BCA-EH
Formal Languages and Automata

School of Computer Science & IT
Uttarakhand Open University,
Haldwani

Uttarakhand Open University

Behind Transport Nagar, Vishwavidyalaya Marg,
Haldwani (Nainital) 263139 Uttarakhand

Toll Free : 1800 180 4025 (10 AM to 5 PM) | (Mon to Sat)

Operator : 05946-286000

Admissions : 05946286002

Book Distribution Unit : 05946-286001

Exam Section : 05946-286022

Website : http://uou.ac.in

Content

UNIT-IV

MINIMIZATION OF DETERMINISTIC FINITE AUTOMATA (DFA)

Block-II

UNIT-V

PUSHDOWN AUTOMATA

UNIT-XII

CHOMSKY HIERARCHY

Title	Formal Languages and Automata
Authors	
Adaption and Typesetting	Dr. Ashutosh Kumar Bhatt Associate Professor School of Computer Science and IT Uttarakhand Open University
ISBN:	
Acknowledgement	
This textbook has been adapted from " National Programme on Technology Enhanced Learning (NPTEL)" available at https://nptel.ac.in/courses/106/103/106103070/	
Published By: Uttarakhand Open University	

Block-I

UNIT-I INTODUCTION TO FORMAL LANGUAGES AND AUTOMATA

1.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Alphabets Strings and Languages, languages and automata and grammar. We also understand String operation, language concatenation and Kleene's Star operation.

1.2 ALPHABETS, STRINGS AND LANGUAGES

1.2.1 LANGUAGES :

A general definition of language must cover a variety of distinct categories: natural languages, programming languages, mathematical languages, etc. The notion of natural languages like English, Hindi, etc. is familiar to us. Informally, language can be defined as a system suitable for expression of certain ideas, facts, or concepts, which includes a set of symbols and rules to manipulate these. 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. We start with some basic concepts and definitions required in this regard.

1.2.2 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. Usually, characters from a typical keyboard are only used as symbols.

1.2.3 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. Σ_1, Σ_2 etc) or sometimes other symbol like G may also be introduced.

Example :

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

$\Sigma = \{a, b, c\}$

$\Sigma = \{a, b, c, \&, z\}$

$\Sigma = \{\#, \nabla, \spadesuit, \beta\}$

1.2.4 STRINGS OR WORDS OVER ALPHABET :

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

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.

1.2.5 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

It is convenient to introduce a notation e for the empty string, which contains no symbols at all. The length of the empty string e is zero, i.e., | e | = 0.

1.2.6 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, $a, b, c \in \Sigma$ (symbols) and u, v, w, x, y, z are strings.

1.2.7 SOME STRING OPERATIONS :

Let $x = a_1a_2a_3 \in a_n$ and $y = b_1b_2b_3 \in b_m$ be two strings. The concatenation of x and y denoted by xy, is the string $a_1a_2a_3 \cdots a_nb_1b_2b_3 \cdots b_m$. That is, the concatenation of x and y denoted by xy is the string that has a copy of x followed by a copy of y without any intervening space between them.

Example : Concatenation of the strings 0110 and 11 is 011011 and concatenation of the strings good and boy is goodboy.

Note that for any string w, we = ew = w. It is also obvious that if | x | = n and | y | = m, then | x + y | = n + m.

u is a prefix of v if v = ux for some string x.
u is a suffix of v if v = xu for some string x.
u is a substring of v if v = xuy for some strings x and y.

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

Prefixes: e, 0, 01, 011.
Suffixes: e, 1, 11, 011.
Substrings: e, 0, 1, 01, 11, 011.

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

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

1.2.8 POWERS OF STRINGS :

For any string x and integer $n \geq 0$, we use x^n to denote the string formed by sequentially concatenating n copies of x. We can also give an inductive definition of x^n as follows: x^n = e, if n = 0 ; otherwise $x^n = xx^{n-1}$

Example : If x = 011, then x^3 = 011011011, x^1 = 011 and $x^0 = e$

1.2.9 POWERS OF ALPHABETS :

We write Σ^k (for some integer k) to denote the set of strings of length k with symbols from Σ. In other words,

Σ^k = { w | w is a string over Σ and | w | = k}. Hence, for any alphabet, Σ^0 denotes the set of all strings of length zero. That is, Σ^0 = { e }. For the binary alphabet { 0, 1 } we have the following.

$\Sigma^0 = \{e\}$.

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

$\Sigma^2 = \{00, 01, 10, 11\}$.

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet Σ is denoted by Σ^*. That is,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

The set Σ^* contains all the strings that can be generated by iteratively concatenating symbols from Σ any number of times.

Example : If Σ = { a, b }, then Σ^* = { e, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}.

Please note that if $\Sigma = F$, then Σ^* that is $\phi^* = \{e\}$. It may look odd that one can proceed from the empty set to a non-empty set by iterated concatenation. But there is a reason for this and we accept this convention.

The set of all nonempty strings over an alphabet Σ is denoted by Σ^+. That is,

$$\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \Sigma^n \cup \cdots$$
$$= \cup \Sigma^k$$

Note that Σ^* is infinite. It contains no infinite strings but strings of arbitrary lengths.

1.2.10 REVERSAL :

For any string $w = a_1 a_2 a_3 \cdots a_n$ the reversal of the string is $w^R = a_n a_{n-1} \cdots a_3 a_2 a_1$.

An inductive definition of reversal can be given as follows:

CHECK YOUR PROGRESS

True/False type questions

1) A grammar is a mechanism used for describing languages._____

2) The Kleene star operation on a language L, denoted as L*._____

3) The transition from one configuration to the next (as defined by the transition function) is called a turn._____

4) The most important feature of the automaton is its control unit._____

5) To concatenate to language L1 and L2 is defined as L1+L2_____

Answers-

1) True

2) True

3) False

4) True

5) False

1.3 LANGUAGE :

A language over an alphabet is a set of strings over that alphabet. Therefore, a language L is any subset of Σ^*. That is, any $L \subseteq \Sigma^*$ is a language.

Example :

1. F is the empty language.

2. Σ^* is a language for any Σ.

3. {e} is a language for any Σ. Note that, $\phi \neq \{e\}$. Because the language F does not contain any string but {e} contains one string of length zero.

4. The set of all strings over { 0, 1 } containing equal number of 0's and 1's.

5. The set of all strings over {a, b, c} that starts with a.

Convention : Capital letters A, B, C, L, etc. with or without subscripts are normally used to denote languages.

1.3.1 SET OPERATIONS ON LANGUAGES :

Since languages are set of strings we can apply set operations to languages. Here are some simple examples (though there is nothing new in it).

Union : A string $x \in L_1 \cup L_2$ iff $x \in L_1$ or $x \in L_2$

Example : { 0, 11, 01, 011 } $\cup$ { 1, 01, 110 } = { 0, 11, 01, 011, 111 }

Intersection : A string $x \in L_1 \cap L_2$ iff $x \in L_1$ and $x \in L_2$.

Example : { 0, 11, 01, 011 } $\cap$ { 1, 01, 110 } = { 01 }

Complement : Usually, Σ^* 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 { $x \in \Sigma^*$ | |x| is odd }.
Similarly we can define other usual set operations on languages like relative complement, symmetric difference, etc.

1.3.2 REVERSAL OF A LANGUAGE :

The reversal of a language L, denoted as L^R, is defined as: $L^R = \{w^R \mid w \in L\}$.

Example :

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

2. Let L = { 1^n0^n | n is an integer }. Then L^R = { 1^n0^n | n is an integer }.

1.3.3 LANGUAGE CONCATENATION :

The concatenation of languages L_1 and L_2 is defined as

$L_1 L_2$ = { xy | $x \in L_1$ and $y \in L_2$ }.

Example : { a, ab }{ b, ba } = { ab, aba, abb, abba }.

Note that ,

1. $L_1 L_2 \neq L_2 L_1$ in general.

2. $L\Phi = \Phi$

3. $L\{\varepsilon\} = L = \{\varepsilon\}$

1.3.4 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 L^n denotes the concatenation of L with itself n times. This is defined formally as follows:

$L_0 = \{e\}$

$L^n = LL^{n-1}$

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

$L_0 = \{e\}$

$L_1 = L\{e\} = L = \{a, ab\}$

$L_2 = L L_1 = \{a, ab\}\{a, ab\} = \{aa, aab, aba, abab\}$

$L_3 = L L_2 = \{a, ab\}\{aa, aab, aba, abab\}$

$= \{aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab\}$ and so on.

1.3.5 KLEENE'S STAR OPERATION :

The Kleene star operation on a language L, denoted as L^* is defined as follows :

L^* = (Union n in N) L^n

$= L^0 \cup L^1 \cup L^2 \cup \cdots$

= { x | x is the concatenation of zero or more strings from L }

Thus L^* is the set of all strings derivable by any number of concatenations of strings in L. It is also useful to define

$L^+ = LL^*$, i.e., all strings derivable by one or more concatenations of strings in L. That is

L^+ = (Union n in N and n >0) L^n

$= L^1 \cup L^2 \cup L^3 \cup \cdots$

Example : Let L = { a, ab }. Then we have,

$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$

= {e} $\cup$ {a, ab} $\cup$ {aa, aab, aba, abab} $\cup \ldots$

$L^+ = L^1 \cup L^2 \cup L^3 \cup \cdots$

= {a, ab} $\cup$ {aa, aab, aba, abab} $\cup \ldots$

Note : e is in L^*, for every language L, including .

The previously introduced definition of Σ^* is an instance of Kleene star.

1.4 AUTOMATA AND GRAMMARS

- 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.

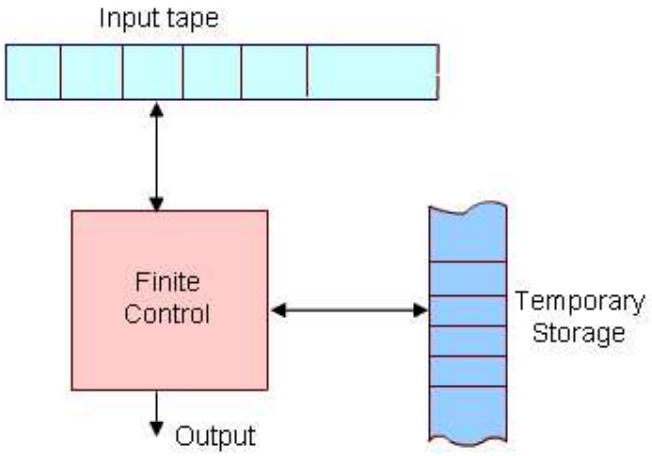


Figure 1: The figure above shows a diagrammatic representation of a generic automation.

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.

1.4.1 GRAMMAR

A grammar is a mechanism used for describing languages. This is one of the most simple but yet powerful mechanism. There are other notions to do the same, of course.

In everyday language, like English, we have a set of symbols (alphabet), a set of words constructed from these symbols, and a set of rules using which we can group the words to construct meaningful sentences. The grammar for English tells us what are the words in it and the rules to construct sentences. It also tells us whether a particular sentence is well-formed (as per the grammar) or not. But even if one follows the rules of the english grammar it may lead to some sentences which are not meaningful at all, because of impreciseness and ambiguities involved in the language. In english grammar we use many other higher level constructs like noun-phrase, verb-phrase, article, noun, predicate, verb etc. A typical rule can be defined as

< sentence > $\rightarrow$ < noun-phrase >< predicate >

meaning that "a sentence can be constructed using a 'noun-phrase' followed by a predicate".

Some more rules are as follows:

< noun-phrase > $\rightarrow$ < article >< noun >

< predicate > $\rightarrow$ < verb >

with similar kind of interpretation given above.

If we take {a, an, the} to be <article>; cow, bird, boy, Ram, pen to be examples of <noun>; and eats, runs, swims, walks, are associated with <verb>, then we can construct the sentence-

a cow runs, the boy eats, an pen walks- using the above rules. Even though all sentences are well-formed, the last one is not meaningful. We observe that we start with the higher level construct <sentence> and then reduce it to <noun-phrase>, <article>, <noun>, <verb> successively, eventually leading to a group of words associated with these constructs.

These concepts are generalized in formal language leading to formal grammars. The word 'formal' here refers to the fact that the specified rules for the language are explicitly stated in terms of what strings or symbols can occur. There can be no ambiguity in it.

Formal definitions of a Grammar

A grammar G is defined as a quadruple.

$G = (N, \Sigma, P, S)$

N is a non-empty finite set of non-terminals or variables,

Σ is a non-empty finite set of terminal symbols such that $N \cap \Sigma = \phi$

$S \in N$, is a special non-terminal (or variable) called the start symbol, and $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ is a finite set of production rules.

The binary relation defined by the set of production rules is denoted by $\rightarrow$, i.e. $\alpha \rightarrow \beta$ iff $(\alpha, \beta) \in P$.

In other words, P is a finite set of production rules of the form $\alpha \rightarrow \beta$, where $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^*$

The production rules specify how the grammar transforms one string to another. Given a string $\delta\alpha\gamma$, we say that the production rule $\alpha \rightarrow \beta$ is applicable to this string, since it is possible to use the rule $\alpha \rightarrow \beta$ to rewrite the α (in $\delta\alpha\gamma$) to β obtaining a new string $\delta\beta\gamma$. We say that $\delta\alpha\gamma$ derives $\delta\beta\gamma$ and is denoted as

$\delta\alpha\gamma \Rightarrow \delta\beta\gamma$

Successive strings are dervied by applying the productions rules of the grammar in any arbitrary order. A particular rule can be used if it is applicable, and it can be applied as many times as described.

We write $\alpha \overset{*}{\Rightarrow} \beta$ if the string β can be derived from the string α in zero or more steps; $\alpha \overset{+}{\Rightarrow} \beta$ if β can be derived from α in one or more steps.

By applying the production rules in arbitrary order, any given grammar can generate many strings of terminal symbols starting with the special start symbol, S, of the grammar. The set of all such terminal strings is called the language generated (or defined) by the grammar.

Formaly, for a given grammar $G=(N,\Sigma,P,S)$ the language generated by G is

$$L(G)=\left\{w\in\Sigma^{*}\mid S\overset{*}{\Rightarrow}w\right\}$$

That is $w\in L(G)$ iff $S\overset{*}{\Rightarrow}w$.

If $w\in L(G)$, we must have for some $n\geq 0$, $S=\alpha_1\Rightarrow\alpha_2\Rightarrow\alpha_3\Rightarrow\cdots\Rightarrow\alpha_n=w$, denoted as a derivation sequence of w, The strings $S=\alpha_1,\alpha_2,\alpha_3,\cdots,\alpha_n=w$ are denoted as sentential forms of the derivation.

Example : Consider the grammar $G=(N,\Sigma,P,S)$, where N = {S}, Σ={a, b} and P is the set of the following production rules

{ S $\rightarrow$ ab, S $\rightarrow$ aSb}

Some terminal strings generated by this grammar together with their derivation is given below.

S $\Rightarrow$ ab

S $\Rightarrow$ aSb $\Rightarrow$ aabb

S $\Rightarrow$ aSb $\Rightarrow$ aaSbb $\Rightarrow$ aaabbb

It is easy to prove that the language generated by this grammar is

$$L(G)=\left\{a^{i}b^{i}\mid i\geq 1\right\}$$

By using the first production, it generates the string ab (for i =1).

To generate any other string, it needs to start with the production S $\rightarrow$ aSb and then the non-terminal S in the RHS can be replaced either by ab (in which we get the string aabb) or the same production S $\rightarrow$ aSb can be used one or more times. Every time it adds an 'a' to the left and a 'b' to the right of S, thus giving the sentential form $a^{i}Sb^{i},\ i\geq 1$. When the non-terminal is replaced by ab (which is then only possibility for generating a terminal string) we get a terminal string of the form $a^{i}b^{i},\ i\geq 1$.

There is no general rule for finding a grammar for a given language. For many languages we can devise grammars and there are many languages for which we cannot find any grammar.

Example: Find a grammar for the language $L=\left\{a^{n}b^{n+1}\mid n\geq 1\right\}$.

It is possible to find a grammar for L by modifying the previous grammar since we need to generate an extra b at the end of the string $a^{n}b^{n},\ n\geq 1$. We can do this by adding a production S $\rightarrow$ Bb where the non-terminal B generates as given in the previous example.

Using the above concept we devise the follwoing grammar for L.

$G = (N, \Sigma, P, S)$ where

N = { S, B }

P = { S $\rightarrow$ Bb, B $\rightarrow$ ab, B $\rightarrow$ aBb }

1.5 CHECK YOUR PROGRESS

Fill in the blanks

1) An alphabet is a _____ nonempty set of symbols.

2) A grammar is a mechanism used for describing _____.

3) For any string $w = a_1 a_2 a_3 \cdots a_n$ the reversal of the string is_____. .

4) _____ is the simplest type of abstract machine we consider.

5) The transition from one configuration to the next (as defined by the transition function) is called a_____

1.6 ANSWER CHECK YOUR PROGRESS

1) Finite

2) Language.

3) $w^R = a_n a_{n-1} \cdots a_3 a_2 a_1$

4) Finite automation.

5) Move

1.7 MODEL QUESTION

Qs-1) What do you understand by Languages and Theory of Computations? What is the most important feature of Automation?

Qs-2) What is move?

Qs-3) What is Grammar? Give the Formal definitions of a Grammar? How will you find grammar for a language?

Qs-4) What are Symbols? Symbols are indivisible objects or entity that cannot be defined. Explain How.

Qs-5) What is Language? Explain about the natural languages, programming languages, mathematical languages?

1.8 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

1.9 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH.

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI.

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons.

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM.

UNIT-II FINITE AUTOMATA

2.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Finite Automata, Finite State Automaton, Multiple Next State, Formal definition of NFA and the Language of an NFA. We also understand States, Transitions and Finite-State Transition System.

2.2 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. The concept of a finite automaton appears to have arisen in the 1943 paper “A logical calculus of the ideas immanent in nervous activity", by Warren McCullock and Walter Pitts. In 1951 Kleene introduced regular expressions to describe the behaviour of finite automata. He also proved the important theorem saying that regular expressions exactly capture the behaviours of finite automata. In 1959, Dana Scott and Michael Rabin introduced non-deterministic automata and showed the surprising theorem that they are equivalent to deterministic automata. We will study these fundamental results. Since those early years, the study of automata has continued to grow, showing that they are indeed a fundamental idea in computing.

2.2.1 STATES, TRANSITIONS AND FINITE-STATE TRANSITION SYSTEM :

Let us first gives 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.

We said that automata are a model of computation. That means that they are a simplified abstraction of `the real thing'. So what gets abstracted away? One thing that disappears is any notion of hardware or software. We merely deal with states and transitions between states. The distinction between program and machine executing it disappears. One could say that an automaton is the machine and the program. This makes automata relatively easy to implement in either hardware or software. From the point of view of resource consumption,

the essence of a finite automaton is that it is a strictly finite model of computation. Everything in it is of a fixed, finite size and cannot be modified in the course of the computation.

2.2.2 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.

Thus, a DFA conceptually consists of 3 parts:

A tape to hold the input string. The tape is divided into a finite number of cells. Each cell holds a symbol from Σ.

A tape head for reading symbols from the tape

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:

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.

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:

2.3 DETERMINISTIC FINITE STATE AUTOMATON :

A Deterministic Finite State Automaton (DFA) is a 5-tuple : $M = (Q, \Sigma, \delta, q_0, F)$

Q is a finite set of states.

Σ is a finite set of input symbols or alphabet.

$\delta : Q \times \Sigma \to Q$ is the "next state" transition function (which is total). Intuitively, δ is a function that tells which state to move to in response to an input, i.e., if M is in state q and sees input a, it moves to state $\delta(q,a)$.

$q_0 \in Q$ is the start state.

$F \subseteq Q$ is the set of accept or final states.

2.3.1 ACCEPTANCE OF STRINGS :

A DFA accepts a string $w = a_1 a_2 \cdots a_n$ if there is a sequence of states $q_0, q_1, \cdots, q_n$ in Q such that

1. q_0 is the start state.
2. $\delta(q_i, q_{i+1}) = a_{i+1}$ for all $0 < i < n$.
3. $q_n \in F$.

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

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

2.3.3 EXTENDED TRANSITION FUNCTION :

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

That is, $\delta^*(q,w)$ 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 accepts w }

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

Example 1 :

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

$Q = \{q_0, q_1\}$

q_0 is the start state

$F = \{q_1\}$

$\delta(q_0, 0) = q_0 \qquad \delta(q_1, 0) = q_1$

$\delta(q_0, 1) = q_1 \qquad \delta(q_1, 1) = q_1$

It is a formal description of a DFA. But it is hard to comprehend. For ex. The language of the DFA is any string over { 0, 1} having at least one 1.

We can describe the same DFA by transition table or state transition diagram as following

TRANSITION TABLE :

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

It is easy to comprehend the transition diagram.

Explanation : We cannot reach find state q_1 w/0 or in the i/p string. There can be any no. of 0's at the beginning. (The self-loop at q_0 on label 0 indicates it). Similarly there can be any no. of 0's & 1's in any order at the end of the string.

2.3.4 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").

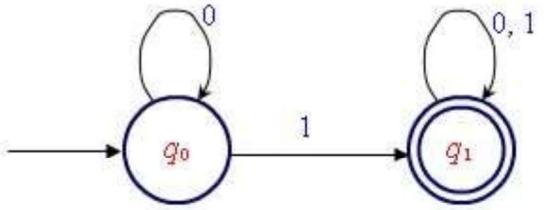
- Rows correspond to states,
- Columns correspond to input symbols,
- Entries correspond to next states
- The start state is marked with an arrow
- The accept states are marked with a star (*).

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

2.3.5 (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, δ . 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 $r \notin F$. Note that there is no formal mechanism for moving the pointer.
6. A language $L \in \Sigma^*$ is said to be regular if $L = L(M)$ for some DFA *M*

2.3.6 REMOVING ∈ TRANSITION

∈- transitions do not increase the power of an *NFA* . That is, any ∈ -*NFA* (*NFA* with ∈ transition), we can always construct an equivalent *NFA* without ∈-transitions. The equivalent *NFA* must keep track where the ∈ *NFA* goes at every step during computation. This can be done by adding extra transitions for removal of every ∈ - transitions from the ∈- *NFA* as follows.

If we removed the ∈ - transition $\delta(p, \in) = q$ from the ∈- *NFA* , then we need to moves from state *p* to all the state γ on input symbol $q \in \Sigma$ which are reachable from state q (in the ∈ - *NFA*) on same input symbol *q*. This will allow the modified *NFA* to move from state *p* to all states on some input symbols which were possible in case of ∈-*NFA* on the same input symbol. This process is stated formally in the following theories.

Theorem if *L* is accepted by an ∈ - *NFA* *N* , then there is some equivalent $NFA\ N'$ without ∈ transitions accepting the same language *L*

Proof:

Let $N = (Q, \Sigma, \delta, q_0, F)$ be the given $\in -NFA$ with

We construct $N' = (Q, \Sigma, \delta', q_0, F')$

Where, $\delta'(q, a) = \left\{ p \middle| p \in \hat{\delta}(q, a) \right\}$ for all $q \in Q$, and $a \in \Sigma$, and

$F' = \begin{cases} F \cup \{q_0\} & \text{if } \hat{\delta}(q_0, \in) \cap F \neq \phi \\ F & \text{otherwise.} \end{cases}$

Other elements of *N′* and *N*

We can show that $L(N) = L(N')$ *i.e.* N' *and* N *are equivalent.*

We need to prove that $\forall w \in \Sigma^*$

$w \in L(N)$ iff $w \in L(N')$ *i.e.*

$\forall w \in \Sigma^*$ $\hat{\delta}'(q_0, w) \in F'$ iff $\hat{\delta}(q_0, w) \in F$

We will show something more, that is,

$$\forall w \in \Sigma^* \quad \hat{\delta}'(q_0, w) = \hat{\delta}(q_0, w)$$

We will show something more, that is, $|w|$

Basis : $|w| = 1$ *, then* $x = a \in \Sigma$

But $\hat{\delta}'(q_0, a) = \hat{\delta}(q_0, a)$ *by definition of* δ'.

Induction hypothesis Let the statement hold for all $w \in \Sigma^*$ *with* $|w| \leq n$.

$$\begin{aligned}
\hat{\delta}'(q_0, w) &= \hat{\delta}'(q_0, xa) && \text{By definition of extension of } \hat{\delta}' \\
&= \delta'\left(\hat{\delta}'(q_0, x), a\right) && \text{By inductions hypothesis.} \\
&= \delta'\left(\hat{\delta}(q_0, x), a\right) && \text{Assuming that} \\
&= \delta'(R, a) && \hat{\delta}(q_0, x) = R, \text{where } R \subseteq Q \\
&= \bigcup_{p \in R} \delta'(p, a) \\
&= \bigcup_{p \in R} \hat{\delta}(p, a) && \text{By definition of } \delta' \\
&= \hat{\delta}(q_0, xa) \\
&= \hat{\delta}(q_0, w) && \text{Since } R = \hat{\delta}(q_0, x)
\end{aligned}$$

To complete the proof we consider the case

When $|w| = 0$ i.e. $w = \in$ then

$\delta'(q_0, \in) = \{q_0\}$ and by the construction of F', $q_0 \in F'$ wherever $\hat{\delta}(q_0, \in)$ constrains a state in *F*.

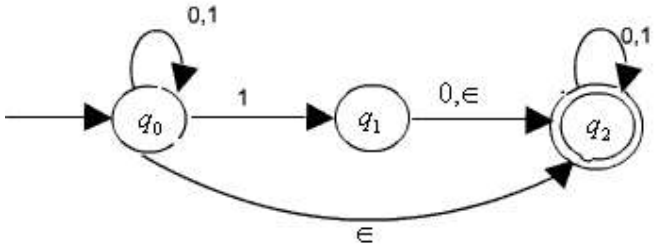
If $F' = F$ (and thus $\hat{\delta}(q_0, \in)$ is not in *F*), then $\forall w$ with $|w| = 1$, w leads to an accepting state in *N′* iff it leads to an accepting state in *N* (by the construction of *N′* and *N*).

Also, if ($w = \in$, thus *w* is accepted by N' iff *w* is accepted by *N* (iff $q_0 \in F$)

If $F' = F \cup \{q_0\}$ (and, thus in *M* we load $\hat{\delta}(q_0, \in)$ in *F*), thus $\in$ is accepted by both *N′* and *N* .

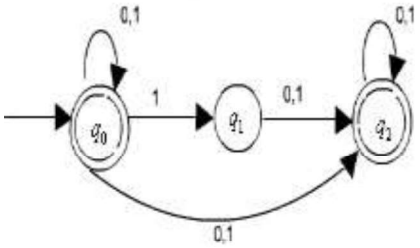
Let $|w| \geq 1$. If *w* cannot lead to q_0 in *N* , then $w \in L(N)$. (Since can add $\in$ transitions to get an accept state). So there is no harm in making q_0 an accept state in *N′*.

Ex: Consider the following *NFA* with $\in$- transition.



Transition table δ

	0	1
$\rightarrow_F q_0$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_2\}$	$\{q_2\}$
$F \quad q_2$	$\{q_2\}$	$\{q_2\}$



Transition table δ' for the equivalent *NFA* without $\in$- moves

Since $\delta(q_0, \in) = q_2 \ \in -NFA$ the start state q_0 must be final state in the equivalent *NFA* .

Since $\delta(q_0, \in) = q_2$ and $\delta(q_2, 0) = q_2$ and $\delta(q_2, 1) = q_2$ we add moves $\delta(q_0, 0) = q_2$ and $\delta(q_0, 1) = q_2$ in the equivalent *NFA* . Other moves are also constructed accordingly.

$\in$ -closures:

The concept used in the above construction can be made more formal by defining the $\in$-closure for a state (or a set of states). The idea of $\in$-closure is that, when moving from a state *p* to a state *q* (or from a set of states S_i to a set of states S_j) an input $a \in \Sigma$, we need to take account of all $\in$-moves that could be made after the transition. Formally, for a given state *q*,

$\in$ -closures: $(q) = \{p \mid p \text{ can be reached from } q \text{ by zero or more } \in\text{-moves}\}$

Similarly, for a given set $R \subseteq Q$

$\in$ -

closures: $(R) = \{p \in Q \mid p \text{ can be reached from any } q \in R \text{ by following zero or more } \in\text{-moves}\}$

So, in the construction of equivalent *NFA N′* without $\in$-transition from any *NFA* with $\in$ moves. the first rule can now be written as $\delta'(q,a) = \in\text{-closure}\left(\delta(q,a)\right)$

2.3.7 EQUIVALENCE OF NFA AND DFA

It is worth noting that a DFA is a special type of NFA and hence the class of languages accepted by DFA s is a subset of the class of languages accepted by NFA s. Surprisingly, these two classes are in fact equal. NFA s appeared to have more power than DFA s because of generality enjoyed in terms of $\in$-transition and multiple next states. But they are no more powerful than DFA s in terms of the languages they accept.

Converting DFA to NFA

Theorem: Every DFA has as equivalent NFA

Proof: A DFA is just a special type of an NFA . In a DFA , the transition functions is defined from $Q\times\Sigma \text{ to } Q$ whereas in case of an NFA it is defined from $Q\times\Sigma \text{ to } 2^Q$ and $D=(Q,\Sigma,\delta,q_0,F)$ be a DFA . We construct an equivalent NFA $N=(Q',\Sigma,\delta',q_0,F)$ as follows.

$\{q_i\}\in Q', \forall q_i \in Q$

$\delta'(\{p\},a)=\{\delta(p,a)\}$, i. e

If $\delta(p,a)=q,$ and $\delta'(\{p\},a)=\{q\}.$

All other elements of N are as in D.

If $w=a_1a_2\cdots,a_n\in L(D)$ then there is a sequence of states $q_0,q_1,q_2\cdots,q_n$ such that $\delta(q_{i-1},a_i)=q_i \text{ and } q_n\in F$

Then it is clear from the above construction of N that there is a sequence of states (in N) $\{q_0\},\{q_1\},\{q_2\},\cdots,\{q_n\}$ such that $\delta'(\{q_{i-1}\},a_i)=\{q_i\}$ and $\{q_n\}\in F$ and hence $w\in L(N).$

Similarly we can show the converse.

Hence , $L(N)=L(D)$

Given any *NFA* we need to construct as equivalent *DFA* i.e. the *DFA* need to simulate the behaviour of the *NFA* . For this, the *DFA* have to keep track of all the states where the NFA could be in at every step during processing a given input string.

There are 2^n possible subsets of states for any *NFA* with *n* states. Every subset corresponds to one of the possibilities that the equivalent *DFA* must keep track of. Thus, the equivalent *DFA* will have 2^n states.

The formal constructions of an equivalent *DFA* for any *NFA* is given below. We first consider an *NFA* without $\in$ transitions and then we incorporate the affects of $\in$ transitions later.

Formal construction of an equivalent *DFA* for a given *NFA* without $\in$ transitions.

Given an $N=(Q,\Sigma,\delta,q_0,F)$ without $\in$- moves, we construct an equivalent *DFA* $D=\left(Q^D,\Sigma,\delta^D,q_0^D,F^D\right)$ as follows

$Q^D=P(Q)$ i.e. $Q^D=\{S|S\subseteq Q\}$,

$q_0^D=\{q_0\}$,

$F^D=\left\{q^D\in Q^D\middle|q^D\cap F\neq\phi\right\}$ (i.e. every subset of Q which as an element in F is considered as a final state in DFA D)

$$\delta^D\left(\{q_1,q_2,\cdots,q_k\},a\right)=\delta(q_1,a)\cup\delta(q_2,a)\cup\cdots\cup\delta(q_k,a)$$

for all $a\in\Sigma$ and $q^D=\{q_1,q_2,\cdots,q_k\}$

where $q_i\in Q,\ 1\le i\le k$.

That is, $\delta^D\left(q^D,a\right)=\bigcup_{q_i\in q^D}\delta(q_i,a)$

To show that this construction works we need to show that L(D)=L(N) i.e.

$\forall w\in\Sigma^*\ \ \hat{\delta}^D\left(q_0^D,w\right)\in F^D$ iff $\hat{\delta}(q_0,w)\cap F\neq\phi$

Or, $\forall w\in\Sigma^*\ \ \hat{\delta}^D\left(\{q_0\},w\right)\cap F\neq$ iff $\hat{\delta}(q_0,w)\cap F\neq\phi$

We will prove the following which is a stranger statement thus required.

$$\forall w\in\Sigma^*,\ \ \hat{\delta}^D\left(\{q_0\},w\right)=\hat{\delta}(q_0,w)$$

Proof : We will show by inductions on $|w|$

Basis If $|w|$=0, then w =$\in$

So, $\delta^D\left(\{q_0\},\in\right)=\{q_0\}=\delta(q_0,\in)$, by definition.

Inductions hypothesis : Assume inductively that the statement holds $\forall w\in\Sigma^*$ of length less than or equal to n.

Inductive step

Let $|w| = n+1$, then $w = xa$ with $|x| = n$ and $a \in \Sigma$.

Now,

$$\begin{aligned}
\hat{\delta}^D(\{q_0\}, w) &= \hat{\delta}^D(\{q_0\}, xa) \\
&= \delta^D\left(\hat{\delta}^D(\{q_0\}, x), a\right), \text{ by inductive extension of } \delta^D \\
&= \delta^D\left(\hat{\delta}(q_0, x), a\right), \text{by induction hypothesis} \\
&= \bigcup_{q_i \in \hat{\delta}(q_0, x)} \delta(q_i, a), \text{ by definition of } \delta^D \\
&= \delta(q_0, xa) \quad \text{by definition of } \hat{\delta} \text{ (extension of } \delta) \\
&= \delta(q_0, w)
\end{aligned}$$

Now, given any NFA with $\in$-transition, we can first construct an equivalent NFA without $\in$ -transition and then use the above construction process to construct an equivalent DFA , thus, proving the equivalence of NFA s and DFA s..

It is also possible to construct an equivalent DFA directly from any given NFA with $\in$-transition by integrating the concept of $\in$ -closure in the above construction.

Recall that, for any $S \subseteq Q$,

$\in$ - closure : $(S) = \{q \in Q \mid q \text{ can be reached from any } p \in S \text{ by following zero or more} \in \text{-transitions}\}$

In the equivalent DFA , at every step, we need to modify the transition functions δ^D to keep track of all the states where the NFA can go on $\in$-transitions. This is done by replacing $\delta(q,a)$ by $\in$-closure $(\delta(q,a))$, i.e. we now compute $\delta^D(q^D, a)$ at every step as follows:

$$\delta^D(q^D, a) = \left\{q \in Q \mid q \in \ \in\text{-closure}\left(\delta(q^D, a)\right)\right\}.$$

Besides this the initial state of the *DFA D* has to be modified to keep track of all the states that can be reached from the initial state of *NFA* on zero or more -transitions. This can be done by changing the initial state q_0^D to $\in$-closure (q_0^D) .

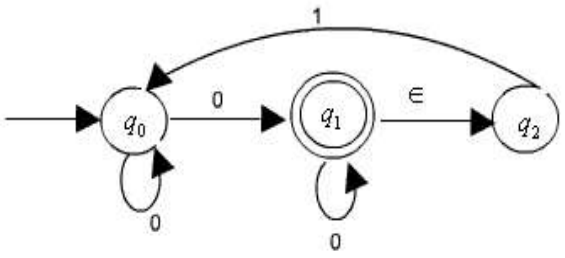
It is clear that, at every step in the processing of an input string by the *DFA D* , it enters a state that corresponds to the subset of states that the *NFA N* could be in at that particular point. This has been proved in the constructions of an equivalent NFA for any $\in$ -*NFA*

If the number of states in the *NFA* is n , then there are 2^n states in the *DFA* . That is, each state in the *DFA* is a subset of state of the *NFA* .

But, it is important to note that most of these 2^n states are inaccessible from the start state and hence can be removed from the *DFA* without changing the accepted language. Thus, in fact, the number of states in the equivalent *DFA* would be much less than 2^n .

Example : Consider the NFA given below.

Transition table



	0	1	∈
$\rightarrow q_0$	$\{q_0, q_1\}$	ϕ	ϕ
$F\ q_1$	$\{q_1\}$	ϕ	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_0\}$

Since there are 3 states in the NFA

There will be $2^3 = 8$ states (representing all possible subset of states) in the equivalent DFA . The transition table of the DFA constructed by using the subset constructions process is produced here.

	0	1
ϕ	ϕ	ϕ
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	ϕ
$F\{q_1\}$	$\{q_1, q_2\}$	$\{q_0\}$
$\{q_2\}$	ϕ	$\{q_0\}$
$F\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$F\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_0\}$
$F\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

The start state of the DFA is $\in$- closures $(q_0) = \{q_0\}$

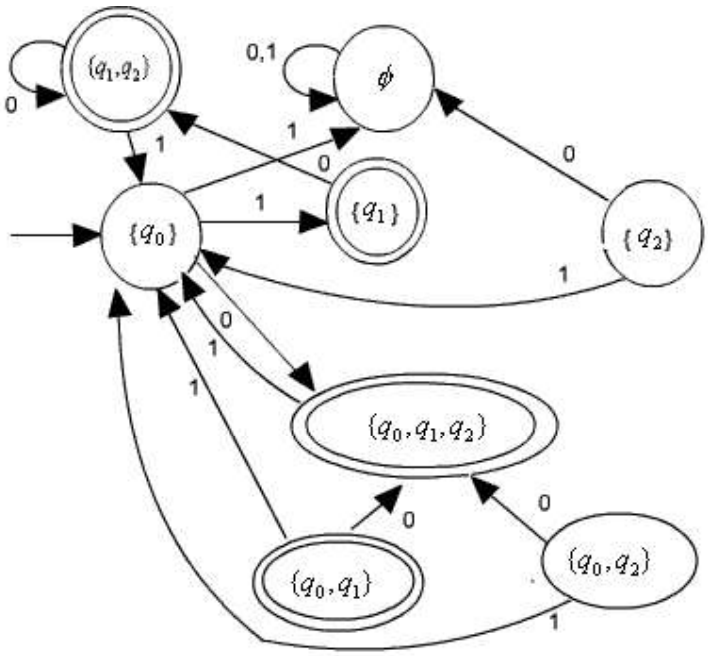
The final states are all those subsets that contains q_1 (since $q_1 \in F$ in the NFA).

Let us compute one entry,

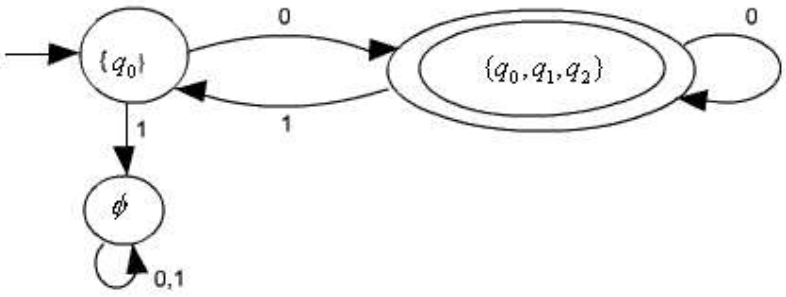
$$\delta^D(\{q_0,0\}) = \in -\text{closure}(\delta(q_0,0))$$
$$= \in -\text{closure}(\{q_0,q_1\})$$
$$= \{q_0,q_1,q_2\}$$

Similarly, all other transitions can be computed.

Corresponding transition fig. for the DFA is shown as



te that states $\{q_1\},\{q_2\},\{q_1,q_2\},\{q_0,q_2\}$ and $\{q_0,q_1\}$ are not accessible and hence can be removed. This gives us the following simplified *DFA* with only 3 states.



It is interesting to note that we can avoid encountering all those inaccessible or unnecessary states in the equivalent *DFA* by performing the following two steps inductively.

1. If q_0 is the start state of the NFA, then make $\in$- closure (q_0) the start state of the equivalent *DFA* . This is definitely the only accessible state.

If we have already computed a set δ of states which are accessible. Then $\forall a \in \Sigma$. compute $\left(\delta^D(S,a)\right)$ because these set of states will also be accessible.

Following these steps in the above example, we get the transition table given below

	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1, q_2\}$	ϕ
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

Non determinism is an important abstraction in computer science. Importance of non determinism is found in the design of algorithms. For examples, there are many problems with efficient nondeterministic solutions but no known efficient deterministic solutions. (Travelling salesman, Hamiltonean cycle, clique, etc). Behaviour of a process is in a distributed system is also a good example of nondeterministic situation. Because the behaviour of a process might depend on some messages from other processes that might arrive at arbitrary times with arbitrary contents.

It is easy to construct and comprehend an NFA than DFA for a given regular language. The concept of NFA can also be used in proving many theorems and results. Hence, it plays an important role in this subject.

In the context of FA nondeterminism can be incorporated naturally. That is, an NFA is defined in the same way as the DFA but with the following two exceptions:

- multiple next state.
- $\in$ - transitions.

CHECK YOUR PROGRESS

True/False type questions

1) Finite-state transition systems can be modeled abstractly by a mathematical model called Finite Automation._____

2) Transitions are changes of states that can occur spontaneously or in response to inputs to the states._____

3) Every DFA has as equivalent NFA_____

4) $\in$- transitions increase the power of an *NFA* ._____

5) A system containing only a finite number of states and transitions among them is called a infinite state transition._____

Answers:

1) True

2) True

3) True

4) False

5) False

2.4 MULTIPLE NEXT STATE :

In contrast to a DFA, the next state is not necessarily uniquely determined by the current state and input symbol in case of an NFA. (Recall that, in a DFA there is exactly one start state and exactly one transition out of every state for each symbol in Σ).

This means that - in a state q and with input symbol a - there could be one, more than one or zero next state to go, i.e. the value of $\delta(q,a)$ is a subset of Q. Thus $\delta(q,a) = \{q_1, q_2, \cdots, q_k\}$ which means that any one of $q_1, q_2, \cdots, q_k$ could be the next state.

The zero next state case is a special one giving $\delta(q,a) = \phi$, which means that there is no next state on input symbol when the automata is in state q. In such a case, we may think that the automata "hangs" and the input will be rejected.

2.4.1 ∈- TRANSITIONS :

In an transition -, the tape head doesn't do anything- it doesnot read and it doesnot move. However, the state of the automata can be changed - that is can go to zero, one or more states. This is written formally as $\delta(q,\in) = \{q_1, q_2, \cdots, q_k\}$ implying that the next state could by any one of $q_1, q_2, \cdots, q_k$ w/o consuming the next input symbol.

2.4.2 ACCEPTANCE :

Informally, an NFA is said to accept its input ω if it is possible to start in some start state and process ω, moving according to the transition rules and making choices along the way whenever the next state is not uniquely defined, such that when ω is completely processed (i.e. end of ω is reached), the automata is in an accept state. There may be several possible paths through the automation in response to an input ω since the start state is not determined and there are choices along the way because of multiple next states. Some of these paths may lead to accpet states while others may not. The automation is said to accept ω if at least one computation path on input ω starting from at least one start state leads to an accept state- otherwise, the automation rejects input ω. Alternatively, we can say that, ω is accepted iff there exists a path with label ω from some start state to some accept state. Since there is no mechanism for determining which state to start in or which of the possible next moves to take (including the ω-transitions) in response to an input symbol we can think that the automation is having some "guessing" power to chose the correct one in case the input is accepted.

Example 1 : Consider the language L = { $\omega \in$ {0, 1}* | The 3rd symbol from the right is 1}. The following four-state automation accepts L.

The m/c is not deterministic since there are two transitions from state q_1 on input 1 and no transition (zero transition) from q_4 on both 0 & 1.

For any string ω whose 3rd symbol from the right is a 1, there exists a sequence of legal transitions leading from the start state q, to the accept state q_4. But for any string ω where 3rd symbol from the right is 0, there is no possible sequence of legal tranisitons leading from q_1 and q_4. Hence m/c accepts L. How does it accept any string $\omega \in$ L?

The m/c starts at q_1 and remains in the state q_1 on any input until the 3rd symbol from the right is encountered. (Of course, ω must satisfy $|\omega| \geq 3$). At this point, if the symbol is 1, it goes to the state q_2 and these enters q_3 & q_4 in the next two steps on any input 0 or 1. But if

the 3rd symbol from the right is q_0, thus it will get stuck at that point, because of no transition defined.

To enter the state q_2 from q_1, the m/c needs the input 1. If the 1 occur prior to the position 4 in the input or more from the right (instead of 3rd), thus it can enter q_2 from q_1 on that input and finally will enter accept state q_4 but at that point some of the input symbols may be left i.e. the input will not be exhausted and hence, the string will not be accepted by the m/c.

2.4.3THE EXTENDED TRANSITION FUNCTION, $\hat{\delta}$:

To describe acceptance by an NFA formally, it is necessary to extend the transition function, denoted as $\hat{\delta}$, takes a state $q \in Q$ and a string $\varpi \in \Sigma^*$, and returns the set of states, S $\subseteq$ Q, that the NFA is in after processing the string ϖ if it starts in state q.

Formally, $\hat{\delta}$ is defined as follows:

1. $\hat{\delta}(q,\in) = \{q\}$ that is, without rending any input symbol, an NFA doesnot change state.
2. Let $\varpi = xa \ \forall$ some $\varpi, x \in \Sigma^*$ and $a \in \Sigma$. Also assume that $\hat{\delta}(q,x) = \{p_1, p_2, \cdots, p_k\}$. Then $\hat{\delta}(q,\varpi) = \bigcup_{i=1}^{k} \delta(p_i, a)$.

That is, $\hat{\delta}(q,\varpi)$ can be computed by first computing $\hat{\delta}(q,x)$, and by then following any transitive from any of these stats that is labelled *a*.

2.5 FORMAL DEFINITION OF NFA :

Formally, an NFA is a quituple $M = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, and F bear the same meaning as for a DFA, but δ, the transition function is redefined as follows:

$$\delta: Q \times (\Sigma \cup \{\in\}) \to P(Q)$$

where P(Q) is the power set of Q i.e. 2^Q.

2.5.1 THE LANGUAGE OF AN NFA :

From the discussion of the acceptance by an NFA, we can give the formal definition of a language accepted by an NFA as follows :

If $N=(Q,\Sigma,\delta,q_0,F)$ is an NFA, then the langauge accepted by N is writtten as L(N) is given by $L(N)=\{w|\ \hat{\delta}(q_0,w)\cap F=\phi\}$.

That is, L(N) is the set of all strings w in Σ^* such that $\hat{\delta}(q_0,w)$ contains at least one accepting state.

2.6 CHECK YOUR PROGRESS

Fill in the blanks

1) A systems can be modeled abstractly by a mathematical model called_____

2) There are_____ states in the DFA

3)_____ transitions do not increase the power of an NFA .

4) _____ are changes of states that can occur spontaneously or in response to inputs to the states.

5) A _____ to hold the input string.

2.7 ANSWER CHECK YOUR PROGRESS

1) Finite Automation.

2) 2^n

3) $\in$

4) Transitions

5) Tape

2.8 MODEL QUESTION

Qs-1) What is finite automation? Also explain States, Transitions and Finite-State Transition System.

Qs-2) What is transition? Explain the difference between state transition diagram or simply a transition diagram.

Qs-3) What is Deterministic Finite State Automaton? DFA conceptually consist of how many parts?

Qs-4) What is Transition table? How can we remove epsilon transition?

Qs-5) What is the difference between NFA and DFA? How can we convert NFA to DFA?

2.8 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

2.9 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-III REGULAR EXPRESSIONS (RE)

3.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Regular Expressions (RE), Regular Expression and Regular Language, Regular Grammars. We also understand Some Decision Algorithms for CFLs.

3.2 REGULAR EXPRESSIONS (RE)

RES: 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) ϕ is a RE

ii) $\in$ □is a RE

iii) $\forall\, a \in S$, a is RE.

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

REs: 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) ϕ is a RE

ii) $\in$ □is a RE

iii) $\forall\, a \in S$, a is RE.

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

Recursive Step :

If r_1 and r_2 are REs over, then so are

i) $r_1 + r_2$

ii) $r_1 r_2$

iii) r_1^*

iv) (r_1)

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 Σ = { 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	ϕ	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. ϕ is the RE describing the empty language i.e. L(ϕ) = ϕ .

2. $\in$ is a RE describing the language {$\in$} i.e. L($\in$) = {$\in$} .

3. $\forall a \in S$, a is a RE denoting the language {a} i.e . L(a) = {a} .

4. If r_1 and r_2 are REs denoting language L(r_1) and L(r_2) respectively, then

i) $r_1 + r_2$ is a regular expression denoting the language L($r_1 + r_2$) = L(r_1) □ L(r_2)

ii) $r_1 r_2$ is a regular expression denoting the language L($r_1 r_2$)=L(r_1) L(r_2)

iii) r_1^* is a regular expression denoting the language $L(r_1^*) = (L(r1))^*$

iv) (r_1) is a regular expression denoting the language L((r_1)) = L(r_1)

Example : Consider the RE (0*(0+1)). Thus the language denoted by the RE is

L(0*(0+1)) = L(0*) L(0+1) .....by 4(ii)

= L(0)*L(0) □□L(1)

= {∈□, 0,00,000,.....} {0} ∪□{1}

= {∈□, 0,00,000,.....} {0,1}

= {0, 00, 000, 0000,.....,1, 01, 001, 0001,.....}

Precedence Rule

Consider the RE ab + c. The language described by the RE can be thought of either L(a)L(b+c) or L(ab)∪L(c) as provided by the rules (of languages described by REs) given already. But these two represents two different languages lending to ambiguity. To remove this ambiguity we can either

1) Use fully parenthesized expression- (cumbersome) or

2) Use a set of precedence rules to evaluate the options of REs in some order. Like other algebras mod in mathematics.

For REs, the order of precedence for the operators is as follows:

i) The star operator precedes concatenation and concatenation precedes union (+) operator.

ii) It is also important to note that concatenation & union (+) operators are associative and union operation is commutative.

Using these precedence rule, we find that the RE ab+c represents the language L(ab) ∪ L(c) i.e. it should be grouped as ((ab)+c).

We can, of course change the order of precedence by using parentheses. For example, the language represented by the RE a(b+c) is L(a)L(b+c).

Example : The RE ab*+b is grouped as ((a(b*))+b) which describes the language L(a)(L(b))* $\cup$ L(b)

Example : The RE (ab)*+b represents the language (L(a)L(b))* $\cup$ L(b).

Example : It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

Example : The regular expression r =(00)*(11)*1 denotes the set of all strings with an even number of 0's followed by an odd number of 1's i.e. $L(r) = \{0^{2n}1^{2m+1} \mid n \geq 0, m \geq 0\}$

Note : The notation r^+ is used to represent the RE rr*. Similarly, r^2 represents the RE rr, r^3 denotes r^2r, and so on.

An arbitrary string over Σ = {0,1} is denoted as (0+1)*.

Exercise : Give a RE r over {0,1} s.t. L(r)={$\omega \in \Sigma^* \mid \omega$ has at least one pair of consecutive 1's}

Solution : Every string in L(r) must contain 00 somewhere, but what comes before and what goes before is completely arbitrary. Considering these observations we can write the REs as (0+1)*11(0+1)*.

Example : Considering the above example it becomes clean that the RE (0+1)*11(0+1)*+(0+1)*00(0+1)* represents the set of string over {0,1} that contains the substring 11 or 00.

Example : Consider the RE 0*10*10*. It is not difficult to see that this RE describes the set of strings over {0,1} that contains exactly two 1's. The presence of two 1's in the RE and any no of 0's before, between and after the 1's ensure it.

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*

Example : Consider a RE r over {0,1} such that

L(r) = { $\varpi \in \{0,1\}^* \mid \varpi$ has no pair of consecutive 1's}

Solution : Though it looks similar to ex ……., it is harder to construct to construct. We observer that, whenever a 1 occurs, it must be immediately followed by a 0. This substring may be preceded & followed by any no of 0's. So the final RE must be a repetition of strings of the form: 00…0100….00 i.e. 0*100*. So it looks like the RE is (0*100*)*. But in this case the strings ending in 1 or consisting of all 0's are not accounted for. Taking these observations into consideration, the final RE is r = (0*100*)(1+ $\in$)+0*(1+$\in$).

Alternative Solution :

The language can be viewed as repetitions of the strings 0 and 01. Hence get the RE as r = (0+10)*(1+$\in$).This is a shorter expression but represents the same language.

CHECK YOUR PROGRESS

True/False type questions

1) The language that is accepted by some FAs are known as Regular language._____

2) A language L is regular iff it has a regular grammar_____

3) Regular grammar and Finite Automata are equivalent_____

4) There are no algorithms to test emptiness of a CFL._____

5) If a language is regular, then there is no RE to describe it._____

Answers-

1) True

2) True

3) True

4) False

5) False

3.3 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) $\cong$ L(r).

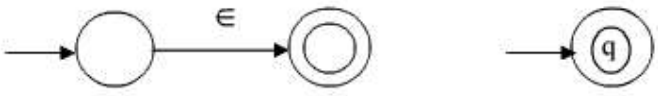
Proof : To prove the lemma, we apply structured index on the expression r. First, we show how to construct FA for the basis elements: ϕ, $\in$ and for any $a \in \Sigma$. Then we show how to combine these Finite Automata into Complex Automata that accept the Union, Concatenation, Kleen Closure of the 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.

Basis :

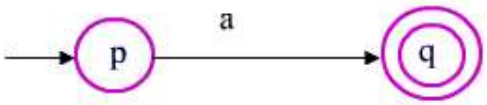
Case (i) : $r = \phi$. Then $L(r) = \phi$. Then $L(r) = \phi$ and the following NFA N recognizes L(r). Formally $N = (Q, \{q\}, \Sigma, \delta, q, F, \phi)$ where Q = {q} and $\delta(q,a) = \phi \; \forall \; a \in S, \; F = \phi$.

Case (ii) : $r = \in$. $L(r) = \{\in\}$, and the following NFA N accepts L(r). Formally $N = (\{q\}, \Sigma, \delta, q, \{q\})$ where $\delta(q,a) = \phi \quad \forall a \in \Sigma$.



Since the start state is also the accept step, and there is no any transition defined, it will accept the only string $\in$ and nothing else.

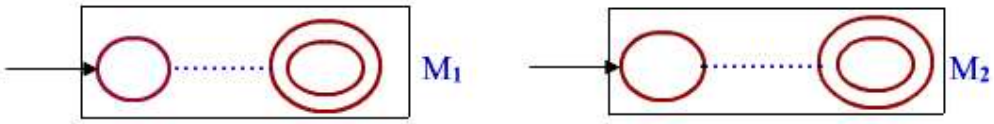
Case (iii) : r = a for some $a \in \Sigma$. Then L(r) = {a}, and the following NFA N accepts L(r).



Formally, $N = (\{p,q\}, \Sigma, \delta, p, \{q\})$ where $\delta(p,q) = \{q\}$, $\delta(s, b) = (\phi)$ for $s \neq p$ or $b \neq a$

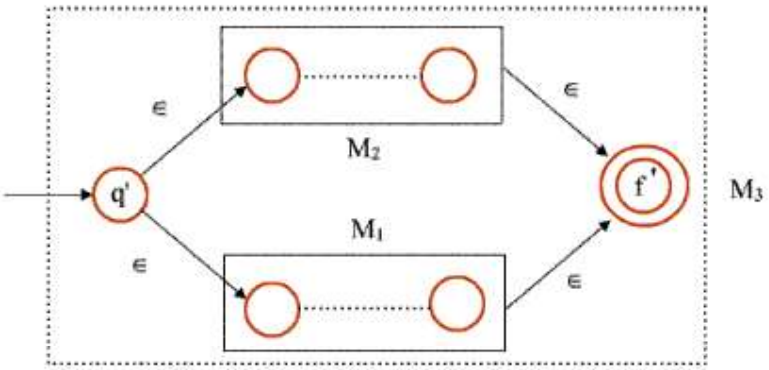
Induction :

Assume that the start of the theorem is true for REs r_1 and r_2. Hence we can assume that we have automata M_1 and M_2 that accepts languages denoted by REs r_1 and r_2, respectively i.e. $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The FAs are represented schematically as shown below.



Each has an initial state and a final state. There are four cases to consider.

Case (i) : Consider the RE $r_3 = r_1 + r_2$ denoting the language $L(r_1) \cup L(r_2)$. We construct FA M_3, from M_1 and M_2 to accept the language denoted by RE r_3 as follows :



- Create a new (initial) start state q' and give $\in$- transition to the initial state of M_1 and M_2 .This is the initial state of M_3.
- Create a final state f' and give $\in$ -transition from the two final state of M_1 and M_2 . f' is the only final state of M_3 and final state of M_1 and M_2 will be ordinary states in M_3.
- All the state of M_1 and M_2 are also state of M_3.
- All the moves of M_1 and M_2 are also moves of M_3. [Formal Construction]
- It is easy to prove that $L(M_3) = L(r_3)$

Proof: To show that $L(M_3) = L(r_3)$ we must show that

$= L(r_1) \cup L(r_2)$

$= L(M_1) = L(M_2)$ by following transition of M_3.

Starts at initial state q' and enters the start state of either M_1 or M_2 follwoing the transition i.e. without consuming any input. WLOG, assume that, it enters the start state of M_1. From this point onward it has to follow only the transition of M_1 to enter the final state of M_1, because this is the only way to enter the final state of M by following the e-transition.(Which is the last transition & no input is taken at hte transition). Hence the whole input w is considered while traversing from the start state of M_1 to the final state of M_1. Therefore M_1 must accept w_3 .

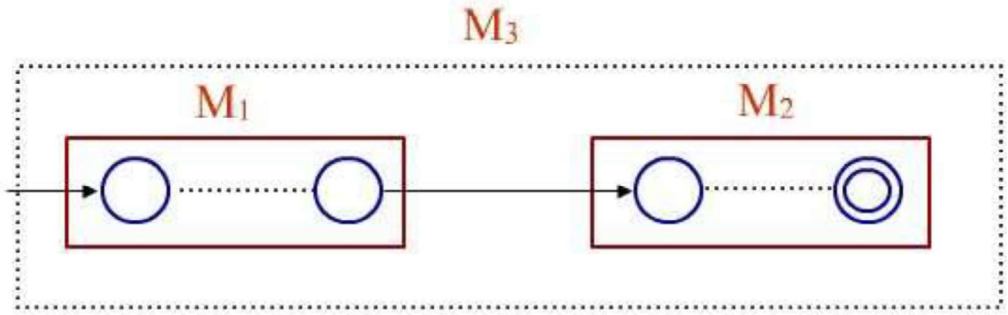
Say, $w \in L(M_1)$ or $w \in L(M_2)$.

WLOG, say $w \in L(M_1)$

Therefore when M_1 process the string w , it starts at the initial state and enters the final state when w consumed totally, by following its transition. Then M_3 also accepts w, by starting at

state q' and taking $\in$ -transition enters the start state of M_1 -follows the moves of M_1 to enter the final state of M_1 consuming input w thus takes $\in$-transition to f'. Hence proved.

Case(ii) : Consider the RE $r_3 = r_1 r_2$ denoting the language $L(r_1)L(r_2)$. We construct FA M_3 from M_1 & M_2 to accept $L(r_3)$ as follows :



Create a new start state q' and a new final state

Add $\in$- transition from

q' to the start state of M_1

q' to f'

final state of M_1 to the start state of M_1

All the states of M_1 are also the states of M_3. M_3 has 2 more states than that of M_1 namely q' and f'.

All the moves of M_1 are also included in M_3.

By the transition of type (b), M_3 can accept $\in$.

By the transition of type (a), M_3 can enters the initial state of M_1 w/o any input and then follow all kinds moves of M_1 to enter the final state of M_1 and then following $\in$-transition

can enter f'. Hence if any $w \in \Sigma^*$ is accepted by M_1 then w is also accepted by M_3. By the transition of type (b), strings accepted by M_1 can be repeated by any no of times & thus accepted by M_3. Hence M_3 accepts $\in$ and any string accepted by M_1 repeated (i.e. concatenated) any no of times. Hence $L(M_3) = (L(M_1))^* = (L(r)_1)^* = r_1^*$.

Case(iv) : Let $r_3 = (r_1)$. Then the FA M_1 is also the FA for (r_1), since the use of parentheses does not change the language denoted by the expression.

FA to RE (REs for Regular Languages) :

Lemma : If a language is regular, then there is a RE to describe it. i.e. if L = L(M) for some DFA M, then there is a RE r such that L = L(r).

Proof : We need to construct a RE r such that $L(r) = \{w \mid w \in L(M)\}$. Since M is a DFA, it has a finite no of states. Let the set of states of M is Q = {1, 2, 3,..., n} for some integer n. [Note : if the *n* states of *M* were denoted by some other symbols, we can always rename those to indicate as 1, 2, 3,..., *n*]. **The required RE is constructed inductively.**

Notations : $r_{ij}^{(k)}$ is a RE denoting the language which is the set of all strings w such that w is the label of a path from state i to state j $(1 \le i, j \le n)$ in M, and that path has no intermediate state whose number is greater then k. (i & j (begining and end pts) are not considered to be "intermediate" so i and /or j can be greater than k) We now construct $r_{ij}^{(k)}$ inductively, for all i, j $\in$ Q starting at k = 0 and finally reaching k = n.

Basis : k = 0, $r_{ij}^{(0)}$ i.e. the paths must not have any intermediate state (since all states are numbered 1 or above). There are only two possible paths meeting the above condition :

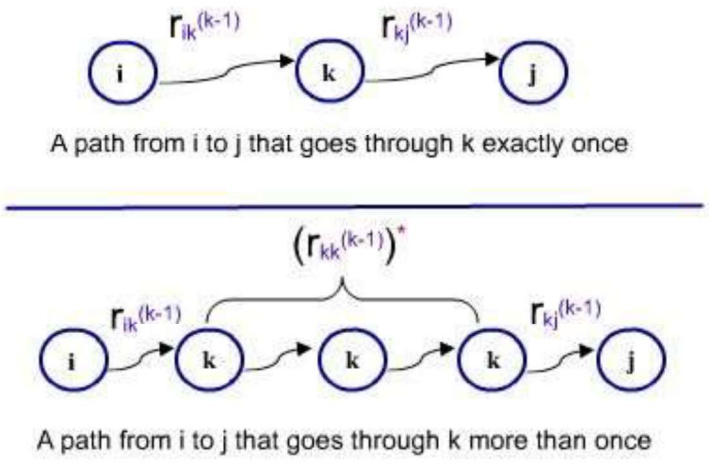
1. **A direct transition from state i to state j.**
   - $r_{ij}^{(0)}$ = a if then is a transition from state i to state j on symbol the single symbol a.
   - $r_{ij}^{(0)} = a_1 + a_2 + \cdots + a_m$ if there are multiple transitions from state i to state j on symbols $a_1, a_2, \cdots, a_m$.
   - $r_{ij}^{(0)}$ = f if there is no transition at all from state i to state j.
2. All paths consisting of only one node i.e. when i = j. This gives the path of length 0 (i.e. the RE $\in$ denoting the string $\in$) and all self loops. By simply adding Î to various cases above we get the corresponding REs i.e.
   - $r_{ii}^{(0)}$ = $\in$ + a if there is a self loop on symbol a in state i .

- $r_{ij}^{(0)} = \in + a_1 + a_2 + \cdots + a_m$ if there are self loops in state i as multiple symbols $a_1, a_2, \cdots, a_m$.
- $r_{ij}^{(0)} = \in$ if there is no self loop on state i.

Induction :

Assume that there exists a path from state i to state j such that there is no intermediate state whose number is greater than k. The corresponding Re for the label of the path is $r_{ij}^{(k)}$.
There are only two possible cases :

1. The path dose not go through the state k at all i.e. number of all the intermediate states are less than
   k. So, the label of the path from state i to state j is tha language described by the RE $r_{ij}^{(k-1)}$.
2. The path goes through the state k at least once. The path may go from i to j and k may appear more than once. We can break the into pieces as shown in the figure 7.



o

Figure 7

3. The first part from the state i to the state k which is the first recurence. In this path, all intermediate states are less than k and it starts at iand ends at k. So the RE $r_{ik}^{(k-1)}$ denotes the language of the label of path.
4. The last part from the last occurence of the state k in the path to state j. In this path also, no intermediate state is numbered greater than k. Hence the RE $r_{kj}^{(k-1)}$ denoting the language of the label of the path.

5. In the middle, for the first occurence of k to the last occurence of k , represents a loop which may be taken zero times, once or any no of times. And all states between two consecutive k's are numbered less than k.

Hence the label of the path of the part is denoted by the RE $\left(r_{ij}^{(k-1)}\right)^*$ .The label of the path from state i to state j is the concatenation of these 3 parts which is

$$r_{ik}^{(k-1)}\left(r_{kk}^{(k-1)}\right)^* r_{kj}^{(k-1)}$$

Since either case 1 or case 2 may happen the labels of all paths from state i to j is denoted by the following RE

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} + r_{ik}^{(k-1)}\left(r_{kk}^{(k-1)}\right)^* r_{kj}^{(k-1)}.$$

We can construct $r_{ij}^{(k)}$ for all i, j $\in$ {1,2,..., n} in increasing order of k starting with the basis k = 0 upto k = n since $r_{ij}^{(k)}$ depends only on expressions with a small superscript (and hence will be available). WLOG, assume that state 1 is the start state and $j_1, j_2, \cdots, j_m$ are the m final states where ji $\in$ {1, 2, ... , n }, $1 \le i \le m$ and $m \le n$. According to the convention used, the language of the automatacan be denoted by the RE

$$r_{1j_1}^{(n)} + r_{1j_2}^{(n)} + \cdots + r_{1j_m}^{(n)}$$

Since $r_{1j_i}^{(n)}$ is the set of all strings that starts at start state 1 and finishes at final state j_i following the transition of the FA with any value of the intermediate state (1, 2, ... , n) and hence accepted by the automata.

3.4 REGULAR GRAMMARS

A grammar $G = (N, \Sigma, P, S)$ is right-linear if each production has one of the following three forms:

- $A \rightarrow cB$,
- $A \rightarrow c$,
- $A \rightarrow \in$

Where A, B $\in N'$ (with A = B allowed) and $c \in \Sigma$. A grammar G is left-linear if each production has once of the following three forms.

$A \rightarrow Bc$, $A \rightarrow c$, $A \rightarrow \in$

A right or left-linear grammar is called a regular grammar.

Regular Grammars and Finite Automata

Regular grammar and Finite Automata are equivalent as stated in the following theorem.

Theorem : A language L is regular iff it has a regular grammar. We use the following two lemmas to prove the above theorem.

Lemma 1 : If L is a regular language, then L is generated by some right-linear grammar.

Proof : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that accepts L.

Let $Q = \{q_0, q_1, \cdots, q_n\}$ and $\Sigma = \{a_1, a_2, \cdots, a_m\}$.

We construct the right-linear grammar $G = (N, \Sigma, P, S)$ by letting

N = Q , $S = q_0$ and $P = \{A \rightarrow cB \mid \delta(A, c) = B\} \cup \{A \rightarrow c \mid \delta(A, c) \in B\}$

[Note: If $B \in F$, then $B \rightarrow \in P$]

Let $w = a_1 a_2 \ldots a_k \in L(M)$. For M to accept w, there must be a sequence of states $q_0, q_1, \cdots, q_k$ such that

$\delta(q_0, a_1) = q_1$

$\delta(q_1, a_2) = q_2$

$\cdots$

$\delta(q_{k-1}, a_k) = q_k$

and $q_k \in F$

By construction, the grammar G will have one production for each of the above transitions. Therefore, we have the corresponding derivation.

$$S = q_0 \underset{G}{\Rightarrow} a_1 q_1 \underset{G}{\Rightarrow} a_1 a_2 q_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a_1 a_2 \cdots a_k q_k \underset{G}{\Rightarrow} a_1 a_2 \cdots a_k = w$$

Hence w $\in$ L(g).

Conversely, if $w = a_1 a_2 \ldots a_k \in L(G)$, then the derivation of w in G must have the form as given above. But, then the construction of G from M implies that

$\delta(q_0, a_1 a_2 \cdots a_k) = q_k$, where $q_k \in F$, completing the proof.

Lemma 2 : Let $G = (N, \Sigma, P, S)$ be a right-linear grammar. Then L(G) is a regular language.

Proof: To prove it, we construct a FA M from G to accept the same language.

$M = (Q, \Sigma, \delta, q_0, F)$ is constructed as follows:

$Q = N \cup \{q_f\}$ (q_f is a special sumbol not in N)

$q_0 = S$, $F = \{q_f\}$

For any $q \in N$ and $a \in \Sigma$ and δ is defined as

$\delta(q, a) = \{p \mid q \to ap \in P\}$ if $q \to a \notin P$

and $\delta(q, a) = \{p \mid q \to ap \in P\} \cup \{q_f\}$, if $q \to a \in P$.

We now show that this construction works.

Let $w = a_1 a_2 \ldots a_k \in L(G)$. Then there is a derivation of w in G of the form

$$S \underset{G}{\Rightarrow} a_1 q_1 \underset{G}{\Rightarrow} a_1 a_2 q_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} a_1 a_2 \cdots a_{k-1} a_k \, (= w)$$

By contradiction of M, there must be a sequence of transitions

$\delta(q_0, a_1) = q_1$

$\delta(q_1, a_2) = q_2$

$\cdots$

$\delta(q_{k-1}, a_k) = q_f$

implying that $w = a_1 a_2 \ldots a_k \in L(M)$ i.e. w is accepted by M.

Conversely, if $w = a_1 a_2 \cdots a_k$ is accepted by M, then because q_f is the only accepting state of M, the transitions causing w to be accepted by M will be of the form given above. These transitions corresponds to a derivationof w in the grammar G. Hence $w \in L(G)$, completing the proof of the lemma.

Given any left-linear grammar G with production of the form $A \to cB \mid c \mid \in$, we can construct from it a right-linear grammar $\hat{G}$ by replacing every production of G of the form $A \to cB$ with $A \to Bc$

It is easy to prove that $L(G) = \left(L\left(\hat{G}\right)\right)^R$. Since $\hat{G}$ is right-linear, $L\left(\hat{G}\right)$ is regular. But then so are $\left(L\left(\hat{G}\right)\right)^R$ i.e. $L(G)$ because regular languages are closed under reversal.

Putting the two lemmas and the discussions in the above paragraph together we get the proof of the theorem-

A language L is regular iff it has a regular grammar.

Example: Consider the regular expression 101*. The DFA for 101* is shown below.

The right linear grammar generating the language denoted by 101* i.e accepted by the above DFA is produced below follwoing the construction process given in the lemma 1.

$S \to 1A \mid 0C$
$A \to 0B \mid 1C \mid 0$

$B \to 1B \mid 0C \mid 1$
$C \to 0C \mid 1C$

Since, C is useless we can eliminate all productins involving C to produce a simpler grammar for 101*

$S \to 1A$
$A \to 0B \mid 0$
$B \to 1B \mid 1$

Example : Consider the grammar

$G: S \to 0A \mid 0$
$A \to 1S$

It is easy to see that G generates the language denoted by the regular expression (01)*0.

The construction of lemma 2 for this grammar produces the follwoing FA.

This FA accepts exactly (01)*1.

3.5 SOMEDECISION ALGORITHMS FOR CFLS

n this section, we examine some questions about CFLs we can answer. A CFL may be represented using a CFG or PDA. But an algorithm that uses one representation can be made to work for the others, since we can construct one from the other.

Testing Emptiness :

Theorem : There are algorithms to test emptiness of a CFL.

Proof : Given any CFL L, there is a CFG G to generate it. We can determine, using the construction described in the context of elimination of useless symbols, whether the start symbol is useless. If so, then $L(G) = \phi$; otherwise not.

Testing Membership :

Given a CFL L and a string x, the membership, problem is to determine whether $x \in L$?

Given a PDA P for L, simulating the PDA on input string x doesnot quite work, because the PDA can grow its stack indefinitely on $\in$ input, and the process may never terminate, even if the PDA is deterministic.

So, we assume that a CFG $G = (N, \Sigma, P, S)$ is given such that L = L(G).

Let us first present a simple but inefficient algorithm.

Convert G to $G' = (N', \Sigma', P', S')$ in CNF generating $L(G) - \{\in\}$. If the input string $x = \in$, then we need to determine whether $S \overset{*}{\underset{G}{\Rightarrow}} \in$ and it can easily be done using the technique given in the context of elimination of $\in$-production. If , $x \neq \in$ then $x \in L(G')$ iff $x \in L(G)$. Consider a derivation under a grammar in CNF. At every step, a production in CNF in used, and hence it adds exactly one terminal symbol to the sentential form. Hence, if the length of the input string x is n, then it takes exactly n steps to derive x (provided x is in $L(G')$).

Let the maximum number of productions for any nonterminal in G' is K. So at every step in derivation, there are atmost k choices. We may try out all these choices, systematically., to

derive the string x in G'. Since there are atmost $K^{|x|}$ i.e. K^n choices. This algorithms is of exponential time complexity. We now present an efficient (polynomial time) membership algorithm.

CYK Algorithm to decide membership in CFL

We now present a cubic-time algorithm due to cocke, Younger and Kasami. It uses the dynamic programming technique-solves smaller sub-problems first and then builds up solution by combining smaller sub-solutions. It determines for each substring y of the given string x the set of all nonterminals that generate y. This is done inductively on the length of y.

Let $G = (N, \Sigma, P, S)$ be the given CFG in CNF. Consider the given string x and let $|x| = n$. Let x_{ij} be the substring of x that begins at position i (i.e. i-th symbol of x) and has length j. Let N_{ij} be the set of all nonterminals A such that $A \overset{*}{\Rightarrow} x_{ij}$.

We write $x = x_{11}x_{21}x_{31}\cdots x_{(n-1)1}x_{n1}$. Where each $x_{i1}\,(1 \le i \le n)$ is a terminal symbol.

$A \overset{*}{\Rightarrow} x_{i1}$ iff $A \to x_{i1} \in P$. Thus we construct the sets N_{i1} for all $1 \le i \le n$.

Combining substrings of length 2, it is clear that, $A \in N_{i2}$ i.e. $A \overset{*}{\Rightarrow} x_{i2}$ iff there is a production $A \to BC$ in G and $B \overset{*}{\Rightarrow} x_{i1}$ and $C \overset{*}{\Rightarrow} x_{i+1,1}$.

That is $A \in N_{i2}$ iff $A \to BC \in P$ and $B \in N_{i1}$ and $C \in N_{i+1,1}$

Thus we can construct the sets N_{i2} from the already constructed sets N_{i1}, by inspecting the grammar.

In general considering substrings x_{ij} of length j, $A \in N_{ij}$ i.e. $A \overset{*}{\Rightarrow} x_{ij}$ iff there is a production $A \to BC$ in G such that $B \overset{*}{\Rightarrow} x_{ik}$ and $C \overset{*}{\Rightarrow} x_{i+k,j-k}$ for some $1 \le k \le j$.

That is $A \in N_{ij}$ iff $B \in N_{ik}$ and $C \in N_{i+k,j-k}$ for some $1 \le k \le j$ such that $A \to BC \in P$. The idea is to divide, x_{ij} into smaller substrings, using all possible wyas (i.e. for different values of k), and construct N_{ij} from already

Combining substrings of length 2, it is clear that, $A \in N_{i2}$ i.e. $A \overset{*}{\Rightarrow} x_{i2}$ iff there is a production $A \to BC$ in G and $B \overset{*}{\Rightarrow} x_{i1}$ and $C \overset{*}{\Rightarrow} x_{i+1,1}$.

That is $A \in N_{i2}$ iff $A \to BC \in P$ and $B \in N_{i1}$ and $C \in N_{i+1,1}$

Thus we can construct the sets N_{i2} from the already constructed sets N_{i1}, by inspecting the grammar.

In general considering substrings x_{ij} of length j, $A \in N_{ij}$ i.e. $A \overset{*}{\Rightarrow} x_{ij}$ iff there is a production $A \to BC$ in G such that $B \overset{*}{\Rightarrow} x_{ik}$ and $C \overset{*}{\Rightarrow} x_{i+k,j-k}$ for some $1 \le k \le j$.

That is $A \in N_{ij}$ iff $B \in N_{ik}$ and $C \in N_{i+k,j-k}$ for some $1 \le k \le j$ such that $A \to BC \in P$. The idea is to divide, x_{ij} into smaller substrings, using all possible wyas (i.e. for different values of k), and construct N_{ij} from already

Limitations of Finite Automata and Non regular Languages :

The class of languages recognized by FA s is strictly the regular set. There are certain languages which are non regular i.e. cannot be recognized by any FA

Consider the language $L = \{a^n b^n \mid n \ge 0\}$

In order to accept is language, we find that, an automaton seems to need to remember when passing the center point between a's and b's how many a's it has seen so far. Because it would have to compare that with the number of b's to either accept (when the two numbers are same) or reject (when they are not same) the input string.

But the number of a's is not limited and may be much larger than the number of states since the string may be arbitrarily long. So, the amount of information the automaton need to remember is unbounded.

A finite automaton cannot remember this with only finite memory (i.e. finite number of states). The fact that *FA* s have finite memory imposes some limitations on the structure of the languages recognized. Inductively, we can say that a language is regular only if in processing any string in this language, the information that has to be remembered at any point is strictly limited. The argument given above to show that $a^n b^n$ is non regular is informal. We now present a formal method for showing that certain languages such as $a^n b^n$ are non regular.

The Pumping Lemma

We can prove that a certain language is non regular by using a theorem called “Pumping Lemma”. According to this theorem every regular language must have a special property. If a language does not have this property, than it is guaranteed to be not regular. The idea behind this theorem is that whenever a FA process a long string (longer than the number of states)

and accepts, there must be at least one state that is repeated, and the copy of the sub string of the input string between the two occurrences of that repeated state can be repeated any number of times with the resulting string remaining in the language.

Pumping Lemma :

Let L be a regular language. Then the following property olds for L.

There exists a number $k \geq 0$ (called, the pumping length), where, if w is any string in L of length at least k i.e. $|w| \geq k$, then w may be divided into three sub strings w = xyz, satisfying the following conditions:

$y \neq \in$ i.e. $|y| > 0$

$|xy| \leq k$

$\forall i \geq 0,\ xy^i z \in L$

Proof : Since L is regular, there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes it, i.e. L = L(M) . Let the number of states in M is n.

Say, $Q = \{q_0, q_1, q_2, \cdots, q_n\}$

Consider a string $w \in L$ such that $|w| \geq n$ (we consider the language L to be infinite and hence such a string can always be found). If no string of such length is found to be in L , then the lemma becomes vacuously true.

Since $w \in L, \hat{\delta}(q_0, w) \in F$. Say $\hat{\delta}(q_0, w) = q_m$ while processing the string w , the DFA M goes through a sequence of states of states. Assume the sequence to be

$q_0, q_3, q_2, q_4, \cdots, q_i, \cdots q_l, \cdots, q_m$

start state final start

Since $|w| \geq n$, the number of states in the above sequence must be greater than n + 1. But number of states in M is only n. hence, by pigeonhole principle at least one state must be repeated.

Let qi and ql be the ql same state and is the first state to repeat in the sequence (there may be some more, that come later in the sequence). The sequence, now, looks like

$q_0, q_3, q_2, q_4, \cdots, q_i, \cdots q_l, \cdots, q_m$

which indicates that there must be sub strings x, y, z of w such that

$\hat{\delta}(q_0, x) = q_i$

$\hat{\delta}(q_i, y) = q_i$

$\hat{\delta}(q_i, z) = q_m$

This situation is depicted in the figure

Since $q_l (= q_i)$ is the first repeated state, we have, $|xy| \leq n$ and at the same time y cannot be empty i.e $|y| > 0$. From the above, it immediately follows that $\hat{\delta}(q_0, xz) = q_m$. Hence $xz = xy^{\circ}z \in L$. Similarly,

$\hat{\delta}(q_0, xy^2z) = q_m$ implying $xy^2z \in L$

$\hat{\delta}(q_0, xy^3z) = q_m$ implying $xy^3z \in L$

and so on.

That is, starting at the loop on state can be omitted, taken once, twice, or many more times, (by the DFA M) eventually arriving at the final state

Thus, accepting the string xz, xyz, xy2z,... i.e. xyiz for all $i \geq 0$

Hence $\forall i \geq 0, xy^iz \in L$.

We can use the pumping lemma to show that some languages are non regular.

Please note, carefully, hat the theorem guarantees the existence of a number $k \geq 0$ as well as the decomposition of the string w to xyz. But it is not known what they are. So, if the theorem is violated for particular values of

3.6 CHECK YOUR PROGRESS

Fill in the blanks:

1) A language that is accepted by some FAs are known as_____

2) Given any CFL L, there is a _____ G to generate it.

3) Regular grammar and _____ are equivalent.

4) There are algorithms to test _____ of a CFL.

5) If L is a regular language, then L is generated by some_____

3.7 ANSWER CHECK YOUR PROGRESS

1) Regular language.

2) CFG

3) Finite Automata

4) Emptiness

5) Right-linear grammar.

3.8 MODEL QUESTION

Qs-1) What is Regular Expression explain with the help of example?

Qs-2) Why Regular expression and Finite automata are equivalent explain with the help of example?

Qs-3) What is CFL? How emptiness of a CFL is tested?

Qs-4) What is precedence rule? Explain.

Qs-5) What is Context Free Grammar(CFG)? For Context Free Grammar(CFG) which grammar is used?

3.9 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

3.10 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-IV MINIMIZATION OF DETERMINISTIC FINITE AUTOMATA (DFA)

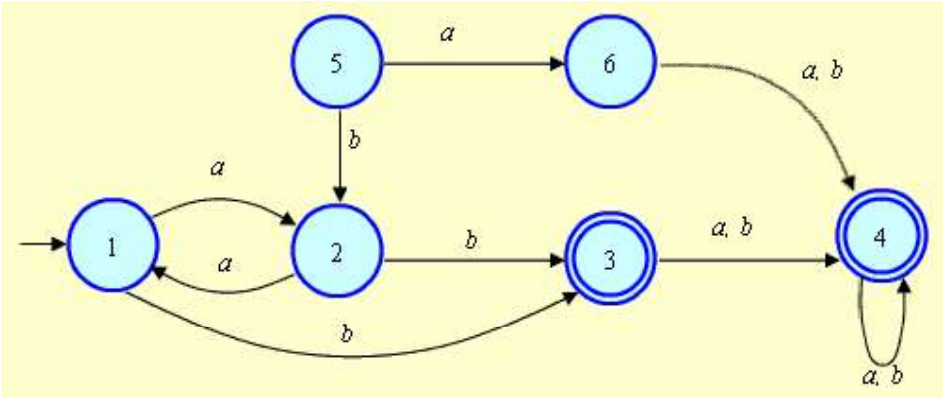
4.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of DFA Isomorphisms, The minimal DFA, A Minimization Algorithm, Some decision properties of Regular Languages, Finite Automata with output, Moore machines, Mealy machines. We also understand Equivalence of Moore and Mealy machines.

4.2 MINIMIZATION OF DETERMINISTIC FINITE AUTOMATA (DFA)

For any regular language L it may be possible to design different DFAs to accept L. Given two DFAs accepting the same language L, it is now natural to ask - which one is more simple? In this case, obviously, the one with less number of states would be simpler than the other. So, given a DFA accepting a language, we might wonder whether the DFA could further be simplified i.e. can we reduce the number of states accepting the same language ?

Consider the follwoing DFA M_1,



A minute observation will reveal that it accepts the language of the regular expression

$a^*b(a+b)^*$

The same language is accepted by the following simpler DFA M_2 as well.

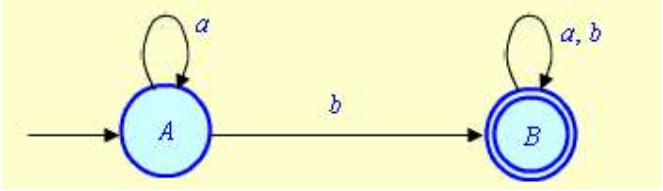


Figure 2

It is a fact that, for any regular language L there is a unique minimal state DFA (the uniqueness is up to isomorphism to be defined next).

For any given DFA M accepting L we can construct the minimal state DFA accepting L by using an algorithm which uses following generic steps.

- First, remove all the states (of the given DFA M) which are not accessible from the start state i.e. sates P for which there is no string $x \in \Sigma^*$ s.t. $\hat{\delta}(q_0, x) = p$. Removing these states, clearly, will not change the language accepted by the DFA.
- Second, remove all the trap states, i.e. all states P from which there is no transition out of it.
- Finally, merge all states which are "equivalent" or "indistinguishable". We need to define formally what is meant by equivalent or indistinguishable states; but at this point we assume that merging these states would not change the accepted language.

Inaccessible states can easily be found out by using a simple research e.g. depth first search. removing trap states are also simple. In the example, states 5 and 6 are inaccessible and hence can be removed, states 1 and 2 are equivalent and can be merged. Similarly states 3 & 4 are also equivalent and can be merged together to have the minimal DFA M_2 as produced above.

To construct the minimal DFA we need to see how to find out indistinguishable or equivalent states for merging.

we start with a definition and then proceed to find method to construct minimal state DFAs.

4.3 DFA ISOMORPHISMS :

Two DFAs are said to be isomorphism if they are identical upto renaming of the states. Formally, DFA isomorphisms are defined as follows.

Definiton : Two DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ are isomorphic if there is a bijection $f : Q_1 \to Q_2$ s.t. the follwoing hold.

1. $f(q_1) = q_2$
2. $\forall q \in Q_1$, $q \in F_1$ iff $f(q) \in F_2$
3. $\forall q \in Q_1$, $\forall a \in \Sigma$
   - $f(\delta_1(q, a)) = \delta_2(f(q), a)$

Theorem : For any regular language L there is a unique (upto isomorphism as defined) DFA that has a minimum number of states. In fact, the minimum DFA is the same as the one that has as states the equivalence classes of $\equiv_L$ (as defined in the context of Myhill-Nerode Theorem).

Proof: Let $M_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$ be the DFA which states are equivalence classes of $\equiv_L$. Let $M = (Q, \Sigma, \delta, q_0, F)$ be any other DFA recognizing L. we have already shown that

1. $\equiv_M$ is a right invariant equivalence relation of finite index s.t. L is the union of some of its equivalence classes.
2. $\equiv_M$ is a refinement of $\equiv_L$.
3. This implies, the number of equivalence classes of $\equiv_M$ (which is equal to the number of states in M) must be greater than or equal to the number of equivalence classes of $\equiv_L$ (which is equal to the number of states in M_L, by construction).
4. That is $|Q| \geq |Q_L|$
5. If $|Q| > |Q_L|$, then we are done, i.e. M_L is the minimum state DFA for L.
6. If $|Q| = |Q_L|$, then to prove the theorem we need to show that DFAs M_L and M are isomorphism.

4.3.1 SHOWING THAT M_L AND M ARE ISOMORPHIC

To show that M_L and M are isomorphic we have to define a bijection $f : Q_L \to Q$ that satisfies all the three conditions given in the definition of DFA isomorphism.

1. Recall that the states of M_L are $[x_1], [x_2], \cdots, [x_k]$ where $x_1, x_2, \cdots, x_k$ are the representatives of each k equivalence classes of $\equiv_L$.
2. Let us define $f : Q_L \to Q$ as follows

   $$f([x_i]) = \delta(q_0, x_i)$$

3. That is, f maps state $[x_i]$ of M_L to the state in M which can be arrived at processing the string x_i from the start state of M. we know that $\forall x_i \in \Sigma^* \quad \delta(q_0, x_i) \in Q$. Hence f is well-defined.
4. f is onto since $|Q_L| = |Q|$
5. To show that f is one-to-one, we need to show that $\forall p, q \in Q_L$ if $f(p) = f(q)$, then p = q . That means, we need to show that $\forall x, y \in \Sigma^*$ if $f([x]) = f([y])$, then $x \equiv_L y$. (since $x_1, x_2, \cdots, x_k$ are the representative of different equivalence classes of $\equiv_L$, this proves that f is one-to-one).

   Let $f([x]) = f([y]) = p \in Q$.

Then $\delta(q_0, x) = \delta(q_0, y) = p$

Therefore $\delta(q_0, xz) = \delta(q_0, yz) = \delta(p, z)$ for any $z \in \Sigma^*$.

Hence, by definition of $\equiv_L$,

$xz \in L$ iff $yz \in L$ or $x \equiv_L y$.

This shows that f is a bijection.

we now show in the following that it satisfies all the three conditions.

1. Note that, since f is a bijection, $x \equiv_L y \Rightarrow f([x]) = f([y])$. Also note that $q_{0L} \equiv_L [\in]$. Hence, $f(q_{0L}) = f([\in]) = \delta(q_0, \in) = q_0$. Therefore, the initial state $[\in]$ of M_2 is mapped to the initial state q_0 of M thus satisfying the first condition.
2. We know that for any $x_i \in \Sigma^*$
3. $x_i \in F$
   $\Leftrightarrow x_i \in L$ (by definition)
4. $\Leftrightarrow \delta(q_0, x_i) \in F$ (Since M accepts L)
5. $\Leftrightarrow f([x_i]) \in F$ (by definition of f)
6. Thus final state of M_L are mapped to final stat of M , satisfying the second condition.
7. Observe that, for any $x_i \in \Sigma^*$, $a \in \Sigma$

$$\begin{aligned} \delta(f[x_i], a) &= \delta(\delta(q_0, x_i), a) && (\text{ by definition of } f) \\ &= \delta(q_0, x_i a) \\ &= f([x_i a]) && (\text{ by definition of } f) \\ &= f(\delta_L(x_i, a)) && (\text{Since } [x_i a] \equiv_L \delta(x_i, a)) \end{aligned}$$

This satisfies the third condition of the definition, thus proving that M_L and M are isomorphic. This also completes the prove that M_L is the minimal state DFA for L since, now, $|Q| \geq |Q_L|$, (i.e. the number of state Q in any arbitrary DFA M accepting the language L must be greater than or equal to the number of states Q_L of the DFA M_L that has as states the equivalence classes of $\equiv_L$.)

4.4 THE MINIMAL DFA

Given DFA M accepting a regular language L, we observe that

1. M_L is the minimal state DFA accepting L.
2. $\equiv_M$ refines $\equiv_L$, implying
   Each equivalence classes of $\equiv_L$ is the union of some equivalence classes of $\equiv_M$.
3. Hence, each staet of M_L (which correspond to the equivalence class of $\equiv_L$) can be obtained by merging states of M. (which correspond to equivalence classes of $\equiv_M$)

But, how do we decide in general when two states can be merged without changing the language accepted?

we now going to devise an algorithm for doing this until no more merging is possible. we start with the following observations.

- It is not possible to merge an accept state p and a non-accepting state q. Because if $p = \hat{\delta}(q_0, x) \in F$ and $q = \hat{\delta}(q_0, y) \notin F$ for some $x, y \in \Sigma^*$, then x must be accepted and y must be rejected after merging p and q. But, now, the resulting merged state can neither be considered as an accept state nor as a non-accepting one.
- If p and q are merged, then we need to merge $\delta(p,a)$ and $\delta(q,a)$, for every $a \in \Sigma$, as well, to maintain determinism.

From the above two observations we conclude that states p and q cannot be merged if $\hat{\delta}(p,x) \in F$ and $\hat{\delta}(q,x) \notin F$ for some $x \in \Sigma^*$.

Using the concept in the previous page, we now define an indistinguishability relation as follows:

Definition : States p and q are indistinguishable if $\forall\, x \in \Sigma^*$ $\hat{\delta}(p,x) \in F$ iff $\hat{\delta}(q,x) \in F$, and is denoted as $p \equiv q$. It is easy to see that indistingushability is an equivalence relation.

In other words we say that sattes p and q are "distinguishable" if $\exists\, x \in \Sigma^*$ s.t $\hat{\delta}(p,x) \in F$ and $\hat{\delta}(q,x) \notin F$ and is denoted as $p \not\equiv q$.

we say that, states p and q of a DFA M accepting a language L can be merged safely (i.e. without changing the accepted language L) if $p \equiv q$ i.e. if p and q are indistinguishable. we can prove this by showing that when p and q are merged. Then they correspond to the same state in M_L.

Formally, $p \equiv q$ iff $\forall\, x, y \in \Sigma^*$, $\hat{\delta}(q_0, x) = p$ and $\hat{\delta}(q_0, y) = q \Rightarrow$ x $\equiv_L$ y.

Proof : (only if) Let $p \equiv q$, $\hat{\delta}(q_0, y) = p$ and $\hat{\delta}(q_0, y) = q$ for some $x, y \in \Sigma^*$. Now, for any $z \in \Sigma^*$ we have

$\hat{\delta}(q_0, xz) = \delta(p, z) \in F$ iff $\hat{\delta}(q_0, yz) = \delta(q, z) \in F$ (since, $p \equiv q$)

So, $xz \in L$ iff $yz \in L \Rightarrow x \equiv_L y$.

(if) Let $p \not\equiv q$, $\hat{\delta}(q_0, x) = p$ and $\hat{\delta}(q_0, y) = q$.

Hence, $\exists z \in \Sigma^*$ s.t.

$\hat{\delta}(p, z) \in F$ and $\hat{\delta}(q, z) \notin F$ ($\because p \not\equiv q$)

Hence $\hat{\delta}(q_0, xz) = \delta(p, z) \in F$ and $\hat{\delta}(q_0, yz) = \delta(q, z) \notin F$.

This implies, $xz \in L$ and $yz \notin L$

so, $x \not\equiv_L y$.

4.5 A MINIMIZATION ALGORITHM :

We now produce an algorithm to construct the minimal state DFA from any given DFA accepting L by merging states inductively.

The algorithm assume that all states are reachable from the start state i.e. there is no inaccessible states. The algorithm keeps on marking pairs of states (p, q) as soon as it determines that p and q are distinguishable i.e. $p \not\equiv q$. The pairs are, of course, unordered i.e. pairs (p, q) and (q , p) are considered to be identical. The steps of the algorithm are given below.

1. For every p, q $\in$ Q , initially unmark all pairs (p, q).
2. If $p \in F$ and $q \notin F$ (or vice versa) then mark (p, q).
3. Repeat the following step until no more changes occur : If there exists an unmarked pair (p, q) such that $(\delta(p, a), \delta(q, a))$ is marked for some $a \in \Sigma$, then mark (p, q).
4. $p \equiv q$ iff (p, q) is unmarked.

The algorithm correctly computes all pairs of states that can be distingusihed i.e. unmarked.

It is easy to show (by induction) that the pair (p, q) is mraked by the above algorithm iff $\exists x \in \Sigma^*$ s.t. $\hat{\delta}(p,x) \in F$ and $\hat{\delta}(q,x) \notin F$ (or vice versa) i.e. if $p \not\equiv q$.

Example : Let us minimize the DFA given below

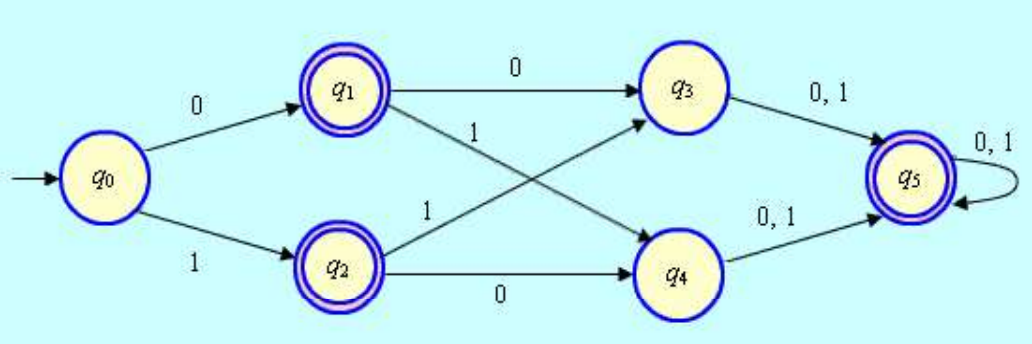


Figure 3

we execute the algorithm and mark a pair by putting an X on the table as shown in figure 4. (Note that the table is a diagonal one having $\binom{n}{2}$ entries for a DFA having n states.)

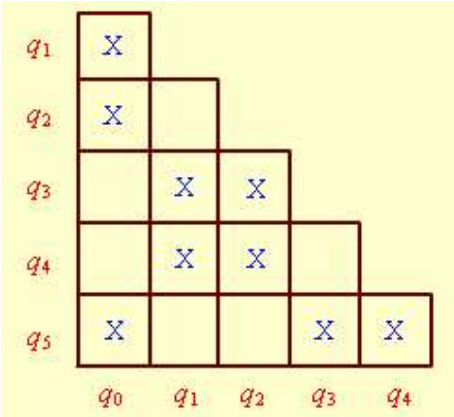


Figure 4

Initially, all cells are unmarked. (i.e. at step 1 of the algorithm) . After step 2, all cells representing pairs of states of which one is accepting and the other is non-accepting are marked by putting an X. The table above shows the status after this step.

In step 3, we consider all unmarked pairs one by one. Considering the unmarked pair (q_0, q_3), we find that q_0 & q_3 go to q_1 and q_5, respectively, on input 0. we use the notation $(q_0, q_3) \xrightarrow{0} (q_1, q_5)$ to indicate this. SInce the pair (q_1, q_5) is not marked, (q_0, q_3) cannot be marked at this point. Again, we see that, $(q_0, q_3) \xrightarrow{1} (q_2, q_5)$ and (q_2, q_5) is

unmarked. Hence, we cannot mark (q_0, q_3) and since we have considered all input symbols (0 & 1) we need to examine other unmarked pairs. The observations and actions are shown below.

- $(q_0, q_4) \xrightarrow{0} (q_1, q_5)$
- $(q_0, q_4) \xrightarrow{1} (q_2, q_5)$, cannot mark (q_0, q_4) since (q_1, q_5) & (q_2, q_5) are unmarked .
- $(q_1, q_2) \xrightarrow{0} (q_3, q_4)$
- $(q_1, q_2) \xrightarrow{1} (q_3, q_4)$, cannot mark (q_1, q_2) since (q_3, q_4) is unmarked .
- $(q_1, q_5) \xrightarrow{0} (q_3, q_5)$, (q_1, q_5) is marked since (q_3, q_5) is already marked.
- $(q_2, q_5) \xrightarrow{0} (q_4, q_5)$, (q_2, q_5) is marked since (q_3, q_5) is already marked.
- $(q_3, q_4) \xrightarrow{0} (q_5, q_5)$, (q_5, q_5) is never marked since it is not in the table & hence (q_3, q_4) is not marked.
- $(q_3, q_4) \xrightarrow{1} (q_5, q_5)$
- The resulting table after this pass is given below.

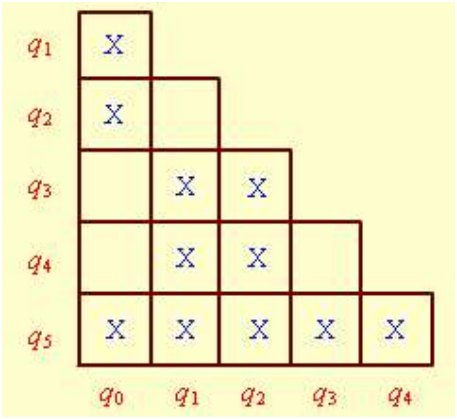


Figure 5

- In the next pass we find that $(q_0, q_3) \xrightarrow{0} (q_1, q_5)$ and (q_1, q_5) is marked in the previous pass .Hence, (q_0, q_3) can be marked now.
- Similarly, $(q_0, q_4) \xrightarrow{1} (q_2, q_5)$ and hence (q_0, q_4) can be marked since (q_2, q_5) has been marked in the previous pass. Other pairs cannot be marked and the resulting table is shown below. By executing step 3 again we observe that no more pairs can be marked and hence the algorithm stops with this table as the final result.
- The unmarked pairs left in the table after execution of the algorithm are (q_1, q_2) and (q_3, q_4) implying $q_1 \equiv q_2$ and $q_3 \equiv q_4$. Now, we merge q_1 & q_2 and q_3 & q_4 to have new states q_{12} & q_{34}, respectively.
- Transitions are adjusted appropriately to obtain the following minimal DFA.

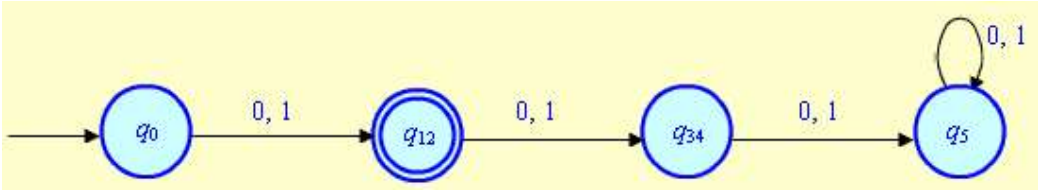


Figure 6

- q_{12} is a final state, since both q_1 & q_2 were final states. Similarly q_{34} is a non-final state.

q_0 goes to q_{12} on input 0 and 1, since q_0 go to q_1 and q_2 respectively on 0 and 1.Similar, justifications suffice for other adjusted transitions.

CHECK YOUR PROGRESS

True/False type questions

1)For any regular language L it may be possible to design different DFAs to accept L._____

2) For any regular language L there is a unique (upto isomorphism as defined) DFA ._____

3) Moore and Mealy machines both produces output_____

4) A Mealy machine is a four-tuple_____

5) Two DFAs are said to be isomorphism if they are not identical upto renaming of the states._____

Answers-

1) True

2) True

3) True

4) False

5) False

4.6 SOME DECISION PROPERTIES OF REGULAR LANGUAGES

At this point we would like to find out answers to some important questions related to regular languages. The questions we consider here all have answers which may be either “yes” or “no”. These are known as decision problems since we used to decide whether the answer is “yes” or “no”. [The reason for considering decision problems is that a regular language is recognized by a FA, which, in response to an input string, either ‘accepts' or ‘rejects' the input string and can be considered as producing “yes” or no “answers”, respectively.]

Consider the following typical and important question:

w and a regular language *L* , is an element of *L* ?

The answer is either yes or no.

While *w* is represent explicitly, we wonder how *L* given to us. Obviously, *L* cannot be given as an enumeration of strings (*L* may be infinite). *L* will be represented either by a *DFA* , *NFA* or regular expression.

The question presented above is called the “membership problem” for the corresponding regular language *L*.

If *L* is represented by a *DFA* , the problem has an easy solution-

- Simulate the DFA on input w
- If the DFA ends in an accepting state, the answer is “yes”. Otherwise, the answer is “no”.

The algorithm is very efficient and it can easily be verified that it takes linear time on the length of the input w

If L is given as an NFA , we can first convert it to an equivalent DFA and than use the above algorithm to find the answer. This is not efficient, since the conversion algorithm from NFA to DFA (by using subset constructions) is expensive.

Similarly, if L is expressed by using a regular expression, we can first convert it to an NFA and than use the above algorithm. We see that this is also an expensive method.

We will consider some more decision problems related to regular languages as given below.

- Given a FA M , is L(M) empty?
- Given a FAM, is L(M) infinite?
- Given two FA s M1 and M2 , do they accept the same language? That is, whether L(M1)=L(M2)?

The list is not extensive. We will consider decision algorithm for the above mentioned problem only.

It is interesting to note that we can use the pumping lemma to determine whether the language accepted by a DFA is empty or infinite. The following theorem states this result.

Theorem : If M is a DFA with n states, than the language accepted by M (i.e. L(M)) is

1. non empty if, and only if, M accepts some string w with $|w| < n$
2. infinite if, and only if, M accepts some string w such that $n \leq |w| < 2n$

Proof:

3. If M accepts a string w with $|w| < n$, then L(M) is clearly non empty. Conversely, let L(M) be non empty, and let w be the shortest string accepted by M. Then it must be the case that $|w| < n$. Otherwise, according to the pumping lemma w can be decomposed as w=xyz satisfying all the three constraints of the pumping lemma. So,

$xy^i z \in L(M), \forall i \geq 0$ For the case i=0, the string $xy^0 z = xz$ is a string which is shorter than w (since $y \neq \in$)

This contradicts that w is the shortest string accepted by M. Hence, $|w| < n$.

- Let M accept a string w with $n \leq |w| < 2n$. Then by pumping lemma w can be decomposed as w=xyz satisfying all the three constraints of the pumping lemma. Hence

  $\forall i \geq 0, xy^i z \in L(M)$

Therefore, L(M) must be infinite.

Conversely, let L(M) be infinite, and let w be the shortest string accepted by M whose length is at least n i.e. w $|w| < n$. (Note that such a string must exist, since L(M) is infinite and there are only a finite number of strings of length less than n). Then, it must be the case that, $n \leq |w| < 2n$. Otherwise (i.e. if $|w| > 2n$, by the pumping lemma we can decompose w as w=xyz satisfying all the constrains of the pumping lemma. So,

$\forall i \geq 0, xy^i z \in L(M)$. For i=0, in particular, $xy^0 z = xz \in L(M)$ is a shorter string than w (since $y \neq \in$), leading to a contradiction. Hence, $n \leq |w| < 2n$.

This theorem gives us the following naive algorithm to determine the emptiness and finiteness of a language L(M) accepted by a DFA M .

Algorithm to decide emptiness

- Run M on all strings of length less than n , where n is the number of states.
- If M accepts any of these, than L(M) is nonempty. Otherwise, L(M) is empty. (From part (1) of the theorem).

But the algorithm is highly inefficient, since the DFA M may have to check all the strings of length less than n and there are $0(|\Sigma|^n)$ strings of such length.

Algorithm to decide finiteness of L(M) .

- Run M on all strings of length between n and 2n

If M accepts any string of these, then L(M) is infinite. Otherwise, L(M) is finite.(From part (2) of the theorem)

Once again, we observe that the algorithm is highly inefficient (i.e. experimental)

But, efficient algorithms exists to decide these problem. We know that a DFA can be represented by a directed graph and for a DFA to accept a string there must exist a path from the start state to any final state. Using this fact, we have the following efficient algorithm to decide emptiness. (Assume, DFA M is given as a directed graph)

- Do DFA from the start state q0
- If any of the final state is reachable from the start state q0 , than L(M) is nonempty. Otherwise, L(M) is empty

We now consider an efficient algorithm to determine whether L(M) is infinite.

We know that all the states which are not reachable from q0 can be detected (along with the associated transition) without changing the accepted language.

Similarly, the accepted language does not change if all the states that cannot lead to an accepting state (also called ‘trap' states) are detected.

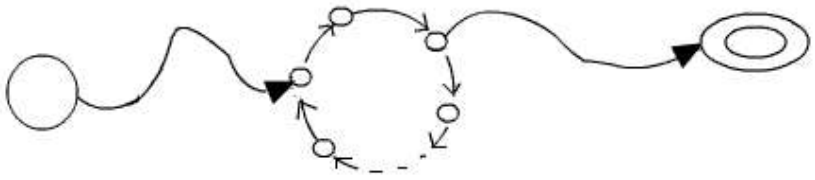
Claim1 : If *L(M)* is infinite, then there must exist a cycle in the directed graph.

Proof : Since *L(M)* is infinite, according to the previous theorem, there exists a string $w \in L(M)$ with $n \le |w| < 2n$ where *n* is the number of states in the *DFA M* . Since the length of the accepted string *w* is greater than the number of states, there must exist a repeated state in the path from *q0* to the final state while processing the string *w*. His repetition of (at least one) state in the path implies the existence of a cycle.

Claim 2: If there is a cycle in the directed graph (for the *DFA M*), then *L(M)* must be infinite.

Proof : We know that all states are reachable from the start state *q0*. Also, there can not be any cycle involving “useless” states, because these have already been removed.

Hence if there exists a cycle, there must be a path from the start state *q0* to one of the states involved in the cycle and, also, there must be a path from on e of the states involved in the cycle to an accepting state. The situation is depicted in the following figure.



So, clearly, starting at , than following the cycle infinitely many times, the DFA can accept infinitely many strings.

Hence, L(M) is infinite.

It is a well-known fact that there exists efficient algorithm to detect a cycle in directed graph. From the above, we have the following efficient algorithm to decide infiniteness of L(M).

- Delete all states not reachable from the start state and delete all states that cannot lead to an accept state (DFS can be used for this).
- If there is a cycle, then L(M) is infinite. Otherwise, L(M) is finite.

It is observed that using the decision algorithm for emptiness and finiteness together with closure properties we can find more decision algorithms. Here is an example.

Example : Given *DFA* s M1 and M2. Is *L(M1) = L(M2)*?

Solution : Observe that $L(M_1) \subseteq L(M_2)$ and $L(M_1) \cap \overline{L(M_2)} = \phi$

Thus *L(M1) = L(M2),* iff $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$ This implies that

L(M1) = L(M2) iff $\left(L(M_1) \cap \overline{L(M_2)}\right) \cup \left(L(M_2) \cap \overline{L(M_1)}\right) = \phi$

Since regular languages are closed under union, intersection and complement, we can construct a *DFA* M3 recognizing the language $\left(L(M_1) \cap \overline{L(M_2)}\right) \cup \left(L(M_2) \cap \overline{L(M_1)}\right)$

If *M3* accepts any string (i.e. $L(M_3) \neq \phi$) then $L(M_1) \neq L(M_2)$) .Otherwise , *L(M1) = L(M2)*

We can use emptiness algorithm to decide if $L(M_3) \neq \phi$)

4.7 FINITE AUTOMATA WITH OUTPUT

The definition of FA that we have already considered allows only two possible outputs is response to an input string, accept or reject. The definition can be extended so that the output can be chosen from some alphabet. Considering two different approaches to associate the output we have two different types of machines in the category- Moore machines and Mealy machines (They are named after the inventors). In a Moore machine the output is associated

with the state, whereas in a Mealy machine the output is associated with the transition. Even though the two models look different, we can prove that they are equivalent.

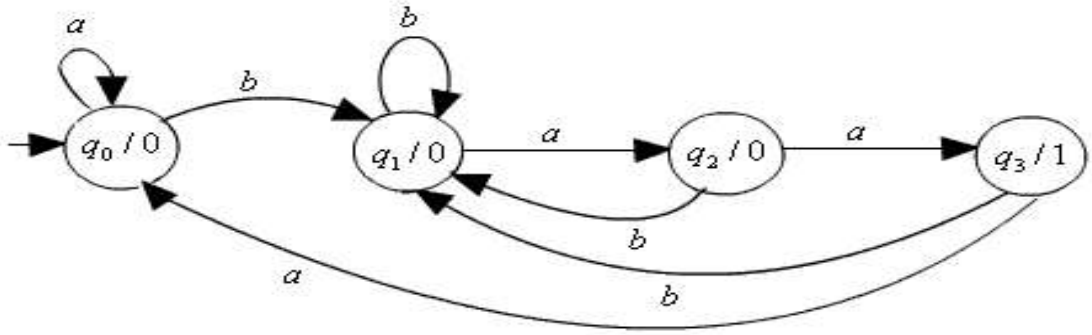
4.7.1 MOORE MACHINES :

A Moore machine is a six-tuple

$M_r = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where Q, Σ, δ and q0 are as in DFA . Γ is the output alphabet and $\lambda : Q \to \Gamma$, is a mapping which gives the output associated with each state. Note that there is no final state and the input and output alphabet need not to be same.

et the sequence of states the machine goes through in response to the input sequence $w = a_1 a_2 \cdots a_m, m \geq 0$ is $q_0, q_1, \cdots, q_m$. Then the output produced by the machine in response to this input $w = a_1 a_2 \cdots a_m$ is defined as $\lambda(q_0)\lambda(q_1)\cdots\lambda(q_m)$ .Note that a Moore machine produces an output without taking any input on state q0. That is, $\lambda(q_0)$ is the output in response to input $\in$. Hence, the length of the output string is always one more than that of the input string.

Example 1: Suppose we wish to determine exactly low many times the sub string baa occurs in the input string. The Moore machine presented by the given transition diagram



Keeps count of this number.

Note that, a state *p* here is annotated with p/a if it the output symbol 'a' is associated with the state *p* i.e. if $\lambda(p) = a$.

Every state outputs 0 except the state q3 which outputs a 1 start state q0, following any path, we arrive at state q3 the last three input symbols read must be $b, a,$ and a . As soon as we arrive at q3, it outputs a 1(prior to that it outputs all 0s) indicating that it has read the sub string baa in the input. From q3 we can arrive at q1 on input b and then again arrive at q3(following some path) provided the last three input symbols read are $b, a,$ and a . Thus,

the machine outputs a 1 as soon as it read the sub string baa ; otherwise, it outputs 0s. So, the number of sub string baa in the input is given by the number of 1s in the output string at the point when the machine finishes processing the input string.

For example, on input $abbabaaababaab$ the machine will go through the states $q_0q_0q_1q_1q_2q_1q_2q_3q_0q_1q_2q_1q_2q_3q_1$ producing the output sequence 0000000100000010 indicating that the sub string baa occurs twice in the input string as the number of 1s in the output string is 2.

he Moore machine can also be represented by a table, where the table to represent the transition (δ) remains same as in FA , but there is a separate column (separated by a double line) to represent the output associated with each state. The tabular form of the Moore machine of the above example is given below.

	a	b	
q_0	q_0	q_1	0
q_1	q_2	q_1	0
q_2	q_3	q_1	0
q_3	q_0	q_1	1

A Moore machine does not define a language of accepted strings, because in response to any input string it produces an output string and there is no concept of final states. The processing of the input string terminates when it outputs the symbol corresponding to the last input symbol.

For a given FA M , accepting the language L(M), if we associate 0 to any nonaccepting state and 1 to each accept state, then the 1's in any output sequence (produced in response to some input sequence) mark the ending of all sub strings of the input starting from the first symbol that are in L(M).

rom this, we can consider *FA* to be a special case of a Moore machine where the output alphabet $\Gamma = \{0,1\}$ and a state *p* is 'accepting' if and only if $\lambda(p) = 1$.

So, a Moore machine can be said to recognize the language of all input strings whose outputs ends in a 1. In the example Moore machine given above if we make *q3* as final state and remove all outputs associated with the states, it will be a *DFA* accepting all string over $\{a,b\}$ that ends with baa.

4.7.2 MEALY MACHINES :

A Mealy machine is a six-tuple, $M_e = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$, where all elements are as in Moore machine, except for λ which is defined as

$$\lambda : Q \times \Sigma \rightarrow \Gamma$$

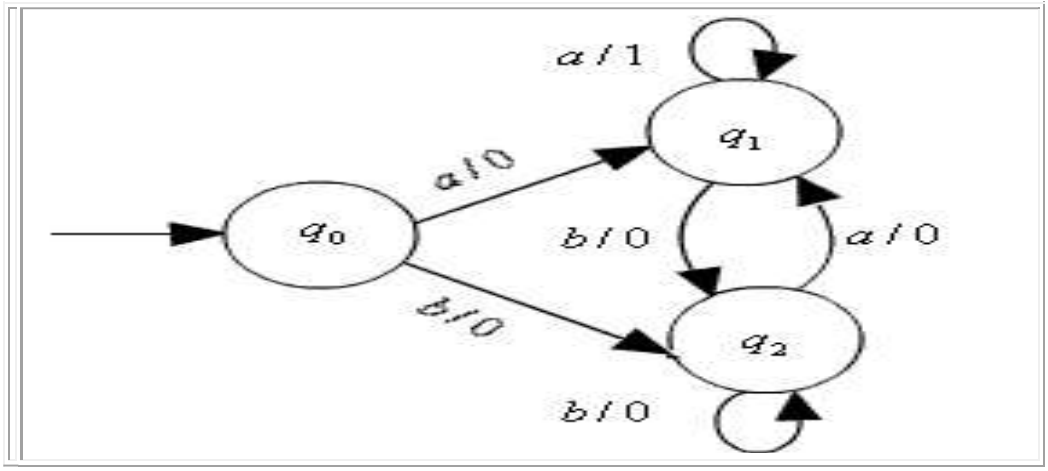
This means that $\lambda(q,a)$ gives the output associated with the transition from state q on input a.
Let the sequence of states the machine goes through in response to the input string $w = a_1 a_2 \cdots a_m \quad m \geq 0$ is $q_0, q_1, \cdots q_{m-1}, q_m$.

Then the output produced by the machine in response to this input $w = a_1 a_2 \cdots a_m$ is defined by $\lambda(q_0 a_1)\lambda(q_1 a_2)\cdots\lambda(q_{m-1}, a_m)$. The length of the output sequence unlike for the Moore machine.

Example:2 Consider the Mealy machine given below.

The machine outputs a 1 in the output string in response to some input string to indicate two consecutive occurrences of a in the input.

For example, the out put corresponding to the input $abbaaabaabb$ is 00001100100.



We can express the Mealy machine in tabular form as indicated below.

The entry b/0 for the raw q1 & column q2indicates that there is a transition from state q1 to state q2 on input b and the output associated with this transition is 0. For no transition defined from state p to state q the entry for raw p & column q will be ϕ

	q_0	q_1	q_2
q_0	ϕ	a /0	b/0
q_1	ϕ	a /1	b/0
q_2	ϕ	a /0	b/0

4.8 EQUIVALENCE OF MOORE AND MEALY MACHINES

Since Moore and Mealy machines both produces output (instead of normal convention of accepting a language by a FA). We can compare them in the sense that they are equivalent if they always produce the same output string in response to the same input string. But there can never be an exact match between the output strings produced by them since the length of the output string of a Moore machine is always one more than that of a Mealy machine in response to the same input string. However, if we ignore the response of a Moore machine for its initial state (i.e. response to input $\in$), then we can define the equivalence of a Moore machine, M_r and a Mealy machine M_e by saying that if for all input string $w, M_r(w) = aM_e(w)$, where a is the output of M_r for its initial state and $M_r(w), M_e(w)$ are outputs of M_r and M_e on w respectively. Then they are equivalent

he following two theorems prove the equivalence of Moore and Mealy machines in this sense.

Theorem : If $M_r = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ is a Moore machine, then there is a Mealy machine M_e and M_r .

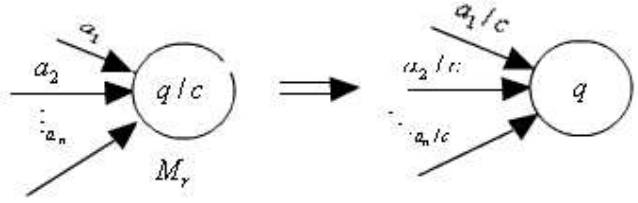
Proof :

We construct a Mealy machine

$M_e = (Q, \Sigma, \Gamma, \delta, \lambda_e, q_0)$ from the given Moore machine M_r , where all the elements except λ_e are as in M_r . λ_e is defined as

$$\lambda_e(q, a) = \lambda(\delta(q, a))$$

That is, the output associated with state *q* in the Moore machine will be associated with the transitions going to the state *q* (from other state) on the same input symbol a in the Mealy machine as shown below –



Now, for any given input string $w = a_1 a_2 \cdots a_m$. If M_r goes through a sequence of states $q_0 q_1 q_2 \cdots q_m, s.t. \delta(q_{i-1}, a_i) = q_i$ then it produces the output sequence $M_r(w) = \lambda(q_0)\lambda(q_1)\cdots\lambda(q_m)$. According to the construction, the Mealy

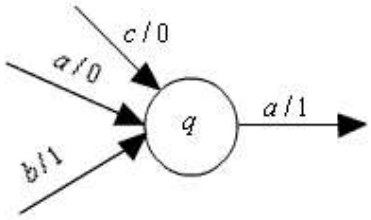
machine M_e also goes through the same sequence of states but produces the following output sequence.

$$\begin{aligned} M_e(w) &= \lambda'(q_0,a_1)\lambda'(q_1,a_2)\cdots\lambda'(q_{m-1},a_m) \\ &= \lambda\big(\delta(q_0,a_1)\big)\lambda\big(\delta(q_1,a_2)\big)\cdots\lambda\big(\delta(q_{m-1},a)\big) \\ &= \lambda(q_1)\lambda(q_2)\cdots\lambda(q_m) \end{aligned}$$

Hence: $M_r(w) = \lambda(q_0)M_e(w)$, proving the equivalence as defined.

Note that, to construct an equivalent Moore machine from a Mealy machine we cannot adopt the reverse process given in the above constructions i.e. simply push the output associated with a transition to the state (where this, transition leads to) to be considered as the output produced by the state. This is because, there may be two transitions going to a state with different outputs associated with it as shown below.

This is an ambiguous situation as we are not sure which output symbol (0 or 1) is to be associated with the state q



This situation can be handled by creating copies of the state *q* for all different outputs associated with incoming transitions (keeping all other things same) as shown below.

Number of states are increase to remember different output symbols associated with moving transitions, and hence, states are considered to be order pairs in the Mealy machine. This construction is presented formally in the theorem given below.

Theorem : If $M_e = (Q,\Sigma,\Gamma,\delta,\lambda_e,q_0)$ is a Mealy machine, then there is a Moore machine M_r equivalent to M_e.

Proof : We construct a Moore machine

$$M_r = \big(Q\times\Gamma,\Sigma,\Gamma,\delta',\lambda',[q_0b_0]\big)$$

Where b0 is any symbol in Γ . Transition δ' is defined as

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

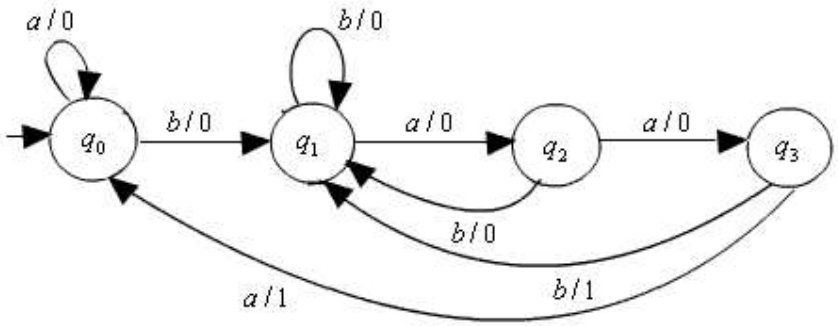
That is, the first component of M_e's state determines the moves of M_e and the second component of M_e is the output associated with some transition in M_r into the state q.

Output functions λ' of M_e is defined as $\lambda'([q,b]) = b$

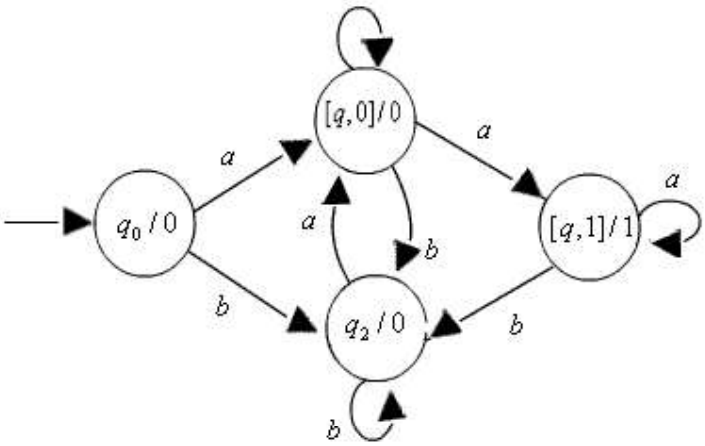
Following the construction, it can easily been shown that if M_e produces the output string $b_1 b_2 \cdots b_m$ in response to the input string $a_1 a_2 \cdots a_m$ after going through the states $q_0 q_1 \cdots, q_m$, then M_r also produces the same output string in response to the same input string after going through the states $[q_0,b_0],[q_1,b_1],\cdots[q_m,b_m]$

Example 3 : The equivalent Mealy machine for the Moore machine given in example 1 is produced below.

Example 3 : The equivalent Mealy machine for the Moore machine given in example 1 is produced below.



Example: 4 The Moore machine which is equivalent to the Mealy machine given is example 2 is shown below.



The states *[q1,0] & [q1,1]* an be renamed as *q1* & *q3* respectively.

4.9 CHECK YOUR PROGRESS

Fill in the blanks:

1) A Moore machine is a _____

2) Two DFA are isomorphic if they accept _____

3) For any regular language L there is a _____

4) Moore and Mealy machines both produces_____

5) If *L(M)* is infinite, then there must exist a _____ in the directed graph.

4.10 ANSWER CHECK YOUR PROGRESS

1) Six tuple

2) Bijection

3) Unique DFA

4) Output

5) Cycle

4.11 MODEL QUESTION

Qs-1) What is Deterministic Finite Automata (DFA) isomorphism explain with the help of example?

Qs-2) What is Equivalence of Moore and Mealy machines?

Qs-3) What is Mealy machine?

Qs-4) What is Moore machine? What do you understand by equivalence of moore and mealy machines

Qs-5) Write Algorithm to decide emptiness?

4.12 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

4.13 SUGGESTED READINGS

1. Martin J. C., "Introduction to Languages and Theory of Computations", TMH

2. Papadimitrou, C. and Lewis, C.L., "Elements of theory of Computations", PHI

3. Cohen D. I. A., "Introduction to Computer theory", John Wiley & Sons

4. Kumar Rajendra, "Theory of Automata (Languages and Computation)", PPM

Block-II

UNIT-V PUSHDOWN AUTOMATA

5.1 LEARNING OBJECTIVES

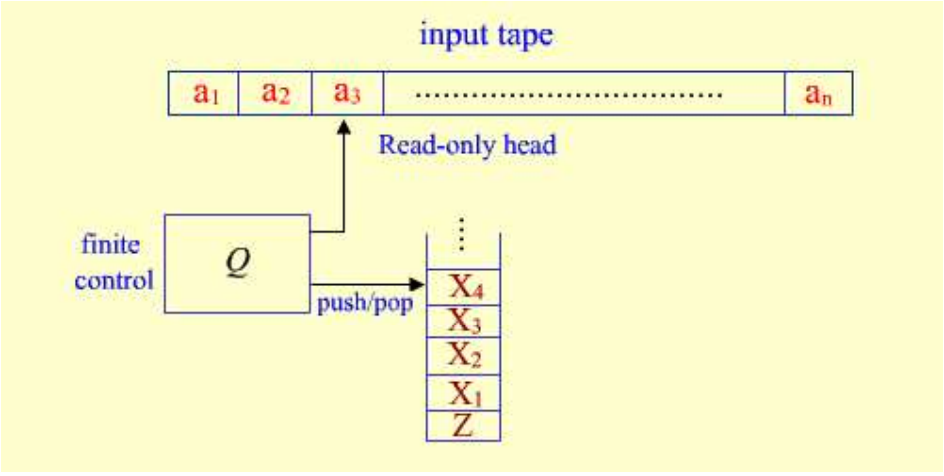
This chapter gives the basic understanding Pushdown Automata, Formal Definitions, Explanation of the transition function, Configuration or Instantaneous Description (ID), Nondeterministic Finite Automata (NFA), Language accepted by a PDA. We also understand Equivalence of PDAs and CFGs, CFA to PDA

.

5.2 PUSHDOWN AUTOMATA

Regular language can be characterized as the language accepted by finite automata. Similarly, we can characterize the context-free language as the language accepted by a class of machines called "Pushdown Automata" (PDA). A pushdown automation is an extension of the NFA.

It is observed that FA have limited capability. (in the sense that the class of languages accepted or characterized by them is small). This is due to the "finite memory" (number of states) and "no external memory" involved with them. A PDA is simply an NFA augmented with an "external stack memory". The addition of a stack provides the PDA with a last-in, first-out memory management cpapability. This "Stack" or "pushdown store" can be used to record a potentially unbounded information. It is due to this memory management capability with the help of the stack that a PDA can overcome the memory limitations that prevents a FA to accept many interesting languages like $\{a^n b^n \mid n \geq 0\}$. Although, a PDA can store an unbounded amount of information on the stack, its access to the information on the stack is limited. It can push an element onto the top of the stack and pop off an element from the top of the stack. To read down into the stack the top elements must be popped off and are lost. Due to this limited access to the information on the stack, a PDA still has some limitations and cannot accept some other interesting languages.



As shown in figure, a PDA has three components: an input tape with read only head, a finite control and a pushdown store.

The input head is read-only and may only move from left to right, one symbol (or cell) at a time. In each step, the PDA pops the top symbol off the stack; based on this symbol, the input symbol it is currently reading, and its present state, it can push a sequence of symbols onto the stack, move its read-only head one cell (or symbol) to the right, and enter a new state, as defined by the transition rules of the PDA.

PDA are nondeterministic, by default. That is, $\in$ - transitions are also allowed in which the PDA can pop and push, and change state without reading the next input symbol or moving its read-only head. Besides this, there may be multiple options for possible next moves.

5.2.1 FORMAL DEFINITIONS

Formally, a PDA M is a 7-tuple M = $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
where,

- Q is a finite set of states,
- Σ is a finite set of input symbols (input alphabets),
- Γ is a finite set of stack symbols (stack alphabets),
- δ is a transition function from $Q \times (\Sigma \cup \{\in\}) \times \Gamma$ to subset of $Q \times \Gamma^*$
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$, is the initial stack symbol, and
- $F^* \subseteq Q$, is the final or accept states.

5.2.2 EXPLANATION OF THE TRANSITION FUNCTION, δ :

If, for any $a \in \Sigma$, $\delta(q,a,z) = \{(p_1, \beta_1), (p_2, \beta_2), \cdots, (p_k, \beta_k)\}$. This means intitutively that whenever the PDA is in state q reading input symbol a and z on top of the stack, it can nondeterministically for any i, $1 \le i \le k$

- go to state p_i
- pop z off the stack
- push β_i onto the stack (where $\beta_i \in \Gamma^*$) (The usual convention is that if $\beta_i = X_1 X_2 \cdots X_n$, then X_1 will be at the top and X_n at the bottom.)
- move read head right one cell past the current symbol a.

- Final states are indicated by double circles and the start state is indicated by an arrow to it from nowhere.

5.3 CONFIGURATION OR INSTANTANEOUS DESCRIPTION (ID) :

- A configuration or an instantaneous description (ID) of PDA at any moment during its computation is an element of $Q \times \Sigma^* \times \Gamma^*$ describing the current state, the portion of the input remaining to be read (i.e. under and to the right of the read head), and the current stack contents. Only these three elements can affect the computation from that point on and, hence, are parts of the ID.
- The start or initalconfiguartion (or ID) on input w is (q_0, w, z_0). That is, the PDA always starts in its start state, q_0 with its read head pointing to the leftmost input symbol and the stack containing only the start/initial stack symbol, z_0.
- The "next move relation" one figure describes how the PDA can move from one configuration to another in one step.

Formally,

$(q, aw, z\alpha) \vdash_M (p, w, \beta\alpha)$

iff $(p, \beta) \in \delta(q, a, z)$

'a' may be $\in$ or an input symbol.

Let I, J, K be IDs of a PDA. We define we write $I \vdash_M^i K$, if ID I can become K after exactly i moves. The relations $\vdash_M^N$ and $\vdash_M^*$ define as follows

$I \vdash_N^0 K$

$I \vdash_N^{n+1} J$ if $\exists K$ such that $I \vdash_N^n K$ and $K \vdash_N^1 J$

$I \vdash_N^* J$ if $\exists\ n \geq 0$ such that $I \vdash_N^n J$.

5.4 NONDETERMINISTIC FINITE AUTOMATA (NFA)

That is, $\vdash_M^*$ is the reflexive, transitive closure of $\vdash_M$. We say that $I \vdash_N^* J$ if the ID J follows from the ID I in zero or more moves.

(Note : subscript M can be dropped when the particular PDA M is understood.)

5.4.1 LANGUAGE ACCEPTED BY A PDA M

There are two alternative definiton of acceptance as given below.

1. Acceptance by final state :

Consider the PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Informally, the PDA M is said to accept its input ϖ by final state if it enters any final state in zero or more moves after reading its entire input, starting in the start configuration on input ϖ.

Formally, we define L(M), the language accepted by final state to be

$$\{ \varpi \in \Sigma^* \mid (q_0, \varpi, Z_0) \vdash_N^* (p, \in, \beta) \text{ for some } p \in F \text{ and } \beta \in \Gamma^* \}$$

2. Acceptance by empty stack (or Null stack) : The PDA M accepts its input ϖ by empty stack if starting in the start configuration on input ϖ, it ever empties the stack w/o pushing anything back on after reading the entire input. Formally, we define N(M), the language accepted by empty stack, to be

$$\{ \varpi \in \Sigma^* \mid (q_0, \varpi, z_0) \vdash_N^* (p, \in, \in) \text{ for some } p \in Q \}$$

Note that the set of final states, F is irrelevant in this case and we usually let the F to be the empty set i.e. F = Q .

Example 1 : Here is a PDA that accepts the language $\{a^n b^n \mid n \geq 0\}$.

$M = (Q, \Sigma, \Gamma, \delta, q_1, Z, F)$

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

$\Sigma = \{a, b\}$

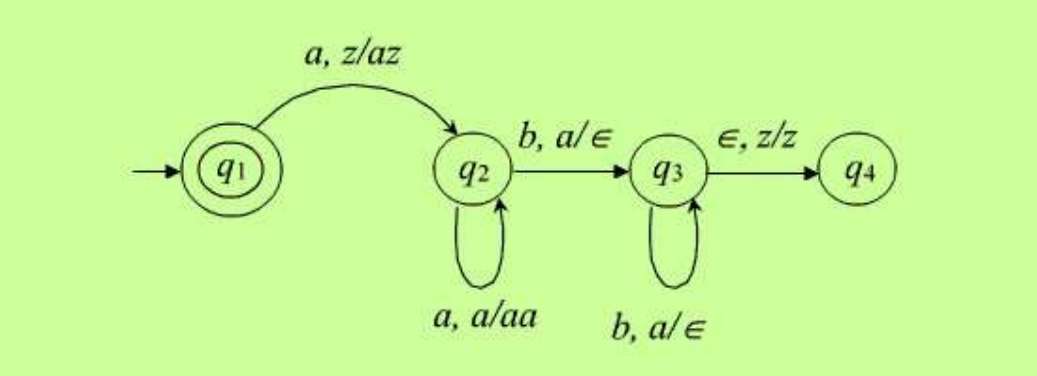
$\Gamma = \{a, b, z\}$

$F = \{q_1, q_4\}$, and δ consists of the following transitions

1. $\delta(q_1, a, z) = \{(q_2, az)\}$
2. $\delta(q_2, a, a) = \{(q_2, aa)\}$
3. $\delta(q_2, b, a) = \{(q_3, \in)\}$

4. $\delta(q_3, b, a) = \{(q_3, \in)\}$
5. $\delta(q_3, \in, z) = \{(q_4, z)\}$

The PDA can also be described by the adjacent transition diagram.



formally, whenever the PDA M sees an input a in the start state q_1 with the start symbol z on the top of the stack it pushes a onto the stack and changes state to q_2. (to remember that it has seen the first 'a'). On state q_2 if it sees anymore a, it simply pushes it onto the stack. Note that when M is on state q_2, the symbol on the top of the stack can only be a. On state q_2 if it sees the first b with a on the top of the stack, then it needs to start comparison of numbers of a's and b's, since all the a's at the begining of the input have already been pushed onto the stack. It start this process by popping off the a from the top of the stack and enters in state q3 (to remember that the comparison process has begun). On state q_3, it expects only b's in the input (if it sees any more a in the input thus the input will not be in the proper form of anbn). Hence there is no more on input a when it is in state q_3. On state q_3 it pops off an a from the top of the stack for every b in the input. When it sees the last b on state q3 (i.e. when the input is exaushted), then the last a from the stack will be popped off and the start symbol z is exposed. This is the only possible case when the input (i.e. on $\in$-input) the PDA M will move to state q_4 which is an accept state.

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.
Let the input be aabb. we start with the start configuration and proceed to the subsequent IDs using the transition function defined

$(q_1, aabb, z) \vdash (q_2, abb, az)$ (using transition 1)

$\vdash (q_2, bb, aaz)$ (using transition 2)

$\vdash (q_3, b, az)$ (using transition 3)

$\vdash (q_3, \in, z)$ (using transition 4)

$\vdash (q_4, \in, z)$ (using transition 5)

q_4 is final state. Hence ,accept. So the string aabb is rightly accepted by M.

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be aabab.

$(q_1, aabab, z) \vdash (q_2, abab, az)$

$\vdash (q_2, bab, aaz)$

$\vdash (q_3, ab, az)$

No further move is defined at this point.

Hence the PDA gets stuck and the string aabab is not accepted.

Example 2 : We give an example of a PDA M that accepts the set of balanced strings of parentheses [] by empty stack. The PDA M is given below.

$M = (\{q\}, \{[,]\}, \{z,[\}, \delta, q, z, \phi)$ where δ is defined as

$\delta(q,[,z) = \{(q,[z)\}$

$\delta(q,[,[) = \{(q,[[)\}$

$\delta(q,],[) = \{(q,\in)\}$

$\delta(q,\in,z) = \{(q,\in)\}$

Informally, whenever it sees a [, it will push the] onto the stack. (first two transitions), and whenever it sees a] and the top of the stack symbol is [, it will pop the symbol [off the stack. (The third transition). The fourth transition is used when the input is exhausted in order to pop z off the stack (to empty the stack) and accept. Note that there is only one state and no final state.

The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ].

$(q,[[\,][\,]][\,],z) \vdash (q,[\,][\,]][\,],[z) \vdash (q,\,][\,]][\,],[[z) \vdash (q,[\,]][\,],[z) \vdash (q,\,]][\,],[[z)$

$\vdash (q,\,]][\,],[[z) \vdash (q,\,][\,],[\,z) \vdash (q,[\,],z) \vdash (q,\,],[z) \vdash (q,\in,z) \vdash (q,\in,\in)$

Equivalence of acceptance by final state and empty stack.

It turns out that the two definitions of acceptance of a language by a PDA - accpetance by final state and empty stack- are equivalent in the sense that if a language can be accepted by empty stack by some PDA, it can also be accepted by final state by some other PDA and vice versa. Hence it doesn't matter which one we use, since each kind of machine can simulate the other.Given any arbitrary PDA M that accpets the language L by final state or empty stack, we can always construct an equivalent PDA M with a single final state that accpets exactly the same language L. The construction process of M' from M and the proof of equivalence of M & M' are given below.

There are two cases to be considered.

CASE I : PDA M accepts by final state, Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ Let qf be a new state not in Q. Consider the PDA $M' = (Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, z_0, \{q_f\})$ where δ' as well as the following transition.

$\delta'(q,\in,X)$ contains $(q_f, X) \quad \forall\ q \in F$ and $X \in \Gamma$. It is easy to show that M and M' are equivalent i.e.

L(M) = L(M')

Let $\omega \in$ L(M) . Then $(q_0, \omega, z_0) \vdash_M^* (q, \in, \gamma)$ for some $q \in F$ and $\gamma \in \Gamma^*$
Then $(q_0, \omega, z_0) \vdash_{M'}^* (q, \in, \gamma) \vdash_{M'}^* (q_f, \in, \gamma)$
Thus M' accepts ω

Conversely, let M' accepts ω i.e. $\omega \in$ L(M'), then $(q_0, \omega, z_0) \vdash_{M'}^* (q, \in, \gamma) \vdash_{M'}^1 (q_f, \in, \gamma)$ for $q \in F$ M' inherits all other moves except the last one from M. Hence $(q_0, \omega, z_0) \vdash_M^* (q, \in, \gamma)$ for some $q \in F$.
Thus M accepts ω. Informally, on any input M' simulate all the moves of M and enters in its own final state q_f whenever M enters in any one of its final status in F. Thus M' accepts a string ω iff M accepts it.

CASE II : PDA M accepts by empty stack.
We will construct M' from M in such a way that M' simulates M and detects when M empties its stack. M' enters its final state q_f when and only when M empties its stack.Thus M' will accept a string ω iff M accepts.
Let $M' = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{X\}, \delta', q_0', X, \{q_f\})$ where $q_0', q_f \notin Q$ and X$\in \Gamma$ and δ' contains all the transition of δ, as well as the following two transitions.
1. $\delta'(q_0, \in, X) = \{(q_0, z_0 X)\}$ and
2. $\delta'(q, \in, X) = \{(q_f, \in)\}, \quad \forall q \in Q$

Transitions 1 causes M' to enter the initial configuration of M except that M' will have its own bottom-of-stack marker X which is below the symbols of M's stack. From this point onward M' will simulate every move of M since all the transitions of M are also in M' .
If M ever empties its stack, then M' when simulating M will empty its stack except the symbol X at the bottom. At this point, M' will enter its final state q_f by using transition rule 2, thereby (correctly) accepting the input. We will prove that M and M' are equivalent.
Let M accepts ω . Then
$(q_0, \omega, z_0) \vdash_M^* (q, \in, \in)$ for some $q \in Q$. But then
$(q_0, \omega, X) \vdash_M^1 (q_0, \omega, z_0 X)$ (by transition rule 1)
$\vdash_M^* (q, \in, X)$ (Since M' includes all the moves of M)
$\vdash_M^1 (q_f, \in, \in)$ (by transition rule 2)
Hence, M' also accepts ω .

Conversely, let M' accepts ω .
Then $(q_0', \omega, X) \vdash_M^1 (q_0, \omega, z_0 X) \vdash_M^* (q, \in, X) \vdash_M^1 (q_f, \in, \in)$ for some $q \in Q$
Every move in the sequence
$(q_0, \omega, z_0 X) \vdash_M^* (q, \in, X)$ were taken from M.

Hence, M starting with its initial configuration will eventually empty its stack and accept the input i.e.

$(q_0, w, z_0) \vdash_M^* (q, \in, \in)$

5.4.2 EQUIVALENCE OF PDAS AND CFGS

We will now show that pushdown automata and context-free grammars are equivalent in expressive power, that is, the language accepted by PDAs are exactly the context-free languages. To show this, we have to prove each of the following:

i) Given any arbitrary CFG G there exists some PDA M that accepts exactly the same language generated by G.

ii) Given any arbitrary PDA M there exists a CFG G that generates exactly the same language accepted by M.

CHECK YOUR PROGRESS

True/False type questions

1) A pushdown automation is an extension of the NFA._____.

2) FA have limited capability._____.

3) The langauge accepted by a class of machines called Pushdown Automata_____.

4) PDA does not accepts by empty stack._____.

5) A PDA has Five components_____.

Answers-

1) True

2) True

3) True

4) False

5) False

5.5 CFA to PDA

We will first prove that the first part i.e. we want to show to convert a given CFG to an equivalent PDA.

Let the given CFG is $G=(N,\Sigma,P,S)$. Without loss of generality we can assume that G is in Greibach Normal Form i.e. all productions of G are of the form .

$A\to cB_1B_2\cdots B_k$ where $c\in\Sigma\cup\{\varepsilon\}$ and $k\geq 0$.

From the given CFG G we now construct an equivalent PDA M that accepts by empty stack. Note that there is only one state in M. Let

$M=(\{q\},\Sigma,N,\delta,q,S,\phi)$, where

- q is the only state
- Σ is the input alphabet,
- N is the stack alphabet ,
- q is the start state.
- S is the start/initial stack symbol, and δ, the transition relation is defined as follows
  1.

  For each production $A\to cB_1B_2\cdots B_k\in P$, $(q,B_1B_2\ldots B_k)\in\delta(q,c,A)$.

We now want to show that M and G are equivalent i.e. L(G)=N(M). i.e. for any $w\in\Sigma^*$. $w\in L(G)$ iff $w\in N(M)$.

If $w\in L(G)$, then by definition of L(G), there must be a leftmost derivation starting with S and deriving w.

i.e. $S\underset{G}{\overset{*}{\Rightarrow}}w$

Again if $w\in N(M)$, then one sysmbol. Therefore we need to show that for any $w\in\Sigma^*$.

$S\underset{G}{\overset{*}{\Rightarrow}}w$ iff $(q,w,s)\vdash(q,\in,\in)$.

But we will prove a more general result as given in the following lemma. Replacing A by S (the start symbol) and γ by $\in$ gives the required proof.

Lemma For any $x,y\in\Sigma^*$, $\gamma\in N^*$ and $A\in N$, $A\underset{G}{\overset{n}{\Rightarrow}}x\gamma$ via a leftmost derivative iff $(q,xy,A)\vdash_M^n(q,y,\gamma)$.

Proof : The proof is by induction on n.

Basis : n = 0

$A \overset{0}{\underset{G}{\Rightarrow}} x\gamma$ iff $A = x\gamma$ i.e. $x = \in$ and $\gamma = A$

iff $(q, xy, A) = (q, y, \gamma)$

iff $(q, xy, A) \vdash_M^0 (q, y, \gamma)$

Induction Step :

First, assume that $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$ via a leftmost derivation. Let the last production applied in their derivation is $B \to c\beta$ for some $c \in \Sigma \cup \{\in\}$ and $\beta \in N^*$.

Then, for some $\omega \in \Sigma^*$, $\alpha \in N^*$

$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha \overset{1}{\underset{G}{\Rightarrow}} \omega c\beta\alpha = x\gamma$

where $x = \omega c$ and $\gamma = \beta\alpha$

Now by the indirection hypothesis, we get,

$(q, \omega cy, A) \vdash_M^n (q, cy, B\alpha)$ .....(1)

Again by the construction of M, we get

$(q, \beta) \in \delta(q, c, B)$

so, from (1), we get

$(q, \omega cy, A) \vdash_M^n (q, cy, B\alpha) \vdash_M^1 (q, y, \beta\alpha)$

since $x = \omega c$ and $\gamma = \beta\alpha$, we get

$(q, xy, A) \vdash_M^{n+1} (q, y, \gamma)$

That is, if $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$, then $(q, xy, A) \vdash_M^{n+1} (q, y, \gamma)$. Conversely, assume that $(q, xy, A) \vdash_M^{n+1} (q, y, \gamma)$ and let

$\delta(q, c, B) = (q, \beta)$ be the transition used in the last move. Then for some $\omega \in \Sigma^*$, $c \in \Sigma \cup \{\in\}$ and $\alpha \in \Gamma^*$

$(q, \omega cy, A) \vdash_M^n (q, cy, B\alpha) \vdash_M^1 (q, y, \beta\alpha)$ where $x = \omega c$ and $\gamma = \beta\alpha$.

Now, by the induction hypothesis, we get

$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha$ via a leftmost derivation.

Again, by the construction of M, $B \to c\beta$ must be a production of G. [Since $(q, \beta) \in \delta(q, c, B)$]. Applying the production to the sentential form $\omega B\alpha$ we get

$$A \overset{n}{\underset{G}{\Rightarrow}} \omega B\alpha \overset{1}{\underset{G}{\Rightarrow}} \omega c\beta\alpha = x\gamma$$

i.e. $A \overset{n+1}{\underset{G}{\Rightarrow}} x\gamma$

via a leftmost derivation.

Hence the proof.

Example : Consider the CFG G in GNF

S $\to$ aAB
A $\to$ a / aA
B $\to$ a / bB

The one state PDA M equivalent to G is shown below. For convenience, a production of G and the corresponding transition in M are marked by the same encircled number.

(1) S $\to$ aAB
(2) A $\to$ a
(3) A $\to$ aA
(4) B $\to$ a
(5) B $\to$ bB

$M = (\{q\}, \{a, b\}, \{S, A, B\}, \delta, q, S, \Sigma)$

We have used the same construction discussed earlier.

5.6 SOME USEFUL EXPLANATIONS :

Consider the moves of M on input aaaba leading to acceptance of the string.

Steps

1. $(q, aaaba, s) \overset{(1)}{\underset{M}{\vdash}} (q, aaba, AB)$
2. $\overset{(2)}{\underset{M}{\vdash}} (q, aba, AB)$
3. $\overset{(3)}{\underset{M}{\vdash}} (q, ba, B)$
4. $\overset{(4)}{\underset{M}{\vdash}} (q, a, B)$
5. $\overset{(5)}{\underset{M}{\vdash}} (q, \in, \in)$ Accept by empty stack.

Note : encircled numbers here shows the transitions rule applied at every step.

Now consider the derivation of the same string under grammar G. Once again, the production used at every step is shown with encircled number.

$$S \overset{(1)}{\underset{G}{\Rightarrow}} aAB \overset{(3)}{\underset{G}{\Rightarrow}} aaAB \overset{(2)}{\underset{G}{\Rightarrow}} aaaB \overset{(5)}{\underset{G}{\Rightarrow}} aaabB \overset{(4)}{\underset{G}{\Rightarrow}} aaaba$$

Steps $\rightarrow$ 1 $\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 4 $\rightarrow$ 5

Observations:

- There is an one-to-one correspondence of the sequence of moves of the PDA M and the derivation sequence under the CFG G for the same input string in the sense that - number of steps in both the cases are same and transition rule corresponding to the same production is used at every step (as shown by encircled number).
- considering the moves of the PDA and derivation under G together, it is also observed that at every step the input read so far and the stack content together is exactly identical to the corresponding sentential form i.e. <what is Read><stack> = <sentential form>

Say, at step 2,

Read so far = a
stack = AB
Sentential form = aAB

From this property we claim that

$(q, x, S) \vdash_M^* (q, \in, \alpha)$ iff $S \overset{*}{\underset{G}{\Rightarrow}} x\alpha$. If the claim is true, then apply with $\alpha = \in$ and we get

$(q, x, S) \vdash_M^* (q, \in, \in)$ iff $S \overset{*}{\underset{G}{\Rightarrow}} x$

or $x \in N(M)$ iff $x \in L(G)$ (by definition)

Thus N(M) = L(G) as desired. Note that we have already proved a more general version of the claim.

5.6.1 PDA and CFG

We now want to show that for every PDA M that accpets by empty stack, there is a CFG G such that L(G) = N(M)

we first see whether the "reverse of the construction" that was used in part (i) can be used here to construct an equivalent CFG from any PDA M.

It can be show that this reverse construction works only for single state PDAs.

- That is, for every one-state PDA M there is CFG G such that L(G) = N(M). For every move of the PDA M $(q, B_1 B_2 \cdots B_K) \in \delta(q, c, A)$ we introduce a production $A \rightarrow cB_1 B_2 \cdots B_K$ in the grammar $G = (N, \Sigma, P, S)$ where N = T and $S = z_0$.

we can now apply the proof in part (i) in the reverse direction to show that L(G) = N(M).

But the reverse construction does not work for PDAs with more than one state. For example, consider the PDA M produced here to accept the langauge $\{a^n b a^n \mid n \geq 1\}$

$M = (\{p, q\}, \{a, b\}, \{z_0, A\}, \delta, p, z_0, \phi)$

Now let us construct CFG $G = (N, \Sigma, P, S)$ using the "reverse" construction.

(Note $N = \{z_0, A\}$, $S = z_0$).

Transitions in M	Corresponding Production in G
$a, z_0 / A$	$z_0 \rightarrow aA$
$a, A / AA$	$A \rightarrow aAA$
$b, A / A$	$A \rightarrow bA$

$a,\ A/\in$ $A \to a$

We can drive strings like aabaa which is in the language.

$$s = z_0 \underset{G}{\Rightarrow} aA \underset{G}{\Rightarrow} aaAA \underset{G}{\Rightarrow} aabAA \underset{G}{\Rightarrow} aabaA \underset{G}{\Rightarrow} aabaa$$

But under this grammar we can also derive some strings which are not in the language. e.g

Therefore, to complete the proof of part (ii) we need to prove the following claim also.

Claim: For every PDA M there is some one-state PDA M' such that $N(M) = N(M')$.

It is quite possible to prove the above claim. But here we will adopt a different approach. We start with any arbitrary PDA M that accepts by empty stack and directly construct an equivalent CFG G.

5.6.2 PDA to CFG

We want to construct a CFG G to simulate any arbitrary PDA M with one or more states. Without loss of generality we can assume that the PDA M accepts by empty stack.

The idea is to use nonterminal of the form <PAq> whenever PDA M in state P with A on top of the stack goes to state q_0. That is, for example, for a given transition of the PDA corresponding production in the grammar as shown below,

And, we would like to show, in general, that $\langle pAq \rangle \overset{*}{\underset{G}{\Rightarrow}} w$ iff the PDA M, when started from state P with A on the top of the stack will finish processing w, arrive at state q and remove A from the stack.

But we have to consider the more general transition rule as shown below.

With this, we are now ready to give the construction of an equivalent CFG G from a given PDA M. we need to introduce two kinds of producitons in the grammar as given below. The reason for introduction of the first kind of production will be justified at a later point. Introduction of the second type of production has been justified in the above discussion.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, \phi)$ be a PDA. We construct from M a equivalent CFG $G = (N, \Sigma, P, S)$

Where

- N is the set of nonterminals of the form <PAq> for $p, q \in Q$ and $A \in \Gamma$ and P contains the follwoing two kind of production
- $S \to \langle q_0 z_0 q \rangle \quad \forall\, q \in Q$
- If $(q_1, B_1 B_2 \cdots B_n) \in \delta(q, a, A)$, then for every choice of the sequence $q_2, q_3, \cdots, q_{n+1}$, $q_i \in Q$, $2 \le i \le n+1$.
- nclude the follwoing production
- $\langle q_A\, q_{n+1} \rangle \to a \langle q_1 B_1 q_2 \rangle \langle q_2 B_2 q_3 \rangle \cdots \langle q_n B_n q_{n+1} \rangle$
- If n = 0, then the production is $\langle q_A\, q_1 \rangle \to a$.
- For the whole excercise to be meaningful we want
- $\langle q_A\, q_{n+1} \rangle \overset{*}{\underset{G}{\Rightarrow}} w$ means there is a sequence of transitions (for PDA M), starting in state q, ending in q_{n+1}, during which the PDA M consumes the input string w and removes A from the stack (and, of course, all other symbols pushed onto stack in A's place, and so on.)
- That is we want to claim that
- $\langle pAq \rangle \overset{*}{\underset{G}{\Rightarrow}} w$ iff $(p, w, A) \vdash (q, \in, \in)$
- If this claim is true, then let $p = q_0,\ A = z_0$ to get $\langle q_0\ z_0\ q \rangle \overset{*}{\underset{G}{\Rightarrow}} w$ iff $(q_0, w, z_0) \vdash (q, \in, \in)$ for some $q \in Q$. But for all $q \in Q$ we have $S \to \langle q_0 z_0 q \rangle$ as production in G. Therefore,
- $S \overset{1}{\underset{G}{\Rightarrow}} \langle q_0 z_0 q \rangle \overset{*}{\underset{G}{\Rightarrow}}$ iff $(q_0, w, z_0) \vdash (q, \in, \in)$ i.e. $S \overset{*}{\underset{G}{\Rightarrow}} w$ iff PDA M accepts w by empty stack or L(G) = N(M)
- Now, to show that the above construction of CFG G from any PDA M works, we need to prove the proposed claim.
- Note: At this point, the justification for introduction of the first type of production (of the form $S \to \langle q_0 z_0 q \rangle$) in the CFG G, is quite clear. This helps use deriving a string from the start symbol of the grammar.
- **Proof :** Of the claim $\langle PAq \rangle \overset{*}{\underset{G}{\Rightarrow}} w$ iff $(P, w, A) \vdash (q, \in, \in)$ for some $w \in \Sigma^*$, $A \in \Gamma$ and $p, q \in Q$

  The proof is by induction on the number of steps in a derivation of G (which of

course is equal to the number of moves taken by M). Let the number of steps taken is n.

- he proof consists of two parts: ' if ' part and ' only if ' part. First, consider the ' if ' part
- If $(P, w, A) \vdash (q, \in, \in)$ then $\langle PAq \rangle \overset{*}{\underset{G}{\Rightarrow}} w$.
- Basis is n =1

  Then $(P, w, A)\,(q, \in, \in)$. In this case, it is clear that $w \in \Sigma \cup \{\in\}$. Hence, by construction $\langle PAq \rangle \rightarrow w$ is a production of G.

5.6.3 INDUCTIVE HYPOTHESIS :

- $\forall i < n \;\; (P, w, A) \vdash (q, \in, \in) \Rightarrow \langle PAq \rangle \overset{*}{\underset{G}{\Rightarrow}} w$
- 5.6.4 INDUCTIVE STEP : $(P, w, A) \vdash (q, \in, \in)$
- For n >1, let w = ax for some $a \in \Sigma \cup \{\in\}$ and $x \in \Sigma^*$ consider the first move of the PDA M which uses the general transition $(q_1, B_1 B_2 \cdots B_n) \in \delta(p, a, A)(p, w, A)$ $= (p, ax, A) \vdash (q_1, x, B_1 B_2 \cdots B_n) \vdash (q, \in, \in)$. Now M must remove $B_1 B_2 \cdots B_n$ from stack while consuming x in the remaining n-1 moves.
- Let $x = x_1 x_2 \cdots x_n$, where $x_1 x_2 \cdots x_n$ is the prefix of x that M has consumed when B_{i+1} first appears at top of the stack. Then there must exist a sequence of states in M (as per construction) $q_2, q_3, \cdots q_n, q_{n+1}$ (with $q_{n+1} = p$), such that
- So, applying inductive hypothesis we get
- $\langle q_i B_i q_{i+1} \rangle \overset{*}{\underset{G}{\Rightarrow}} x_i$, $1 \le i \le n+1$. But corresponding to the original move $(p, w, A) = (p, ax, A) \vdash (q_1, x, B_1 B_2 \cdots B_n)$ in M we have added the following production in G.
- We can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

  i) Let the input be aabb. we start with the start configuration and proceed to the subsequent IDs using the transition function defined
- $(q_1, aabb, z) \vdash (q_2, abb, az)$ (using transition 1)

- $\vdash (q_2, bb, aaz)$ (using transition 2)
- $\vdash (q_3, b, az)$ (using transition 3)

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be aabab.

$(q_1, aabab, z) \vdash (q_2, abab, az)$

$\vdash (q_2, bab, aaz)$

$\vdash (q_3, ab, az)$

No further move is defined at this point.

Hence the PDA gets stuck and the string aabab is not accepted.

The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ].

$(q, [[\,][\,]][\,], z) \vdash (q, [\,][\,]][\,], [z) \vdash (q,][\,]][\,], [[z)$

$\vdash (q, [\,]][\,], [z) \vdash (q,]][\,], [[z) \vdash (q,][\,], [z)$

$\vdash (q, [\,], z) \vdash (q,], [z) \vdash (q, \in, z) \vdash (q, \in, \in)$

Equivalence of acceptance by final state and empty stack.

It turns out that the two definitions of acceptance of a language by a PDA - accpetance by final state and empty stack- are equivalent in the sense that if a language can be accepted by empty stack by some PDA, it can also be accepted by final state by some other PDA and vice versa. Hence it doesn't matter which one we use, since each kind of machine can simulate the other.Given any arbitrary PDA M that accpets the language L by final state or empty stack, we can always construct an equivalent PDA M with a single final state that accpets exactly the same language L. The construction process of M' from M and the proof of equivalence of M & M' are given below.

There are two cases to be considered.

CASE 1 : PDA M accepts by final state, Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Let q_f be a new state not in Q. Consider the PDA $M' = \left(Q \cup \{q_f\}, \Sigma, \Gamma, \delta', q_0, z_0, \{q_f\}\right)$ where δ' as well as the following transition.

$\delta'(q, \in, X)$ contains $(q_f, X) \quad \forall\ q \in F$ and $X \in \Gamma$. It is easy to show that M and M' are equivalent i.e. $L(M) = L(M')$.

Let $w \in L(M)$. Then $(q_0, w, z_0) \vdash_M^* (q, \in, \gamma)$ for some $q \in F$ and $\Gamma \in \Sigma^*$

Then $(q_0, w, z_0) \vdash_{M'}^* (q, \in, \gamma) \vdash_{M'}^* (q_f, \in, \gamma)$.

Thus M' accepts w.

Conversely, let M' accepts w i.e. $w \in L(M')$, then $(q_0, w, z_0) \vdash_{M'}^* (q, \in, \gamma) \vdash_{M'}^1 (q_f, \in, \gamma)$ for some $q \in F$. M' inherits all other moves except the last one from M. Hence $(q_0, w, z_0) \vdash_M^* (q, \in, \gamma)$ for some $q \in F$.

Thus M accepts w. Informally, on any input M' simulate all the moves of M and enters in its own final state q_f whenever M enters in any one of its final status in F. Thus M' accepts a string w iff M accepts it.

CASE 2 : PDA M accepts by empty stack.

we will construct M' from M in such a way that M' simulates M and detects when M empties its stack. M' enters its final state q_f when and only when M empties its stack.Thus M' will accept a string w iff M accepts.

Let $M' = (Q \cup \{q_0', q_f\}, \Sigma, \Gamma \cup \{X\}, \delta', q_0', X, \{q_f\})$ where $q_0', q_f \notin Q$ and $X \in \Gamma$ and δ' contains all the transition of δ, as well as the following two transitions.

1. $\delta'(q_0, \in, X) = \{(q_0, z_0 X)\}$ and

2. $\delta'(q, \in, X) = \{(q_f, \in)\}, \quad \forall q \in Q$

Transitions 1 causes M' to enter the initial configuration of M except that M' will have its own bottom-of-stack marker X which is below the symbols of M's stack. From this point onward M' will simulate every move of M since all the transitions of M are also in M'.

If M ever empties its stack, then M' when simulating M will empty its stack except the symbol X at the bottom. At this point, M' will enter its final state q_f by using transition rule 2, thereby (correctly) accepting the input. we will prove that M and M' are equivalent.

Let M accepts w. Then

$(q_0, w, z_0) \vdash (q, \in, \in)$ for some $q \in Q$. But then,

$(q_0, w, X) \vdash_{M'}^1 (q_0, w, z_0 X)$ (by transition rule 1)

$\vdash_{M'}^* (q, \in, X)$ (since M' include all the moves of M)

$\vdash_{M'}^{1} (q_f, \in, \in)$ (by transition rule 2)

Hence, M' also accepts ϖ.

Conversely, let M' accepts ϖ.

Then $(q_0', \varpi, X) \vdash_{M'}^{1} (q_0, \varpi, z_0 X) \vdash_{M'}^{*} (q, \in, X) \vdash_{M'}^{1} (q_f, \in, \in)$ for some Q .

Every move in the sequence

$(q_0, \varpi, z_0 X) \vdash_{M'}^{*} (q, \in, X)$ were taken from M.

Hence, M starting with its initial configuration will eventually empty its stack and accept the input i.e.

$(q_0, \varpi, z_0) \vdash_{M}^{*} (q, \in, \in)$.

5.7 CONCLUSION

This module explains about the basic understanding Pushdown Automata, Formal Definitions, Explanation of the transition function, Configuration or Instantaneous Description (ID), Nondeterministic Finite Automata (NFA), Language accepted by a PDA, Equivalence of PDAs and CFGs, CFA to PDA.

5.8 CHECK YOUR PROGRESS

Fill in the blanks:

1) A PDA has _____ components.

2) A_____ is an extension of the NFA. Pushdown automation

3) A PDA M is a 7-tuple M _____

4) _____can be used to record a potentially unbounded information.

5) The Grammer accepted by CFG is_____

5.9 ANSWER CHECK YOUR PROGRESS

1) Three

2) Pushdown automation

3) $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

4) "Stack" or "pushdown store"

5) PDA

5.10 MODEL QUESTION

Qs-1) What is PDA explain with example?

Qs-2) How many components are there in PDA explain?

Qs-3) How many tuples are there in PDA write all of them?

Qs-4) What are two language accepted by PDA explain?

Qs-5) What is the full form of CFG And PDA?

5.12 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

5.13 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-VI DETERMINISTIC PUSHDOWN AUTOMATA (PDA)

6.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Deterministic Pushdown Automata (DPDA) and Deterministic Context-free Languages (DCFLs) along with the concepts of DPDAs and FAs: DCFLs and Regular languages. It discusses CFLs and DCFLs, Standard forms of DPDAs, Acceptance by final state and empty stack. We also understand the deep insights of Unambiguous CFGs and DPDAs. Parsing and DPDAs are also elaborated in the chapter.

6.2 DETERMINISTIC PUSHDOWN AUTOMATA (DPDA) and DETERMINISTIC CONTEXT-FREE LANGUAGES (DCFLs)

Pushdown automata that we have already defined and discussed are nondeterministic by default, that is, there may be two or more moves involving the same combinations of state, input symbol, and top of the stock, and again, for some state and top of the stock the machine may either read and input symbol or make an $\in$- transition (without consuming any input).

In deterministic PDA, there is never a choice of move in any situation. This is handled by preventing the above mentioned two cases as described in the definition below.

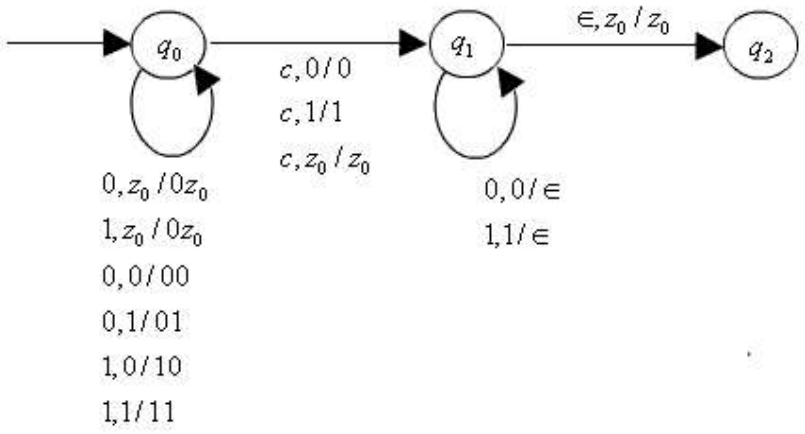
Definition: Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then M is deterministic if and only if both the following conditions are satisfied.

1. $\delta(q, a, X)$ has at most one element for any $q \in Q, a \in \Sigma \cup \{\in\}$, and $X \in \Gamma$ (this condition prevents multiple choice for any combination of q, a and X)
2. If $\delta(q, \in, X) \neq \phi$ and $\delta(q, a, X) = \phi$ for every $a \in \Sigma$

   (This condition prevents the possibility of a choice between a move with or without an input symbol).

A language L is said to be deterministic context-free (DCFL) if there is some DPDA accepting L.

Example: The language $L = \{wcw^t / w \in (0+1)^*\}$ is a DCFL. The following DPDA accepts L



The moves satisfying the conditions given in the definition. As the PDA reads the first half of the input, it remains in the start state q0and pushes the input symbols on the stock. When it reads the symbol c it changes the state from q0 to q1without changing the stock. On state q1 it simply matches input symbols with the stock symbols and erases in case of a match. That is, the symbol the moves satisfying the conditions given in the definition. As the PDA reads the first half of the input, it remains in the start state q0 and pushes the input symbols on the stock. When it reads the symbol c, it changes the state from q0 to q1 without changing the stock. On state q1 it simply matches input symbols with the stock symbols and erases in case of a match. That is, the symbol in wcw^R tells the m/c when to start looking for w^R Once the input is extended, then the symbol z0 on stock indicates a proper match for the input to be wcw^R and hence it accepts by entering state q2, which is a final state.

Example: Consider the language $L = \left\{ ww^R / w \in (0+1)^* \right\}$ In this case there is no way to determine when to start comparison because of absence of the symbol c in the middle/ The PDA in this case has to guess non-deterministically when the middle symbol comes in the input.

6.3 DPDAs and FAs: DCFLs and REGULAR LANGUAGES

Equivalence of DFA & NFA proves that non determination does not add power in case of FA s. But it is not true in case of PDA s, i.e., it can be shown that nondeterministic PDA s are more powerful than DPDA s. In fact, DCFL s is a class of languages that lies properly between the class of regular languages and CFL s. The following discussion proves this fact.

Theorem: If L is a regular language, then there is some DPDA M such that $L = L(M)$.

Proof: Since L is regular, then exists a DFA D such that $L = L(D)$.

The PDA M can be constructed from D (with an additional stock) that simulates all the moves of D on any input just by ignoring its stock. That is if $D=(Q,\Sigma,\{Z\},\delta',Z,F)$ when $\delta'(p,a,Z)=(q,Z)\forall p,q\in Q$ Such that $\delta(p,a)=q$ It is easy to see that $(q_0,w,Z)\overset{*}{\underset{M}{\mapsto}}(p,\in,Z)$ iff $\delta^*(q_0,w)=p$

Again, the language wcw^R can be shown to be non-regular by using pumping lemma. But, the DPDA presented in the example above accepts this language.

Hence the class of DCFL s properly includes the class of regular languages.

6.4 CFLs and DCFLs

We now show that class of languages accepted by DPDA s is properly included in the CFLS.

First, note that, every DCFL is a CFL since every DPDA is a special case of a PDA.

Now, there are two ways to prove the proper inclusion- direct method or indirect method.

In the direct method we need to show that there exists a language which is CFL but not DCFL. We have already argued intuitively that the language ww^R is a CFL but not a DCFL. We will show later that there is a CFL which is not a DCFL.

The indirect method follows the following steps to prove the fact-

First, prove the fact that DCFL s are closed under complement. But, it is fact that CFL 's are not closed under complement. Hence, there must exist a CFL that is not s DCFL .

6.5 STANDARD FORMS of DPDAs

It is possible to put every DPDA in some standard form where the only stack operations are to erase the top symbol without putting anything else on the stack; or to push a single symbol onto the stack on top of the symbol that was previously on top; or to leave the stack unchanged. The following two lemma establishes this fact.

Lemma 1: If L is a DCFL, then L=L(M) for some $DPDAM=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ such that if $\delta(q,a,X)=(p,\alpha)$, then $|\alpha|\leq 2$

Proof: Let the $DPDAM' = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$ accepts L and has a move $\delta'(q,a,X) = (p,\alpha)$ with $|\alpha| > 2$ The DPDAM simulates this move by using some more states and a sequence of moves as follows.

Let $\alpha = Y_1 Y_2 \cdots Y_n$ where $n \geq 3$

Let $p_1, p_2, \cdots p_{n-2}$ are nonaccepting states in M (which is not in M'). The move

$\delta'(q,a,X) = (p,\alpha)$ in M' is redefined as $\delta(q,a,X) = (p_1, Y_{n-1}, Y_n)$ in M.

Then the following moves are introduced in M.

$\delta(p_i, \in, Y_{n-i}) = (p_{i+1}, Y_{n-i-1} Y_{n-i})$ for $1 \leq i \leq n-3$ .finally introduce,

$\delta(p_{n-2}, \in, Y_2) = (p, Y_1 Y_2)$

That is, the DPDA M enters the state p after replacing X with $\alpha = Y_1 Y_2 \cdots Y_n$ (after starting at state q, on input a and with X on the top of the stack). But ---- it now takes a sequence of moves ($n-1$ number of moves to be precise) for the same.

Lemma 2: If L is a DCFL, then L=L(M) for some $DPDAM = (Q, \Sigma, \Gamma, \delta, q_0, Z_0 F)$ such that if $\delta(q,a,X) = (p,\alpha)$,then α is either $\in, X$ or of the form XY for some $Y \in \Gamma$

Proof: Let $L = L(M')$ for some DPDA.

$M' = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F')$

without loss of generality assume that M'

satisfies Lemma 1 given above. We now construct M from M' as follows.

$Q = Q' \times \Gamma'$
$q_0 = [q_0', Z_0']$
$F = F' \times \Gamma'$
$\Gamma = \Gamma' \bigcup \{Z_0\}$ and δ is defined by the following rules:

1. If M' pops its stock, then M pops its stock and remembers the symbol popped (in its finite control) by the move

$\delta([q,X],a,Y)=([p,Y],\in)\forall Y\in\Gamma$ if, $\delta'(q,a,X)=(p,\in)$ is in M'.

2. If M' changes its top symbol of the stock, then M remembers this without changing its top of the stock that is $\forall Y\in\Gamma$.

$\delta([q,X],a,Y)=([p,Z],Y)$ if $\delta(q,a,X)=(p,Z)$ is in M'.

3. M pushes a symbol onto its store whenever the stock size of M' increases, that is $\forall w\in\Gamma$

$\delta([q,X],a,w)=([p,Y],Zw)$ if $\delta'(q,a,X)=(p,YZ)$ is in M'

It can be shown by induction s that L(M)=L(M')

Theorem: The language $L=\{w\in\Sigma^*|w=w^R\}$ is not a DCFL.

Proof: Assume for contradiction that there is some DPDA $P=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ accepting L. Without loss of generality, we can assume that P is in standard form, i.e., every move of P is either of the form

$$\delta(p,a,X)=(q,\in)$$

or one of the forms

$$\delta(p,a,X)=(q,YX)$$

Where $a\in\Sigma\cup\{\in\},X,Y\in\Gamma$ and $p,q_2\in Q$ .Note that, for P to reject an input string w it may not read the whole string (it may enter a configuration at which no transition is defined or it may execute a never-ending sequence of $\in$-moves). On the other hand if P accepts a string w, then it must eventually read the whole string.

We know that if $x\in L,$ Thus xx^R is also in L.

Since P accepts both x and xx^R, the sequence of moves it makes while processing the first part x of the string xx^R must be exactly similar to that it makes on input x irrespective of whether x is followed by any other string or not. This is because of the fact that P is deterministic in nature.

After processing x at the stock content of P be γ with $|\gamma|=K$ for some $k>0$. Now, if P starts reading subsequent symbols from some string y (i.e. P may be assumed to start with the string xy) and finishes reading it, then let the stock content be α. We are sure that $|\alpha|>0$, since P must still be able to process some longer string with xy as the prefix. So, we have

$(q_0, xy, Z_0) \overset{*}{\underset{P}{\mapsto}} (t, \in, \alpha)$ for some $t \in Q$ and $\alpha \in \Gamma^*$ with $|\alpha| > 0$

In the above,

Consider the string y' such that the length of the resulting stack content is minimum i.e.

$(q_0, xy', Z_0) \overset{*}{\underset{P}{\mapsto}} (p, \in, \beta)$ for some $p \in Q, \beta \in \Gamma^*$, then $|\beta| \leq |\alpha|$

So, once P reaches the configuration $(p, \in, \beta)$ after processing xy', it cannot remove any symbol from the stock (since length β cannot be reduced further) in the subsequent moves. Because a move of a DPDA in standard from that involves removing a single symbol from the stack reduces the height of the stack.

Let $\beta = x\beta'$ for some $X \in \Gamma$ and $\beta' \in \Gamma^*$. Since we may consider any string $x \in \Sigma^*$ in xy' there are infinite number of strings of the form xy . But the set of states and stack symbols, Q and Γ, respectively of P are infinite. Hence there must exist two different strings u=xy' and v=xy' in Σ^*. Such that

$$(q_0, u, Z_0) \overset{*}{\underset{P}{\mapsto}} (p, \in, \beta) = (p, \in, X\beta')$$

and

$$(q_0, v, Z_0) \overset{*}{\underset{P}{\mapsto}} (p, \in, \beta) = (p, \in, X\beta')$$

We also know that the symbol X cannot be removed from the stack once P has entered this configuration.

Therefore, for some $Z \in \Sigma^*$, if we consider the strings uZ and vZ, then we must have

$$(q_0, uZ, Z_0) \overset{*}{\underset{P}{\mapsto}} (p, \in, X\beta') \quad \text{and}$$

$$(q_0, vZ, Z_0) \overset{*}{\underset{P}{\mapsto}} (p, \in, X\beta')$$

So, either both uZ and vZ are accepted or both are rejected by P. But since u and v are distinct, for some z one may be in L while other is not. This leads to a contradiction. (Since P should have accepted only one of these two, which is in L and the other should have been rejected.)

Hence our assumption that P accepts L must be false.

6.6 ACCEPTANCE BY FINAL STATE AND EMPTY STACK

We have already proved in case of NPDA that the two methods of acceptance (by empty stack and final state) are equivalent. That is, a language L has an NPDA that accepts by final state if and only if some NPDA accepts it by empty stack. But this is not true for DPDAs. The language recognizing capability of DPDA s that accept by empty stack is much less than that of the other. This is proved in the following lemma.

Lemma1: If a language L is accepted by a DPDA by empty stack, then L has the "prefix properly".

Before giving the proof of the above lemma we first define the "prefix properly" of a language.

Definition: A language L is said to have the prefix properly if whenever $x \in L$, no proper prefix of x is in L

Example: The language wcw^R has the prefix properly; since if $xcx^2 \in L$, then no proper prefix of xcx^2 can be in L. This is because the symbol c identifies the mid-point of the string xcx^2. In many proper prefixes of xcx^2, the symbol c will not be the mid-point of that prefix.

Again, consider the language $(a)^*$ It is quite obvious that there are infinitely many pairs of stings in $(a)^*$ one of which is a prefix of the other e.g., aa and $aaaa$ both are in $(a)^*$ and aa is a proper prefix of $aaaa$. This is a regular language and still not accepted by any DPDA by empty stack.

It is to be noted that prefix property is not a severe restriction. Because we can always introduce a special end marker, say $\$$, not in Σ at the end of every string of a language L to convert it to a language with prefix property. That is $L\$ = \{w\$ \mid w \in L\}$ is a language with prefix property.

Assume for contradiction that the language L accepted by the DPDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \phi)$ by empty stack does not have the prefix property. Hence, there must exist two strings x and xy (with $y \neq \in$) such that P accepts both. Then we have

$(q_0, x, Z_0) \overset{*}{\underset{P}{\mapsto}} (p, \in, \in)$, since $x \in L$.

So, while processing the string xy, the DPDA must arrive at the configuration given below because of its deterministic property.

$$(q_0, xy, Z_0) \overset{*}{\underset{P}{\mapsto}} (p, y, \in)$$

From the point onward the DPDAP cannot move since it has already emptied its stock and $y \neq \in$. Hence, xy is not accepted by P as assumed.

The lemma 2 given below shows that every language accepted by a DPDA by some DPDA that accepts by final state.

Lemma 2: If L is accepted by some DPDA P that accepts by empty stock, then there is some DPDA P'that accepts by final state such that L=L(p').

Proof: If $DPDA\ P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \phi)$ accepts L by empty stack.

$DPDA\ P' = (Q', \Sigma, \Gamma', \delta', q_0', Z_0', F)$ from P which simulate the behaviour of P as follows.

$Q' = Q \cup \{q_0', p'\}$ such that $q_0', p' \notin Q$.

$\Gamma' = \Gamma \cup \{Z_0'\}$

$F = \{p'\}$

δ' Contains all the moves of P and also the following.

1. $\delta'(q_0', \in, Z_0') = (q_0, \in, Z_0 Z_0')$
2. $\delta'(q, x, Z_0') = (p', Z_0')$

By using rule 1, P' simply enters the initial configurations of P pushing Z0above the bottom of stock marker Z0'. Then P' simulates the behaviour of P on any input string. When P accepts

a string, it empties its stock and at that point P0' would expose the bottom of stock marker Z0' and enters the final state P' by using rule 2. So, it is obvious that, an input string X is accepted by P iff it is accepted by P'.

The converse of lemma 2 is not necessarily true. But it can be shown that every language that has the prefix property and is accepted by a DPDA with final state is also accepted by some DPDA that accepts by empty stock, as given in the lemma 3.

Lemma 3: If a language L has the prefix property and is accepted by a DPDAP by final state, then there is some DPDA P' that accepts by empty stock such that L=L(P').

Proof: Let $L = L(P)$ for DPDA

$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ that accepts by final state. We construct P' from P as follows.

$\Gamma' = \Gamma \cup \{Z_0'\}$

P' contains all the moves of P besides the following.

1. The first move of P' is to go to the initial configuration of P by pushing the start symbol Z_0' of P'on top of the stock. From this point onward P' simulates the behavior of P (using P's moves) on any input string. Even if P empties its stack without accepting the input, P' will not empty its stock because of the new start symbol that was pushed on to the top of the stock initially.
2. If P enters an accepting state, P' simply enters the state P'.
3. On state P', the *DPDA* P' erases all the stock symbols without bothering the input eventually emptying its stock. So, P' accepts a string X where P accepts it and vice versa.

Now lemma 1,2,3 together gives us the following theorem.

Theorem: A language L is accepted by a DPDA by empty stock if and only if it has the prefix property and is accepted by some DPDA by final state.

CHECK YOUR PROGRESS

True/False type questions

1) Every DCFL is a CFL since every DPDA is a special case of a PDA._____

2) In deterministic PDA, there is never a choice of move in any situation_____.

3) A parser is an algorithm to determine whether a given string is in the language generated by a given CFG_____

4) Pushdown automata that we have already defined and discussed are deterministic by default_____

5) DPDA does involve backtracing_____

Answers-

1) True

2) True

3) True

4) False

5) False

6.7 UNAMBIGUOUS CFGs and DPDAs

It is interesting to note the language accepted by a DPDA must have an unambiguous grammar. We first prove it for a DPDA that accepts by empty stock and then extend it to a DPDA that accepts by final state.

Theorem: If L is accepted by some DPDAM that accepts by empty stock, then L must have an unambiguous CFG.

Proof: In the construction of an equivalent CFG G for any given DPDAM (that has been discussed in the context of equivalence of PDAs and CFGs) if assume that M is deterministic (that accepts by empty stock), then the resulting grammar G generated can be shown to have unique leftmost derivation for every string (thus, proving that G is unambiguous).

If M accepts a string w by empty stock, then because of deterministic nature of M there must be a unique sequence of moves and M cannot move once it empties its stock. If this sequence of moves is known, we can determine the exact choice of production rules in a leftmost derivation of w under G. Even though there may be many different rules in G for the move. $\delta(q,a,X)=(p,Y_1Y_2\cdots Y_n)$ of M, only one of those will be consistent with the execution of M that actually drive w.

We can now show that if L is accepted by some DPAM that accepts by final state, then L has an unambiguous grammar.

Consider the language for some symbol $\$$ which is not a terminal symbol of M. Since L' has the prefix property. It is accepted by a DPDAM' that accepts by empty stock and, thus, there exists an unambiguous CFG G' with L=L(G') (by the above theorem). We construct a CFG G from G' such that L=L(G) as follows.

G and G' are exactly same except that we introduce a new nonterminal $ and a new production $\$ \rightarrow \in$ in G. Now, if $w\$ \in L(G')$, then G derives the string $w \in L$ following exactly the same sequence of steps except at the last step, when G uses the production $\$ \rightarrow \in$ to get rid of the symbol $. Since G' is unambiguous, G must also be unambiguous.

6.8 PARSING and DPDAs

The context-free languages are of great practical importance, especially, in defining programming languages. For example, we can use CFGs to model the syntax of arithmetic expressions, block structures in programming languages, etc. A compiler for such a programming language must then embody a parser to carry out the process of analysing a given input string in order to determine its grammatical structure with respect to the given grammar. That is, a parser is an algorithm to determine whether a given string is in the language generated by a given CFG and, if so, to construct a purse tree for the string (for further use at a later stage).

We have already seen a cubic-time pursing algorithm (based on dynamic programming technique) that works for any given context-free language. For almost all practical purposes it is considered to be two slow. The most successful parser which have been developed in the recent past are based on the idea of a PDA. Since PDAs and CFGs are found to be equivalent one can develop a parser for CFLs that behave like PDAs. But because of the nondeterministic nature of PDAs, they are still not of immediate practical use in parsing. The parsing process may involve back tracking because of the nondeterministic steps and hence would lead to inefficiency.

On the other hand, a parser rooted in the idea of a DPDA does not involve backtrack----ing and hence expected to work efficiently. Even though the capability of DPDAs are limited in the sense that they accept DCFLs which is a proper subset of CFLs, it turns out that the syntax of most programming languages can be modelled by means of DCFLs. One of the main motivations for studying DCFLs lies in the fact that- they can describe the syntax of programming languages and they can be parsed efficiently using DPDAs. To produce a compiler for a given programming language the syntax is required to be described by some CFG in restricted form that generate only DCFLs. There are different kinds of such CFGs in restricted forms. The LL- and LR-grammars are two important classes in this category.

6.9 CONCLUSION

After reading this module you will know the Deterministic Pushdown Automata (DPDA) and Deterministic Context-free Languages (DCFLs) along with the concepts of DPDAs and FAs: DCFLs and Regular languages. It discusses CFLs and DCFLs, Standard forms of DPDAs, Acceptance by final state and empty stack. It presents the deep insights of Unambiguous CFGs and DPDAs. Parsing and DPDAs are also elaborated in the chapter.

6.10 CHECK YOUR PROGRESS

Fill in the blanks:

1) The full form of DPDA is_____

2) Every DCFL is a _____ since every DPDA is a special case of a _____.

3) A parser is an_____ to determine whether a given string is in the language generated by a given CFG.

4) The parsing process may involve _____ because of the nondeterministic steps and hence would lead to inefficiency.

5) NPDA accepts it by _____

6.11 ANSWER CHECK YOUR PROGRESS

1) Deterministic push down automata.

2) CFL and PDA.

3) Algorithm.

4) Back tracking.

5) Empty stack.

6.12 MODEL QUESTION

Qs-1) What is DPDA and DCFL? Explain their difference with suitable example.

Qs-2) What is UNAMBIGUOUS CFGs? Explain.

Qs-3) What do you understand by determinism? What are two necessary condition for determinism?

Qs-4) What is CFL and DCFL?

Qs-5) For regular language, then there is some DPDA explain?

6.13 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

6.14 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-VII SIMPLIFICATION OF CFG

7.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Simplification of CFG. It explains Chomsky Normal Form (CNF) and Greibach Normal Form (GNF) through various theorems, lemmas and step-wise elaborated solved examples.

7.2 CHOMSKY NORMAL FORM (CNF)

A CFG $G = (N, \Sigma, P, S)$ is in Chomsky Normal Form (CNF) if all production are of the form

A $\rightarrow$ BC or A $\rightarrow$ a

Where A, B, C $\in$ N and a $\in \Sigma$.

Note that since $\in$-production is not allowed, a CFG in CNF cannot generate the empty string $\in$.

Theorem : For any CFG $G = (N, \Sigma, P, S)$ there is a CFG $G' = (N', \Sigma, P', S')$ in Chomsky normal form such that L(G') = L(G)-{$\in$}.

Proof : Without loss of generality we can assume that G doesnot contain any $\in$-production, unit production and useless symbols. (Even if it contains, we can use the procedures already described to remove all those).

We use the following procedure to construct G' from *G*.

- N' and P' are suspects of N and P respectively.
- For each terminal $a \in \Sigma$, introduce a new non terminal Aa in N' and production $Aa \rightarrow a$ in P'. and replace all occurrences of a on the right-hand sides of old productions (except productions of the form $B \rightarrow a$, i.e. where rhs is only one terminal symbol, a with Aa.
- After this step, all productions in P' will be of the form $A \rightarrow a$ or $A \rightarrow B_1 B_2 \cdots B_K,\ K \geq 2$, where $A, B_1, B_2, \cdots B_K \in N'$. (K is greater than 2 because unit productions are already eliminated)
- Clearly, the language generated by this new grammar is not changed; it just takes one more steps than before to generate a terminal symbol.
- For all those productions $A \rightarrow B_1 B_2 \cdots B_K$ with $K \geq 3$, introduce a new non-terminal C in N' and replace this productions with two new productions in P'. $A \rightarrow B_1 a$ and $A \rightarrow B_2 B_3 \cdots B_K$.
- Once again, it is easy to see that this transformed grammar generates the same langauge; it just takes one step more than before to generate a terminal string.
- we repeat the above step until the right-hand sides of every production in P' are of length at most 2.

- The resulting grammar $G' = (N', \Sigma, P', S')$ is now in CNF and clearly $L(G') = L(G) - \{\in\}$.

Example : Consider the CFG : $S \to aSb \mid \in$ generating the language $L = \{a_n b_n \mid n \geq 0\}$. we will construct a CNF to generate the language $L - \{\in\}$ i.e. $\{a_n b_n \mid n \geq 1\}$.

Solutions : We first eliminate $\in$-productions (generating the language $\{a_n b_n \mid n \geq 1\}$) using the procedure already described to get $S \to aSb \mid ab$.

***Step* 1 :** Introduce nonterminals A, B and replace these productions with $S \to ASB \mid AB,\ A \to a,\ B \to b$

***Step* 2 :** Introduce nonterminal C and replace the only production $S \to ASB$ (which is not allowable form in *CNF*) with $S \to AC$ and $C \to SB$

The final grammar in CNF is now

$S \to AC \mid AB$
$C \to SB$
$A \to a$
$B \to b$

CHECK YOUR PROGRESS

True/False type questions

1) A CFG in CNF cannot generate the empty string $\in$. _____

2) A CFG $G = (N, \Sigma, P, S)$ is in **Chomsky Normal Form** (CNF) if all production are of the form

A $\to$ BC or A $\to$ a_____

3) The full form of CFG is Context full grammer._____

4) The full form of GNF is GREIBACH NORMAL FORM._____

5) For every CFG $G = (N, \Sigma, P, S)$ with $\in \notin L(G)$, there is no equivalent CFG $G' = (N', \Sigma, P', S')$ in CNF._____

Answers-

1) True

2) True

3) False

4) False

5) True

7.3 GREIBACH NORMAL FORM (GNF)

A CFG $G=(N, \Sigma, P, S)$ is in Greibach normal form if all productions in P are of the form

$A \rightarrow aB_1B_2 \cdots B_K$

for some $k \geq 0$, where $A, B_1, B_2, \cdots B_K \in N$ and $a \in \Sigma$.

We will now show that every CFG can be transformed to an equivalent CFG in GNF. We first produce two lemmas which help proving this fact.

Given any CFG G containing some left-recursive productions , we can construct an equivalent CFG removing those left-recursive productions by right-recursive productions. The following lemma proves this fact.

Lemma 1 : Let $G=(N, \Sigma, P, S)$ be CFG. Let $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_k$ be the set of all left-recursive A-productions and $A \rightarrow A\beta_1 \mid A\beta_2 \mid \cdots \mid A\beta_k$ be the remaining A-producitons in G. There is a CFG $G'=(N', \Sigma, P', S)$, where $N'=N \cup \{B\}$, $B \notin N$ and P' contains all productions in P except the left-recursive A-productions and also contains the following additional productions

$A \rightarrow \beta_1 B \mid \beta_2 B \mid \cdots \mid \beta_k B$

$B \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$

$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \cdots \mid \alpha_k B$

such that $L(G)=L(G')$

Proof : We first show that L(G) $\subseteq$ L(G ').

$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_k$ are the only productions whcih are in G but not in G'. Consider a leftmost derivation in G that uses a sequence of productions from this set. In such a case, the leftm$A \to \beta_j$nterminal A must $1 \le j \le n$y be disposed off by using a production of the type , later on, for some . That is, we have a derivation as shown below

$$S \overset{*}{\underset{G}{\Rightarrow}} \gamma A\delta \overset{1}{\underset{G}{\Rightarrow}} \gamma A\alpha_{i1}\delta \overset{1}{\underset{G}{\Rightarrow}} \gamma A\alpha_{i2}\alpha_{i1}\delta \overset{*}{\underset{G}{\Rightarrow}} \gamma A\alpha_{ip}\alpha_{ip-1}\cdots\alpha_{i1}\delta \overset{1}{\underset{G}{\Rightarrow}} \gamma\beta_j\alpha_{ip}\alpha_{ip-1}\cdots\alpha_{i1}\delta$$

The same derivation can be achieved in G' as follows:

$$S \overset{*}{\underset{G'}{\Rightarrow}} \gamma A\delta \overset{1}{\underset{G'}{\Rightarrow}} \gamma\beta_j B\delta \overset{1}{\underset{G'}{\Rightarrow}} \gamma\beta_j\alpha_{ip}B\delta \overset{1}{\underset{G'}{\Rightarrow}} \gamma\beta_j\alpha_{ip}\alpha_{ip-1}B\delta$$
$$\overset{*}{\underset{G'}{\Rightarrow}} \gamma\beta_j\alpha_{ip}\alpha_{ip-1}\cdots\alpha_{i2}B\delta \overset{1}{\underset{G'}{\Rightarrow}} \gamma\beta_j\alpha_{ip}\alpha_{ip-1}\cdots\alpha_{i2}\alpha_{i1}\delta$$

Hence any derivation in G is also a derivation in G' and so L(G)$\subseteq$L(G').

To show that L(G')$\subseteq$L(G), we need to follow just the reverse process of the above.

This shows that L(G)=L(G').

Lemma 2 : Let $G = (N, \Sigma, P, S)$ be a *CFG*. Let $A \to \alpha_1 B\alpha_2 \in P$ and $B \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_k$ be the set of all B-productions in P. There is a CFG $G' = (N', \Sigma, P', S')$ where

$$P' = P \cup \{A \to \alpha_1\beta_1\alpha_2 \mid \alpha_1\beta_2\alpha_2 \mid \cdots \mid \alpha_1\beta_k\alpha_2\} - \{A \to \alpha_1 B\alpha_2\}$$

such that $L(G') = L(G)$

Proof : We first show that $L(G) \subseteq L(G')$. It is clear that $A \to \alpha_1 B\alpha_2$ is the only production in G not in G'. If a derivation in G uses this production, then the nonterminal B must eventually be disposed off, later on, by using a production of the form $B \to \beta_j$, $1 \le j \le k$. That is, we have the derivation

$$S \overset{*}{\underset{G}{\Rightarrow}} \gamma A\delta \overset{1}{\underset{G}{\Rightarrow}} \gamma\alpha_1 B\alpha_2\delta \overset{*}{\underset{G}{\Rightarrow}} \eta B\theta \overset{1}{\underset{G}{\Rightarrow}} \eta B_j\theta$$

We can simulate this derivation in G' as follows

$$S \overset{*}{\underset{G'}{\Rightarrow}} \gamma A\delta \overset{1}{\underset{G'}{\Rightarrow}} \gamma\alpha_1\beta_j\alpha_2\delta \overset{*}{\Rightarrow} \eta\beta_j\theta$$

which takes one step less than the previous one.

Hence any derivation in G is also, a derivation in G' and so $L(G) \subseteq L(G')$.

Conversely, if $A \to \alpha_1 \beta_j \alpha_2$ is used (which is not in G) in a derivation in G', then the derivation will be of the form

$$S \overset{*}{\underset{G'}{\Rightarrow}} \gamma A \delta \overset{1}{\underset{G'}{\Rightarrow}} \gamma \alpha_1 \beta_j \alpha_2 \delta \overset{*}{\Rightarrow} \eta \beta_j \theta$$

The production $A \to \alpha_1 \beta_j \alpha_2$, $1 \le j \le k$ are the only producitons which are in G' but not in G. We can now simulate the above derivation in G as follows.

$$S \overset{*}{\underset{G}{\Rightarrow}} \gamma A \delta \overset{1}{\underset{G}{\Rightarrow}} \gamma \alpha_1 \beta_j \alpha_2 \delta \overset{1}{\underset{G}{\Rightarrow}} \gamma \alpha_1 \beta_j \alpha_2 \delta \overset{*}{\underset{G}{\Rightarrow}} \eta \beta_j \theta$$

Hence any derivation in G' is a derivation in G. and so $L(G) = L(G')$.
Hence the proof.

Theorem : For every CFG $G = (N, \Sigma, P, S)$ with $\in \notin L(G)$, there is an equivalent CFG $G' = (N', \Sigma, P', S')$ in CNF.

Proof : Without loss of generality, assume that G is in Chomsky normal form. Let the number of nonterminal in N be M. The following steps construct the equivalent CFG G' from G.

- The first step is to rename the nonterminals in N so that each one has a subscript, starting with 1upto m. So, the set of the nonterminals is now $N' = \{A_1, A_2, \cdots, A_m\}$. This step, certainly doesnot change the resulting language.
- The second step is to process the productions in P so that they satisfy the "Increasing Nonterminals Property" (INP) defined as follows:
- INP is said to be satisfied to be satisfied, if all productions are in form $A_i \to a\alpha$ or $A_i \to A_j \alpha$, where $j > i$ and $\alpha \in N^*$.
- To enforce this property (INP) we start with A_1-productions. Since G is in CNF, all A_1-productions are of the form $A_1 \to a$ or $A_1 \to A_i A_j, \quad 1 \le i, \ j \le m$
- The first one satisfies the property. The second one also satisfies it, unless i =1.

  When i =1, the production is of the form $A_1 \to A_i A_j$ which is a left-recursive one and we can apply lemma 1 to eliminate left-recursion by introducing a new variable , say A_{-1}. So, we have the productions before application of Lemma 1

  $A_1 \to A_1\alpha_1 \mid A_1\alpha_2 \mid \cdots \mid A_1\alpha_k$
- $A_1 \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
- Since G is in CNF, each α_i is some A_j and each β_i must begin with A_j, $j > 1$ or $a \in \Sigma$
- We apply lemma 1 introducing the new nonterminal A_{-1} and the production obtained from above after application of lemma 1 are
- $A_1 \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \mid \beta_1 A_{-1} \mid \beta_2 A_{-1} \mid \cdots \mid \beta_n A_{-1}$
- and $A_1 \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k \mid \alpha_1 A_{-1} \mid \alpha_2 A_{-2} \mid \cdots \mid \alpha_k A_{-1}$

All the above A_1-productions are of the form $A_1 \to a\alpha$ or $A_1 \to A_j \alpha$, where $\alpha \in N'^*, \ a \in \Sigma$ and $j > 1$

And the rhs of all A_{-1}- productions start with some A_j where $j \geq 1$.

So, the resulting grammar, with the new set of productions, say P', satisfies the INP.

- Consider the processing of A_1-productions to be the basis case
- Assume that we have processed $A_k \rightarrow A_i A_j$- through A_{k-1}-productions , this way , introducing new nonterminals A_{-1} through A_{k-1}. That, all these $A_{-(k-1)}, \cdots, A_{-1}, A_1, \cdots, A_{k-1}$ productions satisfy INP is the inductive hypothesis.
- We now process A_k-productions as the inductive step. Since G is in CNF , all A_k-productions are of the form $A_k \rightarrow a$ or $A_k \rightarrow A_i A_j$ the first one satisfies the INP. The second one doesnot satisfy the property whenever $i \leq k$. Consider $A_k \rightarrow A_i A_j$ with $i < k$. By induction hypothesis , all A_i-production satisfy INP, thus of form $A_i \rightarrow a\alpha$ or $A_i \rightarrow A_p \alpha$ with $a \in \Sigma$ and $p > i$. Now, applying lemma 2 to replace A_i with the right hand side of Ai-productions to produce $A_k \rightarrow a\alpha A_j$ (this satisfies INP)
- and $A_k \rightarrow A_p \alpha A_j$

  If $p < k$, we again replace A_p by the rhs of A_p-production.
- After atmost k-iterations, all A_k-productions will be of the form
  $A_k \rightarrow a\alpha$
- $A_k \rightarrow A_k \alpha$

  $A_k \rightarrow A_j \alpha$ with $j > k$, $\alpha \in N'^*$
- The first and the third productions satisfy the INP. The second one doesnot satisfy it, because it is self recursive .
- Apply lemma 1 again, introducing a new nonterminal, A_{-k}, to enforce the INP, exactly in a similar way as we did earlier . So, the resulting productions will satisfy the INP.
- Hence, if we continue to process upto Am-productions, the resulting grammar will satisfy the INP (as proved by induction, above).
- There may be atmost $2m$ nonterminals in N' now,

  namely $\{A_{-m}, A_{-(m-1)}, \cdots, A_{-1}, A_1, \cdots, A_m\}$.
- The third step is to process all these productions, starting with A_m-produtcions down to A_{-m}-production, to get the equivalent CNF G'. m is the highest subscript of any nonterminal A_j. So, by INP, all Am-productions are of the form $A_m \rightarrow a\alpha$, $a \in \Sigma$ and $\alpha \in N'^*$. Thus this production, is in GNF already.
- Consider the A_{m-1}-productions.
- By INP all A_{m-1}-productions must be of the form
- $A_{m-1} \rightarrow a\beta$ (already in GNF)
- and $A_{m-1} \rightarrow A_m \beta$ where $\beta \in N'^*$

- we now apply lemma 2 to replace Am(in the rhs) with the right-hand side of Am-production given above. This gives us A_{m-1}-productions of the form $A_{m-1} \to a\beta$

$A_{m-1} \to a\alpha\beta$ where $\beta \in N^*$

Both type of production now satisfy GNF property.

we inductively process down to the lowest subscripted nonterminal applying lemma 2 wherever necessary.

All productions now satisfy the GNF property i.e. of the form

$A_i \to a\alpha, \quad \alpha \in N^* \quad \& \quad a \in \Sigma$

since we applied either lemma 1 or lemma 2 for any intermediate transformation the resulting grammar, say G', must be equivalent to G i.e. $L(G) = L(G')$.

Example : A $\to$ BB B $\to$ AC | a C $\to$ AB | BA | a . We will construct an equivalent CFG in GNF.

Step 1: Renaming the nonterminal, we get

$A_1 \to A_2A_2$
$A_2 \to A_1A_3 \mid a$
$A_3 \to A_1A_2 \mid A_2A_1 \mid a$

Step 2 : A_1-productions already satisfy INP.

Process A_2- and A_3-productions to enforce the INP.

First consider A_2-productions:

Apply lemma 2 to $A_2 \to A_1A_3$ obtaining $A_2 \to A_2A_2A_3 \mid a$. Now apply lemma 1 to eliminate left-recursion

We get

$A_2 \to a \mid A_{-2}$
$A_{-2} \to A_2A_3 \mid A_2A_3A_{-2}$

which satisfy the INP property.

The resulting grammar is

$A_1 \to A2A2$

$A_2 \to a \mid aA_{-2}$

$A_{-2} \to A_2A_3 \mid A_2A_3A_{-2}$

$A_3 \to A_1A_2 \mid A_2A_1 \mid a$

Next consider A_3-productions. Applying lemma 2 to $A_3 \to A_1A_2$ we get

$A_3 \to A_2A_2A_2 \mid A_2A_1 \mid a$

Applying lemma 2 again on the first two A_3-productions above we get

$A_3 \to aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$

Now, all productions satisfy the INP.

The resulting grammar is:

$A_1 \to A_2A_2 \mid$

$A_2 \to a \mid aA_{-2}$

$A_{-2} \to A_2A_3 \mid A_2A_3A_{-2}$

$A_3 \to aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$

Step 3: All A_3-productions and A_2-productions are already in GNF. Apply lemma 2 to A_1-productions, to get $A_1 \to aA_2 \mid aA_{-2}A_2$.

Similarly, applying lemma 2 to A_{-2}-production we get

$A_{-2} \to aA_3 \mid aA_{-2}A_3 \mid aA_3A_{-2} \mid aA_{-2}A_3A_{-2}$

All the productions are in GNF now. So, the resulting equivalent grammar in GNF is

$A_1 \to aA_2 \mid aA_{-2}A_2$

$A_{-2} \to aA_3 \mid aA_{-2}A_3 \mid aA_3A_{-2} \mid aA_{-2}A_3A_{-2}$

$A_2 \to a \mid aA_{-2}$

$A_3 \to aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$

7.4 CONCLUSION

This module explains about the basic understanding of Simplification of CFG. It discusses Chomsky Normal Form (CNF) and Greibach Normal Form (GNF) with various theorems, lemmas and elaborated solved examples.

7.5 CHECK YOUR PROGRESS

Fill in the blanks:

1) The Full form of CFG is _____

2) The full form of GNF is_____

3) The Greibach normal form if all productions in P are of the form_____

4) The Full form of CNF is_____

5) A CFG_____ is in Chomsky Normal Form (CNF) if all production are of the form

7.6 ANSWER CHECK YOUR PROGRESS

1) Context free Grammer

2) Greibach normal form

3) $A \rightarrow aB_1B_2 \cdots B_K$

4) Chomsky Normal Form

5) $G = (N, \Sigma, P, S)$

7.7 MODEL QUESTION

Qs-1) What is CNF and what are steps for CNF to be in CFG?

Qs-2) What is GNF and what are steps for CNF to be in GNF?

Qs-3) Construct an equivalent CFG in GNF?

Qs-4) What is INP and wHen it is said to be satisfied?

Qs-5) For any CFG $G = (N, \Sigma, P, S)$ there is a CFG $G' = (N', \Sigma, P', S')$ in Chomsky normal form such that L(G') = L(G)$\{\in\}$explain?

7.8 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

7.9 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-VIII CONTEXT FREE LANGUAGES

8.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Context Free Languages (CFLs). It explains Closure Property of Context Free Languages (CFLs), Some Decision Algorithms for CFLs and Testing Finiteness of a CFL through various theorems, lemmas and step-wise elaborated solved examples.

8.2 PUMPING LEMMA FOR CONTEXT FREE LANGUAGES (CFLs)

There is a pumping lemma for CFLs similar to the one for regular language. It can be used in the same way to show that certain languages are not context-free.

Informally, the pumping lemma for CFLs states that for every CFL L, every sufficiently long string can be subdivided into five substrings such that the middle three substrings are not too long, the second and fourth are not both empty, and if these two substrings such that the middle and fourth are not both empty, and if these two substrings are pumped simultaneously zero or more times, then the resulting string will always belong to L.

Theorem (pumping lemma for CFLs)**:**

Let L be any CFL. Then there is a constant n > 0, depending only on L, such that every string $z \in L$ of length at least n can be as ***z = uvwxy*** such that

$$i) |vx| > 0$$
$$ii) |vwx| \leq n$$
$$iii) \forall i \geq 0, \quad uv^i wx^i y \in L$$

Proof : Let G be a CFG in Chomsky Normal Form generating $L - \{\in\}$. Let the number of nonterminals in G is K and let the constant of pumping lemma $n = 2^k$. We will now show that all strings in L with length n or greater can be decomposed to satisfy the conditions of the pumping lemma. Let $\Im \in L(G)$ be such a string i.e. $|\Im| \geq n$ or $|\Im| \geq 2^k$, and $S \overset{*}{\underset{G}{\Rightarrow}} \Im$. Since $|\Im| \geq 2^k$, then height (depth) of the parse tree for $\Im$ is at least $k + 1$. Hence, there is a path of length at least k + 1. in the parse tree for $\Im$. Let p be a path of maximal length from root S to a leaf of the parse tree. Then P must contain at least k + 2 nodes, all of which are labelled by non-terminals except the leaf node which is labelled by a terminal symbol. Hence, there is at least k + 1 non-terminals in that path and since then and only k non-terminals in G, some variable must occur more than once on the path. We select R to be a nonterminal that repeat among the lowest k + 1 non-terminal on this path. To find the lowermost occurrence of R we follow the path up from the leaf keeping track of the labels encountered. Of the first k + 2 nodes only the leaf has a terminal label. The remaining k + 1 node cannot have distinct nonterminal labels and hence we simply pick the first repetition of any non-terminal on that path and call it R.

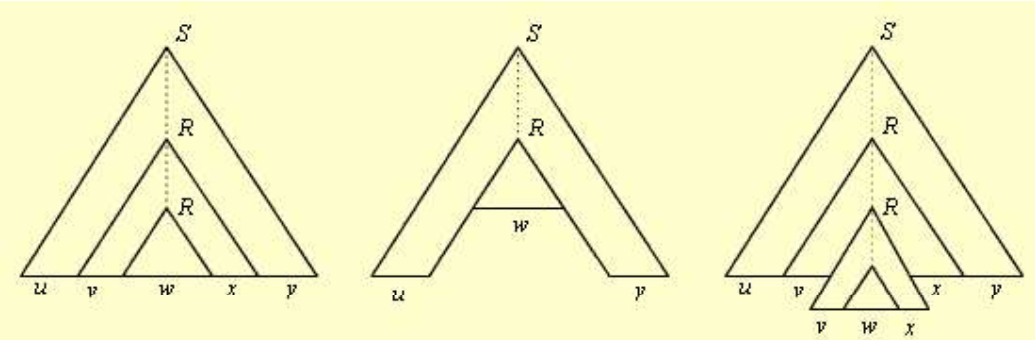


Figure 1

We divide $\mathfrak{Z}$ into uvwxy according to the figure given above. The derivation of $\mathfrak{Z}$ can be given as follows

$$S \overset{*}{\underset{G}{\Rightarrow}} uRy \overset{*}{\underset{G}{\Rightarrow}} uvRxy \overset{*}{\underset{G}{\Rightarrow}} uvwxy$$

Clearly, we have two subderivations

$$R \overset{*}{\underset{G}{\Rightarrow}} vRx \quad \text{and} \quad R \overset{*}{\underset{G}{\Rightarrow}} w$$

The first one corresponds to the subtree rooted at the upper occurence of R and the second one corresponds to the subtree rooted at the lower occurence of R. Both these subtree are generated by the same nonterminal R, so we may substitute one for the ohter and still obtain a valid parse tree. Replacing the larger by the smaller generates the string uwy and hence $uwy \in L$. In this case, the upper occurence of R generates w directly using $R \overset{*}{\underset{G}{\Rightarrow}} w$ instead of generating vwx using $R \overset{*}{\underset{G}{\Rightarrow}} vRx \overset{*}{\underset{G}{\Rightarrow}} vwx$. This is shown in the middle figure.Similarly, replacing ths smaller subtree by the larger one repeatedly, as shown in the last figure, gives of parse trees for the strings $uv^i wx^i y$ at each i >1. That establishes that $uv^i wx^i y \in L(G), \quad \forall i \geq 0$.

We now show that condition (i) and (ii) in the pumping lemma are satisfied by this decomposition. The subderivation $R \overset{*}{\underset{G}{\Rightarrow}} vRx$ must begin with a rule of the form R $\rightarrow$ AB. The second occurence of R is derived from either A or B. If it is derived from A, then the derivation can be written

$$R \overset{*}{\underset{G}{\Rightarrow}} AB \overset{*}{\underset{G}{\Rightarrow}} vRsC \overset{*}{\underset{G}{\Rightarrow}} vRst = vRx$$

The string t is non-empty since it is obtained by a derivation from a non terminal in a CNF form grammar. Hence $x = st$ must be non-empty. If the second occurrence of R is derived from B, a similar argument shows that v is non-empty. Hence, |vx| >0, giving condition (i). The stable rooted at the upper occurrence of R generates vwx. But this R is the first repetition of a variable in the longest path p in the parse tree starting from a leaf. i.e. both occurrence of R fall within the bottom k +1 variables on the path. Hence the subtree rooted at the upper occurrence of R has depth (or height) at most k +1. A tree of the height can generate a string of length at most $n = 2^k$ or less. Therefore $|vwx| \le 2^k$, giving condition (ii).
This completes the proof.

Example: The language $L = \{a^i b^i c^i \mid i \ge 0\}$ is not context-free.

Proof: We apply the pumping lemma to prove it. Assume, for contradiction , that *L* is a CFL.

Let *n* be the constant of the lemma. consider the string $S = a^n b^n c^n$. Evidently, $s \in L$ and $|s| = 3n > n$.

Therefore, according to the Lemma, there exists substrings *u*, *v*, *w*, *x*, *y* such that *s* = *uvwxy* and the following hold.

1. $|vx| > 0$
2. $|vwx| \le n$
3. $\forall i \ge 0, \quad uv^i wx^i y \in L$

vxy cannot start with some *a*, span all *n* *b*'s, and finish with some *c* - condition (ii) above prohibits this.

We now consider all other possibilities of occurence of *vxy* in *S*.

Case 1 : *vxy* occur completely within the leading an symbols. Then pumping up once yields the string $S' = uv^2 wx^2 y = a^m b^n c^n$, where $m > n$ (since number of a's gets increased) . Thus, $S' \notin L$ contradicting the lemma.

Case 2 : If *vxy* occur completely within the middle b_n or trailing c_n symbols, then we can apply exactly similar arguments as in case 1 to arrive at contradictions.

Case 3 : If *vxy* occur partly in the an and partly in the b_n, then pumping up once will yield a string that either contains more *a*'s than *c*'s and more *b*'s than *c*'s or contains some *a*'s after *bb*'s. In both cases, the resulting string is not in *L* and is a contradiction.

Case 4: If *vxy* occur partly in the b_n and partly in the c_n then we can apply exactly similar arguments as in case 3 to arrive at a contradiction.

These are the only possible cases to divide the string into substrings as per the lemma and in every case there is contradiction. Hence, *L* is not a CFL.

Example: $L = \{a^{k^2} \mid k \geq 0\}$ is not context-free.

Proof: Assume contradiction that L is a CFL. There is a constant $n>0$ such that any string in L of length at least n can be pumped according to the pumping lemma.

Consider the string $S = a^{n^2}$ in L. $|S| = n^2 > n$. So, we can write

$S = uvwxy$ such that

1. $|vx|>0$
2. $|vwx| \leq n$ and
3. $uv^i wx^i y \in L$ for $i \geq 0$

By (i) and (ii), $|vx| \leq n$. Thus if we let $i = 1$ in (iii) (i.e. if we pump once), we get a string $s' = a^j$ where $n^2 < j \leq n^2 + n$. Now if we arrange the elements of L in ascending order of length, thus the next element after $S = a^{n^2}$ must be of length $(n+1)^2$ i.e. of length $(n^2 + 2n + 1)$.

Since $n^2 < j < (n^2 + 2n + 1)$, we conclude that S' (which is of length j) is not in L, which contradicts the pumping lemma.

Hence L is not a CFL.

8.3 CLOSURE PROPERTY OF CONTEXT FREE LANGUAGES (CFLs)

We consider some important closure properties of CFLs.

Theorem : If L_1 and L_2 are CFLs then so is $L_1 \cup L_2$

Proof : Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be CFGs generating. Without loss of generality, we can assume that $N_1 \cap N_2 = \phi$. Let S_3 is a nonterminal not in N_1 or N_2. We construct the grammar $G_3 = (N_3, \Sigma_3, P_3, S_3)$ from G_1 and G_2, where

$N_3 = N_1 \cup N_2 \cup \{S_3\}$,

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 \mid S_2\}$

We now show that $L(G_3) = L(G_1) \cup L(G_2) = L_1 \cup L_2$

Thus proving the theorem.

Let $w \in L_1$. Then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$. All productions applied in their derivation are also in G_3. Hence $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$ i.e. $w \in L(G_3)$

Similarly, if $w \in L_2$, then $w \in L(G_3)$

Thus $L_1 \cup L_2 \subseteq L(G_3)$.

Conversely, let $w \in L(G_3)$. Then $S_3 \overset{*}{\underset{G_3}{\Rightarrow}} w$ and the first step in this derivation must be either $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1$ or $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_2$. Considering the former case, we have $S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$

Since N_1 and N_2 are disjoint, the derivation $S_1 \overset{*}{\underset{G_3}{\Rightarrow}} w$ must use the productions of P_1 only (which are also in P_3) Since $S_1 \in N_1$ is the start symbol of G_1. Hence, $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w$ giving $w \in L(G_1)$.

Using similar reasoning, in the latter case, we get $w \in L(G_2)$. Thus $L(G_3) \subseteq L_1 \cup L_2$.

So, $L(G_3) = L_1 \cup L_2$, as claimed.

Theorem: If L_1 and L_2 are CFLs, then so is $L_1 L_2$.

Proof: Let $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ be the CFGs generating L_1 and L_2 respectively. Again, we assume that N_1 and N_2 are disjoint, and S_3 is a nonterminal not in N_1 or N_2. we construct the CFG $G_3 = (N_3, \Sigma_3, P_3, S_3)$ from G_1 and G_2, where

$N_3 = N_1 \cup N_2 \cup \{S_3\}$

$\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 S_2\}$

We claim that $L(G_3) = L(G_1)L(G_2) = L_1L_2$

o prove it, we first assume that $x \in L_1$ and $y \in L_2$. Then $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} y$. We can derive the string *xy* in G_3 as shown below.

$$S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1S_2 \overset{*}{\underset{G_3}{\Rightarrow}} xS_2 \overset{*}{\underset{G_3}{\Rightarrow}} xy$$

since $P_1 \subseteq P$ and $P_2 \subseteq P$. Hence $L_1L_2 \subseteq L(G_3)$.

For the converse, let $w \in L(G_3)$. Then the derivation of *w* in G_3 will be of the form

$S_3 \overset{1}{\underset{G_3}{\Rightarrow}} S_1S_2 \overset{*}{\underset{G_3}{\Rightarrow}} w$ i.e. the first step in the derivation must see the rule $S_3 \to S_1S_2$. Again, since N_1 and N_2 are disjoint and $S_1 \in N_1$ and $S_2 \in N_2$, some string *x* will be generated from S_1 using productions in P_1 (which are also in P_3) and such that $xy = w$.

Thus $S_3 \Rightarrow S_1S_2 \Rightarrow xS_2 \Rightarrow xy = w$

Hence $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$ and $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} y$.

This means that *w* can be divided into two parts *x*, *y* such that $x \in L_1$ and $y \in L_2$. Thus $w \in L_1L_2$ .This completes the proof.

Theorem : If *L* is a CFL, then so is L^*.

Proof : Let $G = (N, \Sigma, P, S)$ be the CFG generating *L*. Let us construct the CFG $G' = (N, \Sigma, P', S)$ from *G* where $P' = P \cup \{S \to SS \mid \in\}$.

We now prove that $L(G') = (L(G))^* = L^*$, which prove the theorem.

G' can generate $\in$ in one step by using the production $S \to \in$ since $P \subseteq P'$, G' can generate any string in *L*. Let $w \in L^n$ for any *n* >1 we can write $w = w_1w_2 \cdots w_n$ where $w_i \in L$ for $1 \le i \le n$. *w* can be generated by G' using following steps.

$$S \overset{n-1}{\underset{G}{\Rightarrow}} SS \cdots S \overset{*}{\underset{G}{\Rightarrow}} w_1Ss \cdots s \overset{*}{\underset{G}{\Rightarrow}} w_1w_2Ss \cdots s \overset{*}{\underset{G}{\Rightarrow}} w_1w_2 \cdots w_n = w$$

First (n-1)-steps uses the production $S \rightarrow SS$ producing the sentential form of n numbers of S's. The nonterminal S in the i-th position then generates w_i using production in P (which are also in P')

It is also easy to see that G can generate the empty string, any string in L and any string $w \in L^n$ for n >1 and none other.

Hence $L(G') = (L(G))^* = L^*$

Theorem : CFLs are not closed under intersection

Proof : We prove it by giving a counter example. Consider the language $L_1 = \{a^i b^i c^j \mid i, j \geq 0\}$ .The following CFG generates L_1 and hence a CFL

$S \rightarrow XC$
$X \rightarrow aXb \mid \in$
$C \rightarrow cC \mid \in$

The nonterminal X generates strings of the form $a^n b^n, n \geq 0$ and C generates strings of the form c^m, $m \geq 0$. These are the only types of strings generated by X and C. Hence, S generates L_1.

Using similar reasoning, it can be shown that the following grammar $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$ and hence it is also a CFL.

$S \rightarrow AX$
$A \rightarrow aA \mid \in$
$X \rightarrow bXc \mid \in$

But, $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ and is already shown to be not context-free.

Hence proof.

Theorem : A CFL's are not closed under complementations

Proof : Assume, for contradiction, that CFL's are closed under complementation. SInce, CFL's are also closed under union, the language $\overline{\overline{L_1} \cup \overline{L_2}}$, where L_1 and L_2 are CFL's must be CFL. But by DeMorgan's law

$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$

This contradicts the already proved fact that CFL's are not closed under intersection.

But it can be shown that the CFL's are closed under intersection with a regular set.

Theorem : If *L* is a CFL and *R* is a regular language, then $L \cap R$ is a CFL.

Proof : Let $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, z_0, F_P)$ be a PDA for *L* and let $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ be a DFA for *R*.

We construct a PDA *M* from *P* and *D* as follows

$$M = (Q_P \times Q_D, \Sigma, \Gamma, \delta_M, (q_P, q_D), z_0, F_P \times F_D)$$

where δ_M is defined as

$\delta_M((p,q),a,X)$ contains $((r,s),\alpha)$ iff

$\delta_D(q,a) = s$ and $\delta_P(p,a,X)$ contains (r,α)

The idea is that *M* simulates the moves of *P* and *D* parallely on input *w*, and accepts *w* iff both *P* and *D* accepts. That means, we want to show that

$$L(M) = L(P) \cap L(D) = L \cap R$$

We apply induction on *n*, the number of moves, to show that

$((q_P, q_D), w, z_0) \overset{n}{\underset{M}{\mapsto}} ((p,q), \in, \gamma)$ iff

$(q_P, w, z_0) \overset{n}{\underset{P}{\mapsto}} (p, \in, \gamma)$ and $\hat{\delta}(q_D, w) = q$

Basic Case is *n*=0. Hence $p = q_P$, $q = q_D$, $\gamma = z_0$ and $w = \in$. For this case it is trivially true

Inductive hypothesis : Assume that the statement is true for *n* -1.

Inductive Step : Let *w* = *xa* and

Let $((q_P, q_D), x_a, z_0) \overset{n-1}{\underset{M}{\mapsto}} ((p', q'), a, \alpha) \overset{1}{\underset{M}{\mapsto}} ((p,q), \in, \gamma)$

By inductive hypothesis, $(q_p, x, z_0) \overset{n-1}{\underset{P}{\mapsto}} (p', \in, \alpha)$ and $\hat{\delta}_D(q_D, x) = q'$

From the definition of δ_M and considering the *n*-th move of the PDA *M* above, we have

$\delta_p(p', a, \alpha) = (p, \in, \gamma)$ and $\delta_D(q', a) = q$

Hence $(q_p, xa, z_0) \overset{n-1}{\underset{p}{\mapsto}} (p', a, \alpha) \overset{1}{\underset{p}{\mapsto}} (p, \in, \gamma)$ and $\hat{\delta}_D(q_D, w) = q$

Inductive hypothesis: Assume that the statement is true for *n* -1.

Inductive Step: Let *w* = *xa* and

Let $((q_p, q_D), x_a, z_0) \overset{n-1}{\underset{M}{\mapsto}} ((p', q'), a, \alpha) \overset{1}{\underset{M}{\mapsto}} ((p, q), \in, \gamma)$

By inductive hypothesis, $(q_p, x, z_0) \overset{n-1}{\underset{P}{\mapsto}} (p', \in, \alpha)$ and $\hat{\delta}_D(q_D, x) = q'$

From the definition of δ_M and considering the *n*-th move of the PDA *M* above, we have

$\delta_p(p', a, \alpha) = (p, \in, \gamma)$ and $\delta_D(q', a) = q$

Hence $(q_p, xa, z_0) \overset{n-1}{\underset{p}{\mapsto}} (p', a, \alpha) \overset{1}{\underset{p}{\mapsto}} (p, \in, \gamma)$ and $\hat{\delta}_D(q_D, w) = q$

If $p \in F_p$ and $q \in F_D$, then $p, q \in F_p \times F_D$ and we got that if *M* accepts *w*, then both *P* and *D* accepts it.

We can show that converse, in a similar way. Hence $L \cap R$ is a CFL (since it is accepted by a PDA *M*)

This property is useful in showing that certain languages are not context-free.

Example: Consider the language

$$L = \left\{ w \in (a,b,c)^* \mid w \text{ contains equal number of } a's, b's \text{ and } c's \right\}$$

Intersecting *L* with the regular set $R = a^*b^*c^*$, we get

$$L \cap R = L \cap a^* b^* c^*$$
$$= \left\{ a^n b^n c^n \mid n \geq 0 \right\}$$

Which is already known to be not context-free. Hence L is not context-free.

Theorem : CFL's are closed under reversal. That is if L is a CFL, then so is L^R

Proof : Let the CFG $G = (N, \Sigma, P, S)$ generates L. We construct a CFG $G' = (N, \Sigma, P', S)$ where

$P = \left\{ A \to \alpha \mid A \to \alpha^R \in P \right\}$. We now show that $L(G') = L^R$, thus proving the theorem.

We need to prove that

$A \underset{G}{\overset{n}{\Rightarrow}} \alpha$ iff $A \underset{G'}{\overset{n}{\Rightarrow}} \alpha^R$.

The proof is by induction on n, the number of steps taken by the derivation. We assume, for simplicity (and of course without loss of generality), that G and hence G' are in CNF.

The basis is n=1 in which case it is trivial. Because α must be either $a \in \Sigma$ or BC with $B, C \in N$.

Hence $A \underset{G}{\overset{1}{\Rightarrow}} a$ iff $A \underset{G'}{\overset{1}{\Rightarrow}} a$ and

Assume that it is true for (n-1)-steps. Let $A \underset{G}{\overset{n}{\Rightarrow}} \alpha$. Then the first step must apply a rule of the form $A \to BC$ and it gives

$A \underset{G}{\overset{1}{\Rightarrow}} BC \underset{G}{\overset{n-1}{\Rightarrow}} \beta\gamma = \alpha$ where $B \underset{G}{\overset{*}{\Rightarrow}} \beta^R$ and $C \underset{G}{\overset{*}{\Rightarrow}} \gamma^R$

By constructing of G', $A \to CB \in P'$

Hence

$A \underset{G'}{\overset{1}{\Rightarrow}} CB \underset{G'}{\overset{n-1}{\Rightarrow}} \gamma^R \beta^R = \alpha^R$

The converse case is exactly similar.

Substitution :

$\forall a \in \Sigma$, let L_a be a language (over Σ). This $s(a) = L_a$ defines a function *S*, called substitution, on $a \in \Sigma$ which is denoted as - for all

This definition of substitution can be extended further to apply strings and langauge as well.

If $w = a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$, is a string in Σ^*, then

$s(w) = s(a_1 a_2 \cdots a_n) = s(a_1)s(a_2)\cdots s(a_n)$.

Similarly, for any language *L*,

$s(L) = \{s(w) \mid w \in L\}$

The following theorem shows that CFLs are closed under substitution.

Thereom: Let $L \subseteq \Sigma^*$ is a CFL, and *s* is a substitution on Σ such that $s(a) = L_a$ is a CFL for all $a \in \Sigma$, thus *s*(*L*) is a CFL

Proof: Let *L* = *L*(*G*) for a CFG $G = (N, \Sigma, P, S)$ and for every $a \in \Sigma$, $L_a = L(G_a)$ for some $G_a = (N_a, \Sigma_a, P_a, S_a)$. Without loss of generality, assume that the sets of non terminals *N* and N_a's are disjoint.

Now, we construct a grammar G', generating *s*(*L*), from *G* and G_a's as follows **:**

- $G' = (N', \Sigma', P', S)$
- $N' = N \cup \bigcup_{a_i \in \Sigma} N_{a_i}$
- $\Sigma' = \bigcup_{a_i \in \Sigma} \Sigma_{a_i}$
- P' consists of
  - $\bigcup_{a_i \in \Sigma} P_{a_i}$ and
  - The production of *P* but with each terminal *a* in the right hand side of a production replaced by S_a everywhere.

We now want to prove that this construction works i.e. $w \in L(G')$ iff $w \in s(L)$.

If Part : Let $w \in s(L)$ then according to the definition there is some string $x = a_1 a_2 \cdots a_n \in L$ and $x_i \in S(a_i)$ for $i = 1, 2, \cdots, n$ such that $w = x_1 x_2 \cdots x_n \left(= s(a_1)s(a_2)\cdots s(a_n)\right)$

We will show that $S \overset{*}{\underset{G}{\Rightarrow}} w$.

From the construction of G', we find that, there is a derivation $S \overset{*}{\underset{G'}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$ corresponding to the string $x = a_1 a_2 \cdots a_n$ (since G' contains all productions of G but every ai replaced with S_{a_i} in the RHS of any production).

Every S_{a_i} is the start symbol of G_{a_i} and all productions of G_{a_i} are also included in G'.

Hence

$$S \overset{*}{\underset{G'}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$$
$$\overset{*}{\underset{G'}{\Rightarrow}} x_1 S_{a_2} \cdots S_{a_n}$$
$$\overset{*}{\Rightarrow} x_1 x_2 \cdots x_n = w$$

Therefore, $w \in L(G')$.

(Only-if Part) Let $w \in L(G')$. Then there must be a derivative as follows :

$S \overset{*}{\underset{G'}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$ (using the production of *G* include in G' as modified by (step 2) of the construction of P'.)

Each S_{a_i} ($i = 1, 2, \cdots, n$) can only generate a string $x_i \in L_{a_i}$, since each N_{a_i}'s and *N* are disjoin. Therefore, we get

$$S \overset{*}{\underset{G'}{\Rightarrow}} S_{a_1} S_{a_2} \cdots S_{a_n}$$

$\overset{*}{\underset{G'}{\Rightarrow}} x_1 S_{a_2} \cdots S_{a_n}$ since $S_{a_1} \overset{*}{\underset{G_{a_1}}{\Rightarrow}} x_1$

$\overset{*}{\underset{G'}{\Rightarrow}} x_1 x_2 S_{a_3} \cdots S_{a_n}$ since $S_{a_2} \overset{*}{\underset{G_{a_2}}{\Rightarrow}} x_2$

$$\overset{*}{\underset{G}{\Rightarrow}} x_1 x_2 \cdots x_n$$
$$= w$$

The string $w = x_1 x_2 \cdots x_n$ is formed by substituting strings x_i for each a_i's and hence $w \in s(L)$.

Theorem : CFL's are closed under homomorphism

Proof : Let $L \subseteq \Sigma^*$ be a CFL, and h is a homomorphism on Σ i.e $h : \Sigma \to \Delta^*$ for some alphabets Δ. Consider the following substitution S: Replace each symbol $a \in \Sigma$ by the language consisting of the only string h(a), i.e. $s(a) = \{h(a)\}$ for all $a \in \Sigma$. Then, it is clear that, $h(L) = s(L)$. Hence, CFL's being closed under substitution must also be closed under homomorphism.

CHECK YOUR PROGRESS

True/False type questions

1)Pumping Lemma can be used in the same way to show that certain languages are context-free._____

2) A CFL's are not closed under complementation_____

3) There are algorithms to test emptiness of a CFL._____

4) CFLs are closed under intersection_____

5) CYK Algorithm to decide membership in CFL_____

Answers-

1) False

2) True

3) True

4) False

5) True

8.4 SOME DECISION ALGORITHMS FOR CFLs

In this section, we examine some questions about CFLs we can answer. A CFL may be represented using a CFG or PDA. But an algorithm that uses one representation can be made to work for the others, since we can construct one from the other.

8.4.1 TESTING EMPTINESS:

Theorem: There are algorithms to test emptiness of a CFL.

Proof : Given any CFL L, there is a CFG G to generate it. We can determine, using the construction described in the context of elimination of useless symbols, whether the start symbol is useless. If so, then $L(G) = \phi$; otherwise not.

8.4.2 TESTING MEMBERSHIP:

Given a CFL *L* and a string *x*, the membership, problem is to determine whether $x \in L$?

Given a PDA *P* for *L*, simulating the PDA on input string *x* does not quite work, because the PDA can grow its stack indefinitely on $\in$ input, and the process may never terminate, even if the PDA is deterministic.

So, we assume that a CFG $G = (N, \Sigma, P, S)$ is given such that $L = L(G)$.

Let us first present a simple but in-efficient algorithm.

Convert *G* to $G' = (N', \Sigma', P', S')$ in CNF generating $L(G) - \{\in\}$. If the input string $x = \in$, then we need to determine whether $S \overset{*}{\underset{G}{\Rightarrow}} \in$ and it can easily be done using the technique given in the context of elimination of $\in$ -production. If , $x \neq \in$ then $x \in L(G')$ iff $x \in L(G)$. Consider a derivation under a grammar in CNF. At every step, a production in CNF in used, and hence it adds exactly one terminal symbol to the sentential form. Hence, if the length of the input string *x* is n, then it takes exactly *n* steps to derive *x* (provided *x* is in $L(G')$).

Let the maximum number of productions for any non terminal in G' is *K*. So at every step in derivation, there are atmost *k* choices. We may try out all these choices, systematically., to derive the string *x* in G'. Since there are atmost $K^{|x|}$ i.e. K^n choices. This algorithms is of exponential time complexity. We now present an efficient (polynomial time) membership algorithm.

8.4.3 CYK ALGORITHM TO DECIDE MEMBERSHIP IN CFL

We now present a cubic-time algorithm due to cocke, Younger and Kasami. It uses the dynamic programming technique-solves smaller sub-problems first and then builds up solution by combining smaller sub-solutions. It determines for each substring *y* of the given string *x* the set of all nonterminals that generate *y*. This is done inductively on the length of *y*.

Let $G = (N, \Sigma, P, S)$ be the given CFG in CNF. Consider the given string *x* and let $|x| = n$.

Let x_{ij} be the substring of *x* that begins at position *i* (i.e. *i*-th symbol of *x*) and has length *j*.

Let N_{ij} be the set of all nonterminals *A* such that $A \overset{*}{\Rightarrow} x_{ij}$.

We write $x = x_{11}x_{21}x_{31}\cdots x_{(n-1)1}x_{n1}$. Where each $x_{i1}\,(1 \le i \le n)$ is a terminal symbol.

$A \overset{*}{\Rightarrow} x_{i1}$ iff $A \to x_{i1} \in P$. Thus we construct the sets N_{i1} for all $1 \le i \le n$.

Combining substrings of length 2, it is clear that, $A \in N_{i2}$ i.e. $A \overset{*}{\Rightarrow} x_{i2}$ iff there is a production $A \to BC$ in *G* and $B \overset{*}{\Rightarrow} x_{i1}$ and $C \overset{*}{\Rightarrow} x_{i+1,1}$.

That is $A \in N_{i2}$ iff $A \to BC \in P$ and $B \in N_{i1}$ and $C \in N_{i+1,1}$

Thus we can construct the sets N_{i2} from the already constructed sets N_{i1}, by inspecting the grammar.

In general considering substrings x_{ij} of length j, $A \in N_{ij}$ i.e. $A \overset{*}{\Rightarrow} x_{ij}$ iff there is a production $A \to BC$ in G such that $B \overset{*}{\Rightarrow} x_{ik}$ and $C \overset{*}{\Rightarrow} x_{i+k,j-k}$ for some $1 \le k \le j$.

That is $A \in N_{ij}$ iff $B \in N_{ik}$ and $C \in N_{i+k,j-k}$ for some $1 \le k \le j$ such that $A \to BC \in P$. The idea is to divide, x_{ij} into smaller substrings, using all possible ways (i.e. for different values of k), and construct N_{ij} from already constructed sets for smaller substrings (i.e. N_{ik} and $N_{i+k,j-k}$) by inspecting the grammar.

These sets for longer substrings of x are constructed inductively until the set N_{1n} for the string $x_{1n} = x$ is constructed.

It is clear from the construction that $S \overset{*}{\Rightarrow} x_{1n} = x$ iff $S \in N_{1n}$

Hence, we can determine whether $x \in L$ by inspecting N_{1n}.

The CYK algorithm is presented next.

CYK-Algorithm

Input: A CFG $G = (N, \Sigma, P, S)$ and a string $x \in \Sigma^*$

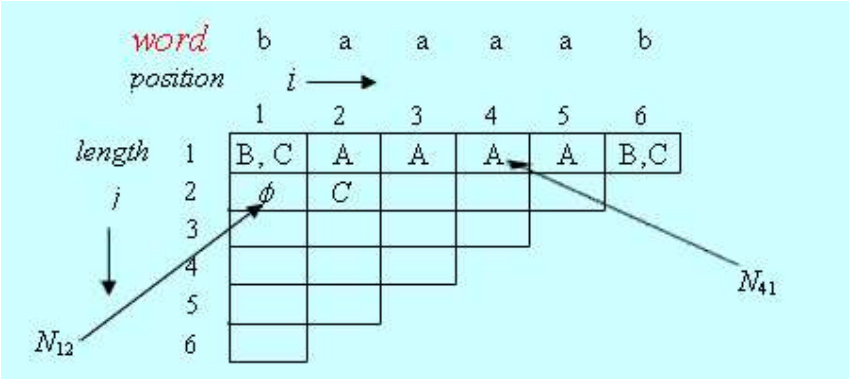
Initialize: $\forall i \quad N_{i1} = \{A \mid A \to x_{i1} \in P\}$

for *j* **:** = 2 to *n* do /* Determine $N_{i2}, N_{i3}, \cdots,$ for all *i* */

for *i* := 1 to *n-j*+1 do /* No sense in considering *i*, *j* with $i + j > n + 1$ for all *i* */
$N_{ij} := \phi$

for *k* := 1 to *j*-1 do /* try substrings of x_{ij} of length *k* */
$N_{ij} = N_{ij} \cup \{A \mid A \to BC \in P,\ B \in N_{i,k},\ C \in N_{i+k,j-k}\}$

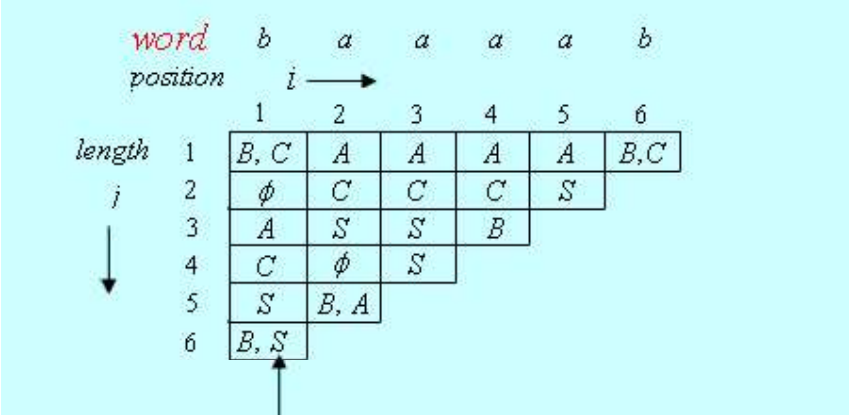
- The correctness of the algorithm can be proved by applying induction on *j* that whenever the outer loop finishes for particular *j*, the set N_{ij} contains all non-terminals A that can derive x_{ij} (for all i).
- It is easy to conclude that the time complexity of this algorithm is $O(n^3)$ where $n = |x|$ and grammar *G* is "fixed" in the sense that the size of the grammar is not considered as input in measuring complexity.
- **Example :** Consider the CFG:
  $S \to AB \mid AC$
  $A \to BC \mid a$
  $B \to CB \mid b$
  $C \to AA \mid b$
  Let us decide the membership for the string ***x* = *baaaab*** using the CYK algorithm.
  The table for N_{ij}'s is shown below.
  Word**:** *b a a a a b*



- Cell *i, j* will contain N_{ij}
- The top row is filled in by the first step of the algorithm e.g. $A \in N_{41}$, because $A \to a\,(= x_{41})$ is a production. We can compute the contents of the second row by using the contents of the first row (already done) and inspecting the grammar. For example, to compute N_{12} (i.e. the set of non-terminals that derive $x_{12} = ba$) we notice that $X \in N_{12}$ if $X \overset{*}{\Rightarrow} ba$ or if $X \to BA$ or $X \to CA$ is a production since no such production exists, we have $N_{12} = \phi$.

Similarly since $C \to AA$ is a production and $A \in N_{21}$, $A \in N_{31}$ we put $C \in N_{22}$.

- $x_{13} = baa$ isider the first element of the th x_{13} r $x_{11}x_{22}$ N_{13} (c $x_{12}x_{21}$ onding to the string). There are two ways to break up , and
- Consider $x_{13} = x_{11}x_{22}$. Since $B \in N_{11}$, $C \in N_{22}$ and $A \to BC$ is a production, we put $A \in N_{13}$. (If we consider, the other way i.e. $x_{13} = x_{12}x_{21}$ we find that $N_{12} = \phi$ and hence no more symbols can be added to N_{13}).
- Continuing this way we fill up the whole table as given below



Figure

$x = x_{16} = baaaab$

Since $S \in N_{16}$, $S \overset{*}{\Rightarrow} x_{16} (= baaaab)$

Hence *baaaab* is a member of the language generated by the grammar.

8.5 TESTING FINITENESS OF A CFL

We now show that there exist algorithms to decide finiteness of a CFL. Let L be a CFL. Then there is some pumping lemma constant n for L. The following algorithm derives the finiteness of L.

8.5.1 DECISION ALGORITHM FOR TESTING FINITENESS OF A CFL:

1. Test all input strings begining with those of length n (in non-decreasing order of length) for membership. (we already have developed algorithm for testing membership).

- If there is a string x with length $n \le |x| < 2n$ such that $x \in L$, then L is infinite otherwise L is finite.

Proof: If $|x| \ge n$ and $x \in L$, then x can be pumped according to the pumping lemma and the language is infinite. We need to test strings of length less than 2n only . Because if there were a string $z = uvwxy$ of length 2n or longer, we can always find a shorter string $uwy \in L$, (by

pumping lemma), but it is atmost n shorter. Thus if there are any strings of length 2n or more we can repeatedly cut out the substring vx to get, eventually, a string whose length is in the range n to 2n-1.

8.6 CONCLUSION

This module explains about the basic understanding of Context Free Languages (CFLs). It discusses Closure Property of Context Free Languages (CFLs), Some Decision Algorithms for CFLs and Testing Finiteness of a CFL through various theorems, lemmas and step-wise elaborated solved examples.

8.7 CHECK YOUR PROGRESS

Fill in the blanks:

1) Pumping Lemma can be used in the same way to show that certain languages are_____

2) CYK Algorithm_____ membership in CFL

3) If L is a CFL and R is a regular language, then _____ is a CFL.

4) A CFL's are not closed under_____

5) CFL's are closed under _____

8.8 ANSWER CHECK YOUR PROGRESS

1) not context-free

2) To decide

3) L intersection R

4) Complementation

5) Reversal.

8.9 MODEL QUESTION

Qs-1) What is Pumping Lemma why it is used?

Qs-2) Context Free Languages (CFLs) are not closed under intersection explain with the help of example?

Qs-3) How to test Finiteness of Context Free Languages (CFL)?

Qs-4) Explain CYK Algorithm?

Qs-5) What is Context Free Languages (CFL)? How to test Emptiness of Context Free Languages (CFL)?

8.10 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

8.11 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

Block-III

UNIT-IX TURING MACHINES

9.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Turing Machines (TMs). It explains Informal Description and Formal Definition of Turing Machines. The chapter discusses Transition Function, Instantaneous Description (IDs) or Configurations of a TM, Moves of Turing Machines, Special Boundary Cases and some more concepts about Configuration and Acceptance through proper elaborations.

9.2 INFORMAL DESCRIPTION

We consider here a basic model of TM which is deterministic and have one-tape. There are many variations, all are equally powerful.

The basic model of TM has a finite set of states, a semi-infinite tape that has a leftmost cell but is infinite to the right and a tape head that can move left and right over the tape, reading and writing symbols.

For any input w with |w|=n, initially it is written on the n leftmost (contiguous) tape cells. The infinitely many cells to the right of the input all contain a blank symbol, B which is a special tape symbol that is not an input symbol. The machine starts in its start state with its head scanning the leftmost symbol of the input w. Depending upon the symbol scanned by the tape head and the current state the machine makes a move which consists of the following:

- writes a new symbol on that tape cell,
- moves its head one cell either to the left or to the right and
- (possibly) enters a new state.

The action it takes in each step is determined by a transition function. The machine continues computing (i.e., making moves) until

- it decides to "accept" its input by entering a special state called accept or final state or
- halts without accepting i.e., rejecting the input when there is no move defined.

On some inputs the TM many keep on computing forever without ever accepting or rejecting the input, in which case it is said to "loop" on that input.

9.3 FORMAL DEFINITION

Formally, a Deterministic Turing machine (DTM) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Where,

- Q is a finite nonempty set of states.
- Γ is a finite non-empty set of tape symbols, called the tape alphabet of *M*.

- $\Sigma \subseteq \Gamma$ is a finite non-empty set of input symbols, called the input alphabet of *M*.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L \times R\}$ is the transition function of *M*,
- $q_0 \in Q$ is the initial or start state.
- $B \in \Gamma \setminus \Sigma$ is the blank symbol
- $F \subseteq Q$ is the set of final state.

So, given the current state and tape symbol being read, the transition function describes the next state, symbol to be written on the tape, and the direction in which to move the tape head (L and R denote left and right, respectively).

9.4 TRANSITION FUNCTION (δ)

- The heart of the TM is the transition function, δ because it tells us how the machine gets one step to the next.
- when the machine is in a certain state q$\in$Q and the head is currently scanning the tape symbol $X \in \Gamma$, and if $\delta(q, x) = (p, Y, D)$, then the machine

1. replaces the symbol *X* by *Y* on the tape
2. goes to state *p*, and
3. the tape head moves one cell (i.e., one tape symbol) to the left (or right) if *D* is *L* (or *R*).

9.5 INSTANTANEOUS DESCRIPTION (IDs) OR CONFIGURATIONS OF A TM

The ID (instantaneous description) of a TM capture what is going out at any moment i.e., it contains all the information to exactly capture the "current state of the computations".

It contains the following:

- The current state, *q*
- The position of the tape head,
- The constants of the tape up to the rightmost nonblank symbol or the symbol to the left of the head, whichever is rightmost.

Note that, although there is no limit on how far right the head may move and write nonblank symbols on the tape, at any finite time, the TM has visited only a finite prefix of the infinite tape.

An ID (or configuration) of a TM *M* is denoted by $\alpha q \beta$ where $\alpha, \beta \in \Gamma^*$ and

- α is the tape contents to the left of the head.
- q is the current state.
- β is the tape contents at or to the right of the tape head.

That is, the tape head is currently scanning the leftmost tape symbol of β. (Note that if $\beta = \in$, then the tape head is scanning a blank symbol)

If q_0 is the start state and w is the input to a TM M then the starting or initial configuration of M is obviously denoted by $q_0 w$.

9.6 MOVES OF TURING MACHINES

To indicate one move we use the symbol $\vdash$. Similarly, zero, one, or more moves will be represented by $\vdash$. A move of a TM M is defined as follows.

- Let $\alpha Z q X \beta$ be an ID of *M* where $X, Z \in \Gamma$, $\alpha, \beta \in \Gamma^*$ and $q \in Q$.
- Let there exists a transition $\delta(q, X) = (p, Y, L)$ of *M*.

Then we write $\alpha Z q X \beta \vdash_M \alpha q Z Y \beta$ meaning that ID $\alpha Z q X \beta$ yields $\alpha Z q Y \beta$

- Alternatively, if $\delta(q, X) = (p, Y, R)$ is a transition of *M*, then we write $\alpha Z q X \beta \vdash \alpha z Y p \beta$ which means that the ID $\alpha z q X \beta$ yields $\alpha z Y p \beta$.

- In other words, when two IDs are related by the relation $\vdash$, we say that the first one yields the second (or the second is the result of the first) by one move.
- If IDj results from IDi by zero, one or more (finite) moves then we write $\vdash$ (If the TM *M* is understand, then the subscript *M* can be dropped from $\vdash$ or $\vdash$).

9.7 SPECIAL BOUNDARY CASES

- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, Y, L)$ be a transition of *M*. Then $\vdash$. That is, the head is not allowed to fall off the left end of the tape.
- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, Y, R)$ then figure (Note that $\alpha Y q$ is equivalent to $\alpha Y q B$)
- Let $q x \alpha$ be an ID and $\delta(q, x) = (p, B, R)$ then figure
- Let $\alpha z q x$ be an ID and $\delta(q, x) = (p, B, L)$ then figure

The language accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, denoted as L(M) is

L(M) = {w | $w \in \Sigma^*$ and figure for some p$\in$F and $\alpha, \beta \in \Gamma^*$ }

In other words, the TM M accepts a string $w \in \Sigma^*$ that cause M to enter a final or accepting state when started in its initial ID (i.e., $q_0 w$). That is a TM M accepts the string $w \in \Sigma^*$ if a sequence of IDs, $ID_1, ID_2, \cdots, ID_k$ exists such that

- ID_1 is the initial or starting ID of *M*
- $ID_i \vdash_M ID_{i+1}$; $1 \leq i < k$
- The representation of IDk contains an accepting state.

The set of strings that M accepts is the language of M, denoted L(M), as defined above.

CHECK YOUR PROGRESS

True/False type questions

1) The basic model of TM has a finite set of states_____

2) Formally, a Deterministic Turing machine (DTM) is a 7-tuple _____

3) The heart of the TM is the transition function_____

4) A deterministic TM is an 5-tuple_____

5) Turing Machine is accepted by Push down automata_____

Answers-

1) True

2) True

3) True

4) False

5) False

9.8 MORE ABOUT CONFIGURATION AND ACCEPTANCE

- An ID $\alpha q \beta$ of *M* is called an accepting (or final) ID if $q \in F$
- An ID $\alpha q x \beta$ is called a blocking (or halting) ID if $\delta(q, x)$ is undefined i.e. the TM has no move at this point.
- ID_j is called reactable from ID_i if $ID_i \vdash_M^* ID_j$
- $q_0 w$ is the initial (or starting) ID if $w \in \Sigma^*$ is the input to the TM and $q_0 \in Q$ is the initial (or start) state of *M*.

On any input string $w \in \Sigma^*$

either

- *M* halts on *w* if there exists a blocking (configuration) ID, I' such that $q_0 w \vdash_M^* I'$.

There are two cases to be considered

- *M* accepts *w* if *I* is an accepting ID. The set of all $w \in \Sigma^*$ accepted by *M* is denoted as *L*(*M*) as already defined
- *M* rejects *w* if I' is a blocking configuration. Denote by reject (*M*), the set of all $w \in \Sigma^*$ rejected by *M*.

or

- *M* loops on *w* if it does not halt on *w*.

Let *loop*(*M*) be the set of all $w \in \Sigma^*$ on which *M* loops for.

It is quite clear that

$$L(M) \cup reject(M) \cup loop(M) = \Sigma^*$$

That is, we assume that a TM *M* halts

- When it enters an accepting ID_1 or
- When it enters a blocking ID_1 i.e., when there is no next move.

However, on some input string, $w \notin L(M)$, it is possible that the TM M loops for ever i.e., it never halts.

It is observed that in the basic TM model there is no apparent way for the machine to "reject" the input string. And, instead of a single accepting state there is a set of accepting states. Considering these two facts, we define a new model which is equivalent (can be shown) to the basic TM model as follows:

A deterministic TM is an 8-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, q_a, q_r)$$

where,

$q_a \in Q$ is the accepting state

$q_r \in Q$ is the rejecting state

No transition are possible from q_a and q_r. All other elements of *M* remain same as defined in case of basic model.

The language accepted by the *M* is defined as

L(*M*) = { $w \mid w \in \Sigma^*$ and $q_0 w \vdash \alpha q_a \beta$ for some $\alpha, \beta \in \Gamma^*$ }

The TM *M* rejects a string $w \in \Sigma^*$ iff

- $q_0 w \vdash \alpha q_a \beta$ for some $\alpha, \beta \in \Gamma^*$ or
- *M* enters in an infinite loop on input *w* i.e., *M* never halts on *w*.
- *M* enters in a blocking ID on input *w* i.e., *M* never halts on *w*. or
- *M* enters in a blocking ID on input *w* i.e., $q_0 w \vdash_M^* \alpha p X \beta$ and $\delta(p, x)$ is undefined.

If M accepts w, we can determine it, because M eventually enters the accepting state q_a. But if M does not accept w, we may not be able to determine this since M may reject w by not halting.

This leads us to categorize the language accepted by the TMs into two broad classes as follows (Described in Next Module).

9.9 CONCLUSION

This module explains about the basic understanding of Turing Machines (TMs). It explains Informal Description and Formal Definition of Turing Machines. The module also discusses Transition Function, Instantaneous Description (IDs) or Configurations of a TM, Moves of Turing Machines, Special Boundary Cases and some important concepts about Configuration and Acceptance with proper elaborations.

9.10 CHECK YOUR PROGRESS

Fill in the Blanks:

1) The basic model of TM has a _____ set of states.

2) TM many keep on computing forever without ever accepting or rejecting the_____

3) A deterministic TM is an_____ 8-tuple

4) The language accepted by Turing Machine is_____

5) The heart of the TM is the_____

9.11 ANSWER CHECK YOUR PROGRESS

Answers:

1) Finite

2) Input

3) 8 Tuple

4) Recursive Language

5) Transition function

9.12 MODEL QUESTION

Qs-1) What is basis model of Turing machine. Explain with the helpnof example?

Qs-2) What is Transition function explain in detail?

Qs-3) What is instantaneous description or configurations of Turing machine?

Qs-4) What are Special boundary cases of Turing Machine?

Qs-5) What are moves of Turing Machine explain your answer?

9.13 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

9.14 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-X RECURSIVELY ENUMERABLE LANGUAGE

10.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Recursive language, Recursively Enumerable (R.E) Language. Recursive (Or Decidable) Languages, Closure Properties, Post Correspondence Problem and Proof Sketch of Un-decidability through various concepts and step-wise elaborated solved examples.

10.2 RECURSIVE LANGUAGE

In mathematics, logic and computer science, a formal language (a set of finite sequences of symbols taken from a fixed alphabet) is called recursive if it is a recursive subset of the set of all possible finite sequences over the alphabet of the language. Equivalently, a formal language is recursive if there exists a total Turing machine (a Turing machine that halts for every given input) that, when given a finite sequence of symbols from the alphabet of the language as input (any string containing only characters in the language's alphabet) accepts only those that are part of the language and rejects all other strings. Recursive languages are also called decidable.

The concept of decidability may be extended to other models of computation. For example, one may speak of languages decidable on a non-deterministic Turing machine. Therefore, whenever an ambiguity is possible, the synonym for "recursive language" used is Turing-decidable language, rather than simply decidable.

The class of all recursive languages is often called R, although this name is also used for the class RP. This type of language was not defined in the Chomsky hierarchy. All recursive languages are also recursively enumerable. All regular, context-free and context-sensitive languages are recursive. There exist three equivalent major definitions for the concept of a recursively enumerable language.

---

1. A recursively enumerable language is a recursively enumerable subset in the set of all possible worlds over the alphabet of the language.
2. A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) which will enumerate all valid strings of the language. Note that if the language is infinite, the enumerating algorithm provided can be chosen so that it avoids repetitions, since we can test whether the string produced for number n is "already" produced for a number which is less than n. If it already is produced, use the output for input n+1 instead (recursively), but again, test whether it is "new".
3. A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input but may either halt and reject or loop forever when presented with a string not in the language. Contrast this to recursive languages, which require that the Turing machine halts in all cases.

All regular, context-free, context-sensitive and recursive languages are recursively enumerable. Post's theorem shows that RE, together with its complement co-RE, correspond to the first level of the arithmetical hierarchy.

10.2.1 Recursively Enumerable (R.E.) Language:

It is also called as TM-recognizable language or semi-decidable language. Simply speaking, a language L is recursively enumerable if some Turing Machine accepts it. Formally, the class of r.e. languages is defined as

{ L | $L \subseteq \Sigma^*$ and $\exists$ TM M such that L = L(M) }

- So, on input string $w \in L$, *M* enters an accepting ID and halts.
- But , on input strings $w \notin L$, *M* either halts entering a bloueing ID (i.e. without entering an accepting ID), or it never halts (i.e. it loops for ever).

10.2.2 Recursive (or decidable) Languages

A language L is recursive if there is some TM M that halts on every input $w \in \Sigma^*$ and L= L(M).

Formally, the class of recursive language is defined as

{ L | $L \subseteq \Sigma^*$ and $\exists$ TM M such that M halts $\forall\, w \in \Sigma^*$ and L = L(M) }

- So, on any input strings $w \in L$, *M* enters an accepting ID and halts and
- On an input string $w \notin L$, *M* halts entering in a blocking ID (or entering in a reject state).

10.2.3 Example

The Halting problem is recursively enumerable but not recursive. Indeed, one can run the Turing Machine and accept if the machine halts, hence it is r.e. On the other hand, the problem is undecidable.

Some other RE, languages are:

- Post correspondence problem
- Mortality (computability theory)
- Entscheidungs problem

10.3 CLOSURE PROPERTIES

Recursively enumerable languages are closed under the following operations. That is, if L and P are two recursively enumerable languages, then the following languages are recursively enumerable as well:

- The Kleene star of L
- The concatenation of L and P
- The union
- The intersection

Note that recursively enumerable languages are not closed under set difference or complementation. The set difference L - P may or may not be recursively enumerable. If L is recursively enumerable, then the complement of L is recursively enumerable if and only if L is also recursive.

10.4 POST CORRESPONDENCE PROBLEM

The Post correspondence problem is an undecidable decision problem that was introduced by Emil Post in 1946. Because it is simpler than the halting problem and the Entscheidungs problem it is often used in proofs of undecidability.

Definition of the problem:

The input of the problem consists of two finite lists and {\displaystyle \beta _{1},\ldots ,\beta _{N}} of words over some alphabet having at least two symbols. A solution to this problem is a sequence of indices {\displaystyle (i_{k})_{1\leq k\leq K}}with and {\displaystyle 1\leq i_{k}\leq N} for all, such that the decision problem then is to decide whether such a solution exists or not.

Example instances of the problem

Example 1

Consider the following two lists:

α_1	α_2	α_3
a	*ab*	*bba*
β_1	β_2	β_3
baa	*aa*	*bb*

A solution to this problem would be the sequence (3, 2, 3, 1), because
Furthermore, since (3, 2, 3, 1) is a solution, so are all of its "repetitions", such as (3, 2, 3, 1, 3, 2, 3, 1), etc.; that is, when a solution exists, there are infinitely many solutions of this repetitive kind.

However, if the two lists had consisted of only {\displaystyle \alpha _{2},\alpha _{3}}

and {\displaystyle \beta _{2},\beta _{3}} from those sets, then there would have been no solution (the last letter of any such α string is not the same as the letter before it, whereas β only constructs pairs of the same letter).

A convenient way to view an instance of a Post correspondence problem is as a collection of blocks of the form

αi
βi

there being an unlimited supply of each type of block. Thus the above example is viewed as

a	ab	bba
baa	aa	bb
i = 1	i = 2	i = 3

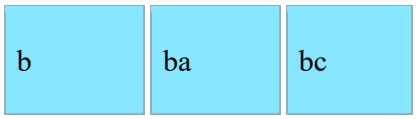
where the solver has an endless supply of each of these three block types. A solution corresponds to some way of laying blocks next to each other so that the string in the top cells corresponds to the string in the bottom cells. Then the solution to the above example corresponds to:

bba	ab	bba	A
bb	aa	bb	Baa
i1 = 3	i2 = 2	i3 = 3	i4 = 1

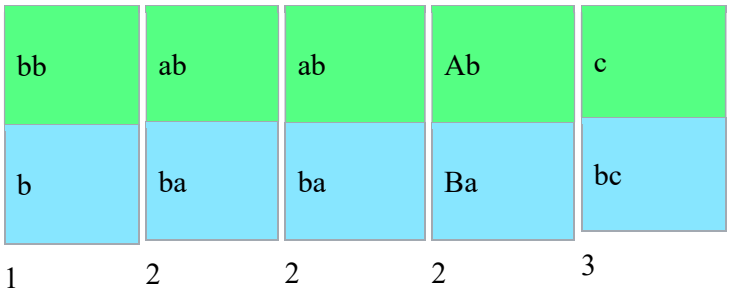
Example 2

Again using blocks to represent an instance of the problem, the following is an example that has infinitely many solutions in addition to the kind obtained by merely "repeating" a solution.

bb	ab	c



In this instance, every sequence of the form (1, 2, 2, . . ., 2, 3) is a solution (in addition to all their repetitions):



CHECK YOUR PROGRESS

True/False type questions

1) A formal language is recursive if there exists a total Turing machine_____

2) Recursively enumerable languages are closed under intersection _____

3) The Halting problem is recursively enumerable but not recursive_____

4) The Post correspondence problem is an decidable decision _____

5) Recursively enumerable languages are not closed under union_____

Answers-

1) True
2) True
3) True
4) False
5) False

10.5 PROOF SKETCH OF UNDECIDABILITY

The most common proof for the undecidability of PCP describes an instance of PCP that can simulate the computation of a Turing machine on a particular input. A match will only occur if the input would be accepted by the Turing machine. Because deciding if a Turing machine will accept an input is a basic undecidable problem, PCP cannot be decidable either. The following discussion is based on Michael Sipser's textbook Introduction to the Theory of Computation.

In more detail, the idea is that the string along the top and bottom will be a computation history of the Turing machine's computation. This means it will list a string describing the initial state, followed by a string describing the next state, and so on until it ends with a string describing an accepting state. The state strings are separated by some separator symbol (usually written #). According to the definition of a Turing machine, the full state of the machine consists of three parts:

- The current contents of the tape.
- The current state of the finite state machine which operates the tape head.
- The current position of the tape head on the tape.

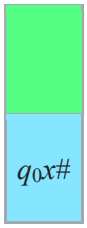
Although the tape has infinitely many cells, only some finite prefix of these will be non-blank. We write these down as part of our state. To describe the state of the finite control, we create new symbols, labelled q1 through qk, for each of the finite state machine's k states. We insert the correct symbol into the string describing the tape's contents at the position of the tape head, thereby indicating both the tape head's position and the current state of the finite control. For the alphabet {0, 1}, a typical state might look something like:

101101110q700110.

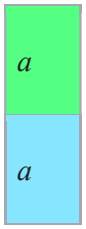
A simple computation history would then look something like this:

q0101#1q401#11q21#1q810.

We start out with this block, where x is the input string and q0 is the start state:

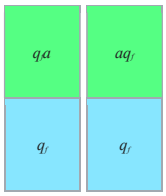


The top starts out "lagging" the bottom by one state, and keeps this lag until the very end stage. Next, for each symbol a in the tape alphabet, as well as #, we have a "copy" block, which copies it unmodified from one state to the next:



We also have a block for each position transition the machine can make, showing how the tape head moves, how the finite state changes, and what happens to the surrounding symbols. For example, here the tape head is over a 0 in state 4, and then writes a 1 and moves right, changing to state 7:

Finally, when the top reaches an accepting state, the bottom needs a chance to finally catch up to complete the match. To allow this, we extend the computation so that once an accepting state is reached, each subsequent machine step will cause a symbol near the tape head to vanish, one at a time, until none remain. If qf is an accepting state, we can represent this with the following transition blocks, where a is a tape alphabet symbol:

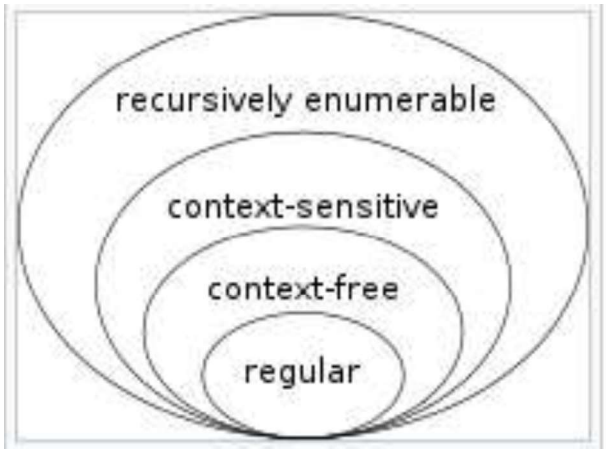


There are a number of details to work out, such as dealing with boundaries between states, making sure that our initial tile goes first in the match, and so on, but this shows the general idea of how a static tile puzzle can simulate a Turing machine computation.

The previous example

q0101#1q401#11q21#1q810.

is represented as the following solution to the Post correspondence problem:



Source: https://en.formulasearchengine.com/wiki/Chomsky_hierarchy#/media/File:Chomsky-hierarchy.svg

10.6 CONCLUSION

This module explains about the basic understanding of Recursive languages. It discusses Recursively Enumerable (R.E) Language. Recursive (Or Decidable) Languages, Closure Properties, Post Correspondence Problem and Proof Sketch of Undecidability through various concepts and step-wise elaborated solved examples.

10.7 CHECK YOUR PROGRESS

Fill in the blanks:

1) Recursive languages are also called _____

2) The Halting problem is recursively enumerable but not _____

3) All regular, context-free, context-sensitive and recursive languages are_____

4) A formal language is recursive if there exists a total_____

5) The Post correspondence problem is an _____

10.8 ANSWER CHECK YOUR PROGRESS

1) Decidable.

2) Recursive

3) Recursively enumerable.

4) Turing machine

5) Undecidable decision

10.9 MODEL QUESTION

Qs-1) Explain closure property for Recursive enumerable?

Qs-2) What is post correspondence problem?

Qs-3) Explain with the diagram Chomsky hierarchy?

Qs-4) Explain Recursive enumerable language?

Qs-5) What are three parts of full state of Turing machine?

Qs-6) Is the set of all definable subsets of the natural numbers recursively enumerable?

10.10 REFERENCES

- https://nptel.ac.in/courses/106/103/106103070/
- Sipser, M. (1996), Introduction to the Theory of Computation, PWS Publishing Co.
- Kozen, D.C. (1997), Automata and Computability, Springer.
- https://en.formulasearchengine.com/wiki/Recursive_language

10.11 SUGGESTED READINGS

1. Martin J. C., "Introduction to Languages and Theory of Computations", TMH

2. Papadimitrou, C. and Lewis, C.L., "Elements of theory of Computations", PHI

3. Cohen D. I. A., "Introduction to Computer theory", John Wiley & Sons

4. Kumar Rajendra, "Theory of Automata (Languages and Computation)", PPM

UNIT-XI POST'S CORRESPONDENCE PROBLEM

11.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Post's Correspondence Problem (PCP). It explains Post's Correspondence System (PCS) through various theorems, corollaries, and lemmas along with step-wise elaborated solved examples.

11.2 POST'S CORRESPONDENCE PROBLEM (PCP)

Theorem 11.1.1: Given any two CFG's G1 and G2 the question "Is $L(G_1) \cap L(G_2) = \phi ?$" is undecidable.

Proof: Assume for contradiction that there exists an algorithm *A* to decide this question. This would imply that PCP is decidable as shown below.

For any Post Correspondence System, *P* construct grammars G_x and G_y by using the constructions elaborated already. We can now use the algorithm *A* to decide whether $L(G_x) \cap L(G_y) = \phi$ and $L(G_x) \cap L(G_y) \neq \phi$ Thus, PCP is decidable, a contradiction. So, such an algorithm does not exist.

If G_x and G_y are CFG's constructed from any arbitrary Post Correspondence System, than it is not difficult to show that $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are also context-free, even though the class of context-free languages are not closed under complementation.

$L(G_x), L(G_y)$ and their complements can be used in various ways to show that many other questions related to CFL's are undecidable. We prove here some of those.

Theorem 11.1.2: Foe any two arbitrary CFG's $G_1 \& G_2$, the following questions are undecidable.

i. Is $L(G_1) = \Sigma^* ?$
ii. Is $L(G_1) = L(G_2) ?$
iii. Is $L(G_1) \subseteq L(G_2) ?$

Proof:

i. If $L(G_1) = \Sigma^*$ then, $\overline{L(G_1)} = \phi$

Hence, it suffice to show that the question "Is $L(G_1) = \phi ?$ " is undecidable.

Since, $\overline{L(G_x)}$ and $\overline{L(G_y)}$ are CFl's and CFL's are closed under union, $L = \overline{L(G_x)} \cup \overline{L(G_y)}$ is also context-free. By DeMorgan's theorem, $\overline{L} = L(G_x) \cap L(G_y)$

If there is an algorithm to decide whether $L(G_1) = \phi$ we can use it to decide whether $\overline{L} = L(G_x) \cap L(G_y) = \phi$ or not. But this problem has already been proved to be undecidable.

Hence there is no such algorithm to decide or not. $L(G_1) = \phi$

Let P be any arbitrary Post correspondence system and G_x and G_y are CFg's constructed from the pairs of strings.

$L_1 = \overline{L(G_x)} \cup \overline{L(G_y)}$ must be a CFL and let G1generates L1. That is,

$$L_1 = L(G_1) = \overline{L(G_x)} \cup \overline{L(G_y)} = \overline{L(G_x) \cap L(G_y)}$$

by De Morgan's theorem, as shown already, any string, $w \in L(G_x) \cap L(G_y)$ represents a solution to the PCP. Hence, $L(G_1)$ contains all but those strings representing the solution to the PCP.

Let $L(G_2) = (\Sigma \cup \{1, 2, \cdots, n\})^*$ for same CFG G2.

It is now obvious that $L(G_1) = L(G_2)$ if and only if the PCP has no solutions, which is already proved to be undecidable. Hence, the question "Is $L(G_1) = L(G_2)$?" is undecidable.

Let G_1 be a CFG generating the language $(\Sigma \cup \{1, 2, \cdots n\})^*$ and G2 be a CFG generating $\overline{L(G_x)} \cup \overline{L(G_y)}$ where G_x and G_y are CFG.s constructed from same arbitrary instance of PCP.

$$L(G_1) \subseteq L(G_2) \text{ iff } \overline{L(G_x)} \cup \overline{L(G_y)} = (\Sigma \cup \{1, 2, \cdots n\})^*$$

i.e. iff the PCP instance has no solutions as discussed in part (ii).

Hence the proof.

Theorem 11.1.3: It is undecidable whether an arbitrary CFG is ambiguous.

Proof : Consider an arbitrary instance of PCP and construct the CFG's G_x and G_y from the ordered pairs of strings.

We construct a new grammar *G* from G_x and G_y as follows.

$G = (N, \Sigma, P, S)$ where

$N = \{S, S_x, S_y\}$,

Σ is same as that of G_x and G_y.

$P = \{P_x, \cup P_y \cup \{S \to S_x | S_y\}$

This construction gives a reduction of PCP to the -------- of whether a CFG is ambiguous, thus leading to the undecidability of the given problem. That is, we will now show that the PCP has a solution if and only if *G* is ambiguous. (where *G* is constructed from an arbitrary instance of PCP).

Proof: Consider an arbitrary instance of PCP and construct the CFG's G_x and G_y from the ordered pairs of strings.

We construct a new grammar *G* from G_x and G_y as follows.

$G = (N, \Sigma, P, S)$ where

$N = \{S, S_x, S_y\}$,

Σ is same as that of G_x and G_y.

$P = \{P_x, \cup P_y \cup \{S \to S_x | S_y\}$

This construction gives a reduction of PCP to the -------- of whether a CFG is ambiguous, thus leading to the undecidability of the given problem. That is, we will now show that the PCP has a solution if and only if *G* is ambiguous. (where *G* is constructed from an arbitrary instance of PCP).

Only if Assume that $i_1, i_2, \cdots i_k$ is a solution sequence to this instance of PCP.

Consider the following two derivation in $i_1, i_2, \cdots i_k$.

$$S \overset{1}{\underset{G}{\Rightarrow}} S_x \overset{1}{\underset{G}{\Rightarrow}} x_{i_1} S_x i_1 \overset{1}{\underset{G}{\Rightarrow}} x_{i_1} x_{i_2} S_x i_2 i_1$$
$$\overset{*}{\underset{G}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_{k-1}} S_x i_{k-1} \cdots i_2 i_1$$
$$\overset{1}{\underset{G}{\Rightarrow}} x_{i_1} x_{i_2} \cdots x_{i_{k-1}} x_{i_k} i_k i_{k-1} \cdots i_2 i_1$$

$$S \overset{1}{\underset{G}{\Rightarrow}} S_y \overset{1}{\underset{G}{\Rightarrow}} y_{i_1} S_y i_1 \overset{1}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} S_y i_2 i_1$$
$$\overset{*}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} \cdots x y_{i_{k-1}} S_y i_{k-1} \cdots i_2 i_1$$
$$\overset{1}{\underset{G}{\Rightarrow}} y_{i_1} y_{i_2} \cdots y_{i_{k-1}} y_k i_k i_{k-1} \cdots i_2 i_1$$

But,

$$x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_k, \text{ since } \cdots i_1, i_2, \cdots i_k$$

is a solution to the PCP. Hence the same string of terminals $\left(x_{i_1} x_{i_2} \cdots x_{i_k}\right)$ has two derivations. Both these derivations are, clearly, leftmost. Hence G is ambiguous.

If It is important to note that any string of terminals cannot have more than one derivation in G_x and G_y Because, every terminal string which are derivable under these grammars ends with a sequence of integers $i_k i_{k-1} \cdots i_2 i_1$ This sequence uniquely determines which productions must be used at every step of the derivation.

Hence, if a terminal string, $w \in L(G)$, has two leftmost derivations, then one of them must begin with the step.

$S \overset{1}{\underset{G}{\Rightarrow}} S_x$ and thus continues with derivation under G_x, and the other must begin with the step $S \overset{1}{\underset{G}{\Rightarrow}} S_y$ and then continues with derivations under G_y.

In both derivations the resulting string must end with a sequence $i_p i_{p-1} \cdots i_2 i_1$ for same $p \geq 1$ The reverse of this sequence must be a solution to the PCP, because the string that precede in one case is $x_1 x_2 \cdots x_{i_{p-1}} x_{i_p}$ and $y_1 y_2 \cdots y_{i_{p-1}} y_{i_p}$ in the other case. Since the string derived in both cases are identical, the sequence $i_1, i_2, \cdots i_{p-1}, i_p$

must be a solution to the PCP.

Hence the proof.

In both derivations the resulting string must end with a sequence $i_p i_{p-1} \cdots i_2 i_1$ for same $p \geq 1$ The reverse of this sequence must be a solution to the PCP, because the string that precede in one case is $x_1 x_2 \cdots x_{i_{p-1}} x_{i_p}$ and $y_1 y_2 \cdots y_{i_{p-1}} y_{i_p}$ in the other case. Since the string derived in both cases are identical, the sequence $i_1, i_2, \cdots i_{p-1}, i_p$

must be a solution to the PCP.

Hence the proof.

CHECK YOUR PROGRESS

True/False type questions

1)Any two CFG's G1 and G2 the question "Is $L(G_1) \cap L(G_2) = \phi ?$" is undecidable._____

2) It is undecidable whether an arbitrary CFG is ambiguous._____

3) PCP over one-letter alphabet is undecidable._____

4) There is no algorithm that determines whether an arbitrary Post Correspondence System has a solution_____

5) Some decidable problem in context-free languages_____

Answers-

1) True

2) True

3) False

4) True

5) False

11.3 POST'S CORRESPONDENCE SYSTEM (PCS)

A post correspondence system consists of a finite set of ordered pairs $(x_i, y_i),\ i = 1,2,\cdots,n,$ where $x_i, y_i \in \Sigma^+$ for some alphabet Σ.Any sequence of numbers $i_1, i_2, \cdots, i_k \quad s-t.$ is called a solution to a Post Correspondence System.

$x_{i_1}, x_{i_2}, \cdots, x_{i_k} = y_{i_1}, y_{i_2}, \cdots, y_{i_k}$ The Post's Correspondence Problem is the problem of determining whether a Post Correspondence system has a solution.

Example 1: Consider the post correspondence system

$$\{(aa, aab), (bb, ba), (abb, b)\}$$

The list 1,2,1,3 is a solution to it.

Because

$$x_1 x_2 x_1 x_3 = y_1 y_2 y_1 y_3$$

$$\underbrace{aa}_{x_1}\underbrace{bb}_{x_2}\underbrace{aa}_{x_1}\underbrace{abb}_{x_3} = \underbrace{aab}_{y_1}\underbrace{ba}_{y_2}\underbrace{aab}_{y_1}\underbrace{b}_{y_3}$$

$$aabbaaabb = aabbaaabb$$

I	x_i	y_i
1	*aa*	*aab*
2	*bb*	*ba*
3	*abb*	*b*

(A post correspondence system is also denoted as an instance of the PCP)

Example 2: The following PCP instance has no solution

I	x_i	y_i
1	*aab*	*aa*
2	*a*	*baa*

This can be proved as follows. (x_2, y_2) cannot be chosen at the start, since than the LHS and RHS would differ in the first symbol (a in LHS and $'b'$ in RHS). So, we must start

with (x_1, y_1). The next pair must be (x_2, y_2) so that the 3 rd symbol in the RHS becomes identical to that of the LHS, which is a . After this step, LHS and RHS are not matching. If (x_1, y_1) is selected next, then would be mismatched in the 7 th symbol (b in LHS and a in RHS). If (x_2, y_2) is selected, instead, there will not be any choice to match the both side in the next step.

Example 3: The list 1,3,2,3 is a solution to the following PCP instance.

i	x_i	y_i
1	1	101
2	10	00
3	011	11

The following properties can easily be proved.

Proposition: The Post Correspondence System

$\{(a^{i_1}, a^{j_1}), (a^{i_2}, a^{j_2}), \cdots, (a^{i_n}, a^{j_n})\}$ has solutions if and only if

$\exists k$ such that $i_k = j_k$ or
$\exists k$ and l such that $i_k > j_k$ and $i_l < j_l$

Corollary: PCP over one-letter alphabet is decidable.

Proposition : Any PCP instance over an alphabet Σ with $|\Sigma| \geq 2$ is equivalent to a PCP instance over an alphabet Γ with $|\Gamma| = 2$

Proof : Let $\Sigma = \{a_1, a_2, \cdots, a_k\}, k > 2.$

Consider $\Gamma = \{0,1\}$ We can now encode every $a_i \in \Sigma, 1 \leq i \leq k$ as $10^i 1.$ any PCP instance over Σ will now have only two symbols, 0 and 1 and, hence, is equivalent to a PCP instance over Γ

Theorem 11.2.1: PCP is undecidable. That is, there is no algorithm that determines whether an arbitrary Post Correspondence System has a solution.

Proof: The halting problem of turning machine can be reduced to PCP to show the undecidability of PCP. Since halting problem of TM is undecidable (already proved), This reduction shows that PCP is also undecidable. The proof is little bit lengthy and left as an exercise.

Some undecidable problem in context-free languages

We can use the undecidability of PCP to show that many problem concerning the context-free languages are undecidable. To prove this we reduce the PCP to each of these problem. The following discussion makes it clear how PCP can be used to serve this purpose.

Let $\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ be a Post Correspondence System over the alphabet Σ. We construct two CFG's Gx and Gy from the ordered pairs x,y respectively as follows.

$G_x=(N_x,\Sigma_x,P_x,S_x)$ and

$G_y=(N_y,\Sigma_y,P_y,S_y)$ where

$G_y=(N_y,\Sigma_y,P_y,S_y)$

$N_x=\{S_x\}$ and $N_y=\{S_y\}$

$\Sigma_x=\Sigma_y=\Sigma\cup\{1,2,\cdots,n\}$,

$P_x=\{S_x\to x_iS_xi,\ S_x\to x_ii \mid i=1,2,\cdots,n\}$

and $P_y=\{S_y\to y_iS_yi,\ S_y\to y_ii \mid i=1,2,\cdots n\}$

it is clear that the grammar G_x generates the strings that can appear in the LHS of a sequence while solving the PCP followed by a sequence of numbers. The sequence of number at the end records the sequence of strings from the PCP instance (in reverse order) that generates the string. Similarly, G_y generates the strings that can be obtained from the RHS of a sequence and the corresponding sequence of numbers (in reverse order).

Now, if the Post Correspondence System has a solution, then there must be a sequence

$i_1i_2,\cdots,i_k$ s.t

$x_{i_1}x_{i_2}\cdots x_{i_k}=y_{i_1}y_{i_2}\cdots y_{i_k}$

According to the construction of G_x and G_y

$S_x\overset{*}{\underset{G_x}{\Rightarrow}}x_{i_1}x_{i_2}\cdots x_{i_k}i_ki_{k-1}\ldots i_2i_1$ and

$S_y\overset{*}{\underset{G_y}{\Rightarrow}}y_{i_1}y_{i_2}\cdots y_{i_k}i_ki_{k-1}\cdots i_2i_1$

In this case

$$x_{i_1} x_{i_2} \cdots x_{i_k} i_k \cdots i_2 i_1 = y_{i_1} y_{i_2} \cdots i_{i_k} i_k \cdots i_2 i_1 = w(say)$$

Hence , $w \in L(G_x)$ and $w \in L(G_y)$ implying

$$L(G_x) \cap L(G_y) \neq \phi$$

Conversely, let $w \in L(G_x) \cap L(G_y)$

Hence, *w* must be in the form w1w2 where $w_1 \in \Sigma^*$ and w2 in a sequence $i_k i_{k-1} \cdots i_2 i_1$ (since, only that kind of strings can be generated by each of G_x and G_y).

Now, the string $w_1 = x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$ is a solution to the Post Correspondence System.

It is interesting to note that we have here reduced PCP to the language of pairs of CFG,s whose intersection is nonempty. The following result is a direct conclusion of the above.

11.4 CONCLUSION

This module explains about the basic understanding of Post's Correspondence Problem (PCP). It discusses Post's Correspondence System (PCS) through various theorems, corollaries and lemmas along with step-wise elaborated solved examples.

11.5 CHECK YOUR PROGRESS

Fill in the blanks

1) A post correspondence system consists of a _____ set of ordered pairs.

2) PCP over one-letter alphabet is _____

3) Given any two CFG's G1 and G2 the question "Is $L(G_1) \cap L(G_2) = \phi ?$" is_____

4) It is undecidable whether an arbitrary CFG is _____

5) The full form of PCP is _____

11.6 ANSWER CHECK YOUR PROGRESS

1) Finite

2) Decidable.

3) Undecidable

4) Ambiguous.

5) Post correspondence problem.

11.7 MODEL QUESTION

Qs-1) Explain Post's Correspondence Problem (PCP) in brief?

Qs-2) It is undecidable whether an arbitrary CFG is ambiguous. explain?

Qs-3) PCP over one-letter alphabet is decidable. Explain?

Qs-4) : For any two arbitrary CFG's G1 and G2, what are three conditions that are undecidable?

Qs-5) Explain the halting problem of turning machine can be reduced to PCP to show the undecidability.

11.8 REFERENCES

https://nptel.ac.in/courses/106/103/106103070/

11.9 SUGGESTED READINGS

1. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH

2. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI

3. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons

4. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM

UNIT-XII CHOMSKY HIERARCHY

12.1 LEARNING OBJECTIVES

This chapter gives the basic understanding of Chomsky Hierarchy. It explains Equivalence of Unrestricted grammars, Turing Machines (TMs), Context-Sensitive Language and Linear-bounded Automata (LBAs). It also discusses the Equivalence of LBAs and Context-sensitive Grammars through various theorems, lemmas and step-wise elaborated solved examples.

12.2 CHOMSKY HIERARCHY

The famous linguistic Noam Chomsky attempted to formalize the notion of grammar and languages in the 1950s. This effort, due to Chomsky, resulted in the definition of the "Chomsky Hierarchy", a hierarchy of language classes defined by gradually increasing the restrictions on the form of the productions. Chomsky numbered the four families of grammars (and languages) that make up the hierarchy and are defined as below.

Let G = (N , Σ, P, S) be a grammar

1. *G* is called a Type-0 or unrestricted, or semi-there or phrase-structure grammar if all productions are of the form $\alpha \to \beta$, where $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^*$.
2. G is a Type-1 or context-sensitive grammar if each production $\alpha \to \beta$ in *P* satisfies $|\alpha| \leq |\beta|$ such that $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^+$. Type-1 grammar , by special dispensation , is allowed to have the production $S \to \in$, provided *S* does not appear on the right-hand side of any production.
3. G is a Type-2 or context-free grammar if each production $\alpha \to \beta$ in P satisfies $|\alpha| = 1$ i.e. α is a single nonterminal.
4. *G* is a Type-3 or right-linear or regular grammar if each production has one of the following three forms: $A \to bC$, $A \to b$, $A \to \in$ where *A,* $C \in$ *N* (with *A* = *C* allowed) and $c \in \Sigma$.
5. The language generated by a Type-*i* grammar is called a Type-*i* language, i = 0,1,2,3. A Type-*i* language is also called a context-sensitive language (CSL). We have already observed that a Type-2 language is also called a Context-Free Language (CFL) and a Type-3 language is also called a regular language. Each class of language in the Chomsky hierarchy is characterized as the language generated by a type of automata. These relationships have been summarised in the following table for convenience.
6.

Grammars	Languages	Automata
Type-0 , phrase-struct , semi-true, unrestricted grammars	Recursively enumerable language	Turing Machine
Type-0 , phrase-struct , semi-true, unrestricted grammars	Context-sensitive language	Linear-bounded automata
Type-2, context-free grammars	Context-free language	Pushdown Automata

Type-3, regular, right-linear, left-linear grammar	Regular Language	Finite Automata

We have already shown

- the equivalence of FAs (regular language) and type-3 or regular grammars, and
- the equivalence of PDAs and CFGS.

We now show the equivalence of

- unrestricted grammars and TMs, and
- context-sensitive grammars and LBAs.[Note that we need to introduce the notion of LBAs first to do this]

12.3 EQUIVALENCE OF UNRESTRICTED GRAMMARS AND TMs

We want to show that a language L = L(M) for some TM M iff L = L(G) for some unrestricted grammar G. The following two theorems completes the proof.

Theorem: Let G = (N, Σ, P, S) be an unrestricted grammar. Then the language L(G) generated by G is recursively enumerable.

Proof : To prove the theorem, we construct a 3-type nondeterministic TM M that accepts L(G). Tape 1 always holds any given input string w $\in \Sigma^*$. A production $\alpha \rightarrow \beta$ of G is represented as $\alpha \# \beta$ where # is a special tape symbol of M such that $\# \notin (N \cup \Sigma)$. All the production of G with this representation are written on tape-2 of M. Two productions are separated by the string ##. The idea is that M's computation simulates derivations of G. Tape 3 is used to simulate the derivative of G. On many input string w, the computation of TM = M consists of the following steps:

1. *w* is written on tape 1.
2. *S* is written on the first cell of tape 3.
3. A production $\alpha \# \beta$ is chosen from tape 2 (we assume that all the productions of *G* are written on tape 2)
4. *M* searches for an instance of the string α on tape 3. If found , then it goes to next step; otherwise the computation halts and *M* rejects *w*.
5. The string α on tape 3 is replaced by the string β (in the RHS of the production $\alpha \rightarrow \beta$). [This step minimies one step in the derivation of *w* in *G*.]
6. The string of tape 3 is compared with that on tape 1 (i.e. with the input *w*). If there is a match , the computation halts in an accepting state (i.e. M accpets w).
7. Repeat step 3 through 7 , to apply other productions.

Note : In step 5, if tape 3 contains $\gamma\alpha\delta$ and α is replaced by β, then it says $\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$. Since α and β may be of different length, the symbols of δ may have to be shifted to fit β between γ and δ.

Let $w \in L(G)$. Then $S \overset{*}{\Rightarrow}_G w$. This derivation will eventually be discovered by one of the nondeterministic computations of M by using the steps given above. Hence, $w \in L(G)$

Conversely, let w$\in$ L(G). Then there is an accepting computation of M for the string w. The actions of M on tape 3 are precisely the strings derivable from S and the only string accepted by M are terminal string in L(G). Hence w $\in$L(G) giving L(M)=L(G). That is, M accepts exactly L(G) and hence L(G) is recursively enumerable.

Theorem : Let L be a recursively enumerable language. Then L = L(G) for some unrestricted grammar G.

Proof : Since L is r.e, it is ampled by a deterministic TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ we want to construct an unrestricted grammar $G = (N, \Sigma, P, S)$ whose derivations simulates the computations of M, such that $L(M) = L(G)$. That is, for any string $w \in \Sigma^*$ $S \overset{*}{\Rightarrow}_G w$ iff symbol for some $\alpha, \beta \in \Gamma^*$ and $q_f \in F$. For this is to happen we need to represent IDs of TM M by strings of terminals & nonterminals in G and must have productions in such that S $\overset{*}{\Rightarrow}_G$ q0w $\overset{*}{\Rightarrow}_G \alpha q_f \beta \overset{*}{\Rightarrow}_G$ w.

That is,

1. The initial ID q_0w must be derivable from *S*.
2. Induction production in *G* to simulate every move of *M*.
3. If *M* eventually enters a final state, then transform the string $\alpha q_f \beta$ to *w*.

Since the string w gets modified during simulation (in step 2), the grammar G has to remember it, so that it can reproduced once M enters a final state. So, G is constructed such that it generates two copies of a representation of some string w$\in \Sigma^*$ and then simulates the behaviour of the TM M on one copy, preserving the other. If M accepts, by entering a final state, then G transforms the second copy to a terminal string; otherwise G doesnot transform the second copy to a terminal string.

Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$, for some k >=1
Construction of G is given below.

- $N = \left(\left(\Sigma \cup \{\in\}\right) \times \Gamma\right) \cup \{S, T, R\}$
- The production in *P* are
    1. $S \rightarrow q_0 T$
    2. $T \rightarrow [a_i, a_i] T$ for all $i = 1, 2, \ldots, k$
    3. $T \rightarrow R$
    4. $R \rightarrow [\in, B]\, R$
    5. $R \rightarrow \in$
- For every move $\delta(q, X) = (P, Y, R)$ of the TM *M*, $q[a, X] \rightarrow [a, Y]\, p$ for all $a \in \Sigma \cup \{\in\}$ and all $q \in Q$ and $X, Y \in \Gamma$.

- For every move $\delta(q, X) = (P, Y, L)$ of the TM M, $[b, Z]\, q\, [a, X] \rightarrow p[b, Z]\, q\, [a, Y]$ for all $a, b \in \Sigma \cup \{\in\}$, all $X, Y, Z \in \Gamma$ and $q \in Q$
- For all $q \in F$, all $a \in \Sigma \cup \{\in\}$ and $X \in \Gamma$
  1. $[a, X]\, q \rightarrow qaq$
  2. $q\, [a, X] \rightarrow qaq.$
  3. $q \rightarrow \in$

We now see that a representation of the initial ID $q_0 w$ of M for a string $w = a_1 a_2 \cdots a_n \in \Sigma^*$ can be derived from S using the two rules 1 and 2 i.e.

$$S \overset{1}{\underset{G}{\Rightarrow}} q_0 T$$

$$\overset{1}{\underset{G}{\Rightarrow}} q_0 [a_1, a_1] T$$

$$\overset{*}{\Rightarrow} q_0 [a_1, a_1][a_2, a_2] \cdots [a_n, a_n] T$$

Assume that M accpets w and it doesnot use more than i calls (i>=0) to the right of w. Then using rule 3 once and rule 4 i-times, and finally rule 5 once, G derives the following string

$$S \overset{*}{\underset{G}{\Rightarrow}} q_0 [a_1, a_1][a_2, a_2] \cdots [a_n, a_n] T \overset{1}{\underset{G}{\Rightarrow}} q_0 [a_1, a_1][a_2, a_2] \cdots [a_n, a_n] R$$ (Using rule 3)

$$\overset{*}{\underset{G}{\Rightarrow}} q_0 [a_1, a_1][a_2, a_2] \cdots [a_n, a_n] \underbrace{[\in, B] \cdots [\in, B]}_{i \text{ numbers}} R$$ (Using rule 4 i-times)

$$\overset{1}{\underset{G}{\Rightarrow}} q_0 [a_1, a_1][a_2, a_2] \cdots [a_n, a_n] \underbrace{[\in, B] \cdots [\in, B]}_{i \text{ numbers}}$$ (using rule 5)

This is the representation of the string

$$a_1 a_2 \cdots a_n \underbrace{BB \cdots B}_{i\text{-times}}$$

For any further derivation from this point, we can use only rules 6 and 7 until we encounter a final state.

Let $|\alpha|$ be the representative of the string α in G. Consider the ID α in G. Consider the ID $\alpha a_i q a_j \beta$ of M for $a_i, a_j \in \Sigma$.

If δ (q, aj)=(P, X, R) is a move of M, then using rule b, we find that

$$[\alpha][a_i, a_i] q \left[a_j, a_j\right][\beta] \overset{1}{\underset{G}{\Rightarrow}} [\alpha][a_i, a_i] \left[a_j, Y\right] p[\beta]$$

which is a correct representation of the next ID $\alpha a_i X p \beta$ and it remembers the symbol a_j in the first component of the nonterminal and modifies it to X in the second component.

If on the other hand, $\delta(q, a_j) = (P, Y, L)$ is move of M, then similarly, it is easy to see that using rule 7 we find a correct representation of the next ID $\alpha p a_i Y \beta$ of M.

Hence, at every step, using rule 6 and 7, the grammar G correctly simulates the computations of M.

If $w \in L(M)$, then M eventually enters a final state. At this point, the derivation in G can use rule 8 to reproduce the original string w from the first component of the representation of

every nonterminal in the resulting string. All the q's can be erased by using $q \to \in$, as many times as required. Therefore $S \overset{*}{\underset{G}{\Rightarrow}} w$ and so $w \in L(G)$.
Conversely, if $w \in L(G)$ there is a derivation of w in G. Proceeding in exactly in opposite direction as discussed above, we discover that for some $\alpha, \beta \in \Gamma$ and $q_f \in F$. Hence $w \in L(M)$, completing the proof.

12.4 CONTEXT-SENSITIVE LANGUAGE AND LBAs

We first introduce the notion of LBAs and then show the equivalence of CSLs and LBAs.

- TM is the most general and powerful computational model developed so far.
- It is interesting to observe that though a large number of variations of TMs exists, all are equivalent to the basic TM model in terms of power or capability i.e. all can accept r.e language only. This implies that it is not possible to increase the power of a TM by putting more features in terms of complex and /or additional structures to the basic model.
- But by putting some kind of restrictions on the use of the structures of the TM, it is possible to limit the power. For example,
    - If only a finite amount of tape is allowed to use with read-only tape that can move only to right one call at a time, we get a FA accepting regular language.
    - If the tape is restricted to be used as stack, it will work like a nondeterministic pushdown automata.
- Similarly, we get another interesting type of automata by restricting the number of tape cells that can be used.
- This new automata, denoted "linear bounded automata" (or LBA), accepts a smaller class of languages than the class of r.e. languages. An LBA is exactly similar to a TM except that on any input $w \in \Sigma^*$ with $|w| = n$, it can use only $(n+2)$ numbers of cells of the input tape.The input string is always put between a left-end marker, <, and a right-end marker, >, which are not puts of the input string. The read-write head cannot move to the left of the left-end marker or to the right of the right-end marker. The two end markers cannot be overwritten in any case.

Formally, a LBA is a nondeterministic TM M = $(Q, \Sigma, \Gamma, \delta, q0, B, <, >, F)$ satisfying the following conditions:

1. The input alphabet, Σ must contain two special symbols < and >, the left and right end markers, respectively which do not appear in any input string.
2. $\delta(q, <)$ can contain only element of the form $(p, <, R)$ and $\delta(q, >)$ can contain only elements of the form $(p, >, L)$ for any $q, p \in Q$.

[Note: All other elements are identical to the corresponding elements of a TM]

The language accepted by M, denoted by L(M) is

L(M) = { $w \mid w \in (\Sigma, \{<, >\})^*$ and $\alpha, \beta \in \Gamma^*$ } for some $\alpha, \beta \in \Gamma^*$ and $q_f \in F$ }

The blank symbol, B is not necessary to be considered as a part of M since it cannot move to the right of right - end marker.
The reason behind using the name "linear bounded automation" is derived from the following fact:

If on every input w with |w| = n, a TM is allowed to use only an amount of tape that is "bounded by some linear function" of n, then the computational power of this TM would be identical to the TM which is restricted to use the amount of tape containing n+2 cells (as given in the definition).

Example : The language $L = \{a^n b^n c^n \mid n \geq 1\}$ is accepted by some LBA.

To show that L is accepted by an LBA. we need to construct a TM to accept L such that during computation on any input w, the read-write head moves neither beyond the right of the rightmost symbol of w nor beyond the left of the leftmost symbol of w. The outline of the TM M accepting L is given below.

On initial state q0, M replaces the first a by X and change state to q1 and the head moves to the right looking for the first b, skipping all other symbols.

This b is then replaced by Y and changes state to q2 and the head moves to the right searching for the first c skipping all other symbols. This c is then replaced by Z and changes state to q3 and the head moves to the left searching for the first X, skipping all other symbols. On reading X in state q3 the head moves to the right (one cell) changing state to q0 again to repeat the same process i.e match each a, b and c and replace them by X, Y and Z, respectively, with the same sequence of state changes.

During this process, if it reads Y (instead of the symbol a) in state q0, then it implies that all a's have been replaced by X's and hence it needs to check that all b's and c's have also been replaced by Y's and Z's, respectively.

This can be done by entering a state, say q4 and moving the head to right looking for any b's or c's left until the right end is discovered (by reading a blank symbol). If not found, the input is accepted; otherwise it is rejected.

It is observed that at no point the read-write head moves past the extreme left and right symbols, except in the last step when it reads the first blank symbol to the right of w.

This TM can be converted to a LBA by including the two end marks and keeping all the moves except the last one. In the last step when M reads a blank symbol, the LBA will read the right endmarker, > and hence a move of the form $\delta(q_4, >) = (q_4, >, L)$ can be included, where , $q_5 \in F$ to save the same purpose. This LBA also accepts the same language L as that of M.

CHECK YOUR PROGRESS

True/False type questions

1) Letters, digits, single characters are known as strings_____

2) LBA is accepted by Regular language_____

3) Type 3 is regular language_____

4) Smallest unit of a grammar that appears in production rules, cannot be further broken down is known as terminal._____

5) Turing machine is most powerful language._____

Answers-

1) True
2) False
3) True
4) True
5) True

12.5 EQUIVALENCE OF LINEAR-BOUNDED AUTOMATA AND CONTEXT-SENSITIVE GRAMMARS

we now show that LBA's and CSG's are equivalent in the sense that the LBA's accept exactly the CSLs except for the fact that an LBA can accept $\in$ while a CSG cannot generate $\in$, that is, L = L (M) for some CSG G. The result is shown by proving the following two theorems.

Theorem : If L is a context-sensitive language, then L is accpeted by one LBA M.

Proof : Since L is a CSL, L = L(G) for some CSG G =(N, Σ, P, S). We now construct an LBA M with a two-track tape to simulate the derivatives of G. The first track holds the input string (including the end markers) while the second track holds the sentential form generated by the simulated derivation. On input <w> on its tape a computation of the LBA M consists of the following sequence of steps.

1. The LBA writes the (start) symbol *S* of *G* on the second track below the leftmost symbol of *w*.
2. If *w* = $\in$ the LBA halts without accepting.
3. The LBA nondeterministically selects a production $\alpha \rightarrow \beta$ and a position in the sentential form written on the second track.
4. It follows next three steps

    1. if a substring on track 2 starting at the selected position doesnot match , α the LBA halts in a rejecting state.

2. If the substring on track 2 starting at the selected position is α but the string obtained by replacing α by β (i.e. applying the rule $\alpha \rightarrow \beta$) has a length greater than $|w|$, then the LBA halts in rejecting state.
3. otherwise, α is replaced by β on track 2.

5. If track 2 contains the string w, then the LBA halts in an accepting state, otherwise, steps 3 through 5 are repeated.

Thus, the LBA M will accpet a string w if $S \overset{*}{\underset{G}{\Rightarrow}} w$. Conversely, a computation of the LBA M with input <w> that falls in an accepting state consists of a sequence of string transformations generated by steps 3 and 4. But these transformations define a deviation of w in G. Thus, the LBA M accepts w iff $S \overset{*}{\underset{G}{\Rightarrow}} w$.

Theorem : Let L be a language accepeted by an LBA M = $(Q, \Sigma_M, \Gamma, \delta, q_0, <, >, F)$. Then L-{ $\in$ } is a context-sensitive language.

Proof : We need to construct an equivalent CSG G that simulates the computation of the LBA M. Note that the techniques used to construct an equivalent unrestricted grammar that simulates the computations of a TM (as given in theorem ....) cannot be adopted directly. The reason is that if the CGS simulated the LBA using distinct symbols for the states and the endmarkers, then it could never erase these symbols later to produce the original input string since it would violate the noncontracting or monotonicity property of a CSG. Because use of a production in a derivation to erase a symbol in a sentential form would produce a starter sentential form. Hence the endmarkers must be incorporated into adjustment tape symbols and similarly the states must be incorporated into the symbols scanned by the tape head. The input alphabet of G is obtained from Σ_M by removing the endmarkers. Nonterminals of G are ordered pairs-the first component is a terminal symbol and the second component is a string consisting of a combination of a tape symbol and (possibly) a state and endmarkers. The construction of the CSG G = (N, Σ_G, P, S) is as follows.

$$\Sigma_G = \Sigma_M - \{<, >\}$$

N = { S, A, [a, X], [a,<X], [a, X>], [a, <X>], [a, qX], [a, q<X], [a, <qX], [a, qX>], [a, Xq >], [a, Xq>], [a, qX>], [a, Xq>], [a, q<X>], [a, <qx>],[a, <Xq>] }

for all $a \in \Sigma_G, X \in \Gamma$ and $q \in Q$

The production in P are given below.

1. $S \rightarrow [a, q_0 < a] A | [a, q_0 < a >] \forall a \in \Sigma_G$
2. $A \rightarrow [a, a] A | [a, a >] \forall a \in \Sigma_G$

Using these two rules we get

S $\overset{*}{\underset{G}{\Rightarrow}}$ [a, q0<a>] or

$$S \overset{*}{\underset{G}{\Rightarrow}} [a, q_0 < a_1][a_2, a_2][a_3, a_3] \cdots [a_m, a_m >]$$

The string that is obtained by concatenating the elements of the first component(s) of the ordered pairs (composite variable(s)) is $a_1 a_2 \cdots a_m$ and represents the input string to the LBA M. Concatenating the second component we get the string $q_0 \langle a_1 a_2 \cdots a_m \rangle$ which is the initial configuration of the LBA M on the input string $a_1 a_2 \cdots a_m$.

Rules 3 and 4 given below are used to simulate the computations of the LBA M.

3. For every move $\delta(q, X) = (p, Y, R)$ of the LBA include $[a_i, q_x]\left[a_j, a_j\right] \rightarrow [a_i, Y]\left[a_j, p a_j\right]$ in *P*
   $\forall a_i, a_j \in \Sigma \cup (\in) \quad X, Y \in \Gamma$ and $p, q \in Q$
4. Similarly for every move $\delta(q, X) = (p, Y, L)$ of the LBA include $[a_i, a_i]\left[a_j, qX\right] \rightarrow [a_i, p a_i]\left[a_j, Y\right]$ in *P*.
5. $[a, q_0 < a] \rightarrow [a, < q_0 a]$ whenever
   $\delta(q_0, <) = (p, <, R)$ is a move of the LBA.
6. $[a, Xq >] \rightarrow [a, pX >]$ whenever
   $\delta(q, >) = (p, > L)$ is a move of the LBA

   The two rules 5 and 6 are used to handle the two special extreme cases as indicate in the definition of the lba.

   Hence every move of the LBA can be simulated by G using the above rules. It is clear that if the LBA ever enters a final state qf , then simulating this step the CSG G will produce a variable $[a, \alpha q \beta]$ in the sentential form. At this point, the derivation must generate the original input string.

   Using the production

7. $[a, \alpha q \beta] \rightarrow a$, $\forall a \in \Sigma_G$ and $\alpha, \beta \in \Sigma^+$, (i.e. α and/or β could include <, > and onre tape symbol).

   The ordered pair $[a, \alpha q \beta]$ is transformed into the terminal symbol contained in the first component.

Now the following rules allow deletion of the second component of an ordered pair (i.e. composite variable) if it is adjacent to a terminal symbol.

8. $[a,\alpha]b \rightarrow ab$

9. $b[a,\alpha] \rightarrow ba \quad \forall a,b \in \Sigma_G$ and all possible $\alpha \in \Sigma^+$

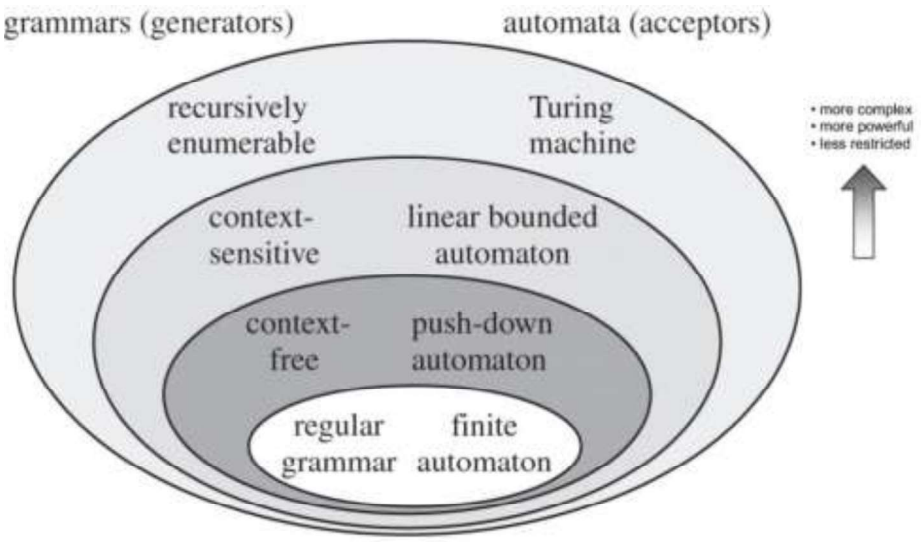
Once, all the second component s are deleted using the above two rules, the original input string is correctly generated. This is the correct derivation the LBA would accept the same string as it had entered one of the final states (implied by use of rule 7).

It is also clear that the CSG G can generate a terminal string only if the LBA accepts it.

Note that the production used here are all context-sensitive. Also, the second components donot produce the string $q_0 \langle\rangle$. Thus the computation with the empty string as input is not simulated by the CSG.

A proof that any string $w \in \Sigma^+$ is accepted by the LBA M iff it is generated by the grammar G is exactly similar to one that was produced in Theorems.

- What are the different levels in the Chomsky hierarchy?



Chomsky Hierarchy Levels. Source: Fitch. 2014.

There are 4 levels – Type-3, Type-2, Type-1, Type-0. With every level, the grammar becomes less restrictive in rules, but more complicated to automate. Every level is also a subset of the subsequent level.

- **Type-3: Regular Grammar** - most restrictive of the set, they generate regular languages. They 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.

- **Type-2: Context-Free Grammar** - generate context-free languages, a category of immense interest to NLP practitioners. Here all rules take the form $A \rightarrow \beta$, where A is a single non-terminal symbol and β is a string of symbols.
- **Type-1: Context-Sensitive Grammar** - the highest programmable level, they generate context-sensitive languages. They have rules of the form $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$ with A as a non-terminal and α, β, γ as strings of terminals and non-terminals. Strings α, β may be empty, but γ must be nonempty.
- **Type-0: Recursively enumerable grammar** - are too generic and unrestricted to describe the syntax of either programming or natural languages.

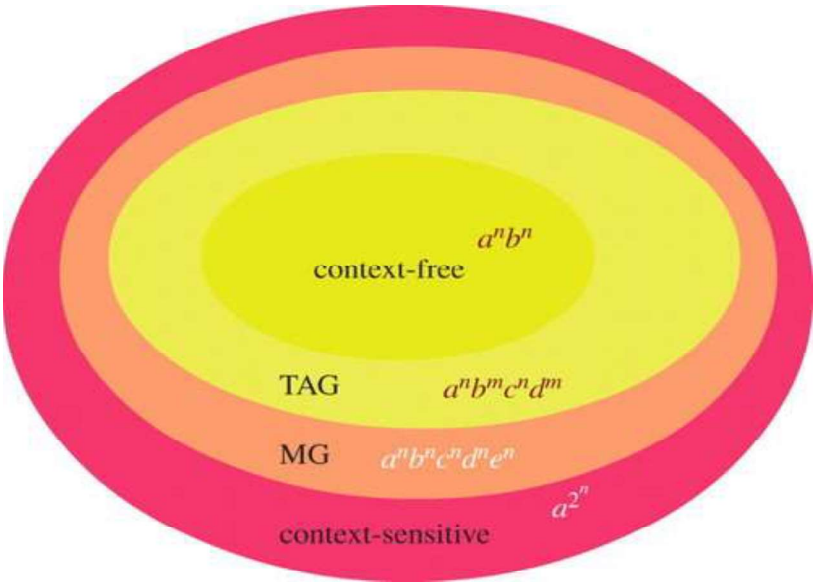
Any language is a structured medium of communication whether it is a spoken or written natural language, sign or coded language, or a formal programming language. Languages are characterised by two basic elements – syntax (grammatical rules) and semantics (meaning). In some languages, the meaning might vary depending upon a third factor called context of usage.

Depending on restrictions and complexity present in the grammar, languages find a place in the hierarchy of formal languages. **Noam Chomsky**, celebrated American linguist cum cognitive scientist, defined this hierarchy in 1956 and hence it's called **Chomsky Hierarchy**.

Although his concept is quite old, there's renewed interest because of its relevance to Natural Language Processing. Chomsky hierarchy helps us answer questions like "Can a natural language like English be described ('parsed', 'compiled') with the same methods as used for formal/artificial (programming) languages in computer science?"

- What are the common terms and definitions used while studying Chomsky Hierarchy?
  - **Symbol** - Letters, digits, single characters. Example - A,b,3
  - **String** - Finite sequence of symbols. Example - Abcd, x12
  - **Production Rules** - Set of rules for every grammar describing how to form strings from the language that are syntactically valid.
  - **Terminal** - Smallest unit of a grammar that appears in production rules, cannot be further broken down.
  - **Non-terminal** - Symbols that can be replaced by other non-terminals or terminals by successive application of production rules.
  - **Grammar** - Rules for forming well-structured sentences and the words that make up those sentences in a language. A 4-tuple **G = (V , T , P , S)** such that V = Finite non-empty set of non-terminal symbols, T = Finite set of terminal symbols, P = Finite non-empty set of production rules, S = Start symbol
  - **Language** - Set of strings conforming to a grammar. Programming languages have finite strings; most natural languages are seemingly infinite. Example – Spanish, Python, Hexadecimal code.
  - **Automaton** - Programmable version of a grammar governed by pre-defined production rules. It has clearly set computing requirements of memory and processing. Example – Regular automaton for regex.

- What are the important extensions to Chomsky hierarchy that find relevance in NLP?



Mildly Context Sensitive Languages. Source: Jäger and Rogers. 2012.

There are two extensions to the traditional Chomsky hierarchy that have proved useful in linguistics and cognitive science:

- **Mildly context-sensitive languages** - CFGs are not adequate (weakly or strongly) to characterize some aspects of language structure. To derive extra power beyond CFG, a grammatical formalism called Tree Adjoining Grammars (TAG) was proposed as an approximate characterization of Mildly Context-Sensitive Grammars. It is a tree generating system that factors recursion and the domain of dependencies in a novel way leading to 'localization' of dependencies, their long distance behaviour following from the operation of composition, called 'adjoining'. Another classification called Minimalist Grammars (MG) describes an even larger class of formal languages.
- **Sub-regular languages** - A sub-regular language is a set of strings that can be described without employing the full power of finite state automata. Many aspects of human language are manifestly sub-regular, such as some 'strictly local' dependencies. Example – identifying recurring sub-string patterns within words is one such common application.

12.6 CONCLUSION

This module explains about the basic understanding of Chomsky Hierarchy. It explains Equivalence of Unrestricted grammars, Turing Machines (TMs), Context-Sensitive Language and Linear-bounded Automata (LBAs). It also discusses the Equivalence of LBAs and Context-sensitive Grammars through various theorems, lemmas and step-wise elaborated solved examples.

12.7 CHECK YOUR PROGRESS

1) Type 3 is _____grammer.

2) Linear bound automata is accepted by_____

3) Smallest unit of a grammar that appears in production rules,is known as_____

4) Set of rules for every grammar describing how to form strings from the language that are syntactically valid is known as _____

5) Recursive enumerable grammer is accepted by_____

12.8 ANSWER CHECK YOUR PROGRESS

Fill in the blanks:

1) Regular
2) Context sensitive language
3) Terminal
4) Production rules
5) Turing machine

12.9 MODEL QUESTION

Qs-1) What is grammer explain? What are its benefits? Expalin.
Qs-2) Explain Chomsky Hierarchy with the help of diagram?
Qs-3) What is sub-regular language?
Qs-4) Explain type1 Grammer?
Qs-5) What is midly context sensitive language? Explain.

12.10 REFERENCES

1. Devopedia. 2021. "Chomsky Hierarchy." Version 9, June 28. Accessed 2021-06-28. https://devopedia.org/chomsky-hierarchy.
2. https://nptel.ac.in/courses/106/103/106103070/

12.11 SUGGESTED READINGS

3. Hopcroft, John and Jeffery Ullman. 1987. "Introduction to Automata theory, languages and computation." Indian Student Edition:Narosa Publishing House.
4. Jäger, Gerhard and James Rogers. 2012. "Formal language theory: refining the Chomsky hierarchy." Philos Trans R Soc Lond B Biol Sci., vol. 367, no. 1598, pp. 1956–1970, July 19. Accessed 2019-08-2019.

5. Martin J. C., “Introduction to Languages and Theory of Computations”, TMH
6. Papadimitrou, C. and Lewis, C.L., “Elements of theory of Computations”, PHI
7. Cohen D. I. A., “Introduction to Computer theory”, John Wiley & Sons
8. Kumar Rajendra, “Theory of Automata (Languages and Computation)”, PPM
9. Roberts, Eric. 2004. "Basics of Automata Theory." Automata Theory, Stanford University, September. Accessed 2019-10-16.