

## Chapter– Two

# **Lexical Analysis**

# Basic Topics in chapter-Two

- *Introduction of lexical analysis*
- *Function of Lexical Analysis*
- *Role of lexical analyzer*
- Lexeme, Token, and patterns
- Finite automata
  - Basic terms: Alphabet, empty string, Strings and languages
- Types of finite automata (DFA and NFA)
- Equivalence of DFA and NDF
- Minimizing the number of states of a DFA
- Regular Language
- Regular expressions
- From regular expressions to finite automata
- Specification of tokens
- Recognition of tokens
- Implementation of a lexical analyzer

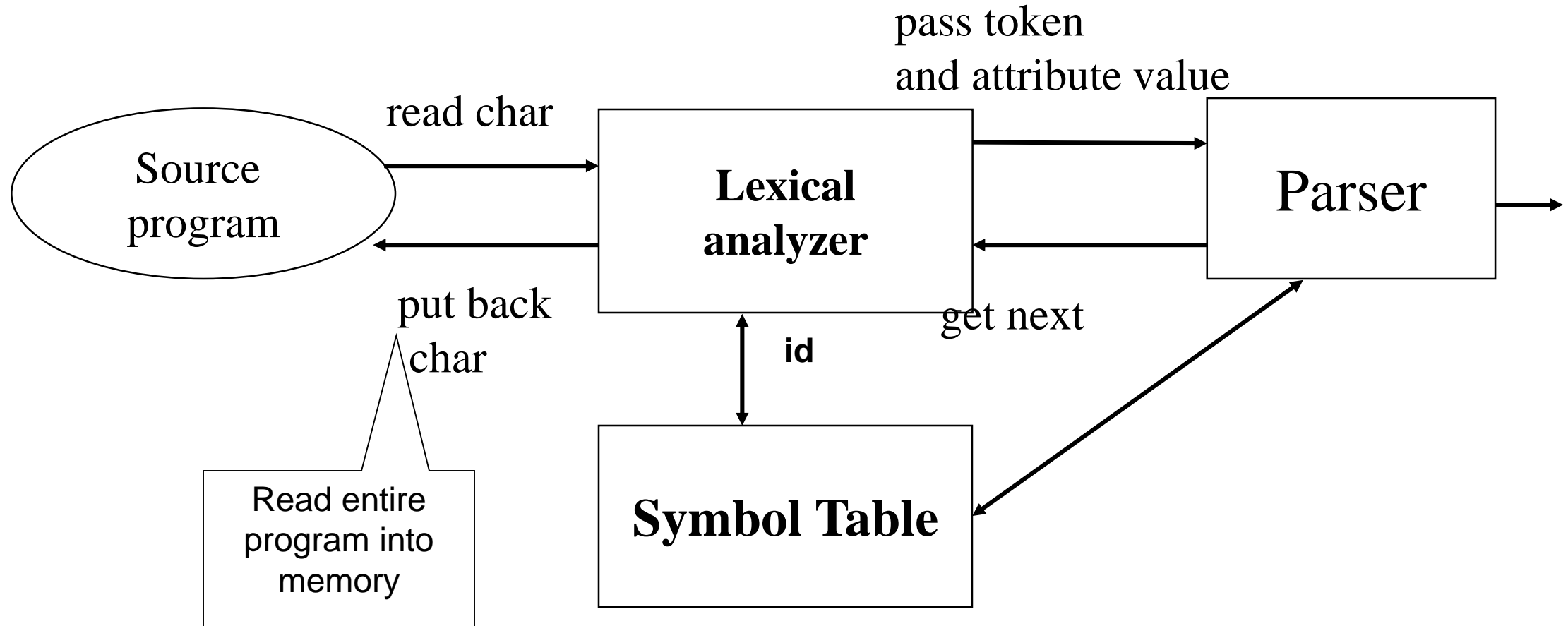
## 2.1. Introduction of lexical analysis

- Lexical analysis is the first phase of a compiler. It is also called **Scanner**
- It takes the modified source code from language preprocessors that are written in the form of sentences.
- The lexical analyzer breaks these source codes into a series of **tokens**, by removing any *whitespace* or *comments* in the source code.
  - ✓ *A **token**- describe a pattern of characters having some meaning in the source program.*  
*Such as: identifier, operators, keywords, numbers, delimiter etc.*
- If the lexical analyzer finds a token invalid, it generates **an error**.
- **Lexical Analysis** can be implemented with the *Deterministic finite Automata*.
- The output is a sequence of tokens that are sent to the parser for syntax analysis

## 2.2. Function of Lexical Analysis

- The main responsibility of LA/ Why LA is Important?
  - Scans and reads the source code character by character
  - *Tokenization* .i.e Grouping input characters into valid tokens
  - Eliminates comments and whitespace (*blank, newline, tab, and perhaps other characters that are used to separate tokens in the input*).
  - Report errors message with line number and display it
  - Retrieve and update symbol table stores identifier ---etc.

## 2.3 The Role of a Lexical Analyzer



# Lexical Analysis Versus Parsing

- There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.
  - i. Simplicity of design*
  - ii. Compiler efficiency is improved.*
  - iii. Compiler portability is enhanced.*

# Tokens, Patterns, and Lexemes

- When discussing lexical analysis, we use three related but distinct terms:
- Those terms are:
  - i. Tokens
  - ii. Lexemes
  - iii. Patterns

## i. Tokens

- **What's a token?**
  - A class or category of input stream
  - Or a set of strings in the same pattern
    - For example: In English: - *noun, verb, adjective, ...*
    - In a programming language: *Identifier, operators, Keyword, Whitespace,*
- The lexical analyzer scans the source program and produces as output a sequence of tokens, which are normally passed, one at a time to the parser.
- A ***token*** is a pair consisting of a token name and an optional attribute value.
  - The **token name** is an abstract symbol representing a kind of lexical unit,
  - The **token names** are the input symbols that the parser processes.
- In what follows, we shall generally write the name of a token in boldface.
- We will often refer to a token by its ***token name***.



# What are Tokens For?

- Classify program substrings according to role
- **Output of lexical analysis** is a stream of tokens . . . which is input to the parser.
- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

# Typical Tokens in a PL

- Tokens correspond to sets of strings.
  - **Symbols/operators:** +, -, \*, /, =, <, >, ->, ...
  - **Keywords:** if, else, while, struct, float, int, begin ...
  - **Integer and Real (floating point) literals:** 123, 123.45, 123E (+ or -), 123.45E (+ or -),
  - **Char (string) literals:** a g f 7 8 + \$ 3 \* #
  - **Identifiers:** strings of letters or digits, starting with a letter
    - letter(letter| digits)\*
  - **Comments:** e.g. /\* statement \*/
  - **White space:** a non-empty sequence of *blank, newline and tab*
- **Example of Non-Tokens:**
  - Comments, preprocessor directive, macros, blanks, tabs, newline etc

## ii. *lexeme*

- Each time the lexical analyzer returns a token to the parser, it has an associated lexeme
- A *lexeme* 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. i.e. the sequence of characters of a token

## iii. *pattern*

- Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token.
- A *pattern* is a description of the form that the lexemes of a token may take. i.e. *A rule that describes a set of strings*
- In the case of a **keyword** as a token, the pattern is just the sequence of characters that form the **keyword**.
- The set of words, or strings of characters, that match a given pattern is called a language.

## Example #1 (Tokens, Patterns and Lexemes)

- int MAX(int a, int b)  
this is source program

Lexeme	Token
int	keywords
MAX	identifier
(	operator
int	keyword
a	identifier
,	operator
int	keyword
b	identifier
)	operator

## Example #2 (Tokens, Patterns and Lexemes)

- gives some typical tokens, their informally described patterns, and some sample lexemes.
- To see how these concepts are used in practice, in programming language.

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

- both **printf** and **score** are lexemes matching the pattern for token id, and **" Total = %,d\n"** is a lexeme matching literal.

## Example #3 (Tokens, Patterns and Lexemes)

Token	Sample Lexemes	Pattern
keyword	if	if
id	abc, n, count,...	Letters(letters digit)*
NUMBER	3.14, 1000	numerical constant
;	;	;

## Example #4

- Consider the program

```
int main()
```

```
{
```

```
// 2 variables
```

```
int a, b;
```

```
a = 10;
```

```
return 0;
```

```
}
```

- All the valid tokens are:

```
'int' 'main' '(' ')' '{' 'int' 'a' ','  
'b' ';' 'a' '=' '10' ';' 'return' '0' ';' '  
'}'
```

## Exercise #1

- Consider the program

```
int max (x, y)  
  
int x, y;  
  
{  
  
/* find max of x and y*/  
  
{  
  
return (x>y ? X:y)  
  
}
```

Q1. How many tokens in this source code

Q2. identify lexeme, token and pattern for the  
given source code

## Exercise #2

Q1. Source code:

```
Printf ("%d has", $ X);
```

How many tokens in this source  
code

Q2. Consider source code:

```
int max (int a, int b){  
if(a>b)  
return a;  
else  
return b;  
}
```

- How many tokens in this source code?
- identify lexeme and token



## • Attributes for Tokens

- When more than one lexeme can match a pattern,
- the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
  - **For example, the pattern for token number matches both 0 and 1,**
- but it is extremely important for the code generator to know which lexeme was found in the source program.

## ✓ Attributes for Tokens(cont'd)

- LA returns to the parser not only a token name,
- but an attribute value that describes the lexeme represented by the token;
- the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.
- Normally, information about an identifier — e.g., its lexeme, its type, and
- the location at which it is first found — is kept in the symbol table.

# Example: Attributes for Tokens

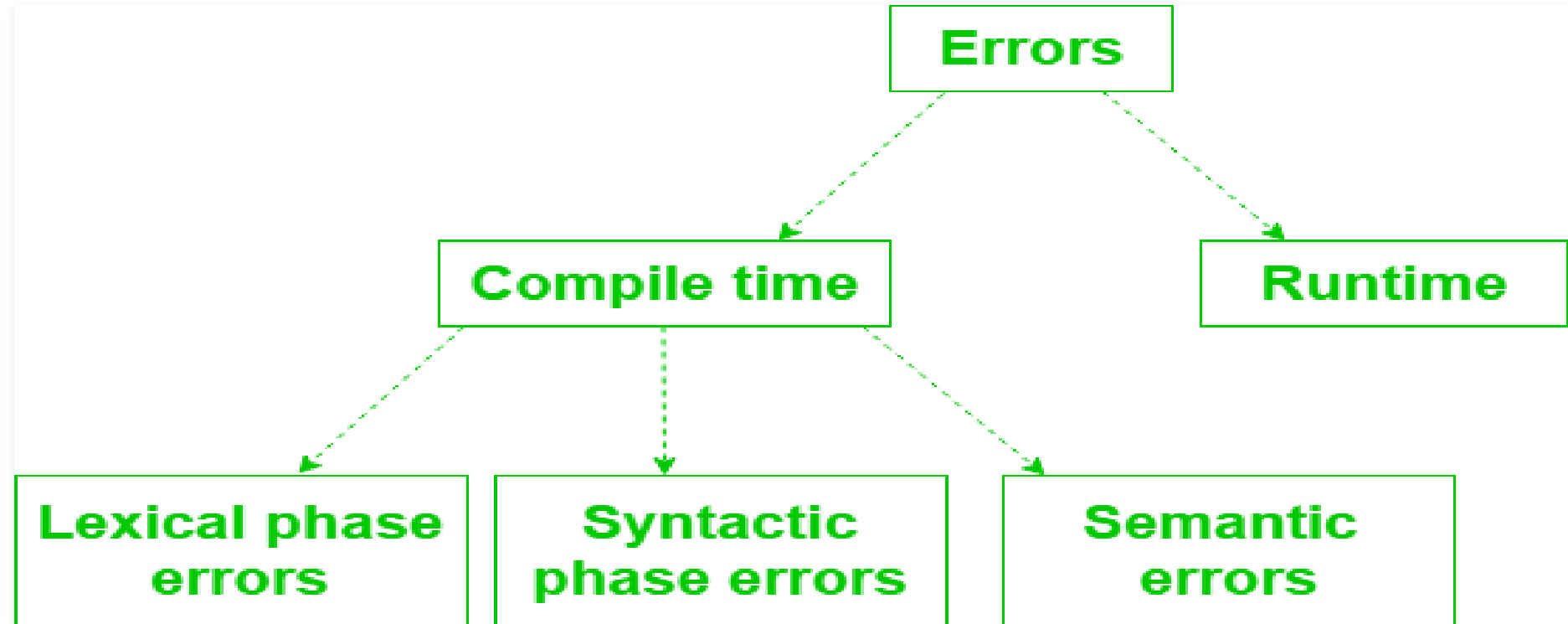
**E = M \* C \*\* 2**

token	Attribute
id	<id, pointer to symbol table entry for E>
operator	<assign-op>
id	<id, pointer to symbol table entry for M>
operator	<mult-op>
id	<id, pointer to symbol table entry for C>
operator	<exp-op>
NUM	<number, integer value 2>

# Error detection and Recovery in Compiler

- In this phase of compilation,
  - ✓ all possible errors made by the user are detected and reported to the user in form of **error messages**.
- This process of locating errors and reporting it to user is called **Error Handling process**.
- Functions of Error handler
  - *Detection,*
  - *Reporting,*
  - *Recovery*

# Classification of Errors



Compile time errors are of three types

# Lexical phase errors

- These errors are detected during the lexical analysis phase Typical lexical errors are
  - Exceeding length of identifier or numeric constants.
  - Appearance of illegal characters
  - Unmatched string

## Example:

#1 : `printf("Geeksforgeeks");$`

This is a lexical error since an illegal character \$ appears at the end of statement.

#2 : `This is a comment */`

- This is an lexical error since end of comment is present but beginning is not present.

## Example #2

- Some errors are out of power of lexical analyzer to recognize:
- for example: `fi (a == f(x)) ...`
- However it may be able to recognize errors like: `d = 2r`
- Such errors are recognized when no pattern for tokens matches a character sequence
- *a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier.*
- *Since `fi` is a valid lexeme for the token `id`, the lexical analyzer must return the token `id` to the parser and let some other phase of the compiler*
- *probably the parser in this case handle an error due to transposition of the letters.*

# Error recovery Strategies

- Successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose (change the order) two adjacent characters



# Finite automata?

- **What is a Finite automata?**
- One of the powerful models of computation which are restricted model of actual computer is called **finite automata**.
- These machines are very similar to CPU of a computer.
- They are restricted model as they lack memory.
- **Finite automata** is called **finite**
  - because number of possible states and number of letter in alphabet are both finite and automaton because the change of state is totally governed by the input.

**Q. Why abstract model are required?** to simplified a things

# Basic terms used in automaton theory

- Everything in mathematics is based on symbols. This is also true for automata theory.

❖ **Alphabets:** is defined as *a finite, non empty set of symbols*.

» Denoted by  $\Sigma$  (Sigma)

- **Example of alphabet is :**

- a set of decimal numbers.  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ,
- All Lower and upper letter alphabet  $\Sigma = \{a, b, \dots, z, A, \dots, Z\}$ .
- Binary Alphabet  $\Sigma = \{0, 1\}$
- A set of ASCII character:  $2^8 = 128$

## ❖ *Strings:*

– A *string* over an alphabet is a finite sequence of symbols from that alphabet, which is usually written next to one another and not separated by commas.

- **Example of strings:**

1. If  $\Sigma a = \{0, 1\}$  then **001001** is a string over  $\Sigma a$ .
2. If  $\Sigma b = \{a, b, , , z\}$  then **axyrpqstcd** is a string over  $\Sigma b$ .

## ❖ *Empty String:*

- The string of zero length. This is denoted by  **$\epsilon$  (epsilon)**.
- The empty string plays the role of 0 in a number system.
- The set of strings, including empty, over an alphabet  $\Sigma$  is denoted by  $\Sigma^*$ . or  $\Sigma^+ = \Sigma^* - \{\epsilon\}$
- **E.g  $\{\epsilon\}$ -means empty string of length zero**
- **$\{a\}$ -one string with whose length is 1**

## ❖ *Length of String:*

- The number of positions for symbols in the string.
- A string is generally denoted by  $w$ .
- The length of string is denoted by  $|w|$ .
- **Example:**  $|10011| = 5$

❖ *Suffix:* If  $w = xv$  for some  $x$ , then  $v$  is a suffix of  $w$ .

❖ *Prefix:* If  $w = vy$  for some  $y$ , then  $v$  is a prefix of  $w$

❖ *Reverse String:* If  $w = w_1 w_2 \dots w_n$  where each  $w_i \in \Sigma$ , the reverse of  $w$  is  $w_n w_{n-1} \dots w_1$ .

❖ *Substring:*  $z$  is a substring of  $w$  if  $z$  appears consecutively within  $w$ .

As an example, ‘deck’ is a substring of ‘abcdeckabcjkl’

## ❖ Kleene Star:

- Another language operation is the “**Kleene Star or kleene closure**” of a language  $L$ , which is *denoted by*  $\Sigma^*$ .
- $\Sigma^*$  is the set of all strings obtained by concatenating zero or more strings from  $\Sigma$
- **Example:**
- $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \dots\}$

**Note:**  $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$

**thus**  $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

## ❖ *Language:*

- Any set of strings over an alphabet  $\Sigma$  is called a **language**.
- The set of all strings, including the empty string over an alphabet  $\Sigma$  is denoted as  $\Sigma^*$ .
- If  $\Sigma$  is an alphabet, and  $L$  subset of  $\Sigma^*$ , then  $L$  is said to be language over alphabet  $\Sigma$ .
- *For example the language of all strings consisting of  $n$  0's followed by  $n$  1's for some  $n \geq 0$ :*  
 $\{\epsilon, 01, 0011, 000111, \dots\}$
- Language in set forms-  $\{w \mid \text{some logical view about } w\}$ 
  - e.g  $L = \{a^n b^n \mid n \geq 1\}$
- Infinite languages  $L$  are denoted as:

$$L = \{w \in \Sigma^* : w \text{ has property } P\}$$

## ❖ Power of an Alphabet:

- if  $\Sigma$  is an alphabet, we can express the set of all strings of a certain length from that alphabet by using the exponential notation.
- $\Sigma^k$  the set of strings length  $k$ , each of whose is in  $\Sigma$
- $\Sigma^0 = \{\epsilon\}$ , regardless of what alphabet  $\Sigma$  is. That is  **$\epsilon$  is the only string of length 0.**
- **If  $\Sigma = \{0,1\}$  then**
  - $\Sigma^1 = \{0,1\}$
  - $\Sigma^2 = \{00,01,10,11\}$
  - $\Sigma^3 = \{000,001,\dots,111\}$  and so on

## ❖ *Concatenation Language:*

- Assume a string  $x$  of length  $m$  and string  $y$  of length  $n$ , the concatenation of  $x$  and  $y$  is written  $xy$ , which is the string obtained by appending  $y$  to the end of  $x$ , as in  $x1x2 \dots xmx1y1y2 \dots yn$ .
- To concatenate a string with itself many times we use the “superscript” notation:

$$\overbrace{xx \dots x}^k = x^k$$



# Example #1

- a. Given  $\Sigma = \{a, b\}$  obtain  $\Sigma^*$ .
- b. Given examples of a finite language in  $\Sigma$ .
- c. Given  $L = \{a^n b^n : n \geq 0\}$ , check if the string aabb, aaaabbbb, abb are in the language L

- **Solution:**

- a.  $\Sigma = \{a, b\}$  therefore we have  $\Sigma^* = \{\epsilon, a, b, ab, aa, bb, aaa, bbb, \dots\}$
- b.  $\{a, aa, aab\}$  is an example of a finite language in  $\Sigma$

## Solution c

- i. aabb  $\rightarrow$  a string in L. ( $n = 2$ )
- ii. aaaabbbb  $\rightarrow$  a string in L. ( $n = 4$ )
- iii. abb  $\rightarrow$  not a string in L (since there is no satisfying this condition)

## Example #2

- **Given:**  $L = \{a^n b^{n+1} : n \geq 0\}$ . It is true that  $L = L^*$  for the following language  $L$ ?

- **Solution:**

- we know  $L^* = L^0 \cup L^1 \cup L^2 \dots$  and  $L^+ = L^1 \cup L^2 \cup L^3 \dots$
- Now for given  $L = \{a^n b^{n+1} : n \geq 0\}$ ., we have

$$\begin{aligned} L^0 &= a^0 b^{0+1} = b \\ L^1 &= a^1 b^{1+1} = ab^2 \\ L^2 &= a^2 b^{2+1} = a^2 b^3 \\ &\dots \end{aligned}$$

- Therefore, we have  $L^* = L^0 \cup L^1 \cup L^2 \dots$
- $= b \cup ab^2 \cup a^2 b^3 \dots$
- Hence it is true that  $L = L^*$

# Formal language vs Natural language

- **Formal language-**

- is a set of strings, each string composed of symbols from a finite symbol set called alphabet. (means *Define language in mathematical fashion*)
- For example, the alphabet for the sheep language is the set  $\Sigma = \{a,b,! \}$ . given a model  $m$  (such as a particular FSA).
- We can use  $L(m)$  to mean “the formal language characterized by  $m$ ”.  *$L(m) = \{baa!, baaa!.....\}$*

- **Natural language:**

- Which is a kind of language that real people speak
- Formal language is different from natural language
- We use a formal language to model part of a natural language such as parts of the phonology, morphology or syntax

# Types of Finite automata

- Two types of finite automata:
  - a. Deterministic Finite Automata (DFA)**
  - b. Non- Deterministic Finite Automata (NFA)**

## a. Deterministic Finite Automata (DFA)

- **Deterministic** means that at each point in processing there is always one unique next state (has no choice point).
- **What is DFA?**
  - It is the simplest model of computation, because we know certain inputs
  - It has a very limited memory
  - In DFA, given current state we know that the next state will be move (transition from one state to another) is uniquely determined by the current configuration.
  - If the internal state, input and contents of the storage are known, it is possible to predict the future behavior of the automaton.
  - This is said to be **DFA** otherwise it is **NFA**.

# Definition of DFA

A Deterministic Finite Automator (DFA) is a 5-tuple

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

where

$Q$	=	Finite state of “internal states”
$\Sigma$	=	Finite set of symbols called “Input alphabet”
$\delta: Q \times \Sigma \rightarrow Q$	=	Transition Function
$q_0 \in Q$	=	Initial state
$F \subseteq Q$	=	Set of Final states

## DAF description or how it's work

- The input mechanism can move only from left to right and reads exactly one symbol on each step and reads symbols of the input word in sequence.
- The transition from one internal state to another is governed by the **transition function  $\delta$** .
- A sequence of states  $q_0, q_1, q_2, \dots, q_n$ , where  $q_i \in Q$  such that  $q_0$  is the start state and  $q_i = \delta(q_{i-1}, a_i)$  for  $0 < i \leq n$ , is a run of the automaton on an input word  $w = a_1, a_2, \dots, a_n \in \Sigma$ .
- If  $\delta(q_0, a) = q_1$ , then if the DFA is in **state  $q_0$**  and the current input symbol is  **$a$** , the DFA will go into state  **$q_1$** .

## Example

$Q = \{q_0, q_1, q_2\}$ ,

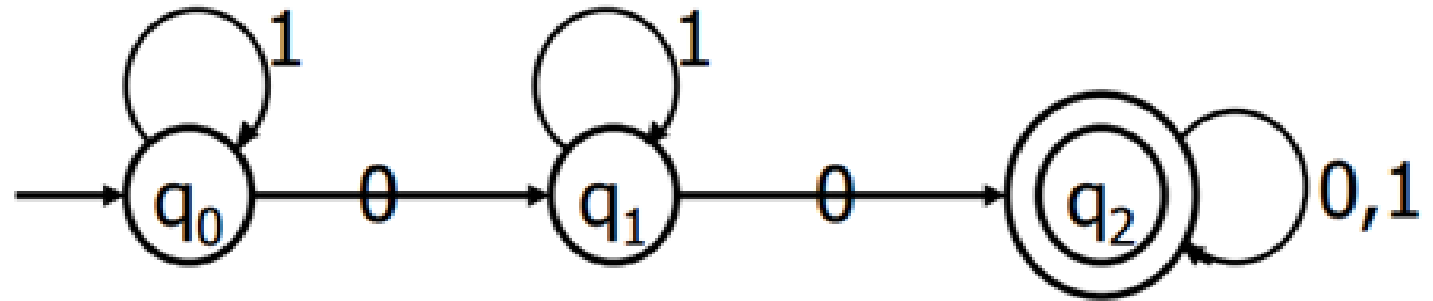
$\Sigma = \{0, 1\}$ ,

$q_0 = q_0$ ,

$F = \{q_2\}$ ,

and  $\delta$  is given by 6 equalities

- $\delta(q_0, 0) = q_1$ ,
- $\delta(q_0, 1) = q_0$ ,
- ...
- $\delta(q_2, 1) = q_2$






# DFA Representations

## a. Transition diagram

- a diagram consisting of circles to represent states and directed line segments to represent transitions between the states.
- One or more actions (outputs may be associated with each transition).
- If any state  $q$  in  $Q$  is the starting state then it is represented by

the circle with arrow as 

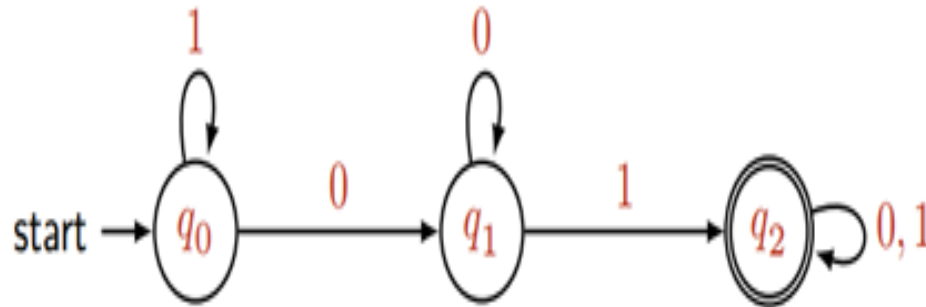
- Nodes corresponding to accepting states are marked by a double circle



# DFA Representations...

## b. Transition Table

- A **state transition table** is a table showing what state or finite state machine will move to, based on the current state and other inputs.
- Example:



Transition table:

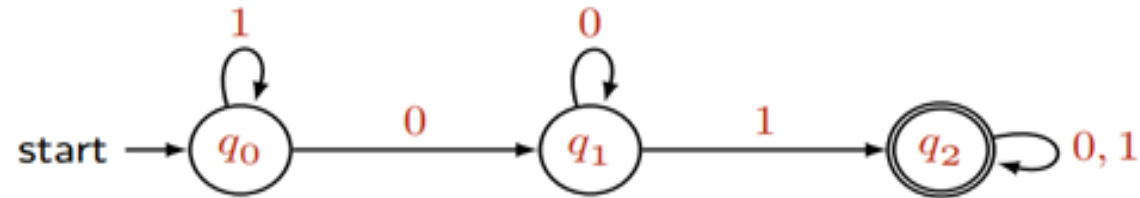
	0	1
→ $q_0$	$q_1$	$q_0$
$q_1$	$q_1$	$q_2$
* $q_2$	$q_2$	$q_2$

# DFA Representations...

## c. Transition detailed descriptions

- The description of state transitions by statement (words) is known as transition detailed descriptions.

- **Example:**



Transition table:

	0	1
$\rightarrow q_0$	$q_1$	$q_0$
$q_1$	$q_1$	$q_2$
$*q_2$	$q_2$	$q_2$

Detailed descriptions:

$Q = \{q_0, q_1, q_2\}$ ,  $\Sigma = \{0, 1\}$ ,  $q_0$  (start state),  
 $\{q_2\}$  (set of accepting states) and  $\delta : Q \times \Sigma \rightarrow Q$   
where

$$\begin{aligned}\delta(q_0, 0) &= q_1, & \delta(q_0, 1) &= q_0, \\ \delta(q_1, 0) &= q_1, & \delta(q_1, 1) &= q_2, \\ \delta(q_2, 0) &= q_2, & \delta(q_2, 1) &= q_2.\end{aligned}$$

# Example: build a sheep talk recognizer

- The sheep language contains any string from the following infinite set:

baaa!

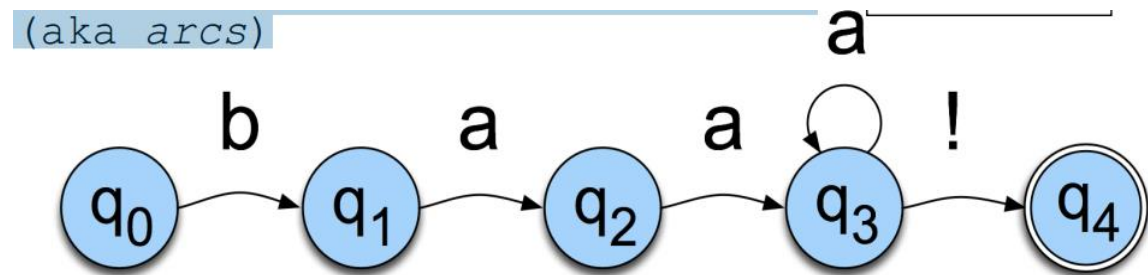
baaaa!

baaaaa! .....

- FSAs (graph notation)**

- Direct graph:**

- Finite set of vertices
- A set of directed links between pairs of vertices (aka *arcs*)



# How machine the read input strings

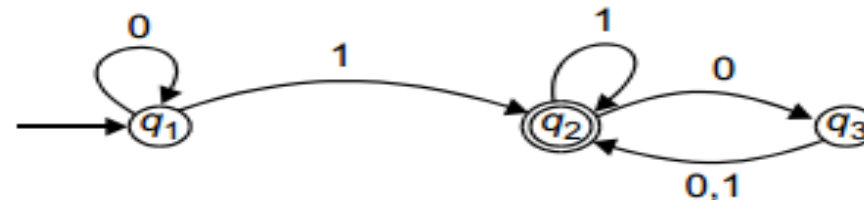
- The machine starts in the start state ( $q_0$ ), and iterates the following process:
  - Check the next letter of the input.
  - If it matches the symbol on an arc leaving the current state, then cross that arc, move to the next state, and also advance one symbol in the input.
  - If we are in the accepting **state ( $q_4$ )** when we run out of input, the machine has successfully recognized an instance of **sheeptalk**.
  - If the machine never gets to the final state, either because it runs out of input, or it gets some input that doesn't match an arc, or if it just happens to get stuck in some **non-final state**, we say the machine **REJECTS** or **fails to accept an input**.

# Example #1

- Q1. Determine the DFA schematic  $M = (Q, \Sigma, \delta, q, F)$ , where  $Q = \{q_1, q_2, q_3\}$ ,  $\Sigma = \{0, 1\}$ ,  $q_1$  is the start state,  $F = \{q_2\}$  and  $\delta$  is given by the table below. Also determine the language  $L$  recognized by the DFA.

Initial state $q$	Symbol $\sigma$	Final state $\delta(q, \sigma)$
$q_1$	0	$q_1$
$q_1$	1	$q_2$
$q_2$	0	$q_3$
$q_2$	1	$q_2$
$q_3$	0	$q_2$
$q_3$	1	$q_2$

- Solution:**



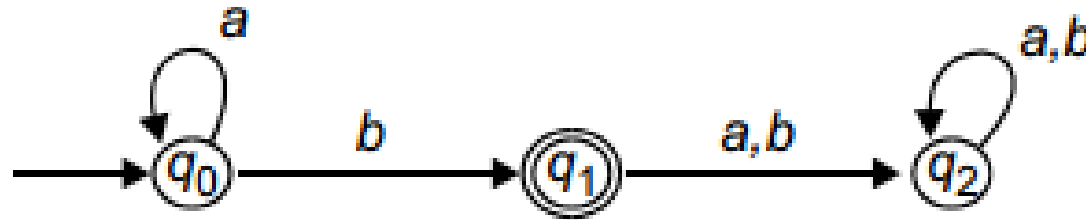
From the given table for  $\delta$ , the DFA is drawn, where  $q_2$  is the only final state. (It is to be noted that a DFA can “**accept**” a string and it can “**recognize**” a language.)

## Example #2

**Q2.** Design a DFA, the language recognized by the automata being  $L = \{a^n b : n \geq 0\}$

- **Solution:**

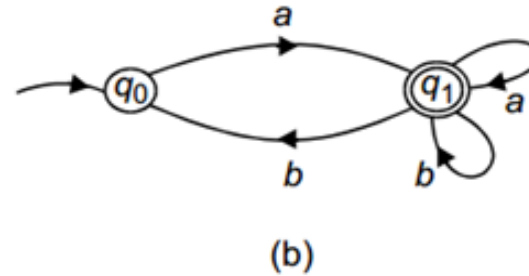
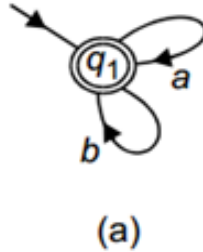
- For the given language  $L = \{a^n b : n \geq 0\}$ , the strings could be  $\{b, ab, a^2b, a^3b, \dots\}$



- Therefore the DFA accepts all strings consisting of arbitrary number of a's, followed by a single b. all other input strings are rejected.

## Example #3

- Q3. Determine the languages produced by the FA shown in figs. (a) and (b)



- Solution:**
- For  $\Sigma = \{a, b\}$ , languages generated =  $\{a,b\}^*$  ( $\epsilon$  will be accepted when initial state equal final state)
- For  $\Sigma = \{a, b\}$ , languages generated =  $\{a,b\}^+$  ( $\epsilon$  is not accepted)



# Exercise #1

**Q1.** Design a DFA,  $M$  which accepts the language  $L(M) = \{w \in (a, b)^* : w \text{ does not contain three consecutive } b\text{'s}\}$ .

- Let  $M = (Q, \Sigma, \delta, q, F)$  where  $Q = \{q_0, q_1, q_2, q_3\}$ ,  $\Sigma = \{a, b\}$ ,  $q_0$  is the initial state,  $F = \{q_0, q_1, q_2\}$  are final state and  $\delta$  is defined as follows:

Initial state $q$	Symbol $\sigma$	Final state $\delta(q, \sigma)$
$q_0$	$a$	$q_0$
$q_0$	$b$	$q_1$
$q_1$	$a$	$q_0$
$q_1$	$b$	$q_2$
$q_2$	$a$	$q_0$
$q_2$	$b$	$q_3$
$q_3$	$a$	$q_3$
$q_3$	$b$	$q_3$

## Exercise #2

Q2. Given  $\Sigma = \{a, b\}$ , construct a DFA that shall recognize the language  $L = \{b^m a b^n\}$ :  
 $m, n > 0\}$

Q3. Given  $\Sigma = \{a, b\}$ , construct a DFA that shall recognize the language  $L = \{a^m b^n\}$ :  
 $m, n > 0\}$

Q4. Determine the FA with the

- a. Set of strings beginning with an 'a'
- b. Set of strings beginning with an 'a' and ending with 'b'
- c. Set of strings having 'aaa' as a subword

## b. Non-Deterministic Finite Automata (NFA)

- A NFA can be different from a deterministic one in that for any input symbol,
- Non deterministic one can transit to more than one states.
- In NFA, given the current state there could be multiple next state
- **epsilon ( $\epsilon$ ) transition.**
- **The questions is how do we choice to which state go to the second input because it is multiple state?**
  - In NFA, the next state may be chosen at random or parallel.

# Definition of NFA

- A **NFA** is defined by a 5-tuple  $M = (Q, \Sigma, \delta, q_0, F)$  where  $Q, \Sigma, \delta, q_0, F$  are defined as follows:
  - $Q$  = Finite set of internal states
  - $\Sigma$  = Finite set of symbols called “Input alphabet”
  - $q_0 \in Q$  is the Initial states
  - $F \subseteq Q$  is a set of Final states
  - $\delta = Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$  A transition function, which is a total function from  $Q \times \Sigma$  to  $2^Q$
  - $\delta: (Q \times \Sigma) \rightarrow P(Q)$ .  $P(Q)$  is the power set of  $Q$ , the set of all subsets of  $Q$
  - $\delta(q,s)$  -The set of all states  $p$  such that there is a transition labeled  $s$  from  $q$  to  $p$   $\delta(q,s)$  is a function from  $Q \times S$  to  $P(Q)$  (but not to  $Q$ )

# Difference between NFA and DFA

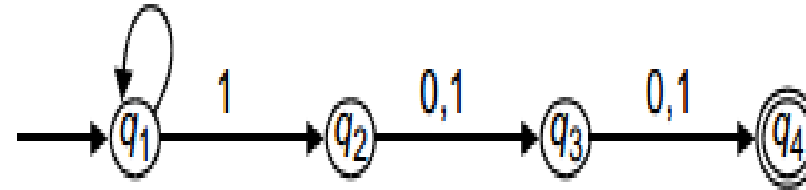
- NFA differs from DFA in that;
  - The range of  $\delta$  in NFA is in the power set  $2^Q$ .
  - A string is accepted by an NFA if there is some sequence of possible moves that will put the machine in the final state at the end of the string.
  - In DFA, For a given state on a given input we reach to a deterministic and unique state.
  - In NFA or NDFA we may lead to more than one state for a given input.
  - In NFA, Empty string can label transitions.
  - *We need to convert NFA to DFA for designing a compiler.*

## NFA: Example #1:

- Obtain an NFA for a language consisting of all strings over  $\{0,1\}$  containing a 1 in the third position from the end.

- Solution:**

- $q_1, q_2, q_3$  are initial states
- $q_4$  is the final state.



- Please note that this is an NFA as

$$\delta(q_2, 0) = q_3 \text{ and } \delta(q_2, 1) = q_3.$$

## NFA: Example #2

- Sketch the NFA state diagram for

$$M = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_3\})$$

with the state table as given below.

$\delta$	0	1
$q_0$	$q_0, q_1$	$q_0, q_2$
$q_1$	$q_3$	$\emptyset$
$q_2$	$\emptyset$	$q_3$
$q_3$	$q_3$	$q_3$

### Solution:

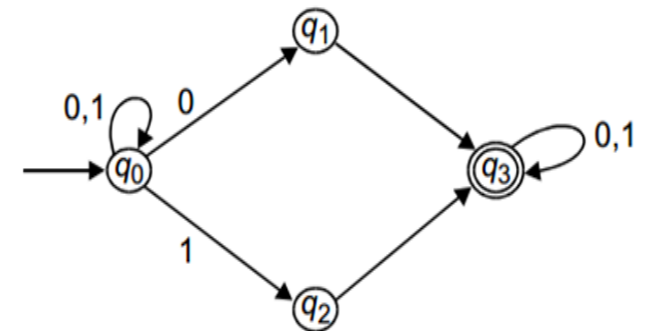
The NFA states are  $q_0, q_1, q_2$  and  $q_3$ .

$$\delta(q_0, 0) = \{q_0, q_1\} \quad \delta(q_0, 1) = \{q_0, q_2\}$$

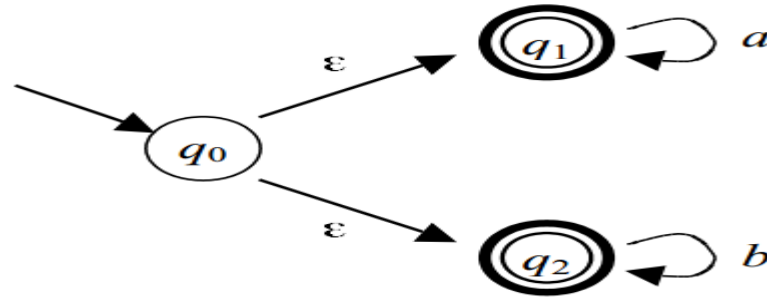
$$\delta(q_1, 0) = \{q_3\} \quad \delta(q_2, 1) = \{q_3\}$$

$$\delta(q_3, 0) = \{q_3\} \quad \delta(q_3, 1) = \{q_3\} .$$

The NFA is as shown below.



# $\epsilon$ - Transitions to Accepting States

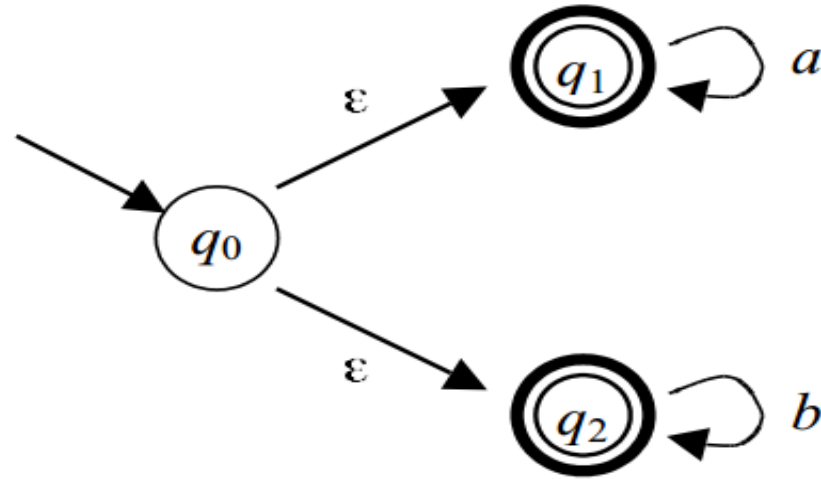


An  $\epsilon$ -transition can be made at any time

- For example, there are three sequences on the empty string
  - No moves, ending in  $q_0$ , rejecting
  - From  $q_0$  to  $q_1$ , accepting
  - From  $q_0$  to  $q_2$ , accepting
- Any state with an  $\epsilon$ -transition to an accepting state ends up working like an accepting state too

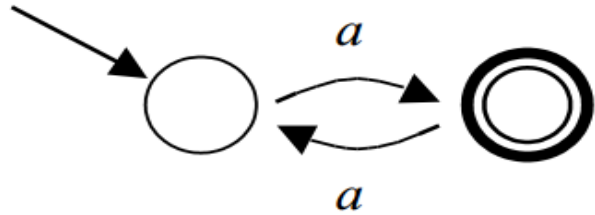


## $\epsilon$ -transitions For NFA Combining

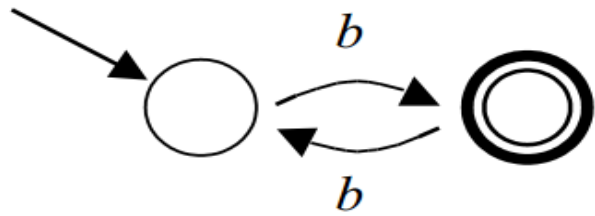


- $\epsilon$ -transitions are useful for combining smaller automata into larger ones
- This machine combines a machine for  $\{a\}^*$  and a machine for  $\{b\}^*$
- It uses an  $\epsilon$ -transition at the start to achieve the union of the two languages

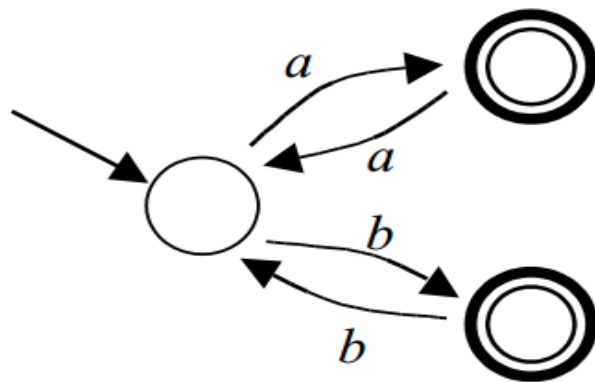
## Example: Incorrect Union



$$A = \{a^n \mid n \text{ is odd}\}$$



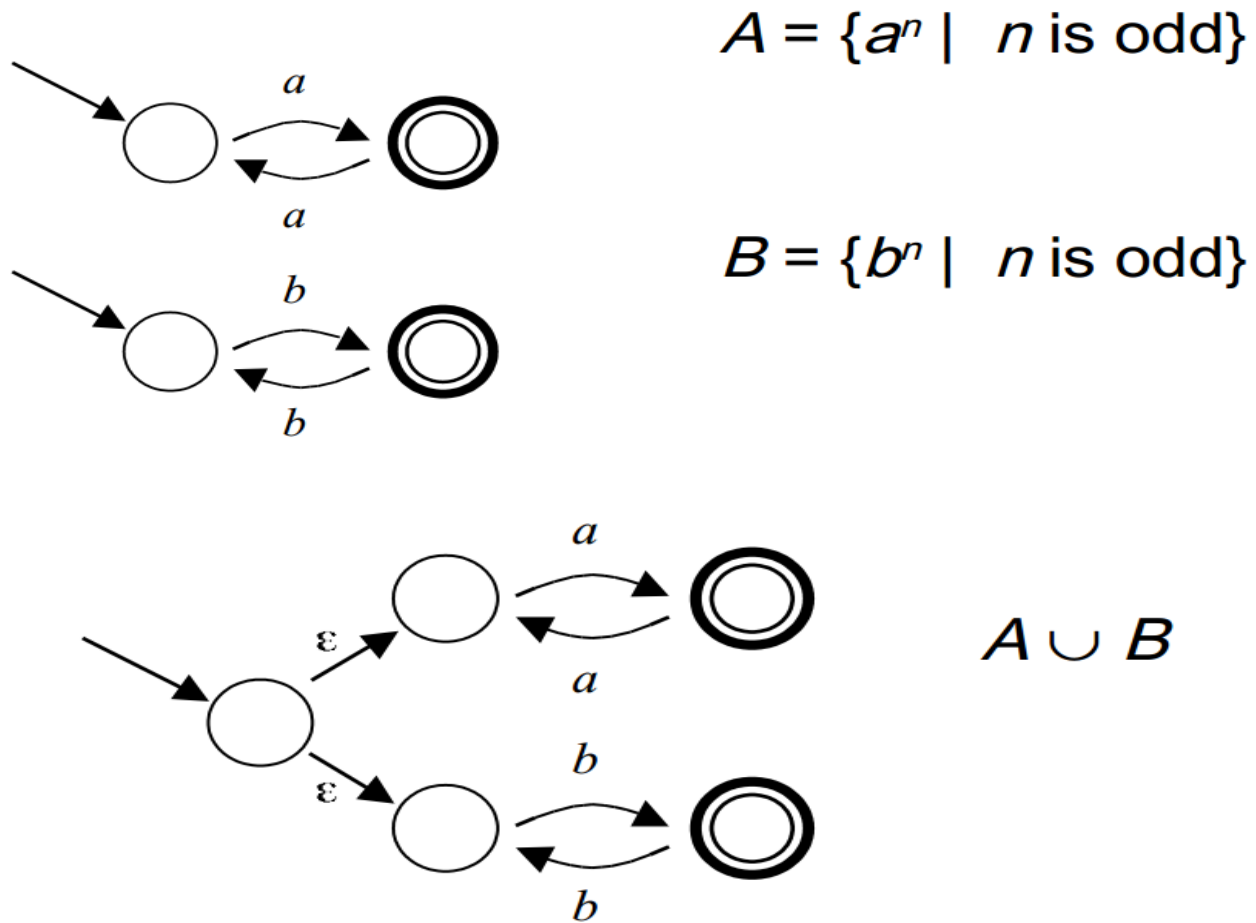
$$B = \{b^n \mid n \text{ is odd}\}$$



$$A \cup B ?$$

No: this NFA accepts *aab*

# Example: Correct Union



# NDF: Exercise #1

- Design a NFA for the language  $L=(ab \cup aba)^*$
- Draw the state diagram for NFA accepting language  $L=(ab)^*(ba)^* \cup aa^*$
- Design a NFA for language  $L=\{0101^n \cup 0100 \mid n \geq 0\}$
- Design a NFA to accept strings with a's and b's such that strings end with '*aa*'

# EQUIVALENCE OF NFA AND DFA

- **Definition:**
  - Two finite accepters  $M1$  and  $M2$  are equivalent iff
$$L(M1) = L(M2)$$
    - i.e., if both accept the same language.
    - Both DFA and NFA recognize the same class of languages.
    - It is important to note that every NFA has an equivalent DFA.
- Let us illustrate the conversion of NFA to DFA through an example.

## Example #1

- Determine a deterministic Finite State Automaton from the given Nondeterministic FSA.

$$M = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$$

with the state table diagram for  $\delta$  given below.

$\delta$	$a$	$b$
$q_0$	$\{q_0, q_1\}$	$\{q_1\}$
$q_1$	$\emptyset$	$\{q_0, q_1\}$

# Solution

Let  $M' = (Q', \Sigma, \delta', q'_0, F')$  be a determine. Finite state automaton (DFA), where

$$Q' = \{[q_0], [q_1], [q_0, q_1], [\emptyset]\},$$

$$q'_0 = [q_0]$$

and  $F' = \{[q_1], [q_0, q_1]\}$

Please remember that  $[ \quad ]$  denotes a single state. Let us now proceed to determine  $\delta'$  to be defined for the DFA.

$\delta'$	$a$	$b$
$[q_0]$	$[q_0, q_1]$	$[q_1]$
$[q_1]$	$\emptyset$	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$
$\emptyset$	$\emptyset$	$\emptyset$

# Solution....

It is to be noted that

since

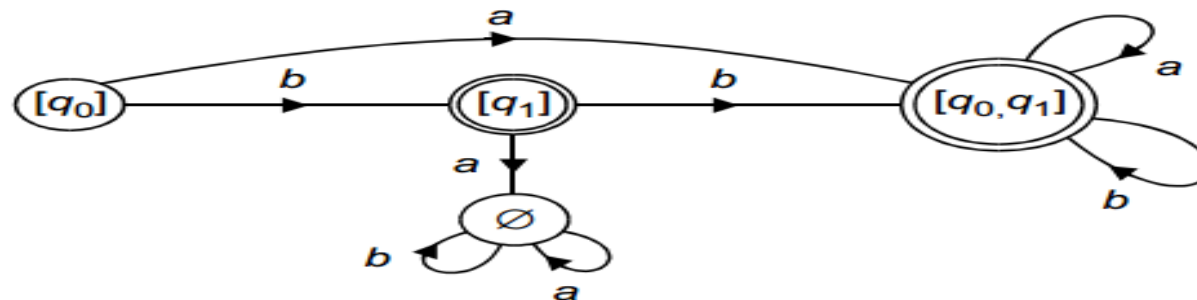
$$\begin{aligned}\delta'([q_0, q_1], a) &= [q_0, q_1] \\ \delta'([q_0, q_1], a) &= \delta(q_0, a) \cup \delta(q_1, a) \\ &= \{q_0, q_1\} \cup \emptyset \\ &= \{q_1, q_1\}\end{aligned}$$

and

since

$$\begin{aligned}\delta'([q_0, q_1], b) &= [q_0, q_1] \\ \delta([q_0, q_1], b) &= \delta(q_0, b) \cup \delta(q_1, b) \\ &= \{q_1\} \cup \{q_0, q_1\} \\ &= \{q_0, q_1\}\end{aligned}$$

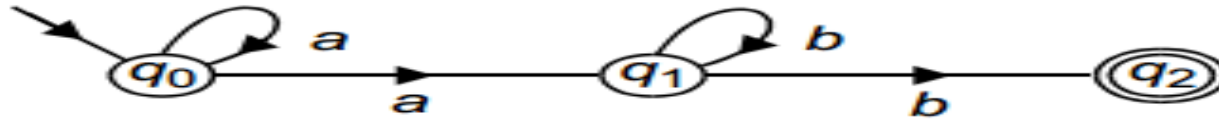
Here any subset containing  $q_1$  is the final state in DFA. This is shown as below.





## Example #2

- Given the NDA as shown in Fig. (a), with  $\delta$  as shown in Fig. (b)



**Fig. (a)**

	<i>a</i>	<i>b</i>
<i>q</i> <sub>0</sub>	{ <i>q</i> <sub>0</sub> , <i>q</i> <sub>1</sub> }	∅
<i>q</i> <sub>1</sub>	∅	{ <i>q</i> <sub>1</sub> , <i>q</i> <sub>2</sub> }
<i>q</i> <sub>2</sub>	∅	∅

**Fig. (b)**

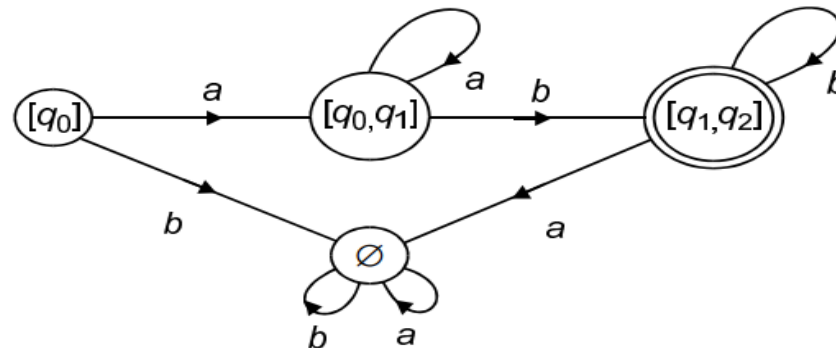
**Determine the equivalent DFA for the above given NFA**

# Solution

- Conversion of NDA to DFA is done through subset construction as shown in the State table diagram below.

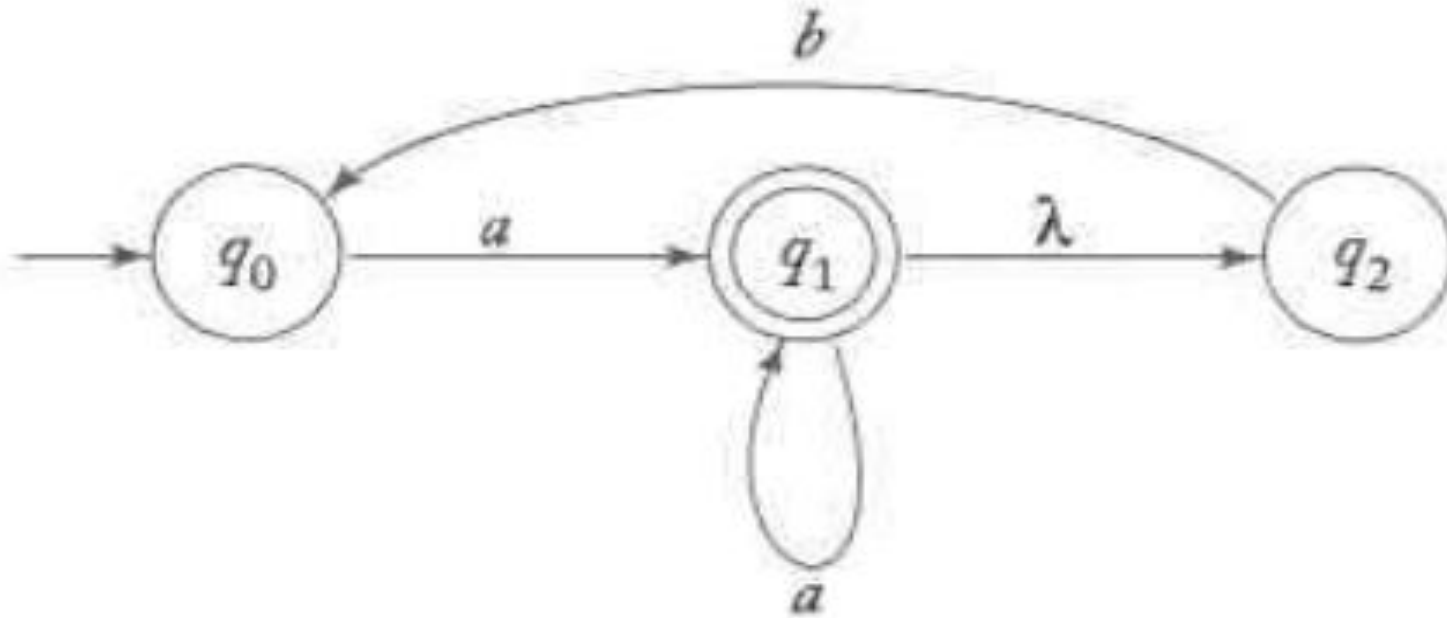
	<i>a</i>	<i>b</i>
$[q_0]$	$[q_0, q_1]$	$\emptyset$
$[q_0, q_1]$	$[q_0, q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$\emptyset$	$[q_1, q_2]$
$\emptyset$	$\emptyset$	$\emptyset$

- The corresponding DFA is shown below. Please note that here any subset containing  $q_2$  is the final state.



## Exercise #2

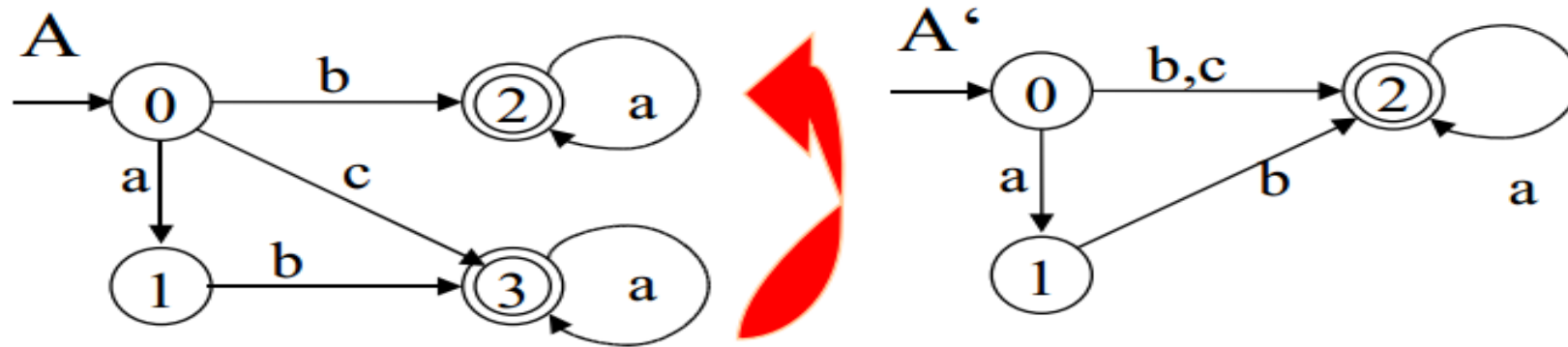
- Find the equivalence DFA for given NFA



# Reducing number of states in Finite Automata

- **DFA Minimization**

- Can we transform a large automaton into a smaller one (provided a smaller one exists)?



- If  $A$  is a DFA, is there an algorithm for constructing an equivalent minimal automaton  $A_{\min}$  from  $A$ ?

# For Example

- $A$  is *equivalent* to  $A'$  i.e.,  $L(A) = L(A')$   
 $A'$  is *smaller* than  $A$  i.e.,  $|A| > |A'|$
- $A$  can be transformed to  $A'$ :
  - States 2 and 3 in  $A$  “*do the same job*”: once  $A$  is in state 2 or 3, it *accepts the same suffix string*. Such states are called *equivalent*.
  - Thus, we can eliminate state 3 without changing the language of  $A$ , by *redirecting* all arcs leading to 3 to 2, instead.

# DFA Minimization using Equivalence Theorem

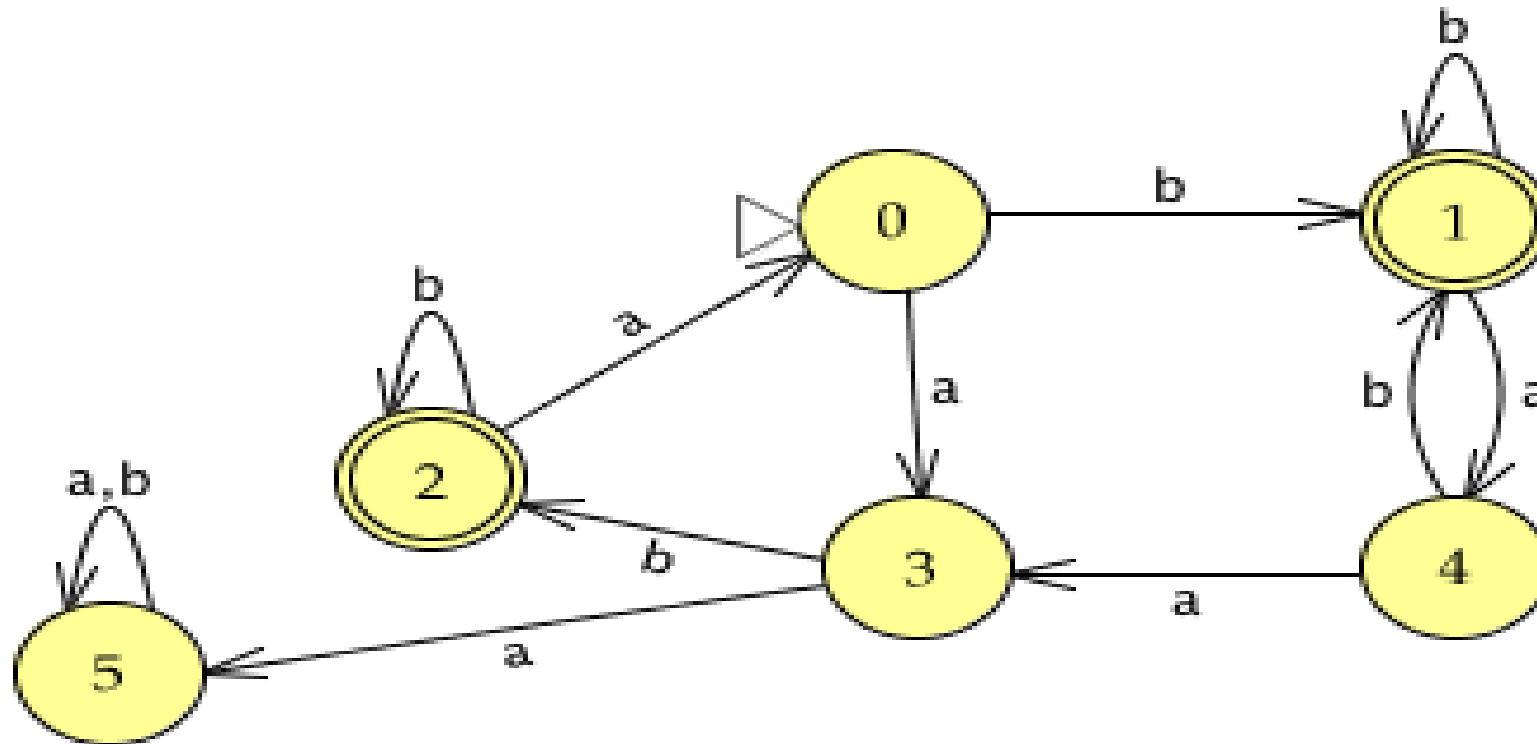
- If A and B are two states in a DFA, we can combine these two states into  $\{A, B\}$  if they are not distinguishable.
- Two states are distinguishable, if there is at least one string S, such that one of  $\delta(A, S)$  and  $\delta(B, S)$  is accepting and another is not accepting.
- Hence, a DFA is minimal if and only if all the states are distinguishable.

# DFA Minimization :Algorithm

- **Step 1** All the states **Q** are divided in two partitions - **final states** and **non-final states** and are denoted by **P<sub>0</sub>**. All the states in a partition are 0th equivalent. Take a counter **k** and initialize it with 0.
- **Step 2** Increment k by 1. For each partition in **P<sub>k</sub>**, divide the states in **P<sub>k</sub>** into two partitions if they are k-distinguishable.
- Two states within this partition **X** and **Y** are k distinguishable if there is an input **S** such that  $\delta(A,S)$  and  $\delta(B,S)$  are  $k-1$ -distinguishable.
- **Step 3** If  $P_k \neq P_{k-1}$ , repeat Step 2, otherwise go to Step 4.
- **Step 4** Combine  $k^{\text{th}}$  equivalent sets and make them the new states of the reduced DFA.

# DFA Minimization :Example

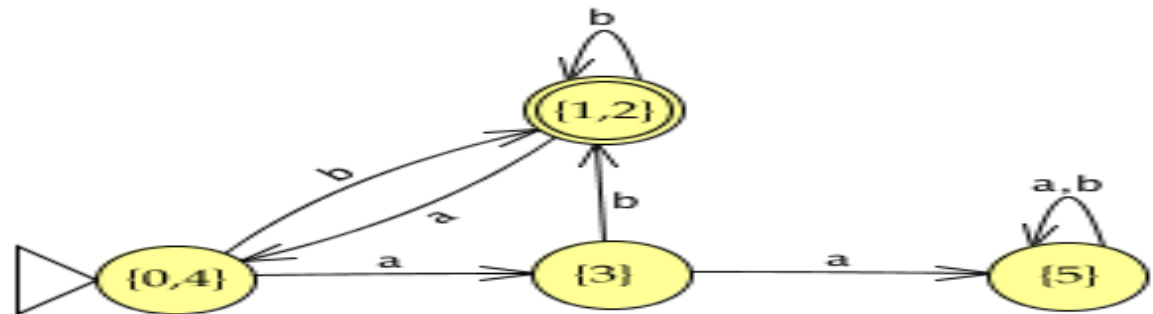
- Let us consider the following DFA -
- We will use the *Partitioning Method*





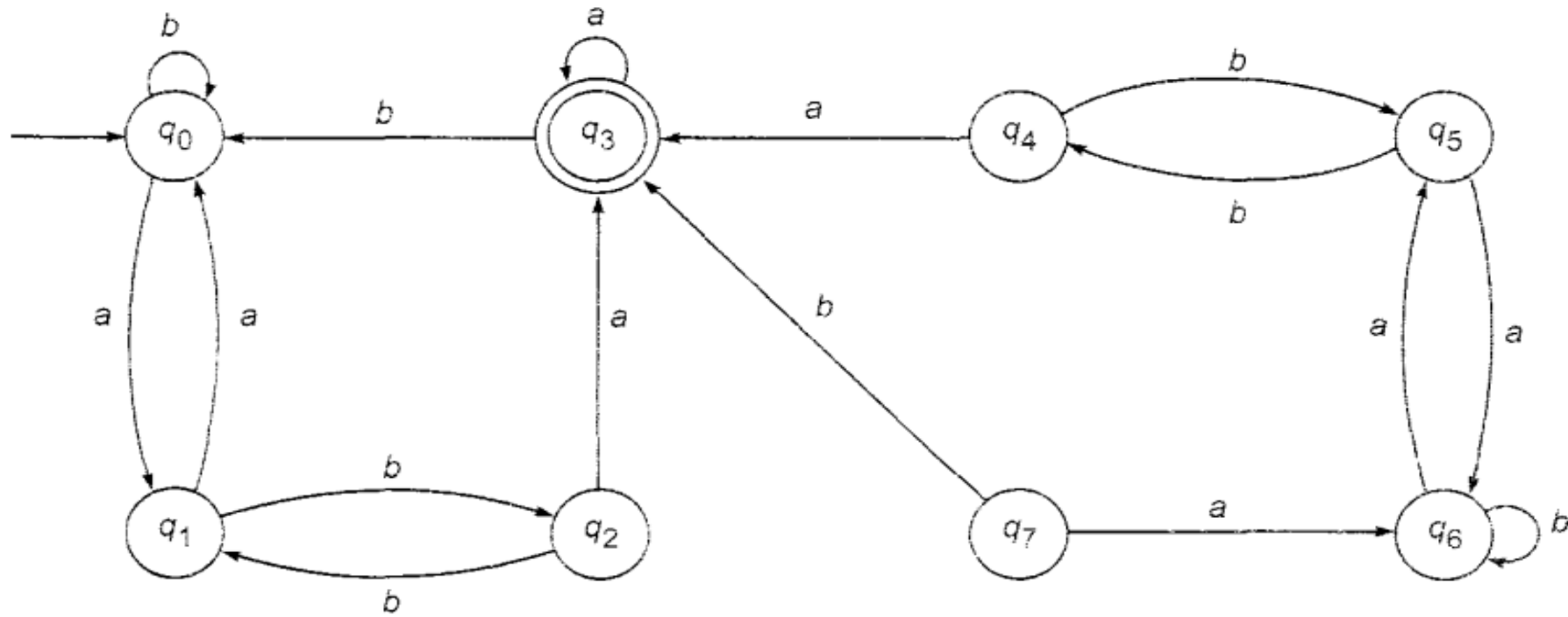
# Solution

- Start by separating final and non-final states:  $\{1,2\}, \{0,3,4,5\}$
- All of  $0,3,4$  go to final states by a b transition,  $5$  does not ( $5$  is a dead state). So we have 3 groups:  $\{1,2\}, \{0,3,4\}, \{5\}$
- An a- transition takes  $3$  to  $5$  and  $0,4$  to  $3$ . So we split  $\{0,3,4\}$  into two groups, getting:  $\{1,2\}, \{0,4\}, \{3\}, \{5\}$
- **Solution:**
- We cannot make any further sub-partitions and so the minimal DFA is:
- 



# Exercise: #3

Construct the minimum state automaton equivalent to the transition diagram given by Fig. 3.14.



**Fig. 3.14** Finite automaton of Example 3.14.

## 2.4 Specification of Tokens

- In theory of compilation regular expressions are an important notation for specifying *lexeme patterns*.
- **Regular expressions** are means for specifying regular languages.
  - Because, tokens are specified with the help of RE
- **Example:** Suppose we wanted to describe the set of valid C identifiers
  - **Letter(letter | digit)\***
- Each regular expression is a pattern specifying the form of strings
  - Because, they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.

## ■ Strings and Languages

- An **alphabet** is any finite set of symbols such as *letters, digits, and punctuation*.
  - The set **{0,1}** is the binary alphabet
  - If x and y are strings, then the concatenation of x and y is also string, denoted **xy**,
  - For example, if x = dog and y = house, then xy = **doghouse**.
  - A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
- In language theory, the terms "**sentence**" and "**word**" are often used as synonyms for "string."
- **|s|** represents the length of a string s,      Ex: banana is a string of length 6

## ■ Strings and Languages (cont'd)

- The **empty string**, is the string of **length zero**.
- The **empty string** is the identity under concatenation; that is, for any string  $s$ ,  $\epsilon S = S\epsilon = s$ .
- A **language** is any countable set of strings over some fixed alphabet.
- *Def. Let  $\Sigma$  be a set of characters. A language over is a set of strings of characters drawn from  $\Sigma$ .*
- Let  $L = \{A, \dots, Z\}$ , then  $["A", "B", "C", "BF" \dots, "ABZ", \dots]$  is consider the language defined by  $L$
- Abstract languages like  $\emptyset$ , the **empty set**, or  $\{\epsilon\}$ , the set containing only the **empty string**, are languages under this definition.

## ■ Terms for Parts of Strings

■ The following string-related terms are commonly used:

1. A **prefix** of string  $s$  is any string obtained by removing zero or more symbols from the beginning of  $s$ . For example: prefixes of **banana**
  - $\epsilon, a, ba, ban, bana, babab, banana$ .
2. A **suffix** of string  $s$  is any string obtained by removing zero or more symbols from the end of  $s$ . For example: suffix of banana
  - $banana, banan, baba, ban, ba, b, \epsilon$
3. A **substring** of  $s$  is obtained by deleting any prefix and any suffix from  $s$ .
  - For instance, **banana, nan, and  $\epsilon$  are substrings of banana.**
4. A **subsequence** of  $s$  is any string formed by deleting zero or more not necessarily consecutive positions of  $s$ .
  - For example: **baan is a subsequence of banana.**

# Operations on Languages

OPERATION	DEFINITION AND NOTATION
<i>Union of <math>L</math> and <math>M</math></i>	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
<i>Concatenation of <math>L</math> and <math>M</math></i>	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
<i>Kleene closure of <math>L</math></i>	$L^* = \bigcup_{i=0}^{\infty} L^i$
<i>Positive closure of <math>L</math></i>	$L^+ = \bigcup_{i=1}^{\infty} L^i$

## Example:

- Let **L** be the set of letters **{A, B, ..., Z, a, b, ..., z}** and
- let **D** be the set of digits **{0,1,.. .9}**.
- L and D are, respectively, the *alphabets of uppercase and lowercase letters and of digits*.
- other languages can be constructed from **L** and **D**, using the operators illustrated above as shown next slide.

## ■ Operations on Languages (cont.)

1. **LUD** is the set of letters and digits –
2. **LD** is set of strings consisting of a letter followed by a digit  
*Ex: A1, a1, B0, etc*
3. **L<sup>4</sup>** is the set of all 4-letter strings. (*ex: aaba, bcef*)
4. **L<sup>\*</sup>** is the set of all strings of letters, including  $\epsilon$ , the empty string.
5. **L(LUD)<sup>\*</sup>** is the set of all strings of letters and digits beginning with a letter.
6. **D<sup>+</sup>** is the set of all strings of one or more digits.



## ■ Regular Expressions (RE)

- A **RE** is a special text string for describing a search pattern.
- **RE** were designed to represent regular languages with a mathematical tool,
  - a tool built from a set of **primitives** and **operations**.
- This representation involves a combination of strings of symbols from some alphabet  $\Sigma$ , parentheses.
- and the operators **+**, **·** and **\***. **Where** (union, concatenation and Kleene star).

# Building Regular Expressions

- Assume that  $\Sigma = \{a, b\}$ 
  - **Zero or more:**  $a^*$  means “zero or more  $a$ ’s”, To say “zero or more  $a$ ’s,”
  - Example:  $\{\epsilon, , , ab abab \dots\}$  you need to say  $(ab)^*$ .
  - **One or more:** Since  $a^+$  means “one or more  $a$ ’s”, you can use  $aa^*$  (or equivalently  $a^*a$ ) to mean “one or more  $a$ ’s”.
  - **Zero or one:**  $(a)?$  It can be described as an optional ‘ $a$ ’ with  $(a + \epsilon)$ .

## Example #1

- A regular expression is obtained from the symbol  $\{0,1\}$ , empty string  $\epsilon$ , and empty-set  $\emptyset$  perform the operations **+**, **·** and **\***

$0 + 1$  represents the set  $\{0, 1\}$

$1$  represents the set  $\{1\}$

$0$  represents the set  $\{0\}$

$(0 + 1) 1$  represents the set  $\{01, 11\}$

$(a + b) \cdot (b + c)$  represents the set  $\{ab, bb, ac, bc\}$

$(0 + 1)^* = \epsilon + (0 + 1) + (0 + 1)(0 + 1) + \dots = \Sigma^*$

$(0 + 1)^+ = (0 + 1)(0 + 1)^* = \Sigma^+ = \Sigma^* - \{\epsilon\}$

## ■ Languages defined by Regular Expressions

- There is a very simple correspondence between regular expressions and the languages they denote:

Regular expression	$L$ (Regular Expression)
$x$ , for each $x \in \Sigma$	$\{x\}$
$\lambda$	$\{\lambda\}$
$\emptyset$	$\{ \}$
$(r_1)$	$L(r_1)$
$r_1^*$	$(L(r_1))^*$
$r_1 r_2$	$L(r_1)L(r_2)$
$r_1 + r_2$	$L(r_1) \cup L(r_2)$

## Example #2

- Let  $\Sigma = \{a, b\}$ 
  - The **RE**  $a|b$  denotes the set  $\{a, b\}$
  - The **RE**  $(a|b)(a|b)$  denotes  $\{aa, ab, ba, bb\}$
  - The **RE**  $a^*$  denotes the set of all strings  $\{\epsilon, a, aa, aaa, \dots\}$
  - The **RE**  $(a|b)^*$  denotes the set of all strings containing  $\epsilon$  and all strings of a's and b's
  - The **RE**  $a|a^*b$  denotes the set containing the string  $a$  and all strings consisting of zero or more a's followed by a  $b$

# Precedence and Associativity

- \*, concatenation, and | are left associative
  - ✓ \* has the highest precedence
  - ✓ . Concatenation has the second highest precedence
  - ✓ | has the lowest precedence

## ■ Regular Definitions

- How to specify tokens?
- Regular definitions
  - Let  $\mathbf{r_i}$  be a regular expression and  $\mathbf{d_i}$  be a distinct name
  - Regular definition is a sequence of definitions of the form

$$\mathbf{d_1} \rightarrow \mathbf{r_1}$$

$$\mathbf{d_2} \rightarrow \mathbf{r_2}$$

.....

$$\mathbf{d_n} \rightarrow \mathbf{r_n}$$

Where each  $\mathbf{r_i}$  is a regular expression over  $\Sigma \cup \{\mathbf{d_1}, \mathbf{d_2}, \dots, \mathbf{d_{i-1}}\}$

**Ex #1:** ■ C identifiers are strings of **letters, digits, and underscores**.

- The regular definition for the language of C identifiers.

– *Letter*  $\rightarrow A | B | C | \dots | Z / a | b | \dots | z$  -

– *digit*  $\rightarrow 0 | 1 | 2 | \dots | 9$

*id*  $\rightarrow letter( letter / digit )^*$

**Ex #2:**

- Unsigned numbers (**integer or floating point**) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. in pascal
- The regular definition:
  - *digit*  $\rightarrow 0 | 1 | 2 | \dots | 9$
  - *digits*  $\rightarrow digit digit^*$
  - *optionalFraction*  $\rightarrow .digits / \epsilon$
  - *optionalExponent*  $\rightarrow ( E( + | - / \epsilon ) digits ) / \epsilon$
  - *number*  $\rightarrow digits optionalFraction optionalExponent$



## Example #3:

- My fax number
  - 91-(512)-259-7586
  - $\Sigma = \text{digits} \cup \{-, (, )\}$
  - Country  $\rightarrow \text{digit}^+$
  - Area  $\rightarrow \text{'(' digit}^+ \text{'})'}$
  - Exchange  $\rightarrow \text{digit}^+$
  - Phone  $\rightarrow \text{digit}^+$
  - Number  $\rightarrow \text{country '-' area '-'}$   
exchange '-' phone

## Example #4:

- My email address : `ska@iitk.ac.in`
  - $\Sigma = \text{letter} \cup \{ @, . \}$
  - Letter  $\rightarrow a | b | \dots | z | A | B | \dots | Z$
  - Name  $\rightarrow \text{letter}^+$
  - Address  $\rightarrow \text{name '@' name '.' name '.'}$   
name

## 2.5 Recognition of Tokens

- In the previous section we learned how to **express patterns using regular expressions**.
- Now, we study how to take 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.
- Given the grammar of branching statement:

$$\begin{array}{lcl} stmt & \rightarrow & \text{if } expr \text{ then } stmt \\ & | & \text{if } expr \text{ then } stmt \text{ else } stmt \\ & | & \epsilon \\ expr & \rightarrow & term \text{ relop } term \\ & | & term \\ term & \rightarrow & id \\ & | & number \end{array}$$

## Recognition of Tokens (cont'd)

- The patterns for the given tokens:

<i>digit</i>	→	[0-9]
<i>digits</i>	→	<i>digit</i> <sup>+</sup>
<i>number</i>	→	<i>digits</i> ( . <i>digits</i> )? ( E [+ -]? <i>digits</i> )?
<i>letter</i>	→	[A-Za-z]
<i>id</i>	→	<i>letter</i> ( <i>letter</i>   <i>digit</i> )*
<i>if</i>	→	<b>if</b>
<i>then</i>	→	<b>then</b>
<i>else</i>	→	<b>else</b>
<i>relop</i>	→	<   >   <=   >=   =   <>

*r*? is equivalent to *r*| $\epsilon$

## Recognition of Tokens (cont'd)

- The **terminals** of the grammar, which are **if**, **then**, **else**, **relop**, **id**, and **number**, are the names of tokens as used by the **lexical analyzer**.
- The lexical analyzer also has the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

$$ws \rightarrow ( \text{blank} \mid \text{tab} \mid \text{newline} )^+$$

- *Tokens are specified with the help of regular expression, then we recognize with transition diagram.*

# Recognition of Tokens: Transition Diagram

- Tokens are recognized with transition diagram:

a. Recognition of Relational operators Ex :

**RELOP** :     < | <= | = | <> | > | >=

b. Recognition of identifier: **ID** = letter(letter | digit) \*

c. Recognition of numbers (integer | floating points)

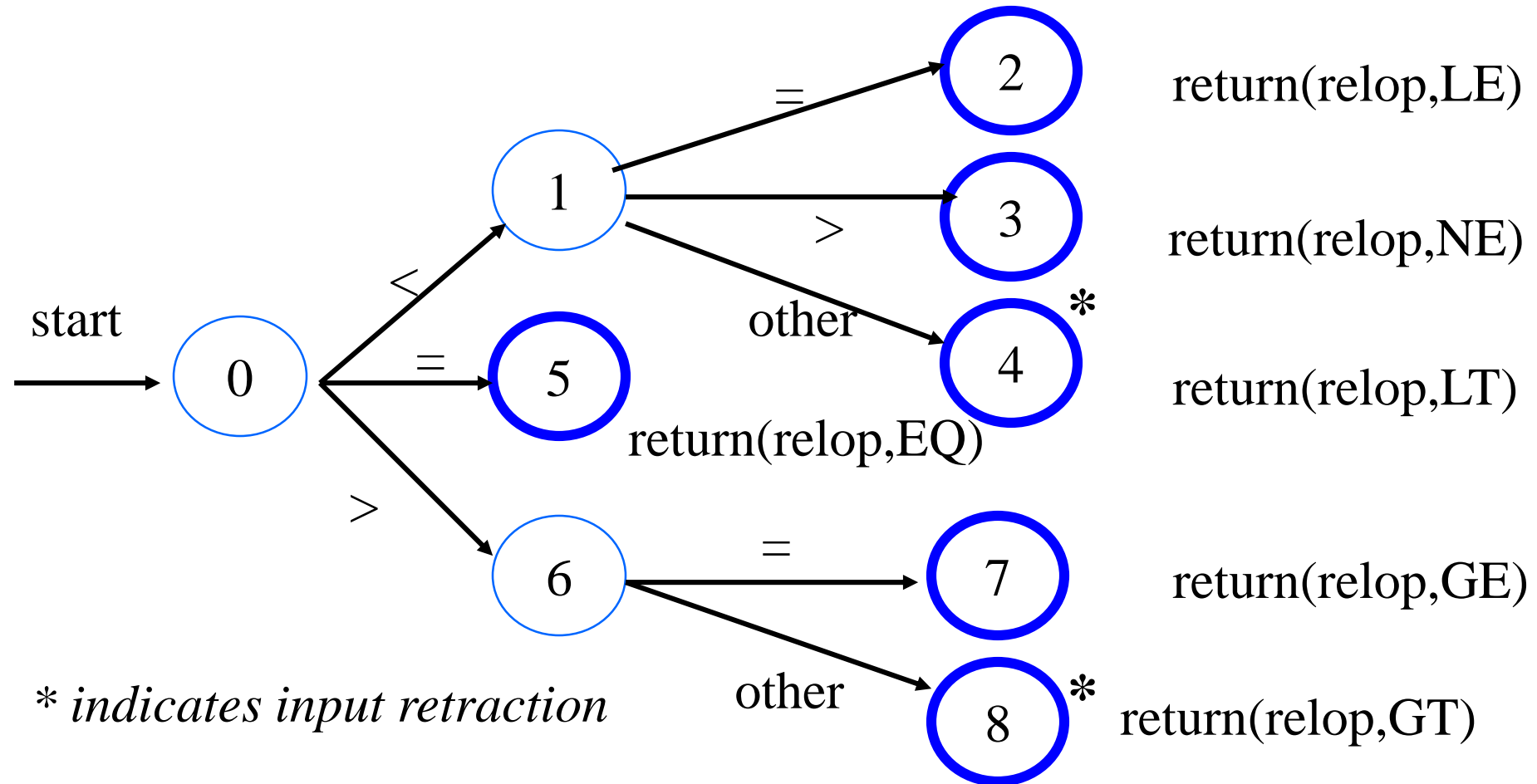
d. Recognition of delimiter or word space EX: *blank, tap, newline*

e. Recognition of keywords such as **if, else, for**

$ws \rightarrow (blank \mid tab \mid newline)^+$

## a. Recognition of Relational operators :Transition Diagram

Ex :RELOP = < | <= | = | <> | > | >=



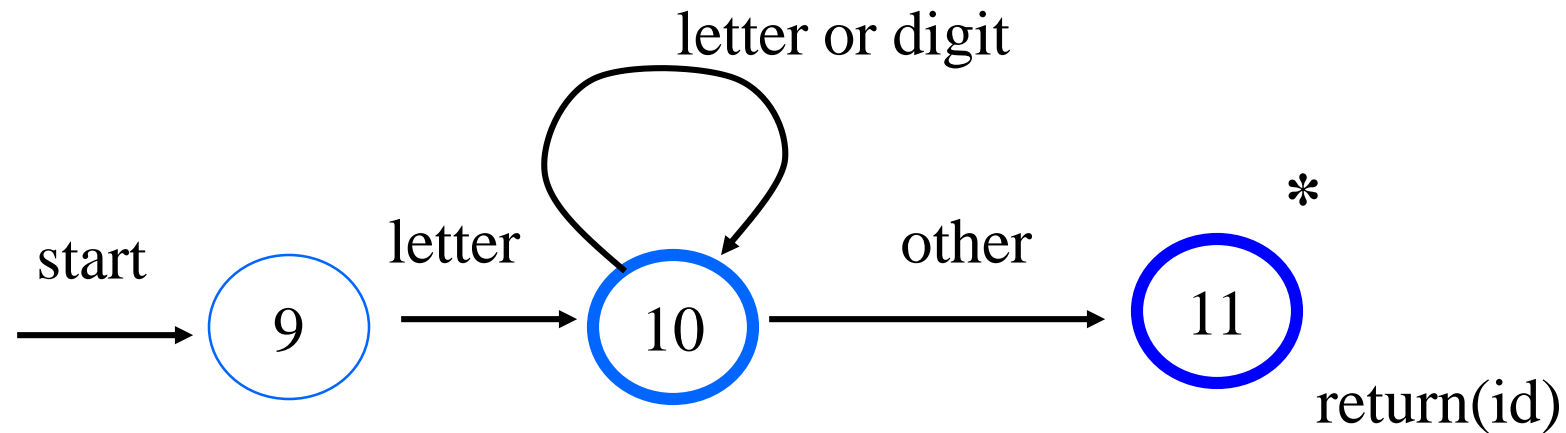
# Mapping transition diagrams into C code

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
                    ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

## b. Recognition of Identifiers

Ex2:  $ID = \text{letter}(\text{letter} \mid \text{digit})^*$

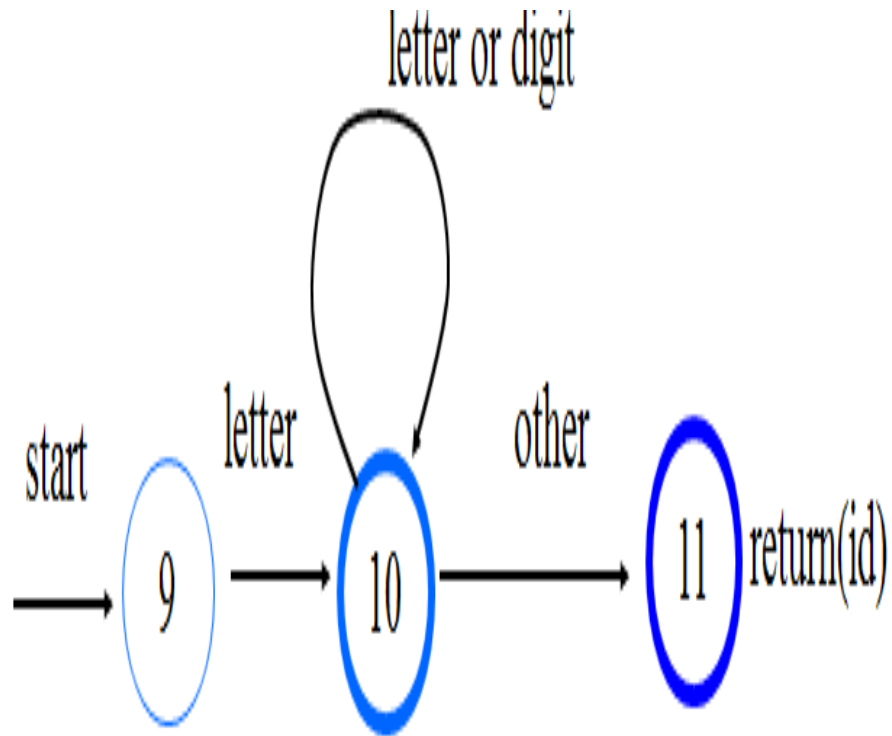
Transition Diagram:



*\* indicates input retraction*



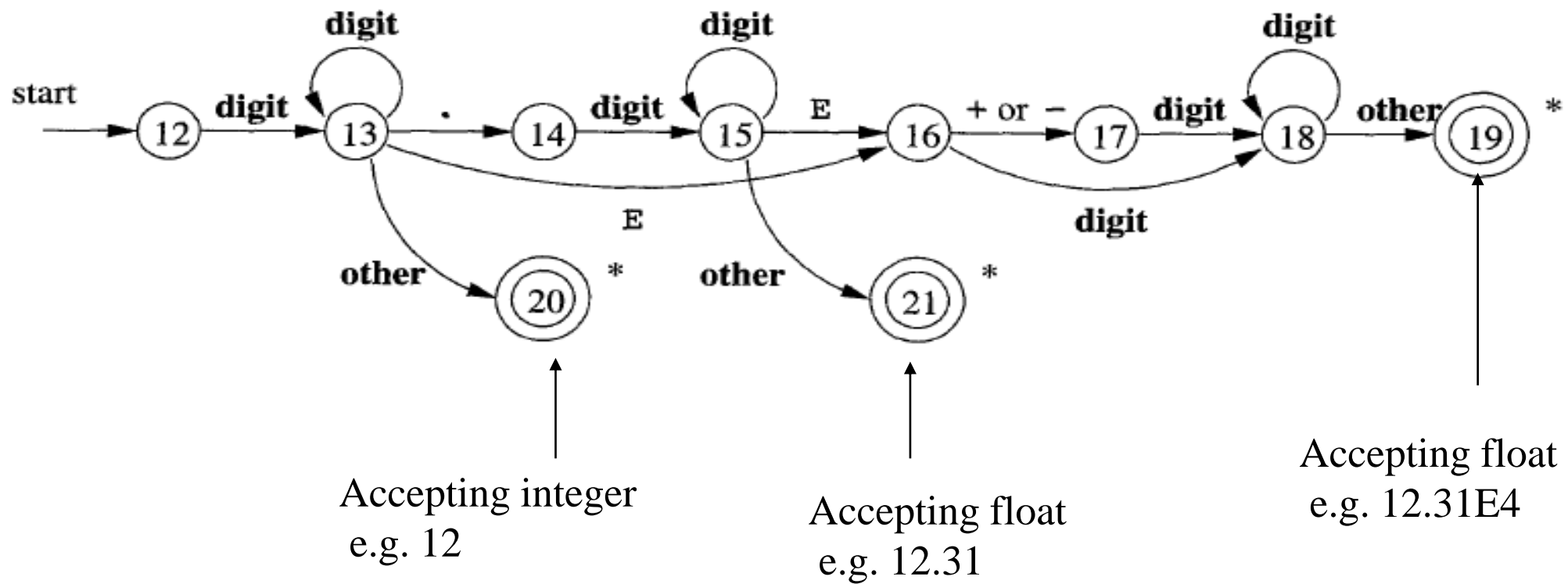
# Mapping transition diagrams into C code



```
switch (state) {  
  case 9:  
    if (isletter( c ) ) state = 10; else state =  
    failure();  
    break;  
  case 10: c = nextchar();  
    if (isletter( c ) || isdigit( c ) ) state = 10; else  
    state =11  
  case 11: retract(1);  
    insert(id);  
    return;
```

## c.Recognition of numbers (integer | floating points)

Unsigned number in Pascal



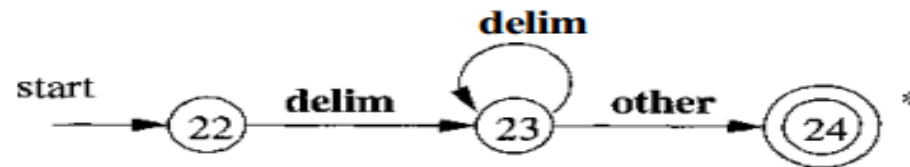
## Example #1

- The lexeme for a given token must be the longest possible
- Assume input to be **12.34E56**
- Starting in the third diagram the accept state will be reached after **12**
- Therefore, the matching should always start with the first transition diagram
- If failure occurs in one transition diagram then *retract* the forward pointer to the start state and activate the next diagram.
- If failure occurs in all diagrams then a lexical error has occurred

## d. Recognition of delimiter

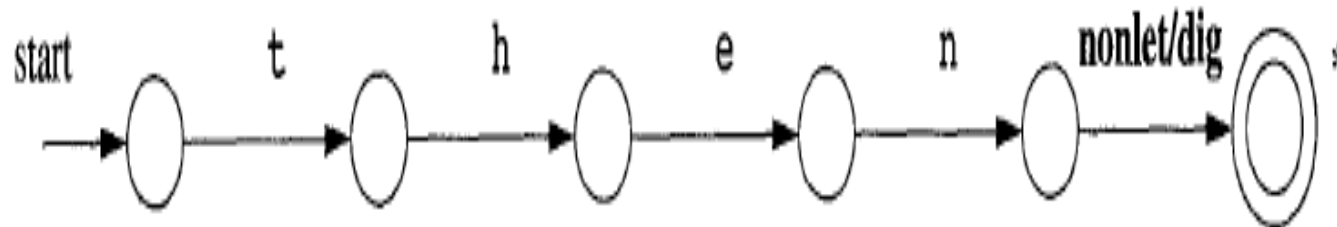
- Whitespace characters, represented by **delim** in that diagram
- typically these characters would be **blank**, **tab**, **newline**, and
- perhaps other characters that are not considered by the language design to be part of any token.

$$ws \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+$$



## e. Recognition of keyword

- Install the reserved words in the symbol table initially.
- A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.
- Create separate transition diagrams for each keyword; the transition diagram for the reserved word: **then**



## Exercise #1

Q. Sketch transition diagram for Recognition of keywords such as: *if, else, for*

## ✓ Tokens, their patterns, and attribute values

- Token information stored in **symbol table**
- Token contains two components: **token name** and **attribute value**.
- For example

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	—	—
<b>if</b>	<b>if</b>	—
<b>then</b>	<b>then</b>	—
<b>else</b>	<b>else</b>	—
Any <i>id</i>	<b>id</b>	Pointer to table entry
Any <i>number</i>	<b>number</b>	Pointer to table entry
<b>&lt;</b>	<b>relop</b>	LT
<b>&lt;=</b>	<b>relop</b>	LE
<b>=</b>	<b>relop</b>	EQ
<b>&lt;&gt;</b>	<b>relop</b>	NE
<b>&gt;</b>	<b>relop</b>	GT
<b>&gt;=</b>	<b>relop</b>	GE

# LEX (Lexical Analyzer Generator)

- **Lex** is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text.
- **Lex** is an acronym that stands for "**lexical analyzer generator**."
- *The main purpose of LEX is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers.*
- The input notation for the **Lex tool** is referred to as the *Lex language* and the tool itself is the *Lex compiler*.
- The *Lex compiler* transforms the input patterns into a transition diagram and generates code, in a file called **lex.yy.c**, that simulates this transition diagram.



# How Lex tool works?

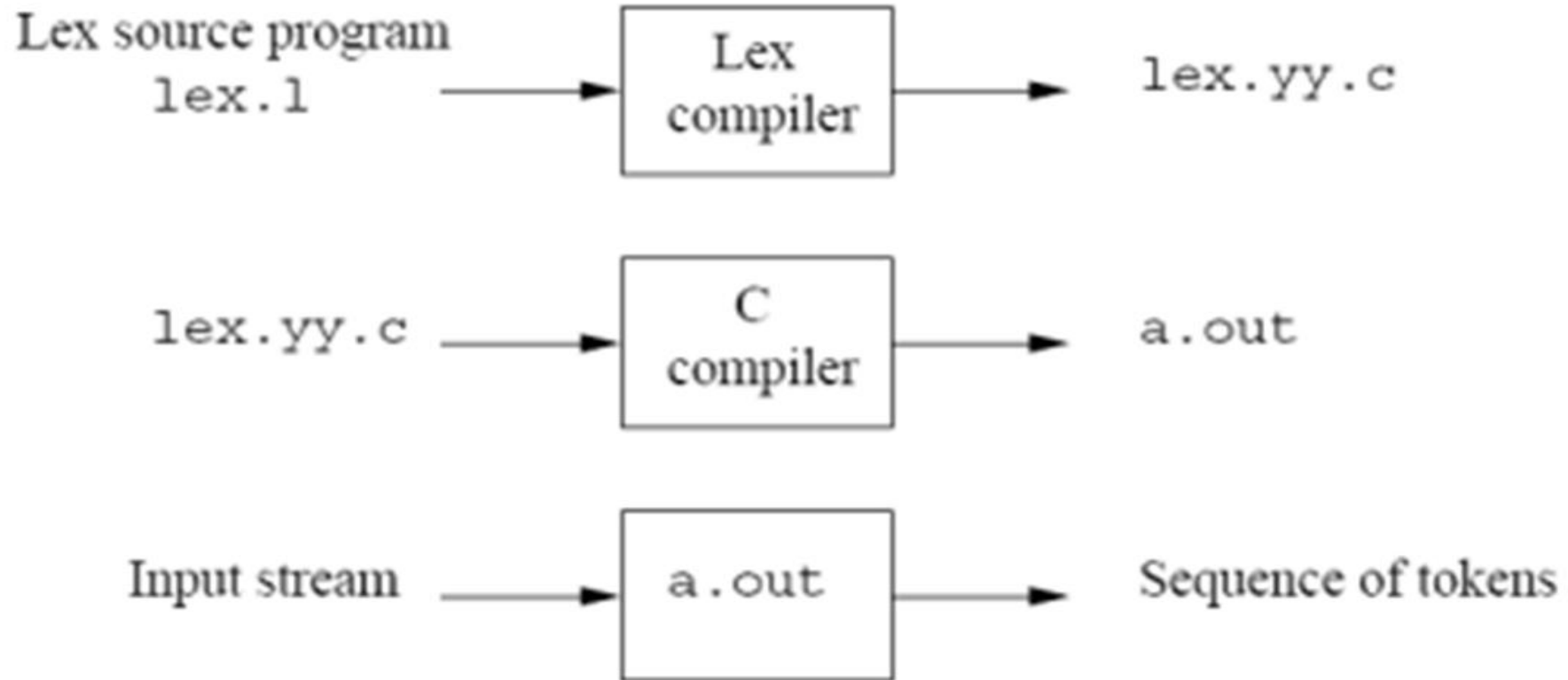


Figure 3.22: Creating a lexical analyzer with Lex

## How Lex tool works?...

- Figure 3.22 suggests how Lex is used:
  - Firstly lexical analyzer creates a program **lex.1** in the *Lex language*.
  - Then Lex compiler runs the **lex.1 program** and produces a C program **lex.yy.c**.
  - Finally C compiler runs the **lex.yy.c** program and produces an object program **a.out**.
  - **a.out** is lexical analyzer that transforms an input stream into a *sequence of tokens*.

# Structure of Lex Programs

- A Lex program is separated into three sections by %% delimiters.
- The general format of Lex source is as follows:

```
% {  
Declarations  
% }  
%%  
translation rules // form: pattern {action}  
%%  
{  
auxiliary functions  
}
```

## ***Declarations sections***

- ✓ Declarations of ordinary C variables, constants and Libraries.

```
% {
```

```
#include<math.h>
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
% }
```

# Translation Rules

- It contains regular expressions and code segments.
- **Translation Rules** define the statement of form

**pattern1 { action1 }**  
**pattern2 { action2 }**  
**.... Pattern n { action }.**

- Where
  - **pattern** describes the regular expression or regular definition
  - **action** describes the actions what action the lexical analyzer should take when pattern

pi matches a lexeme. For example:

<b>[0-9]+</b>	<b>{ return(Integer); } // RE</b>
<b>{DIGIT}+</b>	<b>{ return(Integer); } // RD</b>

## Auxiliary functions

- This section holds additional functions which are used in actions.
- These functions are compiled separately and loaded with lexical analyzer.
- Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.
- Once a match is found, the associated action takes place to produce token.
- The token is then given to parser for further processing.

# Lex Predefined Variables

- ✓ **yytext** -- a string containing the lexeme
- ✓ **yytext** -- the length of the lexeme
- ✓ **yyin** -- the input stream pointer
  - the default input of default main() is **stdin**
- ✓ **yyout** -- the output stream pointer
  - the default output of default main() is **stdout**.

# Lex Library Routines

- **yylex()**
  - The default main() contains a call of yylex()
- **yymore()**
  - return the next token
- **yyless(n)**
  - retain the first n characters in yytext
- **yywarp()**
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1



## Example #1: Lex program to identify the identifier

% {	{
#include<stdio.h>	return 1;
% }	}
% %	int main()
[a-zA-Z]([a-zA-Z] [0-9])*	{
{printf("Identifier\n");}	printf("Enter a string\n");
[0-9]+ {printf("Not a Identifier\n");}	yylex();
% %	
/*call the yywrap function*/	
int yywrap()	}

