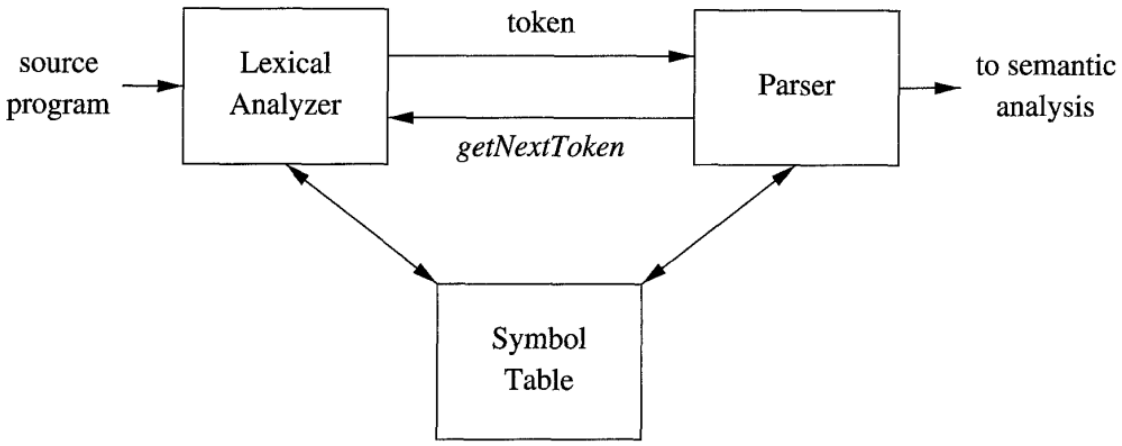
UNIT -II

FINITE AUTOMAT AND LEXICAL ANALYSIS

Role of Lexical Analyzer:

- ✓ Lexical Analyzer is the first part of the Compiler.
- ✓ The main task is to read the input characters of the source program, group them into lexemes.
- ✓ Produce as output a sequence of tokens for each lexeme in the source program
- ✓ These stream of tokens is sent to the parser for syntax analysis.
- ✓ The LA also eliminates the comments, whitespace (blank, newline, tab …), and unessential symbols.

Figure below shows the interaction between the Lexical Analyzer and the Parser. the interaction is implemented by having the parser call the lexical analyzer.



The call, suggested by 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.

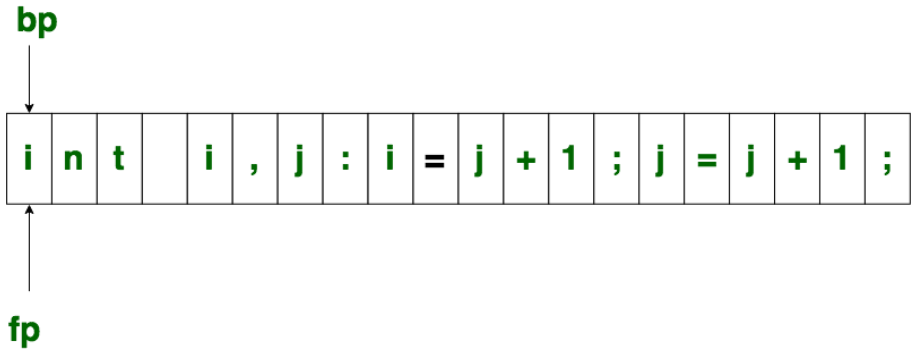
Input Buffering:

- ✓ Input buffering is a technique, where the compiler reads input in blocks (chunks) into a buffer instead of character by character from secondary storage.
- ✓ The lexical analyzer then processes characters from this buffer.

- ✓ The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block.
- ✓ The size of the buffer can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled.
- ✓ This technique reduces the **Overhead** required for process an input character.

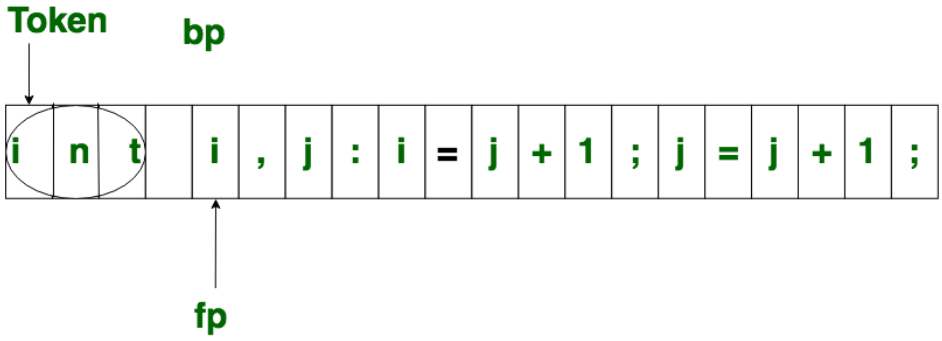
To achieve this, it uses **two pointers:**

- **Begin pointer (bp):** Marks the beginning of the current lexeme.
- **Forward pointer (fp):** Moves ahead to detect the end of the lexeme.



Initial Configuration

Initially both ***bp*** and ***fp*** initially point to *i*. The ***fp*** advances until it encounters a **whitespace**, indicating the **end of the lexeme "int"**. Then, both pointers move forward to the start of the next token as shown in the below figure.



Input buffering

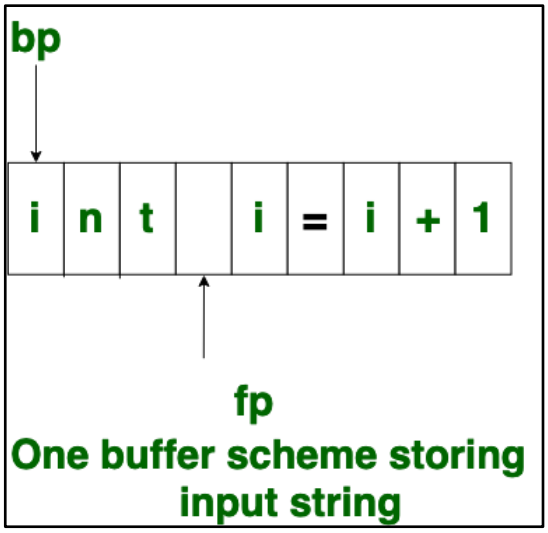
Methods of Input Buffering:

There are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme:

1. One-Buffer Scheme

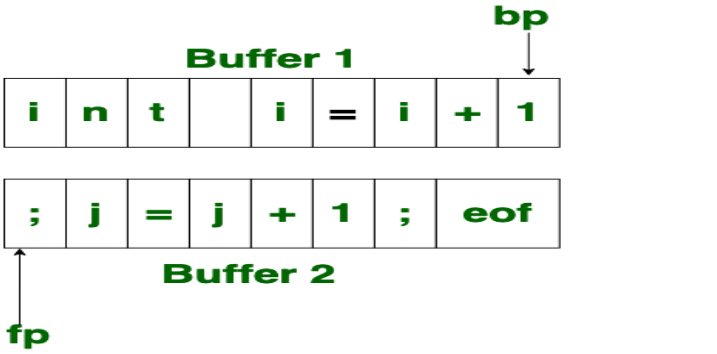
- In this scheme, a single buffer is used to store a block of input.
- The fp scans characters within this buffer until the end of the buffer is reached.

- **Problem:** If a lexeme is longer than the buffer size, part of the lexeme may cross the buffer boundary. Refilling the buffer overwrites the beginning of the lexeme, causing errors.



Two-Buffer Scheme

To overcome the limitations of the one-buffer scheme, the two-buffer scheme is used.



Two buffer scheme storing input string

- Two buffers of equal size are maintained.
- They are filled alternately: when the fp reaches the end of one buffer, the other buffer is refilled with the next block of input.
- At the end of each buffer, a special sentinel character (EOF marker) is placed to indicate buffer boundaries.
- Initially, both bp and fp point to the beginning of the first buffer. The fp moves ahead until a whitespace (end of lexeme) or sentinel is encountered.
- When the sentinel is reached, the analyzer switches to the other buffer. This process continues until the source program is completely scanned.
- Even in this scheme, if a lexeme is longer than the buffer size, it still cannot be scanned completely.

Sentinel Usage: *The sentinel character (eof) at the end of each buffer helps the lexical analyzer detect when to switch buffers without performing repeated boundary checks. This makes scanning more efficient.*

Advantages:

- **Reduced system calls:** Reading in large blocks lowers I/O overhead.
- **Improved performance:** Faster scanning compared to character-by-character reading.
- **Simpler compiler design:** Reduces the complexity of input handling logic.

Disadvantages:

- **Memory overhead:** Larger buffers consume more memory.

- **Management complexity:** Incorrect handling of buffer refills may lead to errors.
- **Lexeme boundary issue:** If a lexeme exceeds the buffer size, scanning may fail.

Lookahead code with sentinels (eof):

```
switch ( *forward++ ) {
        case eof:
            if (forward is at end of first buffer ) {
                reload second buffer;
                forward = beginning of second buffer;
            }
            else if (forward is at end of second buffer ) {
                reload first buffer;
                forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                terminate lexical analysis;
            break;
        Cases for the other characters
}
```

Tokens, Patterns, and Lexemes:

In Lexical Analyzer phase we use **three related but distinct terms**:

- **Token:** A token is a pair consisting of a token name and an optional attribute value.

  <token_name, attribute_value>

  The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.

- **Pattern:** It is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

- **Lexeme**: It is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example: Consider the C statement, find the number of tokens in it

printf ("Total = %d\n", score) ;

- The total number of tokens are: 7

Step: Break into tokens

Printf	Identifier (function name)	**,**	Comma (separator)
(	Left parenthesis (separator)	**score**	Identifier (variable)
"Total = %d\n"	String literal	**)**	Right parenthesis
;	Semicolon (terminator)		

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

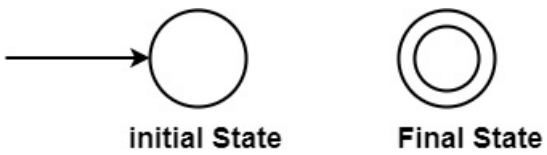
Approaches to Design Lexical Analyzer:

Lexical Analysis can be designed using Transition Diagrams.

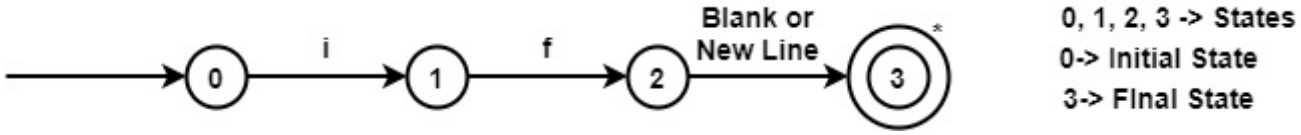
Finite Automata (Transition Diagram) – A Directed Graph or flowchart used to recognize token.

The **transition Diagram** has two parts:

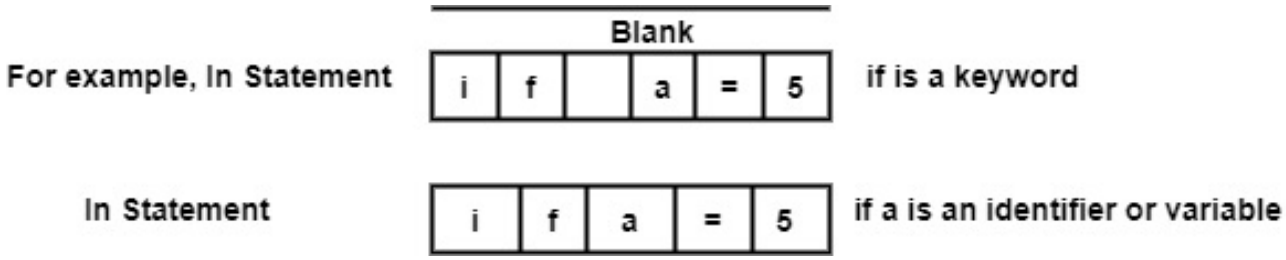
- **States** - It is represented as **circles.**
- **Edges –** States are connected by Edges Arrows.



Example: Draw Transition Diagram for "if" keyword.



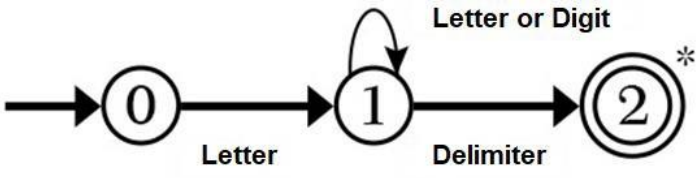
To recognize Token ("**if**"), Lexical Analysis has to read also the next character after "f". Depending upon the next character, it will judge whether the "**if**" keyword or something else is.



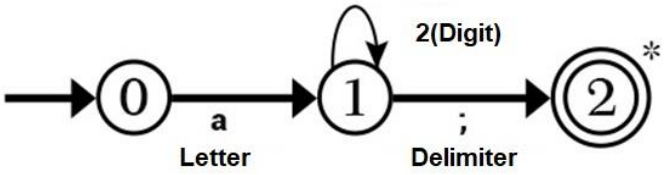
So, Blank space after "if" determines that "**If**" is a keyword.

"*" on Final **State 3** means **Retract**, i.e., **control will again come to previous state 2.** Blank space is not a part of the Token ("if").

Transition Diagram for an Identifier – An identifier starts with a letter followed by letters or Digits. Transition Diagram will be:



Example: In statement **int a2;** Transition Diagram for identifier **a2** will be:



As (;) is not part of Identifier ("**a2**"), so use "*" for **Retract** i.e., coming back to **state 1** to recognize identifier ("**a2**").

The **Transition Diagram for identifier can be converted to Program Code as**:

Coding:

```
State 0: C = Getchar();
        If letter (C) then goto state 1 else fail

State1: C = Getchar();
        If letter (C) or Digit (C) then goto state 1
        else if Delimiter (C) goto state 2
        else Fail

State2: Retract ();
        return (6, Install ());
```

In-state 2, Retract () will take the **pointer one state back**, i.e., **to state 1** & declares that whatever has been **found till state 1 is a token**.

The **lexical Analysis** will **return** the **token to the Parser**, not in the form of an English word but **the form of a pair, i.e., (Integer code, value)**.in

In the case of **identifier**, the **integer code** returned to the parser **is 6** as shown in the table.

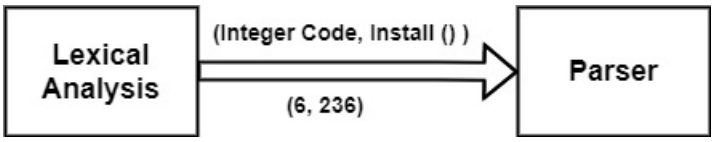
Install () – It will return a pointer to the symbol table, i.e., address of tokens.

The following table shows the integer code and value of various tokens returned by lexical analysis to the parser.

Token	**Integer code**	**Value**
Begin	1	-
End	2	-
If	3	-
Then	4	-
Else	5	-
Identifier	6	Pointer to Symbol Table
Constants	7	Pointer to Symbol Table
<	8	1
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

These integer values are not fixed. Different Programmers can choose other integer codes and values while designing the Lexical Analysis.

Suppose, if the **identifier** is stored at location 236 in the symbol table, then

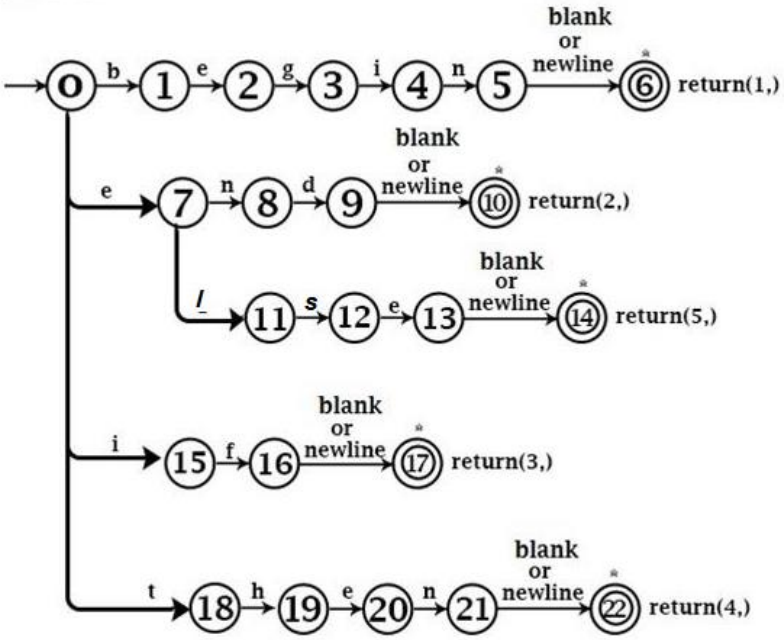


Similarly, if **constant** is stored at location 238 then

Integer code = 7

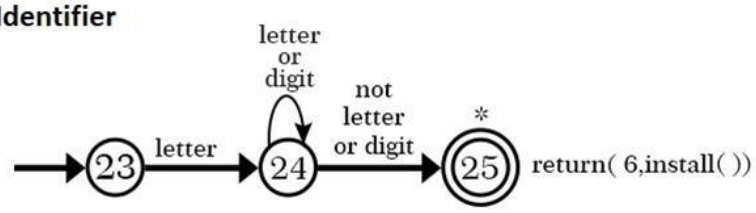
Install () = 238 i.e., Pair will be (7, 238)

Keywords



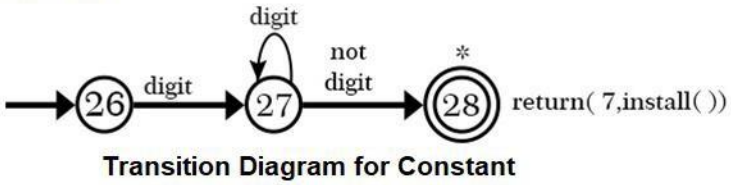
Transition Diagram for Keywords

Identifier



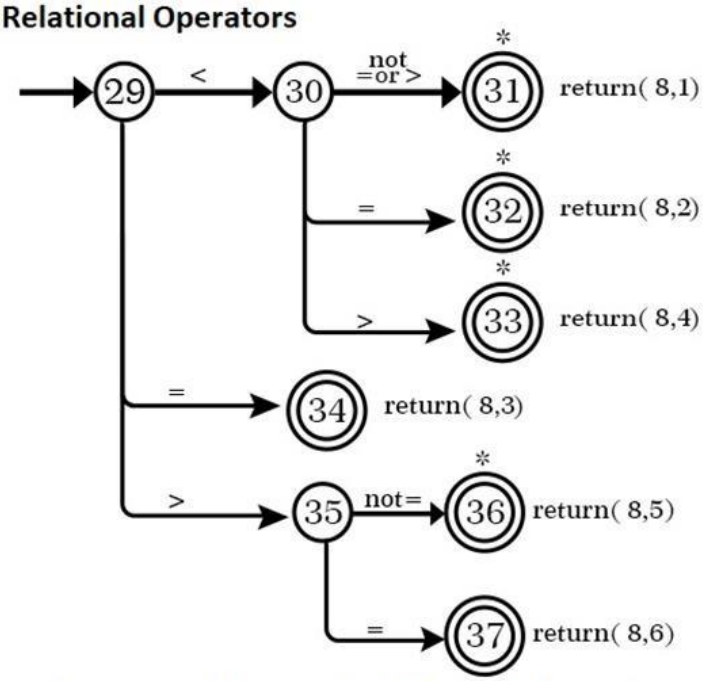
Transition Diagram for Identifier

Constant



Transition Diagram for Constant

Relational Operators



Transition Diagram for Relational Operators

Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in below Figure

OPERATION	DEFINITION AND NOTATION
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	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

- Union is the familiar operation on sets.
- The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them.

- The (Kleene) closure of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of L zero times," is defined to be $\{\varepsilon\}$.
- The positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^o. That is, will not {ε} be in L+ unless it is in L itself.

Example: Let L be a language which contains the set of letters

L = {A, B,... ,Z, **a,b,**...,***z* **}** **and** D is a set of all digits i.e **D = {0,1,..**.9**}**

Some other languages that can be constructed from languages L and D, using the operators in the above figure:

1. *L U D - is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit. followed by one digit.*
2. *LD(Concatenation) is the set df 520 strings of length two, each consisting of one letter followed by one digit.*
3. *L^4 is the set of all 4-letter strings.*
4. *L* is the set of all strings of letters, including ε, the empty string.*
5. *L (L U D)* is the set of all strings of letters and digits beginning with a letter.*
6. *D^+ is the set of strings with one or more digits*

Regular expressions:

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

Definition:

Let **Σ** be an alphabet. The Regular expression over the **Σ** and the languages they denoted (or generated) are defined below:

1. If Φ is an RE, L(Φ) = Φ, is a Language of empty set Φ.
2. If ε is an RL, L(ε) = {ε}, is a Language contains only empty strings
3. For any ***a*** in Σ, *a* is an RE. L(*a*) = {*a*} is a language consisting of only one string
4. If *r* and *s* are Regular expression denoting the languages L_R and L_S respectively
   - (*r*|*s*) is an RE, then L(*r* U *s*) = L(*r*|*s*) = L_R U L_S is also a language.
   - (*rs*) is an RE, then L(*rs*) = L_R.L_S
   - (r^*) is an RE, L(r^*) = $R^* = \bigcup_{i=0}^{\infty} R^i$ (where L^* is called the Kleene closure of L)

Examples:

Let Σ = {a, b}.

1. The regular expression (*a*|b) denotes the language L = {*a*,b}.
2. (*a*|b)(*a*|*b*) denotes {*aa*, *ab*, *ba*, *bb*), the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is *aa*|*ab*|*ba*|*bb*.
3. *a** denotes the language consisting of all strings of zero or more a's, that is, {ε, *a*, *aa*, *aaa*,.. .}.
4. (*a*|*b*)*denotes the set of all strings consisting of zero or more instances of *a* or *b*, that is, all strings of *a*'s and *b*'s: { ε, a, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, ...}. Another regular expression for the same language is (*a***b***)*.
5. *a*|*a***b* denotes the language {a, b, ab, aab,aaab,...), that is, the string *a* and all strings consisting of zero or more *a*'s and **ending** in ***b***.
6. (*a*|*b*)*(*abb*) denotes the set of all strings of *a* and *b* **ends with *abb***: {*abb, aabb, babb, aaabb, ababb, baabb, bbabb, ……*}

Algebraic laws and Identity rules:

A language that can be defined by a regular expression is called a ***regular* set**. If two regular expressions r and s denote the same regular set, we say they are *equivalent* and write r = s. For instance, (alb)= (bla). There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular Definitions (Auxiliary Definitions):

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

$$D_1 \rightarrow r_1$$

$$D_2 \rightarrow r_2$$

. . . .

$$D_n \rightarrow r_n$$

where:

1. Each D_i is a new symbol, not in Σ and not the same as any other of the D's, and
2. Each r_i is a regular expression over the alphabet Σ U {D_1, D_2,... ,D_{i-1}).

By restricting r_i to Σ and the previously defined D's, we avoid recursive definitions, and we can construct a regular expression over Σ alone.

For each r_i.

- ✓ First replacing uses of D_1 in r_2 (which cannot use any of the D's except for D_1).
- ✓ Then replacing uses of D_1 and D_2 in r_3 by r_1 and r_2, and so on.
- ✓ Finally, in r_i we replace each D_i, for i = 1, 2, .. . , n - 1, by the substituted version of r_i, each of which has only symbols of Σ.

Example 1: The **regular *definitions*** of C language *identifiers.*

$$\begin{aligned} \mathit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \mathit{id} &\rightarrow \mathit{letter\_} \ (\ \mathit{letter\_} \mid \mathit{digit} \ )^* \end{aligned}$$

Example 2: The **regular *definition*** for ***Unsigned numbers*** (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.

$$
\begin{array}{rcl}
digit & \rightarrow & 0 \mid 1 \mid \cdots \mid 9 \\
digits & \rightarrow & digit\ digit^* \\
optionalFraction & \rightarrow & .\ digits \mid \epsilon \\
optionalExponent & \rightarrow & (\ \mathtt{E}\ (\ + \mid - \mid \epsilon\ )\ digits\ ) \mid \epsilon \\
number & \rightarrow & digits\ optionalFraction\ optionalExponent
\end{array}
$$

Extensions:

- *One or more instances*: (r)+

  (Two useful algebraic laws, r* = r$^+$ **|** ε and r$^+$ = rr* = r*r relate the Kleene closure and positive closure.)

- *Zero or one instance*: r? (That is, r? is equivalent to (r **|** ε))
- *Character class*: [abc] (Thus, [abc] is shorthand for a**|**b**|**c, and [a-z] is shorthand for
  a **|** b **|**. ..**|**z)

The **shorthanded notation** of above Example 1 and 2 are:

$$
\begin{array}{rcl}
letter\_ & \rightarrow & [\mathtt{A\text{-}Za\text{-}z\_}] \\
digit & \rightarrow & [\mathtt{0\text{-}9}] \\
id & \rightarrow & letter\_\ (\ letter \mid digit\ )^*
\end{array}
$$

$$
\begin{array}{rcl}
digit & \rightarrow & [\mathtt{0\text{-}9}] \\
digits & \rightarrow & digit^+ \\
number & \rightarrow & digits\ (.\ digits)?\ (\ \mathtt{E}\ [\mathtt{+\text{-}}]?\ digits\ )?
\end{array}
$$

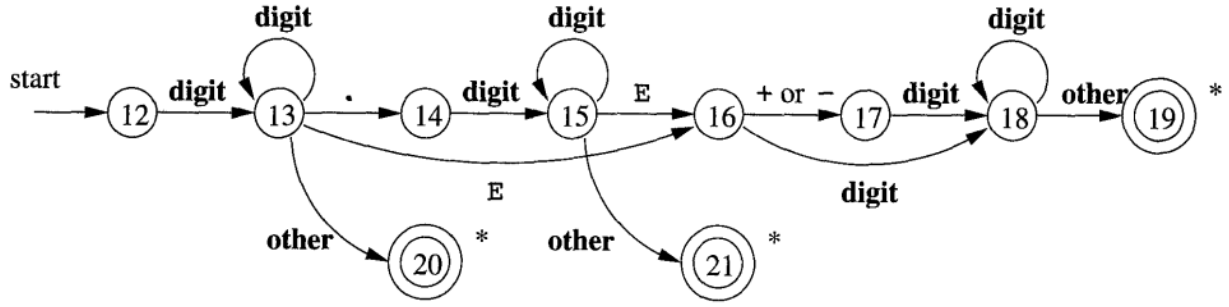
Recognition of Tokens:

The token of any specific language is taking the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

$$
\begin{array}{rcl}
digit & \rightarrow & [\mathtt{0\text{-}9}] \\
digits & \rightarrow & digit^+ \\
number & \rightarrow & digits\ (.\ digits)?\ (\ \mathtt{E}\ [\mathtt{+\text{-}}]?\ digits\ )? \\
letter & \rightarrow & [\mathtt{A\text{-}Za\text{-}z}] \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^* \\
if & \rightarrow & \mathtt{if} \\
then & \rightarrow & \mathtt{then} \\
else & \rightarrow & \mathtt{else} \\
relop & \rightarrow & \mathtt{<} \mid \mathtt{>} \mid \mathtt{<=} \mid \mathtt{>=} \mid \mathtt{=} \mid \mathtt{<>}
\end{array}
$$

For example, **if, then**, **else**, **relop**, **id**, and **number (**the terminals of a grammar**)**, are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions,

The transition diagram for **number** token is shown in Fig.



The **Regular definition of number token** is:

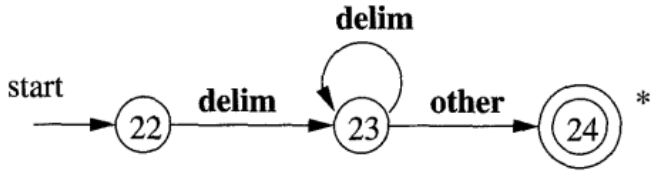
$$number \rightarrow digits\ (.\ digits)?\ (\ \mathtt{E}\ [+-]?\ digits\ )?$$

- Beginning in **state 12**, if we see a **digit**, we go to **state 13**. In that state, we can read any number of additional digits.
- Suppose if the number is 123 an integer, then we will end at state 20 and retract to 13 recognize it as a token type integer.
- Suppose if number 123 followed by a **dot**, (say floating number 123.056), then we have an "**optional fraction**." **State 14** is entered, and we look for one or more additional digits; **state 15** is used for that purpose.
- If we see an E (say 123.056E(+25)), then we have an "**optional exponent**," whose recognition is the job of **states 16** through **19**.
- In **state 15**, instead of ε if see more digit, then we have come to the **end of the fraction**, there is **no exponent**, and we return the lexeme found, via **state 21**.

The lexical analyzer the job of stripping out whitespace, by recognizing the "token" ***ws*** defined by:

$$ws \rightarrow (\ \mathbf{blank}\ |\ \mathbf{tab}\ |\ \mathbf{newline}\ )^{+}$$

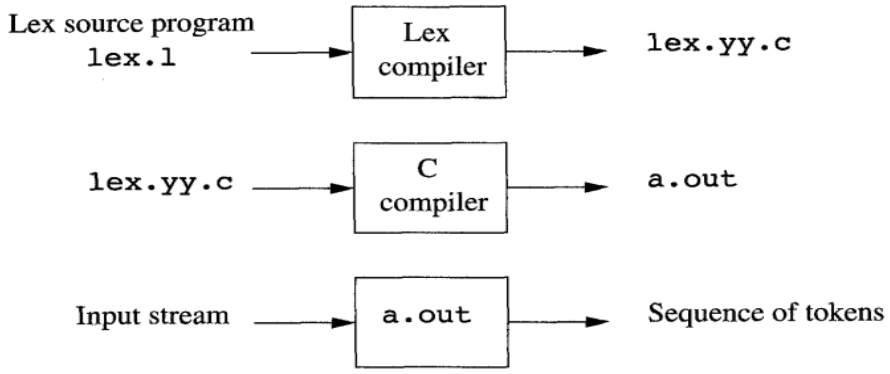
The **transition diagram**, for **whitespace.**



In the diagram, we look for one or more "**whitespace**" characters, represented by **delim** (blank, tab, newline), and other characters that are not considered by the language design to be part of any token.

Language for Specifying Lexical Analyzer:

- The Lexical-Analyzer Generator **Lex** is a tool for automatically generating lexical Analyzer.
- The input source program (LEX program) is a set of *regular expressions together with an action*, which is called **lex.l**
- The Lex compiler transforms **lex.l** to a C program, in a file that is always named **lex. yy.c**.
- The **lex.yy.c** file is then compiled by the C compiler into file called **a.out**
- The C-compiler output (i.e a.out) is a **working lexical analyzer** that can take a ***stream of input*** *characters* and produce a ***stream of tokens***.
- Figure shows how to create a Lexical Analyzer using LEX tool



Structure of Lex Programs

A Lex program has the following form:

declarations

%%

translation rules

%%

auxiliary functions

Declarations Section

- Define variables, constants, and **regular definitions** (shorthand for regular expressions).

Translation Rules Section

- Each rule has the form:

Pattern {Action}

- Pattern = regular expression
- Action = C code to execute when the pattern is matched

Auxiliary Functions Section

- Contains helper C functions, if needed.

Working of Lexical Analyzer:

1. Parser calls the Lexical Analyzer
   - The parser asks for the next token.
2. Lexical Analyzer starts reading the input
   - It reads one character at a time from the remaining input.
3. Find the longest matching pattern
   - It keeps reading until it finds the **longest prefix** of characters that matches a **token pattern (Pi)**.
4. Perform the associated **action (Ai)**
   - Once a **match is found**, the lexical analyzer executes the corresponding **action Ai** (for example, **create a token**).
5. Return or Skip
   - If the token is meaningful (like an identifier, keyword, or operator), it **returns the token to the parser**.
   - If the token is not needed (like **whitespace or a comment**), it **skips it** and continues scanning the next part of the input.
6. Repeat
   - The process repeats until the **end of input** is reached.

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Finite Automata:

Finite automata are **recognizers**; they simply say "yes" or "no" about each possible input string.

*Finite automata come in **two** types:*

- ✓ **Nondeterministic finite automata (NFA)**: N**o restrictions** on the labels of their edges. A symbol in Σ can label several edges out of the same state, and **ε**, the **empty string**, is a possible label.
- ✓ **Deterministic finite automata (DFA)**: For each state, and for each symbol of its input alphabet (Σ) exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of **recognizing the same languages**.

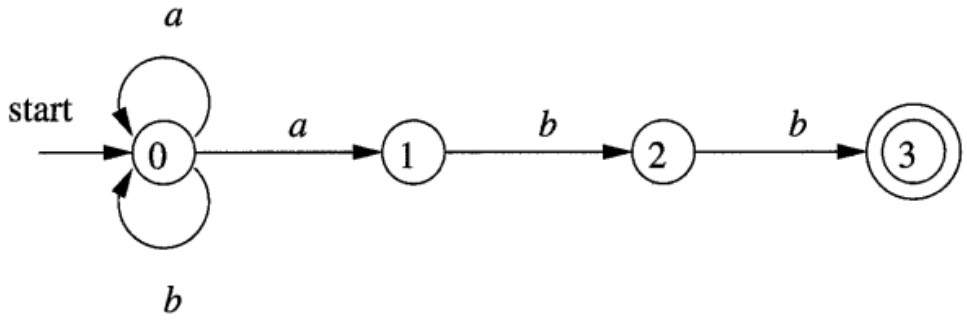
These languages are exactly the same languages, called the ***regular languages***, that describes a regular expression.

Nondeterministic Finite Automata:

A nondeterministic finite automaton (NFA) is a five tuple: {Σ. S, δ, s_o , F}:

1. A finite set of states S.
2. A set of input symbols Σ, the input alphabet. We assume that ε, which stands for the empty string, is never a member of Σ.
3. A transition function δ: S x Σ → 2^S, gives for each state in S, and for each symbol in Σ U {ε} a set of next states.
4. A state s_o ∈S that is distinguished as the start state (or initial state).
5. A set of states F ⊂ S (subset of S), that is distinguished as the accepting states (or final states)

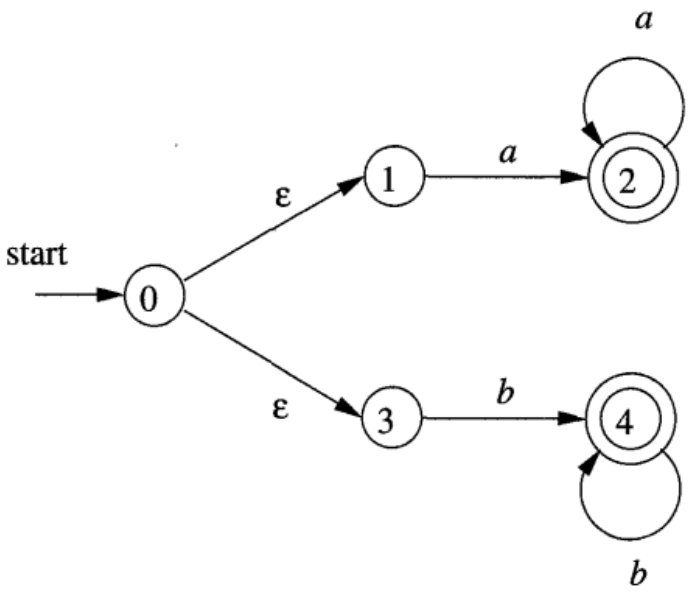
Example: The transition graph for an **NFA** recognizing the **language of regular expression (a|b)*ab**



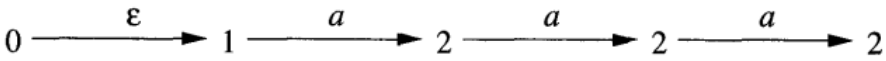
The transition table for the following NFA is

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Example: Construct an NFA accepting L(**aa*|bb*). (**called NFA with ε -moves**)**



String '***aaa***' is accepted because of the path. Note that 'ε "disappear" in a concatenation, so the label of the path is *aaa.*

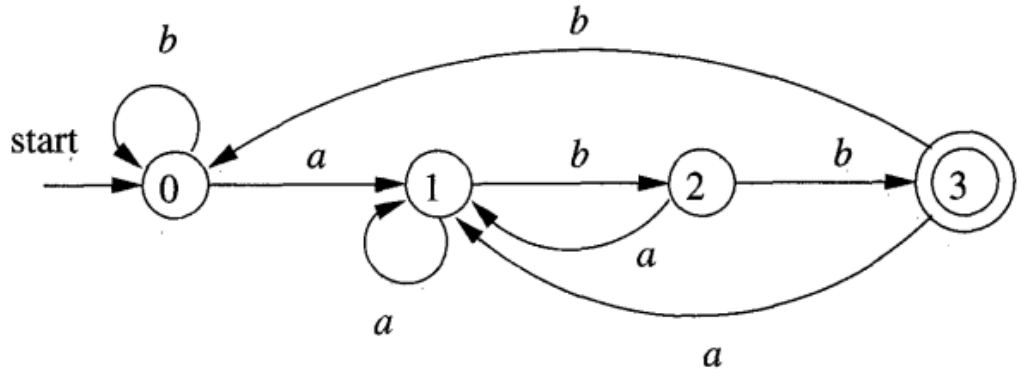


Deterministic Finite Automata:

A deterministic finite automaton (DFA) is a special case of an NFA where:

1. There are **no moves** on input **ε** and
2. For each state ***s*** and input symbol ***a***, there is exactly one edge out of *s* labeled *a.*

Example: The DFA for a language which has a set of all strings of ***a*** and ***b***, end's with ***abb***



Algorithm: Simulating a DFA

```
s = s_o;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

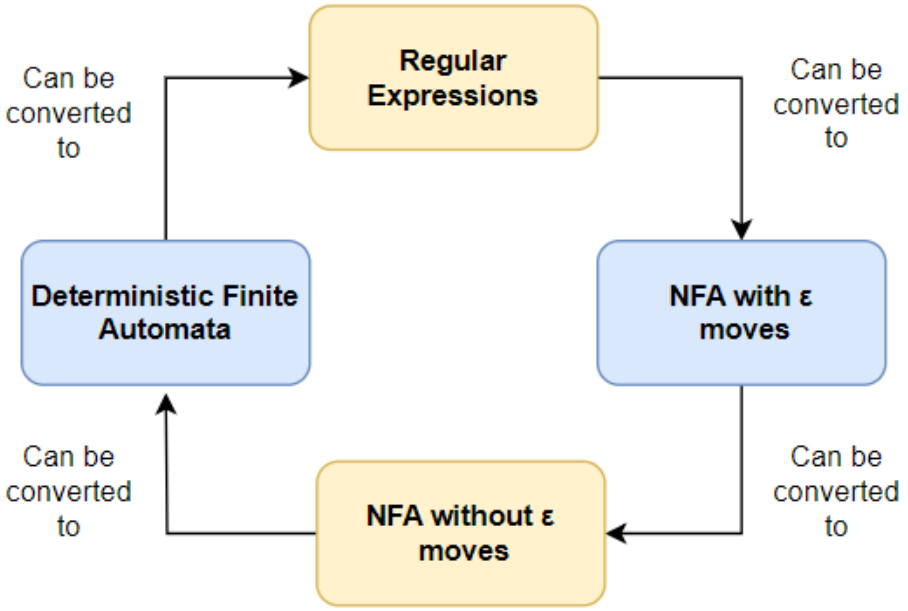
From Regular Expressions to Finite Automata:

The task of a DFA is to find whether a string is in the language denoted by a regular expression.

Regular expression is a natural notation for describing the language of a token.

The direct

It is difficult to construct DFA directly from the regular expression, we use an algorithm that produce an NFA from regular expression and then convert NFA to DFA.

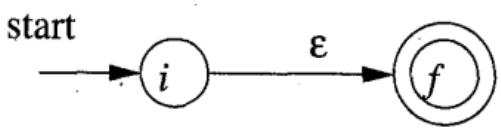


Construction of NFA from a Regular Expressions:

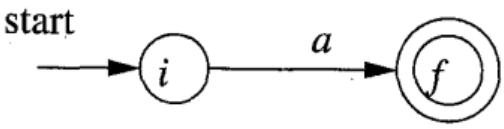
Algorithm : The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA**.**

INPUT: A regular expression R over alphabet Σ.
OUTPUT: An NFA **N** accepting L(R).
METHOD: Begin by parsing R into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressions with no operators, **and inductive rules** for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expressions.

Basis: NFA for regular expression ε

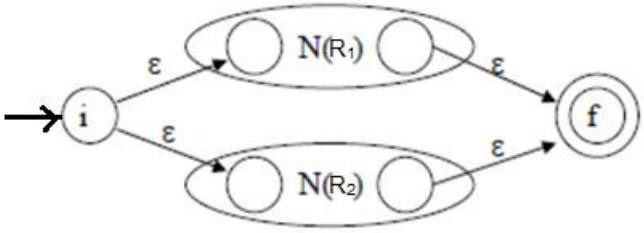


For any subexpression **a** in Σ, construct the NFA

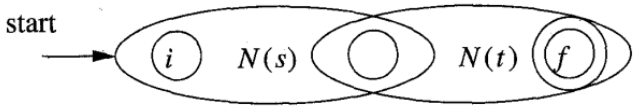


INDUCTION: Suppose N(R_1) and N(R_2) are NFA's for regular expressions R_1 and R_2, respectively. Then

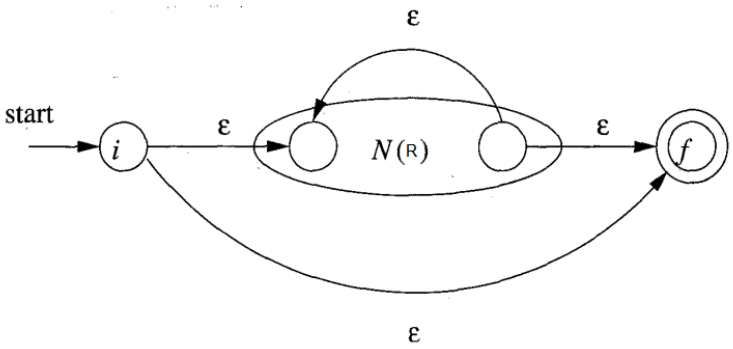
(a) For the $(R_1|R_2)$: Union of R_1 and R_2



(b) For the regular expression $R= (R_1.R_2)$: Concatenation of R_1 and R_2
The start state of N (R1) becomes the start state of N (R), and the accepting state of N(R2) is the only accepting state of N(R).



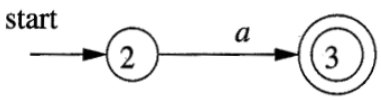
(c) For the regular expression R^* (Kleene Closure of R)

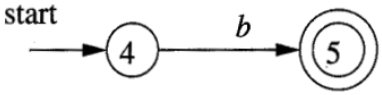


Example: Construct the NFA for a regular expression R = (a|b)*abb.

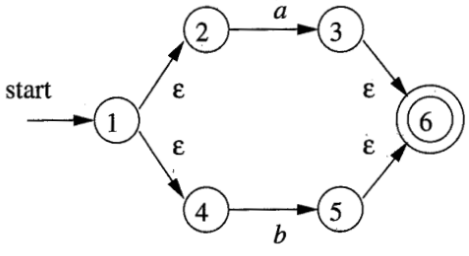
Sol: Let R1 = (a|b)* be a regular expression and R2= abb be another.

Step1: Draw the transition diagrams for individual symbols a and b in the R1





Step 2: Draw the NFA for (a|b)* and NFA for the string as shown in below figure (abb)



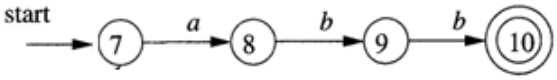


Figure shows the NFA for the regular expression (a|b)*abb