

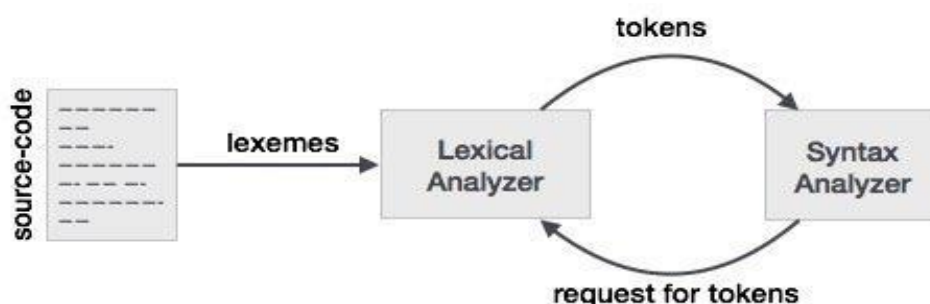
## Q.1 Explain Role of lexical analyzer

Ans.

- Lexical analysis is the first phase of a compiler.
- Its main task is to read the input characters or form of sentences from pre-processor.
- After that produce as output like sequence of token that the parser uses for syntax analysis.

⇒ There are other functions performed by lexical analyzer given below.

1. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
  2. If the lexical analyzer finds a token invalid, it generates an error.
  3. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
- The lexical analyzer works closely with the syntax analyzer.
  - Sometime Lexical analysis divided into cascade of two phase:
    - 1) Scanning
    - 2) Lexical analysis



## Q.2 Explain Token, Lexim, pattern and regular expression.

Ans.

⇒ Token:

- A sequence of characters (alphanumeric) in the source program having some collective meaning is called token.
- In other word it is pair consisting of token name and an optional attribute value.
- Token name is an abstract symbol representing a kind of lexical unit.
- Keyword, operator constants, identifiers, literal strings, punctuation symbols are called token.

⇒ Lexim:

- Lexemes are said to be a sequence of characters (alphanumeric) matched by the pattern for a token.
- There are some predefined rules or description for every lexeme to be identified as a valid token.

⇒ Pattern:

- Rules for lexim are defined by grammar rules means by pattern.
- A pattern explains what can be a token,
- It gives the description for lexim using regular expression.

➤ E.G.:

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

```
int (keyword), value (identifier), = (operator), 100 (constant) and ;  
(symbol).
```

```
const pi = 3.1416;
```

the substring pi is a lexeme for the token "identifier."

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

⇒ Token attribute:

- When more than one lexeme matches a pattern, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler.
- The lexical analyzer collects information about tokens into their associated attributes.
- The token influence parsing decision; the attributed influence the translation of tokens.
- A token has usually only a single attribute- a pointer (index) to the symbol-table entry in which the information about the token is kept.

Q.3 Give definition.

Ans.

⇒ Alphabets:

- Any finite set of symbols.
- E.g. {0,1} is a set of binary alphabets,  
{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of

Hexadecimal alphabets,  $\{a-z, A-Z\}$  is a set of English language alphabets.

⇒ Strings:

- Any finite sequence of alphabets is called a string.
- Length of the string is the total number of occurrence of alphabets.
- e.g., the length of the string tutorialspoint is 14 and is denoted by  $|\text{tutorialspoint}| = 14$ .
- A string having no alphabets.
- i.e. a string of zero length is known as an empty string and is denoted by  $\varepsilon$  (epsilon).

⇒ Language:

- A language is considered as a finite set of strings over some finite set of alphabets.
- Computer languages are considered as finite sets, and mathematically set operations can be performed on them.
- Finite languages can be described by means of regular expressions

⇒ **Special Symbols:**

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#

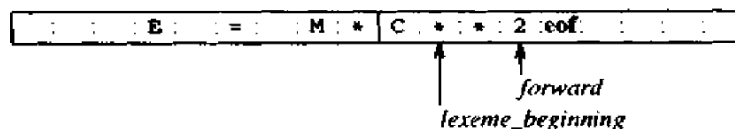
⇒ Terms:

TERM	DEFINITION
<i>prefix of s</i>	A string obtained by removing zero or more trailing symbols of string <i>s</i> ; e.g., <i>ban</i> is a prefix of <i>banana</i> .
<i>suffix of s</i>	A string formed by deleting zero or more of the leading symbols of <i>s</i> ; e.g., <i>nana</i> is a suffix of <i>banana</i> .
<i>substring of s</i>	A string obtained by deleting a prefix and a suffix from <i>s</i> ; e.g., <i>nan</i> is a substring of <i>banana</i> . Every prefix and every suffix of <i>s</i> is a substring of <i>s</i> , but not every substring of <i>s</i> is a prefix or a suffix of <i>s</i> . For every string <i>s</i> , both <i>s</i> and $\epsilon$ are prefixes, suffixes, and substrings of <i>s</i> .
<i>proper prefix, suffix, or substring of s</i>	Any nonempty string <i>x</i> that is, respectively, a prefix, suffix, or substring of <i>s</i> such that $s \neq x$ .
<i>subsequence of s</i>	Any string formed by deleting zero or more not necessarily contiguous symbols from <i>s</i> ; e.g., <i>baaa</i> is a subsequence of <i>banana</i> .

Q.4 Input buffering.

Ans.

- The LA scans the characters of the source program one at a time to discover tokens.
- But there are large number of data for matching and announcing token.
- Because of that la need to look ahead several time and speed of lexical analysis is concern.
- To reduce time of matching we have input buffering scheme use.
- There are several buffering schemes.
- We use a buffering divided into two N-character halves as shown bellow
- N is the number of characters on one disk block



**Fig. 3.3.** An input buffer in two halves.

- We read N input character into each half of the buffer with one system read command or read command for each input character.
- If fewer than N character remain in the input, then a special character eof (end of file) read into the buffering after the input characters.
- eof marks the end of source file and its different from any input character.
- Two pointers to the input buffer are maintained.
- The string of character between the two pointers is the current lexeme.
- Initially, both pointers point to the first character.

- A look ahead pointer or forward pointer is incremented until next lexeme to be found.
- The forward pointer scans ahead with a match for a pattern is found.
- Once the next lexeme determined, the forward pointer is set to the character at its right end.
- After the lexeme is prosed both pointers are set to the character immediately post to the lexeme.
- If the forward pointer is about to move past the halfway mark or right end of buffer the left halfway is filled with N new input characters.
- And forward pointer wraps around to the beginning of the buffer.
- Also reverse for first half.

⇒ Algorithm:

If forward at end of first half then

Begin

Reload second half;

Forward := forward +1;

End

Else if forward at end of second half then

Begin

Reload first half

Move forward to beginning of first half

End

Else

Forward := forward + 1;

### ⇒ **Disadvantages of this scheme**

- This scheme works well most of the time, but the amount of lookahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.
- **e.g.** DECLARE (ARG1, ARG2, . . . , ARGn) in PL/1 program;
- It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

➤ **Sentinel character:** is a special character that can not be the part of the source program.

- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore, the ends of the buffer halves require two tests for each advance of the forward pointer.
- We intend each buffer half to hold a sentinel character at the end.



- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

⇒ **Advantages of this scheme:**

- Most of the time, it performs only one test to see whether forward pointer points to an *eof*.
- Only when it reaches the end of the buffer half or *eof*, it performs more tests.
- Since N input characters are encountered between *eofs*, the average number of tests per input character is very close to 1.

# Algorithm to advance forward pointer to sentinel

Page No.:

Date: / / 2018

version1

version2

code to advance forward pointer

if forward at end of first half then  
begin

reload second half;  
forward := forward + 1;

end  
else if forward at end of second half then begin  
reload first half;  
move forward to beginning of first half;

end  
else

forward := forward + 1;

A sentinel is a special character that can't be a part of the source program. We extend each buffer half to hold a sentinel character at the end.

forward := forward + 1;  
if forward = eof then  
begin

if forward at end of first half then begin

reload second half;  
forward := forward + 1;

end

else if forward at end of second half then begin  
reload first half;  
move forward to beginning of first half.

end

else

terminate lexical analysis.

end

## Q.5 Explain operation language.

Ans.

- There are several operations on language that can be applied.

- For example

Union

Concatenation

Closure

exponentiation

- E.G: let  $L$  be the set  $\{A,B,C,\dots,Z,a,b,c,\dots,z\}$   
 $d=\{0,1,2,3,4,5,6,7,8,9\}$
- Examples of new languages created from  $L$  and  $D$ 
  1.  $L \cup D$  is the set of letters and digits.
  2.  $LD$  is the set of strings consisting of a letter followed by a digit.
  3.  $L^4$  is the set of all four-letter strings.
  4.  $L^*$  is the set of all strings of letters, including  $\epsilon$ , the empty string.
  5.  $L(L \cup D)^*$  is the set of all strings of letters and digits beginning with a letter.
  6.  $D^+$  is the set of all strings of one or more digits.  $\square$

OPERATION	DEFINITION
union of $L$ and $M$ written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of $L$ and $M$ written $LM$	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of $L$ written $L^*$	$L^* = \bigcup_{i=0}^{\infty} L^i$ $L^*$ denotes "zero or more concatenations of" $L$ .
positive closure of $L$ written $L^+$	$L^+ = \bigcup_{i=1}^{\infty} L^i$ $L^+$ denotes "one or more concatenations of" $L$ .

Fig. 3.8. Definitions of operations on languages.

## Q.6 Regular Expressions.

Ans.

- A regular expression is built out of simpler regular expression over alphabet ( $\Sigma$ ) using a set of definition rules.
- Each regular expression ( $r$ ) denote a language  $L(r)$
- The grammar defined by regular expressions is known as **regular grammar**.
- The language defined by regular grammar is known as **regular language** or **regular set**.
- 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.

⇒ Rules:

1. Every letter of the alphabet  $\Sigma$  is a regular expression.
2. Null string  $\epsilon$  and empty set  $\Phi$  are regular expressions.
3. If  $r_1$  and  $r_2$  are regular expressions, then above statements are also give regular expression given below in figure

- (i)  $r_1, r_2$
- (ii)  $r_1 r_2$  (concatenation of  $r_1 r_2$ )
- (iii)  $r_1 + r_2$  (union of  $r_1$  and  $r_2$ )
- (iv)  $r_1^*, r_2^*$  (kleen closure of  $r_1$  and  $r_2$ )
- (v)  $r_1$  as string and  $r_1$  as symbol

Suppose  $r$  and  $s$  are regular expressions denoting the languages  $L(r)$  and  $L(s)$ . Then,

- a)  $(r)|(s)$  is a regular expression denoting  $L(r) \cup L(s)$ .
- b)  $(r)(s)$  is a regular expression denoting  $L(r)L(s)$ .
- c)  $(r)^*$  is a regular expression denoting  $(L(r))^*$ .
- d)  $(r)$  is a regular expression denoting  $L(r)$ .<sup>2</sup>

Q.7 what is regular definition.

Ans.

- A regular definition gives names to certain regular expressions and uses those names in other regular expressions.
- They are used to represent the language for lexical analyzer.
- They assist in finding the type of token that accounts for a particular lexeme
- In other word it is notation convention for lexeme patterns for token.
- And to decline regular expressions using these  $\Sigma$  name in an alphabet of basic symbol.
- Regular definition is a sequence definition of the form.

$D_1 \rightarrow R_1$

$D_2 \rightarrow R_2$

$D_3 \rightarrow R_3$

$D_n \rightarrow R_n$

- Where each  $d_i$  is a distinct name.

- And each  $R_i$  is Regular expression over the extended alphabet  $V \cup \{D_1, D_2, D_3, \dots, D_N\}$ .
- Here is a regular definition for the set of Pascal identifiers that is define as the set of strings of letter and digits beginning with a letter.

letter  $\rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$

digit  $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

id  $\rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$

- Unsigned numbers in pascal are string such as 5280, 39.37, etc.

**6.336E4, or 1.894E-4.** The following regular definition provides a precise specification for this class of strings:

```

digit → 0 | 1 | . . . | 9
digits → digit digit*
optional_fraction → . digits | ε
optional_exponent → ( E ( + | - | ε ) digits ) | ε
num → digits optional_fraction optional_exponent

```

This definition says that an **optional\_fraction** is either a decimal point followed by one or more digits, or it is missing (the empty string). An **optional\_exponent**, if it is not missing, is an **E** followed by an optional **+** or **-** sign, followed by one or more digits. Note that at least one digit must follow the period, so **num** does not match **1.** but it does match **1.0.** □

Q.7 what is Transition diagrams.

Ans.



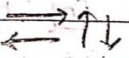
- Lexical analysis use transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
- A transition diagram is similar to a flowchart for (a part of) the lexer.
- We draw one for each possible token.
- It shows the decisions that must be made based on the input seen.
- The two main components are

1. Circles: its use for show Positions in a transition diagram and are called states (think of them as decision points of the lexer).
- The states are connected by arrows, called edges (think of them as the decisions made).
- The living state's have labels indicating the input character that can next appear after the transition diagram has reached states.
- A double circle indicated an accepting state, a state in which a token is found.
- One state is labeled the start state if it is the initial state of the transition diagram.
- $a^*$  indicates that input retraction must take place.
- Transition diagrams are followed one by one trying to determine the next tokens to be returned.
- If failure occurs while we are following one transition diagram, we retract the forward pointer to where it was in the start state of this diagram, and activate the next transition diagram.
- Transition diagrams are depending the actions that take place when a lexical analyzer is called by parser.



## \* Transition Diagram





- It can be defined as stylish flowchart signs:-

- (1)   $\rightarrow$  It is to represent state
- (2)   $\rightarrow$  Final state (Accepting state)
- (3)   $\rightarrow$  Directed Edges containing some label about workflow.
- (4) \*  $\rightarrow$  represent initial state.
- (5) other  $\rightarrow$  unknown signs.

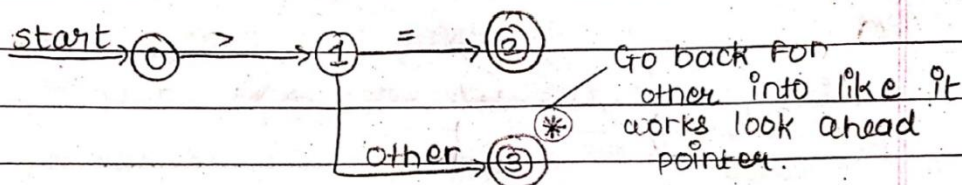
eg. Regular expressions are converted into diagrams.

Regular Expressions a label.

(1) start,  a 

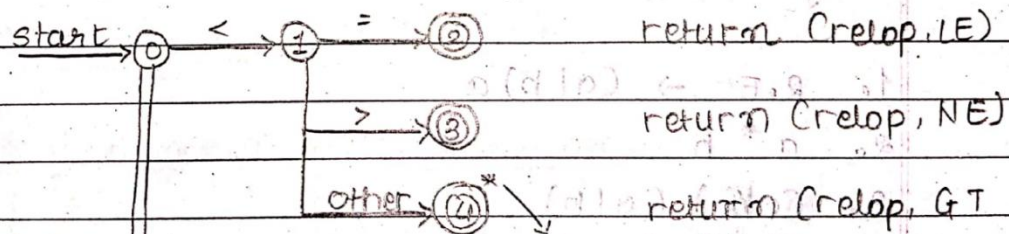
(2) R.E. \* (ablaa)  
 start,  a  b   
 other  \*

$\rightarrow$  Transition Diagram for relop ( $\geq$ ).

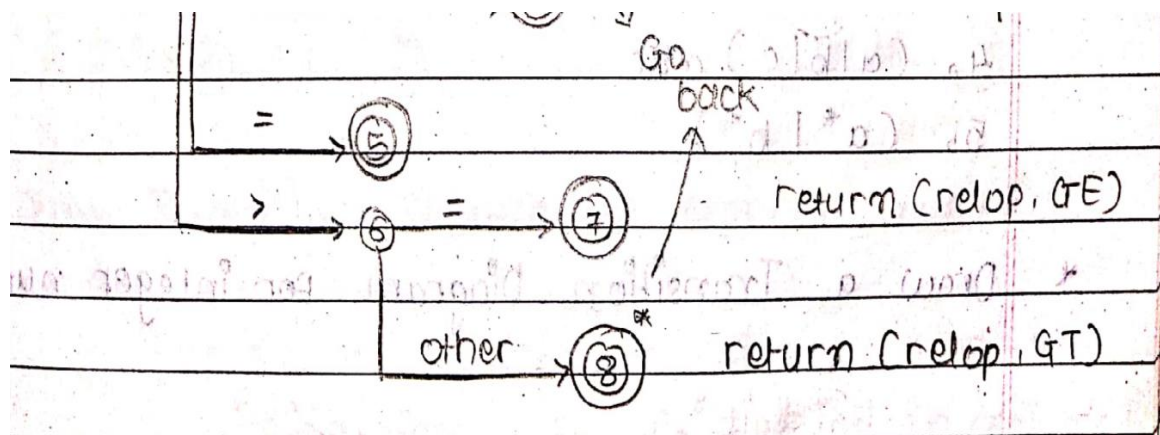


$\rightarrow$  Transition Diagram for relational operators.

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



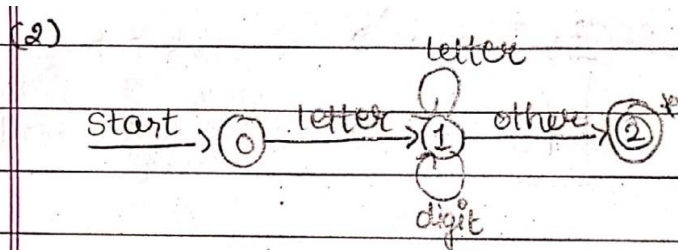
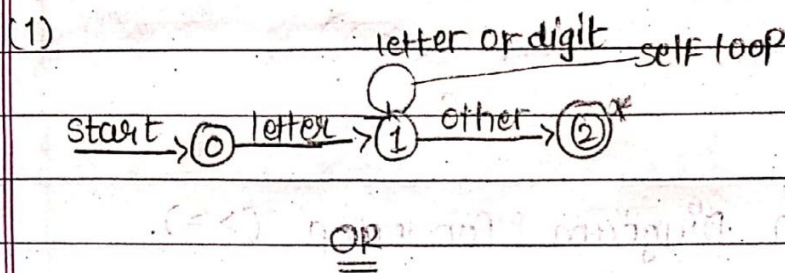




\* Draw a Transition Diagram for valid Identifier

Regular Expression is given as below:-

Letter (Letter | Digit)\*



\* Draw a Transition Diagrams for following regular expressions:-

1. R.E  $\rightarrow (a|b)a$

2.  $a^*b$

3.  $(a|b)(a|b)$

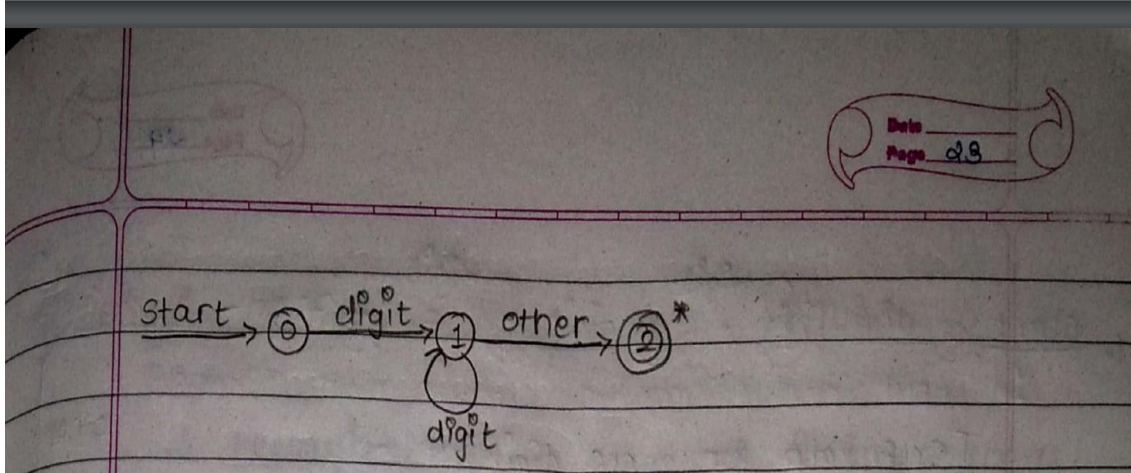
4.  $(a|b|c)abc$

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

\* Draw a Transition Diagram for integer number.

$\text{digit}^+$  or  $\text{digit} \cdot \text{digit}^*$

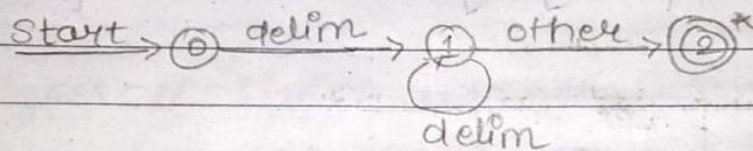
Scanned by CamScanner



\* Draw a Transition Diagram to recognize [white space, Blank, Tab, newline].

$\text{ws}^+$  or  $\text{delim}^+$

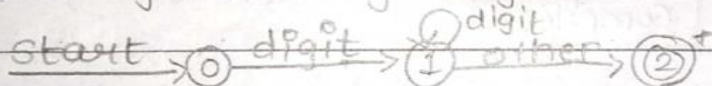
$\text{delim}^+$  or  $\text{delim} \cdot \text{delim}^*$



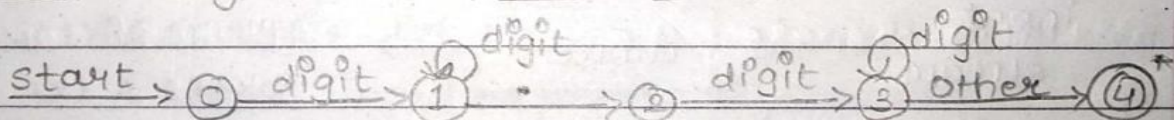
\* Draw Transition Diagram for Real Numbers.

eg 10.25, 27.1, 123.789

$\text{digit}^+ = \text{digit} \text{digit}^*$  ← closure



Final Diagram for Real Numbers.





## Q.8 Deterministic and non-deterministic finite automata.

Ans.

- A *recognizer* for a language is a program that takes a string  $x$ , and answers “yes” if  $x$  is a sentence of that language, and “no” otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.

---

\* Finite Automata: Both automata are capable of recognizing precisely the regular sets.

---

---

\* Non-Deterministic: more than one transition out of a state are possible on the same input symbol.

---

---

\* A language defined by regular expression called as regular sets.

---

- Which one?

deterministic – faster recognizer, but it may take more space

non-deterministic – slower, but it may take less space

Deterministic automata are widely used lexical analyzers.

- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

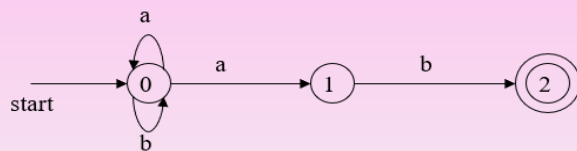
Algorithm1: Regular Expression  $\rightarrow$  NFA  $\rightarrow$  DFA (two steps: first to NFA, then to DFA)

Algorithm2: Regular Expression  $\rightarrow$  DFA (directly convert a regular expression into a DFA)

- **A non-deterministic finite automaton (NFA)** is a mathematical model that consists of:
  - $S$  - a set of states
  - $\Sigma$  - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - $s_0$  - a start (initial) state
  - $F$  – a set of accepting states (final states)
- $\epsilon$ - transitions are allowed in NFAs.

- In other words, we can move from one state to another one without consuming any symbol.
- In NFA, the same character can label two or more transition out of one state.
- An NFA can be represented diagrammatically by a labelled directed graph, called a transition graph in which the node are the states and the labelled edge represent the transition function.

### NFA (Example)



Transition graph of the NFA

0 is the start state  $s_0$   
 $\{2\}$  is the set of final states  $F$   
 $\Sigma = \{a, b\}$   
 $S = \{0, 1, 2\}$

Transition Function:

	a	b
0	$\{0, 1\}$	$\{0\}$
1	—	$\{2\}$
2	—	—

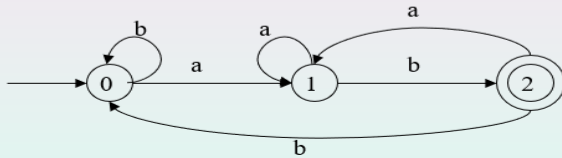
The language recognized by this NFA is  $(a|b)^* a b$

### \*\* Transition Table \*\*

state	input symbol	b
0	$\{0, 1\}$	$\{0\}$
1	—	$\{2\}$
2	—	$\{3\}$
3	—	—

# Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
  - no state has  $\epsilon$ - transition
  - for each symbol  $a$  and state  $s$ , there is at most one labeled edge  $a$  leaving  $s$ .  
i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by  
this DFA is also  $(a|b)^* a b$

- A DFA has almost one transition from each state on input.
- Each entry in the transition table has a single state.
- no state allows with  $\epsilon$ - transition

=> algorithms and example from books



## Q.10 Difference between NFA and DFA.

Ans.

NFA	DFA
1. Input label $\epsilon$ (sign) is allowed	It is not allowed at all.
2. NFA is a kind of mathematical model.	It is a special kind of NFA/Extension of NFA.
3. Edges living from a particular state may have identical input symbols.	Edges living from particular state should have distinct input symbols.
4. NFA diagrams are smaller than DFA diagrams.	DFA diagrams are bigger than NFA diagrams.
5. Backtracking is possible.	There is no chance of backtracking.
6. NFA works less efficiently than DFA.	They are faster than NFA.

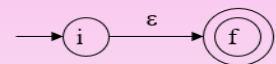
## Q.11 Converting a RE into an NFA using Thomson Construction.

Ans.

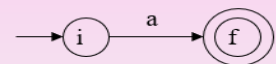
- This is one way to convert a regular expression into an NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create an NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

### Thomson's Construction (cont.)

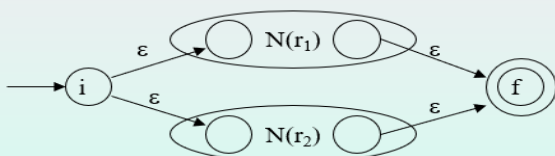
- To recognize an empty string  $\epsilon$



- To recognize a symbol  $a$  in the alphabet  $\Sigma$



- If  $N(r_1)$  and  $N(r_2)$  are NFAs for regular expressions  $r_1$  and  $r_2$ 
  - For regular expression  $r_1 \mid r_2$

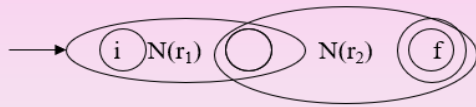


NFA for  $r_1 \mid r_2$



## Thomson's Construction (cont.)

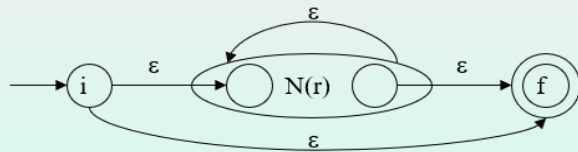
- For regular expression  $r_1 r_2$



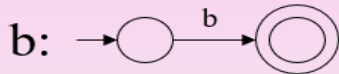
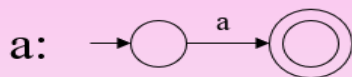
Final state of  $N(r_2)$  become final state of  $N(r_1 r_2)$

NFA for  $r_1 r_2$

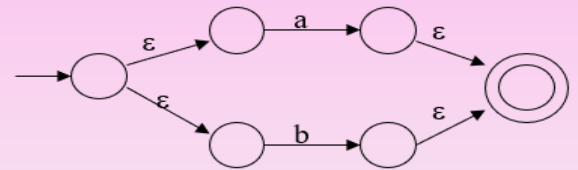
- For regular expression  $r^*$



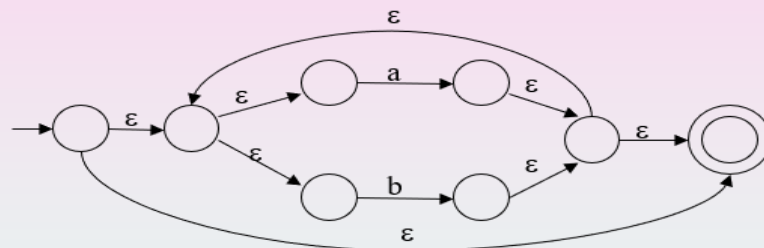
NFA for  $r^*$



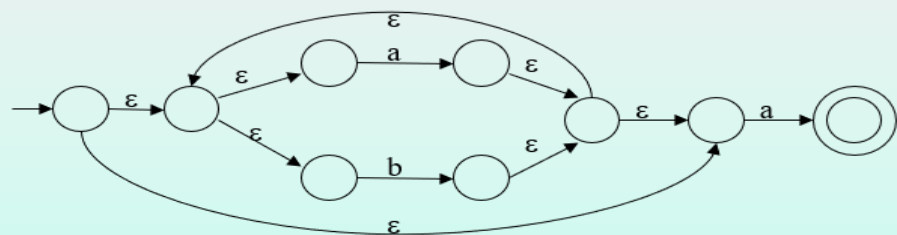
$(a \mid b)$



$(a|b)^*$



$(a|b)^* a$



Ans.

- Few errors are detected at lexical level alone, because a lexical analyzer has a very localized view of a source program.
- For example, if the string **fi** is encountered in a C program for the first time in the context

**fi** ( **a == f(x)**) .....

whether **fi** is a misspelling of the keyword **if** or an undeclared function identifier

Since **fi** is a valid identifier, the lexical analyzer must return the token for an identifier and let latter phase handle any error.

- A lexical analyzer finds an error when it is unable to proceed because none of the patterns matches a prefix of the remaining input.
- The simplest recovery strategy is “panic mode”, to delete successive characters from the remaining input until the lexical analyzer can find a well-formed token.
- Other possible error-recovery actions are:

Deleting an extraneous character

Inserting a missing character

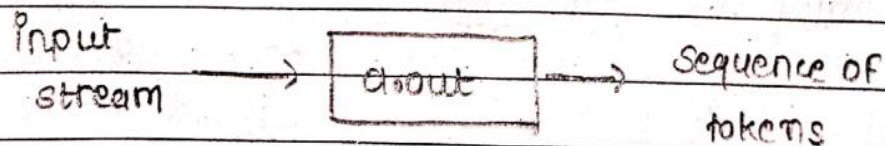
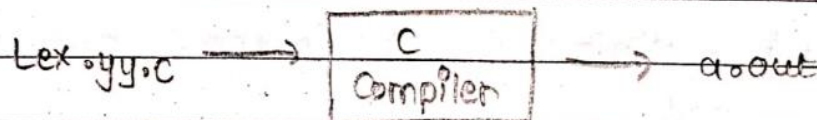
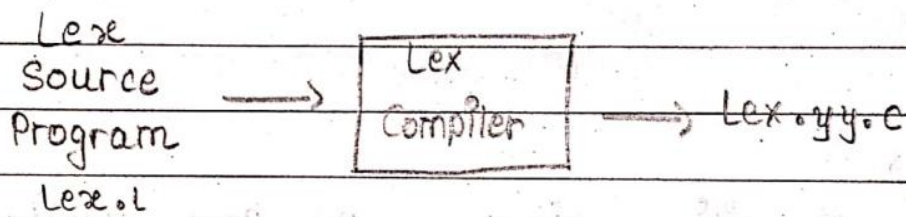
Replacing an incorrect character by a correct character

Transposing two adjacent characters

- Error transformation attempts to repair the input.
- The simplest strategy is to see if a prefix of the remaining input can be transformed into a valid lexeme by a single error transformation.

- This strategy assumes most lexical errors are the result of a single transformation.

\* A language for specifying lexical analyzer



A lex program consist following 3 parts:

1. Declaration
2. Translation Rules
3. Procedures