

## UNIT - I

(1)

### THE STRUCTURE OF COMPILER

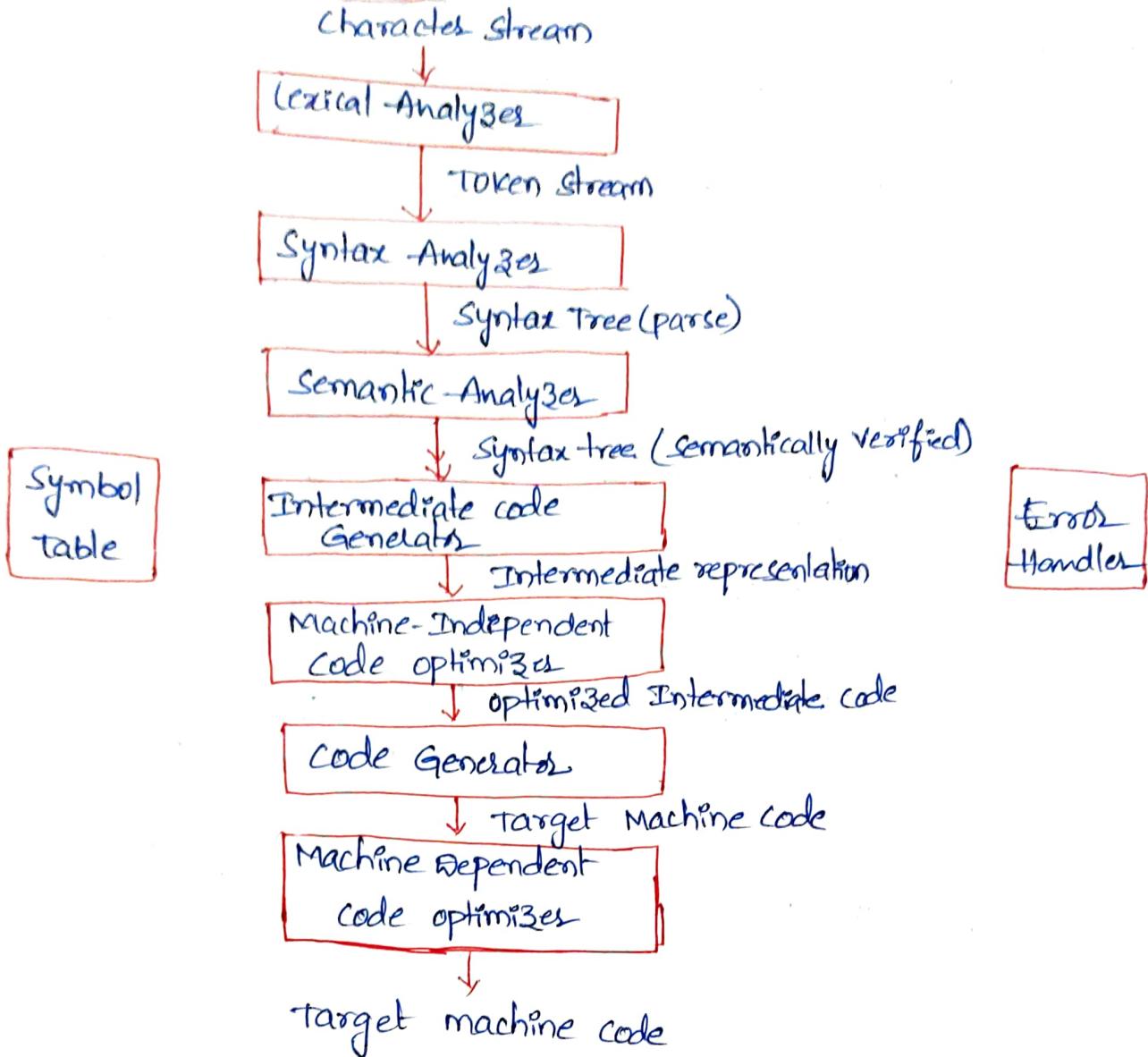


Fig: phases of the compiler

\*All the phases are interact with the symbol table & Error Handler

#### (i) Lexical Analysis:

↳ Lexical Analyzer is also called as scanner.

↳ LA reads the source program characters by characters and converts the corresponding characters into "lexeme" meaningful sequence called "lexeme".

lexeme → A sequence of characters.

↳ For each lexeme the lexical Analyzer produces "TOKEN" as o/p

↳ Token may be a keyword, identifier, operator, constant

↳ Token is represented as <token-name, attribute-value>

↓  
Name of the variable  
1st component

↓  
2nd component

↓  
points to entry in symbol  
table entry

Ex:- Assignment stmt  $a = b + c * 90$

\*  $a = b + c * 90$

\*  $\langle id, 1 \rangle, \langle = \rangle, \langle id, 2 \rangle, \langle + \rangle$

$\langle id, 3 \rangle, \langle * \rangle, \langle 60 \rangle$

\* operators & constants does not  
contain attribute value.

$id_1 = id_2 + id_3 * 60$

↓  
O/p of Lexical Analyzer

lexeme	Token
a	identifier
=	Assignment operator
b	identifier
+	Addition operator
c	identifier
*	Multiplication operator
60	constant

↳ Token Table is a data structure is used to store the  
token information (such as what is variable name, value, & scope of the  
variable ---)

### Syntax Analyzers

↳ Syntax Analyzers is also called as parser.

↳ SA ~~tokens~~ will checks whether the corresponding syntax of the  
source program is correct or not.

↳ If the syntax is not correct, Error Handler reports error to the user.

↳ If the syntax of source program correct,

↳ SA takes i/p as tokens and creates tree like

↳ Syntax Analyzers takes i/p as tokens and creates tree like  
intermediate representation as called the syntax tree.

↳ In the Syntax tree, there are two nodes exists

(i) Interior nodes → represents operations

(ii) children node → represent arguments of the operation

\* Evaluating the expression:  
 The operator which is having higher priority will be evaluated first followed by lowest priority operators. (2)

$$id_1 = id_2 + id_3 * 60$$

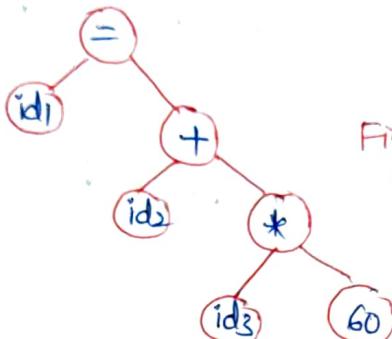
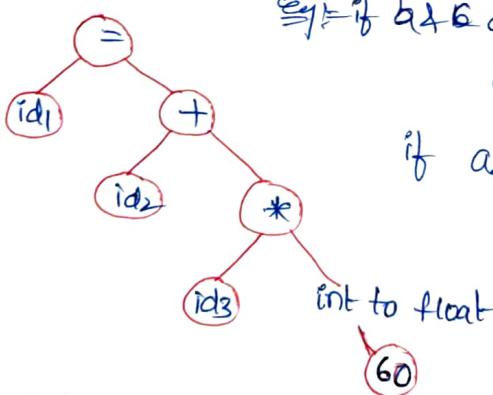


Fig: parse tree

### Semantic Analyzes:

- The Parse tree is given as an I/P to semantic Analyzes.
- Semantic Analyzes will check whether the meaning of the parse tree is correct or not (semantically). Verifies the parse tree.
- For verifying the parse tree semantically, it uses a SW called type checker, it checks the data types of variable, if there any need of type conversion then it will perform type conversion.

e.g:-



if  $a, b$  are int, 1, 2

$$a = b + c * 60 \Rightarrow 1 + 2 * 60 \Rightarrow$$

if  $a, b$  are float 1.5, 2.5

$$a = 1.5 + 2.5 * 60 \Rightarrow$$

Fig: Semantically verified parse tree.

### Intermediate code Generator:

- Semantically verified parse tree is given as an I/P to ICG, and it produces intermediate representation as O/P.

- In process of translating source program into target code compiler may construct one or more intermediate representation (such as, three address code, polish Notations)
  - Syntax tree is also one of the Intermediate representation.
- 3-Address code:-** 3-Rmp rules are
- ① 3-address code assignment instruction should have almost one operator on RHS (should not have more than operators on RHS)
  - ② Compiler must generate temporary variables for storing result.
  - ③ Some instruction may contain fewer than 3-operands.

Eg:-  $id_1 = id_2 + id_3 * 60$

$$t_1 = \text{int to float}(60)$$

$$t_2 = id_3 * t_1$$

$$t_3 = id_2 + t_2$$

$$\cancel{t_3} \rightarrow id_1 = t_3$$

$t_1, t_2, t_3 \rightarrow$  are temporary variables

### Code optimization:-

- Intermediate code representation is given as an i/p to code optimization
- Code optimization is process of eliminating unnecessary statements (shorter code <sup>that</sup> executes fast, consumes less power).

$$t_1 = id * 60.0$$

$$id_1 = id_2 + t_1$$

### Code Generation:-

- It takes optimized code intermediate code from code optimizer and generate machine code,
- In order to produce machine code it uses some assembly languages.

LD	Reg, M	LDF	R <sub>2</sub> , id <sub>3</sub>
ST	Memory, Reg.	MULF	R <sub>2</sub> , R <sub>2</sub> , #60.0
MUL	R <sub>1</sub> , R <sub>2</sub> , R <sub>3</sub>	LDF	R <sub>1</sub> , id <sub>2</sub>
		ADDF	R <sub>1</sub> , R <sub>2</sub>
		STF	:id <sub>1</sub> , R <sub>1</sub>

### Error Handles:-

- ↳ Errors handles interact with all the phases of the compiler.
- If any errors reports the errors in present any of the compiler phase, to the user.

### Grouping of phases into passes:-

→ In implementation

#### (1) Single pass compiler :-

- \* All the phases of compiler are grouped into one part.

#### (2) Two-pass compiler :-

- \* Also called as multi-pass compiler.

Single Pass Compiler → The phases of compiler into are grouped into two-parts.

Source program(HLL)

↓  
Lexical Analysis

Syntax Analysis

Semantic "

ICG

Code optimizer

Code Generation

↓  
target machine code

(Assembly language code)

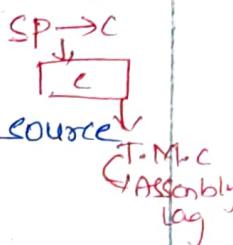
Fig: Single pass compiler

## Two-pass compiler:

1. Analysis part (a) Front end of the compiler
2. Synthesis part (b) Backend of the compiler

### Analysis part:

- ↳ For analysing the code, we are using analysis part.
- ↳ It breakup the source program into pieces and imposes grammatical structure on them, and these structure is used to create intermediate representation source program.
- Analysis part contains the phases that are dependent on source language and independent on target program.
- It detects the errors in source program syntactically & semantically.
- It collects the information about the SP and stored it in symbol Table.



### Synthesis part:

- ↳ It focuses contains the phases that are dependende on Target language and independent of source program.
- ↳ It construct the Target program from Intermediate representation stored in the symbol table.

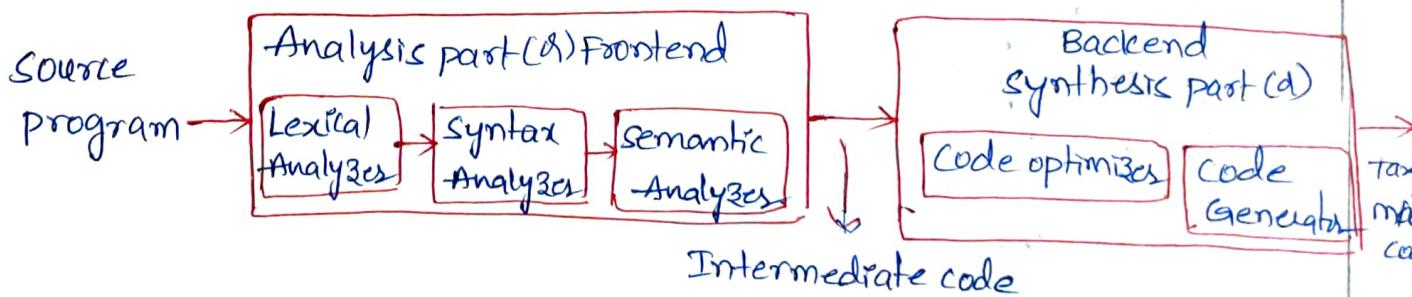


Fig: Two-pass (a) Multi-pass compiler

## ② THE SCIENCE OF BUILDING A COMPILER:-

- ↳ A compiler must accept all the source program, that can follow the language specifications.
- ↳ A source program is infinite & any program can very large, consisting million lines of code.
- ↳ Any transformation performed by the compiler while translating source program must preserve the meaning of program being compiled.
- Compiler writers must not only have complete idea of the compilers they create, but all the programs that their compilers compile.

### Modeling in compiler design and Implementation:-

- The study of compiler is mainly a study of how we design the right mathematical models & choose the right algorithm.
- Some of the most fundamental models are.
  - (i) Finite state machine and Regular Expressions are useful for describing the lexical units of programs & for describing algorithms used by the compiler to recognize those units.
  - (ii) Context Free Grammar is used to describe the syntactic structure of the programming languages.

### The science of code optimization:-

- The "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code.

Compiler optimization must meet the following objectives.

- The optimization must be correct, that is preserve the meaning of compiled program.
- The optimization must improve the performance of many programs.
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.
- It is not guaranteed that the optimizing compiler should produce completely error free code.

### Applications of compilers:

- Implementation of High level languages.
- Optimization for computer Architectures
- Design of New computer architecture
- Program Translation
- SW productivity tool

### Compiler construction tools:

- Compiler writers, any SW developer, use SW development tools such as editors, debuggers, version managers, profilers, test harnesses...
- In addition to these tools, more specialized tools have been created to implement various phases of compiler.

#### 1. lex tool - scanner generator

It takes I/p as RE & produces O/p as stream of tokens.

#### 2. parser generator = yacc tool - it generates parse tree.

3. SPP engines — collection of functions are used to traverse the  
↓  
Semantic Analysis Parse tree & generate Intermediate code.

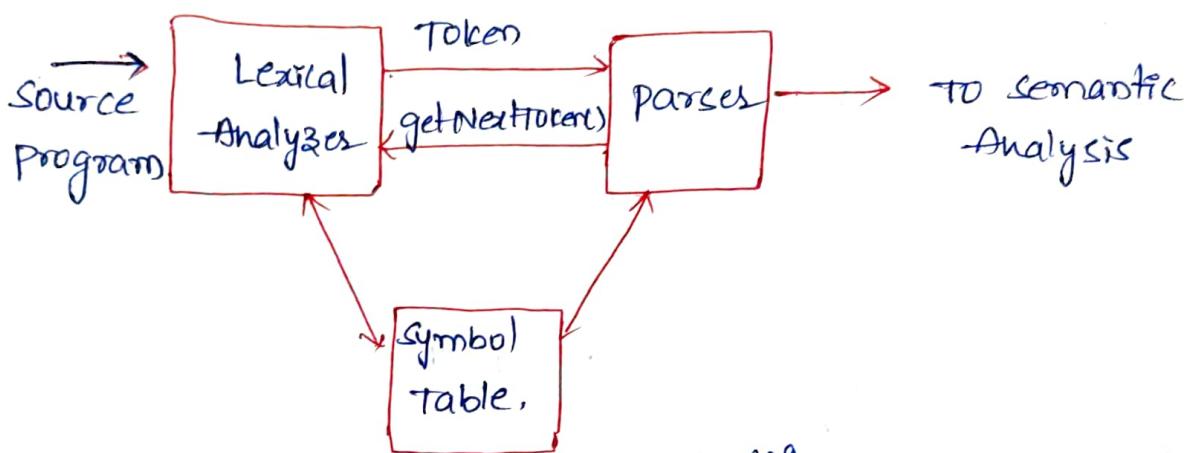
#### 4. Intermediate code generator generator.

4. Data flow Analysis Engine - code optimization is done.

5. compiler - construction tool kit → provide an integrated set of routines for constructing various phases of compiler.

### Role of the Lexical Analyzer:

- It is the first phase of the compiler.
- The main task of lexical analyzer is to read the i/p characters from the source program, and group them into lexemes and produce o/p a sequence of tokens is sent to parser for syntax analysis.
- LA enter lexemes consisting an identifiers into symbol table.
- 



- An interaction is implemented by the <sup>having</sup> parsers call the lexical Analyzer
- The call suggested by the `getNextToken()` command, causes the lexical Analyzer to read streams of characters from its i/p until it can identify the next lexeme and produces the next token, which it returns to the parser.

## (i) The Functions of Lexical Analyzers:

- ① It produces a stream of tokens.
- ② It removes comments from the source program.
- ③ It also removes white spaces, characters such as blank, newline, tab spaces.  
Eg:- `#include <stdio.h> main() { int,`
- ④ It counts the line numbers of source
- ⑤ It generates a symbol table which stores information about identifiers
- ⑥ It provides error messages with corresponding line no & column no.
- ⑦ If the source program uses macro processor, the expansion of macros may be performed by the lexical analyser.

Lexical Analyzers are divided into two processes.

- (i) **Scanning**: consists of simple processes that don't require tokenization of input, such as deletion of comments & whitespaces
- (ii) **Lexical Analysis**: is more complex portion, which produces the tokens from the o/p of the scanner.

## Lexical Analysis versus Parsing:

- ② Separation of LA from SA:
  - (i) To simplify the design of compiler
  - (ii) To increase the efficiency of compiler
  - (iii) To enhance the portability of compiler (working on several platforms)
- (i) If we combine these two phases into single phase then burden on a single phase is increased, so in order to reduce the burden on single phase we will separate LA from SA, so that LA → will do its own task  
SA → will " " " " individually.

- (2) In lexical analysis phase we are using Regular Expressions to recognize the tokens, RE are recognized with the help of finite automata.  
 → Syntax Analysis uses CFG - pushdown automata)  
 \* Specialized buffering techniques are used for reading i/p character can speedup the compiler syntactically.

### (3) Tokens, patterns, & Lexemes:

Token: Token is a pair which consists of two components.

$\langle \text{Token-name}, \text{attribute-value} \rangle \rightarrow \text{TOKEN representation}$ .  
 TOKEN name may be id, op, keyword & constant → A pointer which points to symbol table for token information.

### Lexeme:

Lexeme is a sequence of meaningful characters in source program.  
 → The pattern for a token is identified by the LA as an instance of token.

$E = m^* C^{**} 2$  (Lexemes are matched against the patterns)

### Patterns:

→ Pattern is a rule for describing all these lexemes that can represent a particular token in source language.

RE for id →  $(l|_)(\underline{l}|d|_)*$  ( $\underline{l}$  → 0 or more occurrence)

alphabet & underscore  $l \rightarrow [a-z A-Z] [a-zA-Z0-9]$

Token	Informal Description	Sample Lexeme
if	characters i, f	if
else	character e, else	else
comparision	<or> &lt;=, or >, or =, !=	<=, !=
letter	letter followed by letter/digit	p, s, t, 1, 2
Number	any o number constant	1234567890

## Lexical Error

→ lexical analyzer may not proceed if no pattern matching the prefix of remaining  $^{ip}$

→ These errors are generated during lexical analysis phase.

Commonly generated lexical errors:

1. Spelling errors → (do, do!, if - fi)
2. unmatched string — prompt ("hai");  $\$ \$ \& \# \#$
3. Appearance of illegal character — prompt ("hai");  $\$ \$ \& \# \#$   $\text{printf}("hai")$
4. exceeding the length of identifiers → In C max length of id is 32

### Error Recovery Actions

- (i) Delete one character from the remaining  $^{ip} \rightarrow d \rightarrow \text{Delete}^i, \text{do insert } 0.$
- (ii) Insert a missing character into the remaining  $^{ip} \rightarrow \text{print}("hai")$
- (iii) Replace one character by another character  $\rightarrow \text{do}, d, do$
- (iv) transpose two adjacent characters  $\rightarrow rp$
- (v) Delete successive characters from the remaining  $^{ip}$  until LA recognizes a well-formed token (panic mode - recovery)

## INPUT BUFFERING

technique

→ lexical analyzer uses the input buffering to read input characters from the source program.

→ to read  $^{ip}$  characters from the source program LA uses

two pointers

- (i) lexeme begin pointer

- (ii) forward pointer

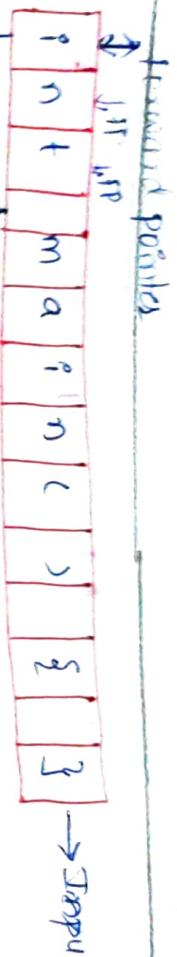
Eg: = int main()

{

}

program  
↓ after saving stored in  
HDD at the compilation stage  
main memory (at the time of  
execution)  
↓  
CPU

(7)



- Lexeme Begin pointer points to the beginning characters of the lexeme.

→ Initially the Forward pointer is placed at beginning character of lexeme.  
→ first Forward pointer reads 1st character (i) and moves one position to the right to read the next character (n)...  
→ whenever Forward pointer reaches a blank space then it identifies it as the end of lexeme. so, the "int" is considered as token.

→ The FP ignores the blank space, lexeme begin pointer and forward pointers are placed at next character location.

lexemeBeginPoint( )



- For each character from secondary memory (HD) one system call is required.
- If the program size is very large, it requires more no. of system calls are required.
- To overcome this problem, we are using Buffering Techniques.  
Buffering: Instead a block of characters are to be read into the buffer using only 1 system call.
  - The use of I/O buffering instead of reading a single character from secondary memory, the block of characters are read.

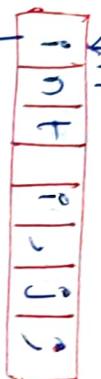
[ i | n | + | m | a | i | n | ( | ) | ; | ; ] /  
↳ It requires only 1-system call to read the program

Buffering can be implemented in 2 ways.

(i) One Buffer Scheme

→ In this technique we are using only one buffer to read the input string.

→ In this technique also we are using (i) lexeme begin pointers and (ii) forward pointers to read the i/p characters from the buffer.



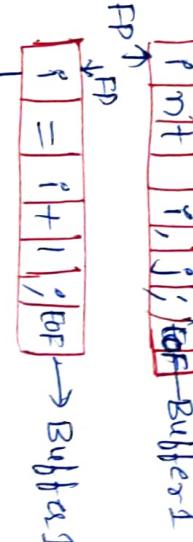
lexeme begin pointer

→ The problem with this approach is if the i/p string size is larger than than the capacity of the buffer, then the buffer has to be overridden pointers to store the remaining i/p string.

$$\text{eg: } \text{i/p string} = 500 \quad \text{Buffer size is } 100 \quad [1|2|3| \dots |100] \rightarrow \text{overridden}$$

(ii) Two-Buffer Scheme:

→ In this technique, it uses two buffer to store i/p characters.



$$\text{eg: } \text{i/p string} = 500$$

$$[1|2| \dots |100] \rightarrow \text{Buffer 1}$$

$$[1|2| \dots |100] \rightarrow \text{Buffer 2}$$

↓

Buffer 1 size = 100

→ In this approach, 1st the buffer is filled.  
→ After the first buffer is completely filled, then only we will fill the second buffer.

→ To determine whether the 1st buffer is completely filled or not,

it uses a special character called "Sentinal" or "EOF".  
use ~~use~~ ~~for~~ (End of the file is sentinel character)

→ whenever the second buffer is completely filled, then the remaining i/p string has to be filled in first buffer, now the first buffer is "overridden".

108

(Offer Letter) 1/20/11 (201-300)

Eq:  $i/p$  String  $SBe = 500$

IP:  $(1-100)$

1	2	3	100	EOF
---	---	---	-----	-----

FP:  $\downarrow$

$\rightarrow$  1st Buffer  $SBe = 100 \rightarrow$ 

201	202	1	1	30
-----	-----	---	---	----

IP Buffer 1

Refill:  $\downarrow$

201 202 - - - B0<sup>80</sup> Buffer1

E.g. - Switch (\*forward++)

Stack → Stack = 100

Case 'eof'.

if (forward ie at end of 1st buffer)  
  {  
    forward\_index = 1 and index = 0  
  }

forward = beginning of 2nd buffer

2nd buffer

else if (forward is at end of 2nd buffer)

forward = beginning of 1st buffer;

traffic first before

Retrogration

## RECOGNITION OF TOKENS:

Tokens are recognized Transition diagram.

- (i) Recognition of identifiers
  - (ii) Recognition of delimiters
  - (iii) Recognition of Relation operators
  - (iv) Recognition of keywords (*if, else, for...*)
  - (v) Recognition of Numbers (*int,float*)

Recognition of identities: The regular definitions for terms

letter → a | b | ... | z | ... A | B | ... | Z |

digit → 0/1 → (b)

Num → digit<sup>+</sup>(digit<sup>+</sup>)<sup>?</sup>(ε[+−]digit<sup>+</sup>)<sup>?</sup>

$\text{if} \rightarrow \text{id}$  then  $\rightarrow$  then  
 $\text{digit} \rightarrow [0-9]$  digit  $\rightarrow$  digit +  
 $\text{delim} \rightarrow \text{blank}$  delimiter (delimiter)\*  
 $\text{WS} \rightarrow$  delimit + delimiter (delimiter)\*  
 (whitespace)

### lexemes

	Token Name	Attribute value
if	if	-
then	then	-
else	else	-
Any id	id	-
Number	Number	points to table entry
<	relOp	LT
$\leq$	relOp	LE
=	relOp	EQ
$\leftrightarrow$	relOp	NE
>	relOp	GT
$\geq$	relOp	GE

→ tokens are recognized with the transition diagram.

### Transition diagram:

→ The intermediate step in the construction of lexical analyzer, i.e. converting patterns into flowcharts called are called transition diagram.

→ Transition diagram have a collection nodes or circles, called "states".

→ The states are connected by arrows, called "edges".  
 → A double circle indicating an accepting state (a final state in which a token is found).

→ Transition diagram is also called as "Finite automata".

→ start (or) initial state is indicated by edge labeled as "start".

any IP symbols have been read.



<, <=, >, >=, ==, !=

retract (comeback by one character)

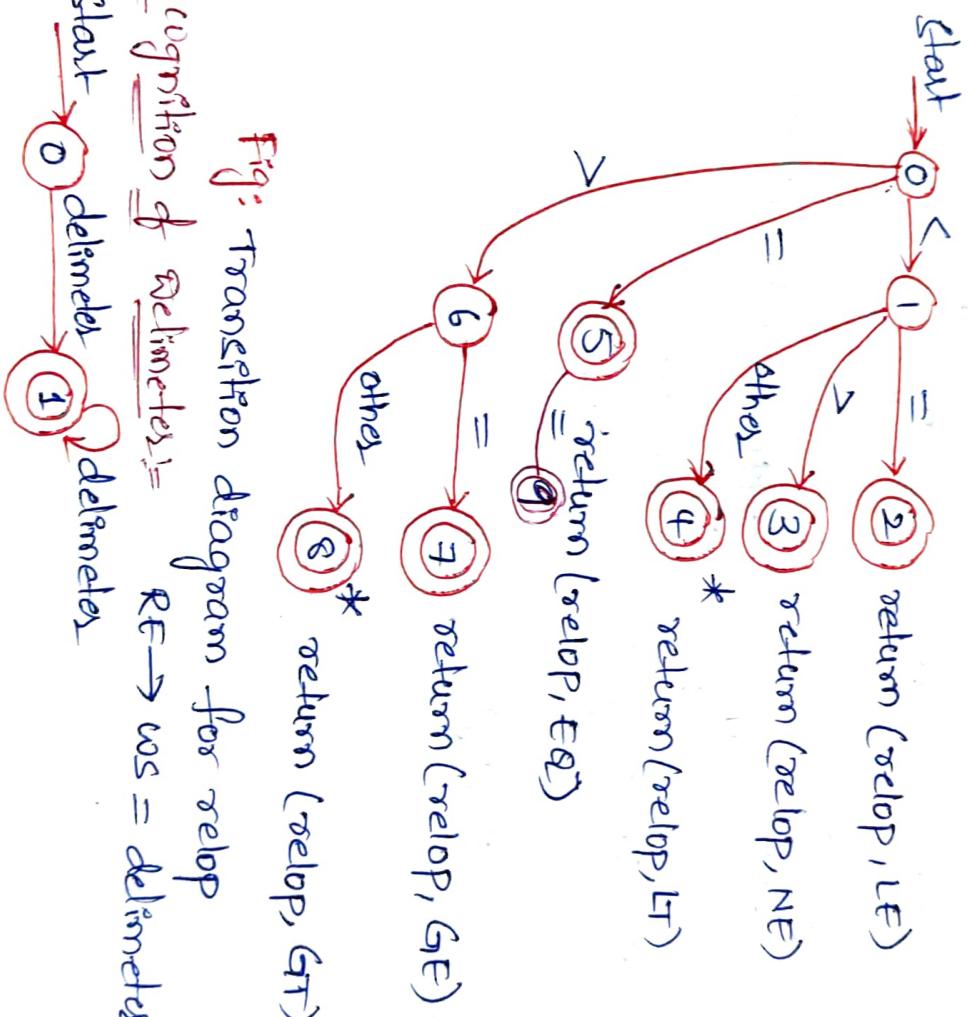


Fig: Transition diagram for relop

Recognition of Delimiter:= RE → ws = delimiter (delimiter)\*

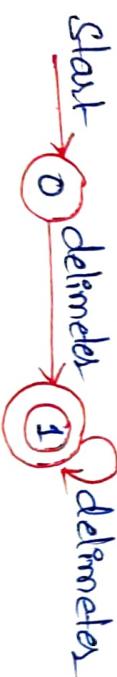


fig: Transition diagram for delimiters

Recognition of keywords:

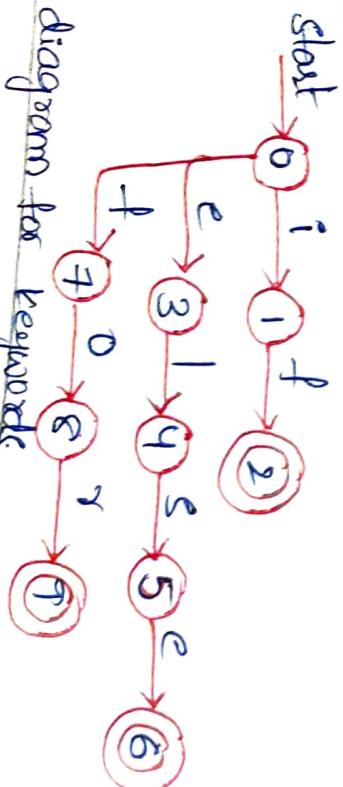
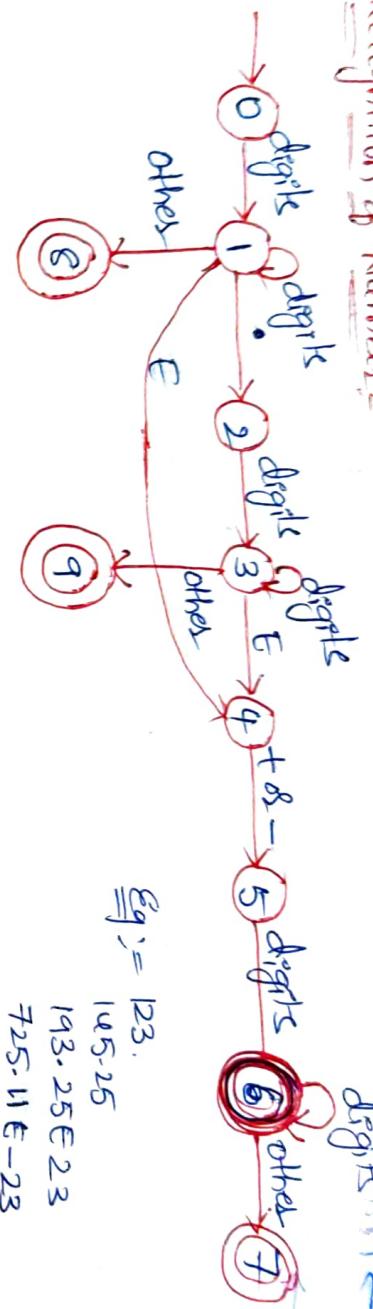


fig: Transition diagram for keywords

## Recognition of Numbers



Eq.: = 123.

145.25

193.25 E 23

725.11 E -23

[? → Represents  
0 or more one  
occurrence]

Recognition of Identifiers: letters (letter/digit)\*

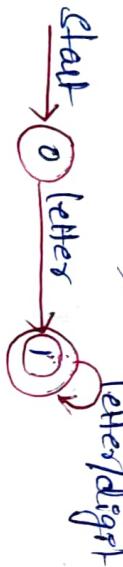
letter/digit

letter → 0 letter → 1 other → 2

return (gettoken(), install\_idx)

Fig: Transition diagram for identifiers

(d)



letter/digit

letter

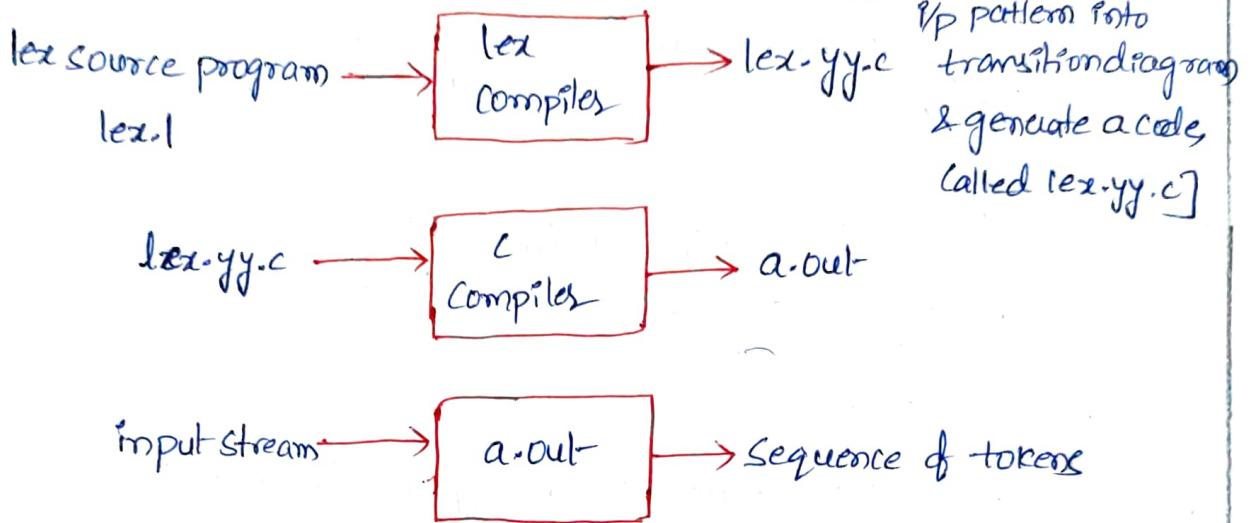
letter

## LEXICAL ANALYZER GENERATOR LEX

(d)  
A language for specifying lexical Analyzer

- Lex is a tool or a language which is used to generate the lexical analyzer.
- Lexical analyzers specifies a Regular Expressions and these RE are used to represent the pattern for tokens.

### Creating lexical Analyzers with



- lex compiler compiles lex.l file and produces a file lex.yy.c
- lex.yy.c program is given as i/p C compiler, it compile this program binary and produces a binary (a) object file called a.out
- Then we need give input to a.out and produces tokens.

### Structure of lex program:

Lex program contains 3-section

- Declarations
- Translation Rules
- Auxiliary Functions

Declarations

%/.

Translation rules

%/.

Auxiliary functions.

### (i) Declarations:

→ Declaration section is mainly useful in order to declare C variables and constants.

Syntax: % {

Variables, constants

% }

Eg: % {

int a, b;

float count=0;

% }

digit [0-9]

letter [A-Za-z]

} RE

→ Declaration section is also used to define the Regular Expression

### (ii) Translation Rules:

→ Translation Rules are used to specify the pattern rules.

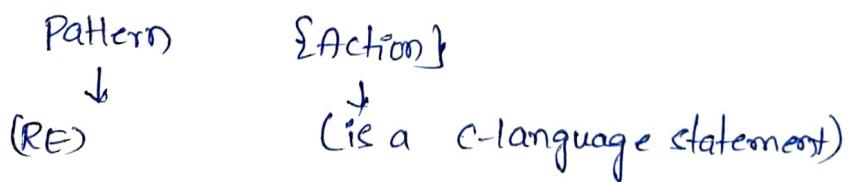
→ Translation Rules are defined in two forms %/ /%

Syntax: %/ /%

Translation rules

%/ /%

→ We can specify the rule in the form of pattern followed by actions



Eg: - %/ /%

Pattern1 {Action1}

Pattern2 {Action2}

||

%/ /%

(ii) Auxiliary Functions → used to define the function.

→ All the functions that are needed are defined in this section.

Eg:- lex program to recognize identifiers, Keywords, Relational operators & Numbers.

lex.l

% {

/\* definition of lex program for recognize tokens  
LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUMBERS, RElop \*/

% }

/\* regular definitions \*/

letter [a-zA-Z]

digit [0-9]

id {letter} ({letter}/{digit})\*

delim [ \t\n ]

ws {delim}+

number {digit}+ (. {digit}+) ? (E [+ -] ? {digit}+) ?

%%

{std}

{ws} /\* no action and no return \*/

if {return (IF);}

then {return (THEN);}

else {return (ELSE);}

{id} yyval = (int) installID(); return (ID);

{number} yyval = (int) installNum(); return (NUMBER);

"<" yyval = LT; return (RElop);

"≤"      {yyval = LE; return(CRELop);}  
 "≡"      {yyval = EQ; return(CRELop);}  
 "<>"      {yyval = NE; return(CRELop);}  
 ">"      {yyval = GT; return(CRELop);}  
 ">="      {yyval = GE; return(CRELop);}  
 "%"     

int installID() { /\* function to install the lexeme, whose first character  
                   is pointed to by yytext, and whose length is yylen,  
                   into the symbol table and return a pointer thereto \*/}  
 int installNum() { /\* similar to Install ID, but puts numerical constant  
                   into a separate table \*/}

Finite Automata

→ Finite Automata

FINITE AUTOMATA

- Finite automata are recognizers; They simply say "yes" or "no" about each possible string.
- For the given language we construct a finite automata, FA is used to check that for the given string is present in the language (or) Not.
- Finite automata is an abstract computing device. It is a mathematical model of a system with discrete inputs, outputs, states and set of transitions from state to state that occurs on input symbols from alphabet  $\Sigma$

There are 2-types of Finite Automata.

### ① NFA (Non-Deterministic Finite automata)

A NFA consists of 5-tuples  $NFA = (Q, \Sigma, \delta, q_0, F)$

$Q(d)$   $\Rightarrow$  A finite set of states

$\Sigma$  = A set of i/p symbols ( $\epsilon$ -is never a member of  $\Sigma$ )

$q_0$  = start (d) initial state

$F$  = set of final states (Accepting state)

$\delta$  = transition function  $= Q \times \Sigma \rightarrow Q$

→ NFA can have null states ( $\epsilon$ -transitions)

→ In NFA on single i/p it is going to multiple states or null states.

→ we can represent NFA or DFA by transition graph.

This graph is very much like to transition diagram except.

(a) A same symbol can label edges from one state to several different states, and

(b) An edge may be labeled by  $\epsilon$ , the empty string

Eg.: Construct a NFA for RE  $(a|b)^*abb$

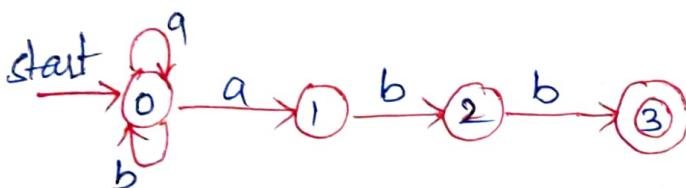


Fig: NFA for string  $(a|b)^*abb$

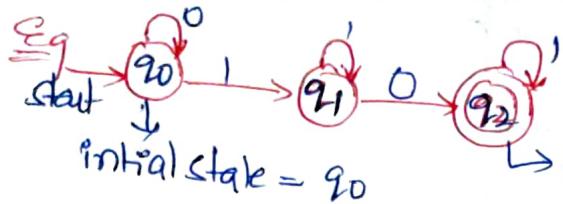
→ It is easy to construct NFA than DFA for a given language

→ Every NFA is not DFA, but each NFA can be translated into DFA

→ In NFA, there exist many paths for specific i/p from the current state to the next state

Transition Graph = (a) Transition diagram

It is a directed graph associated with vertices of the graph corresponds to the state of finite automata.



↳ The states present in b/w initial & final state are called intermediate state

↳ input  $\Sigma = \{0, 1\}$

Transition Table =

→ NFA is also represented by Transition table, whose rows corresponds to state, columns corresponds to i/p symbols.

state	a	b	$\epsilon$
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\{\emptyset\}$	$\emptyset$

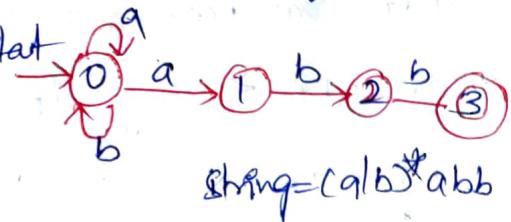


Fig: Transition Table.

Transition Function =

→ The mapping (d) Transition function is denoted by  $\delta$

→ two parameters are passed to transition function:

(i) current state & (ii) i/p symbol

→ Transition function returns a state i.e next state.

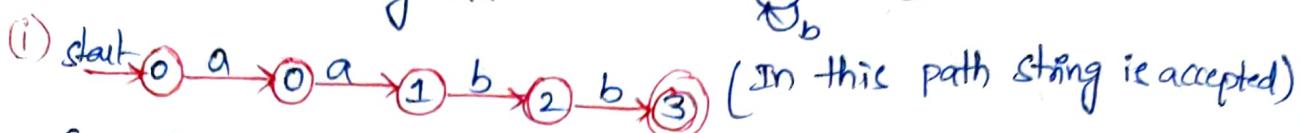
Eg :=  $\delta(q_0, a) = q_1$  (d)  $(q_0, 1) = q_1$

$\downarrow$        $\downarrow$        $\downarrow$   
current state    i/p symbol    next state.

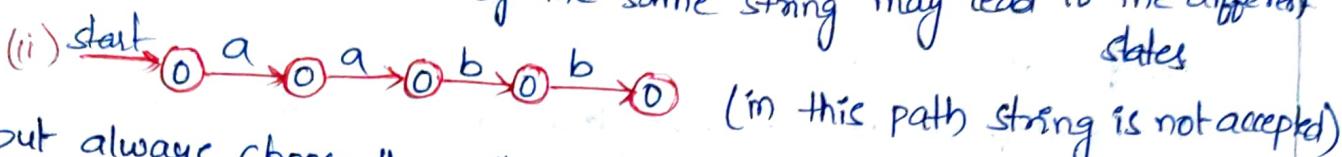
## \* Acceptance of Input string by Automata:

Fig: NFA for  $(a|b)^*abb$

The path labeled by  $aabb$



Several paths labeled by the same string may lead to the different states



but always choose the path which starts from initial state to final state, so the first path is considered.

DFA: (Deterministic FA):

→ DFA is a special case of FA where,

(i) There are no moves on input  $\epsilon$ , and

(ii) For each state and input symbol  $a$ , there is exactly one edge out

of  $S$  labeled  $a$ .

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

$$\delta = Q \times \Sigma = Q$$

$Q$  = A set of finite states

$\Sigma$  = a set of input symbols

$q_0$  = initial (or) start state

$F$  = Final state

$\delta$  = Transition function

Construction of DFA:

Eg#1: String ending with a substring.

Step 1: Decide the min no. of states required to construct DFA

substring length =  $n$  no of states =  $n+1$

Step 2: Decide the string for which you will construct DFA.

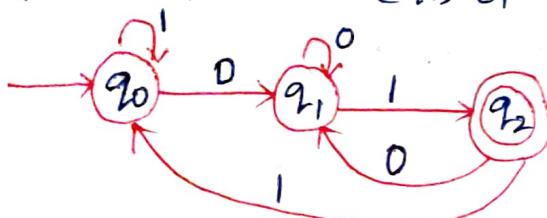
Step 3: Construct the DFA for above decided string (always prefer to go with

Step 4: After construction of DFA, send the left over possible combinations to the starting state. Existing path)

the starting state.

Eg1: String ending with "01" over  $\Sigma = \{0, 1\}$

Regular Expression =  $(0/1)^*01$



min no. of states = 3 substring  $\rightarrow 01$

001

0101

[NOTE]: Always prefer to go with the existing path, create a new path only when can't find a path to go with]

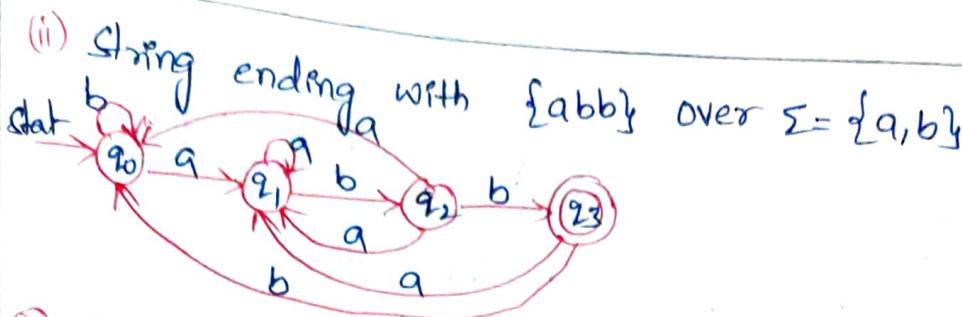


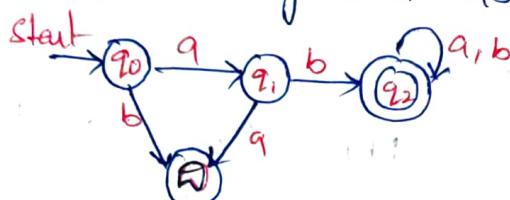
abb  
aabbb  
ababb  
abbabb

Q) String starting with substring.

→ NO of states =  $n+2$

→ In this left over possible combinations are connected to dead state

Eg 1: String starting with "ab" over  $\Sigma = \{a, b\}$

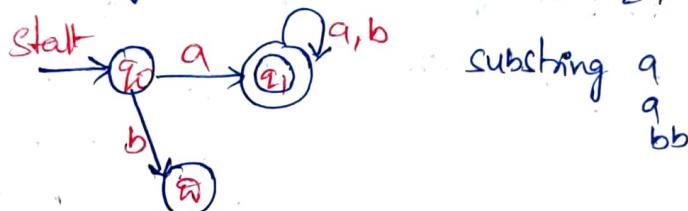


$RE = ab(a/b)^*$  Substring ab

no. of states =  $2+2 = 4$

ab  
ab  
a  
b

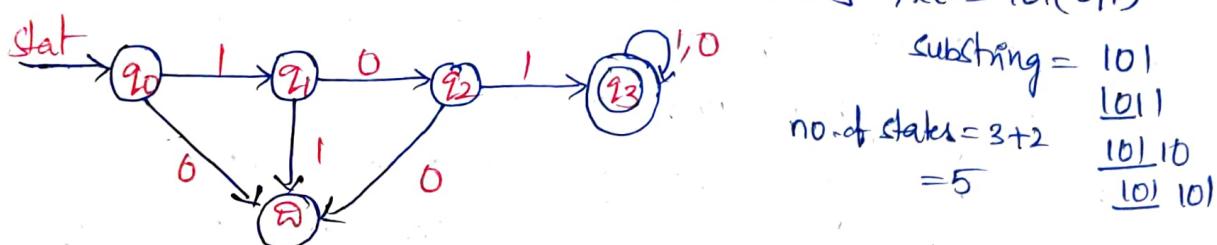
Eg 2: String start with "a" over  $\Sigma = \{a, b\}$   $RE = a(a/b)^*$



substring a  
a  
bb

length of  
no. of states = substring + 2  
= 1 + 2  
= 3

Eg 3: String start with "101" over i/p  $\Sigma = \{0, 1\}$ ,  $RE = 101(0/1)^*$



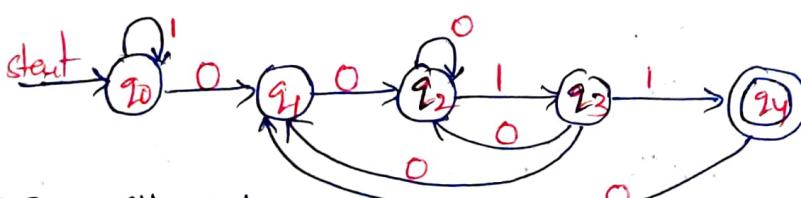
substring = 101

no. of states = 3 + 2 = 5

101  
101  
10  
101  
101

DFA Eg: Construct an DFA that should start with "0011" over i/p  $\Sigma = \{0, 1\}$

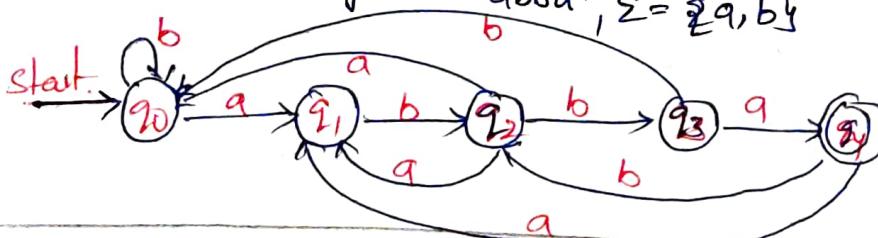
$RE = 0011(0/1)^*$



0011

0 0011  
0 00011  
0 01001  
00110011

Eg: DFA with starting with "abba",  $\Sigma = \{a, b\}$



## From Regular Expression to Automata:

Conversion of an NFA to DFA:

Eq: ① subset construction method:

Eq: The string end with "ab" over alphabet  $\Sigma = \{a, b\}$



Fig: NFA

Step 1: we have to write all the subset of states

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

$$\text{Subset} = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

If the given set NFA contain ~~set~~ states n elements, the no. of possible subsets are  $= 2^n$

$$\text{Here } n=3, \text{ subsets} = 2^3 = 8$$

Step 2: transition table for DFA.

States	Input	
	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow q_0$	$\{q_0, q_1\}$	$q_0$
$q_1$	$\emptyset$	$q_2$
$\star q_2$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\star \{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\star \{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$\star \{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

→ Now, we are going to eliminate the unwanted states.

Step 3: construction of DFA transition table

states	input	
	a	b
→ $q_0$	{ $q_0, q_1$ }	{ $q_0$ }
{ $q_0, q_1$ }	{ $q_0, q_1$ }	{ $q_0, q_2$ }
* { $q_0, q_2$ }	{ $q_0, q_1$ }	{ $q_0$ }

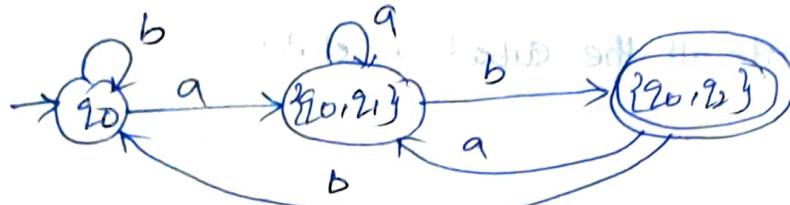
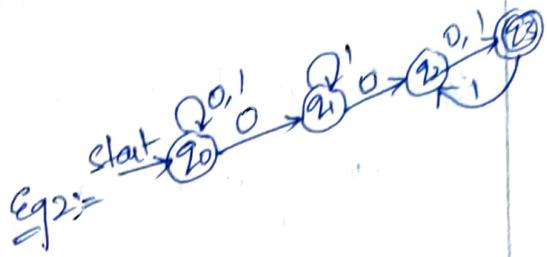
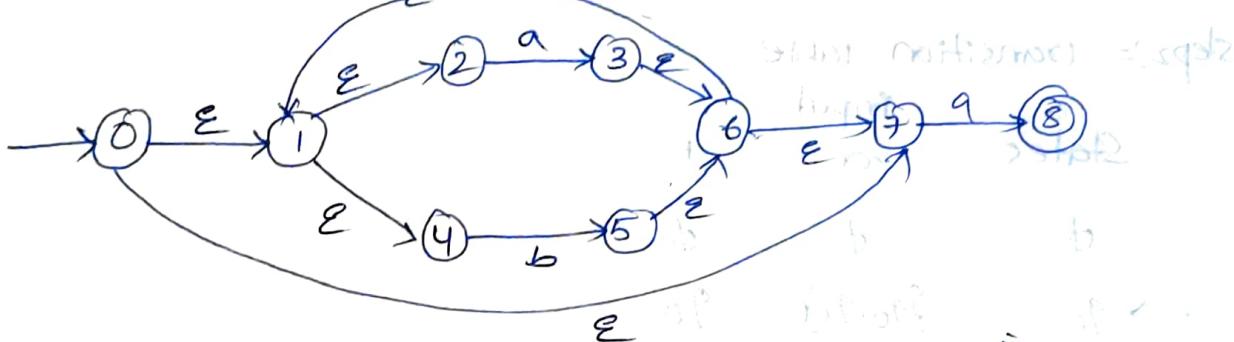


Fig: DFA

②  $\epsilon$ -NFA to DFA =

$$RE = (a+b)^* a \quad \Sigma = \{a, b\}$$



Eg2: NFA for  $(a+b)^* a$

$$A = \epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$$

$\epsilon$ -closure = All  $\epsilon$ -path of the given state is called as  $\epsilon$ -closure

$$\delta(A, a) = \epsilon\text{-closure}(\delta(0, 1, 2, 4, 7), a)$$

$$= \epsilon\text{-closure}(\delta(0, a) \cup \delta(1, a) \cup \delta(2, a) \cup \delta(4, a) \cup \delta(7, a))$$

$$= \epsilon\text{-closure}(3, 8) = \{3, 6, 1, 7, 2, 4, 8\} = 18$$

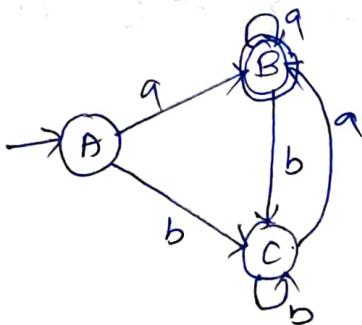
$$\begin{aligned}
 \delta(A, b) &= \text{\varepsilon-closure } (\delta(0, 1, 2, 4, 7), b) \\
 &= \text{\varepsilon-closure } (\delta(0, b) \cup \delta(1, b) \cup \delta(2, b) \cup \delta(4, b) \cup \delta(7, b)) \\
 &= \text{\varepsilon-closure } (5) \\
 &= \text{\varepsilon-closure } \{5, 6, 7, 1, 2, 4\} = C
 \end{aligned}$$

$$\begin{aligned}
 \delta(B, a) &= \text{\varepsilon-closure } (\delta(3, 6, 1, 7, 2, 4, 8), a) \\
 &= \text{\varepsilon-closure } (\delta(3, a) \cup \delta(6, a) \cup \delta(1, a) \cup \delta(7, a) \cup \delta(2, a) \cup \\
 &\quad \delta(4, a) \cup \delta(8, a)) \\
 &= \text{\varepsilon-closure } (8, 3, 8) = B \checkmark \text{ final state}
 \end{aligned}$$

$$\begin{aligned}
 \delta(B, b) &= \text{\varepsilon-closure } (\delta(3, 6, 1, 7, 2, 4, 8), b) \\
 &= \text{\varepsilon-closure } (\delta(3, b) \cup \delta(6, b) \cup \delta(1, b) \cup \delta(7, b) \cup \delta(2, b) \cup \\
 &\quad \delta(4, b) \cup \delta(8, b)) \\
 &= \text{\varepsilon-closure } (5) = (5, 6, 1, 2, 4, 7) = C
 \end{aligned}$$

$$\begin{aligned}
 \delta(C, a) &= \text{\varepsilon-closure } (\delta(5, 6, 1, 2, 4), a) \\
 &= \text{\varepsilon-closure } (\delta(5, a) \cup \delta(6, a) \cup \delta(1, a) \cup \delta(2, a) \cup \\
 &\quad \delta(4, a)) \\
 &= \text{\varepsilon-closure } (8, 3) = B
 \end{aligned}$$

$$\begin{aligned}
 \delta(C, b) &= \text{\varepsilon-closure } (\delta(5, 6, 1, 2, 4), b) \\
 &= \text{\varepsilon-closure } (\delta(5, b) \cup \delta(6, b) \cup \delta(1, b) \cup \delta(2, b) \cup \delta(4, b)) \\
 &= \text{\varepsilon-closure } (5) = C
 \end{aligned}$$



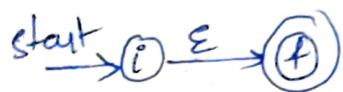
DFA Transition Table

	a	b
A	B	C
B	B	C
C	B	C

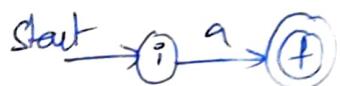
Fig: DFA

## Construction of an NFA from Regular Expression

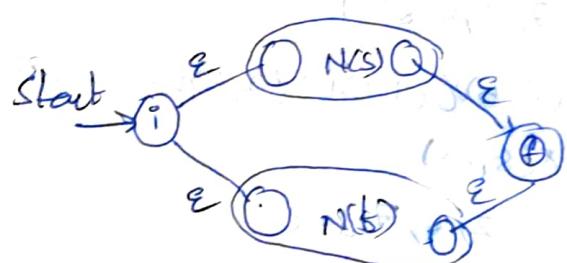
(1) for expression  $\epsilon$  construct NFA (Thompson construction method)



(2) for any subexpression  $a$  in  $\Sigma$ , construct NFA



(3) NFA for the Union of 2 regular expressions



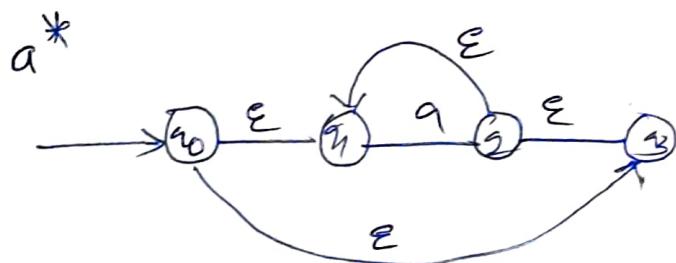
$$\text{Eg } 1 = (a+b) = (a|b)$$

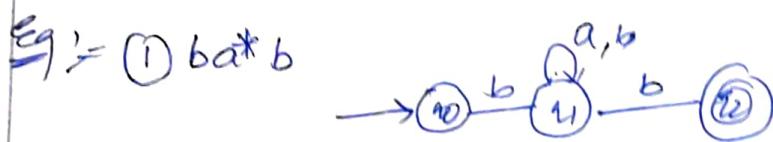
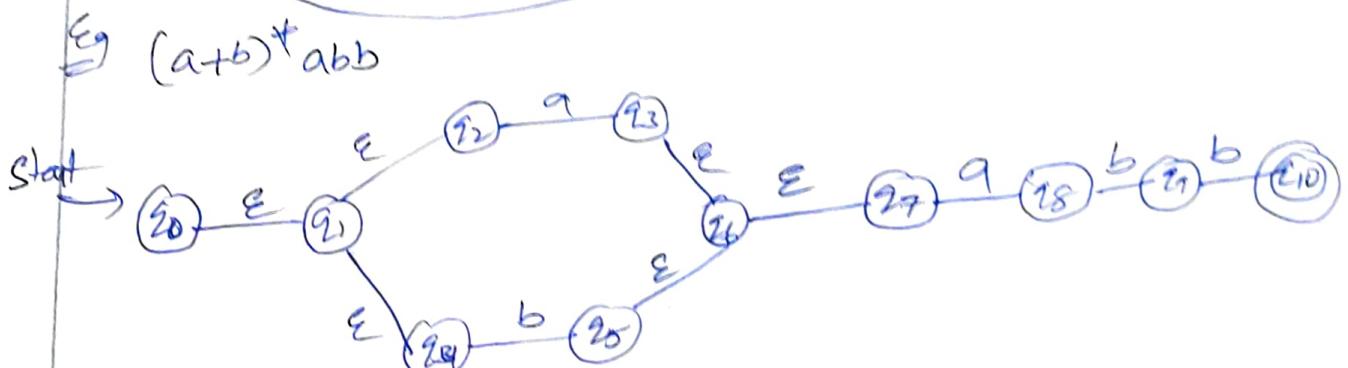
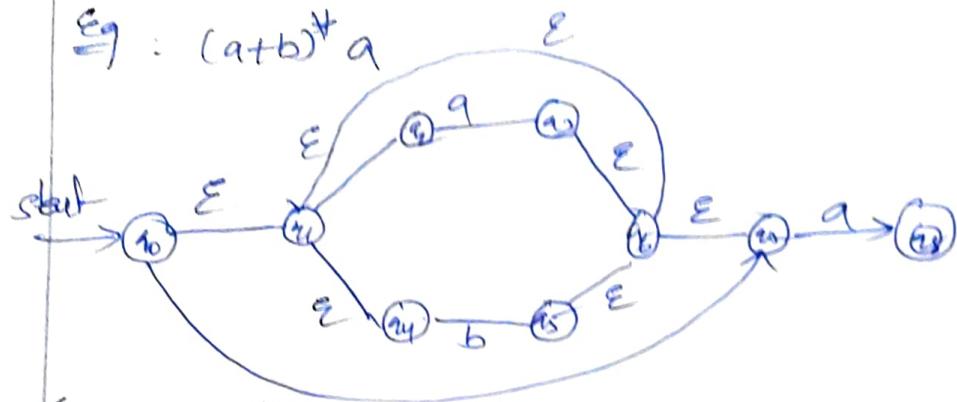
(4) NFA for the concatenation of 2 regular expression



$$\text{Eg } 2 = \text{start} \rightarrow q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \quad (\text{or}) \quad \text{start} \rightarrow q_0 \xrightarrow{a} q_1 \xrightarrow{\epsilon} q_2 \xrightarrow{b} q_3$$

(5) NFA for the closure of RE

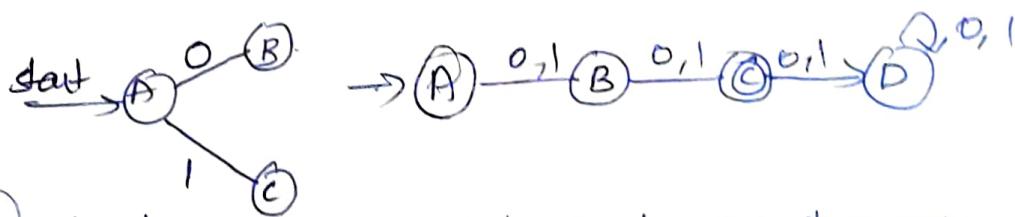




Examples for construction of DFA:

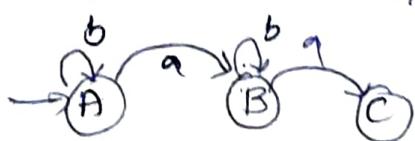
①  $L_1$  = set of all strings start with 0    Eg: 001

② construct the DFA that accepts all string over  $\{0, 1\}$  of length 2  
 $\Sigma = \{0, 1\}$      $L = \{00, 01, 10, 11\}$

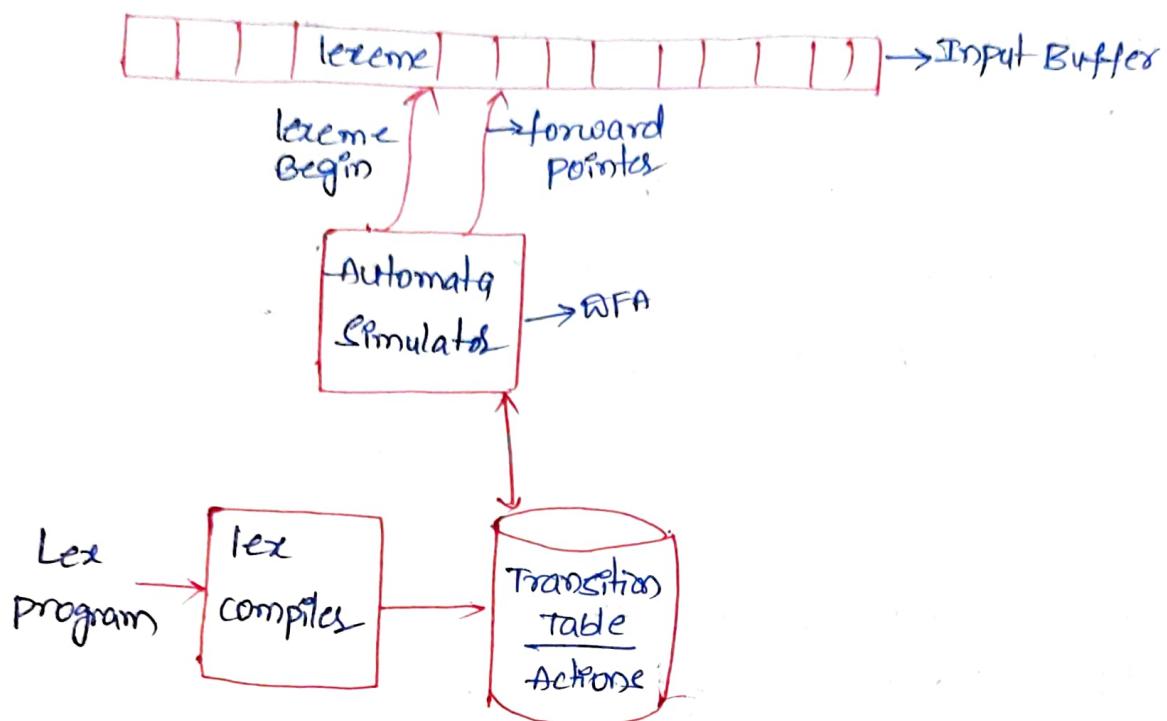


③ construct a DFA that accepts any string over  $\{a, b\}$  that doesn't contain the string aabb in it     $\Sigma = \{a, b\}$

i) const DFA that accepts aabb



## The Structure of the Generated Analyzers:



- This structure describes how an finite automata simulator is working for the lexical Analyzers
- lex program simulates an automata.
- lex program consists of all RE definition and Translation Rules.
- Once a lex program is compiled the Regular Expressions are converted into the finite automata.
- The translation rules are turned into transition table and actions which are used by the a Finite automata simulator, to recognize the tokens.
- Finite automata simulator checks <sup>any</sup> i/p strings that are stored i/p buffer, whether is matching with any pattern, if it matches then it will produce corresponding Token as o/p.

Automata Simulator uses the following components

- (i) Transition Table for automata.
- (ii) Function are passed directly to be through lex to o/p.
- (iii) The action from i/p, which appear as fragments of code to be invoked at the appropriate time by automata simulator

To construct an automata:-

1. convert the regular expression patterns in the lex program to the NFA.
2. we need a single automata that will recognize lexemes matching any of the patterns in the program.
3. we combine all the NFA into one by introducing a new start with  $\epsilon$ -transition.

Eg:- a {action A<sub>1</sub> for pattern P<sub>1</sub>}  
abb {action A<sub>2</sub> for pattern P<sub>2</sub>}  
 $a^*b^*$  {action A<sub>3</sub> for pattern P<sub>3</sub>}

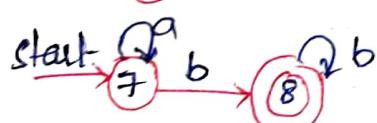


Fig : NFAs for a, abb, and  $a^*b^*$

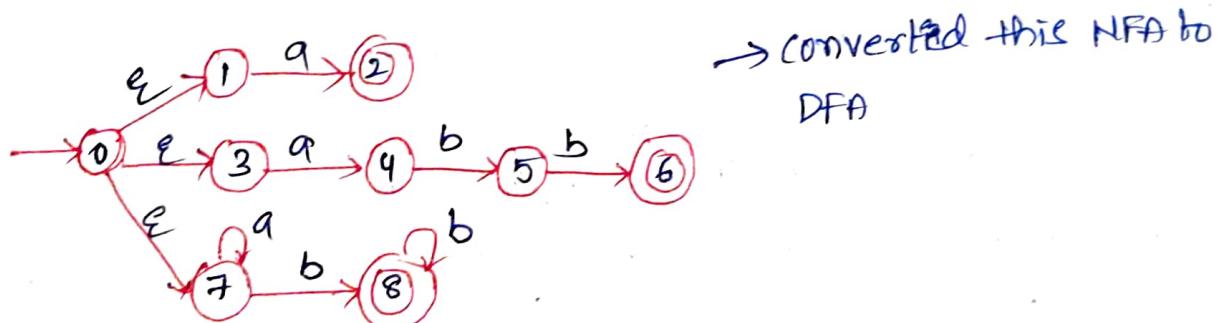
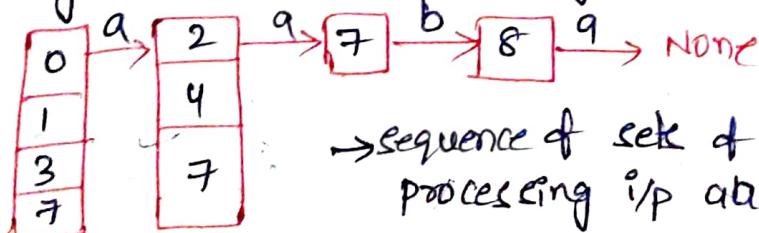


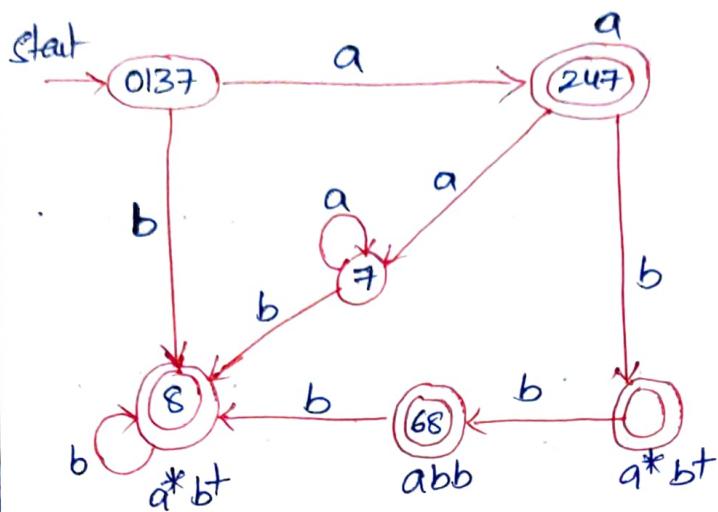
Fig : Combined NFA

i/p string abba pattern matching Based on NFA:-



→ sequence of set of states entered when processing i/p abba

DFA for the above NFA



Eg: Transition graph for DFA handling the patterns  $a$ ,  $abb$ , &  $a^*bt$

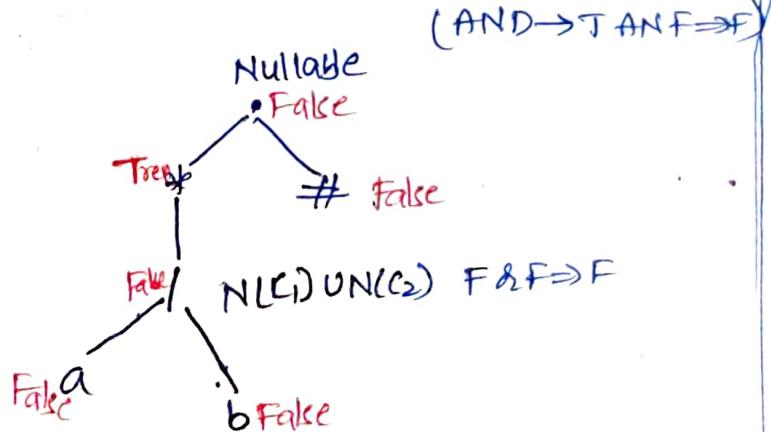
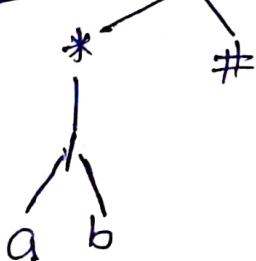
### OPTIMIZATION OF DFA-BASED PATTERN MATCHERS

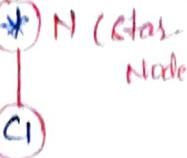
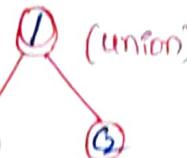
- Convert the Regular Expression into DFA without creating NFA.
- \* Introduce Augmented RE.
- \* Construct a syntax tree for RE.
- \* Number the leaf nodes.
- \* Traverse the syntax tree to construct functions Nullable, Firstpos and Lastpos for each node.
- \* Compute Followpos of each node.
- \* Converting to DFA.

Eg:  $(a/b)^*$

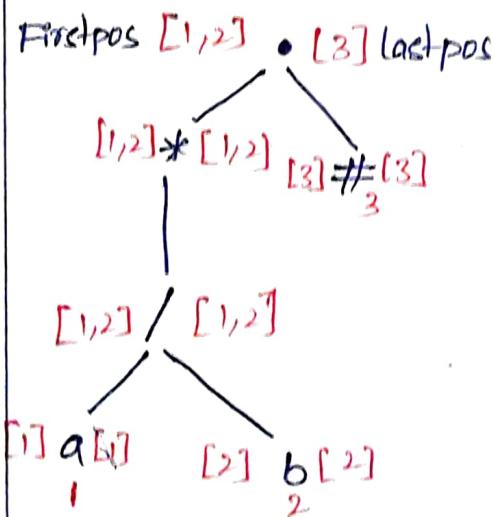
Step 1: (Augmented grammar)  $(a/b)^* \cdot \#$

Step 2: Syntax tree:



Node	Nullable	Firstpos(n)	Lastpos(n)
(i) $N \rightarrow \text{leaf } \epsilon$	True	$\emptyset$	$\emptyset$
(ii) $N \rightarrow \text{leaf } (i)$	False	$\{i\}$	$\{i\}$
(iii) 	True	Firstpos(C1)	Lastpos(C1)
(iv) 	Nullable(C1) or Nullable(C2)	Firstpos(C1) $\cup$ Firstpos(C2)	Lastpos(C1) $\cup$ Lastpos(C2)
(v) 	Nullable(C1) and Nullable(C2)	if (nullable(C1)) is True Firstpos(C1) $\cup$ Firstpos(C2) else Firstpos(C1)	if (nullable(C2)) is True lastpos(C1) $\cup$ lastpos(C2) else lastpos(C2)

Firstpos and lastpos computations:



Node	Followpos
$1 \rightarrow a$	$\{1, 2, 3\}$
$2 \rightarrow b$	$\{1, 2, 3\}$
$3 \rightarrow \#$	"

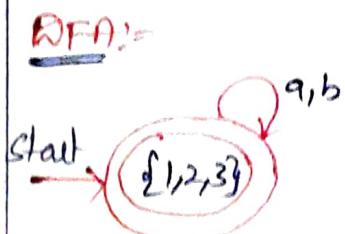
$$DFA = (\Omega, \Sigma, \delta, q_0, F)$$

$$q_0 \rightarrow (1, 2, 3) \quad \Sigma = (a, b) \quad F \rightarrow 1, 2, 3$$

$$\delta \rightarrow ((\{1, 2, 3\}, a), (\{1, 2, 3\}, b))$$

$$\downarrow \quad \downarrow$$

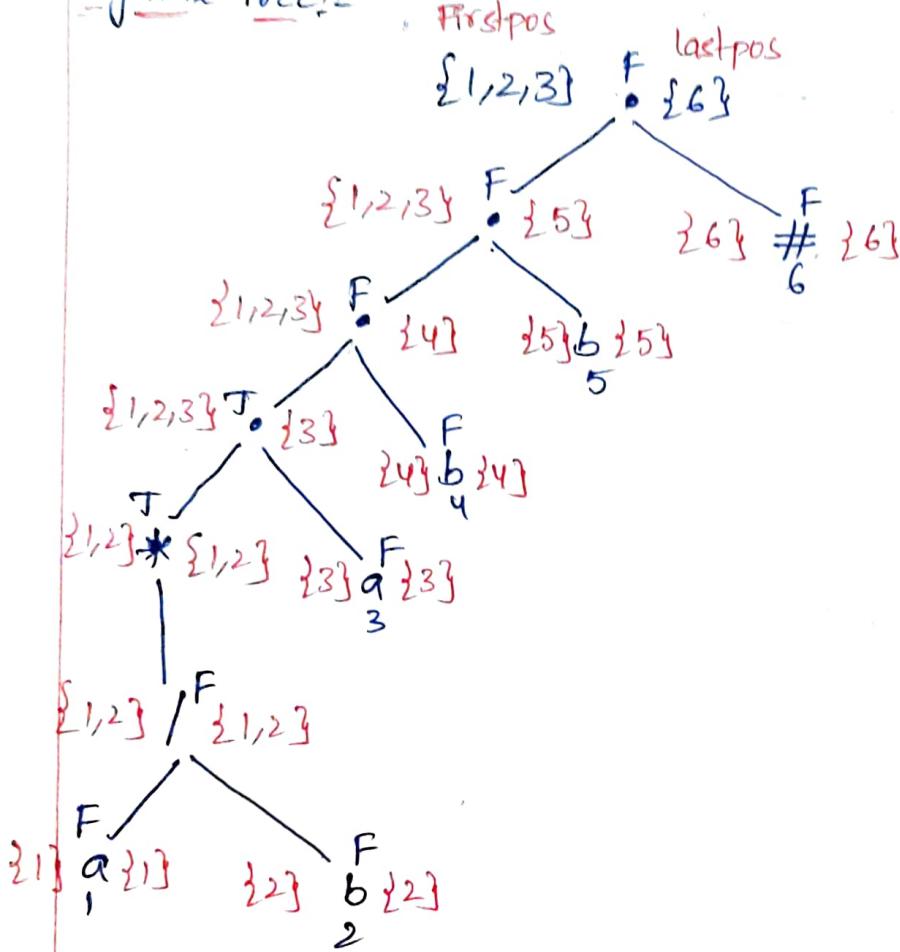
$$(\{1, 2, 3\}, \{1, 2, 3\})$$



Eg 2:  $R \rightarrow (a/b)^* \cdot abb$

Augmented Grammars:  $(a/b)^* \cdot a \cdot b \cdot b \#$

Syntax Tree:



→ For finding Followpos start (\*) and concatenation (.) nodes will be considered.

Star Node: (\*) → if (n is star-node)

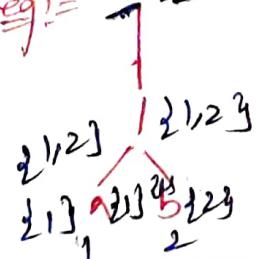
for (each i in lastpos(n))

$$\text{followpos}(i) = \text{followpos}(i) \cup \text{first}(n);$$

→ If n - is a star-node and i "i" is a position in last pos(n),

then all position in firstpos(n) are in followpos(i)

Eg:  $\frac{\text{firstpos}(1, 2)}{\{1, 2\} * \{1, 2\}} \rightarrow \text{lastpos}(n)$



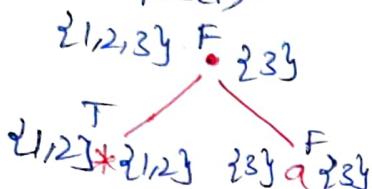
Node followpos()

1	{1, 2}
2	{1, 2}

(i) cat Node (.): If ( $n == c_1 \cdot c_2$  (concat-node))  
 for (each  $i$  in  $\text{lastpos}(c_1)$ )  
 $\text{followpos}(r) = \text{followpos}(i) \cup \text{firstpos}(c_2)$

→ If  $n$  is cat-node with left child  $c_1$  & Right child  $c_2$ , then for every position "i" in  $\text{lastpos}(c_1)$ , all position in  $\text{firstpos}(c_2)$  are in  $\text{followpos}(i)$

$\text{followpos}(i)$ :



Node	followpos
$c_1$	$\{1,2\} \cup \{1,2,3\}$
$\text{lastpos}$	$\{2\}$

Node

followpos

$1 \rightarrow a$   $1,2,3$

$2 \rightarrow b$   $1,2,3$

$3 \rightarrow a$   $4$

$4 \rightarrow b$   $5$

$5 \rightarrow b$   $6$

$6 \rightarrow \#$   $-$

Initial state → 1st Root Node first position  $\{1,2,3\} \Rightarrow A$

$\Sigma = (a, b)$

(B)

1)  $\delta(A, a) \rightarrow \delta(\{1,2,3\}, a) \Rightarrow \{1,3\} = \text{Followpos}(1) \cup \text{Followpos}(3) = \{1,2,3,4\}$

2)  $\delta(A, b) \rightarrow \delta(\{1,2,3\}, b) = \{2\} = \text{Followpos}(2) = \{1,2,3\} \Rightarrow A$

3)  $\delta(B, a) \rightarrow \delta(\{1,2,3,4\}, a) \Rightarrow \{1,3\} = \text{Followpos}(1) \cup \text{Followpos}(3) = \{1,2,3,4\} \Rightarrow B$

4)  $\delta(B, b) = \delta(\{1,2,3,4\}, b) \Rightarrow \{2,4\} = \text{Followpos}(2) \cup \text{Followpos}(4) = \{1,2,3,5\} \Rightarrow C$

5)  $\delta(C, a) = \delta(\{1,2,3,5\}, a) = \{1,3\} = \text{Followpos}(1) \cup \text{Followpos}(3) = \{1,2,3,4\} \Rightarrow B$

6)  $\delta(C, b) = \delta(\{1,2,3,5\}, b) = \{2,5\} = \text{Followpos}(2) \cup \text{Followpos}(5) = \{1,2,3,6\} \Rightarrow D$

7)  $\delta(D, a) = \delta(\{1,2,3,6\}, a) = \{1,3\} = \text{Followpos}(1) \cup \text{Followpos}(3) = \{1,2,3,4\} \Rightarrow B$

8)  $\delta(D, b) = \delta(\{1,2,3,6\}, b) = \{2\} = \text{Followpos}(2) = \{1,2,3\} \Rightarrow A$



(20)

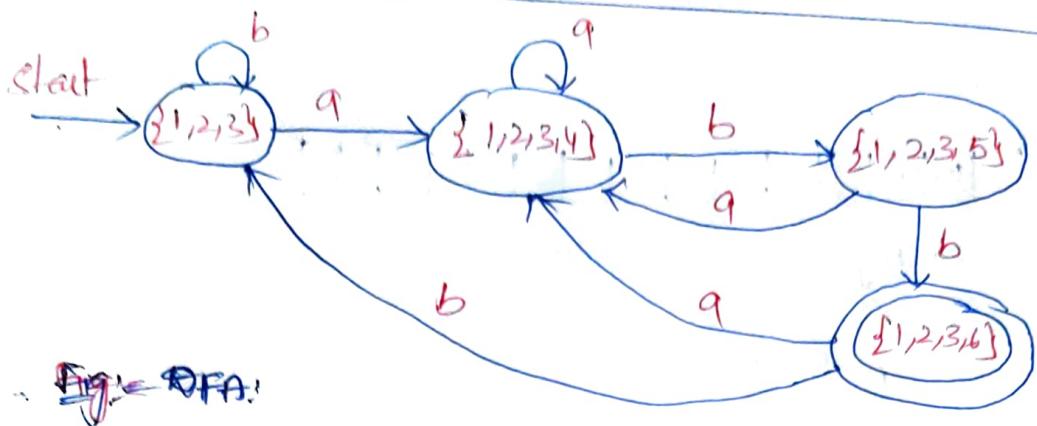


Fig: DFA:

DFA states	a	b
$\{1, 2, 3\} = A$	B	A
$\{1, 2, 3, 4\} = B$	B	C
$\{1, 2, 3, 5\} = C$	B	D
$\{1, 2, 3, 6\} = D$	B	A

Fig: DFA Transition Table