中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

DCS290

Compilation Principle

编 译 原 理

第三章 词法分析

郑馥丹

zhengfd5@mail.sysu.edu.cn

CONTENTS

目 录

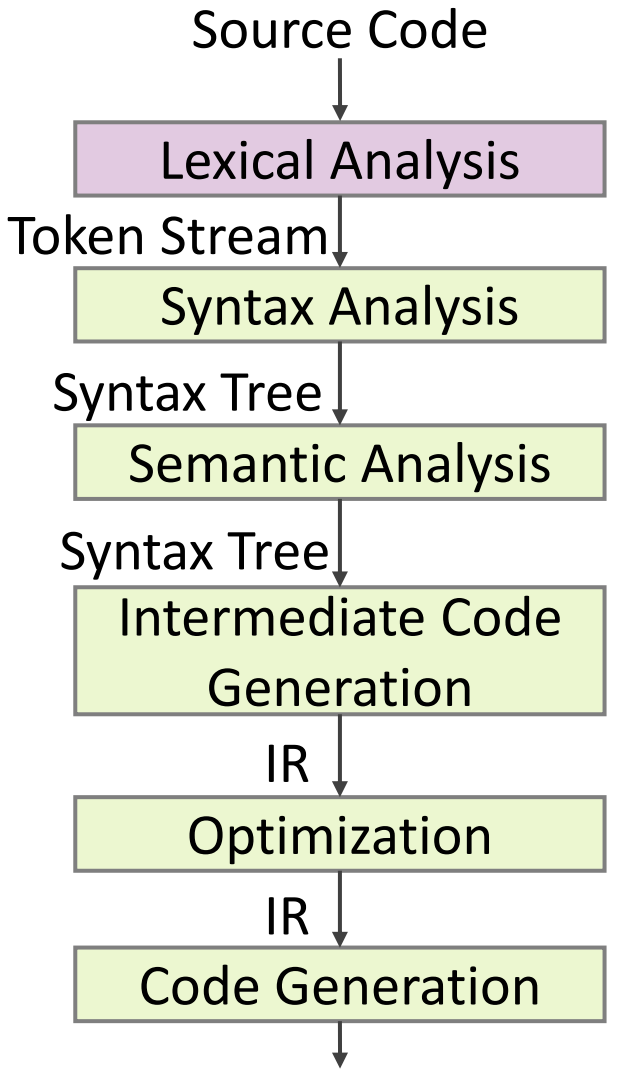
1. 什么是词法分析[Lexical Analysis]?

- 扫描源程序字符流，识别并分解出有词法意义的单词或符号（token）
  - 输入：源程序，输出：token序列
  - token表示：<类别，属性值>
    - ✓ 保留字、标识符、常量、运算符等
  - token是否符合**词法规则**?
    - ✓ 0var, $num

```
void main()
{
    int arr[10], i, x = 1;

    for (i = 0; i < 10; i++)
        arr[i] = x * 5;
}
```

keyword(for)
symbol(()
id(i)
symbol(=)
num(0)
symbol(;)
id(i)
symbol(<)
num(10)
symbol(;)
id(i)
symbol(++)
symbol())

id(arr)
symbol([)
id(i)
symbol(])
symbol(=)
id(x)
symbol(*)
num(5)
symbol(;)



1. 什么是词法分析[Lexical Analysis]?

- 扫描源程序字符流，识别并分解出有词法意义的单词或符号（token）
- 例：
  - 输入：
    - ✓ 字符串“if (i == j)\n\tz = 0; \nelse\n\tz = 1; \n”
  - 目标：将字符串划分为一组**tokens**
  - 步骤：
    - ① 移除注释：/* simple example */
    - ② 识别tokens：‘**if**’ ‘**(**’ ‘**i**’ ‘**==**’ ‘**j**’ ……
    - ③ 识别tokens所属的类别：(**keyword**, ‘**if**’), (**LPAR**, ‘**(**‘), (**id**, ‘**i**’) ……
  - 输出：(**keyword**, ‘**if**’), (**LPAR**, ‘**(**‘), (**id**, ‘**i**’) ……

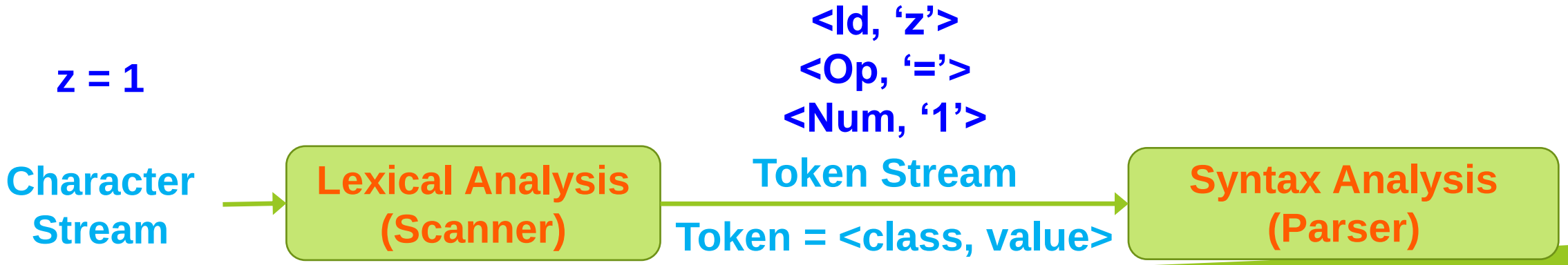
```
/* simple example */
if (i == j)
  z = 0;
else
  z = 1;
```

2. 什么是词[token]?

- Token：词，**最小意义单元**
  - 英语中的token类别：
    - ✓ 名词noun，动词verb，形容词adjective，...
  - 编程语言中的token类别：
    - ✓ 数字number，关键词keyword，空白whitespace，标识符identifier，...
- 每个token类别对应一个字符串集合[a set of strings]
  - number：非空的数字字符串
  - keyword：一组固定的**保留字**(“for”，“if”，“else”，...)
  - whitespace：由空格blank、制表符tab、换行符newline组成的非空序列
  - identifier：用户自定义的要标识的实体名称

3. 词法分析：分词[Tokenization/Scanner]

- 词法分析器也称**标记化Tokenization**或**扫描器Scanner**
  - 将输入字符串划分为token序列
  - 对token进行分类——token类别
    - ✓ **词素[lexeme]**：token类的实例，例如：‘z’, ‘=’, ‘1’
    - ✓ 事先定义好token类别：例如keyword, identifier, whitespace, integer等
- 得到的token会被送入**语法分析器[Syntax Analyzer]**（也称**Parser**）
  - parser依赖token类来识别角色（例如，关键字与标识符的处理方式不同）。



3. 词法分析：设计[Design]

- 定义token类别
  - 描述所有感兴趣的项
  - 依赖于语言、parser的设计
    - ✓ *“if (i == j)\n\tz = 0; \nelse\n\tz = 1; \n”*

      Keyword, identifier, whitespace, integer

5种最常见的编程语言token类别：

- **Identifiers: getBalance, weight,....**
- **Reserved words: if, else, while,...**
- **Constants: 10, 3.14, “abc”, ‘a’,...**
- **Operators: +, -, *, /, <<,...**
- **Punctuation: (,), ;, :,...**

- 识别哪些字符串属于哪个token类别

```
if (i == j)
  z = 0;
else
  z = 1;
```

‘==’ or ‘=’? ➡ 最长匹配原则

keyword or identifier? ➡ 查询keyword表

3. 词法分析：实现[Implementation]

- **实现时需完成2件事情：**
  - **识别字符串的token分类**
  - **返回token的值或词素**
- 一个token是一个二元组**(class, lexeme)**
- 丢弃无意义词
  - 例如：whitespace，comments等
- 如果token类别是无二义性的，则可以在对输入字符串进行一次从左到右的扫描中识别标记
- 但token类别可能有歧义[ambiguous]

3. 词法分析：实现[Implementation]

- 歧义举例
  - C++ 模板语法：Foo<Bar>
  - C++ 流语法：cin >> var
  - 二义性
    - ✓ Foo<Bar<Bar>>
    - ✓ cin >> var

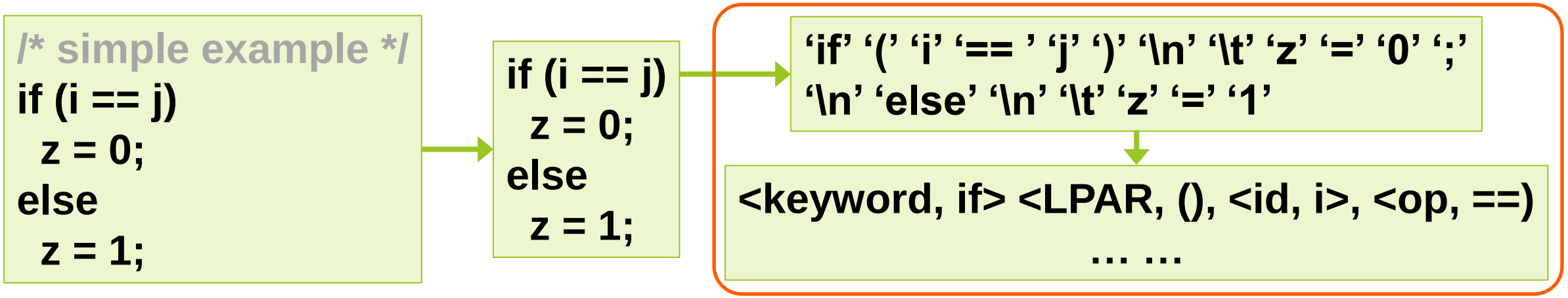
疑问: '>>' 应当成是流操作符，还是两个连续的右尖括号呢?

```
Template <typename T>
T getMax(T x, T y) {
    return (x > y) ? x : y;
}

int main (int argc, char* argv[]) {
    getMax<int>(3, 7);
    getMax<double>(3.0, 2.0);
    getMax<char>('g', 'e');

    return 0;
}
```

- "**向前看[look ahead]**" 可以展望消除歧义
  - 要查看更大的上下文或结构　cin >> var
  - 有时词法分析需要parser的反馈：“解析模板嵌套，这应该是两个独立的> ”
- 如果token没有歧义
  - tokenizing可在无parser反馈的情况下一次完成，可明确区分词法/语法分析

3. 词法分析：总结

- 词法分析
  - 将输入符号串分解成token
  - 识别每个token的类别
- 从左到右扫描
  - 有时候需要“向前看[look ahead]”
  - 应该尽量减少“向前看[look ahead]”的数量（过多会增加复杂性）

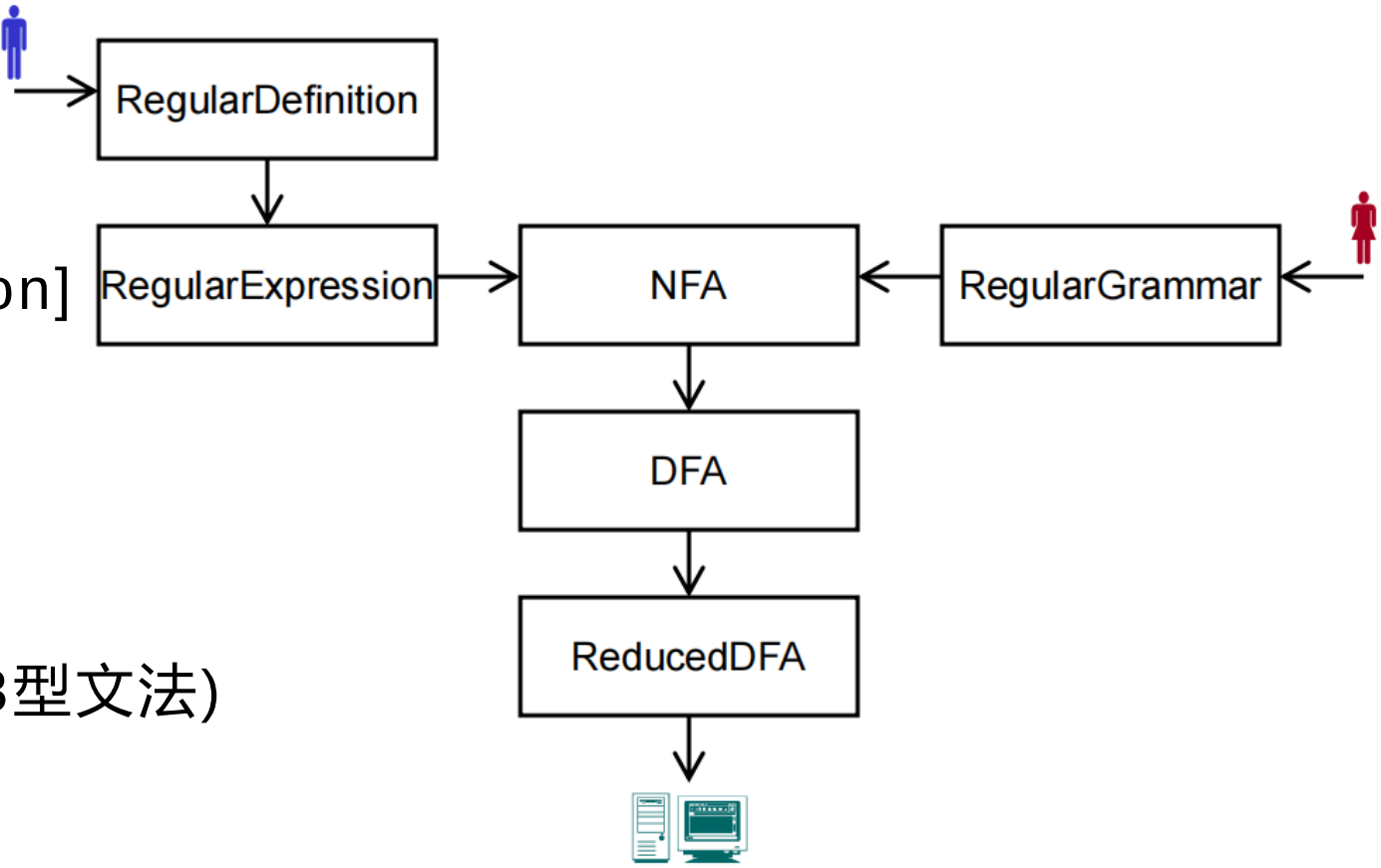


CONTENTS

目 录

1. 词法规范

- 三种词法规范
  - 表达式[Expressions]
    - ✓ 正则表达式[Regular expression]
    - ✓ 正则定义[Regular definition]
  - 语法
    - ✓ 正规文法[Regular grammar](3型文法)
  - 有穷自动机
    - ✓ 确定的有穷自动机[Deterministic Finite Automata (DFA)]
    - ✓ 不确定的有穷自动机[Nondeterministic Finite Automata (NFA)]



2. 正则表达式[Regular expressions, REs]

- 正则表达式
  - 也称：**正规式**
  - 可用于定义token class
  - **简单但功能强大**（能够表达模式）
  - 可以从规范**自动生成**Tokenizer实现（使用翻译工具）
  - 最终实现是**有效的**

2. 正则表达式[Regular expressions, REs]

- 正规集：正规式所能描述的语言集合

① ε和Φ是∑上的正规式，正规集L(ε)={ε}，L(Φ)=Φ；

② 对于任何a∈∑，a是∑上的正规式，它所表示的正规集为L(a)={a}；

③ 假定r和s都是∑上的正规式，他们所表示的正规集分别为L(r)和L(s)，那么，以下也都是正规式和他们所表示的正规集；

正规式	正规集
(r)	L(r)= L ((r))= L (r)
r | s	L(r | s)= L (r)∪ L (s)
r•s	L(r•s)= L (r) L (s)
r^*	L(r^*)=(L (r))*

算符的优先顺序：‘( )’ > ‘*’ > ‘•’ > ‘|’

其中：‘•’和‘|’都是左结合

$ab^*c|d$　　**$a(b^*)c|d$**
$(a(b^*))c|d$
$((a(b^*))c)|d$

④ 仅由有限次使用上述三步定义的表达式才是∑上的正规式，仅由这些正规式所表示的字集才是∑上的正规集。

2. 正则表达式[Regular expressions, REs]

- 例子
  - 令Σ={a，b}， Σ上的正规式和相应的正规集有：

正规式	正规集
a	**{a}**
a|b	**{a,b}**
ab	**{ab}**
(a|b)(a|b)	**{aa,ab,ba,bb}**
a*	**{ε ,a,aa,…任意个a串}**
(a|b)*	**{ε ,a,b,aa,ab ……所有由a和b组成的串}**

2. 正则表达式[Regular expressions, REs]

- 正则表达式的代数性质
  - r | s = s | r　　“|”满足交换律
  - r | (s | t) = (r | s) | t　　“|”的结合律
  - (r s) t = r (s t)　　“.(连接)”的结合律
  - r(s | t) = r s | r t
  - (r | s) t = r t | s t　　分配律
  - εr = rε= r　　ε是“.”的恒等元素
  - r|r=r　　“|”的抽取律　　r*=ε|r|rr|…

2. 正则表达式[Regular expressions, REs]

- 常用表达
  - At least one: $A^+ \equiv AA^*$
  - Option: $A? \equiv A \mid \varepsilon$
  - Characters: $[a_1a_2...a_n] \equiv a_1|a_2|...|a_n$
  - Range: ‘a’ + ‘b’ + … + ‘z’ ≡ [a-z]
  - Excluded range: complement of [a-z] ≡ [^a-z]

2. 正则表达式[Regular expressions, REs]

- 例子

Regular Expression	Explanation
a^*	0 or more a's (ε, a, aa, aaa, aaaa, …)
a^+	1 or more a's (a, aa, aaa, aaaa, …)
(a|b)(a|b)	(aa, ab, ba, bb)
$(a|b)^*$	all strings of a's and b's (including ε)
$(aa|ab|ba|bb)^*$	all strings of a's and b's of **even偶数** length
[a-zA-Z]	**shorthand简写** for "a|b|…z|A|B|…|Z"
[0-9]	shorthand for "0|1|2|…|9"
$0([0-9])^*0$	numbers that start and end with 0
$1^*(0|ε)1^*$	**binary strings that contain at most one zero**
$(0|1)^*00(0|1)^*$	all binary strings that contain '00' as substring

2. 正则表达式[Regular expressions, REs]

- 思考：$(a|b)^*$ 和 $(a^*b^*)^*$ 是否等价？ **等价**

$(a|b)^* = ?$

$(L(a|b))^* = (L(a) \cup L(b))^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$

$= \{a, b\}^0 + \{a, b\}^1 + \{a, b\}^2 + \ldots$

$= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$ **（所有由 a 和 b 组成的字符串）**

$(a^*b^*)^* = ?$

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

$= (\{\varepsilon, a, aa, \ldots\}\{\varepsilon, b, bb, \ldots\})^*$

$= L(\{\varepsilon, a, b, aa, ab, bb, \ldots\})^*$

$= \varepsilon + \{\varepsilon, a, b, aa, ab, bb, \ldots\} + \{\varepsilon, a, b, aa, ab, bb, \ldots\}^2 + \{\varepsilon, a, b, aa, ab, bb, \ldots\}^3 + \ldots$

$= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$ **（所有由 a 和 b 组成的字符串）**

2. 正则表达式[Regular expressions, REs]

- 更多例子
  - Keywords: ‘if’ 或 ‘else’ 或 ‘then’ 或 ‘for’ …
    - ✓ RE = ‘i’‘f’ + ‘e’‘l’‘s’‘e’ + … = ‘if’ + ‘else’ + ‘then’ + …
  - Numbers: 非空数字字符串
    - ✓ digit = ‘0’ + ‘1’ + ‘2’ + ‘3’ + ‘4’ + ‘5’ + ‘6’ + ‘7’ + ‘8’ + ‘9’
    - ✓ 若定义integer = digit digit*，请问‘000’ 是否为integer?
  - Identifier: 以字母开头的字母或数字的字符串
    - ✓ letter = ‘a’ + ‘b’ + … ‘z’ + ‘A’ + ‘B’ + … + ‘Z’ = [a-zA-Z]
    - ✓ 若定义RE = letter(letter + digit)*，请问RE符合C语言中标识符的定义吗?
  - Whitespace: 由空格、换行符、制表符组成的非空序列
    - ✓ (‘ ’ + ‘\n’ + ‘\t’)$^{+}$

2. 正则表达式[Regular expressions, REs]

- 编程语言中的REs https://docs.python.org/3/howto/regex.html

Symbol	Meaning		
\d	Any decimal digit, i.e. [0-9]		
\D	Any non-digit char, i.e., [^0-9]		
\s	Any whitespace char, i.e., [\t\n\r\f\v]		
\S	Any non-whitespace char, i.e., [^ \t\n\r\f\v]		
\w	Any alphanumeric char, i.e., [a-zA-Z0-9_]		
\W	Any non-alphanumeric char, i.e., [^a-zA-Z0-9_]		
.	Any char	\.	Matching “.”
[a-f]	Char range	[^a-f]	Exclude range
^	Matching string start	$	Matching string end
(…)	Capture matches		

2. 正则表达式[Regular expressions, REs]

- 正则表达式vs.正规文法[Regular grammar]

	正则表达式	正规文法
标识符集合	l (l | d) * 其中: l为a-z的字母，d为0-9的数字	• <标识符>→l｜l <字母数字> • <字母数字>→l｜d｜l <字母数字>｜ d <字母数字>
无符号整数	dd *	<无符号整数>→d｜d <无符号整数>

正规式比正规文法更容易让人理解单词是按怎样的规律构成的，且可以从某个正规式**自动地**构造识别程序。

随堂练习(1)

- 写出C语言中标识符的正则表达式

  (_|letter)(_|letter|digit)*

- 写出C语言中十进制数字的正则表达式

  0|([1-9][0-9]*)

- 写出2的倍数的二进制表示的正则表达式

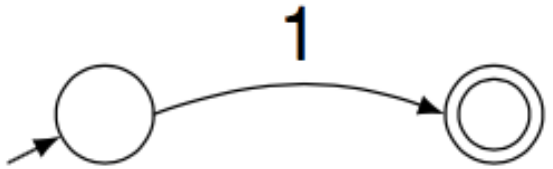
  (0|1)*0

CONTENTS

目 录

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

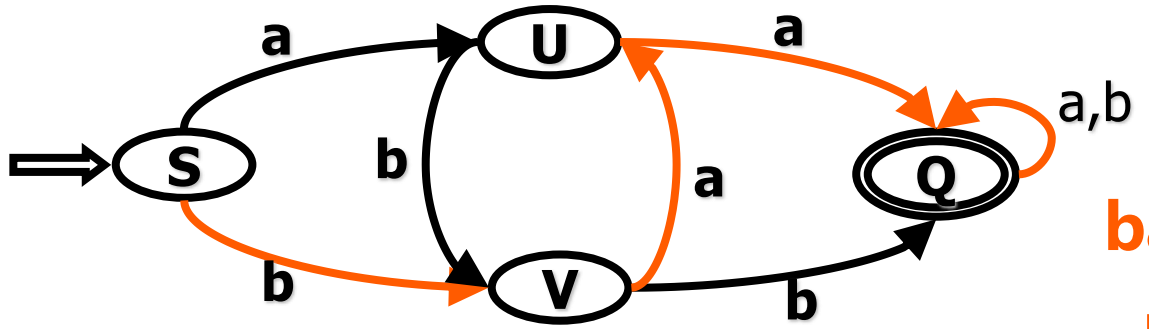
2. 有穷自动机[Finite Automata, FA]

- 正则表达式是定义[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

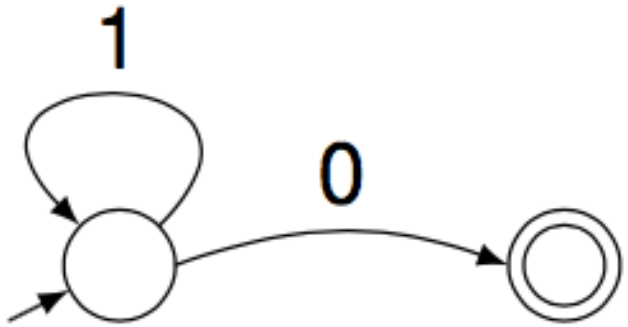
- **FA所能表达的语言即为该FA接受的字符串集合**
  - L(FA) ≡ L(RE)

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’** **L(RE)=1*0**

2. 有穷自动机[Finite Automata, FA]

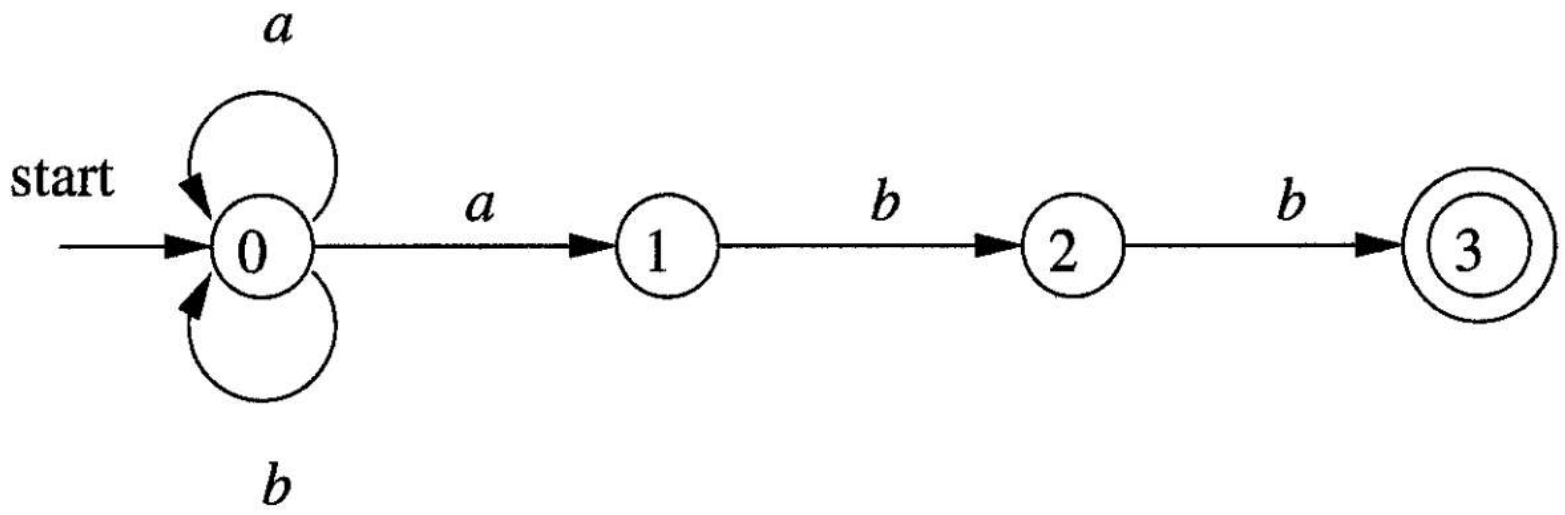
- 例：有右图FA

（1）在字母表∑ = {a, b}上，这个FA的语言是什么?

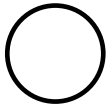
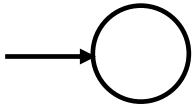
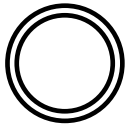
✓ **所有由若干个a和b的组成的串后跟 ‘ abb ’ 的字符串**

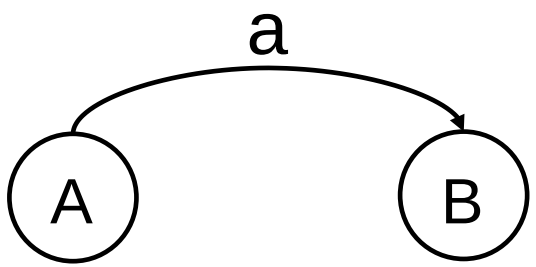
（2）尝试用正则表达式表达这个语言

✓ **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
  - $\mathbf{\delta}$：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
  - $\mathbf{s_0 \in S}$：开始状态/初始状态 
  - $\mathbf{F \subseteq S}$：接受状态/终止状态的**集合** 



δ: A —a→ B

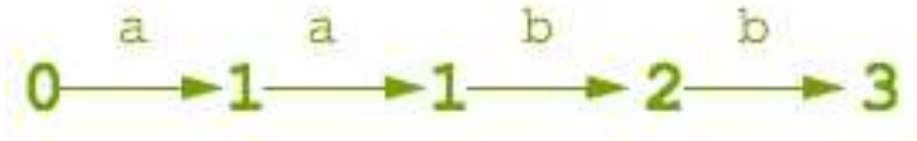
3. DFA和NFA

- **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，对同一个输入可以有**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

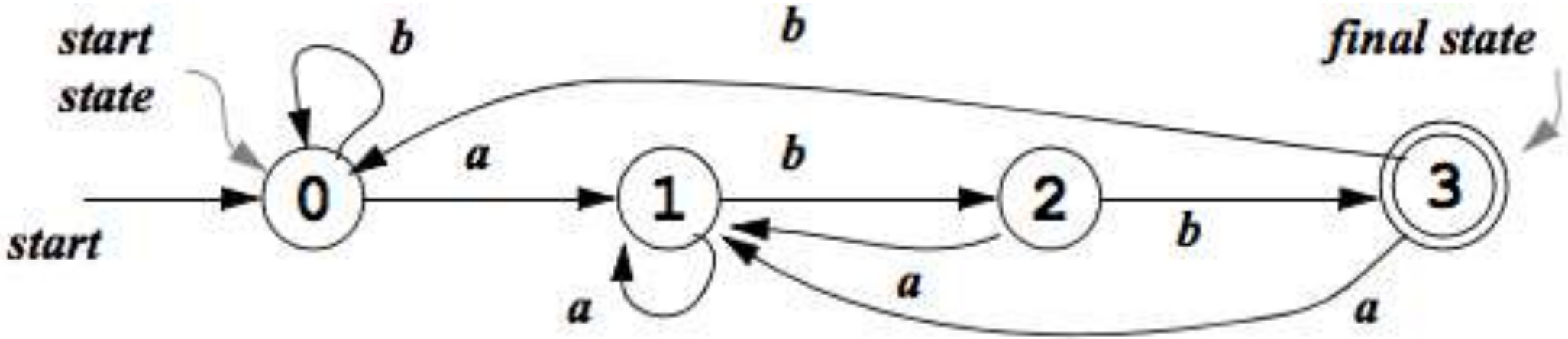
对DFA： $\delta: S \times \Sigma \to S$
对NFA： $\delta: S \times \Sigma \to 2^S$
对ε-NFA： $\delta: S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$

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：

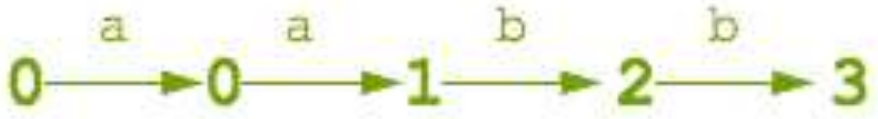


接受



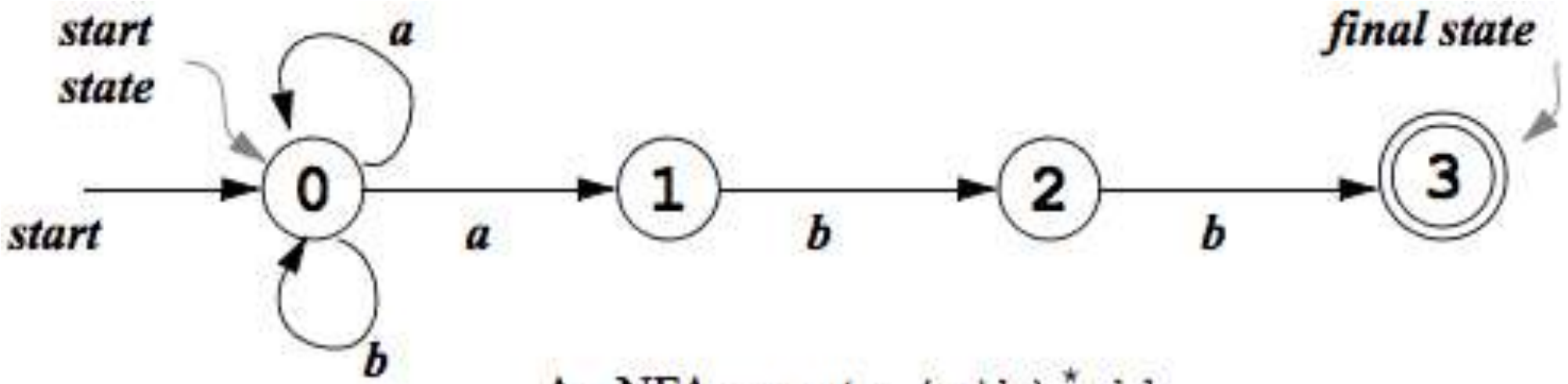
A DFA accepts $(a|b)^*abb$

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： 

  - 不成功序列： 

接受



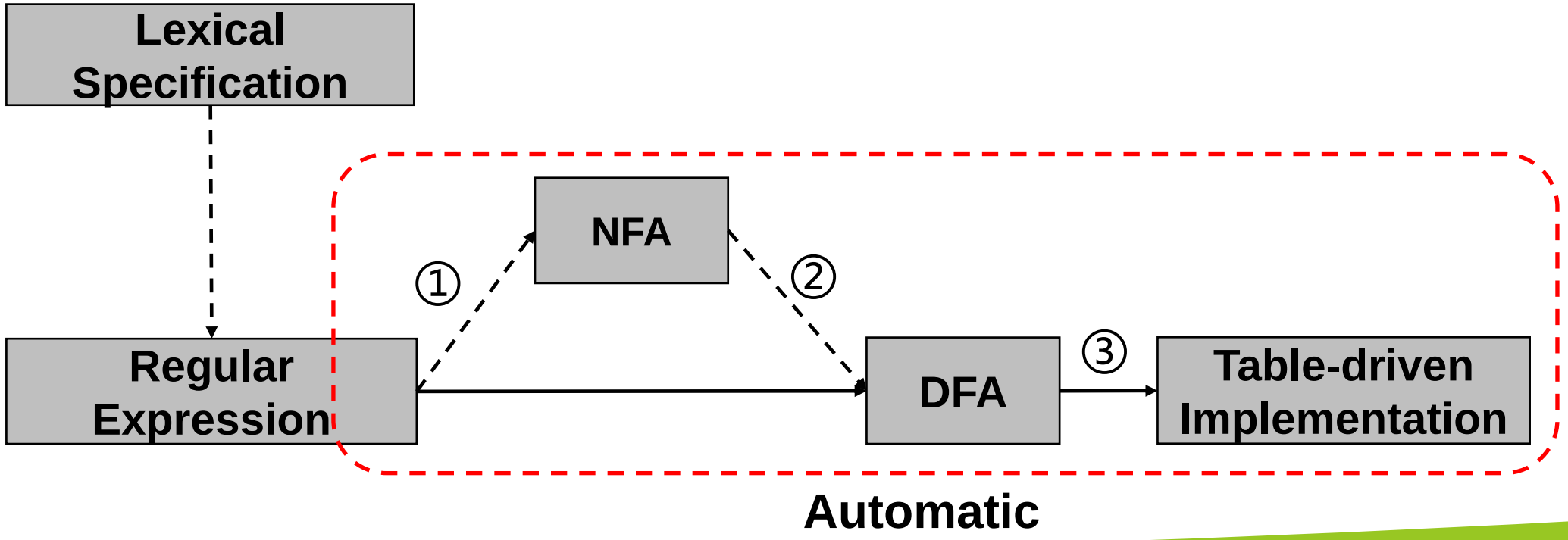
An NFA accepts $(a|b)^*abb$

CONTENTS

目 录

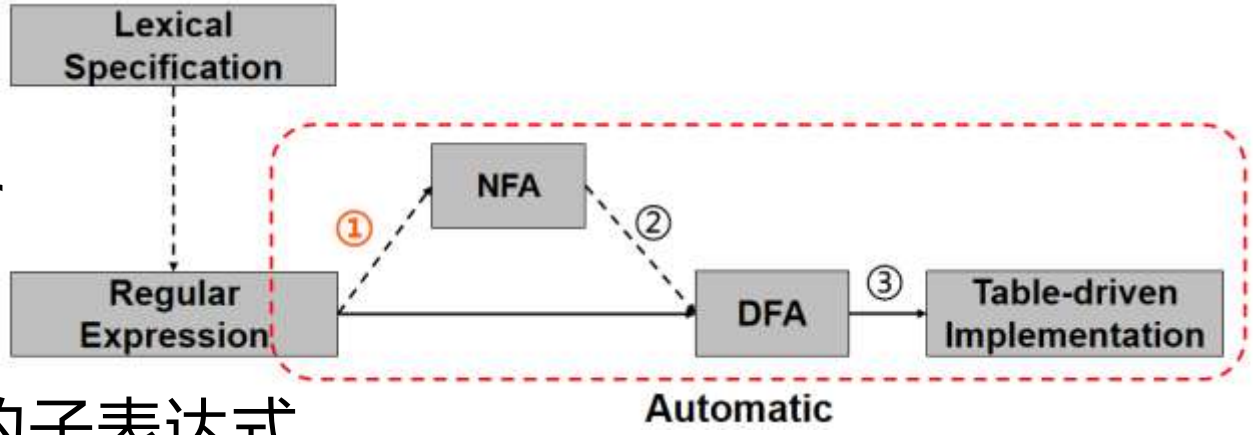
1. 转换流程[Conversion Flow]

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

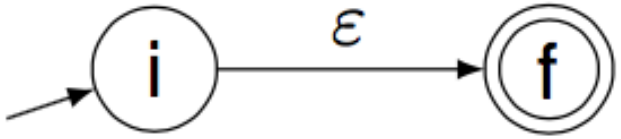


2. RE→NFA

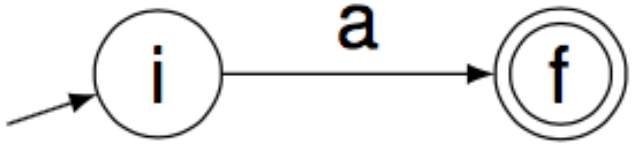
- **RE→NFA采用Thompson构造算法**
  - 输入：字母表Σ上的一个正则表达式r
  - 输出：一个接受L(r)的NFA N
  - 方法：对r进行分析，分解出组成它的子表达式



- **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

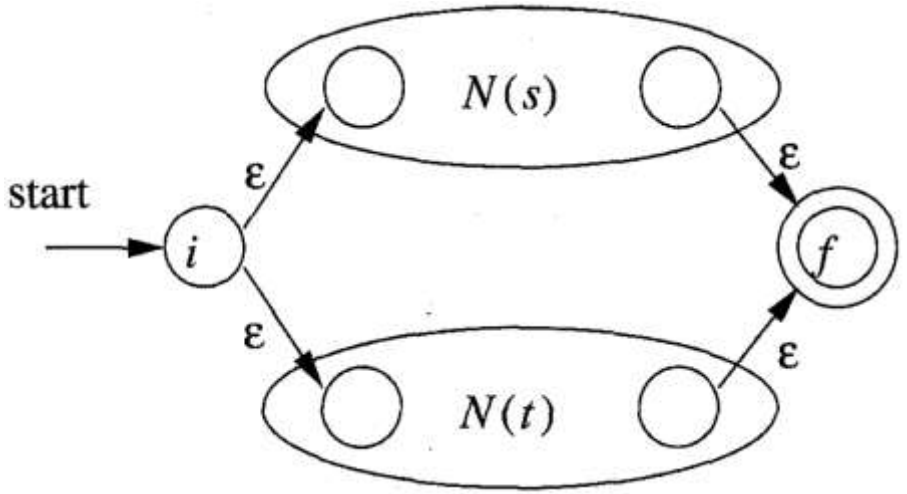
Kenneth Lane Thompson (born February 4, 1943) is an American pioneer of computer science & Computer Chess Development. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor 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.

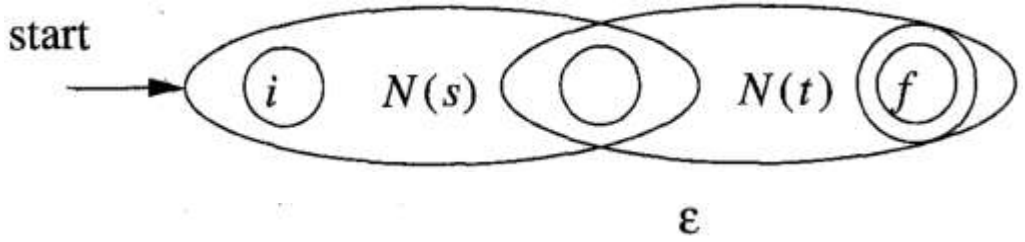
In the 1960s, Thompson also began work on regular expressions. Thompson had developed the CTSS version of the editor QED, which included regular expressions for searching text. QED and Thompson's later editor ed (the standard text editor on Unix) contributed greatly to the eventual popularity of regular 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

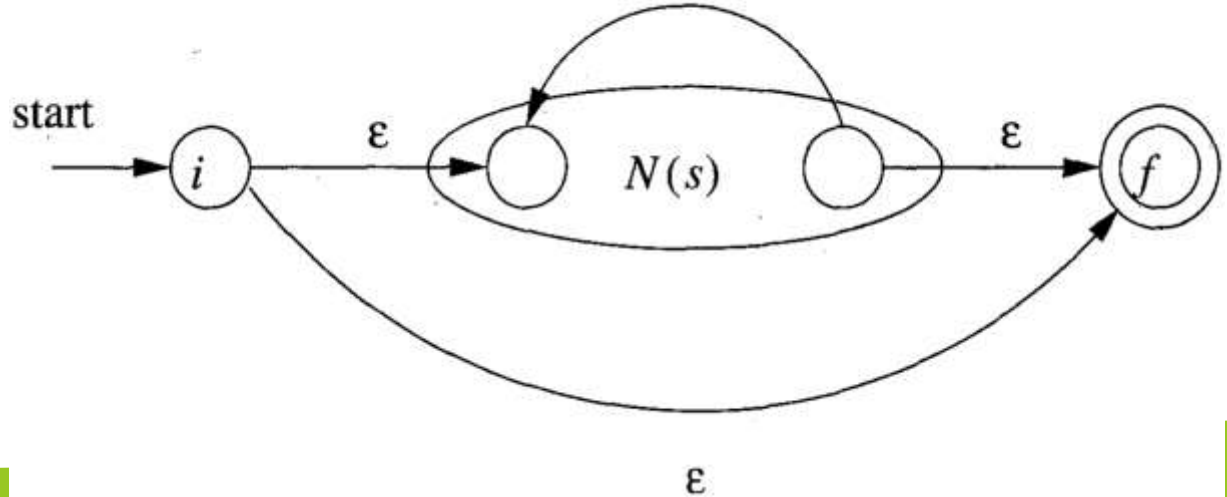
- **Step 2: 处理组合REs（归纳规则）**
  - 对于表达式**r=s|t**
    - 增加2个新状态，4个新的ε转移



  - 对于表达式**r=st**
    - 无需增加新状态或转移

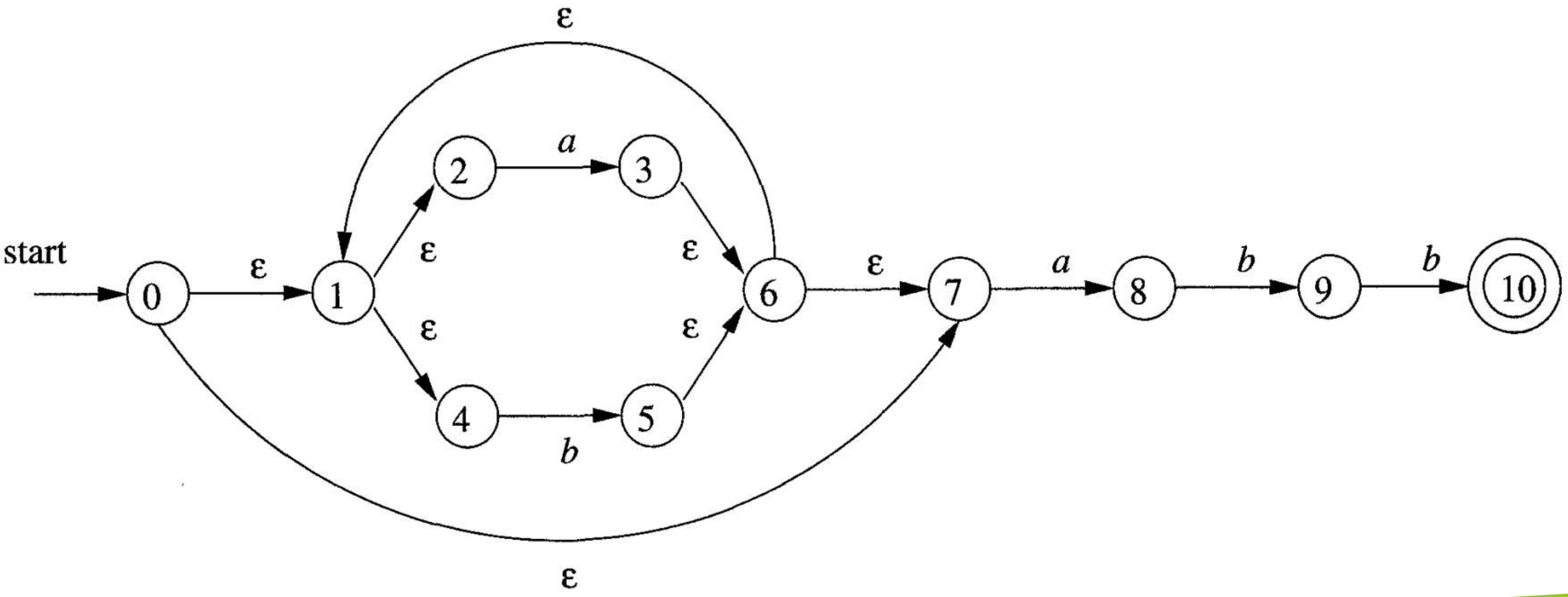


  - 对于表达式**r=s***
    - 增加2个新状态，4个新的ε转移



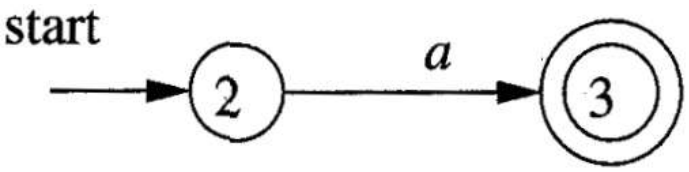
2. RE→NFA

- 例：将正则表达式**(a|b)*abb**转成NFA
- 最终结果：

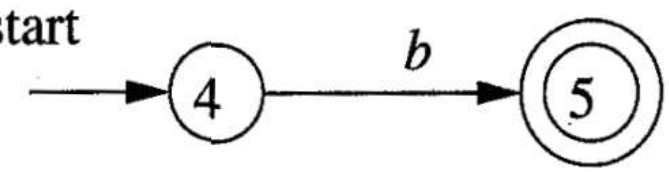


2. RE→NFA

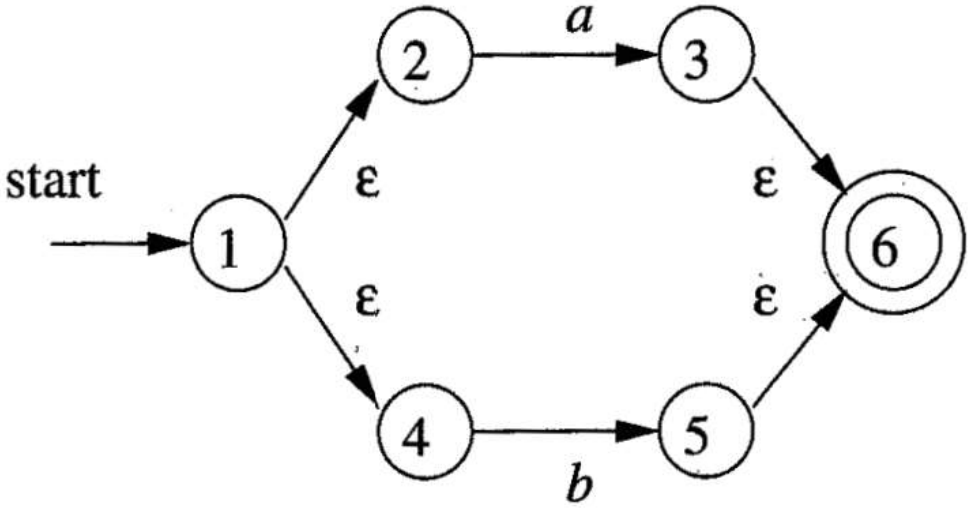
- 例：将正则表达式**(a|b)*abb**转成NFA
  - 对于r1=a：



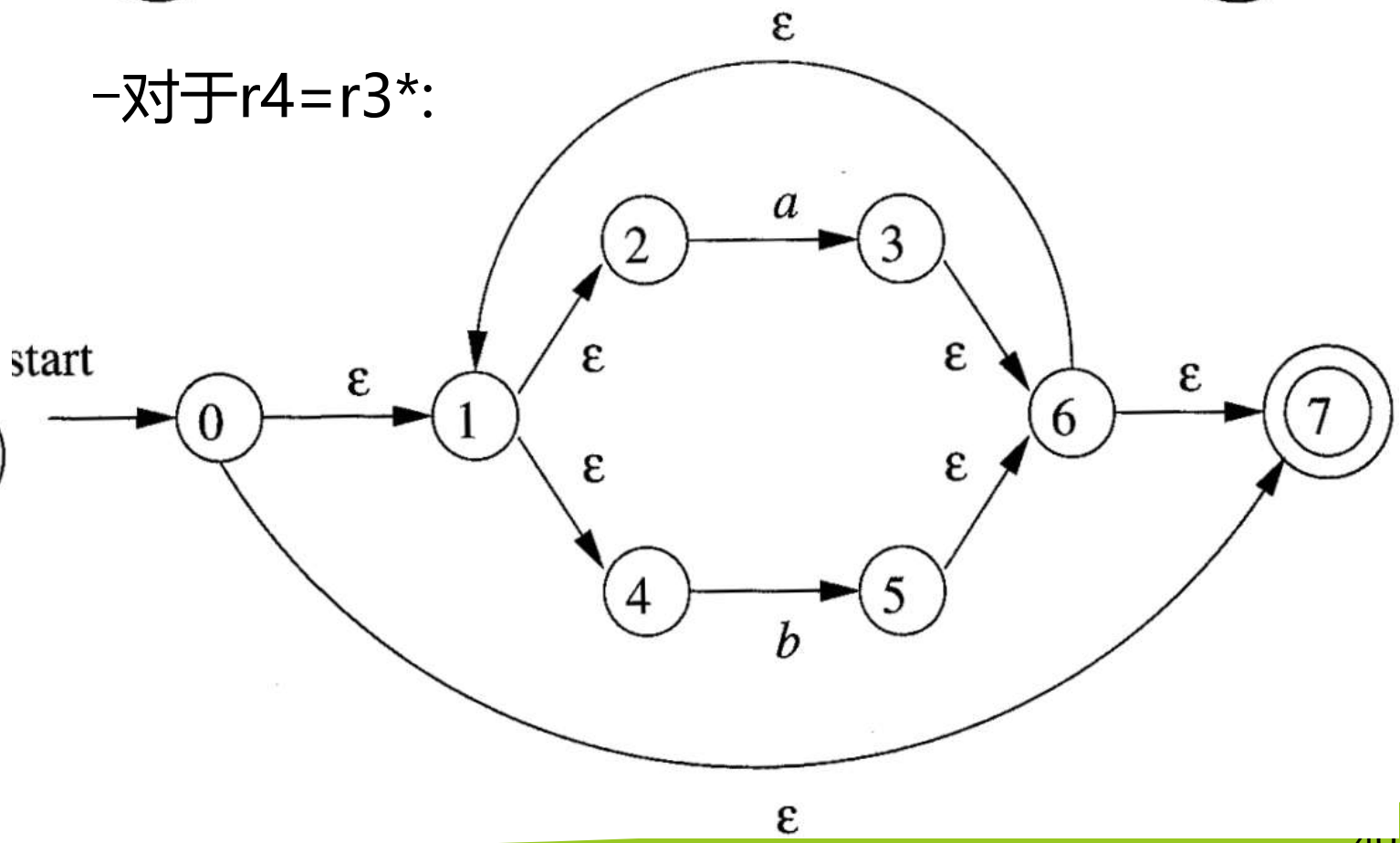
  - 对于r2=b：



  - 对于r3=a|b：

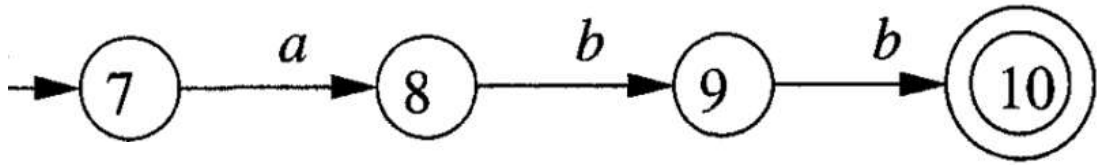


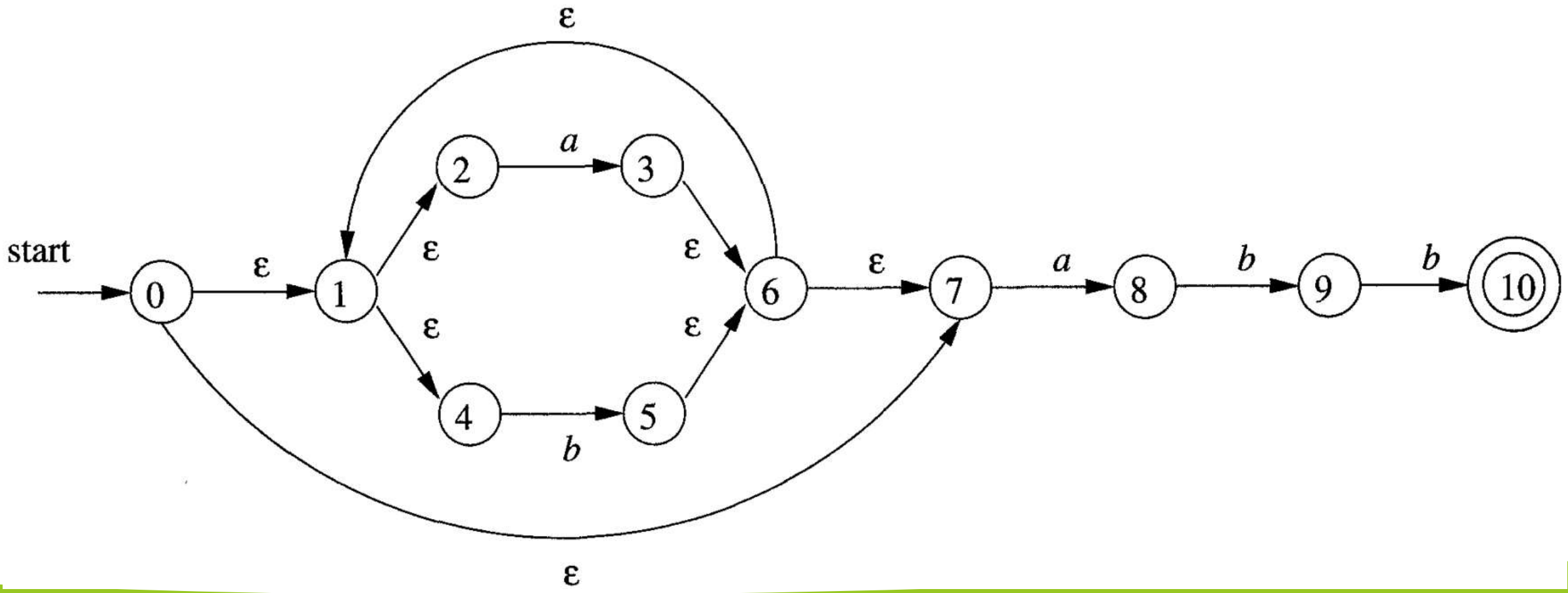
  - 对于r4=r3*：



2. RE→NFA

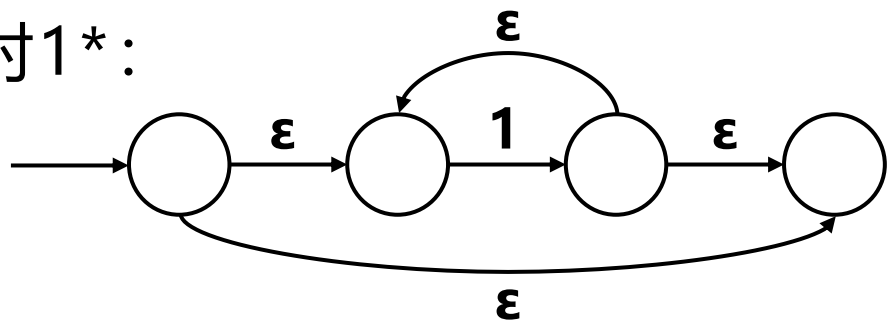
- 例：将正则表达式**(a|b)*abb**转成NFA
  - 对于abb:
  - 最终：



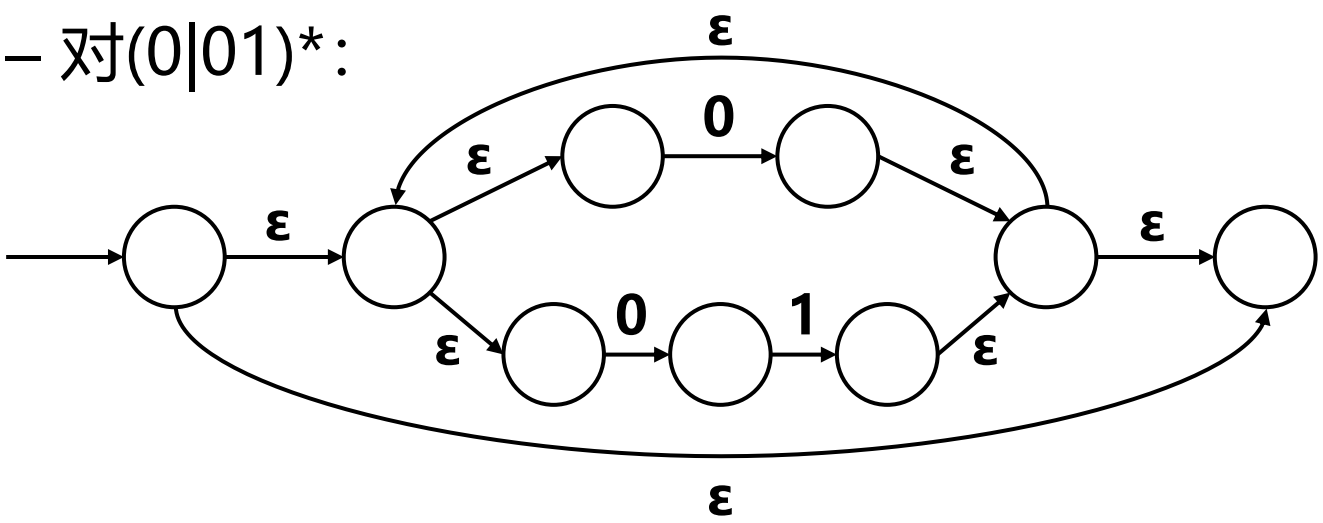


随堂练习(2)

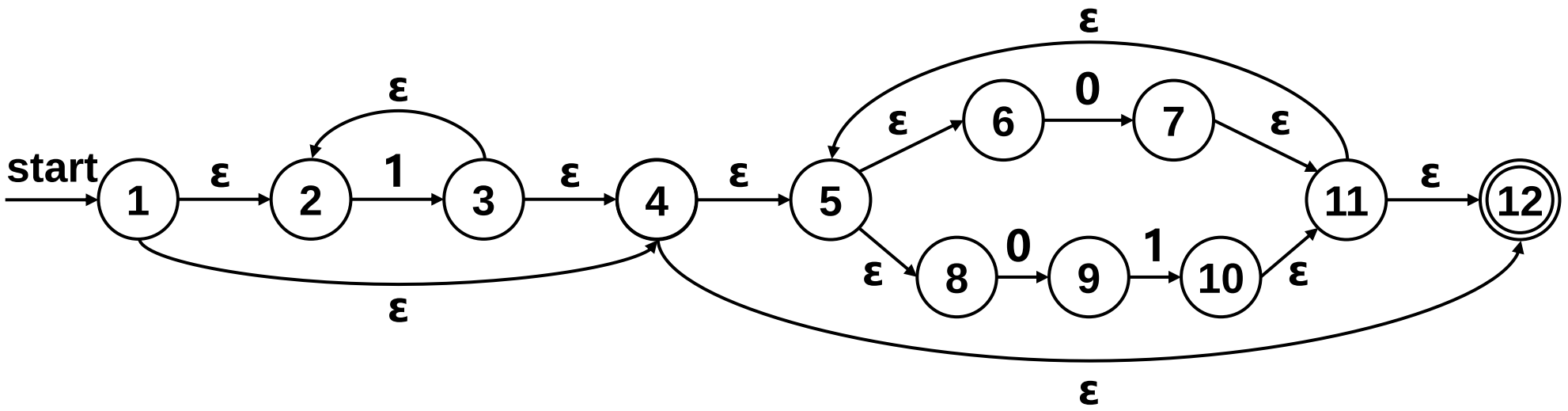
- 将正则表达式**1*(0|01)***转成NFA
  - 对1*:



  - 对(0|01)*:

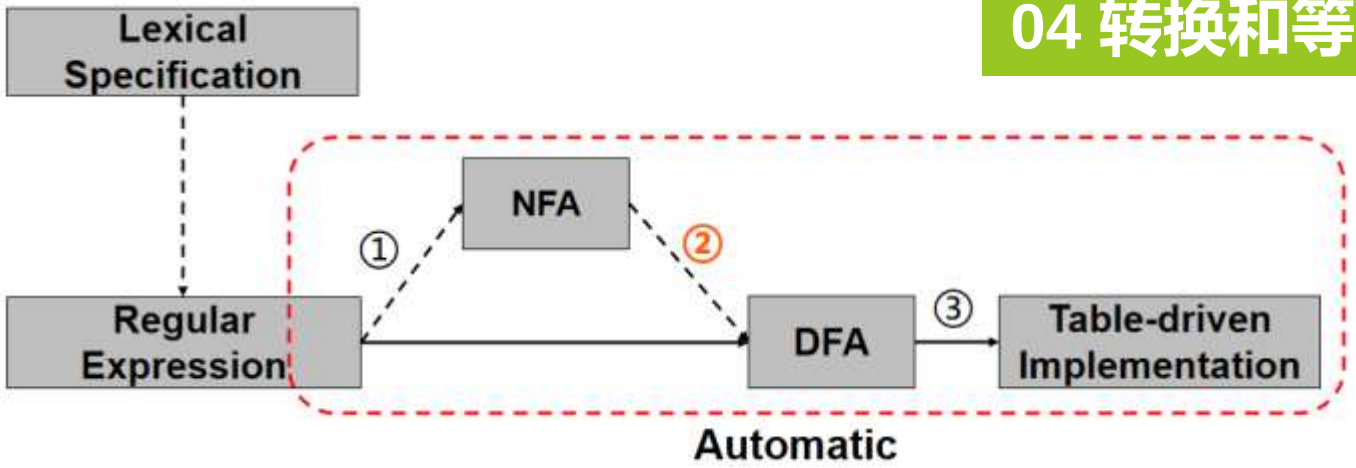


  - 最终:

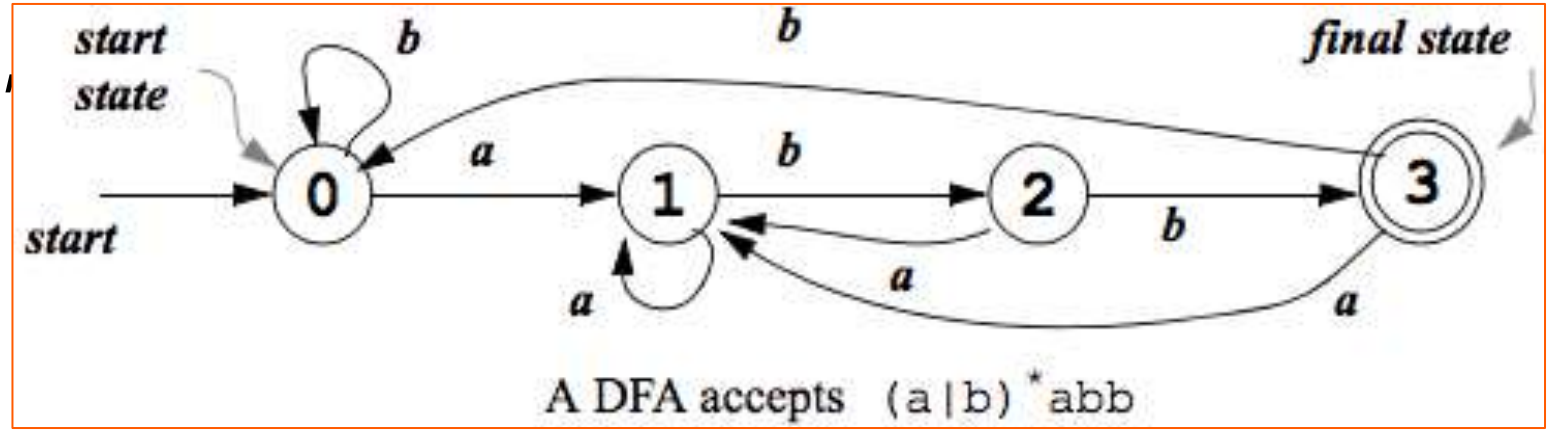


3. NFA→DFA

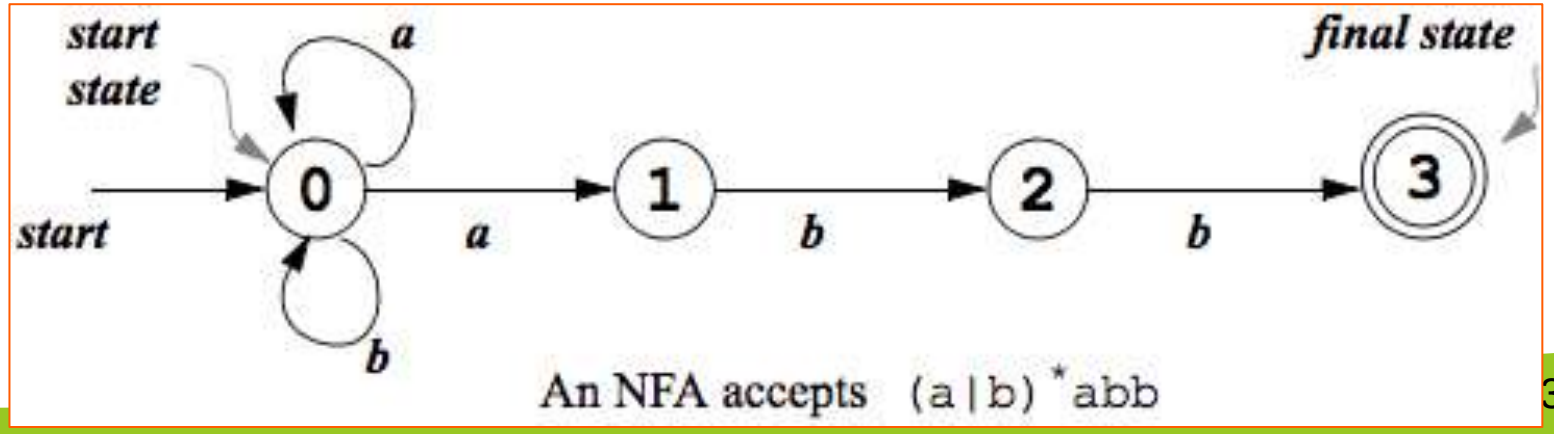
- 对于每个NFA M，存在一个与其**等价**的DFA M’



- 但**NFA的转移是不确定的**，更难以用机器模拟

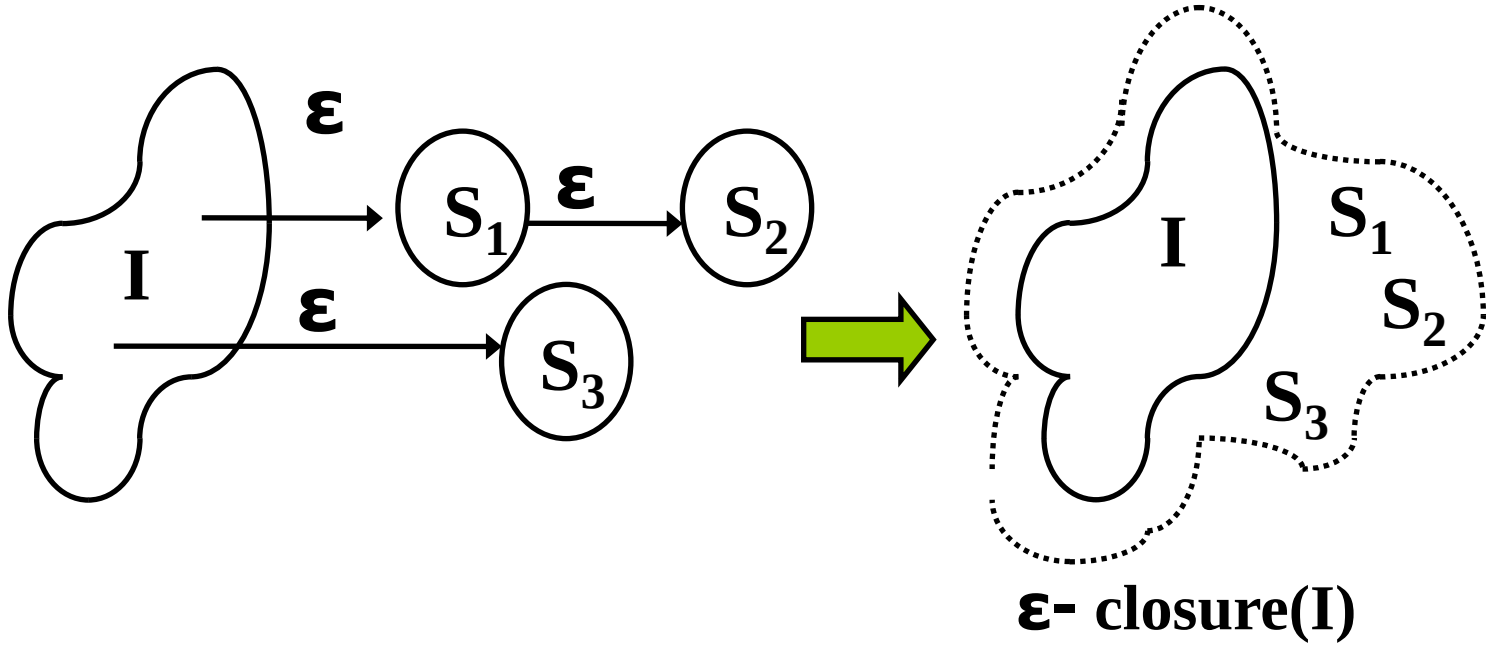


- 故须：NFA→DFA



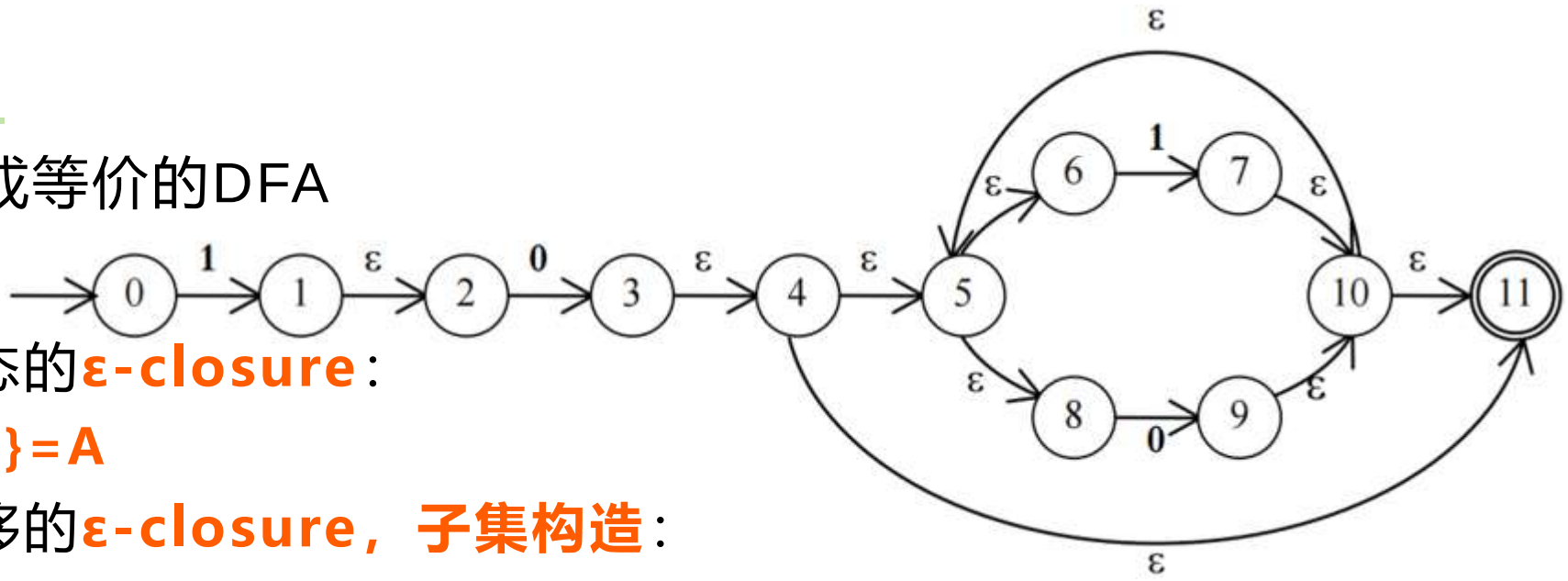
3. NFA→DFA

- **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。



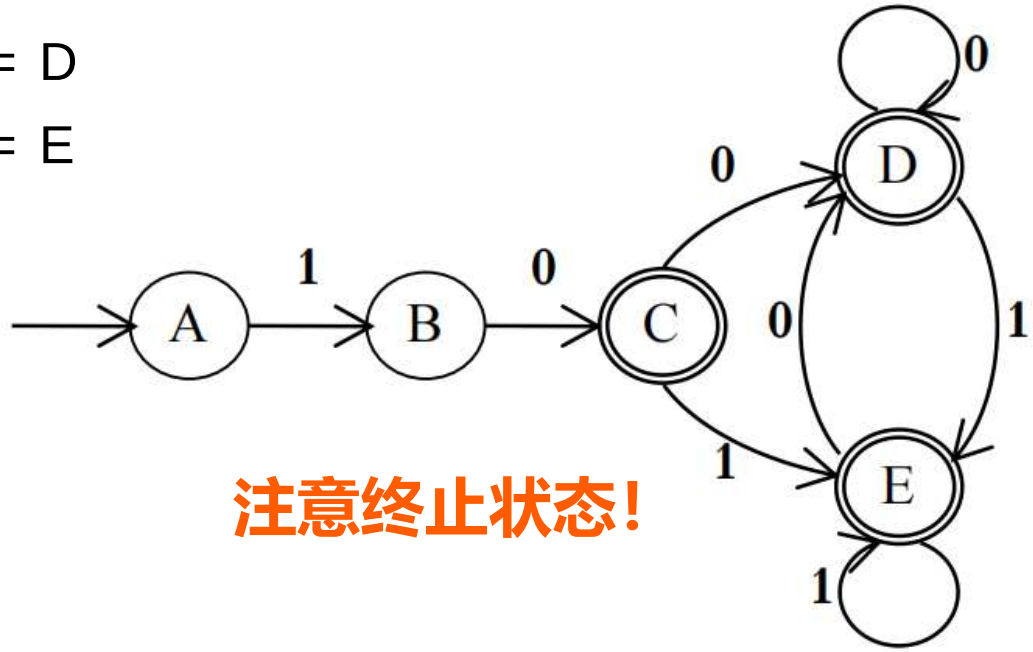
3. NFA→DFA

- 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

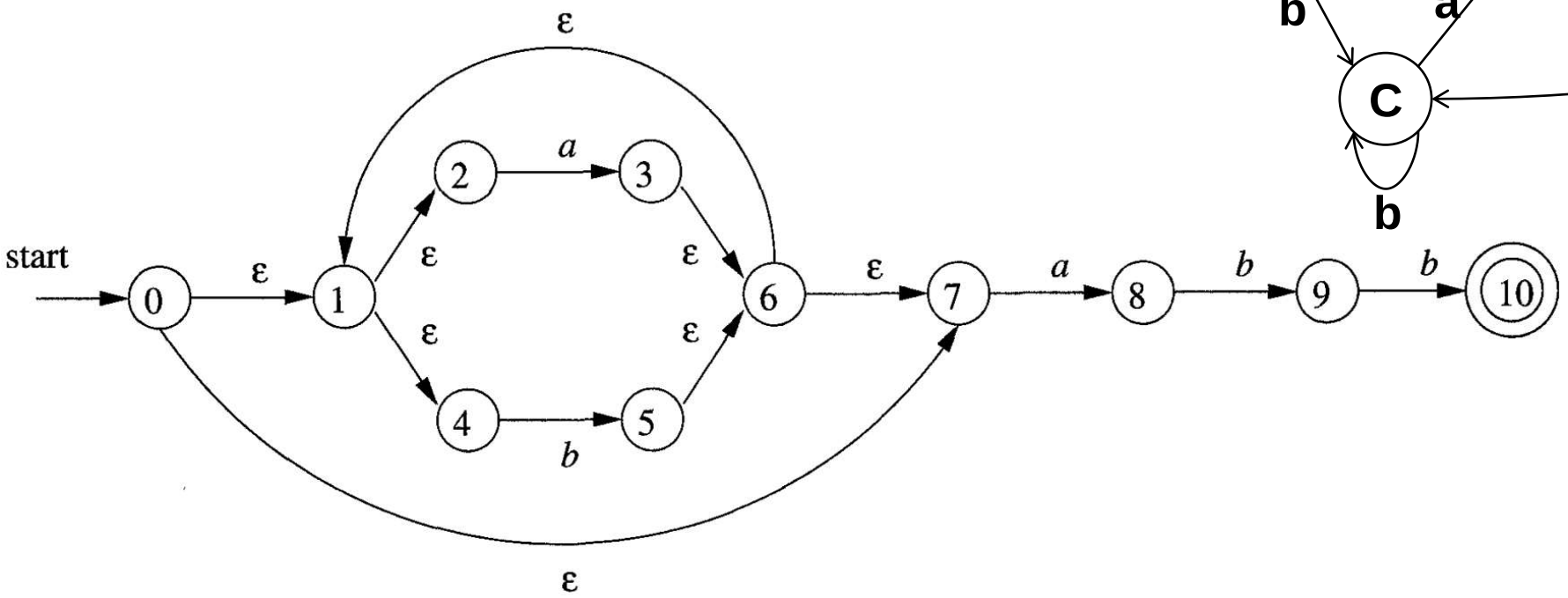
没有新的集合诞生→Step2 结束

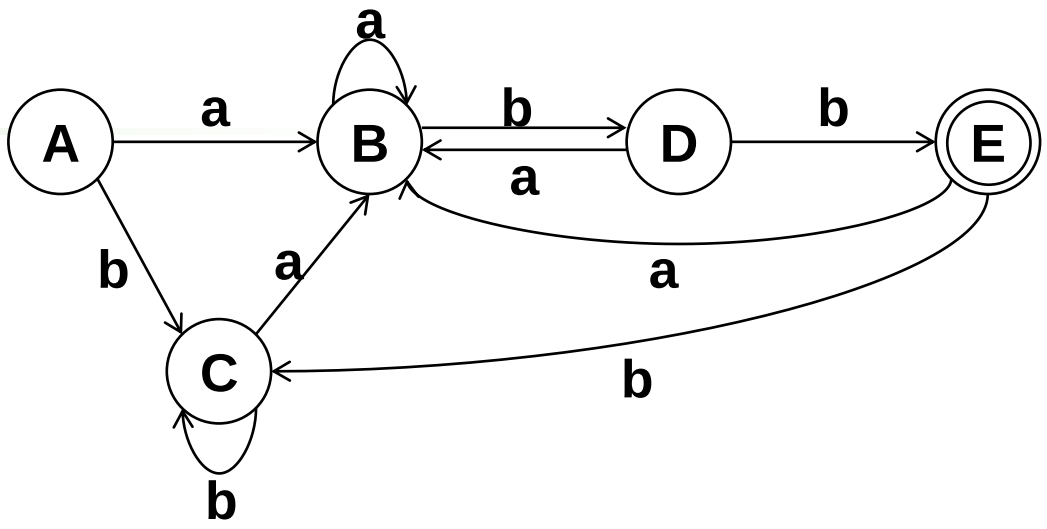


注意终止状态！

随堂练习(3)

- 将下图的ε-NFA转换成等价的DFA





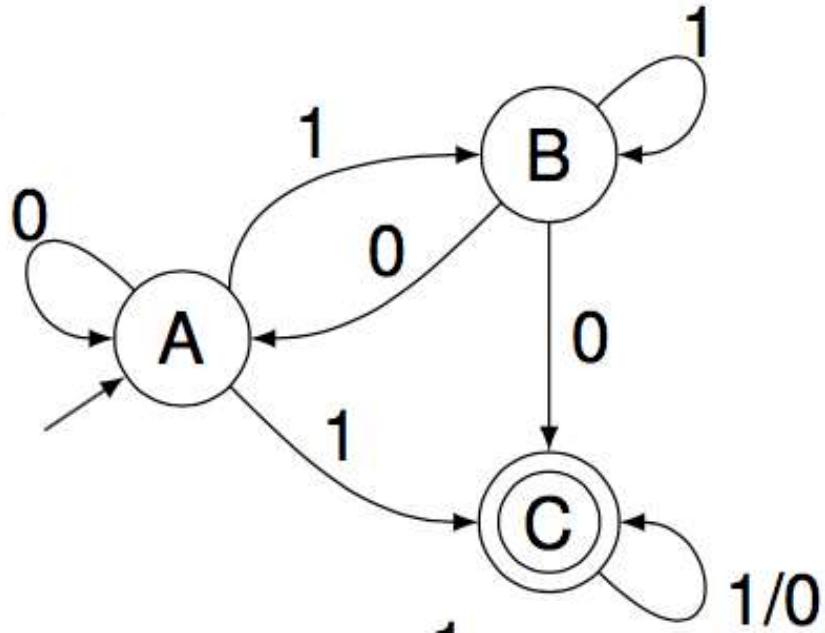
- ε-closure({0})={0,1,2,4,7}=A
- (A,a)=ε-closure({3,8})={1,2,3,4,6,7,8}=B
- (A,b)=ε-closure({5})={1,2,4,5,6,7}=C
- (B,a)=ε-closure({3,8})=B
- (B,b)=ε-closure({5,9})={1,2,4,5,6,7,9}=D
- (C,a)=ε-closure({3,8})=B
- (C,b)=ε-closure({5})=C
- (D,a)=ε-closure({3,8})=B
- (D,b)=ε-closure({5,10})={1,2,4,5,6,7,10}=E
- (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**

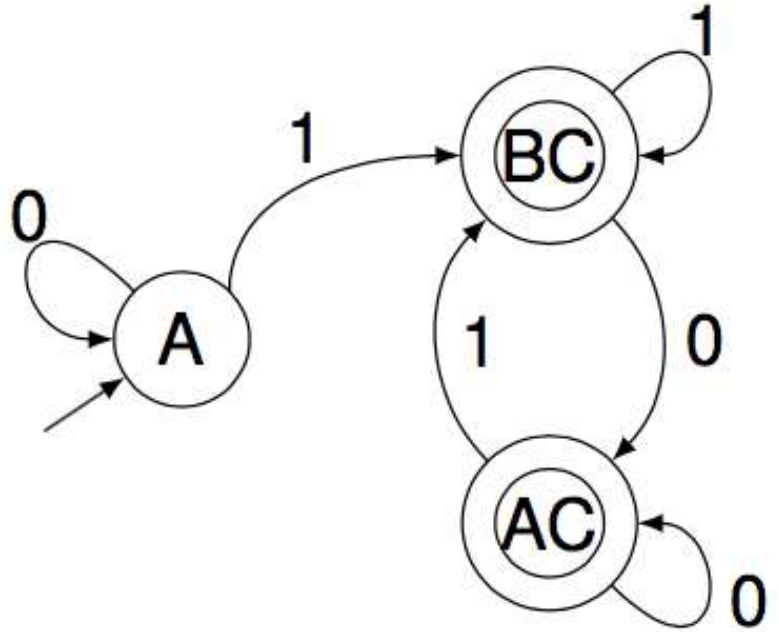


alphabet

state

	0	1
A	A	BC
BC	AC	BC
AC	AC	BC

没有新的集合诞生→Step1 结束

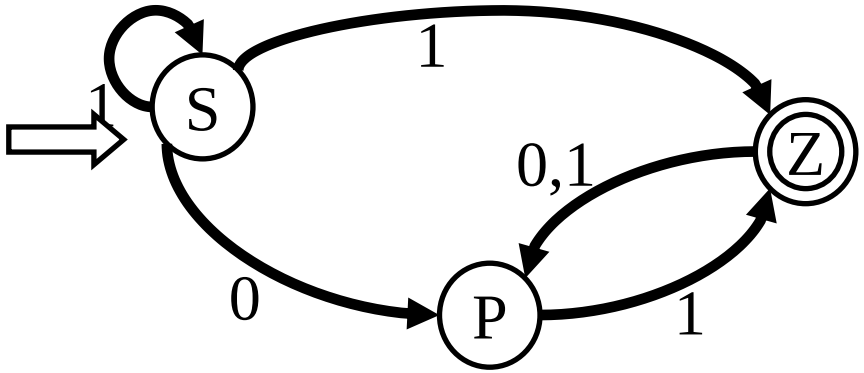


  - Step 2：根据新的状态集合画出DFA

注意终止状态！

随堂练习(4)

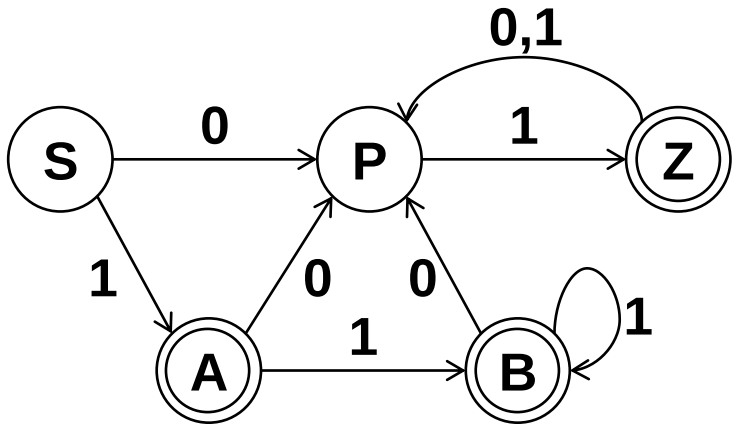
- 将下图的NFA(不含ε转移)转换成等价的DFA



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

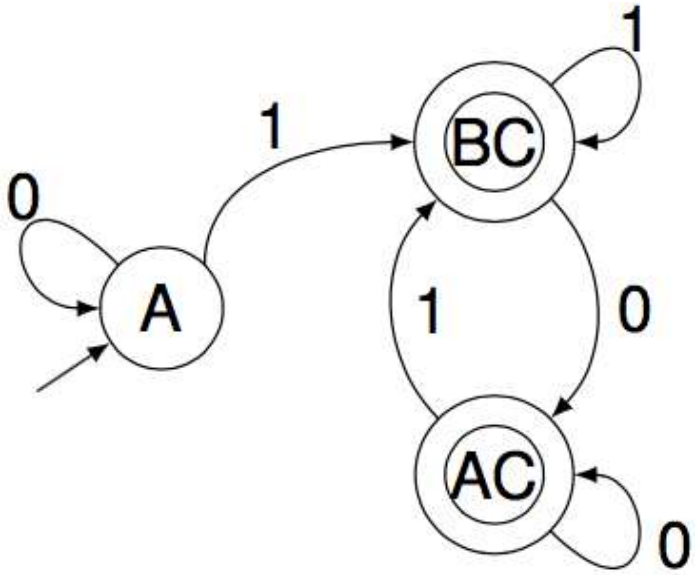
令A=SZ，B=SZP，则有：

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



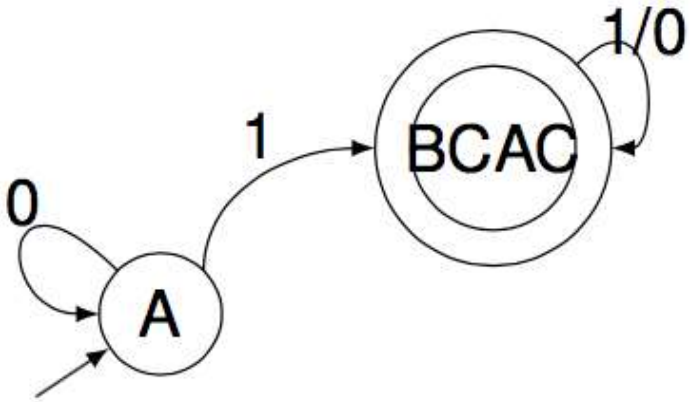
4. DFA化简[Reduction]

- 任务：**去掉多余状态，合并等价状态**
  - 多余状态：从开始状态出发无法到达的状态。
  - 等价状态：两个状态s和t等价的条件是:
    - ✓ 1. **一致性条件**—状态s和t必须同为可接受状态或不可接受状态。
    - ✓ 2. **蔓延性条件**—对于所有输入符号，状态s和状态t必须转换到等价的状态里。



1. BC和AC**同为可接受状态**
2. BC和AC对所有输入符号**都转移到等价的状态里**
   - BC on '0' → AC, AC on '0' → AC
   - BC on '1' → BC, AC on '1' → BC

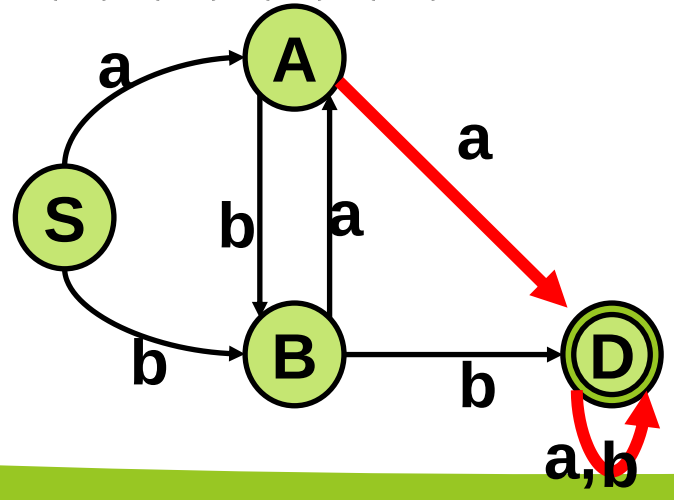
因此：BC和AC是**等价状态，可合并**。

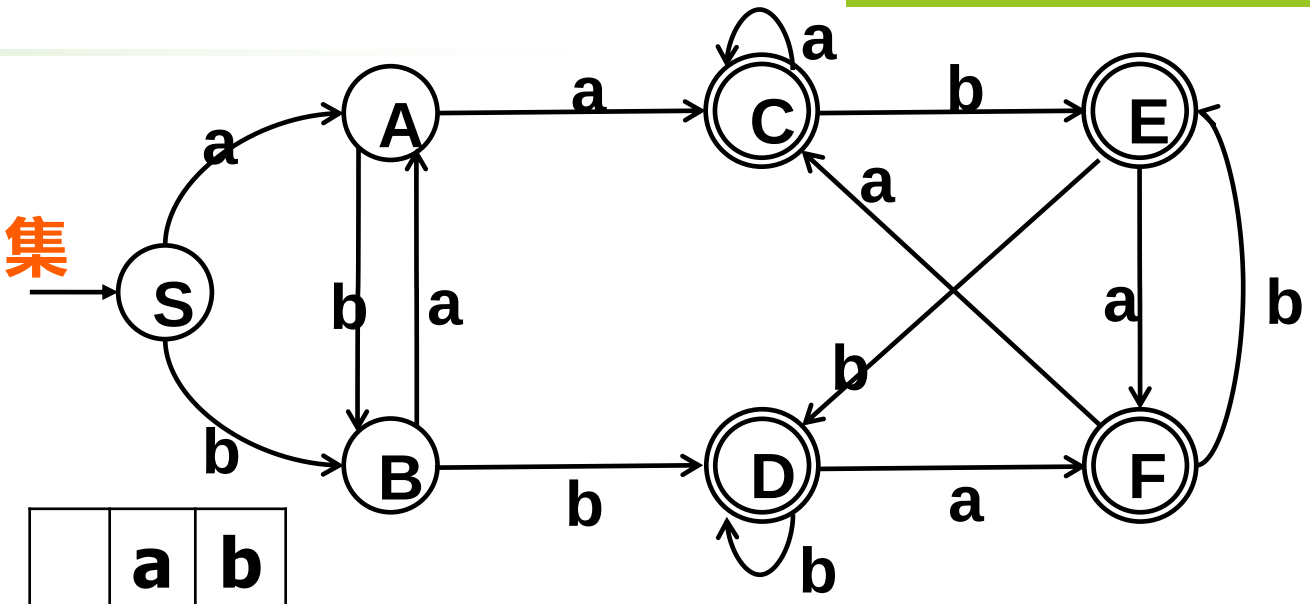


注意：弧也要合并！

4. DFA化简[Reduction]

- 例1：将右图DFA化简
  - **Step 1：将状态分成非终态和终态集**
    - ✓ **{S,A,B} {C,D,E,F}**
  - **Step 2：寻找子集中不等价状态**
    - ✓ {S,A,B}=>{S}{A,B}=>{S}{A}{B}
    - ✓ {C,D,E,F}
  - **Step 3：令D代表{C,D,E,F}**
    - ✓ {S},{A},{B},{D}





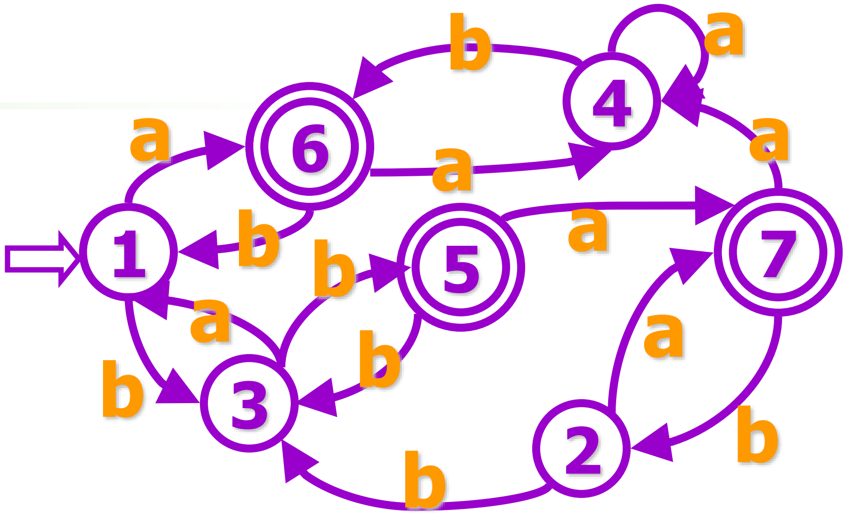
	a	b
S	A	B
A	C	B
B	A	D
C	C	E
D	F	D
E	F	D
F	C	E

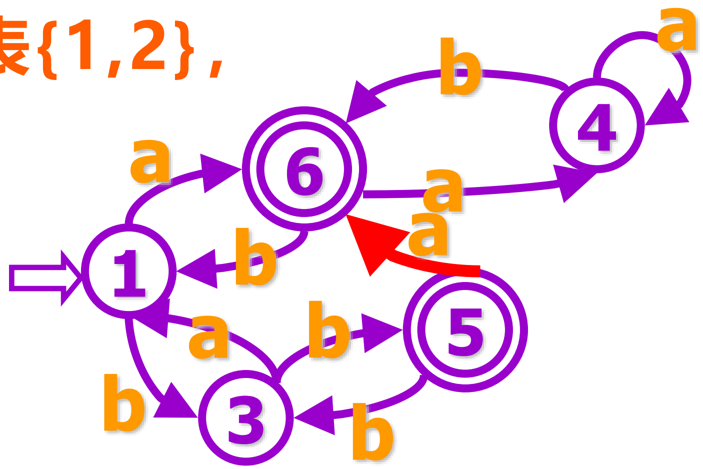
	a	b
S	A	B
A	C	B
B	A	D
CF	C	E
DE	F	D

	a	b
S	A	B
A	C	B
B	A	D
CFDE	CF	DE

4. DFA化简[Reduction]

- 例2：将右图DFA化简
  - Step 1：将状态分成非终态和终态集
    - ✓ {1,2,3,4} {5,6,7}
  - Step 2：寻找子集中不等价状态
    - ✓ {1,2,3,4}=>{1,2}{3,4}
    - ✓ {3,4}=>{3}{4}
    - ✓ {5,6,7}=>{5}{6,7}
    - ✓ 最终：{1,2}{3}{4}{5}{6,7}
  - Step 3：令1代表{1,2}，6代表{6,7}
    - ✓ {1},{3},{4},{5},{6}





	a	b
1	6	3
2	7	3
3	1	5
4	4	6
5	7	3
6	4	1
7	4	2

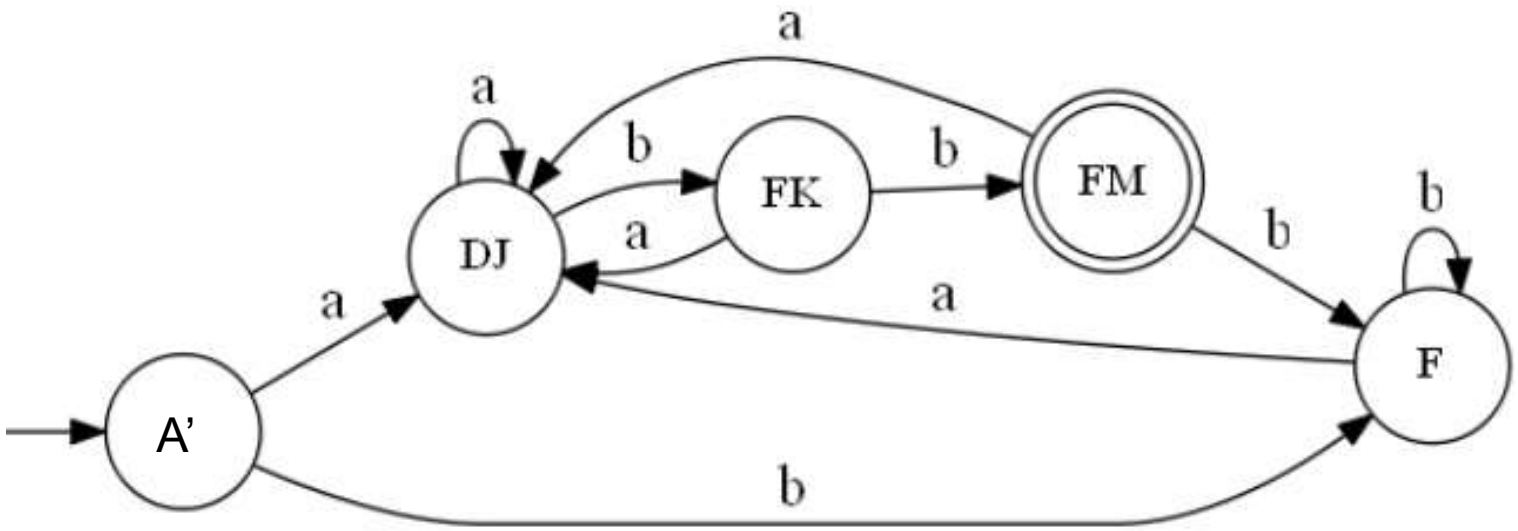
	a	b
1	6	3
2	7	3
3	1	5
4	4	6
5	7	3
6	4	1
7	4	2

	a	b
1	6	3
2	7	3
3	1	5
4	4	6
5	7	3
6	4	1
7	4	2

	a	b
1	6	3
2	7	3
3	1	5
4	4	6
5	7	3
6	4	1
7	4	2

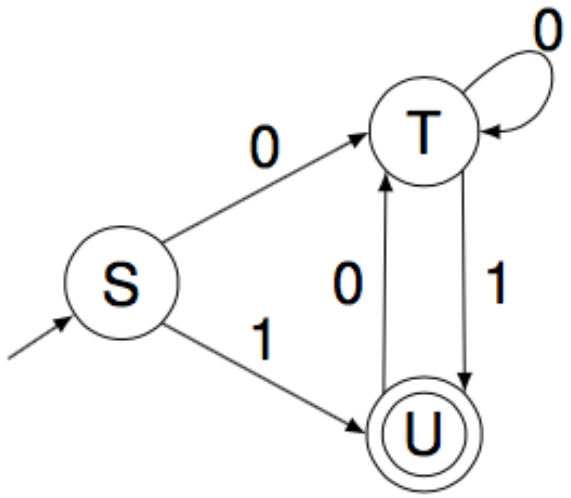
随堂练习(5)

- 将下图DFA化简



5. DFA→Table-drive Implementation

- DFA可转成Table-drive Code



alphabet

state

	0	1
S	T	U
T	T	U
U	T	x

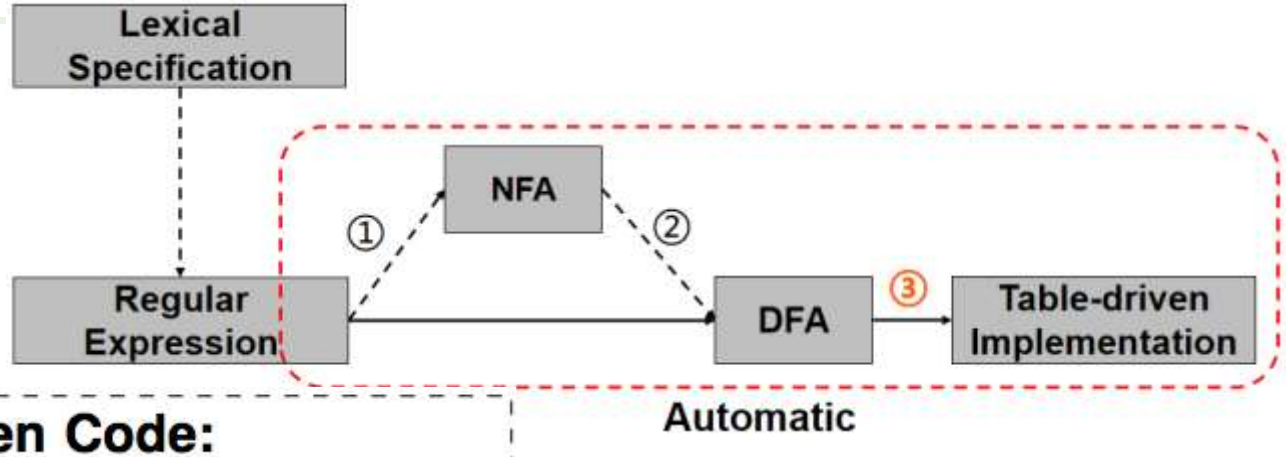


Table-driven Code:

```
DFA() {
    state = "S";
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == "x")
            print("reject");
    }
    if (state ∈ F)
        printf("accept");
    else
        printf("reject");
}
```

Q: which is/are accepted?

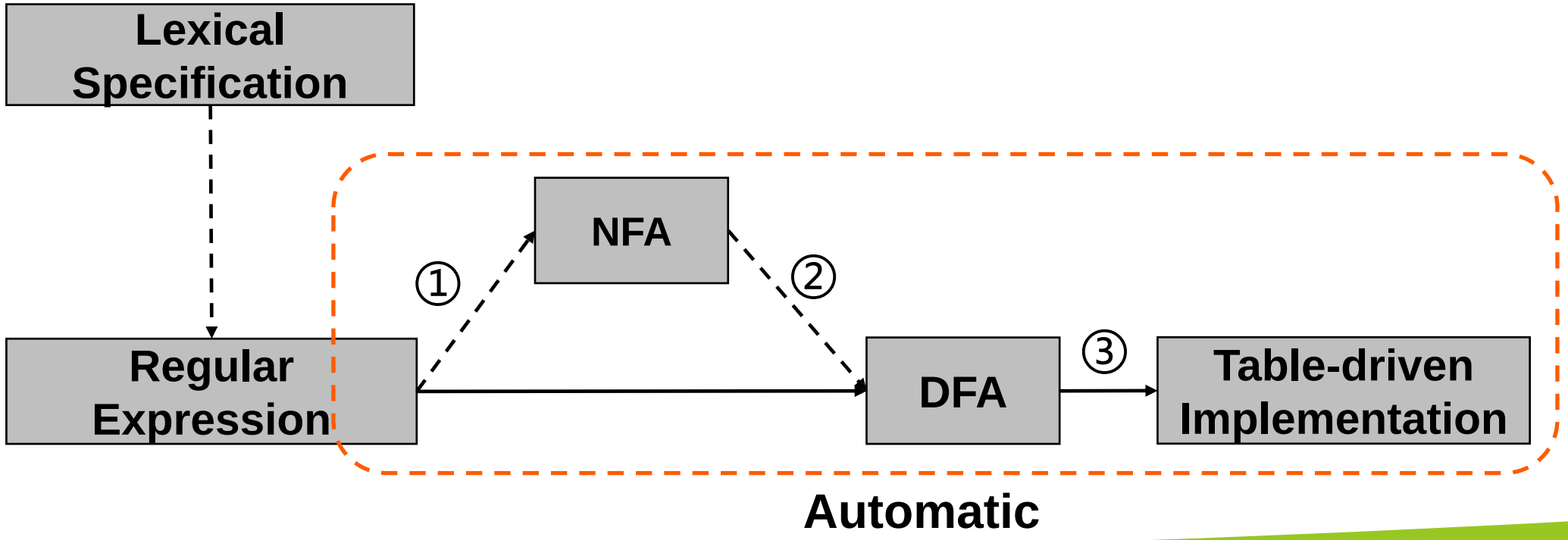
111 ✗

000 ✗

001 ✓

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



6. 一些讨论

- 关于Table-drive Implementation
  - 表格是一种高效实现[efficient]
    - ✓ 仅需**有穷空间O(S x ∑)**
      - 转换表的Size
    - ✓ 仅需**有穷时间O(输入长度)**
      - 状态转换的个数
  - 表格实现的优劣
    - ✓ pros：在给定的状态和输入下能轻松找到转换
    - ✓ cons：当输入字母很大时，会占用大量空间，但大多数状态对大多数输入符号没有任何移动（表格很稀疏）

6. 一些讨论

- 关于**空间复杂度[Space Complexity]**
  - NFA：
    - ✓ 在任何时刻可能有多个状态，假设为N
    - ✓ 空间复杂度：**O(N)**
  - DFA：
    - ✓ 如果NFA有N个状态，则DFA一定在这N个状态的某个子集中
    - ✓ 非空子集：**2^N-1**
    - ✓ 空间复杂度：**$O(2^N)$**，其中N为NFA中的原始状态个数（**指数级状态爆炸**）

6. 一些讨论

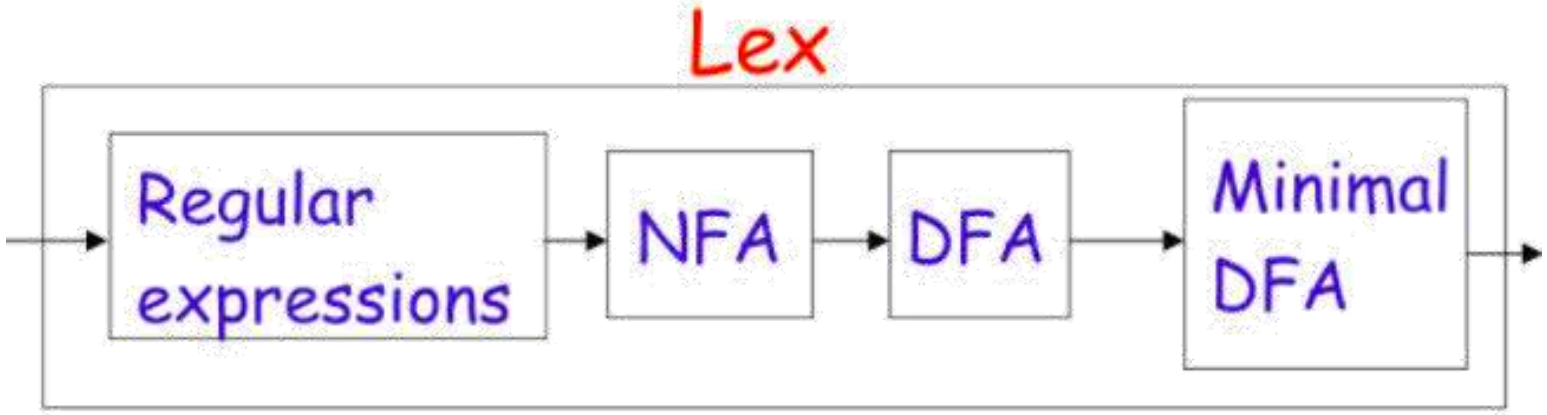
- 关于**时间复杂度[Time Complexity]**
  - DFA
    - ✓ 需要**O(|X|)步**，其中|X|是输入长度
    - ✓ **每一步花费O(1)**常数时间
      - 若当前状态是S且输入为c，则读表T[S,c]，更新当前状态为T[S,c]
    - ✓ 时间复杂度为**O(|X|)**
  - NFA
    - ✓ 需要**O(|X|)步**，其中|X|是输入长度
    - ✓ **每一步花费$O(N^2)$**时间，其中N为状态的个数
      - 当前状态是一组潜在状态，最多N个
      - 对于输入c，必须合并所有T[Spotential, c]，最多N次，每次合并操作花费O(N)时间（为避免状态重复，须遍历状态列表，长度为N）
    - ✓ 时间复杂度为**$O(|X|*N^2)$**

CONTENTS

目 录

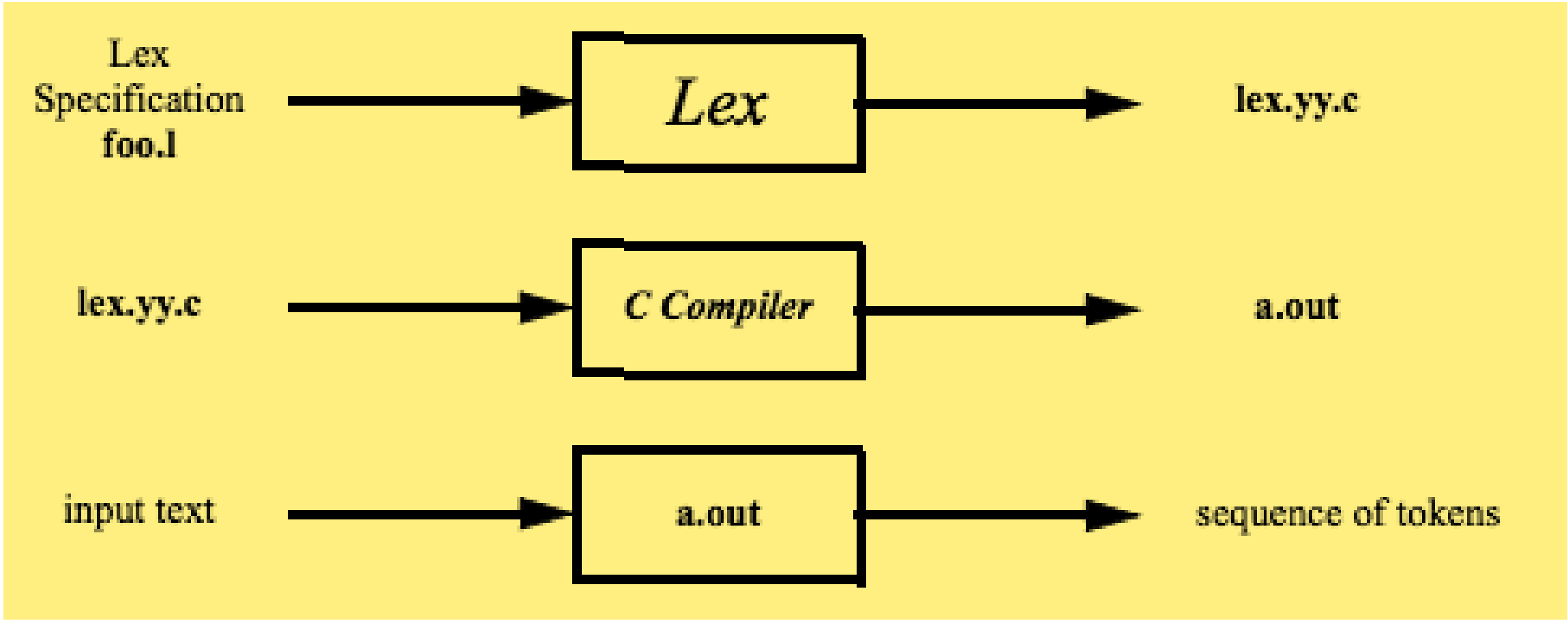
1. 实际实现

- Lex: RE → NFA → DFA → Table



- 大多数其他自动词法分析器也选择DFA而不是NFA
  - 用空间换取速度[Trade off space for speed]

2. Lex


Lex
Specification
foo.l
Lex
lex.yy.c
lex.yy.c
C Compiler
a.out
input text
a.out
sequence of tokens

2. Lex

- Lex规范文件示例

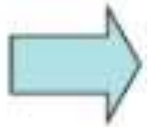
lex

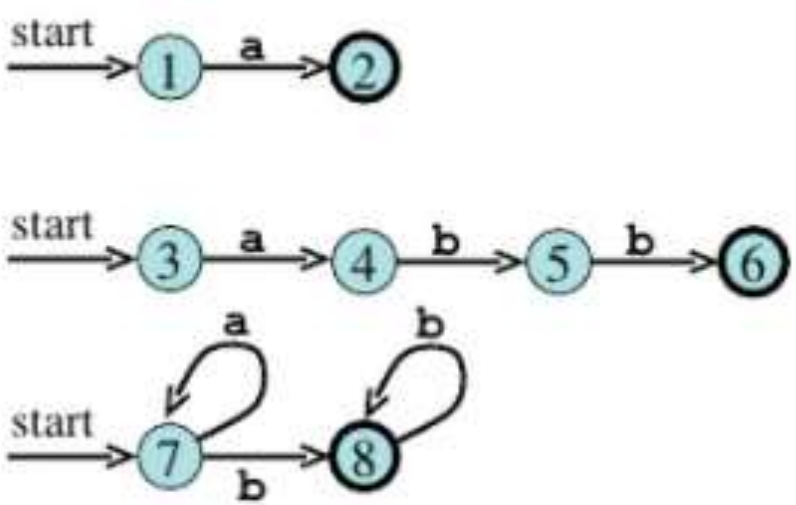
```
%{
#include <stdio.h>
#include <stdlib.h>
int lineno = 1;
%}

letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number {digit}+
```

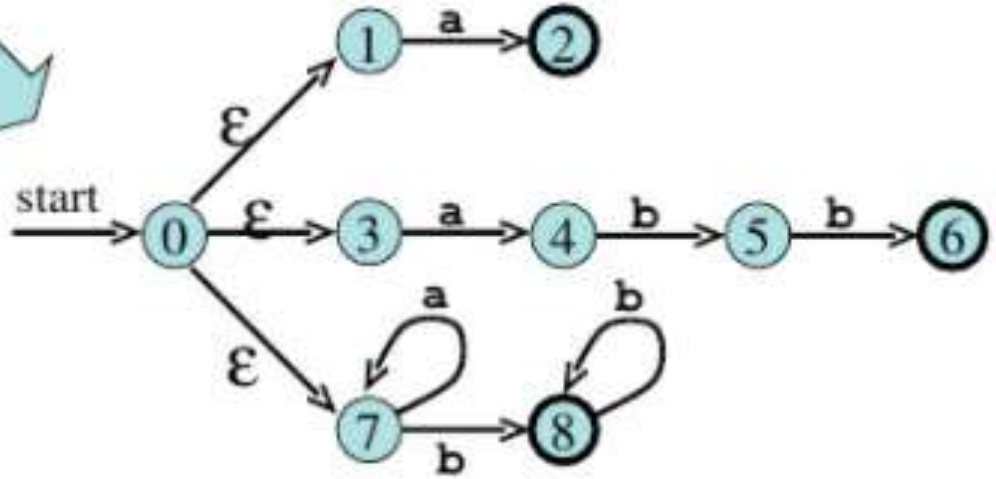
2. Lex

- 例：假设现有3种模式，对应3个NFA
  - 将3个NFA组合成1个NFA
  - 添加开始状态和ε转移

a	{ $action_1$ }
abb	{ $action_2$ }
a*b+	{ $action_3$ }







2. Lex

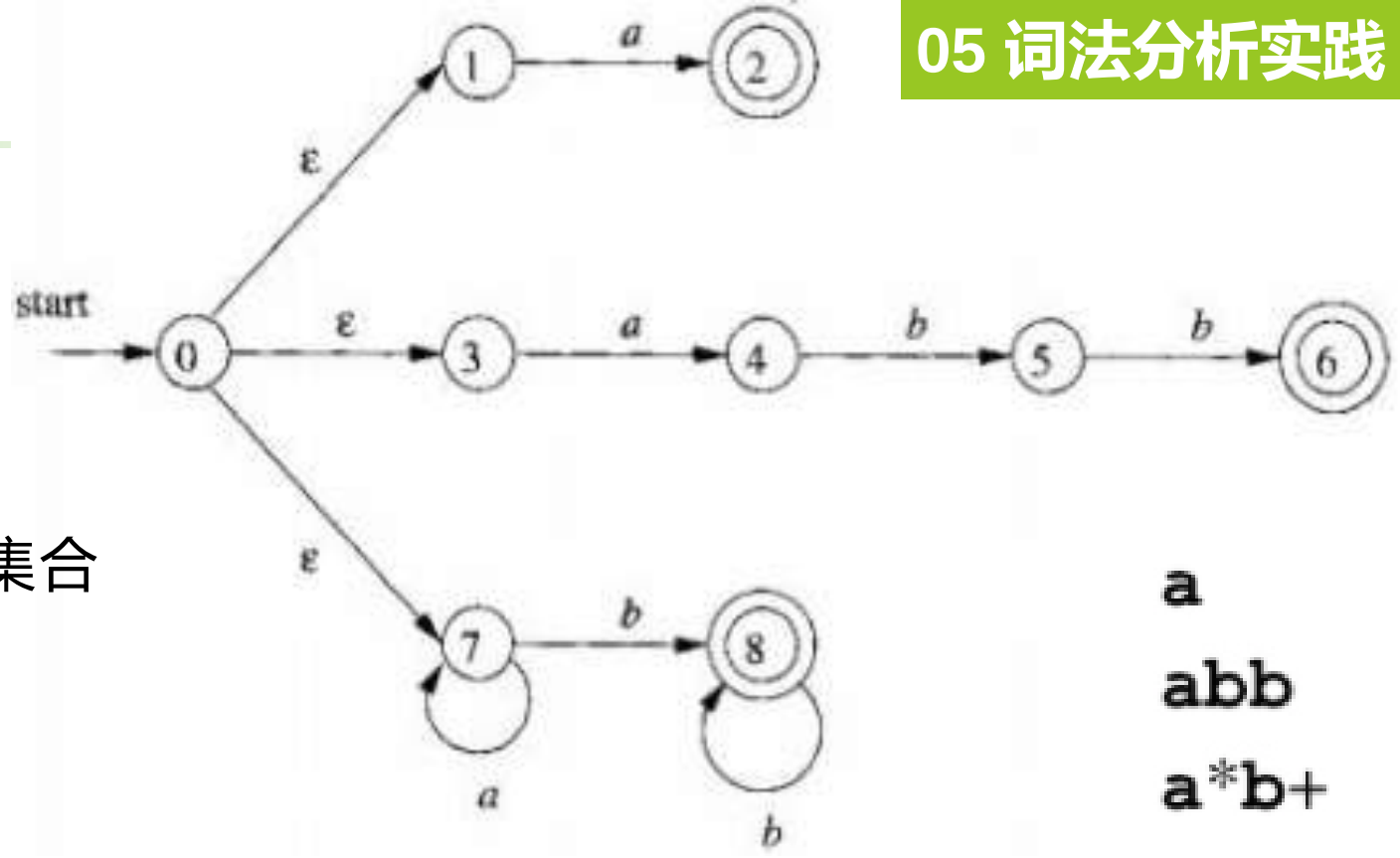
```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }

%%

int main(){
  yylex();
  return 0;
}
```

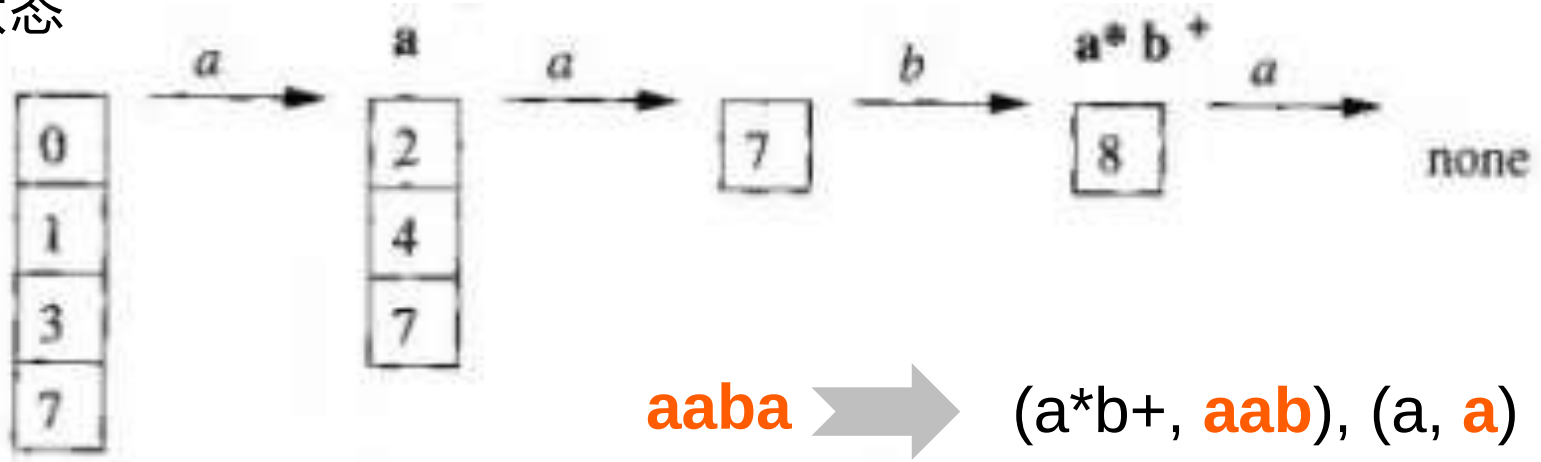
```
root@aa51dde06c76:~/test# echo "aaba" | ./mylex

<ptn3, aab>
<ptn1, a>
root@aa51dde06c76:~/test# echo "abba" | ./mylex

<ptn2, abb>
<ptn1, a>
```

$flex lex.l
$clang lex.yy.c -o mylex -ll

2. Lex

- NFA
  - 输入：**aaba**
    - ✓ ε-closure(0) = {0, 1, 3, 7}
    - ✓ 寻找一组**包含接受状态**的状态集合
    - ✓ 状态8：a*b+匹配成功
      - aab即为词素
      - 继续寻找接受状态

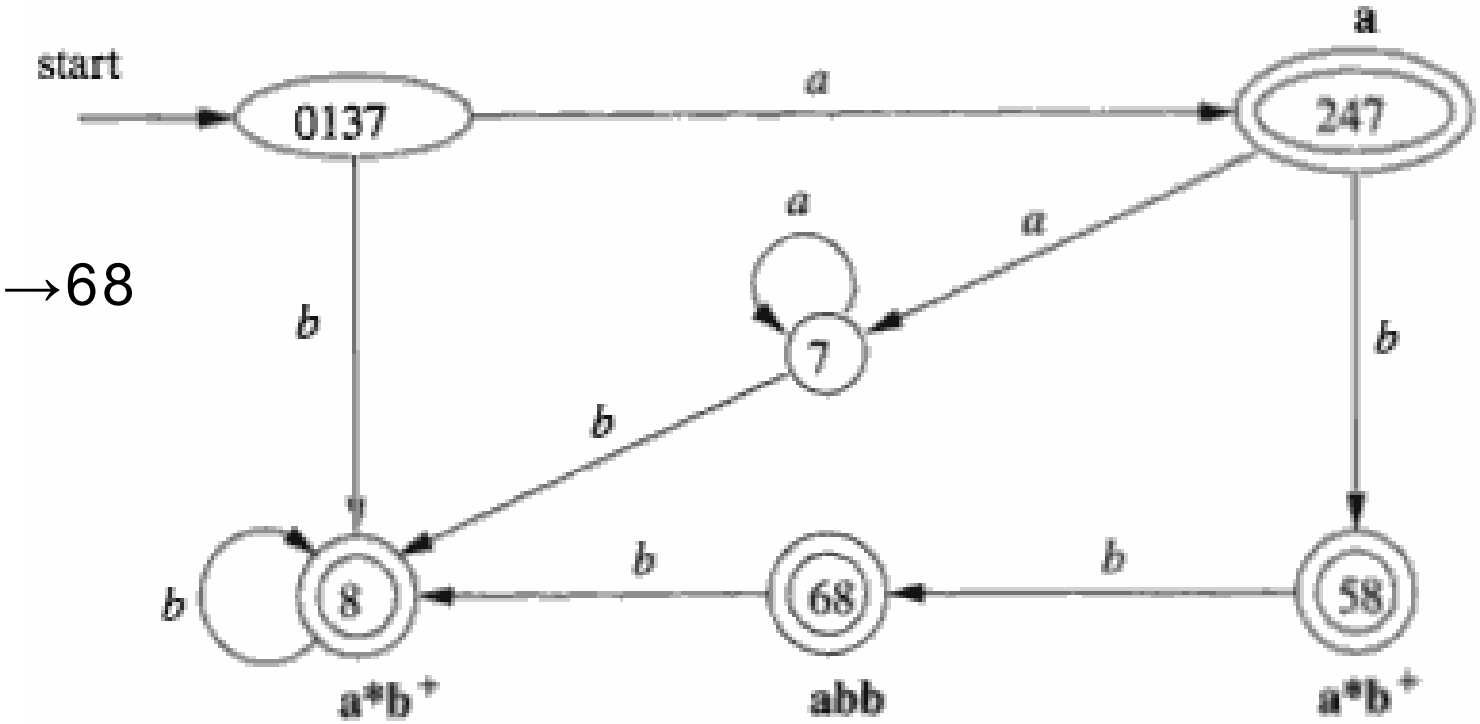




aaba ⇒ (a*b+, aab), (a, a)

2. Lex

- DFA
  - 输入：**abba**
    - ✓ 状态序列：0137→247→58→68
    - ✓ 状态68：abb匹配成功
      - abb即为词素
      - 继续寻找接受状态



a

abb

a*b+

2. Lex

```
a       { action1 }
abb     { action2 }
a*b+    { action3 }
```

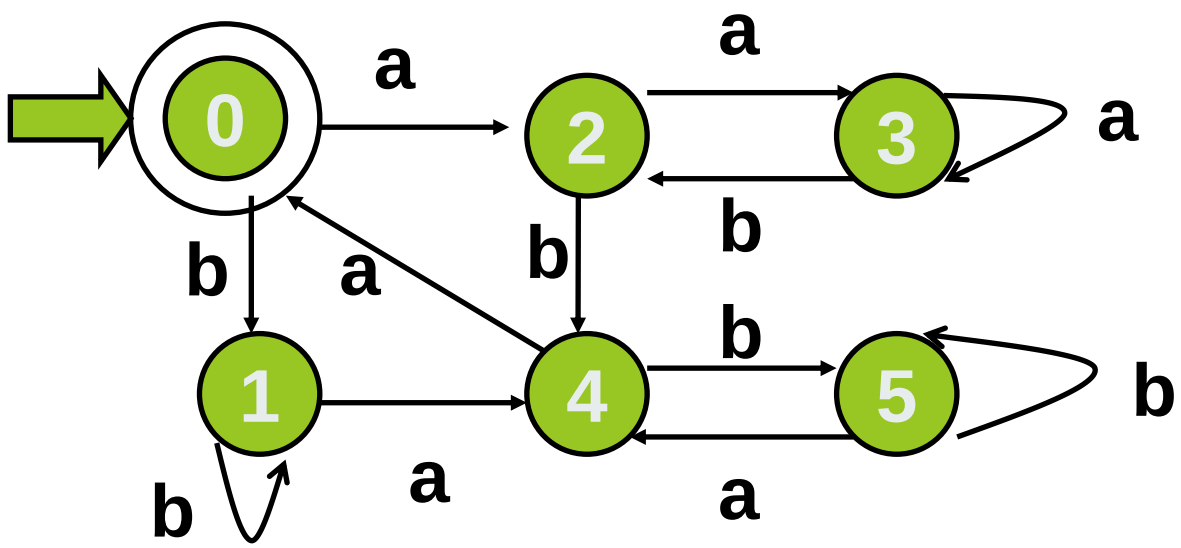
- 若有多种匹配的可能性？
  - **寻找最长匹配**
    - ✓ 例：输入**aabbb**，符合ptn1、ptn2和ptn3，取最长匹配——ptn3
  - **先出现的规则优先级更高**
    - ✓ 例：输入**abb**，符合ptn2和ptn3，取先出现的ptn2

2. Lex

- 如何匹配关键字[keywords]?
  - 方法1：为关键字创建正则表达式，并将它们放在标识符的正则表达式之前，让他们具有更高的优先级
    - ✓ 会导致更臃肿的FA
  - 方法2：使用相同的正则表达式识别关键字和标识符，但使用特殊的**关键字表[keyword table]**进行区分
    - ✓ 会导致更精简的FA
    - ✓ 但是需要额外的表查找
  - **方法2更常用**

第三章课后作业

- 将下图DFA最小化



第三章课后作业

- 提交要求：
  - 文件命名：学号-姓名-第三章作业；
  - 文件格式：.pdf文件；
  - 手写版、电子版均可；若为手写版，则拍照后转成pdf提交，但**须注意将照片旋转为正常角度，且去除照片中的多余信息**；电子版如word等转成pdf提交；
  - 提交到超算习堂（第三章作业）处；
  - 提交ddl：**3月25日晚上12:00**；
  - **重要提示：不得抄袭！**