BLOCK III:
AUTOMATA THEORY

UNIT 1: INTRODUCTION TO LANGUAGES AND GRAMMAR

Unit Structure:

Space for learners:

1.1 INTRODUCTION

To write instructions for machines it is important to learn syntax of the language and to designed computing machines, automata theory is important. For formalizing the notion of a language, we must include all the varieties of languages such as natural language produced by human being and languages for computer. Automata theory is a theoretical branch of Computer Science and Mathematics. It primarily deals with the logic of computation by some simple machine.

The word "automata" is a plural form of the word "automaton". The meaning of the word "automaton" is mechanization i.e., the condition of being automatically operated or controlled. Automating a process means performing it in a machine without the intervention of human. To perform a particular task in a mechanical environment, inputs, energy and control signal are required so thatit can produce the output without the direct involvement of human. Worked performed by machines are more accurate and efficient and it takes less time.

In the context of computer science, an automaton is a machine that can perform the computation in a mechanized manner. An automaton with a finite number of states is called a *finite automaton*. It is very important that the computing machine understands the instructions given by human. And it is necessary to develop languages for writing these instructions so that the machine can understand unambiguously.

This unit is an attempt to give the concept of languages and grammar in the context of computer science.

Space for learners:

1.2 UNIT OBJECTIVES

After going through this unit, you will be able to:

- define alphabet, string, substring empty string, concatenation, Kleene closure etc.
- learn about variables, terminals, productions rules etc.
- define grammar and language in the context of theory of computer science
- determine language generated by a grammar

- learn about Chomsky classification of grammar

Space for learners:

1.3 INTRODUCTION TO FORMAL LANGUAGES

We begin our discussion with the concept of *set*. A set is a collection of elements or objects. For example, a set of three elements a, b and c can be written as S= {a, b, c}. It can be written in any order like S = {b, a, c}.

But a sequence is an ordered collection of elements. A sequence {a, b} is not same as {b, a}.An ordered collection is one in which the arrangement of objects matters.

Characters and their orders are important components in the formation of any language. For instance, the word "cat" does not carry the same meaning "act" though the letters or the elements are same. The most vital component of any language is its character set. In case of language such as English, any word can be formed from the English alphabet set S = {A, B, C, D,……,Z, a, b, c, d,…….,z}. Whereas a sentence is a combination of sequence of symbols from the Roman alphabet along with punctuation marks such as *comma*, *full-stop*, *colon* and *blank-space* which is used to separate two words.

Suppose we want to form words consisting of 5 letters from the set of English alphabets, say S. Then S^5 will be the all possible sequence of word of length 5. Thus, S^n represents the set of all possible n letter sequences. A word becomes valid when it carries some meaning. For example, the word "*cake"* has meaning but the reverse "*ekac"* has no meaning. Whereas, the word "*cat"* and its reverse "*act*" carries definite meaning in English language. The one who understands a language L can able to differentiate the meaningful and meaningless word of that particular language.

The same is true in the context of computer language. For example, in case of C programming language, we write codes for program using the character sets of the C language. Each and every component of statements of the programming language is formed by combining the characters from the character sets so that the compiler can understand and compile the statements of the program.

We can define a language is a set of valid words over its character set. If we denote the set of alphabets or character set by the symbol $\sum$, then $\sum^*$ represents the set of all possible words or strings that can

be constructed using the characters in ∑.It is therefore observed that formal learning of a language includes the following:

- Learning the alphabet
- Words which are formed by various sequence of symbols of its alphabet.
- Sentence formation; Sentences are formed by combining sequence of various words following some rules.

Formal language is designed for use in which natural language is unsuitable. If we want to instruct an abstract machine, we have to use something much more precise. An abstract machine takes instruction given by humans and provides the desired output. So, it is necessary to develop languages for these instructions. The languages that are developed with precise *syntax* and *semantics* are called ***formal languag***e. Syntaxes are precise rules that tell us the symbols we are allowed to use and how to put them together into legal expressions. And semantics tell us the meanings of the symbols and legal expressions. Formal languages play an important role in the development of compilers.

Let us now discuss the basic terminologies which are important and frequently used in formal languages and automata theory.

Space for learners:

1.3.1 Basic Terminologies

Symbols:
Symbol can be any alphabet, letter or any element which is considered as the smallest building block of a language. Symbol can be considered as the atom of a language. For example, a, b, c, x, y, 0, 1, #, ε, etc. are symbols.

Alphabet:

Dictionary meaning of alphabet is a finite set of characters that include letters and is used to write a language. Mathematically we can define alphabet as a finite, non-empty set of symbols. Alphabet of a language is generally denoted by the summation "∑" symbol.

For example,

Binary alphabet consists of 0 and 1 only. It is represented as ∑= {0, 1}

Roman alphabet is denoted by ∑= {a,b,…..,z}.

∑= {a, b, c,ε} is an alphabet.

∑={α, β, γ} is an alphabet.

But, A = {1,2,3, ...} is not an alphabet because it is infinite.

String:

A "string" over an alphabet ∑ is a finite sequence of symbols from that alphabet∑ which is written next to one another and not separated by commas. A string is also known as a *word*. For example,

i) If ∑ ={a,b} be an alphabet; then ab, ba, aa, bb, aba, bab, bbaab, aababa,…are some examples of strings over ∑.
ii) If ∑={0,1} then 110010, 001, 10, 01, 0001, 111101, etc. are some stringsover ∑.
iii) If ∑= {a, b, c, d,….,z} the any combination of symbols *abp, bac, pqre, bcatdpr,* etc.are some strings over ∑.
iv) The set of all strings over an alphabet ∑ is denoted by $\sum^*$. For example, if ∑={0,1} then$\sum^*$= {ε, 0, 1, 00, 11, 01, 10, 111, 000, 010, 101, ... ….}

Length of a string:

Length of a stringwis defined as the number of symbols or element present in the string. For example, length of 110010 is 6. Length of the string *bcatdpr* is 7. Length of a string w is represented within two vertical bars "|w|" as follows:

|00111010| = 8
|101010101110| = 12
|abba|=4
|a|=1
ε is the empty string and has length zero.

Empty String:

The string that has no element i.e., of zero length is called the "empty string". Empty string is denoted by the symbol ε (epsilon). Length of empty string is |ε| =0.

Space for learners:

Reversing a string:
Writing a string in reverse order is known as reversing a string. If w= $w_1w_2w_3\ldots..w_n$ where $w_i \in \sum$, the reverse of the string w is $w_n w_{n-1}w_{n-2}\ldots.w_1$

String concatenation:
When we write one string appending the other string at end,then it is known as *string concatenation*. It is one of the most fundamental operations used for string manipulation.

Let $x = a_1a_2a_3\ldots..a_n$ and $y = b_1b_2b_3\ldots\ldots.b_m$ be two strings of length n and m, then the concatenation of the two strings *x* and *y* is written as *xy,* which is the string obtained by appending y to the end of x. The concatenated string *xy* is $xy = a_1a_2a_3\ldots..a_n\ b_1b_2b_3\ldots\ldots.b_m$

The empty string ε satisfies the property $\varepsilon w = xw = w$ where w is a string.

Substring:
We say that *x* is a *substring* of w if *x* occurs in w, that is w= *uxv* for some strings *u* and *v*. For example, "***put***" is a substring of the string *com**put**er*.

Suffix and Prefix:
If w = xv for some x, then v is a suffix of w. Similarly, if w = ux for some x, then u is the prefix of w.
Again, for w = uxv, the substring x will be the prefix of w if u= ε and x will be the suffix of w if v= ε.

Languages:
We have already been acquainted with the concept of strings. Any set of strings over an alphabet $\sum$ is called a *language*. Language can be finite or infinite. We usually denote a language by the letter L. As $\sum^*$ represents the set of all strings, including the empty string ε over the alphabet $\sum$, we can define a language L over an alphabet $\sum$ as a subset of $\sum^*$. Thus

$L = \{w \in \sum^*$: w has some property P$\}$

Some other examples of language are as follows:

i) $L = \{w \in \{a, b\}^*$: w has an equal number of a's and b's$\}$
ii) $L = \{w \in \sum^*$: w = $w^R\}$ where w^R reverse string of w.

Space for learners:

iii) The set of all strings over {0,1} that start with 0.
iv) L= { ε , 0, 00, 000, ……, } is a language over alphabet { 0}.
v) L= $\{0^n1^n2^n : n \geq 1\}$ is a language.
vi) The set of all strings over {a, b, c} having ab as a substring.
vii) The set of empty string {ε } is also a language over any alphabet.
viii) The empty set $\emptyset$ is a language over any alphabet. {ε }

It should be noted that $\emptyset \neq \{\varepsilon\}$. Because the language does not contain any string but {ε } contains a string ε.Also, length of |{ε }| =1 but $|\emptyset| = 0$

Concatenation of languages:
If L_1 and L_2 are languages over some alphabet $\sum$, their concatenation can be denoted by $L=L_1.L_2$ or, $L = L_1L_2$ where

L={w$\in \sum^*$: w = x . y for some x $L = \{w \in \sum^* : w = x.y$ for some $x \in L_1, y \in L_2\}$.

For example,
i) If L_1={0,1} and L_2= {1,00}, then L_1L_2 = {01,000,11,100}
ii) If L_1={0,1,2} and L_2= {1,00}, then L_1L_2 = {01,000,11,100, 21,200}
iii) If L_1={a, ab, abab} and L_2= {pq, ppqq, pppqqq}, then L_1L_2={apq, appqq, apppqqq, abpq, abppqq, abpppqqq, ababpq, ababppqq, ababpppqqq}
iv) If L_1={b,ba,bab } and L_2= { ε ,b,bb}, then L_1L_2 = {b, bb, bbb, ba, bab, babb, babbb}

Since string concatenation supports the *associative* property, so the concatenation of languages is also *associative*. Thus, if L_1, L_2 and L_3 are three languages then,

$$(L_1L_2)L_3 = L_1(L_2L_3)$$

It is to be noted that $L_1L_2 \neq L_2L_1$

Space for learners:

Kleene Closure:

In terms of formal languages, another important operation is *Kleene closure* or *Kleenestar*. *Kleene closure* of a language L is denoted by L^*.

L^* can be define as

L^*= {w ∈ ∑*: w = $w_1w_2w_3$……w_n, for some n ≥ 0 and some $w_1,w_2,w_3,….,w_n$ ∈ L. It can also be defined as follows:

$$L^* = \bigcup_{n\geq 0} L^n$$

L^*= {Set of all strings over ∑}

Examples:

i) If ∑ = {a,b} and a language over L over ∑, then

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots\ldots\ldots$$

$L^0 = \{\varepsilon\}$

$L^1 = \{a, b\}$

$L^2 = \{aa, ab, ba, bb\}$ and so on.

So, $L^* = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\ldots\}$

ii) If ∑ = {0} and a language over L over ∑, then

$$L^* = \{\varepsilon, 0,00,000,0000, \ldots\ldots\ldots\ldots\}$$

Positive Closure:

If ∑ is an alphabet then positive closure of the language L denoted by L^+ is the set of all strings over ∑ excluding the empty string ε.

$$L^+ = L^* - \{ \varepsilon \}$$

For example, if ∑ = {0}, then

$$L^+ = \{0,00,000,0000, \ldots\ldots\ldots\ldots\}\sum$$

The positive closure of a language L is

$$L^+ = \bigcup_{n\geq 1} L^n$$

Space for learners:

CHECK YOUR PROGRESS-I

1. Given a string 011 over ∑= {0,1}. Find all the substring of the string.
2. For the binary alphabet { 0, 1 }, find $\sum^2$ and $\sum^3$.
3. If L = { a, ab }, find L^* and L^+.

Space for learners:

1.4 REGULAR LANGUAGES AND REGULAR EXPRESSIONS

In this section we are going to introduce the concept of regular languages and regular expressions. In mathematics, we can use operations like + and × to represent expression such as

(2+4)×5

The value of the above arithmetic expression is number 30.

Similarly, we can use regular operations to build up expressions describing languages, which are called regular expressions. The value of a regular expression is a language. As an example,

$(0+1)^*11$

In this case the value is the language L over {0,1} such that every string in L ends with two consecutive one.

We can define a ***regular expression*** over an alphabet ∑ recursively as follows:

- Every character or alphabet belonging to ∑ is a regular expression.
- ∅, empty string ε, and a, for each $a \in \sum$, are regular expressions representing the languages $\emptyset, \{\epsilon\}$ and {a}, respectively.
- If r and s are regular expressions representing the language R and S respectively, then *concatenation* of these represented as *rs* is also a regular expression.
- If r and s are regular expressions representing the language R and S respectively, then the union of these represented as $r \cup s$ or $r+s$ is also a regular expression.
- The Kleene closure r^* is a regular expression representing the language R^*.

A class of languages can be generated by applying operations like union, concatenation, Kleene star etc. on the elements. These

languages are known as **regular languages** and the corresponding finite representations are known as **regular expressions.**

Some regular expressions and their corresponding regular sets are as follows:

Regular expression	**Corresponding regular set**
0	{0}
0+1	{0,1}
a+b+c	{a, b, c}
$(11)^*$	$\{\varepsilon, 11,1111,111111, \ldots.\}$
ab+ba	{ab,ba}
$(a+b)^*c$	{a, ac, acc, accc,…..b, bc, bcc, bccc, …….}
$(abc)^*$	{ε, abc, abcabc,abcabcabc, …….}
$(abc)^*d$	d, abcd,abcabcd,abcabcabcd,……..}
ab^*cd	acd, abcd, abbcd, abbbcd, abbbbcd,…..}
ab(p+q)	{abp, abq}

If r is a regular expression, then the language represented by r is denoted by $L(r)$. Further, a language L is said to be *regular* if there exists a regular expression r such that $L = L(r)$.

1.5 GRAMMAR

It is required to learn the grammar of a language while learning a specific language. For instance, it is required to learn English grammar while learning English language for forming meaning correct sentences. For the formation of sentences in any language, concept of grammar is very necessary. For getting the concept of grammar in the context of computer, let us first take some examples from English grammar. Here, we are considering two types of sentences in English; sentences having a ***noun and a verb*** or those with a ***noun, verb and adverb***.

Noun- verb -adverb	**Noun - verb**
Barun ate quickly.	Barun ate.
Rita walked slowly.	Rita walked.
Neha talks slowly.	Neha sang.
Rishi writes slowly.	Rishi ran.

Space for learners:

We can see that **Noun- verb –adverb** and **Noun- verb** are description of two types of sentences in English grammar. If we replace noun, verb, adverb with some suitable word, we get grammatically correct sentences. In the example, we have seen in the example that sentences are formed by replacing noun with some name like Barun, Rita, Neha, Rishi, verb with ate, walked, talks, writes, etc. and adverb with quickly, slowly, etc.

If we call *noun-verb-adverb* or *Noun-verb* as variables(V), words like Barun, Neha, ate, writes, quickly, slowly, etc. as terminals(T), S be a variable representing a sentence, then following will be the rules (P) for generating two types of sentences:

$$S \rightarrow < noun >< verb >< adverb >$$
$$S \rightarrow < noun >< verb >$$
$$< noun > \rightarrow Barun$$
$$< noun > \rightarrow Rita$$
$$< noun > \rightarrow Neha$$
$$< verb > \rightarrow ate$$
$$< verb > \rightarrow write$$
$$< verb > \rightarrow walked$$
$$< adverb > \rightarrow slowly$$
$$< adverb > \rightarrow quickly$$

Thus, we can describe a grammar by a 4-tuple: Variable (V), Terminals (T), S is a special symbol from V, P is a collection of rules which is termed a productions. The sentences are formed by starting with S, replacing words/strings using the production rules, and terminating when string of terminals is obtained.

A grammar consists of a set of rules (called *productions*) that specify the sequence of characters (or lexical items or sentences) that form allowable programs in the language been defined. Meaningful sentences (or statements) are formed using the grammar of the language. We have learnt that a grammar should have the following components

- A set of non-terminals symbols. These symbols are represented using capital letter like A, B, C, etc.
- A set of terminal symbols. Terminals are generally represented using small case letter like a, b, c etc.

Space for learners:

- A start symbol from the set of non-terminals to represent a sentence from which various sentences of the language can be generated.
- A set of production rules.

Space for learners:

1.5.1 Formal Definition of a Grammar

Noam Chomsky gave a mathematical model of grammar in 1956 which turned out to be useful for writing computer language although it was not useful for describing natural languages. We will briefly discuss the different categories of grammar provided by Noam Chomsky in the next section.

A formal grammar is just a grammar specified using a strictly defined notation. For compiler technology, there are two useful grammars, which are regular grammar and context free grammar. Let us now write the formal definition of a grammar.

A grammar is a quadruple **G=(V, ∑,P,S)**, where

V is a finite set of variables (non-terminals),

$\sum$ is a finite set of terminals. Terminals are denoted by T also.

S is the start symbol, where $S \epsilon V$

P is a finite non-empty set of rules whose elements are $\alpha \rightarrow \beta$, where α, β are strings on $V \cup \sum$.

α has at least one symbol from V. The elements of P are called production rules.

Following points are to be noted while writing and substituting productions:

i) A production rule of a grammar is of the form $A \rightarrow \alpha$ where A is a non-terminal symbol. The production rule $A \rightarrow \alpha$ is same as$(A, \alpha) \epsilon P$. But it is more convenient to write the production as$A \rightarrow \alpha$.

ii) If $S \rightarrow AB$ is a production, then we can replace S by AB, but reverse substitution is not allowed. i.e., we cannot replace AB by S.

iii) $S \rightarrow AB$ is a production but $AB \rightarrow S$ is not.

Examples 1: If G = ({S}, {a}, $\{S \to SS\}, S$), find the language generated by G.

Solution: Here we have V={S}, T={a}, Start symbol S, production rule

P: $S \to SS$

Since we have only one production rule $S \to SS$ in G and it has no terminal on the right-hand side, so we will not get any string from the production. Therefore, the language generated by G is $L(G) = \emptyset$.

Example 2: Consider the Grammar G = (V,T,P, S) where T= {a, b}, P={ $A \to Aa, A \to Ab, A \to a, A \to b, A \to \varepsilon$ }, S={A}. White a common generated by this grammar exacting few strings of the grammar.

Solution:

Here the start symbol is A.

A→ $Aa \to aa$

A→ $Ab \to ba$

A→ $Ab \to Aab \to Aaab \to aaab$

A→ $Ab \to Abb \to Aabb \to babb$

A→ $Ab \to \varepsilon b \to b$

A→ $Aa \to \varepsilon a \to a$

A→ $Aa \to Aaa \to Abaa \to Abbaa \to \varepsilon bbaa$

Hence this grammar can be used to produce the strings of the form $(a+b)^*$

1.6 CHOMSKY CLASSIFICATION OF GRAMMARS

So far, we have seen that a grammar depends on its production rules to derive strings in the associated language. *Noam Chomsky* classified the grammar into four categories which based on their production rules.

Type 3: The first category is known as the *type 3* which is also referred to as *regular grammar*. We have already been acquainted with regular grammar and regular language in our previous section. The production rules for type 3 grammar are of the following forms:

$A \to a$

$A \to aB$

Space for learners:

where A and *B* are some non-terminals and *a* is some terminal in the grammar. Type 3 grammars are recognized by *finite automaton*. In case of type 3 grammar, productions in the left-hand-side consists of a non-terminal only and the productions in the right-hand-side contains either a single terminal or a terminal followed by a single nonterminal.

Type 2: The second category is the *type 2* category which is also known as *context-free grammar* (CFG). The productions of type 2 grammar are of the form

$$A \rightarrow (\sum \cup V)^*$$

The left-hand-side of every production in *type 2* grammar consists of one non terminal only, while the right-hand-side consists of a combination (union) of terminals from $\sum$ and non-terminals from V. The name of the automaton which accepts the type 2 grammar is *pushdown automaton*.

Type 1: The third category of Chomsky classification of grammar is type 1 grammar which is also known as *context-sensitive grammar*. It has the following form of production:

$$(\sum \cup V)^* \rightarrow (\sum \cup V)^*$$

Here, combination of variable and terminals are in both side. But the size of the string produced on the right-hand-side should either be greater than or equal to the size of the string on the left-hand-side of the production. *Linear bounded automaton* recognizes the language generated by type 1 grammar.

Type 0: The fourth category is termed as type 0 grammar. This grammar is also known as *unrestricted grammar*. The language generated by type 0 grammars are accepted by Turing machine. The form of production of type 0 grammar is

$$(\sum \cup V)^* \rightarrow (\sum \cup V)^*$$

The production rules are same as type 1 but it has no restrictions.

Space for learners:

CHECK YOUR PROGRESS-II

4. Choose the correct option:

i. Language of finite automata is generated by
   a) Type 0 grammar
   b) Type 1 grammar
   c) Type 2 grammar
   d) Type 3 grammar

ii. Regular expression of all strings start with ab and ends with ba is
   a) $(a+b)^*ab(a+b)^*$
   b) $ab(a+b)^*ba$
   c) $(a+b)^*ab(b+a)^*$
   d) aba^*b^*ba

iii. L= $\{\varepsilon, b, bb, bbb, bbbb, \ldots \ldots\}$ is represented by
   a) a^+
   b) a^*
   c) both a) and b)
   d) none of these

iv. Given: $\sum$= {a, b}, L= {x$\epsilon\sum^*$|x is a string combination}. $\sum^4$ represents which among the following?
   a) {aa, ab, ba, bb}
   b) {aaaa, abab, ε, abaa, aabb}
   c) {aaa, aab, aba, bbb}
   d) All of these

v. Regular expression for all strings starts with *ab* and ends with *bba* is
   a) $ab(a+b)^*bba$
   b) aba^*b^*bba
   c) $ab(ab)^*bba$
   d) All of these

Space for learners:

1.7 SUMMING UP

- A formal language is a set of strings of symbols drawn from a finite alphabet. It can be specified either by a set of rules that generates the language, or by a machine that accepts or recognizes the language.
- An alphabet $\sum$ is a finite and non-empty set of symbols.
- A string is a finite sequence of symbols from some alphabet.
- A language L over some alphabet $\sum$, is a collection of strings over the alphabet. For example,

  L= {ε,1,111, …} is a language over the alphabet {1}

  L={$0^n1^n2^n$: n≥ 1} is a language.
- The Kleene closure of a language L is denoted by L^*.

  L^* = {Set of all words over $\sum$}

  ={word of length zero, words of length one, words of length two, ..}

  = $L^0 \cup L^1 \cup L^2 \cup$ ... .
- If is an alphabet then positive closure of $\sum$ is denoted by $\sum^+$ and is defined as $\sum^+ = \sum^* - \{\varepsilon\}$
- A grammar consists of four items: a set of terminals $\sum$, a set of non-terminals V, a set of productions P, and a special symbol S, known as start symbol which is a nonterminal and is belongs to V.
- Noam Chomsky classified the grammar into four categories which are based on their type of production.
- Type 3 is known as regular grammar, type 2 is known as or context-free grammar, type 1 in known as context-sensitive grammar and type 0 grammar is known as unrestricted grammar.

1.8 ANSWERS TO CHECK YOUR PROGRESS

1. Substrings of the string 011 are: ε, 0, 1, 01, 11, 011
2.

   $\sum^2$= {00, 01,10,11

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

Space for learners:

3. Given L = { a, ab }. Then we can determine L^* as

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

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

$L^+ = L^1 \cup L^2 \cup \ldots\ldots$

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

4. (i) (d) Type 3 grammar
   (ii) (b) $ab(a+b)^*ba$
   (iii) (b)a^*
   (iv) (b) {aaaa, abab, ε, abaa, aabb}
   (v) (a) $ab(a+b)^*bba$

Space for learners:

1.9 POSSIBLE QUESTIONS

1. Define Kleene star. Give examples.
2. What is a language?
3. Define $\sum^+$.
4. Define empty string.
5. Define prefix and suffix of a string with examples.
6. Define length of a string.
7. Define alphabet with suitable examples.
8. Define a regular language.
9. Give the formal definition of grammar. Write the categories of grammars provided by Noam Chomsky with their production types.
10. Define a grammar of a language.
11. Define regular expressions? Give some examples of regular expression.
12. Write the Regular expression for the following languages/sets
    i) L= {aa, aaaa, aaaaaa, aaaaaaaa,….}
    ii) Language L over {0,1} such that every string in L ends with 11
    iii) Language L over { a, b, c} such that every string in L ends with 11

iv) L= {00,001,0011,00111,….}

v) Set of all string over {0,1}containing exactly one 0

vi) Set of all strings over { a, b} containing exactly two a' s.

vii) Set of all strings over {a, b, c} beginning with c and ending with cc.

Space for learners:

1.10 REFERENCES AND SUGGESTED READINGS

1. Mishra, K. L. P., & Chandrasekaran, N. (2006). *Theory of Computer Science: Automata, Languages and Computation*. PHI Learning Pvt. Ltd.
2. Nagpal, C. K. (2012). *Formal Languages and Automata Theory*. Oxford University Press.
3. Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to automata theory, languages, and computation*. Pearson Education

UNIT 2: INTRODUCTION TO FINITE AUTOMATA

Space for learners:

Unit Structure:

2.1 INTRODUCTION

This Unit discusses fundamental concepts of Theory of Computations. This unit covers the concepts around Deterministic and Non deterministic automata thoroughly with easily understandable examples. Differences, theorems and conversions are also easily discussed with mathematical techniques. It discusses how to construct finite automata for any language, whether it is a DFA, NFA or NFA with empty moves. Basically, for different input symbols, when the machine state is not determined, i.e., machine can move to any states of the automaton, it is called as Nondeterministic finite automata (NFA) and if the machine state is determined then it is known as Deterministic finite automata (DFA). So, let's study their detailed definitions with different type of properties.

Space for learners:

2.2 UNIT OBJECTIVES

After going through this unit, you will be able to –

- Get full concept on DFA and NFA with examples.
- Construct DFA, NFA for any given example
- Convert an NFA into DFA
- Convert a ε-NFA into its equivalent DFA.
- Equivalence of NFA and DFA.
- Discuss the differences between DFA, NFA and ε-NFA.
- Minimize states in a DFA.
- Check a particular string is belongs to finite automata or not.

2.3 DETERMINISTIC FINITE AUTOMATA

It is a finite automaton (FA) where our machine exists only one place at a time. For each input symbol, the machine state is determined, i.e., machine will move to only one certain state, hence it is called as deterministic finite automata. A deterministic finite automata consists of five tuples {**Q, Σ, q_0, F, δ**} where

Q represents finite set of states.

Σ represents set of all input symbols, i.e., Alphabet.

q_0 represents Initial state.

F represents finite set of final state/states.

δ represents Transition Function, which takes two arguments, a state and an input symbol, it returns a single state. So, **δ : Q X Σ -> Q.**

Some real-life examples of Deterministic Finite Automata (DFA) are lifts in buildings, text parsing, video game character behavior, security analysis etc. A Deterministic Finite Automata (DFA) generally represented by digraph, which is known as transition diagram or state diagram, where vertices represent states and arcs shows the transition from one state to another state.

Let us now discuss one example of Deterministic Finite Automata (DFA). Suppose our DFA's tuples look like below-

Q = {a, b}

Σ = {0, 1}

q_0 = {a}

F = {b}

For transition function **δ**, we need to show the mappings from one state to another state on each input symbols, that's why we need a table:

States	**Input Symbols 01**	
a	b	a
b	b	a

So, from the table we can easily understand that on accepting input symbol '0' state 'a' is moving to state 'b' and on accepting input symbol '1' state 'a' remains unchanged, i.e., self-looped and In this manner, we can easily identify the transitions of state 'b' as well. So, let's draw a diagram for the above example:

Space for learners:

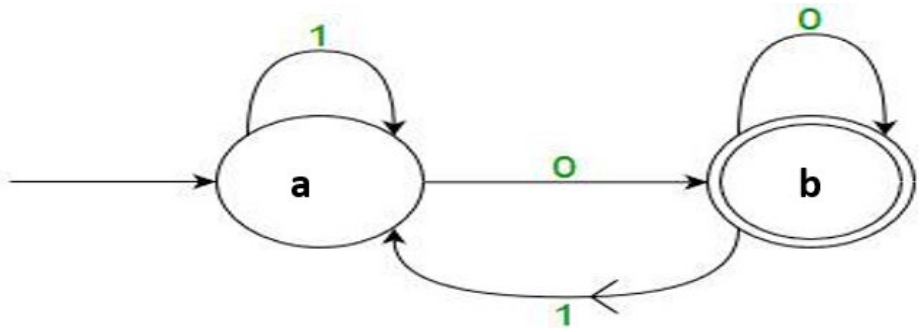


Fig. 1: *A DFA example*

Now, additionally, we can discuss this above diagram more. Check the diagram, see in the final state 'b', input symbol '0' is self-looped and from state 'a', by consuming only '0' we can reach the final state. So, we can easily say, this is a DFA, which will accept all strings ending with zero. Since it is mathematical way to design a finite automata, so there can be numerous number of ways or patterns to design and draw it. But, a finite automaton with minimum number of states is always preferred.

2.4 NONDETERMINISTIC FINITE AUTOMATA (NFA)

2.4.1 Non-Deterministic Finite Automata (NFA)

A kind of finite automata (FA) where our machine can exist in multiple states at the same time. For input symbols, the machine state is not determined, i.e., machine can move to any states of the automaton, hence it is called as nondeterministic finite automata (NFA). That's why Non-Deterministic Automata (NFA) is more complex than DFA. Just like Deterministic Finite Automata (DFA), NFA also consists of five-tuple $\{\mathbf{Q, \Sigma, q_0, F, \delta}\}$, where

Q represents set of all states.

Σ represents set of all input symbols, i.e., Alphabet.

$\mathbf{q_0}$ represents Initial state.

F represents set of final state or states.

Space for learners:

δ represents Transition Function, which takes two arguments, a state and an input symbol, it returns any combination of Q states. So, **$\delta : Q \times \Sigma \rightarrow 2^Q$.**

If we compare this transition function with DFA's transition function, we know that Q is the subset of 2^Q which indicates Q is contained in 2^Q or Q is a part of 2^Q, however, the reverse isn't true. So mathematically, we can say that all DFA is NFA but inverse is not true.

Space for learners:

2.4.2 NFA Detailed Example

Some real-life examples of Nondeterministic Finite Automata (NFA) include playing cards, Tic tac toe and Ludo etc. As we have studied in the section of Deterministic Finite Automata (DFA), same notation technique i.e., digraph is used to draw Nondeterministic Finite Automata (NFA).

Let us now discuss earlier example for Non-Deterministic Automata (NFA). So, suppose our NFA's tuples look like below-

$Q = \{a, b\}$

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

$q_0 = \{a\}$

$F = \{b\}$

For transition function **δ**, we need to show the mappings of states on each input symbols, that's why we need a table:

States	Input Symbols 01
a	aa, b
b	-b

From the above table, we can easily understand that on accepting input symbol '1' state 'a' can move either to state 'b' or to state 'a' and on accepting input symbol '0' state 'a' remains unchanged, i.e., self-looped. So, let's draw a diagram for the above example:

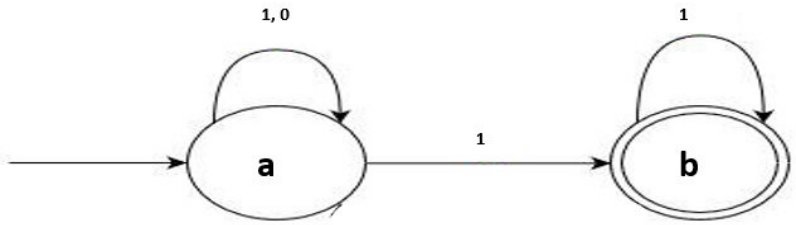


Fig. 2: A NFA example

So, according to the diagram, see in the final state 'b', input symbol '1' is self-looped and from state 'a', by consuming only '1' we can reach the final state. So, we can easily say, this is an NFA, which will accept all strings ending with one. Furthermore, we can see there is no transition for input symbol '0' from final state 'b', this type of mechanisms we can use while constructing a non-deterministic finite Automata.

2.4.3 NFA v/s DFA

We have studied Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA) in previous sections and learnt to know how these automaton works. We have seen that both the types of automata are quite similar to each other, both NFA and DFA have same power and each NFA can be converted into a DFA, but they broadly differ from each other. Their differences are listed below:

Deterministic Finite Automata(DFA)	Nondeterministic Finite Automata(NFA)
For a particular input symbol, machine can be only in one state.	For a particular input symbol, machine can be in one state or multiple states.
In this type of Automata, next state is clearly specified.	In NFA, there can be numerous numbers of possible next states.
Every DFA is NFA.	Every NFA is not DFA.
A string is accepted by DFA, if it is terminates in a final state. Otherwise it rejects the string.	A string is accepted by NFA, if it's at least one of combinational transitions terminates in a final states.
It requires more space.	It requires less space than DFA.
Transition function is $\delta : Q \times \Sigma \rightarrow Q$, which means on any input symbol from our alphabet, machine state will move to only one state from Q.	Transition function is $\delta : Q \times \Sigma \rightarrow 2^Q$, which means on any input symbol from our alphabet, machine state will move to any possible combinations of states from set Q.
It is more difficult to construct.	It is easier to construct.

Space for learners:

CHECK YOUR PROGRESS-I

1: What are DFA and NFA's?

2: What is Transition Table? Give one example.

3: Write down the differences between DFA and NFA.

4: Write the transition function for DFA and NFA.

Space for learners:

2.4.4 Membership Example

Given the NFA M, is 01001 accepted by the NFA? The transition function for the given NFA is

Inputs States	**0**	**1**
->q0	{q0, q3}	{q0, q1}
q1	_	{q2}
q2	{q2}	{q2}
q3	{q4}	_
*q4	{q4}	{q4}

Solution: So, we will use transition function to solve it. As per transition table, q0 & q4 are initial and accepting states respectively.

δ (q0, 0) = {q0, q3}

δ (q0, 01) = δ (δ (q0, 0), 1)
= δ ({q0, q3}, 1)
= δ (q0, 1) U δ (q3, 1)
= {q0, q1}

δ (q0, 010) = δ (δ (q0, 01), 0)
= δ ({q0, q1}, 0)
= δ (q0, 0) U δ (q1, 0)
= {q0, q3}

δ (q0, 0100) = {q0, q3, q4}

$$\delta(q0, 01001) = \delta(\delta(q0, 0100), 1)$$
$$= \delta(\{q0, q3, q4\}, 1)$$
$$= \delta(q0, 1) \cup \delta(q3, 1) \cup \delta(q4, 1)$$
$$= \{q0, q1, q4\}$$

We know, q4 is the final state and q4 is in the final set of states. So, we can say the string 01001 is accepted by the NFA. In similar way, we can check any string is either accepted or rejected by NFA and DFA.

2.4.5 NFA with Empty Moves

A kind of finite automata which contains ε (null or empty) move or instantaneous transition. As we studied, a nondeterministic finite automaton (NFA) can have zero, one, or multiple transitions corresponding to a particular symbol. It is defined to accept the input if there exists some choice of transitions that cause the machine to end up in an accept state. With NFA, we can easily solve complex problems. Epsilon NFA is nothing but an NFA with an additional feature named Epsilon (ε), is a convenient feature with which we can construct even more complex and bigger problems. Both NFA and ε-NFA can recognize same language. An example of ε-NFA is given below:

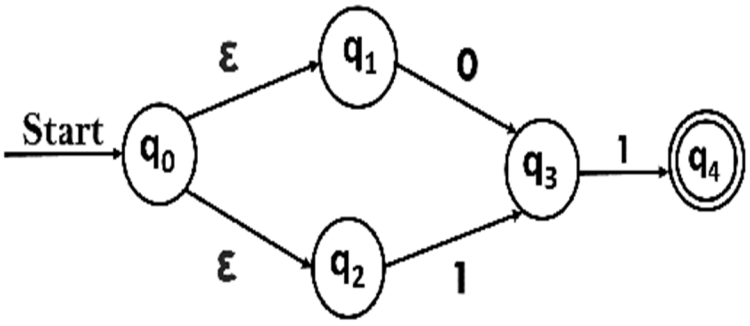


Fig. 3: a ε-NFA Example

Check the above diagram, and see from the state q_0, the machine is moving to state q_1 andq_2 with ε transition that means without input symbol q_0 is changing its state. The transition table will look like:

Space for learners:

States	Input Symbols		
	0	ε	1
q_0	-		q_1 ,q_2 -
q_1	q_3-		-
q_2	-		-q_3
q_3	-		-q_4
q_4	-		- -

Space for learners:

2.4.4.1 ε-closure

ε-closure is calculated for different states of an ε-NFA. ε-closure of a state 'q' means a set of states which can be reached from the state 'q' with ε move (empty/null move) including the self-state. That means set of states that can be reached without any input symbol is ε-closure of a state. Now, Let us find out ε-closure for each state of a ε-NFA given in figure 3:

ε-closure $\{q_0\} = \{q_0, q_1, q_2\}$
ε-closure $\{q_1\} = \{q_1\}$
ε-closure $\{q_2\} = \{q_2\}$
ε-closure $\{q_3\} = \{q_3\}$
ε-closure $\{q_4\} = \{q_4\}$

CHECK YOUR PROGRESS-II

5: Discuss Epsilon NFA with example.

6: What is ε-closure?

2.5 EQUIVALENCE OF DFA AND NFA

In this section, we will discuss the equivalence of DFA and NFA, which means their capability of recognizing language. As we studied deterministic finite automata and non-deterministic finite automata, it

looked like they are different from each other. Their transition diagram, working etc. are different but when comes to recognize a language it turns out to be an equivalent of each other. We can convert an NFA to its equivalent DFA by any conversion algorithm. So, here we will prove the equivalence of NFA and DFA i.e., both NFA and DFA can recognize same type of languages which means for any DFA D, there is an NFA N such that L(N) = L(D) and for any NFA N, there is a DFA D such that L(D) = L(N).

Space for learners:

2.5.1 Equivalence Theorem

Let's formally state the theorem below:

Let for any language, and suppose L is accepted by NFA N = (Σ, Q, q_0, F, δ). There exists a DFA D= (Σ, Q', q'_0, F', δ') which also accepts L. (L(N) = L(D)).

We just need to prove that DFA D is equivalent to NFA N. Through Induction method, we can prove it if we allow each state of DFA D to represent the state or set of states in the NFA N. So, firstly, let's configure the parameters of DFA D (Σ, Q', q'_0, F', δ'), where

$Q' = 2^Q$ and $q'_0 = \{q_0\}$

F' = {q ∈Q'|q ∩ F ≠ Ø}, where F' is the set of states in Q' and F is the set of final states in NFA.

δ' is the transition function of DFA D.

$\delta'(q,a) = U_{p \in q}\ \delta(p,a)$ for q∈Q' and a∈ Σ

We know from the transition function of both NFA and DFA, each state in the set of states Q' in D is nothing but a set of states itself from Q in N. For each state p in state q in Q' of D(p is a single state from Q), determine the transition δ(p,a). δ(p,a) is the union of all δ(p,a).

Now, we can easily prove that $\delta''(q_0',x) = \delta''(q_0,x)$ for every x. i.e., L(D) = L(N)

Basic Step:

Let x be the empty string ε.

$\delta''(q_0',x) = \delta''(q_0', \varepsilon)$

$= q_0'$

$=\{ q_0\}$

$= \delta'(q_0, \varepsilon)$

$= \delta'(q_0, x)$

Inductive Step:

Assume that for any y with $|y|>=0$, $\delta''(q_0',y) = \delta'(q_0',y)$

If we let $n=|y|$, then we need to prove that for a string z with $|z| = n+1$, $\delta''(q_0',z) = \delta'(q_0',z)$. We can then represent the string z as a concatenation of string y and symbol a from the alphabet Σ ($a \in \Sigma$).

So, z=ya

$\delta''(q_0',z) = \delta''(q_0',ya)$

$=\delta'(\delta'' (q_0',y),a)$

$= \delta'(\delta' (q_0,y),a)$ (assumption)

$= U_{p\in \delta' (q0,y)} \delta' (p,a)$

$= \delta' (q_0, ay)$

$= \delta' (q_0, z)$

Now, DFA D accepts a string iff $\delta'' (q_0', x) \in F'$. From the above explanation, it follows that D accepts x iff$\delta'(q_0, x) \cap F \neq \emptyset$. So, a string is accepted by DFA D, if and only if, it is accepted by NFA N.

There is another alternative and easy way to prove this theorem, approach is given below:

Theorem: A language L is accepted by a DFA *if and only if* it is accepted by an NFA.

Proof:

If part:

Prove by showing every NFA can be converted to an equivalent DFA.

Only-if part:

Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA.

By showing these two parts, we can easily solve the above Theorem.

Space for learners:

2.5.2 NFA to DFA Construction

Space for learners:

From the above equivalence theorem, we can conclude that there exists an equivalent DFA for any NFA. In this section we will learn to construct corresponding DFA for an NFA. So, let's discuss subset construction method to construct a DFA for NFA:
Let our NFA is $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$.

Our aim is to build a corresponding DFA $D=\{Q_D, \Sigma, \delta_D, \{q_0\}, F_D\}$ such that $L(D)=L(N)$

Subset Construction:

1. Q_D= all subsets of Q_N (i.e., power set)
2. F_D=set of subsets S of Q_N such that $S \cap F_N \neq \Phi$
3. δ_D:for each subset S of Q_N and for each input symbol a in Σ:

$\delta_D(S,a) = \cup \delta_N(p,a)$

For easy understanding of Subset construction method, we will take an example to construct a corresponding DFA from its NFA. Let's take a language *L = {w | w ends in 01}*, for this NFA will be:

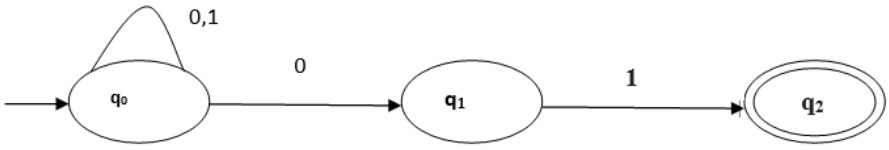


Fig. 4: a NFA Example-to convert to DFA

We need to construct a DFA for this NFA by subset construction method. So, transition table of the NFA will look like:

States	Input Symbols 01	
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
q_2	Ø	Ø

Now, as per algorithm, we first need to find out all subsets of states. We have three states { q_0, q_1, q_2 }, then there subsets are – Ø, { q_0}, { q_1}, { q_2}, { q_0, q_1}, { q_1, q_2}, { q_0, q_0}, { q_0, q_1, q_2} i.e., 2^3=8 subsets of states can be possible. After enumerating all the possible subsets, check the transition table of NFA, from which we have to determine important transitions. We have to give importance to the starting state, here starting state is q_0, now check, from q_0 only we can easily reach to other subsets of states like { q_0, q_1} and { q_2}. So, we will retain only those states which are reachable from {q_0}. So, let's construct the transition table of DFA. Since, our starting state is {q_0}, we will start from this, after writing its mappings, we need to find out the mappings of our new state {q_0, q_1}, for this we need to check the mappings of q_0, q_1 separately in the NFA table and then union it. In this way, we will find the new subset of states and eventually we will move to the final state. Since, our final state in NFA is q_2, so, in DFA table, any combination of subset of states, which contains q_2 will become final state of the corresponding DFA.

States	**Input Symbols 01**	
{q_0}	{ q_0, q_1}	{ q_0}
{ q_0, q_1}	{ q_0, q_1}	{ q_0, q_2}
{ q_0, q_2}	{ q_0, q_1}	{ q_0}

So, from this DFA transition table, we can finally construct our resulting DFA, which is:

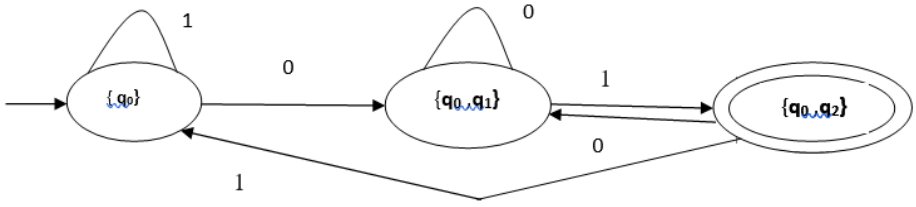


Fig 5: Resulting DFA of an NFA (Figure 4)

Now, we will discuss another method, which is easier than this. Method is known as Lazy Creation; Aim is to avoid enumerating all of power

Space for learners:

sets of states. In this method, we will create states of the resulting DFA by lazy creation of sets. Let us try to understand this method:

Let's take earlier example of figure 4 and check its transition table. Now, checking only that transition table, we can easily construct our DFA transition table by lazy creation of sets. Firstly, we need to check the starting state, from starting state $\{q_0\}$ we can write the states for its different input symbols. Now, see we get a new state named $\{ q_0, q_1\}$, so whenever we get a new state we need to define it first, i.e., we need to find its mappings. Now, we get another new state $\{ q_0, q_2\}$, so we need to define it. After defining it, check if there is any new state present or not, if there is a new state then define it. Define all the new states until no new state is there. When there is no new state present in my transition table, we are ready to draw the diagram of corresponding DFA, So for the example's DFA diagram, check figure 5.

States	**Input Symbols**	
	0	**1**
$\{q_0\}$	$\{ q_0, q_1\}$	$\{ q_0\}$
$\{ q_0, q_1\}$	$\{ q_0, q_1\}$	$\{ q_0, q_2\}$
$\{ q_0, q_2\}$	$\{ q_0, q_1\}$	$\{ q_0\}$

2.5.3 ε-NFA to DFA conversion

We need to recall ε-closure definition from previous sections. We have learnt how to find out ε-closure of each state of a ε-NFA. ε-NFA to DFA conversion is the easier conversion technique among all conversion techniques. Let's take an example of an ε-NFA as in figure 6, then very first we need to find out ε-closure of each states. Steps to convert ε-NFA to DFA are-

Step 1: Take ε-closure for the beginning state of NFA as beginning state of DFA.

Step 2: Find the states that can be traversed from the present for each input symbol.

Space for learners:

Step 3: If any new state is found take it as current state and repeat step 2.

Step 4: Do repeat Step 2 and Step 3 until no new state present in DFA transition table.

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

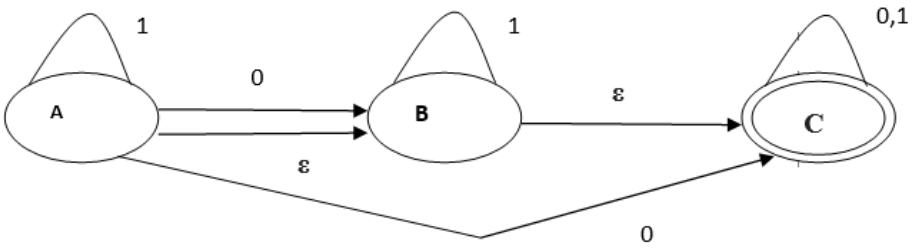


Fig. 6: Epsilon NFA

Transition Table will be:

States	Input Symbols 0ε1		
A	B,C	B	A
B	-	C	B
C	C	C	

For the above example ε-closure are as follows:

ε-closure (A) : {A, B, C}
ε-closure (B) : {B, C}
ε-closure (C) : {C}

Now, using the algorithm steps, we will construct the transition table of DFA:

States	Input Symbols 01	
{A,B,C}	B,C	A,B,C
{B,C}	C	B,C

Space for learners:

{C}	C	C

So, the resulting DFA is:

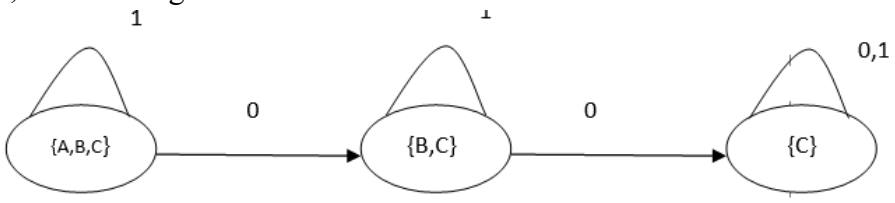


Fig. 7: DFA of an Epsilon NFA (fig. 6)

CHECK YOUR PROGRESS-III

7: What do you mean by lazy creation of sets?

8: Discuss the equivalence theorem of DFA and NFA.

9: What is the power of NFA and DFA in recognizing languages?

2.6 MINIMIZATION OF FA

Minimization of Finite Automata means reducing the useless and redundant states from given Finite automata. Here, we are saying FA, we are mainly indicating DFA. Reducing number of states leads our automaton faster, consumes very less space and eventually become easier to implement. Steps to minimizing DFA are given below:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. δ (q, a) =p
2. δ (r, a) = p

Space for learners:

That means, find the two states which have the same value of a and b and remove one of them.

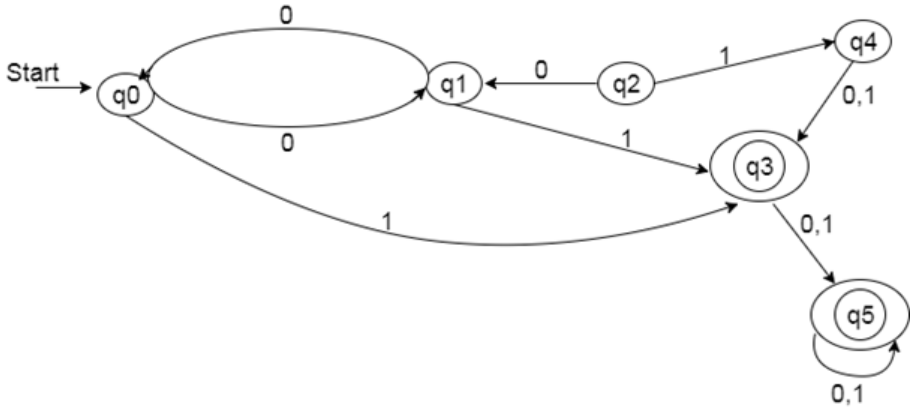
Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Let us take an example to illustrate this:

Suppose we have finite automata -



In the first step, we will try to find unreachable states. From the diagram, we can say that q2 and q4 are the unreachable states. We will remove all unreachable states.

In the second step, we will construct the transition table for the rest of the states. q0 is the initial state, q3 and q5 are the final states.

States	Input Symbols 01
q0	q1 q3
q1	q0 q3
*q3	q5q5
*q5	q5 q5

Space for learners:

In the third step, we will divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

States	Input Symbols 01	
q0	q1	q3
q1	q0	q3

2. Another set contains those rows, which starts from final states.

States	Input Symbols 01	
q3	q5	q5
q5	q5	q5

In fourth step, Set 1 has no similar rows so set 1 will be the same.

In fifth step, in set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

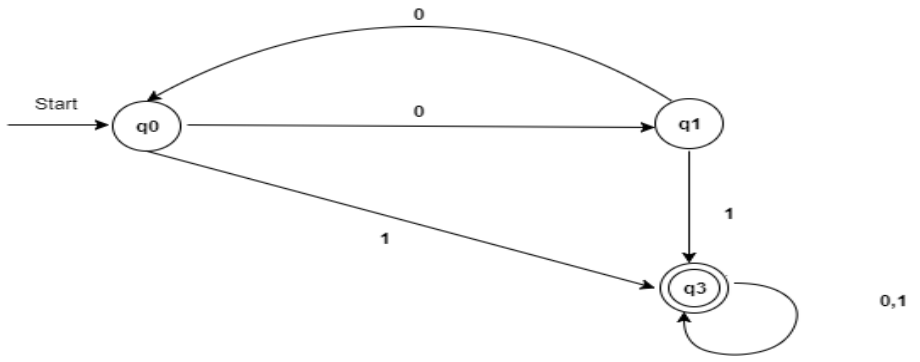
States	Input Symbols 01	
q3	q3	q3

In sixth step, we will combine set 1 and set 2 as:

States	Input Symbols 01	
q0	q1	q3
q1	q0	q3
*q3	q3	q3

Space for learners:

This is nothing but final minimized DFA transition table. Using this table, we can draw our DFA-



Space for learners:

2.7 SUMMING UP

- For each input symbol, if our machine will move to only one certain state, it is a deterministic finite automaton
- If our machine is moving to any combination of states in the machine then it becomes non deterministic in nature.
- For describing complex problems, NFA is used and ε– NFA is a kind of NFA having epsilon feature, which helps states to move to other states or self state without input symbol.
- We can convert NFA, ε– NFA to its equivalent DFA.
- Some finite automata has large number of useless and redundant states, which consumes time and space both, that is why we learnt to reducing states of finite automata.

2.8 ANSWERS TO CHECK YOUR PROGRESS

1. DFA: When our finite automata's state is determined i.e., for each input symbol, machine will move to only one certain state, hence it is called as deterministic finite automata. A deterministic finite automata consists of five tuples {**Q, Σ, q_0, F, δ**} where-

Q represents finite set of states.

Σ represents set of all input symbols, i.e., Alphabet.

q_0 represents Initial state.

F represents finite set of final state/states.

δ represents Transition Function, which takes two arguments, a state and an input symbol, it returns a single state. So, **δ: Q X Σ -> Q.**

Some real-life examples of Deterministic Finite Automata (DFA) are lifts in buildings, text parsing, video game character behavior, security analysis etc.

NFA: When our finite automata's state is not determined i.e., machine can move to any states of the automaton. It consists of five-tuples {**Q, Σ, q_0, F, δ**} where-

Q represents set of all states.

Σ represents set of all input symbols, i.e., Alphabet.

q_0 represents Initial state.

F represents set of final state or states.

δ represents Transition Function, which takes two arguments, a state and an input symbol, it returns any combination of Q states. So,**δ : Q X Σ -> 2^Q.**

2. Transition Table contains the information regarding states and its input symbols. On different input symbols, different states of machine are moving to different states, these information are available in transition table.

For Example, typical transition tables look like-

States	**Input Symbols**	
	0	**1**
a	b	a
b	b	a

3. Refer the section 2.3.3

Space for learners:

4. Transition function of DFA is δ: Q X Σ -> Q, which means the function takes two arguments, a state and an input symbol, it returns a single state.

Transition function of NFA is δ: Q X Σ -> 2^Q, which means the function takes two arguments, a state and an input symbol, it returns any combination of Q states.

5. Refer section 2.3.5

6. ε-closure means set of states that can be reached without any input symbol from any state of the ε-NFA.

7. Lazy creation of sets is a technique to convert a NFA to DFA. It used while we are constructing a DFA from given NFA. Unlike subset construction method, here we don't use to enumerate all the subsets of states. Whenever a new state arrived, we just calculate its mappings. When there is no new state, we draw equivalent DFA.

8. Refer section 2.4.1

9. From the theorem, we can say, A language L is recognized by a DFA if and only if there is an NFA N such that L (N) = L. So, NFA and DFA can recognize same set of languages.

2.9 POSSIBLE QUESTIONS

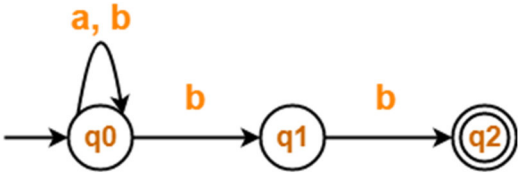
1: Discuss about NFA with example.

2: Draw deterministic and non-deterministic finite automata which accept 00 and 11 at the end of a string containing 0, 1 in it.

3: Write down the differences between NFA and DFA.

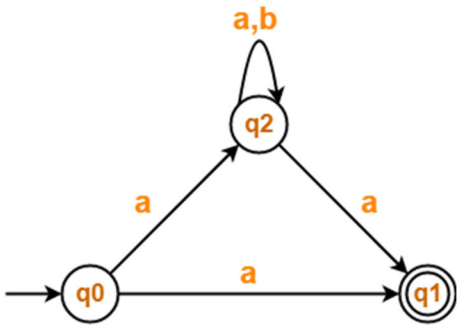
4: Convert Following NFA to its equivalent DFA.

a)

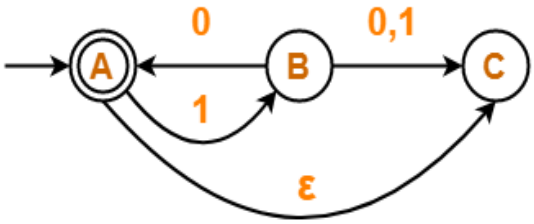


Space for learners:

b)

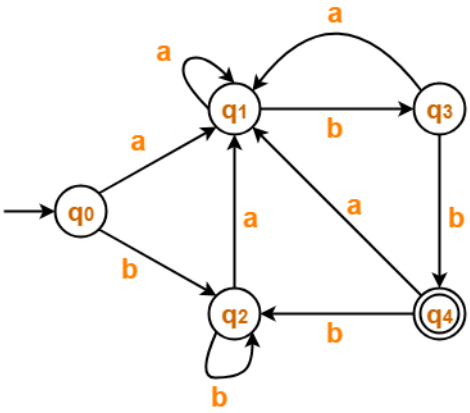


5: Convert the following ε - NFA to its equivalent DFA.



6: Write down the applications of Finite automata.

7: Minimize states of the following FA.



Space for learners:

2.10 REFERENCES AND SUGGESTED READINGS

- Introduction to Automata Theory, Languages and Computation, *John E Hopcroft ,Matwani& Jeffery D. Ullman*
- Introduction to Languages and the Theory of Computation, *John C. Martin*
- Elements of the Theory of Computation, *Lewis & Papadimitriou*
- GeeksforGeeks - https://www.geeksforgeeks.org/
- Gatevidyalay- https://www.gatevidyalay.com/
- Equivalence Theorem - https://www.neuraldump.net/
- Minimization Techniques - https://www.javatpoint.com/

Space for learners:

UNIT 3: REGULAR SETS AND REGULAR EXPRESSIONS

Space for learners:

Unit Structure:

3.1 INTRODUCTION

Regular Expressions are the simplest way to describe set of various strings in a language. Starting from our own utility software like compilers, game player etc to real life, regular expressions has enormous number of applications in various fields. That is why we need to study regular expressions, set and eventually regular language. This Unit basically discusses various definitions, various properties of regular language with examples in easy language. Finite automaton recognizes regular language, so this unit covers the way of designing DFA for its regular language. Regular language has some theorems like other languages, so for proving that we have used easy mathematical ways. Whenever we are studying different definitions and properties of different languages, those mathematical terms are very important in this subject. Only using mathematical terms, pumping lemma concept is introduced and it is very crucial to determine the type of the language. So, basically Regular language is nothing but regular sets which are generated from regular expressions using mathematical operations and this language can be recognized by finite automata.

3.2 UNIT OBJECTIVES

After going through this unit, you will be able to –

- Get full concept on regular expressions with examples.
- Create any regular expressions, then language and eventually a DFA for it.
- What is the relation between NFA, DFA and regular expressions?
- How easily we can proof certain theorems using mathematical notations?
- Explain closure properties of regular language.
- Question about membership for any string in a regular language.
- Get to know about different algebraic laws of languages.
- Explain decision problems of regular language.
- Find a given language is regular or not, using Pumping Lemma.

Space for learners:

3.3 REGULAR SETS AND REGULAR EXPRESSIONS

Space for learners:

Regular Expressions are an effective way to represent a language. These are nothing but string-like simple expressions which can be used to define any finite automata language. These expressions are generally a sequence of patterns. Any set which can be represented by regular expression is called a regular set.

Regular Expressions over an input alphabet Σ are-

- ϕ is a regular expression denoting ϕ.
- ε is a regular expression denoting {ε}.
- For each a ∈ Σ, a is a regular expression denoting {a}.
- If a is regular expression, a* (0 or more times a) is also regular.
- If E and F are regular expressions denoting languages L(E) and L(F), then (E+F) is a regular expression denoting L(E) ∪ L(F)
- If E and F are regular expressions denoting languages L(E) and L(F),then (EF) is a regular expression denoting L(E) L(F)
- If E and F are regular expressions denoting languages L(E) and L(F),then (E*) is a regular expression denoting L(E)*.

Example: Some regular expressions with their meanings are given below:

Regular Expression	Meaning
01	A zero followed by a one(concatenation)
0+1	Either a zero or a one
0*	Any number of zeroes
1^+	One or more number of ones.
(0+1)*	All strings over {0,1}
0*10*10*	Strings containing exactly two ones
(0+1)*11	strings which end with two ones

If a language can be expressed and described by regular expressions then it is known as regular language. And obviously, grammar of that language is called regular grammar.

Space for learners:

3.3.1 Precedence Rules

Precedence rules of regular expression are similar to the rules of general arithmetic expression. We will consider exponentiation first, then multiplication, then addition. We will take Kleene closure as exponentiation, concatenation as multiplication, and union as addition and the precedence rules are identical.

3.3.2 Finite Automata and Regular Expressions

Regular expressions can define the same class of languages as finite automata. DFA, NFA, Epsilon-NFA all can express regular languages.

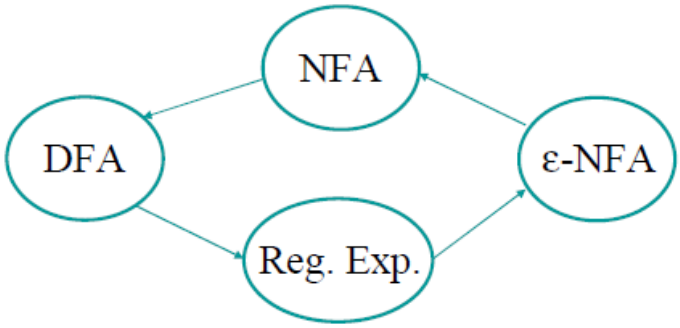


Fig. 1: Relation of Finite automaton and regular expression

CHECK YOUR PROGRESS-I

1: What is a regular expression?

2: How regular languages behave in case of union and concatenation operation?

3: What is the difference between regular expression 0* and 0^{+}.

4: Discuss the relation between regular expressions and Finite automata.

3.3.3 Inductive Definition of Regular Language

Base Case Definition: Let our input alphabet Σ, then {} is a regular language; {ε} is a regular language; {a} is a regular language for any character a in Σ.

Inductive Case Definition: If L_1 and L_2 are regular languages, then $L_1 \cup L_2$, L_1L_2, L_1^* are also regular languages.

Completeness: Regular languages are those languages which can be generated using the above rules.

3.4 CLOSURE PROPERTIES OF REGULAR LANGUAGES

Regular expressions are defined through union, concatenation, and closure.

Union: $L_1 \cup L_2 = \{x \mid x \text{ in } L_1 \text{ or } x \text{ in } L_2\}$.

Concatenation: $L_1L_2 = \{xy \mid x \text{ in } L_1 \text{ and } y \text{ in } L_2\}$.

(Kleene) Closure: $L^* = \cup_{i=0,1,..\infty} L_i$, where $L_0 = \{\varepsilon\}$, $L_i = LL_{i-1}$

Example: $\{01, 10\}^* = \{\varepsilon, 01, 10, 0110, 0101, 1010 \ldots\}$

Since we knew that regular languages are closed under union, concatenation and star operation. So we will try to prove that regular languages are closed under intersection and complement operation as well.

3.4.1 Complement

If L_1 is a regular language, then $\overline{L}_1$ is also a regular language.

Suppose we take a finite automata, let's say a DFA D that accepts L_1, and modifies every non-final states as final states and final states as non-final. That means we are just complementing our DFA D. So, our new DFA $\overline{D}$ is nothing but the compliment of D, it will accept some strings because it has final and non-final sates just like other DFAs. So, DFA $\overline{D}$ will accept some new set of strings, eventually a language. By definition, we know that a language a DFA can recognize its nothing

Space for learners:

but a regular language. So, $\overline{L}_1$ is the regular language which will be recognized by our new DFA $\overline{D}$. Hence, if L_1 is a regular language, then $\overline{L}_1$is also a regular language.

3.4.2 Intersection

For regular languages L_1 and L_2, their $\overline{L}_1 \cap \overline{L}_2$is also a regular language.

It is given that L_1 and L_2 are both regular languages, then -

By definition, $\overline{L}_1$ and $\overline{L}_2$ are also regular languages.

By definition, $\overline{L}_1 \cup \overline{L}_2$ is a regular language.

By definition, complement of $\overline{L}_1 \cup \overline{L}_2$ is also a regular language.

So, by applying de-Morgan's law, we can say $\overline{L}_1 \cap \overline{L}_2$ is also a regular language.

Hence, regular languages are closed under union, intersection, concatenation, complement and star operation.

CHECK YOUR PROGRESS-II

5: Discuss Closure properties of regular language.

6: How regular languages are closed under Complement and Intersection operation?

7: What is Kleene Closure or Star Closure?

3.4.3 Algebraic Laws of Languages

Here are some important algebraic laws of Languages we need to know. If L, M and N are three regular languages, then -

- Union is commutative: $L \cup M = M \cup L$
- Union is associative: $(L \cup M) \cup N = L \cup (M \cup N)$
- Concatenation is associative: $(LM)N = L(MN)$
- ϕ is identity for union: $\phi \cup L = L \cup \phi = L$
- ε is left and right identity for concatenation: $\{\varepsilon\}L = L\{\varepsilon\} = L$
- ϕ is left and right annihilator for concatenation: $\phi L = L \phi = L$

Space for learners:

- Concatenation is left distributive over union: L (M∪N) =LM ∪ LN
- Concatenation is right distributive over union: (M∪N) L = ML ∪ NL
- Union is idempotent: L∪L=L
- Star is idempotent: $(L^*)^* = L^*$
- $\phi^* = \{\varepsilon\}$, $\{\varepsilon\}^* = \{\varepsilon\}$
- $L^+ = LL^* = L^*L$, $L^* = L^+ \cup \{\varepsilon\}$

Space for learners:

3.5 DECISION ALGORITHMS FOR REGULAR SETS

Decision Algorithms for a class of languages are properties which try to provide description of a language and discuss whether or not some properties hold. Decision problems can be solved very quickly, very computationally demanding, or unsolvable. Some decision properties of regular class of languages are given below-

- **Emptiness Problem:** Suppose we have given a regular language L, how to check L is empty or not.

  For this, we have to take the DFA for that regular language; we can easily draw the corresponding DFA for L. Now, we will check if there exists a path from initial state to final state. If there is a path, then it is not empty, otherwise it is empty one.

- **Finiteness Problem:** Suppose we have given a regular language L, how to check L is finite or not.

  For this, again we have to draw the DFA for that regular language. Now, we will check if there is a walk with cycle from initial state to final state. If there is at least one cycle in the path, then it is infinite and if there is no cycle present in the DFA, then L is finite.

- **Equivalence Problem:** Suppose we have given two regular languages L_1 and L_2, how to check if $L_1 = L_2$.

  We have to show the symmetric difference of L_1 and L_2 is empty that is, there is no string belonging to one but not both of the languages. So, symmetric difference of L_1 and L_2 can be expressed as: $(L_1 \cap \overline{L}_2) \cup (L_2 \cap \overline{L}_1)$

  Now we need to show $(L_1 \cap \overline{L}_2) \cup (L_2 \cap \overline{L}_1) = \phi$

For getting ϕ, we have to show ($L_1 \cap \overline{L}_2$) and ($L_2 \cap \overline{L}_1$) = ϕ

By looking at the languages, ($L_1 \cap \overline{L}_2$), if we can write $L_1 \subseteq L_2$ and

By looking at the languages, ($L_2 \cap \overline{L}_1$), if we can write $L_2 \subseteq L_1$, then we can conclude $L_1 = L_2$, or if any of these two $L_1 \subseteq L_2$,$L_2 \subseteq L_1$ is false, then we can conclude $L_1 \neq L_2$

- **Membership Problem:** Given a regular language L, we need to check a string suppose x is belongs to that L or not. Simplest solution for this problem is just to draw a DFA for regular language L and then check string x is accepted or not.

For example, L= $\{a^{3n} \mid n \geq 0\}$, which means the language contains strings of a's where count of numbers of a in the string is divisible by 3. Let's draw a DFA for it:

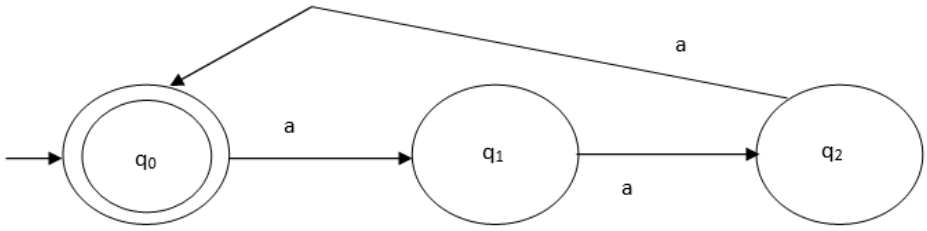


Fig. 2: DFA for the Language L= $\{a^{3n} \mid n \geq 0\}$

From the above diagram, we can easily find out the acceptance status of different strings for this language. Suppose a string is 'aaaaaa', so, our initial state and final state is q_0, we will start moving from q_0, and the string is ending at q_0. Hence, this string belongs to the language. Now, take another example 'aaaaa', & from the diagram we can see that the string is ending at the state q_2. Hence, this string 'aaaaa' does not belong to the language. Since, we have learnt many topics about regular language, now we need to learn how to check a given language is regular or not. For this we will study Pumping Lemma in the next section.

3.6 PUMPING LEMMA FOR REGULAR LANGUAGES

Pumping lemma is used as a proof that the language is not regular. We used pumping lemma as a contradictory measure to proof a language is

Space for learners:

not regular, but if the language satisfies pumping lemma then it can be regular.

Space for learners:

3.6.1 Pumping Lemma

Pumping lemma for Regular Language is -

For any regular language L, there exists an integer n, such that for all w ∈ L with $|w| \geq n$, and x, y, z ∈ Σ, such that w = xyz,
(1) $|xy| \leq n$
(2) $y \neq \varepsilon$
(3) for all $i \geq 0$: $xy^iz \in L$

In the last condition, we are pumping the string 'y'. So whatever the value of i, i.e. string y can be inserted any number of times, but the resultant string should belongs to the language L, If there exists at least one string made from pumping which is not in L, then L is not regular.

CHECK YOUR PROGRESS-III

8: What are the decision problems or algorithms for regular language?

9: State pumping lemma for regular language?

10: Check '001100' string is accepted or rejected by the language L= {x | x ends with at least one zero}.

3.6.2 Examples

Let us discuss some pumping lemma examples-

Approach to solve: Try to find a contradiction to prove that the language is not regular, if we able to do so, then the language is not regular, otherwise it is regular. That's why; we will first assume that the language is a regular one.

Example 1: Checking irregularity of the language **L= {a^nb^n : n>=0}**

So, Let's say our language L= {a^nb^n: n>=0} is regular.

Let m be an integer, we will choose a string w such that w ∈ L and length |w|≥m

Suppose w= $a^m b^m$, according to pumping lemma, w=xyz, so we will divide $a^m b^m$ into three parts and the middle part i.e. y we will pump. Conditions should be fulfilled i.e. |xy| ≤ m, |y|≥1.

So, let's divide it like - $x=a^{m-k}$, $y=a^k$, $z=b^m$, where |k|≥1

From the pumping lemma $xy^iz \in L$, i=0,1,2,3,……..

For i=2, $xy^2z = a^{m-k}a^{2k}b^m = a^{m+k}b^m$

BUT, $a^{m+k}b^m \notin L$, because our language is L= $\{a^n b^n: n>=0\}$, that means equal number of a's followed by equal number of b's. So, It is a contradiction and hence, the language is not regular.

Example 2: Checking irregularity of the language **L= {*ww* | *w* ∈Σ*}**

So, let's say our language L= {*ww* | *w* ∈Σ*} is regular. Here, from the L, we can easily understand that a string w is repeating twice, and the string w is taken from the input alphabet. Assuming our input alphabet Σ ={a,b}, we can take our string as $a^n b a^n b$, where n is an integer.

So, according to pumping lemma, w = $a^n b a^n b$.

So, let's divide it like - $x=a^{n/2}$, $y= a^{n/2}$, $z=ba^n b$

We know to be a regular language, $xy^iz \in L$, i=0,1,2,3,…….., this rule must be satisfied.

BUT, if we take $xy^0z = a^{n/2}ba^n b \notin L$.

For xy^0z, the same string is not repeating twice. It's a simple contradiction and hence, the language is not regular.

Example 3: Checking irregularity of the language **L = { 0^i1^j | i > j}**

So, let's say our language L = $\{0^i1^j | i > j\}$ is regular. By looking at the language definition, we can find out this language has strings which have number of zeroes followed by number of ones, but number of zeroes should be greater than the number of ones. We can take our string as $0^{n+1}1^n$, where n is a positive integer number.

Like earlier example, lets divide our w = xyz, in such a way that |xy| ≤ n, |y|≥1.

Space for learners:

x= 0, $y=0^{n-1}$, $z=01^n$, from the pumping lemma $xy^iz \in L$, i=0,1, 2, 3, ……..

If we check, $xy^2z = 0(0^{n-1})^2\ 01^n = 00^{2n-2}01^n = 0^{2n}1^n \in L$

BUT, if we pump down it, $xy^0z = 0(0^{n-1})^0\ 01^n = 00^001^n = 0^21^n \notin L$, when n>1 and n can be any integer. So, it's contradicting the definition of given language and hence, the language is not regular.

Space for learners:

3.7 SUMMING UP

- Regular Expressions are nothing but strings, which can be used to express and describe a language, regular language.
- Finite automata recognized this regular class of languages.
- While generating regular sets from regular expressions, precedence rules are important to consider.
- Regular languages are closed under union, intersection, concatenation, star and complement operation. Using these mathematical properties we can proof many algorithms on regular languages.
- Like every class of languages, regular languages have also decision problems or algorithms.
- We use pumping lemma to prove that a language is not regular.

3.8 ANSWERS TO CHECK YOUR PROGRESS

1. A regular expression is a string that describes the whole set of strings according to certain rules. Regular Expressions over an input alphabet Σ are-

- ϕ is a regular expression denoting ϕ.
- ε is a regular expression denoting {ε}.
- For each a ∈ Σ, a is a regular expression denoting {a}.
- If a is regular expression, a* (0 or more times a) is also regular.

2. If E and F are regular expressions denoting languages L(E) and L(F), then (E+F) is a regular expression denoting L(E) ∪ L(F) and (EF) is a regular expression denoting L(E) L(F).

3. 0* indicates the sets of any number of zeroes and 0^+ indicates the sets of one or more number of zeroes.

4. Regular expressions can define the same class of languages as finite automata. DFA, NFA, Epsilon-NFA all can express regular languages. For any regular expressions, we can design corresponding finite automata.

5. Closure Properties of Regular languages are-

- Union: $L_1 \cup L_2 = \{x \mid x \text{ in } L_1 \text{ or } x \text{ in } L_2\}$.
- Concatenation: $L_1L_2 = \{xy \mid x \text{ in } L_1 \text{ and } y \text{ in } L_2\}$.
- (Kleene) Closure: $L^* = \cup_{i=0,1,..\infty} L_i$, where $L_0 = \{\varepsilon\}$, $L_i = LL_{i-1}$

Besides from that, Regular languages are also closed under intersection and complement.

6. Regular language is closed under Complement because If L_1 is a regular language, then $\overline{L}_1$ is also a regular language. Regular language is closed under Intersection because for any two regular languages L_1 and L_2, their $\overline{L}_1 \cap \overline{L}_2$ is also a regular language.

7. Kleene or star Closure: If L_1 is a regular langauge, then L_1^* (the Kleene closure of L_1) is also a regular language.

8. The decision problems for regular languages are-

a) Emptiness Problem

b) Finiteness Problem

c) Equivalence Problem

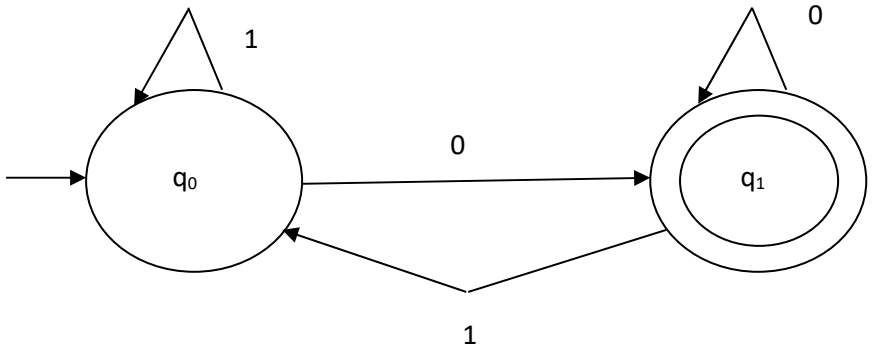
d) Membership Problem

9.Pumping lemma for Regular Language is -

For any regular language L, there exists an integer n, such that for all w ∈ L with $|w| \geq n$, and x,y,z∈ Σ, such that w = xyz,
(1) $|xy| \leq n$
(2) $y \neq \varepsilon$
(3) $xy^iz \in L$, for all $i \geq 0$;

Space for learners:

10. We have given a language L= {x | x ends with at least one zero}, i.e a language which accepts strings which are ending with two zeroes. We need to check the string '001100' is either accepted or rejected. Let's design a DFA for this language:



From the above diagram, we can easily find out the acceptance status of different strings for this language. From our Finite Automata, we can see, our initial state is q_0 and final state is q_1. We have given our string '001100', so we will start from q_0. It is moving like $q_0 \rightarrow q_1 \rightarrow q_1 \rightarrow q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_1$, finally it is ending at q_1, which is our DFA's final state, so the string is accepted by the language.

3.9 POSSIBLE QUESTIONS

1: Which one of the following languages over the alphabet {0,1} is described by the regular expression?
(0+1)*0(0+1)*0(0+1)*
a) The set of all strings containing the substring 00.
b) The set of all strings containing at most two 0's.
c) The set of all strings containing at least two 0's.
d) The set of all strings that begin and end with either 0 or 1.

2: Regular expressions are closed under
a) Union
b) Intersection
c) Kleen star
d) All of the mentioned

3: Which of the following is true?
a) (01)*0 = 0(10)*

Space for learners:

b) (0+1)*0(0+1)*1(0+1) = (0+1)*01(0+1)*
c) (0+1)*01(0+1)*+1*0* = (0+1)*
d) All of the mentioned

4: Write the regular expression for the language accepting all the string containing any number of a's and b's, over the input alphabet $\sum$ = {a,b}.

5: Check the language **L= { *0^n* | *n is a prime number*}** is regular or not.

6: We have studied that two regular languages are equal if they have the same regular expression representation or DFAs. Let L_1 and L_2 denote two regular languages, one of them is given to you as a regular expression while the other is represented as a DFA. How would you verify that they are equal?

7: Discuss the closure properties of regular languages.

8: Discuss the decision properties of regular languages.

9: What will be the regular sets of the following?

(a) (0+1)* (b)(01)* (c)(0+1) (d)$(0+1)^+$

10: What are the applications of Regular expressions and Finite automata?

Space for learners:

3.10 REFERENCES AND SUGGESTED READINGS

- Introduction to Automata Theory, Languages and Computation, *John E Hopcroft, Matwani & Jeffery D. Ullman*
- Introduction to Languages and the Theory of Computation, *John C. Martin*
- Elements of the Theory of Computation, *Lewis & Papadimitriou*
- Sanfoundry - https://www.sanfoundry.com/
- GeeksforGeeks- https://www.geeksforgeeks.org/

UNIT 4: CONTEXT FREE LANGUAGE

Space for learners:

Unit Structure:

4.1 INTRODUCTION

There are certain languages that cannot be described or expressed by finite automata, so we need more powerful mechanism which can recognize complex languages. The recursive structure of CFG is useful for recognizing some set of complex languages. CFG are used for basis of compiler design and implementation, computer vision, linguistics, specification mechanisms for programming languages. We can easily derive any string which belongs to the language of the grammar using derivation techniques. A context free grammar is very flexible because it can be simplified if there are any useless productions or symbols.

Space for learners:

4.2 UNIT OBJECTIVES

This unit covers about context free languages, context free grammar and its derivation techniques. After going through this unit you will be able to:

- Explain about context free languages with examples.
- Create context free grammar for a language.
- Discuss the closure properties of context free languages.
- Find leftmost and rightmost derivation of strings.
- Draw parse tree of a string for a given grammar.
- Check a context free grammar is ambiguous or not.
- Find out useless productions, null productions and unit productions of any context free grammar.
- Simplify any context free grammar.

4.3 CONTEXT FREE LANGUAGE

Context free language (CFL) is a type of language which is generated by a context free grammar (or Type 2 grammar) i.e., a language L is context free if there is a context free grammar (CFG) G, such that L is generated from G. Regular languages are subset of context free

languages. Just like finite automata, which can recognize the set of regular languages, Pushdown automata (PDA) can recognize context free languages. E.g. Arithmetic operations can be generated by context free grammars, so these are context free languages. Simply, Languages which are specified by context free grammars are called context free languages.

Space for learners:

4.3.1 Contexts Free Grammar

It is a formal grammar used to generate possible patterns of strings for a given language. A context free grammar (CFG) is a 4-tuple (V, **Σ**, R, S), where –

V is a set of non-terminals (NT) are also called variables, which are generally denoted by capital letters.

Σ is an alphabet, characters in the alphabet are known as terminals, which are generally denoted by lowercase letters.

R is a set of production or substitution rules that represents the recursive definition of the language. R is a subset of $NT \times (\Sigma \cup NT)^*$. If $(\alpha, \beta) \in R$, if we can write $\alpha \rightarrow \beta$, then $\alpha \rightarrow \beta$ is a production rule, where α contains non-terminal symbols and β may contains terminals or non-terminal symbols or combination of terminal and non-terminal symbols.

S is the starting variable, which is used to derive the string and is one of the variables from the set of V.

4.3.2 CFG Notation

While defining CFG, we used some notations, these are-

- Uppercase letters are non-terminals (NT) and everything else are terminal symbols.
- Start symbol is always from non-terminals set, it will be on the left-hand side of the first production rule.
- Left hand side of the production rule only contains non-terminals, using which we derive strings.
- Right hand side of the production rule can be anything from

terminals and non-terminals set and ε together.

- Rules with common left-hand sides are combined with right-hand sides separated by "|"

As an example, consider the grammar:

S →xSy | ε

The implication is the start symbols is S and the rules are:

S →xSy

S → ε

Space for learners:

4.3.3 Closure Properties of Context Free Language

Closure properties discuss about various operations on Context Free Language is closed. If we are doing an operation on a set and it always produces a member which of the same set type, then we can say the set is closed under that operation. Context free languages have the following closure properties –

Union: Context-free languages are closed under union operation i.e., that if X and Y are both context-free languages, then X∪Y is also a context-free language.

Concatenation: Context-free languages are closed under concatenation operation i.e. that if X and Y are both context-free languages, then XY is also a context-free language.

Kleene Star: Context-free languages are closed under concatenation operation i.e., that if L is a context free language, then L* is also a context free language.

Unlike regular languages, Context free languages are not closed under intersection or complement operation.

For the decision properties of context free languages, emptiness problem, finiteness problem and membership problem all are decidable.

4.3.4 Examples

Let's try to construct the CFG for the language having any number of a's over the set ∑= {a}.

We know the regular expression for above language a*.

Let's try to construct the production rules for this-

$S \rightarrow aS$

$S \rightarrow \varepsilon$

Where S is the starting variable i.e., a non-terminal and a is the input symbol from set ∑ and ε is just a empty string. So, if we want to derive any number of a's then we will start from the starting variable –

S

aS

aaS S is replaced by aS because of first production rule

aaaS S is replaced by aS because of first production rule

aaaaS S is replaced by aS because of first production rule

aaaaaS S is replaced by aS because of first production rule

aaaaaaS S is replaced by aS because of first production rule

aaaaaaε S is replaced by ε because of second production rule

aaaaaa

So, from the derivation we can easily understand that we can get any number of a. If we want to get zero number of a, that means just a empty string, we will first choose second production rule, because S is our starting variable.

Now let's try to construct one intermediate CFG language L = {wxw^R | where w € (a, b)*}, where, w^R is a reverse string of w.

The string that can be generated for a given language is {aaxaa, bxb, abxba, baxab, abbxbba,….}

Production rules for the grammar can be –

$S \rightarrow aSa$

Space for learners:

S → bSb

S → x

From these production rules, we can derive any string of {aaxaa, bxb, abxba, baxab, abbxbba,….}. Suppose for example, String 'abbxbba' can be derived as-

S → aSa

S → abSba S is replaced by bSb using second production rule

S → abbSbba S is replaced by bSb using second production rule

S → abbxbba S is replaced by x using third production rule

Since, at the last line, no non-terminals (in our example, only one NT is given, which is S) are there, hence this is our derived required string.

Space for learners:

CHECK YOUR PROGRESS-I

1: What is Context Free Grammar?

2: Construct a CFG for the language L = $\{0^n1^n \mid n>1\}$

3: Explain closure properties of Context Free language.

4.4 DERIVATION

A derivation is a sequence of steps which begins with the start symbol, uses the production rules to do replacements, and ends with a terminal string.

In one step derivation, u **yields** v in **one-step**, written like u ⇒v, if for some u, v in $(V \cup \sum)^*$, u = xαz and v = xβz where α→βis a rule.

In multistep derivation, u **derives** v, written like u $\Rightarrow^*$v, if there is a chain of one step derivations in the form:

$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow u_3 \Rightarrow u_4 \Rightarrow u_5 \ldots\ldots \Rightarrow v$

4.4.1 Leftmost and Rightmost Derivations

Space for learners:

A leftmost derivation of a sentential form is one in which rules transforming the leftmost non terminal is always applied. Simply, in leftmost derivations, we will always replace the leftmost non-terminals.

A rightmost derivation of a sentential form is one in which rules transforming the rightmost non terminal are always applied. Simply, in rightmost derivations, we will always replace the rightmost non-terminals.

As for example, let's take a simple grammar, consider our earlier example of language a*. For that we have productions rules like-

$S \rightarrow aS$

$S \rightarrow \varepsilon$

Let us consider a string w =aaa

Leftmost Derivation-

$S \rightarrow a\mathbf{S}$

$\rightarrow aa\mathbf{S}$ (Using first production rule)

$\rightarrow aaa\mathbf{S}$ (Using first production rule)

$\rightarrow aaa\varepsilon$ (Using second production rule)

$\rightarrow aaa$

Rightmost Derivation-

$S \rightarrow a\mathbf{S}$

$\rightarrow aa\mathbf{S}$ (Using first production rule)

$\rightarrow aaa\mathbf{S}$ (Using first production rule)

$\rightarrow aaa\varepsilon$ (Using second production rule)

$\rightarrow aaa$

Hence,

leftmost derivation = rightmost derivation

Leftmost and rightmost derivations are just two techniques to derive our strings. So, whatever the strings, if it belongs to the language, we can

get it easily either by leftmost or rightmost derivations. These two derivations techniques will become very easy once we study the concept of parse tree.

Space for learners:

4.4.2 Parse Tree

Parse tree or derivation tree is a geometrical representation of derivations. There always exist a parse tree corresponding to each leftmost derivation and rightmost derivation. A parse tree of a derivation $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow u_3 \ldots\ldots \Rightarrow v$ is a tree in which:

- Each internal node is labeled with a non-terminal symbol.
- Root node of a parse tree is the start symbol of the grammar.
- Each leaf node is labelled with a terminal symbol.
- If a rule T→ T1T2…Tn occurs in the derivation then T is a parent node of nodes labelled T1, T2, …, Tn

As for example, consider the following grammar-

S → aB |bA

S → aS | bAA | a

B → bS | aBB | b

Let us consider a string w = aaabbabbba

Now, let us derive the string w using leftmost derivation.

Derivation-

S → a**B**

→ aa**B**B (Using B → aBB)

→ aaa**B**BB (Using B → aBB)

→ aaab**B**B (Using B → b)

→ aaabb**B** (Using B → b)

→ aaabba**B**B (Using B → aBB)

→ aaabbab**B** (Using B → b)

→ aaabbabb**S** (Using B → bS)

→ aaabbabbb**A** (Using S → bA)

→ aaabbabbba (Using A → a)

So, by looking at the required derived string, we will use our production rules. Let's draw a parse tree for this derivation. Our root node will be S, because S is a starting variable-

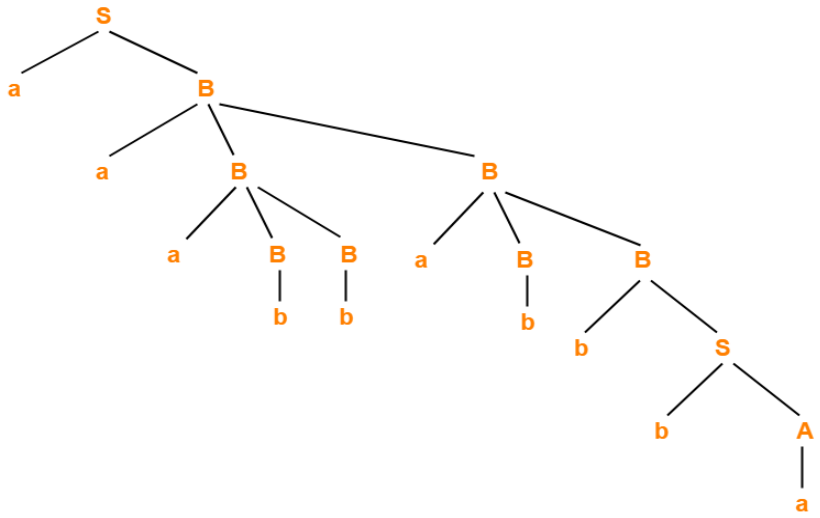


Fig. 1: A Typical Parse Tree

From the figure, if we consider all leaf nodes from leftmost side, we get our string 'aaabbabbba'. So, we can derive strings from any language easily using parse tree that is why it is known as derivation tree.

CHECK YOUR PROGRESS-II

4: What do you mean by leftmost and rightmost derivations?

5: What is multiple steps derivation?

6: What is parse tree? Draw parse tree for the string 'aaaa' for the following CFG:

S → aS

S → ε∈

Space for learners:

4.4.3 Ambiguous Grammar

A grammar G is ambiguous if there is a word w ∈L (G) having are least two different leftmost or rightmost derivations. Simply, for a string in a Context Free Grammar (CFG), more than one leftmost derivation and more than one rightmost derivation exist. For ambiguous grammar, there will be two or more sparse trees for a string. Let's figure this out with an example:

Suppose our grammar is:

$E \rightarrow E + E \mid E * E \mid (E) \mid N$

$N \rightarrow 1N \mid 2N \mid 1 \mid 2$

This is one of arithmetic operation type grammar, where we are using terminals like +, |, * etc. symbols. Let's try to draw parse tree for the string 1 + 2 * 2

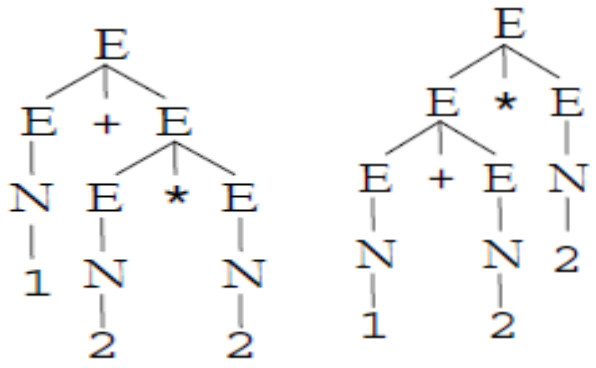


Fig. 2: Two different parse tree for same string

Since for a string, the grammar has more than one parse tree, hence this grammar is an ambiguous grammar. Additionally, from the figure 2, if we calculate parse tree derivations from arithmetic point of view, then left parse tree value and right parse tree value will be 5 and 6 respectively. That is why ambiguity in grammar increases difficulties for parser exponentially.

Let us discuss another example -

Check whether the given grammar is ambiguous or not-

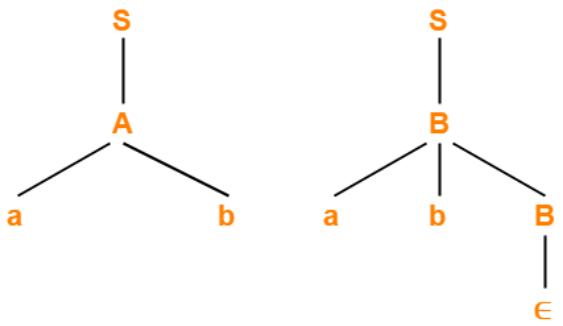
Space for learners:

S → A | B

A → aAb | ab

B → abB |ε

Now, let us draw parse trees for this string ab –



Given grammar is ambiguous because two different parse trees exist for string ab.

4.5 SIMPLIFYING CONTEXT FREE GRAMMARS

While preparing context free grammar, we tend to write some unnecessary redundant productions because CFG allows us to develop a wide variety of grammars. That is why all the grammars are not always optimized i.e. grammar may consists of some useless symbols or productions. Simplification of CFG means reduction of grammar by removing unnecessary productions, while keeping the transformed grammar equivalent to the original grammar. Two grammars are called equivalent if they produce the same language.

4.5.1 Types of Redundant Productions

Useless productions: Productions which do not take part in the derivation of any string. Same is applicable for symbol or variable in context free grammar. Consider the following grammar –

S → aaB| aaS

B → ab | b

Space for learners:

E → ad

Production E → ad will never come in the derivation of any string because it is not reachable from the starting variable S.

Null Productions: The productions of type P → ε are called null productions or ε productions (also called lambda productions). Null productions or ε productions are frequently used to develop context free grammar.

S →ABCd

A → BC

B →bB | ε

C →cC | ε

Productions B → ε and C → ε are both null productions and ε productions.

Unit Productions: The productions of type P → Q are called unit productions. Simply, the production where a non-terminal implies another non terminal is known as unit productions. Consider the following grammar –

S → 0A | 11 | C

A → 0S | 00

C → 01

Production S → C is a unit production in the above grammar.

4.5.2 Elimination of Useless Productions

We have studied about useless productions. Let's try to understand how to eliminate useless productions from a context free grammar with a proper example -

T → aaB | abA | aaT

A → aA

B → ab | b

C → ad

Space for learners:

In the example, the production C → ad is useless, because C is not reachable from S, so it will never occur in the derivation of any string. So, we will eliminate it.

Production A → aA is also useless because we don't have any way to terminate it. If a production never terminates, then it can never produce a string. To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'A' occurs. So, after removing useless symbols and productions, grammar will be –

T → aaB | aaT

B → ab | b

4.5.3 Elimination of Null Or E Productions

We have studied about null or ε productions. For removing null productions from the grammar, we need to do -

Step 1: Find out all non-terminal variables which derives ε, those non terminals are also known as nullable variables.

Step 2: For each production, which contains nullable variables, construct new productions by replacing nullable variables.

Step 3: Combine productions of step 2 with the original productions and remove ε productions.

Consider the following grammar:

S → XYX

X → 0X | ε

Y → 1Y | ε

We need to remove the production rules X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared, so we need to check every possibility while removing ε.

Space for learners:

S → XYX

If the first X at right-hand side is ε and the last X at right-hand side is ε, then we can write

S → YX

S → XY

If Y = ε then

S → XX

If Y and X are ε then,

S → X

If both X are replaced by ε

S → Y

Now, S → XY | YX | XX | X | Y

Let's take another production rule,

X → 0X

If we place ε at right-hand side for X then,

X → 0

X → 0X | 0

Similarly in case of last production rule,

Y → 1Y | 1

So, after removing null productions, our CFG will look like –

S → XY | YX | XX | X | Y

X → 0X | 0

Y → 1Y | 1

CHECK YOUR PROGRESS-III

7: What is ambiguous grammar?

8: Why we need to simplify context free grammar? Explain.

9: What are the different types of redundant productions?

10: Check the following CFG is ambiguous or not?

S → aSb | SS, S → ε

Space for learners:

4.5.4 Elimination of Unit Productions

For removing unit productions i.e., productions of type X → Y, we need to follow -

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

Considering the following grammar:

S → 0A | 1B | C

A → 0S | 00

B → 1 | A

C → 01

In the above example, S → C is a unit production, while removing S → C we have to consider what C implies. Depending on that, we can add a rule to S.

S → 0A | 1B | 01

In the above example, B → A is also a unit production-

B → 1 | 0S | 00

Thus, finally our CFG without unit production is –

S → 0A | 1B | 01

A → 0S | 00

B → 1 | 0S | 00

C → 01

4.6 SUMMING UP

- Context free languages are the languages which are specified by context free grammars.
- Context free grammar is developed to address a complex set of languages, as we have studied it's a 4-tuple grammar.
- For obtaining a string from a CFG, derivations techniques we need such as leftmost derivations, rightmost derivations etc...

Space for learners:

- Parse tree is a geometrical representation of derivation of a string, if for a particular string, there is more than one parse tree then the corresponding grammar is ambiguous.
- We have to eliminate the ambiguity nature of the grammar.
- Sometimes parser faces problem because of the unnecessary productions present in context free grammar that is why we need to check for unnecessary productions in a grammar and if present we have to remove those productions.

Space for learners:

4.7 ANSWERS TO CHECK YOUR PROGRESSES

1. A context free grammar (CFG) is a 4-tuple (V, **Σ**, R, S) grammar, where V is a set of non-terminals (NT) are also called variables, Σ is an alphabet, characters in the alphabet are known as terminals and S is the starting variable. R is a set of production or substitution rules that represents the recursive definition of the language.

2. Given CFG language is $L = \{0^n1^n \mid n \geq 1\}$.

The string that can be generated for a given language is {01, 0011, 000111, 00001111,….}

Production rules for the grammar can be –

$S \rightarrow 0S1$

$S \rightarrow 01$

From these production rules, we can derive any string of {01, 0011, 000111, 00001111…}. Suppose for example, String '000111' can be derived as-

$S \rightarrow 0S1$

$S \rightarrow 00S11$ Using first production rule

$S \rightarrow 000111$ Using second production rule

3. Closure properties of Context free languages are –

Union: Context-free languages are closed under union operation i.e. that if X and Y are both context-free languages, then X∪Y is also a context-free language.

Concatenation: Context-free languages are closed under concatenation operation i.e., that if X and Y are both context-free languages, then XY is also a context-free language.

Kleene Star: Context-free languages are closed under concatenation operation i.e., that if L is a context free language, then L* is also a context free language.

Unlike regular languages, Context free languages are not closed under intersection or complement operation.

4. A leftmost derivation of a sentential form is one in which rules transforming the leftmost non-terminal is always applied. A rightmost derivation of a sentential form is one in which rules transforming the rightmost non terminal is always applied.

5.In multiple steps derivation, u derives v, i.e., $u \Rightarrow^{*} v$, if there is a chain of one step derivations in the form: $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow u_3 \Rightarrow u_4 \Rightarrow u_5 \ldots\ldots \Rightarrow v$

6. A parse tree is a geometrical representation of derivations in which:

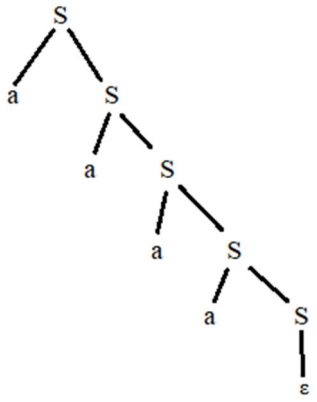
- Each internal node is labeled with a non-terminal symbol.
- Root node of a parse tree is the start symbol of the grammar.
- Each leaf node is labelled with a terminal symbol.
- If a rule T→ T1T2…Tn occurs in the derivation then T is a parent node of nodes labelled T1, T2, …, Tn

We need to draw a parse tree for a string 'aaaa' for a given grammar:

$S \rightarrow aS$

$S \rightarrow \varepsilon$

So, parse tree is -



Space for learners:

7. For a string x in a Context Free Grammar (CFG), if there exist more than one leftmost derivation or rightmost derivations, then it is a ambiguous grammar.

8. Simplification of CFG means reduction of grammar by removing unnecessary productions, while keeping the transformed grammar equivalent to the original grammar. Simplification is required because all the grammars are not always optimized i.e. grammar may consists of some redundant symbols or productions.

9. Redundant productions are –

Useless productions: Productions or symbols which do not take part in the derivation of any string.

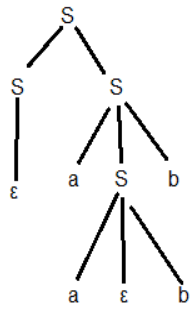
Null Productions: The productions of type P → ε are called null productions or ε productions (also called lambda productions). Null productions or ε productions are frequently used to develop context free grammar.

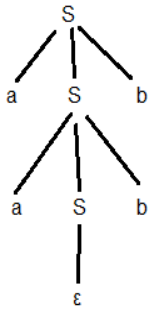
Unit Productions: The productions of type P → Q are called unit productions.

10. We have given following grammar:

S → aSb | SS

S → ε





So, for the string 'aabb', there are two parse trees, hence given CFG is an ambiguous grammar.

Space for learners:

4.8 POSSIBLE QUESTIONS

1. What is Context Free Language?
2. Define Context free grammar. Write some applications of it.
3. Explain the concept of parse tree with suitable example.
4. Explain Closure properties of context free languages.
5. Discuss various derivation techniques of CFG.
6. What are the three ways to simplify a context free grammar?
7. Discuss the simplification ways of Context free grammar with examples.
8. Eliminate useless productions from the following grammar:

   $T \rightarrow abA \mid aaT$

   $A \rightarrow aA$

   $C \rightarrow ad$
9. Check the following grammar is ambiguous or not:

   $E \rightarrow E + E \mid E * E \mid (E) \mid id$
10. Construct CFG without Є production from the grammar:

    S →a | Ab | aBa, A →b | Є, B →b | A.

Space for learners:

4.9 REFERENCES AND SUGGESTED READINGS

- Introduction to Automata Theory, Languages and Computation, *John E Hopcroft, Matwani & Jeffery D. Ullman*
- Introduction to Languages and the Theory of Computation, *John C. Martin*
- Elements of the Theory of Computation, *Lewis & Papadimitriou*
- http://infolab.stanford.edu/
- https://www.geeksforgeeks.org/
- https://www.gatevidyalay.com/
- https://www.cs.wcupa.edu/
- https://www.javatpoint.com/

UNIT 5: PDA AND CHOMSKY NORMAL FORMS

Space for learners:

Unit Structure:

5.1 INTRODUCTION

In this unit we will study thoroughly about pushdown automata, normal forms of context free grammar and lastly the pumping lemma for context free languages. Finite automata cannot able to implement complex problems, that is why pushdown automata comes with an additional element called stack. Pushdown automata can implement a context free languages and stack is used for different mechanisms and memorizing purpose. We can design pushdown automata for any context free languages. In last chapter, we have studied about simplification of CFG, we will study another advanced related topic in this chapter, which is Normal forms of CFG. Normal forms deal with certain rules or forms of writing productions in the grammar. Lastly, we will study the pumping lemma for context free languages, applying which we can find out a language is context free or not.

5.2 UNIT OBJECTIVES

This unit covers Pushdown automata, Normal forms of context free grammar and pumping lemma of Context free languages. After going through this unit you will be able to:

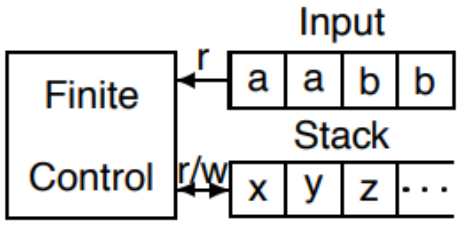
- How Pushdown automata work?
- Designing Pushdown automata for any context free languages.
- Explain about normal forms of context free grammar.
- Convert a context free grammar into its Chomsky's Normal Form (CNF).
- Discuss about Chomsky's Normal Form (CNF) and Greibach Normal Form (GNF)
- Convert a context free grammar into its Greibach Normal Form (CNF).
- Discuss the pumping lemma for context free languages.
- Check a language is context free language or not.

Space for learners:

5.3 PUSHDOWN AUTOMATA (PDA)

Space for learners:

Just as we design DFA for a regular grammar, pushdown automata (PDA) is a way to implement a context free grammar. Pushdown Automata are new type of computational model, which is like finite automata but have an extra memory component called stack. Stack allows PDA to recognize some complex languages that is why A PDA is more powerful than finite automata (FA). A language which can be acceptable by finite automata (FA) can also be acceptable by pushdown automata (PDA). We can visualize a PDA like-



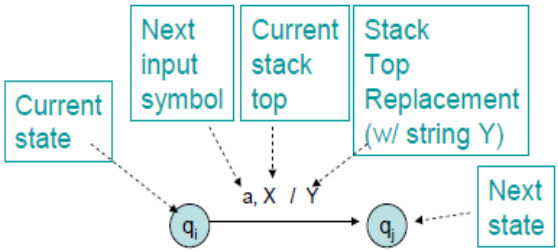
PDA reads input symbol from alphabet and it can read/write to stack. It makes transitions based on input symbol and top of stack.

Formally, a PDA can be defined by 7-tuple (Q, ∑, Γ, δ, q_0, Z_0, F), where:

- Q is the finite number of states
- ∑ is the finite set of input symbols, the alphabet
- Γ is the finite set of stack symbols, symbols which are allowed to push/pop into the stack
- q_0 is the initial state of PDA
- Z_0 is the initial stack symbol
- F is the set of final states
- δ is a transition function: Q x {Σ ∪∈} x Γ → Q x Γ*, i.e., PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.

5.3.1 PDA as a State Diagram

$\delta(q_i, a, X)=\{(q_j, Y)\}$



5.3.2 Instantaneous Description

An instantaneous description of PDA is described by a triple (q, w, α) where:

q is the current state.

w is the unconsumed input.

α is the stack contents.

For transition purpose, we use turnstile notation (⊢ sign), which represents one move of PDA. And for multiple moves, we use ⊢* sign.

For example,

(q, aw, Xβ) ⊢ (p, w, αβ)

In the above example, we took a transition such that we went from state q to p, we consumed input symbol a, and we replaced the top of the stack X with some new string α.

5.3.3 Examples of PDA

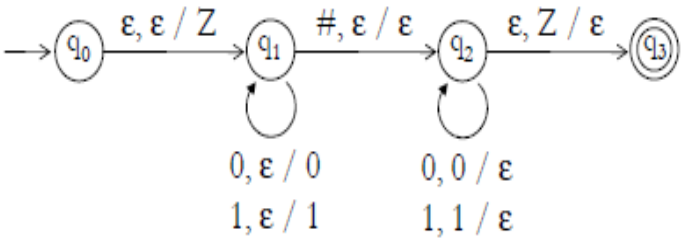
Example 1: Design a PDA for the language L = $\{w\#w^R: w \in \{0, 1\}^*\}$

Solution: From the given language, we can say our strings will look like #, 0#0, 01#10, 0110#0110 etc.

Space for learners:

We can design the PDA using state diagrams only. For better understanding of PDA, we will use state diagrams for constructing PDA, because state diagrams are best mathematical tools for designing PDA.

Space for learners:



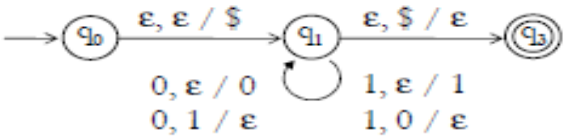
For constructing PDA for a language, we need to think different mathematical mechanisms. There can be numerous numbers of mechanisms for designing a particular PDA. Here we designed this PDA with the mechanism – 'write w on stack and read w^R from the stack' . In this PDA, we are assuming our stack alphabet is {0, 1} and initial stack symbol is Z. So, from q_0 to q_1, we just push a Z into the stack and then at the state q_1, we are pushing 1's and 0's for input symbol 1's and 0's, that means we are writing 'w' at q_1, then by consuming input symbol #, we reached at q_2, here we are popping 1's and 0's for input symbol 1's and 0's. Since our language is of type $w\#w^R$, that is why first we pushed one part and secondly, we popped the other part. If any string is not in type of $w\#w^R$, then machine will never go to the final sate. Hence, we designed our PDA for the given language.

Example 2: Design a PDA for the language L = {w: w has same number of 0's and 1's}

Solution: From the given language, we can say our strings will look like 01, 0110, 011100, 001110 etc., so our input alphabet will be Σ = {0,1}.

As we have seen in the earlier example, we need to develop a mechanism for constructing PDA. Suppose, our stack is keeping track of number of 0's and 1's in the string and if we pop 1 for consuming input symbol 0 and vice versa and finally at the end, if we find our stack

is empty, then we can easily say number of 0's and 1's are equal. So, let's construct it using state diagram -



For the above PDA, let's check membership of a string for the given language. Suppose our string is w= 001110, now for each consumed input symbol, our stack contents will be –

Input symbol	Stack contents
0	$0
0	$00
1	$0
1	$
1	$1
0	$

So, finally PDA's stack is empty, it will move to the state q_3 and which is a final state. Hence, this string is accepted.

CHECK YOUR PROGRESS-I

1: What are Pushdown automata?

2: What is the transition function of PDA?

3: Can we construct a PDA without its state diagram?

4: Construct a PDA for the language L = $\{w\#w^R: w \in \{0,1\}^*\}$

Space for learners:

5.4 NORMAL FORMS

Generally, it's easier to work with context free grammar when it is in normal forms. While parsing in computer, sometimes CFG causes lots of problem such as redundant loops, infinite loops etc. that is why normal forms are often convenient to simplify CFG. There are mainly two normal forms, these are:

5.4.1 Chomsky Normal form (CNF)

A Context free Grammar G is in Chomsky Normal Form where every production is either of the form:

A→BC

A→a

where a is a terminal and A, B, C are non-terminals.

E.g., consider the following grammar G

S → AB

S → c

A → a

B → b

Production rules of Grammar G are in the forms of CNF, so grammar G is in CNF.

When a CFG is not in the form of Chomsky's Normal Form (CNF), then we need to convert it. The conversion requires some easy steps, which are –

Step 1: Remove the start symbol from RHS of production. If the start symbol S is at the right-hand side of any production, create a new production as:

S1 → S, where S1 is the new start symbol.

Step 2: Remove null, useless and unit productions if needed.

Space for learners:

Step 3: Replace terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, the production X → aP can be written as:

X → QP

Q → a

Step 4: Productions which are having more than two non-terminals, change it in the form A→BC.

For example, S → ASB can be decomposed as:

S → QS

Q → AS

Example: Consider the following grammar:

S → ASB

A → aAS|a|ε

B → SbS|A|bb

We need to convert this grammar to its CNF form. So according to step 1, this grammar has start symbol in the RHS, we need to remove them.

S1 → S

S → ASB

A → aAS|a|ε

B → SbS|A|bb

Now, from step 2 we need to simplify our CFG by removing null, unit and useless productions, and this grammar has null productions –

S1 → S

S → ASB|SB

A → aAS|aS|a

B → SbS| A|ε|bb

So, it creates a new null production B → ε, we need to remove it –

S1 → S

Space for learners:

S → AS|ASB| SB| S

A → aAS|aS|a

B → SbS| A|bb

Now, it creates unit production B->A

S1 → S

S → AS|ASB| SB| S

A → aAS|aS|a

B → SbS|bb|aAS|aS|a

Now, another unit production is S1 → S is there –

S1 → AS|ASB| SB| S

S → AS|ASB| SB| S

A → aAS|aS|a

B → SbS|bb|aAS|aS|a

Again, S1 → S and S → S exists, after removing them –

S1 → AS|ASB| SB

S → AS|ASB| SB

A → aAS|aS|a

B → SbS|bb|aAS|aS|a

Now, applying rule of step 3 in the production rule A->aAS |aS and B->SbS|aAS|aS

S1-> AS|ASB| SB

S → AS|ASB| SB

A → XAS|XS|a

B → SYS|bb|XAS|XS|a

X →a

Y→b

In the fourth production, B->bb can't be part of CNF

S1-> AS|ASB| SB

Space for learners:

S → AS|ASB| SB

A → XAS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

Now, according to step 4, in the production rule S1->ASB, we will get

S1-> AS|PB| SB

S → AS|ASB| SB

A → XAS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Similarly, we will do the needful for the production S->ASB,

S1-> AS|PB| SB

S → AS|QB| SB

A → XAS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

Again, for the production A->XAS,

S1-> AS|PB| SB

S → AS|QB| SB

Space for learners:

A → RS|XS|a

B → SYS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

R → XA

Again, for B->SYS,

S1 -> AS|PB| SB

S → AS|QB| SB

A → RS|XS|a

B → TS|VV|XAS|XS|a

X → a

Y → b

V → b

P → AS

Q → AS

R → XA

T → SY

Lastly for B->XAX, Now the grammar will look like –

S1-> AS|PB| SB

S → AS|QB| SB

A → RS|XS|a

B → TS|VV|US|XS|a

X → a

Y → b

V → b

Space for learners:

P → AS

Q → AS

R → XA

T → SY

U → XA

So, this grammar satisfies the conditions of Chomsky's Normal form (CNF), Hence the grammar is in CNF.

5.4.2 Greibach Normal Form (GNF)

Context free Grammar G is in Greibach Normal Form(GNF) where every production is of the form:

A → aα

where, a is a terminal and α consists of any number of non-terminals and if ε is in the language, then we will allow the rule S → ε.

For example, consider a grammar G –

S → aAB | aB

A → aA| a

B → bB | b

In the grammar G, every production is in the form of A → aα, hence grammar G is in GNF.

If we need to convert any context free grammar into its Greibach normal form (GNF), then –

Step 1: Convert given context free grammar into CNF. (Since we have studied this part earlier, so we will use this approach. There are some alternative approaches are available for GNF conversion)

Step 2: If CFG contain left recursions, then remove them. (A production of context free grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS. E.g., S → Sa)

Space for learners:

Step 3: Finally convert the production rules into GNF.

Example: Consider the following grammar -

S → XA|BB

B → b|SB

X → b

A → a

Step 1: We need to convert the grammar into CNF but every production of the grammar is in CNF. So, let's move to step 2.

Step 2: There is no left recursion in the grammar, so we can convert this grammar to GNF.

Step 3: We need to check for the productions that are not in GNF, then we will convert it one by one.

The production rule B->SB is not in GNF-

S → XA|BB

B → b|XAB|BBB

X → b

A → a

So, we substituted S -> XA|BB in production rule B->SB.

The production rules S->XA and B->XAB is not in GNF -

S → bA|BB

B → b|bAB|BBB

X → b

A → a

So, we substituted X->b in production rules S->XA and B->XAB.

Now, B->BBB production is a left recursive production, we need to remove that-

S → bA|BB

B → bC|bABC

C → BBC| ε

Space for learners:

X → b

A → a

We got another problem, because C-> ε is a null production, after removing this -

S → bA|BB

B → bC|bABC|b|bAB

C → BBC|BB

X → b

A → a

The production rules S->BB is not in GNF -

S → bA| bCB|bABCB|bB|bABB

B → bC|bABC|b|bAB

C → BBC|BB

X → b

A → a

So, we substituted B → bC|bABC|b|bAB in production rules S->BB.

The production rules C->BB is not in GNF-

S → bA| bCB|bABCB|bB|bABB

B → bC|bABC|b|bAB

C → BBC

C → bCB|bABCB|bB|bABB

X → b

A → a

So, we substituted B → bC|bABC|b|bAB in production rules C->BB.

The production rules C->BBC is not in GNF -

S → bA| bCB|bABCB|bB|bABB

B → bC|bABC|b|bAB

C → bCBC|bABCBC|bBC|bABBC

Space for learners:

C → bCB|bABCB|bB|bABB

X → b

A → a

So, we substituted B → bC|bABC|b|bAB in production rules C->BBC.

Now, finally every production of this grammar is in greibach normal form (GNF).

5.5 PUMPING LEMMA OF CONTEXT FREE LANGUAGES

Just like regular language's pumping lemma, we can use pumping lemma for Context free languages to check a language is context free or not. Unlike regular languages, in the case of CFL pumping lemma, we break its strings into five parts and pump second and fourth substring. Pumping lemma for Context free languages is -

For every context free language, L, there exists a number n such that for every string z in L, we can write z = uvwxy, where-

1. $|vwx| \leq n$
2. $|vx| \geq 1$
3. For every $i \geq 0$, the string uv^iwx^iy is in L.

For example, suppose a language $L = \{a^nb^nc^n \mid n \geq 0\}$, we need to check the language is CFL or not.

We have studied in the regular languages, whenever we are applying pumping lemmas, we try to show a contradiction which implies this language is not belongs to this class.

Lets there exists a positive integer number n, lets our string is $a^nb^nc^n$. Lets divide it into five parts such that z = uvwxy, considering pumping lemma's conditions $|vwx| \leq n$ and $|vx| \geq 1$

$u=a^n$, $v=b^{n/3}$, $w=b^{n/3}$, $x=b^{n/3}$,$y= c^n$

Now, for i=0,

$uv^iwx^iy = uv^0wx^0y = a^n(b^{n/3})^0b^{n/3}(b^{n/3})^0c^n$

$= a^n\, b^{n/3}c^n \notin L$

Space for learners:

So, it's a simple contradiction, Hence the language is not a context free language.

CHECK YOUR PROGRESS-II

5: What is CNF?

6: What is GNF?

7: State the pumping lemma of context free languages?

5.6 SUMMING UP

- Pushdown Automata (PDA) is a way of constructing context free grammar just like finite automata for regular grammar. Only difference is extra memory element stack here, stack allows PDA to recognize some complex languages that is why A PDA is more powerful than finite automata (FA).
- By using PDA state diagram or instantaneous description we can construct PDA for any context free language.
- A Context free Grammar G is in Chomsky Normal Form where every production is like --- A→BC, A→a, where a is a terminal and A, B, C are non-terminals.
- If the productions are like - A→ aα, where a is a terminal and α consists of any number of non-terminals, then it is said to be is in Greibach Normal Form.
- There are steps to convert a grammar into its CNF, GNF.
- Pumping lemma for context free languages, which is quite similar to the regular language's pumping lemma and it's an easy way to check whether a language is context free or not.

Space for learners:

5.7 ANSWERS TO CHECK YOUR PROGRESSES

Space for learners:

1. Just as we design DFA for a regular grammar, the pushdown automata (PDA) is a way to implement a context free grammar. Pushdown Automata are new type of computational model, which is like finite automata but have an extra memory component called stack. Formally, a PDA can be defined by 7-tuple (Q, ∑, Γ, δ, q_0, Z_0, F), where –

- Q is the finite number of states
- ∑ is the finite set of input symbols, the alphabet
- Γ is the finite set of stack symbols, symbols which are allowed to push/pop into the stack
- q_0 is the initial state of PDA
- Z_0 is the initial stack symbol
- F is the set of final states
- δ is a transition function: Q x {Σ ∪∈} x Γ → Q x Γ*.

2. Transition function of PDA defines the mappings of state to state, which is denoted by δ that implies a PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack and mathematically written as Q x {Σ ∪∈} x Γ → Q x Γ*, where Q is the finite number of states, ∑ is the input alphabet and Γ is the finite set of stack symbols of PDA.

3. Yes, we constructed PDA using state diagrams because it's an easy way to construct a PDA. We can construct PDA by using instantaneous description and turnstile symbol as well, where we need to write each and every moves of your PDA. By seeing these moves, one can easily understand the working principle of designed PDA.

4. Refer section no. 4.3.3.

5. A Context free Grammar G is in Chomsky Normal Form where every production is either of the form: A→BC, A→a

Where a is a terminal and A, B, C are non-terminals and if ε is in the language, then we will allow the rule S → ε.

6. Context free Grammar G is in Greibach Normal Form (GNF) where every production is of the form: $A \rightarrow a\alpha$

Where a is a terminal and α consists of any number of non-terminals and If ε is in the language, then we will allow the rule $S \rightarrow \varepsilon$.

7. Pumping lemma of Context free language is –

For every context free language L, there exists a number n such that for every string z in L, we can write z = uvwxy, where-

1. $|vwx| \leq n$
2. $|vx| \geq 1$

For every $i \geq 0$, the string uv^iwx^iy is in L.

5.8 POSSIBLE QUESTIONS

1. Define Pushdown Automata (PDA).
2. Why Pushdown Automata is more powerful as compared to finite automata?
3. What are the normal forms in CFG?
4. State the pumping lemma for context free languages?
5. Design a PDA for the language L = {w: w has same number of 0's and 1's}.
6. Design a PDA for accepting a language $\{0^n1^m0^n \mid m, n>=1\}$.
7. Convert the following context free grammar into its CNF:

   S->a

   S->aZ

   Z->a
8. Convert the following context free grammar into its CNF:

   $S \rightarrow aXbX$

   $X \rightarrow aY \mid bY \mid \varepsilon$

   $Y \rightarrow X \mid c$
9. Convert the following context free grammar into its GNF:

   S -> BA

Space for learners:

B -> b | SB

A -> a

10. Using Pumping lemma, check L = {ww | w∈{0,1}*} is context free or not.

Space for learners:

5.9 REFERENCES AND SUGGESTED READINGS

- Introduction to Automata Theory, Languages and Computation, *John E Hopcroft, Matwani& Jeffery D. Ullman*
- Introduction to Languages and the Theory of Computation, *John C. Martin*
- Elements of the Theory of Computation, *Lewis & Papadimitriou*
- Examples- https://www.geeksforgeeks.org/
- Javatpoint - https://www.javatpoint.com/