

# COMP409: COMPILER DESIGN

## 1. LEXICAL ANALYSIS

Instructor: Sushil Nepal, Assistant Professor

Block: 9-308

Email: [sushilnepal@ku.edu.np](mailto:sushilnepal@ku.edu.np)

Contact: 9851-151617



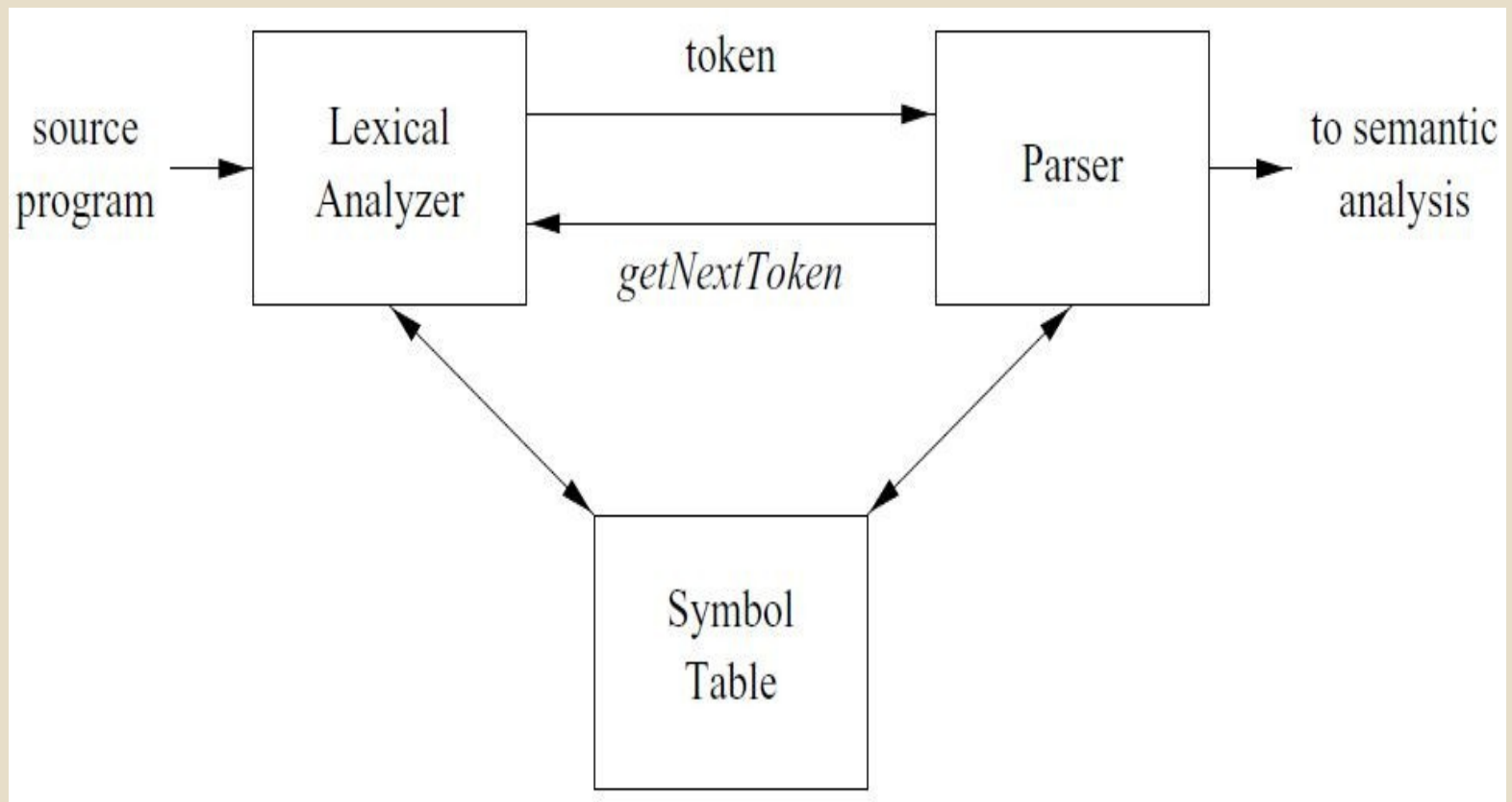
# Lexical Analysis

2

- It is the initial part of reading and analyzing the program text
- The text is read and divided into tokens, each of which corresponds to a symbol in programming language, e.g. a variable name, keyword or number, etc.
- Lexical analyzer (also called lexer) will as its input take a string of individual letters and divide this string into tokens

# The Role of Lexical Analyzer

3



# The Role of Lexical Analyzer

4

- A lexical analyzer doesn't return a list of tokens at once, it returns a token when the parser asks a token from it
- The parser requests the lexical analyzer for the next token whenever it requires one using **getnexttoken( )**
- On the receipt of the command, lexer scans the input and processes until a token is matched

# The Role of Lexical Analyzer

5

- The function of a lexical analyzer is to read the input stream representing the source program, one character at a time and to translate it into valid tokens
- Lexical analyzer may also perform other operations like
  - *removing redundant white spaces (i.e. blanks, tabs and newlines)*
  - *removing token separators (like semicolon)*
  - *removal of comments*
  - *providing line number to the parser for error reporting*

# Reasons for Separation of Lexical Analysis and Parsing

6

- Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
- Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
- Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

# Tokens, Patterns and Lexemes

7

## ■ Tokens

- *A token is a single word of source code input*
- *A token is a pair consisting of a token name and an optional attribute value*
- *Tokens are the separately identifiable blocks with collective meaning*

# Tokens, Patterns and Lexemes

8

- *When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword, etc of the language, these substrings are called the tokens of the language*
- *They are the building block of the programming language*
- *Eg: if, else, identifiers*



# Tokens, Patterns and Lexemes

9

## ■ Lexemes

- *Lexemes are the actual string matched as token*
- *They are the specific characters that make up of a token*
- *For example, **abc** and **123***
- *A token can represent more than one lexeme. i.e. token **intnum** can represent lexemes **123**, **244**, **4545**, etc*

# Tokens, Patterns and Lexemes

10

## ■ Patterns

- *Patterns are rules describing the set of lexemes belonging to a token*
- *For example: “letter followed by letters and digits” and “non-empty sequence of digits”*
- *Regular expressions are usually used to specify patterns*
- *Eg: **intnum** token can be defined as **[0-9][0-9]\****

# Attributes for Tokens

11

- When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme
- This additional information is called as the **attribute** of the token
- Eg: If token ***id*** matched ***var1*** and ***var2*** both, then lexical analyzer must be able to represent ***var1*** and ***var2*** as different identifiers
- For obtaining actual value, each token is associated with attribute, generally pointer to the symbol table

# Attributes for Tokens

12

- Some attributes:
  - *$\langle id, attr \rangle$  where  $attr$  is pointer to the symbol table*
  - *$\langle assignop, \_ \rangle$  no attribute is needed (if there is only one assignment operator)*
  - *$\langle num, val \rangle$  where  $val$  is the actual value of the number*
- Eg:  $dest = source + 5$ 
  - Tokens:  *$\langle id, pt \text{ for } dest \rangle, \langle assignop \rangle, \langle id, pt \text{ for } source \rangle, \langle num, 5 \rangle$*
- Token type and its attribute uniquely identifies a lexeme

# Attributes for Tokens

13

- Example: take statement,

**area = 3.1416 \* r \* r**

**1.getnexttoken( )** returns (**id, attr**) where **attr** is pointer to **area** in symbol table

**2.getnexttoken( )** returns (**assignop**) where no attribute is needed, if there is only one assignment operator

**3. getnexttoken( )** returns (**floatnum, 3.1416**) where **3.1416** is the actual value of floatnum etc

# Lexical Errors

14

- Though error at lexical analysis is normally not common, there is possibility of errors
- When error occurs, the lexical analyzer must not halt the process
- It can print the error message and continue
- Error in this phase is found when there are no matching strings found as given by the pattern

# Lexical Errors

15

- Some error recovery techniques
  - *Deletion of extraneous character*
  - *Inserting missing character*
  - *Replacing incorrect character by correct one*
  - *Transposition of adjacent characters*
- Lexical error recovery is normally an expensive process
- Recovery eg: finding the number of transformations that would make the correct tokens

# Approaches to Implementing Lexical Analyzer

16

1. Use lexical analyzer generator like **Flex** that produces lexical analyzer from the given specification as regular expression. The generator provides routine for reading and buffering the input.
2. Write a lexical analyzer in general programming language like C. We need to use the I/O facility of the language for reading and buffering the input.
3. Use the assembly language to write the lexical analyzer. Explicitly manage the reading of input.



# Approaches to Implementing Lexical Analyzer

17

- These strategies are in increasing order of difficulty and efficiency
- Since we deal with characters in lexical analysis, it is better to take some time during implementation to get efficient result

# Input Buffering

18

- Technique used to speed up reading the source program
- There are many situations where we need to look at least one (if not more) additional character ahead to recognize lexemes in the input
- For eg, **int** is a keyword in C but **intnum** is an identifier so when the scanner reads **i**, **n**, **t**, it has to look for other characters to see whether it is just **int** or some other word

# Input Buffering

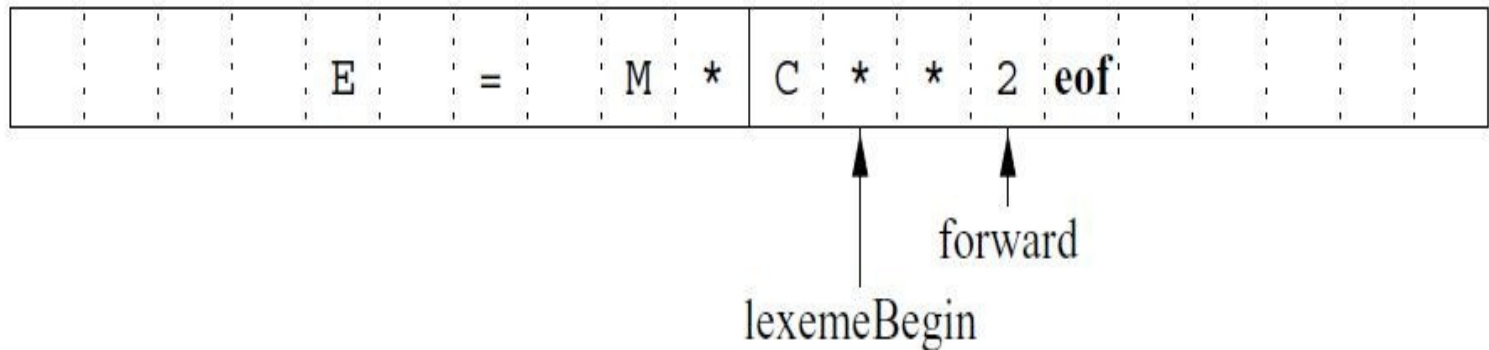
19

- In this case, when the token is read the next time, the scanner needs to move back to rescan the input again for the characters that are not used for the lexeme and this is time consuming
- In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=
- To reduce the overhead and efficiently move back and forth, input buffering technique is used

# Buffer Pairs (2N Buffering)

20

- Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character



# Buffer Pairs

21

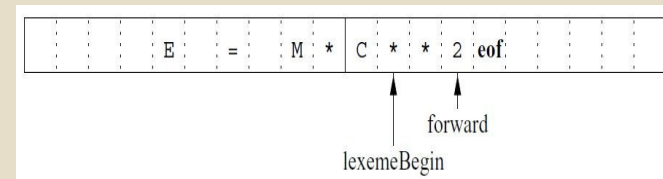
- Each buffer is of the same size  $N$ , and  $N$  is usually the size of a disk block, e.g., 4096 bytes
- Using one system read command we can read  $N$  characters into a buffer, rather than using one system call per character
- If fewer than  $N$  characters remain in the input file, then a special character, represented by **eof** marks the end of the source file

# Buffer Pairs

22

- Two pointers to the input are maintained:
  - Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine
  - Pointer **forward** scans ahead until a pattern match is found
- Once the next lexeme is determined, **forward** is set to the character at its right end
- After the lexeme is recorded, **lexemeBegin** is set to the character immediately after the

# Buffer Pairs



23

- In the above figure, **forward** has passed the end of the next lexeme, and must be retracted one position to its left
- Advancing **forward** requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer

***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;
```



# Sentinels

25

- If we use the 2N buffering scheme, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer
- Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read

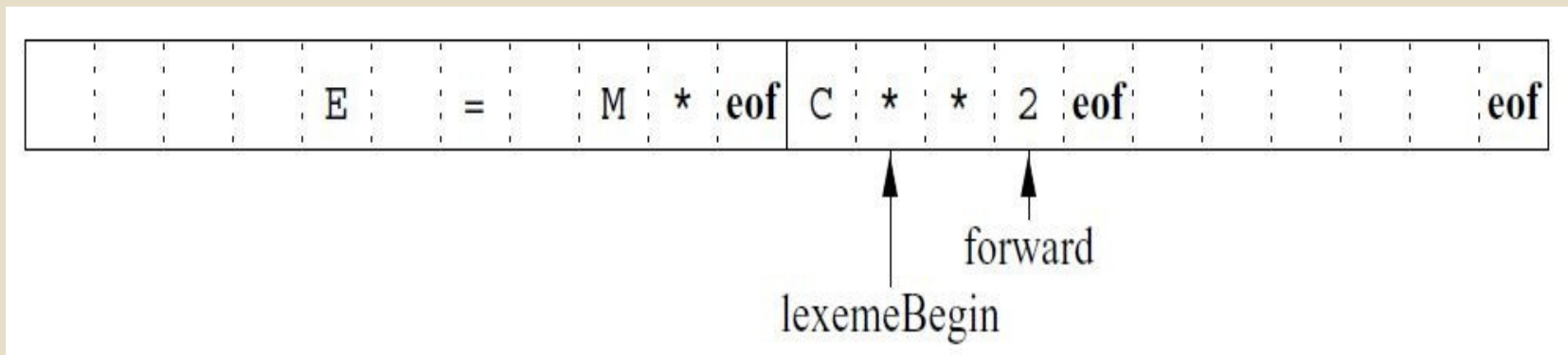
# Sentinels

26

- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**
- **eof** retains its use as a marker for the end of the entire input
- Any **eof** that appears other than at the end of a buffer means that the input is at an end

# Sentinels

27



- *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 /\* eof within a buffer signifying end of input \*/*
- *terminate lexical analysis*
- *end*

# Specification of Tokens

29

- Regular expression is the common way of specifying the patterns for tokens
- Some Definitions:
- **Alphabet:**
  - *Set of symbols that generate language. For e.g.  $\{0-9\}$  is an alphabet that is used to produce all the non-negative integer numbers*
  - *$\{0-1\}$  is an alphabet that is used to produce all the binary strings*

# Specification of Tokens

30

## ■ **String:**

- *Finite sequence of characters from the alphabet*
- *Given the alphabet  $A$ ,  $A^2 = A.A$  is set of strings of length 2, similarly  $A^n$  is set of strings of length  $n$*
- *The **length** of the string  **$w$**  is denoted by  $|w|$  i.e. number of characters (symbols) in  **$w$***
- *We also have  $A^0 = \{\epsilon\}$ , where  $\epsilon$  is called empty string*

# Specification of Tokens

31

## ■ Kleene Closure:

- *Kleene closure of an alphabet  $A$  denoted by  $A^*$  is set of all strings of any length (0 also) possible from  $A$*
- *Mathematically  $A^* = A^0 \cup A^1 \cup A^2 \cup \dots$*
- *For any string,  $w$  over alphabet  $A$ ,  $w \in A^*$*

# Specification of Tokens

32

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>	$L^+ = \bigcup_{i=1}^{\infty} L^i$



# Specification of Tokens

33

- A **language**  $L$  over alphabet  $A$  is the set such that  $L \subseteq A^*$
- The string  $s$  is called **prefix** of  $w$ , if the string  $s$  is obtained by removing zero or more trailing characters from  $w$ . If  $s$  is a proper prefix, then  $s \neq w$ .
- The string  $s$  is called **suffix** of  $w$ , if the string  $s$  is obtained by deleting zero or more leading characters from  $w$ . We say  $s$  is a proper suffix if  $s \neq w$ .
- The string  $s$  is called **substring** of  $w$  if we can obtain  $s$  by deleting zero or more leading or trailing characters from  $w$ . We say  $s$  is a proper substring if  $s \neq w$ .

# Specification of Tokens

34

- **Regular Operators:**

- The following operators are called regular operators and the language formed called regular language.

- $\cdot \rightarrow$  Concatenation operator,  $R.S = \{rs \mid r \in R \text{ and } s \in S\}$

- $* \rightarrow$  Kleene star operator,  $A^* = \bigcup_{i \geq 0} A^i$

- $+ / \cup / | \rightarrow$  Choice/union operator,  $R \cup S = \{t \mid t \in R \text{ or } t \in S\}$

# Specification of Tokens

35

- **Regular Expression (RE):**
- We use regular expression to describe the tokens of a programming language.
- **Basic Symbol**
  - $\epsilon$  is a regular expression denoting language  $\{\epsilon\}$
  - $a \in A$  is a regular expression denoting  $\{a\}$

# Specification of Tokens

36

- If  $r$  and  $s$  are regular expressions denoting languages  $L1(r)$  and  $L2(s)$  respectively, then
  - $r + s$  is a regular expression denoting  $L1(r) \cup L2(s)$
  - $rs$  is a regular expression denoting  $L1(r) \cdot L2(s)$
  - $r^*$  is a regular expression denoting  $(L1(r))^*$
  - $(r)$  is a regular expression denoting  $L1(r)$

# Specification of Tokens

37

## ■ *Practice Questions*

- (a) RE in which every pair of adjacent zero's appear before any pair of adjacent ones.
- (b) RE that gives binary strings having at most two 1s
- (c) RE that denotes the language of all strings that ends with 00 (binary number multiple of 4)
- (d) RE that denotes the set of all strings that describes alternating 1s and 0s

# Specification of Tokens

38

## ■ *Answers*

(a)  $(01)^*(0011)(01)^*$

(b)  $0^*10^*10^* + 0^*10^* + 0^*$

(c)  $(1+0)^*00$

(d)  $(01)^* + (10)^*$

# Specification of Tokens

39

## ■ Properties of RE

- $r+s = s+r$  ( $+$  is commutative)
- $r+(s+t) = (r+s)+t$  ;  $r(st) = (rs)t$  ( $+$  and  $.$  are associative)
- $r(s+t) = (rs)+(rt)$ ;  $(r+s)t = (rt)+(st)$  ( $.$  distributes over  $+$ )
- $\varepsilon r = r\varepsilon$  ( $\varepsilon$  is identity element)
- $r^* = (r+\varepsilon)^*$  (relation between  $*$  and  $\varepsilon$ )
- $r^{**} = r^*$  ( $*$  is idempotent)

# Specification of Tokens

40

- **Regular Definitions:**
- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex
- In those cases, we may use ***regular definitions***
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions



# Specification of Tokens

41

A *regular definition* is a sequence of the definitions of the form:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.

$$d_n \rightarrow r_n$$

where  $d_i$  is a distinct name and

$r_i$  is a regular expression over symbols in

$$\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$$

basic symbols

previously defined names

# Specification of Tokens

42

- Identifiers in Pascal are defined as a string of letters and digits beginning with a letter

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

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

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

- If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex:

$(A \mid \dots \mid Z \mid a \mid \dots \mid z) ( (A \mid \dots \mid Z \mid a \mid \dots \mid z) \mid (0 \mid \dots \mid 9) )^*$

# Recognition of Tokens

43

- A recognizer for a language is a program that takes a string **w**, and answers “YES” if **w** is a sentence of that language, otherwise “NO”
- The tokens that are specified using RE are recognized by using transition diagram or finite automata (FA)
- Starting from the start state we follow the transition defined

# Recognition of Tokens

44

- If the transition leads to the accepting state, then the token is matched and hence the lexeme is returned, otherwise other transition diagrams are tried out until we process all transition diagrams or failure is detected
- Recognizer of tokens takes the language **L** and the string **s** as input and tries to verify whether **s**  $\in$  **L** or not

# Recognition of Tokens

45

- We concentrate on a class of recognizer called Finite Automata (FA)
- There are two types of Finite Automaton:
  - *Deterministic Finite Automaton (DFA)*
  - *Non Deterministic Finite Automaton (NFA)*

# Recognition of Tokens

46

## Design of a Lexical Analyzer

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)

# Recognition of Tokens

47

- Why convert from NFA to DFA?
  - Computer programs generally need to know all possible transitions and states for a given state machine.
  - A non-deterministic finite automaton can have a transition that goes to any number of states for a given input and state.
  - This is a problem for a computer program because it needs precisely one transition for a given input from a given state.
  - The process of converting NFA to DFA eliminates this ambiguity and allows a program to be made.

# Deterministic Finite Automata (DFA)

- FA is deterministic, if there is exactly one transition for each (state, input) pair
- It is a fast recognizer but takes large space
- DFA is a five tuple  $(S, \Sigma, q_0, \delta, F)$  where,
  - $S \rightarrow$  finite set of states
  - $\Sigma \rightarrow$  finite set of input alphabets
  - $q_0 \rightarrow$  starting state
  - $\delta \rightarrow$  transition function i.e.  $\delta : S \times \Sigma \rightarrow S$



# Deterministic Finite Automata

49

- The following is the algorithm for simulating DFA for recognizing given string
- For a given string **w**, in DFA **D**, with start state **q<sub>0</sub>**, with set of final states **F**, the output is “**YES**”, if **D** accepts **w**, otherwise “**NO**”

```
DFASim(D,  $q_0$ ) {
```

```
     $q = q_0$ 
```

```
     $c = \text{getchar}()$ 
```

```
    while ( $c \neq \text{eof}$ ) {
```

```
         $q = \text{move}(q, c)$ 
```

```
// this is  $\delta$  function.
```

```
         $c = \text{getchar}()$ 
```

```
    }
```

```
    if ( $q$  is in  $F$ )
```

```
// if D accepts w
```

```
        return "yes"
```

```
else
```

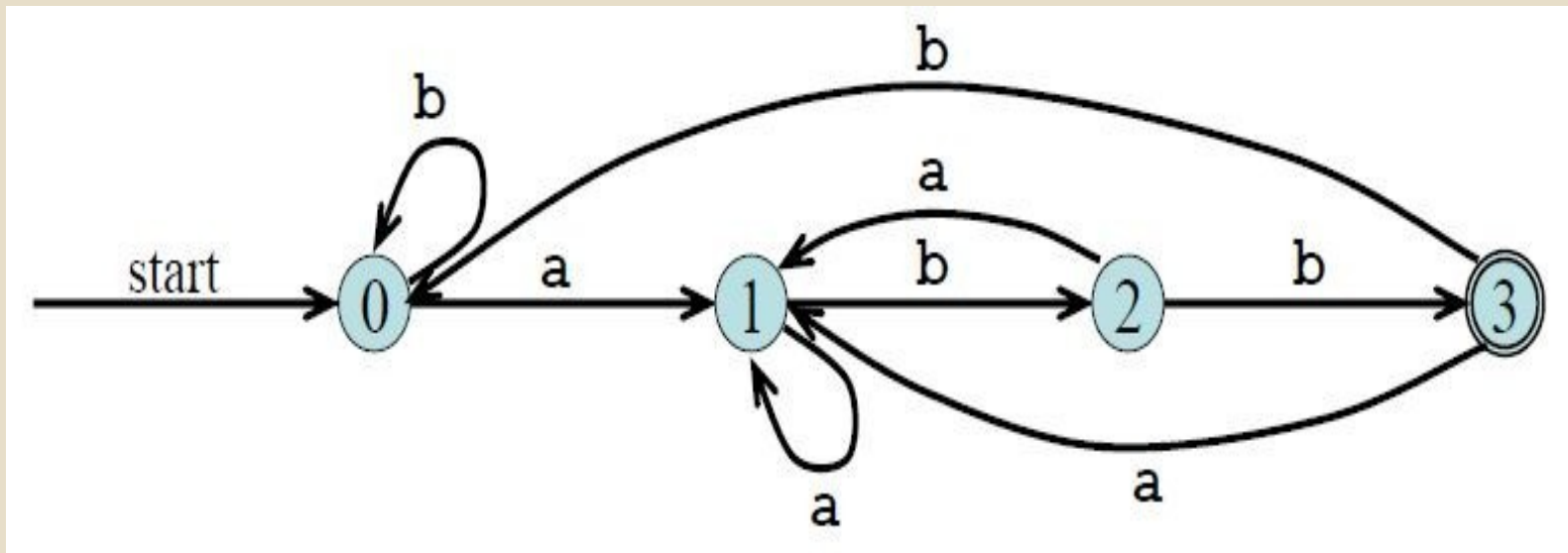
```
    return "no"
```

```
}
```

# Deterministic Finite Automata

51

- The figure shows DFA for the regular expression:  **$(a+b)^*abb$**



# Practice Questions

52

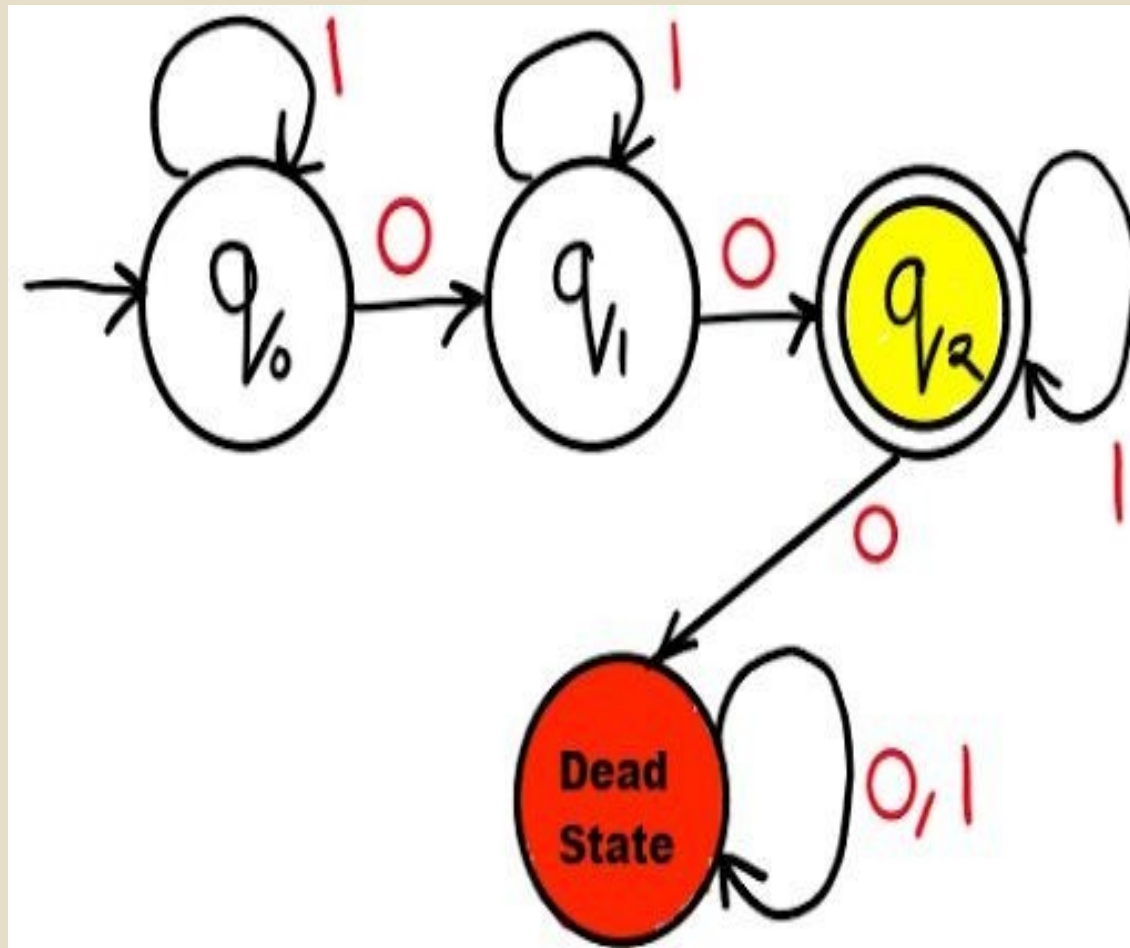
Q) Write the regular expression and draw its corresponding DFA for the following:

(1) Language accepting strings containing exactly two 0s over input alphabets  $\Sigma = \{0, 1\}$ .

(2) Language accepting strings starting and ending with different characters over input alphabets  $\Sigma = \{0, 1\}$ .

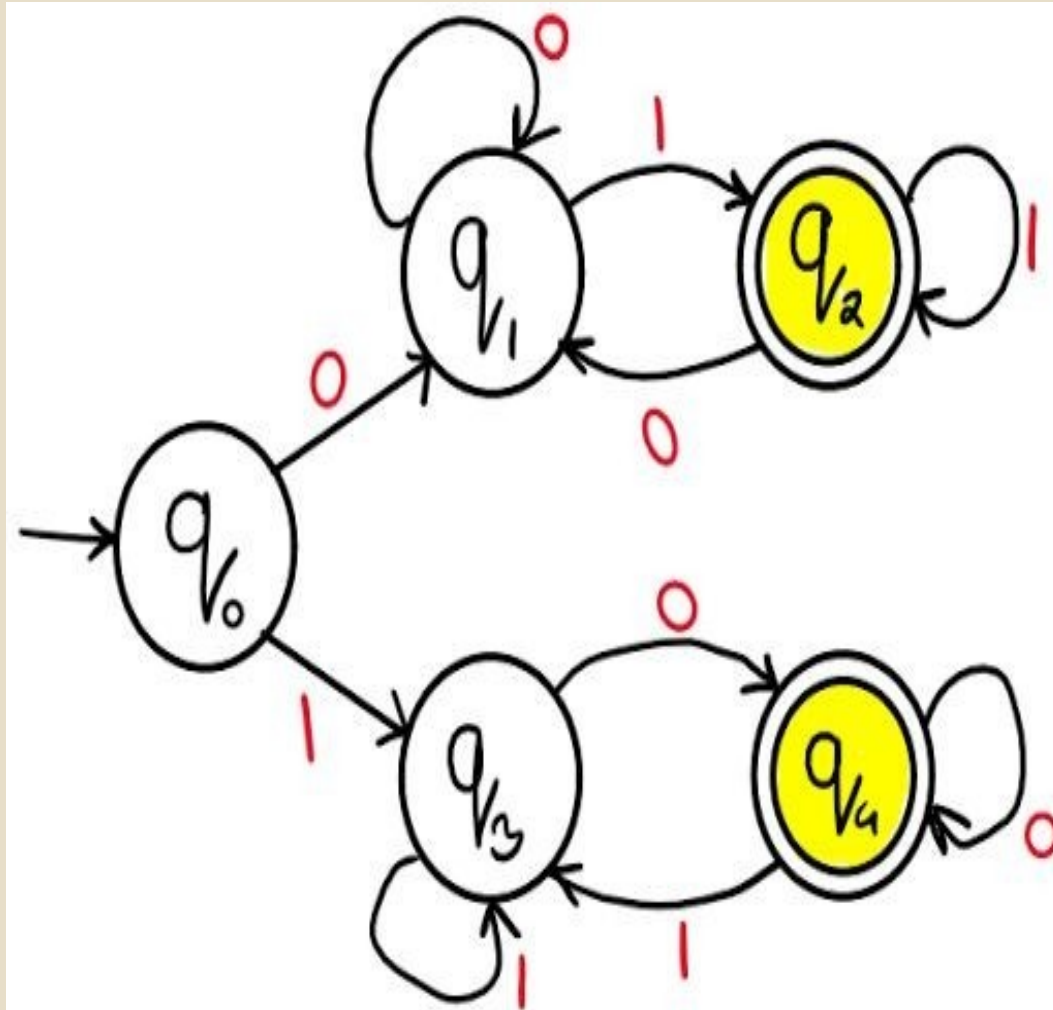
Ans 1:      $1^*01^*01^*$

53



Ans 2:  $(0(0+1)*1)+(1(1+0)*0)$

54



# Non-Deterministic Finite Automata (NFA)

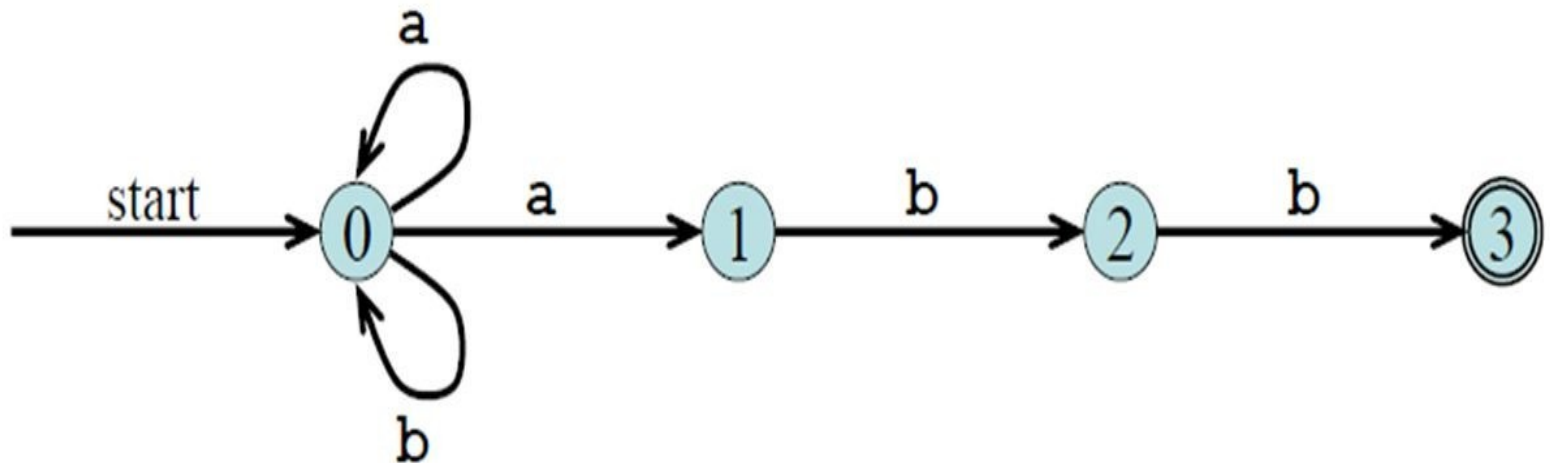
55

- FA is non-deterministic if there can be more than one transition (or none) for each (state, input) pair
- It is a slow recognizer but takes less space
- An NFA is a five tuple  $(S, \Sigma, q_0, \delta, F)$  where,
  - $S \rightarrow$  *finite set of states*
  - $\Sigma \rightarrow$  *finite set of input alphabets*
  - $q_0 \rightarrow$  *starting state*
  - $\delta \rightarrow$  *transition function i.e.  $\delta : S \times \Sigma \rightarrow 2^{|S|}$*
  - $F \rightarrow$  *set of final states  $F \subseteq S$*

# Non-Deterministic Finite Automata

56

- The figure shows NFA for the regular expression:  **$(a+b)^*abb$**





# Practice Questions

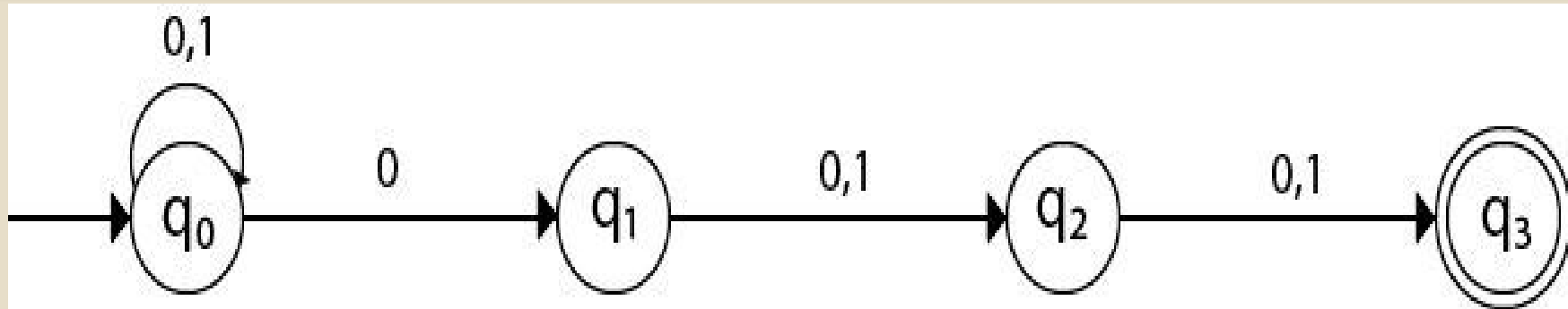
57

(1) Design an NFA with  $\Sigma = \{0, 1\}$  that accepts all strings in which the third symbol from the right end is always 0.

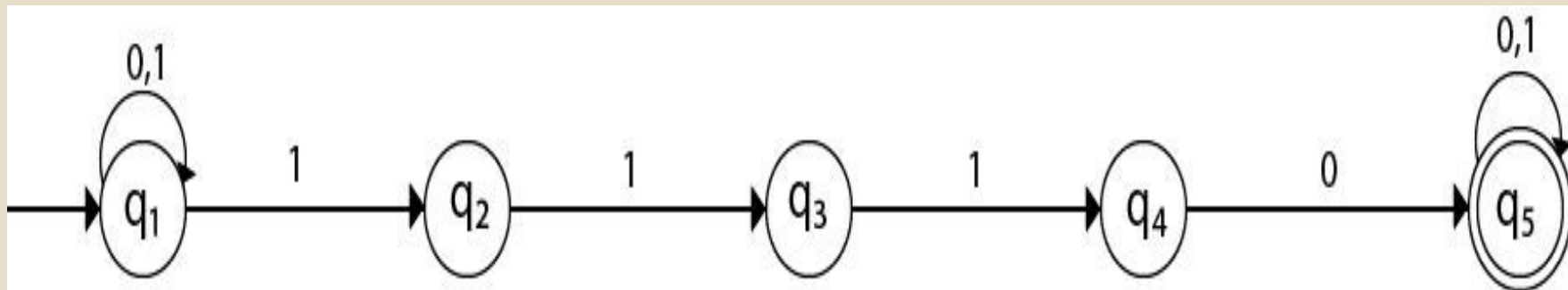
(2) Design an NFA with  $\Sigma = \{0, 1\}$  which accepts all strings containing the substring 1110.

Ans 1:

58



Ans 2:



# $\epsilon$ -NFA

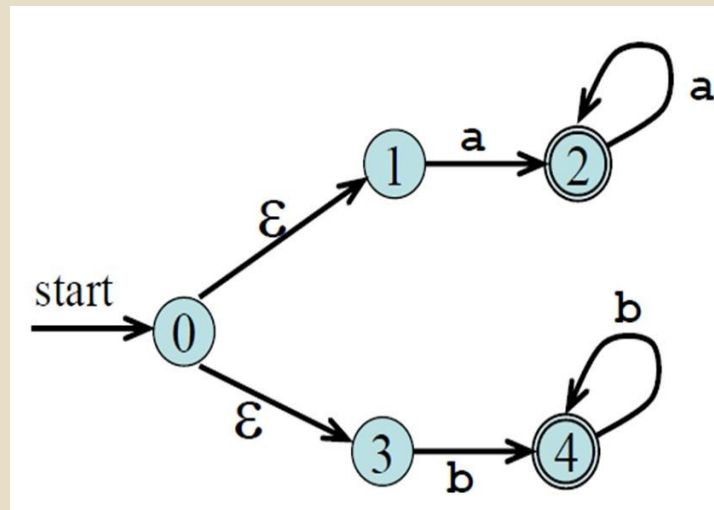
59

- $\epsilon$ -transitions are allowed in  $\epsilon$ -NFAs
- In other words, we can move from one state to another without consuming any symbol
- In case of  $\epsilon$ -NFA the only difference is  $\Sigma = \Sigma \cup \{\epsilon\}$  and hence  $\delta: S \times \Sigma \cup \{\epsilon\} \rightarrow 2^{|S|}$

# $\epsilon$ -NFA

60

- The figure shows a state machine with  $\epsilon$  moves that is equivalent to the regular expression:  **$aa^* + bb^*$**



*//SIMULATING NFA*

*S =  $\epsilon$ -closure(  $\{S_0\}$  )      // set of all states that can be  
accessed from  $S_0$  by  $\epsilon$ - transitions*

*c = getchar( )*

*while(c != eof) {*

*S =  $\epsilon$ -*

*closure(move(S,  
c)) = getchar( )  
}*

*//set of all states that can be  
accessible from a state in S by a  
transition on c*

*if (  $S \cap F \neq \Phi$  ) then*

*return **“YES”***

*else*

*return **“NO”***

# RE to NFA (Thompson's Construction)

- Thompson's construction is a 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 subexpressions are combined

# RE to NFA

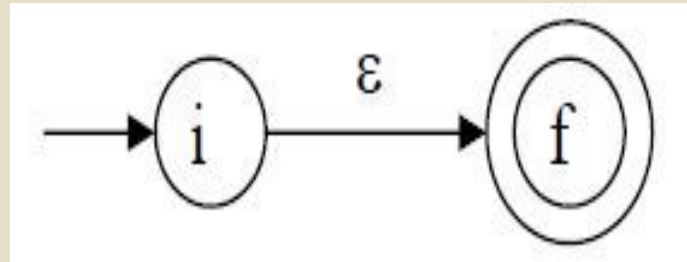
63

- **Input** → RE,  $r$ , over alphabet  $\Sigma$
- **Output** →  $\epsilon$ -NFA accepting  $L(r)$
- **Procedure** → Process in bottom-up manner by creating  $\epsilon$ -NFA for each symbol in  $\Sigma$  including  $\epsilon$ . Then recursively create for other operations as shown below.

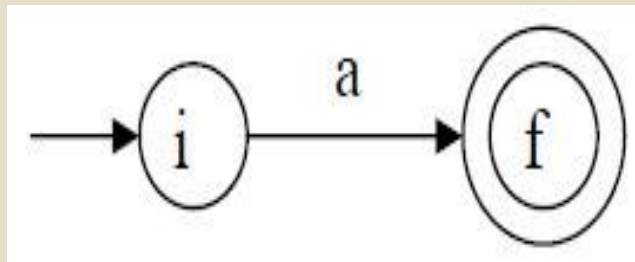
# RE to NFA

64

(1) To recognize an empty string  $\epsilon$



(2) To recognize a symbol **a** in the alphabet  $\Sigma$



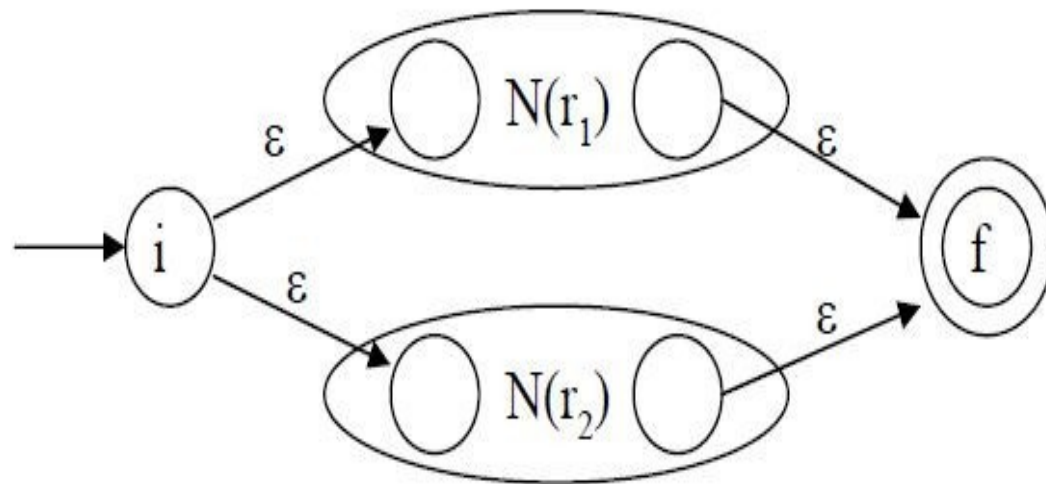


# RE to NFA

65

(3) If  $N(r_1)$  and  $N(r_2)$  are NFAs for regular expressions  $r_1$  and  $r_2$

(A) For regular expression  $r_1 + r_2$



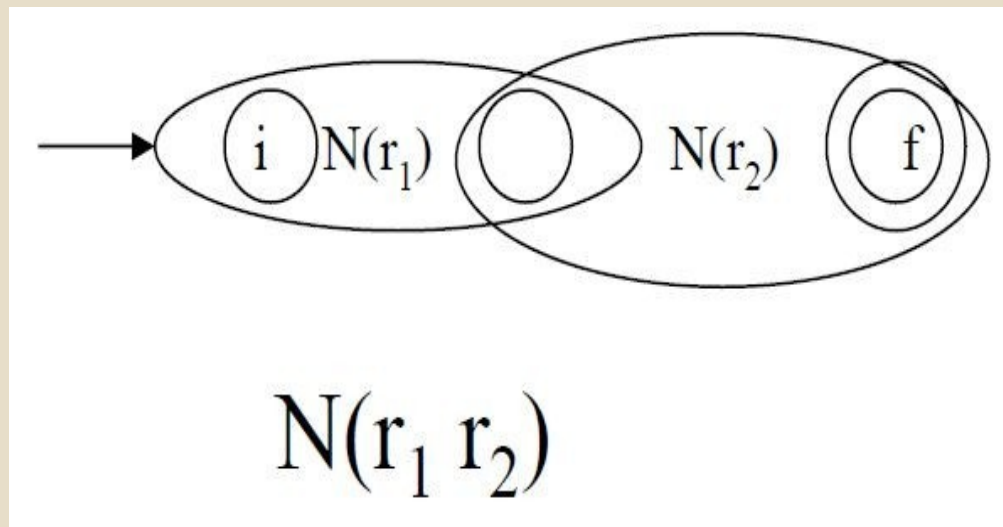
$N(r_1 + r_2)$

# RE to NFA

66

(B) For regular expression  $r_1 \cdot r_2$

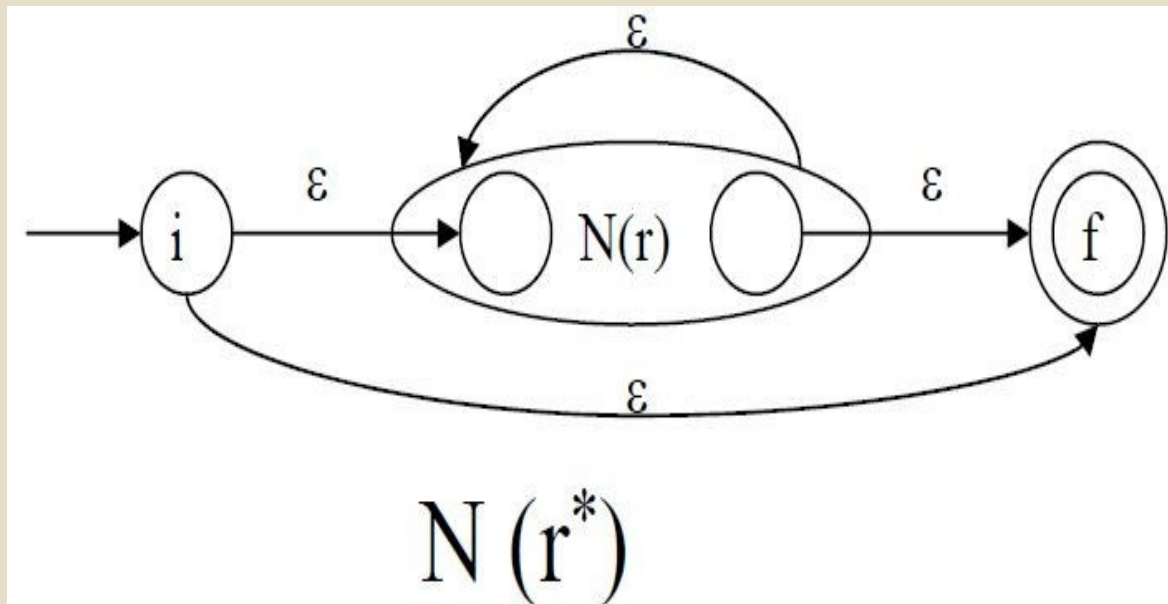
The start state of  $N(r_1)$  becomes the start state of  $N(r_1 r_2)$  and final state of  $N(r_2)$  become final state of  $N(r_1 r_2)$



# RE to NFA

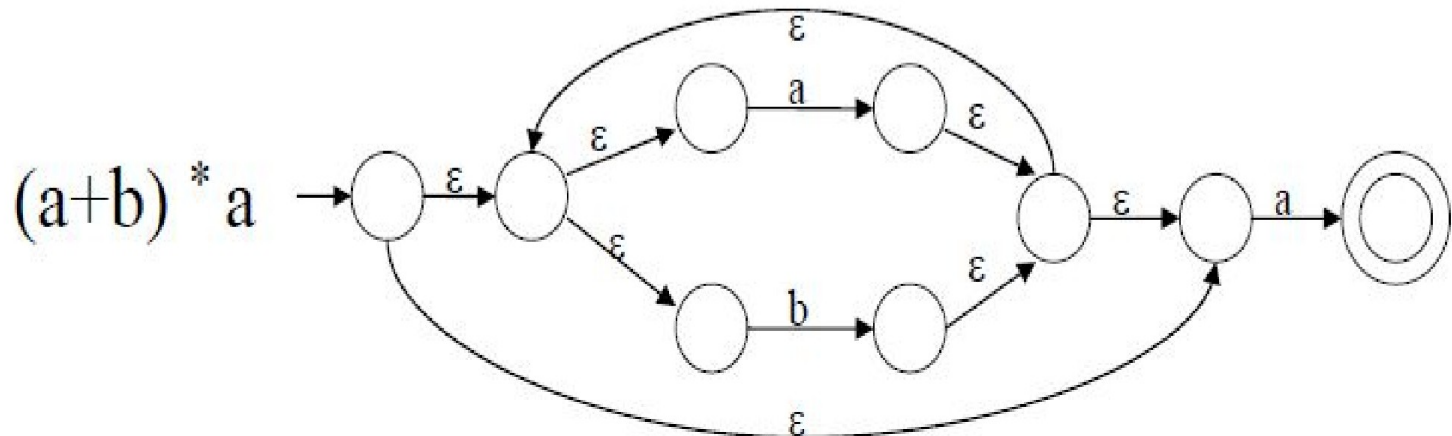
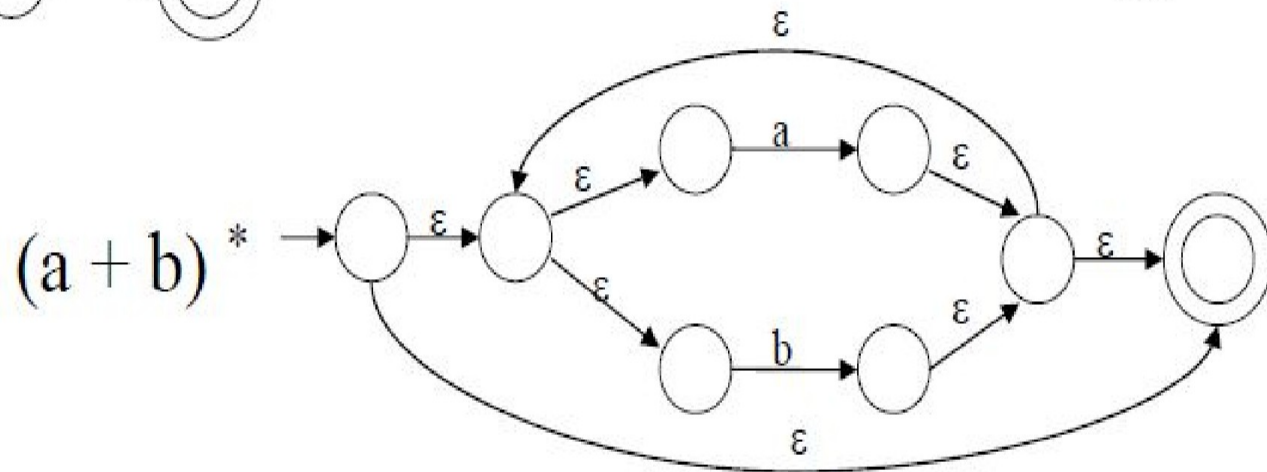
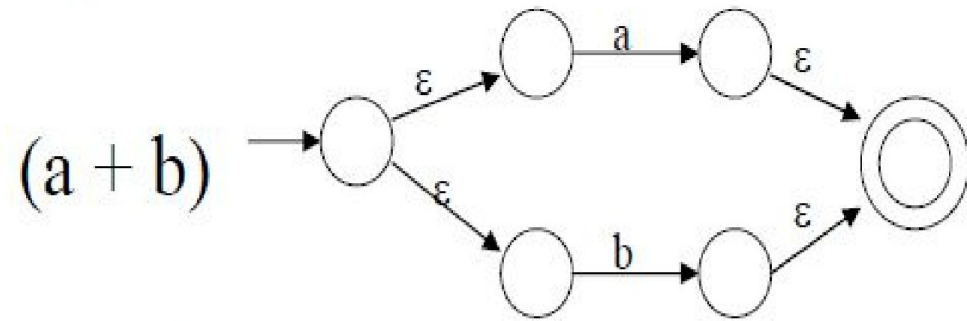
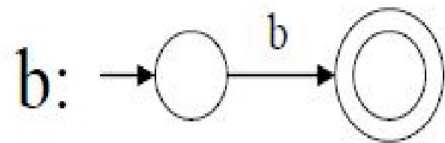
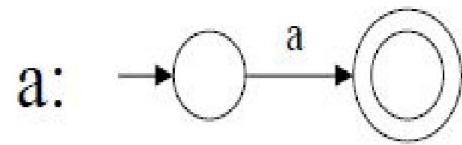
67

(C) For regular expression  $r^*$



Using rules 1 and 2, we construct NFAs for each basic symbol in the expression; we combine these basic NFAs using rule 3 to obtain an NFA for the entire expression

Example:- NFA construction of RE  $(a+b)^* a$



# NFA to DFA (Subset Construction)

69

- The subset construction algorithm converts an NFA into a DFA
- We use the following operations to keep the track of sets of NFA
  - $\epsilon\text{-closure}(s) \rightarrow$  the set of NFA states reachable from state  $s$  on  $\epsilon$ -transition
  - $\epsilon\text{-closure}(T) \rightarrow$  the set of NFA states reachable from state  $s$  in  $T$  on  $\epsilon$ -transition
  - $\text{move}(T, a) \rightarrow$  the set of NFA states to which there is transition on input  **$a$**  from state  $s$  in  $T$

# NFA to DFA

70

- The algorithm produces:
  - **Dstates** is the set of states of the new DFA consisting of sets of states of the NFA
  - **Dtran** is the transition table of the new DFA
- The start state of DFA is  $\varepsilon$ -closure( $q_0$ )
- A set of **Dstates** is an accepting state of DFA if it is a set of NFA states containing at least one accepting state of NFA

# Algorithm

71

Put  $\varepsilon$ -closure( $q_0$ ) as an unmarked state in  $Dstates$

**While** there is an unmarked state  $T$  in  $Dstates$   
**do**

    Mark  $T$

**For** each input symbol  $a \in \Sigma$  **do**

$U = \varepsilon$ -closure(move( $T$ ,  $a$ ))

**If**  $U$  is not in  $Dstates$  **then**

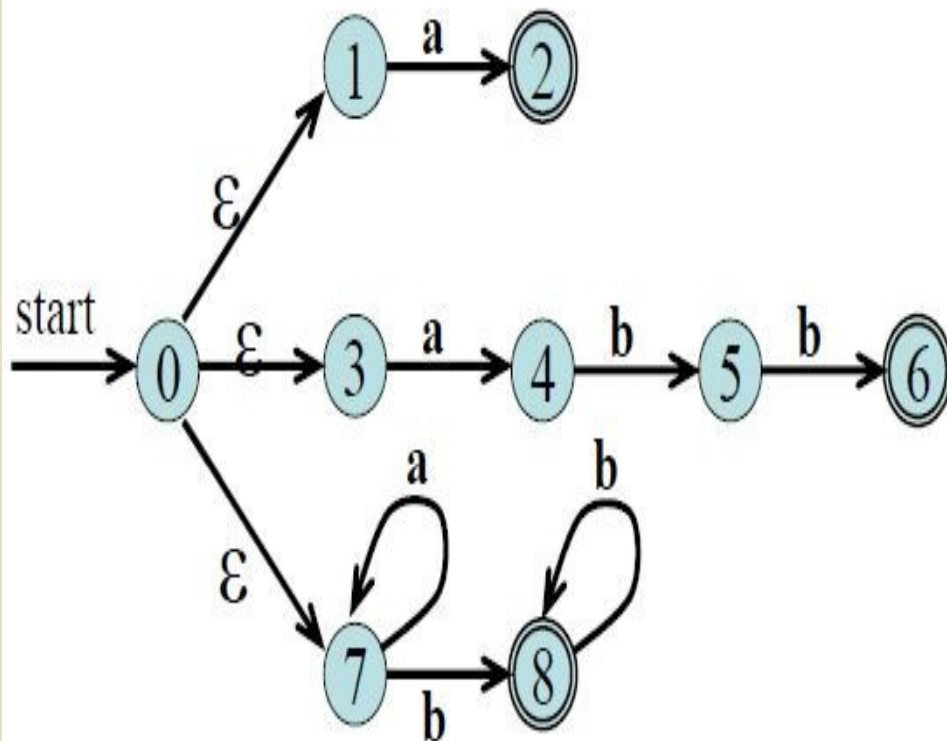
            Add  $U$  as an unmarked state to  
             $Dstates$

**End if**

$Dtran[T, a] = U$

**End do**  
  **End do**

## $\epsilon$ -closure and move Examples



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

$$\text{move}(\{0, 1, 3, 7\}, \mathbf{a}) = \{2, 4, 7\}$$

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

$$\text{move}(\{2, 4, 7\}, \mathbf{a}) = \{7\}$$

$$\epsilon\text{-closure}(\{7\}) = \{7\}$$

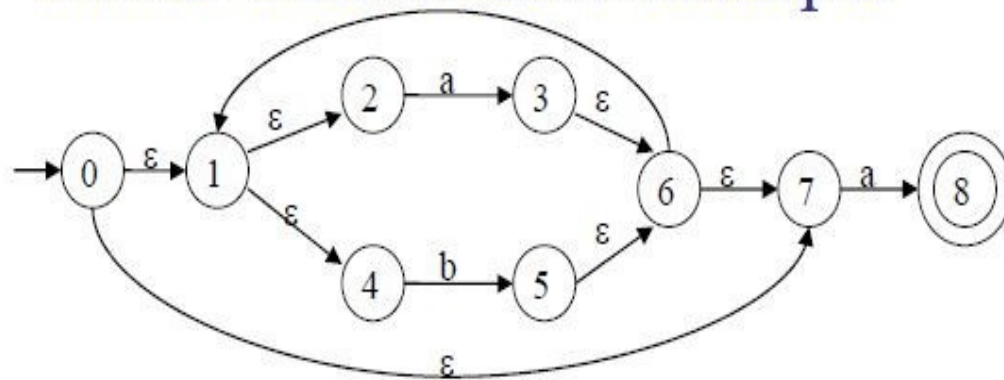
$$\text{move}(\{7\}, \mathbf{b}) = \{8\}$$

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

$$\text{move}(\{8\}, \mathbf{a}) = \emptyset$$



# Subset Construction Example



$$S_0 = \varepsilon\text{-closure}(\{0\}) = \{0,1,2,4,7\}$$

$S_0$  into  $Dstates$  as an unmarked state

$\Downarrow$  mark  $S_0$

$$\varepsilon\text{-closure}(\text{move}(S_0, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1 \quad S_1 \text{ into } Dstates$$

$$\varepsilon\text{-closure}(\text{move}(S_0, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2 \quad S_2 \text{ into } Dstates$$

$$Dtran[S_0, a] \leftarrow S_1 \quad Dtran[S_0, b] \leftarrow S_2$$

$\Downarrow$  mark  $S_1$

$$\varepsilon\text{-closure}(\text{move}(S_1, a)) = \varepsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = S_1$$

$$\varepsilon\text{-closure}(\text{move}(S_1, b)) = \varepsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\} = S_2$$

$$Dtran[S_1, a] \leftarrow S_1 \quad Dtran[S_1, b] \leftarrow S_2$$

$\Downarrow$  mark  $S_2$

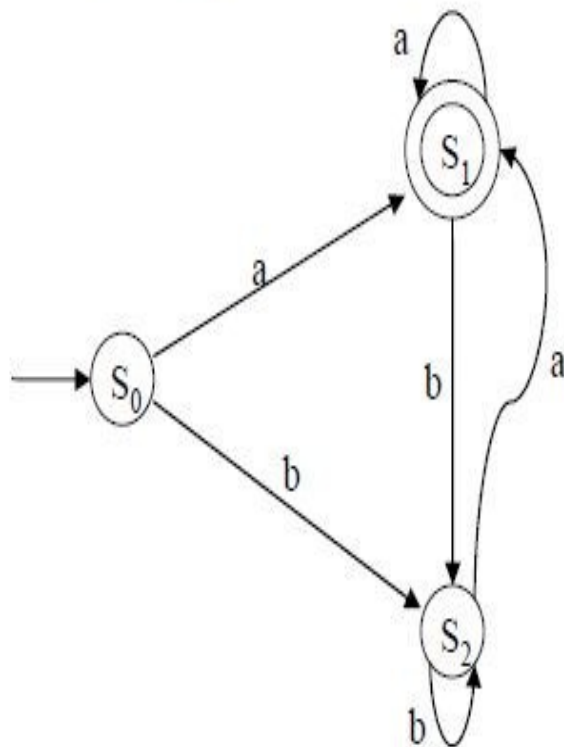
$\epsilon\text{-closure}(\text{move}(S_2, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = S_1$

$\epsilon\text{-closure}(\text{move}(S_2, b)) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = S_2$

$Dtran[S_2, a] \leftarrow S_1 \quad Dtran[S_2, b] \leftarrow S_2$

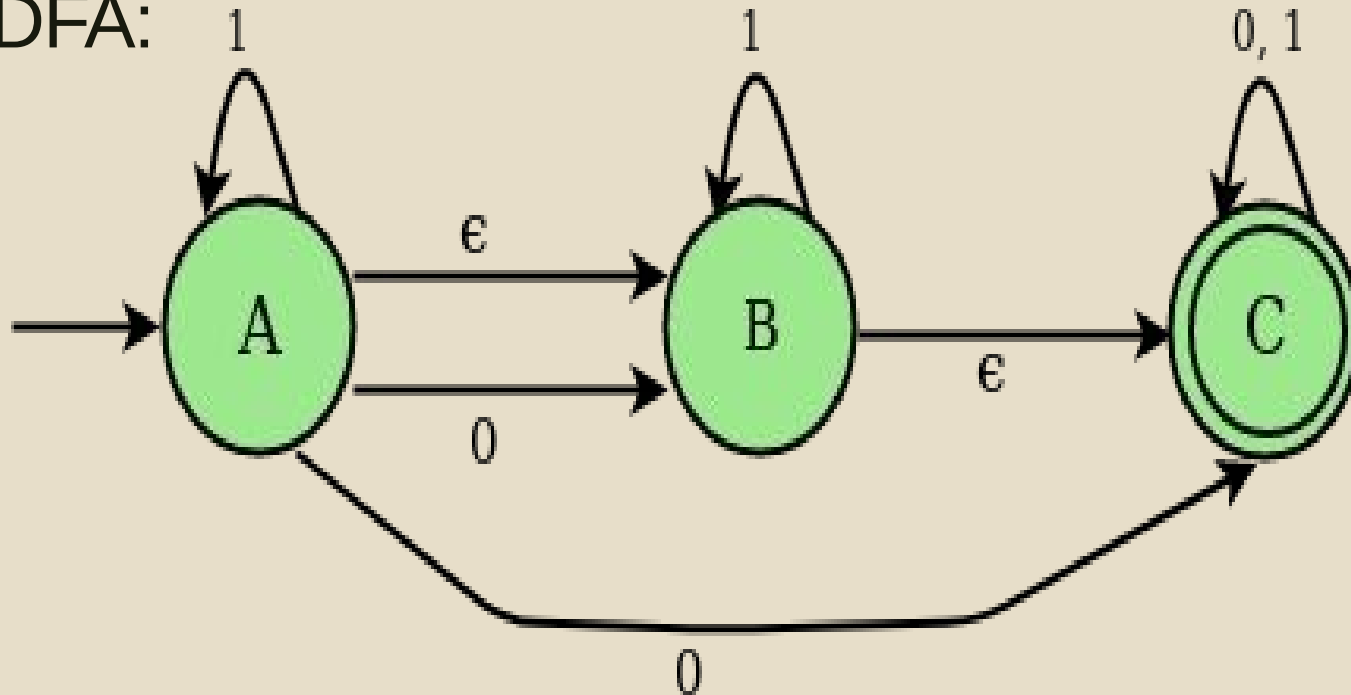
$S_0$  is the start state of DFA since 0 is a member of  $S_0 = \{0, 1, 2, 4, 7\}$

$S_1$  is an accepting state of DFA since 8 is a member of  $S_1 = \{1, 2, 3, 4, 6, 7, 8\}$



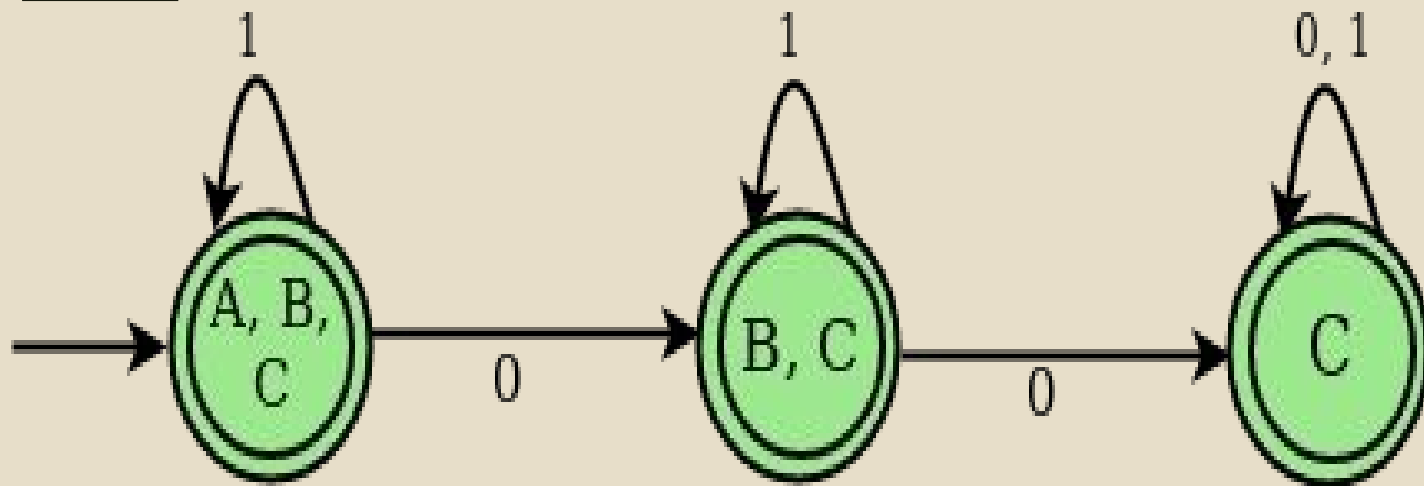
# Practice Question

Q) Convert the following NFA to DFA:



# Practice Question

Ans:



# RE to DFA (directly)

77

## Important States:

- The “important states” of an NFA are those without a null transition; that is, if  **$\text{move}(\{s\}, a) \neq \emptyset$**  for some  **$a$** , then  **$s$**  is an important state
- In an optimal state machine, all states are important states
- The subset construction algorithm uses only the important states when it determines  **$\epsilon\text{-closure}(\text{move}(T, a))$**

# RE to DFA

78

## Augmented Regular Expression:

- The  $\epsilon$ -NFA created from RE has exactly one accepting state and it does not have any transition i.e. it is not important state
- We introduce an “augmented character” **#** from the accepting state to make it an important state
- The regular expression **(r)#** is called the augmented regular expression of the original expression **r**

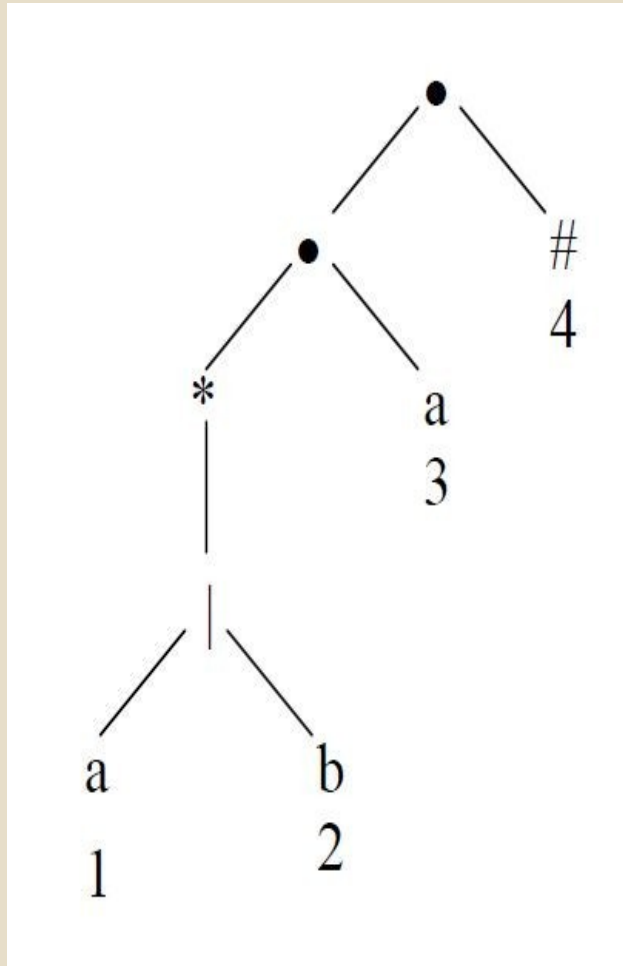
# Procedure:

1. Augment the given regular expression by concatenating it with special symbol # i.e  $r \rightarrow (r)\#$
2. Create the syntax tree for this augmented regular expression
  - *In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.*
3. Then each alphabet symbol (plus #) will be numbered (position numbers)
4. Traverse the tree to construct functions ***nullable***, ***firstpos***, ***lastpos***, and ***followpos***
5. Finally construct the DFA from the ***followpos***

# Syntax Tree Construction:

■  $(a|b)^*a \rightarrow (a|b)^*a\#$

[augmented regular expression]



Syntax tree of  $(a|b)^*a\#$

- each symbol is numbered (positions)
- each symbol is at a leaf
- inner nodes are operators

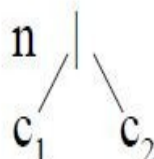
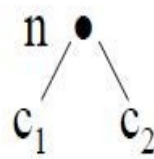
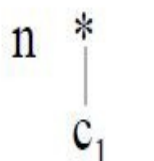


# firstpos, lastpos, nullable

81

- To evaluate ***followpos***, we need three functions to define the nodes (not just for leaves) of the syntax tree:
- **firstpos(n)** → the set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n
- **lastpos(n)** → the set of the positions of the **last** symbols of strings generated by the sub-expression rooted by n
- **nullable(n)** → **true** if the empty string is a member of strings generated by the sub-expression rooted by n, **false** otherwise

# Rules for calculating nullable, firstpos & lastpos

node <u>n</u>	<u>nullable(n)</u>	<u>firstpos(n)</u>	<u>lastpos(n)</u>
is leaf labeled $\epsilon$	true	$\Phi$	$\Phi$
is leaf labeled with position $i$	false	$\{i\}$ (position of leaf node)	$\{i\}$
	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	<b>if</b> $(\text{nullable}(c_1))$ <b>then</b> $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$ <b>else</b> $\text{firstpos}(c_1)$	<b>if</b> $(\text{nullable}(c_2))$ <b>then</b> $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ <b>else</b> $\text{lastpos}(c_2)$
	true	$\text{firstpos}(c_1)$	$\text{lastpos}(c_1)$

a.b.(a+b)\*.

(a+b)\*.b.a.#

1 2 3 4      5 6 7

8 9

#

9

a

8

b

7

\*

+

a

5

b

6

+

a

3

b

4

b

2

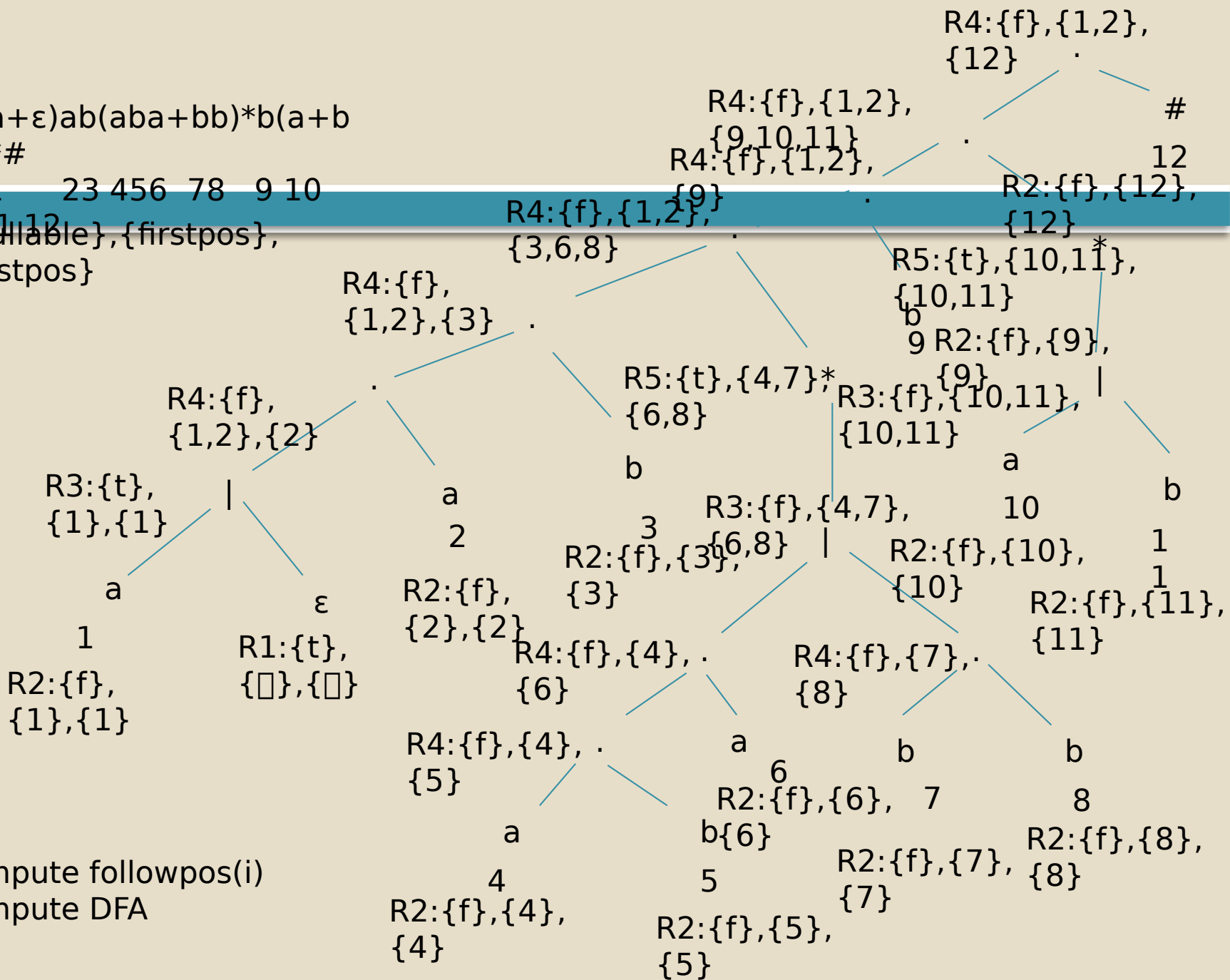
a

1

Compute nullable,  
firstpos, lastpos  
Compute followpos  
Construct DFA

$(a+\epsilon)ab(aba+bb)^*b(a+b)^*\#$

1 23 456 78 9 10  
 {nullable}, {firstpos},  
 {lastpos}



## Evaluating **followpos**

**for** each node  $n$  in the tree **do**

**if**  $n$  is a cat-node with left child  $c1$  and right child  $c2$

**then for** each  $i$  in  $lastpos(c1)$  **do**

$followpos(i) := followpos(i) \cup firstpos(c2)$

**end do**

**else if**  $n$  is a star-node

**for** each  $i$  in  $lastpos(n)$  **do**

$followpos(i) := followpos(i) \cup firstpos(n)$

**end do**

**end if**

**end do**

Compute the **followpos** bottom-up for each node of  
Syntax tree

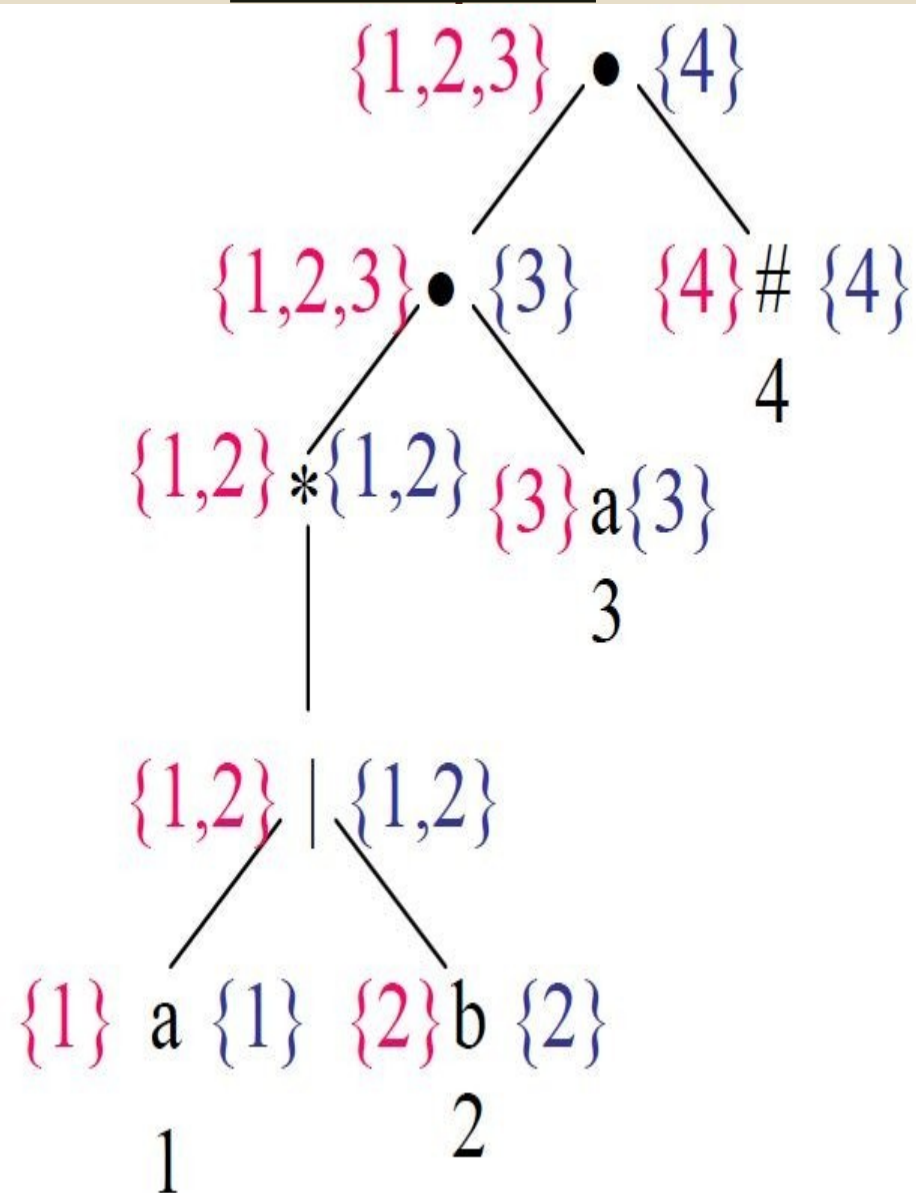
# followpos

86

- We define the function **followpos** for the positions (positions assigned to leaves)
- **followpos(i)** → is the set of positions which can follow the position **i** in the strings generated by the augmented regular expression
- For example,  $(a \mid b)^* a \#$   
                            1  2  3 4
- $\text{followpos}(1) = \{1, 2, 3\}$
- $\text{followpos}(2) = \{1, 2, 3\}$
- $\text{followpos}(3) = \{4\}$
- $\text{followpos}(4) = \{ \}$

## Evaluating followpos

example:



## red - firstpos

## blue - lastpos

Then we can calculate

**followpos**

```
followpos(1) = {1,2,3}
```

```
followpos(2) = {1,2,3}
```

followpos(3) = {4}

```
followpos(4) = { }
```

After we calculate **followpos**, we

## Algorithm for converting RE to DFA

1. Create the syntax tree of  $(r)\#$
2. Calculate the functions: **nullable**, **firstpos**, **lastpos** & **followpos**
3. Put **firstpos(root)** into the states of DFA as an unmarked state



4. *while* (there is an unmarked state  $S$  in the states of DFA) *do*

- mark  $S$
- *for each* input symbol  $a$  *do*
  - let  $s_1, \dots, s_n$  are positions in  $S$  and symbols in those positions is  $a$
  - $S' \leftarrow \text{followpos}(s_1) \cup \dots \cup \text{followpos}(s_n)$
  - $\text{move}(S, a) \leftarrow S'$
  - if ( $S'$  is not empty and not in the states of DFA)
    - put  $S'$  into the states of DFA as an unmarked state

*The start state of DFA is  $\text{firstpos}(\text{root})$*

*The accepting states of DFA are all states containing*

# Example1

For the RE  $(a|b)^*a\#$   
1    2    3 4

$\text{followpos}(1)=\{1,2,3\}, \text{followpos}(2)=\{1,2,3\}, \text{followpos}(3)=\{4\}, \text{followpos}(4)=\{\}$

$S_1 = \text{firstpos}(\text{root}) = \{1,2,3\}$

mark  $S_1$

for a:  $\text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = S_2$      $\text{move}(S_1, a) = S_2$

for b:  $\text{followpos}(2) = \{1,2,3\} = S_1$      $\text{move}(S_1, b) = S_1$

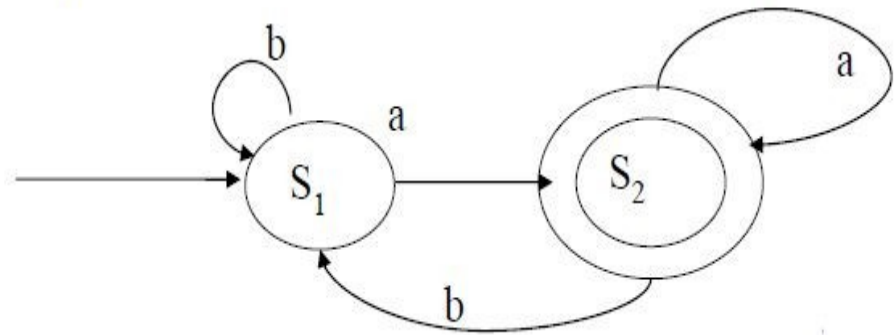
mark  $S_2$

for a:  $\text{followpos}(1) \cup \text{followpos}(3) = \{1,2,3,4\} = S_2$      $\text{move}(S_2, a) = S_2$

for b:  $\text{followpos}(2) = \{1,2,3\} = S_1$      $\text{move}(S_2, b) = S_1$

start state:  $S_1$

accepting states:  $\{S_2\}$



## Example2

For RE----  $(a \mid \epsilon) b c^* \#$

$\text{followpos}(1)=\{2\}$      $\text{followpos}(2)=\{3,4\}$      $\text{followpos}(3)=\{3,4\}$      $\text{followpos}(4)=\{\}$

$S_1 = \text{firstpos}(\text{root}) = \{1,2\}$

mark  $S_1$

for  $a$ :  $\text{followpos}(1)=\{2\}=S_2$      $\text{move}(S_1, a)=S_2$

for  $b$ :  $\text{followpos}(2)=\{3,4\}=S_3$      $\text{move}(S_1, b)=S_3$

mark  $S_2$

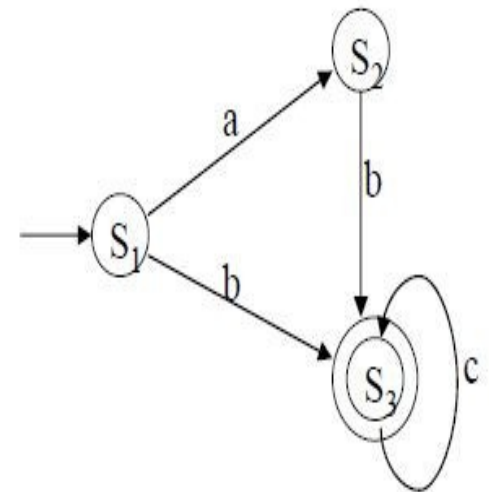
for  $b$ :  $\text{followpos}(2)=\{3,4\}=S_3$      $\text{move}(S_2, b)=S_3$

mark  $S_3$

for  $c$ :  $\text{followpos}(3)=\{3,4\}=S_3$      $\text{move}(S_3, c)=S_3$

start state:  $S_1$

accepting states:  $\{S_3\}$



# State Minimization in DFA

92

- DFA minimization refers to the task of transforming a given DFA into an equivalent DFA which has minimum number of states
- Two states **p** and **q** are called **equivalent** if for all input strings **w**,  $\delta(\mathbf{p}, \mathbf{w})$  is an accepting state iff  $\delta(\mathbf{q}, \mathbf{w})$  is an accepting state
- Otherwise they are called **distinguishable** states

# State Minimization in DFA

93

- String  $w$  distinguishes state  $s$  from state  $t$  if, by starting with DFA  $M$  in state  $s$  and feeding it input  $w$ , we end up in an accepting state, but starting in state  $t$  and feeding it with same input  $w$ , we end up in a non accepting state, or vice-versa
- The procedure finds the states that can be distinguished by some input string
- Each group of states that cannot be distinguished is then merged into a single state

# State Minimization in DFA

94

Suppose there is a DFA  $D = \langle Q, \Sigma, q_0, \delta, F \rangle$  which recognizes a language  $L$ . Then the minimized DFA  $D = \langle Q', \Sigma, q_0, \delta', F' \rangle$  can be constructed for language  $L$  as:

**Step 1:** Divide  $Q$  (set of states) into two sets. One set will contain all final states and the other set will contain all non-final states. This partition is called  $P_0$ .

**Step 2:** Initialize  $k = 1$

# State Minimization in DFA

95

**Step 3:** Find  $\mathbf{P}_k$  by partitioning the different sets of  $\mathbf{P}_{k-1}$ . In each set of  $\mathbf{P}_{k-1}$ , take all possible pair of states. If two states of a set are distinguishable, split the states into different sets in  $\mathbf{P}_k$ .

**Step 4:** Stop when  $\mathbf{P}_k = \mathbf{P}_{k-1}$  (No change in partition)

**Step 5:** All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in  $\mathbf{P}_k$ .

# State Minimization in DFA

96

- In addition to the procedure, we also remove the following states from the DFA:
  - Unreachable State: A state that cannot be reached through any transition from any other state in the DFA
  - Dead State: A **non-final** state, that when an automata reaches, cannot transit into any other state



# State Minimization in DFA

97

- How to find whether two states in partition  $P_k$  are distinguishable?
  - *Two states  $(qi, qj)$  are distinguishable in partition  $P_k$  if for any input symbol  $a$ ,  $\delta(qi, a)$  and  $\delta(qj, a)$  are in different sets in partition  $P_{k-1}$*

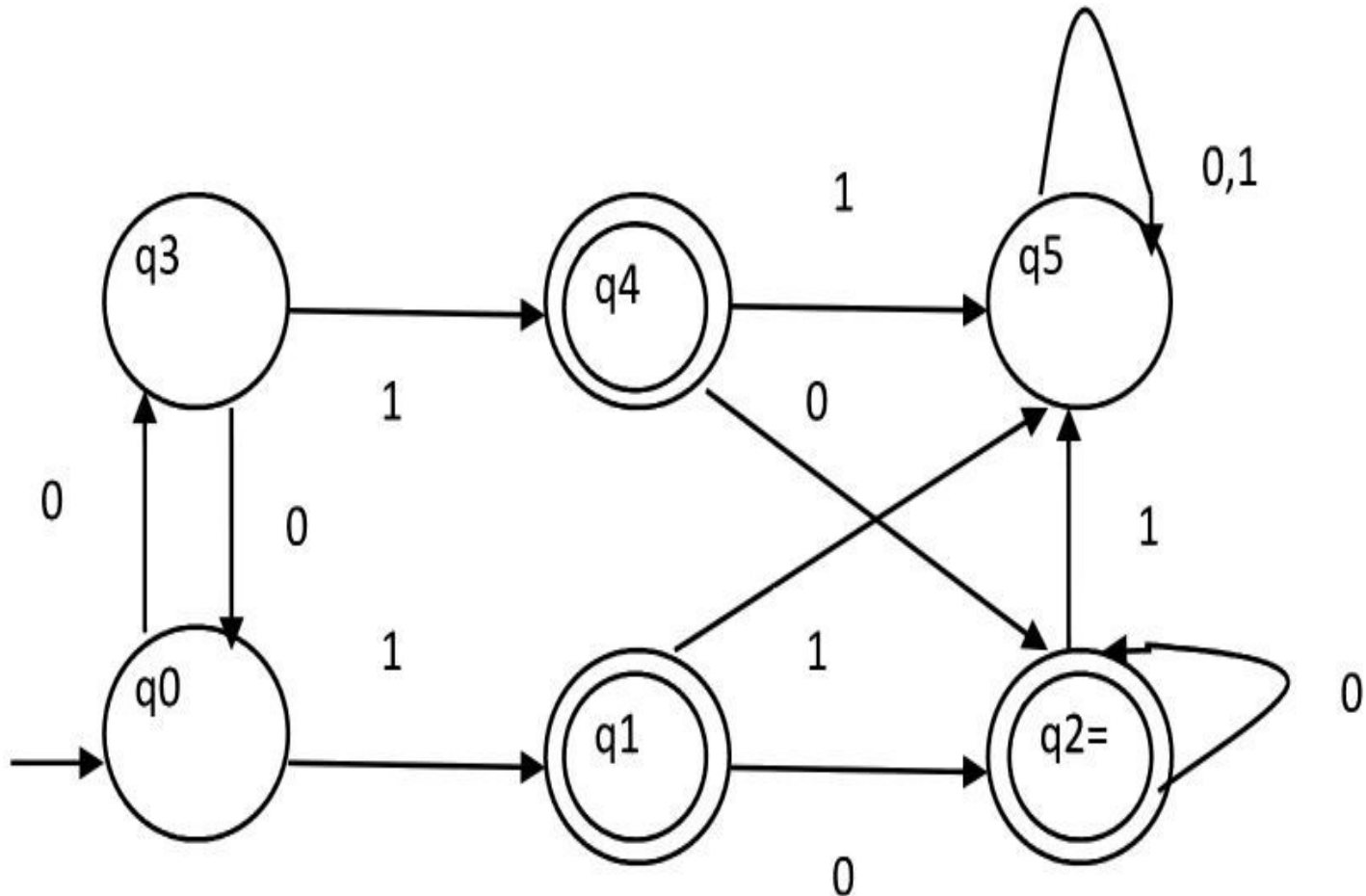
# State Minimization in DFA

98

- Start state of the minimized DFA is the group containing the start state of the original DFA
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA

## Example

Consider the following DFA shown in figure.



1.  $P_0$  will have two sets of states. One set will contain  $q1, q2, q4$  which are final states of DFA and another set will contain remaining states. So  $P_0 = \{ \{ q1, q2, q4 \}, \{ q0, q3, q5 \} \}$ .

2 To calculate  $P_1$ , we will check whether sets of partition  $P_0$  can be partitioned or not:

**For set  $\{ q1, q2, q4 \}$  :**

$\delta ( q1, 0 ) = \delta ( q2, 0 ) = q2$  and  $\delta ( q1, 1 ) = \delta ( q2, 1 ) = q5$ , So  $q1$  and  $q2$  are not distinguishable.

Similarly,  $\delta ( q1, 0 ) = \delta ( q4, 0 ) = q2$  and  $\delta ( q1, 1 ) = \delta ( q4, 1 ) = q5$ , So  $q1$  and  $q4$  are not distinguishable.

Since,  $q1$  and  $q2$  are not distinguishable and  $q1$  and  $q4$  are also not distinguishable, So  $q2$  and  $q4$  are not distinguishable. So,  $\{ q1, q2, q4 \}$  set will not be partitioned in  $P_1$ .

**ii) For set  $\{ q0, q3, q5 \}$  :**

$\delta ( q0, 0 ) = q3$  and  $\delta ( q3, 0 ) = q0$   $\delta ( q0, 1 ) = q1$  and  $\delta ( q3, 1 ) = q4$

Moves of  $q_0$  and  $q_3$  on input symbol 0 are  $q_3$  and  $q_0$  respectively which are in same set in partition  $P_0$ . Similarly, Moves of  $q_0$  and  $q_3$  on input symbol 1 are  $q_1$  and  $q_4$  which are in same set in partition  $P_0$ . So,  $q_0$  and  $q_3$  are not distinguishable.

$\delta(q_0, 0) = q_3$  and  $\delta(q_5, 0) = q_5$  and  $\delta(q_0, 1) = q_1$  and  $\delta(q_5, 1) = q_5$ . Moves of  $q_0$  and  $q_5$  on input symbol 1 are  $q_1$  and  $q_5$  respectively which are in different sets in partition  $P_0$ . So,  $q_0$  and  $q_5$  are distinguishable. So, set  $\{q_0, q_3, q_5\}$  will be partitioned into  $\{q_0, q_3\}$  and  $\{q_5\}$ . So,

$$P_1 = \{ \{q_1, q_2, q_4\}, \{q_0, q_3\}, \{q_5\} \}$$

To calculate  $P_2$ , we will check whether sets of partition  $P_1$  can be partitioned or not:

**iii) For set  $\{q_1, q_2, q_4\}$  :**

$\delta(q_1, 0) = \delta(q_2, 0) = q_2$  and  $\delta(q_1, 1) = \delta(q_2, 1) = q_5$ , So  $q_1$  and  $q_2$  are not distinguishable.

Similarly,  $\delta(q_1, 0) = \delta(q_4, 0) = q_2$  and  $\delta(q_1, 1) = \delta(q_4, 1) = q_5$ , So  $q_1$  and  $q_4$  are not distinguishable.

Since,  $q_1$  and  $q_2$  are not distinguishable and  $q_1$  and  $q_4$  are also not distinguishable, So  $q_2$  and  $q_4$  are not distinguishable. So,  $\{q_1, q_2, q_4\}$  set will not be partitioned in  $P_2$ .

**iv) For set  $\{q_0, q_3\}$  :**

$\delta(q_0, 0) = q_3$  and  $\delta(q_3, 0) = q_0$

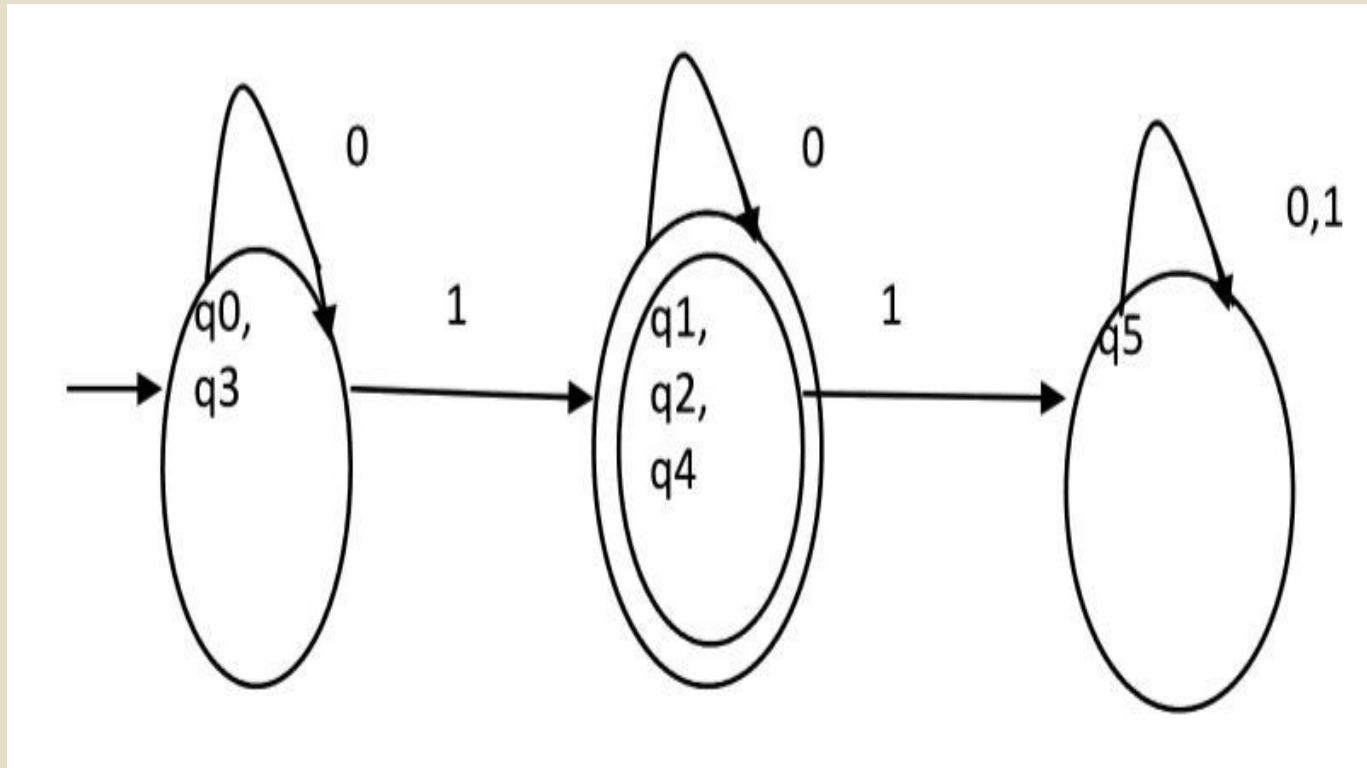
$\delta(q_0, 1) = q_1$  and  $\delta(q_3, 1) = q_4$

Moves of  $q_0$  and  $q_3$  on input symbol 0 are  $q_3$  and  $q_0$  respectively which are in same set in partition  $P_1$ . Similarly, Moves of  $q_0$  and  $q_3$  on input symbol 1 are  $q_1$  and  $q_4$  which are in same set in partition  $P_1$ . So,  $q_0$  and  $q_3$  are not distinguishable.

**v) For set  $\{q_5\}$ :**

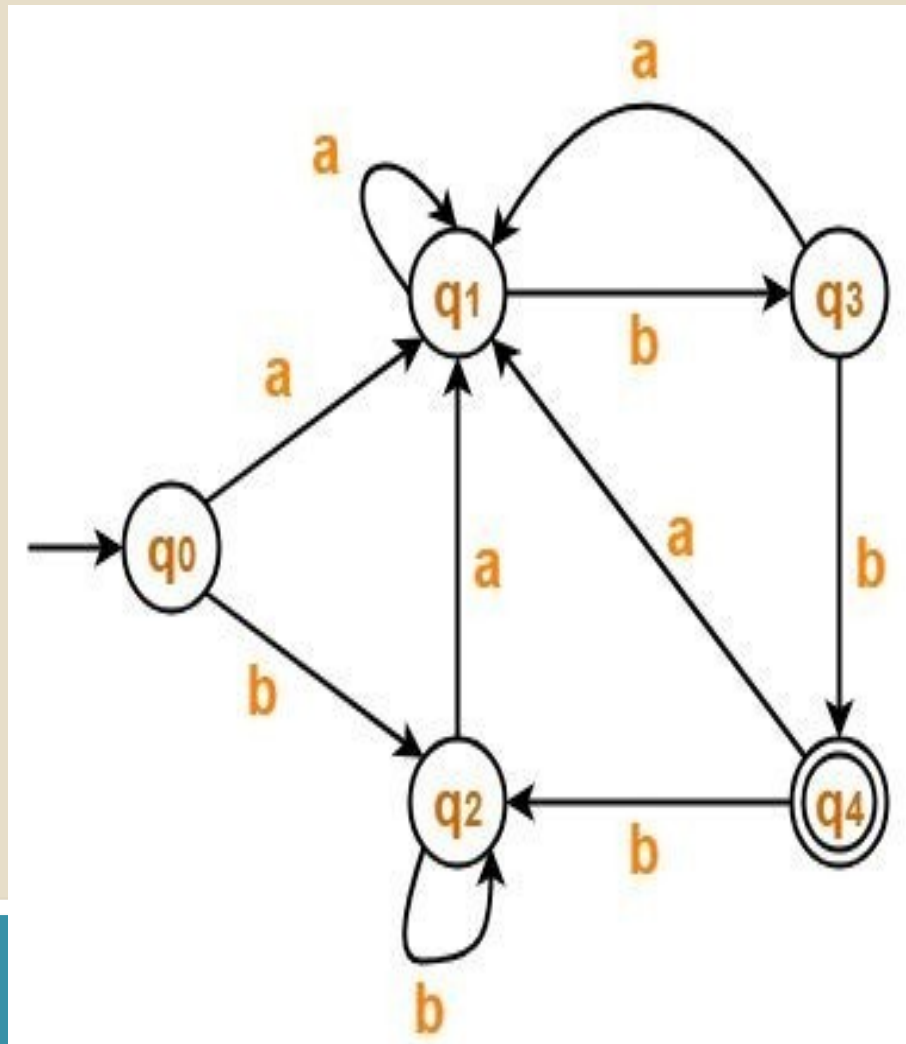
Since we have only one state in this set, it can't be further partitioned. So,  
 $P_2 = \{\{q_1, q_2, q_4\}, \{q_0, q_3\}, \{q_5\}\}$

Here,  $P_1 = P_2$ . So, this is the final partition.  
Partition  $P_2$  means that  $q_1$ ,  $q_2$  and  $q_4$  states are merged into one. Similarly,  $q_0$  and  $q_3$  are merged into one. Minimized DFA corresponding to the given input DFA is shown below:



# Practice Question

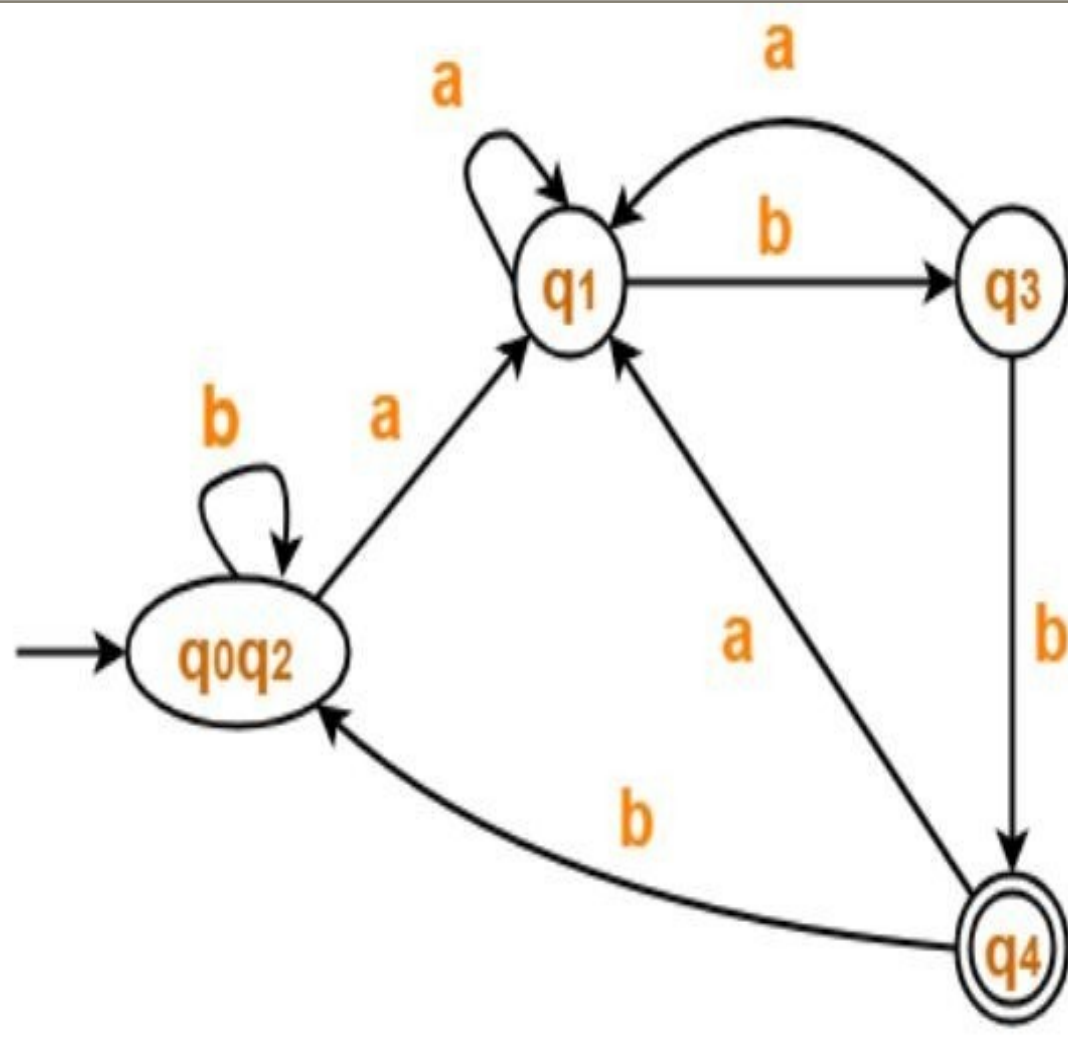
Q) Minimize the following DFA:





An

S:



# Summary: Specification and Recognition

10  
6

Consider the following grammar:

```
stmt  →  if expr then stmt  
        |  if expr then stmt else stmt  
        |   $\epsilon$   
expr  →  term relop term  
        |  term  
term  →  id  
        |  number
```

A grammar for branching statements

- Tokens in the above grammar are:

<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>	→	if
<i>then</i>	→	then
<i>else</i>	→	else
<i>relop</i>	→	<   >   <=   >=   =   <>

Patterns for tokens of branching statements

- Whitespace is identified as shown below:

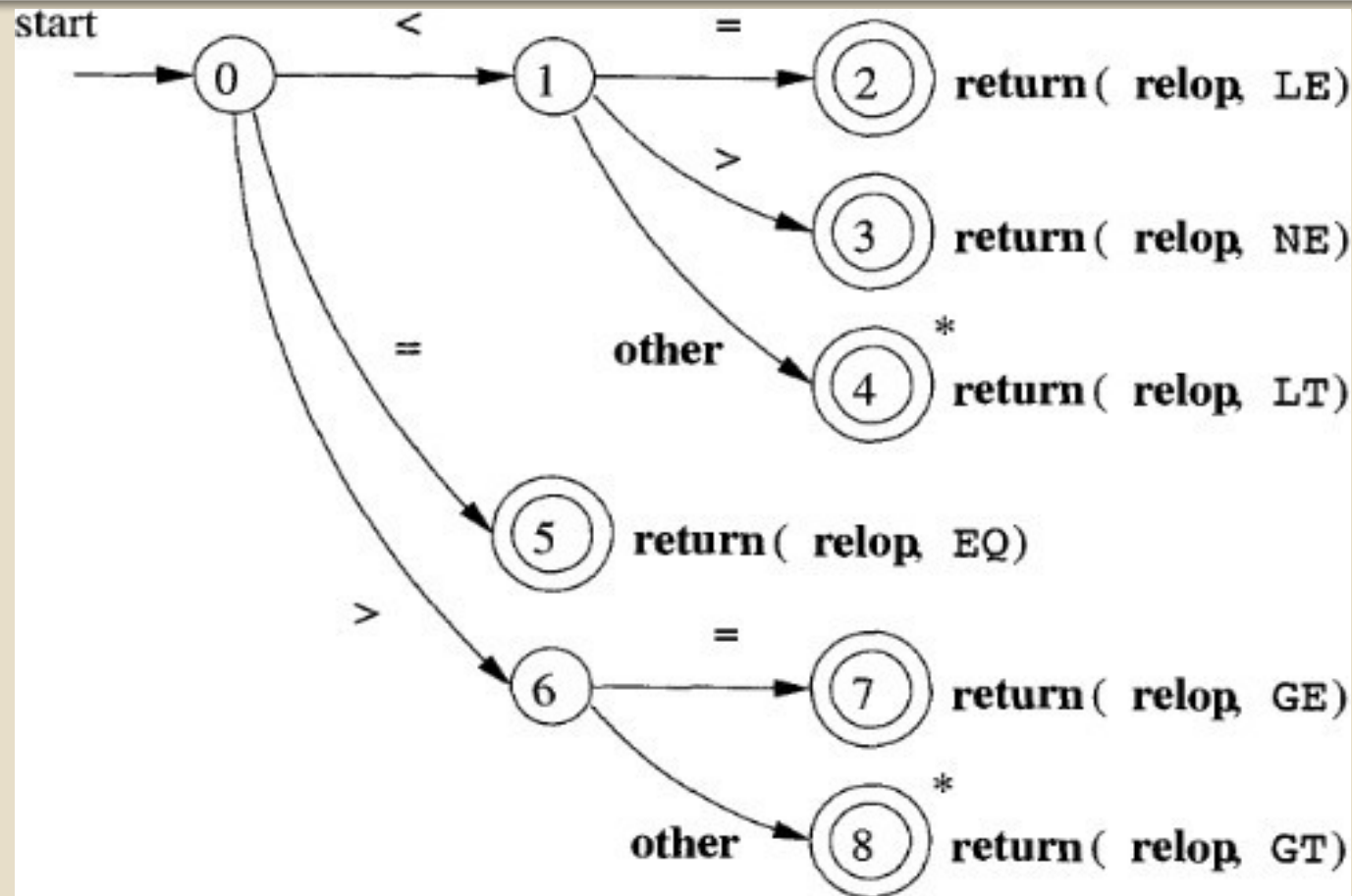
$$ws \rightarrow ( \text{blank} \mid \text{tab} \mid \text{newline} )^+$$

# REGULAR-EXPRESSION PATTERNS FOR TOKENS

10  
8

Regular Expression	Token	Attribute-value
<b>ws</b>	--	--
<b>if</b>	<b>if</b>	--
<b>then</b>	<b>then</b>	--
<b>id</b>	<b>id</b>	pointer to table entry
<b>num</b>	<b>num</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

In the construction of lexical analyzer we first convert the patterns into flowcharts called “transition diagrams”.



Transition diagram for **relop**

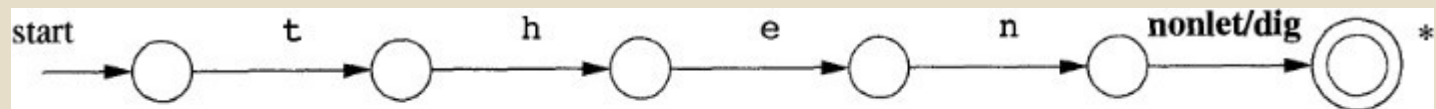
# Implementation of Transition diagram

11

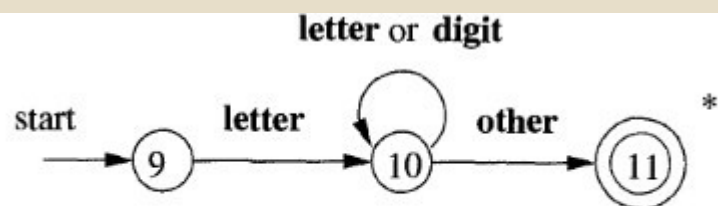
0

```
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:
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

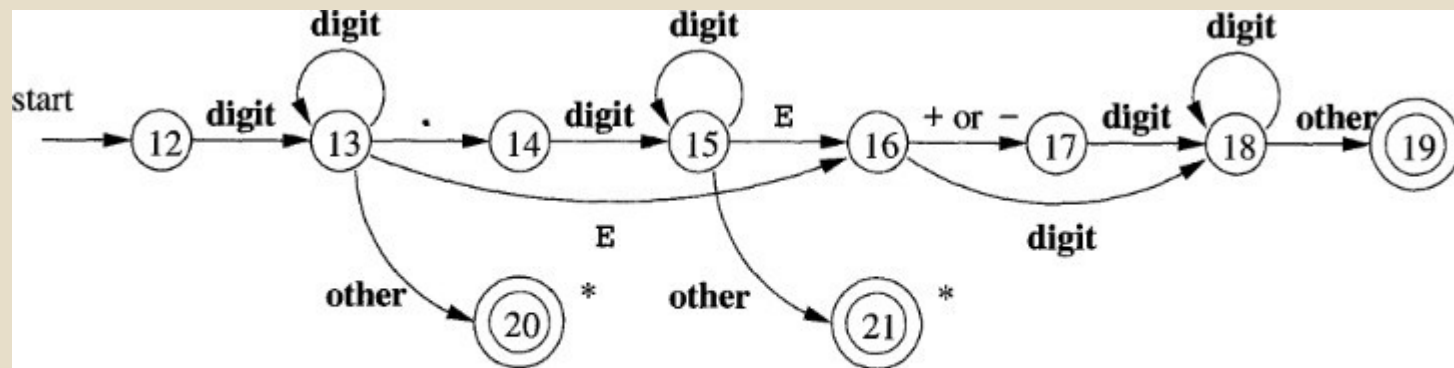
Sketch of implementation of **relop** transition diagram



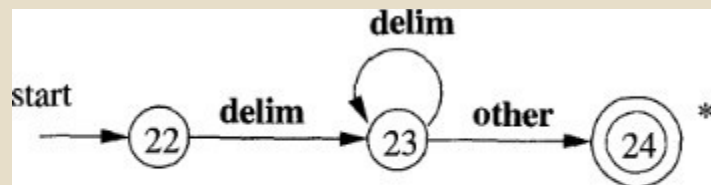
Hypothetical transition diagram for the keyword then



A transition diagram for id's and keywords



A transition diagram for unsigned numbers



A transition diagram for whitespace