terminals; rather than using ordered pair notation, production rules are usually written using an arrow operator with α as its left hand side and β as its right hand side: .It is allowed for β to be the empty string, and in this case it is customary to denote it by ε. The form is called an *ε*-production.It is common to list all right-hand sides for the same left-hand side on the same line, using | (the pipe symbol) to separate them. Rules and can hence be written as . In this case, and is called the first and second alternative, respectively.

Rule application

For any strings , we say *u* directly yields *v*, written as , if with and such that and . Thus, *v* is a result of applying the rule to *u*.

Repetitive rule application

For any strings we say *u* **yields** *v*, written as (or in some textbooks), if such that . In this case, if (i.e.,), the relation holds. In other words, and are the reflexive transitive closure (allowing a word to yield itself) and the transitive closure (requiring at least one step) of , respectively.

Context-free language

The language of a grammar is the set

A language *L* is said to be a context-free language (CFL), if there exists a CFG *G*, such that .

Proper CFGs

A context-free grammar is said to be *proper*,[7] if it has

- no unreachable symbols:
- no unproductive symbols:
- no ε-productions:
- no cycles:

Every context-free grammar can be effectively transformed into a weakly equivalent one without unreachable symbols,[8] a weakly equivalent one without unproductive symbols,[9] and a weakly equivalent one without cycles.[10] Every context-free grammar not producing ε can be effectively transformed into a weakly equivalent one without ε-productions;[11] altogether, every such grammar can be effectively transformed into a weakly equivalent proper CFG.

Example

The grammar , with productions

$S \to aSa$,

$S \to bSb$,

$S \to \varepsilon$,

is context-free. It is not proper since it includes an ε-production. A typical derivation in this grammar is

$S \to aSa \to aaSaa \to aabSbaa \to aabbaa$.

This makes it clear that . The language is context-free, however, it can be proved that it is not regular.

Examples

Well-formed parentheses

The canonical example of a context-free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

S → SS

S → (S)

S → ()

The first rule allows the S symbol to multiply; the second rule allows the S symbol to become enclosed by matching parentheses; and the third rule terminates the recursion.

Well-formed nested parentheses and square brackets

A second canonical example is two different kinds of matching nested parentheses, described by the productions:

S → SS

S → ()

S → (S)

S → []

S → [S]

with terminal symbols [ ] ( ) and nonterminal S.

The following sequence can be derived in that grammar:

([[[()() [ ][ ]]]([ ])])

However, there is no context-free grammar for generating all sequences of two different types of parentheses, each separately balanced disregarding the other, but where the two types need not nest inside one another, for example:

[( ])

or

[[[[(((( ]]]]))))(([ ))(([ ))([ )( ])( ])( ])

A regular grammar

Every regular grammar is context-free, but not all context-free grammars are regular. The following context-free grammar, however, is also regular.

S → a

S → aS

S → bS

The terminals here are *a* and *b*, while the only nonterminal is S. The language described is all

nonempty strings of s and s that end in .

This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side.

Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.

Using pipe symbols, the grammar above can be described more tersely as follows:

S → a | aS | bS

Matching pairs

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

S → aSb

S → ab

This grammar generates the language , which is not regular (according to the pumping lemma for regular languages).

The special character ε stands for the empty string. By changing the above grammar to

S → aSb | ε

we obtain a grammar generating the language instead. This differs only in that it contains the empty string while the original grammar did not.

Algebraic expressions

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → (S)

This grammar can, for example, generate the string

(x + y) * x - z * y / (x + x)

as follows:

S (the start symbol)

→ S - S (by rule 5)

→ S * S - S (by rule 6, applied to the leftmost S)

→ S * S - S / S (by rule 7, applied to the rightmost S)

→ (S) * S - S / S (by rule 8, applied to the leftmost S)

→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)

→ (S + S) * S - S / (S) (etc.)

→ (S + S) * S - S * S / (S)

→ (S + S) * S - S * S / (S + S)

→ (x + S) * S - S * S / (S + S)

→ (x + y) * S - S * S / (S + S)

→ (x + y) * x - S * y / (S + S)

→ (x + y) * x - S * y / (x + S)

→ (x + y) * x - z * y / (x + S)

→ (x + y) * x - z * y / (x + x)

Note that many choices were made underway as to which rewrite was going to be performed next. These choices look quite arbitrary. As a matter of fact, they are, in the sense that the string finally generated is always the same. For example, the second and third rewrites

→ S * S - S (by rule 6, applied to the leftmost S)

→ S * S - S / S (by rule 7, applied to the rightmost S)

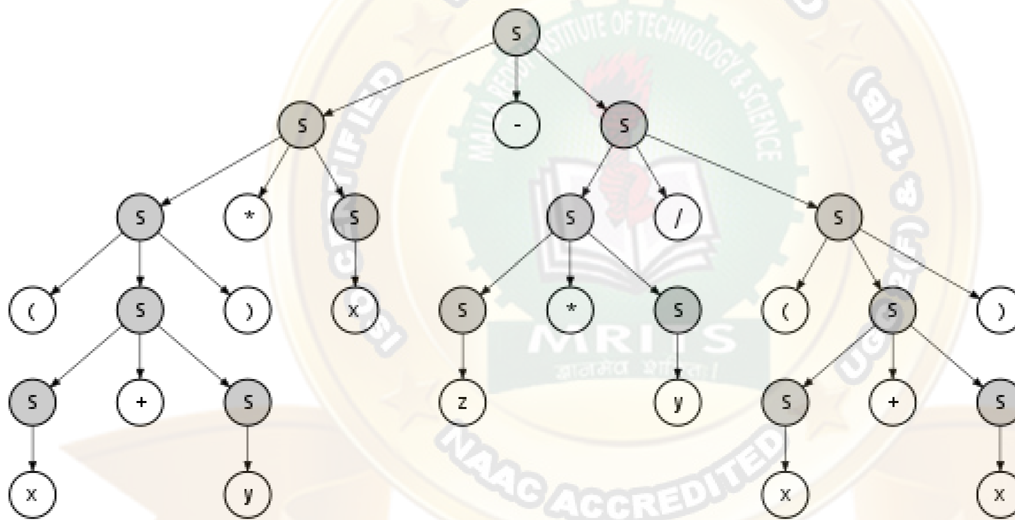
could be done in the opposite order:

→ S - S / S (by rule 7, applied to the rightmost S)

→ S * S - S / S (by rule 6, applied to the leftmost S)

Also, many choices were made on which rule to apply to each selected `S`. Changing the choices made and not only the order they were made in usually affects which terminal string comes out at the end.

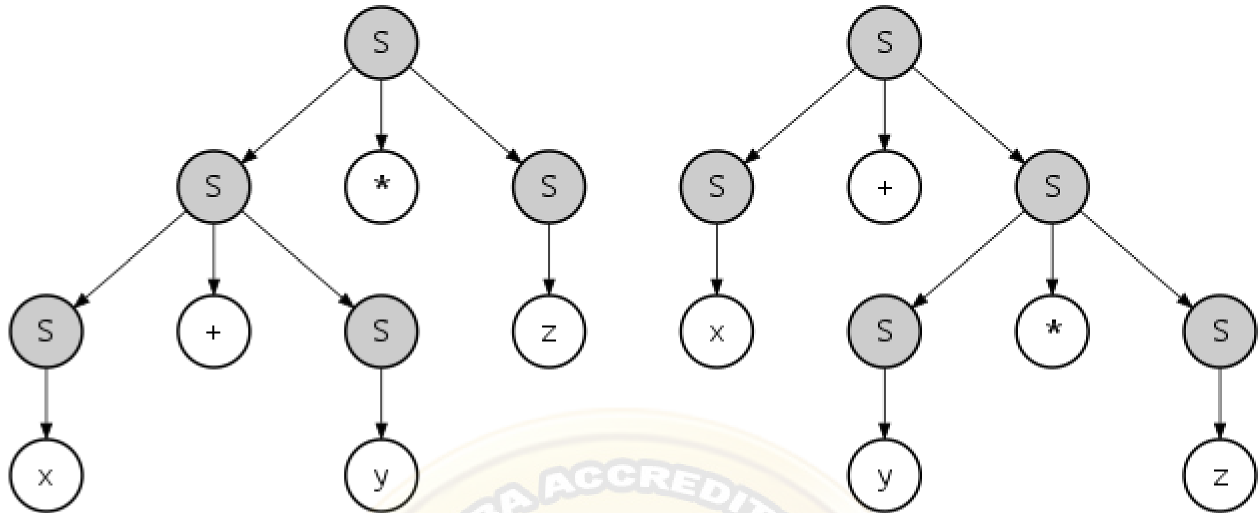
Let's look at this in more detail. Consider the parse tree of this derivation:



Starting at the top, step by step, an S in the tree is expanded, until no more unexpanded `S`es (nonterminals) remain. Picking a different order of expansion will produce a different derivation, but the same parse tree. The parse tree will only change if we pick a different rule to apply at some position in the tree.

But can a different parse tree still produce the same terminal string, which is `( x + y ) * x - z * y / ( x + x )` in this case? Yes, for this particular grammar, this is possible. Grammars with this property are called ambiguous.

For example, `x + y * z` can be produced with these two different parse trees:



However, the *language* described by this grammar is not inherently ambiguous: an alternative, unambiguous grammar can be given for the language, for example:

T → x

T → y

T → z

S → S + T

S → S - T

S → S * T

S → S / T

T → (S)

S → T

(once again picking `S` as the start symbol). This alternative grammar will produce `x + y * z` with a parse tree similar to the left one above, i.e. implicitly assuming the association `(x + y) * z`, which is not according to standard operator precedence. More elaborate, unambiguous and context-free grammars can be constructed that produce parse trees that obey all desired operator precedence and associativity rules.

Further examples

Example 1

A context-free grammar for the language consisting of all strings over {a,b} containing an unequal number of a's and b's:

S → U | V

U → TaU | TaT | UaT

V → TbV | TbT | VbT

T → aTbT | bTaT | ε

Here, the nonterminal T can generate all strings with the same number of a's as b's, the nonterminal U generates all strings with more a's than b's and the nonterminal V generates all strings with fewer a's than b's. Omitting the third alternative in the rule for U and V doesn't restrict the grammar's language.

Example 2

Another example of a non-regular language is . It is context-free as it can be generated by the following context-free grammar:

S → bSbb | A

A → aA | ε

Other examples

The formation rules for the terms and formulas of formal logic fit the definition of context-free grammar, except that the set of symbols may be infinite and there may be more than one start symbol.

Derivations and syntax trees

A *derivation* of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

A derivation is fully determined by giving, for each step:

- the rule applied in that step
- the occurrence of its left-hand side to which it is applied

For clarity, the intermediate string is usually given as well.

For instance, with the grammar:

(1) S → S + S
(2) S → 1
(3) S → a

the string

1 + 1 + a

can be derived with the derivation:

S
→ (rule 1 on the first S)
S+S
→ (rule 1 on the second S)
S+S+S
→ (rule 2 on the second S)
S+1+S
→ (rule 3 on the third S)
S+1+a
→ (rule 2 on the first S)
1+1+a

Often, a strategy is followed that deterministically determines the next nonterminal to rewrite:

- in a *leftmost derivation*, it is always the leftmost nonterminal;
- in a *rightmost derivation*, it is always the rightmost nonterminal.

Given such a strategy, a derivation is completely determined by the sequence of rules applied. For instance, the leftmost derivation

S
→ (rule 1 on the first S)
S+S
→ (rule 2 on the first S)
1+S
→ (rule 1 on the first S)
1+S+S
→ (rule 2 on the first S)
1+1+S
→ (rule 3 on the first S)
1+1+a

can be summarized as

rule 1, rule 2, rule 1, rule 2, rule 3

The distinction between leftmost derivation and rightmost derivation is important because in most parsers the transformation of the input is defined by giving a piece of code for every grammar rule that is executed whenever the rule is applied. Therefore, it is important to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the pieces of code will be executed. See for an example LL parsers and LR parsers.

A derivation also imposes in some sense a hierarchical structure on the string that is derived. For example, if the string "1 + 1 + a" is derived according to the leftmost derivation:

S → S + S (1)

→ 1 + S (2)

→ 1 + S + S (1)

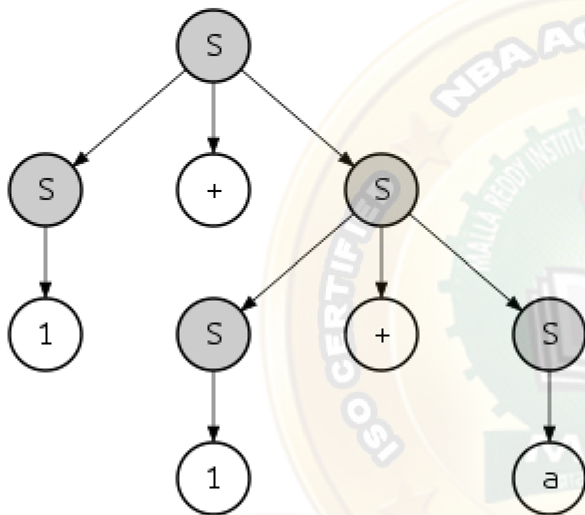
→ 1 + 1 + S (2)

→ 1 + 1 + a (3)

the structure of the string would be:

$\{ \{ 1 \}_S + \{ \{ 1 \}_S + \{ a \}_S \}_S \}_S$

where $\{ ... \}_S$ indicates a substring recognized as belonging to S. This hierarchy can also be seen as a tree:



This tree is called a *parse tree* or "concrete syntax tree" of the string, by contrast with the abstract syntax tree. In this case the presented leftmost and the rightmost derivations define the same parse tree; however, there is another (rightmost) derivation of the same string

S → S + S (1)

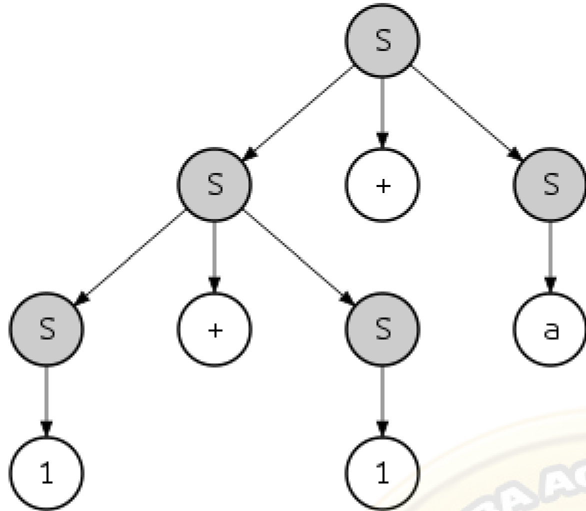
→ S + a (3)

→ S + S + a (1)

→ S + 1 + a (2)

→ 1 + 1 + a (2)

and this defines the following parse tree:



Note however that both parse trees can be obtained by both leftmost and rightmost derivations. For example, the last tree can be obtained with the leftmost derivation as follows:

S → S + S (1)

→ S + S + S (1)

→ 1 + S + S (2)

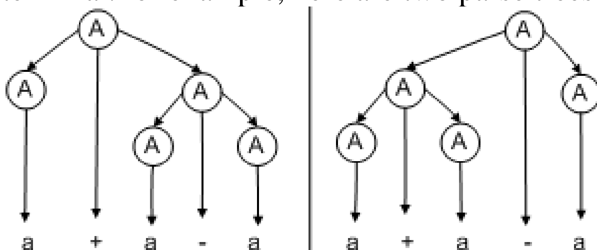
→ 1 + 1 + S (2)

→ 1 + 1 + a (3)

If a string in the language of the grammar has more than one parsing tree, then the grammar is said to be an *ambiguous grammar*. Such grammars are usually hard to parse because the parser cannot always decide which grammar rule it has to apply. Usually, ambiguity is a feature of the grammar, not the language, and an unambiguous grammar can be found that generates the same context-free language. However, there are certain languages that can only be generated by ambiguous grammars; such languages are called *inherently ambiguous languages*.

Leftmost and Rightmost Derivations:

Given a derivation tree for a word, you can "implement" it as a sequence of productions in many different ways. The *leftmost* derivation is the one in which you always expand the leftmost non-terminal. The *rightmost* derivation is the one in which you always expand the rightmost non-terminal.For example, here are two parse trees borrowed from



The leftmost derivation corresponding to the left parse tree is

$$A \rightarrow A+A \rightarrow a+A \rightarrow a+A-A \rightarrow a+a-A \rightarrow a+a-a$$

The rightmost derivation corresponding to the left parse tree is

$$A \rightarrow A+A \rightarrow A+A-A \rightarrow A+A-a \rightarrow A+a-a \rightarrow a+a-a$$

The leftmost derivation corresponding to the right parse tree is

$$A \rightarrow A-A \rightarrow A+A-A \rightarrow a+A-A \rightarrow a+a-A \rightarrow a+a-a$$

The rightmost derivation corresponding to the right parse tree is

$$A \rightarrow A-A \rightarrow A-a \rightarrow A+A-a \rightarrow A+a-a \rightarrow a+a-a$$

Leftmost and Rightmost Derivations

At any stage during a parse, when we have derived some sentential form (that is not yet a sentence) we will potentially have two choices to make:

1. which non-terminal in the sentential form to apply a production rule to
2. which production rule for that non-terminal to apply

Eg. in the above example, when we derived $E\ OP\ E$, we could then have applied a production rule to any of these three non-terminals, and would then have had to choose among all the production rules for either E or OP.

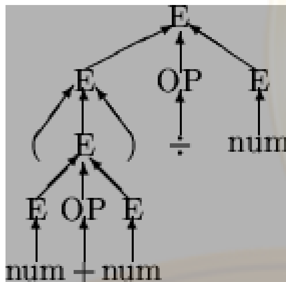
The first decision here is relatively easy to solve: we will be reading the input string from left to right, so it is our own interest to derive the leftmost terminal of the resulting sentence as soon as possible. Thus, in a top-down parse we always choose the leftmost non-terminal in a sentential form to apply a production rule to - this is called a **leftmost derivation**.

If we were doing a bottom-up parse then the situation would be reversed, and we would want to do apply the production rules in reverse to the leftmost symbols; thus we are performing a **rightmost derivation** in reverse.

For example, a bottom-up rightmost derivation would look like:

Rule Applied (in reverse)	*Sent. Form*
	(**num** + **num**) $\div$ **num**
$E \rightarrow$ **num**	(E +**num**) $\div$ **num**
$OP \rightarrow +$	(E OP **num**) $\div$ **num**
$E \rightarrow$ **num**	(E OP E) $\div$ **num**
$E \rightarrow E\ OP\ E$	(E) $\div$ **num**
$E \rightarrow (E)$	E $\div$ **num**
$OP \rightarrow \div$	E OP **num**
$E \rightarrow$ **num**	E OP E
$E \rightarrow E\ OP\ E$	E

Note that this has no effect on the parse tree; we still get:
=1.00mm



THE LANGUAGE OF A GRAMMAR:

A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts. Therefore, a In formal language theory, a **grammar** (when the context is not given, often called a **formal grammar** for clarity) is a set of production rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context—only their form.

Formal language theory, the discipline that studies formal grammars and languages, is a branch of applied mathematics. Its applications are found in theoretical computer science, theoretical linguistics, formal semantics, mathematical logic, and other areas.

grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer"—a function in computing that determines whether a given string belongs to the language or is grammatically incorrect. To describe such recognizers, formal language theory uses separate formalisms, known as automata theory. One of the interesting results

of automata theory is that it is not possible to design a recognizer for certain formal languages.[1] Parsing is the process of recognizing an utterance (a string in natural languages) by breaking it down to a set of symbols and analyzing each one against the grammar of the language. Most languages have the meanings of their utterances structured according to their syntax—a practice known as compositional semantics. As a result, the first step to describing the meaning of an utterance in language is to break it down part by part and look at its analyzed form (known as its parse tree in computer science, and as its deep structure in generative grammar).

Introductory example

A grammar mainly consists of a set of rules for transforming strings. (If it *only* consisted of these rules, it would be a semi-Thue system.) To generate a string in the language, one begins with a string consisting of only a single *start symbol*. The *production rules* are then applied in any order, until a string that contains neither the start symbol nor designated *nonterminal symbols* is produced. A production rule is applied to a string by replacing one occurrence of the production rule's left-hand side in the string by that production rule's right-hand side (*cf.* the operation of the theoretical Turing machine). The language formed by the grammar consists of all distinct strings that can be generated in this manner. Any particular sequence of production rules on the start symbol yields a distinct string in the language. If there are essentially different ways of generating the same single string, the grammar is said to be ambiguous.

For example, assume the alphabet consists of *a* and *b*, the start symbol is *S*, and we have the following production rules:

1. S→aSb.
2. S→ba.

then we start with *S*, and can choose a rule to apply to it. If we choose rule 1, we obtain the string *aSb*. If we then choose rule 1 again, we replace *S* with *aSb* and obtain the string *aaSbb*. If we now choose rule 2, we replace *S* with *ba* and obtain the string *aababb*, and are done. We can write this series of choices more briefly, using symbols: . The language of the grammar is then the infinite set , where is repeated times (and in particular represents the number of times production rule 1 has been applied).

Sentential Form

A *sentential form* is any string derivable from the start symbol. Thus, in the derivation of *a* + *a* * *a* , E + T * F and E + F * a and F + a * a are all sentential forms as are E and *a* + *a* * *a* themselves.

■**Sentence**

A *sentence* is a sentential form consisting only of terminals such as *a* + *a* * *a*. A sentence can be derived using the following algorithm:

Algorithm

Derive String

```
String := Start Symbol
REPEAT
  Choose any nonterminal in String.
  Find a production with this nonterminal on the left-hand side.
  Replace the nonterminal with one of the options on the right-hand
  side of the production.
UNTIL String contains only terminals.
```

Parse Tress:

A **parse tree** or **parsing tree**[or **derivation tree** or **concrete syntax tree** is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. The term *parse tree* itself is used primarily in computational linguistics; in theoretical syntax, the term *syntax tree* is more common.

Parse trees concretely[*clarification needed*] reflect the syntax of the input language, making them distinct from the abstract syntax trees used in computer programming. Unlike Reed-Kellogg sentence diagrams used for teaching grammar, parse trees do not use distinct symbol shapes for different types of constituents.

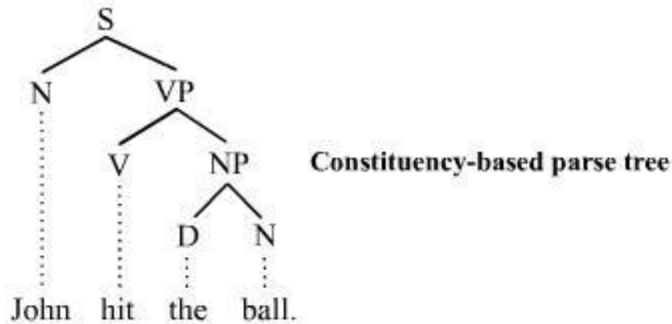
Parse trees are usually constructed based on either the constituency relation of constituency grammars (phrase structure grammars) or the dependency relation of dependency grammars. Parse trees may be generated for sentences in natural languages (see natural language processing), as well as during processing of computer languages, such as programming languages.

A related concept is that of **phrase marker** or **P-marker**, as used in transformational generative grammar. A phrase marker is a linguistic expression marked as to its phrase structure. This may be presented in the form of a tree, or as a bracketed expression. Phrase markers are generated by applying phrase structure rules, and Themselves are subject to further transformational rules.

MRITS

Constituency-based parse trees

The constituency-based parse trees of constituency grammars (= phrase structure grammars) distinguish between terminal and non-terminal nodes. The interior nodes are labeled by non-terminal categories of the grammar, while the leaf nodes are labeled by terminal categories. The image below represents a constituency-based parse tree; it shows the syntactic structure of the English sentence *John hit the ball*:



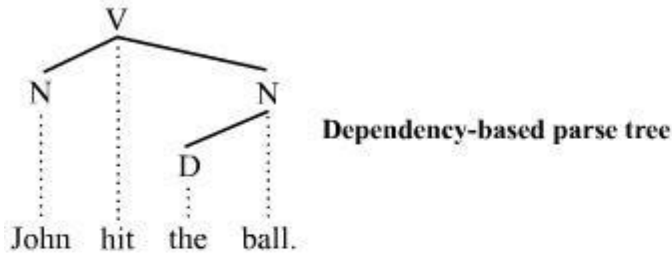
The parse tree is the entire structure, starting from S and ending in each of the leaf nodes (*John*, *hit*, *the*, *ball*). The following abbreviations are used in the tree:

- S for sentence, the top-level structure in this example
- NP for noun phrase. The first (leftmost) NP, a single noun "John", serves as the subject of the sentence. The second one is the object of the sentence.
- VP for verb phrase, which serves as the predicate
- V for verb. In this case, it's a transitive verb *hit*.
- D for determiner, in this instance the definite article "the"
- N for noun

Each node in the tree is either a *root* node, a *branch* node, or a *leaf* node.[2] A root node is a node that doesn't have any branches on top of it. Within a sentence, there is only ever one root node. A branch node is a mother node that connects to two or more daughter nodes. A leaf node, however, is a terminal node that does not dominate other nodes in the tree. S is the root node, NP and VP are branch nodes, and *John* (N), *hit* (V), *the* (D), and *ball* (N) are all leaf nodes. The leaves are the lexical tokens of the sentence.[3] A mother node is one that has at least one other node linked by a branch under it. In the example, S is a parent of both N and VP. A daughter node is one that has at least one node directly above it to which it is linked by a branch of a tree. From the example, *hit* is a daughter node of V. The terms *parent* and *child* are also sometimes

Dependency-based parse trees

The dependency-based parse trees of dependency grammars[4] see all nodes as terminal, which means they do not acknowledge the distinction between terminal and non-terminal categories. They are simpler on average than constituency-based parse trees because they contain fewer nodes. The dependency-based parse tree for the example sentence above is as follows:



This parse tree lacks the phrasal categories (S, VP, and NP) seen in the constituency-based counterpart above. Like the constituency-based tree, constituent structure is acknowledged. Any complete sub-tree of the tree is a constituent. Thus this dependency-based parse tree acknowledges the subject noun *John* and the object noun phrase *the ball* as constituents just like the constituency-based parse tree does.

The constituency vs. dependency distinction is far-reaching. Whether the additional syntactic structure associated with constituency-based parse trees is necessary or beneficial is a matter of debate.

Applications of Context-Free Grammars:

in formal language theory, a **context-free grammar** (**CFG**) is a certain type of formal grammar: a set of that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the rule

replaces with . There can be multiple replacement rules for any given value. For example,

means that can be replaced with either or .In context-free grammars, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is lways a nonterminal symbol. This means that the symbol does not appear in the resulting formal language. So in our case, our language contains the letters and but not Rules can also be applied in reverse to check if a string is grammatically correct according to the grammar.Here is an example context-free grammar that describes all two-letter strings containing the letters and .

If we start with the nonterminal symbol then we can use the rule to turn into . We can then apply one of the two later rules. For example, if we apply to the first we get . If we then apply to the

second we get . Since both and are terminal symbols, and in context-free grammars terminal symbols never appear on the left hand side of a production rule, there are no more rules that can be applied. This same process can be used, applying the second two rules in different orders in order to get all possible strings within our simple context-free grammar.

Languages generated by context-free grammars are known as context-free languages (CFL). Different context-free grammars can generate the same context-free language. It is important to distinguish the properties of the language (intrinsic properties) from the properties of a particular grammar (extrinsic properties). The language equality question (do two given context-free grammars generate the same language?) is undecidable.

Context-free grammars arise in linguistics where they are used to describe the structure of sentences and words in a natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose, but have not really lived up to their original expectation. By contrast, in computer science, as the use of recursively-defined concepts increased, they were used more and more. In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the *Document Type Definition*.[2]

In linguistics, some authors use the term **phrase structure grammar** to refer to context-free grammars, whereby phrase-structure grammars are distinct from dependency grammars. In computer science, a popular notation for context-free grammars is Backus–Naur form, or *BNF*.

Ambiguity in Grammars and Languages:

If a context free grammar **G** has more than one derivation tree for some string **w ∈ L(G)**, it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

Check whether the grammar G with production rules −

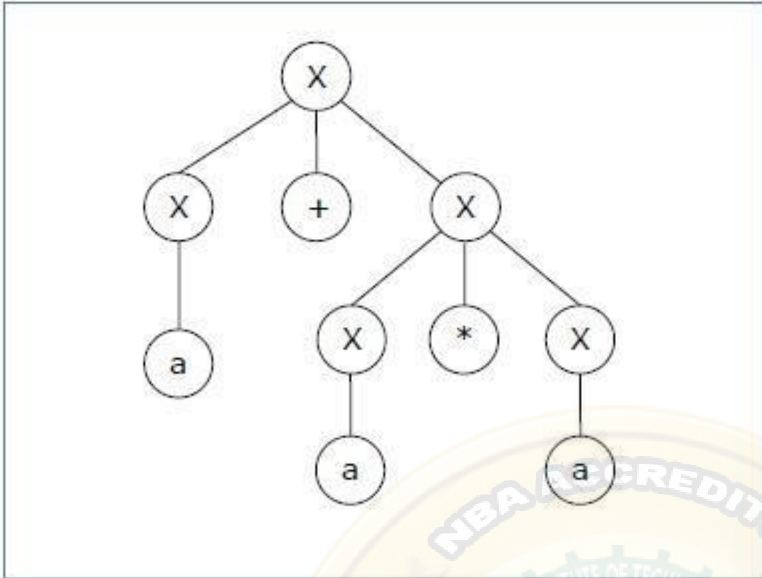
X → X+X | X*X |X| a is ambiguous or not.

Solution

Let’s find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

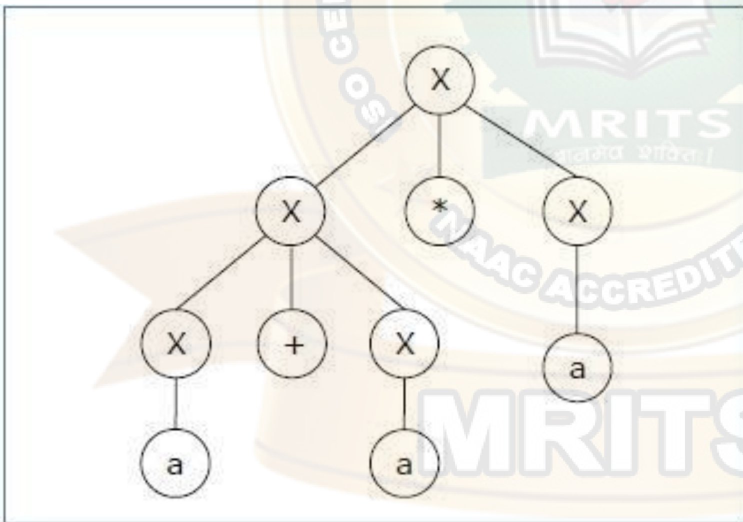
Derivation 1 − X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 −



Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 −



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

Push Down Automata:

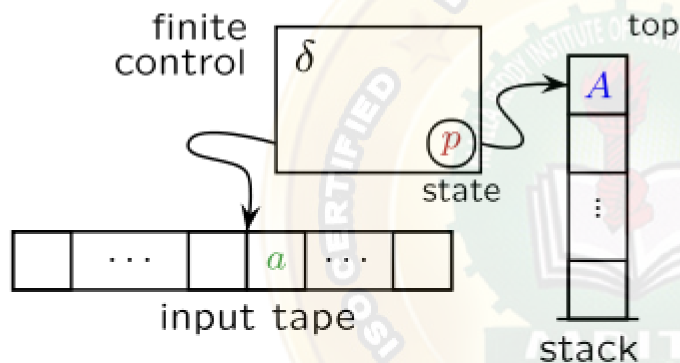
In computer science, a **pushdown automaton** (**PDA**) is a type of automaton that employs a stack.

Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines. Deterministic

pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages, with the former often used in parser design.

The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the top element. A **stack automaton**, by contrast, does allow access to and operations on deeper elements. Stack automata can recognize a strictly larger set of languages than pushdown automata. A nested stack automaton allows full access, and also allows stacked values to be entire sub-stacks rather than just single finite symbols.

Informal description



A diagram of a pushdown automaton

A finite state machine just looks at the input signal and the current state: it has no stack to work with. It chooses a new state, the result of following the transition. A **pushdown automaton (PDA)** differs from a finite state machine in two ways:

1. It can use the top of the stack to decide which transition to take.
2. It can manipulate the stack as part of performing a transition.

A pushdown automaton reads a given input string from left to right. In each step, it chooses a transition by indexing a table by input symbol, current state, and the symbol at the top of the stack. A pushdown automaton can also manipulate the stack, as part of performing a transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the top of the stack. The automaton can alternatively ignore the stack, and leave it as it is.

Put together: Given an input symbol, current state, and stack symbol, the automaton can follow a transition to another state, and optionally manipulate (push or pop) the stack.

If, in every situation, at most one such transition action is possible, then the automaton is called a **deterministic pushdown automaton (DPDA)**. In general, if several actions are possible, then the automaton is called a **general**, or **nondeterministic**, **PDA**. A given input string may drive a

nondeterministic pushdown automaton to one of several configuration sequences; if one of them leads to an accepting configuration after reading the complete input string, the latter is said to belong to the *language accepted by the automaton*.

Definition of the Pushdown Automaton:

Formal definition

We use standard formal language notation: denotes the set of strings over alphabet and denotes the empty string.

A PDA is formally defined as a 7-tuple:

where is a finite set of *states*

- is a finite set which is called the *input alphabet*
- is a finite set which is called the *stack alphabet*
- is a finite subset of , the *transition relation*.
- is the *start state*
- is the *initial stack symbol*
- is the set of *accepting states*

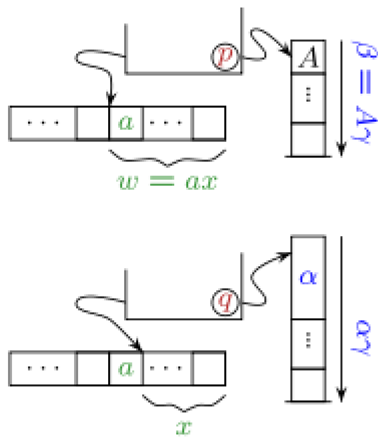
An element is a transition of . It has the intended meaning that , in state , on the input and with as topmost stack symbol, may read , change the state to , pop , replacing it by pushing . The component of the transition relation is used to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched.

In many texts the transition relation is replaced by an (equivalent) formalization, where

- is the *transition function*, mapping into finite subsets of

Here contains all possible actions in state with on the stack, while reading on the input. One writes for example precisely when because . Note that *finite* in this definition is essential.

Computations



a step of the pushdown automaton

In order to formalize the semantics of the pushdown automaton a description of the current situation is introduced. Any 3-tuple is called an instantaneous description (ID) of , which includes the current state, the part of the input tape that has not been read, and the contents of the stack (topmost symbol written first). The transition relation defines the step-relation of on instantaneous descriptions. For instruction there exists a step , for every and every .In general pushdown automata are nondeterministic meaning that in a given instantaneous description there may be several possible steps. Any of these steps can be chosen in a computation. With the above definition in each step always a single symbol (top of the stack) is popped, replacing it with as many symbols as necessary. As a consequence no step is defined when the stack is empty.Computations of the pushdown automaton are sequences of steps. The computation starts in the initial state with the initial stack symbol on the stack, and a string on the input tape, thus with initial description . There are two modes of accepting. The pushdown automaton either accepts by final state, which means after reading its input the automaton reaches an accepting state (in), or it accepts by empty stack (), which means after reading its input the automaton empties its stack. The first acceptance mode uses the internal memory (state), the second the external memory (stack).

Formally one defines

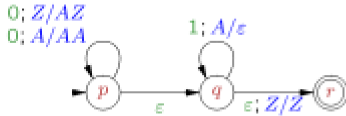
1. with and (final state)
2. with (empty stack)

Here represents the reflexive and transitive closure of the step relation meaning any number of consecutive steps (zero, one or more).For each single pushdown automaton these two languages need to have no relation: they may be equal but usually this is not the case. A specification of the automaton should also include the intended mode of acceptance. Taken over all pushdown automata both acceptance conditions define the same family of languages.

Theorem. For each pushdown automaton one may construct a pushdown automaton such that , and vice versa, for each pushdown automaton one may construct a pushdown automaton such that

Example

The following is the formal description of the PDA which recognizes the language by final state:



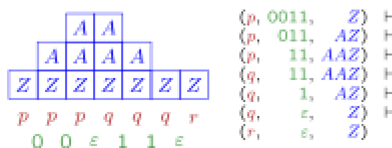
PDA for
(by final state), where

- **states:**
- **input alphabet:**
- **stack alphabet:**
- **start state:**
- **start stack symbol:** *Z*
- **accepting states:**

The transition relation consists of the following six instructions:,,,,, and.In words, the first two instructions say that in state *p* any time the symbol 0 is read, one *A* is pushed onto the stack. Pushing symbol *A* on top of another *A* is formalized as replacing top *A* by *AA* (and similarly for pushing symbol *A* on top of a *Z*).The third and fourth instructions say that, at any moment the automaton may move from state *p* to state *q*.The fifth instruction says that in state *q*, for each symbol 1 read, one *A* is popped.

Finally, the sixth instruction says that the machine may move from state *q* to accepting state *r* only when the stack consists of a single *Z*.There seems to be no generally used representation for PDA. Here we have depicted the instruction by an edge from state *p* to state *q* labelled by (read *a*; replace *A* by).

Understanding the computation process



Accepting computation for 0011

The following illustrates how the above PDA computes on different input strings. The subscript *M* from the step symbol is here omitted.

a. Input string = 0011. There are various computations, depending on the moment the move from state *p* to state *q* is made. Only one of these is accepting.
    i. The final state is accepting, but the input is not accepted this way as it has not been read.
    ii. No further steps possible.
    iii. Accepting computation: ends in accepting state, while complete input has been read.
b. Input string = 00111. Again there are various computations. None of these is accepting.
    i. The final state is accepting, but the input is not accepted this way as it has not been read.
    ii. No further steps possible.
    iii. The final state is accepting, but the input is not accepted this way as it has not been (completely) read.

Description

A pushdown automaton (PDA) is a finite state machine which has an additional stack storage. The transitions a machine makes are based not only on the input and current state, but also on the stack. The formal definition (in our textbook) is that a PDA is this:

M = (K,Σ,Γ,Δ,s,F) where K = finite state set

- Σ = finite input alphabet
- Γ = finite stack alphabet
- s ∈ K: start state
- F ⊆ K: final states
- The transition relation, Δ is a **finite** subset of $(K\times(\Sigma \cup \{\varepsilon\})\times\Gamma^*) \times (K\times\Gamma^*)$

We have to have the **finite** qualifier because the full subset is infinite by virtue of the Γ* component. The meaning of the transition relation is that, for σ ∈ Σ, if ((p,σ,α),(q,β)) ∈ Δ:

- the current state is p
- the current input symbol is σ
- the string at the top of the stack is α then:

- the new state is q
- replace α on the top of the stack by β (pop the α and push the β)

Otherwise, if ((p,ε,α),(q,β)) ∈ Δ, this means that if

- the current state is p
- the string at the top of the stack is α then (not consulting the input symbol), we can

- change the state is q

- replace α on the top of the stack by β

Machine Configuration, yields, acceptance
A machine configuration is an element of $K \times \Sigma^* \times \Gamma^*$.

(p,w,γ) = current state, unprocessed input, stack content)
We define the usual *yields* relationships:

$(p,\sigma w,\alpha\gamma) \vdash (q,w,\beta\gamma)$ if $((p,\sigma,\alpha),(q,\beta)) \in \Delta$ or $(p,w,\alpha\gamma) \vdash (q,w,\beta\gamma)$ if $((p,\varepsilon,\alpha),(q,\beta)) \in \Delta$

As expected, $\vdash^*$ is the reflexive, transitive closure of $\vdash$.

A string w is *accepted* by the PDA if

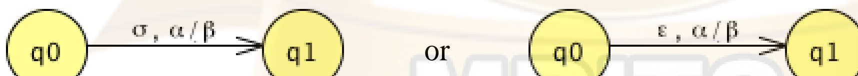
$(s,w,\varepsilon) \vdash^* (f,\varepsilon,\varepsilon)$
Namely, from the start state with empty stack, we

- process the entire string,
- end in a final state
- end with an empty stack.

The language accepted by a PDA M, L(M), is the set of all accepted strings.
The empty stack is our key new requirement relative to finite state machines. The examples that we generate have very few states; in general, there is so much more control from using the stack memory. Acceptance by empty stack only or final state only is addressed in problems 3.3.3 and 3.3.4.

Graphical Representation and ε-transition
The book does not indicate so, but there is a graphical representation of PDAs. A transition

$((p,x,\alpha),(q,\beta))$ where $x = \varepsilon$ or $x \in \Sigma$ would be depicted like this (respectively):



The stack usage represented by

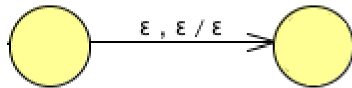
α/β
represents these actions:

- the top of the stack must **match** α
- if we make the transition, **pop** α and **push** β

A PDA is non-deterministic. There are several forms on non-determinism in the description:

- Δ is a relation
- there are ε-transitions in terms of the input
- there are ε-transitions in terms of the stack contents

The true PDA ε-transition, in the sense of being equivalent to the NFA ε-transition is this:

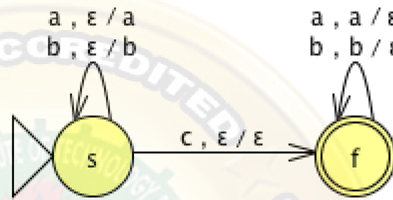


because it consults neither the input, nor the stack and will leave the previous configuration intact.

Palindrome examples

These are examples 3.3.1 and 3.3.2 in the textbook. The first is this:

$\{x \in \{a,b,c\}^* : x = wcw^R \text{ for } w \in \{a,b\}^*\}$

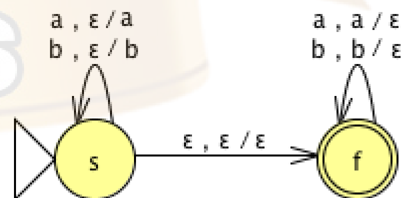


The machine pushes a's and b's in state s, makes a transition to f when it sees the middle marker, c, and then matches input symbols with those on the stack and pops the stack symbol. Non-accepting string examples are these:

ε in state s
ab in state s with non-empty stack
abcab in state f with unconsumed input and non-empty stack
abcb in state f with non-empty stack
abcbab in state f with unconsumed input and empty stack
Observe that this PDA is deterministic in the sense that there are no choices in transitions.

The second example is:

$\{x \in \{a,b\}^* : x = ww^R \text{ for } w \in \{a,b\}^*\}$



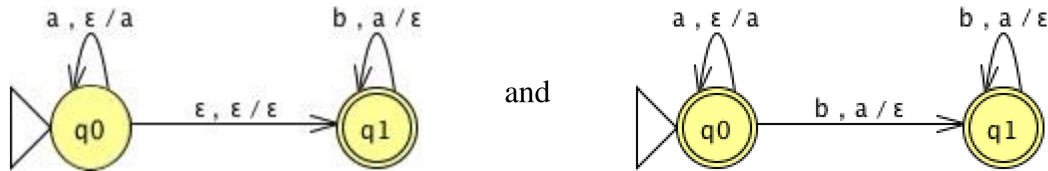
This PDA is identical to the previous one except for the ε-transition

Nevertheless, there is a significant difference in that this PDA must **guess** when to stop pushing symbols, jump to the final state and start matching off of the stack. Therefore this machine is decidedly non-deterministic. In a general programming model (like Turing Machines), we have

the luxury of preprocessing the string to determine its length and thereby knowing when the middle is coming.

The $a^n b^n$ language

The language is L = { w ∈ {a,b}* : w = $a^n b^n$, n ≥ 0 }. Here are two PDAs for L:

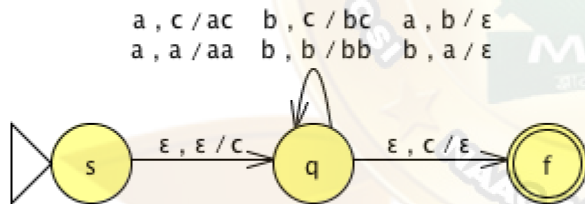


The idea in both of these machines is to stack the a's and match off the b's. The first one is non-deterministic in the sense that it could prematurely guess that the a's are done and start matching off b's. The second version is deterministic in that the first b acts as a trigger to start matching off. Note that we have to make both states final in the second version in order to accept ε.

The equal a's and b's language

This is example 3.3.3 in the textbook. Let us use the convenience notation:

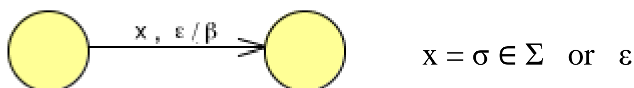
#σ(w) = the number of occurrences of σ in w
The language is L = {w ∈ {a,b}*: #a(w) = #b(w) }. Here is the PDA:



As you can see, most of the activity surrounds the behavior in state q. The idea is have the stack maintain the **excess** of one symbol over the other. In order to achieve our goal, we must know when the stack is empty.

Empty Stack Knowledge

There is no mechanism built into a PDA to determine whether the stack is empty or not. It's important to realize that the transition:



means to do so **without consulting** the stack; it says nothing about whether the stack is empty or not.

Nevertheless, one can maintain knowledge of an empty stack by using a dedicated stack symbol, c, representing the "stack bottom" with these properties:

- it is pushed onto an empty stack by a transition from the start state with no other outgoing or incoming transitions
- it is never removed except by a transition to state with no other outgoing transitions

Behavior of PDA

The three groups of loop transitions in state q represent these respective functions:

- input a with no b's on the stack: push a
- input b with no a's on the stack: push b
- input a with b's on the stack: pop b; or, input b with a's on the stack: pop a

For example if we have seen 5 b's and 3 a's in any order, then the stack should be "bbc". The transition to the final state represents the only non-determinism in the PDA in that it must guess when the input is empty in order to pop off the stack bottom.

DPDA/DCFL

The textbook defines DPDAs (Deterministic PDAs) and DCFLs (Deterministic CFLs) in the introductory part of section 3.7. According to the textbook's definition, a DPDA is a PDA in which no state p has two different outgoing transitions

$((p,x,\alpha),(q,\beta))$ and $((p,x',\alpha'),(q',\beta'))$

which are *compatible* in the sense that both could be applied. A DCFL is basically a language which accepted by a DPDA, but we need to qualify this further.

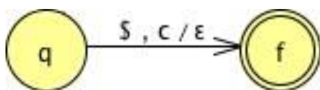
We want to argue that the language $L = \{ w \in \{a,b\}^* : \#a(w) = \#b(w)\}$ is deterministic context free in the sense there is DPDA which accepts it.

In the above PDA, the only non-determinism is the issue of guessing the end of input; however this form of non-determinism is considered artificial. When one considers whether a language L supports a DPDA or not, a dedicated *end-of-input* symbol is always added to strings in the language.

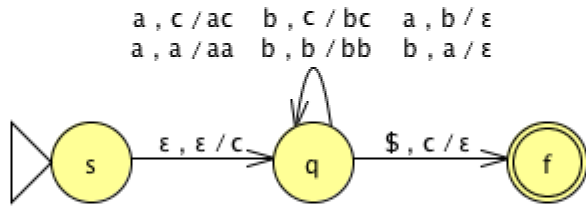
Formally, a language L over Σ is *deterministic context free*, or L is a DCFL , if

L$ is accepted by a DPDA M

where **$** is a dedicated symbol not belonging to Σ. The significance is that we can make intelligent usage of the knowledge of the end of input to decide what to do about the stack. In our case, we would simply replace the transition into the final state by:

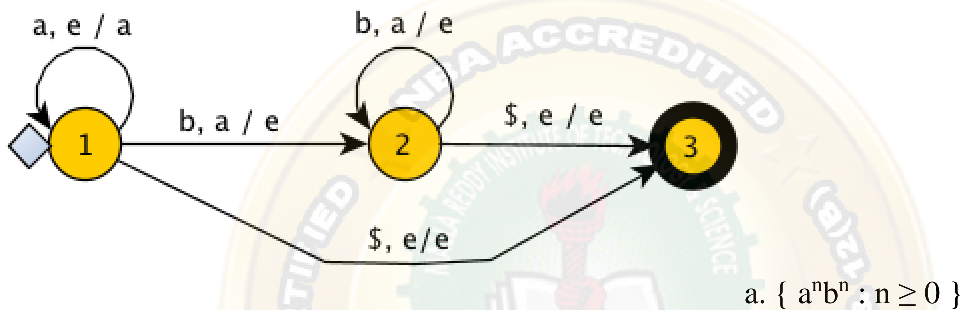


With this change, our PDA is now a DPDA:

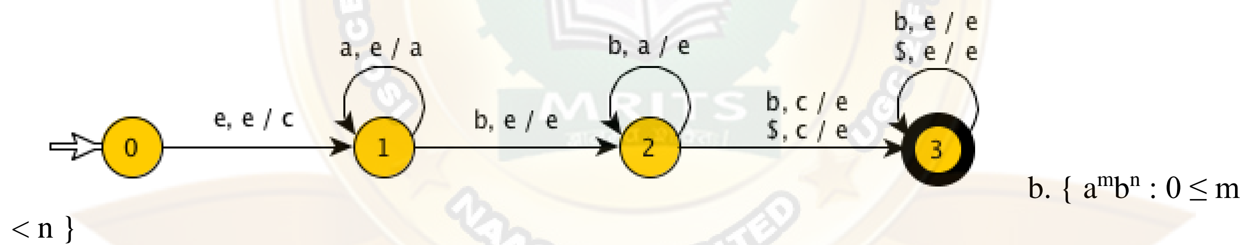


a*b* examples

Two common variations on a's followed by b's. When they're equal, no stack bottom is necessary. When they're unequal, you have to be prepared to recognize that the stacked a's have been completely matched or not.



a. { $a^n b^n : n \geq 0$ }



b. { $a^m b^n : 0 \leq m < n$ }

Let's look at a few sample runs of (b). The idea is that you cannot enter the final state with an "a" still on the stack. Once you get to the final state, you can consume remaining b's and end marker.

We can start from state 1 with the stack bottom pushed on:

success: abb

state	input	stack
1	abb$	c
1	bb$	ac
2	b$	ac
2	$	c
3	ε	ε

success: abbbb

state	input	stack
1	abbbb$	c
1	bbbb$	ac
2	bbb$	ac
2	bb$	c
3	b$	ε
3	$	ε
3	ε	ε

(fail: ab)

state	input	stack
1	ab$	c
1	b$	ac
2	$	ac

(fail: ba)

state	input	stack
1	ba$	c
2	a$	c

Observe that a string like `abbba` also fails due to the inability to consume the very last `a`.

Equivalence of PDA's and CFG's:

If a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar **G**.

Also, if **P** is a pushdown automaton, an equivalent context-free grammar G can be constructed where

L(G) = L(P)

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F)

Step 1 − Convert the productions of the CFG into GNF.

Step 2 − The PDA will have only one state {q}.

Step 3 − The start symbol of CFG will be the start symbol in the PDA.

Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 − For each production in the form **A → aX** where a is terminal and **A, X** are combination of terminal and non-terminals, make a transition **δ (q, a, A)**.

Problem

Construct a PDA from the following CFG.

G = ({S, X}, {a, b}, P, S)

where the productions are −

S → XS | ε , A → aXb | Ab | ab

Solution

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ −

δ(q, ε , S) = {(q, XS), (q, ε)}

δ(q, ε , X) = {(q, aXb), (q, Xb), (q, ab)}

δ(q, a, a) = {(q, ε)}

$\delta(q, 1, 1) = \{(q, \varepsilon)\}$

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule $X_{wx} \rightarrow a X_{yz} b$ in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule $X_{wx} \rightarrow X_{wy}X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the top-down parsing is −

(x+y*z, I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the bottom-up parsing is −

(x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

Deterministic Pushdown Automata:

In automata theory, a **deterministic pushdown automaton** (**DPDA** or **DPA**) is a variation of the pushdown automaton. The class of deterministic pushdown automata accepts the deterministic context-free languages, a proper subset of context-free languages.

Machine transitions are based on the current state and input symbol, and also the current topmost symbol of the stack. Symbols lower in the stack are not visible and have no immediate effect. Machine actions include pushing, popping, or replacing the stack top. A deterministic pushdown automaton has at most one legal transition for the same combination of input symbol, state, and top stack symbol. This is where it differs from the nondeterministic pushdown automaton.

Formal definition

A (not necessarily deterministic) **PDA** can be defined as a 7-tuple:

Where is a finite set of states

- is a finite set of input symbols
- is a finite set of stack symbols
- is the start state
- is the starting stack symbol
- , where is the set of accepting states
- is a transition function, wherewhere is the Kleene star, meaning that is "the set of all finite strings (including the empty string) of elements of ", denotes the empty string, and is the power set of a set .*M* is *deterministic* if it satisfies both the following conditions:

- For any , the set has at most one element.

- For any , if , then for every

There are two possible acceptance criteria: acceptance by *empty stack* and acceptance by *final state*. The two are not equivalent for the deterministic pushdown automaton (although they are for the non-deterministic pushdown automaton). The languages accepted by *empty stack* are those languages that are accepted by *final state* and are prefix-free: no word in the language is the prefix of another word in the language.

The usual acceptance criterion is *final state*, and it is this acceptance criterion which is used to define the deterministic context-free languages.

Languages recognized

If is a language accepted by a PDA , it can also be accepted by a DPDA if and only if there is a single computation from the initial configuration until an accepting one for all strings belonging to . If can be accepted by a PDA it is a context free language and if it can be accepted by a DPDA it is a deterministic context-free language.

Not all context-free languages are deterministic. This makes the DPDA a strictly weaker device than the PDA. For example, the language of even-length palindromes on the alphabet of 0 and 1

has the context-free grammar $S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$. An arbitrary string of this language cannot be parsed without reading all its letters first which means that a pushdown automaton has to try alternative state transitions to accommodate for the different possible lengths of a semi-parsed string.

Restricting the DPDA to a single state reduces the class of languages accepted to the LL(1) languages. In the case of a PDA, this restriction has no effect on the class of languages accepted.

MALL REDDY INSTITUTE OF TECHNOLOGY & SCIENCE AND SCIENCE

LECTURE NOTES

On

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE/IT I-Sem

(Jntuh-R18)

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE I-Sem **L T P C**

3 0 0 3

Course Objectives

1. To provide introduction to some of the central ideas of theoretical computer science from the perspective of formal languages.
2. To introduce the fundamental concepts of formal languages, grammars and automata theory.
3. Classify machines by their power to recognize languages.
4. Employ finite state machines to solve problems in computing.
5. To understand deterministic and non-deterministic machines.
6. To understand the differences between decidability and undecidability.

Course Outcomes

1. Able to understand the concept of abstract machines and their power to recognize the languages.
2. Able to employ finite state machines for modeling and solving computing problems.
3. Able to design context free grammars for formal languages.
4. Able to distinguish between decidability and undecidability.

UNIT - I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory – Alphabets, Strings, Languages, Problems.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines

UNIT - II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages, Statement of the pumping lemma, Applications of the Pumping Lemma.

Closure Properties of Regular Languages: Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

UNIT - III

Context-Free Grammars: Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages. **Push Down Automata**: Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG

UNIT - IV

Normal Forms for Context- Free Grammars: Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form Griebech Normal form.

Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications **Closure Properties of Context-Free Languages**: Closure properties of CFL's, Decision Properties of CFL's

Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

UNIT - V

Types of Turing machine: Turing machines and halting

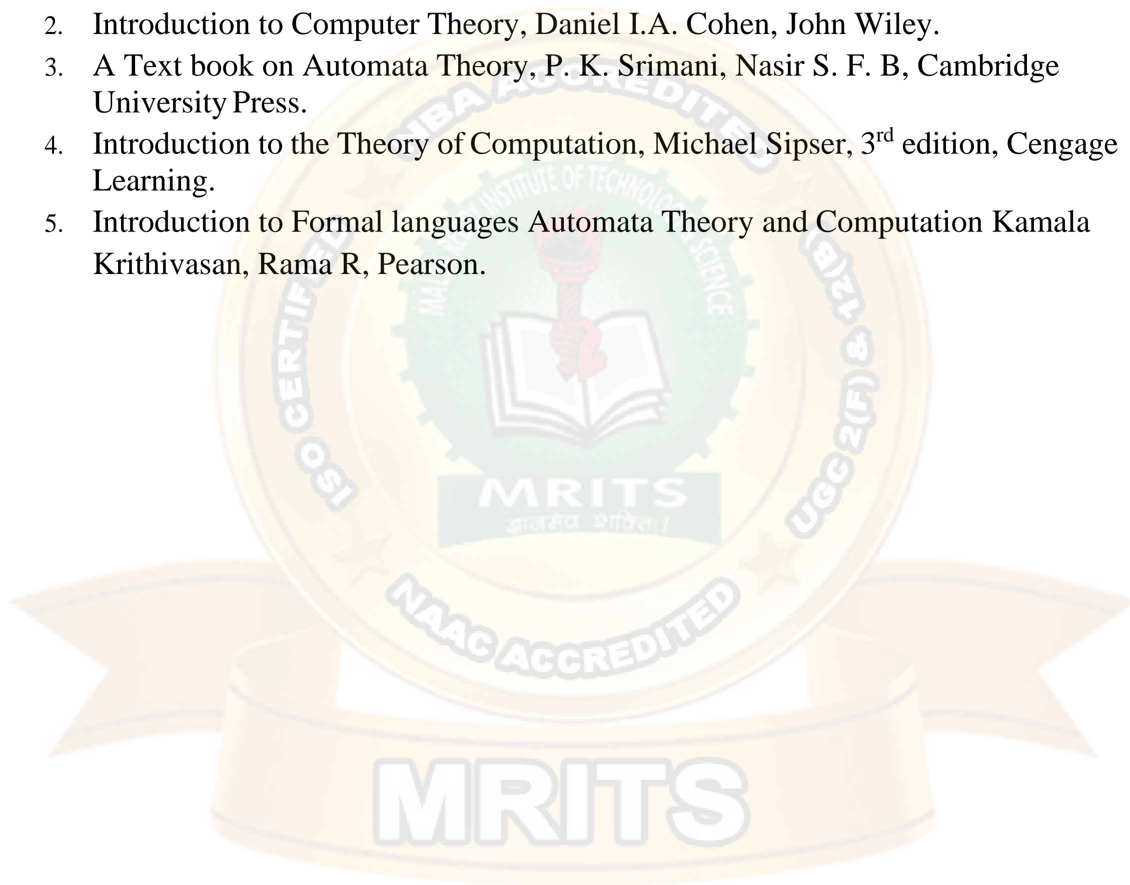
Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem, Other Undecidable Problems, Counter ma

TEXT BOOKS:

1. Introduction to Automata Theory, Languages, and Computation, 3nd Edition, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science – Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCE BOOKS:

1. Introduction to Languages and The Theory of Computation, John C Martin, TMH.
2. Introduction to Computer Theory, Daniel I.A. Cohen, John Wiley.
3. A Text book on Automata Theory, P. K. Srimani, Nasir S. F. B, Cambridge University Press.
4. Introduction to the Theory of Computation, Michael Sipser, 3rd edition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and Computation Kamala Krithivasan, Rama R, Pearson.



UNIT-IV

Normal Forms of Context-Free Grammars

Chomsky Normal Form

A grammar is in *Chomsky Normal Form* if all productions are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where A, B, and C are variables and a is a terminal. Any context-free grammar that does not contain λ can be put into Chomsky Normal Form.

(Most textbook authors also allow the production $S \rightarrow \lambda$ so long as S does not appear on the right hand side of any production.)

Chomsky Normal Form is particularly useful for programs that have to manipulate grammars.

Greibach Normal Form

A grammar is in *Greibach Normal Form* if all productions are of the form

$$A \rightarrow ax$$

where a is a terminal and $x \in V^*$.

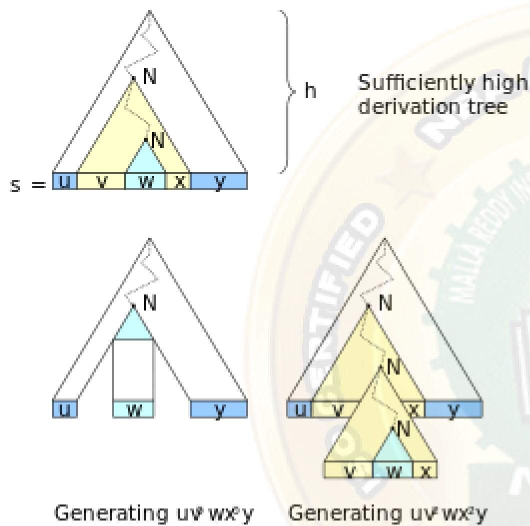
Grammars in Greibach Normal Form are typically ugly and much longer than the cfg from which they were derived. Greibach Normal Form is useful for proving the equivalence of cfgs and npdas. When we discuss converting a cfg to an npda, or vice versa, we will use Greibach Normal Form.

Pumping lemma for context-free languages

In computer science, in particular in formal language theory, the **pumping lemma for context-free languages**, also known as the **Bar-Hillel**[clarification needed] **lemma**, is a lemma that gives a

property shared by all context-free languages and generalizes the pumping lemma for regular languages. As the pumping lemma does not suffice to guarantee that a language is context-free there are more stringent necessary conditions, such as Ogden's lemma.

Formal statement



Proof idea: If *s* is sufficiently long, its derivation tree w.r.t. a Chomsky normal form grammar must contain some nonterminal *N* twice on some tree path (upper picture). Repeating *n* times the derivation part $N \Rightarrow ... \Rightarrow vNx$ obtains a derivation for uv^nwx^ny (lower left and right picture for n=0 and 2, respectively).

If a language *L* is context-free, then there exists some integer $p \geq 1$ (called a "pumping length"[1]) such that every string *s* in *L* that has a length of *p* or more symbols (i.e. with $|s| \geq p$) can be written as

$$s = uvwxy$$

with substrings *u*, *v*, *w*, *x* and *y*, such that

1. $|vwx| \leq p$,
2. $|vx| \geq 1$, and
3. $uv^nwx^ny \in L$ for all $n \geq 0$.

Below is a formal expression of the Pumping Lemma.

Informal statement and explanation

The pumping lemma for context-free languages (called just "the pumping lemma" for the rest of this article) describes a property that all context-free languages are guaranteed to have.

The property is a property of all strings in the language that are of length at least p, where p is a constant—called the *pumping length*—that varies between context-free languages.

Say s is a string of length at least p that is in the language.

The pumping lemma states that s can be split into five substrings, $s = uvwxy$, where vx is non-empty and the length of vwx is at most p, such that repeating v and x any (and the same) number of times in s produces a string that is still in the language (it is possible and often useful to repeat zero times, which removes v and x from the string). This process of "pumping up" additional copies of v and x is what gives the pumping lemma its name.

Finite languages (which are regular and hence context-free) obey the pumping lemma trivially by having p equal to the maximum string length in L plus one. As there are no strings of this length the pumping lemma is not violated.

Usage of the lemma

The pumping lemma is often used to prove that a given language L is non-context-free, by showing that arbitrarily long strings s are in L that cannot be "pumped" without producing strings outside L.

For example, the language $L = \{ a^n b^n c^n \mid n > 0 \}$ can be shown to be non-context-free by using the pumping lemma in a proof by contradiction. First, assume that L is context free. By the pumping lemma, there exists an integer p which is the pumping length of language L. Consider the string $s = a^p b^p c^p$ in L. The pumping lemma tells us that s can be written in the form $s = uvwxy$, where u, v, w, x, and y are substrings, such that $|vwx| \leq p$, $|vx| \geq 1$, and $uv^i wx^i y \in L$ for every integer $i \geq 0$. By the choice of s and the fact that $|vwx| \leq p$, it is easily seen that the substring vwx can contain no more than two distinct symbols. That is, we have one of five possibilities for vwx:

1. $vwx = a^j$ for some $j \leq p$.
2. $vwx = a^j b^k$ for some j and k with $j+k \leq p$.

2.
3. $vwx = b^j$ for some $j \leq p$.
4. $vwx = b^j c^k$ for some j and k with $j+k \leq p$.
5. $vwx = c^j$ for some $j \leq p$.

For each case, it is easily verified that uv^iwx^iy does not contain equal numbers of each letter for any $i \neq 1$. Thus, uv^2wx^2y does not have the form $a^ib^ic^i$. This contradicts the definition of L. Therefore, our initial assumption that L is context free must be false.

While the pumping lemma is often a useful tool to prove that a given language is not context-free, it does not give a complete characterization of the context-free languages. If a language does not satisfy the condition given by the pumping lemma, we have established that it is not context-free.

On the other hand, there are languages that are not context-free, but still satisfy the condition given by the pumping lemma, for example $L = \{ b^jc^kd^l \mid j, k, l \in \mathbb{N} \} \cup \{ a^ib^jc^jd^j \mid i, j \in \mathbb{N}, i \geq 1 \}$: for $s=b^jc^kd^l$ with e.g. $j \geq 1$ choose vwx to consist only of b's, for $s=a^ib^jc^jd^j$ choose vwx to consist only of a's; in both cases all pumped strings are still in L.

The Closure of Context-Free Languages

We have seen that the regular languages are closed under common set-theoretic operations; the same, however, does not hold true for context-free languages.

Lemma: The context-free languages are closed under union, concatenation and Kleene closure. That is, if L_1 and L_2 are context-free languages, so are $L_1 \cup L_2$, $L_1 L_2$ and L_1^*.
Proof:
We will prove that the languages are closed by creating the appropriate grammars.

Suppose we have two context-free languages, represented by grammars with start symbols S_1 and S_2 respectively. First of all, rename all the terminal symbols in the second grammar so that they don't conflict with those in the first. Then:

- To get the union, add the rule $S \to S_1 \mid S_2$
- To get the concatenation, add the rule $S \to S_1 S_2$
- To get the Kleene Closure of L_1, add the rule $S \to S_1 S \mid \varepsilon$ to the grammar for L_1.

$\mathcal{QED}$

Lemma: The context-free languages are not closed under intersection
That is, if L_1 and L_2 are context-free languages, it it not always true that $L_1 \cap L_2$ is also.
Proof:
We will prove the non-closure of intersection by exhibiting a counter-example.
Consider the following two languages:

$$
\begin{aligned}
L_1 &= \{a^i b^j c^k \mid i < j\} \\
L_1 &= \{a^i b^j c^k \mid i < k\}
\end{aligned}
$$

We can prove that these *are* context-free by giving their grammars:

$$
\begin{aligned}
L_1 = \quad S &\rightarrow ABC \\
A &\rightarrow \mathbf{a}A\mathbf{b} \mid \varepsilon \\
B &\rightarrow \mathbf{b}B \mid \mathbf{b} \\
C &\rightarrow \mathbf{c}C \mid \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
L_2 = \quad S &\rightarrow ASC \mid B\mathbf{c} \\
A &\rightarrow \mathbf{a} \\
B &\rightarrow \mathbf{b}B \mid \varepsilon \\
C &\rightarrow \mathbf{c}C \mid \mathbf{c}
\end{aligned}
$$

The intersection of these languages is:

$$L_1 \cap L_2 = \{a^i b^j c^k \mid i < j \; and \; i < k\}$$

We proved in the last section that this language is not context-free.

$\mathcal{QED}$

Lemma: The context-free languages are not closed under complementation.
That is, if L is a context-free language, it it not always true that L' is also.
Proof: *(By contradiction)*
Suppose that context-free languages *are* closed under complementation.
Then if L_1 and L_2 are context-free languages, so are L_1' and L_2'. Since we have proved closure under union, $(L_1' \cup L_2')$ must also be context-free, and, by our assumption, so must its complement $(L_1' \cup L_2')'$.
However, by de Morgan's laws (for sets), $(L_1' \cup L_2')' \equiv (L_1 \cap L_2)$, so this must also be a context-free language.
Since our choice of L_1 and L_2 was arbitrary, we have contradicted the non-closure of intersection, and have thus proved the lemma.

$\mathcal{QED}$

Decision Properties

Now we consider some important questions for which algorithms exist to answer the question/

Is a given string in a CFL?

Given a string *w*, is it in a given CFL?

If we are given the CFL as a PDA, we can answer this ismply by executing the PDA.

If we are given the language as a grammar, we can either

- Convert the grammar to a PDA and execute the PDA, or
- Convert the grammar to Chomsky Normal Form and parse the string to find a derivation for it.

It may not be obvious that our procedures for finding derivations will always terminate. We have seen that, when finding a derivation, we have choices as to which variable to replace and which production to use in the replacement. However,

- We have previously noted that if a string is in the language, then it will have a *leftmost* derivation. So we can systematically always choose to replace the leftmost variable.
  - That leaves the choice of production. We can systematically try all available choices, in a form of backtracking.

    This terminates because we only have a finite number of productions to choose among, and one of the key properties of Chomsky Normal Forms is that each derivation step either increases the total number of symbols by one or leaves the total number of symbols unchanged while replacing a variable by a terminal. So any time the total number of symbols in a derived string exceeds the length of the string we are trying to derive, we know we have chosen an incorrect production somewhere along the way and can backtrack and try a different one.

Actually, we can do better than that. The CYK algorithm can parse a string *w*for a Chomsky Normal Form grammar in $O(|w|^3)$ time.

Is a CFL empty?

We've already seen how to detect whether a variable is nullable.

We apply that test and determine if the grammar's start symbol is nullable.

These are not decision properties for CFLs

No algorithm exists to determine if

- Two CFLs are the same.
    - Note that we *were* able to determine this for regular languages.
- Two CFLs are disjoint (have no strigns in common).

Closure Properties

Substitution

Given a CFG G, if we replace each terminal symbol by a set of strings that is itself a CFL, the result is still a CFL.

Example 3: Expressions over Mirrors

Here is a grammar for simple expressions: $EEEEII \to I \to E+E \to E*E \to (E) \to a \to b$

Here is our grammar for ww^R: $SSS \to 0S0 \to 1S1 \to \epsilon$

Now, suppose that I wanted an expression language in which, instead of the variable name "a", I could use any mirror-imaged string of 0's and 1's.

All I really have to do is to change 'a' in the first grammar to a variable, make that variable the new starting symbol for the ww^R

grammar, then combine the two grammars:
$EEEEIISSS \to I \to E+E \to E*E \to (E) \to a \to S \to 0S0 \to 1S1 \to \epsilon$

It's pretty obvious that this is still a CFG, so the resulting language is still a CFL.

Closure Under Union

Suppose you have CFLs L_1 and L_2.

Consider a language $L=\{1,2\}$. This is clearly a CFL as well.Now substitute L_1 for 1 in L and L_2 for 2 in L. The resulting L-with-substitutions accepts all strings that are in L_1 or L_2, in other words $L_1 \cup L_2$.

Other Closure Properties

Similar substitutions allow us to quickly show that the CFLs are closed under concatenation, Kleene closure, and homomorphisms.

CFLS are *not* closed under intersection and difference.

- But the difference of a CFL and a regular language is still a CFL.

CFLs are closed under reversal. (No surprise, given the stack-nature of PDAs.)

Running time of conversions to Chomsky Normal Form:

A CFG is in Chomsky Normal Form if the Productions are in the following forms −

- A → a
- A → BC
- S → ε

where A, B, and C are non-terminals and **a** is terminal.

Algorithm to Convert into Chomsky Normal Form –

Step 1 − If the start symbol **S** occurs on some right side, create a new start symbol **S'** and a new production **S'→ S**.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Replace each production $A \rightarrow B_1 \ldots B_n$ where **n > 2** with $A \rightarrow B_1C$ where $C \rightarrow B_2 \ldots B_n$. Repeat this step for all productions having two or more symbols in the right side.

Step 5 − If the right side of any production is in the form **A → aB** where a is a terminal and **A, B** are non-terminal, then the production is replaced by **A → XB** and **X → a**. Repeat this step for every production which is in the form **A → aB**.

Problem

Convert the following CFG into CNF

S → ASA | aB, A → B | S, B → b | ε

Solution

(1) Since **S** appears in R.H.S, we add a new state $\mathbf{S_0}$ and $\mathbf{S_0}$**→S** is added to the production set and it becomes −

S_0→S, S→ ASA | aB, A → B | S, B → b | ∈

(2) Now we will remove the null productions −

B → ∈ and A → ∈

After removing B → ε, the production set becomes −

S_0→S, S→ ASA | aB | a, A → B | S | ∈, B → b

After removing A → ∈, the production set becomes −

S_0→S, S→ ASA | aB | a | AS | SA | S, A → B | S, B → b

(3) Now we will remove the unit productions.

After removing S → S, the production set becomes −

S_0→S, S→ ASA | aB | a | AS | SA, A → B | S, B → b

After removing S_0→ S, the production set becomes −

S_0→ ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → B | S, B → b

After removing A→ B, the production set becomes −

S_0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → S | b

B → b

After removing A→ S, the production set becomes −

S_0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → b |ASA | aB | a | AS | SA, B → b

(4) Now we will find out more than two variables in the R.H.S

Here, $S_0 \rightarrow ASA$, $S \rightarrow ASA$, $A \rightarrow ASA$ violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF −

$S_0 \rightarrow AX \mid aB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid AX \mid aB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

(5) We have to change the productions $S_0 \rightarrow aB$, $S \rightarrow aB$, $A \rightarrow aB$

And the final production set becomes −

$S_0 \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$A \rightarrow b$ $A \rightarrow b \mid AX \mid YB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

Introduction to Turing Machines-Problems That Computers Cannot Solve:

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple $(Q, X, \sum, \delta, q_0, B, F)$ where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- **δ** is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.
- q_0 is the initial state
- **B** is the blank symbol
- **F** is the set of final states

Comparison with the previous automaton

The following table shows a comparison of how a Turing machine differs from Finite Automaton and Pushdown Automaton.

Machine	Stack Data Structure	Deterministic?
Finite Automaton	N.A	Yes
Pushdown Automaton	Last In First Out(LIFO)	No
Turing Machine	Infinite tape	Yes

Example of Turing machine

Turing machine M = (Q, X, ∑, δ, q_0, B, F) with

- Q = {q_0, q_1, q_2, q_f}
- X = {a, b}
- ∑ = {1}
- q_0 = {q_0}
- B = blank symbol
- F = {q_f }

δ is given by −

Tape alphabet symbol	Present State 'q_0'	Present State 'q_1'	Present State 'q_2'
a	$1Rq_1$	$1Lq_0$	$1Lq_f$
b	$1Lq_2$	$1Rq_1$	$1Rq_f$

Here the transition $1Rq_1$ implies that the write symbol is 1, the tape moves right, and the next state is q_1. Similarly, the transition $1Lq_2$ implies that the write symbol is 1, the tape moves left, and the next state is q_2.

Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

Time complexity all reasonable functions −

T(n) = O(n log n)

TM's space complexity −

S(n) = O(n)

The Turing Machine, Programming Techniques for Turing Machines:

Programming Techniques for Turing Machines 329

Storage in the State 330

- Storage in the State
I\'m terribly confused on how this process works. Can you detail why the transition function is defined as it is?

A Turing machine has a finite number of states in its CPU. However, the number of states is not always small (like 6). Like a Pentium chip we can store values in it -- as long as there are only finite number of them. For example all real computers have registers but there are only a fixed number of them, AND each register can only hold one of a fixed (and finite) number of bits.

Example: a simple computer with control unit holding a PC with a 16bit address + arithmetic unit holding one double length 64 bit Arithmetic Register. This has stores 40 bits of information plus a small internal control state: {fetch, decode, compute, store, next} perhaps. So we have a finite state machine with 2^80 states!

It is sad that we don't have a high level language for TMs then we could write statements like

1. x=tape.head; perhaps.

Suppose that we wanted to code (in C++) a TM with states 'q0', 'q1',... plus storage 'x' we might write code like this

```
q0: b=tape.head; tape.move_right(); goto q1;
q1: if(tape.head()==B) return FAILS;
```

```
    if(tape.head()==x) tape.move_right(); goto q1;
 ...
```

Example 8.6

I found it very confusing, the idea of storage in states. Could you go over this kind of problem? And is it beneficial to convey the transition function in table format, as it was done in sections 8.1 and 8.2, for this example and what would the table be in this case?

A typical transition table when data is being stored has entries like this

State	Read	Next	Write	Move
(q0,B)	1	(q1,1)	1	R
(q0,B)	0	(q1,0)	0	R

We might interpret this as part of a step that stores the symbol read from the tape in the CPU. Note: the book uses "[]" where I use "()".

Tabulating the 6 states >< 3 symbols on page 331 is not difficult and may help.

But suppose you have a CPU that has 6 binary flags stored in it plus 3 control states. Now you have 3*64 states to write down...

Tabulating all possible states is not very helpful for machines with structured or compound states…. unless you allow variables and conditions to appear in the state descriptions AND expressions in the body of the table.

The other thing missing from the table is the documentation that explains what each transition means.

Multiple Tracks 331

- Multitrack Turing Machine

Can you explain multiple track idea and its relation to the Turing Machine? And maybe show one example in class?

Key idea: \Gamma is the Cartesian product of a finite number of finite sets. (Cartesian product works like a struct in C/C++)

For example: computer tape storage is multi-track tape with something like 8 or 9 bits in each cell.

You can recognise a multi-track machine by looking at the transitions because each will have *tuples* as the read and write symbols.

```
(*,1)/(B,0)->
(*,0)/(B,1)->
(B,0)/(B,0)->
(B,1)/(B,1)->
```

Suppose that all of the above are on a loop from *q* to *q* in a diagram then we have these entries in the table:

State	Next	Read	Write	Move
q	(*,1)	q	(B,0)	R
q	(*,0)	q	(B,1)	R
q	(B,1)	q	(B,1)	R
q	(B,0)	q	(B,0)	R

Both have the effect of taking a marked (*) bits and flipping them.

Exercise: complete the \delta descriptions that fit with the above:

- ...
- \delta(q, (*,i)) = (____________,__________, R),
- \delta(q, (______,i)) = (B,i, R),
- ...

Note the book is using "[]" in place of '()' for tuples. Other books use '<>'!

Error on page 332

In the description of \Sigma does not mention the encoding of "c" as *[B,c]*.

- Multiple Tracks

In example 8.7, i had problem understanding the transition functions, and the duty or assignment for each part in the function.

Can you explain, what each part of the transition function is supposed to mean or do?

Subroutines 333

Subroutines

Would you please explain a little bit more on figure 8.14 and figure 8.15, on how the subroutine Copy works?

Exercises for Section 8.3 334

Extensions to the Basic Turing Machine 336

8.4.1 Multitape Turing Machines 336

Note: try not to confuse a multi-tape machine with a multi-track single tape machine!

You can recognize a multitape machine by its table of transitions. Here is a piece of a 2-tape machine

State	Symbol1	Symbol2	New1	New2	Move1	Move2
q0	a	b	c	d	L	R
...						

Which means that in state *q0*, if the first head sees an "a", and the second head sees a "b" then it write "c" on tape 1 and moves that head left, and it also writes "d" on tape 2 and moves that head right. Notice that we have two symbols that are read, two are written and their are two moves.

The transition would be something like:

```
ab/cd<-->
```

Figures 8.15 8.16 8.17

Could you please explain the two diagrams step by step. Figure 8.15 and figure 8.16 or 8.17

-- Multitape Turing Machines

Why use multi-tape Turing Machines if they are equivalent to single tape Turing Machines?

They are easier to program. ...and run *faster*.

Equivalence of One-Tape and Multitape TM's 337

Running Time and the Many-Tapes-to-One Construction 339

- Introduction to Turing Machines

Could you go over how Many-Tapes-to-One Construction is used in the time complexity of a TM? This wasn't clear to me in the chapter.

Nondeterministic Turing Machines 340

Nondeterministic Turing Machine

Can you clarify the difference between a nondeterministic Turing Machine and a deterministic one?

- NTM

How does a nondeterministic Turing machine differ from a deterministic Turing machine?

Group discussion: what makes a TM non-deterministic?

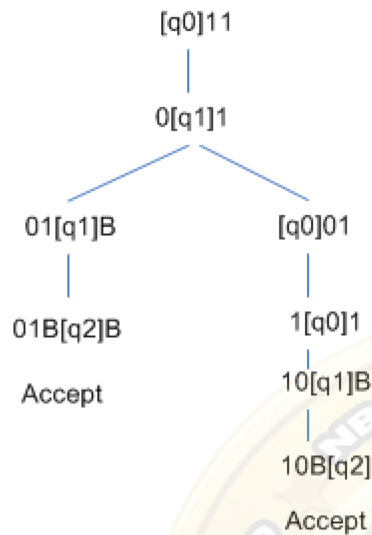
> The machine has choices in some states and with some symbols: there is at least one pair of state><head with two (or more) possible transitions. This spawns extra (hypothetical) threads. The diagram has many arrows leaving a state with the same symbol on it.

As a result we can't trace a sequence of steps like this

```
[q0]000 |- 1[q0]00 |- 11[q0]0 |-
```

because the TM can branch into several possible paths "at once".

One way to handle this is to draw a tree. This can take a lot of paper but works well for humans. For example, here is a tree of IDs generated by the NTM in the execises for this section:



Another approach is embedded in the Proof of Theore, 8.11.

Nondeterministic Turing Machines

Please explain the proof of Theorem 8.11 on Nondeterministic Turning Machines?

As an introduction to the proof it is worth doing exercise 8.4.2 using a list of the IDs reachable from the initial ID. For each of these in turn, add the set of next IDs at the end of the list and cross out the state they came from from the beginning of the list.

In class we traced several the IDs that could come from the NTM in the dreaded Exercise 8.4.4.

1. **Extensions of Turing Machines**
   1. **Multiple tapes**

      There are several tapes and a head for each tape
      The configuration specifies the state, the contents of each tape and the position of the head for each tape.

   2. **Two-way infinite tape**

      The tape is infinite in both directions

   3. **Multiple heads**

      There is one tape with several heads.
      The configuration specifies the state, the contents of the tape and the position of each head.

NBA ACCREDITED MRITS

4. **Two-dimensional tape**

   Two more directions for movement: up and down

All these extensions can be simulated on a standard Turing machine. They do not add to the power of the machine.

Theorem: Any language decided or semidecided, and any function computed by Turing machines with several tapes, heads, two-way infinite tapes, or multi-dimensional tapes, can be decided, semidecided , or computed respectively by a standard Turing machine.

2. **Random access Turing machines**

The tape in Turing machines is sequential, while real computers have random access memory.
To implement such capability, Turing machines are augmented with registers to hold addresses of tape locations.

We assume that each cell of the one-way infinite tape can contain any natural number.
We have also a fixed number of registers, and each can contain any natural number.

The transition function is a sequence of instructions (corresponding to a basic Assembler set of instructions) and the last instruction is "halt".
Thus the transition function simulates a program.

In summary, we have:

- A one-way infinite tape. Each cell can contain any natural number
- A fixed number of registers, each can contain any natural number.
- Transition function - a program (a sequence of instructions, last instruction is halt)
- Fixed set of instructions:

read/write   read from and write onto the tape
load/store   read from and write into the registers
add/sub   addition and subtraction
half
jump   move the head to a specified cell
conditional jump move if the contents of a specified register
                  is positive/zero

Theorem: Any recursive or recursively enumerable language and any recursive function
can be decided, semidecided or computed respectively by a random access Turing machine.

Theorem:

5. Any language decided or semidecided by a random access Turing machine,
   and any function computable by a random access Turing machine,
   can be decided, semidecided, or computed respectively by a standard Turing machine.
6. If the machine halts on an input, the number of steps taken by the standard Turing machine is
   bound by a polynomial in the number of steps of the random access Turing machine
   on the same input.

   The second part of the theorem means the following:

   > If **M** is a random access Turing machine that performs a computation
   > on input of length **n** in **t** steps,
   > then there exists a number **k** and a standard Turing machine **M'**
   > that will do the same computation in $\mathbf{O(t + n)^k}$ steps.

Nondeterministic Turing machines

Non-determinism means that there may be two or more transitions on one and the same "state, symbol" pair. The computation looks like a tree - representing the possible choices.

> A non-deterministic Turing machine **decides a language**, or computes a function, if it halts on all possible inputs.
>
> A **string** in the language **is accepted** if at least one of the possible configurations ends up accepting the input.
>
> A **function is computed** if all possible computations agree on the output.

Learning goals

- Be able to describe the types of Turing machines and state the related theorems.

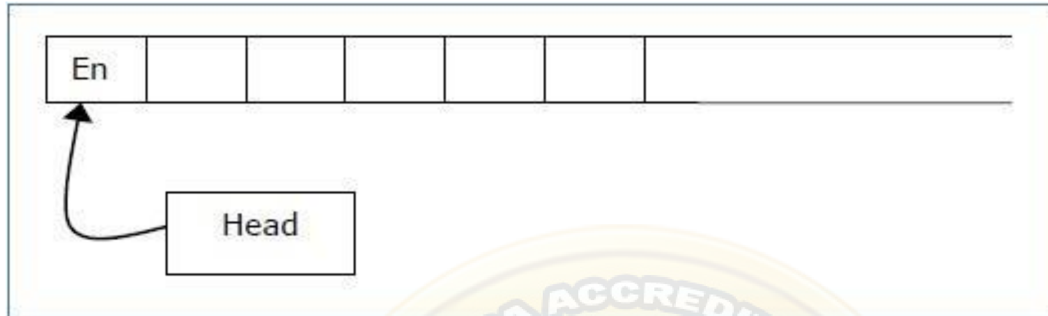
Restricted Turing Machines

Semi-infinite Tapes' TM

- A TM with a *semi-infinite* tape means that there are no cells to the left of the initial head position.

- A TM with a semi-infinite tape simulates a TM with an infinite tape by using a two-track tape.

A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.



It is a two-track tape −

- **Upper track** − It represents the cells to the right of the initial head position.
- **Lower track** − It represents the cells to the left of the initial head position in reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.

The machine starts from the initial state q_0 and the head scans from the left end marker 'End'. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called **accept state** and **reject state**. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note − Turing machines with semi-infinite tape are equivalent to standard Turing machines.

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

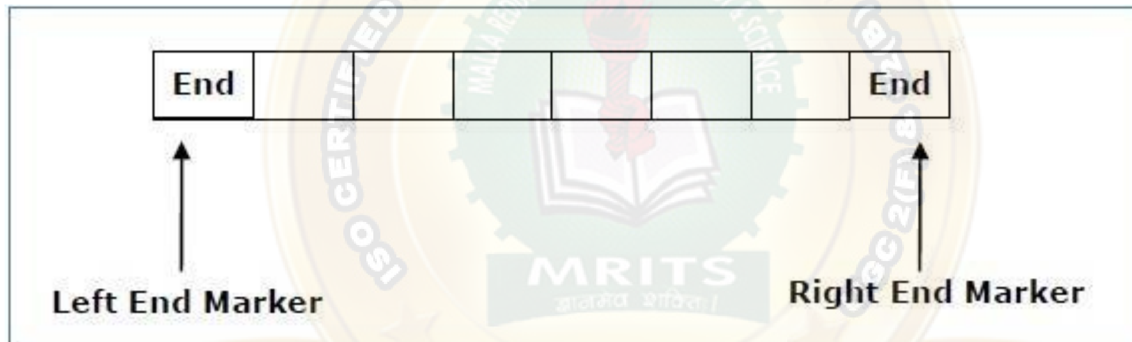
Here,

Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

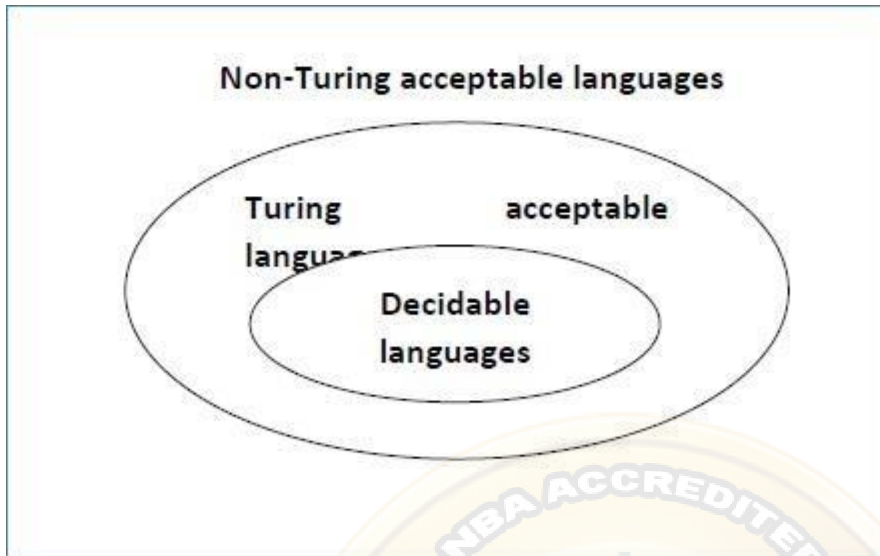
A linear bounded automaton can be defined as an 8-tuple (Q, X, ∑, q_0, ML, MR, δ, F) where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- ∑ is the input alphabet
- q_0 is the initial state
- M_L is the left end marker
- M_R is the right end marker where $M_R \neq M_L$
- **δ** is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- **F** is the set of final states



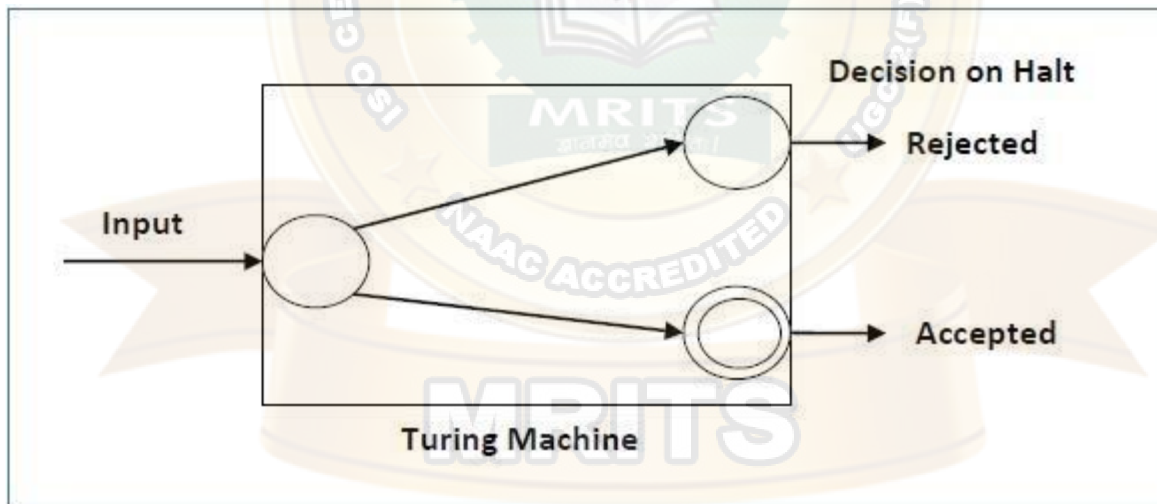
A deterministic linear bounded automaton is always **context-sensitive** and the linear bounded automaton with empty language is **undecidable.**.

A language is called **Decidable** or **Recursive** if there is a Turing machine which accepts and halts on every input string **w**. Every decidable language is Turing-Acceptable.



A decision problem **P** is decidable if the language **L** of all yes instances to **P** is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram −



Example 1

Find out whether the following problem is decidable or not −

Is a number 'm' prime?

Solution

Prime numbers = {2, 3, 5, 7, 11, 13, ... }

Divide the number **'m'** by all the numbers between '2' and '√m' starting from '2'.

If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.

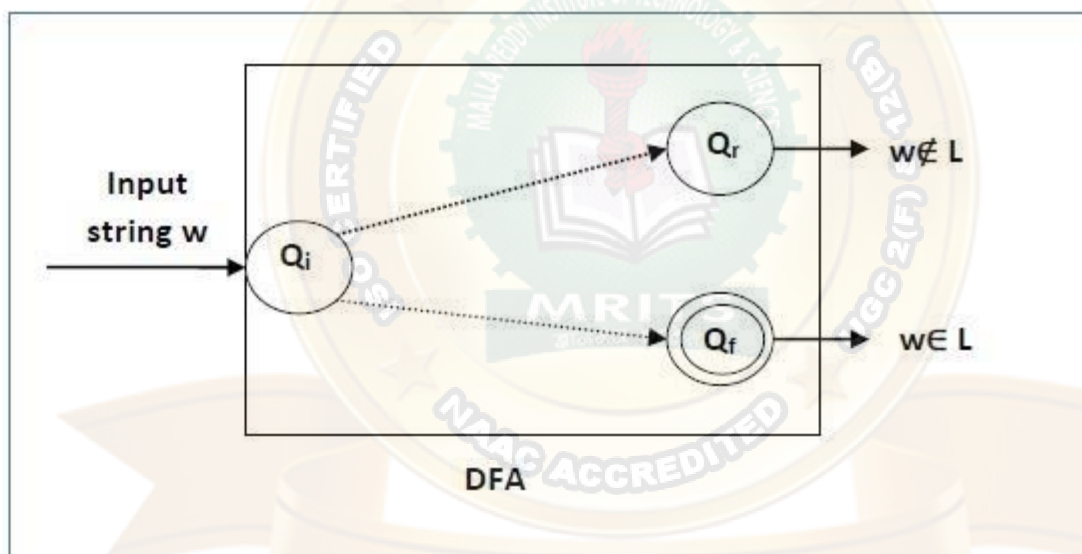
Hence, it is a decidable problem.

Example 2

Given a regular language **L** and string **w**, how can we check if **w** ∈ **L**?

Solution

Take the DFA that accepts **L** and check if **w** is accepted



Some more decidable problems are −

- Does DFA accept the empty language?
- Is $L_1 \cap L_2 = \emptyset$ for regular sets?

Note −

- If a language **L** is decidable, then its complement **L'** is also decidable
- If a language is decidable, then there is an enumerator for it.

A **Turing machine** is a mathematical model of computation that defines an abstract machine, which manipulates symbols on a strip of tape according to a table of rules. Despite the model's

simplicity, given any computer algorithm, a Turing machine capable of simulating that algorithm's logic can be constructed. The machine operates on an infinite memory tape divided into discrete *cells*. The machine positions its *head* over a cell and "reads" (scans) the symbol there. Then, as per the symbol and its present place in a *finite table*of user-specified instructions, the machine (i) writes a symbol (e.g., a digit or a letter from a finite alphabet) in the cell (some models allowing symbol erasure or no writing), hen (ii) either moves the tape one cell left or right (some models allow no motion, some models move the head), then (iii) (as determined by the observed symbol and the machine's place in the table) either proceeds to a subsequent instruction or halts the computation.

The Turing machine was invented in 1936 by Alan Turing, who called it an *a-machine* (automatic machine. With this model, Turing was able to answer two questions in the negative: (1) Does a machine exist that can determine whether any arbitrary machine on its tape is "circular" (e.g., freezes, or fails to continue its computational task); similarly, (2) does a machine exist that can determine whether any arbitrary machine on its tape ever prints a given symbol. Thus by providing a mathematical description of a very simple device capable of arbitrary computations, he was able to prove properties of computation in general—and in particular, the uncomputability of the Entscheidungsproblem ("decision problem.

Thus, Turing machines prove fundamental limitations on the power of mechanical computation. While they can express arbitrary computations, their minimalistic design makes them unsuitable for computation in practice: real-world computers are based on different designs that, unlike Turing machines, use random-access memory.

Turing completeness is the ability for a system of instructions to simulate a Turing machine. A programming language that is Turing complete is theoretically capable of expressing all tasks accomplishable by computers; nearly all programming languages are Turing complete if the limitations of finite memory are ignored.

Overview

A Turing machine is a general example of a CPU that controls all data manipulation done by a computer, with the canonical machine using sequential memory to store data. More specifically, it is a machine (automaton) capable of enumerating some arbitrary subset of valid strings of an alphabet; these strings are part of a recursively enumerable set. A Turing machine has a tape of infinite length that enables read and write operations to be performed.

Assuming a black box, the Turing machine cannot know whether it will eventually enumerate any one specific string of the subset with a given program. This is due to the fact that the halting problem is unsolvable, which has major implications for the theoretical limits of computing.

The Turing machine is capable of processing an unrestricted grammar, which further implies that it is capable of robustly evaluating first-order logic in an infinite number of ways. This is famously demonstrated through lambda calculus.

A Turing machine that is able to simulate any other Turing machine is called a universal Turing machine (**UTM**, or simply a **universal machine**). A more mathematically oriented definition with a similar "universal" nature was introduced by Alonzo Church, whose work on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective methods in logic and mathematics, and provide a precise definition of an algorithm or "mechanical procedure". Studying their abstract properties yields many insights into computer science and complexity theory.

Physical description

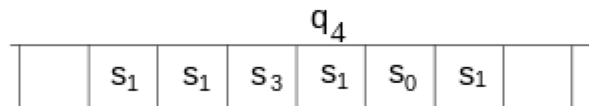
In his 1948 essay, "Intelligent Machinery", Turing wrote that his machine consisted of:

> ...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol, and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.[17] (Turing 1948, p. 3[18])

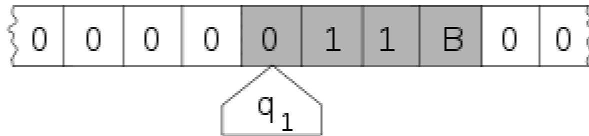
Informal description

For visualizations of Turing machines, see Turing machine gallery.

The Turing machine mathematically models a machine that mechanically operates on a tape. On this tape are symbols, which the machine can read and write, one at a time, using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc. In the original article ("On Computable Numbers, with an Application to the Entscheidungsproblem", see also references below), Turing imagines not a mechanism, but a person whom he calls the "computer", who executes these deterministic mechanical rules slavishly (or as Turing puts it, "in a desultory manner").



The head is always over a particular square of the tape; only a finite stretch of squares is shown. The instruction to be performed (q_4) is shown over the scanned square. (Drawing after Kleene (1952) p. 375.)



Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its *complete configuration*) consists of the internal state, any non-blank symbols on the tape (in this illustration "11B"), and the position of the head relative to those symbols including blanks, i.e. "011B". (Drawing after Minsky (1967) p. 121.)

More precisely, a Turing machine consists of:

- A **tape** divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as '0') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.
- A **head** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.
- A **state register** that stores the state of the Turing machine, one of finitely many. Among these is the special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.
- A finite **table**[19] of instructions[20] that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under the head), tells the machine to do the following *in sequence* (for the 5-tuple models):

1. Either erase or write a symbol (replacing a_j with a_{j1}).
2. Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place).
3. Assume the same or a *new state* as prescribed (go to state q_{i1}).

In the 4-tuple models, erasing or writing a symbol (a_{j1}) and moving the head left or right (d_k) are specified as separate instructions. Specifically, the table tells the machine to (ia) erase or write a symbol *or* (ib) move the head left or right, *and then* (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.

Note that every part of the machine (i.e. its state, symbol-collections, and used tape at any given time) and its actions (such as printing, erasing and tape motion) is *finite*, *discrete* and *distinguishable*; it is the unlimited amount of tape and runtime that gives it an unbounded amount of storage space.

Formal definition

Following Hopcroft and Ullman (1979, p. 148), a (one-tape) Turing machine can be formally defined as a 7-tuple where

- is a finite, non-empty set of *states*;
- is a finite, non-empty set of *tape alphabet symbols*;
- is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
- is the set of *input symbols*, that is, the set of symbols allowed to appear in the initial tape contents;
- is the *initial state*;
- is the set of *final states* or *accepting states*. The initial tape contents is said to be *accepted* by if it eventually halts in a state from .
- is a partial function called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.) If is not defined on the current state and the current tape symbol, then the machine halts;[21]

Anything that operates according to these specifications is a Turing machine.

The 7-tuple for the 3-state busy beaver looks like this (see more about this busy beaver at Turing machine examples):

- (states);
- (tape alphabet symbols);
- (blank symbol);
- (input symbols);
- (initial state);
- (final states);
- see state-table below (transition function).

Initially all tape cells are marked with .

State table for 3 state, 2 symbol busy beaver

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	R	**HALT**

Additional details required to visualize or implement Turing machines

In the words of van Emde Boas (1990), p. 6: "The set-theoretical object [his formal seven-tuple description similar to the above] provides only partial information on how the machine will behave and what its computations will look like."

For instance,

- There will need to be many decisions on what the symbols actually look like, and a failproof way of reading and writing symbols indefinitely.
- The shift left and shift right operations may shift the tape head across the tape, but when actually building a Turing machine it is more practical to make the tape slide back and forth under the head instead.
- The tape can be finite, and automatically extended with blanks as needed (which is closest to the mathematical definition), but it is more common to think of it as stretching infinitely at both ends and being pre-filled with blanks except on the explicitly given finite fragment the tape head is on. (This is, of course, not implementable in practice.) The tape *cannot* be fixed in length, since that would not correspond to the given definition and would seriously limit the range of computations the machine can perform to those of a linear bounded automaton.

Alternative definitions

Definitions in literature sometimes differ slightly, to make arguments or proofs easier or clearer, but this is always done in such a way that the resulting machine has the same computational

power. For example, changing the set to , where *N* ("None" or "No-operation") would allow the machine to stay on the same tape cell instead of moving left or right, does not increase the machine's computational power.

The most common convention represents each "Turing instruction" in a "Turing table" by one of nine 5-tuples, per the convention of Turing/Davis (Turing (1936) in *The Undecidable*, p. 126-127 and Davis (2000) p. 152):

(definition 1): **(q_i, S_j, S_k/E/N, L/R/N, q_m)**

(current state **q_i ,** symbol scanned **S_j ,** print symbol **S_k**/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N ,** new state **q_m)**

Other authors (Minsky (1967) p. 119, Hopcroft and Ullman (1979) p. 158, Stone (1972) p. 9) adopt a different convention, with new state **q_m** listed immediately after the scanned symbol S_j:

(definition 2): **(q_i, S_j, q_m, S_k/E/N, L/R/N)**

(current state **q_i ,** symbol scanned **S_j ,** new state **q_m ,** print symbol **S_k**/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N)**

For the remainder of this article "definition 1" (the Turing/Davis convention) will be used.

Example: state table for the 3-state 2-symbol busy beaver reduced to 5-tuples

Current state	Scanned symbol	Print symbol	Move tape	Final (i.e. next) state	5-tuples
A	0	1	R	**B**	(**A**, 0, 1, R, **B**)
A	1	1	L	**C**	(**A**, 1, 1, L, **C**)
B	0	1	L	**A**	(**B**, 0, 1, L, **A**)
B	1	1	R	**B**	(**B**, 1, 1, R, **B**)
C	0	1	L	**B**	(**C**, 0, 1, L, **B**)
C	1	1	N	**H**	(**C**, 1, 1, N, **H**)

In the following table, Turing's original model allowed only the first three lines that he called N1, N2, N3 (cf. Turing in *The Undecidable*, p. 126). He allowed for erasure of the "scanned square" by naming a 0th symbol S_0 = "erase" or "blank", etc. However, he did not allow for non-printing, so every instruction-line includes "print symbol S_k" or "erase" (cf. footnote 12 in Post (1947), *The Undecidable*, p. 300). The abbreviations are Turing's (*The Undecidable*, p. 119). Subsequent to Turing's original paper in 1936–1937, machine-models have allowed all nine possible types of five-tuples:

	Current m-configuration (Turing state)	Tape symbol	Print-operation	Tape-motion	Final m-configuration (Turing state)	5-tuple	5-tuple comments	4-tuple
N1	q_i	S_j	Print(S_k)	Left L	q_m	(q_i, S_j, S_k, L, q_m)	"blank" = S_0, 1=S_1, etc.	
N2	q_i	S_j	Print(S_k)	Right R	q_m	(q_i, S_j, S_k, R, q_m)	"blank" = S_0, 1=S_1, etc.	
N3	q_i	S_j	Print(S_k)	None N	q_m	(q_i, S_j, S_k, N, q_m)	"blank" = S_0, 1=S_1, etc.	(q_i, S_j, S_k, q_m)
4	q_i	S_j	None N	Left L	q_m	(q_i, S_j, N, L, q_m)		(q_i, S_j, L, q_m)
5	q_i	S_j	None N	Right R	q_m	(q_i, S_j, N, R, q_m)		(q_i, S_j, R, q_m)
6	q_i	S_j	None N	None N	q_m	(q_i, S_j, N, N, q_m)	Direct "jump"	(q_i, S_j, N, q_m)
7	q_i	S_j	Erase	Left L	q_m	(q_i, S_j, E, L, q_m)		
8	q_i	S_j	Erase	Right R	q_m	(q_i, S_j, E, R, q_m)		
9	q_i	S_j	Erase	None N	q_m	(q_i, S_j, E, N, q_m)		(q_i, S_j, E, q_m)

Any Turing table (list of instructions) can be constructed from the above nine 5-tuples. For technical reasons, the three non-printing or "N" instructions (4, 5, 6) can usually be dispensed with. For examples see Turing machine examples.

Less frequently the use of 4-tuples are encountered: these represent a further atomization of the Turing instructions (cf. Post (1947), Boolos & Jeffrey (1974, 1999), Davis-Sigal-Weyuker (1994)); also see more at Post–Turing machine.

The "state"

The word "state" used in context of Turing machines can be a source of confusion, as it can mean two things. Most commentators after Turing have used "state" to mean the name/designator of the current instruction to be performed—i.e. the contents of the state register. But Turing (1936) made a strong distinction between a record of what he called the machine's "m-configuration", and the machine's (or person's) "state of progress" through the computation - the current state of the total system. What Turing called "the state formula" includes both the current instruction and *all* the symbols on the tape:

Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. That is, the **state of the system** may be described by a single expression (sequence of symbols) consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression is called the 'state formula'.

— *The Undecidable, pp. 139–140, emphasis added*

Earlier in his paper Turing carried this even further: he gives an example where he placed a symbol of the current "m-configuration"—the instruction's label—beneath the scanned square, together with all the symbols on the tape (*The Undecidable*, p. 121); this he calls "the *complete configuration*" (*The Undecidable*, p. 118). To print the "complete configuration" on one line, he places the state-label/m-configuration to the *left* of the scanned symbol.

A variant of this is seen in Kleene (1952) where Kleene shows how to write the Gödel number of a machine's "situation": he places the "m-configuration" symbol q_4 over the scanned square in roughly the center of the 6 non-blank squares on the tape (see the Turing-tape figure in this article) and puts it to the *right* of the scanned square. But Kleene refers to "q_4" itself as "the machine state" (Kleene, p. 374-375). Hopcroft and Ullman call this composite the "instantaneous description" and follow the Turing convention of putting the "current state" (instruction-label, m-configuration) to the *left* of the scanned symbol (p. 149).

Example: total state of 3-state 2-symbol busy beaver after 3 "moves" (taken from example "run" in the figure below):

1**A**1

This means: after three moves the tape has ... 000110000 ... on it, the head is scanning the right-most 1, and the state is **A**. Blanks (in this case represented by "0"s) can be part of the total state as shown here: **B**01; the tape has a single 1 on it, but the head is scanning the 0 ("blank") to its left and the state is **B**.

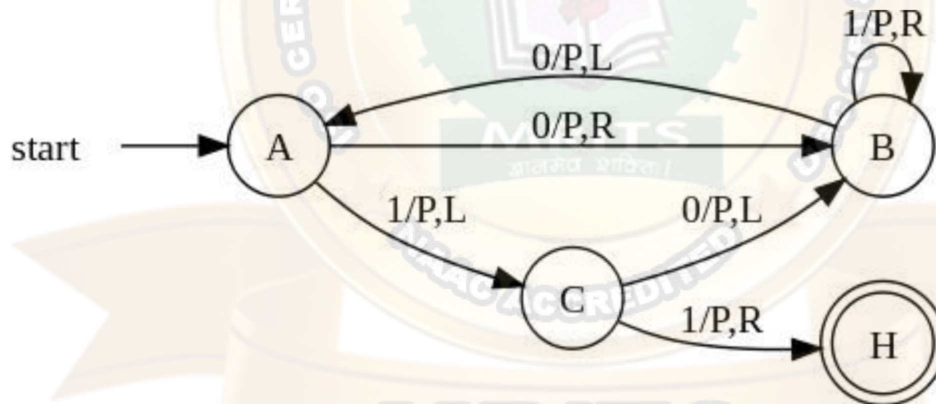
"State" in the context of Turing machines should be clarified as to which is being described: (*i*) the current instruction, or (*ii*) the list of symbols on the tape together with the current instruction, or (*iii*) the list of symbols on the tape together with the current instruction placed to the left of the scanned symbol or to the right of the scanned symbol.

Turing's biographer Andrew Hodges (1983: 107) has noted and discussed this confusion.

Turing machine "state" diagrams

The table for the 3-state busy beaver ("P" = print/write a "1")

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	P	R	**B**	P	L	**A**	P	L	**B**
1	P	L	**C**	P	R	**B**	P	R	**HALT**

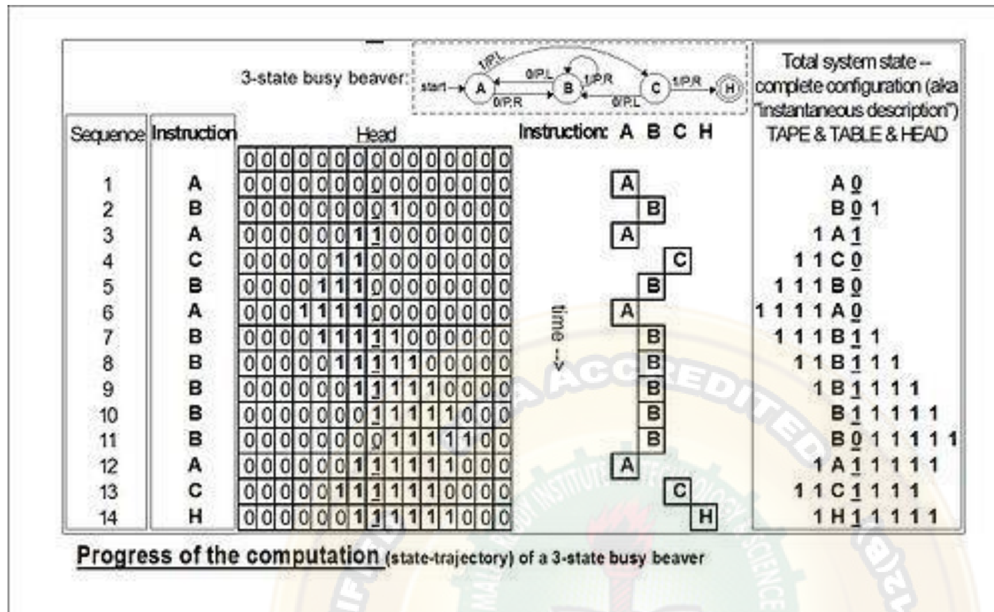


The "3-state busy beaver" Turing machine in a finite state representation. Each circle represents a "state" of the table—an "m-configuration" or "instruction". "Direction" of a state *transition* is shown by an arrow. The label (e.g. **0/P,R**) near the outgoing state (at the "tail" of the arrow) specifies the scanned symbol that causes a particular transition (e.g. **0**) followed by a slash **/**, followed by the subsequent "behaviors" of the machine, e.g. "**P P**rint" then move tape "**R R**ight". No general accepted format exists. The convention shown is after McClusky (1965), Booth (1967), Hill, and Peterson (1974).

To the right: the above table as expressed as a "state transition" diagram.

Usually large tables are better left as tables (Booth, p. 74). They are more readily simulated by computer in tabular form (Booth, p. 74). However, certain concepts—e.g. machines with "reset" states and machines with repeating patterns (cf. Hill and Peterson p. 244ff)—can be more readily seen when viewed as a drawing.

Whether a drawing represents an improvement on its table must be decided by the reader for the particular context. See Finite state machine for more.



The evolution of the busy-beaver's computation starts at the top and proceeds to the bottom.

The reader should again be cautioned that such diagrams represent a snapshot of their table frozen in time, *not* the course ("trajectory") of a computation *through* time and space. While every time the busy beaver machine "runs" it will always follow the same state-trajectory, this is not true for the "copy" machine that can be provided with variable input "parameters".

The diagram "Progress of the computation" shows the three-state busy beaver's "state" (instruction) progress through its computation from start to finish. On the far right is the Turing "complete configuration" (Kleene "situation", Hopcroft–Ullman "instantaneous description") at each step. If the machine were to be stopped and cleared to blank both the "state register" and entire tape, these "configurations" could be used to rekindle a computation anywhere in its progress (cf. Turing (1936) *The Undecidable*, pp. 139–140).

Models equivalent to the Turing machine model

See also: Turing machine equivalents, Register machine, and Post–Turing machine

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf. Minsky (1967)). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church–Turing thesis *hypothesizes* this to be true for any kind of machine: that anything that can be "computed" can be computed by some Turing machine.)

A Turing machine is equivalent to a single-stack pushdown automaton (PDA) that has been made more flexible and concise by relaxing the last-in-first-out requirement of its stack. In addition, a Turing machine is also equivalent to a two-stack PDA with standard last-in-first-out semantics, by using one stack to model the right side and the other stack to model the left side of the Turing machine.

At the other extreme, some very simple models turn out to be Turing-equivalent, i.e. to have the same computational power as the Turing machine model.

Common equivalent models are the multi-tape Turing machine, multi-track Turing machine, machines with input and output, and the *non-deterministic* Turing machine (NDTM) as opposed to the *deterministic* Turing machine (DTM) for which the action table has at most one entry for each combination of symbol and state.

Read-only, right-moving Turing machines are equivalent to NDFAs (as well as DFAs by conversion using the NDFA to DFA conversion algorithm).

For practical and didactical intentions the equivalent register machine can be used as a usual assembly programming language.

An interesting question is whether the computation model represented by concrete programming languages is Turing equivalent. While the computation of a real computer is based on finite states and thus not capable to simulate a Turing machine, programming languages themselves do not necessarily have this limitation. Kirner et al., 2009 have shown that among the general-purpose programming languages some are Turing complete while others are not. For example, ANSI C is not Turing-equivalent, as all instantiations of ANSI C (different instantiations are possible as the standard deliberately leaves certain behaviour undefined for legacy reasons) imply a finite-space memory. This is because the size of memory reference data types is accessible inside the language. However, other programming languages like Pascal do not have this feature, which allows them to be Turing complete in principle. It is just Turing complete in principle, as memory allocation in a programming language is allowed to fail, which means the programming language can be Turing complete when ignoring failed memory allocations, but the compiled programs executable on a real computer cannot.

Choice c-machines, oracle o-machines

Early in his paper (1936) Turing makes a distinction between an "automatic machine"—its "motion ... completely determined by the configuration" and a "choice machine":

...whose motion is only partially determined by the configuration ... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems.

— *The Undecidable, p. 118*

Turing (1936) does not elaborate further except in a footnote in which he describes how to use an a-machine to "find all the provable formulae of the [Hilbert] calculus" rather than use a choice machine. He "suppose[s] that the choices are always between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices $i_1, i_2, ..., i_n$ ($i_1 = 0$ or 1, $i_2 = 0$ or 1, ..., $i_n =$ 0 or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + ... + i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ..." (Footnote ‡, *The Undecidable*, p. 138)

This is indeed the technique by which a deterministic (i.e. a-) Turing machine can be used to mimic the action of a nondeterministic Turing machine; Turing solved the matter in a footnote and appears to dismiss it from further consideration.

An oracle machine or o-machine is a Turing a-machine that pauses its computation at state "**o**" while, to complete its calculation, it "awaits the decision" of "the oracle"—an unspecified entity "apart from saying that it cannot be a machine" (Turing (1939), *The Undecidable*, p. 166–168).

Universal Turing machines

Main article: Universal Turing machine

An implementation of a Turing machine

As Turing wrote in *The Undecidable*, p. 128 (italics added):

It is possible to invent a *single machine* which can be used to compute *any* computable sequence. If this machine **U** is supplied with the tape on the beginning of which is written the string of quintuples separated by semicolons of some computing machine **M**, then **U** will compute the same sequence as **M**.

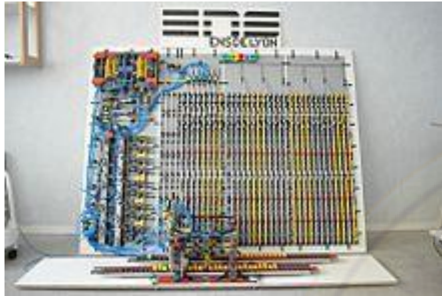
This finding is now taken for granted, but at the time (1936) it was considered astonishing. The model of computation that Turing called his "universal machine"—"**U**" for short—is considered by some (cf. Davis (2000)) to have been the fundamental theoretical breakthrough that led to the notion of the stored-program computer.

Turing's paper ... contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it.

— *Minsky (1967), p. 104*

In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates. This result was obtained in 1966 by F. C. Hennie and R. E. Stearns. (Arora and Barak, 2009, theorem 1.9)

Comparison with real machines



A Turing machine realisation in Lego

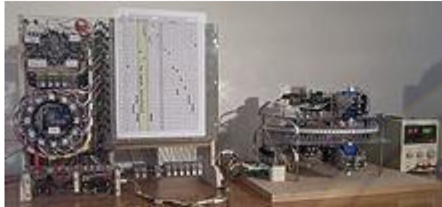
It is often said[*who?*] that Turing machines, unlike simpler automata, are as powerful as real machines, and are able to execute any operation that a real program can. What is neglected in this statement is that, because a real machine can only have a finite number of *configurations*, this "real machine" is really nothing but a linear bounded automaton. On the other hand, Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations. However, Turing machines are not intended to model computers, but rather they are intended to model computation itself. Historically, computers, which compute only on their (fixed) internal storage, were developed only later.

There are a number of ways to explain why Turing machines are useful models of real computers:

1. Anything a real computer can compute, a Turing machine can also compute. For example: "A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms" (Hopcroft and Ullman p. 157). A large enough FSA can also model any real computer, disregarding IO. Thus, a statement about the limitations of Turing machines will also apply to real computers.
2. The difference lies only with the ability of a Turing machine to manipulate an unbounded amount of data. However, given a finite amount of time, a Turing machine (like a real machine) can only manipulate a finite amount of data.
3. Like a Turing machine, a real machine can have its storage space enlarged as needed, by acquiring more disks or other storage media. If the supply of these runs short, the Turing machine may become less useful as a model. But the fact is that neither Turing machines nor real machines need astronomical amounts of storage space in order to perform useful computation. The processing time required is usually much more of a problem.
4. Descriptions of real machine programs using simpler abstract models are often much more complex than descriptions using Turing machines. For example, a Turing machine describing an algorithm may have a few hundred states, while the equivalent deterministic finite automaton

(DFA) on a given real machine has quadrillions. This makes the DFA representation infeasible to analyze.

5. Turing machines describe algorithms independent of how much memory they use. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. Turing machines allow us to make statements about algorithms which will (theoretically) hold forever, regardless of advances in *conventional* computing machine architecture.
6. Turing machines simplify the statement of algorithms. Algorithms running on Turing-equivalent abstract machines are usually more general than their counterparts running on real machines, because they have arbitrary-precision data types available and never have to deal with unexpected conditions (including, but not limited to, running out of memory).



An experimental prototype of a Turing machine

Limitations of Turing machines

Computational complexity theory

Further information: Computational complexity theory

A limitation of Turing machines is that they do not model the strengths of a particular arrangement well. For instance, modern stored-program computers are actually instances of a more specific form of abstract machine known as the random-access stored-program machine or RASP machine model. Like the universal Turing machine the RASP stores its "program" in "memory" external to its finite-state machine's "instructions". Unlike the universal Turing machine, the RASP has an infinite number of distinguishable, numbered but unbounded "registers"—memory "cells" that can contain any integer (cf. Elgot and Robinson (1964), Hartmanis (1971), and in particular Cook-Rechow (1973); references at random access machine). The RASP's finite-state machine is equipped with the capability for indirect addressing (e.g. the contents of one register can be used as an address to specify another register); thus the RASP's "program" can address any register in the register-sequence. The upshot of this distinction is that there are computational optimizations that can be performed based on the memory indices, which are not possible in a general Turing machine; thus when Turing machines are used as the basis for bounding running times, a 'false lower bound' can be proven on certain algorithms' running times (due to the false simplifying assumption of a Turing machine). An example of this is binary search, an algorithm that can be shown to perform more quickly when using the RASP model of computation rather than the Turing machine model.

Concurrency

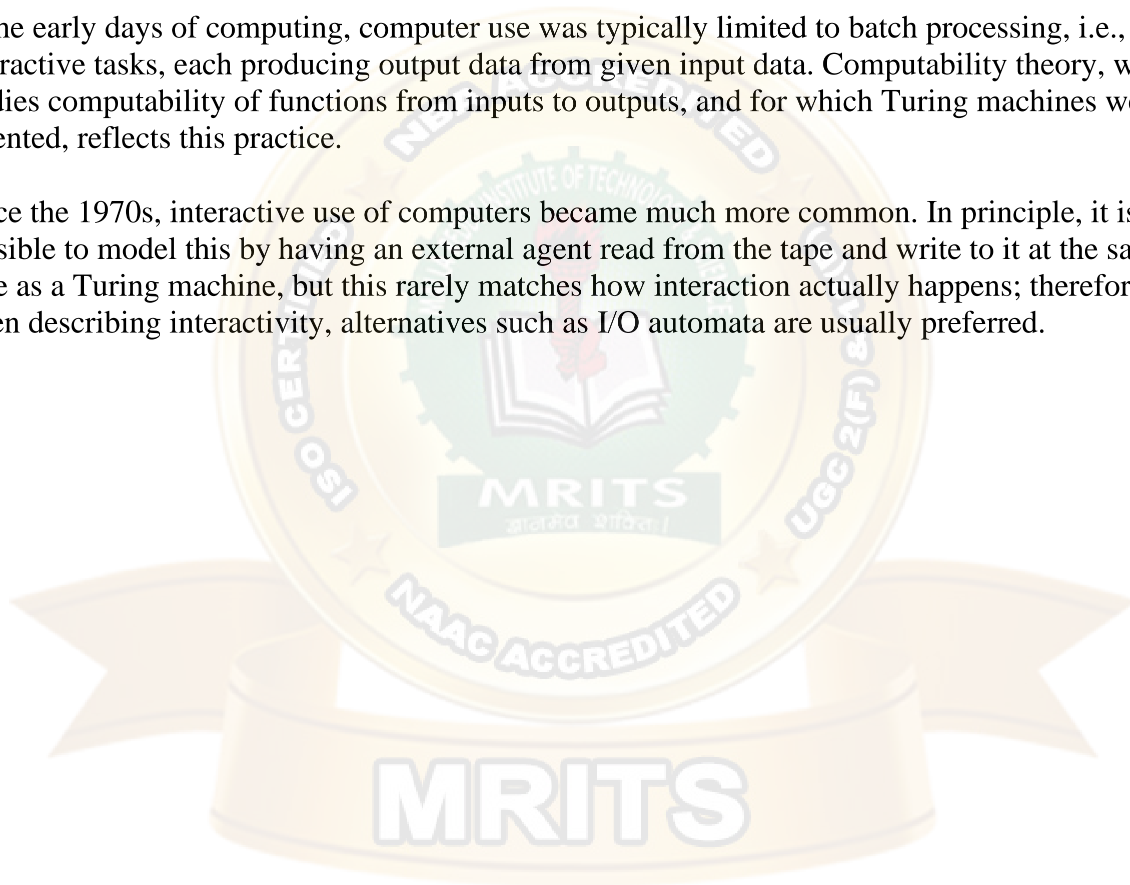
This section **does not cite any sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(April 2015) (Learn how and when to remove this template message)*

Another limitation of Turing machines is that they do not model concurrency well. For example, there is a bound on the size of integer that can be computed by an always-halting nondeterministic Turing machine starting on a blank tape. (See article on unbounded nondeterminism.) By contrast, there are always-halting concurrent systems with no inputs that can compute an integer of unbounded size. (A process can be created with local storage that is initialized with a count of 0 that concurrently sends itself both a stop and a go message. When it receives a go message, it increments its count by 1 and sends itself a go message. When it receives a stop message, it stops with an unbounded number in its local storage.)

Interaction

In the early days of computing, computer use was typically limited to batch processing, i.e., non-interactive tasks, each producing output data from given input data. Computability theory, which studies computability of functions from inputs to outputs, and for which Turing machines were invented, reflects this practice.

Since the 1970s, interactive use of computers became much more common. In principle, it is possible to model this by having an external agent read from the tape and write to it at the same time as a Turing machine, but this rarely matches how interaction actually happens; therefore, when describing interactivity, alternatives such as I/O automata are usually preferred.



MALL REDDY INSTITUTE OF TECHNOLOGY & SCIENCE AND SCIENCE

LECTURE NOTES

On

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE/IT I-Sem

(Jntuh-R18)

CS501PC: FORMAL LANGUAGES AND AUTOMATA THEORY

III Year B.Tech. CSE I-Sem **L T P C**

3 0 0 3

Course Objectives

1. To provide introduction to some of the central ideas of theoretical computer science from the perspective of formal languages.
2. To introduce the fundamental concepts of formal languages, grammars and automata theory.
3. Classify machines by their power to recognize languages.
4. Employ finite state machines to solve problems in computing.
5. To understand deterministic and non-deterministic machines.
6. To understand the differences between decidability and undecidability.

Course Outcomes

1. Able to understand the concept of abstract machines and their power to recognize the languages.
2. Able to employ finite state machines for modeling and solving computing problems.
3. Able to design context free grammars for formal languages.
4. Able to distinguish between decidability and undecidability.

UNIT - I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory – Alphabets, Strings, Languages, Problems.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA, Moore and Melay machines

UNIT - II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages, Statement of the pumping lemma, Applications of the Pumping Lemma.

Closure Properties of Regular Languages: Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

UNIT - III

Context-Free Grammars: Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages. **Push Down Automata**: Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG

UNIT - IV

Normal Forms for Context- Free Grammars: Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form Griebech Normal form.

Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications **Closure Properties of Context-Free Languages**: Closure properties of CFL's, Decision Properties of CFL's

Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

UNIT - V

Types of Turing machine: Turing machines and halting

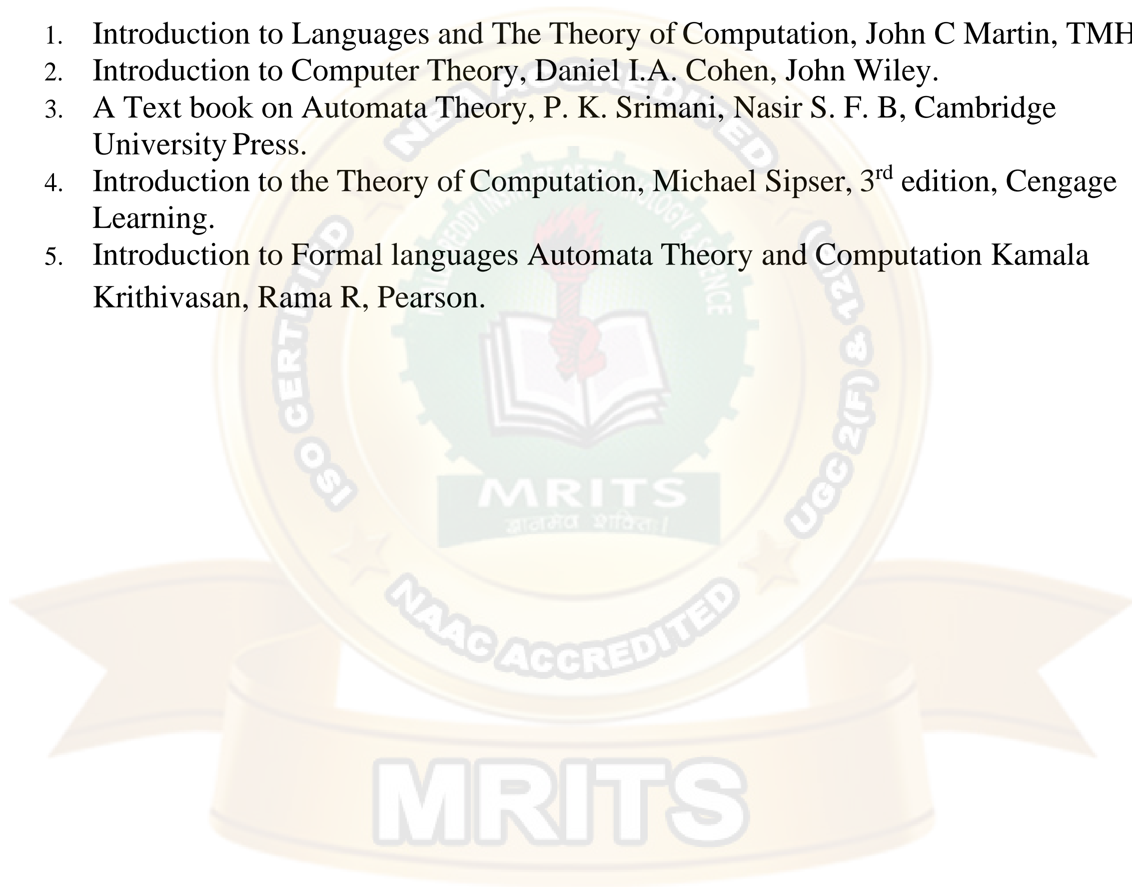
Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem, Other Undecidable Problems, Counter machines

TEXT BOOKS:

1. Introduction to Automata Theory, Languages, and Computation, 3nd Edition, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science – Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCE BOOKS:

1. Introduction to Languages and The Theory of Computation, John C Martin, TMH.
2. Introduction to Computer Theory, Daniel I.A. Cohen, John Wiley.
3. A Text book on Automata Theory, P. K. Srimani, Nasir S. F. B, Cambridge University Press.
4. Introduction to the Theory of Computation, Michael Sipser, 3rd edition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and Computation Kamala Krithivasan, Rama R, Pearson.



UNIT-V

Undecidable problem

From Wikipedia, the free encyclopedia

In computability theory and computational complexity theory, an undecidable problem is a decision problem for which it is known to be impossible to construct a single algorithm that always leads to a correct yes-or-no answer. The halting problem is an example: there is no algorithm that correctly determines whether arbitrary programs eventually halt when run.

Background

A decision problem is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as the set of inputs for which the problem returns yes. These inputs can be natural numbers, but also other values of some other kind, such as strings of a formal language. Using some encoding, such as a Gödel numbering, the strings can be encoded as natural numbers. Thus, a decision problem informally phrased in terms of a formal language is also equivalent to a set of natural numbers. To keep the formal definition simple, it is phrased in terms of subsets of the natural numbers.

Formally, a decision problem is a subset of the natural numbers. The corresponding informal problem is that of deciding whether a given number is in the set. A decision problem A is called decidable or effectively solvable if A is a recursive set. A problem is called partially decidable, semi-decidable, solvable, or provable if A is a recursively enumerable set. This means that there exists an algorithm that halts eventually when the answer is yes but may run for ever if the answer is no. Partially decidable problems and any other problems that are not decidable are called undecidable.

Example: the halting problem in computability theory

In computability theory, the halting problem is a decision problem which can be stated as follows:

> *Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever.*

Alan Turing proved in 1936 that a general algorithm running on a Turing machine that solves the halting problem for all possible program-input pairs necessarily cannot exist. Hence, the halting problem is undecidable for Turing machines.

Non-Recursively Enumerable Languages

A language is a subset of Σ^*. A language is any subset of Σ^*.

We have shown that Turing machines are enumerable. Since recursively enumerable languages are those whose strings are accepted by a Turing machine, the set of recursively enumerable languages is also enumerable.

We have shown that the powerset of an infinite set is not enumerable -- that it has more than $\aleph_0$ subsets. Each of these subsets represents a language. Therefore, there must be languages that are not computable by a Turing machine.

According to Turing's thesis, a Turing machine can compute any effective procedure. Therefore, there are languages that cannot be defined by any effective procedure.

We can find a non-recursively enumerable language X by diagonalization. Using the enumerations described earlier, let string i belong to language X if and only if it does not belong to language i.

Problem. I've just defined a procedure for defining a non-recursively enumerable language. Isn't this a contradiction?

When Recursively Enumerable Implies Recursive

Suppose a language L is recursively enumerable. That means there exists a Turing machine T_1 that, given any string of the language, halts and accepts that string. (We don't know what it will do for strings not in the language -- it could reject them, or it could simply never halt.)

Now let's also suppose that the complement of L, -L = {w: w $\notin$L}, is recursively enumerable. That means there is some other Turing machine T_2 that, given any string of -L, halts and accepts that string.

Clearly, any string (over the appropriate alphabet Σ) belongs to either L or -L. Hence, any string will cause either T_1 or T_2 (or both) to halt. We construct a new Turing machine that emulates both T_1 and T_2, alternating moves between them. When either one stops, we can tell (by whether it accepted or rejected the string) to which language the string belongs. Thus, we have constructed a Turing machine that, for each input, halts with an answer whether or not the string belongs to L. Therefore L and -L are recursive languages.

We have just proved the following theorem: If a language and its complement are both recursively enumerable, then both are recursive.

Recursively Enumerable But Not Recursive

We know that the recursive languages are a subset of the recursively enumerable languages, We will now show that they are a proper subset.

We have shown how to enumerate strings for a given alphabet, w_1, w_2, w_3,..... We have also shown how to enumerate Turing machines, T_1, T_2, T_3,..... (Recall that each Turing machine defines a recursively enumerable language.) Consider the language

$$L = \{w_i: w_i \in L(T_i)\}$$

A little thought will show that L is itself recursively enumerable. But now consider its complement:

$$-L = \{w_i: w_i \notin L(T_i)\}$$

If -L is recursively enumerable, then there must exist a Turing machine that recognizes it. This Turing machine must be in the enumeration somewhere -- call it T_k.

Does w_k belong to L?

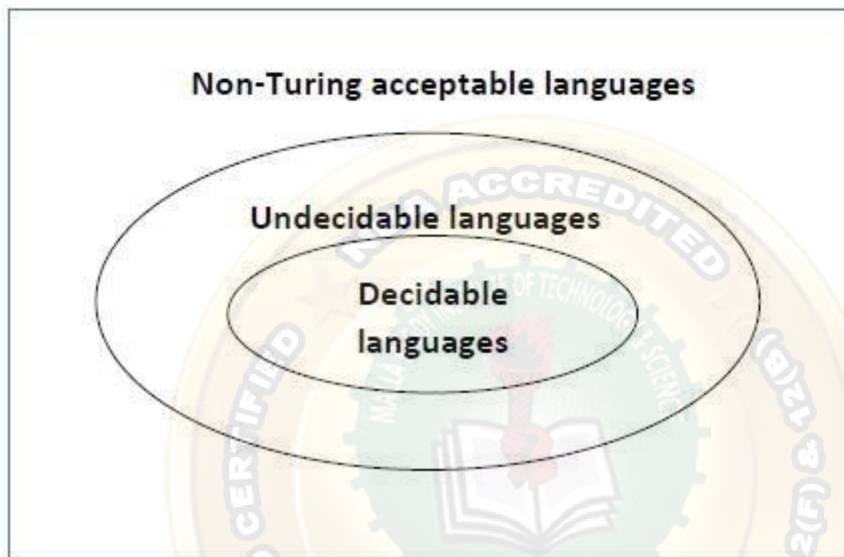
- *If w_k belongs to L then (by the way we have defined L) T_k accepts this string. But T_k accepts only strings that do not belong to L, so we have a contradiction.*
- *If w_k does not belong to L, then it belongs to -L and is accepted by T_k. But since T_k accepts w_k, w_k must belong to L. Again, a contradiction.*

We have now defined a recursively enumerable language L and shown by contradiction that -L is not recursively enumerable.

We mentioned earlier that if a language is recursive, its complement must also be recursive. If language L above were recursive, then -L would also be recursive, hence recursively enumerable. But -L is not recursively enumerable; therefore L must not be recursive.

In computability theory and computational complexity theory, an undecidable problem is a decision problem for which it is known to be impossible to construct a single algorithm that always leads to a correct yes-or-no answer. The halting problem is an example: there is no algorithm that correctly determines whether arbitrary programs eventually halt when run.

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string w (TM can make decision for some input string though). A decision problem P is called "undecidable" if the language L of all yes instances to P is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.



Example

- *The halting problem of Turing machine*
- *The mortality problem*
- *The mortal matrix problem*
- *The Post correspondence problem, etc.*

Undecidable Problems

- *Decida Recall that:*
- *A language is decidable,*
- *if there is a Turing machine (decider)*
- *that accepts the language and halts on every input string ble Languages.*
-
- *For an undecidable language,*
- *the corresponding problem is*
- *undecidable (unsolvable):*

MRITS

- *there is no Turing Machine (Algorithm)*
- *that gives an answer (yes or no)*
- *for every input instance*

- *Post's Correspondence Problem (PCP)*
- *A post correspondence system consists of a finite set of ordered pairs* $(x_i, y_i),\ i = 1, 2, \cdots, n,$ *where* $x_i, y_i \in \Sigma^+$ *for some alphabet* Σ.
- *Any sequence of numbers* $i_1, i_2, \cdots, i_k \quad s-t.$
- *is called a solution to a Post Correspondence System.*
- $x_{i_1}, x_{i_2}, \cdots, x_{i_k} = y_{i_1}, y_{i_2}, \cdots, y_{i_k}$ *The Post's Correspondence Problem is the problem of determining whether a Post Correspondence system has a solutions.*
- *Example 1 : Consider the post correspondence system*
- $\{(aa, aab), (bb, ba), (abb, b)\}$ *The list 1,2,1,3 is a solution to it.*
- *Because*
-
-
-
- $x_1 x_2 x_1 x_3 = y_1 y_2 y_1 y_3$
-

$$\underbrace{aa}_{x_1}\underbrace{bb}_{x_2}\underbrace{aa}_{x_1}\underbrace{abb}_{x_3} = \underbrace{aab}_{y_1}\underbrace{ba}_{y_2}\underbrace{aab}_{y_1}\underbrace{b}_{y_3}$$

$$aabbaaabb = aabbaaabb$$

i	x_i	y_i
1	aa	aab
2	bb	ba
3	abb	b

- *(A post correspondence system is also denoted as an instance of the PCP)*
- *Example 2 : The following PCP instance has no solution*

i	x_i	y_i
1	aab	aa
2	a	baa

-
-
-
-
-
- *his can be proved as follows.* (x_2, y_2) *cannot be chosen at the start, since than the LHS and RHS would differ in the first symbol (* a *in LHS and* $'b'$ *in RHS). So, we must start*

with (x_1, y_1) *. The next pair must be* (x_2, y_2) *so that the 3 rd symbol in the RHS becomes identical to that of the LHS, which is a . After this step, LHS and RHS are not matching. If* (x_1, y_1) *is selected next, then would be mismatched in the 7 th symbol (* b *in LHS and* a *in RHS). If* (x_2, y_2) *is selected, instead, there will not be any choice to match the both side in the next step.*

- *Example3 : The list 1,3,2,3 is a solution to the following PCP instance.*

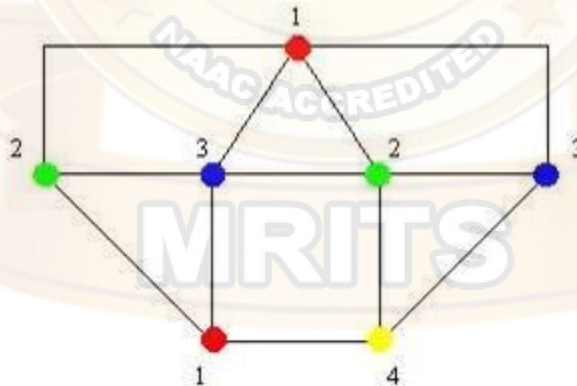
i	x_i	y_i
1	*1*	*101*
2	*10*	*00*
3	*011*	*11*

- *The following properties can easily be proved.*
- *Proposition The Post Correspondence System*
- *Intractable problems*
- *From a computational complexity stance, intractable problems are problems for which there exist no efficient algorithms to solve them.*
- *Most intractable problems have an algorithm – the same algorithm – that provides a solution, and that algorithm is the brute-force search.*
-
- *This algorithm, however, does not provide an efficient solution and is, therefore, not feasible for computation with anything more than the smallest input.*
- *The reason there is no efficient algorithms for these problems is that these problems are all in a category which I like to refer to as "slightly less than random." They are so close to random, in fact, that they do not as yet allow for any meaningful algorithm other than that of brute-force.*
- *If any problem were truly random, there would not be even the possibility of any such algorithm.*

- *EXAMPLE #1: Factoring a number into primes.*
-
- *It is the security provided by the lack of any efficient algorithm to solve this problem that allows RSA and public key encryption to be so widely accepted and successful.*
-

- *EXAMPLE #2: SAT, the satisfiability problem to test whether a given Boolean*
- *formula is satisfiable.*
- *All sets of input must be tried systematically (brute-force) until a satisfying case is discovered.*
- *EXAMPLE #3: Integer partition: Can you partition n integers into two subsets such*
- *that the sums of the subsets are equal?*
- *As the size of the integers (i.e. the size of n) grows linearly, the size of the computations required to check all subsets and their respective sums grows exponentially. This is because, once again, we are forced to use the brute-force method to test the subsets of each division and their sums.*

- *EXAMPLE #4: Graph coloring: How many colors do you need to color a graph*
- *such that no two adjacent vertices are of the same color?*



- *Given any graph with a large number of vertices, we see that we are again faced with resorting to a systematic tracing of all paths, comparison of neighboring colors, backtracking, etc., resulting in exponential time complexity once again.*

- *EXAMPLE #5: Bin packing: How many bins of a given size do you need to hold n*

- *items of variable size?*
-
- *Again, the best algorithm for this problem involves going through all subsets of n items, seeing how they fit into the bins, and backtracking to test for better fits among subsets until all possibles subsets have been tested to achieve the proper answer. Once again, brute-force. Once again, exponential.*

P versus NP problem

The P versus NP problem is a major unsolved problem in computer science. It asks whether every problem whose solution can be quickly verified (technically, verified in polynomial time) can also be solved quickly (again, in polynomial time).

The underlying issues were first discussed in the 1950s, in letters from John Forbes Nash Jr. to the National Security Agency, and from Kurt Gödel to John von Neumann. The precise statement of the P versus NP problem was introduced in 1971 by Stephen Cook in his seminal paper "The complexity of theorem proving procedures"[2] and is considered by many to be the most important open problem in the field.[3] It is one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute to carry a US$1,000,000 prize for the first correct solution.

The informal term quickly, used above, means the existence of an algorithm solving the task that runs in polynomial time, such that the time to complete the task varies as a polynomial function on the size of the input to the algorithm (as opposed to, say, exponential time). The general class of questions for which some algorithm can provide an answer in polynomial time is called "class P" or just "P". For some questions, there is no known way to find an answer quickly, but if one is provided with information showing what the answer is, it is possible to verify the answer quickly. The class of questions for which an answer can be verified in polynomial time is called NP, which stands for "nondeterministic polynomial time".[Note 1]

Consider Sudoku, an example of a problem that is easy to verify, but whose answer may be difficult to compute. Given a partially filled-in Sudoku grid, of any size, is there at least one legal solution? A proposed solution is easily verified, and the time to check a solution grows slowly (polynomially) as the grid gets bigger. However, all known algorithms for finding solutions take, for difficult examples, time that grows exponentially as the grid gets bigger. So Sudoku is in NP (quickly checkable) but does not seem to be in P (quickly solvable). Thousands of other problems seem similar, fast to check but slow to solve. Researchers have shown that a fast solution to any one of these problems could be used to build a quick solution to all the others, a property called NP-completeness. Decades of searching have not yielded a fast solution to any of these problems, so most scientists suspect that none of these problems can be solved quickly. However, this has never been proven.

An answer to the P = NP question would determine whether problems that can be verified in polynomial time, like Sudoku, can also be solved in polynomial time. If it turned out that P ≠ NP, it would mean that there are problems in NP that are harder to compute than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time.

Aside from being an important problem in computational theory, a proof either way would have profound implications for mathematics, cryptography, algorithm research, artificial intelligence, game theory, multimedia processing, philosophy, economics and many other fields

Context

The relation between the complexity classes P and NP is studied in computational complexity theory, the part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps it takes to solve a problem) and space (how much memory it takes to solve a problem).

In such analysis, a model of the computer for which time must be analyzed is required. Typically such models assume that the computer is *deterministic* (given the computer's present state and any inputs, there is only one possible action that the computer might take) and *sequential* (it performs actions one after the other).

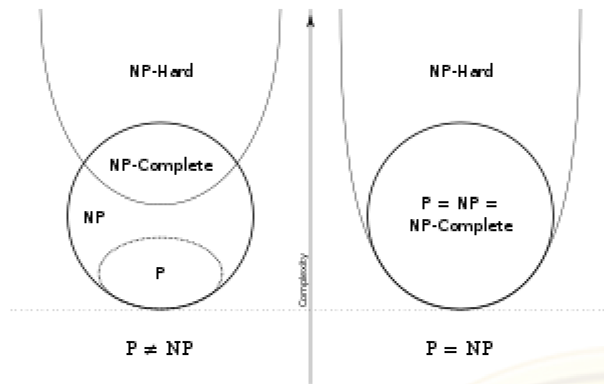
In this theory, the class P consists of all those *decision problems* (defined below) that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input; the class NP consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine.[7] Clearly, $P \subseteq NP$. Arguably the biggest open question in theoretical computer science concerns the relationship between those two classes:

> Is P equal to NP?

In a 2002 poll of 100 researchers, 61 believed the answer to be no, 9 believed the answer is yes, and 22 were unsure; 8 believed the question may be independent of the currently accepted axioms and therefore impossible to prove or disprove.[8]

In 2012, 10 years later, the same poll was repeated. The number of researchers who answered was 151: 126 (83%) believed the answer to be no, 12 (9%) believed the answer is yes, 5 (3%) believed the question may be independent of the currently accepted axioms and therefore impossible to prove or disprove, 8 (5%) said either don't know or don't care or don't want the answer to be yes nor the problem to be resolved.[9]

NP-completeness



Euler diagram for P, NP, NP-complete, and NP-hard set of problems (excluding the empty language and its complement, which belong to P but are not NP-complete)

Main article: NP-completeness

To attack the P = NP question, the concept of NP-completeness is very useful. NP-complete problems are a set of problems to each of which any other NP-problem can be reduced in polynomial time, and whose solution may still be verified in polynomial time. That is, any NP problem can be transformed into any of the NP-complete problems. Informally, an NP-complete problem is an NP problem that is at least as "tough" as any other problem in NP.

NP-hard problems are those at least as hard as NP problems, i.e., all NP problems can be reduced (in polynomial time) to them. NP-hard problems need not be in NP, i.e., they need not have solutions verifiable in polynomial time.

For instance, the Boolean satisfiability problem is NP-complete by the Cook–Levin theorem, so *any* instance of *any* problem in NP can be transformed mechanically into an instance of the Boolean satisfiability problem in polynomial time. The Boolean satisfiability problem is one of many such NP-complete problems. If any NP-complete problem is in P, then it would follow that P = NP. However, many important problems have been shown to be NP-complete, and no fast algorithm for any of them is known.

Based on the definition alone it is not obvious that NP-complete problems exist; however, a trivial and contrived NP-complete problem can be formulated as follows: given a description of a Turing machine M guaranteed to halt in polynomial time, does there exist a polynomial-size input that M will accept?[10] It is in NP because (given an input) it is simple to check whether M accepts the input by simulating M; it is NP-complete because the verifier for any particular instance of a problem in NP can be encoded as a polynomial-time machine M that takes the solution to be verified as input. Then the question of whether the instance is a yes or no instance is determined by whether a valid input exists.

The first natural problem proven to be NP-complete was the Boolean satisfiability problem. As noted above, this is the Cook–Levin theorem; its proof that satisfiability is NP-complete contains technical details about Turing machines as they relate to the definition of NP. However, after this

problem was proved to be NP-complete, proof by reduction provided a simpler way to show that many other problems are also NP-complete, including the Sudoku discussed earlier. In this case, the proof shows that a solution of Sudoku in polynomial time, could also be used to complete Latin squares in polynomial time.[11] This in turn gives a solution to the problem of partitioning tri-partitite graphs into triangles,[12] which could then be used to find solutions for 3-sat,[13] which then provides a solution for general boolean satisfiability. So a polynomial time solution to Sudoku leads, by a series of mechanical transformations, to a polynomial time solution of satisfiability, which in turn can be used to solve any other NP-complete problem in polynomial time. Using transformations like this, a vast class of seemingly unrelated problems are all reducible to one another, and are in a sense "the same problem".

Harder problems

See also: Complexity class

Although it is unknown whether P = NP, problems outside of P are known. A number of succinct problems (problems that operate not on normal input, but on a computational description of the input) are known to be EXPTIME-complete. Because it can be shown that P ≠ EXPTIME, these problems are outside P, and so require more than polynomial time. In fact, by the time hierarchy theorem, they cannot be solved in significantly less than exponential time. Examples include finding a perfect strategy for chess (on an $N \times N$ board). and some other board games. The problem of deciding the truth of a statement in Presburger arithmetic requires even more time. Fischer and Rabin proved in 1974 that every algorithm that decides the truth of Presburger

statements of length n has a runtime of at least for some constant c. Hence, the problem is known to need more than exponential run time. Even more difficult are the undecidable problems, such as the halting problem. They cannot be completely solved by any algorithm, in the sense that for any particular algorithm there is at least one input for which that algorithm will not produce the right answer; it will either produce the wrong answer, finish without giving a conclusive answer, or otherwise run forever without producing any answer at all.

It is also possible to consider questions other than decision problems. One such class, consisting of counting problems, is called #P: whereas an NP problem asks "Are there any solutions?", the corresponding #P problem asks "How many solutions are there?" Clearly, a #P problem must be at least as hard as the corresponding NP problem, since a count of solutions immediately tells if at least one solution exists, if the count is greater than zero. Surprisingly, some #P problems that are believed to be difficult correspond to easy (for example linear-time) P problems.[17] For these problems, it is very easy to tell whether solutions exist, but thought to be very hard to tell how many. Many of these problems are #P-complete, and hence among the hardest problems in #P, since a polynomial time solution to any of them would allow a polynomial time solution to all other #P problems.

Problems in NP not known to be in P or NP-complete

Main article: NP-intermediate

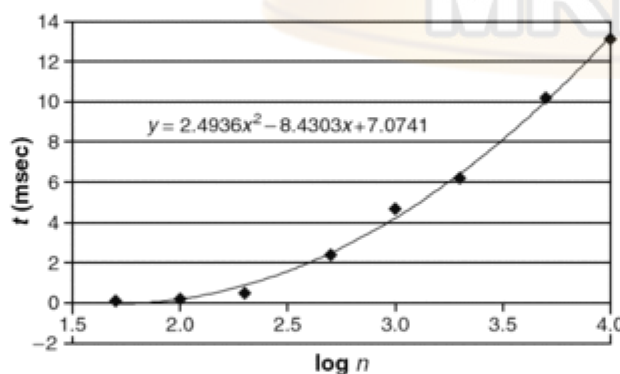
It was shown by Ladner that if P ≠ NP then there exist problems in NP that are neither in P nor NP-complete. Such problems are called NP-intermediate problems. The graph isomorphism problem, the discrete logarithm problem and the integer factorization problem are examples of problems believed to be NP-intermediate. They are some of the very few NP problems not known to be in P or to be NP-complete.

The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic. An important unsolved problem in complexity theory is whether the graph isomorphism problem is in P, NP-complete, or NP-intermediate. The answer is not known, but it is believed that the problem is at least not NP-complete If graph isomorphism is NP-complete, the polynomial time hierarchy collapses to its second level. Since it is widely believed that the polynomial hierarchy does not collapse to any finite level, it is believed that graph isomorphism is not NP-complete. The best algorithm for this problem, due to László Babai and Eugene Luks, has run time $2^{O(\sqrt{n \log n})}$ for graphs with n vertices.

The integer factorization problem is the computational problem of determining the prime factorization of a given integer. Phrased as a decision problem, it is the problem of deciding whether the input has a factor less than k. No efficient integer factorization algorithm is known, and this fact forms the basis of several modern cryptographic systems, such as the RSA algorithm. The integer factorization problem is in NP and in co-NP (and even in UP and co-UP). If the problem is NP-complete, the polynomial time hierarchy will collapse to its first level (i.e., NP = co-NP). The best known algorithm for integer factorization is the general number field sieve, which takes expected time

to factor an n-bit integer. However, the best known quantum algorithm for this problem, Shor's algorithm, does run in polynomial time, although this does not indicate where the problem lies with respect to non-quantum complexity classes.

Does P mean "easy"?



The graph shows time (average of 100 instances in ms using a 933 MHz Pentium III) vs.problem size for knapsack problems for a state-of-the-art specialized algorithm. Quadratic fit suggests that empirical algorithmic complexity for instances with 50–10,000 variables is $O((\log(n))^{2}$.

All of the above discussion has assumed that P means "easy" and "not in P" means "hard", an assumption known as *Cobham's thesis*. It is a common and reasonably accurate assumption in complexity theory; however, it has some caveats.

First, it is not always true in practice. A theoretical polynomial algorithm may have extremely large constant factors or exponents thus rendering it impractical. On the other hand, even if a problem is shown to be NP-complete, and even if P ≠ NP, there may still be effective approaches to tackling the problem in practice. There are algorithms for many NP-complete problems, such as the knapsack problem, the traveling salesman problem and the Boolean satisfiability problem, that can solve to optimality many real-world instances in reasonable time. The empirical average-case complexity (time vs. problem size) of such algorithms can be surprisingly low. An example is the simplex algorithm in linear programming, which works surprisingly well in practice; despite having exponential worst-case time complexity it runs on par with the best known polynomial-time algorithms.[23]

Second, there are types of computations which do not conform to the Turing machine model on which P and NP are defined, such as quantum computation and randomized algorithms.

Reasons to believe P ≠ NP

According to polls,[8][24] most computer scientists believe that P ≠ NP. A key reason for this belief is that after decades of studying these problems no one has been able to find a polynomial-time algorithm for any of more than 3000 important known NP-complete problems (see List of NP-complete problems). These algorithms were sought long before the concept of NP-completeness was even defined (Karp's 21 NP-complete problems, among the first found, were all well-known existing problems at the time they were shown to be NP-complete). Furthermore, the result P = NP would imply many other startling results that are currently believed to be false, such as NP = co-NP and P = PH.

It is also intuitively argued that the existence of problems that are hard to solve but for which the solutions are easy to verify matches real-world experience.]

If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found.

— *Scott Aaronson, MIT*

On the other hand, some researchers believe that there is overconfidence in believing P ≠ NP and that researchers should explore proofs of P = NP as well. For example, in 2002 these statements were made: The main argument in favor of P ≠ NP is the total lack of fundamental progress in

the area of exhaustive search. This is, in my opinion, a very weak argument. The space of algorithms is very large and we are only at the beginning of its exploration. [...] The resolution of Fermat's Last Theorem also shows that very simple questions may be settled only by very deep theories.

— *Moshe Y. Vardi, Rice University*

Being attached to a speculation is not a good guide to research planning. One should always try both directions of every problem. Prejudice has caused famous mathematicians to fail to solve famous problems whose solution was opposite to their expectations, even though they had developed all the methods required.

— *Anil Nerode, Cornell University*

Consequences of solution

One of the reasons the problem attracts so much attention is the consequences of the answer. Either direction of resolution would advance theory enormously, and perhaps have huge practical consequences as well.

P = NP

A proof that P = NP could have stunning practical consequences if the proof leads to efficient methods for solving some of the important problems in NP. It is also possible that a proof would not lead directly to efficient methods, perhaps if the proof is non-constructive, or the size of the bounding polynomial is too big to be efficient in practice. The consequences, both positive and negative, arise since various NP-complete problems are fundamental in many fields.

Cryptography, for example, relies on certain problems being difficult. A constructive and efficient solutionto an NP-complete problem such as 3-SAT would break most existing cryptosystems including:

- Existing implementations of public-key cryptography, a foundation for many modern security applications such as secure financial transactions over the Internet.
- Symmetric ciphers such as AES or 3DES, used for the encryption of communications data.
- Cryptographic hashing as the problem of finding a pre-image that hashes to a given values must be difficult to be useful, and ideally should require exponential time. However, if P=NP, then finding a pre-image M can be done in polynomial time, through reduction to SAT. These would need to be modified or replaced by information-theoretically secure solutions not inherently based on P-NP equivalence.

On the other hand, there are enormous positive consequences that would follow from rendering tractable many currently mathematically intractable problems. For instance, many problems in operations research are NP-complete, such as some types of integer programming and the travelling salesman problem. Efficient solutions to these problems would have enormous implications for logistics. Many other important problems, such as some problems in protein

structure prediction, are also NP-complete;[30] if these problems were efficiently solvable it could spur considerable advances in life sciences and biotechnology.

But such changes may pale in significance compared to the revolution an efficient method for solving NP-complete problems would cause in mathematics itself. Gödel, in his early thoughts on computational complexity, noted that a mechanical method that could solve any problem would revolutionize mathematics:[31][32]

If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem.

Similarly, Stephen Cook says[33]

...it would transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of a reasonable length, since formal proofs can easily be recognized in polynomial time. Example problems may well include all of the CMI prize problems.

Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to find after problems have been stated—for instance, Fermat's Last Theorem took over three centuries to prove. A method that is guaranteed to find proofs to theorems, should one exist of a "reasonable" size, would essentially end this struggle.

Donald Knuth has stated that he has come to believe that P = NP, but is reserved about the impact of a possible proof:[34]

[...] I don't believe that the equality P = NP will turn out to be helpful even if it is proved, because such a proof will almost surely be nonconstructive.

P ≠ NP

A proof that showed that P ≠ NP would lack the practical computational benefits of a proof that P = NP, but would nevertheless represent a very significant advance in computational complexity theory and provide guidance for future research. It would allow one to show in a formal way that many common problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems. Due to widespread belief in P ≠ NP, much of this focusing of research has already taken place.

Also P ≠ NP still leaves open the average-case complexity of hard problems in NP. For example, it is possible that SAT requires exponential time in the worst case, but that almost all randomly selected instances of it are efficiently solvable. Russell Impagliazzo has described five hypothetical "worlds" that could result from different possible resolutions to the average-case complexity question. These range from "Algorithmica", where P = NP and problems like SAT

can be solved efficiently in all instances, to "Cryptomania", where P ≠ NP and generating hard instances of problems outside P is easy, with three intermediate possibilities reflecting different possible distributions of difficulty over instances of NP-hard problems. The "world" where P ≠ NP but all problems in NP are tractable in the average case is called "Heuristica" in the paper. A Princeton University workshop in 2009 studied the status of the five worlds.

Results about difficulty of proof

Although the P = NP problem itself remains open despite a million-dollar prize and a huge amount of dedicated research, efforts to solve the problem have led to several new techniques. In particular, some of the most fruitful research related to the P = NP problem has been in showing that existing proof techniques are not powerful enough to answer the question, thus suggesting that novel technical approaches are required.

As additional evidence for the difficulty of the problem, essentially all known proof techniques in computational complexity theory fall into one of the following classifications, each of which is known to be insufficient to prove that P ≠ NP:

Classification	Definition
Relativizing proofs	Imagine a world where every algorithm is allowed to make queries to some fixed subroutine called an oracle (a black box which can answer a fixed set of questions in constant time. For example, a black box that solves any given travelling salesman problem in 1 step), and the running time of the oracle is not counted against the running time of the algorithm. Most proofs (especially classical ones) apply uniformly in a world with oracles regardless of what the oracle does. These proofs are called *relativizing*. In 1975, Baker, Gill, and Solovay showed that P = NP with respect to some oracles, while P ≠ NP for other oracles.[38] Since relativizing proofs can only prove statements that are uniformly true with respect to all possible oracles, this showed that relativizing techniques cannot resolve P = NP.
Natural proofs	In 1993, Alexander Razborov and Steven Rudich defined a general class of proof techniques for circuit complexity lower bounds, called *natural proofs*.[39] At the time all previously known circuit lower bounds were natural, and circuit complexity was considered a very promising approach for resolving P = NP. However, Razborov and Rudich showed that, if one-way functions exist, then no natural proof method can distinguish between P and NP. Although one-way functions have never been formally proven to exist, most mathematicians believe that they do, and a proof of their existence would be a much stronger statement than P ≠ NP. Thus it is unlikely that natural proofs alone can resolve P = NP.
Algebrizing proofs	After the Baker-Gill-Solovay result, new non-relativizing proof techniques were successfully used to prove that IP = PSPACE. However, in 2008, Scott Aaronson and Avi Wigderson showed that the main technical tool used in the IP = PSPACE proof, known as

arithmetization, was also insufficient to resolve P = NP.[40]

These barriers are another reason why NP-complete problems are useful: if a polynomial-time algorithm can be demonstrated for an NP-complete problem, this would solve the P = NP problem in a way not excluded by the above results.

These barriers have also led some computer scientists to suggest that the P versus NP problem may be independent of standard axiom systems like ZFC (cannot be proved or disproved within them). The interpretation of an independence result could be that either no polynomial-time algorithm exists for any NP-complete problem, and such a proof cannot be constructed in (e.g.) ZFC, or that polynomial-time algorithms for NP-complete problems may exist, but it is impossible to prove in ZFC that such algorithms are correct. However, if it can be shown, using techniques of the sort that are currently known to be applicable, that the problem cannot be decided even with much weaker assumptions extending the Peano axioms (PA) for integer arithmetic, then there would necessarily exist nearly-polynomial-time algorithms for every problem in NP. Therefore, if one believes (as most complexity theorists do) that not all problems in NP have efficient algorithms, it would follow that proofs of independence using those techniques cannot be possible. Additionally, this result implies that proving independence from PA or ZFC using currently known techniques is no easier than proving the existence of efficient algorithms for all problems in NP.

Claimed solutions

While the P versus NP problem is generally considered unsolved, many amateur and some professional researchers have claimed solutions. Gerhard J. Woeginger has a comprehensive list. As of 2018, this list contained 62 purported proofs of P = NP, 50 of P ≠ NP, 2 proofs the problem is unprovable, and one proof that it is undecidable. An August 2010 claim of proof that P ≠ NP, by Vinay Deolalikar, a researcher at HP Labs, received heavy Internet and press attention after being initially described as "seem[ing] to be a relatively serious attempt" by two leading specialists. The proof has been reviewed publicly by academics, and Neil Immerman, an expert in the field, has pointed out two possibly fatal errors in the proof. In September 2010, Deolalikar was reported to be working on a detailed expansion of his attempted proof. However, opinions expressed by several notable theoretical computer scientists indicate that the attempted proof is neither correct nor a significant advancement in the understanding of the problem. This assessment prompted a May 2013 *The New Yorker* article to call the proof attempt "thoroughly discredited".

Logical characterizations

The P = NP problem can be restated in terms of expressible certain classes of logical statements, as a result of work in descriptive complexity.

Consider all languages of finite structures with a fixed signature including a linear order relation. Then, all such languages in P can be expressed in first-order logic with the addition of a suitable least fixed-point combinator. Effectively, this, in combination with the order, allows the definition of recursive functions. As long as the signature contains at least one predicate or function in addition to the distinguished order relation, so that the amount of space taken to store such finite structures is actually polynomial in the number of elements in the structure, this precisely characterizes P.

Similarly, NP is the set of languages expressible in existential second-order logic—that is, second-order logic restricted to exclude universal quantification over relations, functions, and subsets. The languages in the polynomial hierarchy, PH, correspond to all of second-order logic. Thus, the question "is P a proper subset of NP" can be reformulated as "is existential second-order logic able to describe languages (of finite linearly ordered structures with nontrivial signature) that first-order logic with least fixed point cannot?".[52] The word "existential" can even be dropped from the previous characterization, since P = NP if and only if P = PH (as the former would establish that NP = co-NP, which in turn implies that NP = PH).

Polynomial-time algorithms

No algorithm for any NP-complete problem is known to run in polynomial time. However, there are algorithms known for NP-complete problems with the property that if P = NP, then the algorithm runs in polynomial time on accepting instances (although with enormous constants, making the algorithm impractical). However, these algorithms do not qualify as polynomial time because their running time on rejecting instances are not polynomial. The following algorithm, due to Levin (without any citation), is such an example below. It correctly accepts the NP-complete language SUBSET-SUM. It runs in polynomial time on inputs that are in SUBSET-SUM if and only if P = NP:

```
// Algorithm that accepts the NP-complete language SUBSET-SUM.
//
// this is a polynomial-time algorithm if and only if P = NP.
//
// "Polynomial-time" means it returns "yes" in polynomial time when
// the answer should be "yes", and runs forever when it is "no".
//
// Input: S = a finite set of integers
// Output: "yes" if any subset of S adds up to 0.
// Runs forever with no output otherwise.
// Note: "Program number P" is the program obtained by
// writing the integer P in binary, then
// considering that string of bits to be a
// program. Every possible program can be
// generated this way, though most do nothing
// because of syntax errors.
FOR N = 1...∞
  FOR P = 1...N
    Run program number P for N steps with input S
    IF the program outputs a list of distinct integers
    AND the integers are all in S
```

```
    AND the integers sum to 0
  THEN
    OUTPUT "yes" and HALT
```

If, and only if, P = NP, then this is a polynomial-time algorithm accepting an NP-complete language. "Accepting" means it gives "yes" answers in polynomial time, but is allowed to run forever when the answer is "no" (also known as a *semi-algorithm*).

This algorithm is enormously impractical, even if P = NP. If the shortest program that can solve SUBSET-SUM in polynomial time is b bits long, the above algorithm will try at least 2^b-1 other programs first.

Formal definitions

P and NP

Conceptually speaking, a *decision problem* is a problem that takes as input some string w over an alphabet Σ, and outputs "yes" or "no". If there is an algorithm (say a Turing machine, or a computer program with unbounded memory) that can produce the correct answer for any input string of length n in at most cn^k steps, where k and c are constants independent of the input string, then we say that the problem can be solved in *polynomial time* and we place it in the class P. Formally, P is defined as the set of all languages that can be decided by a deterministic

polynomial-time Turing machine. That is,

where

and a deterministic polynomial-time Turing machine is a deterministic Turing machine M that satisfies the following two conditions:

1. M halts on all input w and

2. there exists such that , where O refers to the big O notation and

NP can be defined similarly using nondeterministic Turing machines (the traditional way). However, a modern approach to define NP is to use the concept of *certificate* and *verifier*. Formally, NP is defined as the set of languages over a finite alphabet that have a verifier that runs in polynomial time, where the notion of "verifier" is defined as follows.

Let L be a language over a finite alphabet, Σ.

$L \in$ NP if, and only if, there exists a binary relation and a positive integer k such that the following two conditions are satisfied:

1. For all , such that $(x, y) \in R$ and ; and
2. the language over is decidable by a deterministic Turing machine in polynomial time.

A Turing machine that decides L_R is called a *verifier* for L and a y such that $(x, y) \in R$ is called a *certificate of membership* of x in L.

In general, a verifier does not have to be polynomial-time. However, for L to be in NP, there must be a verifier that runs in polynomial time.

Example

Let

Clearly, the question of whether a given x is a composite is equivalent to the question of whether x is a member of COMPOSITE. It can be shown that COMPOSITE ∈ NP by verifying that it satisfies the above definition (if we identify natural numbers with their binary representations).

COMPOSITE also happens to be in P.[53][54]

NP-completeness
Main article: NP-completeness

There are many equivalent ways of describing NP-completeness.

Let L be a language over a finite alphabet Σ.

L is NP-complete if, and only if, the following two conditions are satisfied:

1. $L \in$ NP; and

2. any L' in NP is polynomial-time-reducible to L (written as), where if, and only if, the following two conditions are satisfied:

    1. There exists $f : \Sigma^* \rightarrow \Sigma^*$ such that for all w in Σ^* we have: ; and
    2. there exists a polynomial-time Turing machine that halts with $f(w)$ on its tape on any input w.

Alternatively, if $L \in$ NP, and there is another NP-complete problem that can be polynomial-time reduced to L, then L is NP-complete. This is a common way of proving some new problem is NP-complete.

In computational complexity theory, an NP-complete decision problem is one belonging to both the NP and the NP-hard complexity classes. In this context, NP stands for "nondeterministic polynomial time". The set of NP-complete problems is often denoted by NP-C or NPC.

Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place; the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly

as the size of the problem grows. As a consequence, determining whether it is possible to solve these problems quickly, called the P versus NP problem, is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. NP-complete problems are often addressed by using heuristic methods and approximation algorithms.

Overview

NP-complete problems are in NP, the set of all decision problems whose solutions can be verified in polynomial time; *NP* may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine. A problem *p* in NP is NP-complete if every other problem in NP can be transformed (or reduced) into *p* in polynomial time.

NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved—this is called the P versus NP problem. But if *any NP-complete problem* can be solved quickly, then *every problem in NP* can, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every NP-complete problem (that is, it can be reduced in polynomial time). Because of this, it is often said that NP-complete problems are *harder* or *more difficult* than NP problems in general.

Formal definition

See also: formal definition for NP-completeness (article *P = NP*)

A decision problem is NP-complete if:

1. is in NP, and
2. Every problem in NP is reducible to in polynomial time.[1]

can be shown to be in NP by demonstrating that a candidate solution to can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1.[2]

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or

any other Turing-equivalent abstract machine) for , we could solve all problems in NP in polynomial time.

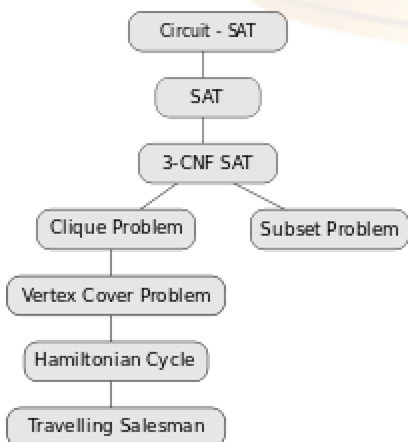
Background

The concept of NP-completeness was introduced in 1971 (see Cook–Levin theorem), though the term *NP-complete* was introduced later. At 1971 STOC conference, there was a fierce debate among the computer scientists about whether NP-complete problems could be solved in polynomial time on a deterministic Turing machine. John Hopcroft brought everyone at the conference to a consensus that the question of whether NP-complete problems are solvable in polynomial time should be put off to be solved at some later date, since nobody had any formal proofs for their claims one way or the other. This is known as the question of whether P=NP.

Nobody has yet been able to determine conclusively whether NP-complete problems are in fact solvable in polynomial time, making this one of the great unsolved problems of mathematics. The Clay Mathematics Institute is offering a US $1 million reward to anyone who has a formal proof that P=NP or that P≠NP.

The Cook–Levin theorem states that the Boolean satisfiability problem is NP-complete (a simpler, but still highly technical proof of this is available). In 1972, Richard Karp proved that several other problems were also NP-complete (see Karp's 21 NP-complete problems); thus there is a class of NP-complete problems (besides the Boolean satisfiability problem). Since the original results, thousands of other problems have been shown to be NP-complete by reductions from other problems previously shown to be NP-complete; many of these problems are collected in Garey and Johnson's 1979 book *Computers and Intractability: A Guide to the Theory of NP-Completeness*.[3] For more details refer to *Introduction to the Design and Analysis of Algorithms* by Anany Levitin.

NP-complete problems



Some NP-complete problems, indicating the reductions typically used to prove their NP-completeness

Main article: List of NP-complete problems

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices. Consider these two problems:

- Graph Isomorphism: Is graph G_1 isomorphic to graph G_2?
- Subgraph Isomorphism: Is graph G_1 isomorphic to a subgraph of graph G_2?

The Subgraph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be hard, but is not thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

To the right is a diagram of some of the problems and the reductions typically used to prove their NP-completeness. In this diagram, an arrow from one problem to another indicates the direction of the reduction. Note that this diagram is misleading as a description of the mathematical relationship between these problems, as there exists a polynomial-time reduction between any two NP-complete problems; but it indicates where demonstrating this polynomial-time reduction has been easiest.

There is often only a small difference between a problem in P and an NP-complete problem. For example, the 3-satisfiability problem, a restriction of the boolean satisfiability problem, remains NP-complete, whereas the slightly more restricted 2-satisfiability problem is in P (specifically, NL-complete), and the slightly more general max. 2-sat. problem is again NP-complete. Determining whether a graph can be colored with 2 colors is in P, but with 3 colors is NP-complete, even when restricted to planar graphs. Determining if a graph is a cycle or is bipartite is very easy (in L), but finding a maximum bipartite or a maximum cycle subgraph is NP-complete. A solution of the knapsack problem within any fixed percentage of the optimal solution can be computed in polynomial time, but finding the optimal solution is NP-complete.

Solving NP-complete problems

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, and it is unknown whether there are any faster algorithms.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- Approximation: Instead of searching for an optimal solution, search for a solution that is at most a factor from an optimal one.
- Randomization: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. Note: The Monte Carlo method is not an example of an efficient algorithm in this specific sense, although evolutionary approaches like Genetic algorithms may be.
- Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- Parameterization: Often there are fast algorithms if certain parameters of the input are fixed.
- Heuristic: An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

One example of a heuristic algorithm is a suboptimal greedy coloring algorithm used for graph coloring during the register allocation phase of some compilers, a technique called graph-coloring global register allocation. Each vertex is a variable, edges are drawn between variables which are being used at the same time, and colors indicate the register assigned to each variable. Because most RISC machines have a fairly large number of general-purpose registers, even a heuristic approach is effective for this application.

Completeness under different types of reduction

In the definition of NP-complete given above, the term *reduction* was used in the technical meaning of a polynomial-time many-one reduction.

Another type of reduction is polynomial-time Turing reduction. A problem is polynomial-time Turing-reducible to a problem if, given a subroutine that solves in polynomial time, one could write a program that calls this subroutine and solves in polynomial time. This contrasts with many-one reducibility, which has the restriction that the program can only call the subroutine once, and the return value of the subroutine must be the return value of the program.

If one defines the analogue to NP-complete with Turing reductions instead of many-one reductions, the resulting set of problems won't be smaller than NP-complete; it is an open question whether it will be any larger.

Another type of reduction that is also often used to define NP-completeness is the logarithmic-space many-one reduction which is a many-one reduction that can be computed with only a logarithmic amount of space. Since every computation that can be done in logarithmic space can also be done in polynomial time it follows that if there is a logarithmic-space many-one reduction then there is also a polynomial-time many-one reduction. This type of reduction is more refined than the more usual polynomial-time many-one reductions and it allows us to distinguish more classes such as P-complete. Whether under these types of reductions the definition of NP-complete changes is still an open problem. All currently known NP-complete problems are NP-complete under log space reductions. All currently known NP-complete problems remain NP-complete even under much weaker reductions.[4] It is known, however, that AC^0 reductions define a strictly smaller class than polynomial-time reductions.[5]

Naming

According to Donald Knuth, the name "NP-complete" was popularized by Alfred Aho, John Hopcroft and Jeffrey Ullman in their celebrated textbook "The Design and Analysis of Computer Algorithms". He reports that they introduced the change in the galley proofs for the book (from "polynomially-complete"), in accordance with the results of a poll he had conducted of the theoretical computer science community.[6] Other suggestions made in the poll[7] included "Herculean", "formidable", Steiglitz's "hard-boiled" in honor of Cook, and Shen Lin's acronym "PET", which stood for "probably exponential time", but depending on which way the P versus NP problem went, could stand for "provably exponential time" or "previously exponential time".[8]

Common misconceptions

The following misconceptions are frequent.

"NP-complete problems are the most difficult known problems." Since NP-complete problems are in NP, their running time is at most exponential. However, some problems provably require more time, for example Presburger arithmetic.

- *"NP-complete problems are difficult because there are so many different solutions."* On the one hand, there are many problems that have a solution space just as large, but can be solved in polynomial time (for example minimum spanning tree). On the other hand, there are NP-problems with at most one solution that are NP-hard under randomized polynomial-time reduction (see Valiant–Vazirani theorem).
- *"Solving NP-complete problems requires exponential time."* First, this would imply $P \neq NP$, which is still an unsolved question. Further, some NP-complete problems actually have algorithms running in super polynomial, but sub exponential time such as $O(2^{\sqrt{n}}n)$. For example, the independent set and dominating set problems are NP-complete when restricted to planar graphs, but can be solved in sub exponential time on planar graphs using the planar separator theorem.[10]
- *"All instances of an NP-complete problem are difficult."* Often some instances, or even most instances, may be easy to solve within polynomial time. However, unless P=NP, any polynomial-time algorithm must asymptotically be wrong on more than polynomial many of the exponentially many inputs of a certain size.[11]
- *"If P=NP, all cryptographic ciphers can be broken."* A polynomial-time problem can be very difficult to solve in practice if the polynomial's degree or constants are large enough. For example, ciphers with a fixed key length, such as Advanced Encryption Standard, can all be broken in constant time (and are thus already in P), though with current technology that constant may exceed the age of the universe.