# Compiler Design

# Lecture-4

Introdution to Lexical Analysis

## Topics Covered

- Approaches to implement a lexical analyzer
- The role of the lexical analyzer
- Transition Diagram
- Finite Automata

## Section 0 Approaches to implement a lexical analyzer

1. Simple approach
   - Construct a diagram that illustrates the structure of the tokens of the source language , and then to hand-translate the diagram into a program for finding tokens

Notes: Efficient lexical analyzers can be produced in this manner

## Section 0 Approaches to implement a lexical analyzer

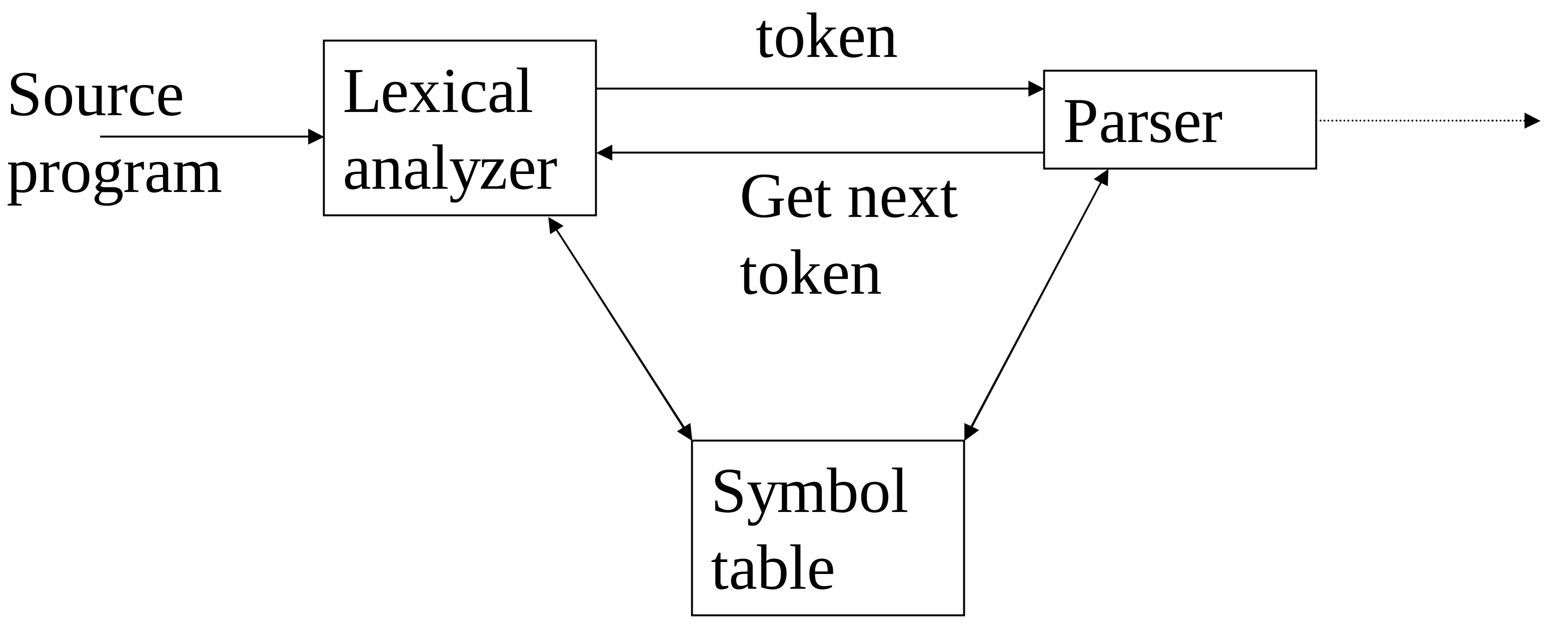
2. Pattern-directed programming approach
   - Pattern Matching technique
   - Specify and design program that execute actions triggered by patterns in strings
   - Introduce a pattern-action language called Lex for specifying lexical analyzers
     - Patterns are specified by regular expressions
     - A compiler for Lex can generate an efficient finite automation recognizer for the regular expressions

## First phase of a compiler

1. Main task
   - To read the input characters
   - To produce a sequence of tokens used by the parser for syntax analysis
   - As an assistant of parser

# The role of the lexical analyzer

## 2. Interaction of lexical analyzer with parser



## The role of the lexical analyzer

3. Processes in lexical analyzers
   - Scanning
     - Pre-processing
       - Strip out comments and white space
       - Macro functions
   - Correlating error messages from compiler with source program
     - A line number can be associated with an error message
   - Lexical analysis

# CHAPTER 3 LEXICAL ANALYSIS

## Section 1 The role of the lexical analyzer

4. Terms of the lexical analyzer

– Token

- Types of words in source program
- Keywords, operators, identifiers, constants, literal strings, punctuation symbols(such as commas,semicolons)

– Lexeme

- Actual words in source program

– Pattern

- A rule describing the set of lexemes that can represent a particular token in source program
- **Relation** {<.<=,>,>=,==,<>}

# The role of the lexical analyzer

## 5. Attributes for Tokens

–A pointer to the symbol-table entry in which the information about the token is kept

E.g  E=M*C**2

<**id**, pointer to symbol-table entry for E>

<**assign_op**,>

<**id**, pointer to symbol-table entry for M>

<**multi_op**,>

<**id**, pointer to symbol-table entry for C>

<**exp_op**,>

<**num**,integer value 2>

## The role of the lexical analyzer

6. Lexical Errors
   - Deleting an extraneous character
   - Inserting a missing character
   - Replacing an incorrect character by a correct character
   - Transposing two adjacent characters(such as , fi=>if)
   - Pre-scanning

## The role of the lexical analyzer

7. Input Buffering
   - Two-buffer input scheme to look ahead on the input and identify tokens
   - Buffer pairs
   - Sentinels(Guards)

## Specification of Tokens

1. Regular Definition of Tokens
   - Defined in regular expression

**e.g. Id** $\rightarrow$ **letter**(**letter**|**digit**)

**letter** $\rightarrow$A|B|…|Z|a|b|…|z

**digit** $\rightarrow$0|1|2|…|9

Notes: Regular expressions are an important notation for specifying patterns. Each pattern matches  a set of strings, so regular expressions will serve as as names for sets of strings.

# Specification of Tokens

2. Regular Expression & Regular language
   - Regular Expression
     - A notation that allows us to define a pattern in a high level language.
   - Regular language
     - Each regular expression r denotes a language L(r) (the set of sentences relating to the regular expression r)

Notes: Each word in a program can be expressed in a regular expression

## Specification of Tokens

3. The rule of regular expression over alphabet Σ

1) ε is a regular expression that denote {ε}
   - ε is regular expression
   - {ε} is the related regular language

2) If *a* is a symbol in Σ, then *a* is a regular expression that denotes {*a*}
   - a is regular expression
   - {a} is the related regular language

## Specification of Tokens

3. The rule of regular expression over alphabet $\Sigma$

3) Suppose $\alpha$ and $\beta$ are regular expressions, then $\alpha|\beta$, $\alpha\beta$, $\alpha^*$ ,$\beta^*$ is also a regular expression

Notes: Rules 1) and 2) form the basis of the definition; rule 3) provides the inductive step.

# Specification of Tokens

## 4. Algebraic laws of regular expressions

1) $\alpha|\beta = \beta|\alpha$

2) $\alpha|(\beta|\gamma)=(\alpha|\beta)|\gamma \quad \alpha(\beta\gamma) = (\alpha\,\beta)\gamma$

3) $\alpha(\beta|\gamma) = \alpha\beta \mid \alpha\gamma \qquad (\alpha|\beta)\gamma = \alpha\gamma \mid \beta\gamma$

4) $\varepsilon\alpha = \alpha\varepsilon = \alpha$

5) $(\alpha^*)^* = \alpha^*$

6) $\alpha^* = \alpha^+|\varepsilon \qquad \alpha^+ = \alpha\,\alpha^* = \alpha^*\alpha$

7) $(\alpha|\beta)^* = (\alpha^* \mid \beta^*)^* = (\alpha^*\,\beta^*)^*$

# Specification of Tokens

## 4. Algebraic laws of regular expressions

8) If $\varepsilon \notin L(\gamma)$, then

$\alpha = \beta | \gamma \alpha \iff \alpha = \gamma^* \beta$

$\alpha = \beta | \alpha \gamma \iff \alpha = \beta \gamma^*$

Notes: We assume that the precedence of * is the highest, the precedence of | is the lowest and they are left associative

# Specification of Tokens

5. Notational Short-hands

a)One or more instances

$(r)^+$ digit$^+$

b)Zero or one instance

r? is a shorthand for r|ε (E(+|-)?digits)?

c)Character classes

[a-z] denotes a|b|c|…|z

[A-Za-z] [A-Za-z0-9]

## Recognition of Tokens

1. Task of recognition of token in a lexical analyzer
   - Isolate the lexeme for the next token in the input buffer
   - Produce as output a pair consisting of the appropriate token and attribute-value, such as <id,pointer to table entry> , using the translation table given in the Fig in next page

# Recognition of Tokens

1. Task of recognition of token in a lexical analyzer

| Regular expression | Token | Attribute-value |
|---|---|---|
| if | **if** | - |
| **id** | **id** | Pointer to table entry |
| < | **relop** | LT |

# Recognition of Tokens

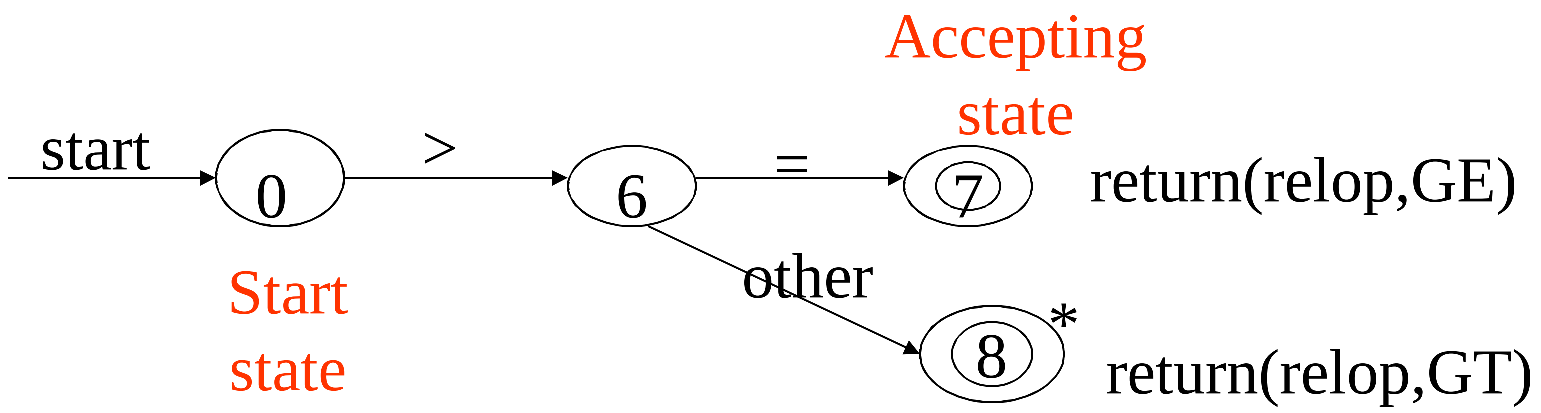
2. Methods to recognition of token
   - Use Transition Diagram

# CHAPTER 3 LEXICAL ANALYSIS

## Section 3 Recognition of Tokens

### 3. Transition Diagram(Stylized flowchart)

– Depict the actions that take place when a lexical analyzer is called by the parser to get the next token



Notes: Here we use '*' to indicate states on which input retraction must take place

# Recognition of Tokens

## 4. Implementing a Transition Diagram

- Each state gets a segment of code
- If there are edges leaving a state, then its code reads a character and selects an edge to follow, if possible
- Use nextchar() to read next character from the input buffer

# Recognition of Tokens

4. Implementing a Transition Diagram

```
while (1) {
  switch(state) {
    case 0: c=nextchar();
            if (c==blank || c==tab || c==newline){
                state=0;lexeme_beginning++}
            else if (c== ‘<‘) state=1;
            else if (c==‘=‘)  state=5;
            else if(c==‘>’) state=6 else state=fail();
             break
     case 9: c=nextchar();
            if (isletter( c)) state=10;
            else state=fail(); break
  … }}}
```

# Recognition of Tokens

## 5. A generalized transition diagram

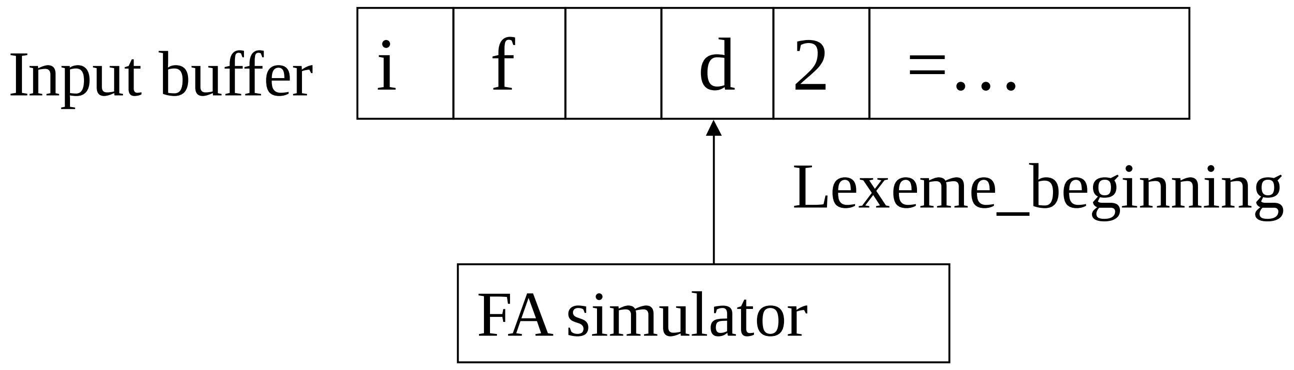
Finite Automation

– Deterministic or non-deterministic FA

– Non-deterministic means that more than one transition out of a state may be possible on the the same input symbol

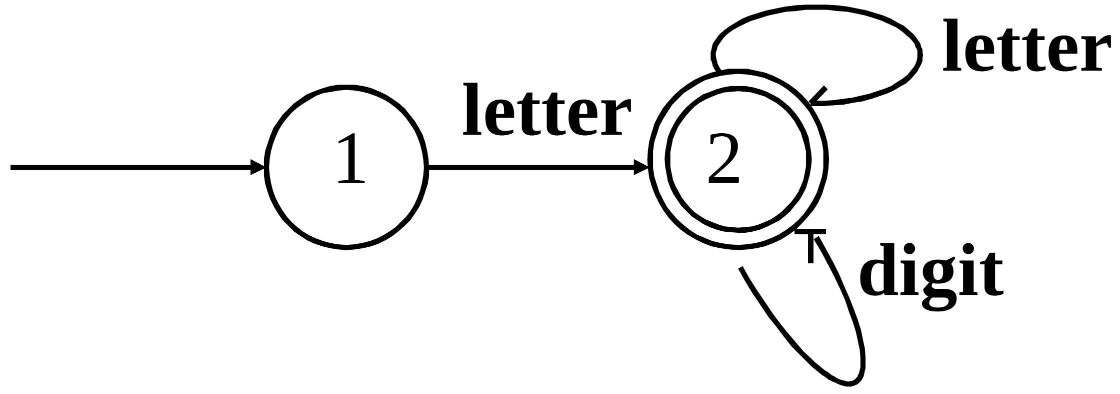
# Recognition of Tokens

6. The model of recognition of tokens



# Recognition of Tokens

E.g：The FA simulator for Identifiers is:



– Which represent the rule:
**identifier=letter(letter|digit)***

## Finite automata

1. Usage of FA
   - Precisely recognize the regular sets
   - A regular set is a set of sentences relating to the regular expression
2. Sorts of FA
   - Deterministic FA
   - Non-deterministic FA

# Finite automata

3. Deterministic FA (DFA)

DFA is a quintuple, M(S,Σ,move,$s_0$,F）

– S: a set of *states*

– Σ: the input symbol alphabet

– move: a transition function, mapping from S× Σ to S, *move(s,a)=s'*

– $s_0$: the *start* state, $s_0 \in S$

– F: a set of states *F* distinguished as *accepting* states, $F \subseteq S$

## Finite automata

3. Deterministic FA (DFA)

Note: 1) In a DFA, no state has an ε-transition;

2)In a DFA, for each state *s* and input symbol *a*, there is at most one edge labeled *a* leaving *s*

3)To describe a FA,we use the transition graph or transition table

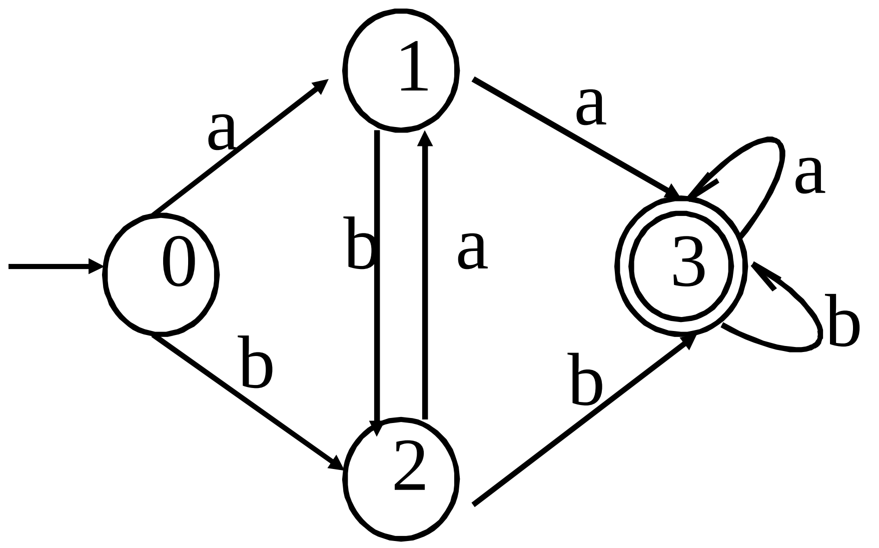
4)A DFA accepts an input string x if and only if there is some path in the transition graph from start state to some accepting state e.g. DFA M=({0,1,2,3},{a,b},move,0,{3})

Move: move(0,a)=1 m(0,b)=2 m(1,a)＝3 m(1,b)＝2

m(2,a)=1 m(2,b)=3 m(3,a)＝3 m(3,b)＝3

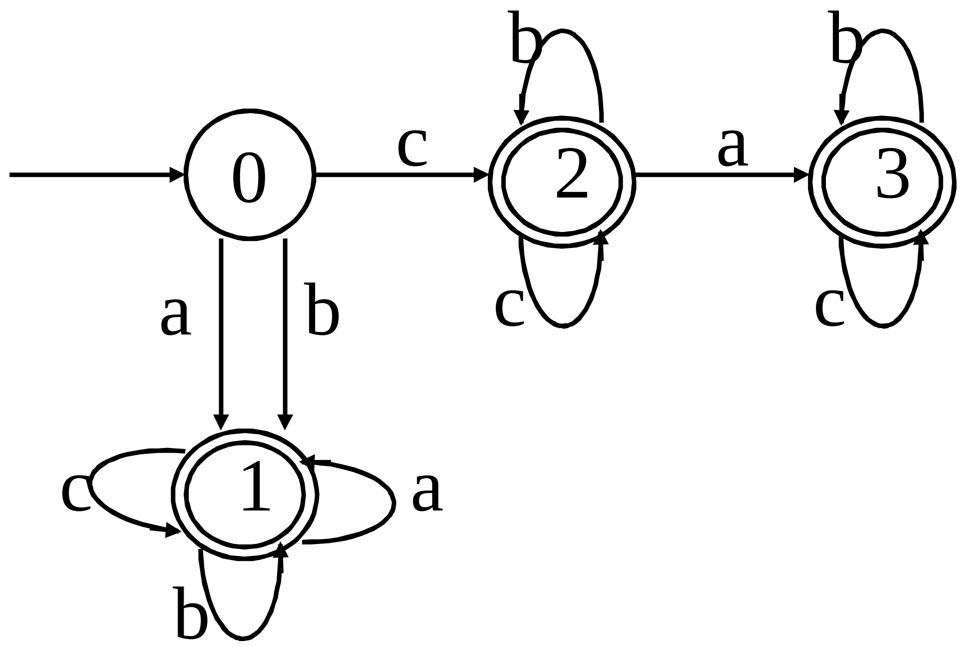
Transition table

| state \ input | a | b |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 3 | 3 | 3 |



Transition graph

E.g. Construct a DFA M，which can accept the strings which begin with *a* or *b*, or begin with *c* and contain at most one *a*。



So ,the DFA is

M=({0,1,2,3,},{a,b,c},move,0,{1,2,3})

move： move(0,a)=1 move(0,b)=1
move(0,c)=1 move(1,a)=1
move(1,b)=1 move(1,c)=1
move(2,a)=3 move(2,b)=2
move(2,c)=2 move(3,b)=3
move(3,c)=3

# Finite automata

4. Non-deterministic FA (NFA)

NFA is a quintuple, M(S,Σ,move,$s_0$,F）

- S: a set of *states*
- Σ: the input symbol alphabet
- move: a mapping from S× Σ to S, *move(s,a)=$2^S$, $2^S$* ⊆S
- $s_0$: the *start* state, $s_0 \in$ S
- F: a set of states *F* distinguished as *accepting* states, F⊆S

## Finite automata

4. Non-deterministic FA (NFA)

Note:1) In a NFA,the same character can label two or more transitions out of one state;

2) In a NFA,ε is a legal input symbol.

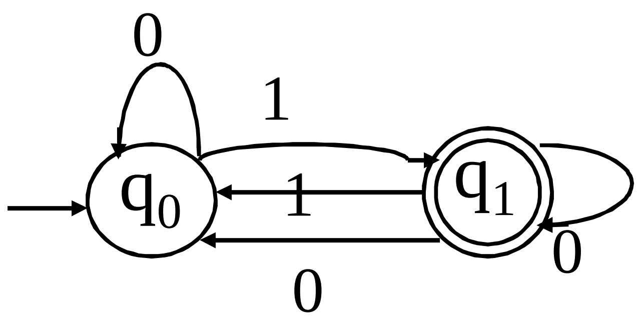
3) A DFA is a special case of a NFA

4)A NFA accepts an input string x if and only if there is some path in the transition graph from start state to some accepting state. A path can be represented by a sequence of state transitions called moves.

5)The language defined by a NFA is the set of input strings it accepts

## e.g. An NFA M＝({$q_0$,$q_1$},{0,1},move,$q_0$,{$q_1$})

| input State | 0 | 1 |
|---|---|---|
| $q_0$ | $q_0$ | $q_1$ |
| $q_1$ | $q_0$, $q_1$ | $q_0$ |



The language defined by the NFA is
0*10*|0*10*((1|0)0*10*)*

## Finite automata

5. Conversion of an NFA into a DFA

a)Reasons to conversion

Avoiding ambiguity

b)The algorithm idea

Subset construction: The following state set of a state in a NFA is thought of as a following STATE of the state in the converted DFA

# Finite automata

## 5. Conversion of an NFA into a DFA

c)The pre-process-- $\varepsilon$-closure(T)

Obtain $\varepsilon$-closure(T) $T \subseteq S$

(1) $\varepsilon$-closure(T) definition

A set of NFA states reachable from NFA state *s* in *T* on $\varepsilon$-*transitions* alone

## Finite automata

5. Conversion of an NFA into a DFA

c)The pre-process--- ε-closure(T)

(2)ε-closure(T) algorithm

```
push all states in T onto stack;
initialize ε-closure(T)  to T;
 while stack is not empty do {
    pop the top element of the stack into t;
    for each state u with an edge from t to u
       labeled ε do
      {
      if u is not in ε-closure(T) {
         add u to ε-closure(T)
         push u into stack}}}
```

# CHAPTER 3 LEXICAL ANALYSIS

## Section 4 Finite automata

### 5. Conversion of an NFA into a DFA

d)Subset Construction algorithm

– Input. An NFA $N=(S,\Sigma,move,S_0,Z)$

– Output. A DFA $D= (Q,\Sigma,\delta,I_0,F)$, accepting the same language

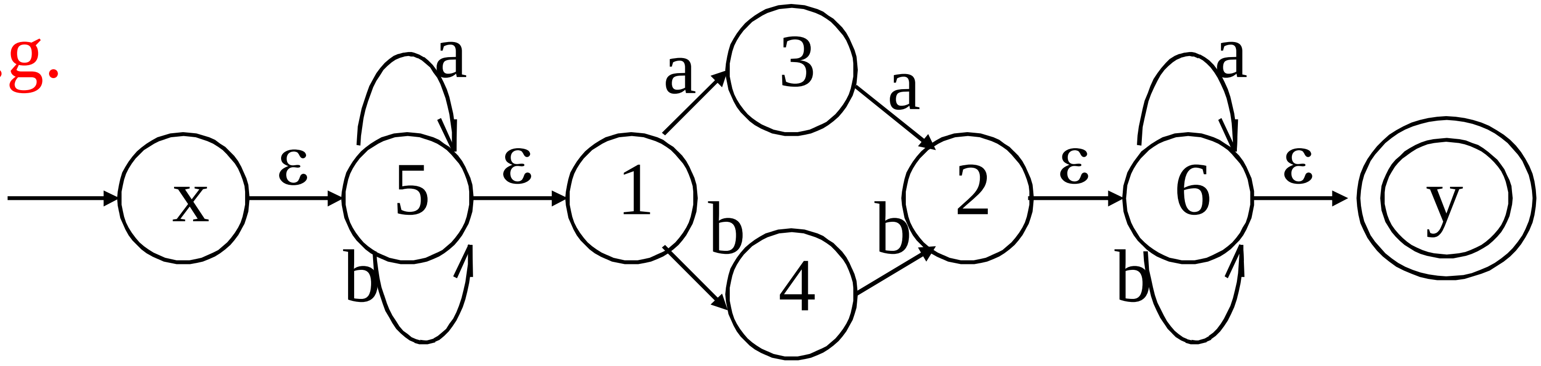
# Finite automata

## 5. Conversion of an NFA into a DFA

d)Subset Construction algorithm

```
(1)$I_0 = \varepsilon$-closure($S_0$), $I_0 \in Q$
(2)For each $I_i$ , $I_i \in Q$,
    let $I_t = \varepsilon$-closure(move($I_i$,a))
     if $I_t \notin Q$, then put $I_t$ into Q
(3)Repeat step (2), until there is no new state
    to put into Q
(4)Let $F=\{I \mid I \in Q, 且 I \cap Z <> \Phi\}$
```

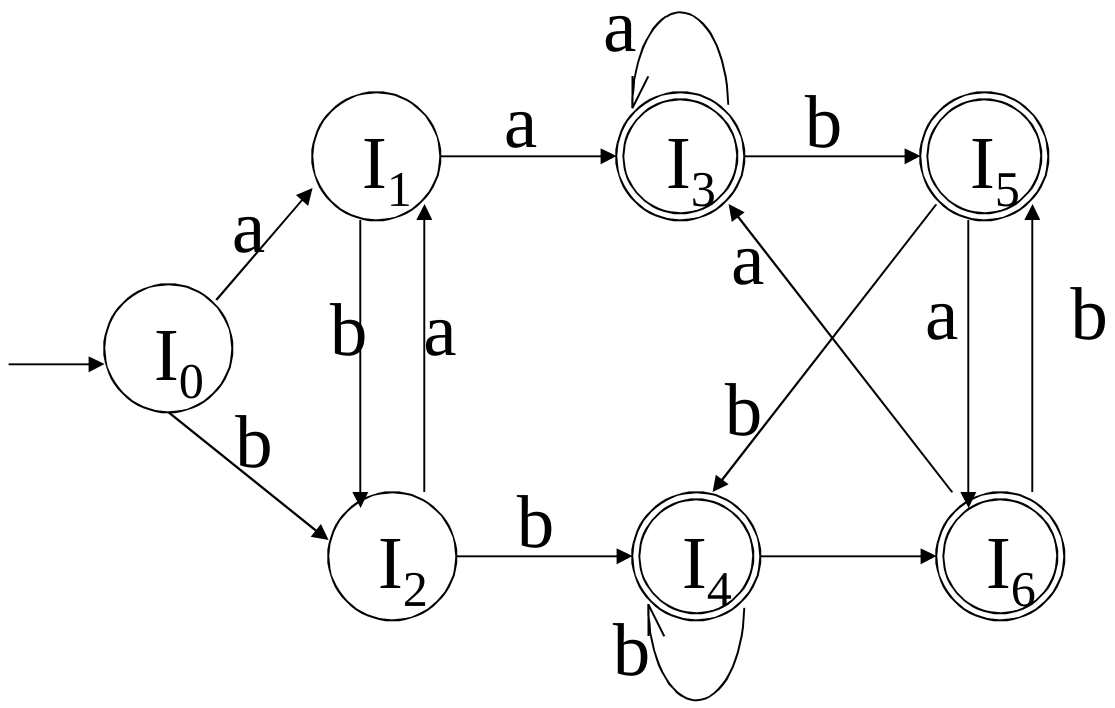
e.g.



| I | a | b |
|---|---|---|
| $I_0=\{x,5,1\}$ | $I_1=\{5,3,1\}$ | $I_2=\{5,4,1\}$ |
| $I_1=\{5,3,1\}$ | $I_3=\{5,3,2,1,6,y\}$ | $I_2=\{5,4,1\}$ |
| $I_2=\{5,4,1\}$ | $I_1=\{5,3,1\}$ | $I_4=\{5,4,1,2,6,y\}$ |
| $I_3=\{5,3,2,1,6,y\}$ | $I_3=\{5,3,2,1,6,y\}$ | $I_5=\{5,1,4,6,y\}$ |
| $I_4=\{5,4,1,2,6,y\}$ | $I_6=\{5,3,1,6,y\}$ | $I_4=\{5,4,1,2,6,y\}$ |
| $I_5=\{5,1,4,6,y\}$ | $I_6=\{5,3,1,6,y\}$ | $I_4=\{5,4,1,2,6,y\}$ |
| $I_6=\{5,3,1,6,y\}$ | $I_3=\{5,3,2,1,6,y\}$ | $I_5=\{5,1,4,6,y\}$ |

| I | a | b |
|---|---|---|
| $I_0$ | $I_1$ | $I_2$ |
| $I_1$ | $I_3$ | $I_2$ |
| $I_2$ | $I_1$ | $I_4$ |
| $I_3$ | $I_3$ | $I_5$ |
| $I_4$ | $I_6$ | $I_4$ |
| $I_5$ | $I_6$ | $I_4$ |
| $I_6$ | $I_3$ | $I_5$ |

DFA is



# Finite automata

## 5. Conversion of an NFA into a DFA

d)Subset Construction algorithm

Notes:

1)Both DFA and NFA can recognize precisely the regular sets;

2)DFA can lead to faster recognizers

3)DFA can be much bigger than an equivalent NFA

## Finite automata

6. Minimizing the number of States of a DFA

a)Basic idea

Find all groups of states that can be distinguished by some input string. At beginning of the process, we assume two distinguished groups of states: the group of non-accepting states and the group of accepting states. Then we use the method of partition of equivalent class on input string to partition the existed groups into smaller groups .

# Finite automata

6. Minimizing the number of States of a DFA

b)Algorithm

- Input. A DFA M={S,Σ,move, $s_0$,F}
- Output. A DFA M' accepting the same language as M and having as few states as possible.

## Finite automata

### 6. Minimizing the number of States of a DFA

#### b)Algorithm

(1)Construct an initial partition $\prod$ of the set of states with two groups: the accepting states *F* and the non-accepting states *S-F*. $\prod_0 = \{I_0^1, I_0^2\}$

# Finite automata

## 6. Minimizing the number of States of a DFA

### b)Algorithm

(2) For each group *I* of $\prod_i$ ,partition *I* into subgroups such that two states *s* and *t* of *I* are in the same subgroup if and only if for all input symbols *a*, states *s* and *t* have transitions on *a* to states in the same group of $\prod_i$ ; replace *I* in $\prod_{i+1}$_ by the set of subgroups formed.

# Finite automata

## 6. Minimizing the number of States of a DFA

b)Algorithm

(3) If $\prod_{i+1} = \prod_i$ ,let $\prod_{final} = \prod_{i+1}$ and continue with step (4). Otherwise,repeat step (2) with $\prod_{i+1}$

(4) Choose one state in each group of the partition $\prod_{final}$ as the representative for that group. The representatives will be the states of the reduced DFA M'. Let *s* and *t* be representative states for *s*'s and *t*'s group respectively, and suppose on input *a* there is a transition of *M* from *s* to *t*. Then *M'* has a transition from *s* to *t* on *a*.

# Finite automata

## 6. Minimizing the number of States of a DFA

### b)Algorithm

(5) If M’ has a dead state(a state that is not accepting and that has transitions to itself on all input symbols),then remove it. Also remove any states not reachable from the start state.

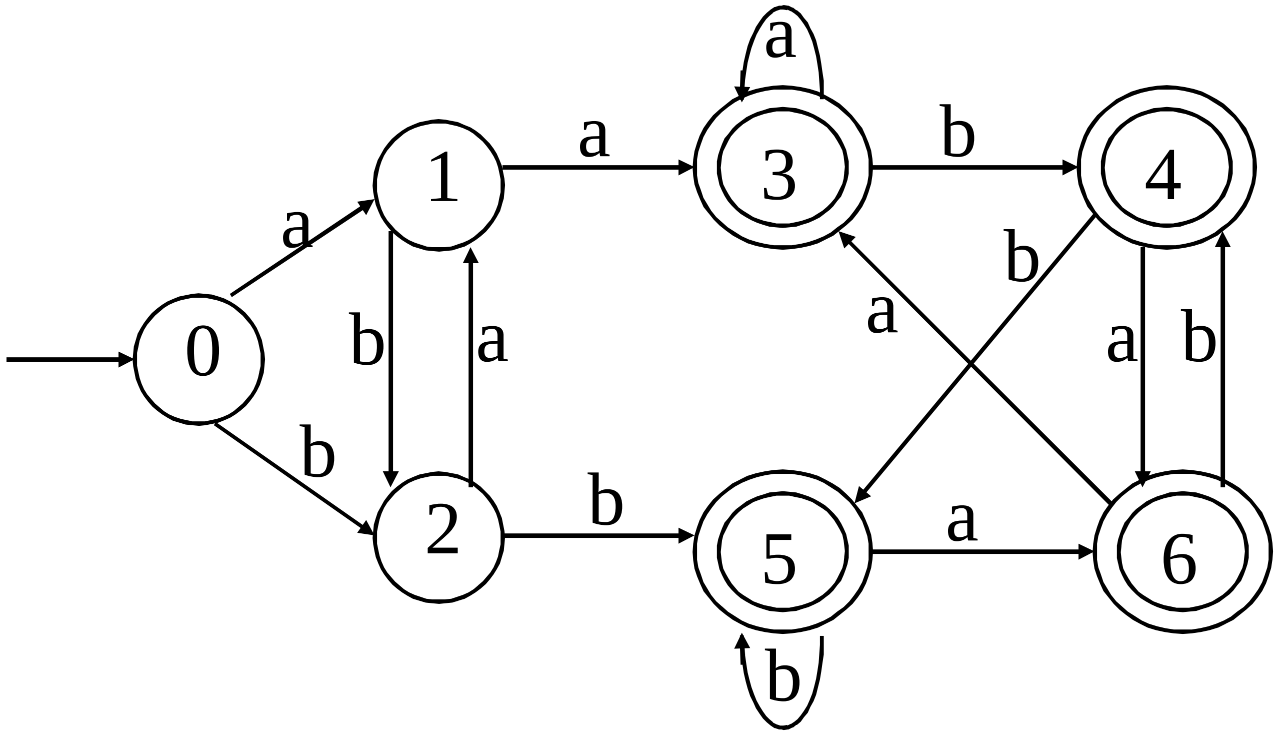
## Finite automata

### 6. Minimizing the number of States of a DFA

#### b)Algorithm

Notes: The meaning that string *w distinguishes* state *s* from state *t is that b*y starting with the 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 input w, we end up in a non-accepting state, or vice versa.

## e.g. Minimize the following DFA.



1. Initialization: $\prod_0 = \{\{0,1,2\},\{3,4,5,6\}\}$

2.1 For Non-accepting states in $\prod_0$ :

- a: move({0,2},a)={1} ; move({1},a)={3} . 1,3 do not in the same subgroup of $\prod_0$.
- So ,$\prod_1` = \{\{1\}, \{0,2\}, \{3,4,5,6\}\}$
- b: move({0},b)={2}; move({2},b)={5}. 2,5 do not in the same subgroup of $\prod_1$'.
- So, $\prod_1`` = \{\{1\}, \{0\}, \{2\}, \{3,4,5,6\}\}$

2.2 For accepting states in $\prod_0$ :

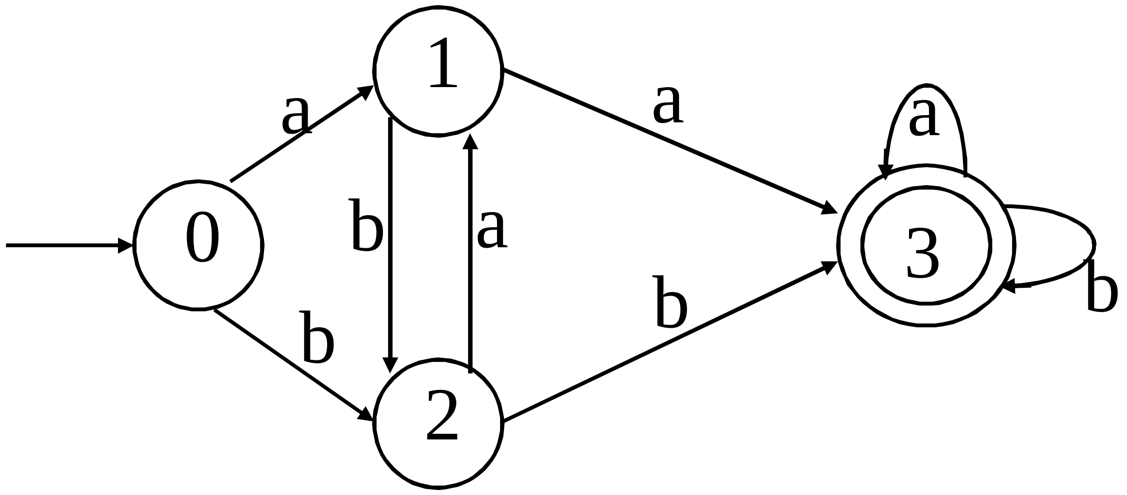
- a: move({3,4,5,6},a)={3,6}, which is the subset of {3,4,5,6} in $\prod_1$"
- b: move({3,4,5,6},b)={4,5}, which is the subset of {3,4,5,6} in $\prod_1$"
- So, $\prod_1$ ={{1}, {0}, {2}, {3,4,5,6}}.

3.Apply the step (2) again to $\prod_1$ ,and get $\prod_2$.

- $\prod_2$ ={{1},{0},{2},{3,4,5,6}}= $\prod_1$ ,
- So, $\prod_{final}$ = $\prod_1$

4. Let state 3 represent the state group {3,4,5,6}

## So, the minimized DFA is :



# Regular expression to an NFA

1. The reasons about regular expression to a NFA

   Strategy for building a recognizer from a regular expression is to construct an NFA from a regular expression and then to simulate the behavior of the NFA on an input string.

## Regular expression to an NFA

2. Construction of an NFA from a regular expression

a)Basic idea

Syntax-directed in that it uses the syntactic structure of the regular expression to guide the construction process.

## Regular expression to an NFA

2. Construction of an NFA from a regular expression

a)Algorithm

- Input. A regular expression *r* over an alphabet Σ
- Output. An NFA *N* accepting *L( r)*

## Regular expression to an NFA

2. Construction of an NFA from a regular expression

a)Algorithm

– Method

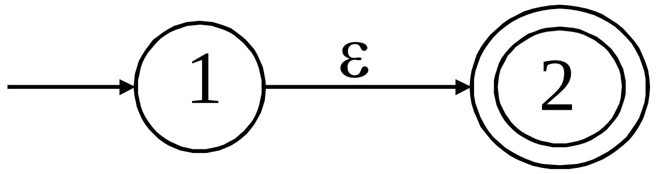
(1) Parse *r* into its constituent sub-expressions.

(2) Use rules in the next pages to construct NFA's for each of the basic symbols in *r*(those that are either ε or an alphabet symbol).

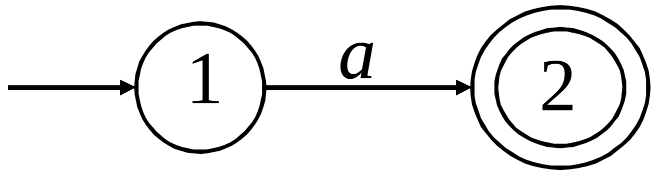
(3)Use rules in the next and next page to combine these NFA's inductively, and obtain the NFA for the entire expression.

## Rules

1. For ε,

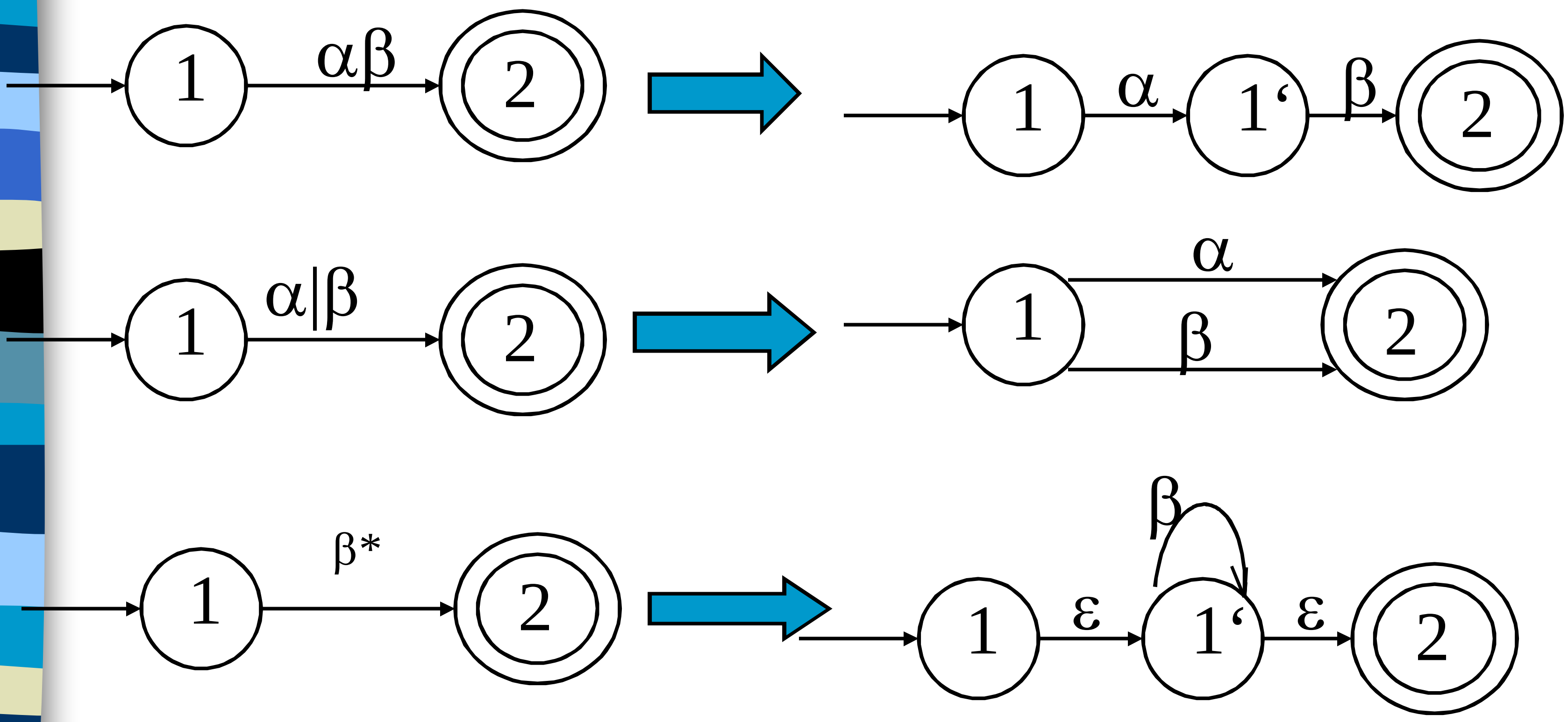


2. For *a* in Σ,

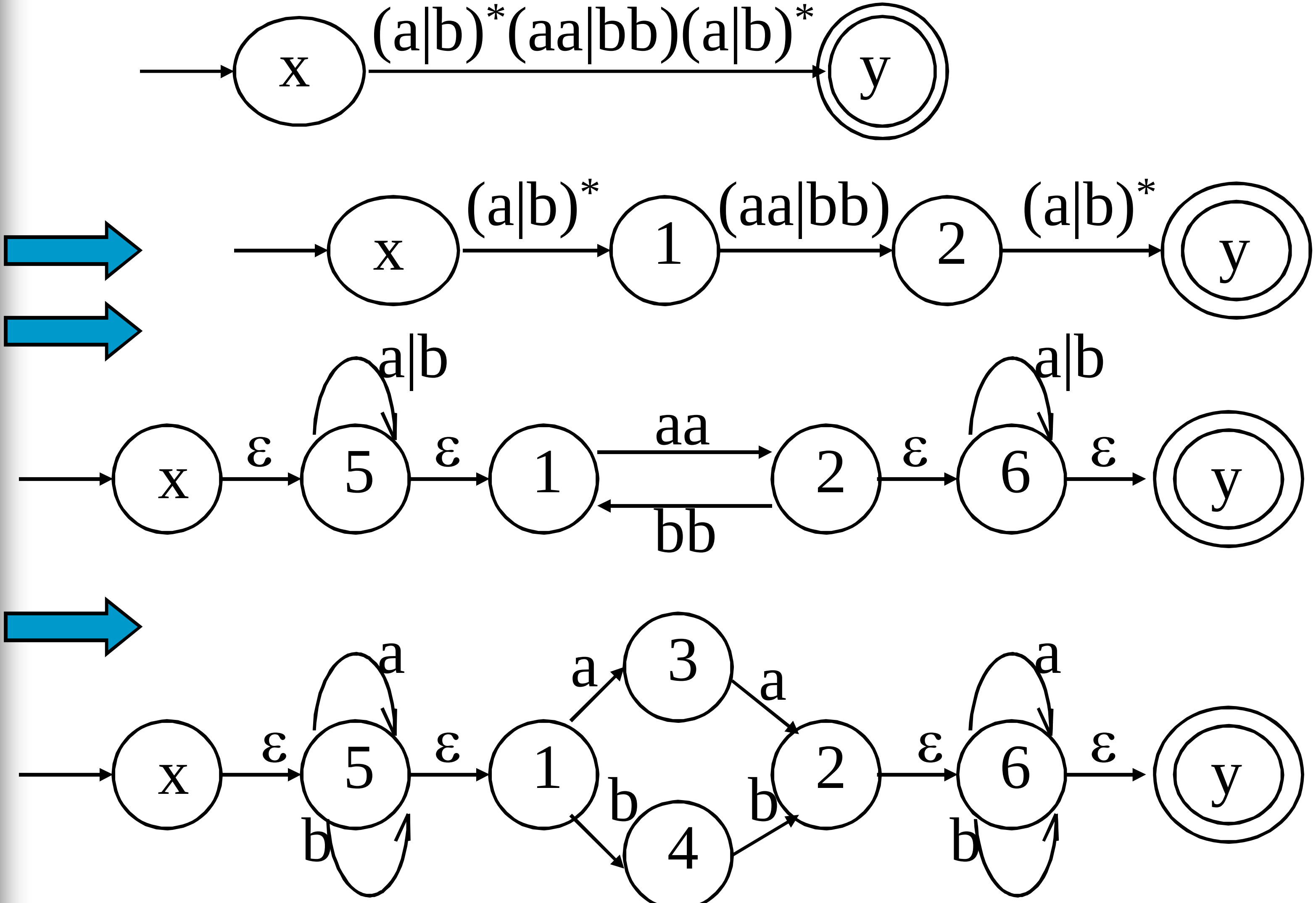


# Rules

3. Rules for complex regular expressions



e.g. Let us construct *N( r)* for the regular expression *r=(a|b)*(aa|bb)(a|b)**



## FA to Regular expression

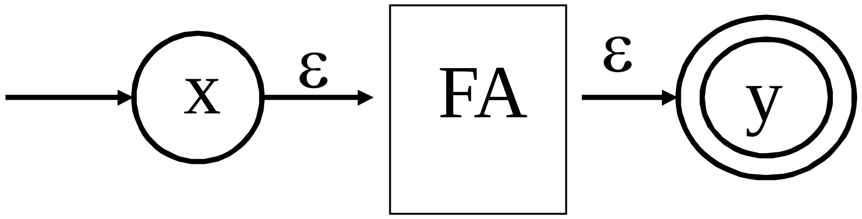
1. Basic ideas

   Reduce the number of states by merging states

2. Algorithm
   - Input: An FA M
   - Output: A regular expression *r* over an alphabet Σ recognize the same language as FA M

# FA to Regular expression

2. Algorithm
   - Method:
     - Extend the concept of FA, let the arrows can be marked by regular expressions.
     - Add two nodes x,y to the FA M and get M' that recognize the same regular language.

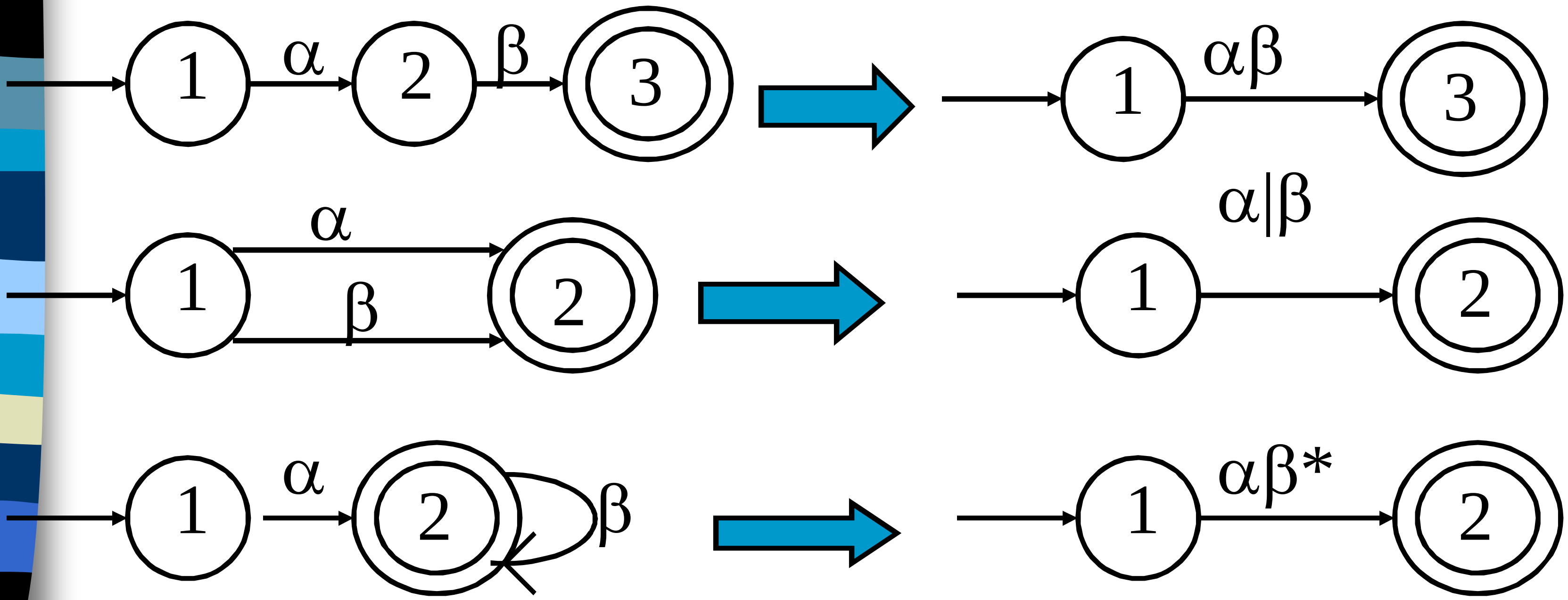


# FA to Regular expression

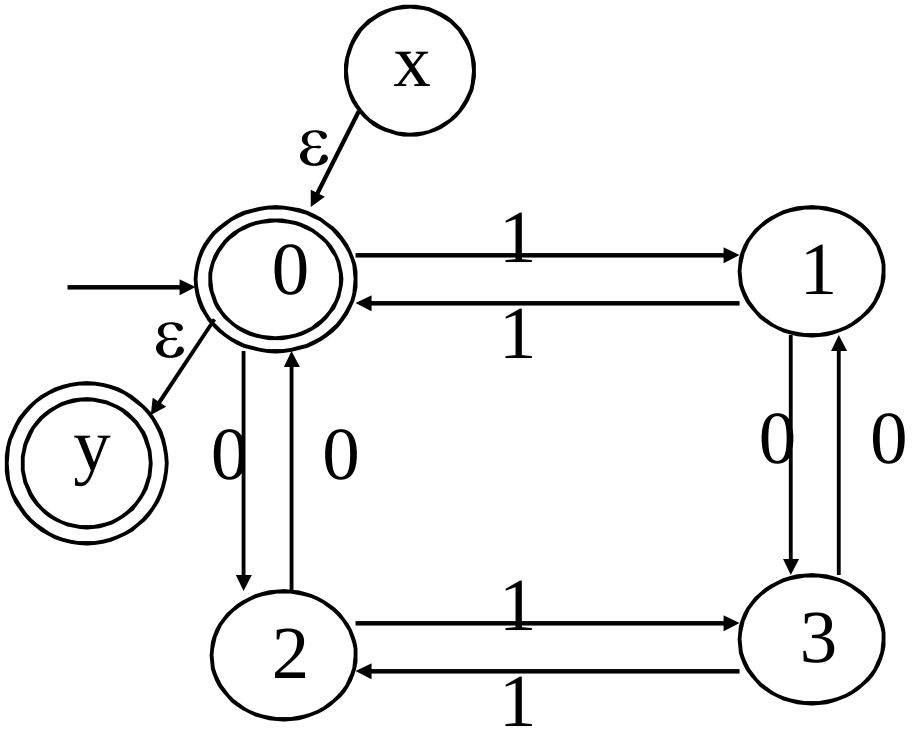
2. Algorithm

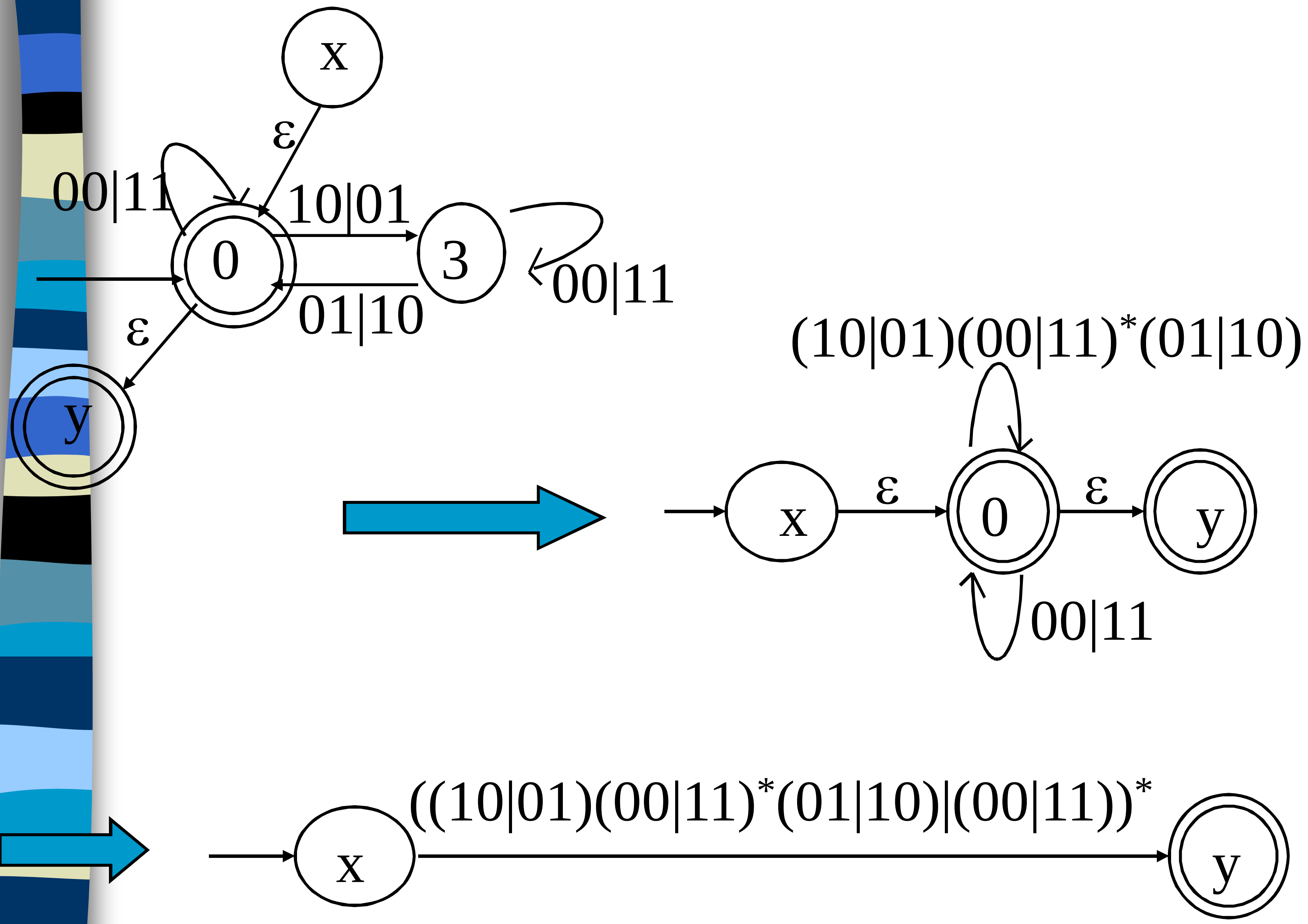
– Method:

- Use  the following rules to combine the regular expression in the FA's inductively, and obtain the entire expression for the FA



## E.g. Construct the regular expression for the following DFA M.




x
ε
00|11
10|01
0
3
00|11
01|10
ε
y
(10|01)(00|11)*(01|10)
x
ε
0
ε
y
00|11
((10|01)(00|11)*(01|10)|(00|11))*
x
y

# Regular Grammar to an NFA

1. Basic properties

- For each regular grammar $G=(V_N,V_T,P,S)$, there is an FA $M=(Q,\Sigma,f,q0,Z)$, and $L(G)=L(M)$.
- For each FA M, there is a right-linear grammar and a left-linear grammar recognize the same language. $L(M)=L(G_R)=L(G_L)$

## Regular Grammar to an NFA

2. Right-linear grammar to FA
   - Input :G=($V_N$,$V_T$,P,S)
   - Output : FA M=(Q, $\Sigma$,move,$q_0$,Z)
   - Method :
     - Consider each non-terminal symbol in G as a state, and add a new state T as an accepting state.
     - Let Q=$V_N$∪{T} , $\Sigma = V_T$ , $q_0 =$S; if there is the production S$\to \varepsilon$ , then Z={S,T}, else Z={T} ;

# Regular Grammar to an NFA

2. Right-linear grammar to FA
   - Method :
     - For each production, construct the function *move*.

     a) For the productions similar as $A_1 \to aA_2$, construct move($A_1$,a)= $A_2$.

     b) For the productions similar as $A_1 \to a$, construct move($A_1$,a)= T.

     c) For each *a* in $\Sigma$, move(T,a)=$\Phi$, that means the accepting states do not recognize any terminal symbol.

E.g. A regular grammar

G=({S,A,B},{a,b,c},P,S) P: $S \rightarrow aS \mid aB$

$B \rightarrow bB \mid bA$

$A \rightarrow cA \mid c$

Construct a FA for the grammar G.

Answer: let $M=(Q,\Sigma,f,q_0,Z)$

1) Add a state T , So $Q=\{S,B,A,T\}$; $\Sigma=\{a,b,c\}$; $q_0=S$; $Z=\{T\}$.

2) f：

$f(S,a)=S \quad f(S,a)=B$

$f(B,a)=B \quad f(B,b)=A$

$f(A,c)=A \quad f(A,c)=T$

$$\begin{tikzcd}
{} \arrow[r] & S \arrow[loop above, "a"] \arrow[r, "a"] & B \arrow[loop above, "b"] \arrow[r, "b"] & A \arrow[loop above, "c"] \arrow[r, "c"] & T
\end{tikzcd}$$

## Regular Grammar to an NFA

3. FA to Right-linear grammar
   - Input : $M=(S, \Sigma, f, s_0, Z)$
   - Output : $Rg=(V_N, V_T, P, s_0)$
   - Method :
     - If $s_0 \notin Z$, then the Productions are;

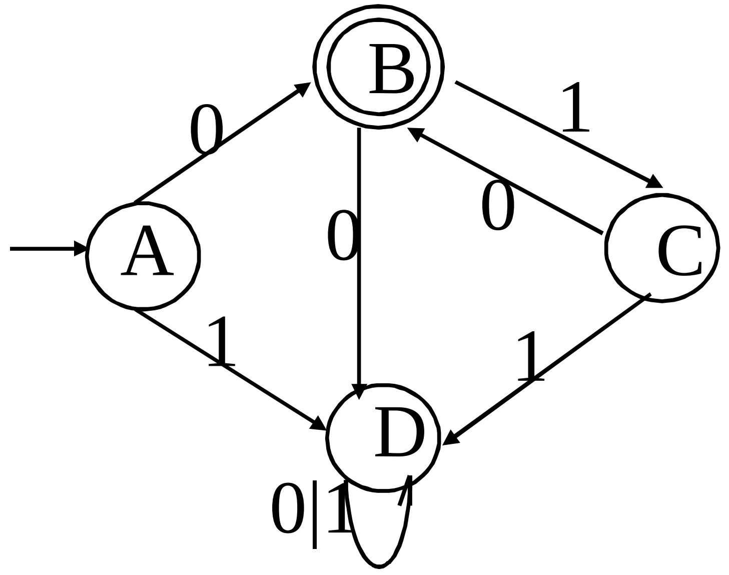
     a) For the mapping $f(A_i,a)=A_j$ in M, there is a production $A_i \rightarrow aA_j$;

     b) If $A_j \in Z$, then add a new production $A_i \rightarrow a$, then we get $A_i \rightarrow a|aA_j$;

## Regular Grammar to an NFA

3. FA to Right-linear grammar
   - Method :
     - If $s_0 \in Z$, then we will get the following productions besides the productions we've gotten based on the former rule:
     - For the mapping $f(s_0,\varepsilon)=s_0$, construct new productions, $s_0' \to \varepsilon \mid s_0$, and $s_0'$ is the new starting state.

## e.g. construct a right-linear grammar for the following DFA M=({A,B,C,D},{0,1},f,A,{B})



Answer:Rg=({A,B,C,D},{0,1},P,A)

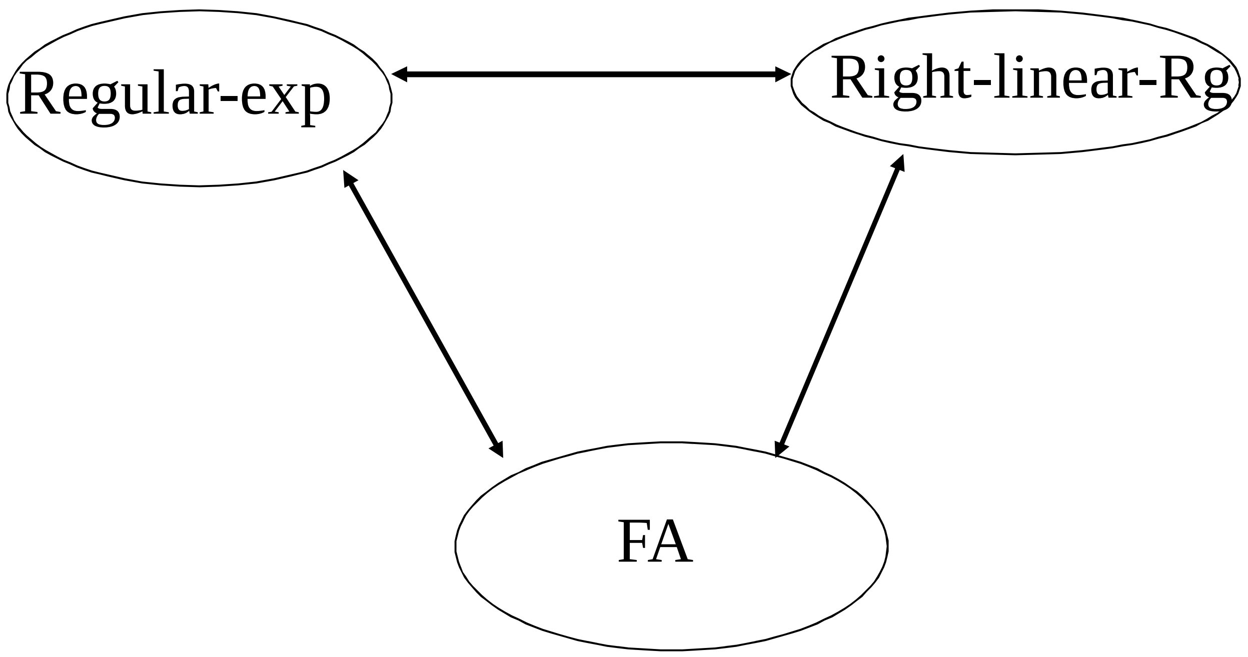
$A \rightarrow 0B \mid 1D \mid 0$

$B \rightarrow 1C \mid 0D$

$C \rightarrow 0B \mid 1D \mid 0$

$D \rightarrow 0D \mid 1D$

L(Rg)=L(M)=$0(10)^*$


Regular-exp
Right-linear-Rg
FA

## Design of a lexical analyzer generator

1. Lexical analyzer generator

   A software tool that automatically constructs a lexical analyzer from related language specification

2. Typical lexical analyzer generator

   Lex

## Design of a lexical analyzer generator

3. Lex

  a) Lexical analyzer generating tool

    Lex compiler

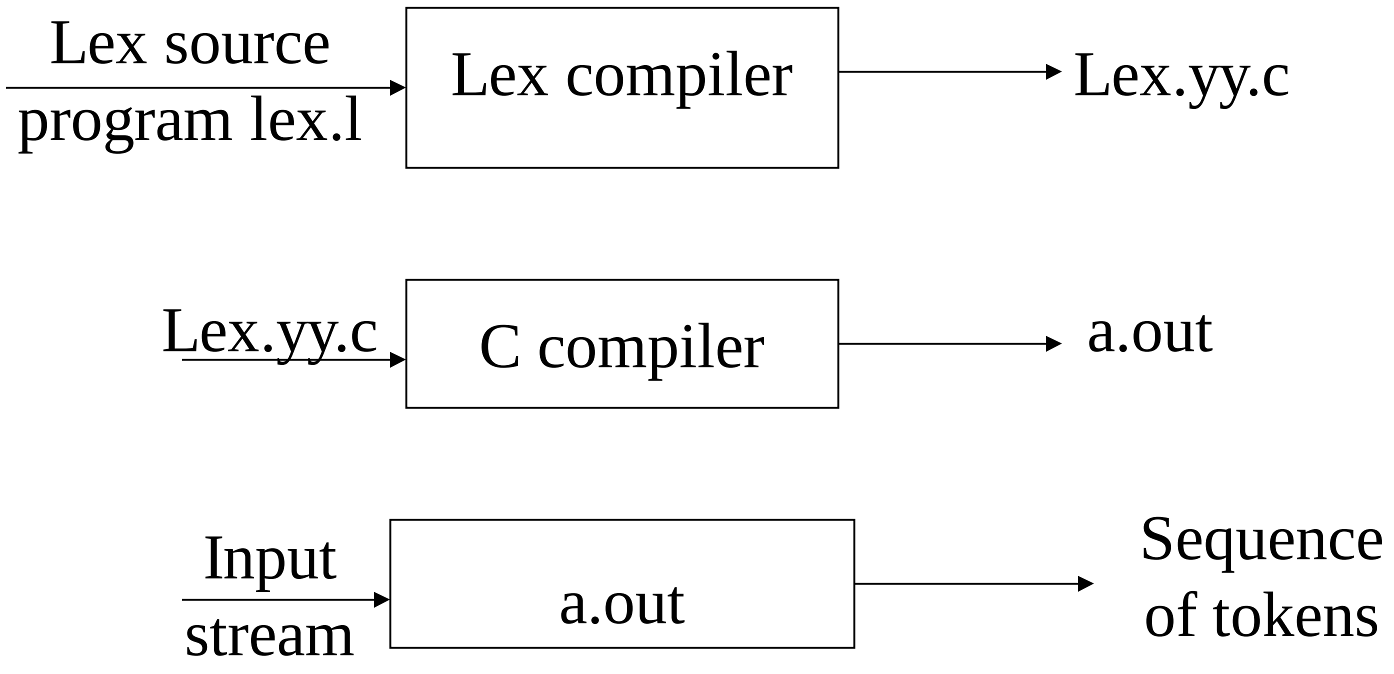
  b)Input specification

    Lex language program

# Design of a lexical analyzer generator

3. Lex

c) The process that creates a lexical analyzer with Lex



## Design of a lexical analyzer generator

3. Lex

d) Lex specification

A Lex program consists of three parts:

```
declaration
%%
translation rules
%%
auxiliary procedures
```

## Design of a lexical analyzer generator

3. Lex

d) Lex specification

(1)Declaration

Include declarations of variables, manifest constants and regular definitions

Notes: A manifest constant is an identifier that is declared to represent a constant

# Design of a lexical analyzer generator

3. Lex

d) Lex specification

(1)Declaration

```
%{
  /*definitions of manifest constants
  LT,LE,EQ,GT,GE,IF,THEN,ELSE,ID*/
 %}
/*regular expression*/
delim [\t\n]
ws      {delim}+
letter   [A-Za-z]
digit    [0-9]
id        {letter}({letter}|{digit})*
```

# Design of a lexical analyzer generator

3. Lex

   d) Lex specification

      (2)Translation Rules

         $p_1$ {$action_1$} /*p—pattern(Regular exp) */

         …

         $p_n$ {$action_n$}

   e.g {if} {return(IF);}

         {id} {yylval=install_id();return(ID);}

# Design of a lexical analyzer generator

3. Lex

d) Lex specification

(3)auxiliary procedures

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

Notes:The auxiliary procedures can be compiled separately and loaded with the lexical analyzer.

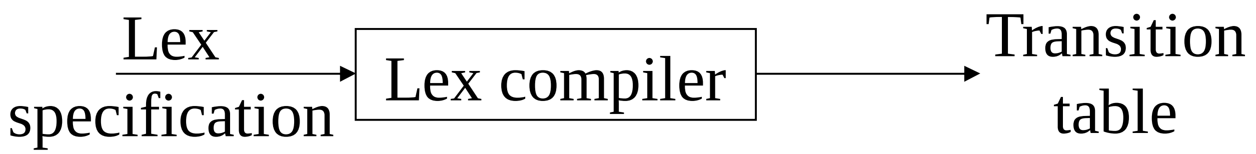
```
%{ /* this part is copied literally to the generated C code */
#define IF 5
#define ID 12
#define INTEGER 13
#define REAL 14
%}
delim          [ \t\n]+
letter         [A-Za-z]
digit          [0-9]
id             {letter}({letter}|{digit})*
integer        {digit}+
real           {digit}+\.({digit}+)?|({digit}+)?\. {digit}+
%%
{delim}        {/* no action and no return */}
if             {return IF;}
{id}           {yylval = install_id(); return ID;}
{real}         {yylval = convert_real(); return REAL;}
{integer}      {yylval = convert_int(); return INTEGER;}
%%
install_id()   { ... procedure to enter or lookup in the symbol table the string
               pointed to by yytext with length yyleng ...}
```

```
%{
/* a lex program that adds line numbers to lines of text,
printing the new text to the standard output */
#include <stdio.h>
int lineno = 1;
%}
line        .*\n
%%
{line}      {printf("%5d %s", lineno++, yytext); }
%%
main()
{ yylex(); return 0;}
```

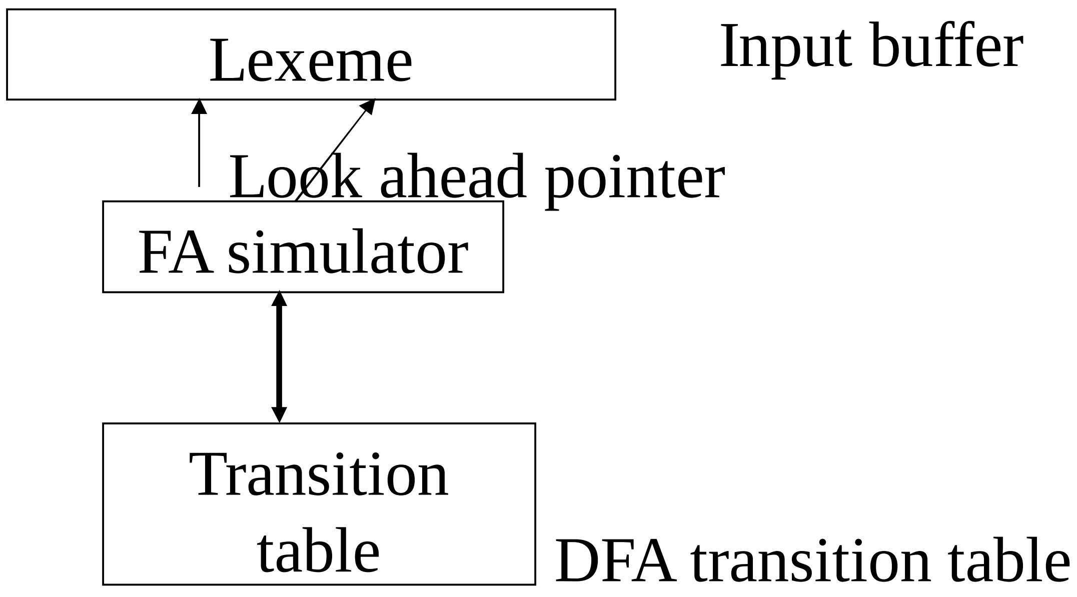
# Design of a lexical analyzer generator

3. Lex

e) Model of Lex compiler



Lex specification
Lex compiler
Transition table



Lexeme
Input buffer
Look ahead pointer
FA simulator
Transition table
DFA transition table