DCS290

Compilation Principle

编 译 原 理

第三章 词法分析（2）

郑馥丹
zhengfd5@mail.sysu.edu.cn

CONTENTS

目录

01 概述 Introduction

02 词法规范 Lexical Specification

03 有穷自动机 Finite Automata

04 转换和等价 Transformation and Equivalence

05 词法分析实践 Lexical Analysis in Practice

1. 转换图[Transition Diagram]

- **结点[Note]：状态**
  - 词法分析器在扫描输入串的过程中寻找和某个模式匹配的词素
  - 转换图的每一个状态代表一个在此过程中可能发生的情况
  - **初始状态**(Start/Initial State)：**只有一个**，一般由没有出发结点的箭头表示
  - **最终状态**(Accepting/Final States)：**可以有多个，用双圈表示**

- **边[Edge]：有向[directed]，标记有符号[symbol(s)]**
  - 从一个状态指向另一个状态
  - 如果处于某个状态S，且下一个输入符号是a，则会寻找一条从S状态离开且标号为a的边

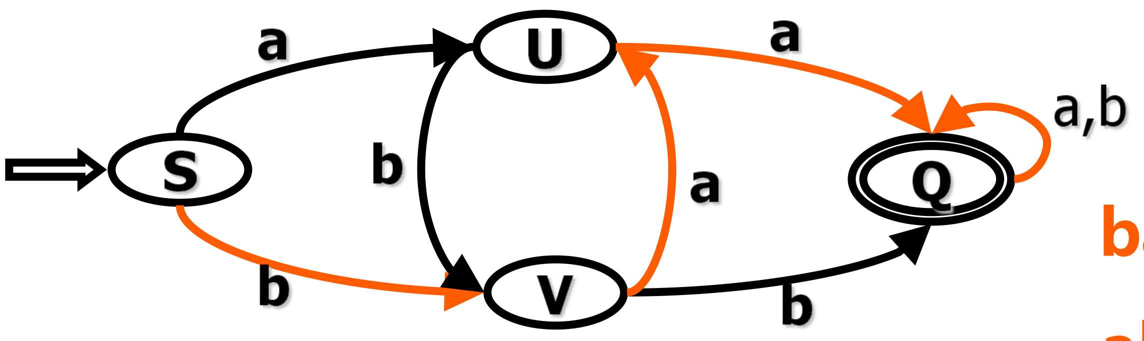
2. 有穷自动机[Finite Automata, FA]

- 正则表达是定义[Regular Expression = **specification**]
- 自动机是实现[Finite Automata = **implementation**]

- 自动机[Automata]：一个机器或一个程序
- **有穷**自动机[Finite Automata, FA]：具有**有穷个状态**的程序
- 有穷自动机基于转换图
  - **有状态和标记的边**
  - **有一个唯一的开始状态和一个或多个最终状态**

2. 有穷自动机[Finite Automata, FA]

- 有穷自动机FA是一个对字符串进行**分类（接受、拒绝）**的程序
  - 是一个识别语言的程序
  - Lex tool：将REs(specification)转换成FAs(implementation)
  - 对于给定的字符串x，如果存在一条从FA的**初始结点**到某一个最终结点的通路，且该**通路上所有边上的符号连接成的字符串等于x**，则称**x可被FA识别**，或称**x被FA接受**
    - ✓ 否则，称**x被FA拒绝**



baab被FA接受

abcd被FA拒绝

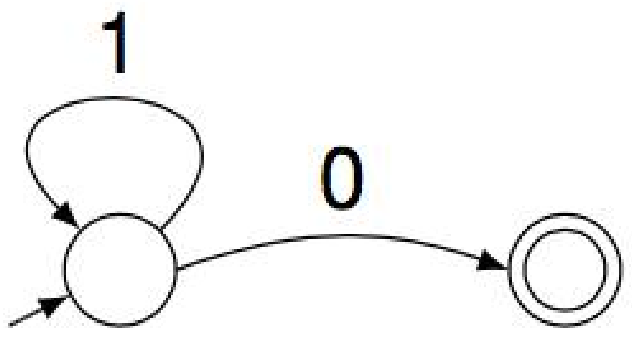
- **FA所能表达的语言即为该FA接受的字符串集合**
  - L(FA) ≡ L(RE)

2. 有穷自动机[Finite Automata, FA]

- 例：有右图FA

（1）判断以下字符串是否被FA接受

- ✓ 0 ✓
- ✓ 1 ✗
- ✓ 11110 ✓
- ✓ 11101 ✗
- ✓ 11100 ✗
- ✓ 1111110 ✓



（2）在字母表∑ = {0, 1}上，这个FA的语言是什么？

- ✓ **任意多个‘1’后跟一个‘0’**

2. 有穷自动机[Finite Automata, FA]

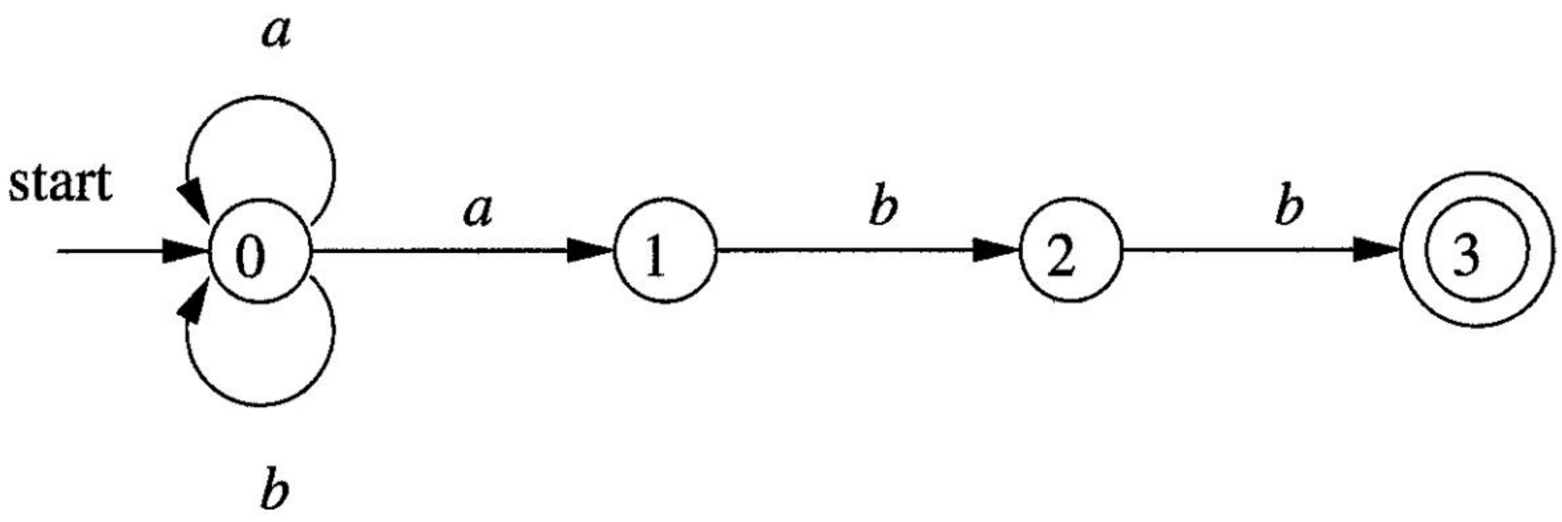
- 例：有右图FA

（1）在字母表∑ = {a, b}上，这个FA的语言是什么?

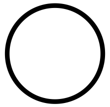
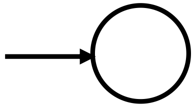
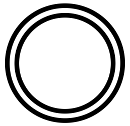
✓ **所有以若干个a和b的组成的串后跟‘abb’的字符串**

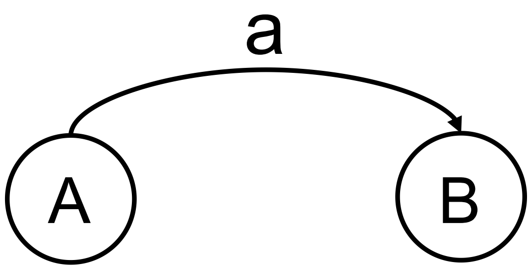
（2）尝试用正则表达式表达这个语言

✓ **L(RE) = (a|b)*abb**



2. 有穷自动机[Finite Automata, FA]

- 一个有穷自动机是一个五元组：$\mathbf{M = (S, \Sigma, \delta, s_0, F)}$，其中：
  - $\mathbf{S}$：有穷状态集合 
  - $\mathbf{\Sigma}$：输入字母表
  - $\boldsymbol{\delta}$：一个**转换函数[transition function]**，它为每个状态和$\Sigma \cup \{\varepsilon\}$中的每个符号都给出了相应的后继状态集合
  - $\mathbf{s_0} \in \mathbf{S}$：开始状态/初始状态 
  - $\mathbf{F} \subseteq \mathbf{S}$：接受状态/终止状态的**集合** 



$\boldsymbol{\delta}$: $A \xrightarrow{a} B$

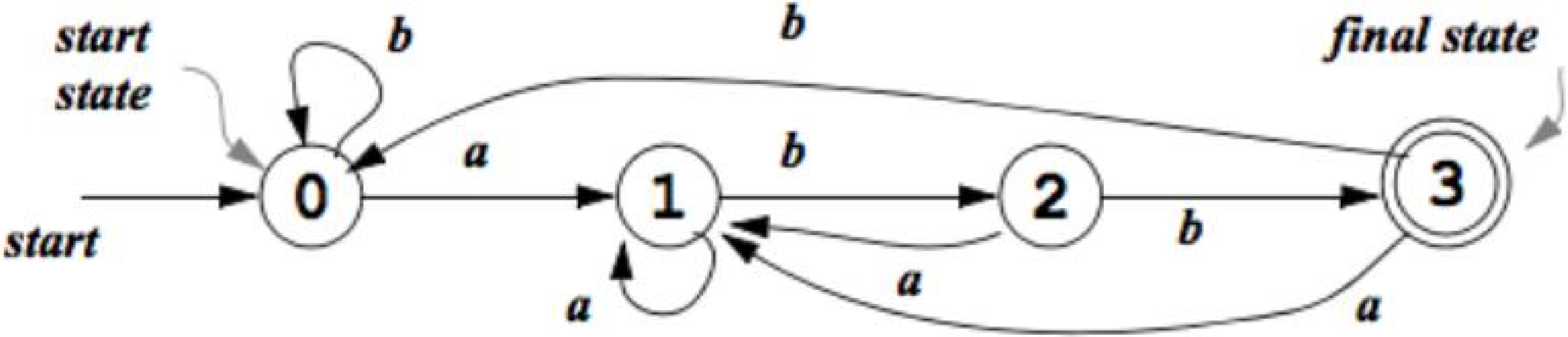
3. DFA和NFA

- **DFA**(Deterministic Finite Automata): 机器在任何给定时间只能以**一种**状态存在(**确定**)
  - 每个状态对于每个输入只对应**一个转换**
  - **没有ε转移[ε-moves]**
  - 只通过状态图的**一条路径**
- **NFA**(Nondeterministic Finite Automata)：机器可以同时以**多种**状态存在(**非确定**)
  - 在给定状态下，对一个输入可以有**多个转换**
  - 可以**有ε转移[ε-moves]**
  - **多条路径**：可以选择采取哪条路径
    - ✓ 如果其中**某些**路径在输入结束时导致接受状态，则NFA接受

对DFA：δ：$S \times \Sigma \rightarrow S$
对NFA：δ：$S \times \Sigma \rightarrow 2^S$
对ε-NFA：δ：$S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$

3. DFA和NFA

- DFA示例
  - **只有一种**可能的转移[moves]序列：**要么**导致最终状态并**接受**，**要么拒绝**输入字符串
  - 输入字符串：aabb
  - 成功序列： $0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$ **接受**

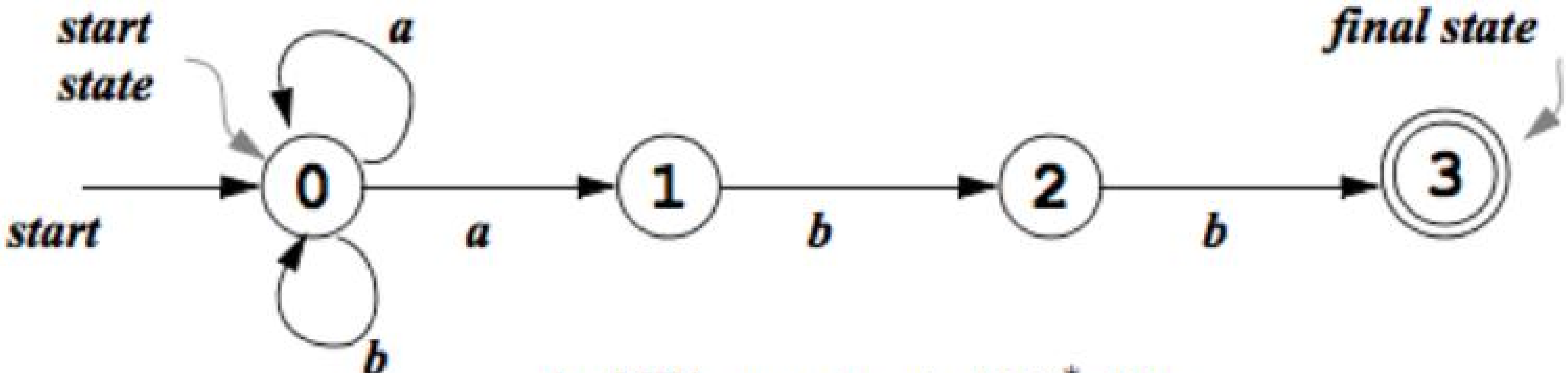


A DFA accepts $(a|b)^*abb$

3. DFA和NFA

- NFA示例
  - 有**多个**可能的moves：**只要有一个moves序列导致最终状态，即认为接受**
  - 输入字符串：aabb
  - 成功序列：$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$
  - 不成功序列：$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

接受



An NFA accepts $(a|b)^*abb$

CONTENTS

目 录

01 概述 Introduction

02 词法规范 Lexical Specification

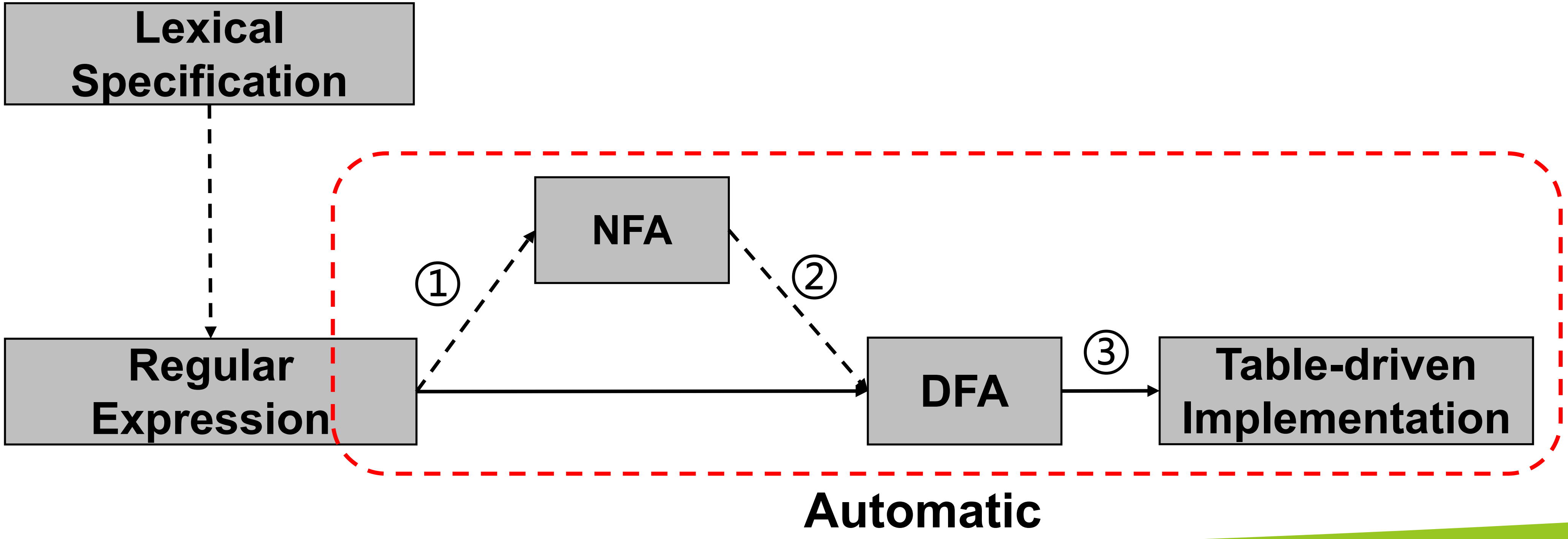
03 有穷自动机 Finite Automata

04 转换和等价 Transformation and Equivalence

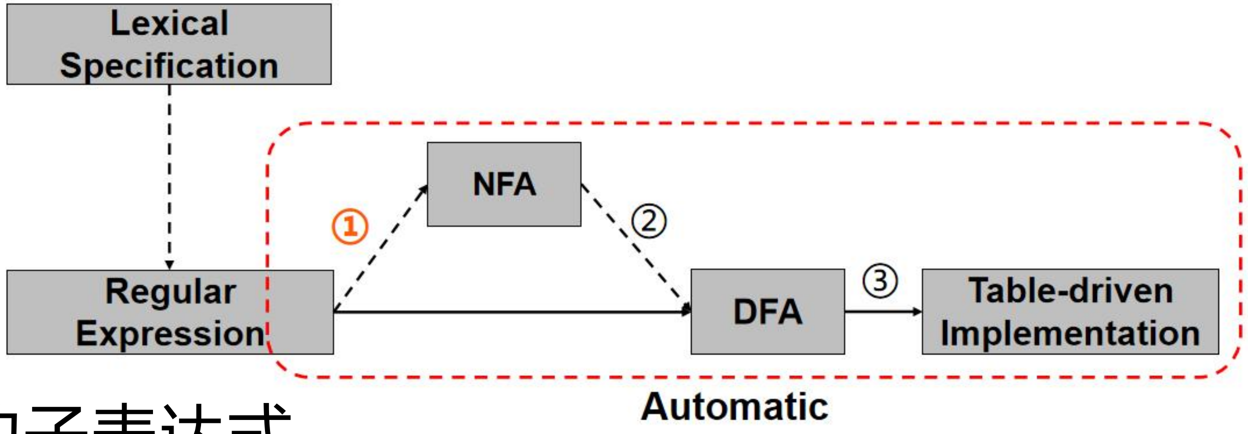
05 词法分析实践 Lexical Analysis in Practice

1. 转换流程[Conversion Flow]

- 流程：RE→NFA→DFA→Table-drive Implementation
  - ① RE→NFA
  - ② NFA→DFA
  - ③ DFA→Table-drive Implementation

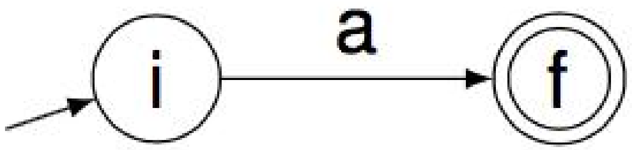


2. RE→NFA



- **RE→NFA**采用**Thompson构造算法**
  - 输入：字母表Σ上的一个正则表达式r
  - 输出：一个接受L(r)的NFA N
  - 方法：对r进行分析，分解出组成它的子表达式
- **Step 1: 处理原子REs（基本规则）**
  - 对于表达式**ε**：
    - 增加新状态i，作为NFA的初始状态
    - 增加另一个新状态f，作为NFA的接受状态

  - 对于表达式**a**：
    - 同上



Thompson

Ken Thompson

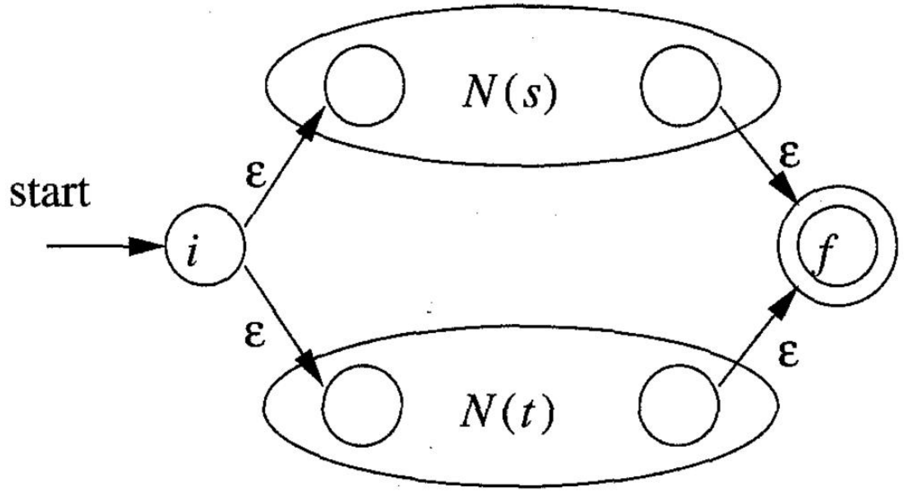
Kenneth Lane Thompson (born February 4, 1943) is an American pioneer of computer science & Computer Chess Development. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language, and was one of the creators and early developers of the Plan 9 operating system. Since 2006, Thompson has worked at Google, where he co-developed the Go programming language.

Other notable contributions included his work on regular expressions and early computer text editors QED and ed, the definition of the UTF-8 encoding, and his work on computer chess that included the creation of endgame tablebases and the chess machine Belle. He won the Turing Award in 1983 with his long-term colleague Dennis Ritchie.

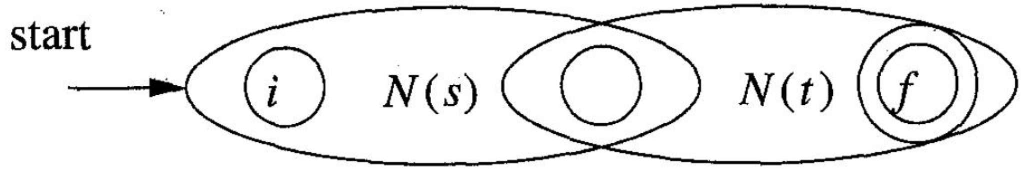
In the 1960s, Thompson also began work on regular expressions. Thompson had developed the CTSS version of the editor QED, which included regular expressions for searching text. QED and Thompson's later editor ed (the standard text editor on Unix) contributed greatly to the eventual popularity of regular expressions, and regular expressions became pervasive in Unix text processing programs. Almost all programs that work with regular expressions today use some variant of Thompson's notation. He also invented Thompson's construction algorithm used for converting regular expressions into nondeterministic finite automata in order to make expression matching faster.[12]

2. RE→NFA

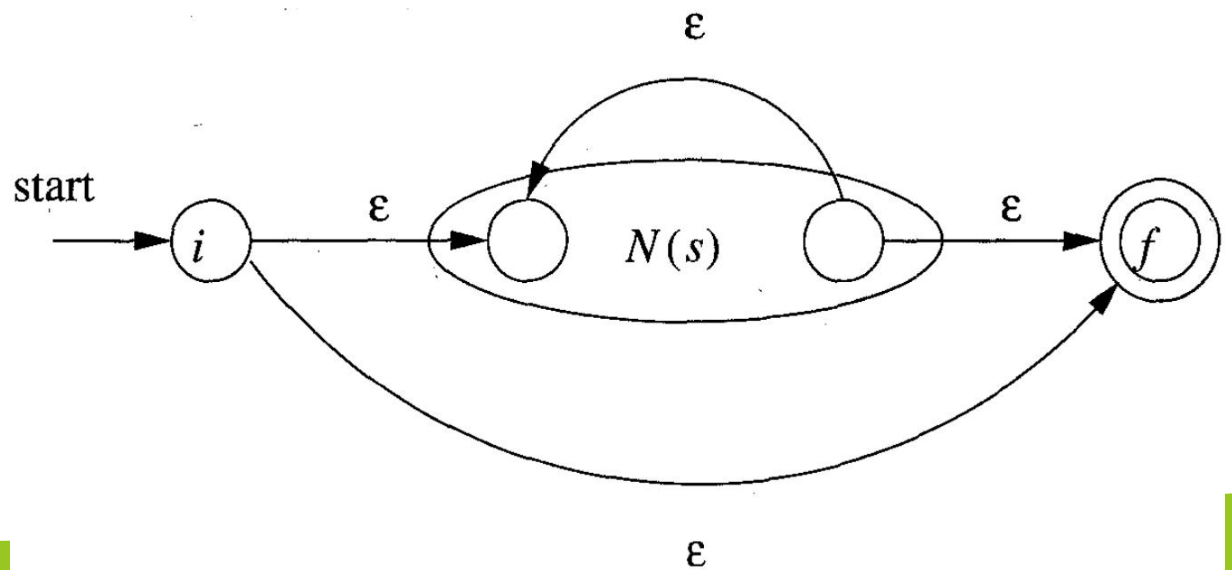
- **Step 2: 处理组合REs（归纳规则）**
  - 对于表达式**r=s|t**
    - 增加2个新状态，4个新的ε转移



  - 对于表达式**r=st**
    - 无需增加新状态或转移

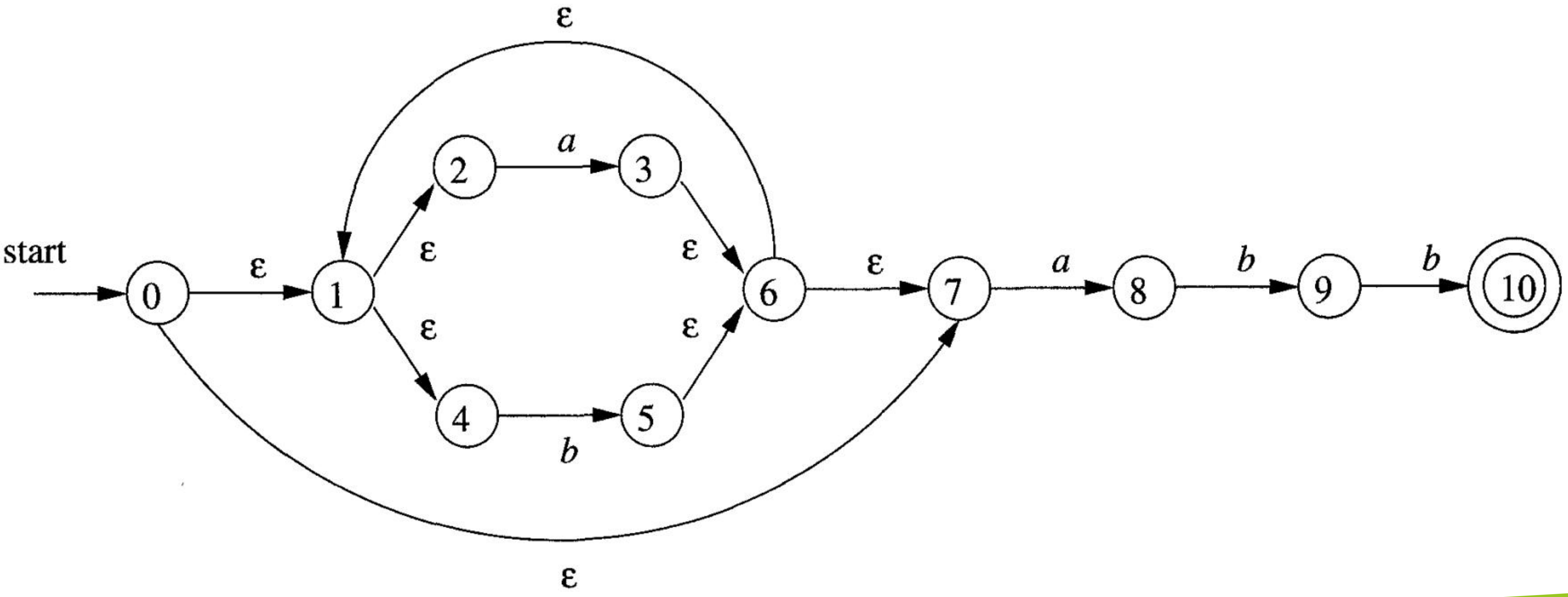


  - 对于表达式**r=s***
    - 增加2个新状态，4个新的ε转移



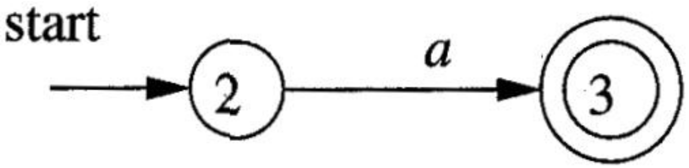
2. RE→NFA

- 例：将正则表达式(a|b)*abb转成NFA
- 最终结果：

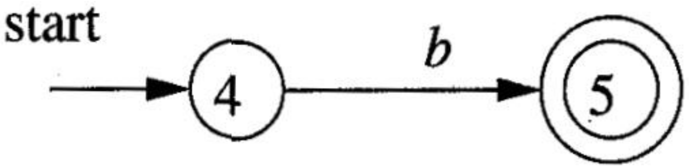


2. RE→NFA

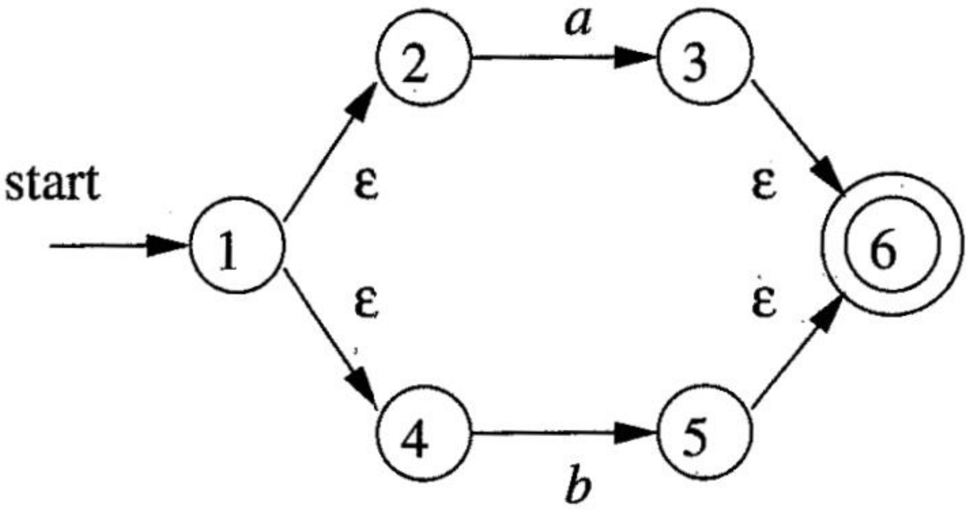
- 例：将正则表达式**(a|b)*abb**转成NFA
  - 对于r1=a：



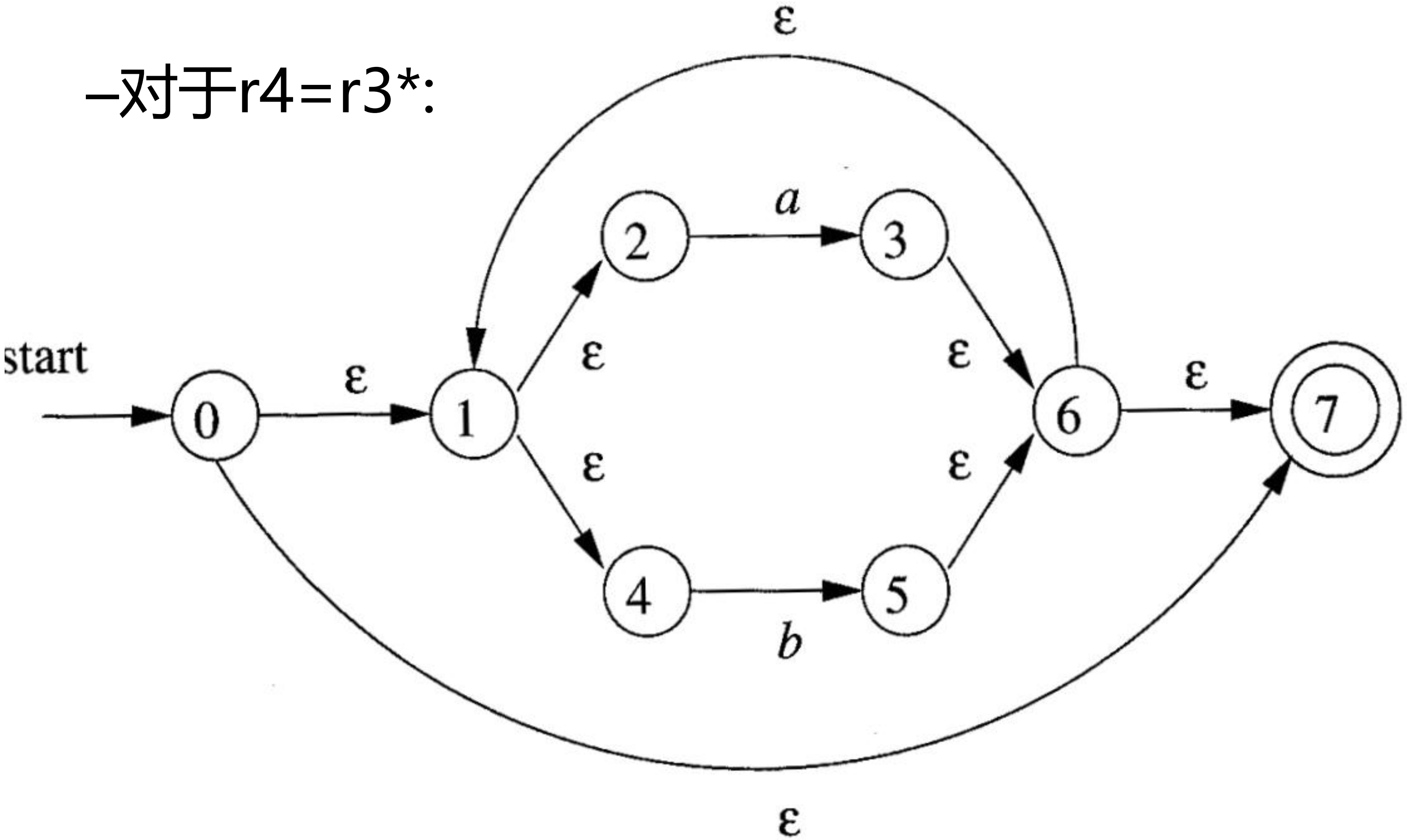
  - 对于r2=b：



  - 对于r3=a|b：

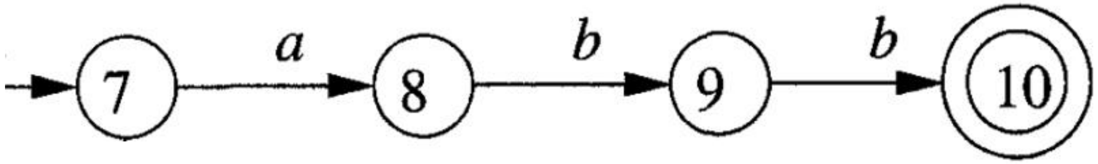


  - 对于r4=r3*:

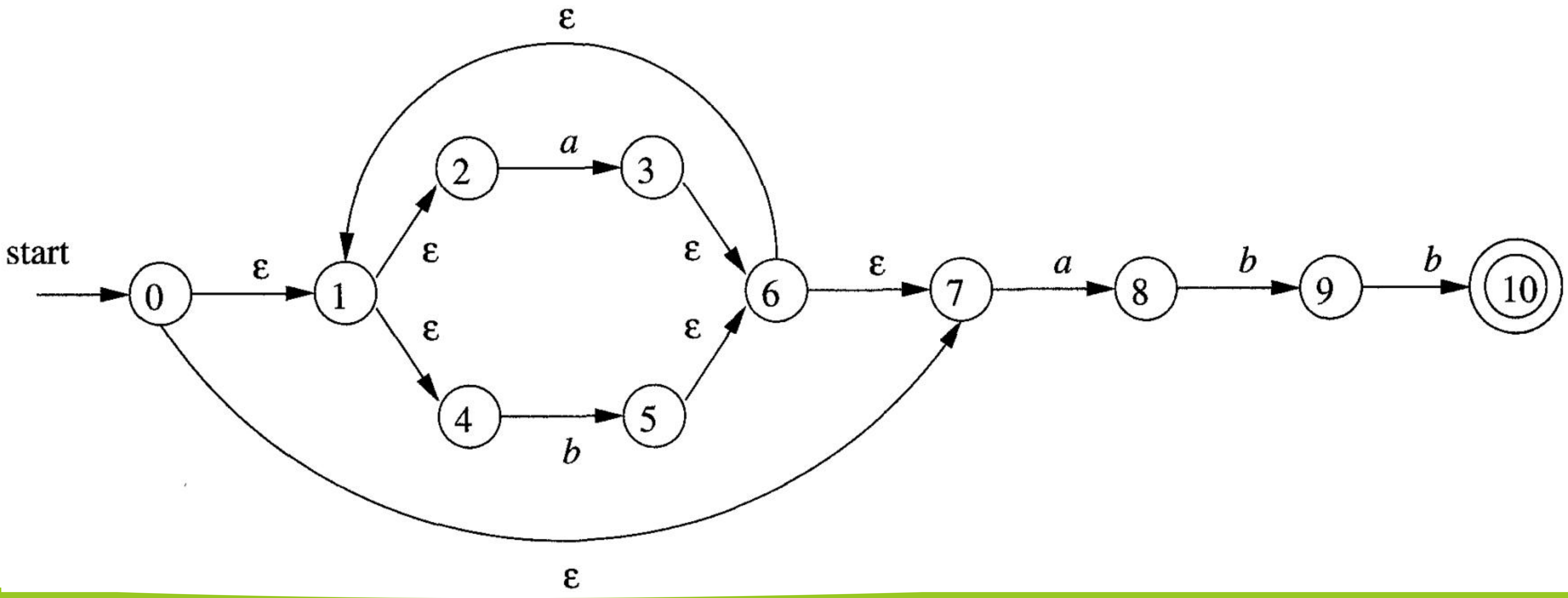


2. RE→NFA

- 例：将正则表达式**(a|b)*abb**转成NFA
  - 对于abb:



  - 最终：

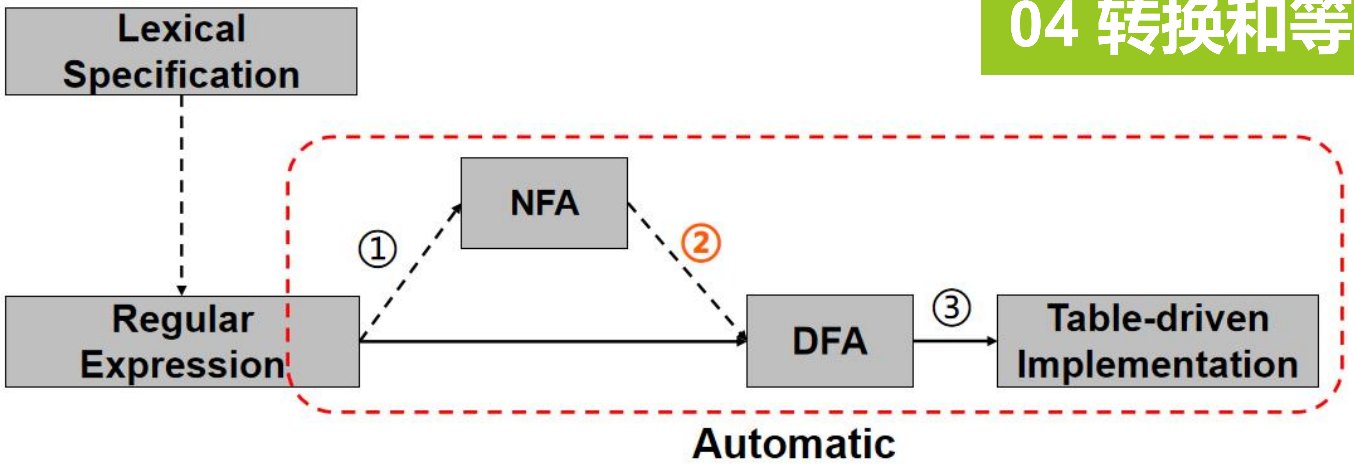


随堂练习(2)

- 将正则表达式**1*(0|01)***转成NFA

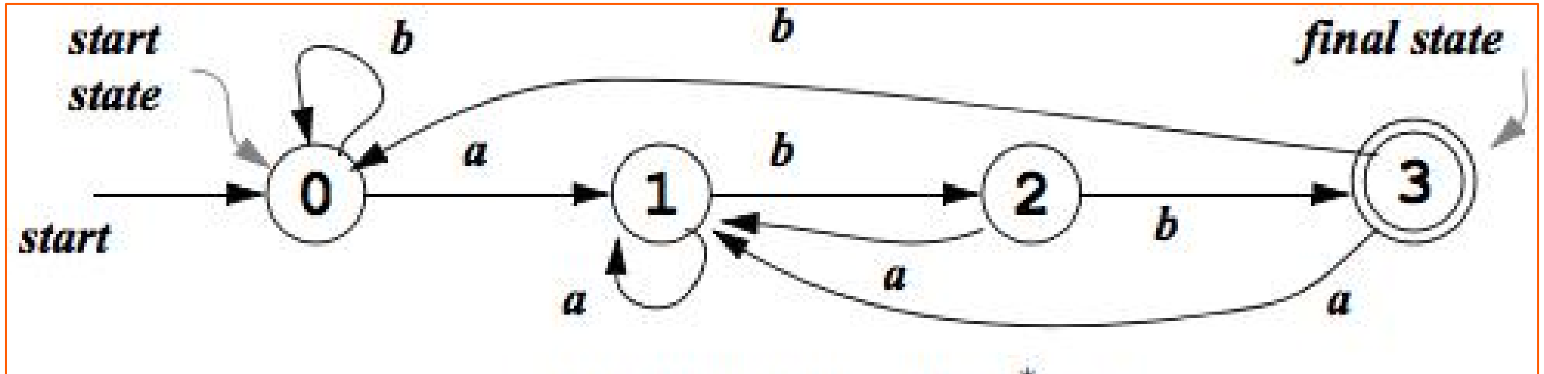
3．NFA→DFA

- 对于每个NFA M，存在一个与其**等价**的DFA M’

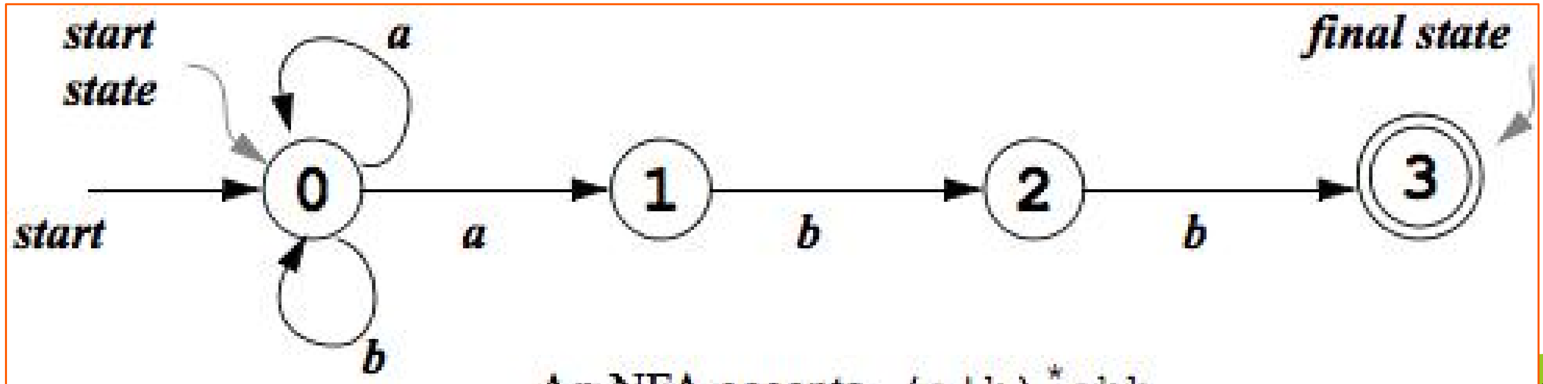


- 但**NFA的转移是不确定的**，更难以用机器模拟

- 故须：NFA→DFA



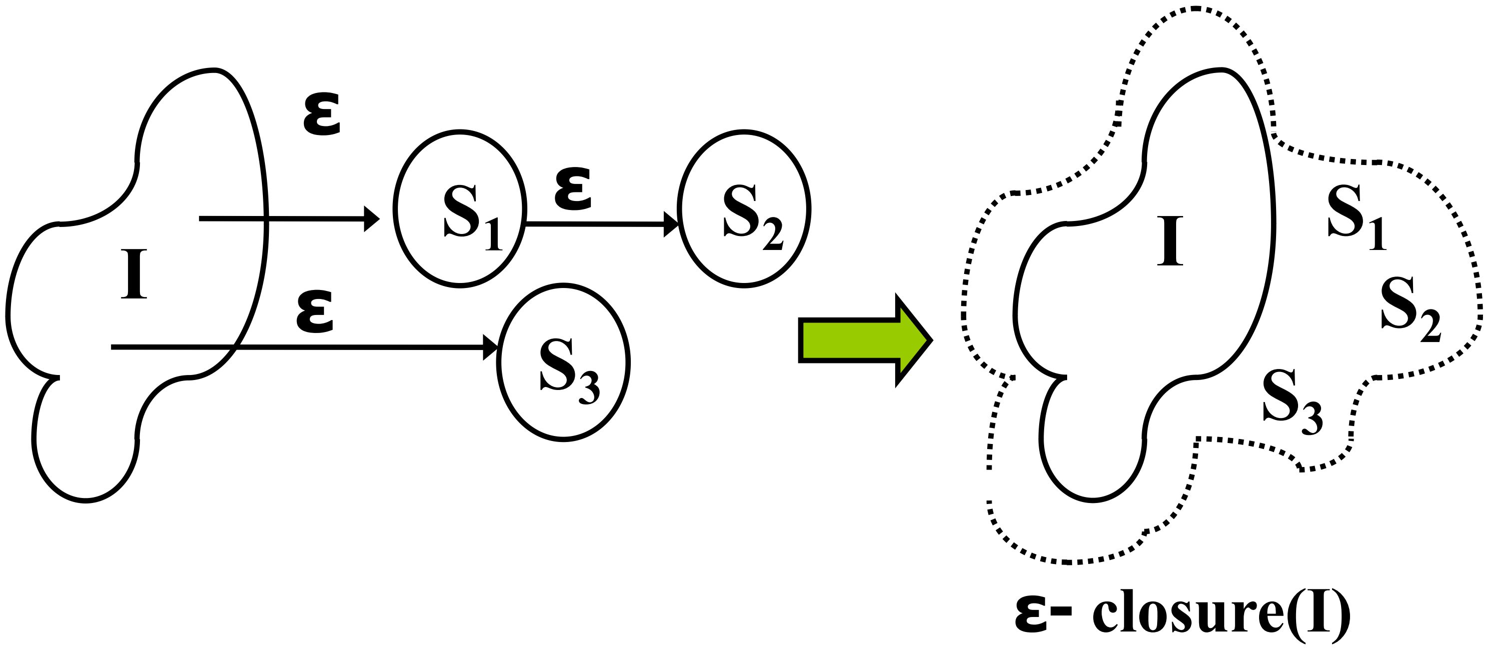
A DFA accepts (a|b)*abb



An NFA accepts (a|b)*abb

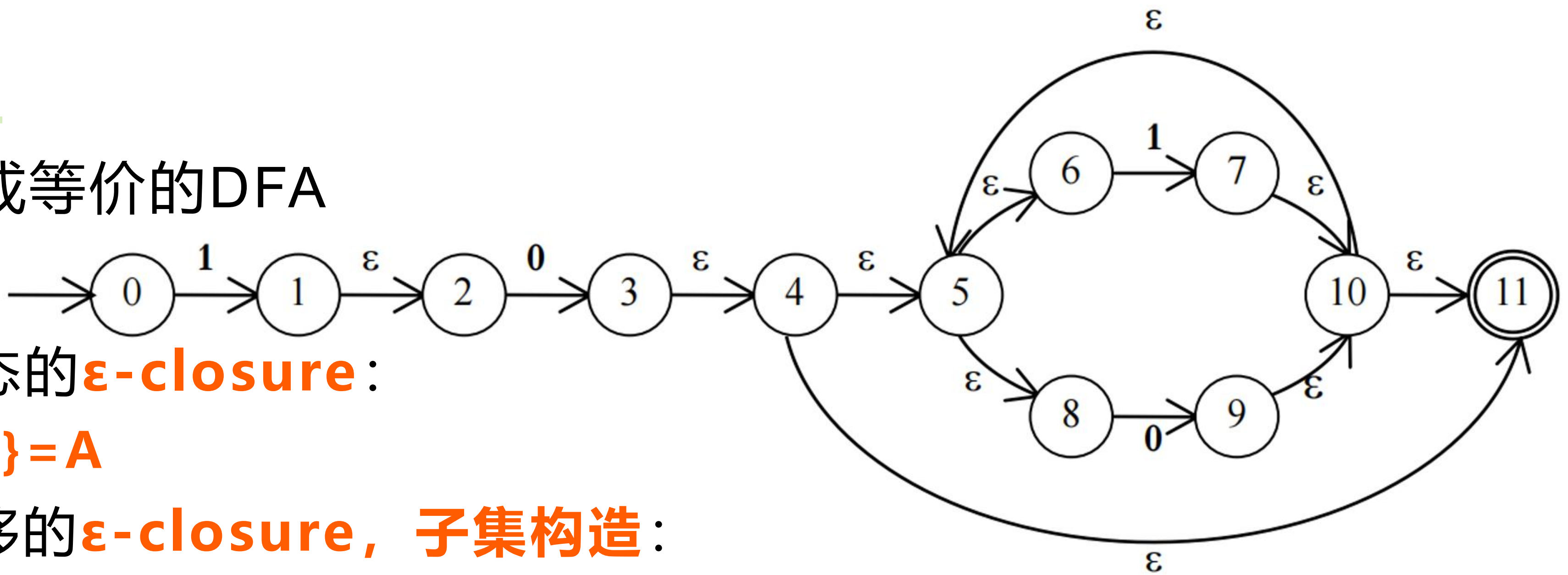
3. NFA→DFA

- **ε-NFA→DFA**
  - **ε闭包构造[ε-closure construction]**算法
    - ✓ 状态集合I的ε闭包**ε-closure(I)**是状态集I中的所有状态S以及经**任意条ε弧**而能到达的状态的集合。



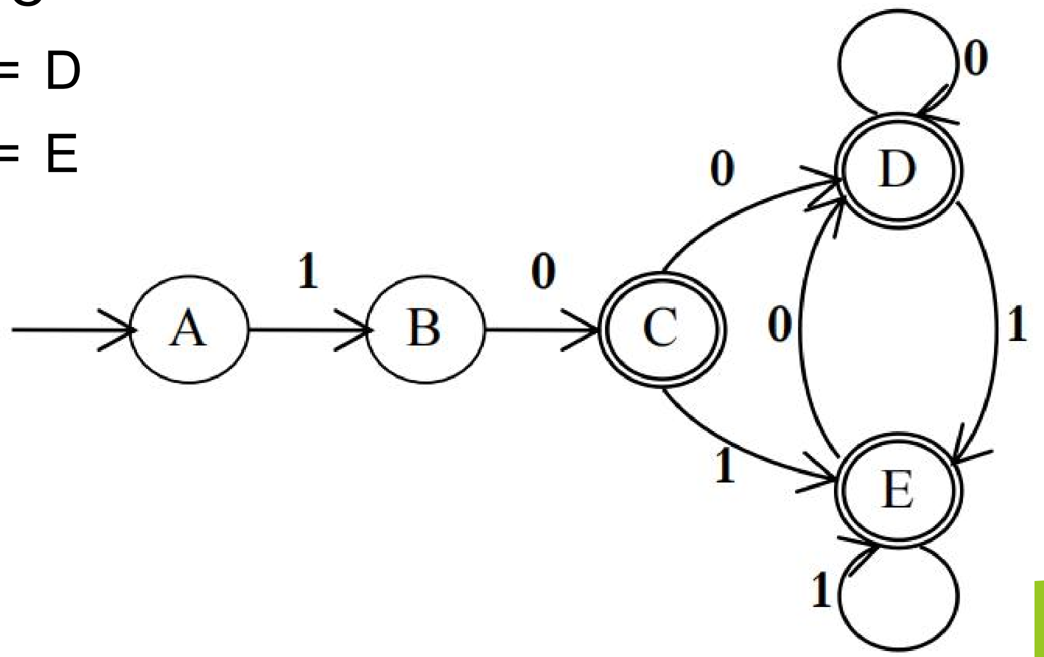
3. NFA→DFA

- 例1：将右图**ε-NFA**转换成等价的DFA

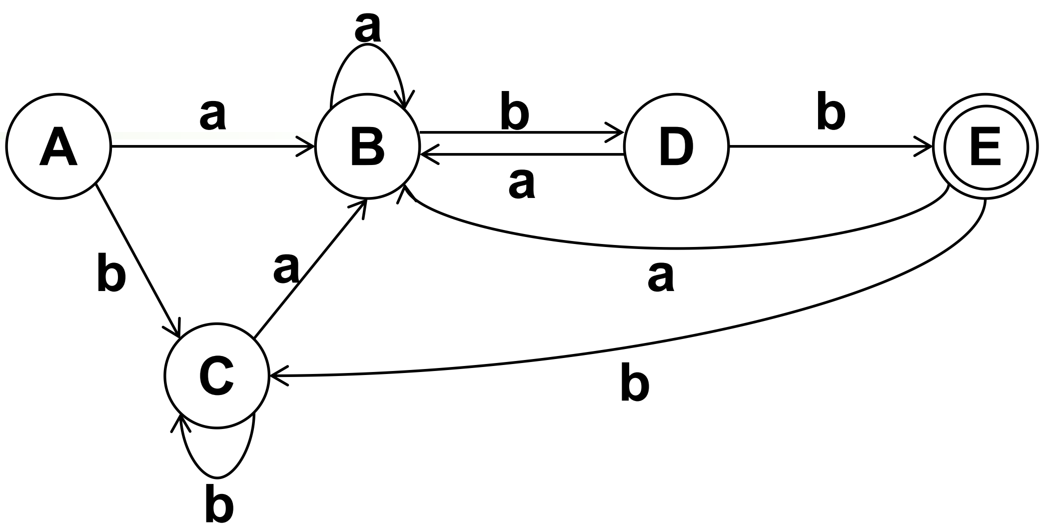


  - **Step 1**：计算初始状态的**ε-closure**：
    - ✓ **ε-closure({0})={0}=A**
  - **Step 2**：计算状态转移的**ε-closure，子集构造**：
    - ✓ (A，1) = ε-closure({1}) = {1，2} = B
    - ✓ (B，0) = ε-closure({3}) = {3，4，5，6，8，11} = C
    - ✓ (C，0) = ε-closure({9}) = {5，6，8，9，10，11} = D
    - ✓ (C，1) = ε-closure({7}) = {5，6，7，8，10，11} = E
    - ✓ (D，0) = ε-closure({9}) = D
    - ✓ (D，1) = ε-closure({7}) = E
    - ✓ (E，0) = ε-closure({9}) = D
    - ✓ (E，1) = ε-closure({7}) = E
  - **Step 3**：根据新的状态集合画出DFA

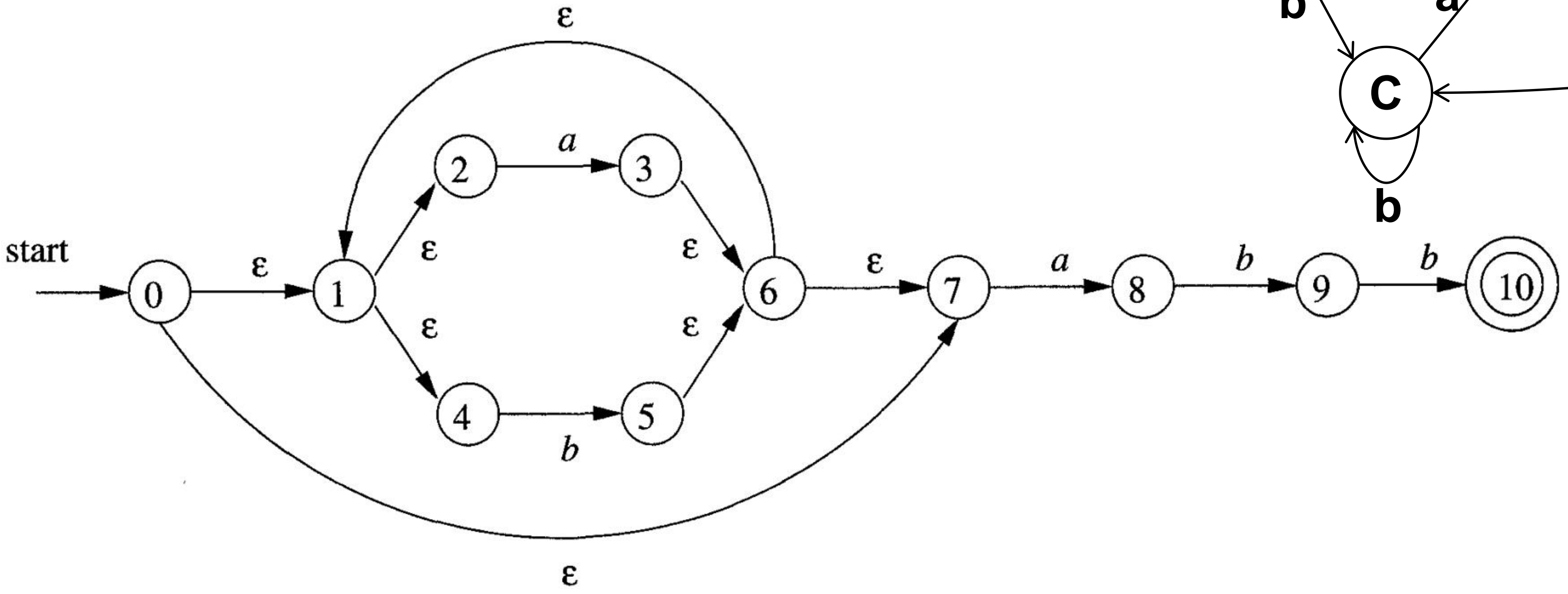
没有新的集合诞生→Step2 结束



随堂练习(3)



- 将下图的ε-NFA转换成等价的DFA



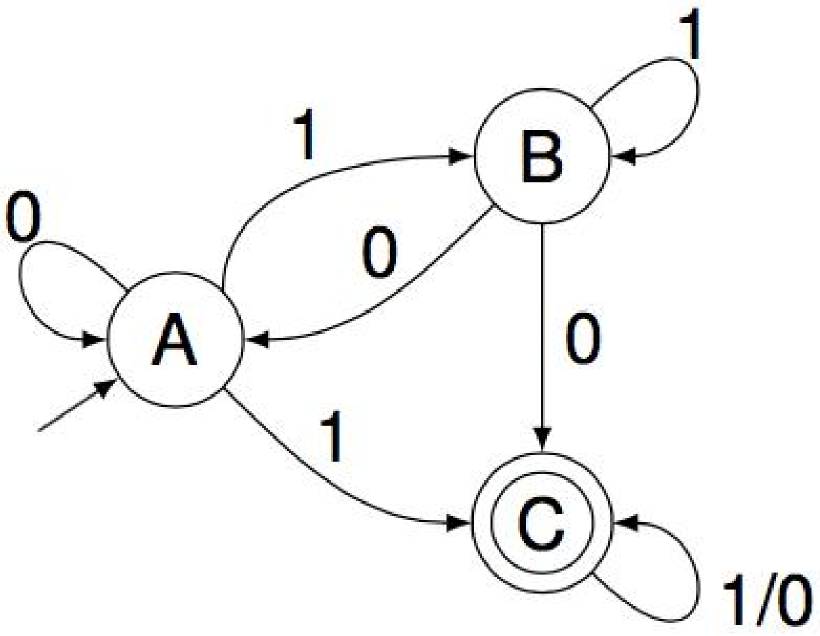
- ε-closure({0})={0,1,2,4,7}=A
- (A,a)=ε-closure({3,8})={1,2,3,4,6,7,8}=B
- (A,b)=ε-closure({5})={1,2,4,5,6,7}=C
- (B,a)=ε-closure({3,8})=B
- (B,b)=ε-closure({5,9})={1,2,4,5,6,7,9}=D
- (C,a)=ε-closure({3,8})=B
- (C,b)=ε-closure({5})=C
- (D,a)=ε-closure({3,8})=B
- (D,b)=ε-closure({5,10})={1,2,4,5,6,7,10}=E
- (E,a)=ε-closure({3,8})=B
- (E,b)=ε-closure({5})=C
- **没有新的集合诞生-->结束**

3. NFA→DFA

- **NFA(不含ε转移)→DFA**
  - **子集构造[subset construction]**算法
    - ✓ 让DFA的每个状态对应NFA的一个状态集合
    - ✓ 即DFA用它的一个状态记录在NFA读入一个输入符号后可能达到的所有状态

3. NFA→DFA

- 例2：将右图NFA**(不含ε转移)**转换成等价的DFA



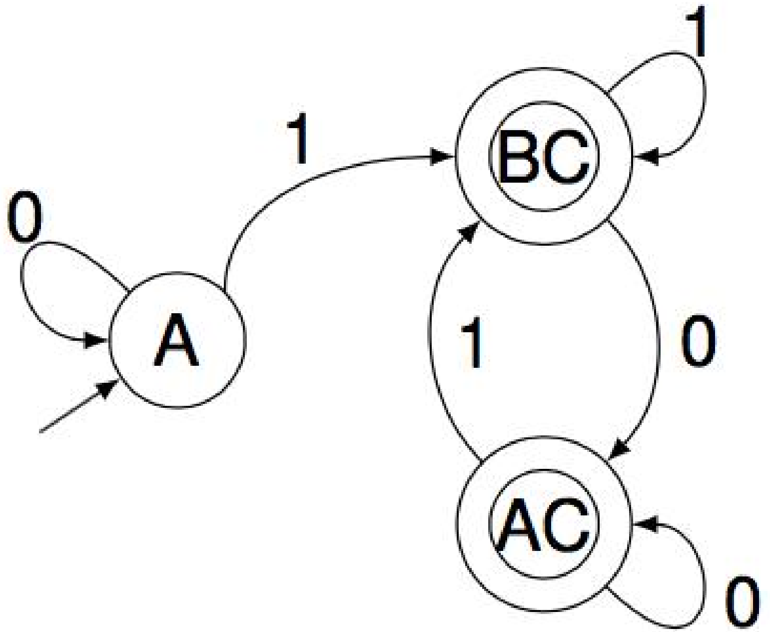
  - 可借助**状态转换矩阵**来进行
  - Step 1：**子集构造**

alphabet

state

	0	1
A	A	BC
BC	AC	BC
AC	AC	BC

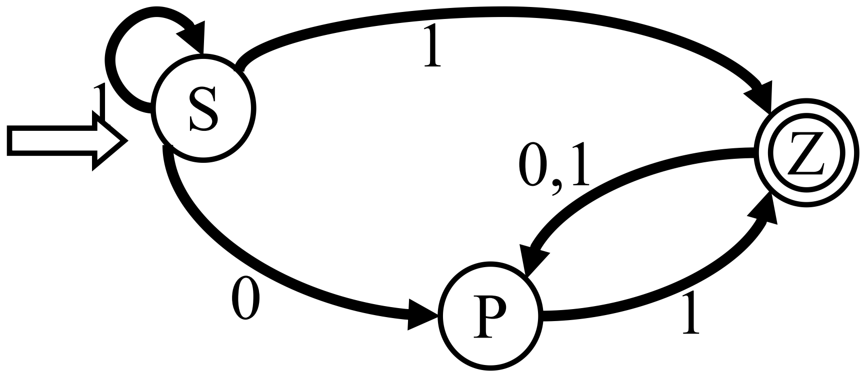
没有新的集合诞生→Step1 结束



  - Step 2：根据新的状态集合画出DFA

随堂练习(4)

• 将下图的NFA(不含ε转移)转换成等价的DFA



令A=SZ，B=SZP，则有：

	0	1
S	P	SZ
P	-	Z
SZ	P	SZP
Z	P	P
SZP	P	SZP

	0	1
S	P	A
P	-	Z
A	P	B
Z	P	P
B	P	B