# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

L-1

## LECTURE HANDOUTS

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** Introduction- Basic Mathematical Notation and techniques

**Introduction : ( Maximum 5 sentences)** :

- In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines.
- These abstract machines are called automata.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- automata theory
- computability theory
- mathematical abstraction

**Detailed content of the Lecture:**

**What is TOC**?

In theoretical computer science, the **theory of computation** is the branch that deals with whether and how efficiently problems can be solved on a model of computation, using an algorithm. The field is divided into three major branches: automata theory, computability theory and computational complexity theory.

In order to perform a rigorous study of computation, computer scientists work with a mathematical abstraction of computers called a model of computation. There are several models in use, but the most commonly examined is the Turing machine.

**Automata theory**

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

This automaton consists of

• **states** (represented in the figure by circles),
• and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.

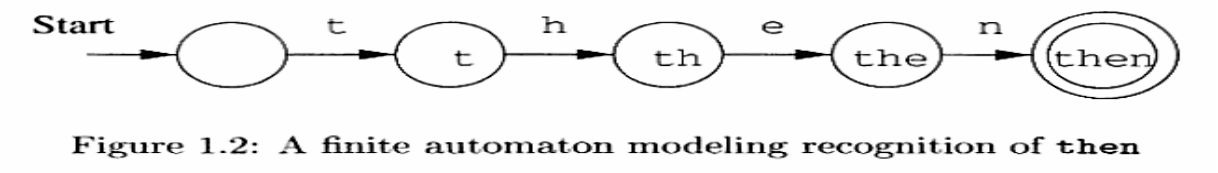



Figure 1.2: A finite automaton modeling recognition of then

**Introduction to formal proof:**

**Basic Symbols used :**

U – Union

∩- Conjunction

ϵ - Empty String

ϕ– NULL set

>- negation

‘– compliment

= > implies

**Additive inverse:** a+(-a)=0

**Multiplicative inverse:** a*1/a=1

Universal set U={1,2,3,4,5}

Subset A={1,3}

A’ ={2,4,5}

**Absorption law:** AU(A∩B) = A, A∩(AUB) = A

**De Morgan’s Law:**

(AUB)’ =A’∩B’

(A_B)’ = A’ U B’

Double compliment

(A’)’ =A

A ∩A’ = ϕ

**Logic relations:**

a ->b = > a U b

>(a_b)=>a U >b

**Relations:**

Let a and b be two sets a relation R contains aXb.

Relations used in TOC:

**Reflexive:** a = a

**Symmetric:** aRb = > bRa

**Transition:** aRb, bRc = > aRc

If a given relation is reflexive, symmentric and transitive then the relation is called equivalence relation.

**Deductive proof:** Consists of sequence of statements whose truth lead us from some initial statement called the hypothesis or the give statement to a conclusion statement.

The theorem that is proved when we go from a hypothesis $H$ to a conclusion $C$ is the statement “if $H$ then $C$.” We say that $C$ is *deduced* from $H$. An example

**Additional forms of proof:**

Proof of sets

Proof by contradiction

Proof by counter example

**Direct proof (AKA) Constructive proof:**

If *p* is true then *q* is true

Eg: if a and b are odd numbers then product is also an odd number.

Odd number can be represented as 2n+1

a=2x+1, b=2y+1

product of a X b = (2x+1) X (2y+1)

= 2(2xy+x+y)+1 = 2z+1 (odd number)

**Proof by contrapositive:**

**Theorem 1.10:** $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

| | Statement | Justification |
|---|---|---|
| 1. | $x$ is in $R \cup (S \cap T)$ | Given |
| 2. | $x$ is in $R$ or $x$ is in $S \cap T$ | (1) and definition of union |
| 3. | $x$ is in $R$ or $x$ is in both $S$ and $T$ | (2) and definition of intersection |
| 4. | $x$ is in $R \cup S$ | (3) and definition of union |
| 5. | $x$ is in $R \cup T$ | (3) and definition of union |
| 6. | $x$ is in $(R \cup S) \cap (R \cup T)$ | (4), (5), and definition of intersection |

Figure 1.5: Steps in the "if" part of Theorem 1.10

| | Statement | Justification |
|---|---|---|
| 1. | $x$ is in $(R \cup S) \cap (R \cup T)$ | Given |
| 2. | $x$ is in $R \cup S$ | (1) and definition of intersection |
| 3. | $x$ is in $R \cup T$ | (1) and definition of intersection |
| 4. | $x$ is in $R$ or $x$ is in both $S$ and $T$ | (2), (3), and reasoning about unions |
| 5. | $x$ is in $R$ or $x$ is in $S \cap T$ | (4) and definition of intersection |
| 6. | $x$ is in $R \cup (S \cap T)$ | (5) and definition of union |

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

To see why "if $H$ then $C$" and "if not $C$ then not $H$" are logically equivalent, first observe that there are four cases to consider:

1. $H$ and $C$ both true.
2. $H$ true and $C$ false.
3. $C$ true and $H$ false.
4. $H$ and $C$ both false.

**Proof by Contradiction:**

H and not C implies falsehood.

That is, start by assuming both the hypothesis $H$ and the negation of the conclusion $C$. Complete the proof by showing that something known to be false follows logically from $H$ and not $C$. This form of proof is called *proof by contradiction.*

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if $S$ is any statement, then the statement "$S$ is not a theorem" is itself a statement without parameters, and thus can

**Example :** | 011 | = 4, |11| = 2, | b | = 1
**Convention :** We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end denote strings over an alphabet.

**Example :** Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.
Prefixes: ε, 0, 01, 011.
Suffixes: ε, 1, 11, 011.
Substrings: ε, 0, 1, 01, 11, 011.
Note that x is a prefix (suffix or substring) to x, for any string x and ε is a prefix (suffix or substring) to any string.
A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and $x \neq y$.
In the above example, all prefixes except 011 are proper prefixes.

**Video Content / Details of website for further learning (if any):**
https://www.youtube.com/watch?v=1aEinrlyp8w
https://www.youtube.com/watch?v=00cXiux2Kjk

**Important Books/Journals for further learning including the page nos.:**
Hopcroft J.E., Motwani R. and Ullman J.D, Introduction to Automata Theory, Languages and Computations, Pearson Education Second Edition T1(46-49

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

L-2

## LECTURE HANDOUTS

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** Finite State System

**Introduction : ( Maximum 5 sentences)** :

Automata (singular : automation) are a particularly simple, but useful, model of computation.They were initially proposed as a simple model for the behavior of neurons.

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

**State Transitions**, **Languages, Symbols ,Alphabets**

**Detailed content of the Lecture:**

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

**States, Transitions and Finite-State Transition System :**

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

- Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.
- *Transitions* are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction).
- Some examples of state transition systems are: digital systems, vending machines, etc. A system containing only a finite number of states and transitions among them is called a *finite-state transition system*.
- Finite-state transition systems can be modeled abstractly by a mathematical model called *finite automation*

**Languages :**

The languages we consider for our discussion is an abstraction of natural languages. That is, our focus here is on formal languages that need precise and formal definitions. Programming languages belong to this category.

**Symbols :**

Symbols are indivisible objects or entity that cannot be defined. That is, symbols are the atoms of the world of languages. A symbol is any single object such as ♠, *a*, 0, 1, #, begin, or do.

**Alphabets :**

An alphabet is a finite, nonempty set of symbols. The alphabet of a language is normally denoted By Σ. When more than one alphabets are considered for discussion, then subscripts may be used (e.g. $\Sigma 1, \Sigma 2$ etc) or sometimes other symbol like G may also be

Σ={a,b,c}
Σ= {a,b,c,&,z}
Σ= {#,β,♠,∇}

**Example** :

**Strings or Words over Alphabet :**

A string or word over an alphabet is a finite sequence of concatenated symbols of .

**Example** : 0110, 11, 001 are three strings over the binary alphabet { 0, 1 } .

aab, abcb, b, cc are four strings over the alphabet { a, b, c }.

It is not the case that a string over some alphabet should contain all the symbols from the alphabet. For example, the string cc over the alphabet { a, b, c } does not contain the symbols a and b. Hence, it is true that a string over an alphabet is also a string over any superset of that alphabet.

**Length of a string :**

The number of symbols in a string w is called its length, denoted by |w|.

**Example :** | 011 | = 4, |11| = 2, | b | = 1

**Convention :** We will use small case letters towards the beginning of the English alphabet to denote symbols of an alphabet and small case letters towards the end to denote strings over an alphabet. That is a,b,c,ϵ,Σ (symbols) $u,v,w,x,y,z$ and are strings.

**Example :** Consider the string 011 over the binary alphabet. All the prefixes, suffixes and substrings of this string are listed below.

Prefixes: ε, 0, 01, 011.

Suffixes: ε, 1, 11, 011.

Substrings: ε, 0, 1, 01, 11, 011.

Note that x is a prefix (suffix or substring) to x, for any string x and ε is a prefix (suffix or substring) to any string.

A string x is a proper prefix (suffix) of string y if x is a prefix (suffix) of y and $x \neq y$.

In the above example, all prefixes except 011 are proper prefixes.

**Video Content / Details of website for further learning (if any):**

https://www.youtube.com/watch?v=nqvDdT4h4iQ

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D, Introduction to Automata Theory, Languages and Computations, Pearson Education Second Edition .T1(63-70)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)
Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

L-3

**LECTURE HANDOUTS**

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** Basic Definition of Finite Automaton

**Introduction : ( Maximum 5 sentences)** :

- Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.
- A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.
- An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- **Finite-State System**

**Detailed content of the Lecture:**

**Finite Automata**

Automata (singular : automation) are a particularly simple, but useful, model of computation. They were initially proposed as a simple model for the behavior of neurons.

**States, Transitions and Finite-State Transition System :**

Let us first give some intuitive idea about *a state of a system* and *state transitions* before describing finite automata.

- Informally, *a state of a system* is an instantaneous description of that system which gives all relevant information necessary to determine how the system can evolve from that point on.
- *Transitions* are changes of states that can occur spontaneously or in response to inputs to the states. Though transitions usually take time, we assume that state transitions are instantaneous (which is an abstraction). Some examples of state transition systems are: digital systems, vending machines, etc.
- A system containing only a finite number of states and transitions among them is called a *finite-state transition system*.

**Finite-state transition** systems can be modeled abstractly by a mathematical model called

## FINITE AUTOMATION

Automata: A algorithm or program that automatically recognizes if a particular string belongs to the language or not, by checking the grammar of the string.

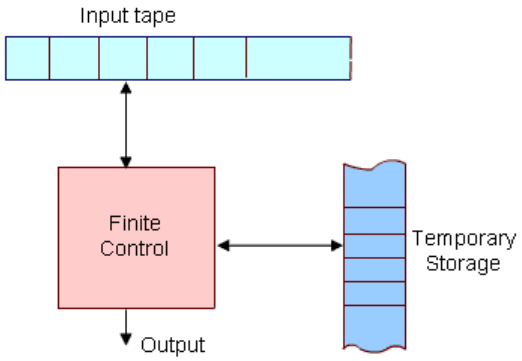
- An automata is an abstract computing device (or machine). There are different varities of such abstract machines (also called models of computation) which can be defined mathematically.

Every Automaton fulfills the three basic requirements.

- Every automaton consists of some essential features as in real computers. It has a mechanism for reading input. The input is assumed to be a sequence of symbols over a given

alphabet and is placed on an input tape(or written on an input file). The simpler automata can only read the input one symbol at a time from left to right but not change. Powerful versions can both read (from left to right or right to left) and change the input.The automaton can produce output of some form. If the output in response to an input string is binary (say, accept or reject), then it is called an accepter. If it produces an output sequence in response to an input sequence, then it is called a transducer(or automaton with output).

- The automaton may have a temporary storage, consisting of an unlimited number of cells, each capable of holding a symbol from an alphabet ( whcih may be different from the input alphabet). The automaton can both read and change the contents of the storage cells in the temporary storage. The accusing capability of this storage varies depending on the type of the storage.
- The most important feature of the automaton is its control unit, which can be in any one of a finite number of interval states at any point. It can change state in some defined manner determined by a transition function.




**Operation of the automation is defined as follows.**

- At any point of time the automaton is in some integral state and is reading a particular symbol from the input tape by using the mechanism for reading input. In the next time step the automaton then moves to some other integral (or remain in the same state) as defined by the transition function.
- The transition function is based on the current state, input symbol read, and the content of the temporary storage. At the same time the content of the storage may be changed and the input read may be modifed.
- The automation may also produce some output during this transition.The internal state, input and the content of storage at any point defines the configuration of the automaton at that point. The transition from one configuration to the next ( as defined by the transition function) is called a *move.*
- Finite state machine or *Finite Automation* is the simplest type of abstract machine we consider. Any system that is at any point of time in one of a finite number of interval state and moves among these states in a defined manner in response to some input, can be modeled by a finite automaton. It doesnot have any temporary storage and hence a restricted model of computation.

**Union :** A string $x \in L_1 \cup L_1$ iff $x \in L_1$ or $x \in L_2$
**Example :** { 0, 11, 01, 011 } U { 1, 01, 110 } = { 0, 11, 01, 011, 111 }
**Intersection :** A string $x \in L_1 \cap L_2$ iff $x \in L_1$ or $x \in L_2$
**Example :** { 0, 11, 01, 011 }∩ { 1, 01, 110 } = { 01 }
**Complement :** Usually, is the universe that a complement is taken with respect to.
Thus for a language L, the complement is L(bar) = $\{ x \in \Sigma^* \mid x \notin L \}$.
**Example :** Let L = { x | |x| is even }. Then its complement is the language $\{ x \in \Sigma^* \mid |x| \text{ is odd} \}$.Similarly we can define other usual set operations on languages like relative complement, symmetric difference, etc.
**Reversal of a language :**
The reversal of a language *L*, denoted as $L^R$ , is defined as: $L^R – (w^R \mid w \epsilon L)$

1. Let L = { 0, 11, 01, 011 }. Then $L^R$ = { 0, 11, 10, 110 }.

**Language concatenation :** The concatenation of languages $L_1$ and $L_2$ is defined as
**Example :** { *a*, *ab* }{ *b*, *ba* } = { *ab*, *aba*, *abb*, *abba* }.

**Iterated concatenation of languages :** Since we can concatenate two languages, we also repeat

this to concatenate any number of languages. Or we can concatenate a language with itself any number of times. The operation denotes the concatenation of L* with itself n times. This is defined formally as follows:

$L_0 = (e)$

$L^* = L\ L^{n-1}$

**Example :** Let L = { a, ab }. Then according to the definition, we have

$L_0 = (e)$

$L_1 = L(e) = L = (a,ab)$

$L_2 = L\ L_1 = (a,ab)\ (a,ab) = (aa,aab,aba,abab)$

$L_3 = L\ L_2 = (a,ab)\ (aa,aab,aba,abab) – (aaa,aab.aaba,aabab,abaa,abaab,ababa,ababab)$

And so on.

**Video Content / Details of website for further learning (if any):**

https://www.youtube.com/watch?v=nqvDdT4h4iQ

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D, Introduction to Automata Theory, Languages and Computations, Pearson Education Second Edition T1(63-70)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

**L-4**

## LECTURE HANDOUTS

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** Deterministic Finite State Automaton(DFA)

**Introduction : ( Maximum 5 sentences)** :

- Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string one symbol at a time and then, after the input has been completely read, decides whether to accept or reject the input.
- As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far.
- A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- **Finite State Automaton**
- **Regular Expression and Regular Language**

**Detailed content of the Lecture:**

**Deterministic Finite (-state) Automata**

Informally, a DFA (Deterministic Finite State Automaton) is a simple machine that reads an input string -- one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can change its state, to reflect how it reacts to what it has seen so far. A machine for which a deterministic code can be formulated, and if there is only one unique way to formulate the code, then the machine is called deterministic finite automata.

Thus, a DFA conceptually consists of 3 parts:

1. A *tape* to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from .
2. A *tape head* for reading symbols from the tape
3. A *control* , which itself consists of 3 things:
   - finite number of states that the machine is allowed to be in (zero or more states are designated as *accept* or *final* states),
   - a current state, initially set to a start state,
   - a state transition function for changing the current state.

An automaton processes a string on the tape by repeating the following actions until the tape head has traversed the entire string:

1. The tape head reads the current tape cell and sends the symbol s found there to the control. Then the tape head moves to the next cell.
2. he control takes s and the current state and consults the state transition function to get the next state, which becomes the new current state.

Once the entire string has been processed, the state in which the automation enters is examined.

If it is an accept state , the input string is accepted ; otherwise, the string is rejected . Summarizing all the above we can formulate the following formal definition: **Automata theory**

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

This automaton consists of

• **states** (represented in the figure by circles),

• and **transitions** (represented by arrows).

As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its ***transition function*** (which takes the current state and the recent symbol as its inputs).

Uses of Automata: compiler design and parsing.

**Deterministic Finite State Automaton :** A Deterministic Finite State Automaton (DFA) is a 5-tuple : A = (Q, Σ, δ, q0, F) where

• Q is a finite set of states

• Σ is the finite set of input symbols

• δ is a transition function (Q X Σ->Q )

• q0 is the start state or initial state

• F is a set of final or accepting states

**Language Accepted or Recognized by a DFA :**

The language accepted or recognized by a DFA M is the set of all strings accepted by M , and is denoted by

$$L(M)_{i.e}\ L(M) = \{w \in \Sigma^* / M \text{ accepts } w\}$$

The notion of acceptance can also be made more precise by extending the transition function δ.

**Extended transition function :**

Extend $\delta : Q \times \Sigma \rightarrow Q$ (which is function on symbols) to a function on strings, i.e. .

$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

That is $\hat{\delta}(q,w)$ , is the state the automation reaches when it starts from the state q and finish processing the string w. Formally, we can give an inductive definition as follows:

The language of the DFA M is the set of strings that can take the start state to one of the accepting states i.e.

$L(M) = \{w \in \Sigma^* / M \text{ accepts } w\}$

$= \{ w \in \Sigma^* / \hat{\delta}(q_0,w) \in F\}$

**Example 1 :**

$M= (Q, \Sigma, \delta, q_0, F)$

$Q=\{q_0,q_1\}$

$q_0$ ,is the start state

**Transition table :**

It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the “next state”).

• Rows correspond to states,

• Columns correspond to input symbols,

• Entries correspond to next states

• The start state is marked with an arrow

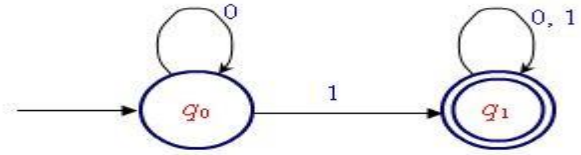
• The accept states are marked with a star (*).

| | 0 | 1 |
|---|---|---|
| $\rightarrow q_0$ | $q_0$ | $q_1$ |
| $*q_1$ | $q_1$ | $q_1$ |

**(State) Transition diagram :**

A state transition diagram or simply a transition diagram is a directed graph which can be constructed as follows:

1. For each state in Q there is a node.
2. There is a directed edge from node q to node p labeled a iff $\delta(q,a)=p$ . (If there are several input symbols that cause a transition, the edge is labeled by the list of these symbols.)
3. There is an arrow with no source into the start state.
4. Accepting states are indicated by double circle.




5. Here is an informal description how a DFA operates. An input to a DFA can be any string $w \in \Sigma^*$ . Put a pointer to the start state q. Read the input string w from left to right, one symbol at a time, moving the pointer according to the transition function, $\delta$. If the next symbol of w is a and the pointer is on state p, move the pointer to $\delta(p,a)$. When the end of the input string w is encountered, the pointer is on some state, r. The string is said to be accepted by the DFA if $r \in F$ and rejected if $r \in F$ . Note that there is no formal mechanism for moving the pointer.
6. A language is said to be regular if L = L(M) for some DFA M.

**Regular Expressions: Formal Definition**

We construct REs from primitive constituents (basic elements) by repeatedly applying certain recursive rules as given below. (In the definition)

**Definition** : Let S be an alphabet. The regular expressions are defined recursively as follows.

**Basis** :

i) $\phi$ is a RE
ii) $\epsilon$ is a RE
iii) $\forall a \in S$ is RE.

These are called primitive regular expression i.e. Primitive Constituents

**Recursive Step :**

If $r1$ and $r2$ are REs over, then so are
i) $r1+r2$
ii) $r1\ r2$
iii) $r1^*$
iv) $(r1)$

**Closure :** r is RE over only if it can be obtained from the basis elements (Primitive REs) by a finite no of applications of the recursive step (given in 2).

**Example :** Let = { 0,1,2 }. Then (0+21)*(1+ F ) is a RE, because we can construct this expression by applying the above rules as given in the following step.

| **Steps** | **RE Constructed** | **Rule Used** |
|---|---|---|
| 1 | 1 | Rule 1(iii) |
| 2 | $\phi$ | Rule 1(i) |
| 3 | 1+ $\phi$ | Rule 2(i) & Results of Step 1, 2 |
| 4 | (1+ $\phi$ ) | Rule 2(iv) & Step 3 |
| 5 | 2 | 1(iii) |
| 6 | 1 | 1(iii) |
| 7 | 21 | 2(ii), 5, 6 |
| 8 | 0 | 1(iii) |
| 9 | 0+21 | 2(i), 7, 8 |
| 10 | (0+21) | 2(iv), 9 |
| 11 | (0+21)* | 2(iii), 10 |
| 12 | (0+21)* | 2(ii), 4, 11 |

**Language described by REs :** Each describes a language (or a language is associated with every RE). We will see later that REs are used to attribute regular languages.

**Notation :** If r is a RE over some alphabet then L(r) is the language associate with r . We can define the language L(r) associated with (or described by) a REs as follows.

1. $\phi$ is the RE describing the empty language i.e. $L(\phi) = \phi$.
2. $\epsilon$ is a RE describing the language $\{ \epsilon \}$ i.e. $L(\epsilon) = \{ \epsilon \}$ .
3. $\forall\ a \in S$ is a RE denoting the language $\{a\}$ i.e . $L(a) = \{a\}$ .

4. If $^{r1}$and $^{r2}$ are REs denoting language $L(r_1)$ and $L(r_2)$ respectively, then
5. $r_1 + r_2$ is a regular expression denoting the language $L(r_1 + r_2) = L(r_1) È L(r_2)$.
**Example :** The RE $ab^*+b$ is grouped as $((a(b^*))+b)$ which describes the language $L(a)(L(b))^* L(b)$
**Example :** The RE $(ab)^*+b$ represents the language $(L(a)L(b))^* È L(b)$.
**Example :** Consider the language of strings over {0,1} containing two or more 1's.
**Solution :** There must be at least two 1's in the RE somewhere and what comes before, between, and after is completely arbitrary. Hence we can write the RE as (0+1)*1(0+1)*1(0+1)*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string
i) 0*10*1(0+1)*
ii) (0+1)*10*10*
**Regular Expression and Regular Language :**
**Equivalence(of REs) with FA :**
Recall that, language that is accepted by some FAs are known as Regular language. The two concepts : REs and Regular language are essentially same i.e. (for) every regular language can be developed by (there is) a RE, and for every RE there is a Regular Langauge. This fact is rather suprising, because RE approach to describing language is fundamentally differnet from the FA approach. But REs and FA are equivalent in their descriptive power. We can put this fact in the focus of the following Theorem.
**Theorem :** A language is regular iff some RE describes it.
This Theorem has two directions, and are stated & proved below as a separate lemma
**RE to FA :**
**REs denote regular languages :**
**Lemma :** If $L(r)$ is a language described by the RE $r$, then it is regular i.e. there is a FA such that $L(M) = L(r)$.
**Proof :** To prove the lemma, we apply structured index on the expression $r$. First, we show how to construct FA for the basis elements: , and for any . Then we show how to combine these Finite Automata into Complex Automata that accept the Union, Concatenation, Kleen Closure of the languages accepted by the original smaller automata.
Use of NFAs is helpful in the case i.e. we construct NFAs for every REs which are represented by transition diagram only.

**Video Content / Details of website for further learning (if any):**
https://www.youtube.com/watch?v=1aEinrlyp8w
https://www.youtube.com/watch?v=00cXiux2Kjk

**Important Books/Journals for further learning including the page nos.:**
Hopcroft J.E., Motwani R. and Ullman J.D, Introduction to Automata Theory, Languages and Computations, Pearson Education Second Edition .T1(70-76)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

**L-5**

## LECTURE HANDOUTS

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** NDFA

**Introduction : ( Maximum 5 sentences)** :

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a give
- The finite automata are called NFA when there exist many paths for specific input from the current st
- Every NFA is not DFA, but each NFA can be translated into DFA.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- **NFA than DFA**

**Detailed content of the Lecture:**

- NFA stands for non-deterministic finite automata. It is easy to construct an C for a given regular language
- The finite automata are called NFA when there exist many paths for specific input from the current state
- Every NFA is not DFA, but each NFA can be translated into DFA.
- NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next

In the following image, we can see that from state q0 for input a, there are two next states q1 and q2, similarly, fr
Thus it is not fixed or determined that with a particular input where to go next. Hence this FA is called non-deter

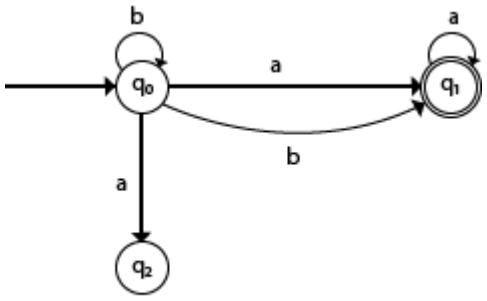



Fig:- NDFA

*Formal definition of NFA:*

NFA also has five states same as DFA, but with different transition function, as shown follows:

$\delta: Q \times \sum \rightarrow 2^Q$

1. Q: finite set of states
2. Σ: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

## *Graphical Representation of an NFA*

An NFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by the double circle.

### Example 1:

1. Q = {q0, q1, q2}
2. Σ = {0, 1}
3. q0 = {q0}
4. F = {q2}

**Solution:**

Transition diagram:

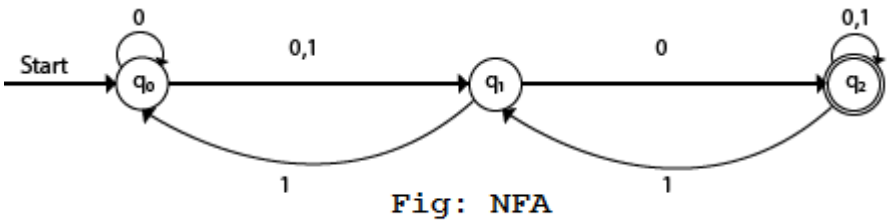



Fig: NFA

| Present State | Next state for Input 0 | Next State of Input 1 |
|---|---|---|
| →q0 | q0, q1 | q1 |
| q1 | q2 | q0 |
| *q2 | q2 | q1, q2 |

In the above diagram, we can see that when the current state is q0, on input 0, the next state will be q0 or q1, and on state is q1, on input 0 the next state will be q2 and on 1 input, the next state will be q0. When the current state is q2, next state will be q1 or q2.

**Video Content / Details of website for further learning (if any):**

https://www.youtube.com/watch?v=nqvDdT4h4iQ

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-6**

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D, Introduction to Automata Theory, Languages and Computations,

**Course Teacher**

**Verified by HOD**

**Course Name with Code** : Theory of Computation -16CSD11

**LECTURE HANDOUTS**

**Course Teacher** :K.PA[illegible]

**U** **CSE** : I Finite Automata **Date of Lectu** **IV/II/A**

**Topic of Lecture:** Finite Automaton with €- moves

**Introduction : ( Maximum 5 sentences)** :

- A **finite automaton** (FA) is a simple idealized machine used to recognize patterns within input taken from some character set (or alphabet) C.
- The job of an FA is to accept or reject an input depending on whether the pattern defined by the

FA occurs in the input.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- **finite automaton**
- **Regular Expression**

**Detailed content of the Lecture:**

There are two methods to convert FA to regular expression –

**1. State Elimination Method –**

**Step 1 –**

If the start state is an accepting state or has transitions in, add a new non-accepting start state and add an €-transition between the new start state and the former start state.

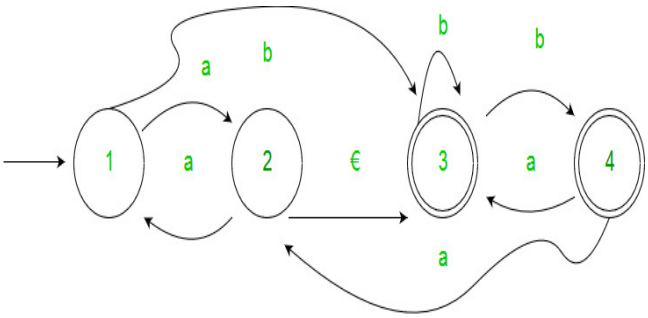
**Step 2 –**

If there is more than one accepting state or if the single accepting state has transitions out, add a new accepting state, make all other states non-accepting, and add an €-transition from each former accepting state to the new accepting state.

**Step 3 –**

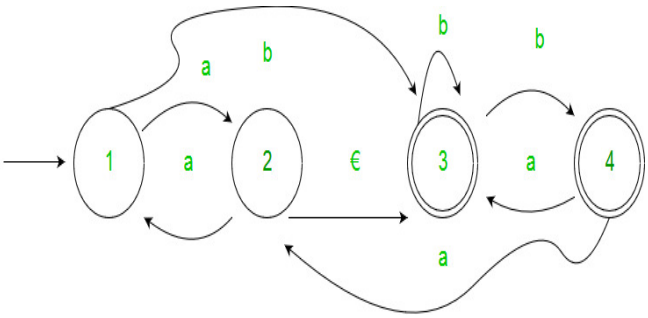
For each non-start non-accepting state in turn, eliminate the state and update transitions accordingly.

**Example** :-

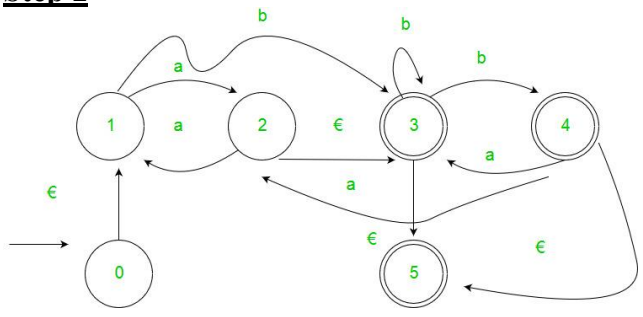



**Solution :-**

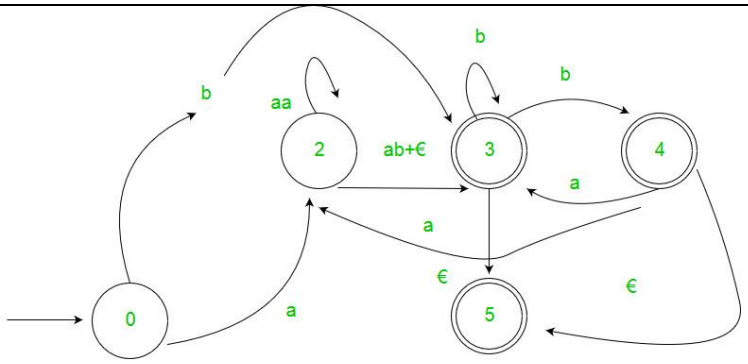
**Step 1**

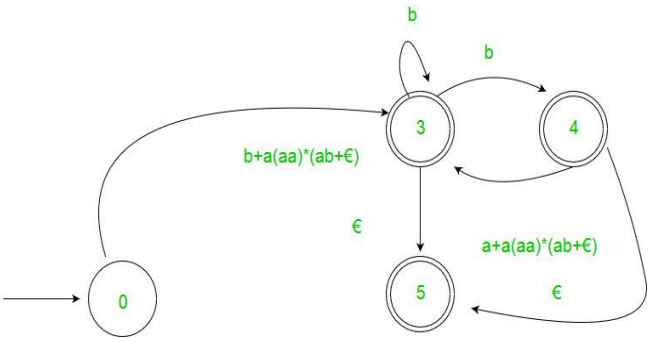


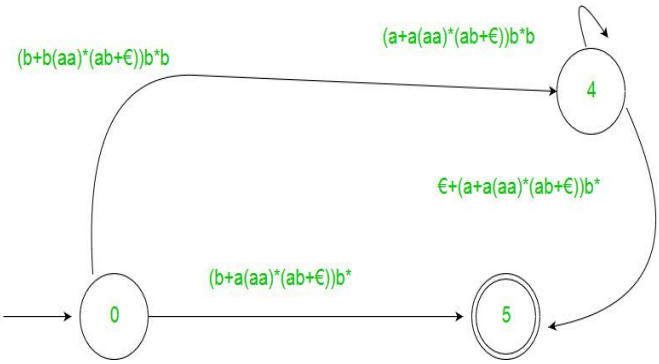

**Step 2**




**Step 3**




(b+a((aa)*(ab+€))b*+((b+a(aa)*(ab+€))b*b((a+a(aa)*(ab+€))b*b)*(€+(a+a(aa)*(ab+€))b*)


**2. Arden's Theorem –** Let P and Q be 2 regular expressions. If P does not contain null string, then following equation in R, viz R = Q + RP, Has a unique solution by R = QP*

**Assumptions –**

- The transition diagram should not have €-moves.
- It must have only one initial state.

**Using Arden's Theorem to find Regular Expression of Deterministic Finite automata –**

1. For getting the regular expression for the automata we first create equations of the given form for all the states

   **$q_1 = q_1w_{11} + q_2w_{21} + \ldots + q_nw_{n1} + €$ ($q_1$ is the initial state)**

   **$q_2 = q_1w_{12} + q_2w_{22} + \ldots + q_nw_{n2}$**

   .

   .

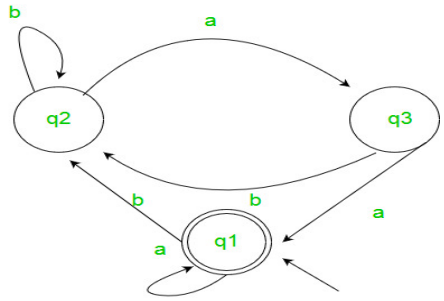
   .

   **$q_n = q_1w_{1n} + q_2w_{2n} + \ldots + q_nw_{nn}$**

   $w_{ij}$ is the regular expression representing the set of labels of edges from $q_i$ to $q_j$

**Note –** For parallel edges there will be that many expressions for that state in the expression.

2. Then we solve these equations to get the equation for $q_i$ in terms of $w_{ij}$ and that expression is the required solution, where $q_i$ is a final state.




**Solution** :-

Here the initial state is $q_2$ and the final state is $q_1$.

The equations for the three states $q_1$, $q_2$, and $q_3$ are as follows ?

$q_1 = q_1a + q_3a + €$ ( € move is because $q_1$ is the initial state)

$q_2 = q_1b + q_2b + q_3b$

$q_3 = q_2a$

Now, we will solve these three equations ?

$q_2 = q_1b + q_2b + q_3b$

$= q_1b + q_2b + (q_2a)b$ (Substituting value of $q_3$)

$= q_1b + q_2(b + ab)$

$= q_1b\ (b + ab)^*$ (Applying Arden's Theorem)

$q_1 = q_1a + q_3a + €$

$= q_1a + q_2aa + €$ (Substituting value of $q_3$)

$= q_1a + q_1b(b + ab^*)aa + €$ (Substituting value of $q_2$)

$= q_1(a + b(b + ab)^*aa) + €$

$= € (a+ b(b + ab)^*aa)^*$

$= (a + b(b + ab)^*aa)^*$

Hence, the regular expression is $(a + b(b + ab)^*aa)^*$.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/generating-regular-expression-from-finite-automata/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second. T1(90-97)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-7**

**LECTURE HANDOUTS**

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** Equivalence of NFA and DFA

**Introduction : ( Maximum 5 sentences)** :

- Finite state automata (FSA), also known as finite state machines (FSM), are usually classified as being deterministic (DFA) or non-deterministic (NFA). A deterministic finite state automaton has exactly one transition from every state for each possible input.
- For a given string, the path through a DFA is deterministic since there is no place along the way where the machine would have to choose between more than one transition. Given this definition it isn't too hard to figure out what an NFA is.
- Unlike in DFA, it is possible for states in an NFA to have more than one transition per input symbol. Additionally, states in an NFA may have states that don't require an input symbol at

all, transitioning on the empty string ε.

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- **NFA, DFA, FSA**

**Detailed content of the Lecture:**

Before continuing, let's formally state the theorem we are proving:

## Theorem

Let language $L \subseteq \Sigma^*$, and suppose L is accepted by NFA $N = (\Sigma, Q, q_0, F, \delta)$. There exists a DFA $D= (\Sigma, Q', q'_0, F', \delta')$ that also accepts L. (L(N) = L(D)).

By allowing each state in the DFA D to represent a set of states in the NFA N, we are able to prove through induction that D is equivalent to N. Before we begin the proof, let's define the parameters of D:

- Q' is equal to the powerset of Q, $Q' = 2^Q$
- $q'_0 = \{q_0\}$
- F' is the set of states in Q' that contain any element of F, $F' = \{q \in Q' | q \cap F \neq \emptyset\}$
- δ' is the transition function for D. $\delta'(q,a) = \bigcup_{p \in q} \delta(p,a)$ for $q \in Q'$ and $a \in \Sigma$.

I think it's worth further explaining δ'. Remember that each state in the set of states Q' in D is a set of states itself from Q in N. For each state p in state q in Q' of D (p is a single state from Q), determine the transition δ(p,a). δ(q,a) is the union of all δ(p,a).

Now we will prove that $\delta(q'_0, x) = \delta(q_0, x)$ for every *x*. ie, L(D) = L(N)

**Theorem:** For any string w, $\delta_1^*(q_{1,0}, w) = \delta_2^*(q_{2,0}, w)$.

**Proof:** This is going to be proven by induction on w.

**Basis Step:** For $w = \Lambda$,

$\delta_2^*(q_{2,0}, \Lambda) = q_{2,0}$ by the definition of $\delta_2^*$.
$= \{q_{1,0}\}$ by the construction of DFA $M_2$.
$= \delta_1^*(q_{1,0}, \Lambda)$ by the definition of $\delta_1^*$.

**Inductive Step:** Assume that $\delta_1^*(q_{1,0}, w) = \delta_2^*(q_{2,0}, w)$ for an arbitrary string w. --- Induction Hypothesis

For the string w and an arbitrry symbol a in $\Sigma$,

$\delta_1^*(q_{1,0}, wa) = \bigcup_{p \in \delta_1^*(q_{1,0}, w)} \delta_1(p, a)$
$= \delta_2(\delta_1^*(q_{1,0}, w), a)$
$= \delta_2(\delta_2^*(q_{2,0}, w), a)$
$= \delta_2^*(q_{2,0}, wa)$

Thus for any string w $\delta_1^*(q_{1,0}, w) = \delta_2^*(q_{2,0}, w)$ holds.

## *NFA to DFA Conversion Example*

From the proof, we can tease out an algorithm that will allow us to convert any non-deterministic finite state automaton (NFA) to an equivalent deterministic finite state automaton (DFA). That is, the language accepted by the DFA is identical that accepted by the NFA.

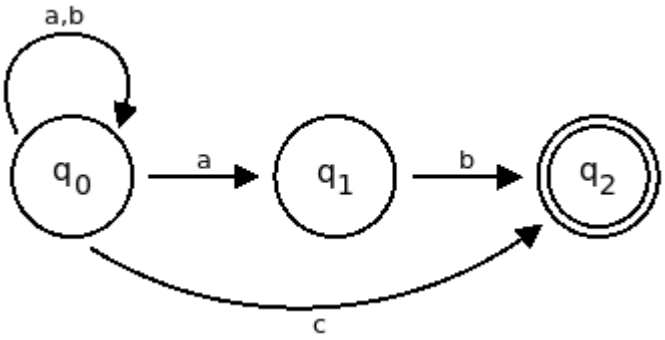
## Algorithm

Given NFA N = (Σ, Q, $q_0$, F, δ) we want to build DFA D= (Σ, Q', $q'_0$, F', δ'). Here's how:

1. Initially Q' = ∅.
2. Add $q_0$ to Q'.
3. For each state in Q find the *set* of possible states for each input symbol using N's transition table, δ. Add this set of states to Q', if it is not already there.
4. The set of final states of D, F', will be all of the the states in Q' that contain in them a state that is in F.

Working through an example may aid in understanding these steps.

Consider the following NFA N = (Σ, Q, $q_0$, F, δ)




Σ = {a, b, c}
Q = {$q_0$, $q_1$, $q_2$}
F = {$q_2$}

We wish to construct DFA D= (Σ, Q', $q'_0$, F', δ'). Following the steps in the conversion algorithm:

1. Q' = ∅
2. Q' = {$q_0$}
3. For every state in Q, find the set of states for each input symbol using using N's transition table, δ. If the set is not Q, add it to Q'.We can start building the transition table δ' for D by first examining $q_0$.

.

| state | a | b | c |
|---|---|---|---|
| $q_0$ | {$q_0$, $q_1$} | $q_0$ | $q_2$ |

$q_0$ is already in Q' so we don't add it. {$q_0$, $q_1$} is considered a single state, so we add it and $q_2$ to Q'.

Q' = {$q_0$, {$q_0$, $q_1$}, $q_2$}

δ' now looks like:

| state | a | b | c |
|---|---|---|---|
| $q_0$ | {$q_0$, $q_1$} | $q_0$ | $q_2$ |
| {$q_0$, $q_1$} | ? | ? | ? |

$q_2$ ? ? ?

1. Now calculate the transitions for $\{q_0, q_2\}$:

   $\delta'(\{q_0, q_2\}, a) = \delta(q_0,a) \cup \delta(q_2,a) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
   $\delta'(\{q_0, q_2\}, b) = \delta(q_0,b) \cup \delta(q_2,b) = \{q_0\} \cup \emptyset = \{q_0\}$
   $\delta'(\{q_0, q_2\}, c) = \delta(q_0,c) \cup \delta(q_2,c) = \{q_2\} \cup \emptyset = \{q_2\}$

   There are no new states to add to Q'. The updated transition table looks like:

| **state** | **a** | **b** | **c** |
|---|---|---|---|
| $q_0$ | $\{q_0, q_1\}$ | $q_0$ | $q_2$ |
| $\{q_0, q_1\}$ | $\{q_0, q_1\}$ | $\{q_0, q_2\}$ | $q_2$ |
| $q_2$ | – | – | – |
| $\{q_0, q_2\}$ | $\{q_0, q_1\}$ | $q_0$ | $q_2$ |

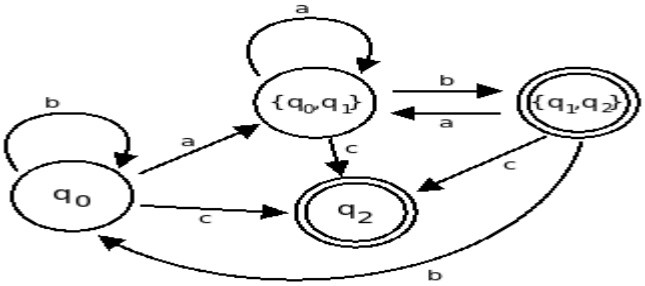
Since we've inspected all of the states in Q and no longer have any states to add to Q', we arefinished. D's set of final states F' are those states that include states in F. So, F' = $\{q_2, \{q_0, q_2\}\}$.

For our completed DFA D= ($\Sigma$, Q', $q'_0$, F', $\delta$'):

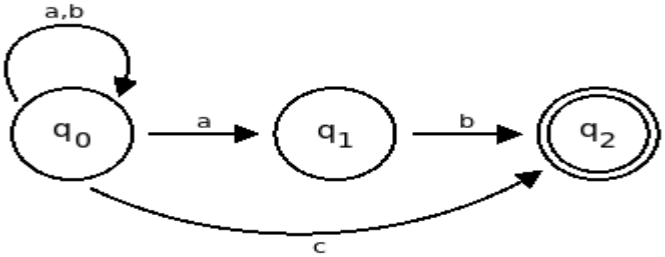
- $\Sigma = \{a, b\}$
- $Q' = \{q_0, \{q_0, q_1\}, q_2, \{q_0, q_2\}\}$
- $q'_0 = q_0$
- $F' = \{q_2, \{q_0, q_2\}\}$
- $\delta'$ is:

| **state** | **a** | **B** | **c** |
|---|---|---|---|
| $q_0$ | $\{q_0, q_1\}$ | $q_0$ | $q_2$ |
| $\{q_0, q_1\}$ | $\{q_0, q_1\}$ | $\{q_0, q_2\}$ | $q_2$ |
| $q_2$ | – | – | – |
| $\{q_0, q_2\}$ | $\{q_0, q_1\}$ | $q_0$ | $q_2$ |

The transition graph for this DFA looks like:




Again for reference, the original NFA was:



| |
|---|
| **Video Content / Details of website for further learning (if any):**<br>https://www.neuraldump.net/2017/11/nfa-and-dfa-equivalence-theorem-proof-and-example/<br>http://www.digimat.in/nptel/courses/video/106104148/L07.html |
| **Important Books/Journals for further learning including the page nos.:**<br>Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second. T1(77-82) |

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-8**

**LECTURE HANDOUTS**

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

| |
|---|
| **Topic of Lecture:** Equivalence of NDFA's with and without €-moves |
| **Introduction : ( Maximum 5 sentences)** :<br>NFA's with ε −Transitions<br>• We extend the class of NFAs by allowing instantaneous (ε) transitions:<br>1. The automaton may be allowed to change its state without reading the input symbol.<br>2. In diagrams, such transitions are depicted by labeling the appropriate arcs with ε.<br>3. Note that this does not mean that ε has become an input symbol. On the contrary, we assume that the symbol ε does not belong to any alphabet. |
| **Prerequisite knowledge for Complete understanding and learning of Topic:**<br>**( Max. Four important topics)**<br>Deterministic and Non-daterministic Automata |
| **Detailed content of the Lecture:**<br>A ε-NFA is a quintuple A=(Q,Σ,δ,q0,F) where – Q is a set of states – Σ is the alphabet of input symbols – q0∈Q is the initial state – F ⊆ Q is the set of final states – δ: Q × Σε □⟶ P(Q) is the transition function • Note ε is never a member of Σ • Σε is defined to be (Σ ∪ ε)<br>**ε-NFA** |

• ε -NFAs add a convenient feature but (in a sense) they bring us nothing new:
they do not extend the class of languages that can be represented. Both NFAs and ε-NFAs recognize exactly the same languages.
• ε-transitions are a convenient feature: try to design an NFA for the even or divisible by 3 language that does not use them! – Hint, you need to use something like the product construction from union-closure of DFAs

**ε-Closure**

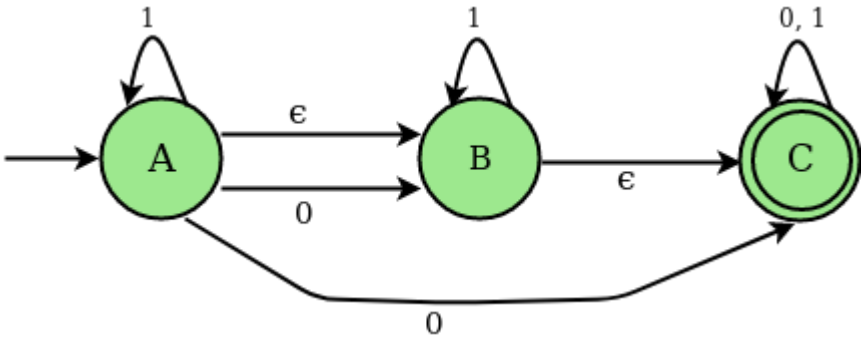
• ε-closure of a state
• The ε-closure of the state q, denoted ECLOSE(q), is the set that contains q, together with all states that can be reached starting at q by following only ε-transitions.
• In the above example: • ECLOSE(P) ={P,Q,R,S} • ECLOSE(R)={R,S} • ECLOSE(x)={x} for the remaining 5 states {Q,Q1,R1,R2,R2}

**Computing eclose:**

• Compute eclose by adding new states until no new states can be added
• Start with [P]
• Add Q and R to get [P,Q,R]
• Add S to get [P,Q,R S]
• No new states can be added

**Elimination of ε-Transitions**

• Given an ε-NFA N, this construction produces an NFA N' such that L(N')=L(N). • The construction of N' begins with N as input, and takes 3 steps: 1. Make p an accepting state of N' iff ECLOSE(p) contains an accepting state of N. 2. Add an arc from p to q labeled a iff there is an arc labeled a in N from some state in ECLOSE(p) to q. 3. Delete all arcs labeled ε.




**Transition state table for the above NFA**

| STATES | 0 | 1 | EPSILON |
|---|---|---|---|
| A | B, C | A | B |
| B | – | B | C |
| C | C | C | – |

**Video Content / Details of website for further learning (if any):**

http://web.cecs.pdx.edu/~sheard/course/CS581/notes/NfaEpsilonDefined.pdf

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second. T1(97-98)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

Estd. 2000

IQAC

**L-9**

## LECTURE HANDOUTS

**CSE**

**IV/II/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **I Finite Automata** **Date of Lecture:**

**Topic of Lecture:** Minimization of DFA

**Introduction : ( Maximum 5 sentences)** :

- DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states

**Prerequisite knowledge for Complete understanding and learning of Topic:**
**( Max. Four important topics)**

- **DFA**

**Detailed content of the Lecture:**

DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states.

**Minimization of DFA**

Suppose there is a DFA D < Q, Σ, q0, δ, F > which recognizes a language L. Then the minimized DFA D < Q', Σ, q0, δ', F' > can be constructed for language L as:

**Step 1:** We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called $P_0$.

**Step 2:** Initialize k = 1

**Step 3:** Find $P_k$ by partitioning the different sets of $P_{k-1}$. In each set of $P_{k-1}$, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in $P_k$.

**Step 4:** Stop when $P_k = P_{k-1}$ (No change in partition)

**Step 5:** All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in

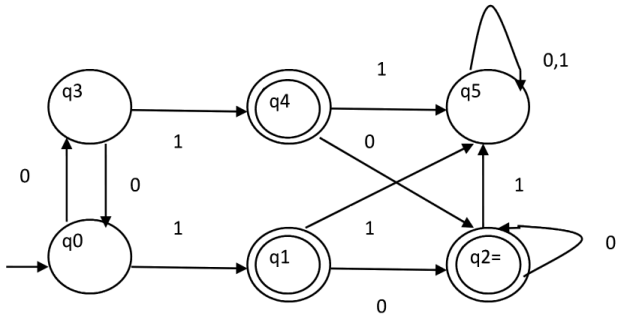
$P_k$.

**How to find whether two states in partition $P_k$ are distinguishable ?**

Two states ( qi, qj ) are distinguishable in partition $P_k$ if for any input symbol a, δ ( qi, a ) and δ ( qj, a ) are in different sets in partition $P_{k-1}$.

**Example**

Consider the following DFA shown in figure.




**Step 1.** P0 will have two sets of states. One set will contain q1, q2, q4 which are final states of DFA and another set will contain remaining states. So P0 = { { q1, q2, q4 }, { q0, q3, q5 } }.

**Step 2.** To calculate P1, we will check whether sets of partition P0 can be partitioned or not: **i) For set { q1, q2, q4 } :**

δ ( q1, 0 ) = δ ( q2, 0 ) = q2 and δ ( q1, 1 ) = δ ( q2, 1 ) = q5, So q1 and q2 are not distinguishable.

Similarly, δ ( q1, 0 ) = δ ( q4, 0 ) = q2 and δ ( q1, 1 ) = δ ( q4, 1 ) = q5, So q1 and q4 are not distinguishable.

Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P1.

**ii) For set { q0, q3, q5 } :**

δ ( q0, 0 ) = q3 and δ ( q3, 0 ) = q0

δ ( q0, 1) = q1 and δ( q3, 1 ) = q4

Moves of q0 and q3 on input symbol 0 are q3 and q0 respectively which are in same set in partition P0.

Similarly, Moves of q0 and q3 on input symbol 1 are q3 and q0 which are in same set in partition P0. So, q0 and q3 are not distinguishable.

δ ( q0, 0 ) = q3 and δ ( q5, 0 ) = q5 and δ ( q0, 1 ) = q1 and δ ( q5, 1 ) = q5

Moves of q0 and q5 on input symbol 1 are q3 and q5 respectively which are in different set in partition P0. So, q0 and q5 are distinguishable. So, set { q0, q3, q5 } will be partitioned into { q0, q3 } and { q5 }. So,

P1 = { { q1, q2, q4 }, { q0, q3}, { q5 } }

To calculate P2, we will check whether sets of partition P1 can be partitioned or not:

**iii)For set { q1, q2, q4 } :**

δ ( q1, 0 ) = δ ( q2, 0 ) = q2 and δ ( q1, 1 ) = δ ( q2, 1 ) = q5, So q1 and q2 are not distinguishable.

Similarly, δ ( q1, 0 ) = δ ( q4, 0 ) = q2 and δ ( q1, 1 ) = δ ( q4, 1 ) = q5, So q1 and q4 are not distinguishable.

Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P2.

**iv)For set { q0, q3 } :**

δ ( q0, 0 ) = q3 and δ ( q3, 0 ) = q0

δ ( q0, 1 ) = q1 and δ ( q3, 1 ) = q4

Moves of q0 and q3 on input symbol 0 are q3 and q0 respectively which are in same set in partition P1.

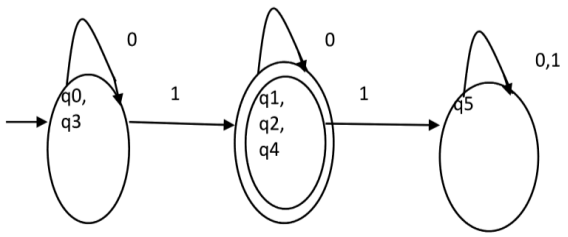
Similarly, Moves of q0 and q3 on input symbol 1 are q3 and q0 which are in same set in partition P1. So, q0 and q3 are not distinguishable.

**v) For set { q5 }:**

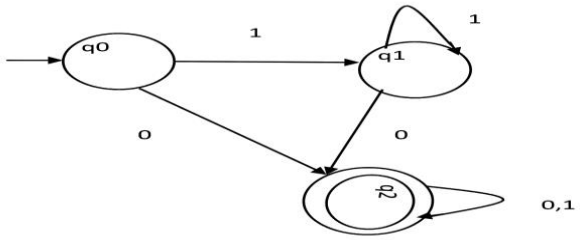
Since we have only one state in this set, it can't be further partitioned. So,

P2 = { { q1, q2, q4 }, { q0, q3 }, { q5 } }

Since, P1=P2. So, this is the final partition. Partition P2 means that q1, q2 and q4 states are merged into one. Similarly, q0 and q3 are merged into one. Minimized DFA corresponding to DFA of Figure 1 is shown in Figure 2 as:



**Question :** Consider the given DFA. Which of the following is false?
1. Complement of L(A) is context-free.
2. L(A) = L ( ( 11 * 0 + 0 ) ( 0 + 1 )* 0* 1* )
3. For the language accepted by A, A is the minimal DFA.
4. A accepts all strings over { 0, 1 } of length atleast two.




A. 1 and 3 only
B. 2 and 4 only
C. 2 and 3 only
D. 3 and 4 only

**Solution :** Statement 4 says, it will accept all strings of length atleast 2. But it accepts 0 which is of length 1. So, 4 is false.
Statement 3 says that the DFA is minimal. We will check using the algorithm discussed above.
P0 = { { q2 }, { q0, q1 } }
P1 = { q2 }, { q0, q1 } }. Since, P0 = P1, P1 is the final DFA. q0 and q1 can be merged. So minimal DFA will have two states. Therefore, statement 3 is also false.
So correct option is (D).
This article has been contributed by Sonal Tuteja.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Video Content / Details of website for further learning (if any):**
https://www.geeksforgeeks.org/minimization-of-dfa/

**Important Books/Journals for further learning including the page nos.:**
Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second. T1(173-181)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-16**

**CSE**

**II/IV/A**

**Course Name with Code** : Theory of Computation -16CSD11

**Course Teacher** :K.PAPITHASRI

**Unit** : II Regular Languages. **Date of Lecture:**

**Topic of Lecture: Chomsky hierarchy of language**

**Introduction : ( Maximum 5 sentences)** :

- Chomsky Hierarchy is a broad classification of the various types of grammar available
- These include Unrestricted grammar, context-free grammar, context-sensitive grammar and restricted grammar
- Grammars are classified by the form of their productions.
- Each category represents a class of languages that can be recognized by a different automaton.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Finite Automata-NFA,DFA
- Regular Expression
- Regular Languages

**Detailed content of the Lecture:**

Chomsky Hierarchy is a broad classification of the various types of grammar available

These include Unrestricted grammar, context-free grammar, context-sensitive grammar and restricted grammar

Grammars are classified by the form of their productions.

Each category represents a class of languages that can be recognized by a different automaton

The classes are nested, with type 0 being the largest and most general, and type 3 being the smallest and most restricted.

**Type 0 :Unrestricted grammar :** This type of grammar requires LHS to contain atleast one non-terminal

**Applications:** Unrestricted language recursively enumerable automaton Turing machine.

– $x \rightarrow y$ – $x \in ((V \cup T)*V\ (V \cup T)*)$ – $y \in (V \cup T)*$

– LHS contains at least one non-terminal

– generate recursively enumerable languages

– recognized by Turing Machines

– example :

generates L = {ww | w ∈ (a + b)*}

Generate wCwR E

C is a temporary center marker

E is a wall

∗ Reverse WRwR by pushing symbols in WRwR, leftmost first, over the wall with a pusher P
∗ Clean up by removing C and E

V = {S, T, C, P}

T = {a, b} P :

1. S → TE
2. T → aTa
3. T→ | bTb
4. T→ | C
5. C → CP
6. Paa → aPa
7. Pab → bPa
8. Pba → aPb
9. Pbb → bPb
10. PaE → Ea
11. PbE → Eb
12. CE → ϵ

**Type 1 : Context-sensitive grammar**

**Applications**: context-sensitive language context-sensitive automaton linear-bounded automaton – x → y

or S → ϵ and S is not in RHS.

– x∈ $((V \cup T)*V\ (V \cup T)*)$

– y∈ $(V \cup T)+$ – |x| ≤ |y| – non-contracting

– generate context-sensitive languages

– recognized by linear-bounded automata

– example

V = {S, A, B}

T = {a, b, c}

P :

1. S → abc
2. | aAbc
3. Ab → bA
4. Ac → Bbcc
5. bB → Bb
6. aB → aa
7. aB → aaA

generates L = $a^n b^n c^n | n \geq 1$

**Type 2 : Context-free** -It is a grammar that is used to generate languages recursively and requires it to be in Chomsky Normal Form or the Griebach Normal Form

**Applications:** context-free language context-free automaton pushdown automaton

– x → y

– x∈ V

– y∈ (V ∪ T)*

– generate context-free languages

– recognized by pushdown automata

– example

V = {S}

T = {a, b}

P :

1. S → aSb
2. S → |ϵ

generates L = {a^nb^n| n ≥ 0}

**Type 3 : Regular grammar**

– A regular grammar is right-linear or left- linear (but not both)

– A, B ∈ V

– y∈ T*

– Right-Linear productions have the form A → yB or A → y

– Left-Linear productions have the form A → By or A → y

– generate regular languages

– recognized by finite state automata

– example

V = {S}

T = {a}

P :

1. S → aS
2. . S → |ϵ

generates L = $a_n$;|n≥0

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/layers-of-osi-model/

https://www.youtube.com/watch?v=EzLMMsRR6Js

https://nptel.ac.in/courses/106106049/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second .T1(82-85).

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-17**

**LECTURE HANDOUTS**

**CSE** | **II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Regular Languages- Regular Expression

**Introduction : ( Maximum 5 sentences)** :

- Regular languages are *languages that can be generated from one-element languages by applying certain standard operations a finite number of times*.
- Regular expressions can be thought of as *the algebraic description of a regular language*.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Finite state System

Finite Automata

**Detailed content of the Lecture:**

**Regular Expressions and Regular Languages**

- Regular languages are *languages that can be generated from one-element languages by applying certain standard operations a finite number of times*. They are the languages that can be recognized by finite automata. These simple operations include concatenation, union and kleen closure. By the use of these operations regular languages can be represented by an explicit formula.
- Regular expressions can be thought of as *the algebraic description of a regular language*. Regular expression can be *defined by the following rules*:

1. Every letter of the alphabet $\sum$ is a regular expression.
2. Null string $\epsilon$ and empty set $\Phi$ are regular expressions.
3. If r1 and r2 are regular expressions, then
   (i)r1, r2
   (ii)r1r2 (concatenation of r1r2 )
   (iii) r1 + r2 ( union of r1 and r2 )
   (iv) r1*, r2* ( kleen closure of r1 and r2 ) are also regular expressions

4. If a string can be derived from the rules 1, 2 and 3 then it is also a regular expression.

Note that a* means zero or more occurrence of a in the string while a+ means that one or more occurrence of a in the string. That means a* denotes language L = {є , a, aa, aaa, ….} and a+ represents language L = {a, aa, aaa, ….}. And also note that there can be more than one regular expression for a given set of strings.

**Order for precedence for the operations** is: kleen > concatenation > union. This rule allows us to lessen the use of parentheses while writing the regular expression. For example a + b*c is the simplified form of (a + ((b)*c)). Note that (a + b)* is not the same as a + b*, a + b* is (a + (b)*).

**Example 1:** Write a regular expression for a set of strings of 0s and 1s with even number of 0s.

**Solution:**

The set of strings would be 00, 001, 0011, 1001, 1010, …. The regular expression for the above set of strings can be written as (((00)*1*) + (01*0))*.

For a regular expression r, we denote the language it represents as L(r).

Some **rules applicable on regular languages** are as follows:
For two regular expressions r1 and r2

1. r1 + r2 is a regular expression denoting union of L(r1) and L(r2). That is
   L(r1 + r2) = L(r1) U L(r2)
2. r1r2 is a regular expression denoting the concatenation of L(r1) and L(r2). That is
   L(r1r2) = L(r1) . L(r2)
3. r* is a regular expression denoting the closure of L(r). That is
   L(r*) = L(r)*

These rules will be used to create regular expressions from a given languages.
For *example* consider the language L = {$a^n$ $b^m$ : (n + m) is even}. For writing the regular expression we will have to consider two cases
Case 1: n and m both are even
Case 2: n and m are both odd.
Let the regular expression for case 1 be r1, then
r1 = (aa)* (bb)* (as even numbers can be represented as 2n)
Let the regular expression for case 2 be r1, then
r2 = (aa)* a (bb)* b (as odd numbers can be represented as 2n + 1)
Let the regular expression for the language L be r, then
r = r1 + r1 (as it will be the union of both the cases described above)
r = ((aa)* (bb)*) + (aa)* a (bb)* b)) is the regular expression for the given language.

**Example 2:** Write a regular expression for the language containing odd number of 1s, ∑ = {0,1}.
**Solution:**

The language will contain at least one 1. It may contain any number of 0s anywhere in the string. So the language we have to write a regular expression for is 1, 01, 01101, 0111, 111, . . . . This language

| |
|---|
| can be represented by the following regular expression<br>0*(10*10*)10* |
| **Video Content / Details of website for further learning (if any):**<br><br>https://www.sanfoundry.com/automata-theory-regular-expressions-languages/<br><br>https://www.youtube.com/watch?v=37Yr4fS0-po |
| **Important Books/Journals for further learning including the page nos.:**<br><br>Harry R Lewis and Christos H Papadimitriou Elements of the Theory of Computation Prentice Hall of India, Pearson Education Second T2(85-112) |

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-18**

**LECTURE HANDOUTS**

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Equivalence of finite Automaton

**Introduction : ( Maximum 5 sentences)** :

- 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.
- Just as we built a small FA for each operator and operand in a regular expression, we will now build a small regular expression for each state in the DFA.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Definition of Automata and Mathematical Notations

**Detailed content of the Lecture:**

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.

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.

Operators 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.

**Examples**

The set of strings over {0,1} that end in 3 consecutive 1's.

```
(0 | 1)* 111
```

The set of strings over {0,1} that have at least one 1.

```
0* 1 (0 | 1)*
```

The set of strings over {0,1} that have at most one 1.

```
0* | 0* 1 0*
```

The set of strings over {A..Z,a..z} that contain the word "main".

```
Let <letter> = A | B | ... | Z | a | b | ... | z

<letter>* main <letter>*
```

The set of strings over {A..Z,a..z} that contain 3 x's.

```
<letter>* x <letter>* x <letter>* x <letter>*
```

**Unix Operator Extensions**

Regular expressions are used frequently in Unix:

- In the command line
- Within text editors
- In the context of pattern matching programs such as *grep* and *egrep*

To facilitate construction of regular expressions, Unix recognizes additional operators. These operators can be defined in terms of the operators given above; they represent a notational convenience only.

- character classes: '[' <list of chars> ']'
- start of a line: '^'
- end of a line: '$'
- wildcard matching any character except newline: '.'
- optional instance: R? = epsilon | R
- one or more instances: R+ == RR*

**Equivalence of Regular Expressions and Finite Automata**

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.

**Constructing an FA from an RE**

We begin by showing how to construct an FA for the operands in a regular expression.

- If the operand is a character c, then our FA has two states, s0 (the start state) and sF (the final, accepting state), and a transition from s0 to sF with label c.
- If the operand is epsilon, then our FA has two states, s0 (the start state) and sF (the final, accepting state), and an epsilon transition from s0 to sF.
- If the operand is null, then our FA has two states, s0 (the start state) and sF (the final, accepting state), and no transitions.

Given FA for R1 and R2, we now show how to build an FA for R1R2, R1|R2, and R1*. Let A (with start state a0 and final state aF) be the machine accepting L(R1) and B (with start state b0 and final state bF) be the machine accepting L(R2).

- The machine C accepting L(R1R2) includes A and B, with start state a0, final state bF, and an epsilon transition from aF to b0.
- The machine C accepting L(R1|R2) includes A and B, with a new start state c0, a new final state cF, and epsilon transitions from c0 to a0 and b0, and from aF and bF to cF.
- The machine C accepting L(R1*) includes A, with a new start state c0, a new final state cF, and epsilon transitions from c0 to a0 and cF, and from aF to a0, and from aF to cF.

**Video Content / Details of website for further learning (if any):**

https://www.cs.rochester.edu/u/nelson/courses/csc_173/fa/re.html

https://www.youtube.com/watch?v=dBwx2PVicTY

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second .T1(110-120)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-19**

**LECTURE HANDOUTS**

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Equivalence of regular expression

**Introduction : ( Maximum 5 sentences)** :

- 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.
- Just as we built a small FA for each operator and operand in a regular expression, we will now build a small regular expression for each state in the DFA.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Definition of Automata and Mathematical Notations

**Detailed content of the Lecture:**

- The machine C accepting L(R1R2) includes A and B, with start state a0, final state bF, and an epsilon transition from aF to b0.
- The machine C accepting L(R1|R2) includes A and B, with a new start state c0, a new final state cF, and epsilon transitions from c0 to a0 and b0, and from aF and bF to cF.
- The machine C accepting L(R1*) includes A, with a new start state c0, a new final state cF, and epsilon transitions from c0 to a0 and cF, and from aF to a0, and from aF to cF.

**Eliminating Epsilon Transitions**

If we can eliminate epsilon transitions from an FA, then our construction of an FA from a regular expression (which yields an FA with epsilon transitions) can be completed.

Observe that epsilon transitions are similar to nondeterminism in that they offer a choice: an epsilon transition allows us to stay in a state or move to a new state, regardless of the input symbol.

If starting in state s1, we can reach state s2 via a series of epsilon transitions followed by a transition on input symbol x, we can replace all of the epsilon transitions with a single transition from s1 to s2 on symbol x.

**Algorithm for Eliminating Epsilon Transitions**

We can build a finite automaton F2 with no epsilon transitions from a finite automaton F1 containing epsilon transitions as follows:

1. The states of F2 are all the states of F1 that have an entering transition labeled by some symbol other than epsilon, plus the start state of F1, which is also the start state of F2.
2. For each state in F1, determine which other states are reachable via epsilon transitions only. If a state of F1 can reach a final state in F1 via epsilon transitions, then the corresponding state is a final state in F2.
3. For each pair of states i and j in F2, there is a transition from state i to state j on input x if there exists a state k that is reachable from state i via epsilon transitions in F1, and there is a transition in F1 from state k to state j on input x.

**Constructing an RE from an FA**

To construct a regular expression from a DFA (and thereby complete the proof that regular expressions and finite automata have the same expressive power), we replace each state in the DFA one by one with a corresponding regular expression.

Just as we built a small FA for each operator and operand in a regular expression, we will now build a small regular expression for each state in the DFA.

The basic idea is to eliminate the states of the FA one by one, replacing each state with a regular expression that generates the portion of the input string that labels the transitions into and out of the state being eliminated.

**Algorithm for Constructing an RE from an FA**

Given a DFA F we construct a regular expression R such that
L(F) == L(R).

We preprocess the FA, turning the labels on transitions into regular expressions. If there is a transition with label {a,b}, then we replace the label with the regular expression a | b. If there is no transition from a state to itself, we can add one with the label NULL.

For each accepting state sF in F, eliminate all states in F except the start state s0 and sF.

To eliminate a state sE, consider all pairs of states sA and sB such that there is a transition from sA to sE with label R1, a transition from sE to sE with label R2 (possibly null, meaning no transition), and a transition from sE to sB with label R3. Introduce a transition from sA to sB with label R1R2*R3. If there is already a transition from sA to sB with label R4, then replace that label with R4|R1R2*R3.

After eliminating all states except s0 and sF:

- If s0 == sF, then the resulting regular expression is R1*, where R is the label on the transition from s0 to s0.
- If s0 != sF, then assume the transition from s0 to s0 is labeled R1, the transition from s0 to sF is labeled R2, the transition from sF to sF is labeled R3, and the transition from sF to s0 is labeled R4. The resulting regular expression is R1*R2(R3 | R4R1*R2)*

Let RFi be the regular expression produced by eliminating all the states except s0 and sFi. If there are n final states in the DFA, then the regular expression that generates the strings accepted by the original DFA is RF1 | RF2 | ... RFn.

**Video Content / Details of website for further learning (if any):**

https://www.cs.rochester.edu/u/nelson/courses/csc_173/fa/re.html

https://www.youtube.com/watch?v=dBwx2PVicTY

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second .T1(110-120)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-20**

**LECTURE HANDOUTS**

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Pumping Lemma for Regular sets

**Introduction : ( Maximum 5 sentences)** :

- **pumping lemma** may refer to: **Pumping lemma** for regular languages, the fact that all sufficiently long strings in such a language have a substring that can be repeated arbitrarily many times, usually used to prove that certain languages are not regular
- **pumping lemma for regular languages** is a lemma that describes an essential property of all regular languages.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Regular Expression

Regular Languages

**Detailed content of the Lecture:**

There are two Pumping Lemmas, which are defined for
1. Regular Languages, and
2. Context – Free Languages

**Pumping Lemma for Regular Languages**

For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists $u, v, w \in \Sigma*$, such that $x = uvw$, and

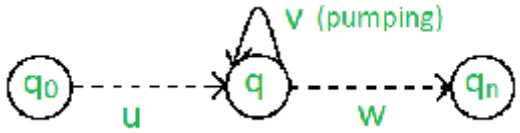
(1) $|uv| \leq n$

(2) $|v| \geq 1$

(3) for all $i \geq 0$: $uv^iw \in L$

In simple terms, this means that if a string v is 'pumped', i.e., if v is inserted any number of times, the resultant string still remains in L.

Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.
The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.



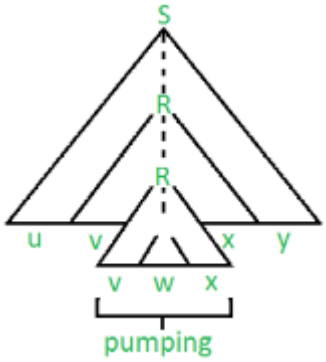

For example, let us prove $L_{01} = \{0^n1^n \mid n \geq 0\}$ is irregular.
Let us assume that L is regular, then by Pumping Lemma the above given rules follow.
Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w such that (1) – (3) hold.

We show that for all u, v, w, (1) – (3) does not hold.
If (1) and (2) hold then $x = 0^n1^n = uvw$ with $|uv| \leq n$ and $|v| \geq 1$.
So, $u = 0^a$, $v = 0^b$, $w = 0^c1^n$ where : $a + b \leq n$, $b \geq 1$, $c \geq 0$, $a + b + c = n$
But, then (3) fails for i = 0
$uv^0w = uw = 0^a0^c1^n = 0^{a+c}1^n \notin L$, since $a + c \neq n$.


**Pumping Lemma for Context-free Languages (CFL)**

Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language. For any language L, we break its strings into five parts and pump second and fourth substring.
Pumping Lemma, here also, is used as a tool to prove that a language is not CFL. Because, if any one string does not satisfy its conditions, then the language is not CFL.
Thus, if L is a CFL, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, w, x, $y \in \Sigma*$, such that $x = uvwxy$, and
(1) $|vwx| \leq n$
(2) $|vx| \geq 1$

(3) for all $i \geq 0$: $uv^iwx^iy \in L$




For above example, $0^n1^n$ is CFL, as any string can be the result of pumping at two places, one for 0 and other for 1.
Let us prove, $L_{012} = \{0^n1^n2^n \mid n \geq 0\}$ is not Context-free.
Let us assume that L is Context-free, then by Pumping Lemma, the above given rules follow.
Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w, x, y such that (1) – (3) hold.
We show that for all u, v, w, x, y (1) – (3) do not hold.
If (1) and (2) hold then $x = 0^n1^n2^n = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$.
(1) tells us that vwx does not contain both 0 and 2. Thus, either vwx has no 0's, or vwx has no 2's.
Thus, we have two cases to consider.
Suppose vwx has no 0's. By (2), vx contains a 1 or a 2. Thus uwy has 'n' 0's and uwy either has less than 'n' 1's or has less than 'n' 2's.
But (3) tells us that $uwy = uv^0wx^0y \in L$.
So, uwy has an equal number of 0's, 1's and 2's gives us a contradiction. The case where vwx has no 2's is similar and also gives us a contradiction. Thus L is not context-free.

## PROPERTIES OF REGULAR SETS

**Property 1**. *The union of two regular set is regular.*
**Proof** –
Let us take two regular expressions
$RE_1 = a(aa)^*$ and $RE_2 = (aa)^*$
So, $L_1 = \{a, aaa, aaaaa,.....\}$ (Strings of odd length excluding Null)
and $L_2 = \{ \varepsilon, aa, aaaa, aaaaaa,.....\}$ (Strings of even length including Null)
$L_1 \cup L_2 = \{ \varepsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa,.....\}$
(Strings of all possible lengths including Null)
$RE (L_1 \cup L_2) = a^*$ (which is a regular expression itself)
**Hence, proved.**
**Property 2.** *The intersection of two regular set is regular.*
**Proof** –
Let us take two regular expressions
$RE_1 = a(a^*)$ and $RE_2 = (aa)^*$
So, $L_1 = \{ a,aa, aaa, aaaa, ....\}$ (Strings of all possible lengths excluding Null)
$L_2 = \{ \varepsilon, aa, aaaa, aaaaaa,.....\}$ (Strings of even length including Null)
$L_1 \cap L_2 = \{ aa, aaaa, aaaaaa,.....\}$ (Strings of even length excluding Null)
$RE (L_1 \cap L_2) = aa(aa)^*$ which is a regular expression itself.

**Hence, proved.**

**Property 3.** *The complement of a regular set is regular.*

**Proof** −

Let us take a regular expression −

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, .....} (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L' = {a, aaa, aaaaa, .....} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

**Hence, proved.**

**Property 4.** *The difference of two regular set is regular.*

**Proof** −

Let us take two regular expressions −

$RE_1$ = a (a*) and $RE_2$ = (aa)*

So, $L_1$ = {a, aa, aaa, aaaa, ....} (Strings of all possible lengths excluding Null)

$L_2$ = { ε, aa, aaaa, aaaaaa,.....} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ....}

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

**Hence, proved.**

**Property 5.** *The reversal of a regular set is regular.*

**Proof** −

We have to prove $\mathbf{L^R}$ is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

$L^R$ = {10, 01, 11, 01}

RE ($L^R$) = 01 + 10 + 11 + 10 which is regular

**Hence, proved.**

**Property 6.** *The closure of a regular set is regular.*

**Proof** −

If L = {a, aaa, aaaaa, .....} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa,……} (Strings of all lengths excluding Null)

RE (L*) = a (a)*

**Hence, proved.**

**Property 7.** *The concatenation of two regular sets is regular.*

**Proof −**

Let $RE_1$ = (0+1)*0 and $RE_2$ = 01(0+1)*

Here, $L_1$ = {0, 00, 10, 000, 010, .....} (Set of strings ending in 0)

and $L_2$ = {01, 010,011,.....} (Set of strings beginning with 01)

Then, $L_1 L_2$ = {001,0010,0011,0001,00010,00011,1001,10010,.....}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Hence, proved.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/pumping-lemma-in-theory-of-computation/

| |
|---|
| https://www.tutorialspoint.com/automata_theory/regular_sets.htm<br><br>https://www.youtube.com/watch?v=cyAMe39JQ3Y |
| **Important Books/Journals for further learning including the page nos.:**<br><br>John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2003). Introduction to Automata Theory, Languages, and Computation. T1(146-147) |

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-21**

**LECTURE HANDOUTS**

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Problems based on Pumping Lemma

**Introduction : ( Maximum 5 sentences)** :

- *Pumping Lemma for Regular* Languages In simple terms, this means that if a string v is '*pumped*', i.e., if v is inserted any number of times, the resultant string still remains in L.
- *Pumping Lemma* is used as a proof for irregularity of a language. Thus, if a language is *regular*, it always satisfies *pumping lemma*.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Regular Expression

Regular Languages

**Detailed content of the Lecture:**

**Theorem**

Let L be a regular language. Then there exists a constant **'c'** such that for every string **w** in **L** −**|w| ≥ c**
We can break **w** into three strings, **w = xyz**, such that −

- |y| > 0
- |xy| ≤ c
- For all k ≥ 0, the string $xy^kz$ is also in L.

## *Applications of Pumping Lemma*

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If L is regular, it satisfies Pumping Lemma.
- If L does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that L is regular.
- So, the pumping lemma should hold for L.
- Use the pumping lemma to obtain a contradiction −
  - Select **w** such that **|w| ≥ c**
  - Select **y** such that **|y| ≥ 1**
  - Select **x** such that **|xy| ≤ c**
  - Assign the remaining string to **z.**
  - Select **k** such that the resulting string is not in **L.**

**Hence L is not regular.**

**Problem**

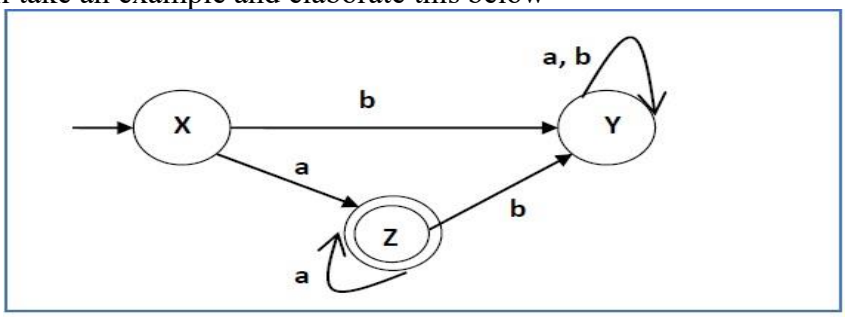
Prove that $\mathbf{L = \{a^i b^i \mid i \geq 0\}}$ is not regular.

***Solution*** −

- At first, we assume that **L** is regular and n is the number of states.
- Let w = $a^n b^n$. Thus |w| = 2n ≥ n.
- By pumping lemma, let w = xyz, where |xy| ≤ n.
- Let x = $a^p$, y = $a^q$, and z = $a^r b^n$, where p + q + r = n, p ≠ 0, q ≠ 0, r ≠ 0. Thus |y| ≠ 0.
- Let k = 2. Then $xy^2z = a^p a^{2q} a^r b^n$.
- Number of as = (p + 2q + r) = (p + q + r) + q = n + q
- Hence, $xy^2z = a^{n+q} b^n$. Since q ≠ 0, $xy^2z$ is not of the form $a^n b^n$.
- Thus, $xy^2z$ is not in L. Hence L is not regular.

If (Q, ∑, δ, $q_0$, F) be a DFA that accepts a language L, then the complement of the DFA can be obtained by swapping its accepting states with its non-accepting states and vice versa.

We will take an example and elaborate this below −




**DFA accepting language L**

This DFA accepts the language

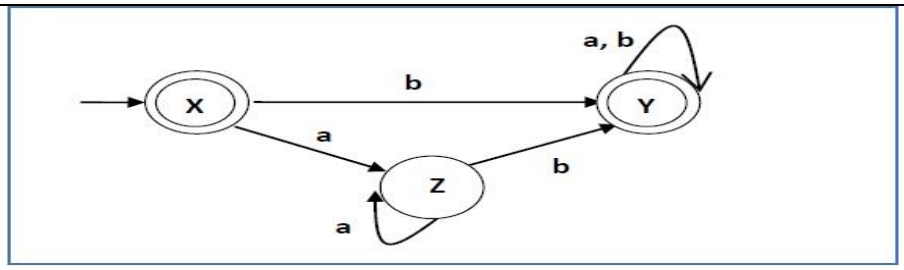
L = {a, aa, aaa , ..... }

over the alphabet

∑ = {a, b}

So, RE = $a^+$.

Now we will swap its accepting states with its non-accepting states and vice versa and will get the following −




DFA accepting complement of language L

This DFA accepts the language
L' = {ε, b, ab ,bb,ba, ..... }
over the alphabet
∑ = {a, b}

**Note** − If we want to complement an NFA, we have to first convert it to DFA and then have to swap states as in the previous method.

***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

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/pumping_lemma_for_regular_grammar.htm/

https://nptel.ac.in/courses/106106049/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second .T1(147-149)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-22**

## LECTURE HANDOUTS

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Closure Properties of Regular Languages.

**Introduction : ( Maximum 5 sentences)** :

- **Closure properties** on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language.
- Closure refers to some operation on a language, resulting in a new language that is of same "type" as originally operated on i.e., regular.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Regular Expression

Regular Sets

**Detailed content of the Lecture:**

**Closure properties** on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language. Closure refers to some operation on a language, resulting in a new language that is of same "type" as originally operated on i.e., regular.

Regular languages are closed under following operations.

**Consider L and M are regular languages:**

1. **Kleen Closure:**
   RS is a regular expression whose language is L, M. R* is a regular expression whose language is L*.

2. **Positive closure:**
   RS is a regular expression whose language is L, M. $R^+$ is a regular expression whose language is $L^+$.
3. **Complement:**
   The complement of a language L (with respect to an alphabet $E$ such that $E^*$ contains L) is $E^*$–L. Since $E^*$ is surely regular, the complement of a regular language is always regular.
4. **Reverse Operator:**
   Given language $L$, $L^R$ is the set of strings whose reversal is in L.
   Example: L = {0, 01, 100};
   $L^R$ ={0, 10, 001}.
   **Proof:** Let E be a regular expression for L. We show how to reverse E, to provide a

   regular expression for $E^R$ for $L^R$

5. **Complement:**
   The complement of a language L (with respect to an alphabet $E$ such that $E^*$ contains L) is $E^*$–L. Since $E^*$is surely regular, the complement of a regular language is always regular.
6. **Union:**
   Let L and M be the languages of regular expressions R and S, respectively.Then R+S is a regular expression whose language is(L U M).
7. I**ntersection:**
   Let L and M be the languages of regular expressions R and S, respectively then it a regular expression whose language is L intersection M.
   **proof:** Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs consisting of final states of both A and B.
8. **Set Difference operator:**
   If L and M are regular languages, then so is L -M = strings in L but not M.
   Proof: Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs, where A-state is final but B-state is not.
9. **Homomorphism:**

A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet. Example: h(0) = ab; h(1) = E. Extend to strings by h(a1⋯an) =h(a1)⋯h(an).

Example: h(01010) = abababab.

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

10. **Inverse Homomorphism :** Let h be a homomorphism and L a language whose alphabet is the output language of h. $h^{-1}(L)$ = {w | h(w) is in L}.

**Note:** There are few more properties like symmetric difference operator, prefix operator, substitution which are closed under closure properties of regular language.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/closure-properties-of-regular-languages/

https://www.youtube.com/watch?v=9pDdSxxJJ-k

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second. T1(151-160)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-23**

## LECTURE HANDOUTS

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Closure Properties of Regular Languages.(Example)

**Introduction : ( Maximum 5 sentences)** :

- **Closure properties** on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language.
- Closure refers to some operation on a language, resulting in a new language that is of same "type" as originally operated on i.e., regular.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

Regular Expression

Regular Sets

**Detailed content of the Lecture:**

**Closure properties** on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language. Closure refers to some operation on a language, resulting in a new language that is of same "type" as originally operated on i.e., regular.
Regular languages are closed under following operations.
**Consider L and M are regular languages:**

5. **Kleen Closure:**
   RS is a regular expression whose language is L, M. R* is a regular expression whose language is L*.
6. **Positive closure:**
   RS is a regular expression whose language is L, M. $R^+$ is a regular expression whose language is $L^+$.

7. **Complement:**
The complement of a language L (with respect to an alphabet $E$ such that $E^*$ contains L) is $E^*$–L. Since $E^*$ is surely regular, the complement of a regular language is always regular.

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

5. **Complement:**
The complement of a language L (with respect to an alphabet $E$ such that $E^*$ contains L) is $E^*$–L. Since $E^*$is surely regular, the complement of a regular language is always regular.

6. **Union:**
Let L and M be the languages of regular expressions R and S, respectively.Then R+S is a regular expression whose language is(L U M).

7. **Intersection:**
Let L and M be the languages of regular expressions R and S, respectively then it a regular expression whose language is L intersection M.
**proof:** Let A and B be DFA's whose languages are L and M, respectively. Construct C, the product automaton of A and B make the final states of C be the pairs consisting of final states of both A and B.

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

9. **Homomorphism:**
A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet. Example: h(0) = ab; h(1) = $E$. Extend to strings by h(a1⋯an) =h(a1)⋯h(an).
Example: h(01010) = abababb.

If L is a regular language, and h is a homomorphism on its alphabet, then h(L)= {h(w) | w is in L} is also a regular language.

**Proof:** Let E be a regular expression for L. Apply h to each symbol in E. Language of resulting R, E is h(L).

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

**Inverse Homomorphism**

**Example :**Let Σ = {a, b}, and Δ = {0, 1}. Let L = (00 ∪ 1)∗ and h(a) = 01 and h(b) = 10.

• h −1 (1001) = {ba}, h −1 (010110) = {aab}

• h −1 (L) = (ba) ∗

• What is h(h −1 (L))? (1001)∗ ( L Note: In general h(h −1 (L)) ⊆ L ⊆ h −1 (h(L)), but neither containment is necessarily an equal

**Note:** There are few more properties like symmetric difference operator, prefix operator, substitution which are closed under closure properties of regular language.

**Question :** Consider the language L1,L2,L3 as given below.
L1 = { $a^m b^n$ | m, n >= 0 }
L2 = { $a^n b^n$ | n >= 0 }
L3 = { $a^n b^n c^n$ | n >= 0 }
Which of the following statements is NOT TRUE?
A. Push Down Automata (PDA) can be used to recognize L1 and L2
B. L1 is a regular language
C. All the three languages are context free
D. Turing machine can be used to recognize all the three languages
**Solution :** Option (A) says PDA can be used to recognize L1 and L2. L1 contains all strings with any no. of a followed by any no. of b. So, it can be accepted by PDA. L2 contains strings with n no. of a's followed by n no. of b's. It can also be accepted by PDA. So, option (A) is correct.
Option (B) says that L1 is regular. It is true as regular expression for L1 is a*b*.
Option (C) says L1, L2 and L3 are context free. L3 languages contains all strings with n no. of a's followed by n no. of b's followed by n no. of c's. But it can't be accepted by PDA. So option ( C) is not correct.
Option (D) is correct as Turing machine can be used to recognize all the three languages.

**Question :** The language L = { $0^i 1 2^i$ | i ≥ 0 } over the alphabet {0, 1, 2} is :
A. Not recursive
B. Is recursive and deterministic CFL
C. Is regular
D. Is CFL bot not deterministic CFL.

**Solution :** The above language is deterministic CFL as for 0's, we can push 0 on stack and for 2's we can pop corresponding 0's. As there is no ambiguity which moves to take, it is deterministic. So, correct option is (B). As CFL is subset of recursive, it is recursive as well.

**Question :** Consider the following languages:
L1 = { $0^n1^n$| n≥0 }
L2 = { wcwr | w ε {a,b}* }
L3 = { wwr | w ε {a,b}* }
Which of these languages are deterministic context-free languages?
A. None of the languages
B. Only L1
C. Only L1 and L2
D. All three languages
**Solution :** Languages L1 contains all strings in which n 0's are followed by n 1's. Deterministic PDA can be constructed to accept L1. For 0's we can push it on stack and for 1's, we can pop from stack. Hence, it is DCFL.
L2 contains all strings of form wcwr where w is a string of a's and b's and wr is reverse of w. For example, aabbcbbaa. To accept this language, we can construct PDA which will push all symbols on stack before c. After c, if symbol on input string matches with symbol on stack, it is popped. So, L2 can also be accepted with deterministic PDA, hence it is also DCFL.
L3 contains all strings of form wwr where w is a string of a's and b's and wr is reverse of w. But we don't know where w ends and wr starts. e.g.; aabbaa is a string corresponding to L3. For first a, we will push it on stack. Next a can be either part of w or wr where w=a. So, there can be multiple moves from a state on an input symbol. So, only non-deterministic PDA can be used to accept this type of language. Hence, it is NCFL not DCFL.
So, correct option is (C). Only, L1 and L2 are DCFL.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/closure-properties-of-regular-languages/

https://www.youtube.com/watch?v=9pDdSxxJJ-k

https://www.geeksforgeeks.org/closure-properties-of-context-free-languages/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second. T1(151-160)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

**(An Autonomous Institution)**

**(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)**

**Rasipuram - 637 408, Namakkal Dist., Tamil Nadu**

**L-24**

**LECTURE HANDOUTS**

**CSE**

**II/IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **II Regular Languages.** **Date of Lecture:**

**Topic of Lecture:** Application of Finite Automata

**Introduction : ( Maximum 5 sentences)** :

- Automata is a machine that can accept the Strings of a *Language L* over an *input alphabet* . So far we are familiar with the Types of Automata .
- FA is less powerful than any other machine.
- For the designing of lexical analysis of a compiler.
- For recognizing the pattern using regular expressions

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

1. Finite Automata
2. NFA

**Detailed content of the Lecture:**

## Applications of various Automata

Automata is a machine that can accept the Strings of a *Language L* over an *input alphabet* .
So far we are familiar with the Types of Automata . Now, let us discuss the expressive power of Automata and further understand its Applications.

**Expressive Power of various Automata:**
The Expressive Power of any machine can be determined from the class or set of Languages accepted by that particular type of Machine. Here is the increasing sequence of expressive power of machines :

As we can observe that FA is less powerful than any other machine. It is important to note that DFA

and NFA are of same power because every NFA can be converted into DFA and every DFA can be converted into NFA .
The Turing Machine i.e. TM is more powerful than any other machine.

(i) Finite Automata (FA) equivalence:

```
Finite Automata
≡ PDA with finite Stack
≡ TM with finite tape
≡ TM with unidirectional tape
≡ TM with read only tape
```

(ii) Pushdown Automata (PDA) equivalence:

```
PDA ≡ Finite Automata with Stack
```

(iii) Turing Machine (TM) equivalence:

```
Turing Machine
≡ PDA with additional Stack
≡ FA with 2 Stacks
```

The **Applications** of these Automata are given as follows:

**1. Finite Automata (FA) –**

- For the designing of lexical analysis of a compiler.
- For recognizing the pattern using regular expressions.
- For the designing of the combination and sequential circuits using Mealy and Moore Machines.
- Used in text editors.
- For the implementation of spell checkers.

**2. Push Down Automata (PDA) –**

- For designing the parsing phase of a compiler (Syntax Analysis).
- For implementation of stack applications.
- For evaluating the arithmetic expressions.
- For solving the Tower of Hanoi Problem.

**3. Linear Bounded Automata (LBA) –**

- For implementation of genetic programming.
- For constructing syntactic parse trees for semantic analysis of the compiler.

**4. Turing Machine (TM) –**

- For solving any recursively enumerable problem.
- For understanding complexity theory.
- For implementation of neural networks.
- For implementation of Robotics Applications.

| |
|---|
| • For implementation of artificial intelligence. |
| **Video Content / Details of website for further learning (if any):**<br><br>https://www.geeksforgeeks.org/applications-of-various-automata/ |
| **Important Books/Journals for further learning including the page nos.:** |

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-31**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Introduction– Types of Grammar

**Introduction : ( Maximum 5 sentences)** :

- The theory of formal languages finds its applicability extensively in the fields of Computer Science.
- **Noam Chomsky** gave a mathematical model of grammar in 1956 which is effective for writing computer languages.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Regular Languages
- Regular Expression
- Finite Automaton

**Detailed content of the Lecture:**

Noam Chomsky gave a **mathematical model of grammar**. This model is used to write computer languages effectively.

**A grammar can be represented as a 4 tuple:**

**(N, T, P, S)**

- **N** denotes the set of variables or non-terminal symbols.
- **T** denotes the set of terminal symbols.
- **S** is a special variable called start symbol and **S** belongs to **N**.
- **P** is production rules for terminals and non-terminals **α → β**, where **α** and **β** are strings on **VN ∪ ∑** and least one symbol of **α** belongs to **VN**.

**Example:**

**Grammar G1 − ({S, C, D}, {c, d}, S, {S → CD, C → c, D → d})**

Here,

- **S**, **C**, and **D** Are Non-terminal symbols;
- **c** and **d** are Terminal symbols

- **S** is the Start symbol, **S** belongs to **N**
- Productions, **P**: **S → CD, C → c, D → d**

**Example:**

**Grammar G2 − (({S, C}, {c,d}, S,{S → cCd, cC → ccCd, C → ε } )**

Here,

- **S** and **C** Are Non-terminal symbols.
- **c** and **d** are Terminal symbols.
- **ε** is an empty string.
- **S** is the Start symbol, **S** belongs to **N**
- Production **P** : **S → cCd, cC → ccCd, C → ε**

**Derivations**

If a grammar **G** has a production **α → β**, it denotes that **x α y** derives **x β y** in **G**. This derivation can be represented as: **x α y ⇒ G x β y**

**Example:**

Let **grammar be − G = ({S, C}, {c, b}, S, {S →cCd, cC → ccCd, C → ε } )**

Derived strings:

- **S ⇒ cCd** using production **S →cCd**
- **⇒ccCdd** using production **cC →cCd**
- **⇒cccCddd** using production **cC→ cCd**
- **⇒cccddd** using production **C→ ε**

**Language**

It is The set of all strings a grammar can derive. It is said to be generated by that grammar.

**L(G)= {X|X∈ ∑*, S ⇒g X}**

**If L(G1) = L(G2), the Grammar G1 and Grammar G2 are equivalent.**

**Example:**

Let grammar be: **G: N = {S, C, D} T = {c, d} P = {S → CD, C → c, D → d}**

Here, **S** produces **CD**, and we can replace **C** by **c**, and **C** by **d**. Here, the only accepted string is **cd**, i.e., **L(G) = {cd}**

**Example:**

Suppose we have the following grammar: **G: N = {S, C, D} T = {c,d} P = {S → CD, C → cC|c, D → dD|d}**

The language generated by this grammar

**L(G) = {cd, c2d, cd2, c2d2, ...}**

**= {cm dn | m ≥ 1 and n ≥ 1}**

Now, We'll consider some given languages and then convert it into a grammar **G** which is responsible for production of those languages.

Construction of a Grammar Generating a Language

We'll consider some languages and convert it into a grammar G which produces those languages.

**Example**

**Problem** − Suppose, L (G) = $\{a^m b^n \mid m > 0 \text{ and } n \geq 0\}$. We have to find out the grammar G which produces L(G).

**Solution** −

Since L(G) = $\{a^m b^n \mid m > 0 \text{ and } n \geq 0\}$, the set of strings accepted can be rewritten as −

L(G) = {a, aa, ab, aaa, aab ,abb, …….}

Here, the start symbol has to take at least one ‘a’ followed by any number of ‘b’ including null.

To accept the string set {a, aa, ab, aaa, aab, abb, …….}, we have taken the productions −

$S \rightarrow aA$, $A \rightarrow aA$ , $A \rightarrow B$, $B \rightarrow bB$ ,$B \rightarrow \lambda$

$S \rightarrow aA \rightarrow aB \rightarrow a\lambda \rightarrow a$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aa\lambda \rightarrow aa$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow ab\lambda \rightarrow ab$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaaA \rightarrow aaaB \rightarrow aaa\lambda \rightarrow aaa$ (Accepted)

$S \rightarrow aA \rightarrow aaA \rightarrow aaB \rightarrow aabB \rightarrow aab\lambda \rightarrow aab$ (Accepted)

$S \rightarrow aA \rightarrow aB \rightarrow abB \rightarrow abbB \rightarrow abb\lambda \rightarrow abb$ (Accepted)

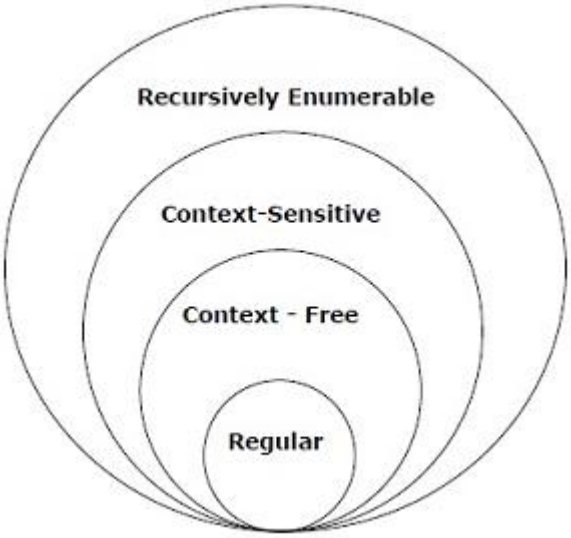
Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar −

G: ({S, A, B}, {a, b}, S, {$S \rightarrow aA$, $A \rightarrow aA \mid B$, $B \rightarrow \lambda \mid bB$ })

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

| Grammar Type | Grammar Accepted | Language Accepted | Automaton |
|---|---|---|---|
| Type 0 | Unrestricted grammar | Recursively enumerable language | Turing Machine |
| Type 1 | Context-sensitive grammar | Context-sensitive language | Linear-bounded automaton |
| Type 2 | Context-free grammar | Context-free language | Pushdown automaton |
| Type 3 | Regular grammar | Regular language | Finite state automaton |




It shows the scope of each type of grammar

**Type - 3 Grammar**

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example
X → ε
X → a | aY
Y → b

**Type - 2 Grammar**

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example
S → X a
X → a
X → aX
X → abc
X → ε

**Type - 1 Grammar**

Type-1 grammars generate context-sensitive languages. The productions must be in the form

**$\alpha A \beta \rightarrow \alpha \gamma \beta$**

where **$A \in N$** (Non-terminal)

and **$\alpha, \beta, \gamma \in (T \cup N)^*$** (Strings of terminals and non-terminals)

The strings **$\alpha$** and **$\beta$** may be empty, but **$\gamma$** must be non-empty.

The rule **$S \rightarrow \varepsilon$** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example
AB → AbBc
A → bcA
B → b

**Type - 0 Grammar**

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **$\alpha \rightarrow \beta$** where **$\alpha$** is a string of terminals and nonterminals with at least one non-terminal and **$\alpha$** cannot be null. **$\beta$** is a string of terminals and non-terminals.

Example
S → ACaB
Bc → acB
CB → DB
aD → Db

**Video Content / Details of website for further learning (if any):**

https://www.includehelp.com/toc/introduction-to-grammars-in-theory-of-computation.aspx

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(189-190)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-32**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Context Free Grammars and Languages

**Introduction : ( Maximum 5 sentences)** :

- A **context-free grammar** is a set of recursive rules used to generate patterns of strings.
- A **context-free grammar** can describe all regular **languages** and more, but they cannot describe all possible **languages**.
- **Context-free grammars** are studied in fields of theoretical computer science, compiler design, and linguistics.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Mathematical model of grammar(Grammar Type)**
- **Languages**
- **Automaton**

**Detailed content of the Lecture:**

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

- **G** describes the grammar
- **T** describes a finite set of terminal symbols.
- **V** describes a finite set of non-terminal symbols
- **P** describes a set of production rules
- **S** is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

**Example**

$L= \{wcw^R \mid w \in (a, b)^*\}$

**Production rules:**

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow c$

Now check that abbcbba string can be derived from the given CFG.

$S \Rightarrow aSa$
$S \Rightarrow abSba$
$S \Rightarrow abbSbba$
$S \Rightarrow abbcbba$

By applying the production $S \rightarrow aSa$, $S \rightarrow bSb$ recursively and finally applying the production $S \rightarrow c$, we get the string abbcbba.

# Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficientparser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

**Closure Properties of Context Free Languages**

Context Free languages are accepted by pushdown automata but not by finite automata. Context free languages can be generated by context free grammar which has the form :

A -> ρ (where A ∈ N and ρ ∈ (T ∪ N)* and N is a non-terminal and T is a terminal)

**Properties of Context Free Languages**

**Union :** If L1 and If L2 are two context free languages, their union L1 ∪ L2 will also be context free. For example,

$L1 = \{ a^n b^n c^m \mid m >= 0 \text{ and } n >= 0 \}$ and $L2 = \{ a^n b^m c^m \mid n >= 0 \text{ and } m >= 0 \}$

$L3 = L1 \cup L2 = \{ a^n b^n c^m \cup a^n b^m c^m \mid n >= 0, m >= 0 \}$ is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.

**Note:** So CFL are closed under Union.

**Concatenation :** If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free. For example,

$L1 = \{ a^n b^n \mid n >= 0 \}$ and $L2 = \{ c^m d^m \mid m >= 0 \}$

$L3 = L1.L2 = \{ a^n b^n c^m d^m \mid m >= 0 \text{ and } n >= 0 \}$ is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's. So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.

**Note:** So CFL are closed under Concatenation.

**Kleene Closure :** If L1 is context free, its Kleene closure L1* will also be context free. For example,

$L1 = \{ a^n b^n \mid n >= 0 \}$

$L1^* = \{ a^n b^n \mid n >= 0 \}^*$ is also context free.

**Note :**So CFL are closed under Kleen Closure.

**Intersection and complementation :** If L1 and If L2 are two context free languages, their intersection

L1 ∩ L2 need not be context free. For example,
L1 = { $a^nb^nc^m$ | n >= 0 and m >= 0 } and L2 = ($a^mb^nc^n$ | n >= 0 and m >= 0 }
L3 = L1 ∩ L2 = { $a^nb^nc^n$ | n >= 0 } need not be context free.
L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.
Similarly, complementation of context free language L1 which is $\sum^*$ – L1, need not be context free.
**Note :** So CFL are not closed under Intersection and Complementation.

**Deterministic Context-free Languages**
Deterministic CFL are subset of CFL which can be recognized by Deterministic PDA. Deterministic PDA has only one move from a given state and input symbol, i.e., it do not have choice. For a language to be DCFL it should be clear when to PUSh or POP.
For example, L1= { $a^nb^nc^m$ | m >= 0 and n >= 0} is a DCFL because for a's, we can push on stack and for b's we can pop. It can be recognized by Deterministic PDA. On the other hand, L3 = { $a^nb^nc^m$ ∪ $a^nb^mc^m$ | n >= 0, m >= 0 } cannot be recognized by DPDA because either number of a's and b's can be equal or either number of b's and c's can be equal. So, it can only be implemented by NPDA. Thus, it is CFL but not DCFL.
**Note :** DCFL are closed only under complementation and Inverse Homomorphism.
**Question :** Consider the language L1,L2,L3 as given below.
L1 = { $a^mb^n$ | m, n >= 0 }
L2 = { $a^nb^n$ | n >= 0 }
L3 = { $a^nb^nc^n$ | n >= 0 }
Which of the following statements is NOT TRUE?
A. Push Down Automata (PDA) can be used to recognize L1 and L2
B. L1 is a regular language
C. All the three languages are context free
D. Turing machine can be used to recognize all the three languages
**Solution :** Option (A) says PDA can be used to recognize L1 and L2. L1 contains all strings with any no. of a followed by any no. of b. So, it can be accepted by PDA. L2 contains strings with n no. of a's followed by n no. of b's. It can also be accepted by PDA. So, option (A) is correct.
Option (B) says that L1 is regular. It is true as regular expression for L1 is a*b*.
Option (C) says L1, L2 and L3 are context free. L3 languages contains all strings with n no. of a's followed by n no. of b's followed by n no. of c's. But it can't be accepted by PDA. So option ( C) is not correct.
Option (D) is correct as Turing machine can be used to recognize all the three languages.
**Question :** Consider the following languages:
L1 = { $0^n1^n$| n≥0 }
L2 = { wcwr | w ε {a,b}* }
L3 = { wwr | w ε {a,b}* }
Which of these languages are deterministic context-free languages?
A. None of the languages
B. Only L1
C. Only L1 and L2
D. All three languages
**Solution :** Languages L1 contains all strings in which n 0's are followed by n 1's. Deterministic PDA can be constructed to accept L1. For 0's we can push it on stack and for 1's, we can pop from stack. Hence, it is DCFL.
L2 contains all strings of form wcwr where w is a string of a's and b's and wr is reverse of w. For example, aabbcbbaa. To accept this language, we can construct PDA which will push all symbols on

stack before c. After c, if symbol on input string matches with symbol on stack, it is popped. So, L2 can also be accepted with deterministic PDA, hence it is also DCFL.
L3 contains all strings of form wwr where w is a string of a's and b's and wr is reverse of w. But we don't know where w ends and wr starts. e.g.; aabbaa is a string corresponding to L3. For first a, we will push it on stack. Next a can be either part of w or wr where w=a. So, there can be multiple moves from a state on an input symbol. So, only non-deterministic PDA can be used to accept this type of language. Hence, it is NCFL not DCFL.
So, correct option is (C). Only, L1 and L2 are DCFL.

**Video Content / Details of website for further learning (if any):**

https://www.javatpoint.com/capabilities-of-cfg

https://www.javatpoint.com/context-free-grammar

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second .T1(190-193)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-33**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Context Free Grammars and Languages –( **Derivations and Languages – Ambiguity)**

**Introduction : ( Maximum 5 sentences)** :

- A **context-free grammar** is a set of recursive rules used to generate patterns of strings.
- A **context-free grammar** can describe all regular **languages** and more, but they cannot describe all possible **languages**.
- **Context-free grammars** are studied in fields of theoretical computer science, compiler design, and linguistics.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Mathematical model of grammar(Grammar Type)**
- **Languages**
- **Automaton**
- **Context Free Grammars**

**Detailed content of the Lecture:**

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

- **G** describes the grammar
- **T** describes a finite set of terminal symbols.
- **V** describes a finite set of non-terminal symbols
- **P** describes a set of production rules
- **S** is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by

terminal symbols.

# Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.
- We have two options to decide which non-terminal to be replaced with production rule.

## Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

**Production rules:**

S = S + S
S = S - S
S = a | b |c

Input:

a - b + c

## The left-most derivation is:

S = S + S
S = S - S + S
S = a - S + S
S = a - b + S
S = a - b + c

## Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

S = S + S
S = S - S
S = a | b |c

Input:

a - b + c

**The right-most derivation is:**

S = S - S
S = S - S + S
S = S - S + c

S = S - b + c
S = a - b + c

## Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

## The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

**Example:**

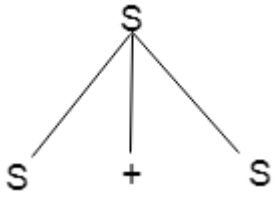
**Production rules:**

T= T + T | T * T
T = a|b|c

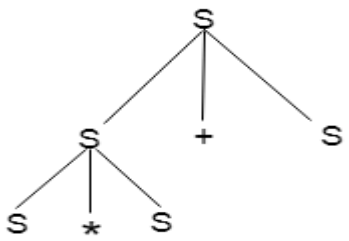
**Input:**

a * b + c

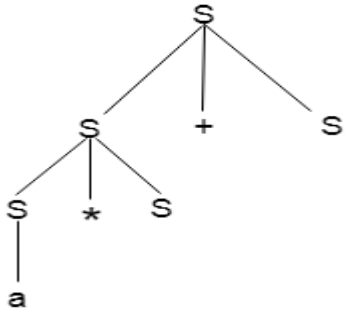
**Step 1:**



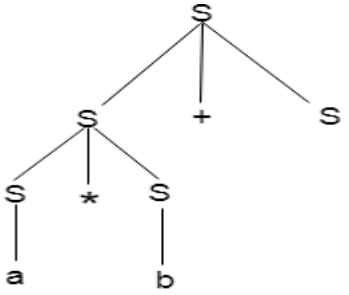

**Step 2:**



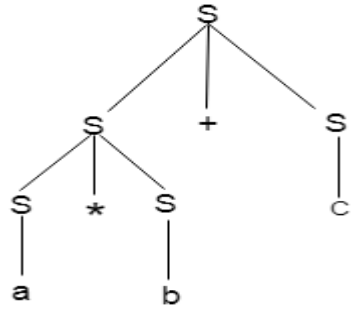
**Step 3:**




**Step 4:**




**Step 5:**




## Ambiguity

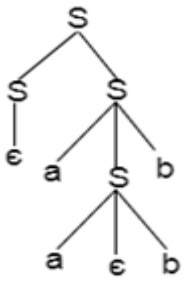
A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

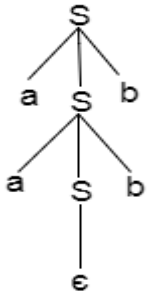
**Example:**

S = aSb | SS

S = ∈

For the string aabb, the above grammar generates two parse trees:




If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

**Parser**

- Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.
- A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.
- Parsing is of two types: top down parsing and bottom up parsing.

**Video Content / Details of website for further learning (if any):**

https://www.javatpoint.com/derivation

https://www.javatpoint.com/parse-tree

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(193-197)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-34**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Context Free Grammars and Languages - **(Relationship between derivation and derivation trees)**

**Introduction : ( Maximum 5 sentences)** :

- A **context-free grammar** is a set of recursive rules used to generate patterns of strings.
- A **context-free grammar** can describe all regular **languages** and more, but they cannot describe all possible **languages**.
- **Context-free grammars** are studied in fields of theoretical computer science, compiler design, and linguistics.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Mathematical model of grammar(Grammar Type)**
- **Languages**
- **Automaton**
- **Context Free Grammars**

**Detailed content of the Lecture:**

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

- **G** describes the grammar
- **T** describes a finite set of terminal symbols.
- **V** describes a finite set of non-terminal symbols
- **P** describes a set of production rules
- **S** is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by

terminal symbols.

# Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.
- We have two options to decide which non-terminal to be replaced with production rule.

## Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

**Production rules:**

S = S + S
S = S - S
S = a | b |c

Input:

a - b + c

## The left-most derivation is:

S = S + S
S = S - S + S
S = a - S + S
S = a - b + S
S = a - b + c

## Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

S = S + S
S = S - S
S = a | b |c

Input:

a - b + c

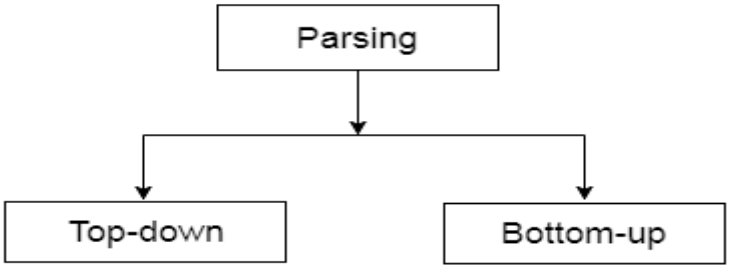
**The right-most derivation is:**

S = S - S
S = S - S + S
S = S - S + c

S = S - b + c
S = a - b + c

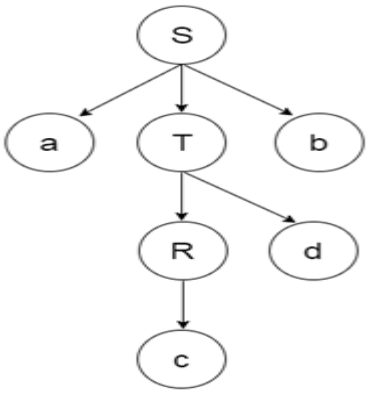
**Parser**

- Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.
- A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.
- Parsing is of two types: top down parsing and bottom up parsing.




# Top down paring

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.
- Parse Tree representation of input string "acdb" is as follows:

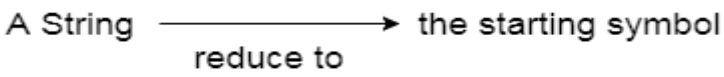



**Bottom up parsing**

- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

## Shift reduce parsing

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.



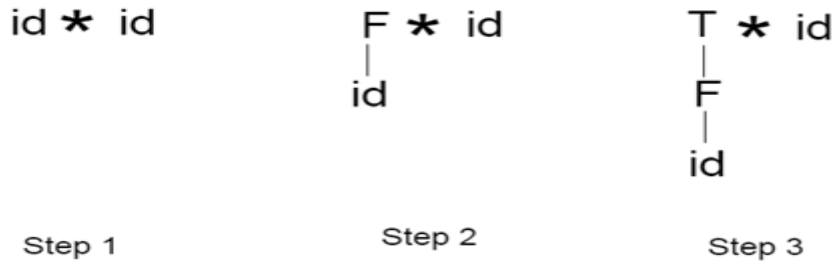

- Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

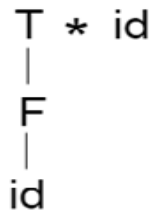
## Example

**Production**

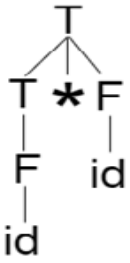
E → T
T → T * F
T → id
F → T
F → id

Parse Tree representation of input string "id * id" is as follows:

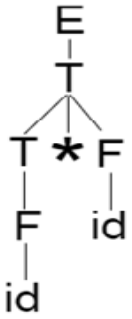





Step 4




Step 5




Step 6

Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing

a. LR( 1 )
b. SLR( 1 )
c. CLR ( 1 )
d. LALR( 1 )

## There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".
a ⋖ b means that terminal "a" has the lower precedence than terminal "b".
a ≐ b means that the terminal "a" and "b" both have same precedence.

**Video Content / Details of website for further learning (if any):**

https://www.javatpoint.com/derivation

https://www.javatpoint.com/parse-tree

https://www.javatpoint.com/parser

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(193-197)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**L-35**

## LECTURE HANDOUTS

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Context Free Grammars and Languages- **(Simplification of CFG Elimination of Useless symbols)**

**Introduction : ( Maximum 5 sentences)** :

- As we have seen, various languages can efficiently be represented by a context-free grammar.
- All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal).

**Prerequisite knowledge for Complete understanding and learning of Topic:**

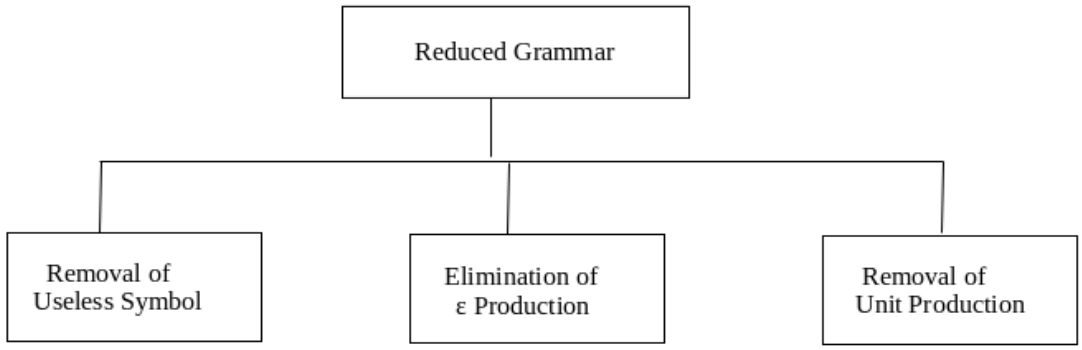
**( Max. Four important topics)**

- **Context Free Grammars**

**Detailed content of the Lecture:**

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as $X \rightarrow Y$ where X and Y are non-terminal.
3. If $\varepsilon$ is not in the language L then there need not to be the production $X \rightarrow \varepsilon$.




**Removal of Useless Symbols**

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

$T \rightarrow aaB \mid abA \mid aaT$
$A \rightarrow aA$
$B \rightarrow ab \mid b$
$C \rightarrow ad$

In the above example, the variable 'C' will never occur in the derivation of any string, so the production $C \rightarrow ad$ is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production $A \rightarrow aA$ is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production $A \rightarrow aA$, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

**Elimination of ε Production**

The productions of type $S \rightarrow \varepsilon$ are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

**Step 1:** First find out all nullable non-terminal variable which derives ε.

**Step 2:** For each production $A \rightarrow a$, construct all production $A \rightarrow x$, where x is obtained from a by removing one or more non-terminal from step 1.

**Step 3:** Now combine the result of step 2 with the original production and remove ε productions.

**Example:**

Remove the production from the following CFG by preserving the meaning of it.

$S \rightarrow XYX$
$X \rightarrow 0X \mid \varepsilon$
$Y \rightarrow 1Y \mid \varepsilon$

**Solution:**

Now, while removing ε production, we are deleting the rule $X \rightarrow \varepsilon$ and $Y \rightarrow \varepsilon$. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

$S \rightarrow XYX$

If the first X at right-hand side is ε. Then

$S \rightarrow YX$

Similarly if the last X in R.H.S. = ε. Then

$S \rightarrow XY$

If $Y = \varepsilon$ then

$S \rightarrow XX$

If Y and X are ε then,

$S \rightarrow X$

If both X are replaced by ε

$S \rightarrow Y$

Now,

$S \rightarrow XY \mid YX \mid XX \mid X \mid Y$

Now let us consider

$X \rightarrow 0X$

If we place ε at right-hand side for X then,

$X \rightarrow 0$
$X \rightarrow 0X \mid 0$

Similarly $Y \rightarrow 1Y \mid 1$

Collectively we can rewrite the CFG with removed ε production as

$S \rightarrow XY \mid YX \mid XX \mid X \mid Y$
$X \rightarrow 0X \mid 0$
$Y \rightarrow 1Y \mid 1$

**Video Content / Details of website for further learning (if any):**

https://www.javatpoint.com/automata-simplification-of-cfg

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(279-280)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-36**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Context Free Grammars and Languages **(Unit productions - Null productions)**

**Introduction : ( Maximum 5 sentences)** :

- As we have seen, various languages can efficiently be represented by a context-free grammar.
- All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal).
- The productions of type 'A -> B' are called unit productions
- To create a unit production free grammar 'Guf' from the original grammar 'G' , we follow the procedure mentioned below.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Context Free Grammars**

**Detailed content of the Lecture:**

**Removing Unit Productions**

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

**Step 1:** To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

**Step 2:** Now delete X → Y from the grammar.

**Step 3:** Repeat step 1 and step 2 until all unit productions are removed.
For example:

S → 0A | 1B | C
A → 0S | 00

B → 1 | A

C → 01

**Solution:**

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

S → 0A | 1B | 01

A → 0S | 00

B → 1 | 0S | 00

C → 01

**Null production**

Null productions are of the form A -> ϵ. ... We cannot remove all ϵ-productions from a grammar if the language contains ϵ as a word, but if it doesn't we can remove all. In a given CFG, we call a non-terminal N nullable if there is a production N -> ϵ or there is a derivation that starts at N and leads to ϵ: N => … => ϵ.

**Elimination of null production from context free grammar**

If ε belongs to the language then we are supposed to generate it and thus we will not remove it.
Using below example we will understand the whole concept.

**Example 1**

S -> aSb/aAb/ab/a

A -> ε

How to know whether ε is generated in the CFG or not ?
Find all the Variable which are generating ε
So only **A** is genrating ε Thus ε does not belong to the langauge.

**Now we will proceed with elimination of NULL production:**

Replace NULL producing symbol with and without in R.H.S. of remaining states
And drop the productions which has ε directly. eg. A -> ε

S -> aSb/aAb/ab/ab/a But we no need to write "ab" twice

So,

**S -> aSb/aAb/ab/a**

Example 2

S -> AB

A -> aAA/ε

B -> bBB/ε

**Nullable** Variables are {A, B, S}
Because start state also a Nullable symbol so ε belongs to given CFG
We will proceed with the method:

**S -> AB/A/B/ε**
**A -> aAA/aA/a**
**B -> bAA/bA/b**

**Video Content / Details of website for further learning (if any):**

https://www.javatpoint.com/automata-simplification-of-cfg

https://scanftree.com/automata/elimination-of-null-production-from-context-free-grammar

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(279-280)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-37**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Greibach Normal form

**Introduction : ( Maximum 5 sentences)** :

- Greibach normal form (GNF) if the right-hand sides of all production rules start with a terminal symbol, optionally followed by some variables.
- A non-strict form allows one exception to this format restriction for allowing the empty word (epsilon, ε) to be a member of the described language.
- The normal form was established by Sheila Greibach and it bears her name.

**Prerequisite knowledge for Complete understanding and learning of Topic:(Max.Four important topics)**

- Context Free Grammars

**Detailed content of the Lecture:**

A context free grammar (CGF) is in Greibach Normal Form (GNF) if all production rules satisfy one of the following conditions:

- A non-terminal generating a terminal (e.g.; X->x)
- A non-terminal generating a terminal followed by any number of non-terminals (e.g.; X->xX1X2…Xn).Start symbol generating ε. (e.g.; S-> ε)

**Consider the following grammars:**

G1 = {S->aA|bB, B->bB|b, A->aA|a}

G2 = {S->aA|bB, B->bB|ε, A->aA|ε}

**How to convert CFG to GNF**

**Step 1.** Convert the grammar into CNF.

If the given grammar is not in CNF, convert it to CNF. You can refer following article to convert CFG to CNF: Converting Context Free Grammar to Chomsky Normal Form:

**Step 2.** Eliminate left recursion from grammar if it exists.

If CFG contains left recursion, eliminate them. You can refer following article to eliminate left recursion:

Parsing | Set 1 (Introduction, Ambiguity and Parsers)

**Step 3.** Convert the production rules into GNF form.
If any production rule is not in the GNF form, convert them. Let us take an example to convert CFG to GNF. Consider the given grammar G1:
S → XA|BB

B → b|SB

X → b

A → a

As G1 is already in CNF and there is not left recursion, we can skip step 1and 2 and directly move to step 3.
The production rule B->SB is not in GNF, therefore, we substitute S -> XA|BB in production rule B->SB as:
S → XA|BB

B → b|XAB|BBB

X → b

A → a

The production rules S->XA and B->XAB is not in GNF, therefore, we substitute X->b in production rules S->XA and B->XAB as:
S → bA|BB

B → b|bAB|BBB

X → b

A → a

Removing left recursion (B->BBB), we get:
S → bA|BB

B → bC|bABC

C → BBC| ε

X → b

A → a

Removing null production (C-> ε), we get:
S → bA|BB

B → bC|bABC|b|bAB

C → BBC|BB

X → b

A → a

The production rules S->BB is not in GNF, therefore, we substitute B → bC|bABC|b|bAB in production rules S->BB as:

S → bA| bCB|bABCB|bB|bABB

B → bC|bABC|b|bAB

C → BBC|BB

X → b

A → a

The production rules C->BB is not in GNF, therefore, we substitute B → bC|bABC|b|bAB in production rules C->BB as:
S → bA| bCB|bABCB|bB|bABB

B → bC|bABC|b|bAB

C → BBC

C → bCB|bABCB|bB|bABB

X → b

A → a

The production rules C->BBC is not in GNF, therefore, we substitute B → bC|bABC|b|bAB in production rules C->BBC as:
S → bA| bCB|bABCB|bB|bABB

B → bC|bABC|b|bAB

C → bCBC|bABCBC|bBC|bABBC

C → bCB|bABCB|bB|bABB

X → b

A → a This is the GNF form for the grammar G1.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/converting-context-free-grammar-greibach-normal-form/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(280-293)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-38**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Chomsky Normal Form

**Introduction : ( Maximum 5 sentences)** :

- In formal language theory, a context-free grammar G is said to be in Chomsky normal form if all of its production rules are of the form: A → BC, or A → a, or S → ε,
- where A, B, and C are nonterminal symbols, a is a terminal symbol, S is the start symbol, and ε denotes the empty string.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Context Free Grammars**
- **Greibach Normal form**

**Detailed content of the Lecture:**

A context free grammar (CFG) is in Chomsky Normal Form (CNF) if all production rules satisfy one of the following conditions:

- A non-terminal generating a terminal (e.g.; X->x)
- A non-terminal generating two non-terminals (e.g.; X->YZ)
- Start symbol generating ε. (e.g.; S-> ε)

Consider the following grammars,

G1 = {S->a, S->AZ, A->a, Z->z}

G2 = {S->a, S->aZ, Z->a}

The grammar G1 is in CNF as production rules satisfy the rules specified for CNF. However, the grammar G2 is not in CNF as the production rule S->aZ contains terminal followed by non-terminal which does not satisfy the rules specified for CNF.

**Note**

- For a given grammar, there can be more than one CNF.
- CNF produces the same language as generated by CFG.
- CNF is used as a preprocessing step for many algorithms for CFG like CYK(membership algo), bottom-up parsers etc.
- For generating string w of length 'n' requires '2n-1' production or steps in CNF.

- Any Context free Grammar that do not have ε in its language has an equivalent CNF.

**How to convert CFG to CNF?**

**Step 1.** Eliminate start symbol from RHS.
If start symbol S is at the RHS of any production in the grammar, create a new production as:
S0->S
where S0 is the new start symbol.

**Step 2.** Eliminate null, unit and useless productions.
If CFG contains null, unit or useless production rules, eliminate them. You can refer the this article to eliminate these types of production rules.

**Step 3.** Eliminate terminals from RHS if they exist with other terminals or non-terminals. e.g,; production rule X->xY can be decomposed as:
X->ZY
Z->x
**Step 4.** Eliminate RHS with more than two non-terminals.
e.g,; production rule X->XYZ can be decomposed as:
X->PZ
P->XY

**Example –** Let us take an example to convert CFG to CNF. Consider the given grammar G1:
S → ASB

A → aAS|a|ε

B → SbS|A|bb

**Step 1.** As start symbol S appears on the RHS, we will create a new production rule S0->S. Therefore, the grammar will become:
S0->S

S → ASB

A → aAS|a|ε

B → SbS|A|bb

**Step 2.** As grammar contains null production A-> ε, its removal from the grammar yields:
S0->S

S → ASB|SB

A → aAS|aS|a

B → SbS| A|ε|bb

Now, it creates null production B→ ε, its removal from the grammar yields:

S0->S

S → AS|ASB| SB| S

A → aAS|aS|a

B → SbS| A|bb

Now, it creates unit production B->A, its removal from the grammar yields:

S0->S

S → AS|ASB| SB| S

A → aAS|aS|a

B → SbS|bb|aAS|aS|a

Also, removal of unit production S0->S from grammar yields:

S0-> AS|ASB| SB| S

S → AS|ASB| SB| S

A → aAS|aS|a

B → SbS|bb|aAS|aS|a

Also, removal of unit production S->S and S0->S from grammar yields:

S0-> AS|ASB| SB

S → AS|ASB| SB

A → aAS|aS|a

B → SbS|bb|aAS|aS|a

**Step 3.** In production rule A->aAS |aS and B-> SbS|aAS|aS, terminals a and b exist on RHS with non-terminates. Removing them from RHS:

S0-> AS|ASB| SB

S → AS|ASB| SB

A → XAS|XS|a

B → SYS|bb|XAS|XS|a

X →a

Y→b

Also, B->bb can't be part of CNF, removing it from grammar yields:

S0-> AS|ASB| SB

S → AS|ASB| SB

A → XAS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

**Step 4:** In production rule S0->ASB, RHS has more than two symbols, removing it from grammar yields:

S0-> AS|PB| SB

S → AS|ASB| SB

A → XAS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Similarly, S->ASB has more than two symbols, removing it from grammar yields:

S0-> AS|PB| SB

S → AS|QB| SB

A → XAS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

Similarly, A->XAS has more than two symbols, removing it from grammar yields:

S0-> AS|PB| SB

S → AS|QB| SB

A → RS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

R → XA

Similarly, B->SYS has more than two symbols, removing it from grammar yields:

S0 -> AS|PB| SB

S → AS|QB| SB

A → RS|XS|a

B → TS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

R → XA

T → SY

Similarly, B->XAX has more than two symbols, removing it from grammar yields

S0-> AS|PB| SB

S → AS|QB| SB

A → RS|XS|a

B → TS|VV|US|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

R → XA

T → SY

U → XA

So this is the required CNF for given grammar.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/converting-context-free-grammar-chomsky-normal-form/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(280-293)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-39**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **III. Context Free Grammar** **Date of Lecture:**

**Topic of Lecture:** Problems related to CNF and GNF

**Introduction : ( Maximum 5 sentences)** :

- In formal language theory, a context-free grammar G is said to be in Chomsky normal form if all of its production rules are of the form: A → BC, or A → a, or S → ε,
- CNF produces the same language as generated by CFG.
- Greibach normal form (GNF) if the right-hand sides of all production rules start with a terminal symbol, optionally followed by some variables.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Context Free Grammars
- Greibach Normal form

**Detailed content of the Lecture:**

**Greibach Normal Form**

A CFG is in Greibach Normal Form if the Productions are in the following forms −

$A \rightarrow b$

$A \rightarrow bD_1 \ldots D_n$

$S \rightarrow \varepsilon$

where A, $D_1$,....,$D_n$ are non-terminals and b is a terminal.

**Algorithm to Convert a CFG into Greibach 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** − Remove all direct and indirect left-recursion.

**Step 5** − Do proper substitutions of productions to convert it into the proper form of GNF.

**Problem:**

**Convert the following CFG into CNF**

S → XY | Xn | p

X → mX | m

Y → Xn | o

**Solution:**

Here, **S** does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Now after replacing

X in S → XY | Xo | p

with

mX | m

we obtain

S → mXY | mY | mXo | mo | p.

And after replacing

X in Y → $X_n$ | o

with the right side of

X → mX | m

we obtain

Y → mXn | mn | o.

Two new productions O → o and P → p are added to the production set and then we came to the final GNF as the following −

S → mXY | mY | mXC | mC | p

X → mX | m

Y → mXD | mD | o

O → o

P → p

**Chomsky Normal Form**

A CFG is in Chomsky Normal Form if the Productions are in the following forms −

- A → a

- $A \rightarrow BC$
- $S \rightarrow \varepsilon$

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 $\mathbf{A \rightarrow B_1 \ldots B_n}$ where $\mathbf{n > 2}$ with $\mathbf{A \rightarrow B_1 C}$ where $\mathbf{C \rightarrow B_2 \ldots B_n}$. 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 $\mathbf{S_0 \rightarrow S}$ is added to the production set and it becomes −

$S_0 \rightarrow S$, 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 −

$S_0 \rightarrow S$, S→ ASA | aB | a, A → B | S | ∈, B → b

After removing A → ∈, the production set becomes −

$S_0 \rightarrow S$, 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 −

$S_0 \rightarrow S$, S→ ASA | aB | a | AS | SA, A → B | S, B → b

After removing $S_0 \rightarrow$ S, the production set becomes −

$S_0 \rightarrow$ ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → B | S, B → b

After removing A→ B, the production set becomes −

$S_0 \rightarrow$ ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

$A \rightarrow S \mid b$

$B \rightarrow b$

After removing $A \rightarrow S$, the production set becomes −

$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA$, $S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$

$A \rightarrow b \mid ASA \mid aB \mid a \mid AS \mid SA$, $B \rightarrow b$

**(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 −

$S_0 \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$S \rightarrow AX \mid YB \mid a \mid AS \mid SA$

$A \rightarrow b$ $A \rightarrow b \mid AX \mid YB \mid a \mid AS \mid SA$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/greibach_normal_form.htm

https://www.geeksforgeeks.org/converting-context-free-grammar-chomsky-normal-form/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(293-297)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-46**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Pushdown Automata- Definitions

**Introduction : ( Maximum 5 sentences)** :

- We have already discussed finite automata. But finite automata can be used to accept only regular languages.
- Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Context Free Grammars

**Detailed content of the Lecture:**

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is −
**"Finite state machine" + "a stack"**

## PDA Components:

**Input tape:** The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.

**Finite control:** The finite control has some pointer which points the current symbol which is to be read.

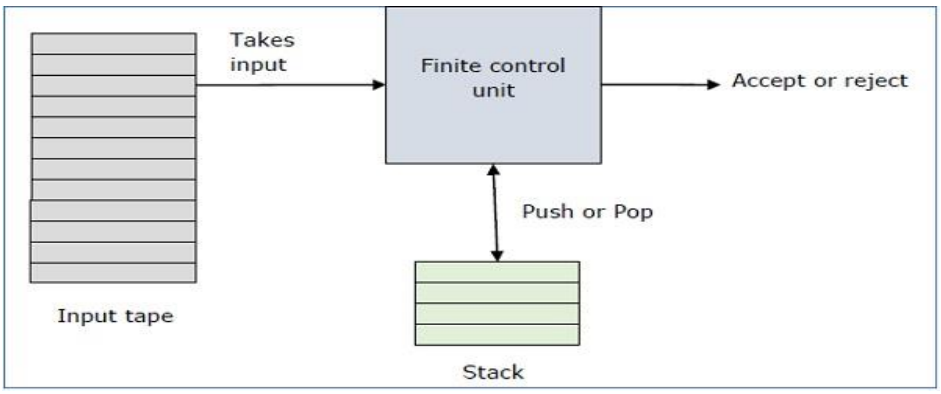
**Stack:** The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

The stack head scans the top symbol of the stack.

A stack does two operations −

- **Push** − a new symbol is added at the top.
- **Pop** − the top symbol is read and removed.

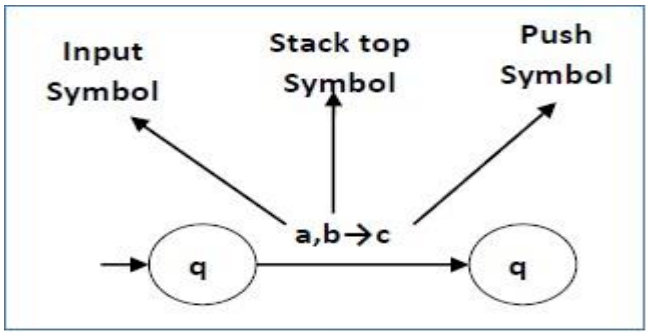
A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.




A PDA can be formally described as a 7-tuple (Q, ∑, S, δ, $q_0$, I, F) −

- **Q** is the finite number of states
- ∑ is input alphabet
- **S** is stack symbols
- **δ** is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*
- **$q_0$** is the initial state ($q_0$ ∈ Q)
- **I** is the initial stack top symbol (I ∈ S)
- **F** is a set of accepting states (F ∈ Q)

The following diagram shows a transition in a PDA from a state $q_1$ to state $q_2$, labeled as a,b → c




This means at state **$q_1$**, if we encounter an input string **'a'** and top symbol of the stack is **'b'**, then we pop **'b'**, push **'c'** on top of the stack and move to state **$q_2$**.

**Terminologies Related *to PDA***

**Instantaneous Description**

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- **q** is the state
- **w** is unconsumed input
- **s** is the stack contents

**Turnstile Notation**

⊢ sign describes the turnstile notation and represents one move.

⊢* sign describes a sequence of moves.

**For example,**
(p, b, T) ⊢ (q, w, α)
In the above example, while taking a transition from state p to q, the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α.

**Example 1:**
Design a PDA for accepting a language $\{a^n b^{2n} \mid n>=1\}$.

**Solution:** In this language, n number of a's should be followed by 2n number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon as we read 'b' then for every single 'b' only one 'a' should get popped from the stack.
The ID can be constructed as follows:

1. δ(q0, a, Z) = (q0, aaZ)
2. δ(q0, a, a) = (q0, aaa)

Now when we read b, we will change the state from q0 to q1 and start popping corresponding 'a'. Hence,

δ(q0, b, a) = (q1, ε)

Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state q1 only.

δ(q1, b, a) = (q1, ε)

After reading all b's, all the corresponding a's should get popped. Hence when we read ε as input symbol then there should be nothing in the stack. Hence the move will be:

δ(q1, ε, Z) = (q2, ε)

Where
PDA = ({q0, q1, q2}, {a, b}, {a, Z}, δ, q0, Z, {q2})
We can summarize the ID as:

δ(q0, a, Z) = (q0, aaZ)

δ(q0, a, a) = (q0, aaa)

δ(q0, b, a) = (q1, ε)

δ(q1, b, a) = (q1, ε)

δ(q1, ε, Z) = (q2, ε)

Now we will simulate this PDA for the input string "aaabbbbbb".

δ(q0, aaabbbbbb, Z) ⊢ δ(q0, aabbbbbb, aaZ)

⊢ δ(q0, abbbbbb, aaaaZ)

⊢ δ(q0, bbbbbb, aaaaaaZ)

⊢ δ(q1, bbbbb, aaaaaZ)

⊢ δ(q1, bbbb, aaaaZ)

⊢ δ(q1, bbb, aaaZ)

⊢ δ(q1, bb, aaZ)

⊢ δ(q1, b, aZ)

⊢ δ(q1, ε, Z)

⊢ δ(q2, ε) ACCEPT

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/pushdown_automata_introduction.htm

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(243-247)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

L-47

**LECTURE HANDOUTS**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Moves - Instantaneous descriptions

**Introduction : ( Maximum 5 sentences)** :

- We have already discussed finite automata. But finite automata can be used to accept only regular languages.
- Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Pushdown Automata

**Detailed content of the Lecture:**

**Instantaneous Description**

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- **q** is the state
- **w** is unconsumed input
- **s** is the stack contents

**Final State Acceptability**

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA $(Q, \sum, S, \delta, q_0, I, F)$, the language accepted by the set of final states F is −

$L(PDA) = \{w \mid (q_0, w, I) \vdash^* (q, \varepsilon, x), q \in F\}$

for any input stack string **x**.

**Empty Stack Acceptability**

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

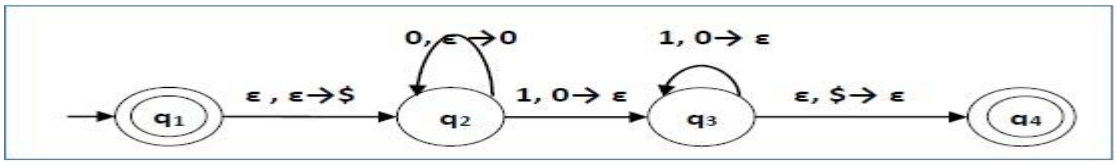
For a PDA $(Q, \sum, S, \delta, q_0, I, F)$, the language accepted by the empty stack is −

$L(PDA) = \{w \mid (q_0, w, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$

**Example**

Construct a PDA that accepts $\mathbf{L = \{0^n\ 1^n \mid n \geq 0\}}$

**Solution**




PDA for L= {$0^n\ 1^n$ | n≥0}

This language accepts L = {ε, 01, 0011, 000111, ..... }

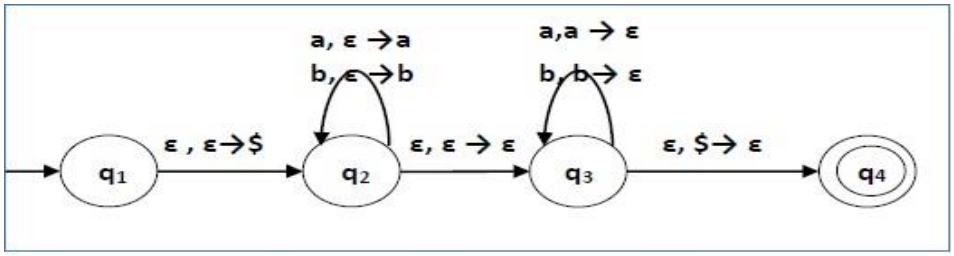
Here, in this example, the number of **'a'** and **'b'** have to be same.

- Initially we put a special symbol **'$'** into the empty stack.
- Then at state $\mathbf{q_2}$, if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0.
- Then at state $\mathbf{q_3}$, if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element.
- If the special symbol '$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state $q_4$.

**Example**

Construct a PDA that accepts L = { $ww^R$ | w = (a+b)* }

**Solution**




PDA for L= {$ww^R$ | w = (a+b)*}

Initially we put a special symbol '$' into the empty stack. At state $\mathbf{q_2}$, the **w** is being read. In state $\mathbf{q_3}$, each 0 or 1 is popped when it matches the input. If any other input is given, the PDA will go to a dead state. When we reach that special symbol '$', we go to the accepting state $\mathbf{q_4}$.

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/pda_context_free_grammar.htm

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(247-251)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-48**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Deterministic pushdown automata

**Introduction : ( Maximum 5 sentences)** :

- In automata theory, a deterministic pushdown automaton 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.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Pushdown Automata
- Instantaneous Description

**Detailed content of the Lecture:**

**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.

Pushdown Automata. Recall from the lecture that a pushdown automaton(PDA)is syntactically a tuple $A=\langle Q,\Sigma,\Gamma,\delta,q0,z0,F\rangle$ where$Q$is a finite set of states, $\Sigma$ and $\Gamma$ are two finite alphabets of input and stack symbols, $\delta\subseteq Q\times\Gamma\times(\Sigma\cup\{\varepsilon\})\times Q\times\Gamma^*$is a finite set of transitions,$q0\in Q$is the initial state,$z0\in\Gamma$ the initial stack content, and $F\subseteq Q$ is the set of accepting states.

A configuration of A is a pair $q, \gamma\in Q\times\Gamma^*$ consisting of the current state q and the current stack contents $\gamma$; we

assume that the top of the stack is to the right. A transition $(q,z,a,q',\gamma')$ can be applied in q, $\gamma$ if $\gamma=\gamma'' z$ and leads to the configuration $q',\gamma''\gamma'$, which we write $q,\gamma'' z\ a\longrightarrow_A q',\gamma''\gamma'$. We lift this definition to finite words $w\in\Sigma^*$and write $q, \gamma w\longrightarrow_A q',\gamma'$ if there exists a sequence of transitions labelled by w from $q,\gamma$ to $q',\gamma'$.

The language of A is $L(A)\text{def}=\{w\in\Sigma^*| \exists qf\in F,\exists\gamma\in\Gamma^*, q0,z0w\rightarrow Aqf,\gamma\}$. Its language by empty stack is $N(A)\text{def}=\{w\in\Sigma^*|\exists q\in Q, q0,z0w\longrightarrow_A q,\varepsilon\}$.

**Exercise 1(Deterministic Pushdown Automata.)**

A PDA is deterministic(a DPDA)if, for all $q\in Q$ and $z\in\Gamma$ ,either there is an applicable $\varepsilon$-transition $(q,z,\varepsilon,q',\gamma')$ for some $q'$ and $\gamma'$, and then it is the only applicable transition:$|\{(a,q',\gamma')|(q,z,a,q',\gamma')\in\delta\}|=1$,

or

there is no applicable $\varepsilon$-transition and for all $b\in\Sigma$, there is at most one applicable transition with label b:$|\{(q',\gamma')|(q,z,b,q',\gamma')\in\delta\}|\leq 1$.

A context-free language L is deterministic if there exists a DPDA A with L=L(A).

1. Show that the set of centered palindromes $L_{pal}\text{def}=\{w\$w^R|w\in \{a,b\}^*\}$is deterministic ($\cdot R$ denotes the 'mirror', aka 'reversal' operation on finite words).
2. We can generalise the previous example. Let $h: \Sigma^*\rightarrow\Delta^*$be a homomorphism and let $\$\in\Sigma\cup\Delta$ be a symbol. Show that the language $L_h\text{def}=\{w\$(h(w))^R|w\in\Sigma+\}$ is deterministic.
3. Show that a regular language is also deterministic context-free

**Non-deterministic Pushdown Automata**

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

**Example:**

Design PDA for Palindrome strips.

**Solution:**

Suppose the language consists of string L = {aba, aa, bb, bab, bbabb, aabaa, .....}. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

**Simulation of abaaba**

1. $\delta(q1, abaaba, Z)$ Apply rule 1
2. $\vdash \delta(q1, baaba, aZ)$ Apply rule 5
3. $\vdash \delta(q1, aaba, baZ)$ Apply rule 4
4. $\vdash \delta(q1, aba, abaZ)$ Apply rule 7
5. $\vdash \delta(q2, ba, baZ)$ Apply rule 8
6. $\vdash \delta(q2, a, aZ)$ Apply rule 7
7. $\vdash \delta(q2, \varepsilon, Z)$ Apply rule 11
8. $\vdash \delta(q2, \varepsilon)$ Accept

**Closure Properties**

In the following, we will need two more closure properties of deterministic context-free languages—that will hopefully be proven during the second part of the course:

**Theorem 1**(Closure under Complementation).

Given a DPDA A over input alphabet $\Sigma$, we can compute a DPDA $A'$ over $\Sigma$ such that $L(A') = \Sigma^*\backslash L(A)$.

**Theorem 2**(Closure under Left Quotient).Given a DPDA A over input alphabet $\Sigma$ and a word $w\in\Sigma^*$, we can

compute a DPDA A ′over Σ such that $L(A') = w^{-1} \cdot L(A)$.

Undecidability Results

We prove in this section several undecidability results for context-free languages.

**Exercise 4**(Basic Undecidable Problems).

In the following, we are given as input of the decision problem one or two PDAs A1andA2 over some alphabet Σ.

1. (Emptiness of Intersection) Show by a reduction from the Post Correspondence [Problem that the problem whether L(A1)∩L(A2) is empty is undecidable, even if A1 and A2 are deterministic.
2. Show the following corollaries:
   (a) (Inclusion) whether L(A1)⊆L(A2) is undecidable, even ifA1andA2aredeterministic.
   (b) (Universality) whether L(A1) = Σ*is undecidable.

**Video Content / Details of website for further learning (if any):**

http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node37.html

http://www.lsv.fr/~schmitz/teach/2016_formlang/homework01.pdf

https://www.javatpoint.com/non-deterministic-pushdown-automata

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(270-274)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Estd. 2000

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-49**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Non-Deterministic pushdown automata

**Introduction : ( Maximum 5 sentences)** :

- In automata theory, a deterministic pushdown automaton 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.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Pushdown Automata
- Instantaneous Description

**Detailed content of the Lecture:**

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

**Example:**

Design PDA for Palindrome strips.

**Solution:**

Suppose the language consists of string L = {aba, aa, bb, bab, bbabb, aabaa, .....}. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

**Simulation of abaaba**

9. $\delta(q1, abaaba, Z)$ Apply rule 1
10. $\vdash \delta(q1, baaba, aZ)$ Apply rule 5
11. $\vdash \delta(q1, aaba, baZ)$ Apply rule 4
12. $\vdash \delta(q1, aba, abaZ)$ Apply rule 7

13. $\vdash \delta(q2, ba, baZ)$ Apply rule 8
14. $\vdash \delta(q2, a, aZ)$ Apply rule 7
15. $\vdash \delta(q2, \varepsilon, Z)$ Apply rule 11
16. $\vdash \delta(q2, \varepsilon)$ Accept

**Closure Properties**

In the following, we will need two more closure properties of deterministic context-free languages—that will hopefully be proven during the second part of the course:

**Theorem 1**(Closure under Complementation).

Given a DPDA A over input alphabet $\Sigma$, we can compute a DPDA $A'$ over $\Sigma$ such that $L(A') = \Sigma^* \backslash L(A)$.

**Theorem 2**(Closure under Left Quotient).Given a DPDA A over input alphabet $\Sigma$ and a word $w \in \Sigma^*$, we can compute a DPDA $A'$over $\Sigma$ such that $L(A') = w^{-1} \cdot L(A)$.

Undecidability Results

We prove in this section several undecidability results for context-free languages.

**Exercise 4**(Basic Undecidable Problems).

In the following, we are given as input of the decision problem one or two PDAs A1andA2 over some alphabet $\Sigma$.

3. (Emptiness of Intersection) Show by a reduction from the Post Correspondence [Problem that the problem whether $L(A1) \cap L(A2)$ is empty is undecidable, even if A1 and A2 are deterministic.
4. Show the following corollaries:
   (a) (Inclusion) whether $L(A1) \subseteq L(A2)$ is undecidable, even ifA1andA2aredeterministic.
   (b) (Universality) whether $L(A1) = \Sigma^*$is undecidable.

**Video Content / Details of website for further learning (if any):**

http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node37.html

http://www.lsv.fr/~schmitz/teach/2016_formlang/homework01.pdf

https://www.javatpoint.com/non-deterministic-pushdown-automata

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(270-274)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-50**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Equivalence of Pushdown automata and CFL

**Introduction : ( Maximum 5 sentences)** :

- PDA & Context-Free Grammar. 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.
- In the next two topics, we will discuss how to convert from PDA to **CFG** and vice versa.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Deterministic pushdown automaton

**Detailed content of the Lecture:**

If a grammar **G** is context-free, we can build an equivalent non deterministic 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 )}
δ(q, 1, 1) = {(q, ε )}

***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**

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)

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/pda_context_free_grammar.html

https://www.tutorialspoint.com/automata_theory/pda_and_parsing.html

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(261-269)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

L-51

## LECTURE HANDOUTS

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Problem based on Equivalence of Pushdown automata and CFL

**Introduction : ( Maximum 5 sentences)** :

- PDA & Context-Free Grammar. 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.
- In the next two topics, we will discuss how to convert from PDA to **CFG** and vice versa.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Deterministic pushdown automaton

**Detailed content of the Lecture:**

If a grammar **G** is context-free, we can build an equivalent non deterministic 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 )}
δ(q, 1, 1) = {(q, ε )}

***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**

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)

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/pda_context_free_grammar.html

https://www.tutorialspoint.com/automata_theory/pda_and_parsing.html

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(261-269)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-52**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Pumping lemma for CFL

**Introduction : ( Maximum 5 sentences) :**

- Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language.
- For any language L, we break its strings into five parts and pump second and fourth substring .

Prerequisite knowledge for Complete understanding and learning of Topic:

( Max. Four important topics)

- Deterministic pushdown automaton

**Detailed content of the Lecture:**

**Pumping Lemma**

- **Pumping Lemma** is used as a **proof** for irregularity of a language. Thus, if a language is regular, it always satisfies **pumping lemma**.
- If there exists at least one string made from **pumping** which is not in L, then L is surely not regular.

There are two Pumping Lemmas, which are defined for

1. Regular Languages, and
2. Context – Free Languages

**Pumping Lemma for Regular Languages**

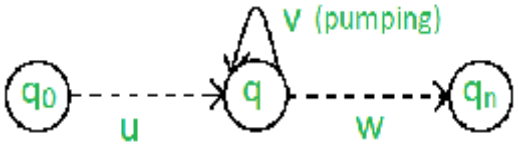
For any regular language L, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, $w \in \Sigma*$, such that $x = uvw$, and

(1) $|uv| \leq n$

(2) $|v| \geq 1$

(3) for all $i \geq 0$: $uv^iw \in L$

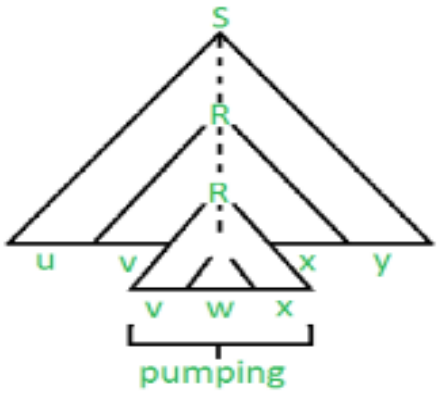
In simple terms, this means that if a string v is 'pumped', i.e., if v is inserted any number of times, the resultant string still remains in L.
Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.
The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.




For example, let us prove $L_{01} = \{0^n1^n \mid n \geq 0\}$ is irregular.
Let us assume that L is regular, then by Pumping Lemma the above given rules follow.
Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w such that (1) – (3) hold.

We show that for all u, v, w, (1) – (3) does not hold.
If (1) and (2) hold then $x = 0^n1^n = uvw$ with $|uv| \leq n$ and $|v| \geq 1$.
So, $u = 0^a$, $v = 0^b$, $w = 0^c1^n$ where : $a + b \leq n$, $b \geq 1$, $c \geq 0$, $a + b + c = n$
But, then (3) fails for i = 0
$uv^0w = uw = 0^a0^c1^n = 0^{a+c}1^n \notin L$, since $a + c \neq n$.


**Pumping Lemma for Context-free Languages (CFL)**
Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language. For any language L, we break its strings into five parts and pump second and fourth substring.
Pumping Lemma, here also, is used as a tool to prove that a language is not CFL. Because, if any one string does not satisfy its conditions, then the language is not CFL.
Thus, if L is a CFL, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists u, v, w, x, $y \in \Sigma*$, such that x = uvwxy, and
(1) $|vwx| \leq n$
(2) $|vx| \geq 1$
(3) for all $i \geq 0$: $uv^iwx^iy \in L$




For above example, $0^n1^n$ is CFL, as any string can be the result of pumping at two places, one for 0 and other for 1.
Let us prove, $L_{012} = \{0^n1^n2^n \mid n \geq 0\}$ is not Context-free.
Let us assume that L is Context-free, then by Pumping Lemma, the above given rules follow.
Now, let $x \in L$ and $|x| \geq n$. So, by Pumping Lemma, there exists u, v, w, x, y such that (1) – (3) hold.
We show that for all u, v, w, x, y (1) – (3) do not hold.

If (1) and (2) hold then $x = 0^n1^n2^n = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$.
(1) tells us that vwx does not contain both 0 and 2. Thus, either vwx has no 0's, or vwx has no 2's.

Thus, we have two cases to consider.
Suppose vwx has no 0's. By (2), vx contains a 1 or a 2. Thus uwy has 'n' 0's and uwy either has less than 'n' 1's or has less than 'n' 2's.
But (3) tells us that $uwy = uv^0wx^0y \in L$.
So, uwy has an equal number of 0's, 1's and 2's gives us a contradiction. The case where vwx has no 2's is similar and also gives us a contradiction. Thus L is not context-free.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/pumping-lemma-in-theory-of-computation/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(297-301)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-53**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Problems based on pumping Lemma

**Introduction : ( Maximum 5 sentences) :**

- Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language.
- For any language L, we break its strings into five parts and pump second and fourth substring .

Prerequisite knowledge for Complete understanding and learning of Topic:

( Max. Four important topics)

- Deterministic pushdown automaton

**Detailed content of the Lecture:**

**Pumping Lemma**

- **Pumping Lemma** is used as a **proof** for irregularity of a language. Thus, if a language is regular, it always satisfies **pumping lemma**.
- If there exists at least one string made from **pumping** which is not in L, then L is surely not regular.

There are two Pumping Lemmas, which are defined for

1. Regular Languages, and
2. Context – Free Languages

**Problem:** $L = \{ w \mid w \in \{a, b\}^*, w = w^R \}$

Proof by contradiction:

Assume L is regular. Then the pumping lemma applies.

From the pumping lemma there exists an n such that every $w \in L$ longer than n can be represented as x y z with $|y| \neq 0$ and $|x\ y| \leq n$.

Let us choose the palindrome $a^n b a^n$ .

Again notice that we were clever enough to choose a string which:

**Problem:** L = $\{a^n b^{2n}\}$

Assume L is regular. From the pumping lemma there exists a p such that every w ∈ L such that |w| ≥ p can be represented as x y z with |y| # 0 and |xy| ≤ p. Let us choose $a^p b^{2p}$ . Its length is 3p ≥ p. Since the length of xy cannot exceed p, y must be of the form a for some k > 0.

From the pumping lemma $a^{p-k}\ b^{2p}$ must also be in L but it is not of the right form. Hence the language is not regular.

**Problem:** L = { w ∈ {a, b}* | w has an equal number of a's and b's}

Let us show this by contradiction: assume L is regular. We know that the language generated by a*b* is regular. We also know that the intersection of two regular languages is regular. Let M = $(a^n b^n \mid n \geq 0\}$ = L(a*b*) ∩ L. Therefore if L is regular M would also be regular. but we know that M is not regular. Hence, L is not regular.

**Problem:** L = $\{a^n b a^{3n} \mid n \geq 0\}$

Assume L is regular. From the pumping lemma there exists a p such that every w ∈ L such that |w| ≥ p can be represented as x y z with |y| #0 and |xy| ≤ p. Let us choose $a^p b a^{3p}$ . Its length is 4p+1 ≥ p. Since the length of xy cannot exceed p , y must be of the form $a^k$ for some k > 0. From the pumping lemma $a^{p-k} b a^{3p}$ must also be in L but it is not of the right form. Hence the language is not regular.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/pumping-lemma-in-theory-of-computation/

http://athena.ecs.csus.edu/~gordonvs/135/resources/pumpingLemmaExerALRsol.pdf

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(301-304)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-54**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **IV Pushdown Automata.** **Date of Lecture:**

**Topic of Lecture:** Closure Properties of Context free Languages

**Introduction : ( Maximum 5 sentences) :**

Context-free languages are **closed** under −

- Union
- Concatenation
- Kleene Star operation

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Deterministic pushdown automaton**

**Detailed content of the Lecture:**

**Closure Properties of Context Free Languages**

Context Free languages are accepted by pushdown automata but not by finite automata. Context free languages can be generated by context free grammar which has the form :

A -> ρ (where A ∈ N and ρ ∈ (T ∪ N)* and N is a non-terminal and T is a terminal)

**Properties of Context Free Languages**

**Union :** If L1 and If L2 are two context free languages, their union L1 ∪ L2 will also be context free. For example,

L1 = { $a^n b^n c^m$ | m >= 0 and n >= 0 } and L2 = { $a^n b^m c^m$ | n >= 0 and m >= 0 }

L3 = L1 ∪ L2 = { $a^n b^n c^m$ ∪ $a^n b^m c^m$ | n >= 0, m >= 0 } is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.

**Note:** So CFL are closed under Union.

**Concatenation :** If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free. For example,

L1 = { $a^n b^n$ | n >= 0 } and L2 = { $c^m d^m$ | m >= 0 }

L3 = L1.L2 = { $a^n b^n c^m d^m$ | m >= 0 and n >= 0} is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's. So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.
**Note:** So CFL are closed under Concatenation.

**Kleene Closure :** If L1 is context free, its Kleene closure L1* will also be context free. For example,
L1 = { $a^nb^n$ | n >= 0 }
L1* = { $a^nb^n$ | n >= 0 }* is also context free.
**Note :**So CFL are closed under Kleen Closure.

**Intersection and complementation :** If L1 and If L2 are two context free languages, their intersection L1 ∩ L2 need not be context free. For example,
L1 = { $a^nb^nc^m$ | n >= 0 and m >= 0 } and L2 = ($a^mb^nc^n$ | n >= 0 and m >= 0 }
L3 = L1 ∩ L2 = { $a^nb^nc^n$ | n >= 0 } need not be context free.
L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.
Similarly, complementation of context free language L1 which is ∑* – L1, need not be context free.
**Note :** So CFL are not closed under Intersection and Complementation.

**Deterministic Context-free Languages**
Deterministic CFL are subset of CFL which can be recognized by Deterministic PDA. Deterministic PDA has only one move from a given state and input symbol, i.e., it do not have choice. For a language to be DCFL it should be clear when to PUSh or POP.
**For example**, L1= { $a^nb^nc^m$ | m >= 0 and n >= 0} is a DCFL because for a's, we can push on stack and for b's we can pop. It can be recognized by Deterministic PDA. On the other hand, L3 = { $a^nb^nc^m$ ∪ $a^nb^mc^m$ | n >= 0, m >= 0 } cannot be recognized by DPDA because either number of a's and b's can be equal or either number of b's and c's can be equal. So, it can only be implemented by NPDA. Thus, it is CFL but not DCFL.

**Note :** DCFL are closed only under complementation and Inverse Homomorphism.

**Question :** The language L = { $0^i12^i$ | i ≥ 0 } over the alphabet {0, 1, 2} is :
A. Not recursive
B. Is recursive and deterministic CFL
C. Is regular
D. Is CFL bot not deterministic CFL.
**Solution :** The above language is deterministic CFL as for 0's, we can push 0 on stack and for 2's we can pop corresponding 0's. As there is no ambiguity which moves to take, it is deterministic. So, correct option is (B). As CFL is subset of recursive, it is recursive as well.
**Question :** Consider the following languages:
L1 = { $0^n1^n$| n≥0 }
L2 = { wcwr | w ε {a,b}* }
L3 = { wwr | w ε {a,b}* }
Which of these languages are deterministic context-free languages?
A. None of the languages
B. Only L1
C. Only L1 and L2

D. All three languages

**Solution :** Languages L1 contains all strings in which n 0's are followed by n 1's. Deterministic PDA can be constructed to accept L1. For 0's we can push it on stack and for 1's, we can pop from stack. Hence, it is DCFL.
L2 contains all strings of form wcwr where w is a string of a's and b's and wr is reverse of w. For example, aabbcbbaa. To accept this language, we can construct PDA which will push all symbols on stack before c. After c, if symbol on input string matches with symbol on stack, it is popped. So, L2 can also be accepted with deterministic PDA, hence it is also DCFL.
L3 contains all strings of form wwr where w is a string of a's and b's and wr is reverse of w. But we don't know where w ends and wr starts. e.g.; aabbaa is a string corresponding to L3. For first a, we will push it on stack. Next a can be either part of w or wr where w=a. So, there can be multiple moves from a state on an input symbol. So, only non-deterministic PDA can be used to accept this type of language. Hence, it is NCFL not DCFL.
So, correct option is (C). Only, L1 and L2 are DCFL.

**simplification of CFG**
In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called **simplification of CFGs**. Simplification essentially comprises of the following steps −

- Reduction of CFG
- Removal of Unit Productions
- Removal of Null Productions

**Reduction of CFG**
CFGs are reduced in two phases −
**Phase 1** − Derivation of an equivalent grammar, **G'**, from the CFG, **G**, such that each variable derives some terminal string.
**Derivation Procedure** −
Step 1 − Include all symbols, $\mathbf{W_1}$, that derive some terminal and initialize **i=1**.
Step 2 − Include all symbols, $\mathbf{W_{i+1}}$, that derive $\mathbf{W_i}$.
Step 3 − Increment **i** and repeat Step 2, until $\mathbf{W_{i+1} = W_i}$.
Step 4 − Include all production rules that have $\mathbf{W_i}$ in it.
**Phase 2** − Derivation of an equivalent grammar, **G"**, from the CFG, **G'**, such that each symbol appears in a sentential form.
**Derivation Procedure** −
Step 1 − Include the start symbol in $\mathbf{Y_1}$ and initialize $\mathbf{i = 1}$.
Step 2 − Include all symbols, $\mathbf{Y_{i+1}}$, that can be derived from $\mathbf{Y_i}$ and include all production rules that have been applied.
Step 3 − Increment **i** and repeat Step 2, until $\mathbf{Y_{i+1} = Y_i}$.

**Problem**
Find a reduced grammar equivalent to the grammar G, having production rules, P: S → AC | B, A → a, C → c | BC, E → aA | e

**Solution**
**Phase 1** −
T = { a, c, e }

$W_1$ = { A, C, E } from rules A → a, C → c and E → aA
$W_2$ = { A, C, E } U { S } from rule S → AC
$W_3$ = { A, C, E, S } U ∅
Since $W_2 = W_3$, we can derive G' as −
G' = { { A, C, E, S }, { a, c, e }, P, {S}}
where P: S → AC, A → a, C → c , E → aA | e
**Phase 2** −
$Y_1$ = { S }
$Y_2$ = { S, A, C } from rule S → AC
$Y_3$ = { S, A, C, a, c } from rules A → a and C → c
$Y_4$ = { S, A, C, a, c }
Since $Y_3 = Y_4$, we can derive G" as −
G" = { { A, C, S }, { a, c }, P, {S}}
where P: S → AC, A → a, C → c

**Removal of Unit Productions**
Any production rule in the form A → B where A, B ∈ Non-terminal is called **unit production.**.
**Removal Procedure −**
**Step 1** − To remove **A → B**, add production **A → x** to the grammar rule whenever **B → x** occurs in the grammar. [x ∈ Terminal, x can be Null]
**Step 2** − Delete **A → B** from the grammar.
**Step 3** − Repeat from step 1 until all unit productions are removed.

**Problem**
Remove unit production from the following −
S → XY, X → a, Y → Z | b, Z → M, M → N, N → a
**Solution** −
There are 3 unit productions in the grammar −
Y → Z, Z → M, and M → N
**At first, we will remove M → N.**
As N → a, we add M → a, and M → N is removed.
The production set becomes
S → XY, X → a, Y → Z | b, Z → M, M → a, N → a
**Now we will remove Z → M.**
As M → a, we add Z→ a, and Z → M is removed.
The production set becomes
S → XY, X → a, Y → Z | b, Z → a, M → a, N → a
**Now we will remove Y → Z.**
As Z → a, we add Y→ a, and Y → Z is removed.
The production set becomes
S → XY, X → a, Y → a | b, Z → a, M → a, N → a
Now Z, M, and N are unreachable, hence we can remove those.
The final CFG is unit production free −
S → XY, X → a, Y → a | b
**Removal of Null Productions**
In a CFG, a non-terminal symbol **'A'** is a nullable variable if there is a production **A → ε** or there is a derivation that starts at **A** and finally ends up with
ε: A → .....… → ε
**Removal Procedure**
**Step 1** − Find out nullable non-terminal variables which derive ε.

**Step 2** − For each production **A → a**, construct all productions **A → x** where **x** is obtained from **'a'** by removing one or multiple non-terminals from Step 1.
**Step 3** − Combine the original productions with the result of step 2 and remove **ε - productions**.

**Problem**
Remove null production from the following −
S → ASA | aB | b, A → B, B → b | ∈
**Solution** −
There are two nullable variables − **A** and **B**
**At first, we will remove B → ε.**
After removing **B → ε**, the production set becomes −
S→ASA | aB | b | a, A ε B| b | &epsilon, B → b
**Now we will remove A → ε.**
After removing **A → ε**, the production set becomes −
S→ASA | aB | b | a | SA | AS | S, A → B| b, B → b
This is the final production set without null transition.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/closure-properties-of-context-free-languages/

https://www.tutorialspoint.com/automata_theory/cfg_simplification.htm

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(305-315)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-61**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Definitions of Turing machines – Models

**Introduction : ( Maximum 5 sentences)** :

- A **Turing Machine** (TM) is a mathematical **model** which consists of an infinite length tape divided into cells on which input is given.
- 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.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Automaton

**Detailed content of the Lecture:**

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).

A Turing machine consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions. A TM is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- ∑ is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Let us construct a turing machine for L={0n1n|n>=1}

- Q = {q0,q1,q2,q3} where q0 is initial state.
- T = {0,1,X,Y,B} where B represents blank.
- ∑ = {0,1}
- F = {q3}
-

**Transition function δ is given in Table 1 as:**

| | 0 | 1 | X | Y | B |
|---|---|---|---|---|---|
| **q0** | (q1,X,R) | | | (q3,Y,R) | |
| **q1** | (q1,0,R) | (q2,Y,L) | | (q1,Y,R) | |
| **q2** | (q2,0,L) | | (q0,X,R) | (q2,Y,L) | |
| **q3** | | | | (q3,Y,R) | **Halt** |

**Illustration**

Let us see how this turing machine works for 0011. Initially head points to 0 which is underlined and state is q0 as:

| B | 0 | 0 | 1 | 1 | B |
|---|---|---|---|---|---|

The move will be δ(q0, 0) = (q1, X, R). It means, it will go to state q1, replace 0 by X and head will move to right as:

| B | X | 0 | 1 | 1 | B |
|---|---|---|---|---|---|

The move will be δ(q1, 0) = (q1, 0, R) which means it will remain in same state and without changing any symbol, it will move to right as:

| B | X | 0 | 1 | 1 | B |
|---|---|---|---|---|---|

The move will be δ(q1, 1) = (q2, Y, L) which means it will move to q2 state and changing 1 to Y, it will move to left as:

| B | X | 0 | Y | 1 | B |
|---|---|---|---|---|---|

Working on it in the same way, the machine will reach state q3 and head will point to B as shown:


Using move δ(q3, B) = halt, it will stop and accepted.

**Note:**

- In non-deterministic turing machine, there can be more than one possible move for a given state and tape symbol, but non-deterministic TM does not add any power.
- Every non-deterministic TM can be converted into deterministic TM.

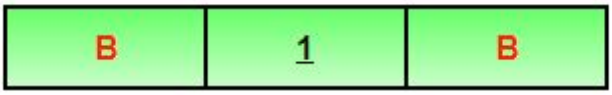
**Question:** A single tape Turing Machine M has two states q0 and q1, of which q0 is the starting state. The tape alphabet of M is {0, 1, B} and its input alphabet is {0, 1}. The symbol B is the blank symbol used to indicate end of an input string. The transition function of M is described in the following table.

| | 0 | 1 | B |
|---|---|---|---|
| q0 | q1,1,R | q1,1,R | Halt |
| q1 | q1,1,R | q0,L,R | q0,B,L |

The table is interpreted as illustrated below. The entry (q1, 1, R) in row q0 and column 1 signifies that if M is in state q0 and reads 1 on the current tape square, then it writes 1 on the same tape square, moves its tape head one position to the right and transitions to state q1. Which of the following statements is true about M?

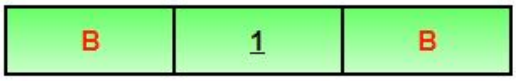
1. M does not halt on any string in (0 + 1)+
2. M does not halt on any string in (00 + 1)*
3. M halts on all string ending in a 0
4. M halts on all string ending in a 1

**Solution:** Let us see whether machine halts on string '1'. Initially state will be q0, head will point to 1 as:

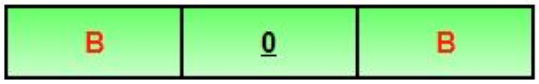



Using δ(q0, 1) = (q1, 1, R), it will move to state q1 and head will move to right as:


Using δ(q1, B) = (q0, B, L), it will move to state q0 and head will move to left as:

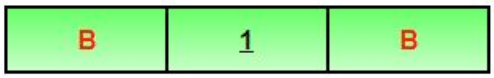


It will run in the same way again and again and not halt.
Option D says M halts on all string ending with 1, but it is not halting for 1. So, option D is incorrect.
Let us see whether machine halts on string '0'. Initially state will be q0, head will point to 1 as:

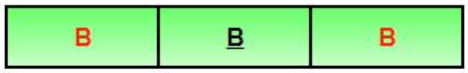



Using δ(q0, 0) = (q1, 1, R), it will move to state q1 and head will move to right as:


Using δ(q1,B)=(q0,B,L), it will move to state q0 and head will move to left as:




It will run in the same way again and again and not halt.
Option C says M halts on all string ending with 0, but it is not halting for 0. So, option C is incorrect.
Option B says that TM does not halt for any string (00 + 1)*. But NULL string is a part of (00 + 1)* and TM will halt for NULL string. For NULL string, tape will be,




Using δ(q0, B) = halt, TM will halt. As TM is halting for NULL, this option is also incorrect.
So, option (A) is correct.

**Video Content / Details of website for further learning (if any):**

https://www.geeksforgeeks.org/turing-machine-in-toc/

https://www.youtube.com/watch?v=jZI1v_l42pE

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(342-349)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-62**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Computable languages and functions

**Introduction : ( Maximum 5 sentences)** :

- According to the Church–**Turing** thesis, **computable functions** are exactly the **functions** that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space.
- Equivalently, this thesis states that a **function** is **computable** if and only if it has an algorithm.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Turing Machine

Detailed content of the Lecture:

A Turing machine is a computing device that on input a finite sequence of strings might produce a string. The machine has two parts: a control unit and a tape. The control unit is characterized by a finite set Q of control states, a start state $q0 \in Q$ and a transition function $\delta$. The tape is a one-way infinite sequence of cells and each cell contains a symbol from a tape alphabet $\Gamma$. In $\Gamma$ there is a special symbol t, called the blank. The control unit is always located above a cell of the tape and it can move left or right. A computation is performed as follows. Initially the control unit is located at the leftmost cell of the tape and it is in the start stateq0. All cells of the tape are blank except at the beginning of the tape, where the input strings are written, separated by blanks.

During the computation, the control unit transforms the tape step by step. At each step, depending on the current state and the symbol below the control head, the transition function, specifies.

- which symbol is written in the cell below the computation head,
- whether the computation head moves one cell left or right,
- the new state of the computation head.

The transition function is a partial function. If it is undefined for some state and symbol, or the head tries to move left on the first cell of the tape, the computation is finished. If this happens, and the tape contains $r\underline{\infty}$for some string r that has no blanks, then r is the output of the computation. More precisely, the transition function is a function $\delta: Q \times \Gamma \to \Gamma \times \{-1,1\} \times Q$. A value $\delta(q,a) = (b,m,r)$ means that if the control head is in a state q above a cell containing $a \in \Gamma$, it write $sb \in \Gamma$, moves one cell left ($m=-1$) or right ($m= 1$), and finally, changes to the control state $r \in Q$. A computation is just a repetition

of such transformations. Notation. For a finite set A, let A∞ be the set of infinite sequences of elements in A. The ith element of such a T∈A∞ is written as Ti∈A. Let a∞=aa.... Let␣ the a fixed symbol. N={1,2,3,...}.At each computation step, the state of the Turing machine is fully described by the position i∈N of the computation head, the state q∈Q of the computation head and the content T∈Γ∞ of the tape, i.e., by (i,q,T) ∈N×Q×Γ∞.

**Definition 1.**
A deterministic Turing machine is a 4-tuple(Q,Γ,δ,q0) where Q and Γ are finite sets,q0∈Q,t∈Γ, and δ:Q×Γ→Γ×{−1,1}×Qis a partial function. A Turing machine defines a directed graph whose vertices are elements of N×Q×Γ∞ (called states).There is an edge from a state(i,q,T) to the state(i+m, r, T1...Ti−1bTi+1...)if and only if δ(q,Ti) = (b,m,r). There are no other edges. If for r,u,v,...,z∈(Γ\ {t})*the path starting in the state(1,q0,u␣ v␣ ...␣ z␣ ∞) is finite and terminates in(1,q,r␣ ∞), we say that U outputs r on input u,v,...,z and write U(u,v,...,z) =r, otherwise the value is undefined.

We also use Turing machines that recognize a language. In the part of computational complexity we use the non-deterministic variant of such machines.

**Expanding the input.**
Let a∈Σ be a symbol. The function f: Σ*→Σ*=x1ax2a...x|x| a is computable. Probably, you can already implement a TM yourself to prove this claim. Here is my solution.
The idea to implement the machine is to first insert the leftmost a symbol, and thus move x2...x|x| one position to the right. After this move, the machine needs to return to x2 and insert an a to the right ofx2and hence movex3...x|x| and so on. This approach has a problem: when the machine needs to return, it does not know where it was last because the input can contain a letters. For this it inserts a symbol ̇a∈Σ, rather than a. After moving the remainder of the input one cell to the right, the machine returns to the ̇a symbol, replaces it by a, skips a letter and then again inserts ̇a. Assume t ␣∈Σ. The machine is given by

$$(\{s,l,j,j'\}\cup\{ra{:}a\text{␣}\ \Sigma\},\Sigma,\{t,\text{␣}\ ,\dot{a}\}\cup\Sigma,\delta,s),$$

With a similar technique, we can insert a string aa or $a^k$ for some k≥1 after each letter of the input string. One can also construct a machine that does the reverse: compressing the input by deleting all bits at positions different from 1 modulo k.

**Video Content / Details of website for further learning (if any):**
https://www.hse.ru/mirror/pubs/share/202374148

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(328-331)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-63**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Techniques for Turing machine construction

**Introduction : ( Maximum 5 sentences)** :

- A **Turing Machine** (TM) is a mathematical **model** which consists of an infinite length tape divided into cells on which input is given.
- 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.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Turing Machine**

**Detailed content of the Lecture:**

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).

A Turing machine consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions. A TM is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- ∑ is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps $Q \times T \rightarrow Q \times T \times \{L,R\}$. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

# Accepted Language & Decided Language

A TM accepts a language if it enters into a final state for any input string w. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.
A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.
There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

## Designing a Turing Machine

The basic guidelines of designing a Turing machine have been explained below with the help of a couple of examples.

**Example 1**

Design a TM to recognize all strings consisting of an odd number of α's.

***Solution***

The Turing machine **M** can be constructed by the following moves −

- Let $\mathbf{q_1}$ be the initial state.
- If **M** is in $\mathbf{q_1}$; on scanning α, it enters the state $\mathbf{q_2}$ and writes **B** (blank).
- If **M** is in $\mathbf{q_2}$; on scanning α, it enters the state $\mathbf{q_1}$ and writes **B** (blank).
- From the above moves, we can see that **M** enters the state $\mathbf{q_1}$ if it scans an even number of α's, and it enters the state $\mathbf{q_2}$ if it scans an odd number of α's. Hence $\mathbf{q_2}$ is the only accepting state.

Hence,
$M = \{\{q_1, q_2\}, \{1\}, \{1, B\}, \delta, q_1, B, \{q_2\}\}$
where δ is given by −

| Tape alphabet symbol | Present State '$q_1$' | Present State '$q_2$' |
|---|---|---|
| α | $Bq_2$ | $Bq_1$ |

**Example 2**

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

***Solution***

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.
The Turing Machine, **M**, can be constructed by the following moves −

- Let $\mathbf{q_0}$ be the initial state.
- If **M** is in $\mathbf{q_0}$, on reading 0, it moves right, enters the state $\mathbf{q_1}$ and erases 0. On reading 1, it enters the state $\mathbf{q_2}$ and moves right.
- If **M** is in $\mathbf{q_1}$, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters $\mathbf{q_2}$ and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state $\mathbf{q_3}$.
- If **M** is in $\mathbf{q_2}$, on reading either 0 or 1, it moves right. On reaching B, it moves left and enters the state $\mathbf{q_4}$. This validates that the string comprises only of 0's and 1's.
- If **M** is in $\mathbf{q_3}$, it replaces B by 0, moves left and reaches the final state $\mathbf{q_f}$.
- If **M** is in $\mathbf{q_4}$, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state $\mathbf{q_f}$.

Hence,
$M = \{\{q_0, q_1, q_2, q_3, q_4, q_f\}, \{0,1, B\}, \{1, B\}, \delta, q_0, B, \{q_f\}\}$
where δ is given by −

| Tape alphabet symbol | Present State '$q_0$' | Present State '$q_1$' | Present State '$q_2$' | Present State '$q_3$' | Present State '$q_4$' |
|---|---|---|---|---|---|
| 0 | $BRq_1$ | $BRq_1$ | $ORq_2$ | - | $OLq_4$ |
| 1 | $1Rq_2$ | $1Rq_2$ | $1Rq_2$ | - | $1Lq_4$ |
| B | $BRq_1$ | $BLq_3$ | $BLq_4$ | $OLq_f$ | $BRq_f$ |

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/accepted_and_decided_language.htm

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(355-361)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-64**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Multi head and Multi tape Turing Machines

**Introduction : ( Maximum 5 sentences)** :

- A multi-tape Turing machine is like an ordinary Turing machine with several tapes.
- Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank...

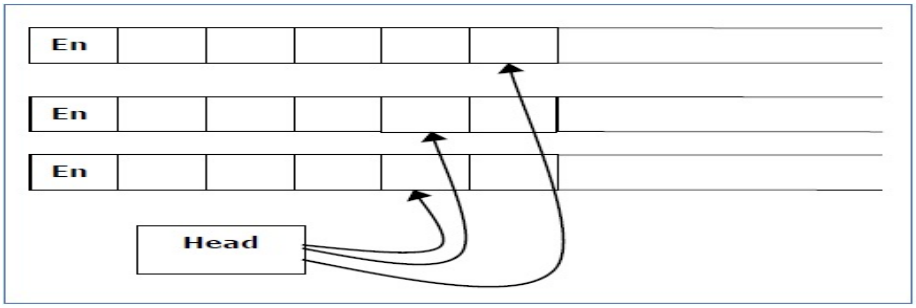
**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Turing Machine**

**Detailed content of the Lecture:**

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.




A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, $q_0$, F) where −

- **Q** is a finite set of states

- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where
  $\delta$: $Q \times X^k \rightarrow Q \times (X \times \{Left\_shift, Right\_shift, No\_shift\})^k$
  where there is **k** number of tapes
- $\mathbf{q_0}$ is the initial state
- **F** is the set of final states

**Note** − Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

## Multi-track Turing Machine

Multi-track Turing machines, a specific type of Multi-tape Turing machine, contain multiple tracks but just one tape head reads and writes on all tracks. Here, a single tape head reads n symbols from **n** tracks at one step..

A Multi-track Turing machine can be formally described as a 6-tuple $(Q, X, \sum, \delta, q_0, F)$ where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- $\sum$ is the input alphabet
- **δ** is a relation on states and symbols where
  $\delta(Q_i, [a_1, a_2, a_3,....]) = (Q_j, [b_1, b_2, b_3,....]$, Left_shift or Right_shift)
- $\mathbf{q_0}$ is the initial state
- **F** is the set of final states

**Note** − For every single-track Turing Machine **S**, there is an equivalent multi-track Turing Machine **M** such that **L(S) = L(M)**.

## Non-Deterministic Turing Machine

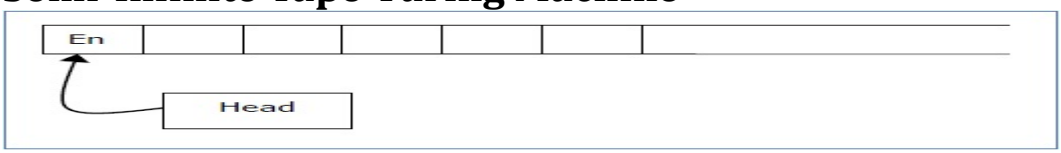
In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 6-tuple $(Q, X, \sum, \delta, q_0, B, F)$ where −

- **Q** is a finite set of states
- **X** is the tape alphabet
- $\sum$ is the input alphabet
- **δ** is a transition function;
  $\delta : Q \times X \rightarrow P(Q \times X \times \{Left\_shift, Right\_shift\})$.
- $\mathbf{q_0}$ is the initial state
- **B** is the blank symbol
- **F** is the set of final states

## Semi-Infinite Tape Turing Machine




t 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 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.

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/non_deterministic_turing_machine.htm

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(361-365)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-65**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Universal Turing machine - The Halting problem – Partial Solvability

**Introduction : ( Maximum 5 sentences)** :

- **universal Turing machine** (UTM) is a **Turing machine** that simulates an arbitrary **Turing machine** on arbitrary input.
- The **universal machine** essentially achieves this by reading both the description of the **machine** to be simulated as well as the input to that **machine** from its own tape
- The **halting problem** is a decision **problem** about properties of computer programs on a fixed **Turing**-complete model of computation,
- i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a **Turing machine**.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Turing Machine**

**Detailed content of the Lecture:**

**The Halting problem**

**Input** − A Turing machine and an input string **w**.

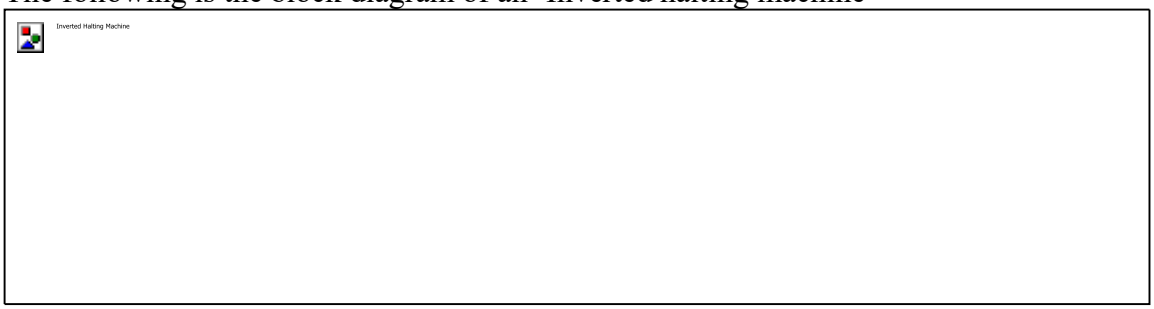
**Problem** − Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

**Proof** − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −


Now we will design an **inverted halting machine (HM)'** as −

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' −




Further, a machine **$(HM)_2$** which input itself is constructed as follows −

- If $(HM)_2$ halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.

## PARTIAL SOLVABILITY

Problem types

There are basically three types of problems namely

•Decidable / solvable /recursive

•Undecidable /unsolvable

•Semi decidable / partial solvable / recursively enumerable Decidable / solvable problem

A problem is said to be **Decidable** if we can always construct a corresponding **algorithm** that can answer the problem correctly. We can intuitively understand Decidable problems by considering a

simple example. Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000. To find the **solution** of this problem, we can easily devise an algorithm that can enumerate all the prime numbers in this range.

Now talking about Decidability in terms of a Turing machine, a problem is said to be a Decidable problem if there exists a corresponding Turing machine which **halts** on every input with an answer- **yes or no**. It is also important to know that these problems are termed as **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

**Semi- Decidable Problems –**
Semi-Decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the Turing Machine. Such problems are termed as **Turing Recognisable** problems.

**Examples –** We will now consider few important **Decidable problems**:

- Are two **regular** languages L and M **equivalent**?
  We can easily check this by using Set Difference operation.
  L-M =Null and M-L =Null.
  Hence (L-M) U (M-L) = Null, then L,M are equivalent.
- Membership of a CFL?
  We can always find whether a string exists in a given CFL by using an algorithm based on dynamic programming.
- Emptiness of a CFL
  By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

**Undecidable Problems –**
The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

We can understand Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: a^n + b^n = c^n.

If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an **Undecidable**

**Problem**.

**Examples –** These are few important **Undecidable Problems**:

- Whether a CFG generates all the strings or not?
  As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.
- Whether two CFG L and M equal?

Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

- Ambiguity of CFG?
  There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.
- Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?
  It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.
- Is a language Learning which is a CFL, regular?
  This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/turing_machine_halting_problem.htm

https://www.geeksforgeeks.org/decidable-and-undecidable-problems-in-theory-of-computation/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(352-353)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

Estd. 2000

IQAC

L-66

**LECTURE HANDOUTS**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** The Halting problem – Partial Solvability

**Introduction : ( Maximum 5 sentences)** :

- **universal Turing machine** (UTM) is a **Turing machine** that simulates an arbitrary **Turing machine** on arbitrary input.
- The **universal machine** essentially achieves this by reading both the description of the **machine** to be simulated as well as the input to that **machine** from its own tape
- The **halting problem** is a decision **problem** about properties of computer programs on a fixed **Turing**-complete model of computation,
- i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a **Turing machine**.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- **Turing Machine**

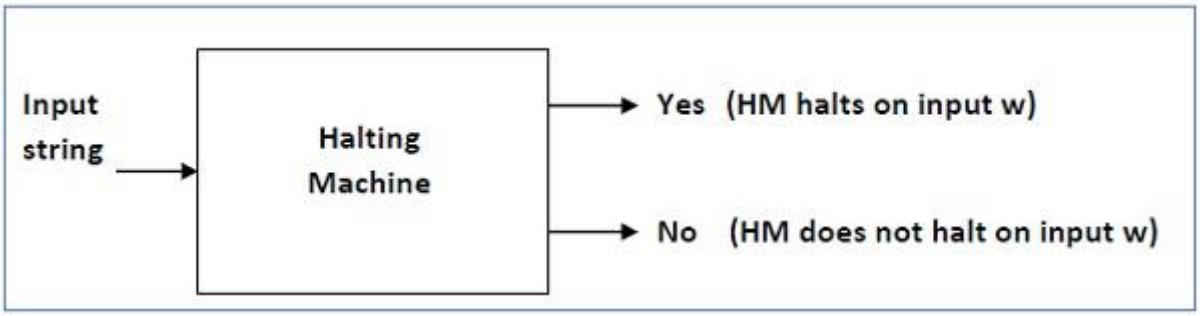
**Detailed content of the Lecture:**

**The Halting problem**

**Input** − A Turing machine and an input string **w**.

**Problem** − Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

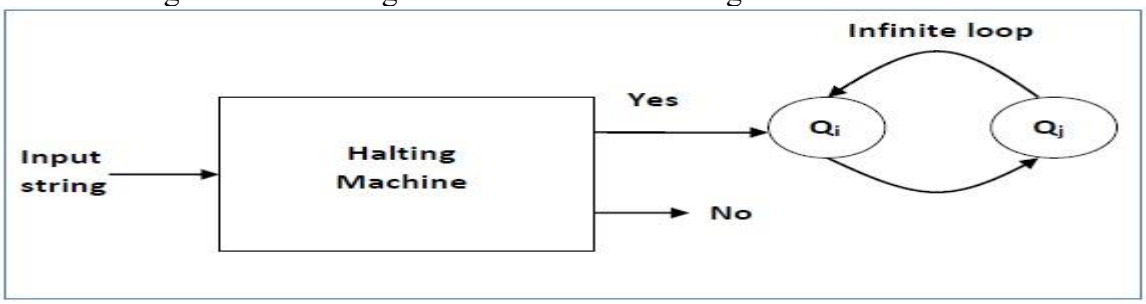
**Proof** − At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a **Halting machine** that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine −




Now we will design an **inverted halting machine (HM)'** as −

- If **H** returns YES, then loop forever.
- If **H** returns NO, then halt.

The following is the block diagram of an 'Inverted halting machine' −




Further, a machine **$(HM)_2$** which input itself is constructed as follows −

- If $(HM)_2$ halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is **undecidable**.

**PARTIAL SOLVABILITY**

Problem types

There are basically three types of problems namely

•Decidable / solvable /recursive

•Undecidable /unsolvable

•Semi decidable / partial solvable / recursively enumerable Decidable / solvable problem

A problem is said to be **Decidable** if we can always construct a corresponding **algorithm** that can answer the problem correctly. We can intuitively understand Decidable problems by considering a

simple example. Suppose we are asked to compute all the prime numbers in the range of 1000 to 2000. To find the **solution** of this problem, we can easily devise an algorithm that can enumerate all the prime numbers in this range.

Now talking about Decidability in terms of a Turing machine, a problem is said to be a Decidable problem if there exists a corresponding Turing machine which **halts** on every input with an answer- **yes or no**. It is also important to know that these problems are termed as **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

**Semi- Decidable Problems –**
Semi-Decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the Turing Machine. Such problems are termed as **Turing Recognisable** problems.

**Examples –** We will now consider few important **Decidable problems**:

- Are two **regular** languages L and M **equivalent**?
  We can easily check this by using Set Difference operation.
  L-M =Null and M-L =Null.
  Hence (L-M) U (M-L) = Null, then L,M are equivalent.
- Membership of a CFL?
  We can always find whether a string exists in a given CFL by using an algorithm based on dynamic programming.
- Emptiness of a CFL
  By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

**Undecidable Problems –**
The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

We can understand Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: a^n + b^n = c^n.

If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an **Undecidable**

**Problem**.

**Examples –** These are few important **Undecidable Problems**:

- Whether a CFG generates all the strings or not?
  As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.
- Whether two CFG L and M equal?

Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

- Ambiguity of CFG?
  There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.
- Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?
  It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.
- Is a language Learning which is a CFL, regular?
  This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/automata_theory/turing_machine_halting_problem.htm

https://www.geeksforgeeks.org/decidable-and-undecidable-problems-in-theory-of-computation/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(352-353)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-67**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Problems about Turing machine

**Introduction : ( Maximum 5 sentences)** :

- A **problem** is decidable if we can construct a **Turing machine** which will halt in finite amount of time for every input and give answer as 'yes' or 'no'.
- A decidable **problem** has an algorithm to determine the answer for a given input.

**Prerequisite knowledge for Complete understanding and learning of Topic:**

**( Max. Four important topics)**

- Turing Machine

**Detailed content of the Lecture:**

A Turing machine is a computing device that on input a finite sequence of strings might produce a string. The machine has two parts: a control unit and a tape. The control unit is characterized by a finite set Q of control states, a start state $q0 \in Q$ and a transition function $\delta$. The tape is a one-way infinite sequence of cells and each cell contains a symbol from a tape alphabet $\Gamma$. In $\Gamma$ there is a special symbol t, called the blank. The control unit is always located above a cell of the tape and it can move left or right. A computation is performed as follows. Initially the control unit is located at the leftmost cell of the tape and it is in the start stateq0. All cells of the tape are blank except at the beginning of the tape, where the input strings are written, separated by blanks.

During the computation, the control unit transforms the tape step by step. At each step, depending on the current state and the symbol below the control head, the transition function, specifies.

- which symbol is written in the cell below the computation head,
- whether the computation head moves one cell left or right,
- the new state of the computation head.

## Undecidability and Reducibility

**Decidable Problems**

A problem is decidable if we can construct a Turing machine which will halt in finite amount of time for every input and give answer as 'yes' or 'no'. A decidable problem has an algorithm to determine the answer for a given input.

**Examples**

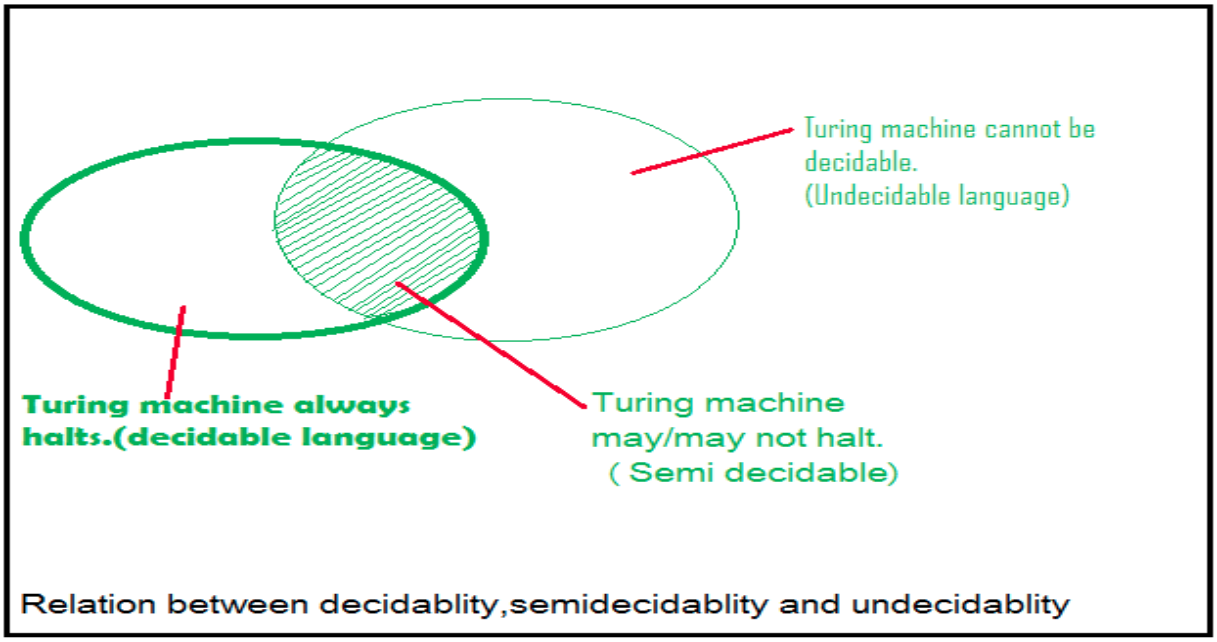
- **Equivalence of two regular languages:** Given two regular languages, there is an algorithm and Turing machine to decide whether two regular languages are equal or not.
- **Finiteness of regular language:** Given a regular language, there is an algorithm and Turing machine to decide whether regular language is finite or not.
- **Emptiness of context free language:** Given a context free language, there is an algorithm whether CFL is empty or not.

**Undecidable Problems**

A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.

**Examples**

- **Ambiguity of context-free languages:** Given a context-free language, there is no Turing machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.
- **Equivalence of two context-free languages:** Given two context-free languages, there is no Turing machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.
- **Everything or completeness of CFG:** Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet (∑*)is undecidable.
- **Regularity of CFL, CSL, REC and REC:** Given a CFL, CSL, REC or REC, determining whether this language is regular is undecidable.



Relation between decidablity,semidecidablity and undecidablity

**Rice's Theorem**
Every non-trivial (answer is not known) problem on Recursive Enumerable languages is undecidable.e.g.; If a language is Recursive Enumerable, its complement will be recursive enumerable or not is undecidable.

**Reducibility and Undecidability**
Language A is reducible to language B (represented as $A \leq B$) if there exists a function f which will convert strings in A to strings in B as:

$$w \in A \iff f(w) \in B$$

Theorem 1: If $A \leq B$ and B is decidable then A is also decidable.
Theorem 2: If $A \leq B$ and A is undecidable then B is also undecidable.

**Question: Which of the following is/are undecidable?**

1. G is a CFG. Is $L(G)=\phi$?
2. G is a CFG. Is $L(G)=\sum^*$?
3. M is a Turing machine. Is L(M) regular?
4. A is a DFA and N is an NFA. Is L(A)=L(N)?

A. 3 only
B. 3 and 4 only
C. 1, 2 and 3 only
D. 2 and 3 only

**Explanation:**

- Option 1 is whether a CFG is empty or not, this problem is decidable.
- Option 2 is whether a CFG will generate all possible strings (everything or completeness of CFG), this problem is undecidable.
- Option 3 is whether language generated by TM is regular is undecidable.
- Option 4 is whether language generated by DFA and NFA are same is decidable. So option D is correct.

**Question: Which of the following problems are decidable?**

1. Does a given program ever produce an output?
2. If L is context free language then L' is also context free?
3. If L is regular language then L' is also regular?
4. If L is recursive language then L' is also recursive?

A. 1,2,3,4
B. 1,2
C. 2,3,4
D. 3,4

**Explanation:**

- As regular and recursive languages are closed under complementation, option 3 and 4 are decidable problems.
- Context free languages are not closed under complementation, option 2 is undecidable.
- Option 1 is also undecidable as there is no TM to determine whether a given program will produce an output. **So, option D is correct.**

Some more **Undecidable Problems** related to Turing machine:

- **Membership** problem of a Turing Machine?
- **Finiteness** of a Turing Machine?
- **Emptiness** of a Turing Machine?
- Whether the language accepted by Turing Machine is regular or CFL?

**Problem:**

**Given two regular languages L1 and L2, is the problem of finding whether a string 'w' exists in both L1 and L2, a decidable problem or not.**

First we make two Turing machines TM1 and TM2 which simulate the DFAs of languages L1 and L2 respectively. We know that a DFA always halts, so a Turing machine simulating a DFA will also always halt. We enter the string 'w' in TM1 and TM2. Both Turing machines will halt and give us an answer. We can connect the outputs of the Turing machines to a modified 'AND' gate which will output 'yes' only when both the Turing machines answer 'yes'. Otherwise it will output 'no'.
Since this system of two Turing machines and a modified AND gate will always stop, this problem is a decidable problem.

There are a lot of questions on this topic. There is no universal algorithm to solve them. Most of the questions require unique and ingenious proofs. Here is where experience is needed. By solving a lot of these problems, one can become very quick in coming up with proofs for these problems on the spot. So, keep practicing.

*The words 'language' and 'problem' can be used synonymously in Theory of computation. For eg. The 'Halting problem' can also be written as 'L = {<M, w> | Turing machine 'M' halts on input 'w'}'. Here 'L' is a language.

**Video Content / Details of website for further learning (if any):**
https://www.geeksforgeeks.org/undecidability-and-reducibility-in-toc/
https://www.geeksforgeeks.org/decidable-and-undecidable-problems-in-theory-of-computation/

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(353-355)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-68**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Recursive and recursively enumerable languages

**Introduction : ( Maximum 5 sentences) :**

- A **problem** is decidable if we can construct a **Turing machine** which will halt in finite amount of time for every input and give answer as 'yes' or 'no'.
- A decidable **problem** has an algorithm to determine the answer for a given input.

Prerequisite knowledge for Complete understanding and learning of Topic:

( Max. Four important topics)

- Turing machines

**Detailed content of the Lecture:**

**Recursive Language**

A language L is recursive (decidable) if L is the set of strings accepted by some Turing Machine (TM) that halts on every input.

**Example**

When a Turing machine reaches a final state, it halts. We can also say that a Turing machine M halts when M reaches a state q and a current symbol 'a' to be scanned so that $\delta(q, a)$ is undefined.

There are TMs that never halt on some inputs in any one of these ways, So we make a distinction between the languages accepted by a TM that halts on all input strings and a TM that never halts on some input strings.

**Recursive Enumerable Language**

A language L is recursively enumerable if L is the set of strings accepted by some TM.

If L is a recursive enumerable language then −

If $w \in L$ then a TM halts in a final state,

If $w \notin L$ then a TM halts in a non-final state or loops forever.

If L is a recursive language then −

If w ∈ L then a TM halts in a final state,

If w ∉ L then TM halts in a non-final state.

**Recursive Languages are also recursive enumerable**

Proof − If L is a recursive then there is TM which decides a member in language then −

- M accepts x if x is in language L.
- M rejects on x if x is not in language L.

According to the definition, M can recognize the strings in language that are accepted on those strings.

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/what-is-a-recursive-and-recursively-enumerable-language

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(301-304)

**Course Teacher**

**Verified by HOD**

# MUTHAYAMMAL ENGINEERING COLLEGE

(An Autonomous Institution)

(Approved by AICTE, New Delhi, Accredited by NAAC & Affiliated to Anna University)

Estd. 2000

Rasipuram - 637 408, Namakkal Dist., Tamil Nadu

**LECTURE HANDOUTS**

**L-69**

**CSE**

**II/ IV/A**

**Course Name with Code** : **Theory of Computation -16CSD11**

**Course Teacher** :**K.PAPITHASRI**

**Unit** : **V Turing machines** **Date of Lecture:**

**Topic of Lecture:** Turing machine as an enumerator

**Introduction : ( Maximum 5 sentences) :**

- A **problem** is decidable if we can construct a **Turing machine** which will halt in finite amount of time for every input and give answer as 'yes' or 'no'.
- A decidable **problem** has an algorithm to determine the answer for a given input.

Prerequisite knowledge for Complete understanding and learning of Topic:

( Max. Four important topics)

- Turing machines

**Detailed content of the Lecture:**

An **enumerator** is an automaton that lists, possibly with repetitions, elements of some set *S*, which it is said to enumerate. A set enumerated by some enumerator is said to be recursively enumerable. An enumerator is equivalent to a Turing machine.

Formal definition

An enumerator is usually represented as a 2-tape Turing machine. One working tape, and one print tape. It can be defined by a 7-tuple, following the notation of a Turing machine: where

- It is a finite, non-empty set of *states*.
- It is a finite, non-empty set of the *output / print alphabet*
- It is a finite, non-empty set of the *working tape alphabet*

- It is the transition function.
- It is the start state
- It is the print state.

- It is the reject state with

Equivalence of Enumerator and Turing Machines

A language over a finite alphabet is Turing Recognizable if and only if it can be enumerated by an enumerator. This shows Turing recognizable languages are also recursively enumerable.

**An Enumerable Language is Turing Recognizable**

It's very easy to construct a Turing Machine that recognizes the enumerable language . We can have two tapes. On one tape we take the input string and on the other tape, we run the enumerator to enumerate the strings in the language one after another. Once a string is printed in the second tape we compare it with the input in the first tape. If its a match, then we accept the input, else reject.

.

**Video Content / Details of website for further learning (if any):**

https://www.tutorialspoint.com/what-is-a-recursive-and-recursively-enumerable-language

**Important Books/Journals for further learning including the page nos.:**

Hopcroft J.E., Motwani R. and Ullman J.D,Introduction to Automata Theory, Languages and Computations ,Hall of India, Pearson Education Second . T1(301-304)

**Course Teacher**

**Verified by HOD**