Automata

Automata theory is the study of abstract computing devices or "machines".

Or

It is a branch of computer science that deals with designing abstract self propelled computing devices that follow a predetermined sequence of operations automatically.

Self Propelled - able to move by its own power.

An automaton with a finite number of states is called a Finite Automaton.

Automata - what is it?

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

Why Study Automata Theory?

Finite automata are a useful model for many important kinds of hardware and software. Some of the most important kinds which applies automata theory.

1] Software for designing and checking the behavior of digital circuits.

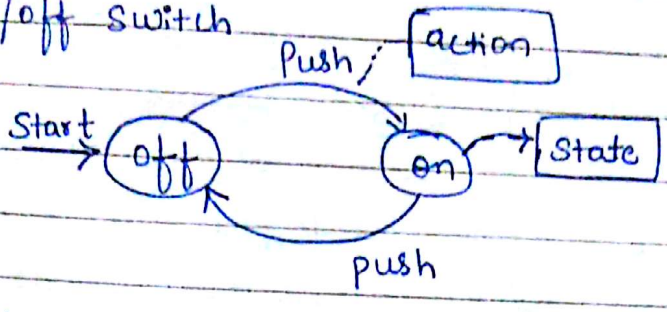
2] The "lexical analyzer" of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuation.

3] Software for scanning large bodies of text, such as collections of web pages, to find occurences of words, phrases, or other patterns.

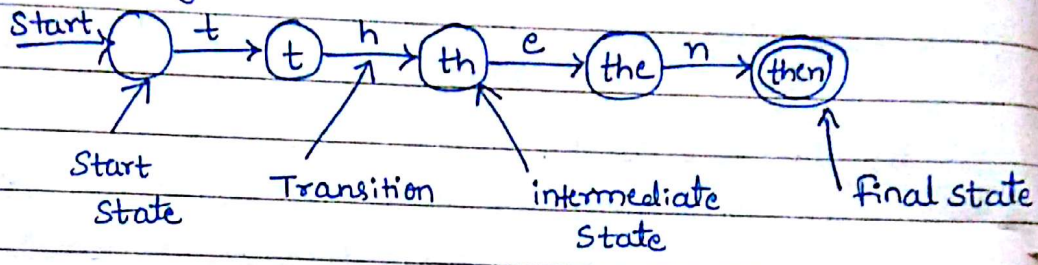
4] Software for verifying systems of all types that have a finite number of distinct states, such as communications protocals or protocals for secure exchange of information.

Finite Automata : Examples

- on/off switch



- Modelling recognition of the word "then".



Structural Representations:

There are two important notations play an important role in the study of automata and their applications.

1] Grammars: These are useful models when designing software that processes data with recursive structure. The best known example is a "parser", the component of a compiler that deals with the recursively nested features of the typical programming language, such as expressions - arithmetic, conditional, and so on.

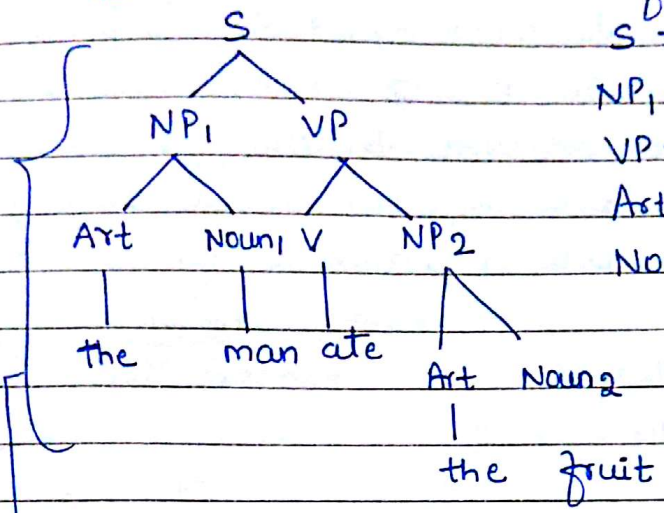
For instance, a grammatical rule like $E \Rightarrow E+E$ states that an expression can be formed by taking any two expressions and connecting them by a plus sign; this rule is typical of how expressions of real programming languages are formed.

2] Regular Expressions: These also denote the structure of data, especially text strings.

Grammars :- 1959 — N. chomsky.

Parsing - He look at the parse tree of natural language sentences and tried to define what is a grammar.

Ex:- The man ate the fruit.



S — sentence
NP_1 - Noun phrase
VP - verb phrase
Art - Article.
$Noun_1$ and $Noun_2$ are nouns.

→ Parse tree (syntax tree)

↓

derivation tree

* Sentense Symbol to Sentense — Top down down approach - Derivation Tree
  S to Sentence (the man ate the fruit)
* Sentence to Sentense Symbol — Bottom up approach - parse tree.
  Sentence to S

Production rules can be defined from the parse tree.

$\langle S \rangle \rightarrow \langle NP_1 \rangle \langle VP \rangle$

$\langle NP_1 \rangle \rightarrow \langle Art \rangle \langle Noun_1 \rangle$

$\langle Art \rangle \rightarrow$ The

$\langle Noun_1 \rangle \rightarrow$ man

$\langle V \rangle \rightarrow$ ate

$\langle NP_2 \rangle \rightarrow \langle Art \rangle \langle Noun_2 \rangle$

$\langle Art \rangle \rightarrow$ The

$\langle Noun_2 \rangle \rightarrow$ fruit.

$\langle S \rangle \Rightarrow \langle NP_1 \rangle \langle VP \rangle$

$\Rightarrow \langle Art \rangle \langle N_1 \rangle \langle VP \rangle$

$\Rightarrow$ The $\langle N_1 \rangle \langle VP \rangle$

$\Rightarrow$ The man $\langle V \rangle \langle NP_2 \rangle$

$\Rightarrow$ The man ate $\langle Art \rangle \langle N_2 \rangle$

$\Rightarrow$ The man ate the $\langle N_2 \rangle$

$\Rightarrow$ The man ate the fruit.

Using production rules we can derive sentence. We can use only one rule at a time during derivation of a sentence.

$\rightarrow$ rewritten as

$\Rightarrow$ directly derives

$V, NP_2, VP, Art, Noun_1, Noun_2$ are rewritten as something else - so, these are called as non-terminals.

The, man, ate, the, fruit - These, we cannot rewrite as something else - So, these are calleds as terminals.

Sentential form (sequential form)

↓ strings over terminals and non-terminals.

Ex:- $\alpha \Rightarrow \beta$

we will get β by using rule.

Total alphabet - can be defined as the union of non-terminals and terminals.

$V = NUT.$

S → start symbol → where production begins.

$S \Rightarrow aSb$

↓ start symbol.

Grammar = (N, T, P, S)

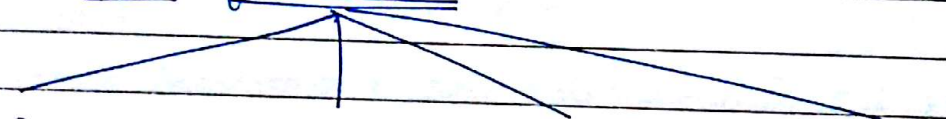
N – finite set of non-terminals
T – finite set of terminals
P – finite set of production rules
S – Start symbol

Two types of derivation :-
→ Rightmost derivation
→ leftmost derivation

Rightmost Derivation - If we replace rightmost nonterminal by terminal then it is rightmost derivation.

Leftmost Derivation - If we replace leftmost non-terminal by terminal then it is called as leftmost derivation.

Types of Grammar



Type 0
Unrestricted
phrase structured
grammar

Type 1
Context
sensitive
grammar

Type 2
Context
free
grammar

Type 3
Regular
grammar

Right linear

Left linear

→ No restrictions
→ Most powerful grammar

$\alpha \rightarrow \beta$

$\alpha \in (V+T)^+$

α belongs to any combination of variables and terminals
α should not have empty string
i.e $\in$ (epsilon)

Type 0 :- (Unrestricted phrase structured Grammar)

→ As the name itself tells, this grammar is unrestricted means this grammar have no restrictions.

production rule of the form:

$$\alpha \rightarrow \beta$$

$\alpha \in (V+T)^+$ $(\alpha \in V^*NV^*)$

$\beta \in (V+T)^*$ ↑ atleast one nonterminal

$\alpha \in (V+T)^+$

α should have ϵ (epsilon).

On β side we can have ϵ. $[(V+T)^*]$

Eg:- $aAb \rightarrow bB$

$aA \rightarrow \epsilon$.

In this example we have, ϵ on β side.

→ This grammar is most powerful grammar.

Type 1 :- (context sensitive grammar)

If we put restrictions on Type 0 grammar then we can derive Type 1 grammar.

→ This grammar has a restriction on length of α and β.

$$\alpha \rightarrow \beta$$

$$|\alpha| \leq |\beta|$$

$\alpha, \beta \in (V+T)^+$

Eg:- $aAb \rightarrow bbb$

$aA \rightarrow bbb$

$aAb \rightarrow bb$ This rule is not allowed.

here, $|\alpha|$ is not less than or equal to $|\beta|$. But this type of rule allowed in Type 0.

Type 2 :- (Context free grammar)

If we apply more restrictions on Type 1 grammar, then we can derive Type 2 grammar.

$$A \rightarrow \alpha$$

What is context ?

$$aAb \longrightarrow bb$$

left context of A

Right context of A.

But in the CFGs we write production as →

$$V \longrightarrow (V+T)^*$$

$$\Downarrow$$

$$\in A \in$$

no context

means left and right context of this variable is $\in$. thats why these are known as context free grammar.

$$A \rightarrow \alpha$$

In this production rule, we donot have left and right Context. But when we take Type 1 grammar we have both left and right context.

$$\alpha \in (V+T)^*$$

E.g:- $A \rightarrow \in$

$A \rightarrow BCD$

(can have many variables at RHS side)

Type 3 :- (Regular Grammar)

Right Linear | Left Linear

$A \rightarrow xB \mid x$

⇓ Right Linear

$A \rightarrow Bx \mid x$

⇓ Left Linear

$A, B \in V$

$x \in \Sigma^*$

Regular grammar :-

$S \rightarrow as \mid b$

$S \rightarrow aS \mid c$

$S \rightarrow Sa \mid b$

$A \rightarrow \in$

$A \rightarrow ba$

} R.Gr can have atmost one variable at RHS.

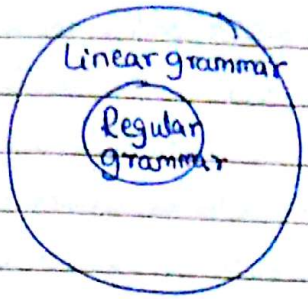
Linear Grammar:- In any grammar if there exist exactly one variable on the LHS and atmost one variable on RHS, is known as linear grammar.

following are example of the linear grammar

$S \rightarrow aSb \mid bSa \mid \in$ → middle linear

$A \rightarrow aB \mid b$ → left linear

$S \rightarrow as \mid bS \mid a$



Regular grammar = left linear, Right linear

Linear grammar = middle linear
left linear
Right linear.

So, Regular grammars are subset of Linear grammars

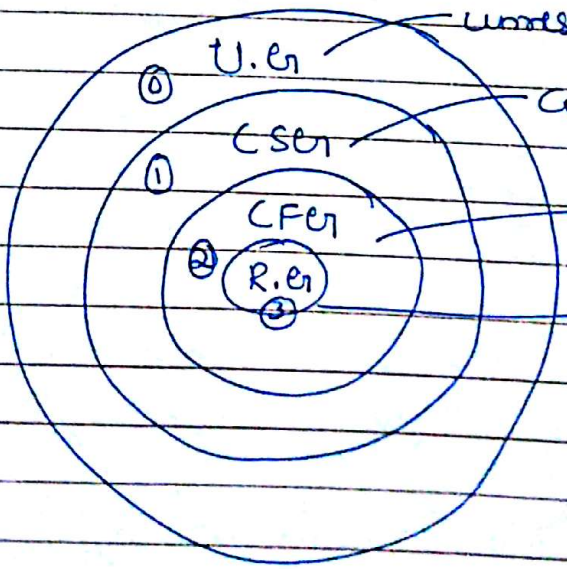
If a ~~linear~~ Linear grammar has both left and right linear grammar then it is not regular grammar.

Eg:- $S \to aS$
$S \to Sb$
$S \to a$
} Linear but not regular. because It contains both left and right linear grammar.

$S \to aSb \mid ab$ } Linear but not regular. because in regular grammar we should not have middle linear grammar.

CHOMSKY HEIRARCHY

Type 3 ⊂ 2, 1, 0 — Type 3 is the subset of 2, 1, 0
Type 2 ⊂ 1, 0 — Type 2 is the subset of 1, 0
Type 1 ⊂ 0 — Type 1 is the subset of 0.



If we put restrictions on Unrestricted, we will get Context Sensitive grammar.

If we put restrictions on context sensitive, we get Context free grammar.

If we put restrictions on context free grammar, we get regular grammar.

If grammar is Type 3 then it will be Type 0, Type 1, Type 2.

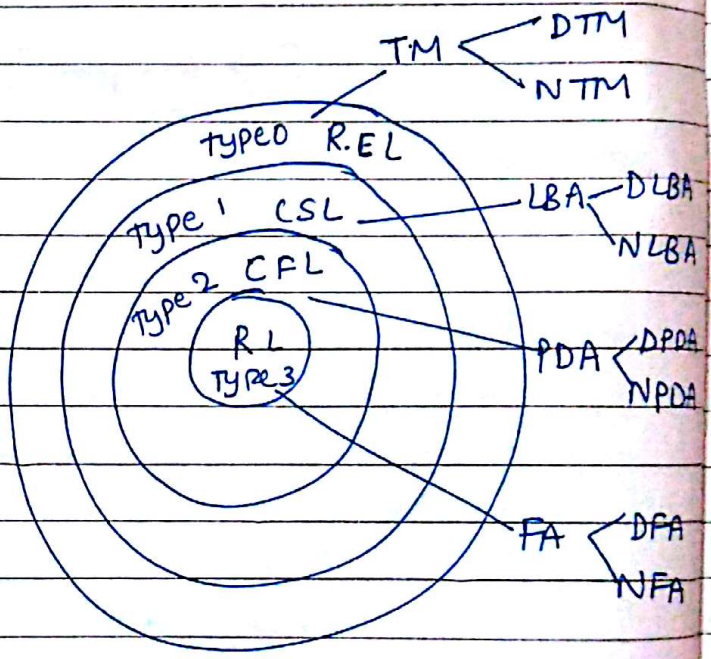
Type 3 C Type 0, 1, 2.

REL = Recursive Enumerable language

CSL = Context sensitive Language

CFL = context free Language

RL = Regular Language



Turing machine (TM) → Deterministic Turing machine
→ Non-Deterministic Turing machine

Linear Bounded Automata (LBA) → Deterministic Linear Bounded Automata
→ Non-deterministic " " "

Push Down Automata (PDA) → Deterministic PDA
→ Non-Deterministic PDA

Finite Automata (FA) → DFA – Deterministic FA
→ NFA – Non-Deterministic FA

* CFL but not RL → ✓
* CSL but not REL → ✗
* REL but not RL → ✓
* RL but not CSL → ✗

(i) CFL but not RL. ✓

If language is CFL ~~means~~ then it is not regular language
Because RL is the subset of CFL.
CFL is super set of RL.

(ii) CSL but not REL ✗

CSL is subset of REL. The statement CSL but not REL is not correct

(iii) REL but not RL. ✓

REL is supesset of RL. RL is subset of REL.

(iv) RL but not CSL ✗

CSL is supset of RL. So the statement is incorrect

Expressive Power:- No. of languages accepted by a particular machine is known as expressive power of that machine.

Language accepted by:-

Finite automata — Regular language
Push down automata — CFL, RL
Linear Bounded Automata — CSL, CFL, RL
Turing machine — REL, CSL, CFL, RL

Expressive power — TM > LBA > PDA > FA

Expressive Power of DFA = Expressive power of NFA

Because, the expressive power of DFA and NFA is same, hence every NFA is converted into DFA, i.e. for a language L if NFA exist the DFA also should exist.

Expressive power of NPDA > Expressive power of DPDA

hence every, NPDA cannot be converted into DPDA i.e for a language L, NPDA exist DPDA may or may not exist.

Expressive power of DTM = Expressive power of NTM.

Hence every NTM can be converted into DTM.
i.e for a language L if NTM exist DTM should also exist,

Date / 120

The expressive power of DLBA and NLBA is unknown

- the default FA is DFA
- the default PDA is NPDA
- the default TM is DTM

Identify types of grammar

1) $S \to aS \mid a$ (Right sensitive) – Type 3
Type (3, 2, 1, 0)

2) $S \to AB$, $A \to a$, $B \to b$ } Type 2 (✓)
(2, 1, 0)

3) $S \to aSb \mid bSb \mid a \mid b$
Type 2
(2, 1, 0)

4) $S \to aS \mid bS \mid \epsilon$
only right linear ⇒ Regular grammar
(3, 2, 1, 0)

Note- FA → Does not have memory element.
So, regular languages are useful only to represent strings without having comparison.

PDA = FA + Stack

If language contains comparisons then the suitable grammar is CFG.

TM = FA + 2 Stack

FA + 2 stack or 3 stack + 4 stack

TM > PDA > FA

The Central Concepts of Automata Theory

Alphabets

An Alphabet is a finite, non empty set of symbols. Conventionally, we use the symbol Σ for an alphabet. Common alphabets include:

1) $\Sigma = \{0, 1\}$, the binary alphabet.
2) $\Sigma = \{a, b, \ldots, z\}$, the set of all lower-case letters.
3) The set of all ASCII characters, or the set of all printable ASCII characters.

Strings

A string is a finite sequence of symbols chosen from some alphabet.

Eg:- 01101 is a string from binary alphabet $\Sigma = \{0, 1\}$

Empty String

The empty string is the string with zero occurrences of symbols. This string, denoted as ϵ, is a string that may be chosen from any alphabet whatsoever.

Length of a string

It is often useful to classify strings by their length, that is, the number of positions for symbols in the string.

Eg:- 01101 has length 5.

$|\epsilon| = 0$

$|011| = 3$

Date 1 / 120

Prefix of string :-

String formed by taking any number of leading symbols.

W = abc, then ∈, a, ab, abc are the prefixes.

Suffix :-

String formed by taking trailing symbols of string.

W = abc, then ∈, c, cb, abc.

Power of an alphabet.

If Σ is a alphabet, we can express the set of all strings of a certain length from the alphabet by using an exponential notation.

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

$\Sigma^1 = \{a, b, c\}$

$\Sigma^2 = \{aa, ab, bc, ca, ba, bb, bc, ca, cb, cc\}$

$\Sigma^3 = \{aaa, aab, aac, abb, abc, \ldots\}$

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

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots\ldots$

$\Sigma^* = \Sigma^+ \cup \{\in\}$

Concatenation of Strings

If x and y are two strings,

$x = a_1, a_2 \ldots a_i$

$y = b_1, b_2 \ldots b_j$

$xy = a_1 a_2 \ldots a_i b_1 b_2 \ldots b_j$

Eg:- $x = 1101$, $y = 11010011$

$xy = 1101\,11010011$

Language –

set of strings obtained from an alphabet.

Eg:-

1) The language of all strings consisting of n 0's followed by n 1's for some $n \geq 0$;

$\{\epsilon, 01, 0011, 000111\}$

2) The set of strings of 0's and 1's with an equal number of each:

$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$

3) The set of binary numbers whose value is prime.

$\{10, 11, 101, 111, 1011, \ldots\}$

$L_1 \cup L_2 = \{s \mid s \in L_1 \text{ or } s \in L_2\}$

$L_1 \cdot L_2 = \{s_1 s_2 \mid s_1 \in L_1 \text{ or } s_2 \in L_2\}$

$L_1^* = \bigcup_{i=0}^{\infty} L^i$

$L_1^+ = \bigcup_{i=1}^{\infty} L^i$

Consider language L_1 defined on the alphabet set,

$L_1 = \{a, b, c, d\}$

$L_2 = \{1, 2\}$

$L_1 \cdot L_2 = \{a1, a2, b1, b2, c1, c2, d1, d2\}$

$L_1 \cup L_2 = \{a, b, c, d, 1, 2\}$

$L_1^3 = \{abc, abd, aaa, \ldots\}$

$L_1^* = \{\epsilon, a, b, c, d, aa, ab, \ldots\}$

Contains strings of all possible lengths including null string.

L_1^+ = { Contains strings of all possible lengths excluding null string }

Language generated by grammar can be represented as L(G).

1) N = {S} , T = {a, b}

P = { 1. $S \to aSb$

2. $S \to ab$ }

What is the language derived from this?

Sol:- Type 2 grammar. (S is non termina and aSb is string)

$S \to aSb$

$S \to ab$

$S \overset{2}{\Rightarrow} ab$ $\quad ab \in L(G)$

$S \overset{1}{\Rightarrow} aSb$

$\overset{2}{\Rightarrow} aabb$ $\quad a^2b^2 \in L(G)$

Terminal string

$S \overset{1}{\Rightarrow} aSb$

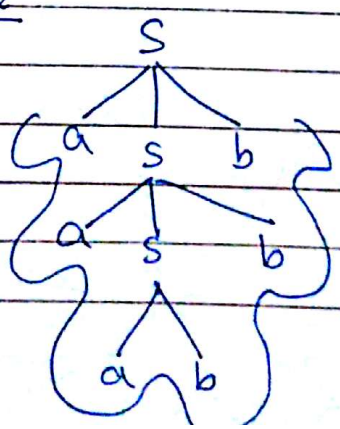
$\overset{1}{\Rightarrow} aaSbb$

$\overset{2}{\Rightarrow} aaabbb$

$\Rightarrow a^3b^3$ $\quad a^3b^3 \in L(G)$

$L(G) = \{a^n b^n / n \geq 1\}$

derivation tree



String generated is aaabbb result of the tree.

2) $N = \{S\}$, $T = \{a, b, c\}$

$P = \{$ 1. $S \to aSa$

2. $S \to bSb$

3. $S \to c \}$

Type 2 grammar (3)

Sol:- $S \overset{3}{\Rightarrow} c$ $\quad c \in L(G)$

$S \overset{1}{\Rightarrow} aSa$

$\overset{3}{\Rightarrow} aca$ $\quad aca \in L(G)$

$S \overset{2}{\Rightarrow} bSb$

$\overset{3}{\Rightarrow} bcb$ $\quad bcb \in L(G)$

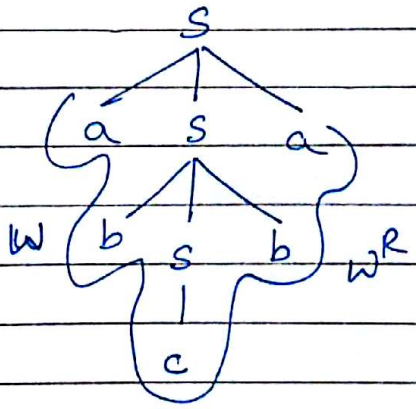
$S \overset{1}{\Rightarrow} aSa$

$\overset{2}{\Rightarrow} abSba$

$\overset{3}{\Rightarrow} abcba$ $\quad abcba \in L(G)$

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

Derivation tree



3) $N=\{S\}$, $T=\{a,b\}$, $P=\{1.\ S\to aSb$
$2.\ S\to ab$
$3.\ S\to SS\}$

Sol:- Type 2 grammar, This is not a linear grammar. (contains two nonterminals)
$S\to SS$.

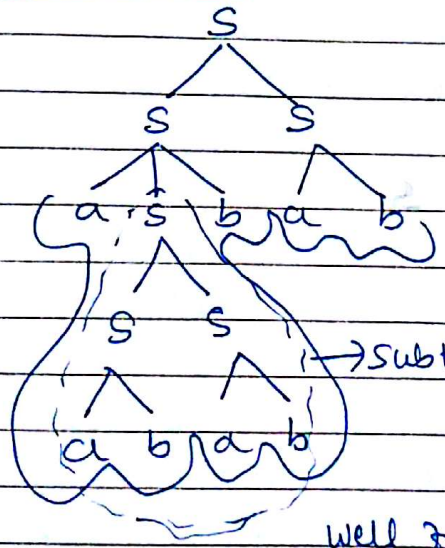
$S \overset{2}{\Rightarrow} ab$

$ab \in L(G)$

$S \overset{3}{\Rightarrow} SS$
$\overset{2}{\Rightarrow} abS$
$\overset{2}{\Rightarrow} abab$

$S \Rightarrow SS$
$\Rightarrow aSbS$
$\Rightarrow aSbab$
$\Rightarrow aSSbab$
$\Rightarrow aababbab$

Derivation tree -



If we replace a by '(' and b by ')' then,
aababbab
((()())())

L(G) = consists of well formed strings of parenthesis.

Well formed string of parenthesis is known as Dyck set.

4) $N=\{S, S_1\}$ $T=\{a, b\}$ $P=\{1. S \to aSb$

$2. S \to ab$

$3. S_1 \to SS_1$

$4. S_1 \to \in\}$

Sol ⇒ $S_1 \Rightarrow SS_1$

$\Rightarrow S\in$ $\quad S_1 \Rightarrow \in$

$\Rightarrow S$ $\quad \in L(G)$

$\Rightarrow aSb$

$\Rightarrow aabb$

$\Rightarrow aabb$ $\quad aabb \in L(G)$

$a^2b^2 \in L(G)$

$S_1 \Rightarrow SS_1$

$\Rightarrow SSS_1$

$\Rightarrow SS\in$

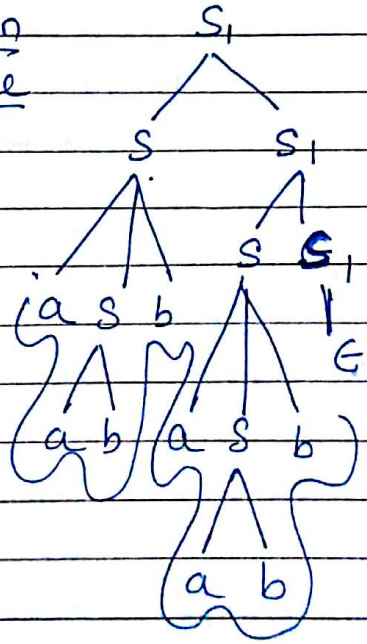
$\Rightarrow aSb\ aSb$

$\Rightarrow aabb\ aabb$

$\Rightarrow aabbbaabb$

$a^4b^4 \in L(G)$

Derivation tree



$L=\{a^nb^n / n \geq 1\}$

$L(G) = L^*$

$L^* = L^0 \cup L_1 \cup L_2 \cup \dots \cup L_n$.

5) $N=\{S\}$, $T=\{a,b\}$, $P=\{1.\ S\to aS,\ 2.\ S\to bS$
$3.\ S\to a,\ 4.\ S\to b\}$

Sol:- Type 3 grammar (regular)

$S \Rightarrow b \quad b\in L(G)$
$S \Rightarrow a \quad a\in L(G)$

$S \Rightarrow aS$
$\Rightarrow aa \in L(G)$
$S \Rightarrow aS \Rightarrow ab \in L(G)$

$S \Rightarrow bS \Rightarrow bb \in L(G)$
$S \Rightarrow bS \Rightarrow ba \in L(G)$

$L(G) = \Sigma^+$
$\Sigma = \{a,b\}$
string of length 2.
any string of a's b's.

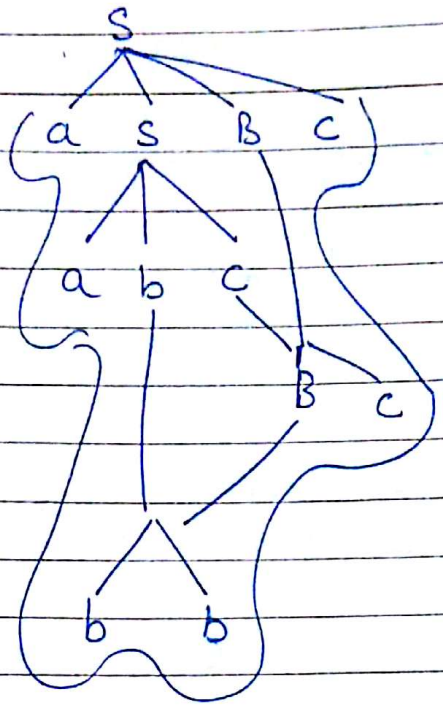
6) $N=\{S,B\}$ $T=\{a,b,c\}$ $P=\{1.\ S\to aSBC$
$2.\ S\to abc$
$3.\ cB\to Bc$
$4.\ bB\to bb\}$

Sol:- Type 1 grammar
$|\alpha| \le |\beta|$.

$S \overset{2}{\Rightarrow} abc \quad abc \in L(G)$

$S \overset{1}{\Rightarrow} a\underline{S}BC$
$\overset{3}{\Rightarrow} aa\underline{bc}\underline{B}C$
$\overset{4}{\Rightarrow} aa\underline{bB}cc$
$\Rightarrow aabbcc \quad a^2b^2c^2 \in L(G)$

Derivation tree –



aabbcc

$L(G) = \{a^n b^n c^n / n \geq 1\}$

7) N = {S} T = {a} P = { 1. S → aS
2. S → a }

Sol^n – Type 3 grammar.

S ⇒ a a ∈ L(G)

S ⇒ aS
⇒ aa

$L(G) = \{a^n / n \geq 1\}$

S ⇒ aS
⇒ aaS
⇒ aaa aaa ∈ L(G)

S ⇒ aS
⇒ aaS
⇒ aaaS
⇒ aaaa aaaa ∈ L(G)

8] $N = \{S, A, B\}$ $T = \{a, b\}$

$P = \{$ 1. $S \to aB$
2. $B \to b$
3. $B \to bS$
4. $B \to aBB$
5. $S \to bA$
6. $A \to a$
7. $A \to aS$
8. $A \to bAA \}$

Sol:- $S \overset{1}{\Rightarrow} aB \overset{2}{\Rightarrow} ab$ Type 3 grammar

$S \overset{5}{\Rightarrow} bA \overset{6}{\Rightarrow} ba$

$S \overset{1}{\Rightarrow} aB \overset{3}{\Rightarrow} abS \overset{5}{\Rightarrow} abbA \overset{6}{\Rightarrow} abba$

$S \overset{1}{\Rightarrow} aB$
$\overset{4}{\Rightarrow} aaBB$
$\overset{4}{\Rightarrow} aaaBBB$
$\overset{3}{\Rightarrow} aaabSBB$
$\overset{3}{\Rightarrow} aaababBB$
$\overset{3}{\Rightarrow} aaababbB$
$\overset{3}{\Rightarrow} aaababbb$

$S \overset{5}{\Rightarrow} bA$
$\overset{8}{\Rightarrow} bbAA$
$\overset{7}{\Rightarrow} bbaSA$
$\overset{7}{\Rightarrow} bbaSaS$
$\overset{1}{\Rightarrow} bbaaBaS$
$\overset{5}{\Rightarrow} bbaaBabA$
$\overset{2}{\Rightarrow} bbaababa$

$L(G) = \{$ ~~consists of~~ strings consisting of equal no. of a's and b's $\}$

Dyck set

→ In this set string has equal no. of a's and b's.
But restriction is, left parenthesis should come before matching right parenthesis.
Eg:- aabaabbab, abab

Regular set

→ here, also string contains equal no. of a's and b's
Eg:- abab, baab, baba
a = '(', b = ')'
abab = ()() ✓
baab =)(() X
baba =)()(X

a = '(', b = ')'

(()()) ()- aababbab

()() - abab

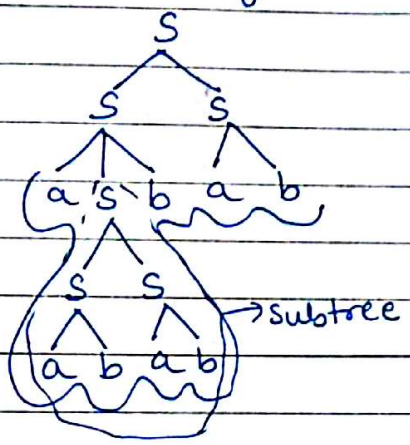
* All strings in set follows restriction.

* Nesting is here.

Eg. (()()) ()

↓

nesting



* All strings in set should not follow restriction which applied on Dyck set that left parenthesis should come before matching right parenthesis.

Some strings follows but some strings does not follow

* Nesting is not here

Regular Expressions

Regular expressions are used for representing certain set of strings in an algebraic fashion.

1) Any terminal symbol i.e symbols ~~bcds~~ in belongs to alphabet (Σ) including Λ, $\emptyset$ are regular expressions.

Λ or ϵ (empty string)

$\emptyset$ (null symbol)

2) the union of two regular expressions is also a regular expression. if R_1 and R_2 are two regular expressions then $R_1 + R_2$ is also a regular expression.

3) The concatenation of two regular expressions is also regular expression.

if R_1, R_2 are two regular expressions then $R_1 \cdot R_2$ is also regular expression.

4] The iteration (or closure) of a regular expression is also a regular expression.

$R \rightarrow R^*$

$a^* = \epsilon, a, aa, aaa, aaaa, \ldots\ldots$

If R is a Regular expression then L(R) is the regular language.

$\emptyset$	denotes	$\emptyset$	$L(\emptyset) = \emptyset$
ϵ	denotes	$\{\epsilon\}$	$L(\epsilon) = \{\epsilon\}$
a	denotes	$\{a\}$	$L(a) = \{a\}$
(R_1)	denotes		$L(R_1)$
$R_1 + R_2$	denotes		$L(R_1) \cup L(R_2)$
$R_1 \cdot R_2$	denotes		$L(R_1) \cdot L(R_2)$
$(R_1)^*$	denotes		$(L(R_1))^*$

Describe the following set as Regular expressions.

1) $\{0,1,2\}$ 0 or 1 or 2 $R = 0+1+2$

2) $\{\Lambda, ab\}$ $R = \Lambda + ab$

3) $\{abb, a, b, bba\}$ abb or a or b or bba

$R = abb + a + b + bba$

4) $\{\Lambda, 0, 00, 000, \ldots\}$ or $\{\epsilon, 0, 00, 000, \ldots\}$

$R = 0^*$

5] $\{1, 11, 111, 1111, \ldots\}$

$R = 1^+$.

Star binds Tightest

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

Concatenation Binds tighter than Union.

$abUc = (ab)Uc$

Other notations -

$ab|c = (ab)|c$

$a^* = \{a\}^*$

$a^+ = a\,a^* = \{a\}^+$ "one or more".

Parsing Practice -

$aabUc\ aa\ bUc\ aa = ?$

$(aab)\ U\ (caab)\ U\ (caa)$

$aab\ |\ caab\ |\ caa = ?$

$= (aab)\ |\ (caab)\ |\ (caa)$

$d\ U\ ab^*\ cd^* = ?$

$= (d)\ U\ (a(b^*)\ c\ (d^*))$

$= (d)\ U\ (a(b^*)c\ (d^*))$

→ Each Regular expression describes a language.

→ Regular languages are closed under union, concatenation and closure (star).

→ Regular expressions describe regular languages.

Identities of Regular Expressions

1) $\emptyset + R = R$
2) $\emptyset . R + R . \emptyset = \emptyset$
3) $\epsilon . R = R . \epsilon = R$
4) $\epsilon^* = \epsilon$ and $\emptyset^* = \epsilon$
5) $R + R = R$
6) $R^* . R^* = R^*$
7) $R R^* = R^* R$
8) $(R^*)^* = R^*$
9) $\epsilon + R R^* = \epsilon + R^* R = R^*$
10) $(PQ)^* P = P(QP)^*$
11) $(P+Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$
12) $(P+Q)R = PR + QR$ and
    $R(P+Q) = RP + RQ$.

$(0+10^*)$, $L = \{0, 1, 10, 100, 1000, 10000, \ldots\}$

(0^*10^*) , $L = \{1, 01, 10, 010, 0010, \ldots\}$

$(0+\epsilon)(1+\epsilon)$, $L = \{\epsilon, 0, 1, 01\}$

$(a+b)^*$ = set of strings of a's and b's of any length including the null string.
So, $L = \{\epsilon, a, b, aa, ab, ba, ba, \ldots\}$

$(a+b)^*abb$ Set of strings of a's and b's ending with abb. So,
$L = \{abb, aabb, babb, aaabb, ababb, \ldots\}$

$(11)^*$ Set consisting of even number of 1's including empty string.
So, $L = \{\epsilon, 11, 1111, 111111, \ldots\}$

$(aa)^*(bb)^*b$ set of strings consisting of even no. of a's followed by odd number of b's

So, L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, ...}

$(aa+ab+ba+bb)^*$ String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba, and bb, including null.

So, L = {aa, ab, ba, bb, aaab, aaba, ...}

General applications of Automata -

- Biology
- cellular Automata
  - ✓ Game for life
  - ✓ Pascal's Triangle
  - ✓ Brain's Brain

Types of automata and Applications

- Turing Machine -

  Applications:
  - Turing machine Counting
  - Turing machine subtraction
  - 3-State busy Beaver
  - 4-State busy Beaver
  - Justification.

Date / /20

Linear Bounded Automata -

Applications -

- Genetic programming.
- Parse trees.
- Emptiness problem.

Push Down Automata -

Applications -

- Online transaction process system
- UPPAAL Tool
- Tower of Hanoi (Recursive solution)
- Timed automata model.
- Deterministic Top down Parsing LL grammar
- Context free Language.
- Predictive Bottom up parsing LR grammar.
- Converting a PDA.

Finite Automata -

Applications -

- Finite State programming.
- Event-Driven Finite State machine.
- Virtual FSM
- DFA based text filter in Java.
- Acceptors and Recognizers
- UML state diagrams.
- Transducers
- Hardware applications
- & FS.

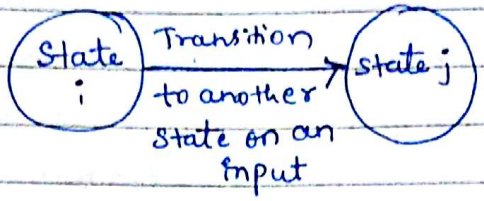
Regular expressions are a notation to represent lexeme patterns for a token.

- They are used to represent the language for lexical analyzer.
- They assist in building finding the type of token that accounts for a particular lexeme.

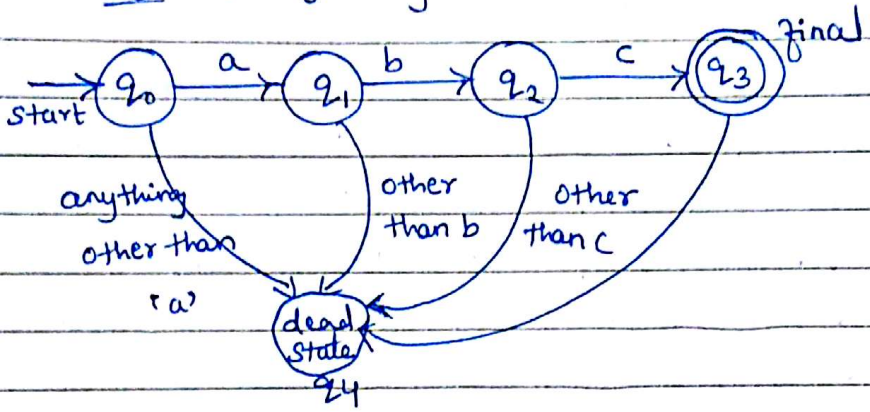
Overview of 4 types of Automata -

(1) Finite Automata:

Finite: fixed number of states.



E.g:- Recognizing "abc"



abc

$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{c} q_3$ (final state reached)

↓ accept string abc.

abd

$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{d} q_4$ (dead state)

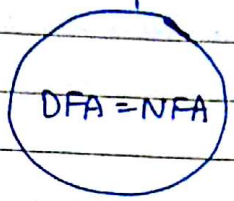
↓ reject string abc

* string ends in final state ⇒ string is accepted
* String ends in non-final state ⇒ rejected.

Date / /20

Finite automata :- 2 types [Deterministic
Non deterministic

DFA has same power as NFA



* Both recognize regular language.

(2) Pushdown Automata

Note : DFA or NFA has no memory (Only states)
PDA has a DFA + stack of infinite size.

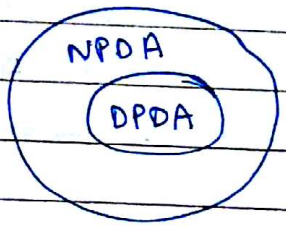
$\{0^n 1^n\}$ set of strings

$\{0^n 1^n \mid n \geq 1\}$

$\{01, 0011, 000111, 00001111, \ldots\}$

Equal no. of 0's and 1's → finite automata cannot recognize this string. because this string is infinite. But in finite automata we have only finite no. of states.

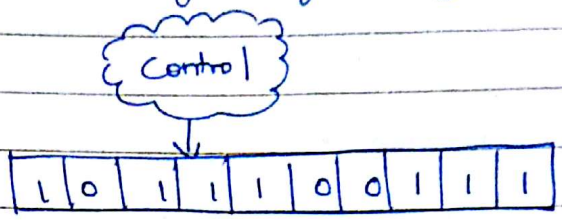
→ PDA recognizes this type of string.
→ PDA recognizes context free languages.
Non-deterministic PDA has more power than deterministic PDA.
→ NDPA recognizes the some languages which cannot be recognized by DPDA.



2) Linear Bounded automata:-

Note:- In PDA, we have stack as a temporary storage. In stack we can see only top element not the bottom element - PDA.

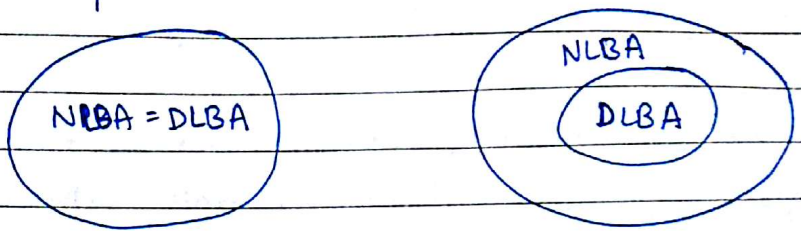
→ A restricted form of Turing machine.



The tape is limited in size to the size of the input.

→ LBAs are not as powerful as the full Turing machines.

Deterministic Linear Bounded Automata has same power or more power than deterministic Linear Bounded Automata.



Assume our tape alphabet can be larger than the input alphabet.

We can use a larger the alphabet to store more information in our limited tape.

Or equivalently,

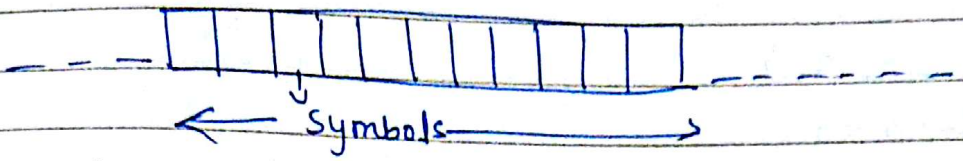
We can restrict our machine to using only a small portion of the tape.

$$A_{LBA} = \{ \langle M, w \rangle \mid M \text{ is an LBA and } M \text{ accepts } w \text{ is decidable} \}$$

Date 1/20

4) Turing machine -

DFA + infinite tape (array)

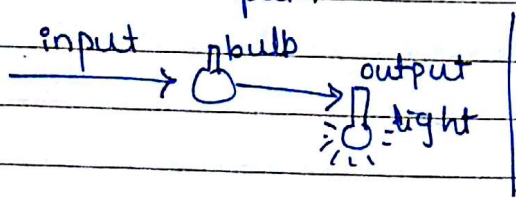


Turing machine recognizes recursively enumerable language.

*** Alan Turing → Father of computer science.

Finite Automata → (Mathematical model)

DFA - exact output.



A machine works

we assume →
Starting state q_0 {off state}
Final state q_f {on state}

Definition of a Deterministic Finite automaton -

A deterministic finite automaton consists of:

1) A finite set of states, often denoted Q.
2) A finite set of input symbols, often denoted Σ.
3) A transition function that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted 'δ'.

$$Q \times \Sigma \longrightarrow Q$$

δ : transition function.

4) A start state, one of the states in Q.
5) A set of final or accepting states F.

DFA is five tuple notation:

$$A = (Q, \Sigma, \delta, q_0, F).$$