Lexical Analysis.

Role of Lexical Analyzer:

- The lexical analyzer is the first phase of compiler.
- Its main task is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program

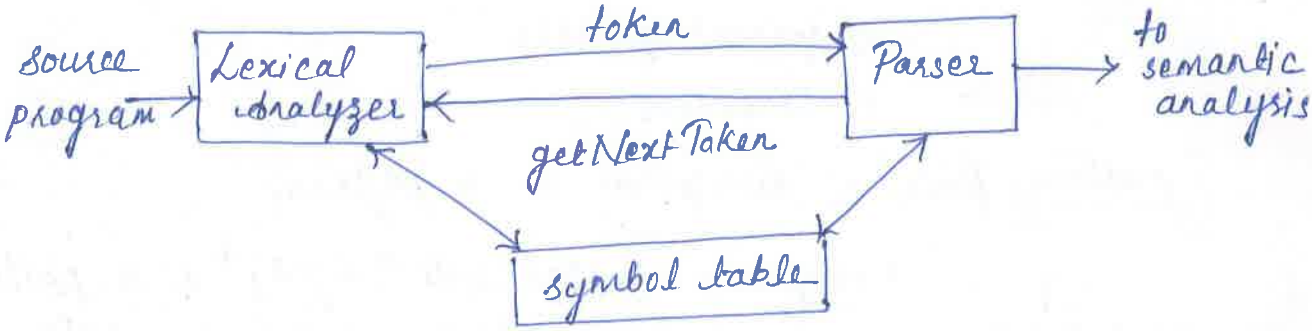


fig: The Role of lexical analyzer.

- The "getNextToken" command causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the Parser.
- Sometimes, lexical analyzers are divided into a cascade of two processes.

a) scanning
b) lexical analysis (ie) it generates the series of tokens

Issues in Lexical Analysis:

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical and syntax analysis (parsing).

→ Simplicity of design
→ Compiler efficiency is improved
→ Compiler portability is enhanced.

Tokens, Patterns and lexemes:

When discussing lexical analysis, we use 3 related but different terms:

Token: It is a sequence of character that can be treated as a single logical entity. Typical tokens are:

a) Identifiers
b) Keywords
c) Operands
d) Special symbols
e) Constants

pattern: Rule of description is a pattern.

Example: letter(letter|digit)* is a pattern to symbolize a set of strings which consist of a letter followed by a letter or digit.

Lexemes: Sequence of characters in a token is a lexeme.

Example: 100.01, counter, const .. etc are lexemes.

if(a<b)

Here "if", "(", "a", "<", "b", ")" are all lexemes

Lexemes	token
if	Keyword
(,)	operator
a, b	identifier
<	operator

The blank and newline characters can be ignored. These stream of tokens will be given to syntax analyzer.

Input Buffering:

- The lexical analyzer scan the input string from left to right one character at a time.
- It uses 2 pointers : begin_ptr(bP) , forward-ptr(fP)
  lexeme

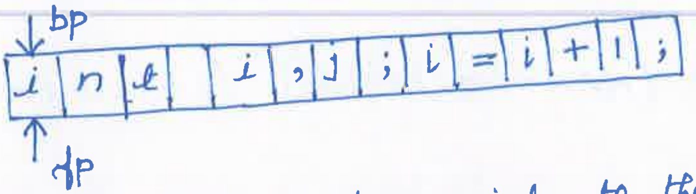
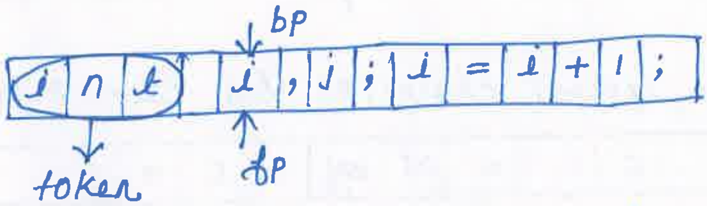


fig : Initial Configuration

- Initially both pointers point to the first character of the input string.
- The fp moves ahead to search for end of lexeme. Blank space indicates end of lexeme.
- When fp encounters white space, it ignore & move ahead.

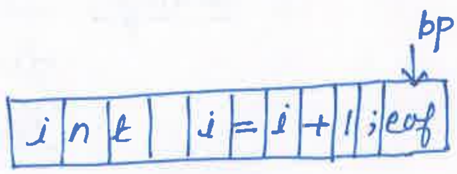


- There are two buffering scheme.

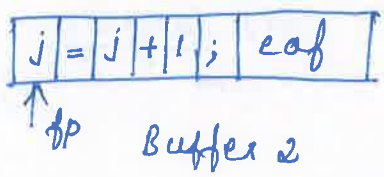
1) One buffer scheme: Only one buffer is used to store the input string. The problem is overwriting the 1st part of lexeme.

2) Two buffer scheme: To overcome the problem of one buffer scheme, two buffers are used.

- To identify the boundary of first buffer "eof" is placed at the end of first buffer.
- Alternatively both buffers can be filled up until end of the input program & stream of tokens is identified
- This "eof" character introduced at the end is called "sentinel"



Buffer 1



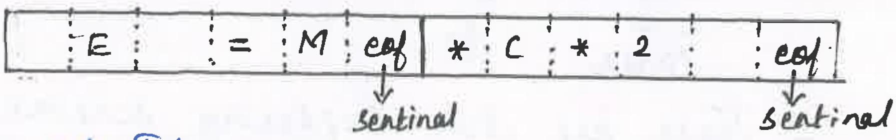
Buffer 2

Problem: If length of lexeme is longer than length of buffer then scanning input cannot be scanned completely.

Algorithm for input buffering using sentinels:

```
if (fp == eof(buff1))   /* encounters end of first buffer */
{
    reload the second buffer ;
    fp = beginning of second buffer ;
}
else if (fp == eof(buff2))   /* encounters end of second
                                               buffer */
{
    reload the first buffer ;
    fp = beginning of first buffer ;
}
else          /* fp == eof(input) */
    terminate scanning
```

<u>sentinel</u> is a special character but cannot be a part of source program.



Specification of Tokens:

To specify tokens regular expressions are used.

i) <u>Alphabets</u> : It is any finite set of symbols.

Example: letters, digits, and punctuation

$\{0, 1\} \rightarrow$ binary alphabet.

ii) <u>String</u>: It is a finite sequence of symbols drawn from alphabet.

Example: "word" $\rightarrow$ String of length 4

Empty String denoted ϵ, length is zero

(iii) <u>language</u> : It is any countable set of strings over some fixed alphabet.

Term	Description
prefix of string s	A string obtained by removing zero or more symbols from the end of s. Ex: ban, banana & ϵ are prefixes of banana.
suffix of string s	A string obtained by removing zero or more symbols from the beginning of s. Ex: nana, banana & ϵ are suffix of banana.
Substring of string s	It is obtained by deleting any prefix & any suffix from s. Ex: banana, nan & ϵ are substrings of banana
Subsequence of string s.	A string formed by deleting zero or more not necessarily consecutive positions of s. Ex: baan is a subsequence of banana.

Operations on Language:

Operation	Definition
Union of L and M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
Concatenation of L and M	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Lexical errors:

Its include misspellings of identifiers, keywords or operators.

Eg: The use of an identifier elipseSize instead of ellipseSize and missing quotes around text intended as a string.

Regular Expressions:

- The R.E's are used to describe the tokens of a programming Language. A R.E is obtain from a set of defining rules.
- The Language denoted by a R.E is called "Regular set."

Rules:

(i) Basic Rules:

1) ϵ is a R.E, and $L(\epsilon)$ is $\{\epsilon\}$
2) If 'a' is a symbol in Σ, then 'a' is a RE, and $L(a) = \{a\}$. that is the language with one string, of length one.

(ii) Induction Rules:

Suppose r and s are R.E's denoting languages $L(r)$ and $L(S)$.

(i) $(r)|(S)$ is a RE denoting the language $L(r) \cup L(S)$.

(ii) $(r)(S)$ is a RE denoting the language $L(r)L(S)$.

(iii) $(r)^*$ is a RE denoting $(L(r))^*$

(iv) (r) is a RE denoting $L(r)$

Precedence of operation in RE:

a) $*$ has highest Precedence

b) Concatenation $(\cdot)$ has second highest Precedence

c) $|$ has lowest Precedence

Regular Definition: If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$\vdots$

$d_n \rightarrow r_n$

where,

d_1, d_2 - distinct name

r_1, r_2 .. etc - RE

Example: Regular definition of the language of C identifiers.

letter → |A|B|...|Z| a|b|..|z|

digit → 0|1|...|9

id → letter (letter|digit)*

Notational Shorthands:

1) one or more instances: unary postfix operator + ie) r^+
2) zero or more instances: unary postfix operator * ie) r^*
3) zero or one instance: unary postfix operator ? ie) $r?$

   $r?$ is a shorthand for $r|\epsilon$

4) Character classes: [a-z] denotes the RE a|b|c|...|z

Example: unsigned numbers (integer (or) floating point) are strings such as 5280, 0.01234, 6.336E4, 1.89E-4.

Regular definition

digit → 0|1|...|9

digits → digit digit* (digit)+

optionalFraction → . digits|ϵ

optional Exponent → (E(+|-|ϵ) digits) | ϵ

number → digits optionalFraction Optional Exponent.

Using notational shorthands.

digit → [0-9]

digits → digit$^+$

number → digits(. digits)? (E[+-]? digits)?

Example: Construct Regular Expression:

1) Construct R.E for a L containing the strings of length 2 over $\Sigma = \{0,1\}$

Soln: R.E $= (0 \overset{(or)}{+} 1)(0+1)$

ie) $L = 00, 01, 10, 11$

2) L containing strings which end with "abb" over $\Sigma = \{a,b\}$

Soln: $L = \underline{abb}, a\underline{abb}, b\underline{abb}, aa\underline{abb}, ab\underline{abb}, bb\underline{abb} \ldots$ etc

So, R.E $= (a+b)^* abb$

3) Write R.E for recognizing identifier.

Soln: Identifier : combination of letters or letter and digits but having first character as letter always.

Letter $= (A, B, \ldots Z, a, b \ldots z)$
digit $= (0, 1, 2 \ldots 9)$

R.E $=$ letter (letter + digit)*

4) L accepting all combinations of a's except null string over $\Sigma = \{a\}$

Soln: $L = \{a, aa, aaa \ldots\}$

R.E $= a^+ \rightarrow$ called +ive closure.

5) L containing all the strings with any number of a's & b's

Soln: $L = \{\varepsilon, a, aa, b, bb, ab, ba, bab \ldots\}$

R.E $= (a+b)^* \rightarrow$ any combination of a & b even a null string.

6) L containing all strings with any number of a's & b's except null.

R.E $= (a+b)^+$

7) L containing string will be such that any number of a's followed by any number of b's followed by any number of c's.

R.E $= a^* b^* c^*$

8) L consist of exactly two b's over the set $\Sigma = \{a, b\}$

R.E $= a^* \underline{b} a^* \underline{b} a^*$

$\downarrow$
any number of a's or null string

Algebraic Properties of Regular Expression.

Algebraic laws obeyed by R.E's are Algebraic properties.

Properties	Meaning
$r_1 \mid r_2 = r_2 \mid r_1$	$\mid$ is commutative
$r_1 \mid (r_2 \mid r_3) = (r_1 \mid r_2) r_3$	$\mid$ is associative
$(r_1 r_2) r_3 = r_1 (r_2 r_3)$	Concatenation is associative
$r_1 (r_2 \mid r_3) = r_1 r_2 \mid r_1 r_3$ $(r_2 \mid r_3) r_1 = r_2 r_1 \mid r_3 r_1$	Concatenation is distributive over $\mid$
$\varepsilon r = r \varepsilon = r$	ε is identity
$r^* = (r \mid \varepsilon)^*$	Relation between ε and $*$
$r^{**} = r^*$	$*$ is idempotent.

Example: construct Regular Expression:

9) Construct the R.E for the language accepting all the strings which are ending with 00 over the set $\Sigma = \{0,1\}$

R.E = (Any combination of 0's & 1's) 00

R.E = $(0+1)^* 00$

The valid strings are $1\underline{00}$, $01\underline{00}$, $10\underline{00}$ ...

10) Write R.E for the language accepting the strings which are starting with 1 and ending with 0, over the set $\Sigma = \{0,1\}$

→ The 1st symbol in R.E should be 1
last symbol should be 0.

$\therefore$ R.E = $1 \underbrace{(0+1)^*}_{} 0$

↳ any combination of 0 & 1 including null.

Finite Automata (FA):

* The generalized transition diagram for a RE is called as Finite automata.
* It is the directed graph in which the nodes of are the states and edges are the transition.
* FA is a collection of 5 tuples $(Q, \Sigma, \delta, q_0, F)$. where,

$Q \rightarrow$ finite set of states

$\Sigma \rightarrow$ input set

$\delta \rightarrow$ transition function $\longrightarrow$ 2 parameters are passed to δ

$q_0 \rightarrow$ Initial state ie) $q_0 \in Q$

$F \rightarrow$ final state. ie) $F \in Q$

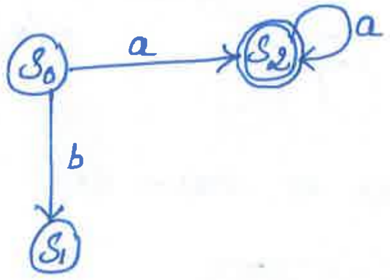
$q_1 = \delta(q_0, a)$ means from current state q_0 with i/p 'a'. the next state transition is q_1.

FA

DFA (Deterministic Finite Automata) | NFA (Non-Deterministic Finite Automata)

DFA (Deterministic Finite Automata)

* The FA is called DFA if there is only one path for a specific i/p from current state to next state.



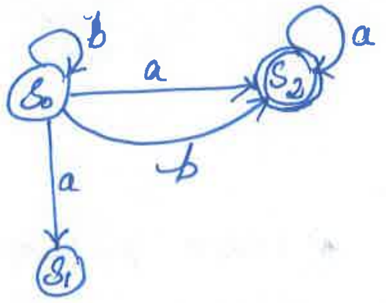
* DFA consist of 5-tuple

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

* No state has an ϵ transition.
* DFA is a subset of NFA and It is determined that with a particular i/p where to go next.

NFA (Non-Deterministic Finite Automata)

* The FA is called NFA if there is many paths for a specific i/p from current state to next state.



* NFA consist of 5-tuple

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

* The state can have ϵ transition
* It is not determined that with a particular i/p where to go next. Hence this FA is called NFA

Steps to implement the FA:

1) Conversion of RE to NFA (Thompson's construction)
2) Conversion of NFA to DFA (Subset construction)
3) Minimization of DFA.

1) Conversion of RE to NFA: (Thompson's construction)

Algorithm:

Input: A R.E 'r' over an alphabet Σ. → finite state of input

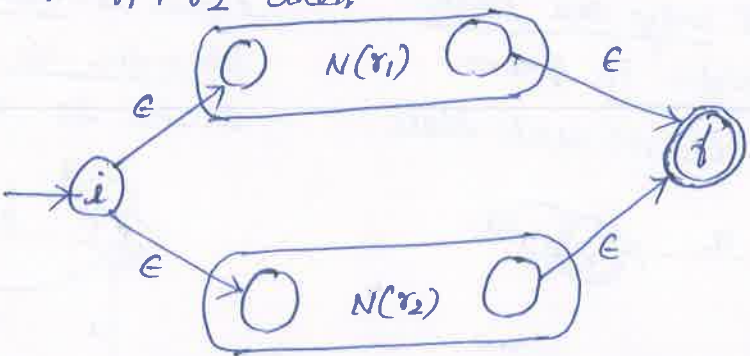
Output: An NFA 'N' accepting $L(r)$

Method:

1) For $r = \epsilon$

2) when $r = a$ for $\Sigma = \{a\}$ the NFA is

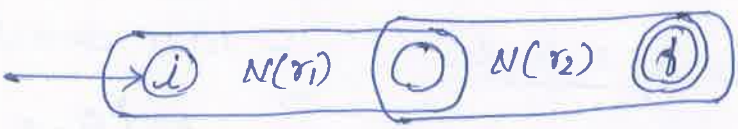
3) when $r = r_1 + r_2$ then NFA can be drawn as-



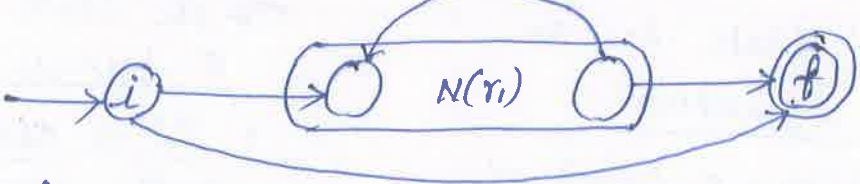
$N(r_1) \rightarrow$ NFA for regular Expression r_1

$N(r_2) \rightarrow$ NFA for r_2

4) When $r = r_1 . r_2$ then NFA can be drawn as-



5) when $r = (r_1)^*$ then NFA can be drawn as-

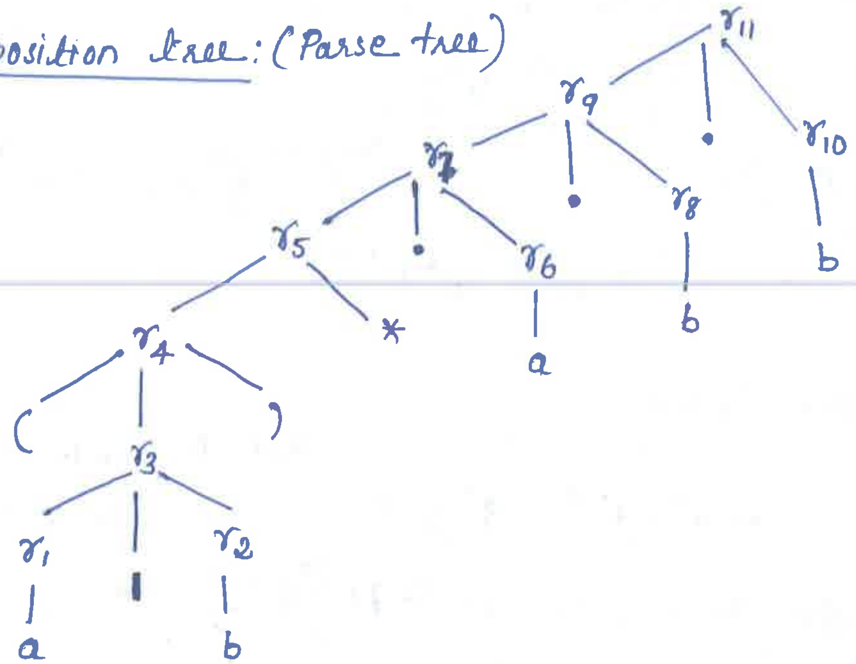


6) For the parenthesized regular expression (r), the NFA is same as that of r

Example

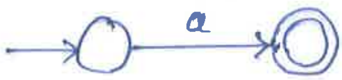
1) Construct NFA for the RE, $r = (a|b)^* abb$

Decomposition tree: (Parse tree)

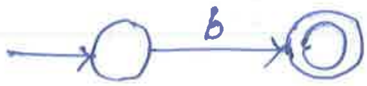


Construction of NFA:

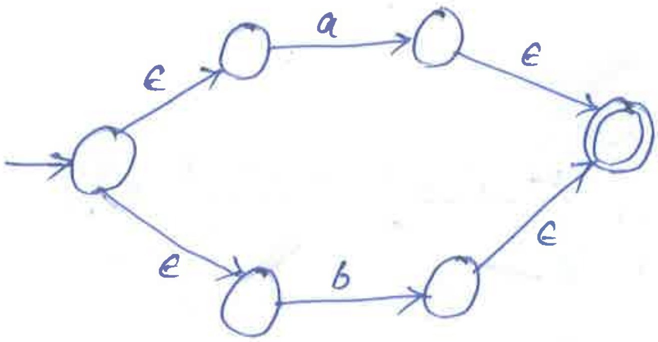
1) NFA for $r_1 = a$



2) NFA for $r_2 = b$



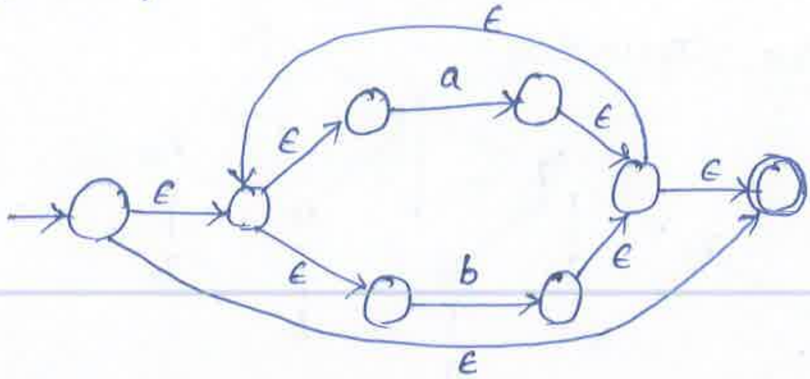
3) NFA for $r_3 = r_1 | r_2$ ie) $a|b$



4) NFA for $r_4 = (r_3)$ ie) $(a|b)$

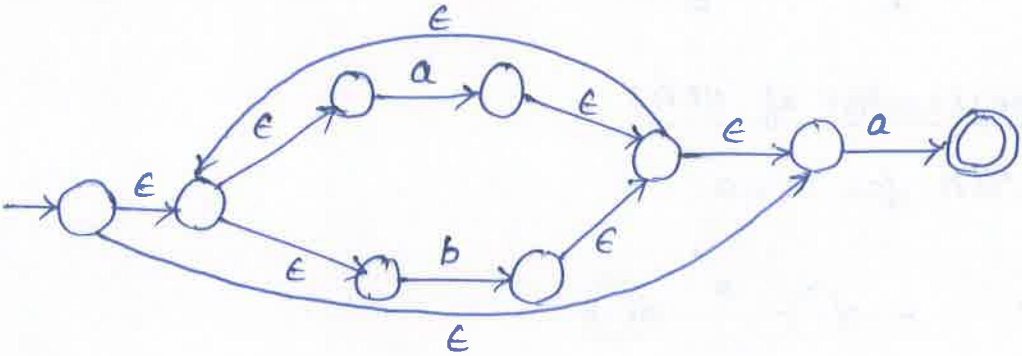
The NFA is same as r_3

5) NFA for $r_5 = r_4^*$ ie) $(a|b)^*$



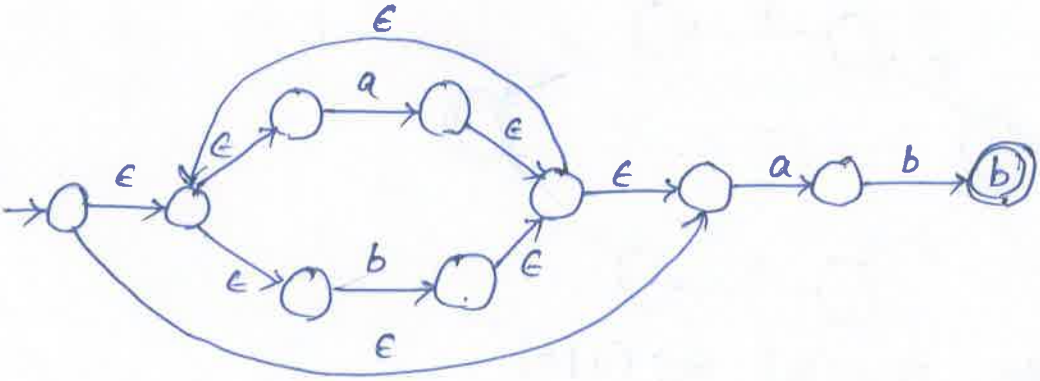
6) NFA for $r_6 = a$

7) NFA for $r_7 = r_5 \cdot r_6$ ie) $(a|b)^* a$

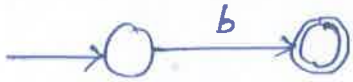


8) NFA for $r_8 = b$

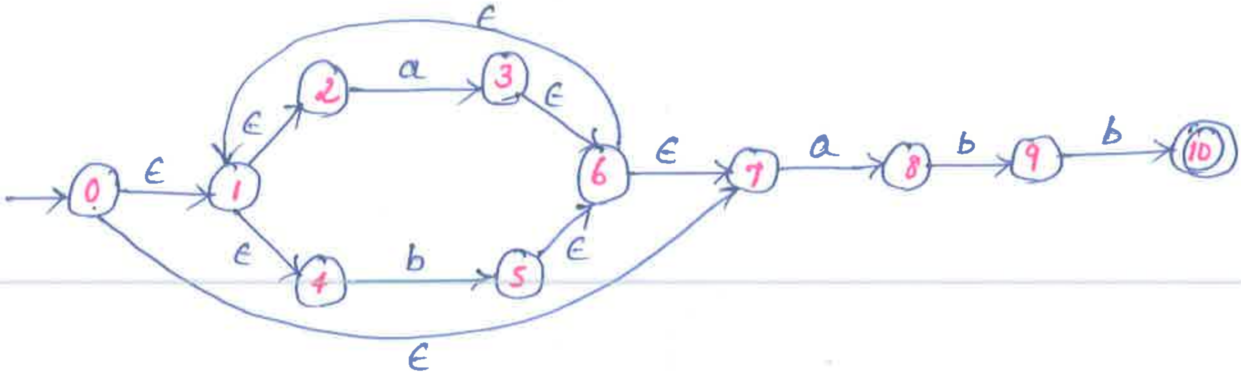
9) NFA for $r_9 = r_7 \cdot r_8 = (a|b)^* ab$



10) NFA for $r_{10} = b$

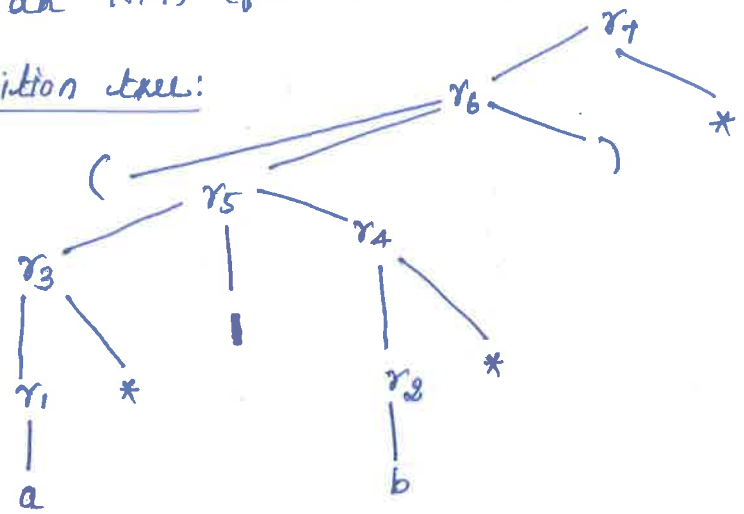


11) NFA for $r_{11} = r_9 r_{10} = (a|b)^* abb$

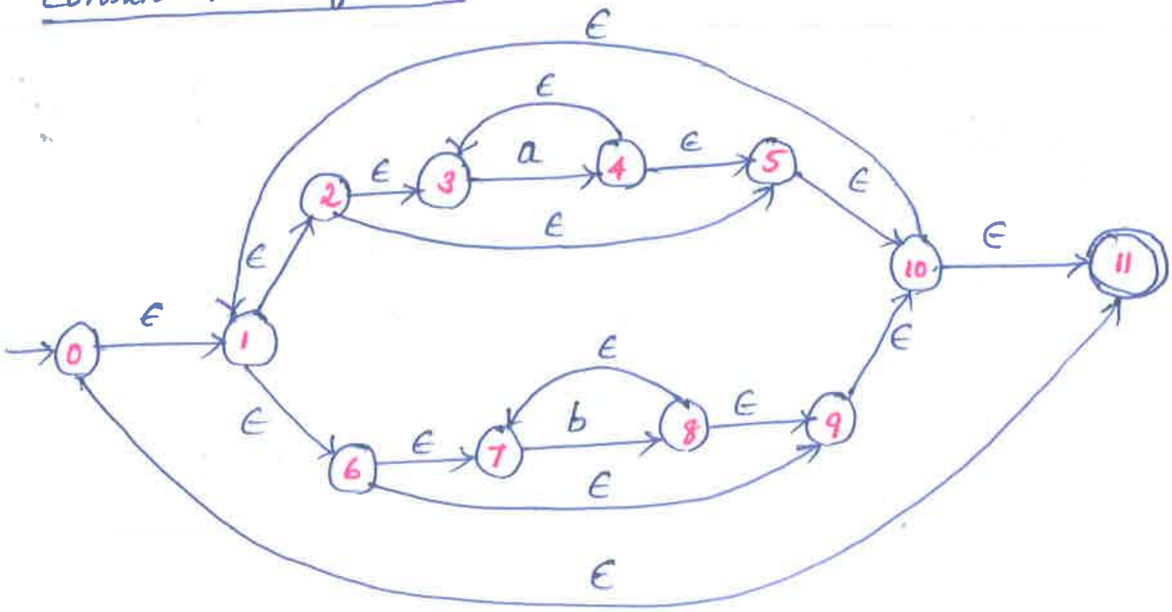


2) Construct an NFA for $(a^* | b^*)^*$

Decomposition tree:

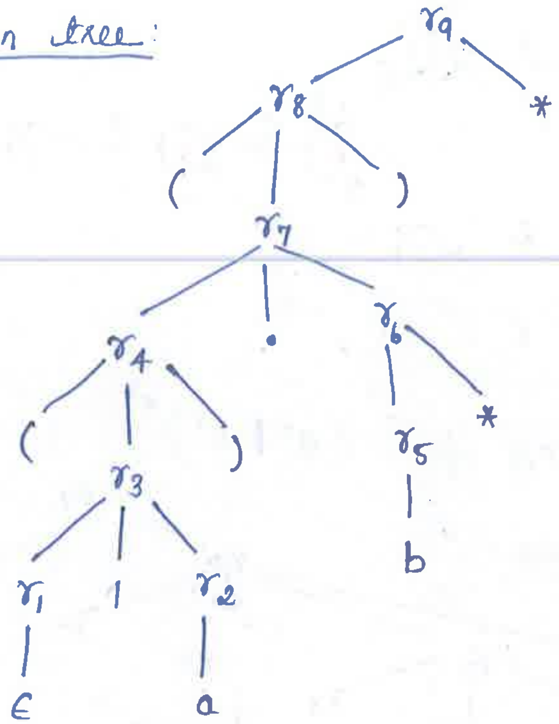


Construction of NFA:

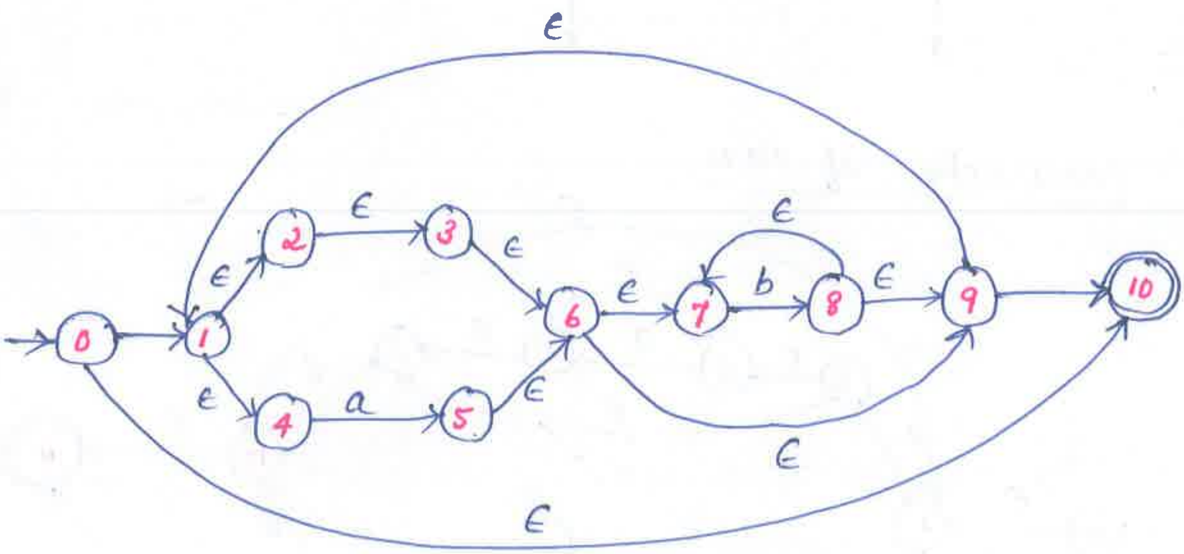


3) Construct NFA for an R.E $((\epsilon|a)b^*)^*$

Decomposition tree:



Construction of NFA:



2) Conversion of NFA to DFA (Subset Construction)

Algorithm:

Input: An NFA N.
Output: A DFA D accepting the same language as N.

Method:

This Algorithm constructs a transition table D_{tran} for D. Each state of D is a set of NFA states and we construct D_{tran} So, D will simulate "in parallel" all possible moves N can make on a given input string.

The functions that describe basic computations on the states of N that are needed in the algorithm are as follows:

Operation	Description.
(i) ϵ_closure(s) →	Set of NFA states reachable from NFA state s on ϵ_transitions alone.
(ii) ϵ_closure(T) →	Set of NFA states reachable from some NFA state s in set T on ϵ_transitions alone;
(iii) move(T, a) →	Set of NFA states to which there is a transition on i/p symbol 'a' from some state s in T. refer book 153

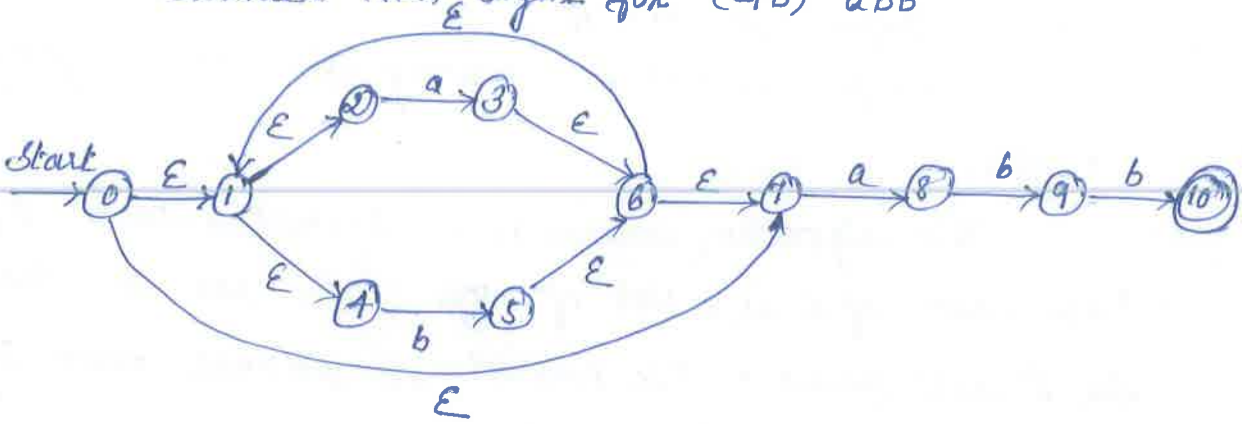
```
initially, ε_closure(S0) is the only state in Dstates, and it is
unmarked;
   while (there is an unmarked state T in Dstate)
   {
      mark T;
      for (each input symbol a)
      {
        U = ε_closure (move(T,a));
        if (U is not in Dstates)
                 add U as an unmarked state to Dstates;
        Dtran[T,a] = U;
      }
   }
```

fig :- The subset Construction

Example:

1) Convert NFA to DFA :

Consider NFA digm. for $(a|b)^*abb$



Solution: step 1 : construct Transition Table D_{tran} for DFA

The start state A of the equivalent DFA is ε_closure (0),

(or) $A = \{0, 1, 2, 4, 7\}$ // reachable states from state 0 via a path all of whose edges have label ε.

The input symbol alphabet is $\{a, b\}$

1) $D_{tran}[A, a]$ is,

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

$= \epsilon\text{-closure}(3, 8) = \epsilon\text{-closure}(3) \cup \epsilon\text{-closure}(8)$ write for ε states except 0 state

$= \{1, 2, 3, 4, 6, 7, 8\}$

$D_{tran}[A, a] = B$

2) $D_{tran}[A, b]$ is,

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

$= \epsilon\text{-closure}(5)$ write for ε except 0

$= \{1, 2, 4, 5, 6, 7\}$

$D_{tran}[A, b] = C$

3) $D_{tran}[B, a]$ is

$\epsilon\text{-closure}(\text{move}(B,a)) = \epsilon\text{-closure}(3, 8) \rightarrow \{1, 2, 3, 4, 6, 7, 8\}$

$D_{tran}[B, a] = B$

4) $D_{tran}[B, b]$

$$\epsilon\_closure(move(B, b)) = \epsilon\_closure(5, 9)$$
$$= \{5, 9, 6, 7, 1, 2, 4\}$$
$$= D$$

5) $D_{tran}[C, a]$

$$\epsilon\_closure(move(C, a)) = \epsilon\_closure(3, 8) \Rightarrow \{1, 2, 3, 4, 6, 7, 8\}$$
$$= B$$

6) $D_{tran}[C, b]$

$$\epsilon\_closure(move(C, b)) = \epsilon\_closure(5) \Rightarrow \{1, 2, 4, 5, 6, 7\}$$
$$= C$$

7) $D_{tran}[D, a]$

$$\epsilon\_closure(move(D, a)) = \epsilon\_closure(8, 3) \Rightarrow \{1, 2, 3, 4, 6, 7, 8\}$$
$$= B.$$

8) $D_{tran}[D, b]$

$$\epsilon\_closure(move(D, b)) = \epsilon\_closure(10, 5) \Rightarrow \{10^*, 5, 6, 7, 1, 2, 4\}$$
$$= E \text{ (final state)}$$

9) $D_{tran}[E, a]$

$$\epsilon\_closure(move(E, a)) = \epsilon\_closure(8, 3) \Rightarrow \{1, 2, 3, 4, 6, 7, 8\}$$
$$= B$$

10) $D_{tran}[E, b]$

$$\epsilon\_closure(move(E, b)) = \epsilon\_closure(5) \Rightarrow \{1, 2, 4, 5, 6, 7\}$$
$$= C.$$

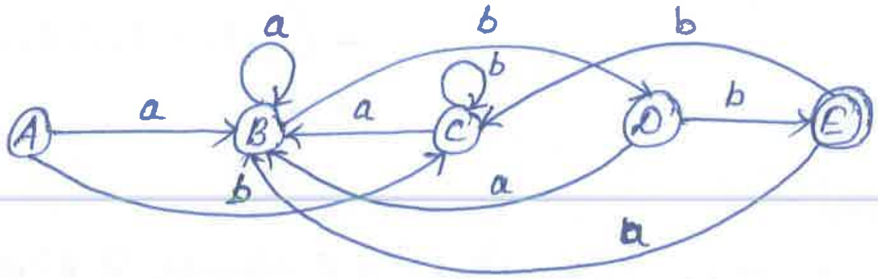
Transition Table:

DFA STATES	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

Since A is a start state and E is a final state.

Step IV

Construction of DFA



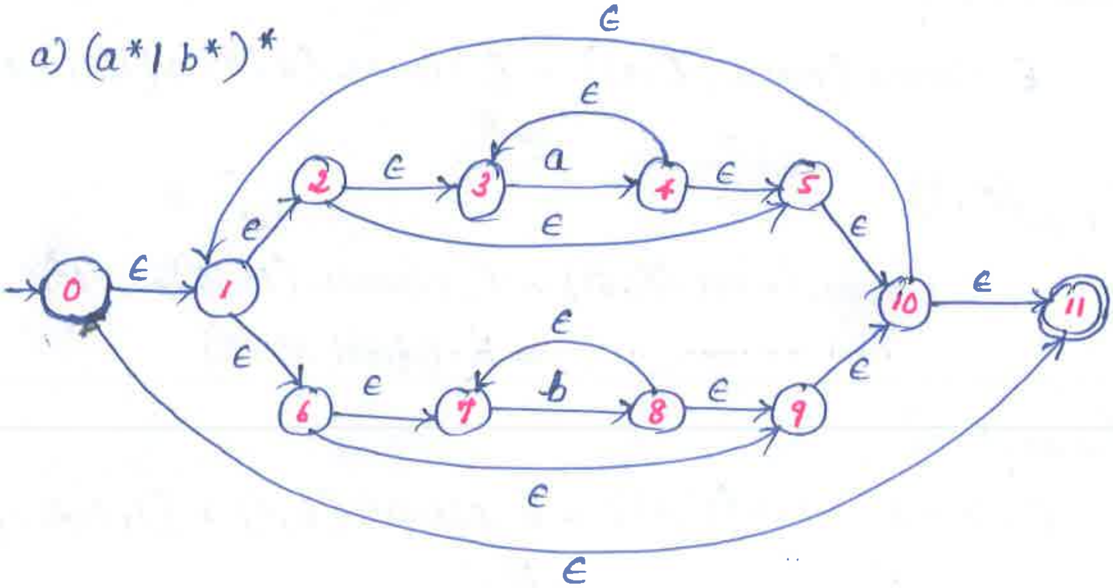
2) Construct NFA to DFA.

a) NFA for $(a^* | b^*)^*$

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

show the sequence of moves made by each in processing the input string ababbab

a) $(a^* | b^*)^*$



Solution:

Step I: Construct Transition Table Dtran for DFA

ϵ_closure(0) = $\{0, 1, 2, 3, 6, 7, 5, 10, 11^*, 9\} = {}^*A$

Input symbol alphabet is $\{a, b\}$

1) Dtran[A, a] is

ϵ_closure(move(A, a)) = ϵ_closure(4)

$= \{4, 5, 10, 1, 2, 3, 11, 9, 6, 7\}$

$= \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11^*\}$

$= {}^*B$

2) $D_{tran}[A,b]$

ϵ_closure(move(A,b)) = ϵ_closure(8)

= {8, 9, 10, 11, 1, 2, 3, 5, 7, 6}

= {1, 2, 3, 5, 6, 7, 8, 9, 10, 11}

= *C

3) $D_{tran}[B,a]$

ϵ_closure(move(B,a)) = ϵ_closure(4)

= B

4) $D_{tran}[B,b]$

ϵ_closure(move(B,b)) = ϵ_closure(8)

= C

5) $D_{tran}[C,a]$

ϵ_closure(move(C,a) = ϵ_closure(4)

= B

6) $D_{tran}[C,b]$

ϵ_closure(move(C,b) = ϵ_closure(8)

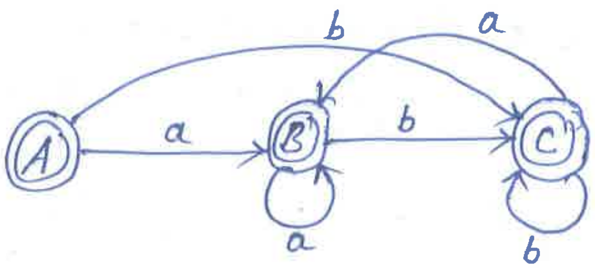
= C

Transition table:

DFA STATES	Input symbol	
	a	b
→ *A	B	C
*B	B	C
*C	B	C

Since, A is start state and A, B, C are the final states

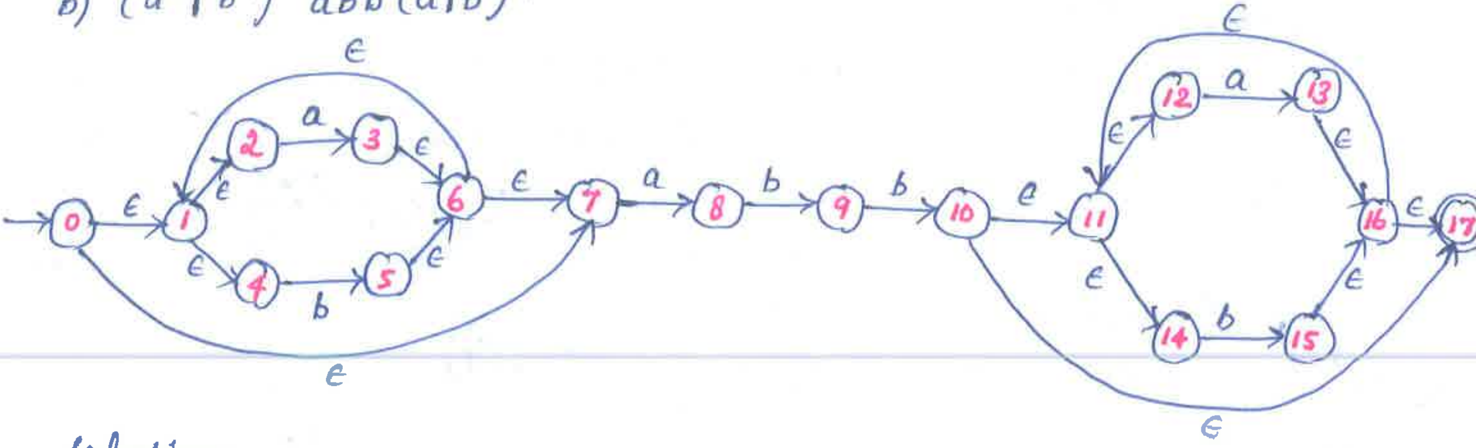
Step II Construction of DFA



The sequence of moves made by each in processing the input string ababbab

$\rightarrow A \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{b} C \xrightarrow{a} B \xrightarrow{b} C$

b) $(a \mid b)^* abb (a|b)^*$



Solution:

Step I: Construct Transition Table D_{tran} for DFA

$\epsilon\_closure(0) = \{0, 1, 2, 4, 7\} = A$

Input symbol $\{a, b\}$

1) $D_{tran}[A, a]$

$\epsilon\_closure(move(A, a)) = \epsilon\_closure(3, 8)$
$= \{3, 6, 1, 2, 4, 7, 8\}$
$= \{1, 2, 3, 4, 6, 7, 8\} = B$

2) $D_{tran}[A, b]$

$\epsilon\_closure(move(A, b)) = \epsilon\_closure(5)$
$= \{5, 6, 7, 1, 2, 4\}$
$= \{1, 2, 4, 5, 6, 7\} = C$

3) $D_{tran}[B, a]$

$\epsilon\_closure(move(B, a)) = \epsilon\_closure(3, 8)$
$= \{1, 2, 3, 4, 6, 7, 8\} = B$

4) $D_{tran}[B, b]$

$\epsilon\_closure(move(B, b)) = \epsilon\_closure(5, 9)$
$= \{1, 2, 4, 5, 6, 7, 9\} = D$

5) $D_{tran}[C, a]$

$\epsilon\_closure(move(C, a)) = \epsilon\_closure(3, 8)$
$= B$

6) $D_{tran}[C, b]$

$\epsilon\_closure(move(C, b)) = \epsilon\_closure(5)$
$= C$

7) $D_{tran}[D, a]$

$$\epsilon\_closure(move(D, a)) = \epsilon\_closure(3, 8)$$
$$= B$$

8) $D_{tran}[D, b]$

$$\epsilon\_closure(move(D, b)) = \epsilon\_closure(5, 10)$$
$$= \{1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17^*\} = {}^*E$$

9) $D_{tran}[E, a]$

$$\epsilon\_closure(move(E, a)) = \epsilon\_closure(3, 8, 13)$$
$$= \{1, 2, 3, 4, 6, 7, 8, 13, 16, 17, 11, 12, 14\}$$
$$= \{1, 2, 3, 4, 6, 7, 8, 11, 12, 13, 14, 16, 17^*\}$$
$$= {}^*F$$

10) $D_{tran}[E, b]$

$$\epsilon\_closure(move(E, b)) = \epsilon\_closure(5, 15)$$
$$= \{1, 2, 4, 5, 6, 7, 5, 16, 17, 11, 12, 14\}$$
$$= \{1, 2, 4, 5, 6, 7, 11, 12, 14, 15, 16, 17^*\}$$
$$= {}^*G$$

11) $D_{tran}[F, a]$

$$\epsilon\_closure(move(F, a)) = \epsilon\_closure(3, 8, 13)$$
$$= F$$

12) $D_{tran}[F, b]$

$$\epsilon\_closure(move(F, b)) = \epsilon\_closure(5, 9, 15)$$
$$= \{1, 2, 4, 5, 6, 7, 9, 11, 12, 14, 15, 16, 17^*\}$$
$$= {}^*H$$

13) $D_{tran}[G, a]$

$$\epsilon\_closure(move(G, a)) = \epsilon\_closure(3, 8, 13)$$
$$= F$$

14) $D_{tran}[G, b]$

$$\epsilon\_closure(move(G, b)) = \epsilon\_closure(5, 15)$$
$$= G$$

15) $D_{tran}[H, a]$

$$\epsilon\_closure(move(H, a)) = \epsilon\_closure(3, 8, 13)$$
$$= F$$

16) $D_{tran}[H, b]$

$\epsilon\text{-closure}(move(H, b)) = \epsilon\text{-closure}(5, 10, 15)$

$= \{1, 2, 4, 5, 6, 7, 10, 11, 12, 14, 17^*, 15, 16\}$

$=^* I$

17) $D_{tran}[I, a]$

$\epsilon\text{-closure}(move(I, a)) = \epsilon\text{-closure}(3, 8, 13)$

$= F$

18) $D_{tran}[I, b]$

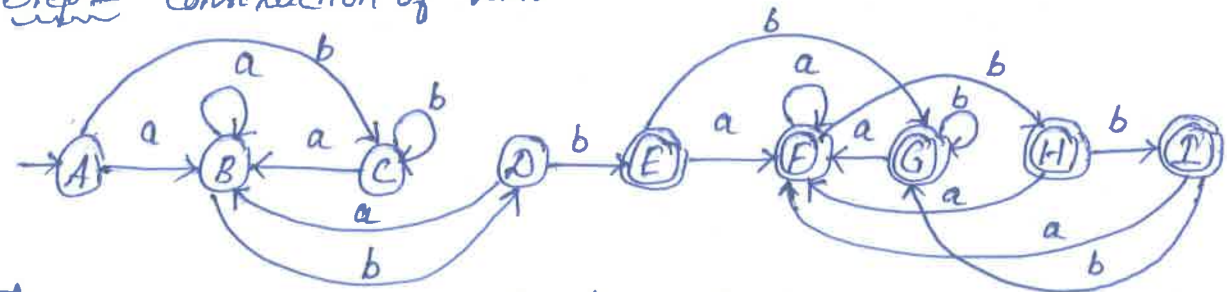
$\epsilon\text{-closure}(move(I, b)) = \epsilon\text{-closure}(5, 15)$

$= G$

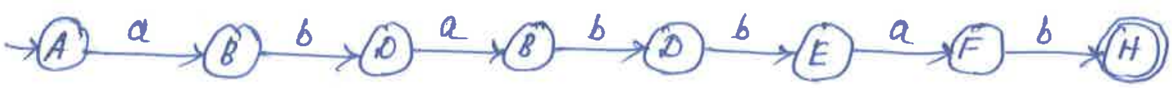
Transition Table:

DFA States	Input Symbol	
	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
* E	F	G
* F	F	H
* G	F	G
* H	F	I
* I	F	G

Step II construction of DFA



The sequence of moves made by each in processing the i/p "ababbab".



3) Minimization of DFA

Algorithm: (Minimizing the number of states of a DFA)

Input : A DFA D with set of states S, input alphabet Σ, start state s_0, and set of accepting states F.

Output: A DFA D' accepting the same language as D and having as few states as possible.

Method:

1) start with an initial partition π with 2 groups. The accepting states F and non-accepting states $S-F$.

2) Apply the procedure to construct a new partition π_{new}.

```
initially, let πnew = π;
for (each group G of π)
{
  partition G into subgroups such that 2 states
  s and t are in the same subgroup if and only if
  for all input symbols 'a', states s and t have
  transitions on 'a' to states in the same group of π.
  /* at worst, a state will be in a subgroup by itself */
  replace G in πnew by the set of all subgroups formed;
}
```

3) If $\pi_{new} = \pi$, let $\pi_{final} = \pi$ and continue with step (4). Otherwise repeat step(2) with π_{new} in place of π.

4) Choose one state in each group of π_{final} as the <u>representative</u> for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows.

a) The start state of D' is the representative of the group containing the start state D.

b) The accepting states of D' are the representatives of those groups that containing an accepting state of D.

c) Let 's' be the representative of some group G of π_{final}, and let the transition of D from 's' to t, on input 'a'.

Let r be the representative of t's group H. Then in D', there is a transition from s to r on i/p a. Note that in D, every state in group G must go to some state of group H on i/p a, or else group G would have been split according to step (2)

Example:

1) Minimize the states in the following DFA's.

State	Input symbol	
	a	b
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

Step 1: Initial Partition π consist of 2 groups. {A,B,C,D} {E}

{E} → cannot split because it has only one state.

{A, B, C, D} → non-accepting states

{E} → accepting state.

Step 2: Construct π_{new}.

Input symbol → {a,b}

Round (i) consider the group {A,B,C,D}

→ On input 'a', each of these states goes to B, member of same group

→ on input 'b', {A,B,C} go to members of group {A,B,C,D} while D goes to E, a member of another group.

So, π_{new} = {A,B,C} {D} {E}

Round(ii) Consider the group {A, B, C}

→ On input 'a', each of these states goes to B

→ On input 'b', {A, C} go to the members of {A, B, C} while B goes to D, a member of another group.

So, $\pi_{new} = \{A, C\}\{B\}\{D\}\{E\}$

↓

Round (iii): cannot split this group because A & C go to same state for the input a & b.

Step 3: construct minimum-state DFA.

→ It has 4 states corresponding to the 4 groups of π_{final}.

→ Let us pick A, B, D, E as the representatives of these groups.

↓

(representative of {A,C} group)

State	Input symbol a	b
→A	B	A
B	B	D
D	B	E
*E	B	A

A → is the initial state

E → accepting state

2) Minimize the states in the following DFA's

States	Input symbol a	b
→A	B	A
B	B	C
C	B	D
*D	E	D
*E	E	F
*F	E	D

Step 1: Initial partition π consist of 2 groups {A, B, C} {D, E, F}

{A, B, C} → non-accepting states
{D, E, F} → accepting states:

Step 2: Construct π_{new}

Input symbol {a, b}

Round (i): consider the group {A, B, C}

→ on input 'a', each states go to B, a member of same group.
→ On input 'b', {A, B} go to members of same group {A, B, C} while C goes to D, the member of another group {D, E, F}

So, π_{new} = {A, B} {C} {D, E, F}

Round (ii): consider the group {A, B}

→ On input 'a', {A, B} go to members of same group.
→ On input 'b', {B} goes to C, a member of another group

So, π_{new} = {A} {B} {C} {D, E, F}

Round (iii) consider the group {D, E, F}

→ On input 'a', {D, E, F} go to members of same group
→ On input 'b', {D, F} have a transition to D, while E has a transition to F.

So, π_{new} = {A} {B} {C} {D, F} {E}

D is the representative

Step 3: Construct minimum state DFA π_{final} = {A} {B} {C} {D} {E}

State	a	b
→A	B	A
B	B	C
C	B	D
* D	E	D
* E	E	D

A → initial state
D, E → accepting state.

Reduced DFA:

$$\pi_{final} = \{A\}\{B\}\{C\}\{D,E\}$$

D is a representative

States	Input symbol	
	a	b
→A	B	A
B	B	C
C	B	D
*D	D	D