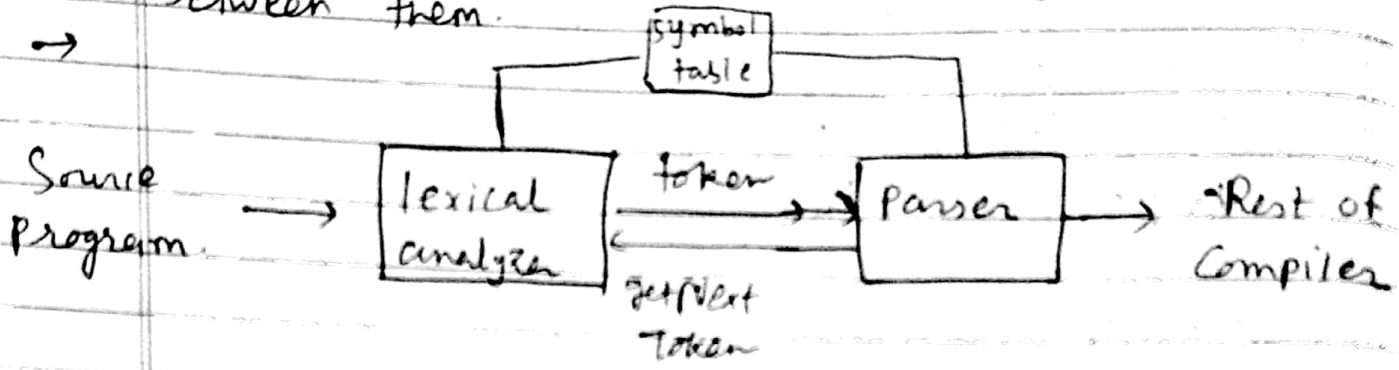
Chapter : 2 Lexical Analyzer

→ Role of lexical analysis and its issues:
→ How do parser and scanner communicate. Explain with the block diagram communication between them.

→



→ lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that parser uses for syntax analysis.

→ Upon receiving "get next token" command from parser, the lexical analyzer reads the input character until it can identify the next token.

→ lexical analysis removes the blank, whitespace or comment from source program.

Issues in lexical analysis:
There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:
→ Compiler efficiency is improved.
→ Compiler portability is enhanced.
→ Simpler design is perhaps the most important consideration.

lexical error : Misspelling of identifiers, keyword.
Syntax error : grammatical rules, missing operator
Semantic error : type mismatch, undeclared variable.

Symbol table :

→ Symbol table is a data structure used by language translator such as Compiler or interpreter.

→ It is used to store names encountered in the source program, along with the relevant attributes for those names.

→ information about following entities is stored in symbol table
  → variable / identifier
  → procedure / function
  → keyword
  → Constant
  → class name
  → label name

Input buffer :
Temporary Storage :
→ We use buffer divided into two N-character halves
→ Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme. forward pointer,
→ Input buffer is needed because lexical analyzer does not rely on white space or menks.

We have two buffer Scheme:

1) Buffer pairs
2) Sentinels.

1) Buffer pairs:

→ Scheme: Consist of two buffers, each consists of N - Character size which are reloaded alternatively.

→ N - Numbers of characters on one disk block e.g. 4096 bytes

→ N Characters are read from input file to the buffer using one System read command.

→ eof is inserted at the end if number of character is less than N.

Two pointers are there / maintained

1 lexeme begin
2 forward

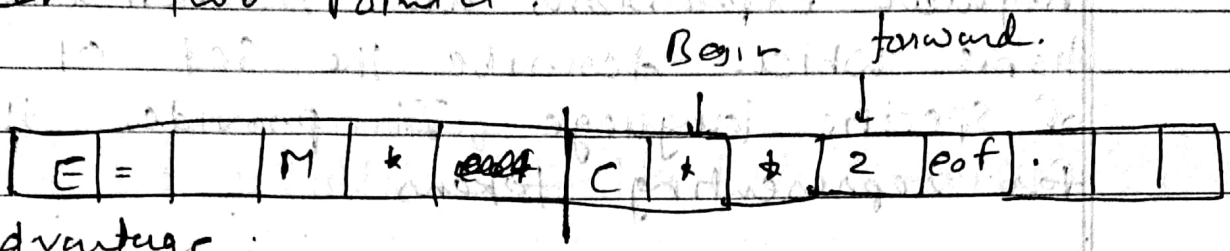
1 lexeme begin points to the beginning of current lexeme which is yet to be found

2 forward: scans ahead until a match for pattern is found.

→ once lexeme is found, lexeme begin is set to a character Immediately after the lexeme which is just found.

forward Scans ahead until a match for a pattern is found.

→ Current lexeme is the set of characters between two pointer:



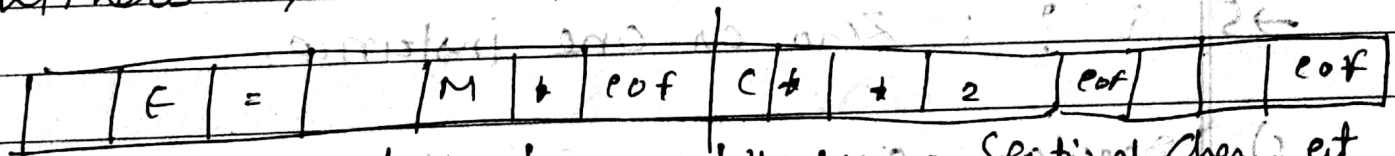
disadvantage:

→ ~~Two~~ test required

1 Whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input and move forward to the begining of the newly loaded buffer.

2 What character read

Sentinels: / Techniques for speeding up the process.



Reduces two test to one by using Sentinel character at end. We can combine the buffer end test with the test for the current character. if we extend each buffer to hold Sentinel Character At the end.

eof → use as a marker for the end of entire input

eof appears other then the end of buffer means that the input is at end.

lexical analysis searches for the pattern defined by language rules.

PAGE No.
DATE

Regular expression:

→ Sequence of character that define search pattern / R.E. is an important notation for specifying patterns.

→ Regular expressions are mathematical Symbolisms which describe the set of strings of specific language. It provides the notation for representing tokens.

Operation on languages:

1 $r|s$: Union : r or s : a|b

2 $r.s$: Concatenation : r followed by s : ab

3 $(r)^*$: 0 or more strings : kleen star instance matching r

4 $(r)^+$: one or more strings : Positive star matching r

→5 $r?$: Zero or one instance.

1) Regular expression for identifier:

: letter (letter + digit)* : l(l+d)*

2) 0 or 1 = 0+1

3) String having zero or more a : a^*

4) String having one or more a : a^+

5) All binary string : $(0+1)^*$

Regular definition :
A Regular definition gives names to certain regular expressions and uses those names in other regular expressions.

for identifier : set of strings of letters and digits beginning with a letters.

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

digit → 0 | 1 | 2 | ..... | 9

id → letter (letter + digit)*

language : Set of character which make string.

ex. $\Sigma = \{a, b\} \mid \{0, 1\}$

→ R.E. for any no. of a and b. except ε/Λ.

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

→ $(a + b)^+$

→ R.E for string Stenting with a

$a(a + b)^*$

→ R.E for String Start With 1 and end With 0.

→ $1(0+1)^*0$

→ odd no. of 1

$0^*(1\,0^*1\,0^*)^*\,1\,0^*$

→ even no of 1

$(0^* \mid 0^*1\,0^*1\,0^*)^*$

Finite automata (NFA and DFA)

→ FA or finite state machine is a 5-tuple
(S, Σ, S_0, F, δ)

S : finite set of states.
// { q_0, q_1, ... }

Σ : finite alphabet of input symbol.

S_0 : Initial state / starting state

F : Set of accepting state / final state

δ : transition function / next state gives.

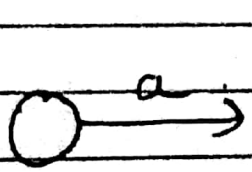
$\delta(q_0, a) = q_1$

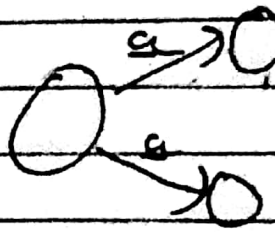
There are two types of finite automata.

1 Deterministic finite automata:
have for each state exactly one edge leaving out for each symbol

2 NFA:
are the others kind. There are no restrictions on the edges of leaving a state. There can be several with same symbol as label and some edges can be labeled with ε

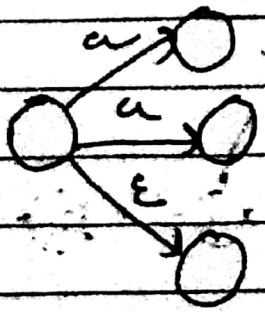
DFA: a:

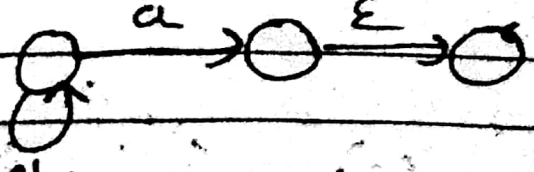




wrong.

2 NFA:



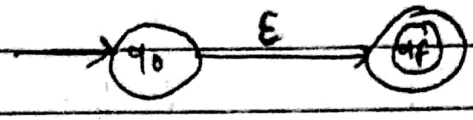


Construction of NFA from regular expression

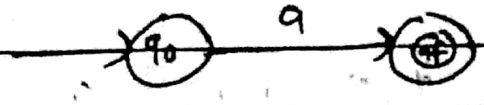
Thompson rule:

1) $r = \epsilon$

regular expression : r

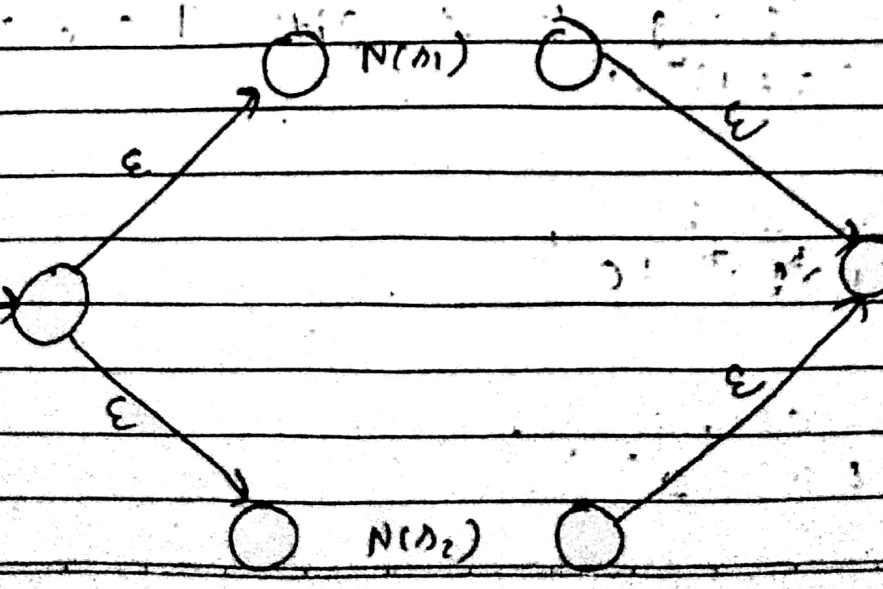


2) $r = a$

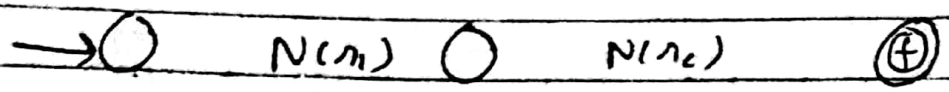


3) $r = r_1 + r_2$

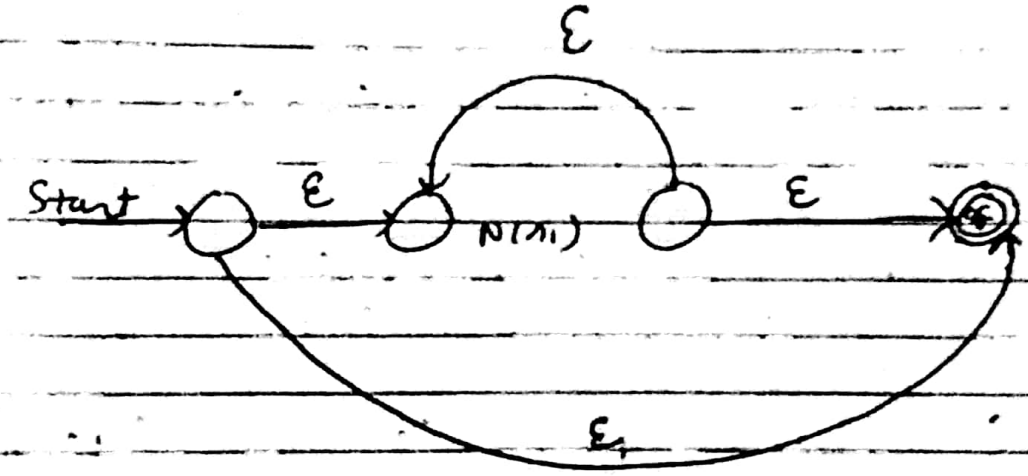
N(r1) : NFA for r1



4) $r = r_1 . r_2$

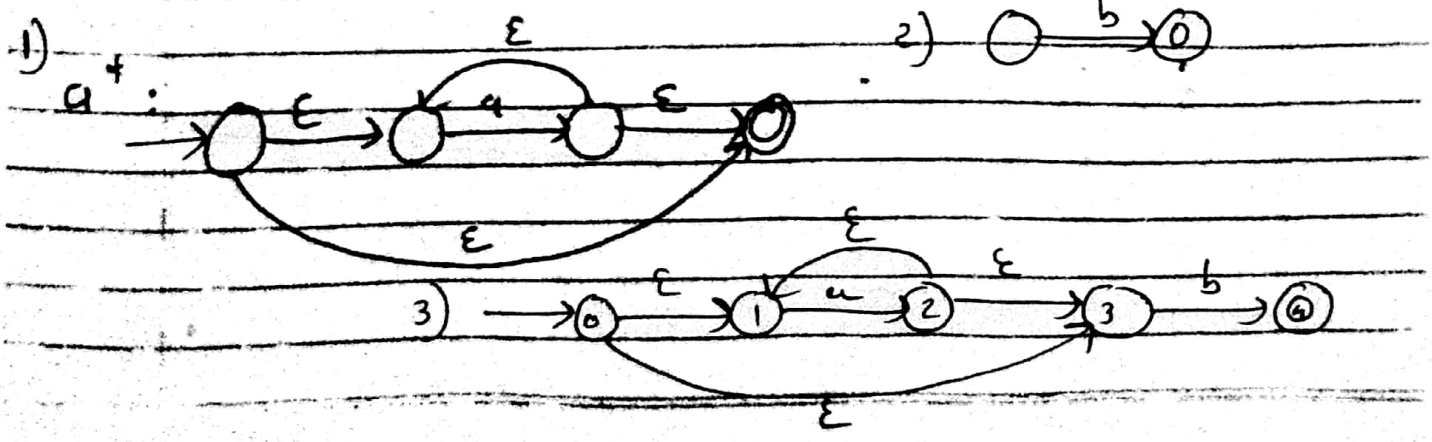


5) $r = (r_1)^*$



Construct NFA equivalent to $r = a^* b$ using Thompson's notation:

→ 1 $a^* b$

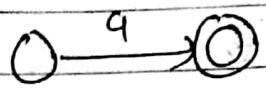


Scanned by CamScanner

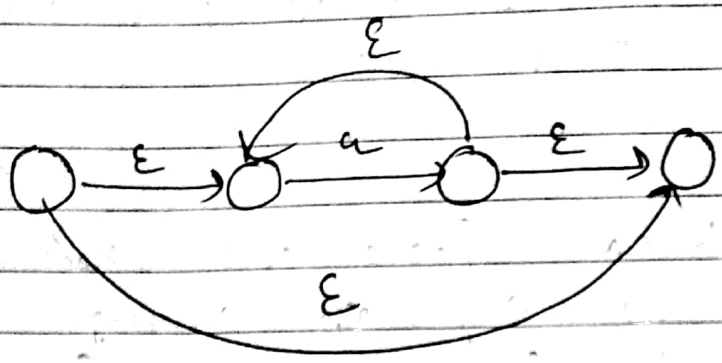
Thompson's notation:

2 $a a^* (b+c) a^k c \#$

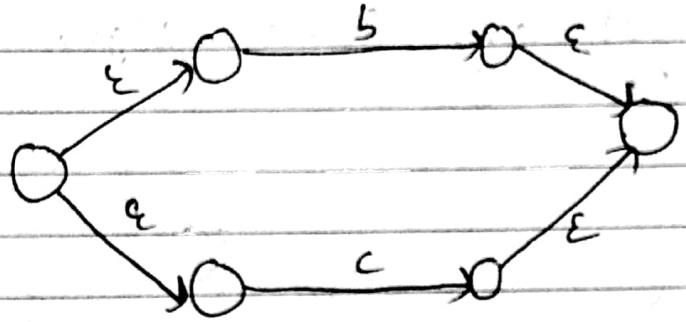
a



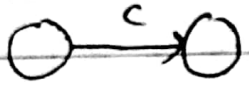
a^*



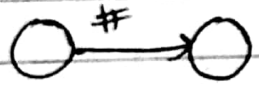
b + c

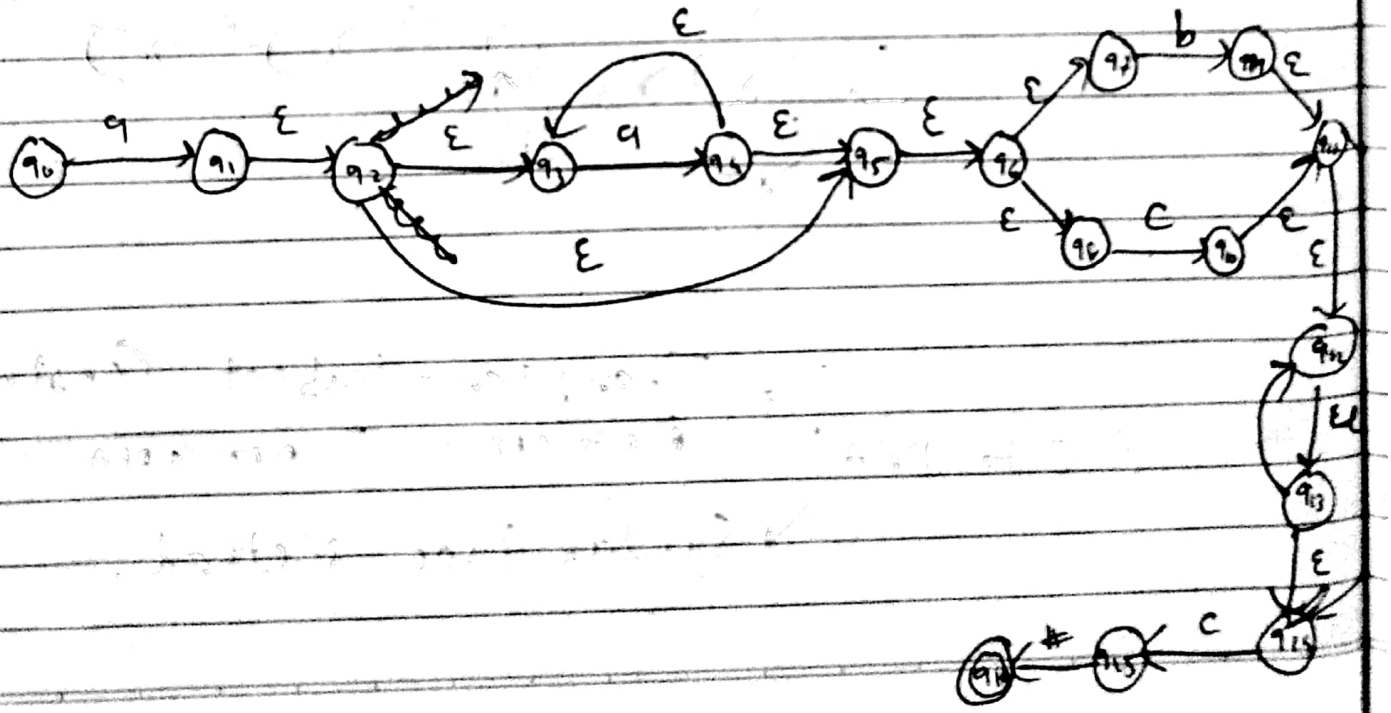


c

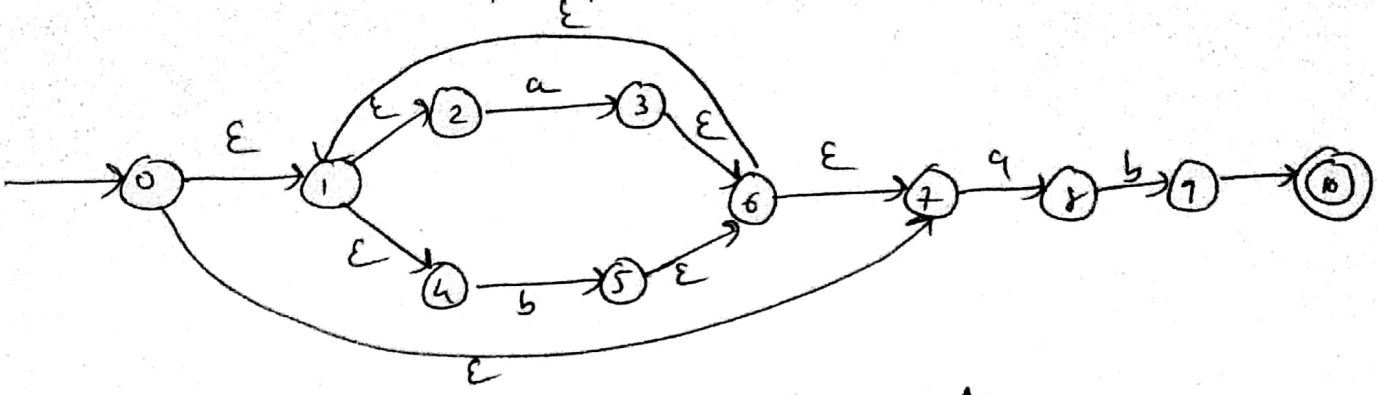


#





$(a+b)^* abb$: Convert it into DFA using thompson - subset Construction method.



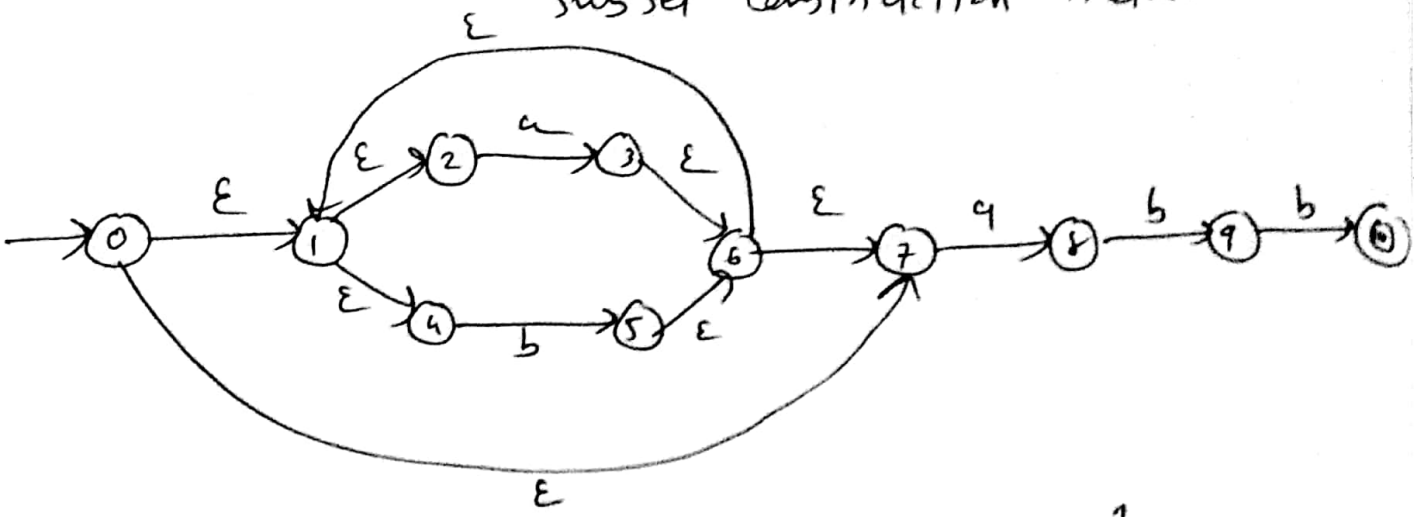
ε-closure (0) : {0, 1, 2, 4, 7} → A
(1) : {1, 2, 4}
(2) : {2}
(3) : {3, 6, 1, 2, 4, 7}
(4) : {4}
(5) : {5, 6, 1, 2, 4, 7}
(6) : {6, 1, 2, 4, 7}
(7) : {7}
(8) : {8}
(9) : {9}
(10) : {10}

Transition table :

	a	b
A {0, 1, 2, 4, 7}	ε-closure (3, 8) = {3, 8, 6, 1, 2, 4, 7} B	ε-closure (5) : {5, 6, 1, 2, 4, 7} C
B {3, 8, 6, 1, 2, 4, 7}	ε-closure (3, 8) : {3, 8, 6, 1, 2, 4, 7} B	ε-closure (9, 5) {9, 5, 6, 1, 2, 4, 7} D
C {5, 6, 1, 2, 4, 7}	ε-closure (3, 8) : {3, 8, 6, 1, 2, 4, 7} B	ε-closure (5) : {5, 6, 1, 2, 4, 7} C
D {9, 5, 6, 1, 2, 4, 7}	ε-closure (3, 8) : {3, 8, 6, 1, 2, 4, 7} B	ε-closure (10, 5) : {10, 5, 6, 1, 2, 4, 7} E
E {10, 5, 6, 1, 2, 4, 7}	ε-closure (3, 8) : {3, 8, 6, 1, 2, 4, 7} B	ε-closure (5) {5, 6, 1, 2, 4, 7} C

→ Draw DFA :→

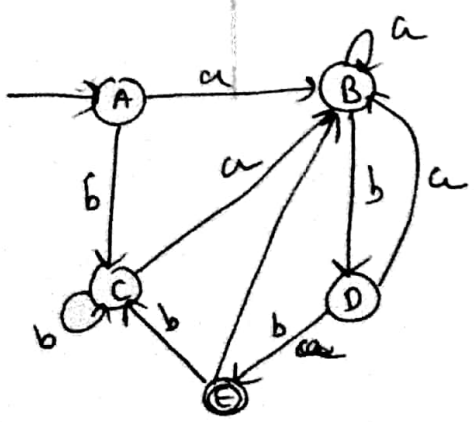
2) $(a+b)^* abb$: Convert it into DFA using thompson subset Construction method.



ε-closure (0) = {0, 1, 2, 4, 7}.

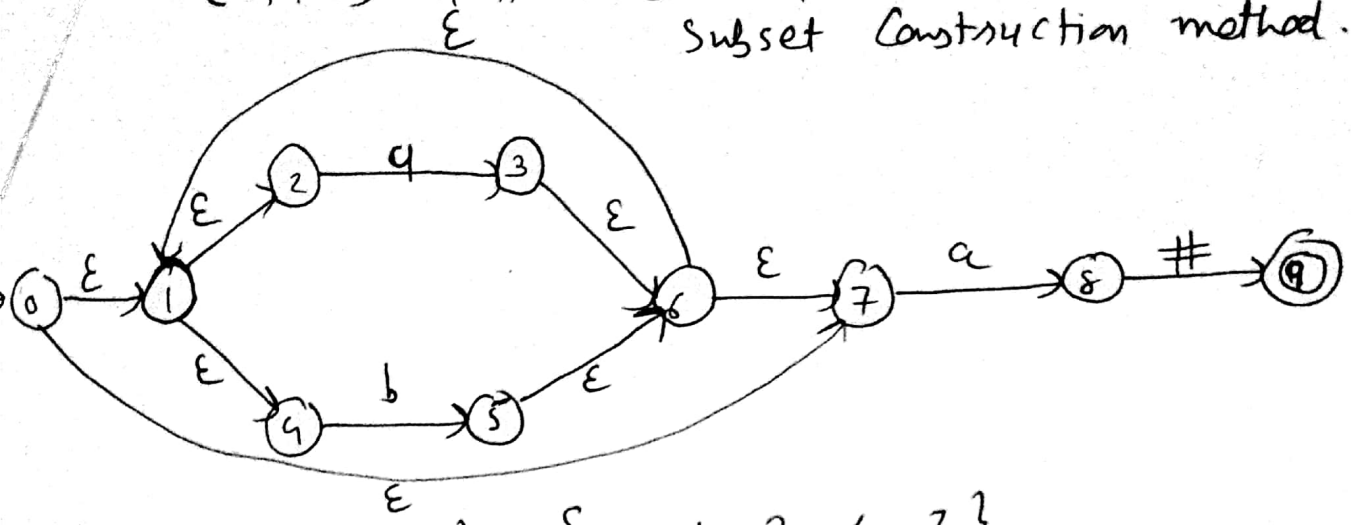
	a	b
A{0,1,2,4,7}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} C
B{0,1,3,8,6,1,2,4,7}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} D
C{5,6,1,2,4,7}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} C
D{9,5,6,1,2,4,7}	{3, 8, 6, 1, 2, 4, 7} B	{10, 5, 6, 1, 2, 4, 7} E
E{10,5,1,2,4,7,6}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} C

DFA :



Final state: E.

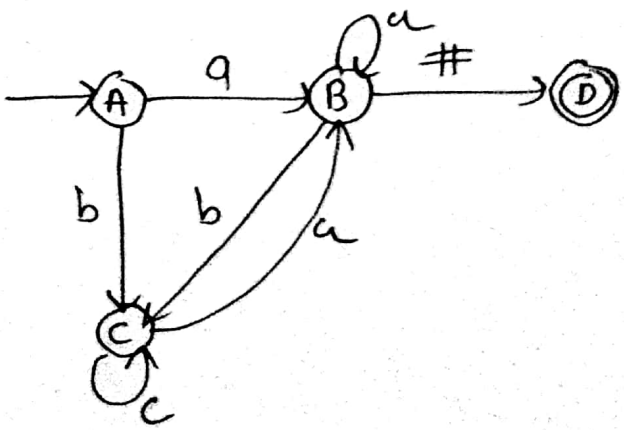
$(a+b)^* a\#$: Convert into DFA using thompson-Subset Construction method.



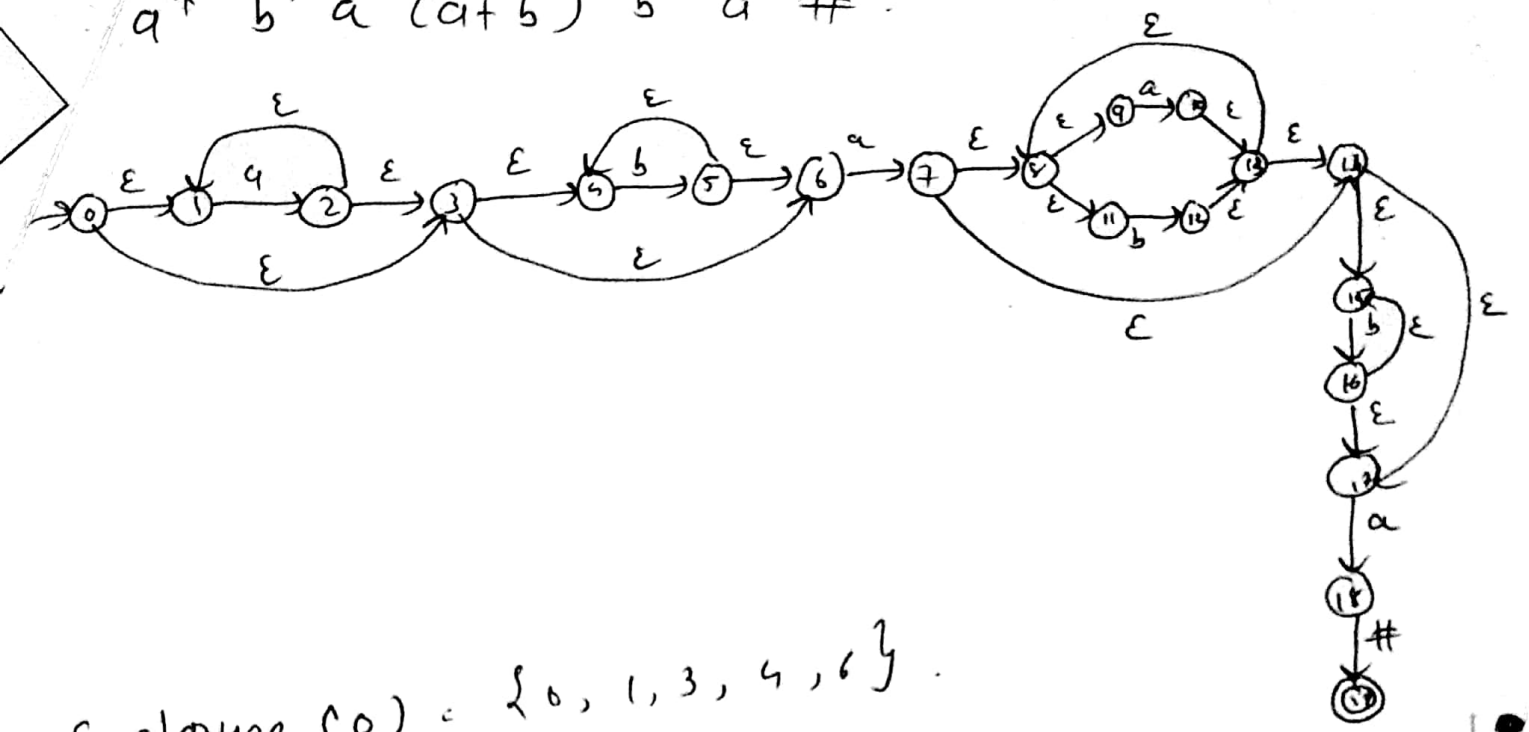
ε-closure(0) = {0, 1, 2, 4, 7}

	a	b	#
A {0,1,2,4,7}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} C	ε
B {3, 8, 6, 1, 2, 4, 7}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} C	{9} D
C {5, 6, 1, 2, 4, 7}	{3, 8, 6, 1, 2, 4, 7} B	{5, 6, 1, 2, 4, 7} C	ε
D {9}	ε	ε	ε

	a	b	#
A	B	C	ε
B	B	C	D
C	B	C	ε
D	ε	ε	ε



$a^* b^* a (a+b)^* b^* a \#$.



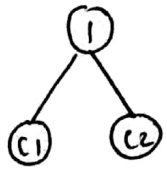
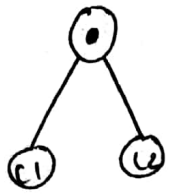
ε-closure (0) : {0, 1, 3, 4, 6}.

	a	b	#
A {0, 1, 3, 4, 6}	{2, 7, 1, 3, 4, 6, 8, 9, 11, 14, 15, 17} B	{5, 4, 6} C	∅
B {2, 7, 1, 3, 4, 6, 8, 9, 11, 14, 15, 17}	{2, 7, 10, 18, 8, 9, 11, 14, 15, 3, 1, 4, 6, 17, 13} D	{5, 12, 16, 4, 6, 13, 8, 9, 11, 14, 15, 17} E	∅
C {5, 4, 6}	{7, 8, 9, 11, 14, 15, 17} F	{5, 4, 6} C	∅
D {2, 7, 10, 18, 8, 9, 11, 14, 15, 3, 1, 4, 6, 17, 13}	{2, 7, 10, 18, ..} D	{5, 12, 16, ..} E	{19} or.
E {5, 12, 16, 4, 6, 13, 8, 9, 11, 14, 15, 17}	{7, 18, 8, 9, 11, 14, 10, 13, 15, 17} H	{5, 12, 16, ..} E	∅
F {7, 8, 9, 11, 14, 15, 17}	{10, 18, 13, 14, 15, 8, 9, 11, 17} I	{12, 16, 8, 9, 11, 14, 15} J	∅

	a	b	#
$\{19\}$	$\emptyset$	$\emptyset$	$\emptyset$
H $\{7, 18, 8, 9, 11, 14, 10, 13, 15, 17\}$	$\{\underline{10}, \underline{18}, \ldots\}$ I	$\{\underline{12}, \underline{16} \ldots\}$ J.	$\{19\}$
I $\{10, 18, 13, 14, 15, 8, 9, 11, 17\}$	$\{\underline{10}, \underline{18} \ldots\}$ I	$\{\underline{12}, \underline{16}, \ldots\}$ J	$\{19\}$ or.
J $\{\underline{12}, \underline{16}, 8, 9, 11, 14, 15\}$	$\{\underline{10}, \underline{18}, \ldots\}$ I	$\{\underline{12}, 16, \ldots\}$ J	$\emptyset$

Syntax tree Method:

Rules:

	Node n	firstpos (n)	lastpos (n)
1)	n is a leaf labeled with ε	∅	∅
2)	n is a leaf labeled with position i	{i}	{i}
3)		firstpos (c1) ∪ firstpos (c2)	lastpos (c1) ∪ lastpos (c2)
4)		if nullable (c1) then firstpos (c1) ∪ firstpos (c2) else firstpos (c1)	if nullable (c2) then lastpos (c1) ∪ lastpos (c2) else lastpos (c2)
5)		firstpos (c1)	lastpos (c1)

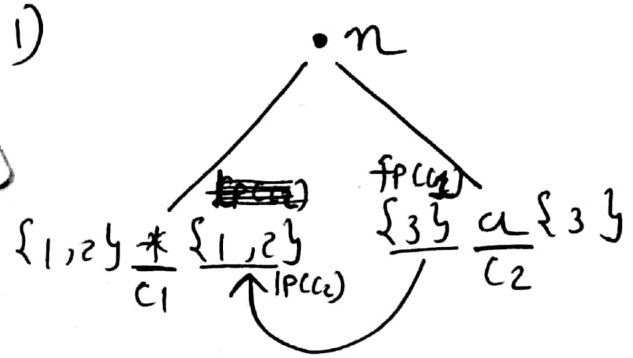
Computing followpos (o) :

1) If n is a cat node with left child c_1 and right c_2, then for every position i in lastpos(c_1), all positions in firstpos(c_2) are in followpos(i).

2) If n is star node and i is the position in lastpos(n), then all positions in firstpos(n) are in followpos(i).

In Syntax tree,

1)



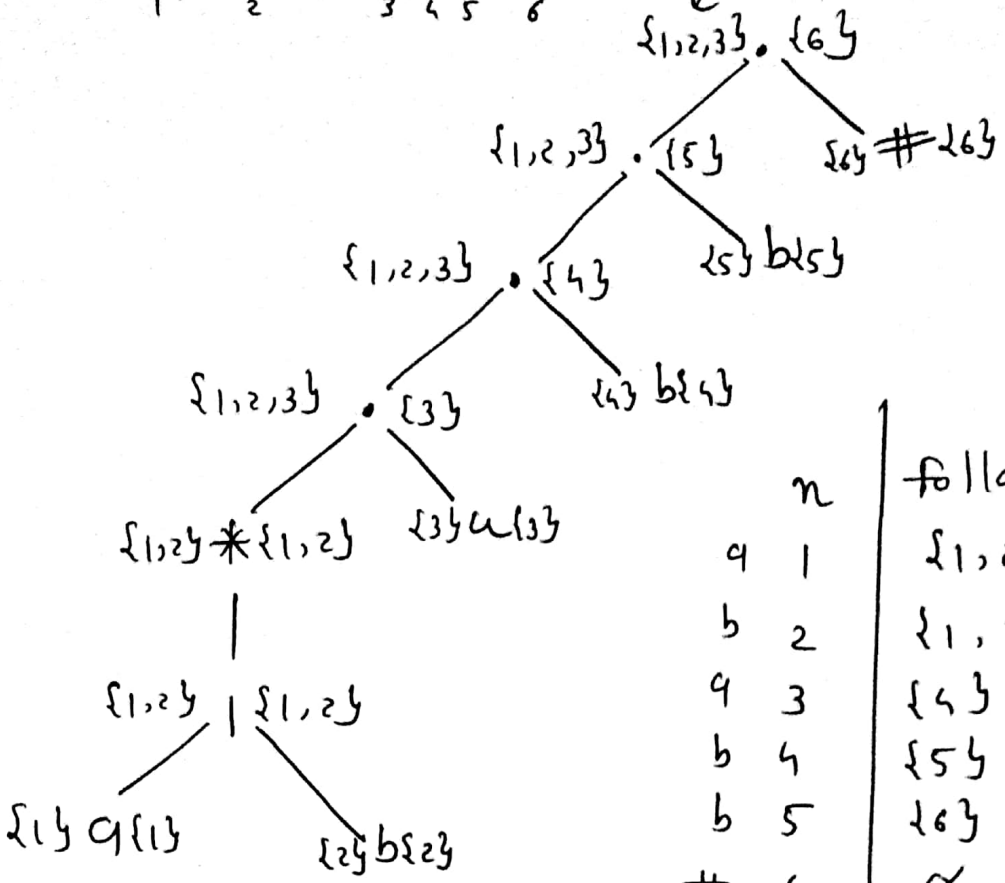
n	followpos(n)
1	3
2	3

2)

n	followpos(n)
1	1, 2
2	1, 2

$(a_1 + b_2)^* a_3 b_4 b_5 \#_6$

starting node : A.

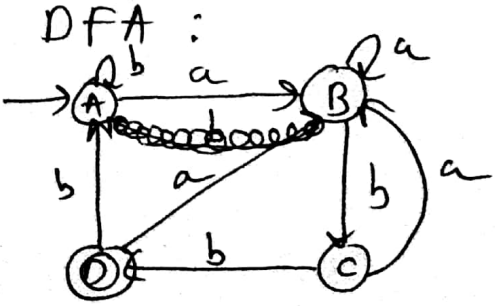


	n	follow pos
a	1	{1, 2, 3}
b	2	{1, 2, 3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	∅

Transition table

	a	b
A {1, 2, 3}	{1, 2, 3, 4} B	{1, 2, 3} A
B {1, 2, 3, 4}	{1, 2, 3, 4} B	{1, 2, 3, 5} C
C {1, 2, 3, 5}	{1, 2, 3, 4} B	{1, 2, 3, 6} D
D {1, 2, 3, 6}	{1, 2, 3, 4} B	{1, 2, 3} A

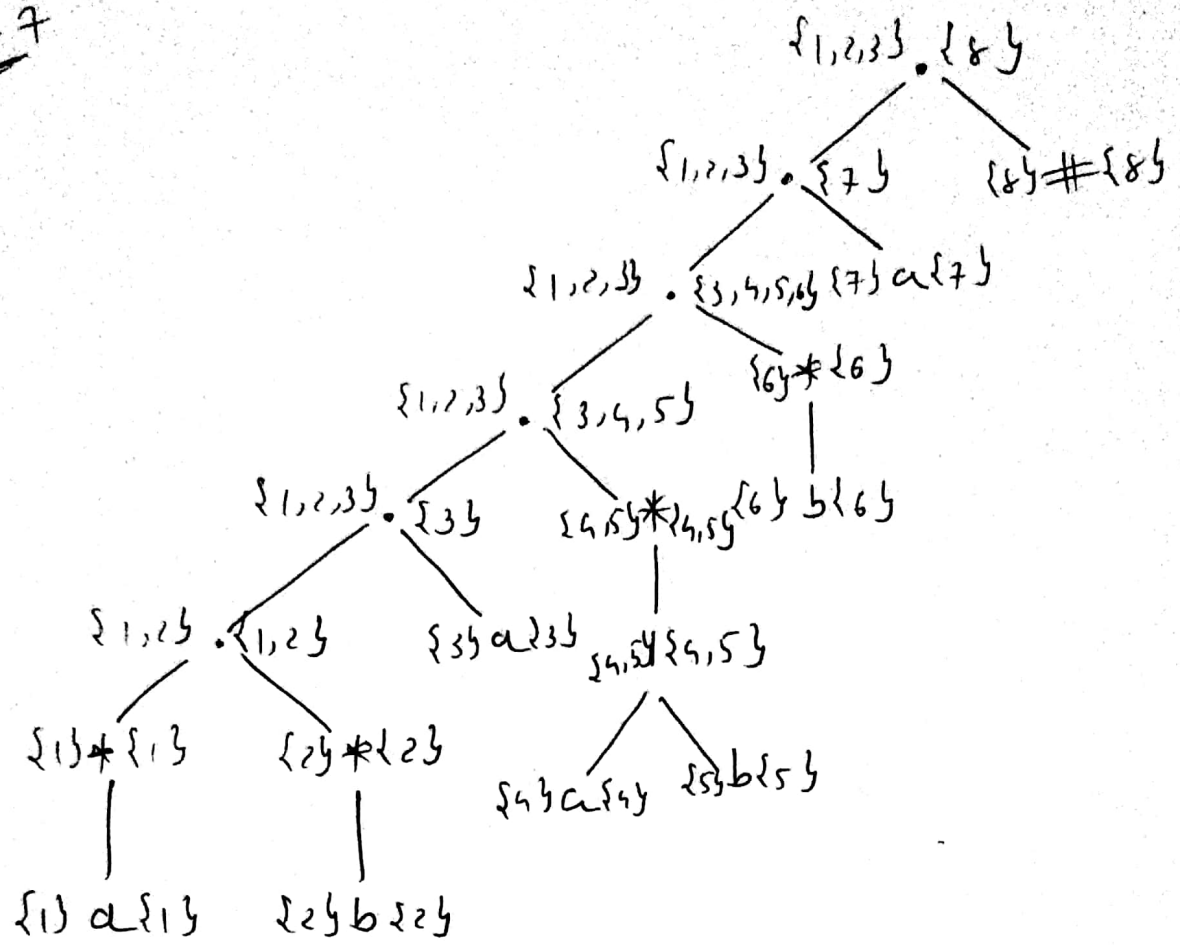
DFA :



final state : D

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

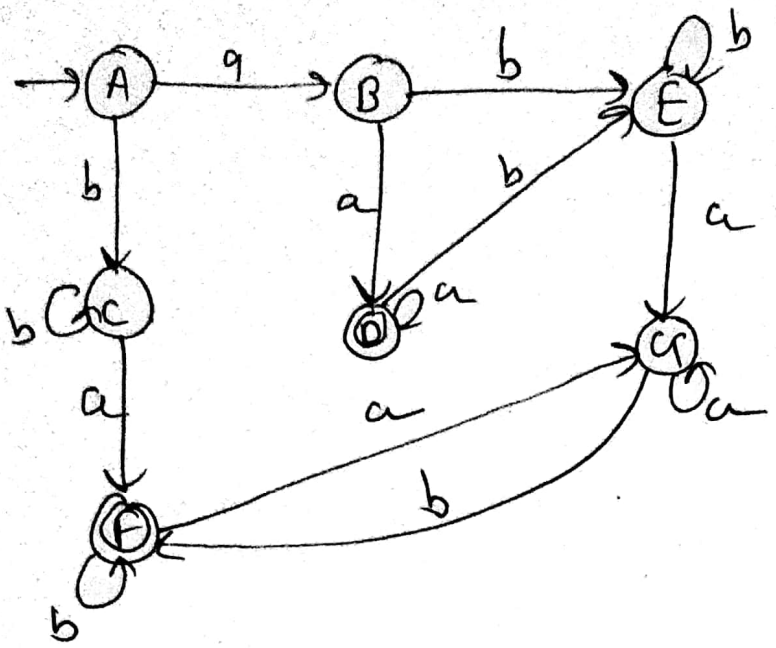
QB-7



n		followpos
a	1	1, 2, 3
b	2	2, 3
a	3	4, 5, 6, 7
a	4	4, 5, 6, 7
b	5	4, 5, 6, 7
b	6	6, 7
a	7	8
#	8	$\emptyset$

	a	b
A {1, 2, 3}	{1, 2, 3, 4, 5, 6, 7} B	{2, 3} C
B {1, 2, 3, 4, 5, 6, 7}	{1, 2, 3, 4, 5, 6, 7, 8} D	{2, 3, 4, 5, 6, 7} E
C {2, 3}	{4, 5, 6, 7} F	{2, 3} C
D {1, 2, 3, 4, 5, 6, 7, 8}	{1, 2, 3, 4, 5, 6, 7, 8} D	{2, 3, 4, 5, 6, 7} E
E {2, 3, 4, 5, 6, 7}	{4, 5, 6, 7, 8} G	{2, 3, 4, 5, 6, 7} E
F {4, 5, 6, 7, 8}	{4, 5, 6, 7, 8} G	{4, 5, 6, 7} F
G {4, 5, 6, 7}	{4, 5, 6, 7, 8} G	{4, 5, 6, 7} F

DFA :



final state : D , F

QB-A $(a^* \mid b^*)^* \#$ (positions: a_1, b_2, $\#_3$)

$\{1,2,3\} \cdot \{3\}$

$\{1,2\} * \{1,2\}$ $\qquad$ $\{3\} \# \{3\}$

$\{1,2\} \mid \{1,2\}$

$\{1\} * \{1\}$ $\qquad$ $\{2\} * \{2\}$

$\{1\} a \{1\}$ $\qquad$ $\{2\} b \{2\}$

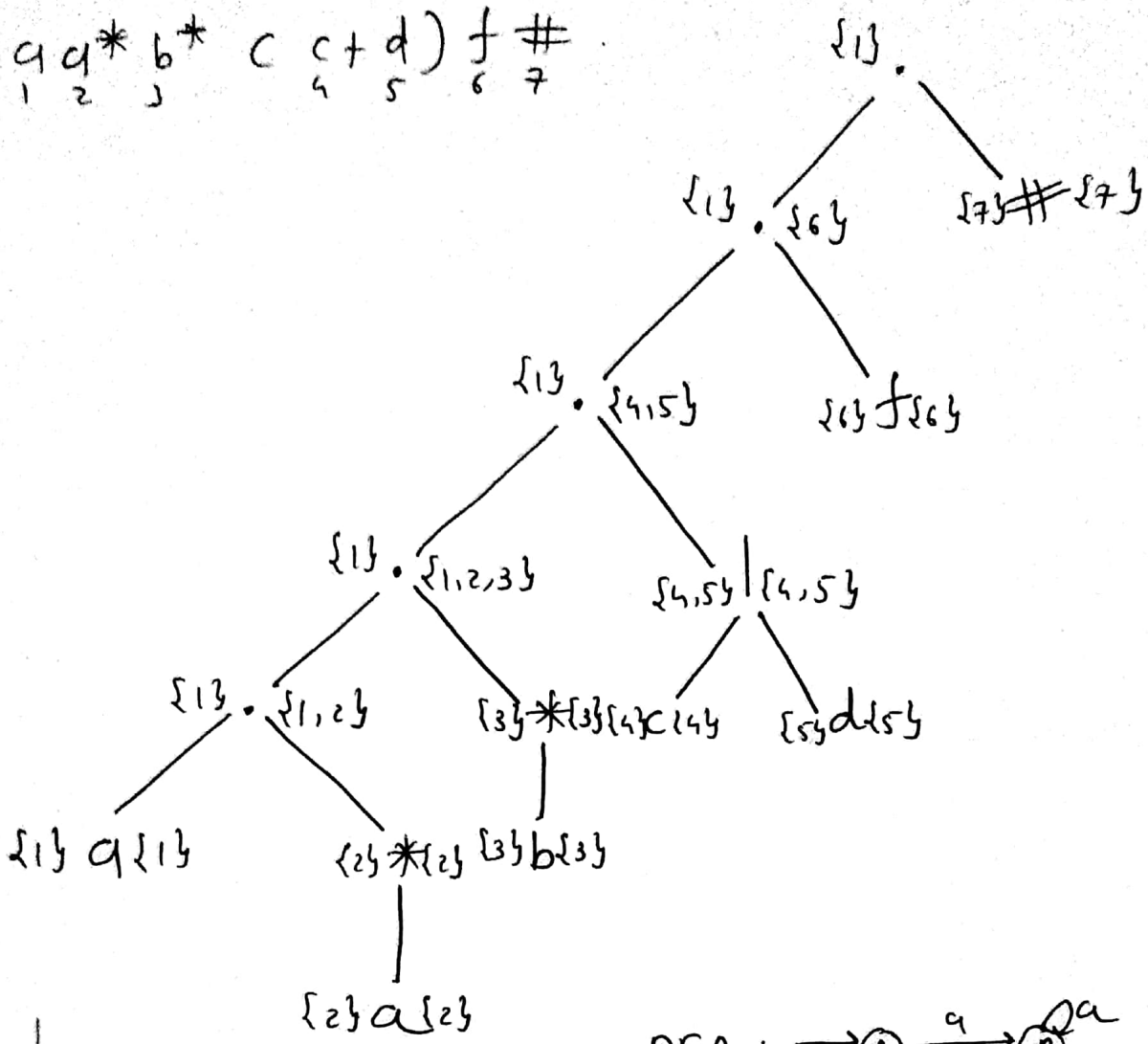
		follow pos
a	1	1, 2, 3
b	2	1, 2, 3
a	3	$\emptyset$

	a	b
$\{1,2,3\}$ A	$\{1,2,3\}$ A	$\{1,2,3\}$ A

DFA:

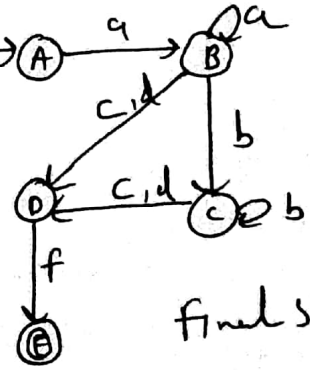
QB-24 $a^+ b^* (c|d) f \#$ (QB-24)

$a_1 a_2^* b_3^* (c_4 + d_5) f_6 \#_7$



n		follow pos
a	1	2, 3, 4, 5
a	2	2, 3, 4, 5
b	3	3, 4, 5
c	4	6
d	5	6
f	6	7
#	7	∅

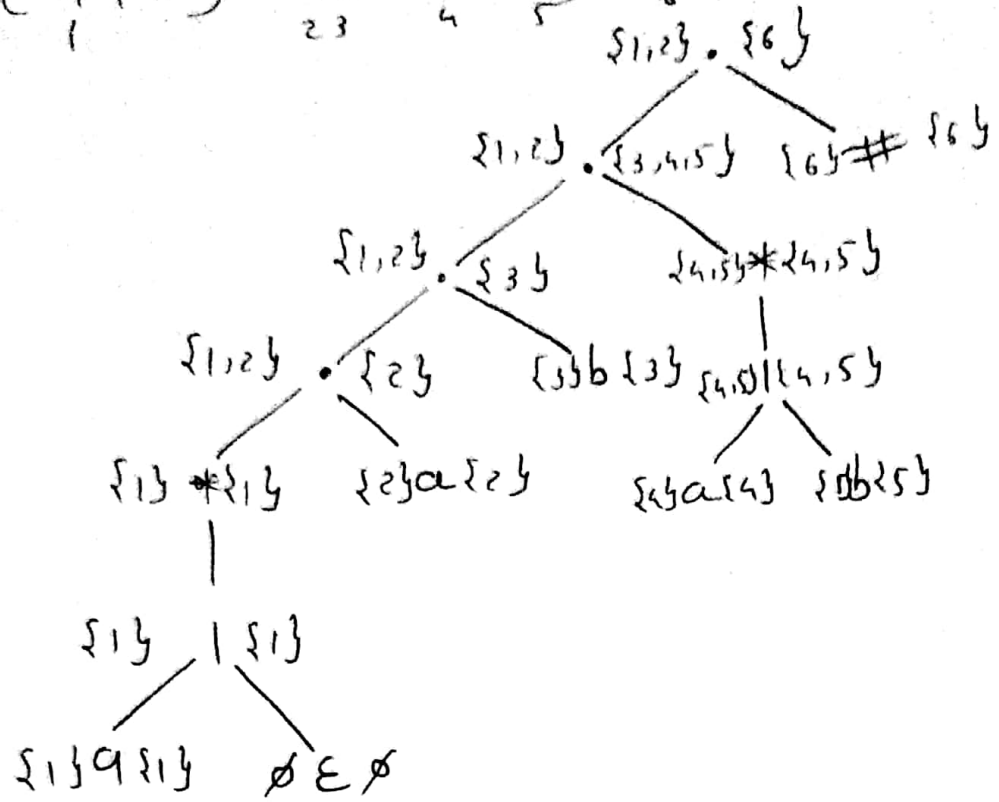
DFA :



final state E.

	a	b	c	d	f
A {1}	{2, 3, 4, 5} B	∅	∅	∅	∅
B {2, 3, 4, 5}	{2, 3, 4, 5} B	{3, 4, 5} C	{6} D	{6} D	∅
C {3, 4, 5}	∅	{3, 4, 5} C	{6} D	{6} D	∅
D {6}	∅	∅	∅	∅	{7} E
E {7}	∅	∅	∅	∅	∅

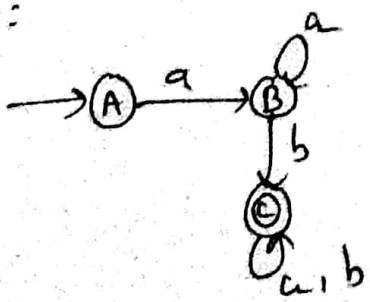
QB-33 $(a|\varepsilon)^* ab (a|b)^* \#$



n		follow pos
a	1	1, 2
a	2	3
b	3	4, 5, 6
a	4	4, 5, 6
b	5	4, 5, 6
#	6	∅

	a	b
{1, 2} A	{1, 2, 3} B	∅
{1, 2, 3} B	{1, 2, 3} B	{4, 5, 6} C
{4, 5, 6} C	{4, 5, 6} C	{4, 5, 6} C

DFA:



3) Direct Method:

1) Convert the NFA with ε (built using Thompson's Method) into NFA without ε.

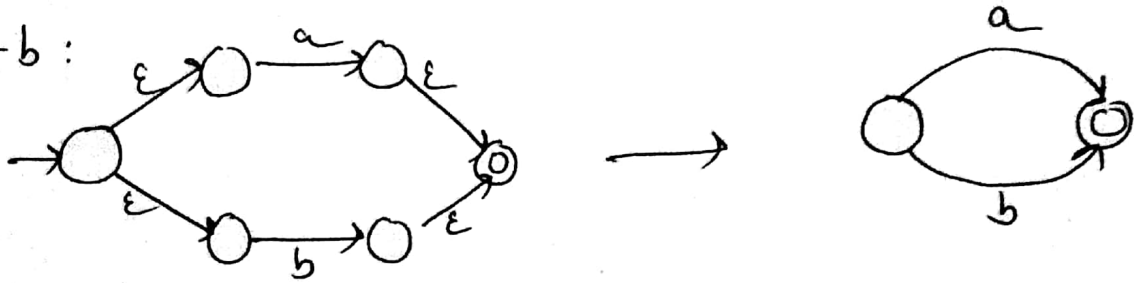
2) Then the NFA without ε is converted into DFA.

Rules : NFA - ε

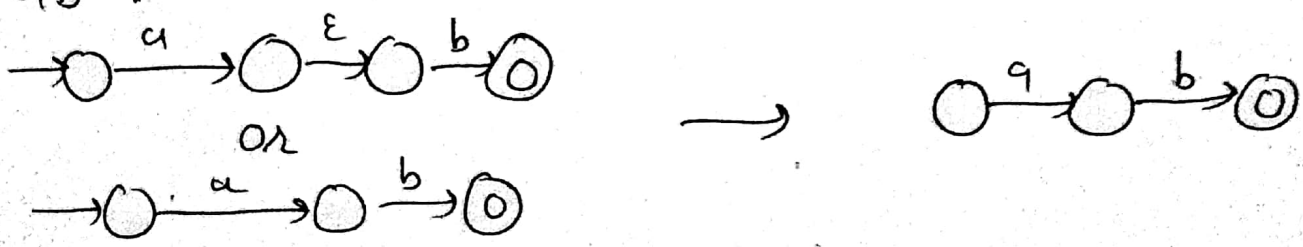
NFA - without ε

1) a*:

2) a+b:



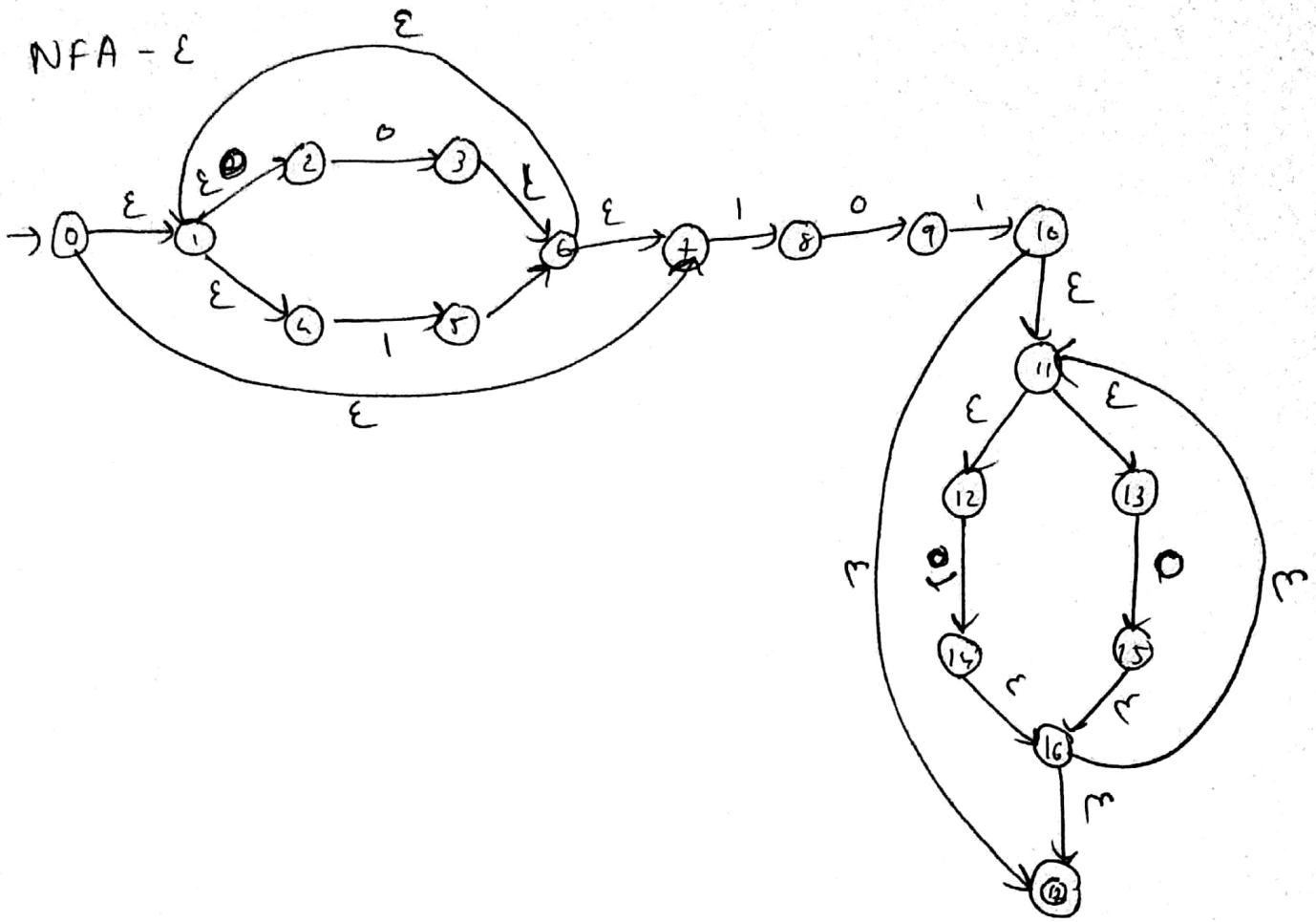
3) ab:



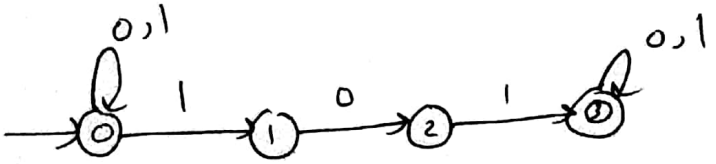
QB-11 $(0+1)^*$ 101 $(0+1)^*$

using direct Method :

1) NFA - ε



2) NFA - without ε .



3) Transition table :

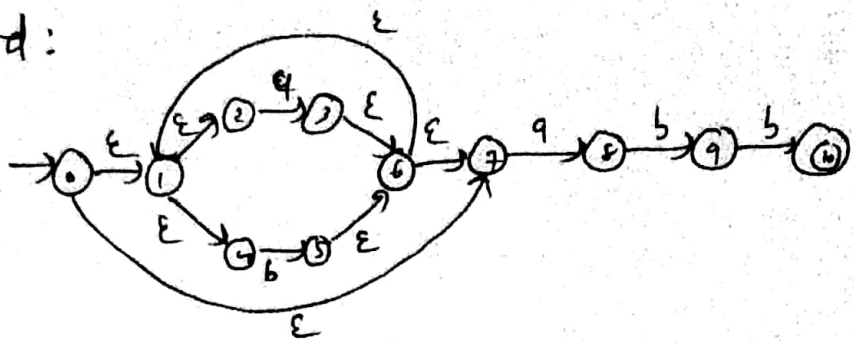
	0	1
A {0}	{0} A	{0,1} B
B {0,1}	{0,2} C	{0,1} B
C {0,2}	{0} A	{0,1,3} D
D {0,1,3}	{0,2,3} E	{0,1,3} D
E {0,2,3}	{0,3} F	{0,1,3} D
F {0,3}	{0,3} F	{0,1,3} D

4) Draw DFA : Final state : D, E, F

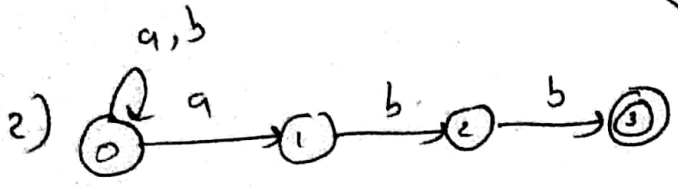
Using Direct Method:

$(a+b)^* abb$

1)



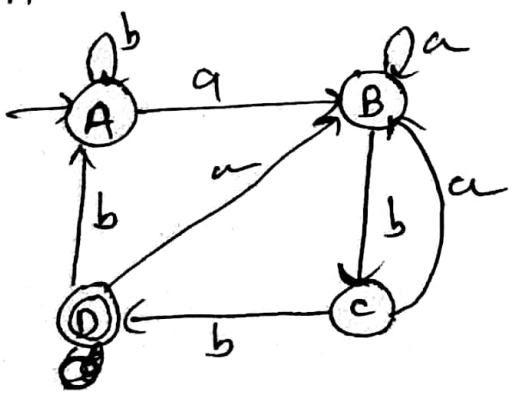
2)



3)

	a	b
{0} A	{0, 1} B	{0} A
{0, 1} B	{0, 1} B	{0, 2} C
{0, 2} C	{0, 1} B	{0, 3} D
{0, 3} D	{0, 1} B	{0} A

4) DFA:



final state : D

optimization of DFA:

Rules:

→ start with an Initial partion Π with two groups F and S-F, the accepting and non accepting states of DFA.

→ partition it into subgroups such that two states s and t are in same subgroup if and only if for all input symbols a, states s and have transitions on a to states in the same group of Π.

ex: $(a+b)^*$ abb

States	a	b
A	B	C
B	B	D
C	B	C
D	B	E
final → E	B	C

group : 1 — A, B, C, D

group : 2 — E

group : 1 — A, B, C

group : 2 — D

group : 3 — E

group : 1 — A, C

group : 2 — B

group 3 — D

group 4 — E

minimized transition table:

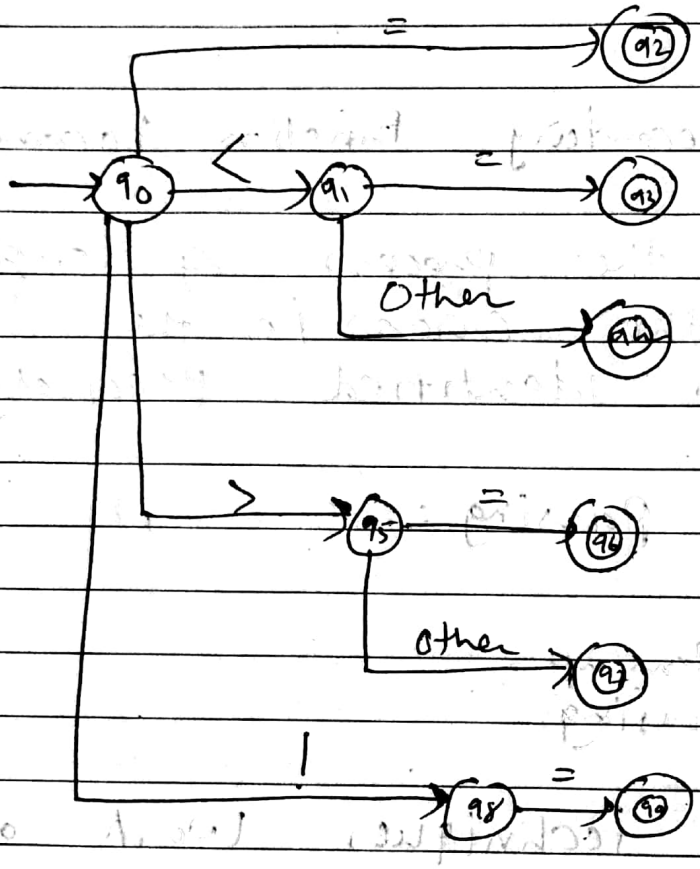
States	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Draw optimized DFA.

Transition diagram :

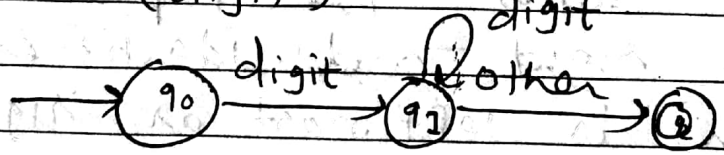
(1) Write r.e. for relation operators. Design a transition diagram for them.

r.e : (< | > | <= | >= | = | !=)

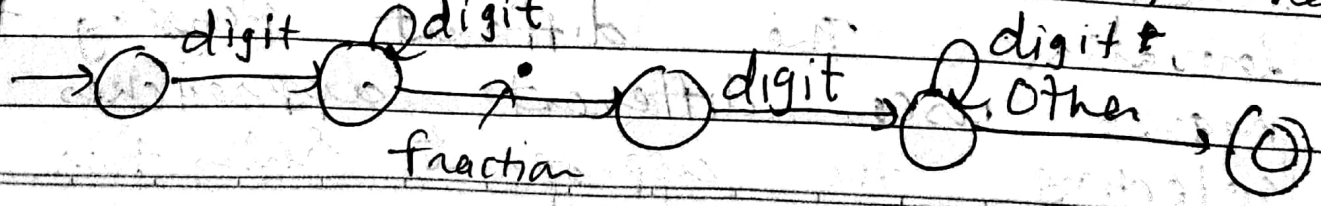


(2) Transition diagram for unsigned numbers.

1 r.e : (digit)+ 5280 Int.



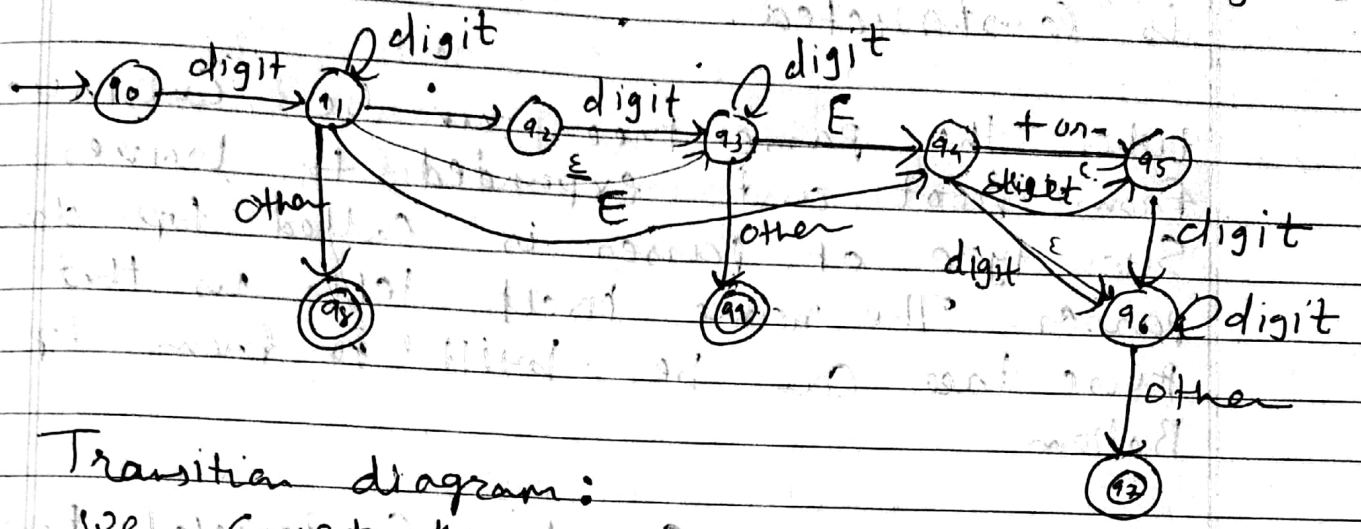
2 r.e : (digit)+ . (digit)+ 39.37 real no.



9. Unsigned no. used in pascal.

Real, Expanent, int. : 6.336E4, 1.894E-4

→3 r.e : digit + (. (digit +) ? (E (+ | -) ? digit +) ?



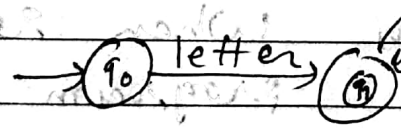
Transition diagram:

We convert the pattern into stylized flowchart called transition diagram

→ In finite automata transitions are marked with single letter of alphabet. In Transitan diagram they can be marked with letters or strings (combination of letters)

→ Transition diagram is a way of visually representing finite state machine.

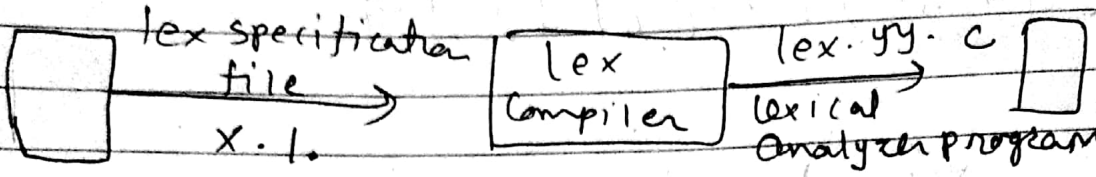
→4 r.e : letter (letter + digit)*

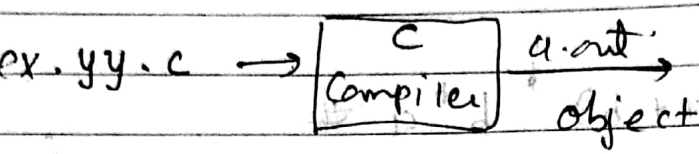


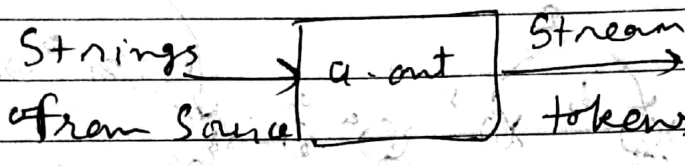
→ Positions in a transition diagram are drawn as a circle and are called states.

→ states are connected by arrows called edges

A language Specifying for lexical Analyzer:







→ lex Scans the Source program in order to get the stream of tokens and these tokens are related together so that Varions programing Construct Such as expressions, block Statements, Procedures, Control structures can be realised.

→ LEx Specification file can be created using extension .l.
for ex : x.l

→ lex.yy.c is a c program which is actually a lexical analyzer program. The LEx specification file stores the regular expression for the tokens and lex.yy.c Consist of tabular representation of the transition diagrams Constructed for the regular expression.

lexer is very much faster in finding the tokens as compared to the handwritten lex. program in C.

→ The lexeme can be recognized with the help of this tabular representation of transition diagram.

→ Finally compiler compiles this generated lex.yy.c and produces an object program a.out. When some input stream is given to a.out as then sequence of tokens get generated.

Structure of LEX:

LEX program consists of three parts:

1 Declaration part
2 Rule section.
3 Auxiliary Procedure Section.

%{
Declaration section // Declaration of variable, Constant can be done
%}
%%

Rule Section {C code // Regular expression
ex: int | printf | if : keyword

%%

Auxiliary Procedure section. } functions/ procedures required.

Rule section: Consist of regular expression with associated with actions

PAGE NO.
DATE

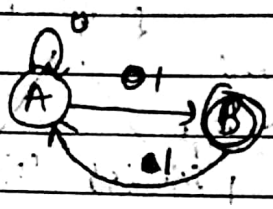
Q.B: What are regular expressions? Find the regular expression described by DFA.

$\{\{A, B\}, \{0, 1\}, \delta, A, \{B\}\}$

	0	1
A	A	B
B	∅	A

B is accepting state.

Describe the language defined by the regular expression.



$[0^* 1 (11)^*]$

or $[0^* 1]^{**}$