Chapter 3
Lexical Analysis

Zhen Gao
gaozhen@tongji.edu.cn

Scanner

- **Lexical analysis is the foundation of compilation**
- **The program that performs lexical analysis is called a lexical analyzer (or scanner)**
- **Lexical analysis tasks**
  - **Scan the source program left to right, character by character**
  - **Recognize and generate a token stream from the input string**

Outline

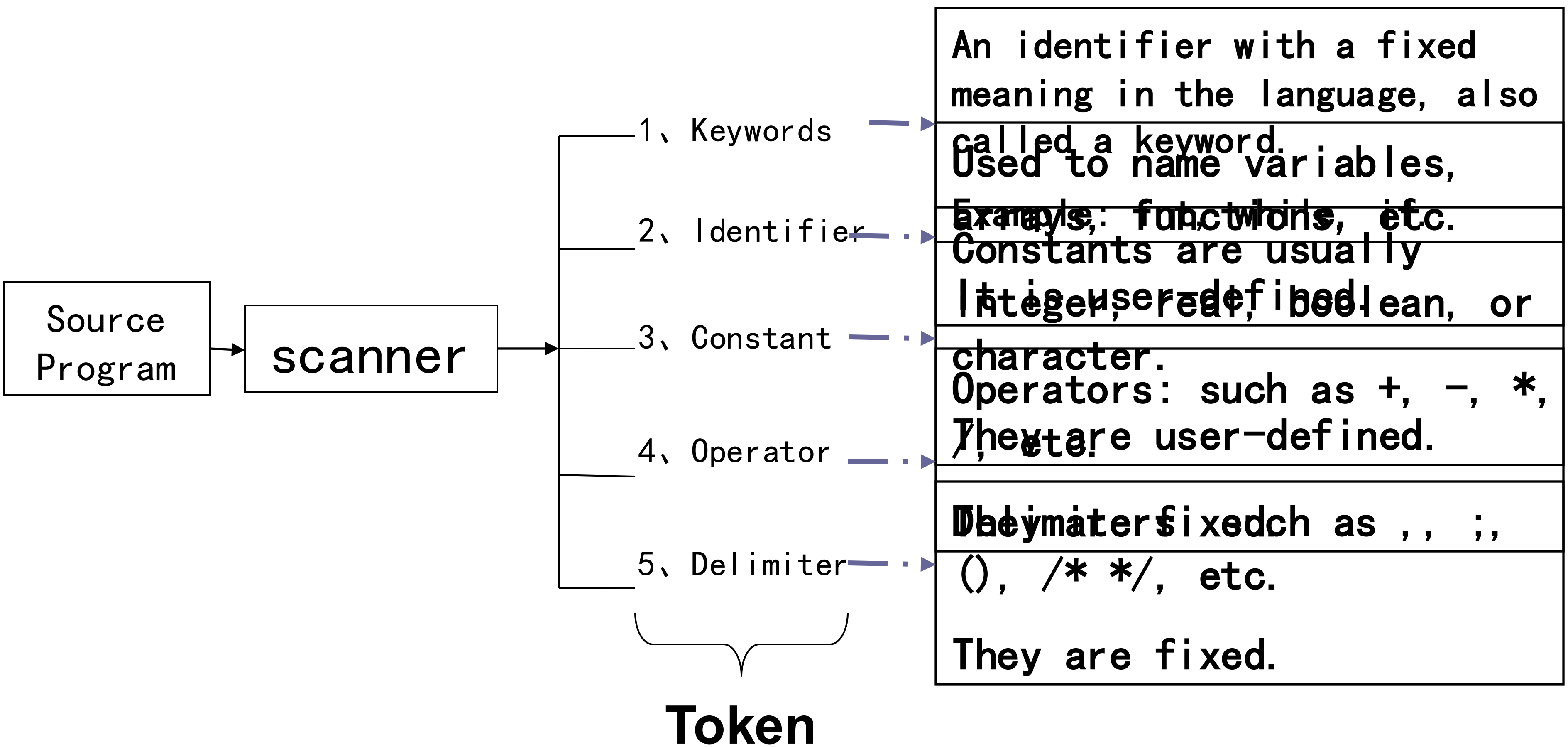
3.1 Requirements for Scanner

3.2 Design of Scanner

3.3 Regular Expressions and Finite Automata

3.4 Automatic Generation of Scanner

Functions of a Scanner



*

Token Representation

(token type, attribute value)

- **Token type:** information needed by syntax analysis
- **Attribute value:** information usually needed by other compilation stages, also called **token value**

Example: In `int i, j;`, `i` and `j` , tokens are “**identifier**”, and their attribute values are the ”**symbol table entries**”

Token

- **Token : Usually encoded as integers**
- **Encoding usually depend on processing convenience**
  - **Identifiers: single category**
  - **Constants: by type (int, real, boolean)**
  - **Keywords: one per keyword**
  - **Operators: one per symbol or by common traits**
  - **Delimiters: one per symbol**

Attribute Value

- **Attribute values reflect the features or characteristics of a token**
  - **Identifiers**: value is a pointer to its symbol table entry or internal string
  - **Constants**: value is a pointer to its constant table entry or in binary form
  - **Keywords, Operators, Delimiters:** one token per item; no separate attribute value needed

Example: Code segment: while (i>=j) i--;

<while , - >

< (, - >

< id , ptr-i>

< >= , - >

< id , ptr-j>

<) , - >

< id , ptr-i>

< ………. >

< ; , - >

Symbol Table				
No	ID	Addr	type	…… .
224	j	AF80	INT	
227	i	DF88	INT	

*

FORTRAN Compilation Example

- **IF (5·EQ·M) GOTO 100**
- **the FORTRAN Scanner outputs the following token sequence:**

IF	(34, _)
(	(2, _)
5	(20, binary of '5’)
EQ	(6, _)
M	(26, 'M’)
)	(16, _)
GOTO	(30, _)
100	(19, binary of '100')

Implementation of Scanner

- **Completely Independent:** Scanner is a separate pass; reads the whole source; output goes to Parser
  - **Advantage:** Simple, clear, organized structure
- **Relatively Independent:** Scanner is a subroutine called by Parser as needed
  - **Advantage:** No intermediate files; more efficient

Outline

3.1 Requirements for Scanner

3.2 Design of Scanner

3.3 Regular Expressions and Finite Automata

3.4 Automatic Generation of Scanner

Token Recognition: Lookahead

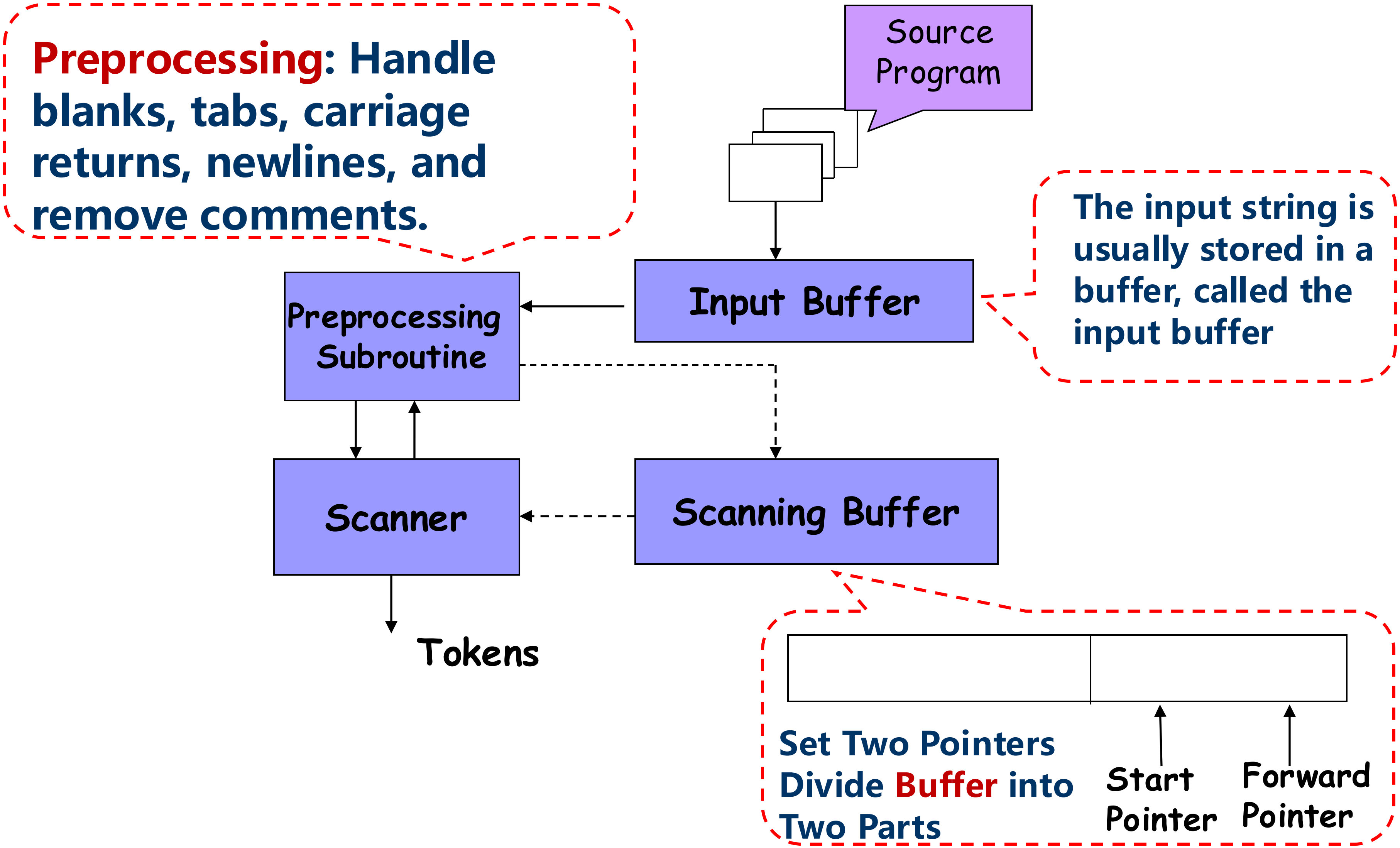
- **Keyword Recognition**

Example: Two valid sentences in standard FORTRAN:

1、DO99K = 1,10 — DO as a keyword

2、DO99K = 1.10 — DO as part of an identifier

Structure of Scanner

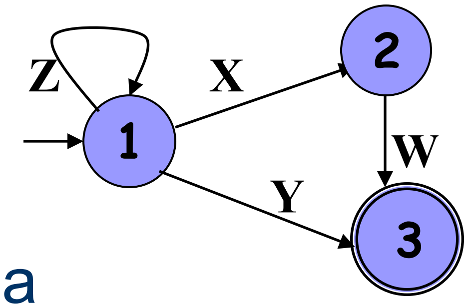


Regular Grammar

- **In most programming languages, lexical rules for tokens can be described by regular grammars:**
  - **<Identifier> → letter | <Identifier> letter | <Identifier> digit**
  - **<Integer> → digit | <Integer> digit**
  - **<Operator> → + | - | * | / …**
  - **<Delimiter> → ; | , | (|) …**

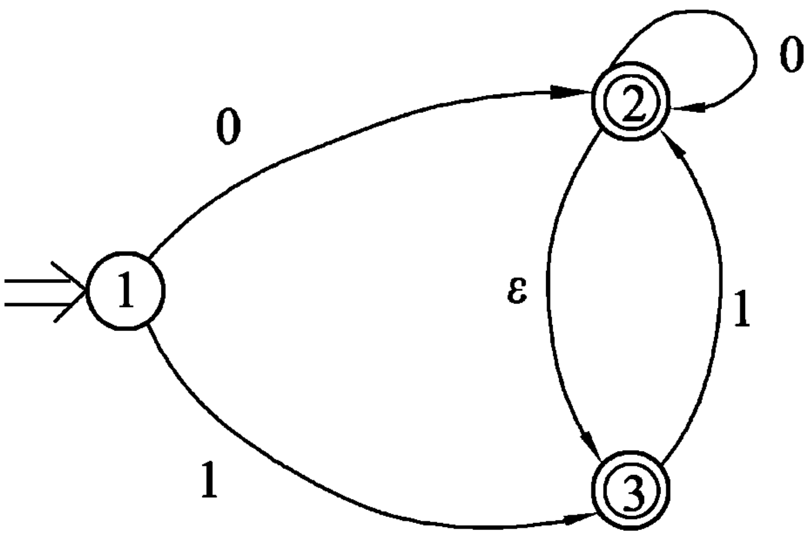
State Transition Diagram

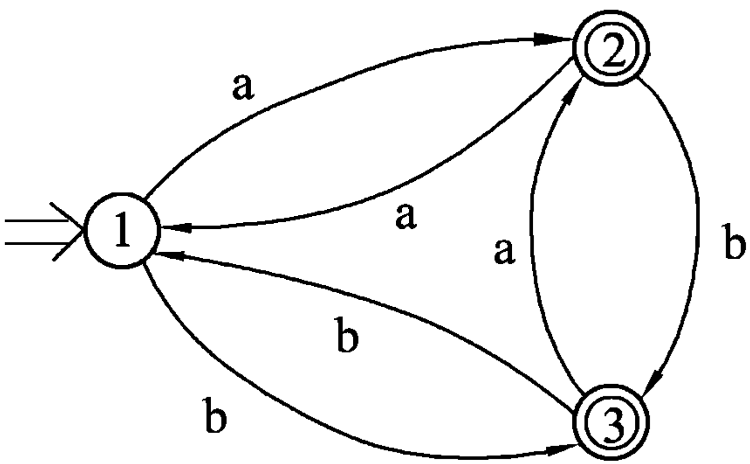
- **String Recognition**
  - A state transition diagram can **accept (recognize) certain strings**.
  - **Path**: sequence of edge labels from **start state** to a **final state**.
  - A string **β** is **accepted** if there exists a path generating β.
  - If no such path exists, the string **β is not accepted**.



State Transition Diagram

- **Language Recognized by a State Transition Diagram**
  - Let **L(TG)** be the set of strings accepted by a state transition diagram **TG**.



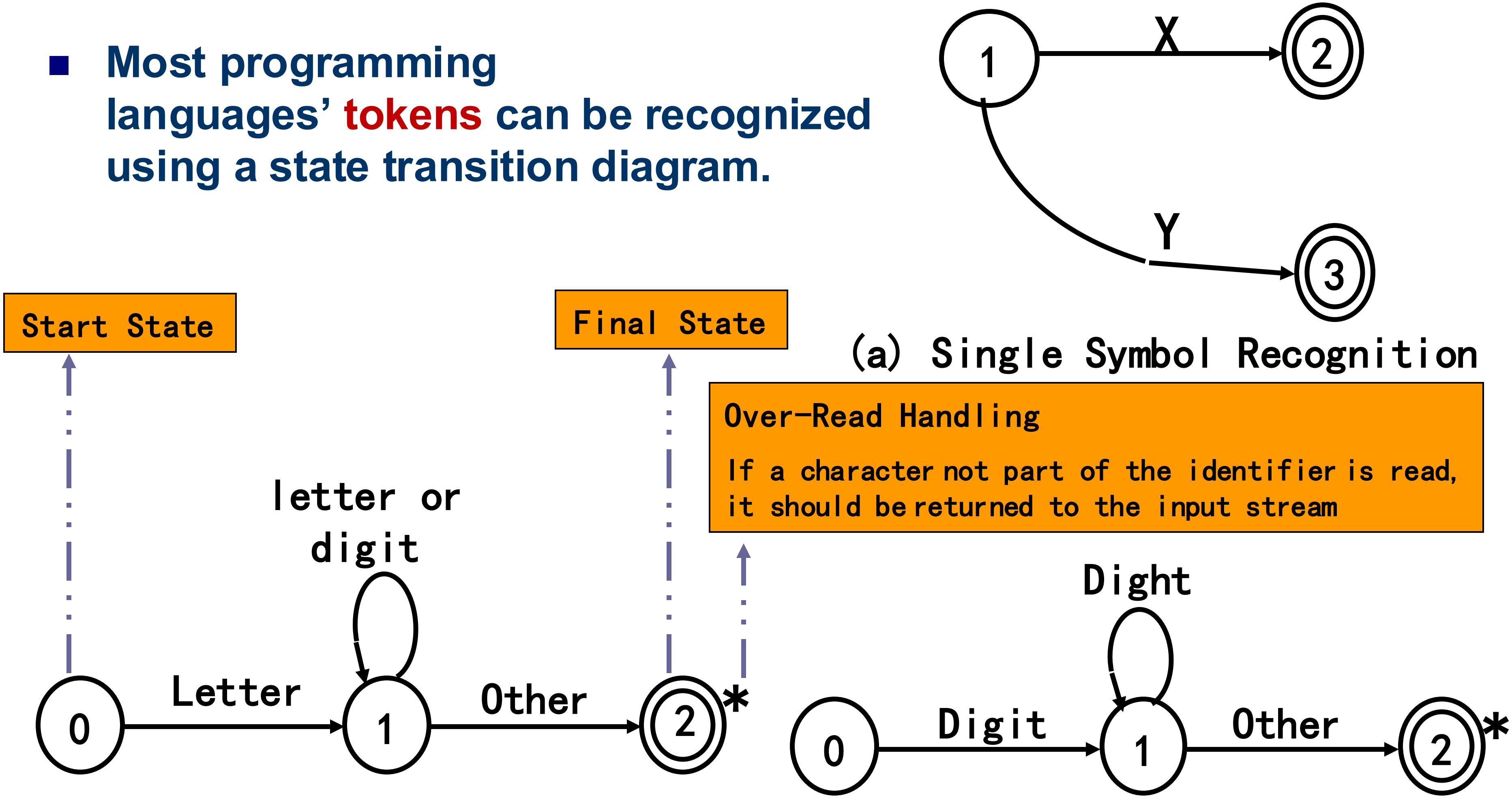


L(TG)={ 0， 1， 00， 01， 11， 001， 010， …}

L(TG)={ a， b， ab， ba， aaa， bbb， aab， bba， …}

State Transition Diagram

- **Most programming languages' tokens can be recognized using a state transition diagram.**



(a) Single Symbol Recognition

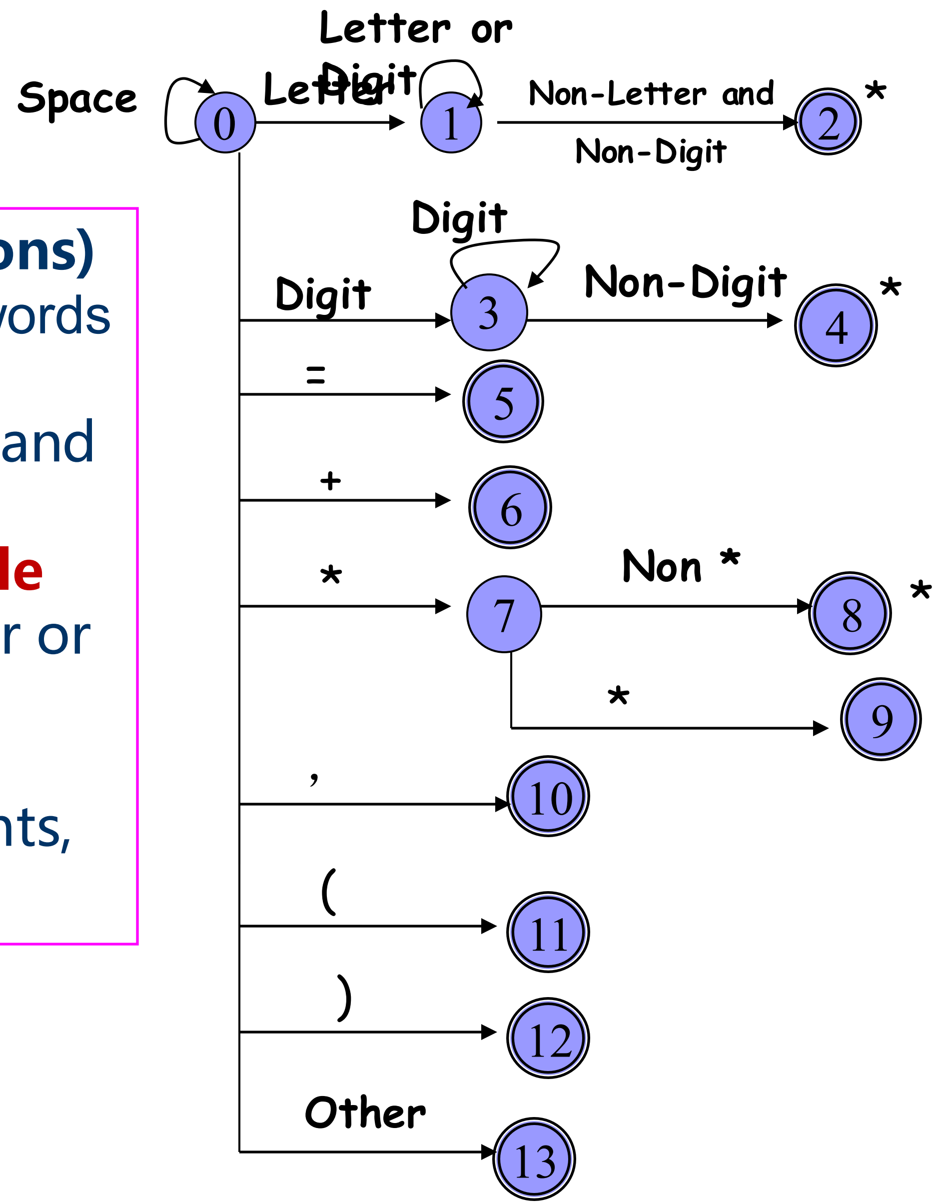
(b) Identifier Recognition

(c) Integer Recognition

Example: All Tokens and Their Internal Representations in a Small Language

Tokens	Token Type	Mnemonic Symbol	Attribute Value
DIM	1	$DIM	-
IF	2	$IF	-
DO	3	$DO	-
STOP	4	$STOP	-
END	5	$END	-
Identifier	6	$ID	Internal String
Constant	7	$INT	Binary Form
=	8	$ASSIGN	-
+	9	$PLUS	-
*	10	$STAR	-
**	11	$POWER	-
,	12	$COMMA	-
(	13	$LPAR	-
)	14	$RPAR	-

State Transition Diagram Recognizing All Tokens of a Small Language



Conventions (Restrictions)

- ✓ **Keywords**: reserved words
- ✓ Reserved words are treated as identifiers and recognized using a **reserved word table**
- ✓ If there is no operator or delimiter between keywords, identifiers, or constants, add a **space**

*

Simulating a DFA

```
s = s0;
c = nextChar() ;
while ( c != eof ) {
      s = move(s, c);
      c = nextChar() ;
}
if ( s is in F ) return " yes " ;
else return "no " ;
```

Function: Recognize whether a string is a valid token

Outline

3.1 Requirements for Scanner

3.2 Design of Scanner

3.3 Regular Expressions and Finite Automata

3.4 Automatic Generation of Scanner

Learning Approach

Lexical rules

→ Regular expressions

→ Automata (NFA → DFA → Minimized DFA)

→ Scanner

Regular Expressions and Finite Automata

- To better use state transition diagrams for constructing scanner and discussing **automatic generation** of scanners, the concept of the diagram needs to be formalized
  - **Regular expressions** and regular sets
  - Deterministic Finite Automata (**DFA**)
  - Non-deterministic Finite Automata (**NFA**)
  - **Equivalence** of regular expressions and finite automata
  - **Minimization** of DFA

Regular Expressions and Regular Sets

- **Recursive definition of regular expressions and sets over alphabet ∑:**
  - **ε and φ are regular expressions over ∑, representing the sets {ε} and ∅**
  - **Any symbol a ∈ ∑ is a regular expression representing the set {a}**
  - **Combination rules:**
    - **U | V → union L(U) ∪ L(V)**
    - **U · V → concatenation L(U) L(V)**
    - **U* → Kleene star (L(U))***
  - **Generation rule: Only expressions obtained by finite applications of the above rules are regular expressions over ∑**

Operators in Regular Expressions

- **“|”** → read as **“or”**
- **“.”** → read as **“concatenation”**
- **“*”** → read as **“closure”**
- Operator precedence: * **>. > |**
- Concatenation operator **“.”** can often be omitted; parentheses can be omitted if no ambiguity arises
- Two regular expressions are **equivalent (U = V)** if they represent the same regular set

*

Examples of Three Operations

Example 1. if L = { 001,10,111 }, M = {ε , 001},

$L \cup M$ = { ε , 10, 001, 111 }

Example 2. if L = { 001,10,111 }, M = {ε , 001},

LM = { 001, 10, 111, 001001, 10001, 111001 }

Example 3. if L = { 0, 11 },

L^* = { ε , 0, 11, 00, 011, 110, 1111, 000, 0011, 0110, 01111, 1100, 11011, 11110, 111111, ⋯ }

*

Example1 : Let ∑ = {a, b}

Regular Expressions | **Regular Sets**

$(a|b)(a|b)$ → $L(a|b)(a|b)=L(a|b)\cdot L(a|b)$

$=(L(a)\cup L(b))\cdot(L(a)\cup L(b))$

$=\{a,b\}\cdot\{a,b\}=\{aa,ab,ba,bb\}$

ba^* → $L(ba^*)=L(b)L(a^*)=L(b)(L(a))^*$

$=\{b\}\{a\}^*=\{b\}\{\varepsilon,a,aa,aaa,\ldots\}$

$=\{b,ba,baa,baaa,\ldots\}$

Regular Expressions **Regular Sets**

a(a|b)*

(a|b)*(aa|bb)(a|b)*

?

Example Regular expression for "identifier"

(A| B |…| Z | a | b |… z|_)(A | B |…| Z | a |b|…|z|_|0|1|…|9)*

Example Regular expression for "integer"

(0 | 1 | 2 |…| 9)(0 | 1 | 2 |…| 9)*

Algebraic Laws of Regular Expressions

- Let **U, V, W** be regular expressions:

(1) U|V=V|U	**Commutative Law**
(2) U|(V|W)=(U|V)|W	**Associative Law**
(3) U(VW)=(UV)W	**Associative Law**
(4) U(V|W)=UV|UW	**Distributive Law**
(5) (V|W)U=VU|WU	**Distributive Law**
(6) εU=Uε=U	

Exercise

- Write Regular Expressions
  - (1) Binary strings ending with 01
  - (2) Decimal integers divisible by 5

Learning Approach

Lexical rules

→ Regular expressions

→ Automata (NFA → DFA → Minimized DFA)

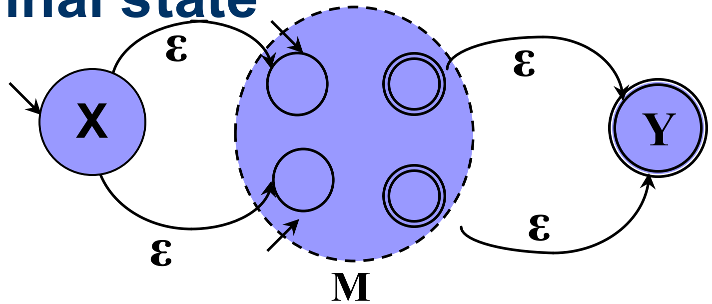
→ Scanner

Equivalence of Regular Expressions and Finite Automata

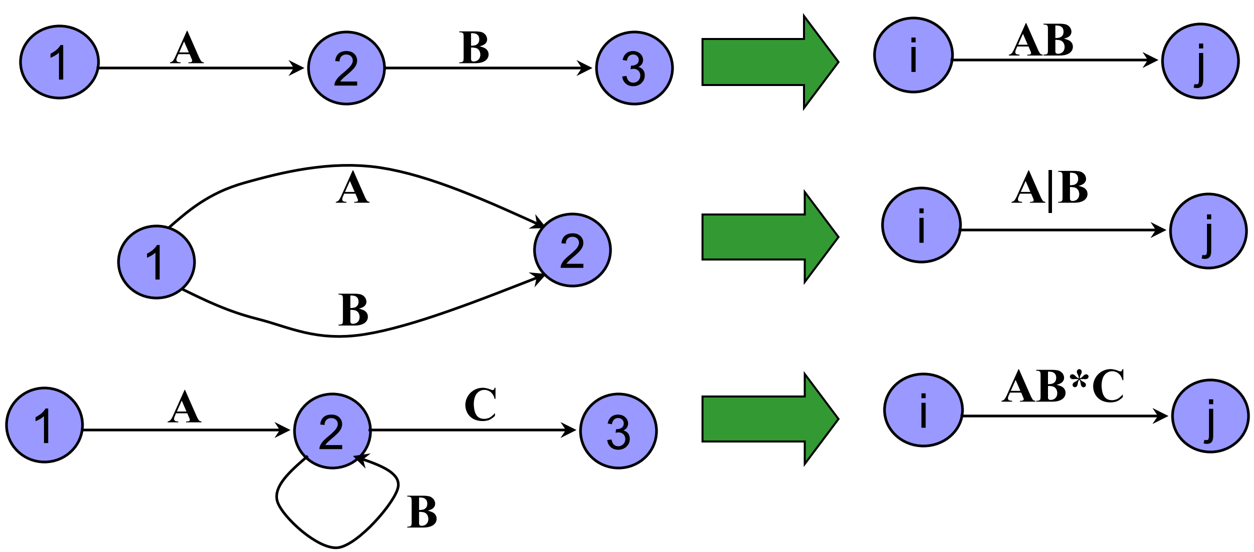
- Regular expressions and finite automata are **equivalent**
  - For any FA **M**, there exists a regular expression **V** such that **L(V) = L(M)**
  - For any regular expression **V**, there exists an FA **M** such that **L(M) = L(V)**

NFA →Regular Expression(State Elimination Method)

(1) **Add X node** and **Y node** to ensure a **unique start state** and a **unique final state**

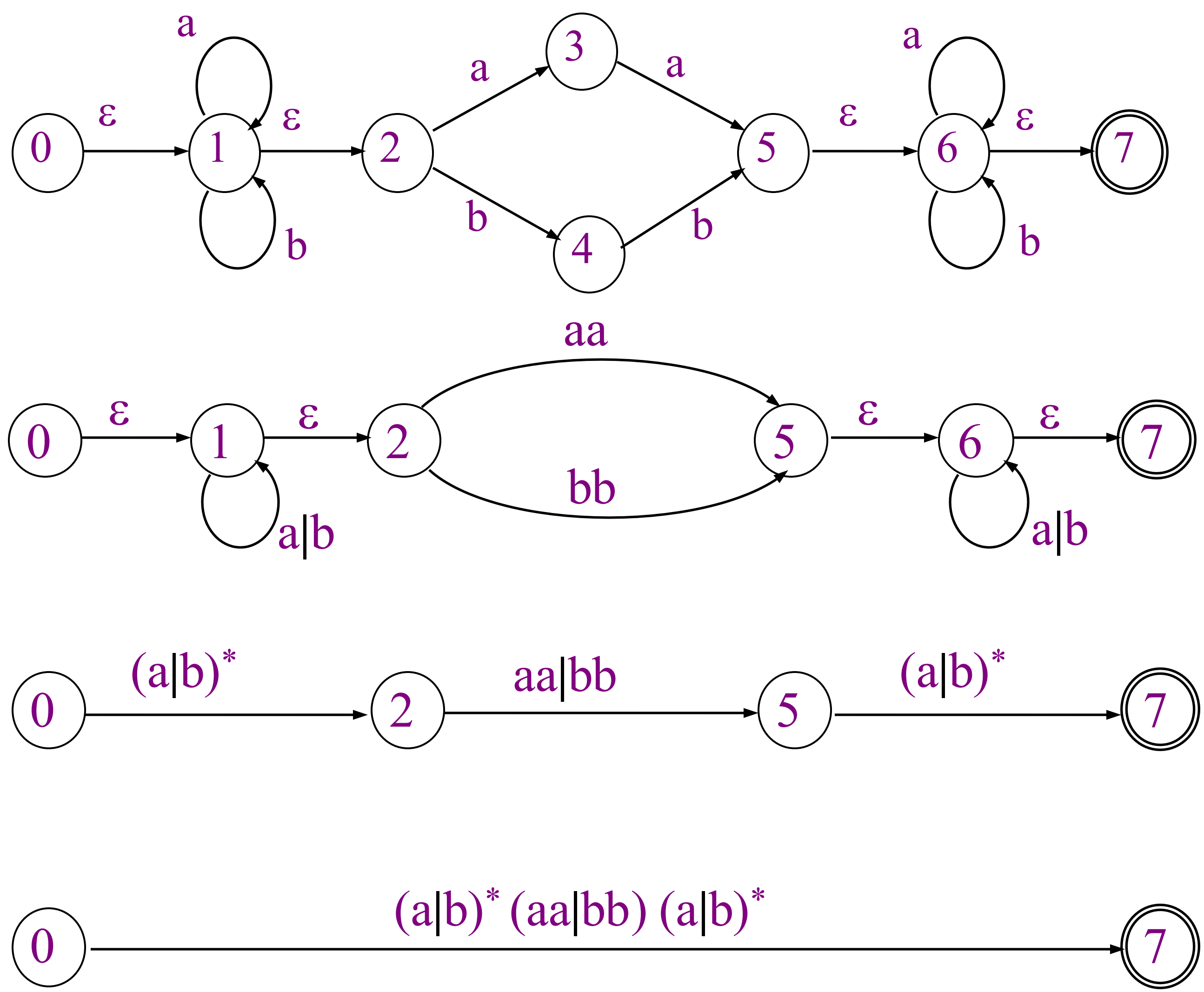


(2) Repeatedly **eliminate** states and merge edges using the rules, until only **X** and **Y** remain → the regular expression on the edge from **X** to **Y** is the result



*

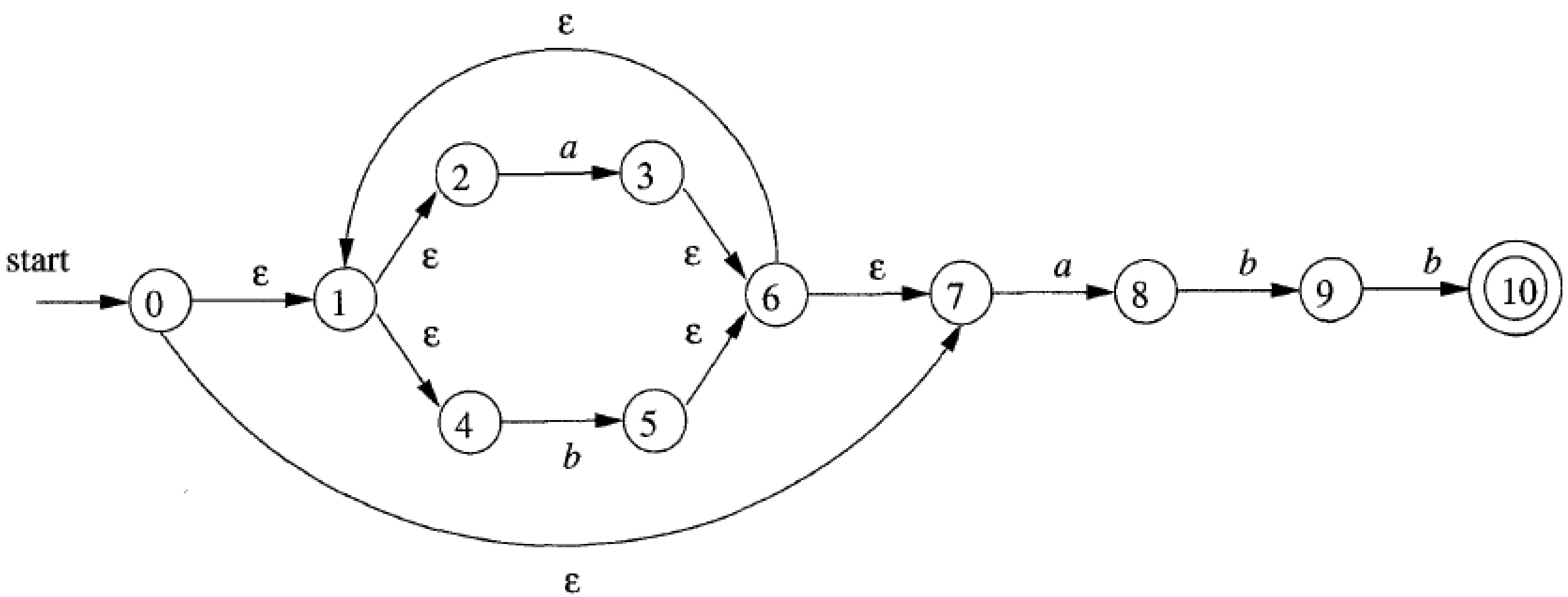
State Elimination Method



*

Quiz

NFA → Regular Expression

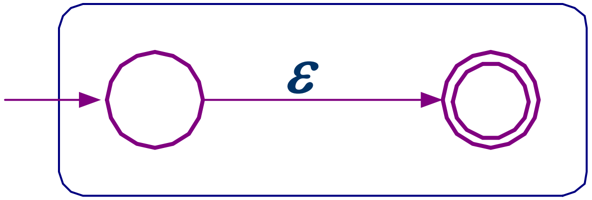


*

Regular Expression → NFA (Thompson's Algorithm)

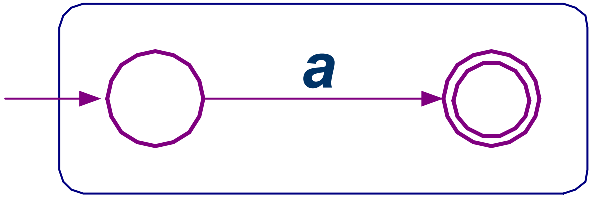
Basic Rules:

1 For ε , construct as



2 For ϕ , construct as

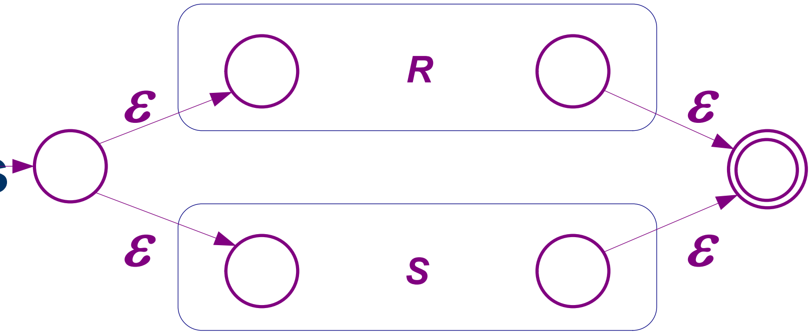
*3 For **a** , construct as*



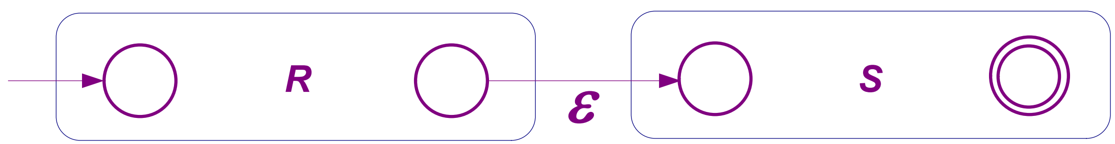
*

Induction:

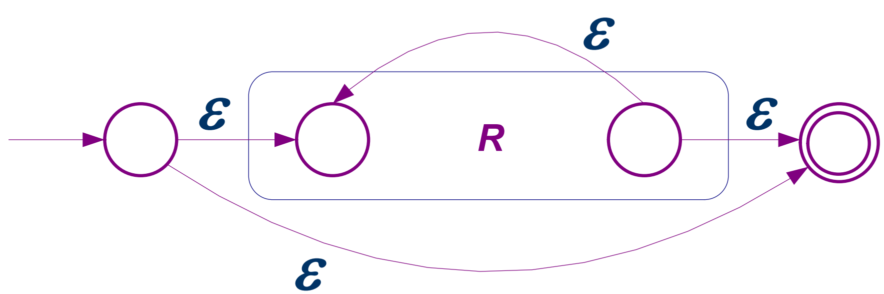
***1 For R|S* , construct as**



***2 For RS* , construct as**

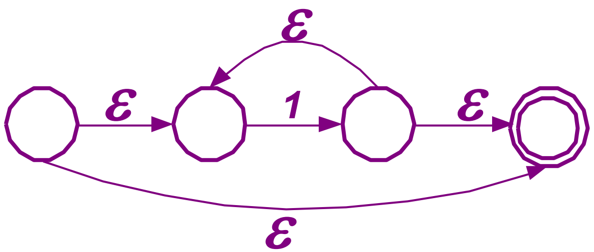


***3 For R** , construct as**

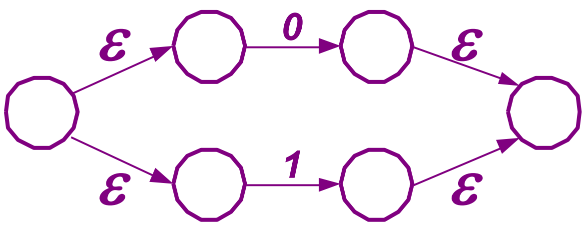


- **Example** : Given regular expression 1*0(0|1)*, construct an equivalent **NFA**.

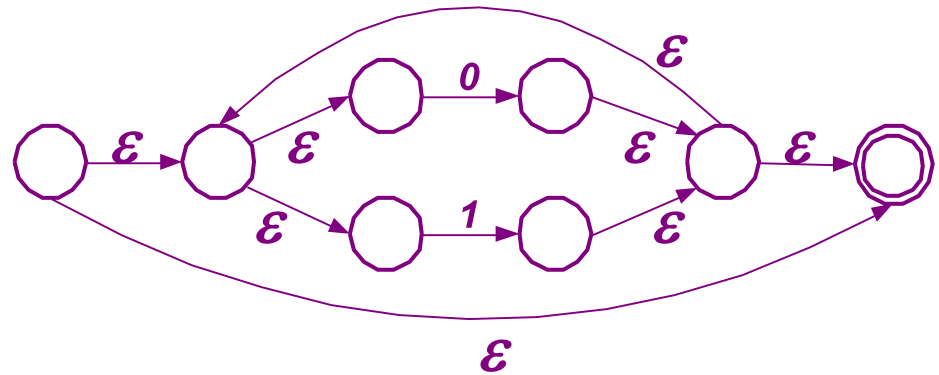
1*



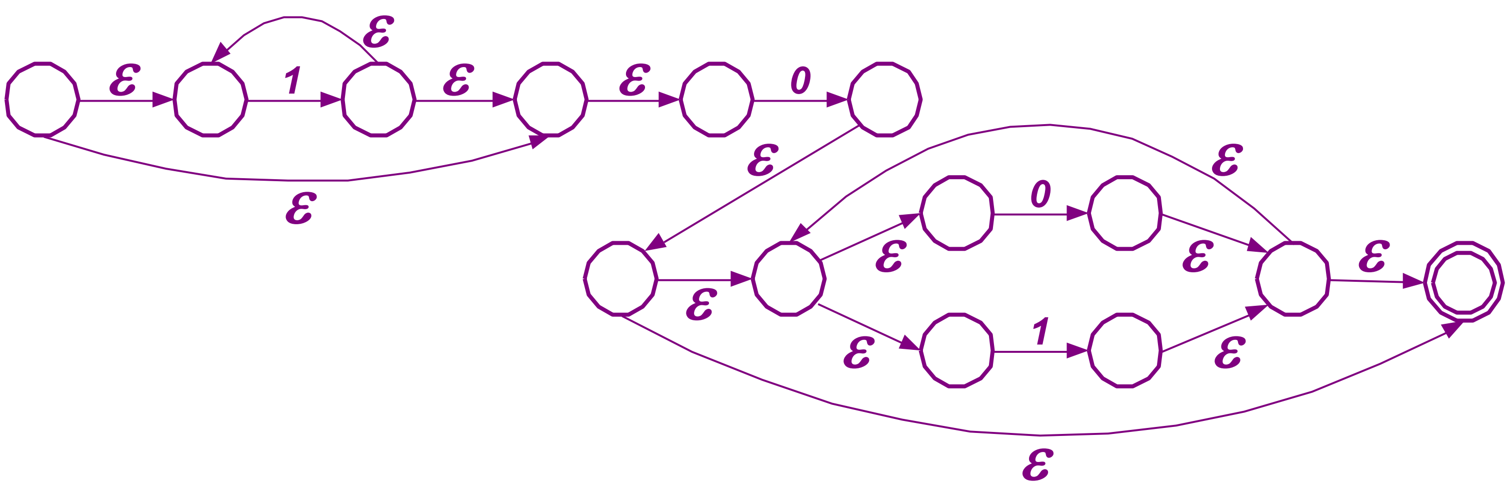
0|1



(0|1)* ⇒



1*0(0|1)* ⇒



Exercise

- (alb) * abb(alb) *

Learning Approach

Lexical rules

→ Regular expressions

→ Automata (NFA → DFA → Minimized DFA)

→ Scanner

*

Finite Automaton (FA)

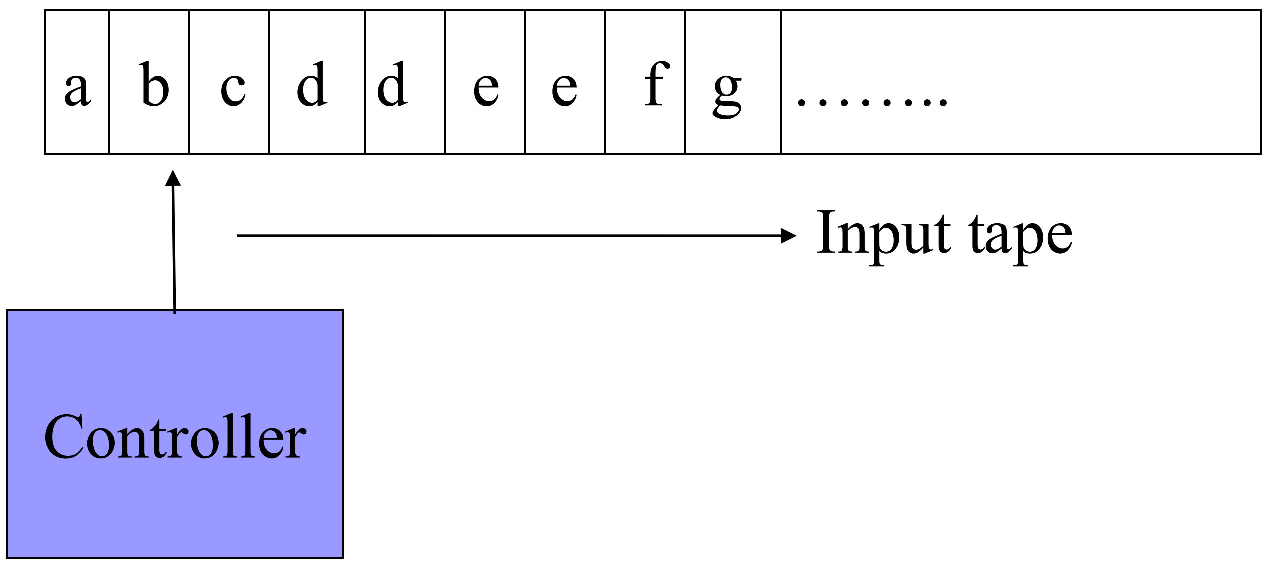
- **Finite State Machine (FSM)**
  - A machine or control structure designed to **automatically imitate a predetermined sequence of operations** or **respond to an encoded instruction**
  - Widely applied in many fields
  - An important tool in **computer science and engineering**

State + Input + Rules → State Transition

*

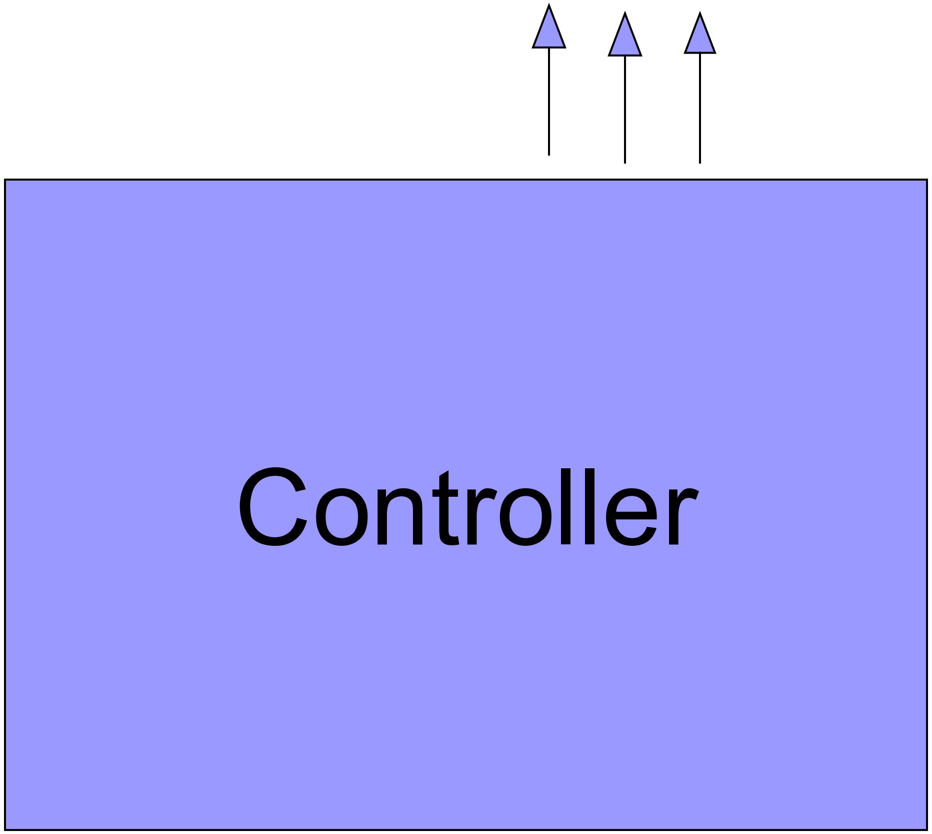
Model of FA

- FA can be understood as a **controller**
- It reads characters on an **input tape**



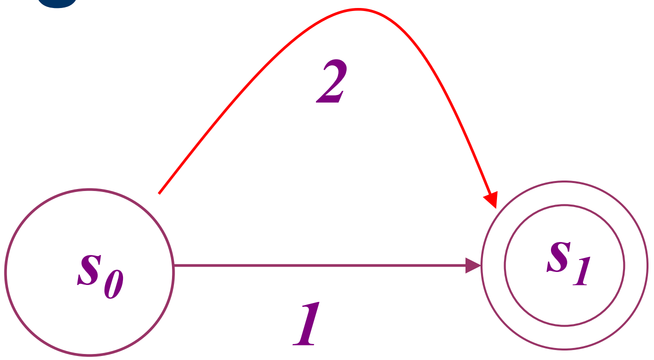
Example

Input（Characters）： 0 0 1 0



Deterministic Finite Automaton (DFA)

- A DFA is a **5-tuple**: **M = (S, Σ, δ, s_0, F)**
  - **S**: finite set of states; each element is a state
  - **Σ**: finite input alphabet;
  - **δ**: transition function: **S × Σ → S**
  - **s_0**: start state, **$s_0 \in S$**
  - **F**: set of final states, **$F \subseteq S$**



*

State Transition Matrix of a DFA

- **A DFA can be represented by a state transition matrix**
  - **Rows**: represent the states.
  - **Columns**: represent the input symbols.
  - **Matrix entries**: represent the value of δ(s, a), i.e., the next state when the automaton is in state *s* and reads input symbol *a*.

Example： DFA M= ({0,1,2,3}， {a,b}, δ, 0, {3})

where

δ(0,a)=1 δ(0,b)=2

δ(1,a)=3 δ(1,b)=2

δ(2,a)=1 δ(2,b)=3

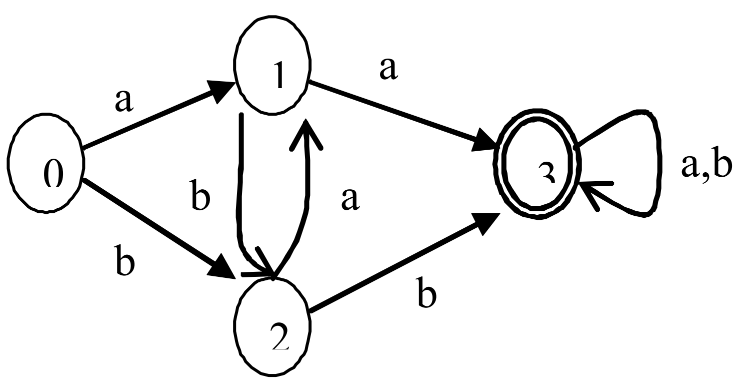
δ(3,a)=3 δ(3,b)=3

State	a	b
0	1	2
1	3	2
2	1	3
3	3	3

DFA and Transition Diagram

- **A DFA can also be represented as a deterministic state transition diagram**
  - **States** → **nodes** in the diagram.
  - **Transitions** → directed **edges** labeled with input symbols.
  - **No Ambiguity** → For each state and input symbol, there is exactly **one** outgoing edge

状态	a	b
0	1	2
1	3	2
2	1	3
3	3	3



*

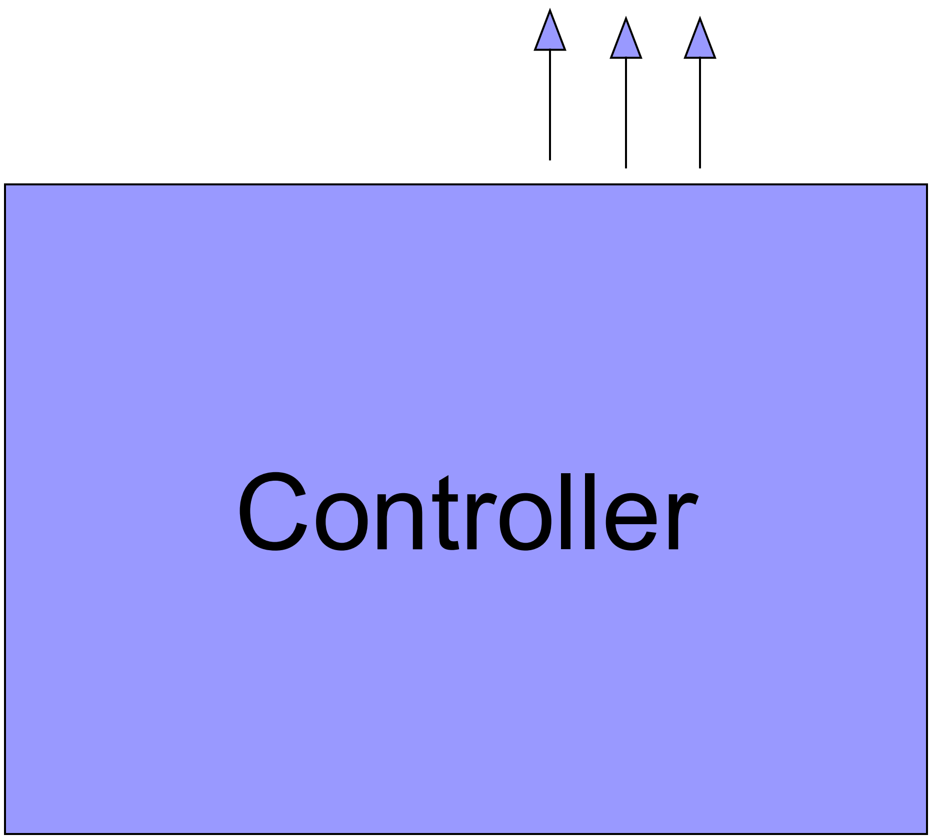
Extended Transition Function δ′

- δ′: the transition function that handles an input string (not just one symbol)
  - Definition:
    - $\delta': S \times \Sigma^* \rightarrow S$
    - For any state $s \in S$:
      - $\delta'(s, \varepsilon) = s$
      - If ω is a string and *a* is a symbol, then: $\delta'(s, \omega a) = \delta(\delta'(s, \omega), a)$
- For a DFA:
  - $\delta'(s, a) = \delta(\delta'(s, \varepsilon), a) = \delta(s, a)$
  - That is, for a single symbol, δ and δ′ are the same.

*

Extended Transition Function δ′

Input（Characters）： 0 0 1 0



$\delta'(q_0, \varepsilon) = q_0$

$\delta'(q_0, 0) = \delta(q_0, 0) = q_2$

$\delta'(q_0, 00) = \delta(q_2, 0) = q_0$

$\delta'(q_0, 001) = \delta(q_0, 1) = q_1$

$\delta'(q_0, 0010) = \delta(q_1, 0) = q_3$

Language Accepted by a DFA

- **Accepted string:**
  - A string is accepted if, after reading the entire input, the DFA ends in a final (accepting) state.
  - Otherwise, the string is rejected.
- **Language of a DFA: The set of all strings accepted by the DFA.**
  - $L(M) = \{\alpha \mid \delta'(s_0, \alpha) \in F\}$
- **Special case:**
  - If $s_0 \in F$, then the empty string ε is accepted.

Simulating a DFA

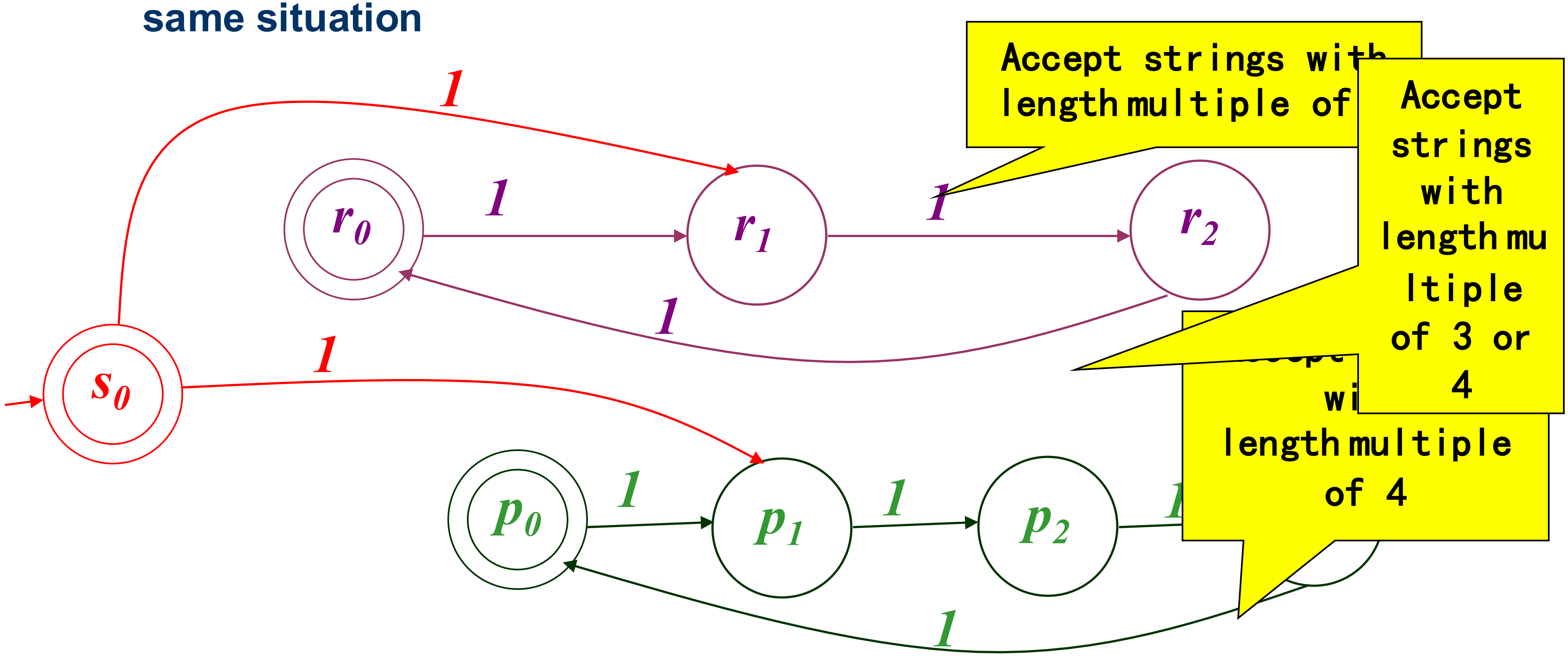
```
s = s0;
c = nextChar() ;
while ( c != eof ) {
      s = move(s, c);
      c = nextChar() ;
}
if ( s is in F ) return " yes " ;
else return "no " ;
```

Exercise

- River Crossing Puzzle
  - A man needs to ferry a wolf, a goat, and a cabbage across a river.
    - The boat holds only the man + one item.
    - Wolf + goat alone → wolf eats goat.
    - Goat + cabbage alone → goat eats cabbage.
- Task
  - Use a finite automaton to describe the crossing method.

Non-deterministic Finite Automaton (NFA)

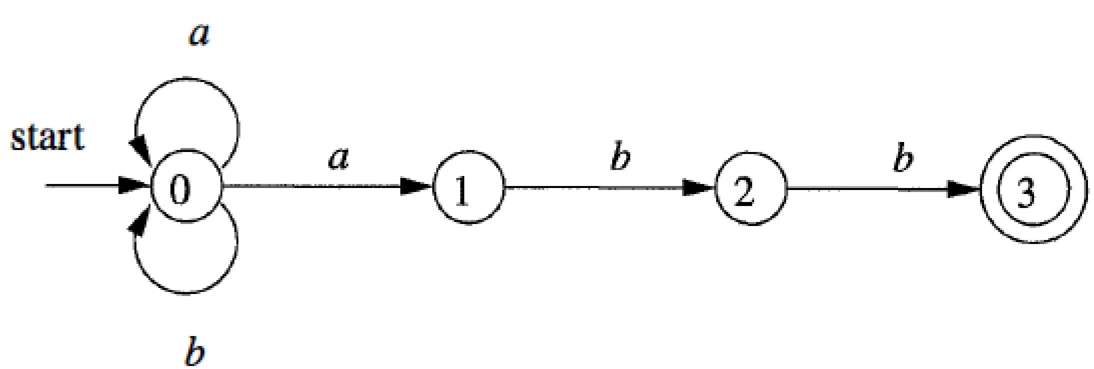
- Modify the DFA model so that in some state, for a given input, there can be multiple transitions to different states
- That is, the automaton has the ability to make different choices in the same situation



*

Non-deterministic Finite Automaton (NFA)

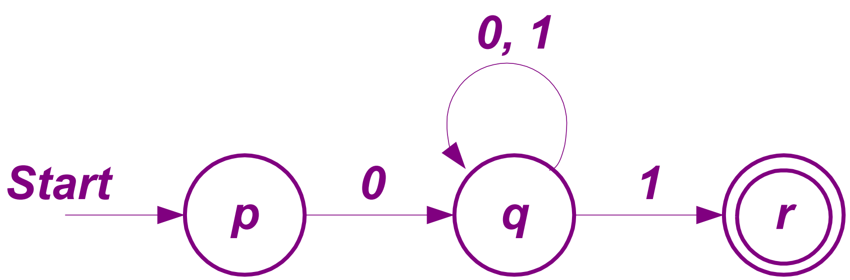
- An NFA **M** is a 5-tuple:
  - $M = (S, \Sigma, \delta, S_0, F)$
    - **S** and **Σ** are defined as before
    - **δ: S × Σ → 2^S** (subset of states)
      - For a state **s ∈ S** and input symbol **a**:
        - $\delta(s, a) = S' \subseteq S$
    - **S_0 ⊆ S** is a non-empty set of start states
    - **F ⊆ S** is the set of final states



*

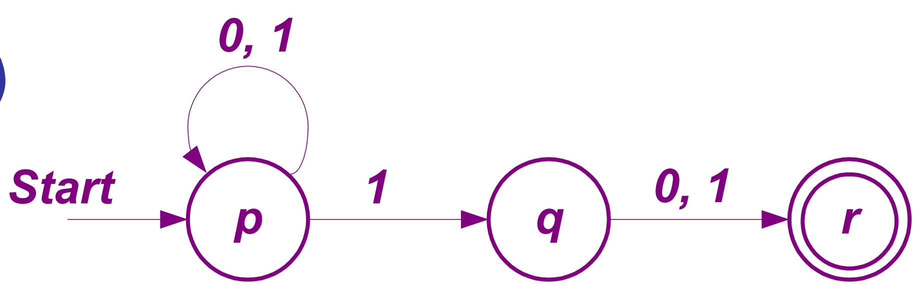
NFA Represented by Transition Diagram and Transition Matrix

(1)



	0	1
→ p	{ q }	ϕ
q	{ q }	{ q, r }
* r	ϕ	ϕ

(2)



	0	1
→ p	{ p }	{ p, q }
q	{ r }	{ r }
* r	ϕ	ϕ

Note:
Each entry in the transition matrix is a set of states
Can include the empty set (Φ), meaning some state -input combinations may have no transitions

*

Simulating a NFA

$S = \epsilon\text{-}closure(s_0);$

$c = nextChar();$

while (c != **eof**) {

  $S = \epsilon\text{-}closure(move(S, c));$

  $c = nextChar();$

}

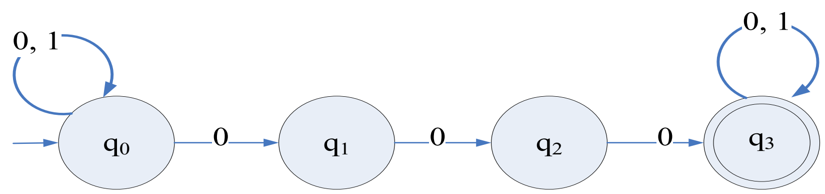
if ($S \cap F$!= $\emptyset$) **return** "yes";

else return "no";

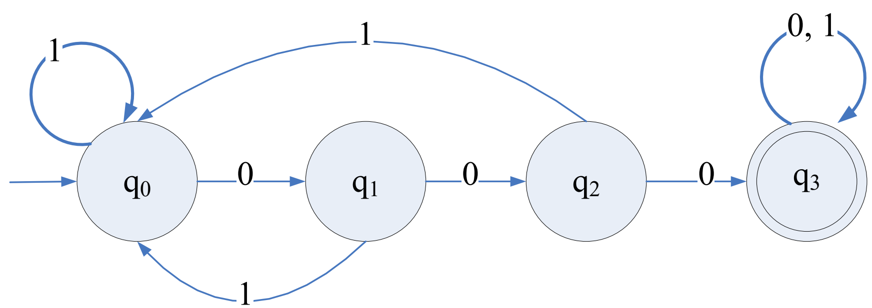
NFA and DFA Comparison

Example: Construct an NFA that recognizes a language over {0,1}

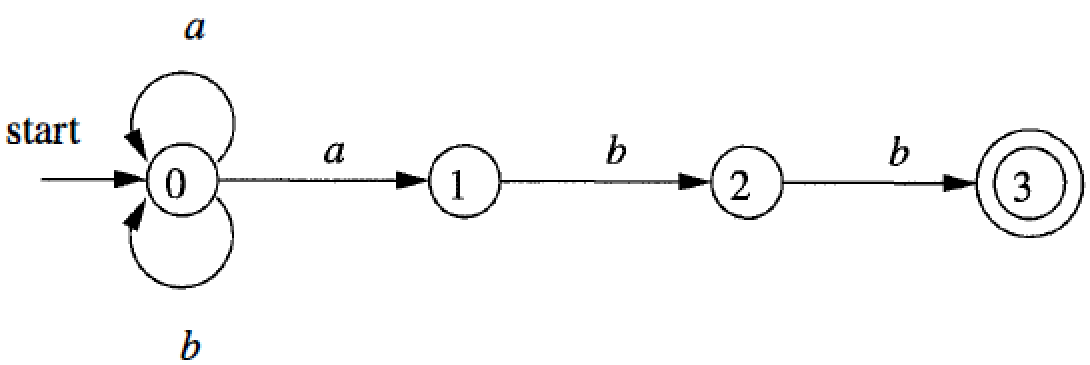
$L=\{x000y \mid x,y \in \{0,1\}^*\}$。



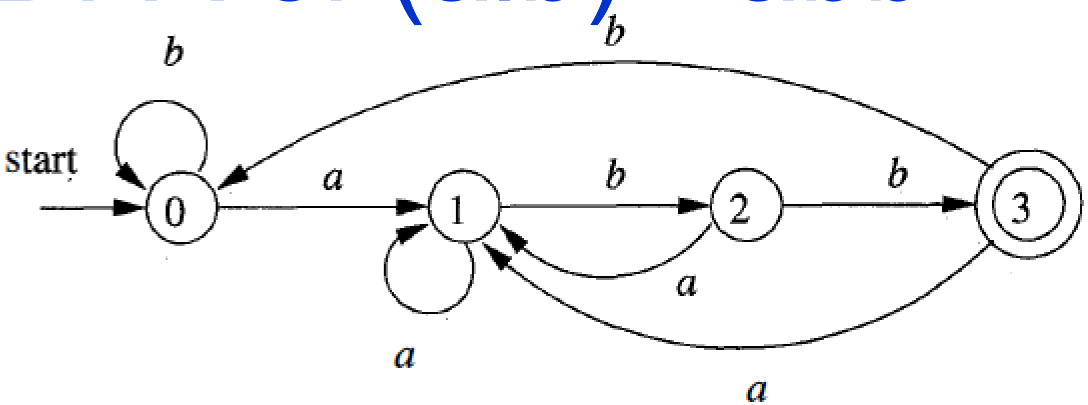
Corresponding DFA



NFA of (alb) * abb



DFA of (alb) * abb



NFA and DFA Comparison

- **Input Symbols**
  - In DFA: each state has a transition for **every symbol** in the alphabet
  - In NFA: a state may have **no transition** for some symbols, or may allow **ε-transitions**
- **Transition States**
  - In DFA: the next state is **deterministic** (only one)
  - In NFA: the next state is **non-deterministic** (can be multiple)

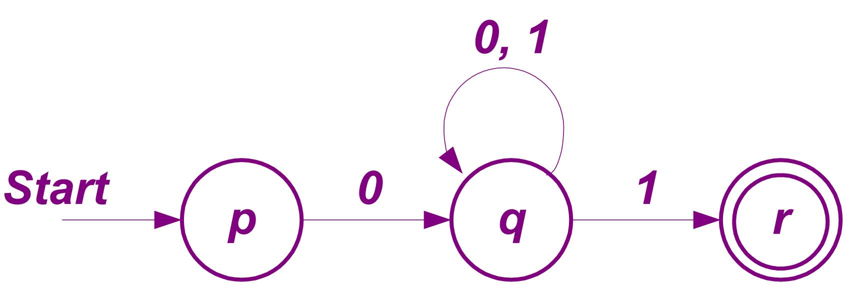
Extended Transition Function

- Difference from DFA:
  - $\delta: S \times \Sigma \rightarrow 2^S$
- Extended function:
  - $\delta': S \times \Sigma^* \rightarrow 2^S$
  - $\delta'(s, \varepsilon) = \{s\}$
  - $\delta'(s, \omega a) = \{p \mid \exists r \in \delta'(s, \omega) \wedge p \in \delta(r, a)\}$
    - $\delta'(s, \omega a)$ is the union of all possible states reached by reading **a** from each state in $\delta'(s, \omega)$.
    - If $\delta'(s, \omega) = \{r_1, r_2, \ldots, r_k\}$,then $\delta'(s, \omega a) = \bigcup \delta\,(r_i, a)$, where $\omega \in \Sigma^*, a \in \Sigma, r_i \in S$.

*

Extended transition function

	0	1
$\rightarrow p$	$\{q\}$	ϕ
q	$\{q\}$	$\{q, r\}$
$* r$	ϕ	ϕ



$\delta'(p, \varepsilon) = \{p\}$

$\delta'(p, 0) = \{q\}$

$\delta'(p, 01) = \{q, r\}$

$\delta'(p, 010) = \{q\}$

$\delta'(p, 0100) = \{q\}$

$\delta'(p, 01001) = \{q, r\}$

*

Language Accepted by an NFA

- **If after reading a string, the NFA enters a set of states that contains at least one final state in F, then the NFA accepts the string.**
- **For** $M = (S, \Sigma, \delta, S_0, F)$ **,the language of** M **is:**

$$L(M) = \{\alpha \mid \delta'(S_0, \alpha) \cap F \neq \emptyset\}$$

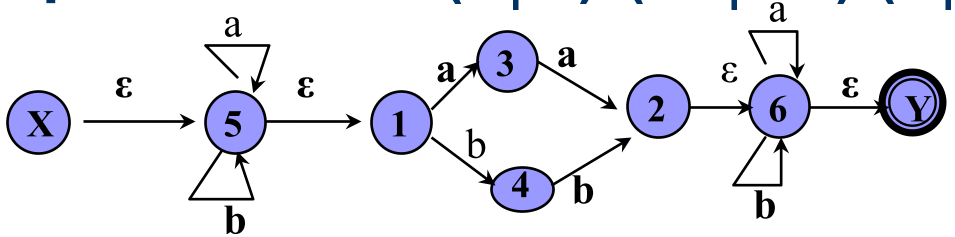
- **For any input string** $\alpha \in \Sigma^*$**:**
  - **If there exists a path from some start state** S_0 **to some final state in** F
  - **And the concatenation of edge labels equals** α **(ε-transitions ignored)**
  - **Then** α **is accepted (recognized) by the NFA** M**.**

Equivalence of NFA and DFA

- **A DFA is a special case of an NFA** → any language accepted by a DFA is also accepted by an NFA.
- **Question:** Can every language accepted by an NFA be accepted by some DFA?
  - **Answer: Yes**
- **Proof strategy**: For any NFA, construct a DFA that accepts the same language, where each DFA state corresponds to a **subset** of NFA states.

Example: Convert NFA to DFA

Regular expression V = (a | b)*(aa | bb) (a | b)*



1） use Subset Construction to create the state transition matrix

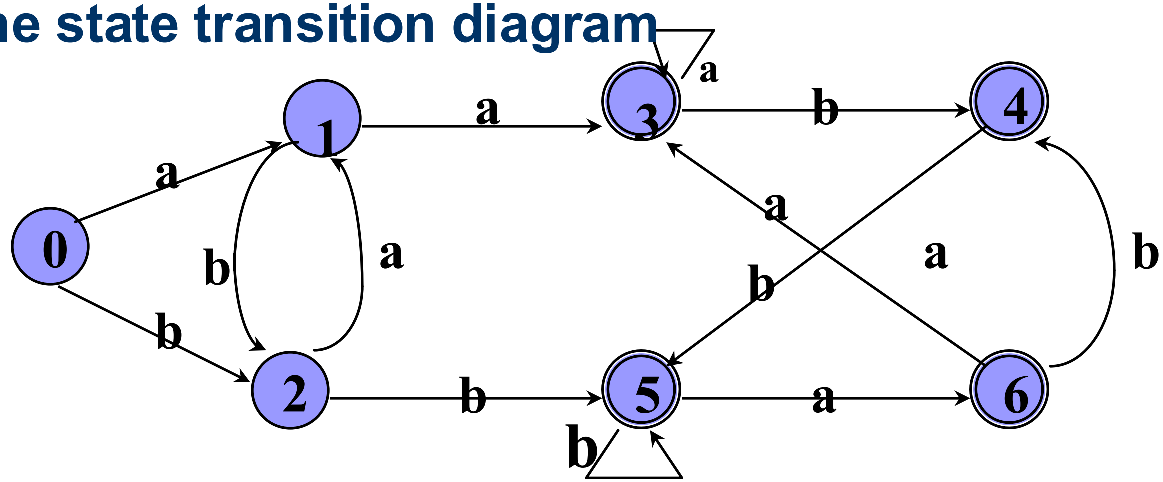
I	I_a	I_b
{X, 5, 1 }	{5, 3, 1 }	{5, 4, 1 }
{5, 3, 1 }	{5, 3, 1, 2, 6, Y }	{5, 4, 1 }
{5, 4,1 }	{5, 3, 1 }	{5, 4, 1, 2, 6, Y }
{5, 3, 1, 2, 6, Y }	{5, 3, 1, 2, 6, Y }	{5, 4, 1, 6, Y }
{5, 4, 1, 2, 6, Y}	{5, 3, 1, 6, Y }	{5, 4, 1, 2, 6, Y }
{5, 4, 1, 6, Y}	{5, 3, 1, 6, Y }	{5, 4, 1, 2, 6, Y }
{5, 3, 1, 6, Y }	{5, 3, 1, 2, 6, Y }	{5, 4, 1, 6, Y }

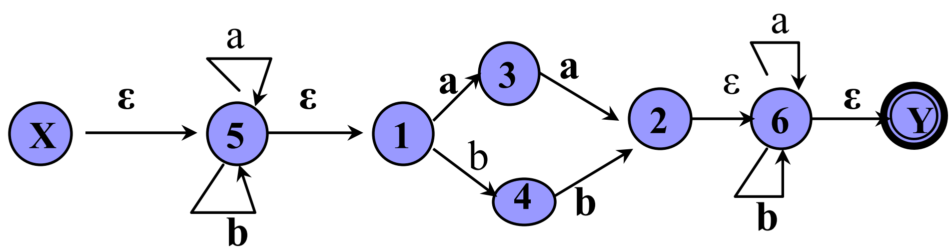
2) Rename the state subsets to get a new state transition matrix

I		I_a	I_b
{X, 5, 1 }	0	{5, 3, 1 }	{5, 4, 1 }
{5, 3, 1 }	1	{5, 3, 1, 2, 6, Y }	{5, 4, 1 }
{5, 4,1 }	2	{5, 3, 1 }	{5, 4, 1, 2, 6, Y }
{5, 3, 1, 2, 6, Y }	3	{5, 3, 1, 2, 6, Y }	{5, 4, 1, 6, Y }
{5, 4, 1, 6, Y }	4	{5, 3, 1, 6, Y }	{5, 4, 1, 2, 6, Y }
{5, 4, 1, 2, 6, Y}	5	{5, 3, 1, 6, Y }	{5, 4, 1, 2, 6, Y }
{5, 3, 1, 6, Y }	6	{5, 3, 1, 2, 6, Y }	{5, 4, 1, 6, Y }

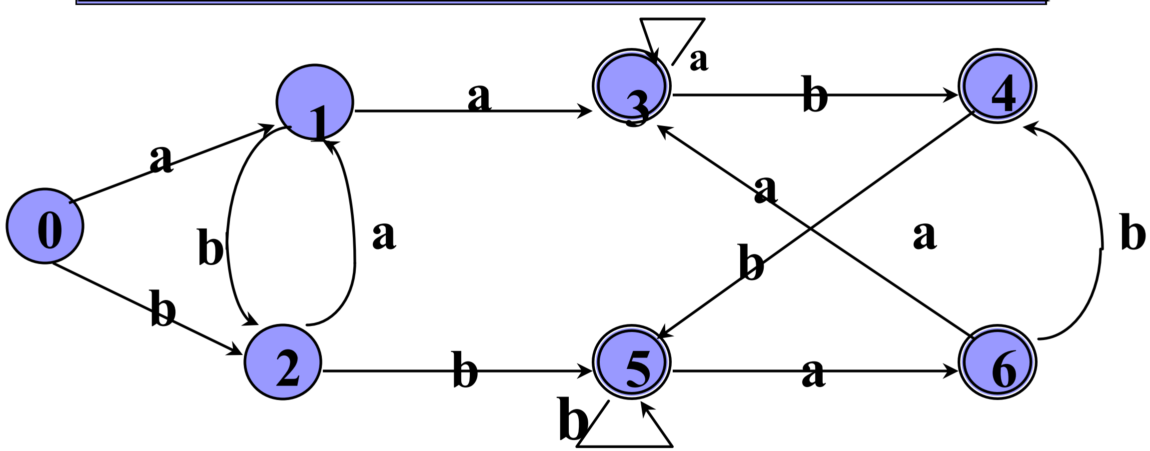
s	a	b
0	1	2
1	3	2
2	1	5
3	3	4
4	6	5
5	6	5
6	3	4

3) Draw the state transition diagram





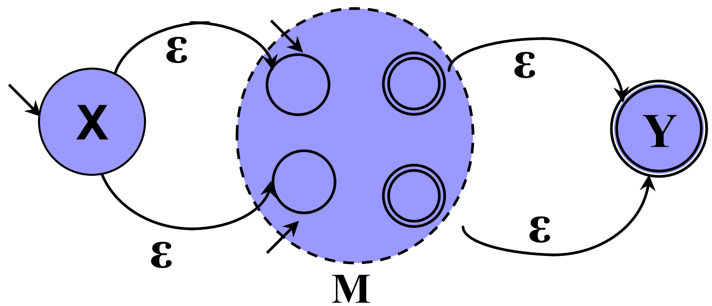
I		I_a		I_b	
{X, 5, 1 }	0	{5, 3, 1 }	1	{5, 4, 1 }	2
{5, 3, 1 }	1	{5, 3, 1, 2, 6, Y }	3	{5, 4, 1 }	2
{5, 4,1 }	2	{5, 3, 1 }	1	{5, 4, 1, 2, 6, Y }	5
{5, 3, 1, 2, 6, Y }	3	{5, 3, 1, 2, 6, Y }	3	{5, 4, 1, 6, Y }	4
{5, 4, 1, 6, Y }	4	{5, 3, 1, 6, Y }	6	{5, 4, 1, 2, 6, Y }	5
{5, 4, 1, 2, 6, Y}	5	{5, 3, 1, 6, Y }	6	{5, 4, 1, 2, 6, Y}	5
{5, 3, 1, 6, Y }	6	{5, 3, 1, 2, 6, Y }	3	{5, 4, 1, 6, Y }	4



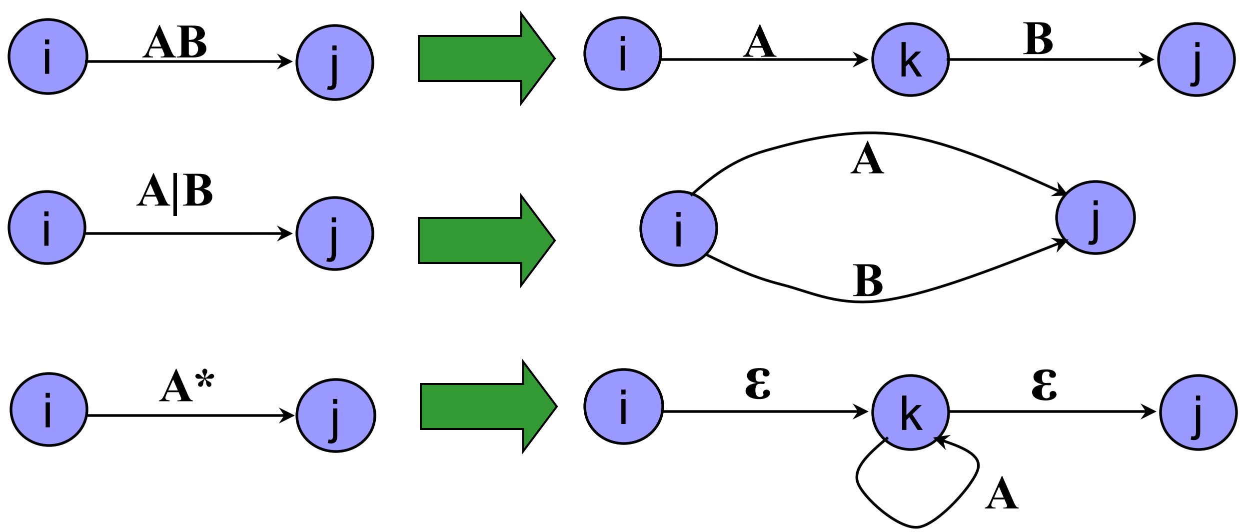
Proof of NFA and DFA Equivalence

(1) Modify the state-transition diagram of NFA M to obtain M′

Introduce new start node X and new final node Y, with X, Y ∉ S



(2) Extend nodes and add edges according to the following rules



*

(2) Further transform M' into a DFA

- Let **I** be a subset of M"s states. The **ε-CLOSURE(I)** is defined as:
  - If **q** ∈ **I**, then **q** ∈ **ε-CLOSURE(I)**
  - If **q** ∈ **I**, then any state **q'** reachable from **q** via any number of **ε-transitions** is also in **ε-CLOSURE(I)**
- Let **I** be a subset of M"s states and **a** ∈ **Σ**. Define:
  - $I_a = \varepsilon\text{−}CLOSURE(J)$
  - where **J** is the set of all states reachable from any state in **I** via an **a-transition**

*

(2) Further transform M′ into a DFA (continued)

1) **Construct the state transition matrix;**

Let Σ = {a, b}, create a table in the following form:

I	I_a	I_b
ε_CLOSURE({X})		

2) Treat each subset in the first column of the table as a **new state** and rename them
   - The subset in the first row, first column becomes the **DFA start state**
   - Any subset containing the original final state **Y** becomes a **DFA final state**

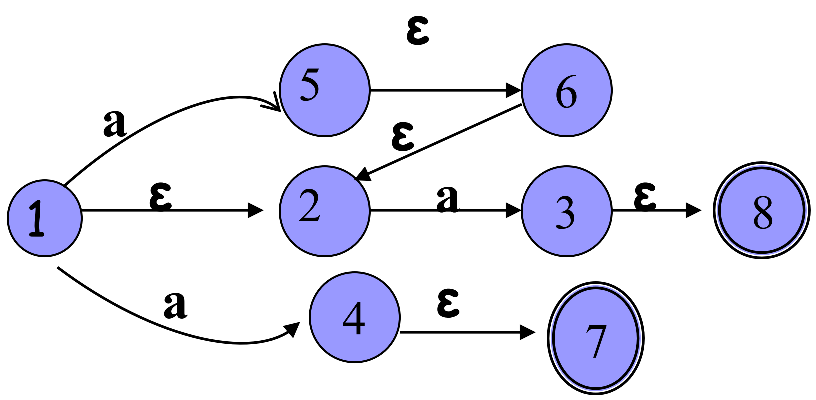
3) **Draw the new DFA**

*

Example. **For the state-transition diagram shown**

Let $I=\{1\}$, $\varepsilon\_CLOSURE(I)=\{1,2\}$

Let $I=\{1,2\}$, $I_a=\varepsilon\_CLOSURE\{5,4,3\}=\{5,6,2,4,7,3,8\}$

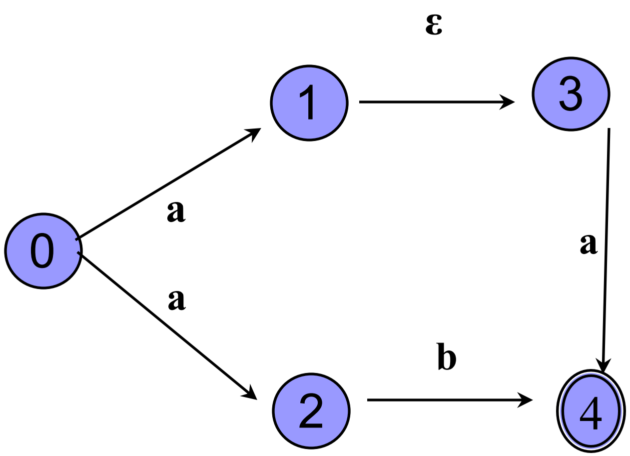


*

Quiz-Canvas

2) Lexical Analysis - Convert NFA to DFA using Subset Construction

Exercise



DFA State-Minimization

- **Definition**
  - Minimizing a DFA **M** means finding a DFA **M′** with fewer states such that **L(M) = L(M′)**
- **Terminology**
  - **Equivalent states (s and t):**
    - For any string α, if starting from **s** leads to a final state, then starting from **t** also leads to a final state, and vice versa
  - **Distinguishable states (s and t):**
    - States **s** and **t** are not equivalent
    - Example: **s** reaches a final state on input α, but **t** does not
    - In particular, **final and non-final states** are distinguishable

DFA State-Minimization

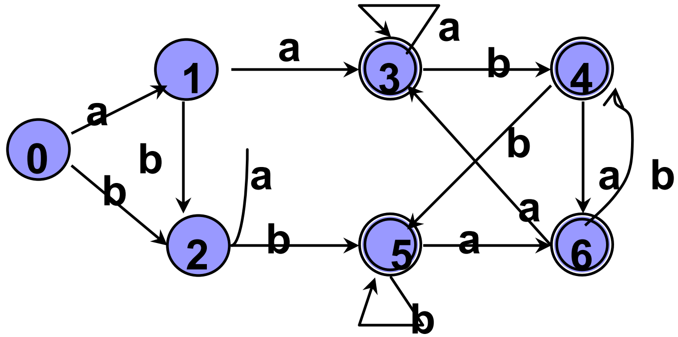
- The minimization process partitions the state set of DFA **M** into **disjoint subsets** such that:
  - **States in different subsets are distinguishable**
  - **States in the same subset are equivalent**

DFA State-Minimization

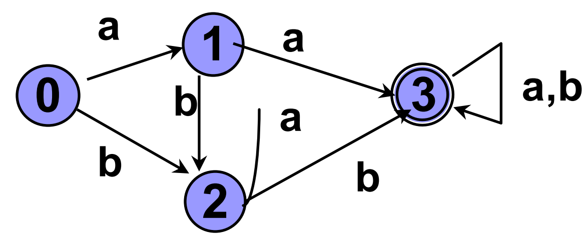
- **Initial Partition**
  - Divide the state set S into two subsets: $\prod= \{ I^{(1)}, I^{(2)} \}$, where $I^{(1)}$ is the set of final states, $I^{(2)}$ is the set of non-final states
- **Refinement**
  - Suppose the current partition is: $\prod= \{I^{(1)}, I^{(2)}, \dots, I^{(m)} \}$, For each subset $I^{(k)}$, check whether it can be further divided:
  - If $I^{(k)}_a$ is not contained in a single subset of $\prod$, split $I^{(k)}$
  - If $I^{(k)}_a$ spans N subsets, divide $I^{(k)}$ into N groups
- **Repeat**
  - Keep refining until no further splitting occurs — i.e., the number of subsets in $\prod$ no longer increases.

```
Input: DFA M = (Q, Σ, δ, q0, F)

1. Partition P = { F, Q \ F }    // 接受态和非接受态
2. repeat
3.       P_old = P
4.       For each group G in P:
5.           Split G into subgroups where states have different
             transitions under some symbol a ∈ Σ (according to P_old)
6.       Update P with these subgroups
7. until P = P_old    // 没有进一步划分

8. Construct minimized DFA M':
       - States = groups in P
       - Start state = group containing q0
       - Accept states = groups containing F
       - Transitions: δ'([q], a) = [δ(q, a)]

Output: Minimized DFA M'
```

Before



After



Initial partition $\prod_0 = \{ I^{(1)}, I^{(2)} \}$, $I^{(1)} = \{3, 4, 5, 6\}$, $I^{(2)} = \{0, 1, 2\}$

Examine $I^{(1)}_a = \{3, 6\} \subseteq \{3, 4, 5, 6\}$

$I^{(1)}_b = \{4, 5\} \subseteq \{3, 4, 5, 6\}$

$I^{(1)}$ Unchanged

Examine $I^{(2)}_a = \{1, 3\}$, Since $\{1\}_a = \{3\}$, $\{0,2\}_a = \{1\}$

Split {0, 1, 2} into {1}, {0, 2}

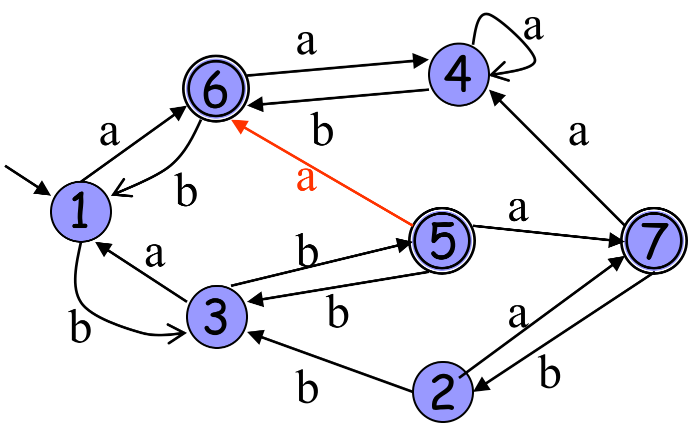
$\prod_1 = \{\{1\}, \{0, 2\}, \{3, 4, 5, 6\}\}$

Examine $\{0, 2\}_b = \{2, 5\}$, so, split {0, 2} into {0}, {2}

$\prod_2 = \{\{0\}, \{1\}, \{2\}, \{3, 4, 5, 6\}\}$

Let state 3 represent the subset {3, 4, 5, 6}, Draw new DFA

*



$\Pi_0=\{\{1,2,3,4\},\{5,6,7\}\}$

$\{1,2,3,4\}_a=\{6,7,1,4\}$ is not contained in a single subset of Π_0, $\rightarrow$ need to split.

$\{1,2\}_a=\{6,7\} \subseteq \{5,6,7\}$,

$\{3,4\}_a=\{1,4\} \subseteq \{1,2,3,4\}$,

$\rightarrow \Pi_1=\{\{1,2\},\{3,4\},\{5,6,7\}\}$

$\{3,4\}_a=\{1,4\}$, not contained in a single subset of Π_1

$\rightarrow \Pi_2=\{\{1,2\},\{3\},\{4\},\{5,6,7\}\}$

$\{5,6,7\}_a=\{7,4\}$,

$\rightarrow \Pi_3=\{\{1,2\},\{3\},\{4\},\{5\},\{6,7\}\}$

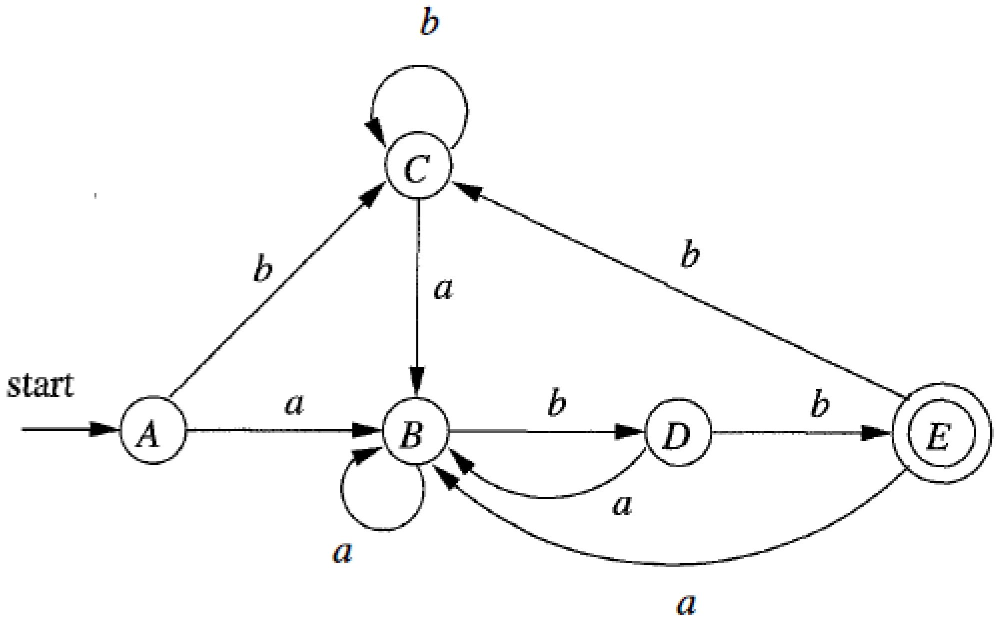
*

Quiz-Canvas

2) Lexical Analysis - DFA State-minimization

Exericise

DFA State-Minimization



Learning Approach

Lexical rules

→ Regular expressions

→ Automata (NFA → DFA → Minimized DFA)

→ Scanner

Wait a minute…

Equivalence of Regular Grammars and Finite Automata

More

Regular Grammar → Automata

Automata → Regular Grammar

Regular Grammars and Finite Automata

- **If L(G) = L(M) → G and M are equivalent**
  - **Every right/left-linear grammar G → some finite automaton M**
  - **Every finite automaton M → some right/left-linear grammar G**

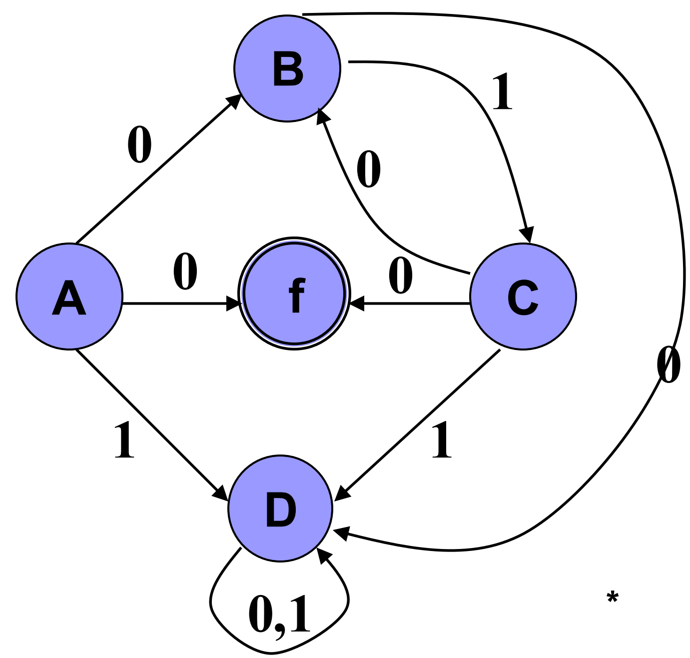
G=({A,B,C,D},{0,1},A, ε),

A →0|0B|1D

B →0D|1C

C →0|0B|1D

D →0D|1D



*

Constructing NFA from **Right-Linear Grammar**

Given $G=(V_N, V_T, S, \mathscr{E})$。

Treat each nonterminal in V_N as a state; add new final state $f \notin V_N$

Define $M= (V_N \cup\{f\}, V_T, \delta, S, \{f\})$

- **Transition rules:**

  (a) If $A \rightarrow a$, then $\delta(A, a) = f$;

  (b) If $A \rightarrow aA_1 | aA_2 | \ldots | aA_k$,then $\delta(A, a) = \{A_1, \ldots, A_k\}$;

  $A \in V_N$, $a \in V_T \cup \{\varepsilon\}$

Constructing NFA from **Left-Linear Grammar**

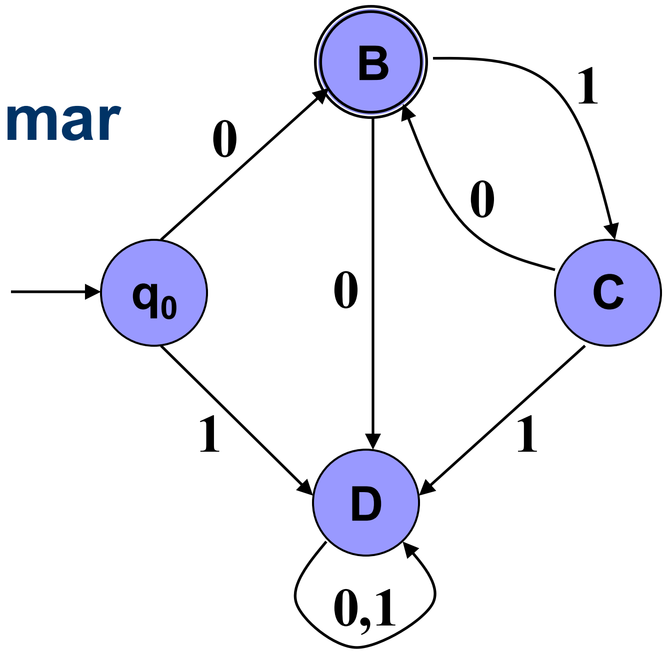
Example: Given left-linear grammar

$G=(\{B,C,D\},\{0,1\},B,\ £)$,

$B \rightarrow C0|0$

$C \rightarrow B1$

$D \rightarrow D0|D1|C1|B0|1$



Given $G=(V_N, V_T, S,\ £)$

- **Treat each nonterminal in V_N as a state; add new state $q_0 \notin V_N$**
- **Define $M = (V_N \cup \{q_0\}, V_T, \delta, q_0, \{S\})$**
- **Transition rules $(a \in V_T \cup \{\varepsilon\})$:**
  - **(a) If $A \rightarrow a$, then $\delta(q_0, a) = A$**
  - **(b) If $A_1 \rightarrow Aa$, $A_2 \rightarrow Aa$, … $A_k \rightarrow Aa$, then $\delta(A, a) = \{A_1, \ldots, A_k\}$**

*

Exercise

(1) Right-Linear Grammar → Equivalent FA

G=({A,B},{l,d},A, £),

A →l | lB

B →l | d | lB | dB

(2) Left-Linear Grammar → Equivalent FA

G=({A},{l,d},A, £),

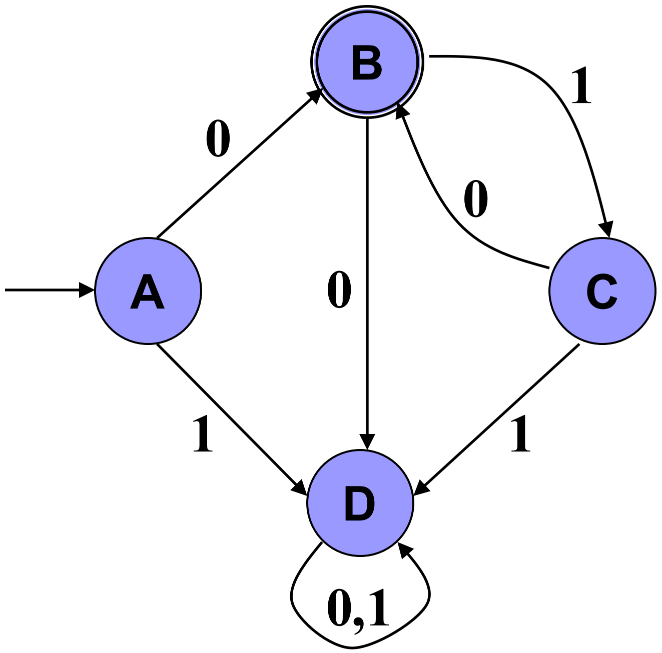
A →l | Al | Ad

More

Regular Grammar → Automata

Automata → Regular Grammar

Automata → Regular Grammar



Right-Linear Grammar

G=({A,B,C,D},{0,1},A, £),

A →0|0B|1D

B →0D|1C

C → 0|0B|1D

D →0D|1D

Left-Linear Grammar

G=({B,C,D},{0,1},B, £),

B →C0|0

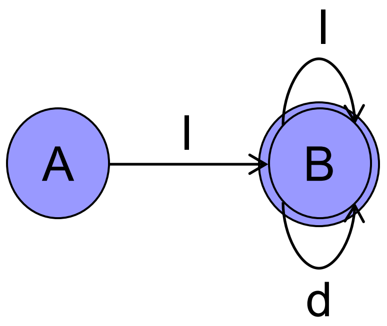
C → B1

D →D0|D1|C1|B0|1

*

Exercise

Generate the left-linear and right-linear grammars equivalent to the following DFA



Quiz-Canvas

2) Lexical Analysis - Regular Grammar and Automata

Outline

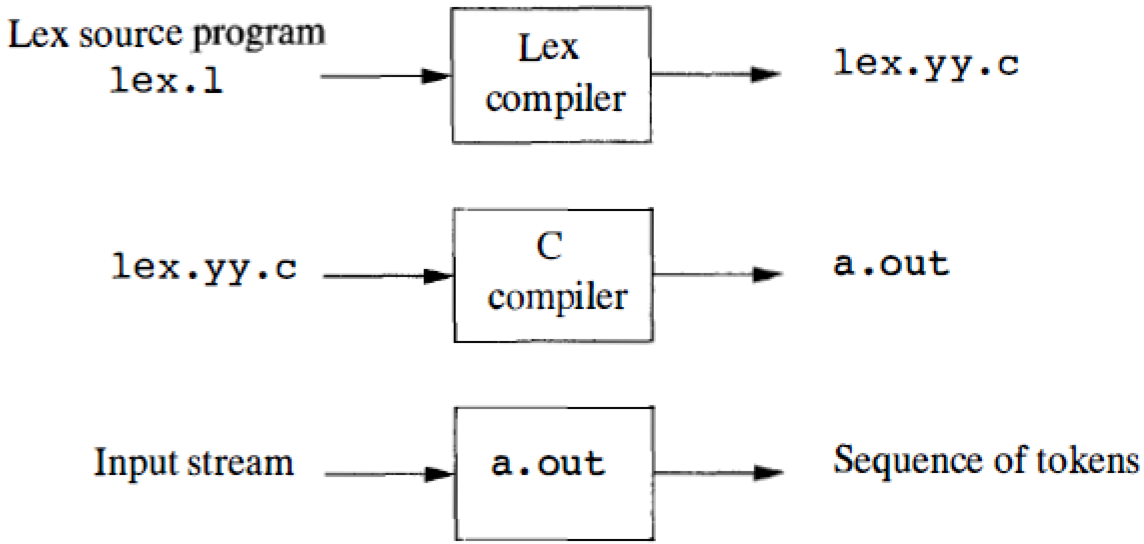
3.1 Requirements for Scanner

3.2 Design of Scanner

3.3 Regular Expressions and Finite Automata

3.4 Automatic Generation of Scanner

Automatic generation of Scanner: LEX

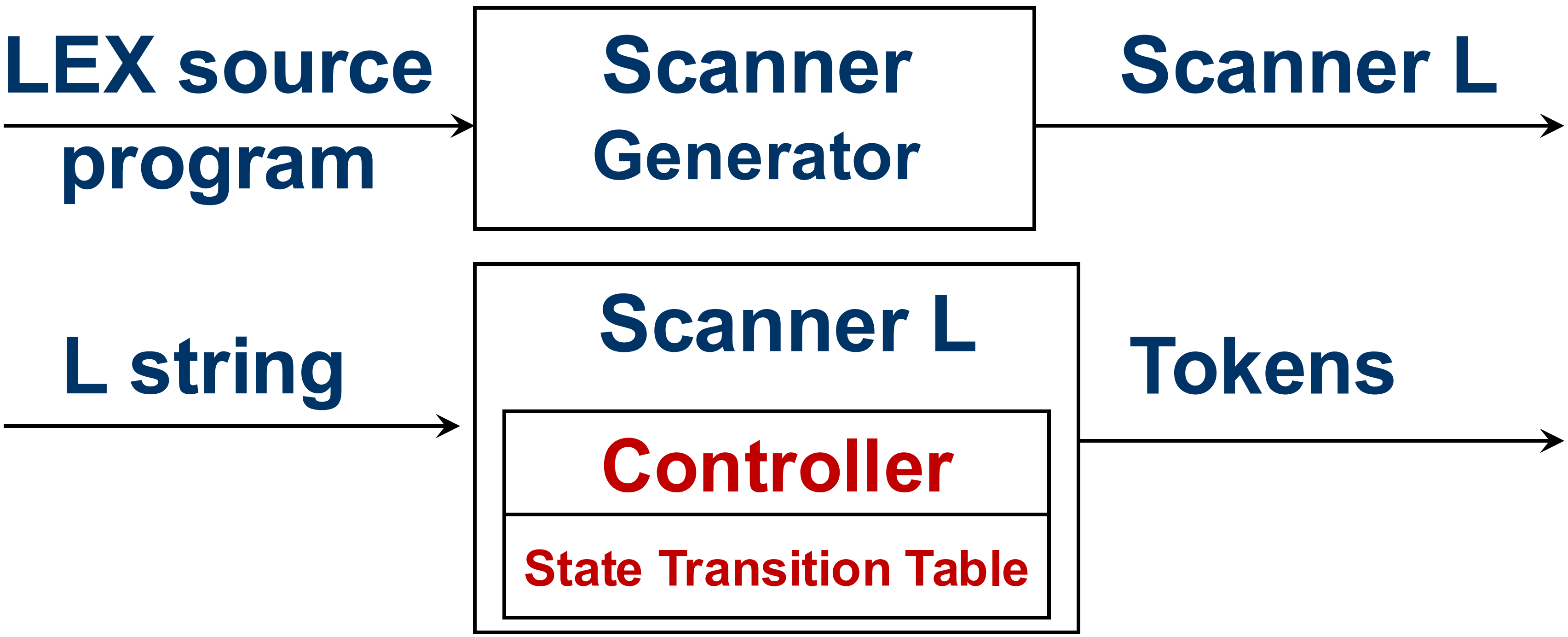


Lex 和 Yacc 从入门到精通

熊春雷

Automatic generation of Scanner: LEX

- LEX program = regular expressions + corresponding Actions
- Action: small code specifying what to do when a token is recognized



LEX source program

```
declarations
%%
translation rules
%%
auxiliary functions
```

(1) Declarations / Auxiliary Definitions

Auxiliary Definitions of Regular Expression

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

r_i: regular expression

d_i: abbreviation for r_i

r_i can use only characters from Σ and previously defined abbreviations d_1, d_2, … , d_{i-1}

(2) Translation rules

P_1	$\{A_1\}$
P_2	$\{A_2\}$
...	
P_m	$\{A_m\}$

P_i:a regular expression over $\sum \cup \{d_1, d_2, . . . ,d_n\}$

A_i:action to be taken when token P_i is recognized; a small piece of code

*

Example: LEX program to recognize tokens of a small language

AUXILIARY DEFINITIONS

letter→ A | B | . . . | Z

digit→ 0 | 1 | . . . | 9

RECOGNITION RULES /* 识别规则 */

Regular Expression

1	DIM	{RETURN (1, _)}
2	IF	{RETURN (2, _)}
3	DO	{RETURN (3, _)}
4	STOP	{RETURN (4, _)}
5	END	{RETURN (5, _)}
6	letter(letter | digit)*	{RETURN (6, getSymbolTableEntryPoint())}
7	digit (digit)*	{RETURN (7, getConstTableEntryPoint())}
8	=	{RETURN (8, _)}
9	+	{RETURN (9, _)}
10	*	{RETURN (10, _)}
11	**	{RETURN (11, _)}
12	,	{RETURN (12, _)}
13	(	{RETURN (13, _)}
14	)	{RETURN (14, _)}

*

Example: Declarations

Identifier

letter →A | B | … | Z

digit → 0 | 1 | … | 9

iden →letter (letter | digit)*

Integer constant

integer→ digit(digit)*

sign →+ | - | ε

signedinteger →sign integer

Real constant without exponent

decimal →signedinteger . integer
| signedinteger . | sign . Integer

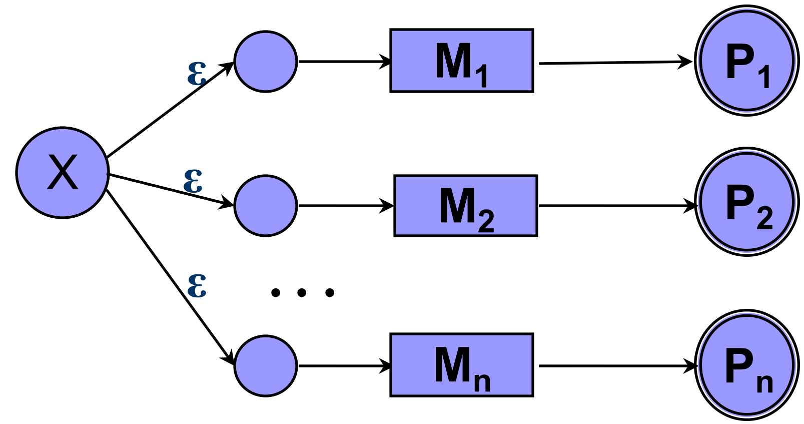
Example: 123.456 | -45. | +.78

Real constant with exponent

exponential →(decimal
| signedinteger) E signedinteger

*

Implementation of LEX



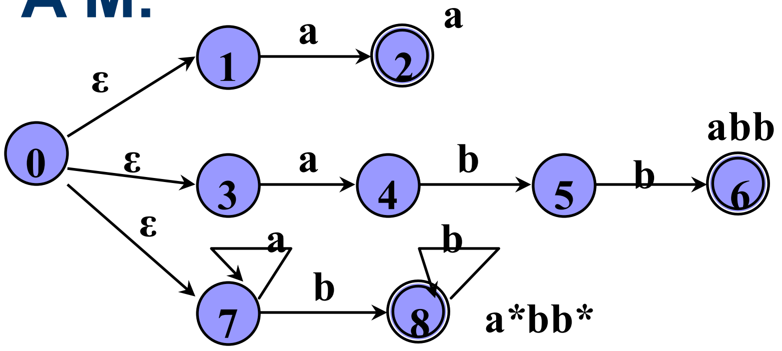
- **Method**
  - **LEX compiler transforms a LEX source program into a scanner by constructing the corresponding DFA**
- **Steps**
  - **Construct an NFA M_i for each recognition rule P_i**
  - **Introduce a new start state X, combine NFAs into NFA M**
  - **Convert M to DFA using subset construction and simplify**
  - **Transform the DFA into a Scanner**
- **Notes**
  - **Match the longest substring (longest match principle)**
  - **If multiple longest substrings match, choose the earliest P_i (priority match principle)**

*

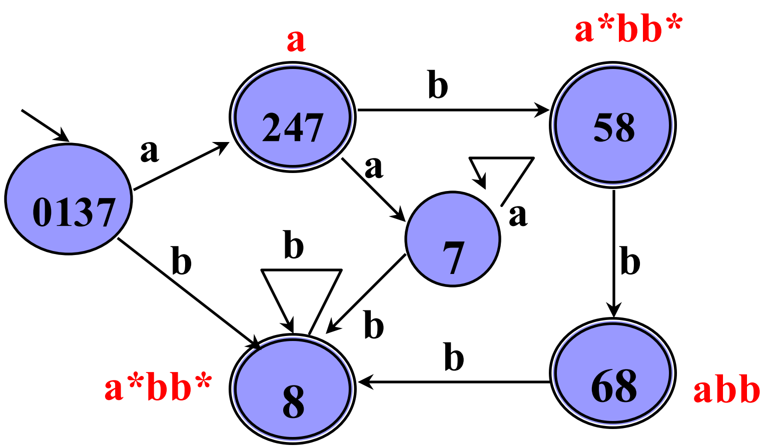
Example. LEX program:

a { }
abb { }
a*bb* { }

NFA M:



State	a	b	Tokens
0 1 3 7	2 4 7	8	
2 4 7	7	5 8	a
8		8	a*bb*
7	7	8	
5 8		6 8	a*bb*
6 8		8	abb



Longest match principle
Priority match principle

Input: abbbabb

Output: abbb abb

Survey

- **Algorithm 1 (Thompson's Algorithm)**
  - **Regular Expression → NFA**
- **Algorithm 2 (Subset Construction)**
  - **NFA → DFA**
- **Algorithm 3 (State Equivalence)**
  - **DFA state-minimization**
- **Others:**
  - **Conversion between FA and regular grammar**

Exercise

- **P64, 12 (a)**
  - NFA → DFA
- **P64 12(b)**
  - DFA minimization
- **P65， 15**
  - Left- and right-linear grammar conversion

*

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

감사합니다
Korean

Tack så mycket
Swedish

धन्यवाद
Hindi

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

谢谢
Chinese

Dankon
Esperanto

Thank You !

Tak
Danish

ありがとうございます
Japanese

Trugarez
Breton

Danke
German

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic