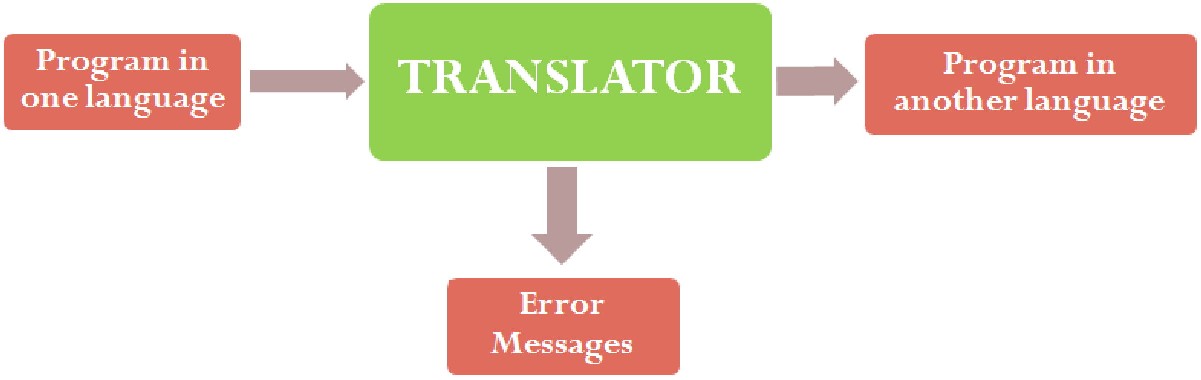
COMPILER DESIGN

UNIT-1

Translators

- A translator is a program that
  - takes as input a program written in one programming language (the source language)
  - produces as output a program in another language (object or target language).
- It takes program written in source code and converts it into machine code.
- It also discovers and identifies the error during translation.



Need for Translators

- Binary representation used for communication within a computer system is called machine language.
- With machine language we can communicate with the computer in terms of bits.

- There are three main kinds of programming languages:
  - Machine language
  - Assembly language
  - High Level language

Machine Language

- Machine language
  - Computer can understand only one language i.e. machine language.
  - Normally written as strings of binary 0's and 1's.

- A program written in machine language has the following disadvantages:
  - Difficult to read & understand
  - Machine dependent
  - Error prone.

- Due to these limitations, some languages have been designed which are easily understandable by the user and also machine independent.
- A **software program** is required which can convert this machine independent language into machine language.
- This software program is called a **Translator.**

Assembly Language

- To overcome limitations of machine language, assembly language was introduced in 1952.
- Instructions can be written in the form of letters and symbols and not in binary form.
- For example,
  - to add two numbers

    ADD A, B

- **Advantages** of Assembly language over machine language:
  - Uses mnemonics (symbols) instead of bits.
  - More readable.
  - Permits programmers to use labels to identify and name particular memory words that holds instructions or data.
  - Locating and correcting errors is easier.

- The main **disadvantage** of assembly language is that the programs written in assembly language are **machine dependent**.

High Level Language

- A high level language writes a program which can be easily understandable by the user.
- While writing program in high level language, programmer need not know the internal structure of the computer.
- Machine independent language.
  - For example,
    - to add two numbers simply write,

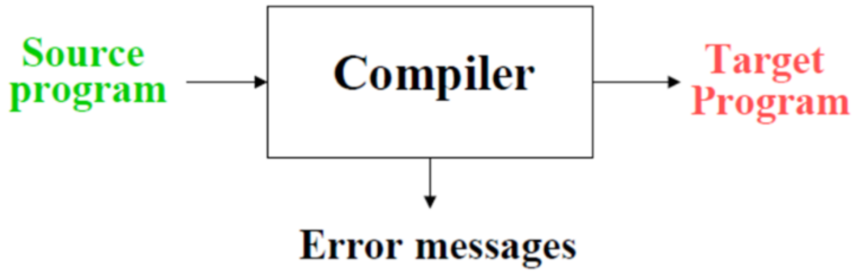
      C=a+b
- Some of the high level languages are:
  - C, C++, Java etc.

Types of Translators

- Compilers
- Assemblers
- Interpreters
- Macros
- Preprocessors
- Linkers & Loaders

Compilers

- A compiler is a translator which converts high level language (source program) into low level language (object program or machine program)



- Also generates diagnostic messages encountered during compilation of program.

Advantages & Disadvantages of Compiler

- Advantages:
  - Compiler translates complete program in a single run.
  - It takes less time.
  - More CPU utilization.
  - Easily supported by many high level languages like C, C++ etc.

- Disadvantages:
  - Not flexible.
  - Consumes more space.
  - Error localization is difficult.
  - Source program has to be compiled for every modification.

Interpreters

- An interpreter like compiler is a translator that translates high level language program(source program) into low level language (object program or machine program)
- An interpreter reads a source program written in a high level language as well as data for this program.
- It runs the program against the data to produce results.

- Advantages:
  - Translates the program line by line.
  - Flexible
  - Error localization is easier.

- Disadvantages:
  - Consumes more time as it is slower.
  - CPU utilization is less.
  - Less efficient.

Other Translators

- Assembler
  - An assembler is a translator that translates the assembly language instructions into machine code.
- Macros
  - A macro is a translator that translates assembly language instructions into machine code.
  - It is a variation of assembler.
- Preprocessor
  - It is a program that transform the source code before compilation.
  - Preprocessor tells the compiler to include some header files into the program.

Linkers and Loaders

- A linker is a program that combines object modules to form an executable program
- Loader is a program which accepts the input as linked modules & loads them into main memory for execution by the computer.

Analysis-Synthesis Model of Compiler

- There are two parts of compilation:
- Analysis
  - Breaks up the source program into constituent pieces
  - Creates an intermediate representation of source program
- Synthesis
  - Constructs the desired source program from an intermediate representation

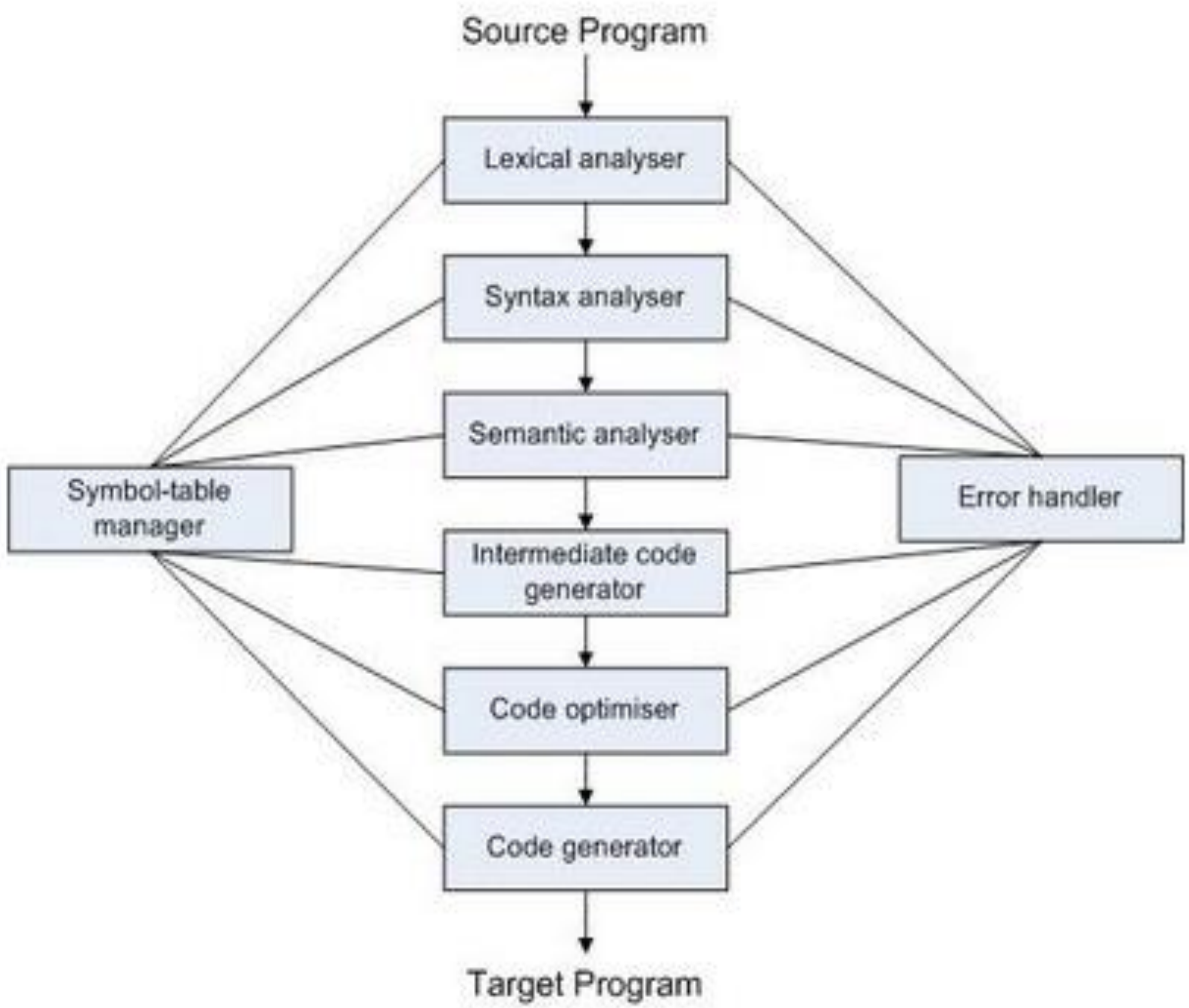
Analysis of source program

- In compilation, analysis consists of three parts:
  - Linear analysis
    - Stream of characters in source program is read from left to right and grouped into tokens.
    - These tokens are the sequence of characters having a collective meaning.
  - Hierarchical analysis
    - Characters or tokens are grouped hierarchically into nested collections with collective meaning.
  - Semantic analysis
    - Certain checks are performed to ensure that the components of a program fit together meaningfully.

Phases of a Compiler

- A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions.
- It is difficult to implement the whole process in one step.
- This process is broken down into subtasks called **Phases**.
- Each phase is interdependent on other phase.
- Output of one phase will be input to another phase.
- First phase of compiler takes as input source program and last phase produces the required object program.

Phases of Compiler



Lexical Analysis

- The lexical analysis is an interface between the source program and the compiler.
- It reads the source program character by character separating them into groups that logically belongs together.
- The sequence of character groups are called **tokens**.
- The character sequence that forms a token is called a **"lexeme"**.
- The software that performs lexical analysis is called a **lexical analyzer** or **scanner**.

An Example

- Consider the statement:

  Sum:=bonus+basic*50

- The statement is grouped into 7 tokens as follows:

S. No.	Lexeme (Sequence of characters)	Token (Category of lexeme
1	Sum	Identifier
2	:=	Assignment operator
3	Bonus	Identifier
4	+	Addition operator
5	Basic	Identifier
6	*	Multiplication operator
7	50	Integer constant

- The output of lexical analysis phase is of the form:

  [id1,500] := [id2,700] + [id3,800] * [const,900]

Syntax analysis

- Second phase of compiler and performed by software called **Parser** or **Syntax Analyser**.
- Creates the syntactic structure (Parse Tree)of the given program.
- Parser receives input in the form of token from its previous phase and determines the structural elements of the program and their relationship.
- Then parser constructs a parse tree from various tokens obtained from lexical analyzer.

- There are 2 types of parsers:
- **Bottom-up Parser**
  - Constructs parse tree from leaves & scan towards the root of tree.
- **Top-down Parser**
  - Construct parse tree from root level and move downwards towards leaves.

Semantic Analysis

- Semantic refers to the "meaning of the program".
- Following are the functions performed by semantic analyzer:
  - Type checking
    - Checks or verifies that each operator has operands that are permitted by source language definition or there should by type compatibility between operator & operands.
  - Implicit type conversion
    - Changing one data type to another automatically when data type of operands of an operator are different or there is any mismatch.
    - E.g. int+real $\xrightarrow{\textbf{Type conversion}}$ real+real=real
    - a+b*10 $\xrightarrow{\textbf{Semantic Analysis}}$ **a + b * int to real (10)**

Intermediate Code Generation

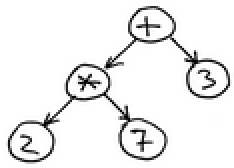
- After performing syntax and semantic analysis on program, compiler generates an intermediate code
  - Intermediate between source language and machine language

- Types of intermediate code:
  - Postfix notation
    - E.g. (a+b)*(c+d)

      ab+cd+*
  - Three address code
    - These are statements of the form c=a op b
    - i.e. there can be atmost three addresses, two for operands and one for result.
    - Each instruction has atmost one operator on right hand side.
    - E.g. d=a*b+c
    - Three address code: **t1=a*b**

      t2=t1+c

      d=t2
  - Syntax trees
    - It is condensed form of parse tree in which leaves are identifiers and interior nodes will be operators.
    - E.g. 2*7+3



Code Optimization

- This phase improves the intermediate code so that faster running object code can be produced.
- It performs the following tasks:
  - Improve target code
  - Eliminate redundant information (common sub-expressions)
  - Remove unnecessary operation.
  - Replaces slow instructions with faster ones.
- Types of Optimization:
  - Local optimization
  - Loop optimization
  - Global data flow analysis

Types of Optimization

- Local Optimization
  - Removes common sub expressions or redundant information
  - E.g.

```
S=P+Q+R
Z=P+Q+M
```

Local Optimization →

```
T1=P+Q
S=T1+R
Z=T1+M
```

- Loop Optimization
  - It is very important to optimize loops so as to increase the performance of the whole program.
  - Statement which computes same value every time when the loop is executed is called "Loop invariant computation".
  - These statements can be takes outside the loop resulting in decreasing the execution time of loop and the whole program.

```
int a=5;
int c;
for (int i=1;i<=5;i++)
{       cout<<i;
        c=a+2;
}
```

Loop Optimization →

```
int a=5;
int c;
c=a+2;
for (int i=1;i<=5;i++)
{       cout<<i;
}
```

- Global Data Flow Analysis
  - Performs optimization by examining the information flow between various data items.
  - Determines information regarding the definition and use of data in a program

Code Generation

- Final phase of compilation process
- Converts optimized intermediate code given by code optimizer into Assembly/ Machine language.

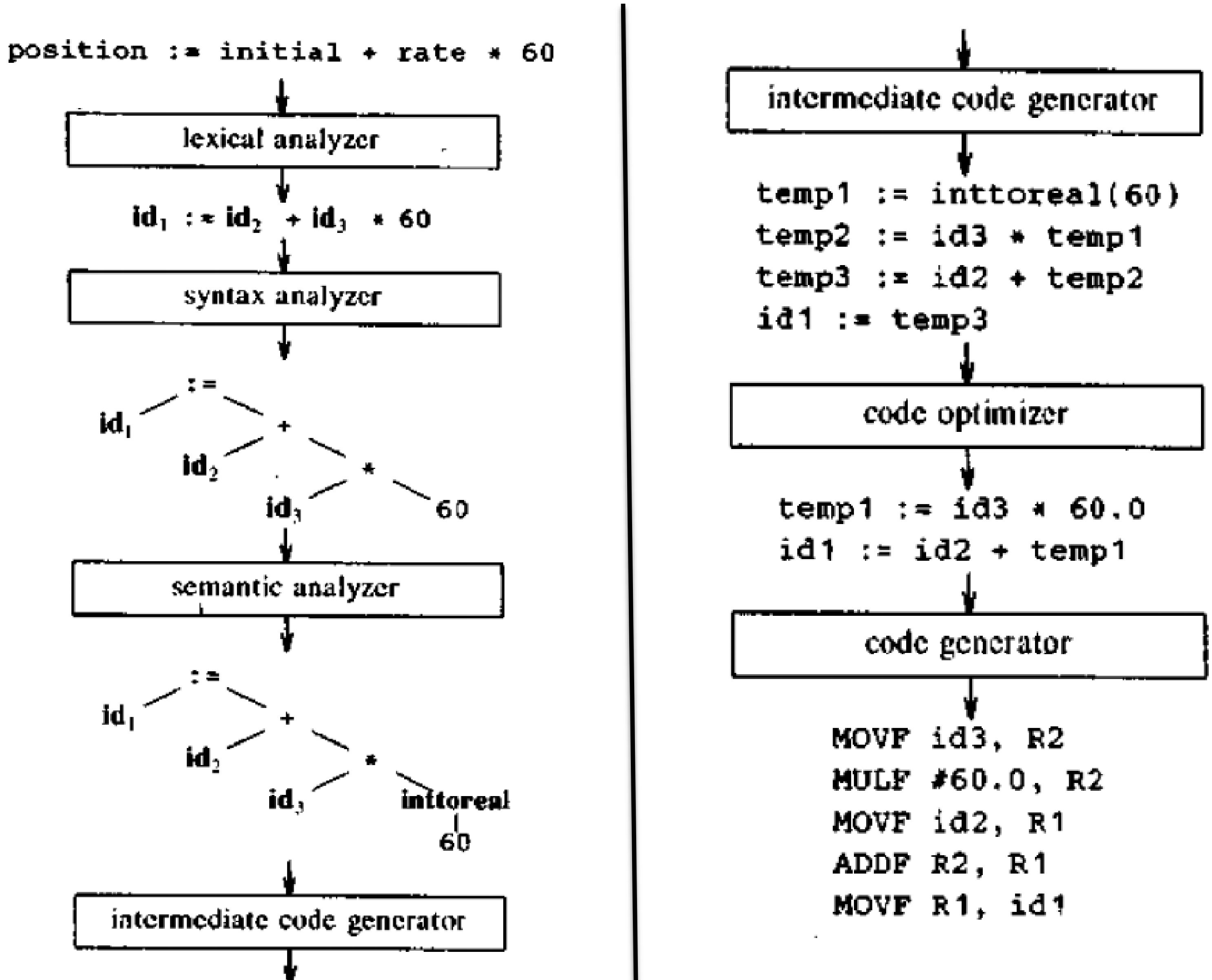
- Main tasks of code generation:
  - Register Allocation
    - What names in a program should be stored in registers
  - Register Assignment
    - In which register, names should be stored.

Symbol Table & Error Handler

- Symbol Table
  - It is a data structure which contains tokens.
  - Keeps record of each token & its attributes (i.e. identifier name, data types, location etc.)
  - This information will be used later by semantic analyzer and code generator.

- Error Handler
  - It detects and reports errors occurred at each phase of compiler.

Example

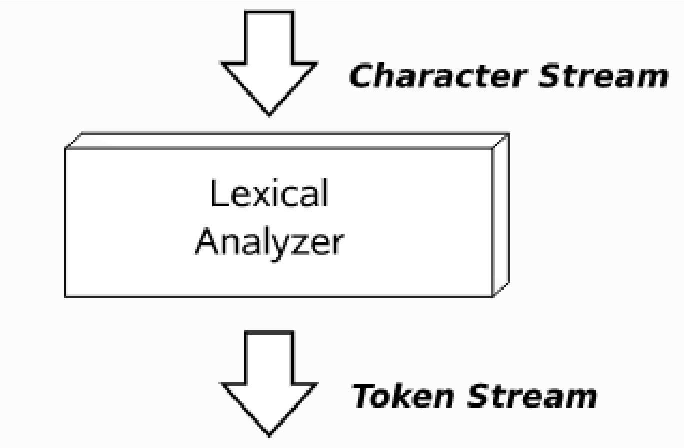

position := initial + rate * 60
lexical analyzer
id_1 := id_2 + id_3 * 60
syntax analyzer
:=
id_1
+
id_2
*
id_3
60
semantic analyzer
:=
id_1
+
id_2
*
id_3
inttoreal
60
intermediate code generator
intermediate code generator
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Compiler Construction Tools

- Software tools developed to create one or more phases of compiler are called compiler construction tools.
- Some of these are:
  - Scanner generator
    - Generates lexical analyzers
    - Basic lexical analyzer is produced by finite automata which takes input in the form of regular expressions.
  - Parser generator
    - Produces syntax analyzers which takes input in the form of programming language based on context free grammar.
  - Syntax directed translation engines
    - Produces intermediate codes.
  - Data flow engines
    - Used in code optimization.
    - Produces optimized code.
  - Automatic code generators
    - Takes input in the form of intermediate code and convert it into machine language.

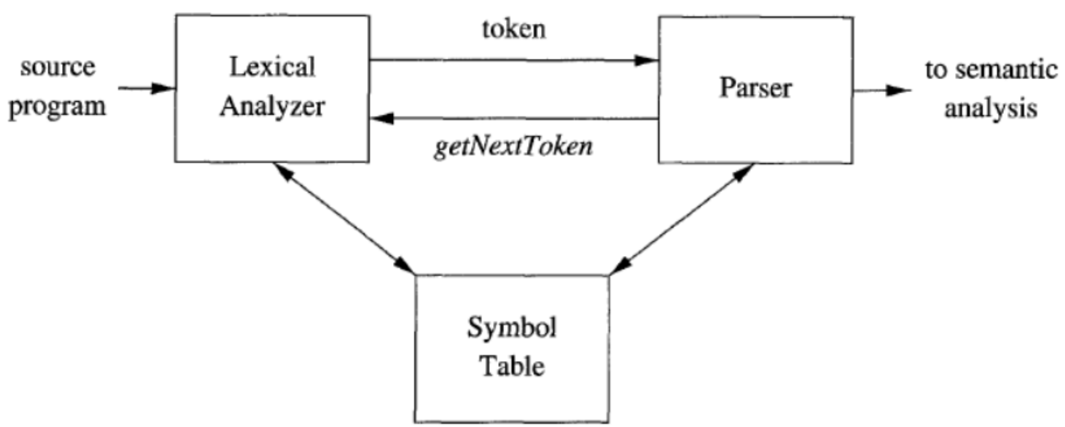
Lexical analysis

- First phase of compiler
- Reads source program one character at a time and convert it into sequence of tokens.



Role of Lexical Analyzer

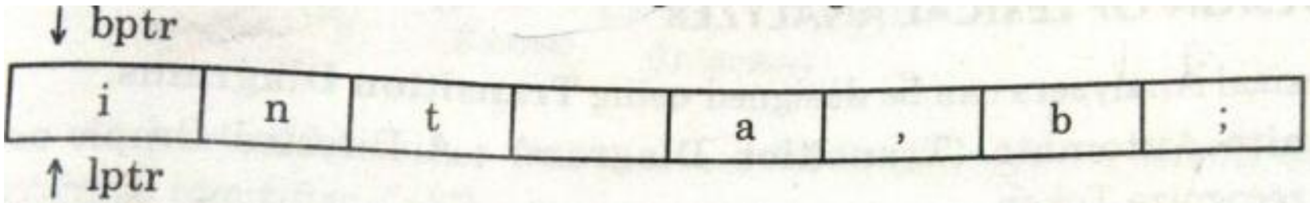
- Main functions of lexical analyzer are:
  - Separate tokens from program and return those tokens to parser.
  - Eliminate comments, white spaces, new line characters etc. from string.
  - Inserts tokens into symbol table.
  - Returns a numeric code fro each token to parser.

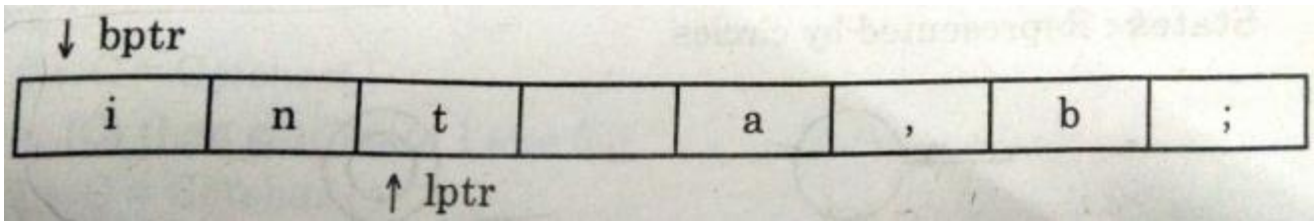


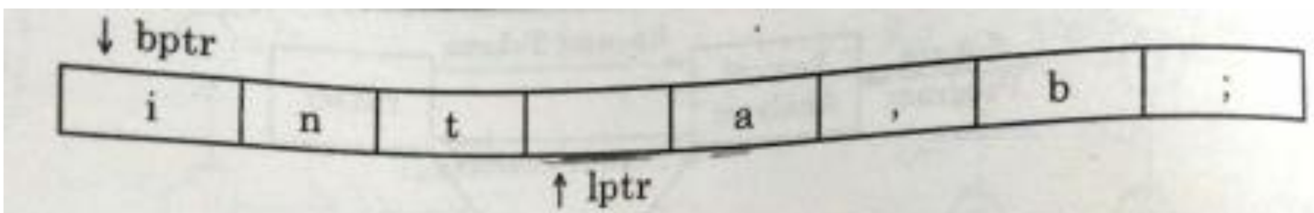
Input Buffering

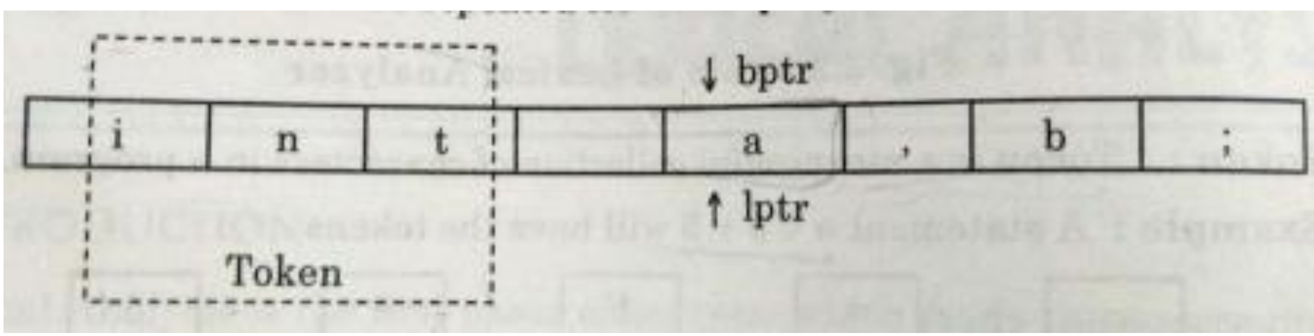
- To identify tokens, lexical analyzer has to access secondary memory every time.
- It is costly and time consuming.
- So, input strings are stored into buffer and scanned by lexical analyzer.
- Lexical analyzer scans input strings from left to right one character at a time to identify tokens.

- It uses two pointers to scan tokens:
  - Begin pointer (bptr)
    - Points to beginning of string to be read.
  - Look ahead pointer (lptr)
    - It moves ahead to search for end of token.


bptr
i
n
t
a
,
b
;
lptr


bptr
i
n
t
a
,
b
;
lptr

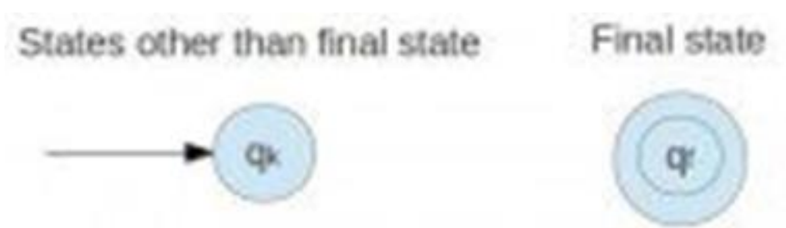

bptr
i
n
t
a
,
b
;
lptr


bptr
i
n
t
a
,
b
;
lptr
Token

Design of Lexical Analyzer

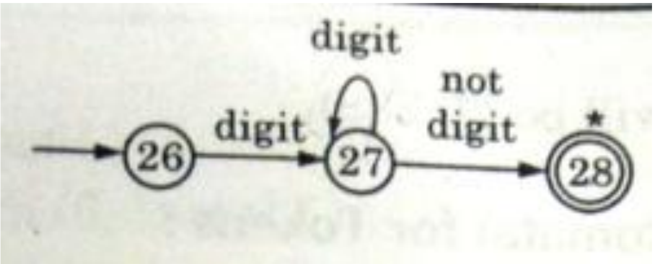
- Can be designed using Finite Automata or Transition Diagrams.
- Finite Automata (Transition Diagram)
  - It is a directed graph or flowchart used to recognize token.

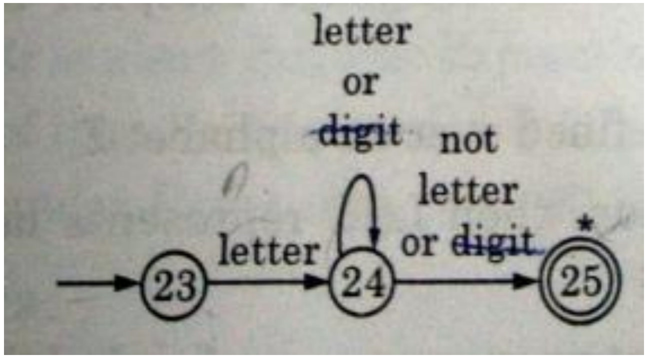
- Transition diagram has 2 parts:
  - States
    - Represented by circles.



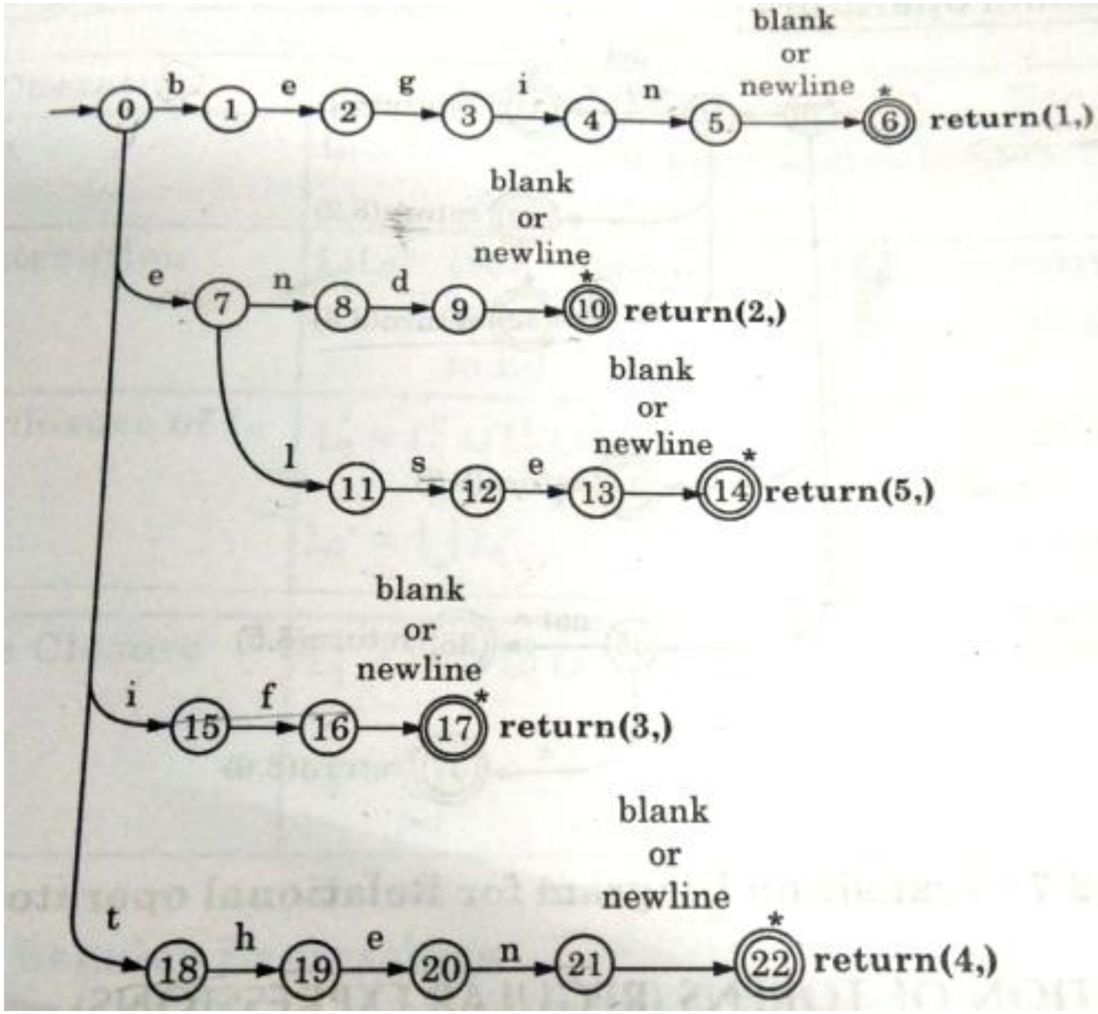
  - Edges
    - States are connected by edges arrows.

Transition Diagram (Constants and Identifiers)

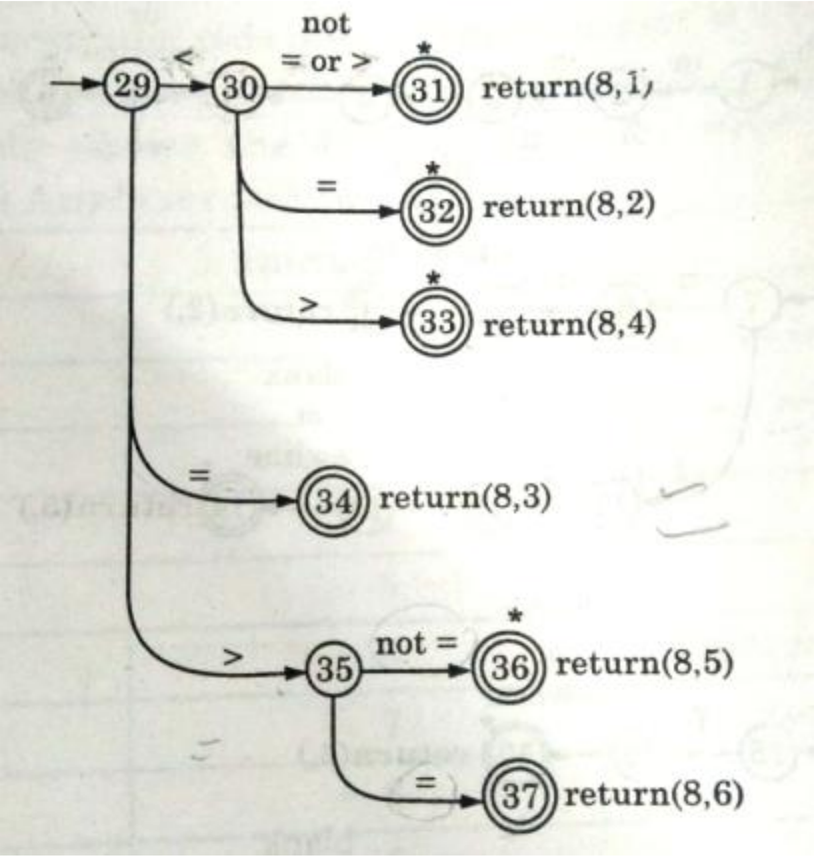




Transition Diagram (Keywords)



Transition Diagram (Relational Operators)



Specification of Tokens(Regular Expressions)

- Regular expressions are used to specify tokens.
- Provides convenient and useful notation.
- RE's define the language accepted by Finite Automata (Transition Diagram).
- RE's are defined over an alphabet Σ.
- If R is a regular expression, then L(R) represents language denoted by RE.

- Language
  - It is a collection of strings over some fixed alphabet.
  - Empty string can be denoted by ε.
  - E.g.
    - If L= set of strings of 0's and 1's of length two
    - Then, L= {00, 01, 10, 11}

Operations on Languages

- Operations that can be performed on Languages are:
  - Union
  - Concatenation
  - Kleen Closure
  - Positive Closure

- Union
  - $L_1 \cup L_2 =$ {set of strings in L_1 & set of strings in L_2}
- Concatenation
  - $L_1L_2 =$ {set of strings in L_1 followed by strings in L_2}
- Kleen Closure
  - $L_1^* = L_1^0 \cup L_1^1 \cup L_1^2 \cup \ldots\ldots.$
- Positive Closure
  - $L_1^+ = L_1^1 \cup L_1^2 \cup L_1^3 \cup \ldots\ldots.$

Rules of Regular Expressions

- ε is a Regular expression.
- Union of two Regular expressions R_1 and R_2 is also a Regular expression.
- Concatenation of two Regular expressions R_1 and R_2 is also a Regular expression.
- Closure of Regular Expression is also a Regular Expression.
- If R is a Regular Expression then (R) is also a Regular Expression.

Algebraic Laws

- $R1 | R2 = R2 | R1$ or $R1+R2 = R2+R1$ (Commutative)

- $R1 | (R2 | R3) = (R1 | R2)+R3$ (Associative)
or $R1+(R2+R3) = (R1+R2)+R3$

- $R1(R2R3) = (R1R2)R3$ (Associative)

- $R1(R2 | R3) = R1R2 \mid R1R3$ (Distributive)
or $R1(R2+R3) = R1R2+R1R3$

- $\varepsilon R = R \varepsilon = R$ (Concatenation)

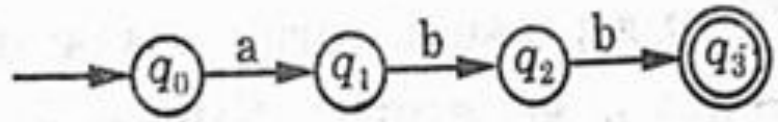
Recognition of Token (Finite Automata)

- It is a machine or a recognizer for a language that is used to check whether string is accepted by a language or not.
- In Finite Automata,
  - **Finite** means finite number of states .
  - **Automata** means Automatic machine which works without any interference of human being.

Finite Automata

- FA can be represented by 5 tuple (Q, Σ, δ, q_0, F)
  - Where,
    - Q: finite non empty set of states
    - Σ: Finite set of input symbols
    - δ: Transition function
    - q_0 : Initial state
    - F: Set of final states

Example *Design Finite Automata which accepts string "abb"*



States : $Q = \{q_0, q_1, q_2, q_3\}$

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

Transition Function δ : $\{\delta\ (q_0, a) = q_1,\ \ \delta\ (q_1, b) = q_2, \delta(q_2, b) = q_3\}$

Initial State : q_0

Final State (F) : $\{q_3\}$

Types of Finite Automata

- **Deterministic Finite Automata**
  - Deterministic means on each input there is one and only one state to which automata can have transition from its current state

- **Non-Deterministic Finite Automata**
  - Non-Determinsitic means there can be several possible transitions.
  - Output is non-deterministic for a given output.

Deterministic Finite Automata (DFA)

- DFA is a 5 tuple $(Q, \Sigma, \delta, q_0, F)$
  - Where,
    - Q: finite non empty set of states
    - Σ: Finite set of input symbols
    - δ: Transition function to move from current state to next state.

      $$\boldsymbol{\delta : Q \times \Sigma \rightarrow Q}$$

    - q_0 : Initial state
    - F: Set of final states

Non- Deterministic Finite Automata (NFA)

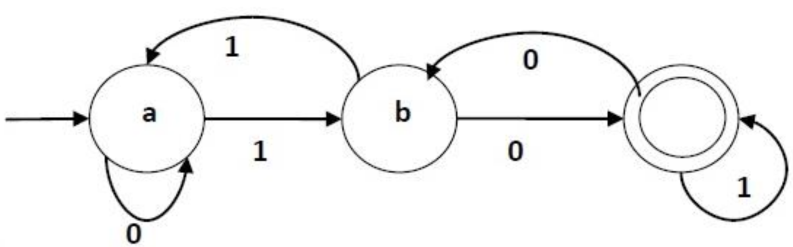
- NFA is a 5 tuple $(Q, \Sigma, \delta, q_0, F)$
  - Where,
    - Q: finite non empty set of states
    - Σ: Finite set of input symbols
    - δ: Transition function to move from current state to next state.

      $$\boldsymbol{\delta : Q \times \Sigma \text{ -> } 2^Q}$$
    - q_0 : Initial state
    - F: Set of final states

Difference between DFA and NFA

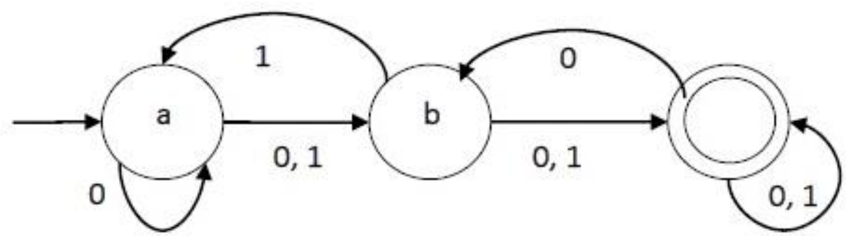
DFA

- Every transition from one state to other is unique & deterministic in nature.
- Null transitions (ε) are not allowed.
- Transition function
  $\boldsymbol{\delta} : Q \times \Sigma \text{->} Q$
- Requires less memory as transitions & states are less.



NFA

- There can be multiple transitions for an input i.e. non-deterministic.
- Null transitions (ε) are allowed means transition from current state to next state without any input.
- Transition function
  $\boldsymbol{\delta} : Q \times \Sigma \text{->} 2^Q$
- Requires more memory.



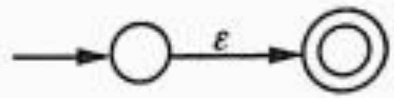
Conversion of Regular expression to NFA

Input : A Regular Expression R

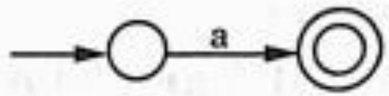
Output : NFA accepting language denoted by R

Method :

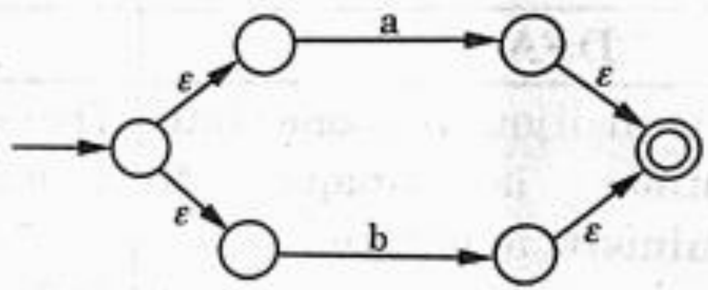
1. For ε, NFA is



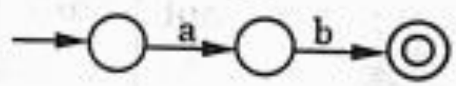
2. For a, NFA is



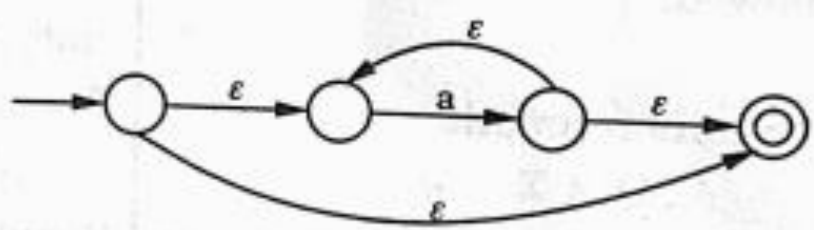
3. For a + b, or a | b NFA is



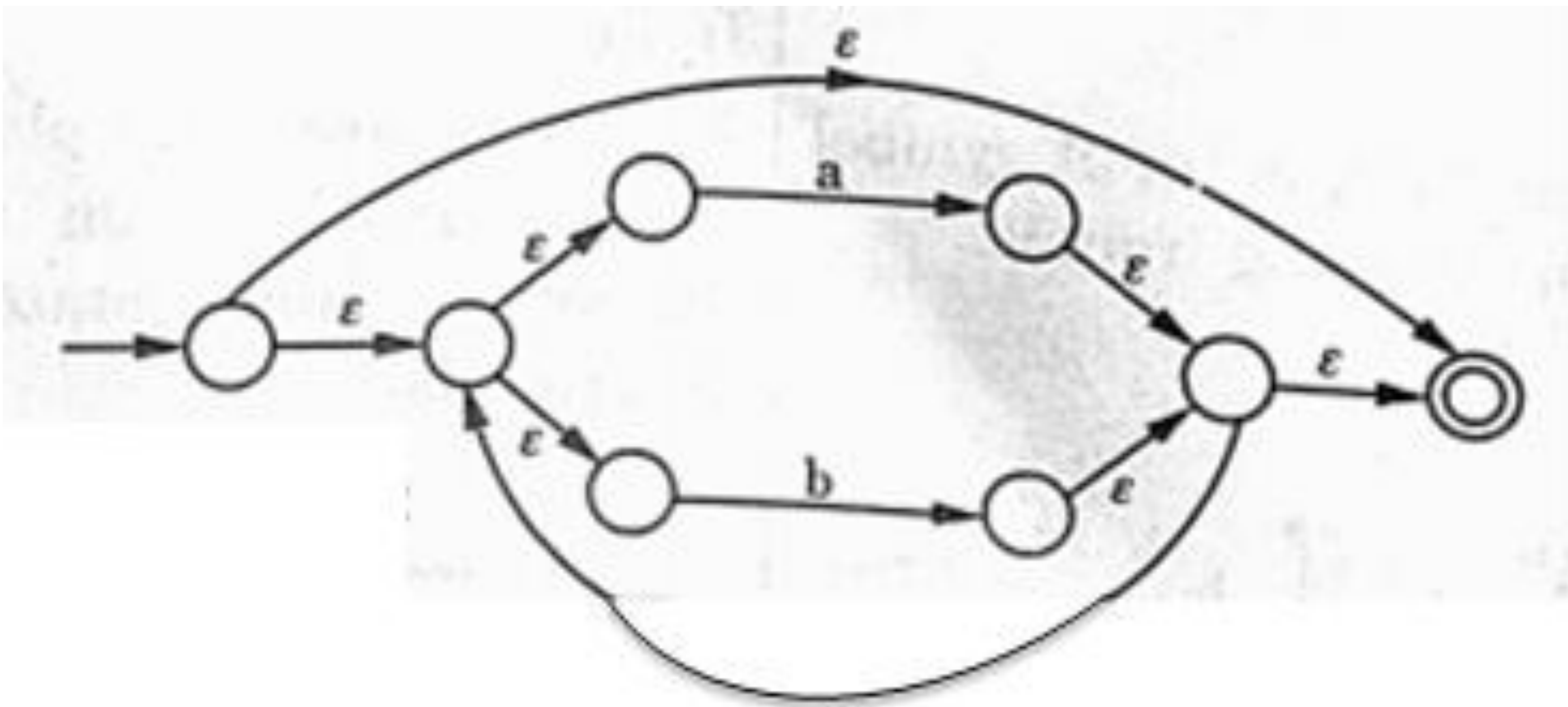
4. For ab, NFA is



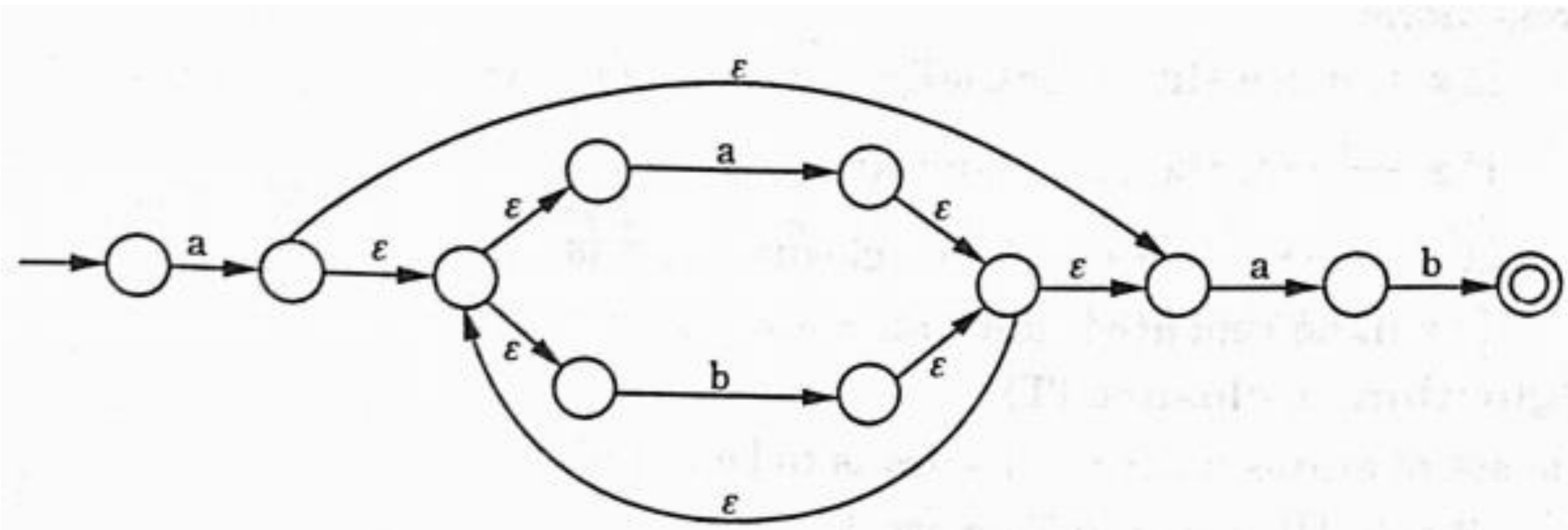
5. For a*, NFA is



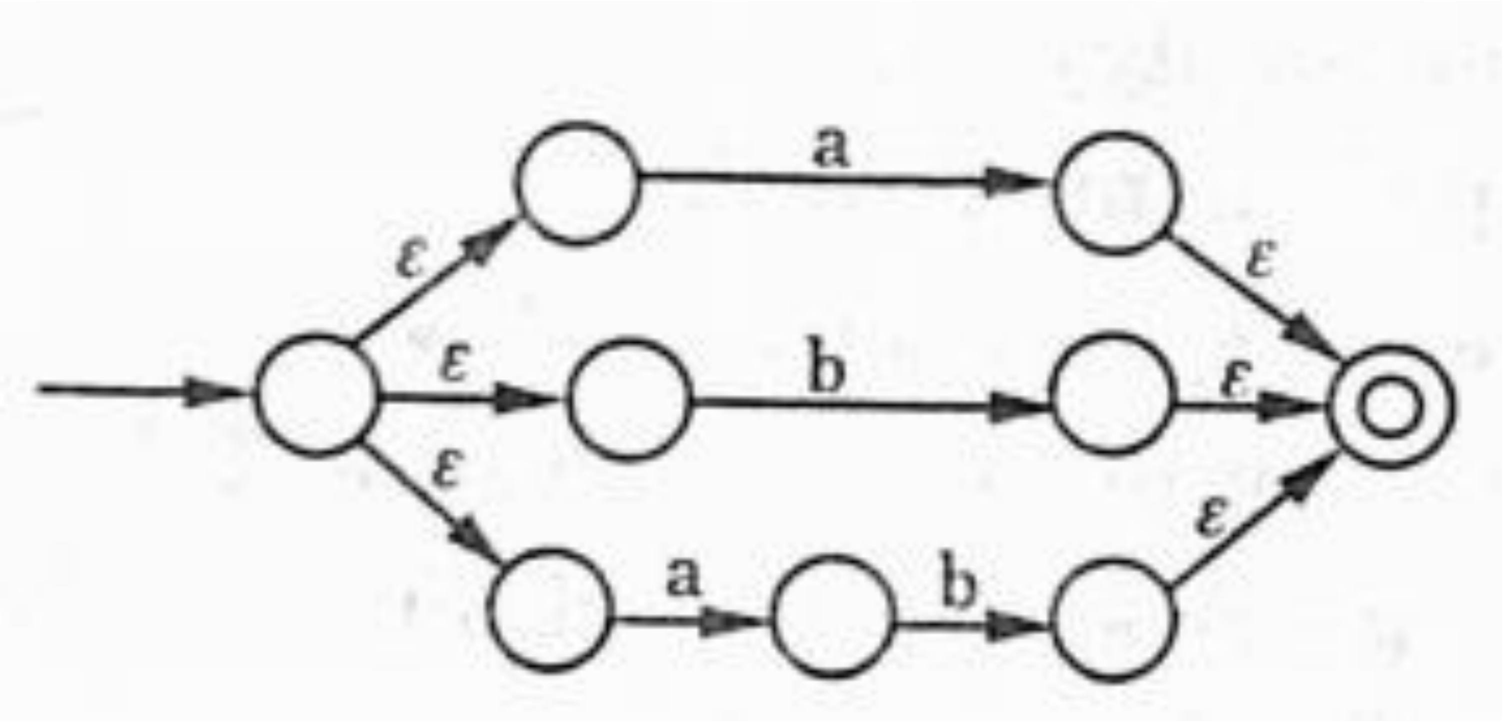
NFA for RE (a+b)*



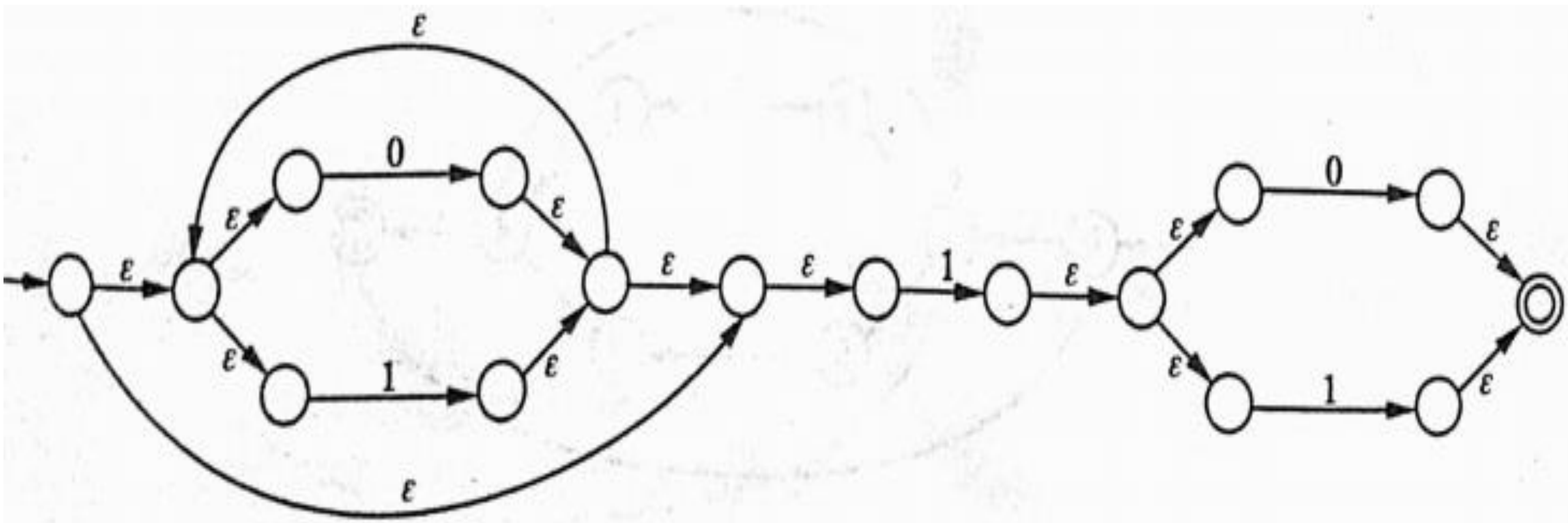
NFA for a(a+b)*ab



NFA for a+b+ab



NFA for (0+1)*1(0+1)



ε-closure (s)

- ε-closure (s): it is a set of states that can be reached from state s on ε-transitions alone.

1. If s, t, u are states. Initially, ε- closure (s) = {s}
2. If $s \xrightarrow{\varepsilon} t$, then ε- closure (s)= {s,t}

   If $s \xrightarrow{\varepsilon} t \xrightarrow{\varepsilon} u$, then ε-closure (s) = {s,t,u}

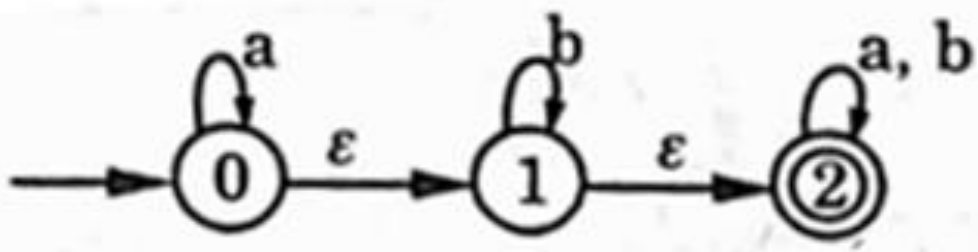
   It will be repeated, until all states are covered.

Algorithm : ε-closure (T)

T is set of states whose ε-closure is to be found

```
1.  Push All states in T on stack.
2.  ε-closure (T) = T
3.  While (stack not empty)
4.  {     Pop s, top element of Stack
5.          for each state t, with edge s --ε--> t
6.                {If t is not present in ε-closure (T)
7.                      {ε– closure (T) = ε-closure (T) ∪{t}
8.                      Push t on stack
9.  }}}
```

Example: Find ε-closure of all states

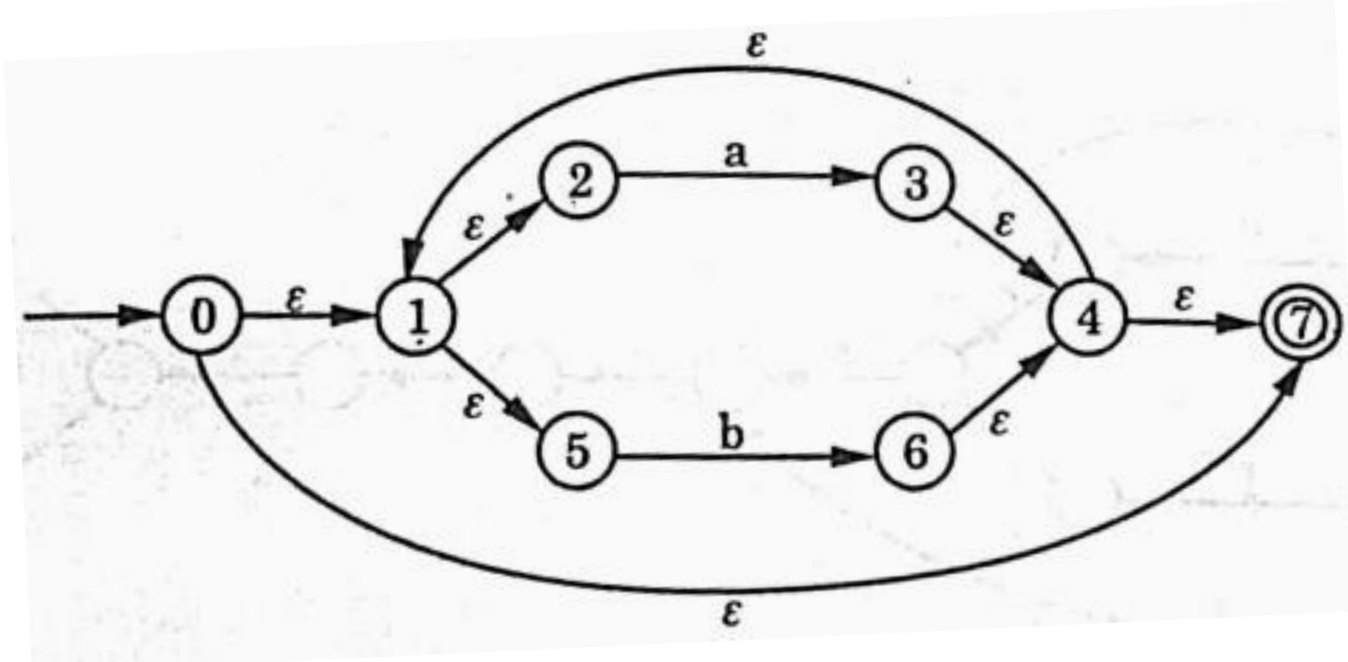


ε–closure $(0) = \{0,1,2\}$

ε–closure $(1) = \{1,2\}$

ε-closure $(2) = \{2\}$.

Example: Find ε-closure of states 0,1,4

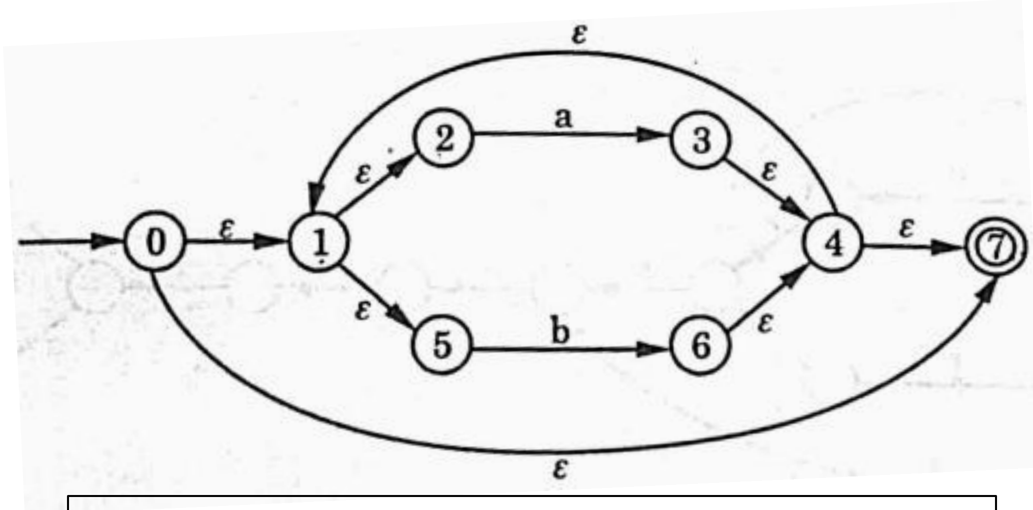


ε–closure $(0) = \{0,1,2,5,7\}$

ε–closure $(1) = \{1,2,5\}$

ε-closure $(4) = \{4,7,1,2,5\}$.

Example: Find ε-closure of all states



ε-closure(0) = {0, 1, 2, 5, 7}
ε-closure(1) = {1, 2, 5}
ε-closure(2) = {}
ε-closure(3) = {}
ε-closure(4) = {4, 7, 1, 2, 5}
ε-closure(5) = {}
ε-closure(6) = {}
ε-closure(7) = {}

NFA to DFA Conversion

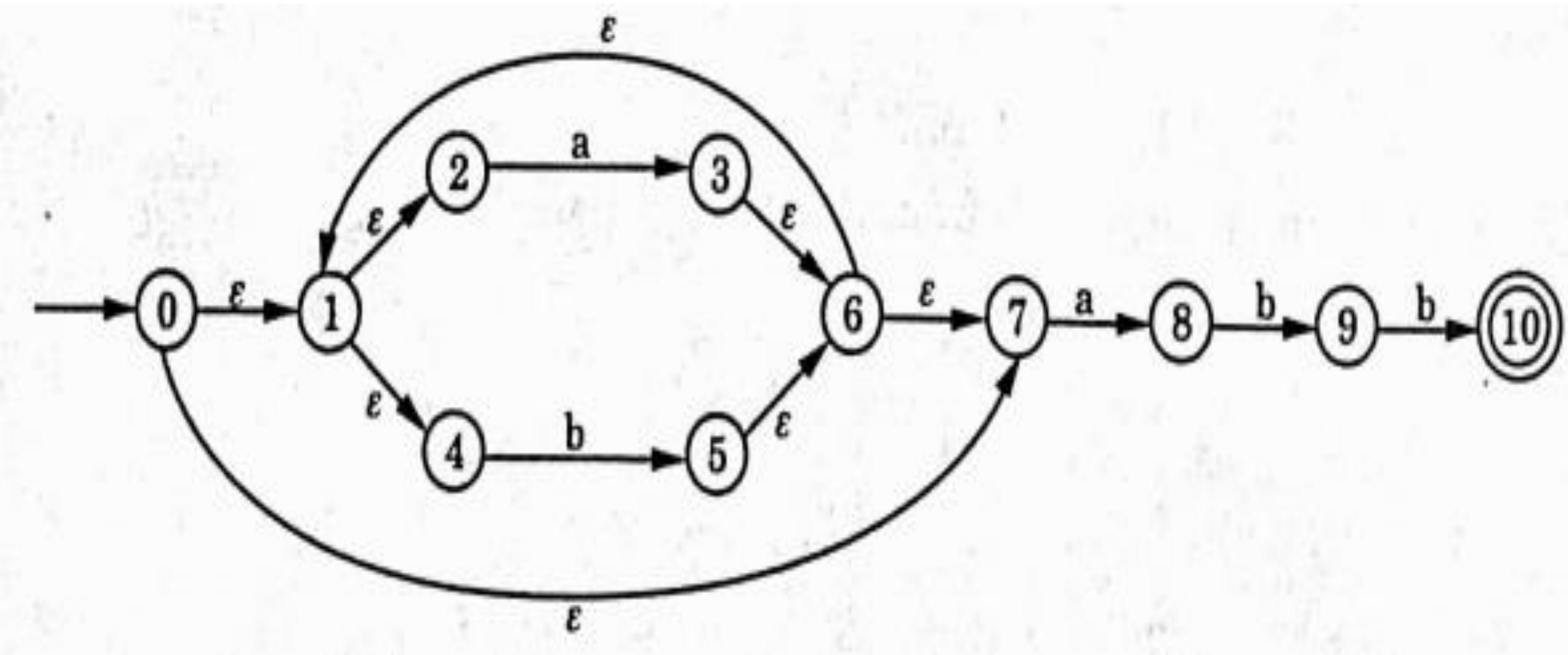
Algorithm NFA - to-DFA :

Input : NFA with set of states $N = \{ n_0, n_1, \ldots n_n\}$, with start state n_0

Output : DFA, with set of states $D' = \{ d_0, d_1, d_2 \ldots d_n\}$, with start state d_0

1. $d_0 = \varepsilon$-closure (n_0)
2. $D' = \{d_0\}$
3. **set** d_0 unmarked
4. **while** there is an unmarked state d in D'
5. { **set** d marked
6. { **For** each input symbol 'a'
7. {Let T be set of states in NFA to which there is a transition on 'a' from some state n_i in d
8. $d' = \varepsilon$-closure (T).
9. **If** d' is not already present in D'
10. { $D' = D' \cup \{d'\}$
11. Add transition $d \rightarrow d'$, labeled 'a'
12. **set** d' unmarked
13. }}}}

Example: Draw NFA for RE (a+b)*abb.
Convert NFA to DFA



Now , we can apply algorithm to convert it into DFA

$A = \varepsilon\text{-closure}(0)$ [step 1 of Algo]

$= \{0, 1, 2, 4, 7\}$

$\therefore \quad D' = \{A\}$ [step 2]

Apply steps (4-12) of Algo on state A

For state A

The transitions of symbols a, b from state A

$$\begin{array}{lccccccc} A = & \{ & 0, & 1, & 2, & 4, & 7 & \} \\ & & \downarrow & \downarrow & a\downarrow & b\downarrow & a\downarrow & \\ T = & \{ & -, & -, & 3, & 5, & 8 & \} \end{array}$$

[step 6,7]

$\therefore$ **For Input Symbol a ,**

$T_a = \{3,8\}$

$\therefore \quad B = \varepsilon\text{-closure}(T_a)$

$= \varepsilon\text{-closure}(\{3,8\})$

$= \varepsilon\text{-closure}(3) \cup \varepsilon\text{-closure}(8)$

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

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

For Input Symbol b, [step 6,7]

$T_b = \{5\}$

$\therefore \quad C = \varepsilon\text{-closure}(T_b)$ [step 8]

$= \varepsilon\text{-closure}(5)$

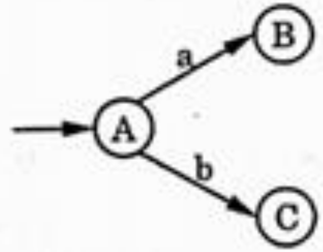
$= \{5, 6, 7, 1,2, 4\}$

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

$\therefore \quad D' = \{A\} \cup \{B,C\}$ [step 9, 10]

$\therefore \quad D' = \{A,B,C\}$

∴ Add transformation from A to B and A to C



State

For ~~step~~ B

∴ $B = \{1, 2, 3, 4, 6, 7, 8\}$

Transitions on symbols a, b from B are :

∴ B =	{	1,	2,	3,	4,	6,	7,	8 }
		↓	a ↓	↓	b ↓	↓	a ↓	b ↓
T =	{	–,	3,	–,	5,	–,	8,	9 }

For input symbol a

∴ $T_a = \{3,8\}$

∴ $\varepsilon\text{-closure}(T_a) = \varepsilon\text{-closure}\ \{3,8\}$

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

$= B$

For input symbol b

$T_b = \{5, 9\}$

$\varepsilon\text{-closure}\ (T_b)$

$= \varepsilon\text{-closure}\ (\{5,9\})$

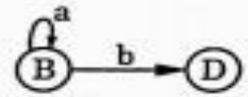
$= \varepsilon\text{-closure}\ (5) \cup \varepsilon\text{-closure}\ (9)$

$= \{5, 6, 7, 1, 2, 4\} \cup \{9\}$

$= \{1,2,4,5,6,7,9\} = D$

∴ $D' = \{A,B,C\} \cup \{D\}. = \{A,B,C,D\}$

$\therefore$ **Add Transitions from B to B and from B to D**



For state C

Since,

$$\begin{array}{lccccccc}
C = \{ & 1, & 2, & 4, & 5, & 6, & 7 & \} \\
 & \downarrow & a\downarrow & b\downarrow & \downarrow & \downarrow & a\downarrow & \\
T = \{ & -, & 3, & 5, & -, & -, & 8 & \}
\end{array}$$

For input symbol a

$\therefore \quad T_a = \{3,8\}$

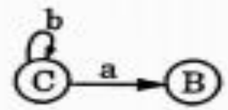
$\therefore \quad \varepsilon\text{-closure } \{3,8\} = B$

For input symbol b

$T_b = \{5\}$

$\therefore \quad \varepsilon\text{-closure } (5) = C$

Add Transition from C to B and C to C



For state D

$$\begin{array}{lcccccccc}
\therefore \quad D = \{ & 1, & 2, & 4, & 5, & 6, & 7, & 9 & \} \\
 & \downarrow & a\downarrow & b\downarrow & \downarrow & \downarrow & a\downarrow & b\downarrow & \\
T = \{ & -, & 3, & 5, & -, & -, & 8, & 10 & \}
\end{array}$$

For input symbol a

$\therefore \quad T_a = \{3,8\}$

$\therefore \quad \varepsilon\text{-closure } (T_a) = B$

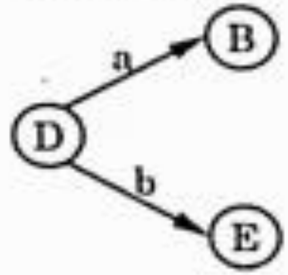
For input symbol b

$T_b = \{5,10\}$

$$\begin{aligned}
\therefore \quad \varepsilon\text{-closure } (\{5,10\}) &= \varepsilon\text{-closure } (5) \cup \varepsilon\text{-closure } (10) \\
&= \{5, 6, 7, 1, 2, 4,\} \cup \{10\} \\
&= \{1, 2, 4, 5, 6, 7, 10\} = E
\end{aligned}$$

$\therefore \quad D' = \{A,B,C,D\} \cup \{E\} = \{A,B,C,D,E\}$

Add transition from D to B and D to E



For state E

∴ $E =$	{	1,	2,	4,	5,	6,	7,	10 }
		↓	a ↓	b ↓	↓	↓	a ↓	↓
$T =$	{	–,	3,	5,	–,	–,	8,	– }

For input symbol a

∴ $T_a = \{3,8\}$

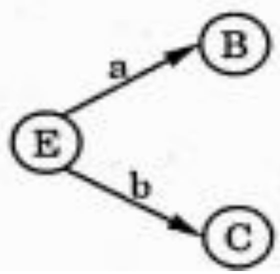
∴ ε-closure $(\{3,8\}) = B$

For input symbol b

$T_b = \{5\}$

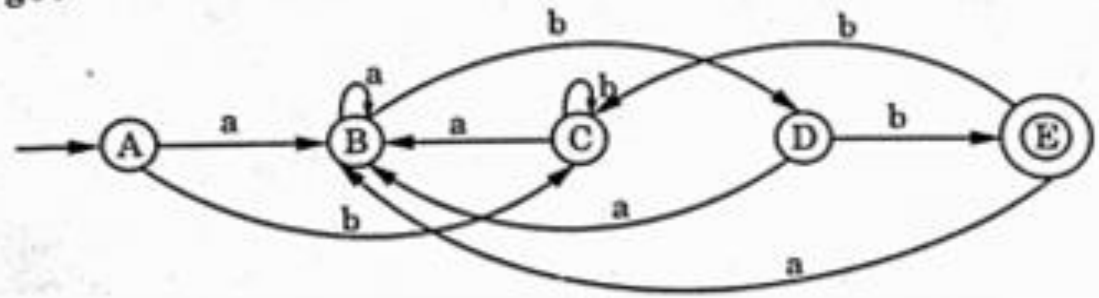
ε-closure $(5) = C$

Add Transition from E to B and E to C



$\therefore$
$A = \{0, 1, 2, 4, 7\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$
$C = \{1, 2, 4, 5, 6, 7\}$
$D = \{1, 2, 4, 5, 6, 7, 9\}$
$E = \{1, 2, 4, 5, 6, 7, 10\}$

Therefore states of DFA will be $D' = \{A, B, C, D, E\}$. Joining all transitions Diagrams., we get



		a	b
(Initial state)	A	B	C
	B	B	D
	C	B	C
	D	B	E
(Final state)	E	B	C

$\because$ E contains state 10, which is final state in NFA

$\therefore$ E, itself will be final state in DFA.

Minimizing number of states of DFA

- Minimizing means reducing the number of states in DFA.
- The states should be eliminated in such a way that resulting DFA should not effect the language accepted by DFA.

Algorithm : Minimization of DFA

Input : DFA D1 with set of states Q with set of final states F.

Output : DFA D2 which accepts same language as D1 and having minimum no. of states as possible.

Method :

(1) Make a partition 'π' of set of states with two subsets :

(*a*) Final state 'F'

(*b*) Non Final states 'Q–F'

$\therefore$ $\pi = \{ F, Q - F\}$

(2) Apply following procedure to make π_{new} from π.

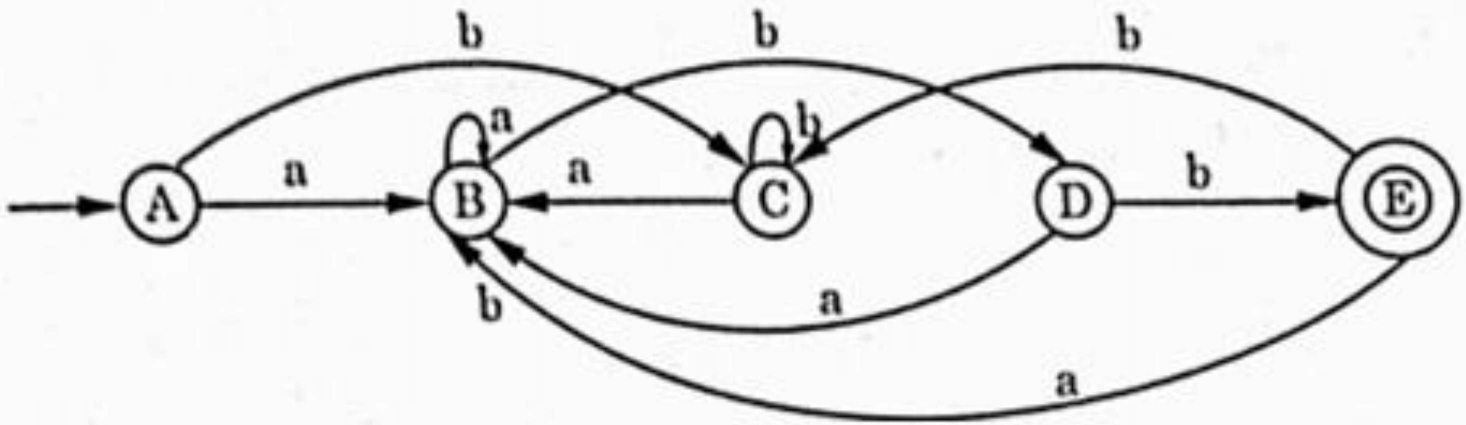
For each set S of π.

Partition S into Subsets such that two states p & q of S are in same subset of S iff for each input symbol 'a' states p & q have transitions to states in same set of π. Replace S in π_{new} by set of subsets formed.

(3) If $\pi_{new} = \pi$, Let $\pi_{final} = \pi$ & continue with step 4. Else repeat step 2 for $\pi = \pi_{new}$.

(4) Choose one state from each set of π_{final} as representative of that set. These states will be states of Minimized DFA D2.

Convert following DFA to Minimized DFA.



Make a Transition Table

		a	b
(Initial state)	A	B	C
	B	B	D
	C	B	C
	D	B	E
(Final state)	E	B	C

Make a partition 'π' of set of states *ie*, $\pi = \{F, Q - F\}$

$\therefore \quad \pi_0 = \{\{E\}, \{A, B, C, D\}\}$

3 (*a*) For input symbol a, on {A, B, C, D} of π_0

$$\left.\begin{aligned} A &\xrightarrow{a} B \\ B &\xrightarrow{a} B \\ C &\xrightarrow{a} B \\ D &\xrightarrow{a} B \end{aligned}\right\} \text{All B's lie in same set } \{A, B, C, D\} \text{ of } \pi_0$$

(*b*) For input symbol b on {A, B, C, D} of π_0

$$\begin{aligned} A &\xrightarrow{b} C \\ B &\xrightarrow{b} D \\ C &\xrightarrow{b} C \end{aligned} \quad \text{Lie in same set } \{A, B, C, D\} \text{ of } \pi_0$$

$$D \xrightarrow{b} E \quad \text{Lie in other set } \{E\} \text{ of } \pi_0$$

$\therefore$ {A, B, C, D} of π_0 will be split into {A, B, C} and {D}

$\therefore$ $\pi_1 = \{\{E\}, \{A, B, C\}, \{D\}\}$

4. (*a*) For input a, on {A, B, C} of π_1

$$\left.\begin{array}{l} A \xrightarrow{a} B \\ B \xrightarrow{a} B \\ C \xrightarrow{a} B \end{array}\right\}$$ All B's lie in same set of π_1

(*b*) For input *b* on {A, B, C} of π_1

$A \xrightarrow{b} C$

$B \xrightarrow{b} D$

$C \xrightarrow{b} C$

Lie in same set {A, B, C} of π_1

Lie in other set {D} of π_1

$\therefore$ {A, B, C} in π_1 will be split into {A, C} and {B}

$\therefore$ $\pi_2 = \{\{E\}, \{A, C\}, \{B\}, \{D\}\}$

5. Check, if {A, C} can be splited further

(*a*) For input *a*, on {A, C} of π_2

$$\left.\begin{array}{l} A \xrightarrow{a} B \\ C \xrightarrow{a} B \end{array}\right\} \text{lie in same set of } \pi_2$$

(*b*) For input *b*, on {A, C} of π_2

$$\left.\begin{array}{l} A \xrightarrow{b} C \\ C \xrightarrow{b} C \end{array}\right\} \text{Lie in same set of } \pi_2$$

$\therefore$ {A, C} will not be splitted

$\therefore$ $\pi_3 = \{\{E\}, \{A, C\}, \{B\} \{D\}\}$

$\therefore$ $\pi_3 = \pi_2 = \pi_{final} = \{\{E\}, \{A, C\}, \{B\}, \{D\}\}$

$\therefore$ There will be **4 states of Minimized DFA** corresponding to **5 states of given-DFA.**

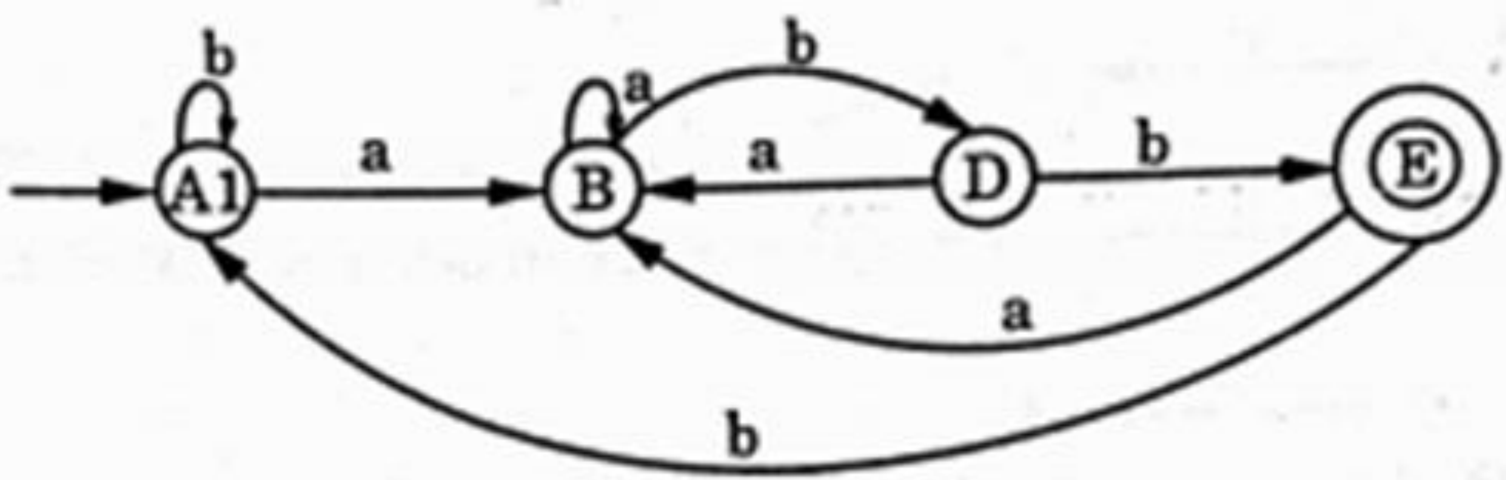
- { A,C} can be renamed as A1
- B
- D
- E

∴ 4 States of Minimized DFA *i.e.* A1, B, D,E will be joined by seeing the transitions from the given DFA Table .

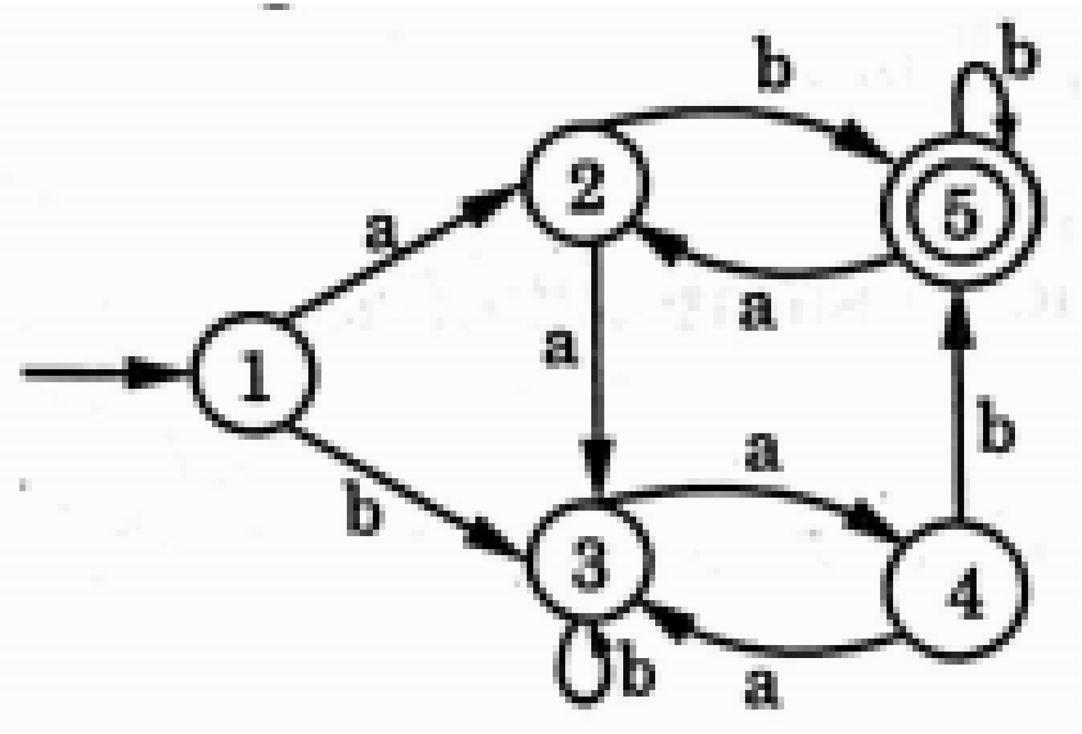
∴ Minimized or Reduced Automata will be

		a	b
(Initial state)	A1	B	A1
	B	B	D
	D	B	E
(Final state)	E	B	A1

where A1 = {A,C}



Example: Minimize the following DFA



1. Make a Transition Table.

		a	b
(Initial state)	1	2	3
	2	3	5
	3	4	3
	4	3	5
(Final state)	5	2	5

2. $\pi_0 = \{\{5\}, \{1,2,3,4\}\}$

3. (*a*) For input a, on {1, 2,3, 4} of π_0

$$\left.\begin{array}{l} 1 \xrightarrow{a} 2 \\ 2 \xrightarrow{a} 3 \\ 3 \xrightarrow{a} 4 \\ 4 \xrightarrow{a} 3 \end{array}\right\} \text{All States lie in same set } \{1, 2, 3,4\} \text{ of } \pi_0$$

(*b*) For input b, on {1,2,3,4} of π_0.

$1 \xrightarrow{b} 3$

$2 \xrightarrow{b} 5$

$3 \xrightarrow{b} 3$

$4 \xrightarrow{b} 5$

Lie in same set {1, 2, 3,4} of π_0

Lie in same set {5} of π_0

$\therefore$ {1, 2, 3, 4} will be split into {1, 3} and {2,4}

$\therefore$ $\pi_1 = \{\{5\}, \{1, 3\}, \{2, 4\}\}$

4. (a) For input symbol a on {1,3} of π_1

$$\left.\begin{array}{l} 1 \xrightarrow{a} 2 \\ 3 \xrightarrow{a} 4 \end{array}\right\} \quad \text{Lie in same set } \{2,\ 4\} \text{ of } \pi_1$$

Similarly for input symbol a on {2,4} of π_1

$$\left.\begin{array}{l} 2 \xrightarrow{a} 3 \\ 4 \xrightarrow{a} 3 \end{array}\right\} \quad \text{Lie in same set } \{1,\ 3\} \text{ of } \pi_1$$

(b) For input symbol b on {1,3} of π_1

$$\left.\begin{array}{l} 1 \xrightarrow{b} 3 \\ 3 \xrightarrow{b} 3 \end{array}\right\} \quad \text{Lie in same set} \{1,\ 3\} \text{ of } \pi_1$$

Similarly for input symbol b on {2,4} of π_1

$$\left.\begin{array}{l} 2 \xrightarrow{b} 5 \\ 4 \xrightarrow{b} 5 \end{array}\right\} \quad \text{Lie in same set } \{5\} \text{ of } \pi_1$$

$\therefore$ subset in π_1 *ie* {1,3} & {2,4} will not be splitted.

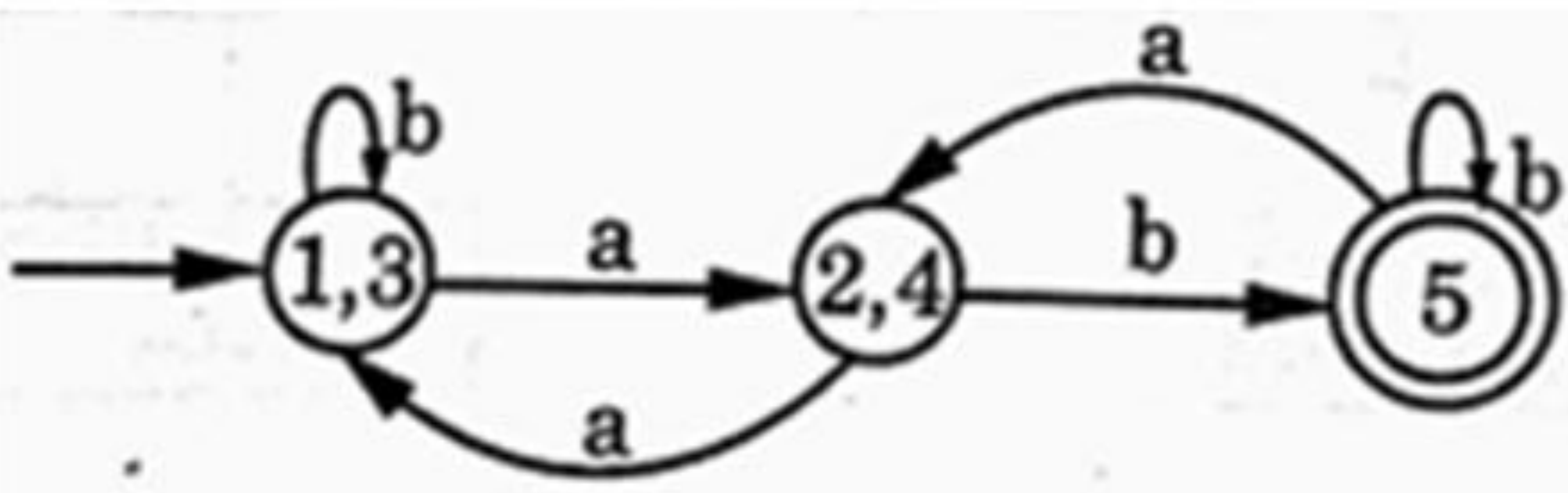
$\therefore$ $\pi_{\text{final}} = \{\{5\},\ \{1,\ 3\},\ \{2,\ 4\}\}$

$\therefore$ There will be 3 states of DFA

ie, {5} , { 1, 3} and {2, 4}

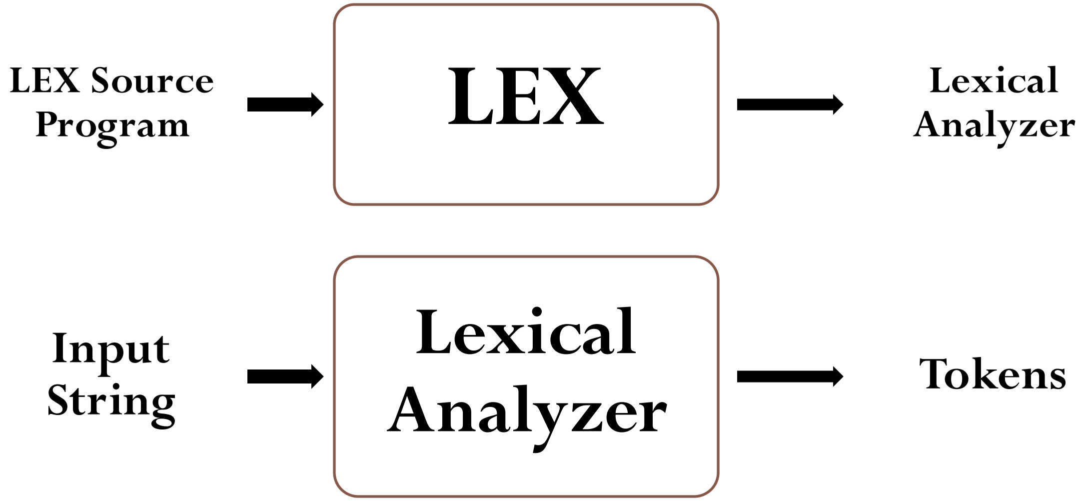
		a	b
(initial state)	13	24	13
	24	13	5
(Final state)	5	24	5

Minimized DFA will be.



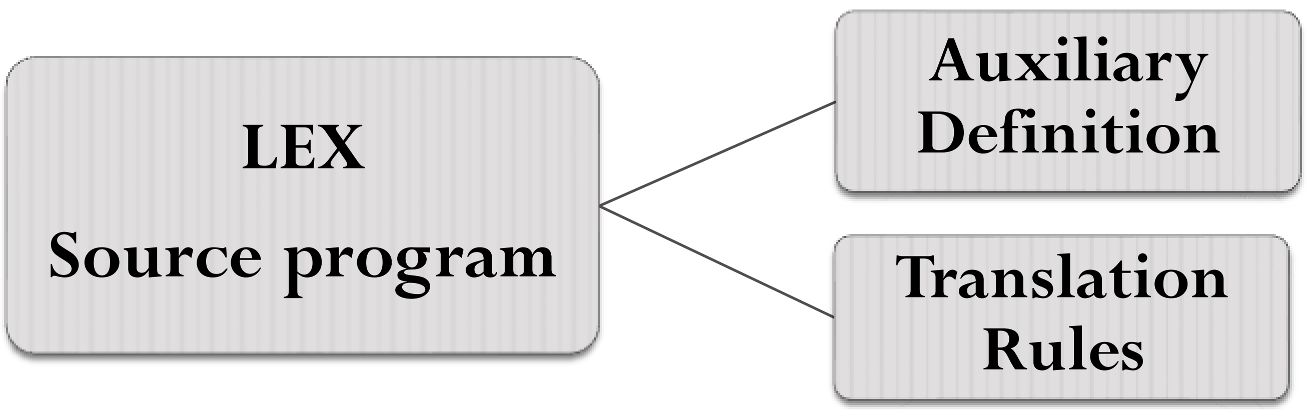
Language for Lexical Analyzers

- LEX is a source program used for the specification of lexical analyzer.
  - It is a tool or software which automatically generates Lexical Analyzer (Finite Automata).
  - It takes as input a LEX source program and produces Lexical Analyzer as its output.
  - Then Lexical Analyzer will convert the input string entered by user into tokens as its output.



LEX Source Program

- Language for specifying or representing Lexical Analyzer.
- Components of LEX source program:
  - Auxiliary Definitions
  - Translation Rules



Auxiliary Definitions

- It denotes the Regular Expressions of the form:

$$\text{Distinct Names}\left\{\begin{array}{lcl} D_1 & = & R_1 \\ D_2 & = & R_2 \\ & \vdots & \\ & \vdots & \\ D_n & = & R_n \end{array}\right\}\text{Regular Expressions}$$

Where,

- Distinct name (D_i) -> shortcut name of Regular Expression
- Regular Expression (R_i) -> Notation to represent collection of input symbols.

Auxiliary Definition for Identifiers

$$\begin{array}{l} D_1 \\ D_2 \\ D_3 \end{array} \left\{ \begin{array}{r l} \text{Letter} & = A \mid B \mid \ldots\ldots\ldots \mid Z \\ \text{digit} & = 0 \mid 1 \mid 2 \mid \ldots\ldots\ldots \mid 9 \\ \text{identifier} & = \text{letter } (\text{letter} \mid \text{digit})^* \end{array} \right\} \begin{array}{l} R_1 \\ R_2 \\ R_3 \end{array}$$

Auxiliary Definition for signed Numbers :

integer = digit digit*

sign = + | –

signedinteger= sign integer

Auxiliary Definition for Decimal Numbers :

decimal = signedinteger . integer | sign. integer

Auxiliary Definition for Exponential Numbers :

Exponential – No= (decimal | signedinteger) E signedinteger

Auxiliary Definition for Real Numbers :

Real-No.= decimal | Exponential – No

Translation Rules

- It is a set of rules or actions which tells Lexical Analyzer what it has to do.

  or
- what it has to return to parser on encountering token.
- It consists of statements of the form:

  $P_1 \{Action_1\}$

  $P_2 \{Action_2\}$

  :

  :

  $P_n \{Action_n\}$

  Where,

  - P_i -> pattern or Regular Expression consisting of input alphabets & Auxiliary definition names.
  - $Action_i$ -> it is a piece of code which gets executed whenever token is recognised.

Example

Translation Rules for "Keywords"

Patterns or Regular Expressions		Actions
begin		{return 1}
end		{return 2}
if		{return 3}
then		{return 4}
else		{return 5}

Translation Rules for "Identifiers"

letter (letter +digit)* {Install( ); return 6}

- If Lexical analyzer recognizes an "identifier", the action taken by the Lexical Analyzer is
  - to install or store the name in symbol table
  - return value 6 as integer code to the parser.

Implementation of Lexical Analyzer

- LEX generates Lexical Analyzer as its output by taking LEX program as its input.
- LEX program is a collection of patterns (Regular expressions) & their corresponding actions.
- Patterns represent the tokens to be recognized by lexical analyzer to be generated.
- For each pattern, a corresponding NFA will be designed.
- There can be n number of patterns.
- A start state is taken and using ε-transition, all these NFAs can be connected together to make combined NFA.
- The final state of each NFA show that it has found its own token P_i.
- Convert the NFA to DFA.
- The final state shows which token we have found.
  - If states in DFA does not include any final state of NFA, there will be error condition.

Example

Convert the following LEX program into Lexical Analyzer.

AUXILARY DEFINITIONS

—

—

—

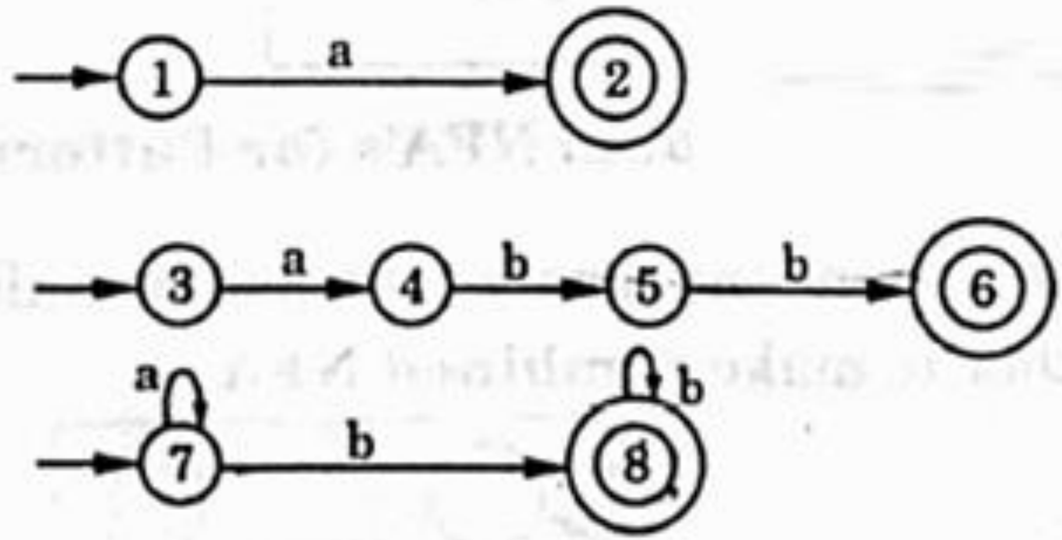
TRANSLATION RULES

a { }

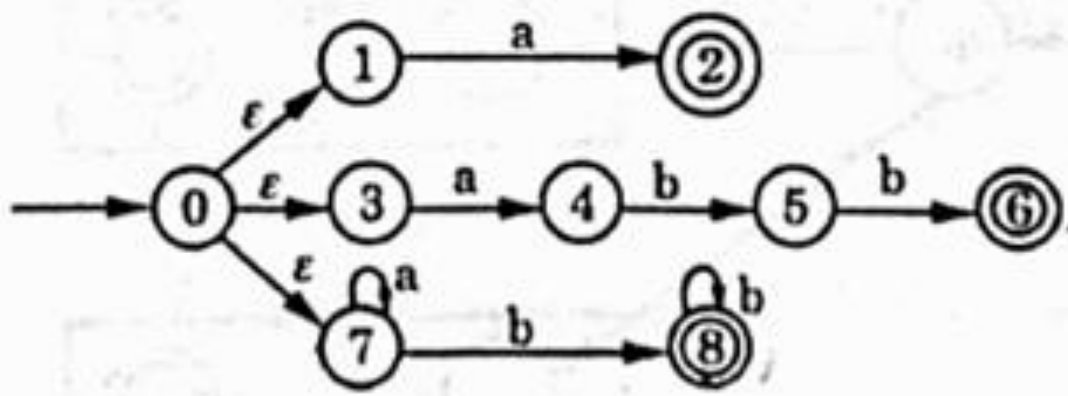
abb { }

$a^* b^+$ { }

1. Convert the patterns into NFA's



2. Make a Combined NFA



3. **Convert NFA to DFA**

$A = \varepsilon\text{-closure}(0) = \{0, 1, 3, 7\}$

The transition on symbols a, b from state A

$\boxed{\text{For State A}}$

$$
\begin{array}{lcccccccl}
T_a & = & \{ & -, & 2, & 4, & 7 & \} & = \{2, 4, 7\} \\
 & & & a\uparrow & a\uparrow & a\uparrow & a\uparrow & & \\
A & = & \{ & 0, & 1, & 3, & 7 & \} & \\
 & & & b\downarrow & b\downarrow & b\downarrow & b\downarrow & & \\
T_b & = & \{ & -, & -, & -, & 8 & \} & = \{8\}
\end{array}
$$

$\therefore\ \varepsilon\text{-closure}(T_a)$

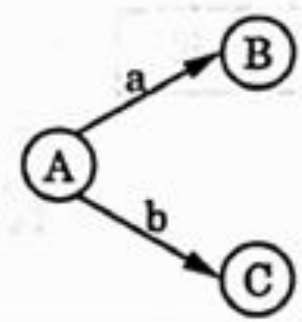
$= \varepsilon\text{-closure}(\{2, 4, 7\})$

$= \{2, 4, 7\} = B$

$\therefore\ \varepsilon\text{-closure}(T_b)$

$= \varepsilon\text{-closure}(\{8\})$

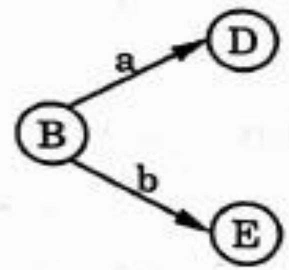
$= \{8\} = C$



For State B

$$
\begin{array}{lcccccl}
T_a & = \{ & -, & -, & 7 & \} & = \{7\} \\
 & & a\uparrow & a\uparrow & a\uparrow & & \\
B & = \{ & 2, & 4, & 7 & \} & \\
 & & b\downarrow & b\downarrow & b\downarrow & & \\
T_b & = \{ & -, & 5, & 8 & \} & = \{5, 8\}
\end{array}
$$

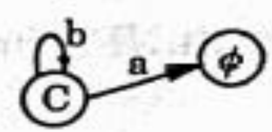
$\therefore$ ε-closure $(7) = \{7\} = D$ | ε-closure $(\{5, 8\})$ $= \{5, 8\} = E$



For State C

$$
\begin{array}{ll}
T_a = & \{-\} = \phi \\
 & a \uparrow \\
C = & \{8\} \\
 & b \downarrow \\
T_b = & \{8\}
\end{array}
$$

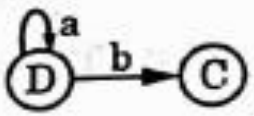
$\therefore$ ε-closure $(\phi) = \phi$ | ε-closure $(8) = \{8\} = C$



For State D

$$T_a = \{ 7 \} = \phi$$
$$a \uparrow$$
$$D = \{ 7 \}$$
$$b \downarrow$$
$$T_b = \{ 8 \}$$

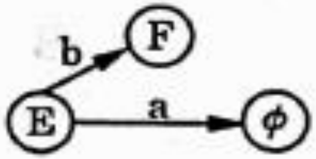
$\therefore$ ε-closure (7) = { 7 } =D | ε-closure (8) = { 8 } = C



For State E

$$\begin{array}{llll} T_a = & \{ \ -, & - \ \} & = \phi \\ & \ \ a\uparrow & a\uparrow & \\ E = & \{ \ 5, & 8 \ \} & \\ & \ \ b\downarrow & b\downarrow & \\ T_b = & \{ \ 6, & 8 \ \} & = \{6, 8\} \end{array}$$

$\therefore$ ε-closure (ϕ) = ϕ | ε-closure {(6,8)} = { 6,8 } = F



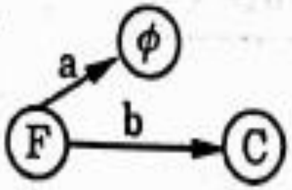
For State F

$$T_a = \{\ -,\ -\ \} = \phi$$

$$\begin{array}{ll} & a\uparrow \quad a\uparrow \\ F = \{ & 6, \quad 8\} \\ & b\downarrow \quad b\downarrow \end{array}$$

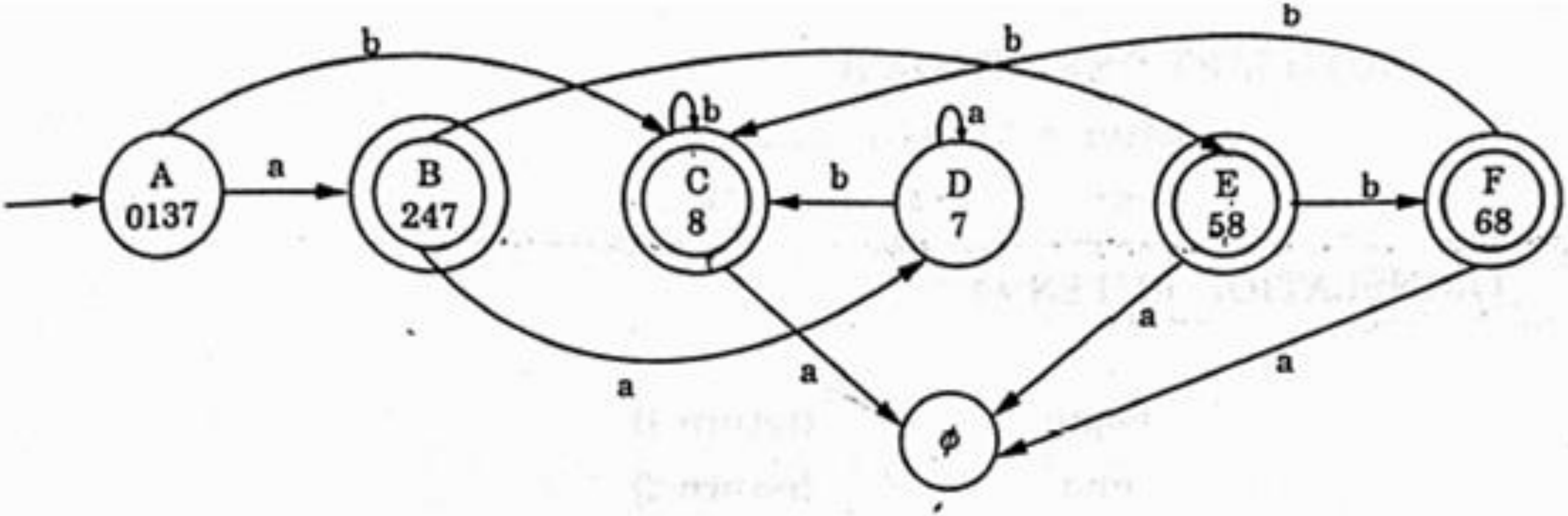
$$T_b = \{\ -,\ 8\} = \{8\}$$

$\therefore$ ε-closure $(\phi) = \phi$ | ε-closure $(8) = \{8\} = C$



$\therefore$ Combining all transition Diagrams, we get complete DFA. Since state 2, 6, 8 are final states in NFA.

$\therefore$ States in NFA having there states *i.e.* 247, 8, 58, 68 are final states



State	a	b	Tokens Recognize
0137	247	8	none
247	7	58	a
8	ϕ	8	$a^* b^+$
7	7	8	none
58	ϕ	68	$a^* b^+$
68	ϕ	8	abb
ϕ	ϕ	ϕ	none

Tokens Recognized :

- 0137 → No state in {0,1,3,7} is Final state. Therefore , no token will be Recognized by this state.
- 247 → State 2 in these states is final state ∵ state 2 accepts a in combined NFA. Therefore, 247 will accept a.
- 8 → ∵ 8 is Final State in combined NFA. It accepts a^*b^+ in combined NFA.
- 7 → ∵ 7 is not final state & therefore it accepts nothing.
- 58 → ∵ 8 is Final state but 5 is non-Final state. State 8 accepts $a^* b^+$ in combined NFA. Therefore 58 will accept a^*b^+
- 68 → Both states 6 & 8 are final states. But 6 accepts abb and 8 accepts $a^* b^+$ in combined NFA. But abb comes before $a^* b^+$ in Translation rules given in Question. Therefore state 68 will accept token abb.

END OF UNIT-I