
编译原理

2. 词法分析

rainoftime.github.io
浙江大学
计算机科学与技术学院

课程内容

1. Introduction
- 2. Lexical Analysis**
3. Parsing
4. Abstract Syntax
5. Semantic Analysis
6. Activation Record
7. Translating into Intermediate Code
8. Basic Blocks and Traces
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

本讲内容

1

词法分析概述

2

正则表达式

3

有穷自动机

4

词法分析器自动生成

5

Lex工具

1. 词法分析概述

什么是词法分析？

词法分析 (Lexical Analysis)

- 程序是以**字符串**的形式传递给编译器的

```
if (i == j)
```

```
    z = 0;
```

```
else
```

```
    z = 1;
```



```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- 词法分析: 将输入字符串识别为**有意义的子串**
 - 其他辅助任务: 过滤注释、空格, etc.

词法分析 (Lexical Analysis, Lexer, Scanner)

- **词法分析: 将输入字符串识别为有意义的子串**
 - Partition input string into substrings (**lexeme**)
 - Classify them according to their role (**tokens**)
- **Token (词法记号, 单词)**
 - English: noun, verb, adjective, ...
 - Programming language: keyword, identifier, ..
- **Lexeme (词素)**
 - A member (string) of the set (token) such as “else”, “if”
 - An instance of the token

例: Token和Lexeme

- **Token: 关键字、操作符、标识符、字符串等**

```
if (i == j) print("equal");  
else num 5 = 1;
```

Token	Lexeme	Token的非形式化定义
if	if	字符i, f
else	else	字符e, l, s, e
relation	< , <= , = , ...	< 或 <= 或 = 或 ...
id	sum, count, D5	由字母开头的字母数字串
number	3.1, 10, 2.8 E12	任何数值常数

例: 词法分析

- 词法分析: 字符流 → Token流

```
float match0(char *s) /* find a zero */
{ if (!strcmp(s, "0.0", 3))
    return 0.;
}
```

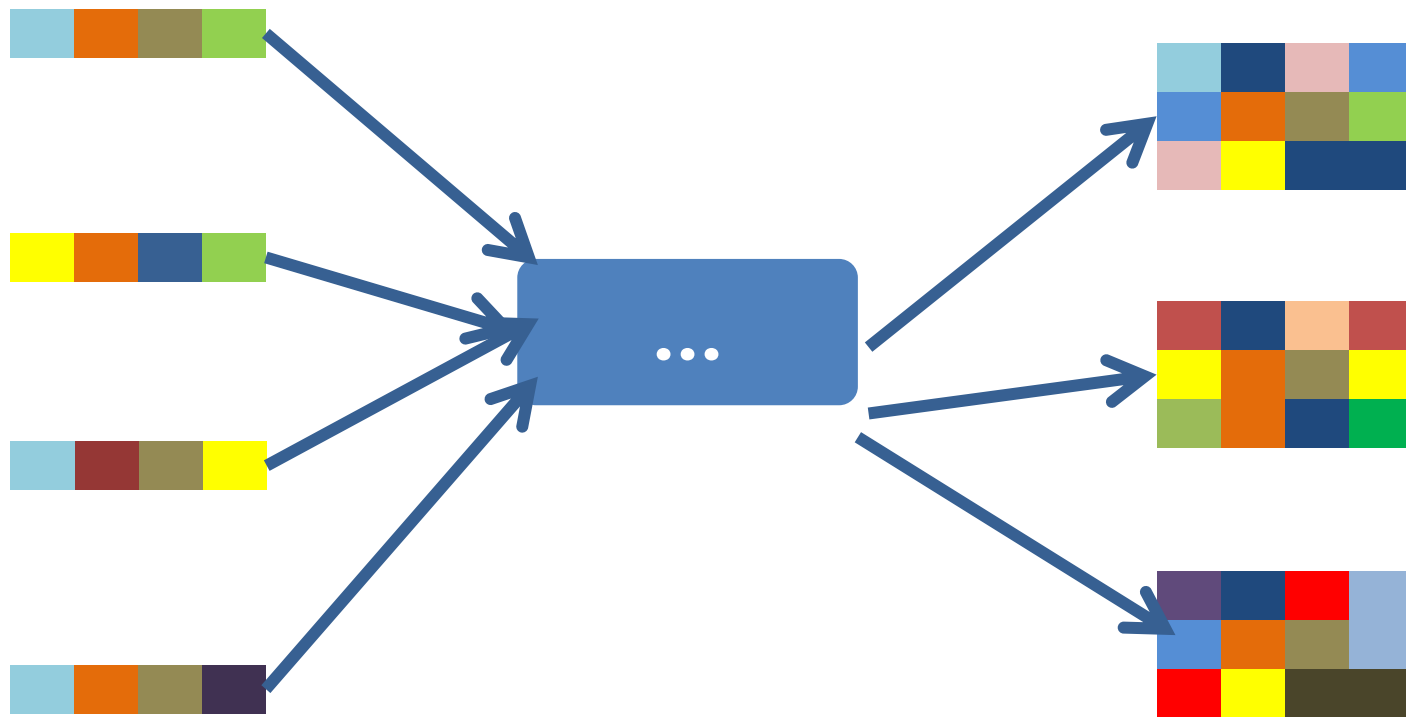
FLOAT	ID(match0)	LPAREN	CHAR	STAR	ID(s)
RPAREN	LBRACE	IF	LPAREN	BANG	
ID(strcmp)	LPAREN	ID(s)	COMMA	STRING(0.0)	
COMMA	NUM(3)	RPAREN	RPAREN	RETURN	
REAL(0.0)	SEMI	RBRACE	EOF		

词法分析器的构造

Program = Specification + Implementation
“What” “How”

声明式的规范

词法分析器

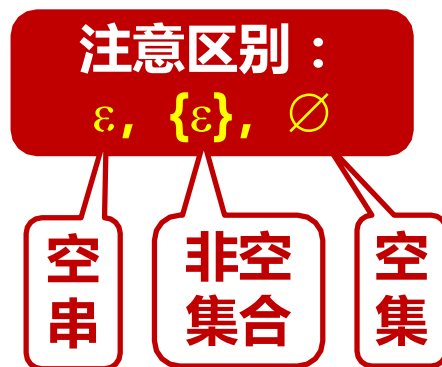


2. 正则表达式

如何形式化地描述词法?

字母表和串

- **字母表 (alphabet) : 符号的有限集合**
 - 字母、数字、标点符号、...
- **串 (String, word) : 字母表中符号的有穷序列**
 - 串 s 的**长度**，通常记作 $|s|$ ，是指 s 中**符号的个数**
 - **空串**是长度为0的串，用 ε (*epsilon*) 表示



串上的运算

- **连接(concatenation):** y 附加到 x 后形成的串记作 xy
 - 例如, 如果 $x=dog$ 且 $y=house$, 那么 $xy=doghouse$
 - 空串是连接运算的**单位元**, 即对于任何串 s 都有 $\varepsilon s = s\varepsilon = s$

- **幂运算**

$$\begin{cases} s^0 = \varepsilon, \\ s^n = s^{n-1}s, n \geq 1 \end{cases}$$

串 s 的 n 次幂: 将 n 个 s 连接起来

- $s^1 = s^0 s = \varepsilon s = s$, $s^2 = ss$, $s^3 = sss$, ...
- 例: 如果 $s=ba$, 那么 $s^1=ba$, $s^2=baba$, $s^3=bababa$, ...

形式语言

- **语言**：字母表 Σ 上的一个串集

- 例: $\{\varepsilon, 0, 00, 000, \dots\}$, $\{\varepsilon\}$, \emptyset

- **句子**：属于语言的串

- **语言的运算**

优先级：

幂>连接>并

运算	定义和表示
L 和 M 的并	$L \cup M = \{s \mid s \text{ 属于 } L \text{ 或者 } s \text{ 属于 } M\}$
L 和 M 的连接	$LM = \{st \mid s \text{ 属于 } L \text{ 且 } t \text{ 属于 } M\}$
L 的幂	$\begin{cases} L^0 = \{\varepsilon\} \\ L^n = L^{n-1}L, n \geq 1 \end{cases}$
L 的Kleene闭包	$L^* = \bigcup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+ = \bigcup_{i=1}^{\infty} L^i$

例: 形式语言及其运算

给定语言 $L=\{a,b\}$, $M=\{cc,dd\}$

并	$L \cup M = \{s \mid s \in L \text{ 或 } s \in M\}$	$L \cup M = \{ a, b, cc, dd \}$
连接	$LM = \{st \mid s \in L \text{ 且 } t \in M\}$	$LM = \{ acc, add, bcc, bdd \}$
幂	$L^0 = \{\epsilon\}$, $L^i = L^{i-1}L$	$L^1=L$, $L^2=LL=\{aa,ab,ba,bb\}$
闭包	$L^* = L^0 \cup L^1 \cup L^2 \cup \dots$?
正闭包	$L^+ = L^1 \cup L^2 \cup \dots$?

正则表达式(Regular Expression, RE)

- 正则表达式 r 定义正则语言, 记为 $L(r)$

- ε 是一个 RE, $L(\varepsilon) = \{\varepsilon\}$
- 如果 $a \in \Sigma$, 则 a 是一个 RE, $L(a) = \{a\}$
- 假设 r 和 s 都是 RE, 分别表示语言 $L(r)$ 和 $L(s)$
 - $r|s$ 是一个 RE, $L(r|s) = L(r) \cup L(s)$
 - rs 是一个 RE, $L(rs) = L(r) L(s)$
 - r^* 是一个 RE, $L(r^*) = (L(r))^*$ Kleene 闭包
 - (r) 是一个 RE, $L((r)) = L(r)$

$((a)(b)^*)|(c)$ 可以写成 $ab^*|c$

优先级:
闭包 $*$ > 连接 > 选择 $|$

正则表达式的一些定律

定律	描述
$r \mid s = s \mid r$	\mid 是可以交换的
$r \mid (s \mid t) = (r \mid s) \mid t$	\mid 是可结合的
$r (s \mid t) = (r s) \mid r t$	连接是可结合的
$r (s \mid t) = r s \mid r t ;$ $(s \mid t) r = s r \mid t r$	连接对 \mid 是可分配的
$\varepsilon r = r \varepsilon = r$	ε 是连接的单位元
$r^* = (r \mid \varepsilon)^*$	闭包中一定包含 ε
$r^{**} = r^*$	$*$ 具有幂等性

例: 正则表达式及其定义的语言

- 令 $\Sigma = \{a, b\}$, 以下RE定义了不同正则语言
 - $L(a|b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$
 - $L((a|b)(a|b)) = L(a|b) L(a|b) = \{a, b\} \{a, b\} = \{aa, ab, ba, bb\}$
 - $L(a^*) = (L(a))^* = \{a\}^* = \{\varepsilon, a, aa, aaa, \dots\}$ 任意长度的a
 - $L((a|b)^*) = (L(a|b))^* = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
任意长度的ab串

正则定义

- 正则定义:

- 对于比较复杂的语言，为了构造简洁的正则式，可先构造简单的正则式，再将这些正则式组合起来，形成一个与该语言匹配的正则序列
- 正则定义是具有如下形式的**定义序列**：

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

给一些**RE命名**，并在之后的**RE**中像使用字母表中的符号一样使用这些名字

1. 各个 d_i 的名字都不同
2. 每个 r_i 都是 $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ 上的正则表达式

例: 正则定义

- 整数的正则定义

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

可以简记为 **[0-9]**

number \rightarrow **digit digit***

另一种写法

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

number \rightarrow **digit⁺** (正闭包)

例: 正则定义

- C语言的标识符的正则定义

- C语言的标识符是字母、数字和下划线组成的串

$$digit \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$$
$$letter_ \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _$$
$$id \rightarrow letter_ (letter_ \mid digit)^*$$

- 可以简化为

$$letter_ \rightarrow [A-Za-z_]$$
$$digit \rightarrow [0-9]$$
$$id \rightarrow letter_ (letter_ \mid digit)^*$$

正则表达式 → 词法分析的规约(Specification)

词法分析: 字符流到Token-lexeme对

1. Select a set of tokens

- Number, Keyword, Identifier, ...

2. Write a R.E. for the lexemes of each token

- Number = **digit⁺**
- Keyword = **'if' | 'else' | ...**
- Identifier = **letter (letter | digit)***
- LeftPar = **'('**
- ...

正则规则的二义性

- 给定if8, 它是单个标识符 还是两个token if 和 8?

```
if
[a-z][a-z0-9]*
[0-9]+
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)
("--"[a-z]*"\n")|(" "|"\n"|"\\t")+

```

```
{return IF;}
{return ID;}
{return NUM;}
{return REAL;}
{ /*do nothing*/ }
{ error(); }
```

正则规则的二义性

最长匹配 Longest match:

- The longest initial substring of the input that can match any regular expression is taken as the next token.

规则优先 Rule priority:

- For a *particular* longest initial substring, the first regular expression that can match determines its token-type.
- This means that the order of writing down the regular-expression rules has significance.

Thus,

- `if8` matches as an identifier by the longest-match rule
- `if` matches as a reserved word by rule-priority.

3. 有穷自动机(Finite Automata)

如何判定一个串匹配某个正则表达式?
如何形式化地描述这个匹配过程?

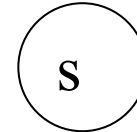
有穷自动机 (Finite Automata, FA)

- **有穷自动机**: $M = (S, \Sigma, move, s_0, F)$
 1. S : **有穷状态集**
 2. Σ : **输入符号集合/字母表**
 3. $move(s, a)$: **转换函数** , 表示从状态 s 出发 , 读入输入 a 时转化到的状态
 4. s_0 : **开始状态**(或初始状态) , $s_0 \in S$
 5. F : **接收状态** (或终止状态) 集合 , $F \subseteq S$

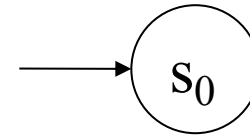
有穷自动机的表示: 转换图

- **转换图** (*Transition Graph*)

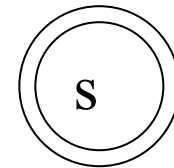
- 状态



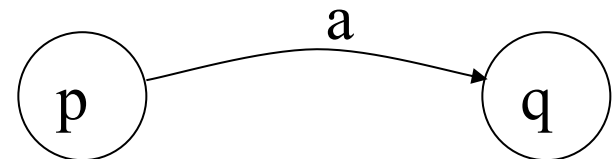
- 初始状态/开始状态



- 终止状态/接收状态(可以有多个)



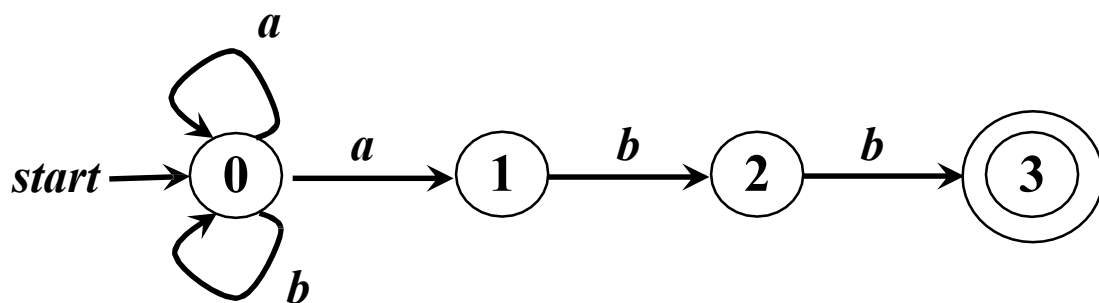
- 状态转换



如果对于**输入** a ，存在一个从状态 p 到状态 q 的转换，就在 p 、 q 之间画一条有向边，并标记上 a

有穷自动机的表示: 转换表

- 转换表



转换图

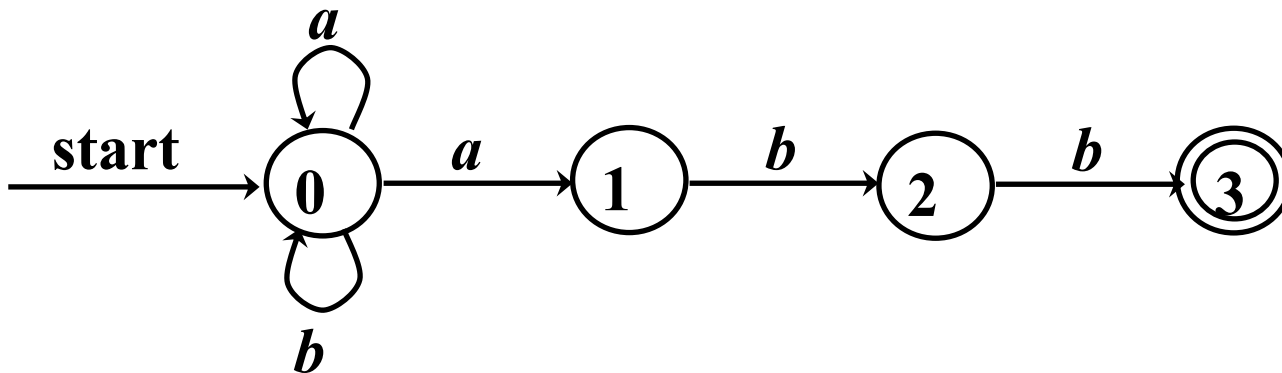
状态 \ 输入	输入	
	<i>a</i>	<i>b</i>
0	{ 0,1 }	{ 0 }
1	∅	{ 2 }
2	∅	{ 3 }
3 •	∅	∅

转换表

如果转换函数没有给出对应于某个状态-输入对的信息，就把 \emptyset 放入相应的表项中

有穷自动机接收的串

- 给定输入串 x ，如果存在一个对应于串 x 的从**初始状态**到**某个终止状态**的转换序列，则称**串 x** 被该**FA接收**



考虑输入: **ababb**

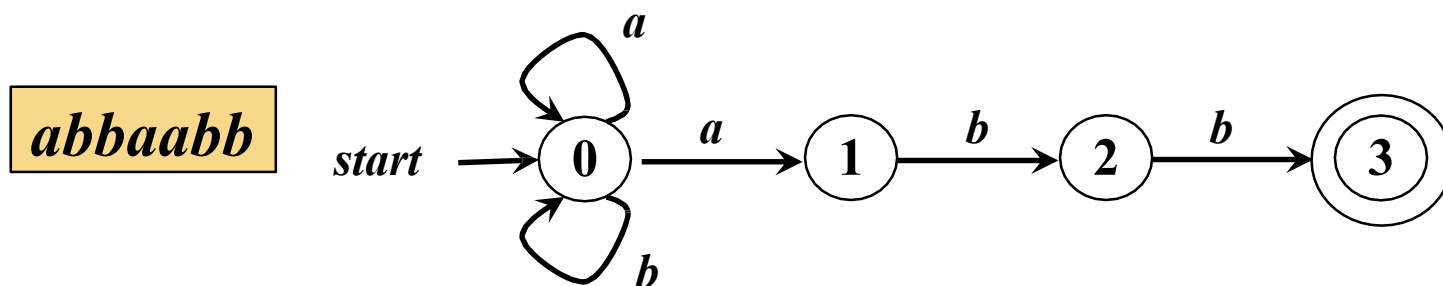
$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$ **ACCEPT !**

$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} ?$

有穷自动机接收（定义）的语言

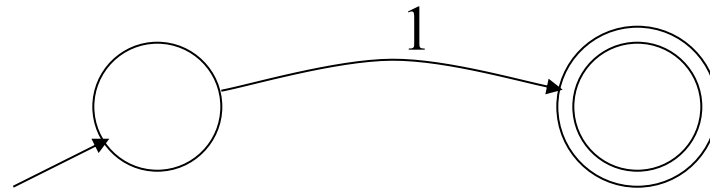
- 给定输入串 x ，如果存在一个对应于串 x 的从初始状态到某个终止状态的转换序列，则称串 x 被该FA接收
- 由一个有穷自动机 M 接收的所有串构成的集合，称为该FA接收（或定义）的语言，记为 $L(M)$



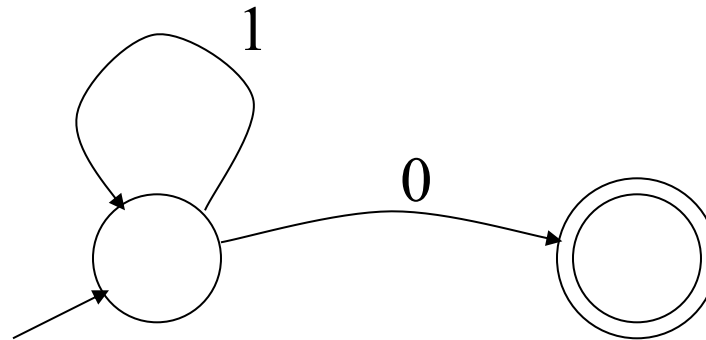
$L(M)$ = 所有以 abb 结尾的字母表 $\{a, b\}$ 上的串的集合

例: FA定义/接收的语言

- A finite automaton that accepts only “1”

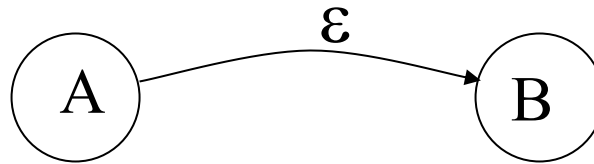


- A finite automaton accepting any number of 1's followed by a single 0

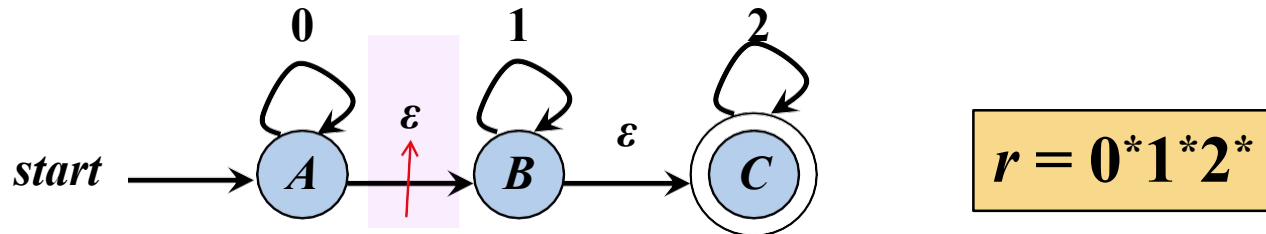


状态转换: Epsilon Moves

- **ϵ -moves:** 一种特殊的状态转换方式
 - 自动机可以不读入任何输入，而从状态A转移到状态B



- 例: 以下FA定义的语言是？



$$r = 0^*1^*2^*$$

不接受任何条件就能进入状态B

有穷自动机的分类

- **根据状态转换方式的不同:**
 - 非确定有穷自动机
(Nondeterministic finite automata, NFA)
 - 确定性有穷自动机
(Deterministic finite automata, DFA)

非确定有穷自动机 NFA

$$M = (S, \Sigma, move, s_0, F)$$

1. S : 有穷状态集
2. Σ : 输入符号集合, 即输入字母表。假设 ε 不是 Σ 中的元素
3. **$move: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$** 。 $move(s, a)$ 表示从状态 s 出发, 沿着标记为 a 的边所能到达的**状态集合**
4. s_0 : 开始状态(或初始状态), $s_0 \in S$
5. F : 接收状态 (或终止状态) 集合, $F \subseteq S$

- 在状态 s 时读入 a , 可能迁移到多个不同的状态
- 可能有 ε -moves (不读入任何输入而迁移到其他状态)

非确定有穷自动机 NFA

• 1976年图灵奖获得者：

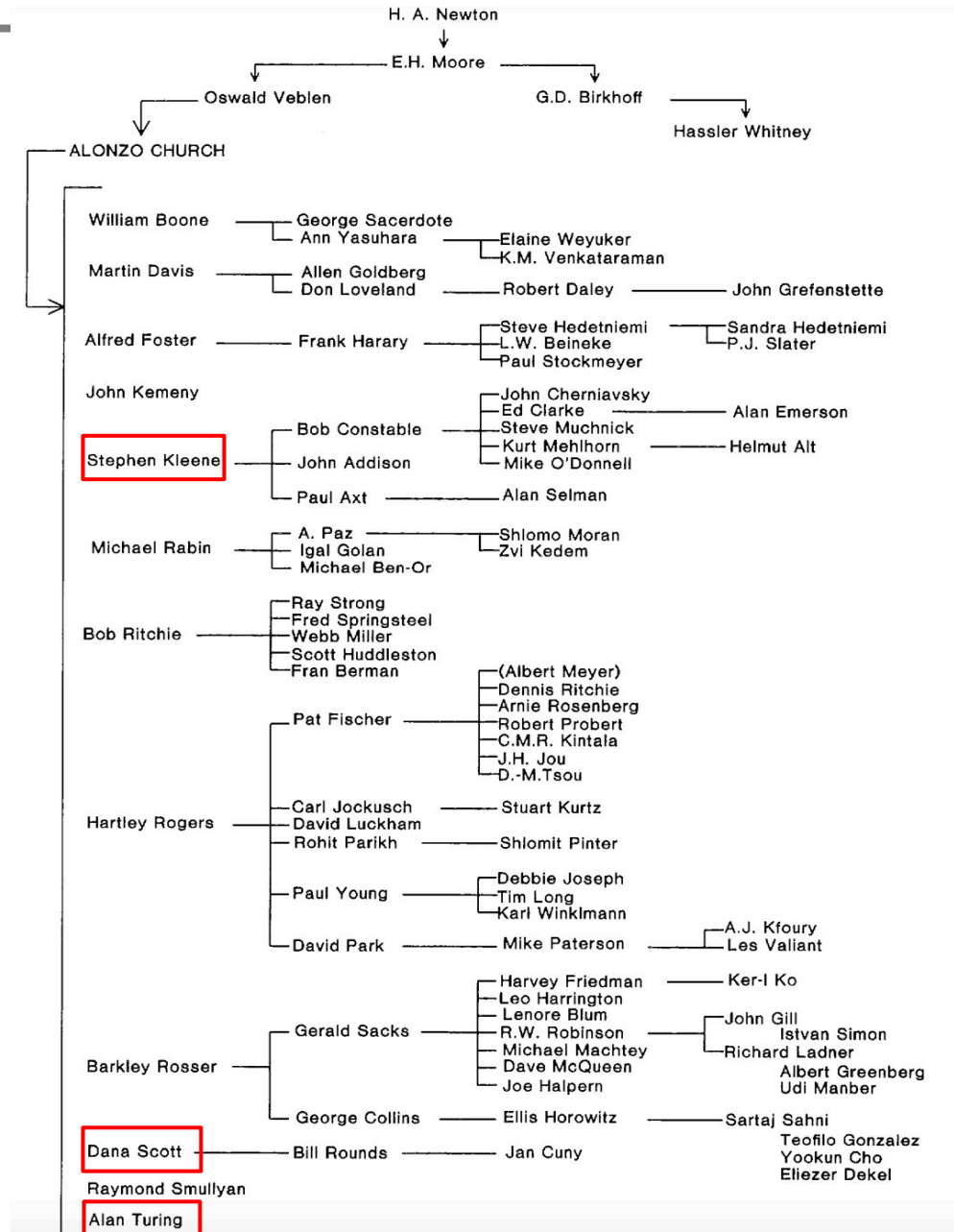
- **Michael O. Rabin** --- PhD, Princeton; Prof, Harvard
- **Dana S. Scott** --- PhD, Princeton; Prof, CMU



Michael O. Rabin



Dana S. Scott



确定性有穷自动机 DFA

$$M = (S, \Sigma, move, s_0, F)$$

1. S : 有穷状态集
2. Σ : 输入符号集合, 即输入字母表。假设 ε 不是 Σ 中的元素
3. **move: $S \times \Sigma \rightarrow S$** 。 $\delta(s,a)$ 表示从状态 s 出发, 沿着标记为 a 的边所能到达的**状态**
4. s_0 : 开始状态(或初始状态), $s_0 \in S$
5. F : 接收状态 (或终止状态) 集合, $F \subseteq S$

- 在状态 s 时读入 a , 可能迁移到的状态是**确定的**
- 没有 ε -moves

- **DFA和NFA主要差异在转换函数**

- DFA的 $move : S \times \Sigma \rightarrow S$

- NFA的 $move : S \times (\Sigma \cup \{\epsilon\}) \rightarrow P(S)$

- 在状态s时读入a, 可能迁移到多个不同的状态

- 可能有 ϵ -moves (不读入任何输入而迁移到其他状态)

- **DFA和NFA的等价性**

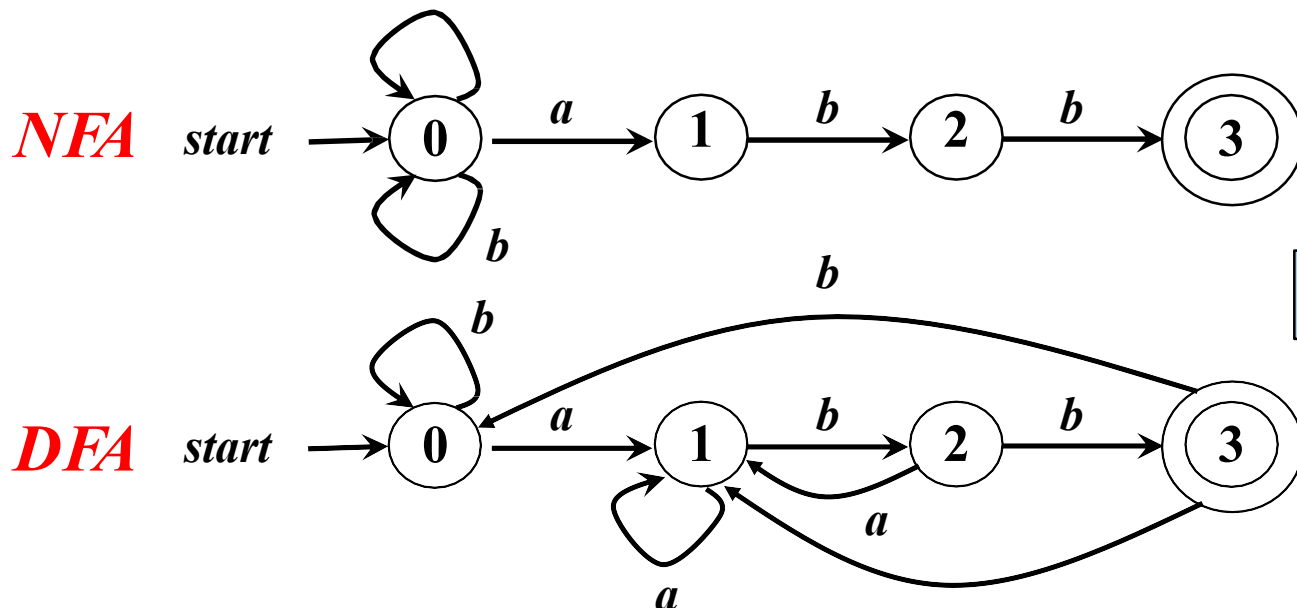
- 对任何NFA N , 存在定义同一语言的DFA D

- 对任何DFA D , 存在定义同一语言的NFA N

NFA vs. DFA vs. RE

- 正则表达式 \Leftrightarrow DFA \Leftrightarrow NFA

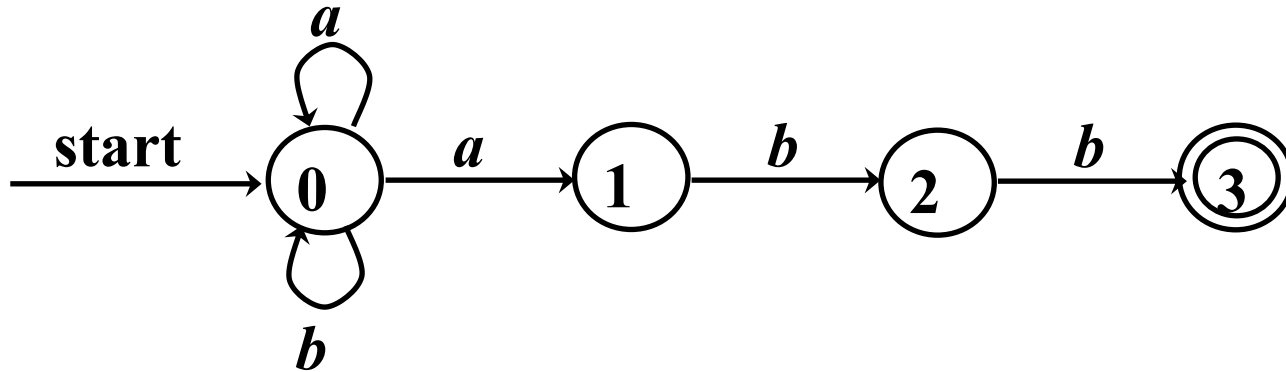
- 对任何NFA, 存在定义同一(正则)语言的DFA
- 对任何DFA, 存在定义同一(正则)语言的NFA



词法分析：如何构造FA，来识别用RE刻画(Token)？

回顾: 构造NFA识别字符串?

- 给定输入字符串 x ，如果存在一个对应于串 x 的从初始状态到某个终止状态的转换序列，则称串 x 被该FA接收



考虑输入: **ababb**

$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

$0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} ?$ **Failure**

NFA需对多种路径试探+失败回退，效率很低

构造DFA识别字符串

- 输入

以文件结束符eof结尾的字符串 x 。

DFA D : 开始状态 s_0 , 接收状态集 F , 转换函数 $move$

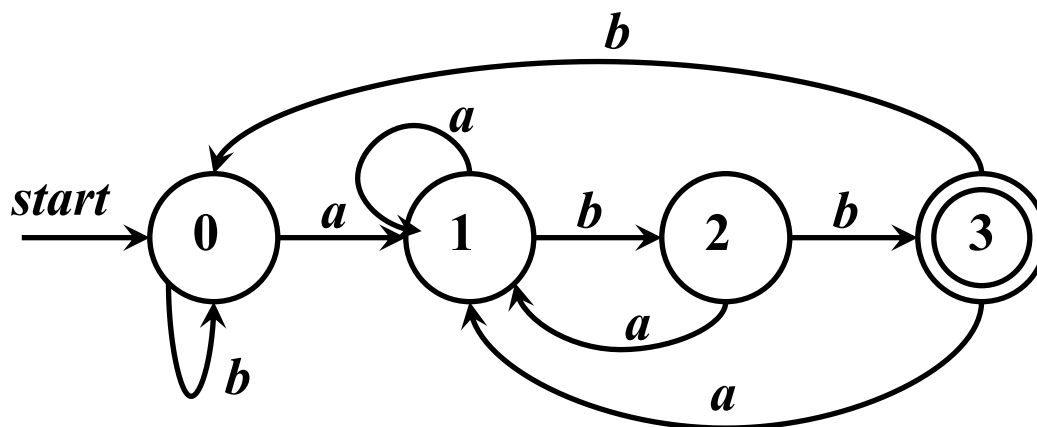
- 输出

如果 D 接收 x , 则回答 “yes” , 否则回答 “no”

```
 $s \leftarrow s_0; c \leftarrow nextChar();$   
while  $c \neq eof$  do  
     $s \leftarrow move(s, c);$   
     $c \leftarrow nextChar();$   
end;  
if  $s$  is in  $F$  then return “yes”  
else return “no”
```

- 函数 $nextChar()$ 返回 x 的下一个符号
- 函数 $move(s, c)$ 表示从状态 s 出发, 沿着标记为 c 的边所能到达的状态

例: 识别语言 $(a|b)^*abb$ 的DFA



例. Input: **ababb**

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

如何构造上面的DFA?

4. 词法分析器的自动生成

- 从正则表达式到自动机

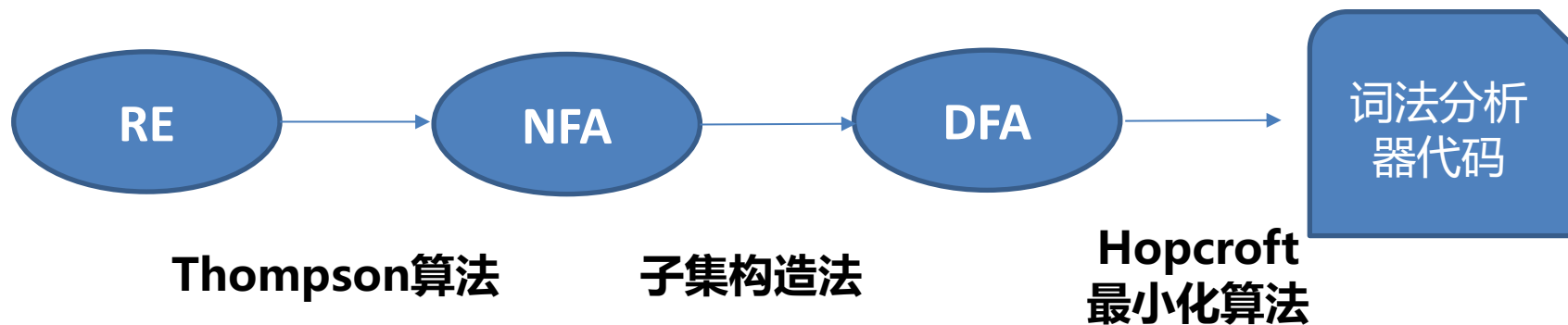
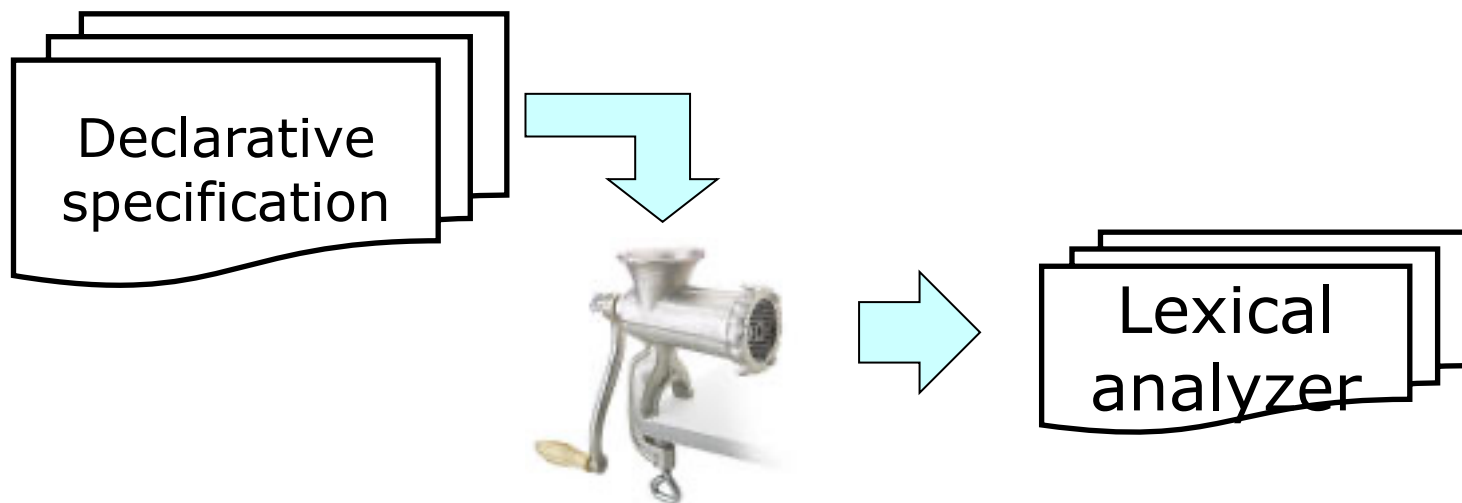
- RE → NFA

- NFA → DFA(子集构造法)

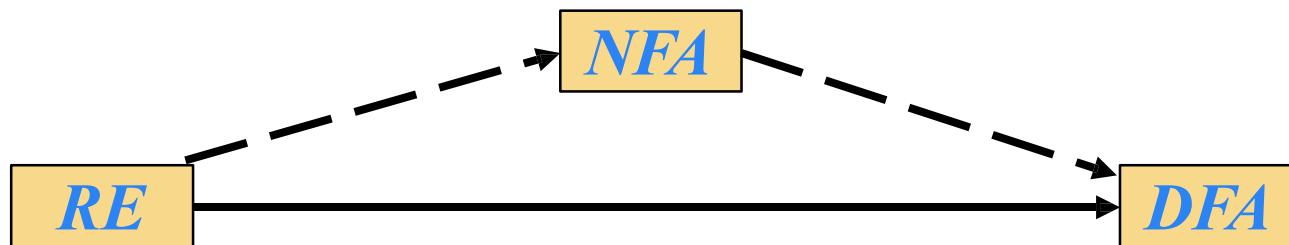
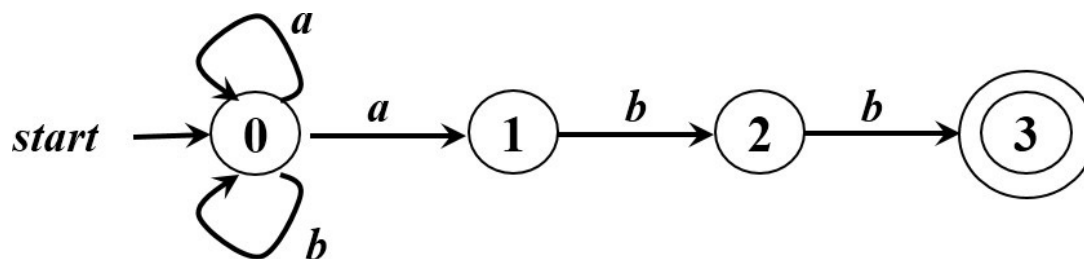
- DFA简化

给定RE，如何自动构造其DFA?

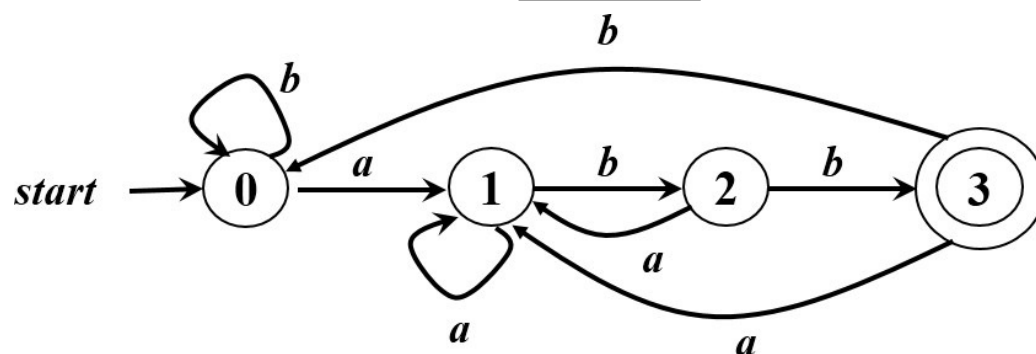
词法分析器的自动生成: RE \rightarrow DFA



词法分析器的自动生成: RE \rightarrow DFA



$r = (a|b)^*abb$

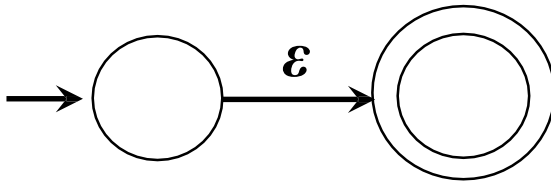


正则表达式到NFA

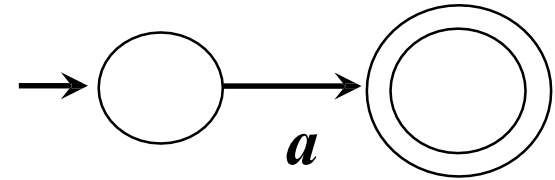
- **输入：正则表达式 r ；输出：定义它的NFA, 记为 $N(r)$**
- **Thompson算法：基于对RE的结构做归纳**
 - 对基本的RE直接构造: ϵ, a
 - 对复合的RE递归构造: $st, s \mid t, s^*$
 - 重要特点: $N(r)$ 仅一个接受状态，且没有出边

正则表达式到NFA: 处理 ϵ 和 a

- 直接构造: 识别 ϵ 和字母表中一个符号 a 的NFA
 - 重要特点: 仅一个接受状态, 接受状态没有出边



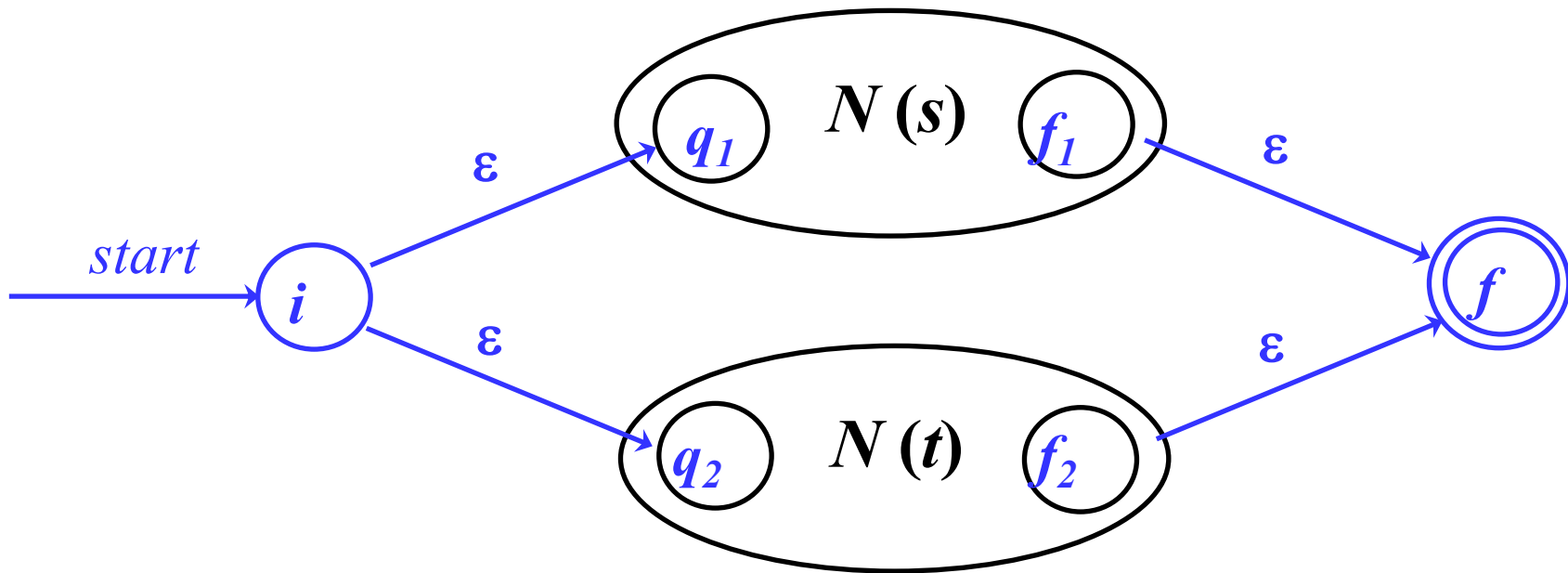
识别正则表达式 ϵ 的NFA



识别正则表达式 a 的NFA

正则表达式到NFA: 处理s|t

- 递归构造: **选择** $s \mid t$
 - 重要特点**: 仅一个接受状态, 接受状态没有出边

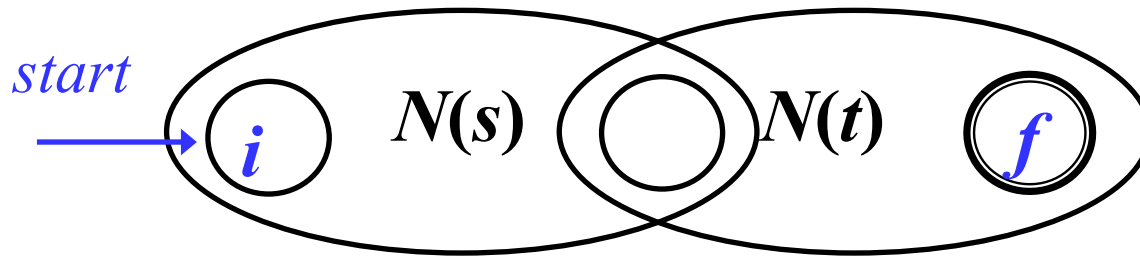


识别正则表达式 $s \mid t$ 的NFA

正则表达式到NFA: 处理st

- 递归构造: **连接** st
 - **重要特点**: 仅一个接受状态, 接受状态没有出边

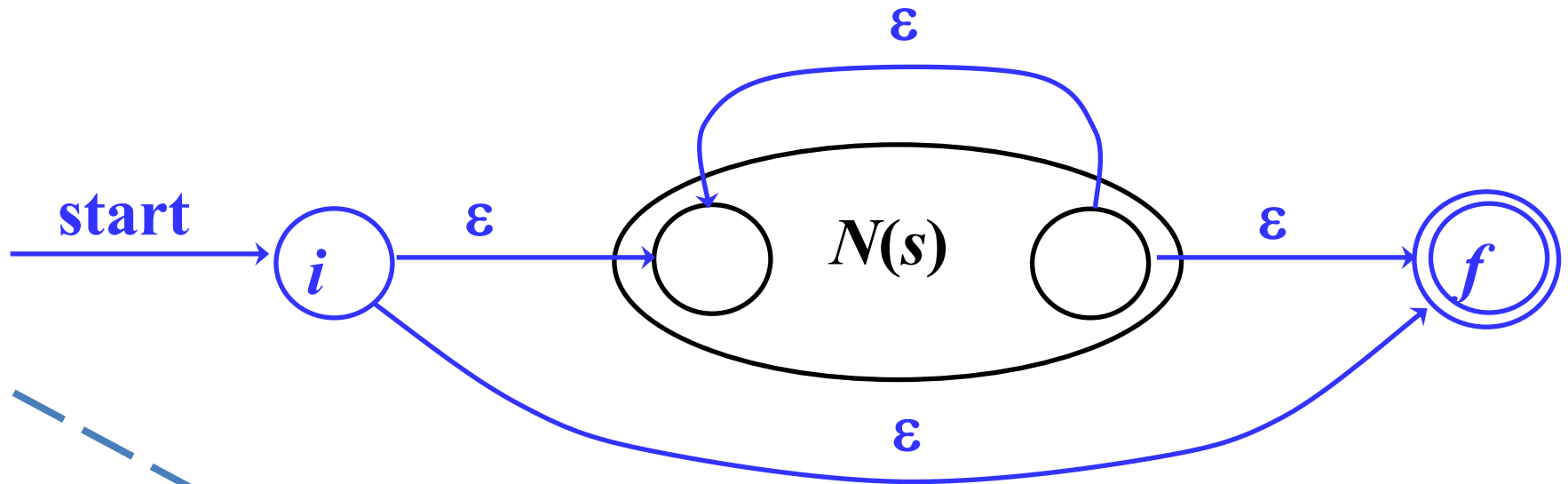
$N(s)$ 的接受状态和 $N(t)$ 的开始状态合并



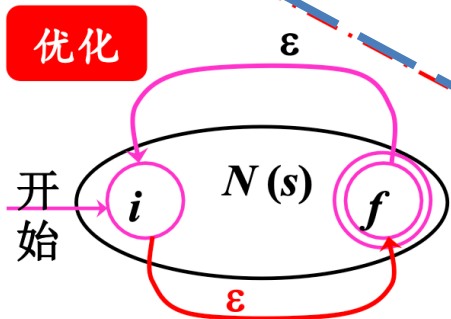
识别正则表达式 st 的NFA

正则表达式到NFA: 处理 s^*

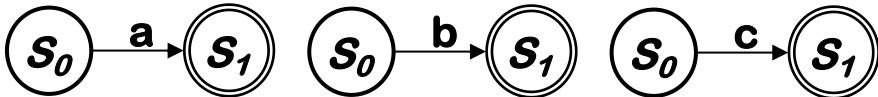
- 递归构造: **闭包** s^*
 - 重要特点**: 仅一个接受状态, 接受状态没有出边



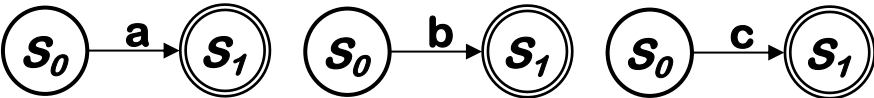
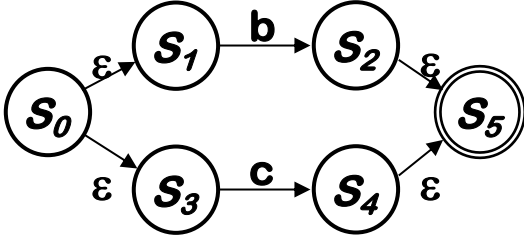
识别正则表达式 s^* 的NFA



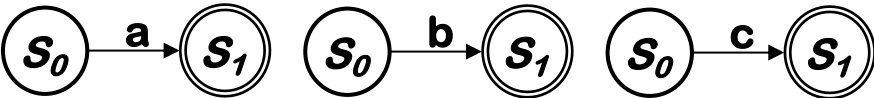
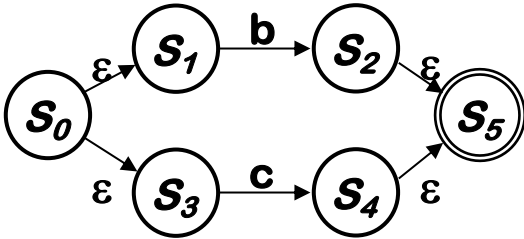
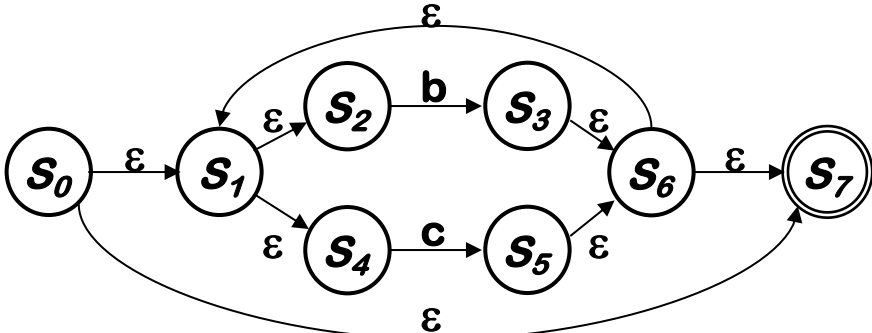
例: 正则表达式 $r=a(b|c)^*$ 到NFA

1.	a, b, c	
----	---------	--

例: 正则表达式 $r=a(b|c)^*$ 到NFA

1.	a, b, c	
2.	$b \mid c$	

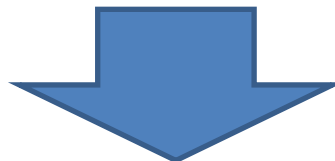
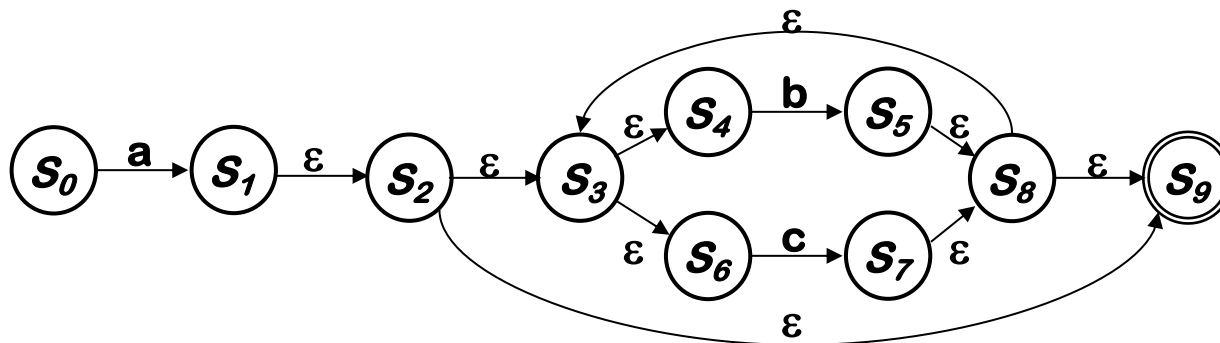
例: 正则表达式 $r=a(b|c)^*$ 到NFA

1.	a, b, c	
2.	$b \mid c$	
3.	$(b \mid c)^*$	

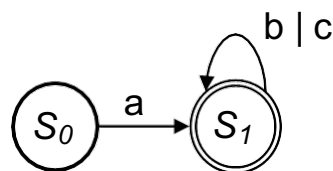
例: 正则表达式 $r=a(b|c)^*$ 到NFA

1.	a, b, c	
2.	$b \mid c$	
3.	$(b \mid c)^*$	
4.	$a(b \mid c)^*$	

例: 正则表达式 $r=a(b|c)^*$ 到NFA



We could do a bit better (如果人工构造的话) ☺



4. 词法分析器的自动生成

- 从正则表达式到自动机

- $RE \rightarrow NFA$

- $NFA \rightarrow DFA$ (子集构造法)

- DFA简化

给定RE，如何自动构造其DFA?

NFA到DFA的转换

- **子集构造法(subset construction)原则**
 - DFA的每个状态是NFA的**状态集合的一个子集**
 - 读了输入 a_i 后NFA能到达的所有状态： s_1, s_2, \dots, s_k ，则DFA到达一个状态，对应于NFA的 $\{s_1, s_2, \dots, s_k\}$
- **定义NFA状态(集)上的一些操作**

操作	描述
$\varepsilon\text{-closure}(s)$	NFA状态 s 的 ε-闭包 : s 经 ε 转换所能到达的状态集合
$\varepsilon\text{-closure}(T)$	T 中所有状态的 ε-闭包的并集 , 即 $\bigcup_{s \in T} \varepsilon\text{-closure}(s)$

NFA到DFA的转换

- 子集构造法(subset construction)过程

1. NFA的初始状态的 ϵ -闭包对应于DFA的初始状态

2. 针对每个DFA状态(对应NFA状态子集A),

求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集

$$S = \epsilon\text{-closure}(\text{move}(A, a_i))$$

NFA从状态集A出发, 读入 a_i 后能到达的状态集合

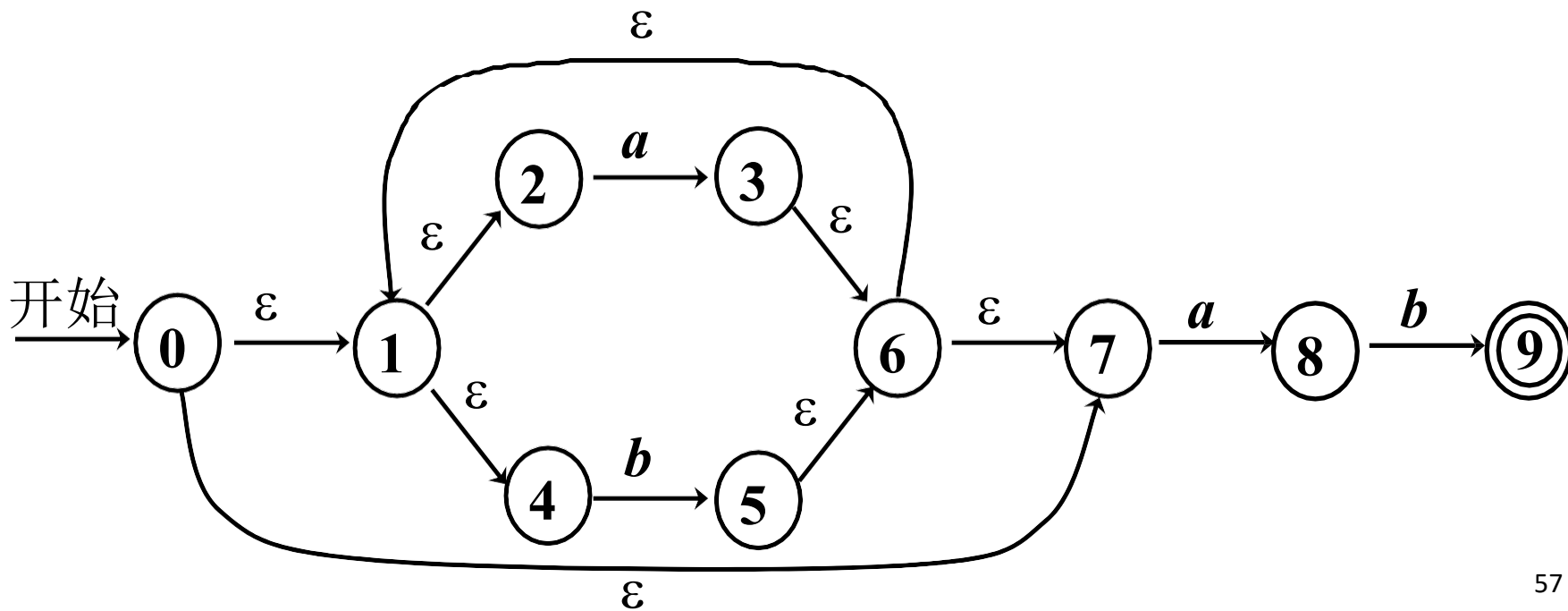
该集合S

- 要么对应于DFA中的一个已有状态
- 要么是一个要新加的DFA状态

逐步构造DFA的状态转换表, 直到不动点

例: NFA到DFA的变换

- $(a|b)^*ab$ 对应的NFA如下，把它变换为DFA

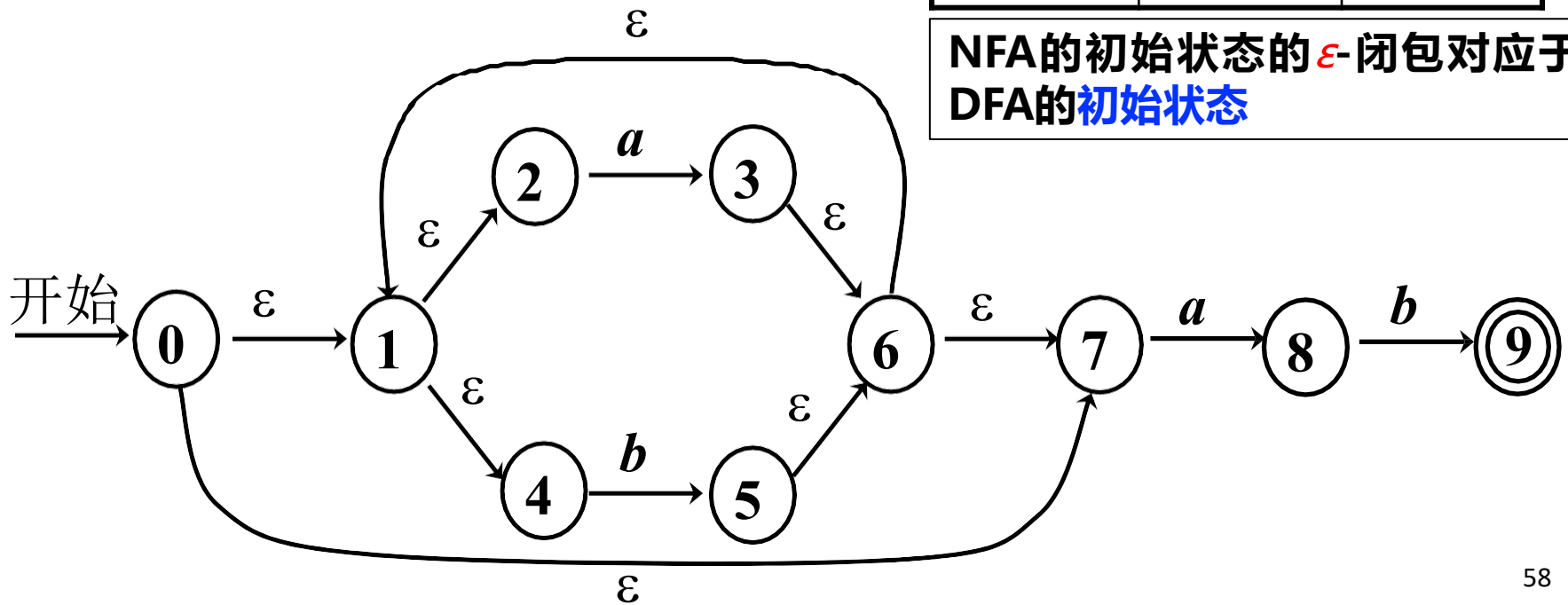


例: NFA到DFA的变换

DFA的状态转换表

状态	输入符号	
	<i>a</i>	<i>b</i>

NFA的初始状态的 ϵ -闭包对应于DFA的初始状态

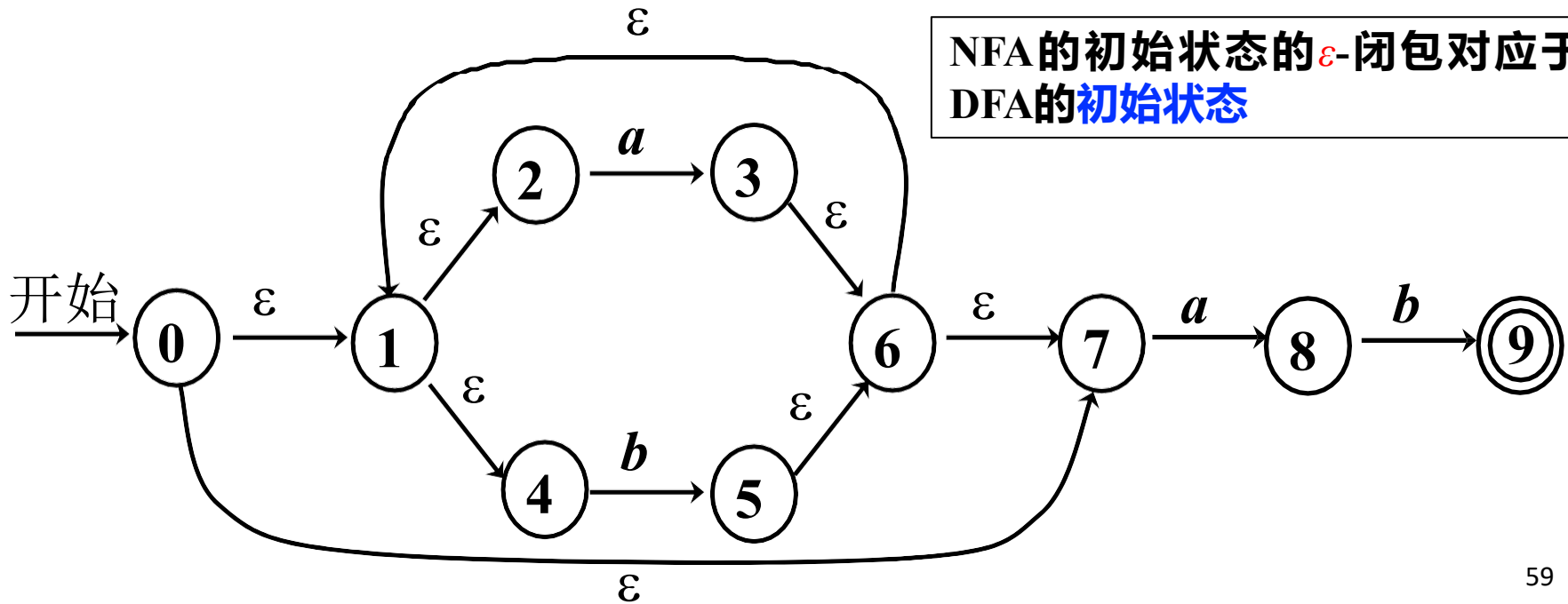


例: 计算 ϵ -closure(0)

$$A = \{\mathbf{0}, 1, 2, 4, 7\}$$

识别DFA初始状态

状态	输入符号	
	<i>a</i>	<i>b</i>
A		



例: 计算 ϵ -closure(move(A,a))

$A = \{0, 1, 2, 4, 7\}$

$B = \{1, 2, \textcolor{red}{3}, 4, 6, 7, \textcolor{red}{8}\}$

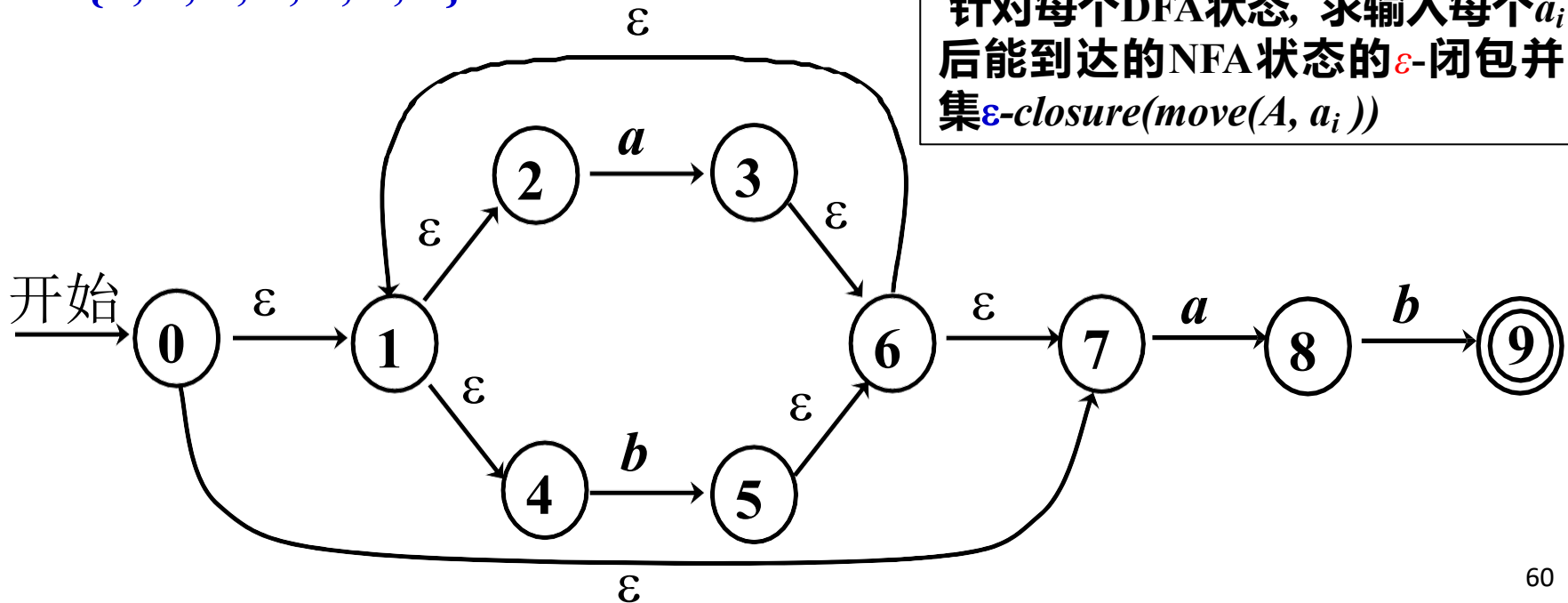
ϵ -closure (move ($\{0,1,2,4,7\},a$))

$= \epsilon$ -closure($\{3,8\}$)

$= \{1, 2, 3, 4, 6, 7, 8\}$

状态	输入符号	
	a	b
A	$\textcolor{red}{B}$	

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 ϵ -closure(move(A, a_i))



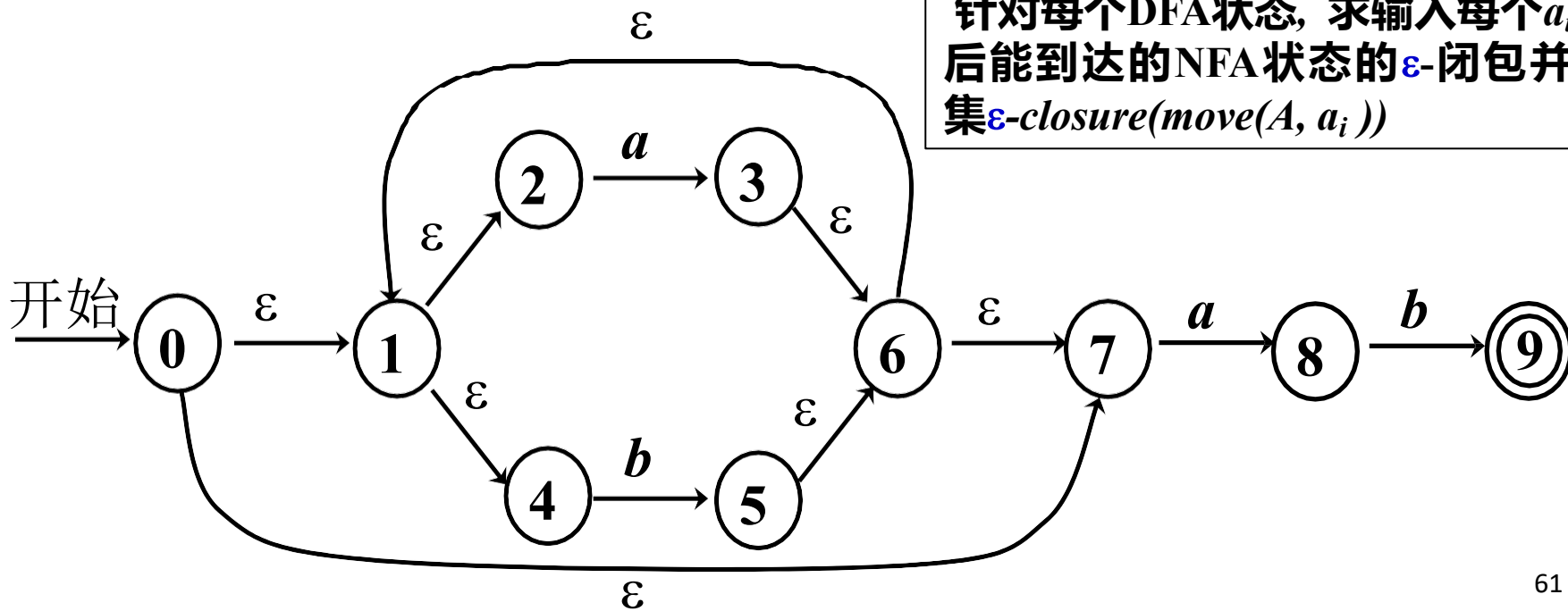
例: 更新状态转换表

$A = \{\mathbf{0}, 1, 2, 4, 7\}$

$B = \{1, 2, \mathbf{3}, 4, 6, 7, \mathbf{8}\}$

状态	输入符号	
	a	b
A	B	
B		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 $\epsilon\text{-closure}(\text{move}(A, a_i))$



例: 计算 ϵ -closure(move(A,b))

$A = \{0, 1, 2, 4, 7\}$

$B = \{1, 2, 3, 4, 6, 7, 8\}$

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

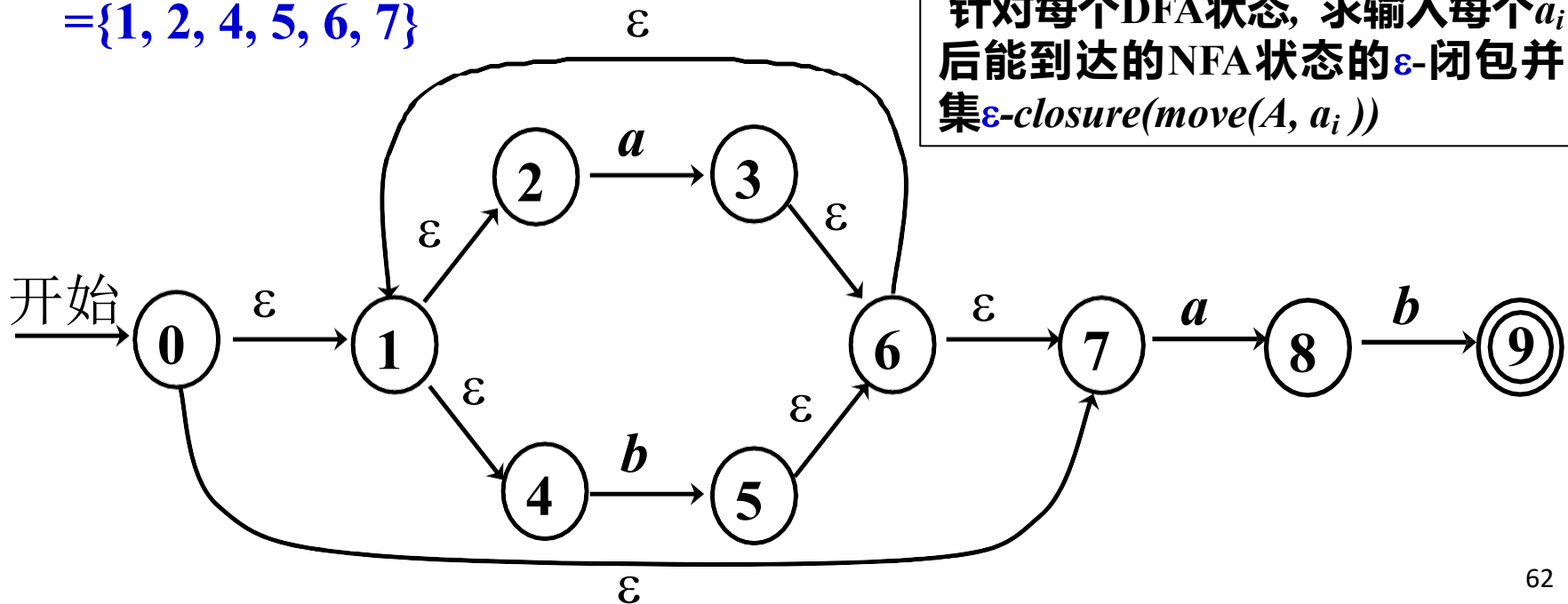
ϵ -closure (move ({0,1,2,4,7},b))

= ϵ -closure({5})

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

状态	输入符号	
	a	b
A	B	C
B		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 ϵ -closure(move(A, a_i))



例: 计算 ϵ -closure(move(B,a))

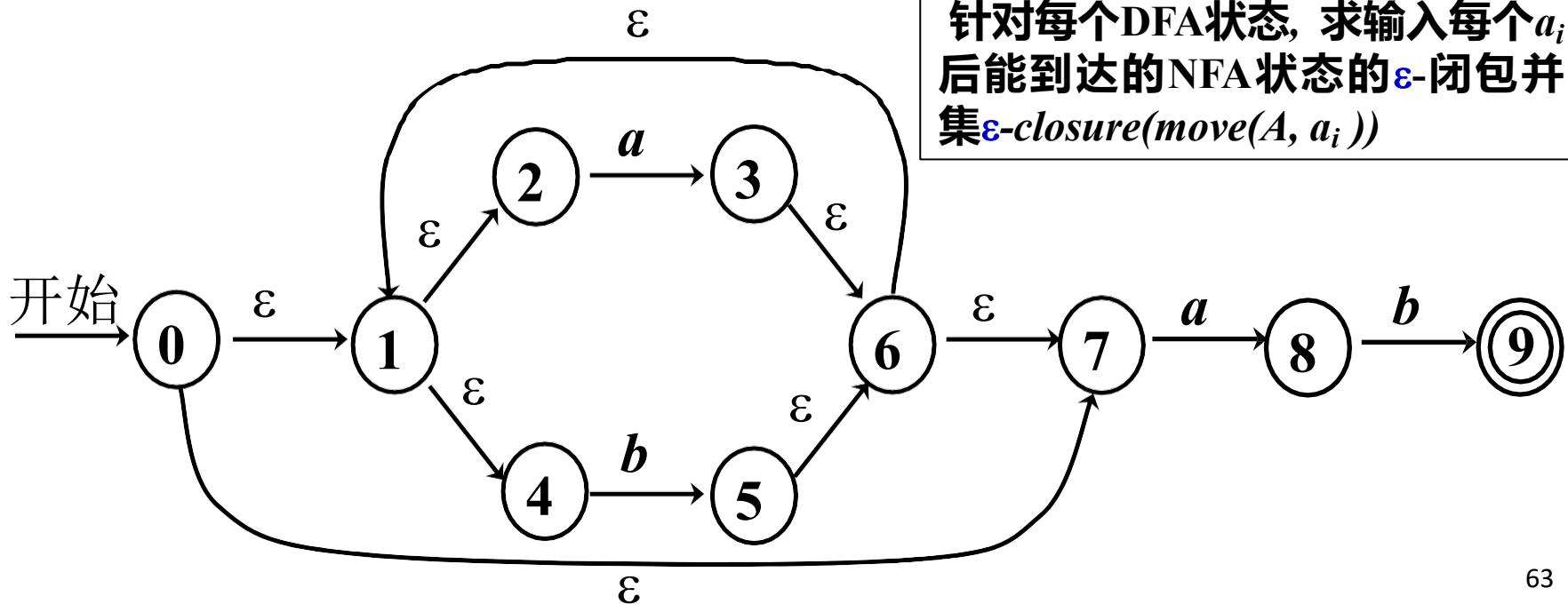
$A = \{0, 1, 2, 4, 7\}$

$B = \{1, 2, \textcolor{red}{3}, 4, 6, 7, \textcolor{red}{8}\}$

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

状态	输入符号	
	a	b
A	B	C
B		
$\textcolor{red}{C}$		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 $\epsilon\text{-closure}(\text{move}(A, a_i))$



例: 更新状态转换表

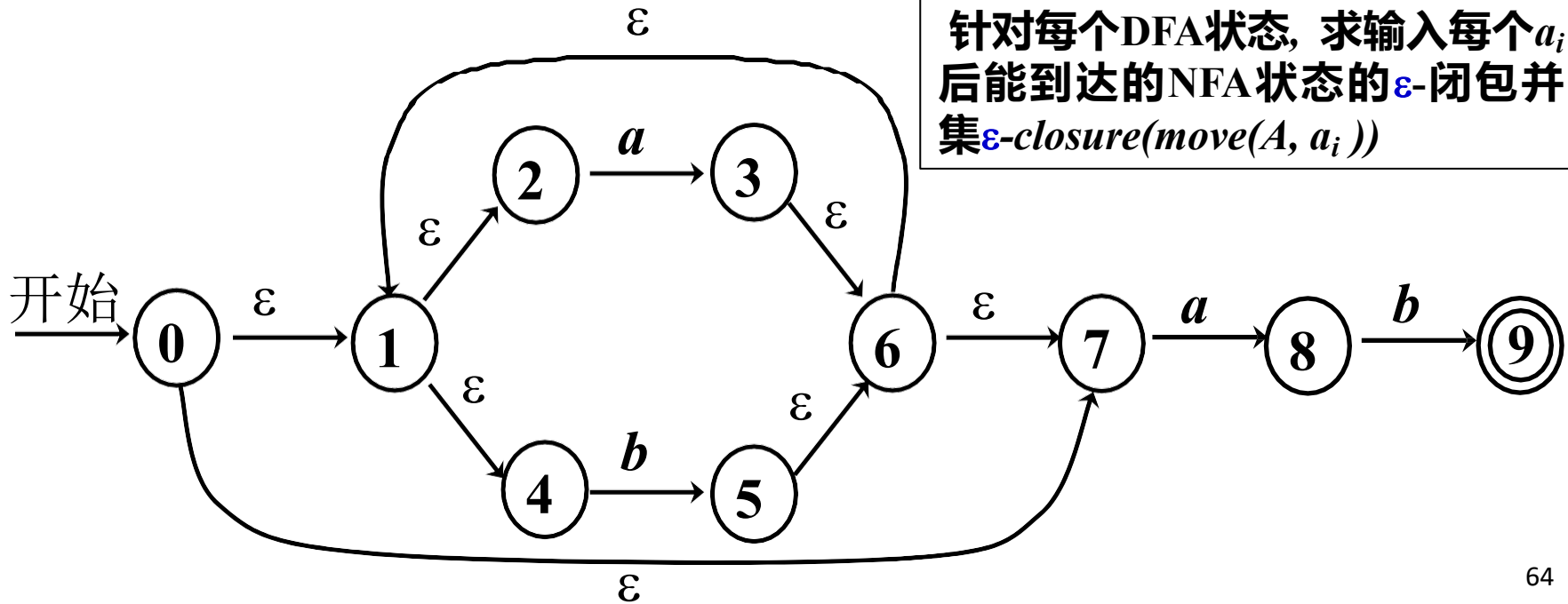
$A = \{0, 1, 2, 4, 7\}$

$B = \{1, 2, 3, 4, 6, 7, 8\}$

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

状态	输入符号	
	a	b
A	B	C
B	B	
C		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集
 $\epsilon\text{-closure}(\text{move}(A, a_i))$



例: 计算 ϵ -closure(move(B,b))

$A = \{\mathbf{0}, 1, 2, 4, 7\}$

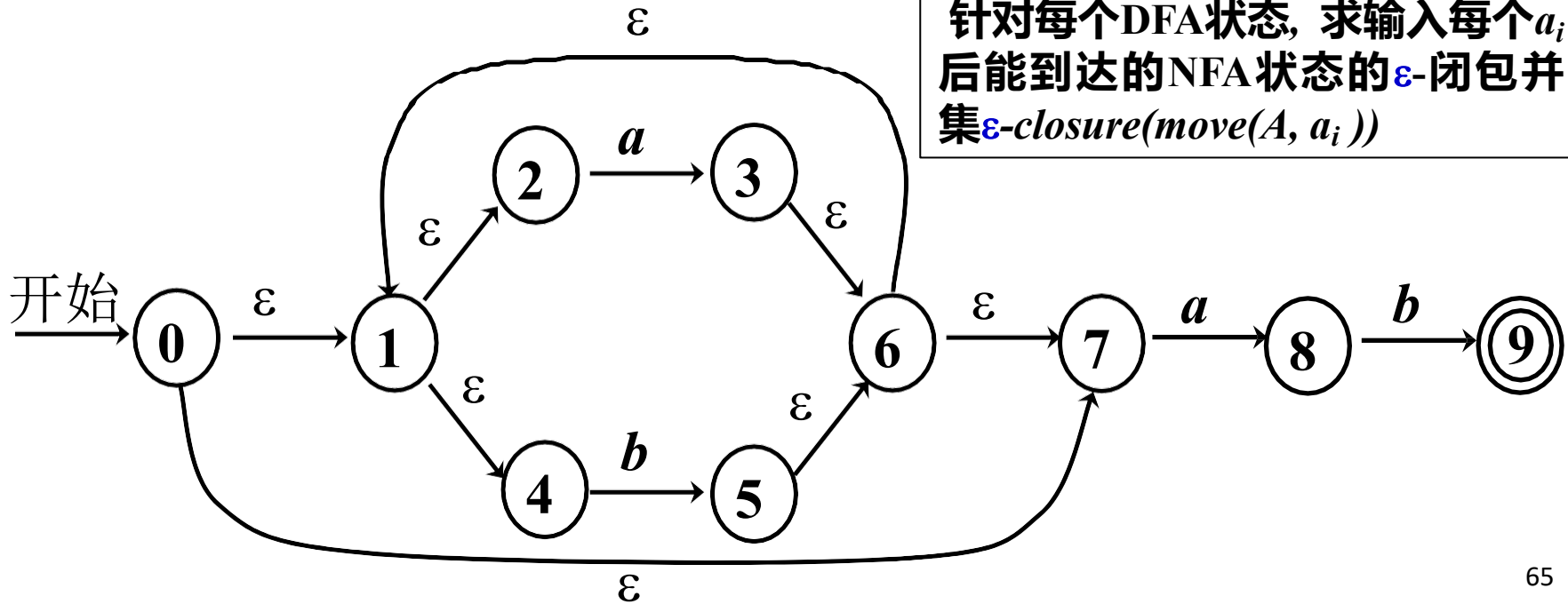
$B = \{1, 2, \mathbf{3}, 4, 6, 7, \mathbf{8}\}$

$C = \{1, 2, 4, \mathbf{5}, 6, 7\}$

$D = \{1, 2, 4, \mathbf{5}, 6, 7, \mathbf{9}\}$

状态	输入符号	
	a	b
A	B	C
B	B	\mathbf{D}
C		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 ϵ -closure(move(A, a_i))



例: 更新状态转换表

$A = \{\mathbf{0}, 1, 2, 4, 7\}$

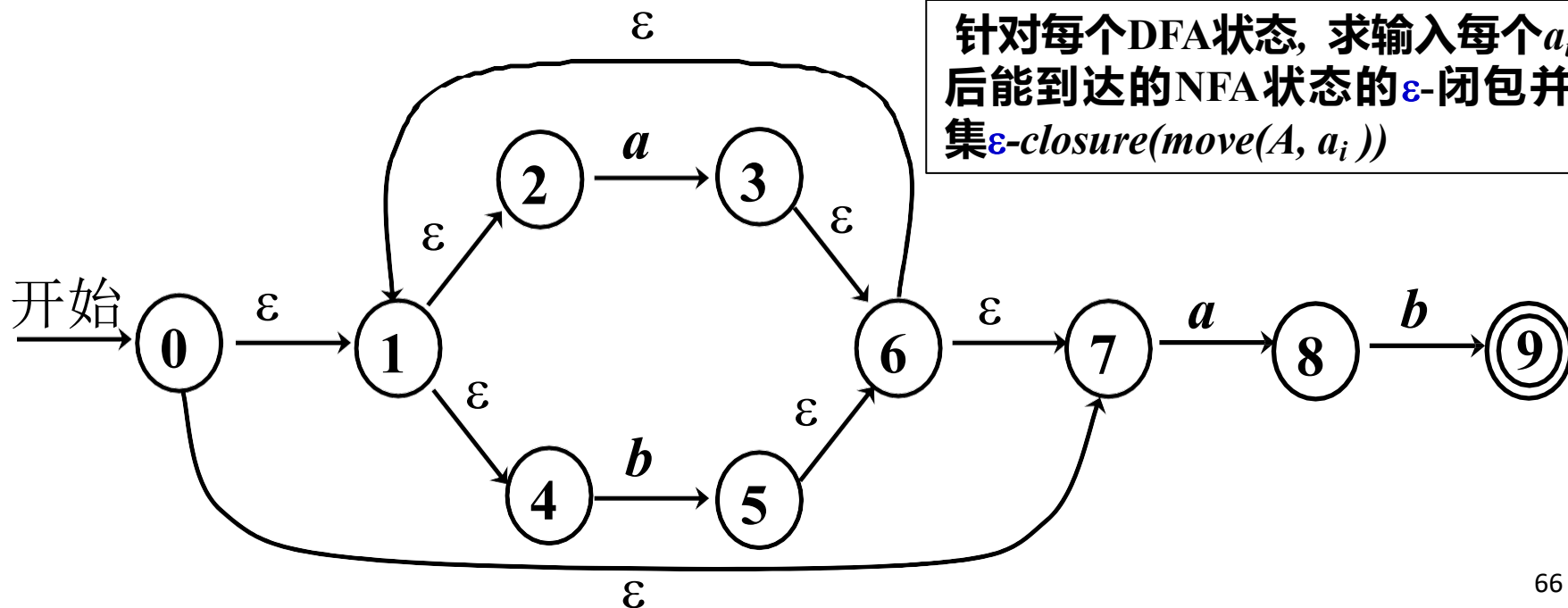
$B = \{1, 2, \mathbf{3}, 4, 6, 7, \mathbf{8}\}$

$C = \{1, 2, 4, \mathbf{5}, 6, 7\}$

$D = \{1, 2, 4, \mathbf{5}, 6, 7, \mathbf{9}\}$

状态	输入符号	
	a	b
A	B	C
B	B	D
C		
D		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 $\epsilon\text{-closure}(\text{move}(A, a_i))$



例: 计算C的状态转换

$A = \{\mathbf{0}, 1, 2, 4, 7\}$

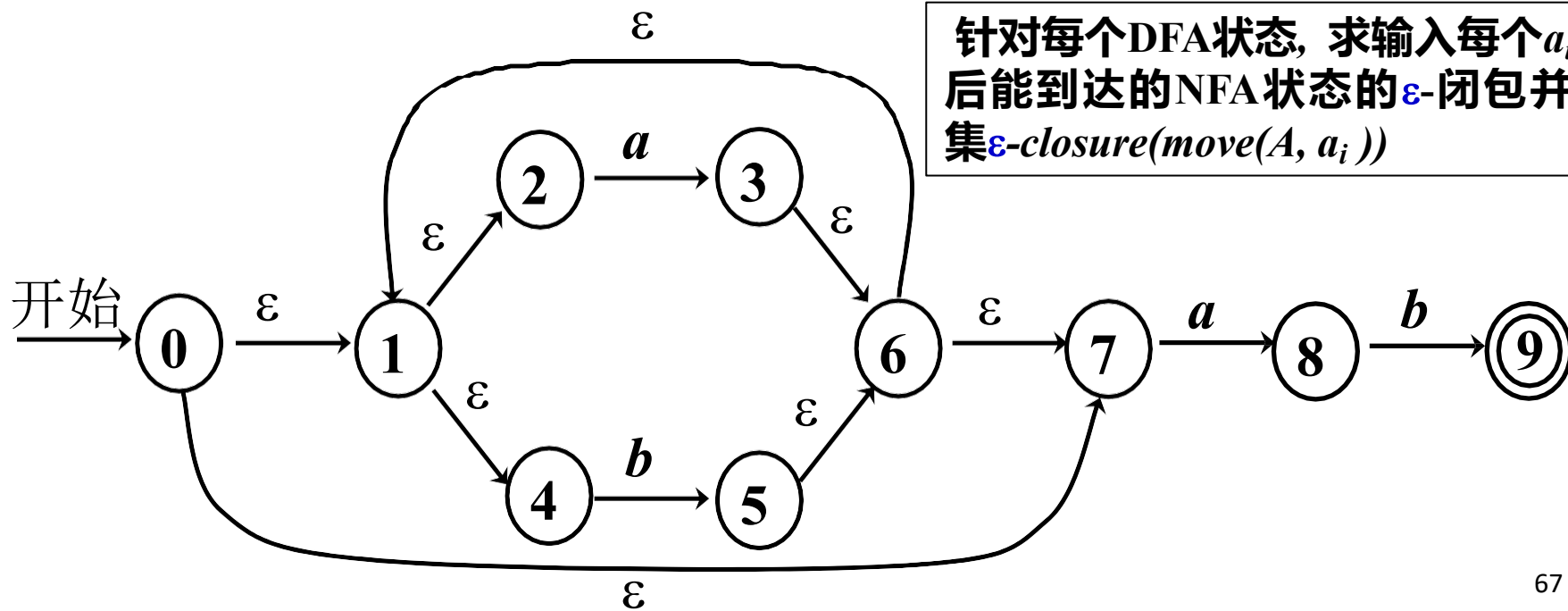
$B = \{1, 2, \mathbf{3}, 4, 6, 7, \mathbf{8}\}$

$C = \{1, 2, 4, \mathbf{5}, 6, 7\}$

$D = \{1, 2, 4, \mathbf{5}, 6, 7, \mathbf{9}\}$

状态	输入符号	
	a	b
A	B	C
B	B	D
C	\mathbf{B}	\mathbf{C}
D		

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 $\epsilon\text{-closure}(\text{move}(A, a_i))$



例: 计算D的状态转换

$A = \{0, 1, 2, 4, 7\}$

$B = \{1, 2, 3, 4, 6, 7, 8\}$

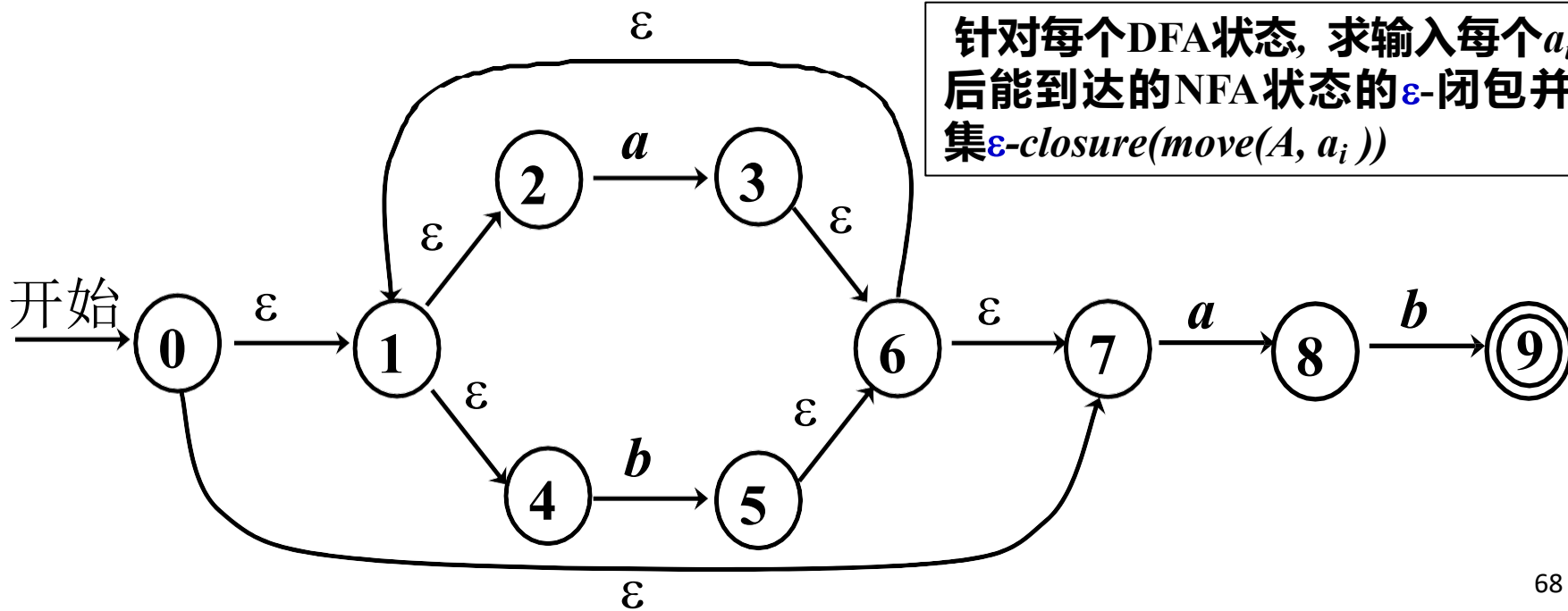
$C = \{1, 2, 4, 5, 6, 7\}$

$D = \{1, 2, 4, 5, 6, 7, 9\}$

算法到达了“不动点”

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	B	C

针对每个DFA状态, 求输入每个 a_i 后能到达的NFA状态的 ϵ -闭包并集 $\epsilon\text{-closure}(\text{move}(A, a_i))$



例: 确定DFA的终止状态

$A = \{\textcolor{red}{0}, 1, 2, 4, 7\}$

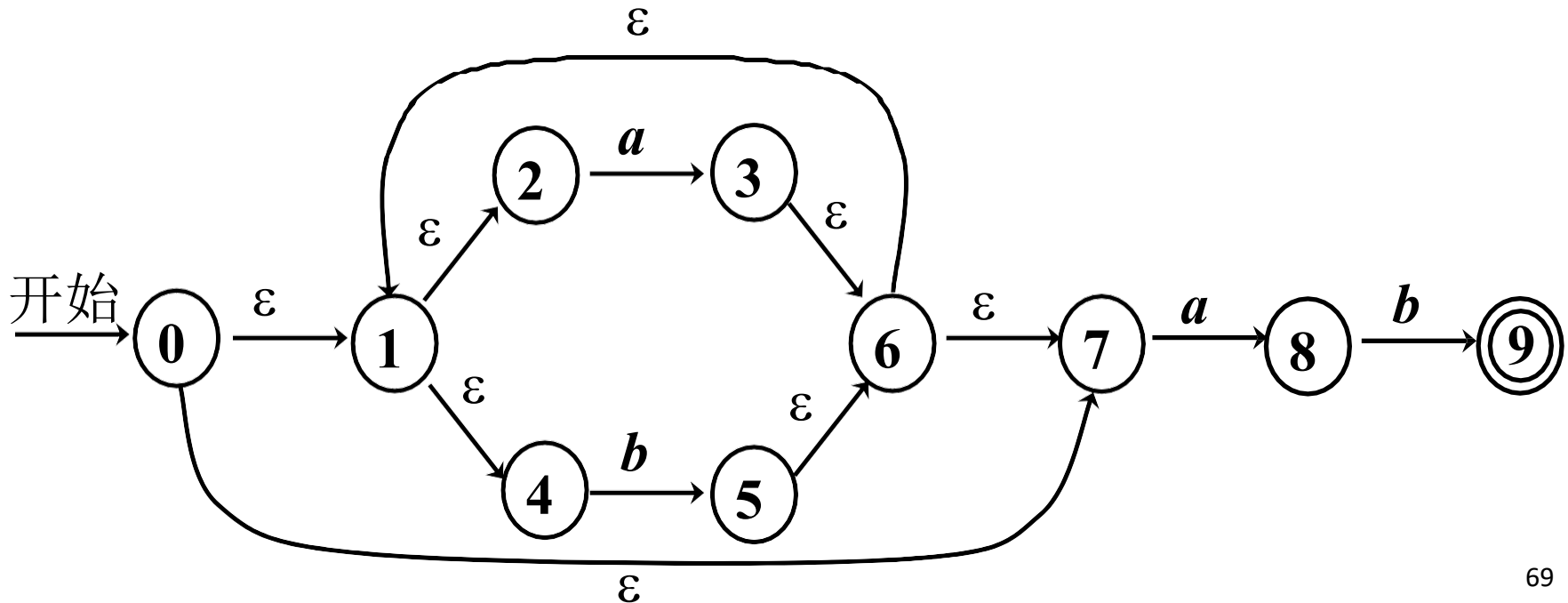
$B = \{1, 2, \textcolor{red}{3}, 4, 6, 7, \textcolor{red}{8}\}$

$C = \{1, 2, 4, \textcolor{red}{5}, 6, 7\}$

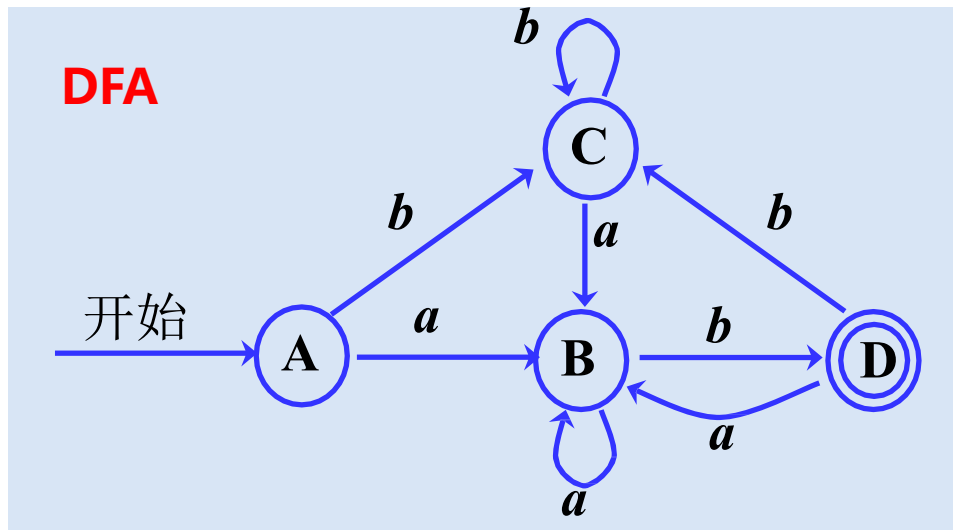
$D = \{1, 2, 4, \textcolor{red}{5}, 6, 7, \textcolor{red}{9}\}$

The DFA's final states are the sets of states that contain at least one final state from the NFA

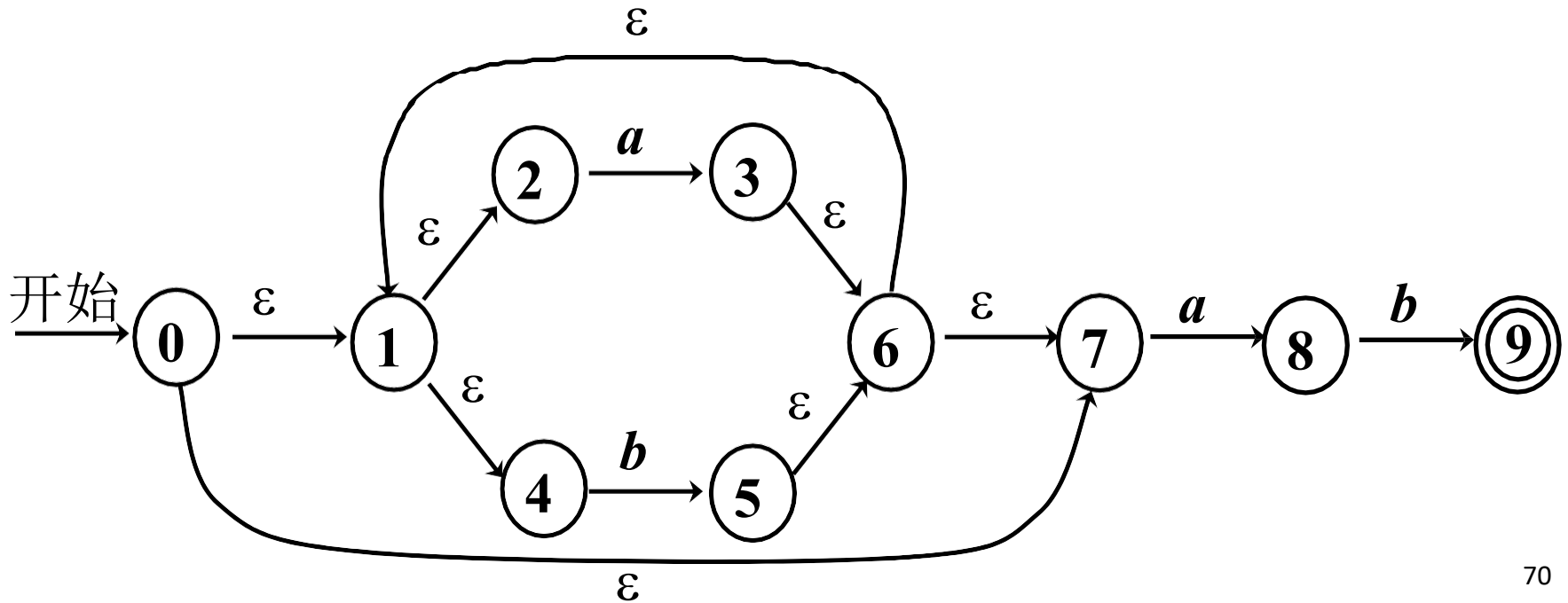
状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	$\textcolor{red}{B}$	$\textcolor{red}{C}$



例: DFA转换表 → DFA转换图

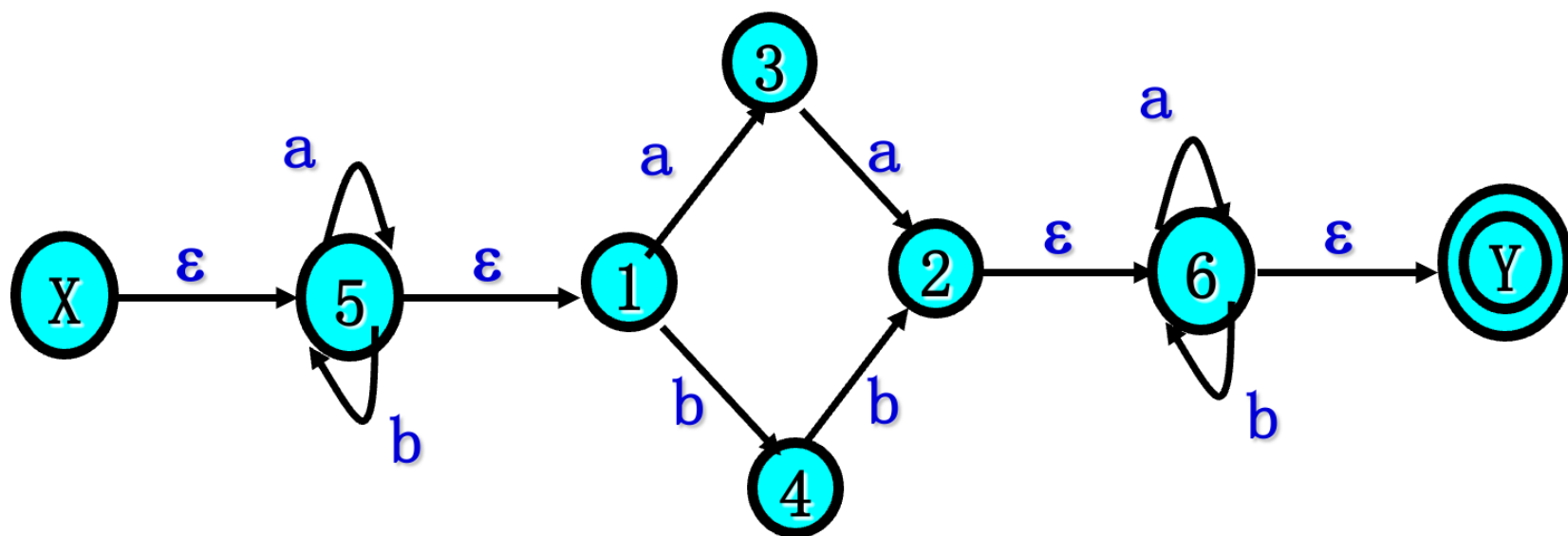


状态	输入符号	
	<i>a</i>	<i>b</i>
<i>A</i>	<i>B</i>	<i>C</i>
<i>B</i>	<i>B</i>	<i>D</i>
<i>C</i>	<i>B</i>	<i>C</i>
<i>D</i>	<i>B</i>	<i>C</i>



课后练习: NFA \rightarrow DFA转换

- 将以下NFA转成DFA



4. 词法分析器的自动生成

- 从正则表达式到自动机

- RE \rightarrow NFA

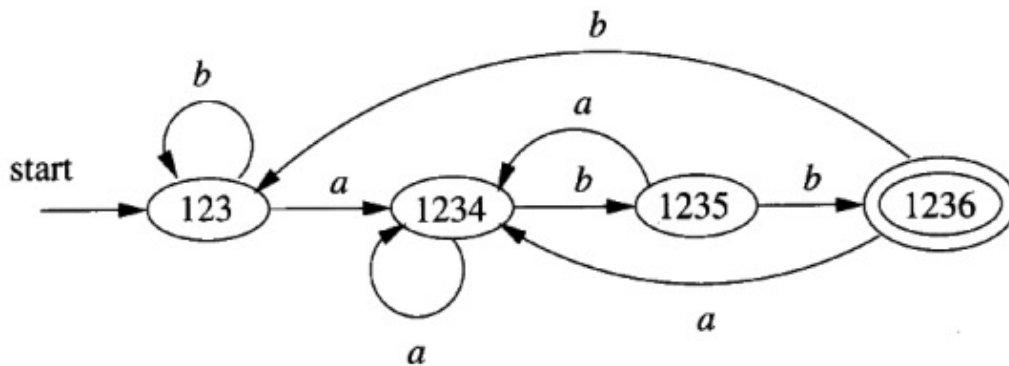
- NFA \rightarrow DFA(子集构造法)

- DFA简化

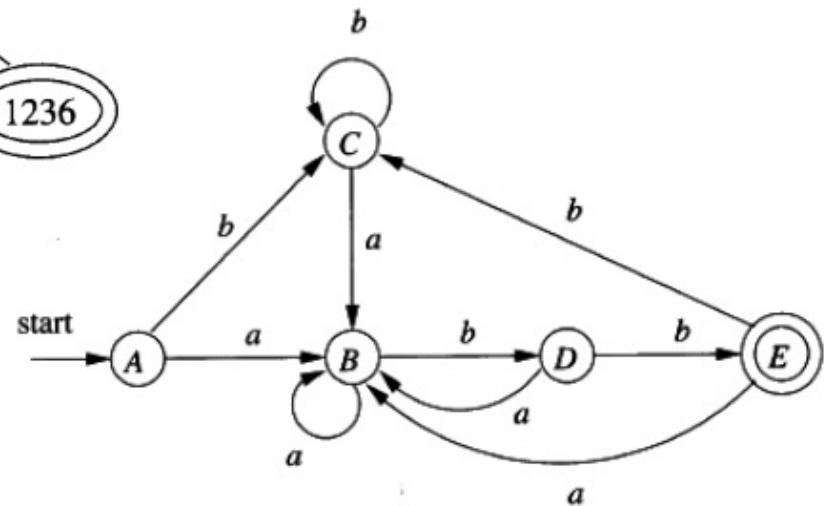
给定RE，如何自动构造其DFA?

DFA状态数量的最小化

- 一个正则语言可对应于多个识别此语言的DFA
- 通过DFA的最小化可得到**状态数量最少**的DFA (不计同构，这样的DFA是**唯一**的)



两个等价的DFA：都识别
 $(a|b)^*abb$



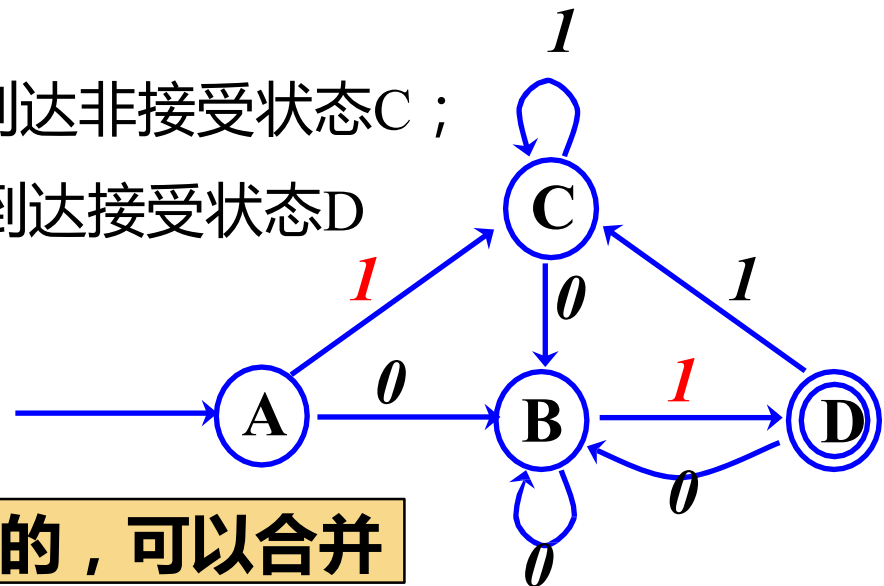
可区分的状态(Distinguishable States)

- 可区分的状态**

- 如果存在串 x ，使得从 s 、 t 出发，一个到达接受状态，一个到达非接受状态，那么 x 就区分了 s 和 t

例：考虑下图的DFA

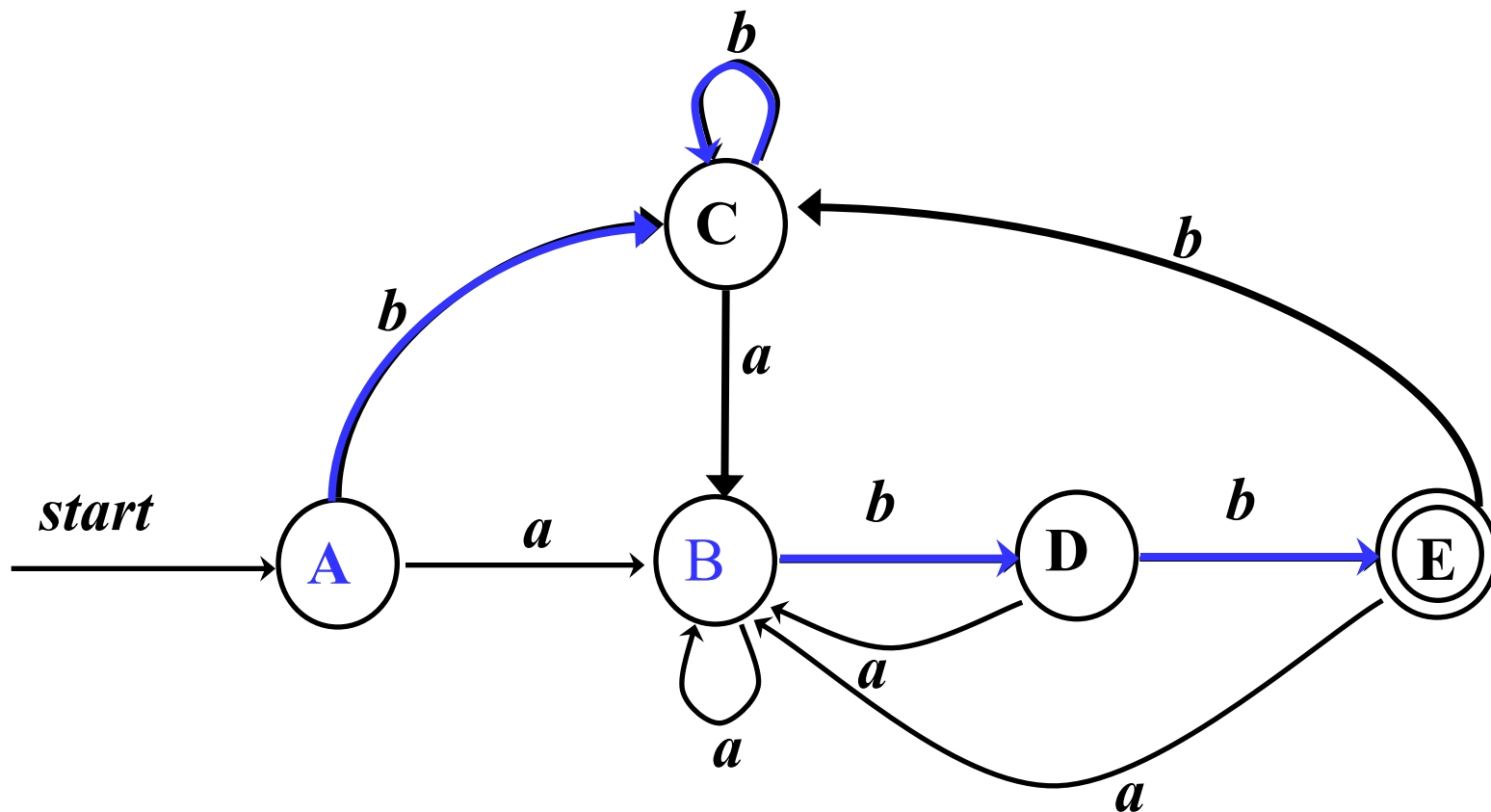
- ε 区分任何接受状态和非接受状态
- A 和 B 是可区分的状态：
 - 从A出发，读入串 1 后到达非接受状态C；
 - 从B出发，读过串 1 后到达接受状态D



不可区分的两个状态就是等价的，可以合并

例: 可区分状态

$(a|b)^*abb$ 对应的DFA



bb 区分状态A和状态B

DFA最小化算法

- **(推论)DFA状态等价的条件:**
 - **一致性条件** : s 、 t 同为终态或非终态
 - **蔓延性条件** : 对**所有**输入符号, s 、 t 必须转换到**等价**的状态集中, 同时具有传递性
- **DFA简化算法**
 - 划分部分: 根据以上条件迭代式划分等价类,
 - 构造部分: 从划分得到的等价类中选取**代表**, 并重建DFA

DFA最小化算法 (划分部分)

- 初始划分:接受状态组和非接受状态组 $\Pi = \{ S - F, F \}$
- 迭代，不断划分

for (Π 中的每个元素/集合G) {
 细分G，使得G中的s、t仍然在同一组中 **iff**
 对任意输入a，s、t都到达 Π 中的同一组；
 Π_{new} = 将 Π 中的G替换为细分得到的小组
}

- 如果 $\Pi_{\text{new}} == \Pi$ ，令 $\Pi_{\text{final}} = \Pi$ ，算法完成；否则 $\Pi = \Pi_{\text{new}}$ ，转步骤2

集合G的每个状态读入同一字符后，都落入相同的某个集合，那么就不用细分S

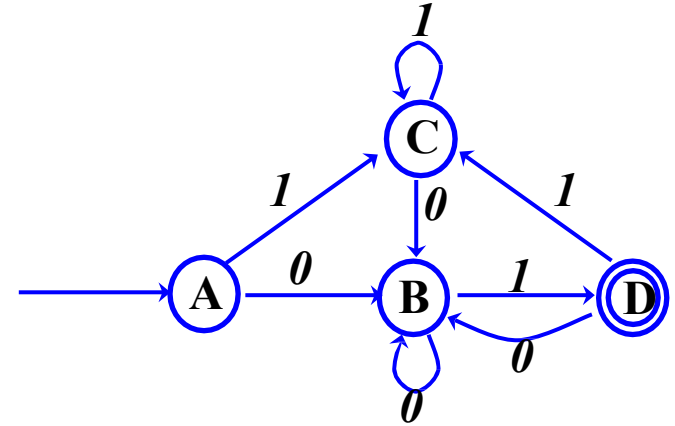
DFA最小化算法 (构造部分)

- 在 Π_{final} 的每个组中选择一个状态作代表，作为最小化DFA中的状态
 - 开始状态就是包含原开始状态的组的代表
 - 接受状态就是包含了原接受状态的组的代表 (这个组一定只包含接受状态)
- 转换关系构造如下
 - 如果s是G的代表，而原DFA中s在a上的转换到达t，且t所在组的代表为r，那么最小化DFA中有从s到r的在a上的转换

例: DFA的化简

按是否可区分识别等价类

- 1. 等价类: $\{A, B, C\}, \{D\}$



状态 \ 输入	0	1
A	B	C
B	B	D
C	B	C
D	B	C

集合G的每个状态读入同一字符后，都落入相同的某个集合，那么不用细分G

例: DFA的化简

按是否可区分识别等价类

1. $\{A, B, C\}, \{D\}$

2. 测试A, B, C是否可区分

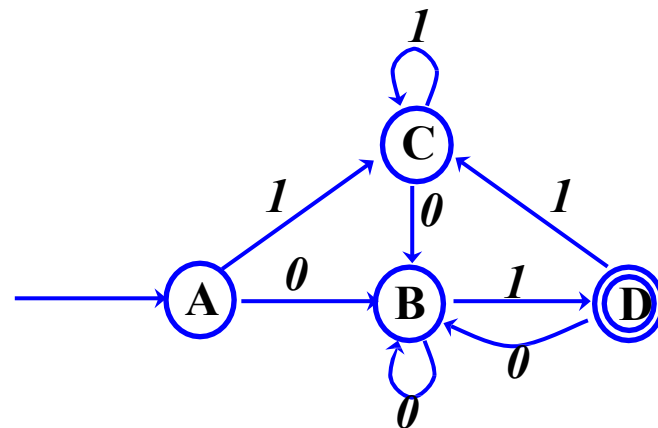
$move(\{A, B, C\}, 0) = \{B\}$

$move(\{A, B, C\}, 1) = \{C, D\}$



接收1时, 出现了分歧

等价类: $\{A, C\}, \{B\}, \{D\}$



状态 \ 输入	0	1
A	B	C
B	B	D
C	B	C
D	B	C

集合G的每个状态读入同一字符后, 都落入相同的某个集合, 那么不用细分G

例: DFA的化简

按是否可区分识别等价类

1. $\{A, B, C\}, \{D\}$
2. $\{A, C\}, \{B\}, \{D\}$
3. 测试A, C是否可区分

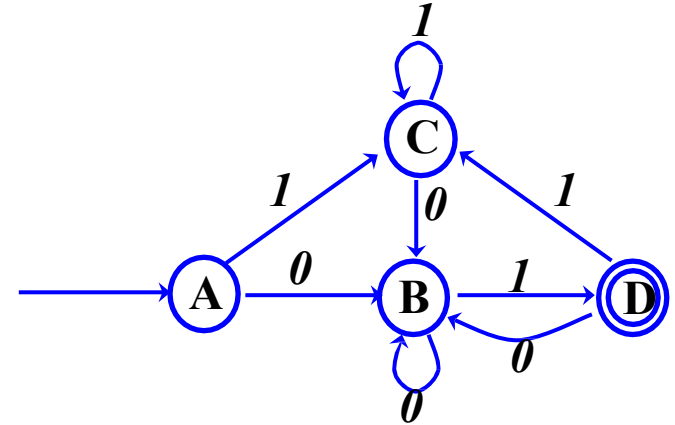
$$\text{move}(\{A, C\}, 0) = \{B\}$$

$$\text{move}(\{A, C\}, 1) = \{C\}$$



无法再区分A和C

等价类 $\{A, C\}, \{B\}, \{D\}$



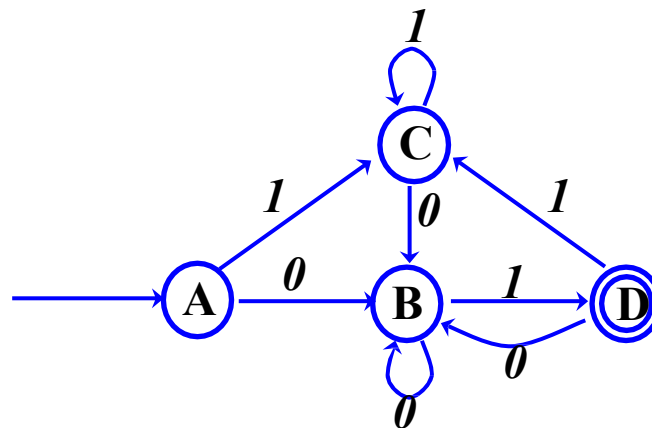
状态 \ 输入	0	1
A	B	C
B	B	D
C	B	C
D	B	C

集合G的每个状态读入同一字符后，都落入相同的某个集合，那么不用细分G

例: DFA的化简

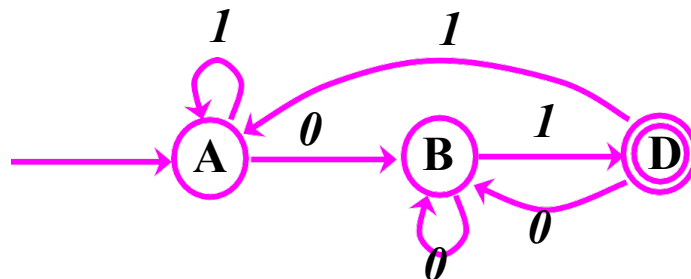
按是否是接受状态来区分

1. $\{A, B, C\}, \{D\}$
2. $\{A, C\}, \{B\}, \{D\}$
3. $\{A, C\}, \{B\}, \{D\}$



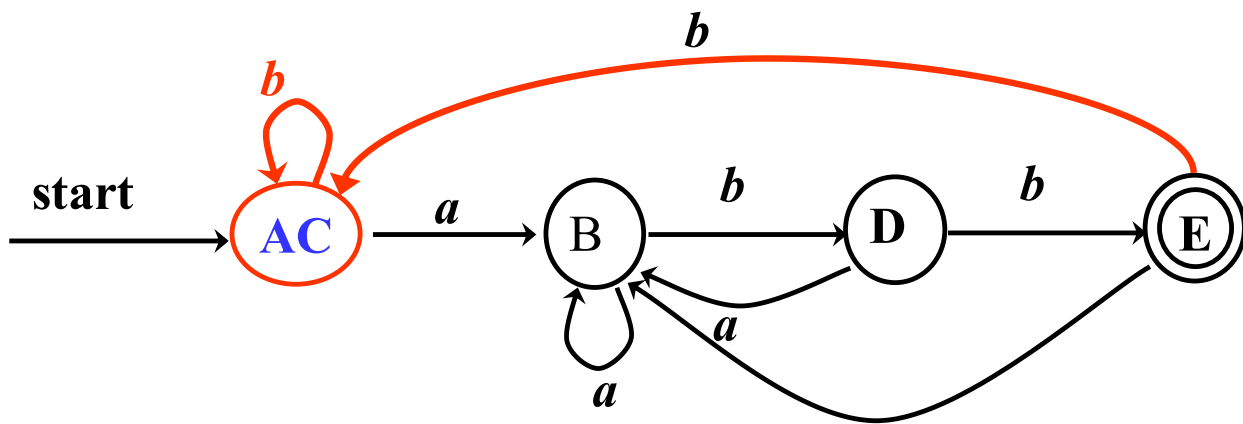
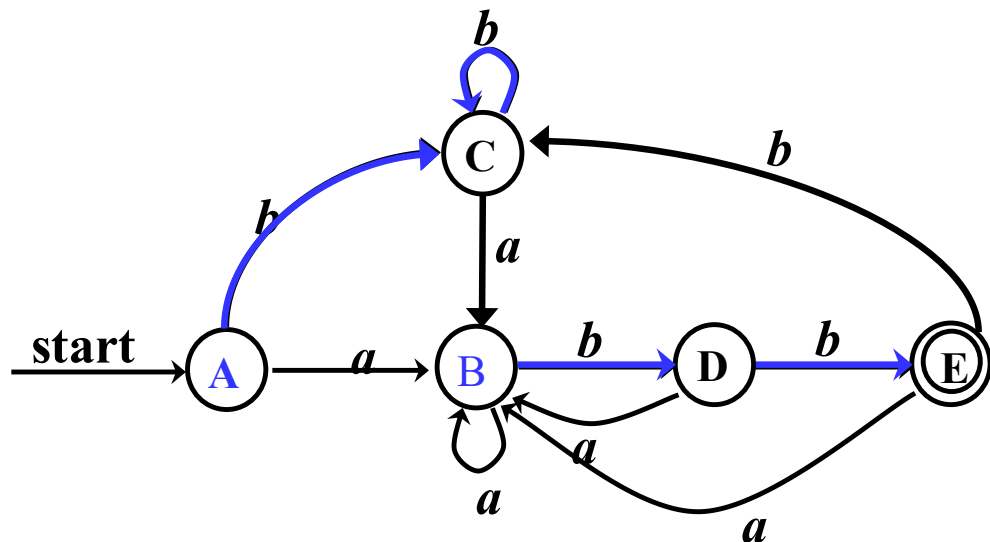
每个组中选择一个状态作代表，
作为最小化DFA中的状态

- A代表 $\{A, C\}$
- B代表 $\{B\}$
- D代表 $\{D\}$



简化后的DFA

练习: DFA的化简



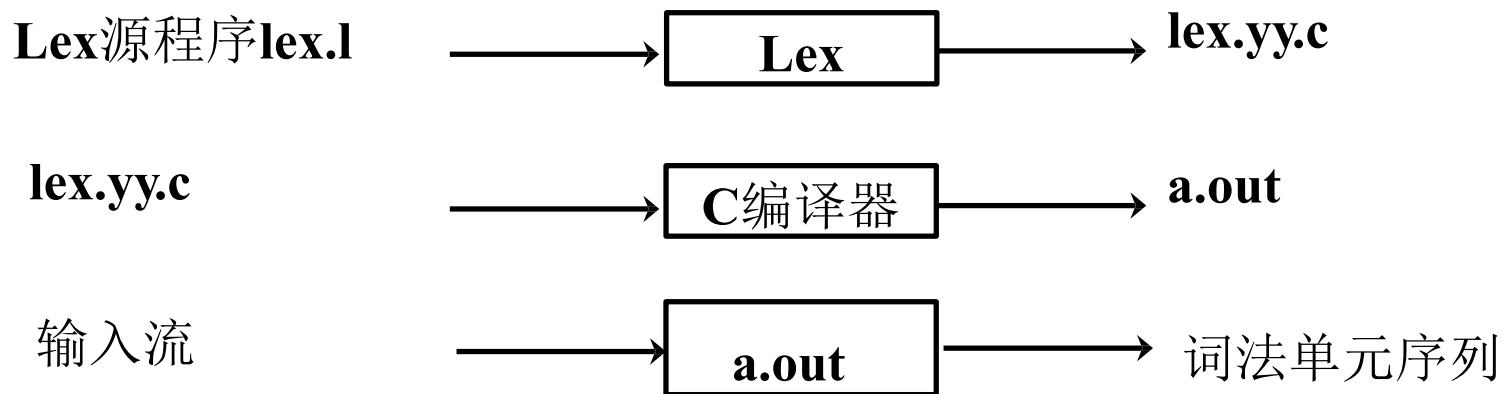
5. Lex 词法分析工具

词法分析和语法分析器的自动生成

- Lex, Yacc
- Flex, Bison
- ANTLR
- ...

用Lex生成词法分析器

- Lex/Flex是一个有用的词法分析器生成工具
- 通常和Yacc一起使用，生成编译器的前端



用Lex创建一个词法分析器

Lex程序的结构

- **声明部分**

- 常量：表示常数的标识符
- 正则定义

- **转换规则**

- 模式 { 动作 }
- 模式是正则表达式
- 动作表示识别到相应模式时应采取的处理方式
- 处理方式通常用是C语言代码表示

- **辅助函数**

- 各个动作中使用的函数

声明部分

%%

转换规则

%%

辅助函数

Lex程序的形式

例: Lex文件—声明部分

%{和}%之间的内容一般被直接拷贝到lex.yy.c中； 这里的内容就是一段注释； LT、LE等的值在Yacc源程序中定义

%{

**/* 常量LT, LE, EQ, NE, GT, GE,
WHILE, DO, ID, NUMBER, RELOP的定义*/**

%}

/* 正则定义 */

delim [\t \n]

ws {**delim**}+

letter [A-Za-z]

digit [0-9]

id {letter}({letter}|{digit})*

number {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

例: Lex文件—翻译规则部分

```
{ws}                /* 没有动作, 也不返回 */}  
while              {return (WHILE);}  
do                 {return (DO);}  
{id}                {yylval = install_id ( ); return (ID);}  
{number}           {yylval = install_num( );  
                    return (NUMBER);}  
  
“ < ”             {yylval = LT; return (RELOP);}  
“ <= ”           {yylval = LE; return (RELOP);}  
“ = ”             {yylval = EQ; return (RELOP);}  
“ <> ”           {yylval = NE; return (RELOP);}  
“ > ”             {yylval = GT; return (RELOP);}  
“ >= ”           {yylval = GE; return (RELOP);}
```

没有返回, 表示 继续识别其它的 词法单元

把识别到的标识符 加入标识符表

识别到数字常量, 加入常量表

例: Lex文件—辅助函数

- Lex处理源程序时，辅助函数被拷贝到lex.yy.c中
- 辅助函数可在规则中直接调用

```
installId ( ) {  
    /* 把词法单元装入符号表并返回指针。  
       yytext指向该词法单元的第一个字符，  
       yyleng给出的它的长度 */  
}
```

```
installNum ( ) {  
    /* 类似上面的过程，但词法单元不是标识符而是数 */  
}
```

词法分析器的工作方式

- **Lex生成的词法分析器作为一个函数被调用**
- **在每次调用过程中，不断读入余下的输入符号**
- **发现最长的、与某个模式匹配的输入前缀时**
 - 调用相应的动作，该动作进行相关处理
 - 之后词法分析器继续寻找其它词素

Lex中的冲突解决方法

- **冲突**：多个输入前缀与某个模式相匹配，或者一个前缀与多个模式相匹配
- **Lex解决冲突的方法**
 - **Longest match**: 多个前缀可能匹配时，选择最长的前缀
 - 比如，词法分析器把<=当作一个词法单元识别
 - **Rule Priority**: 最长前缀与多个模式匹配时，选择列在前面的模式
 - 如果保留字的规则在标识符的规则之前，词法分析器将识别出 保留字

例: Rule Priority

R = Whitespace | ‘new’ | Integer | Identifier

分析“new foo”

- “new” 匹配 **R**, 更精确地说是 ‘new’
- 但是它可能也匹配 **Identifier**, 此时该选哪个?

一般地, 如果 $x_1 \dots x_i \sqsubseteq L(R_j)$ 和 $x_1 \dots x_i \sqsubseteq L(R_k)$

规则: 选择先列出的正则定义(j 如果 $j < k$)

- 例如: 须将 ‘new’ 列在 **Identifier** 的前面

词法分析总结

- **词法分析器的作用和接口**
- **重要概念及其转换技巧/方法**
 - 非形式描述的Token , 正则表达式
 - 正则表达式 \rightarrow NFA
 - NFA \rightarrow DFA: 子集构造法
 - DFA \rightarrow 最简DFA
- **Lex工具的使用**



Thank you all for your attention

计算 ε -closure (T)

实际上是一个图搜索过程 (只考虑 ε 标号边)

将 T 的所有状态压入 $stack$ 中 ;

将 ε -closure (T) 初始化为 T ;

while ($stack$ 非空){

 将栈顶元素 t 给弹出栈中 ;

 for (每个满足如下条件的 u : 从 t 出发有一个标号为 ε 的转换到达状态 u)

 if (u 不在 ε -closure (T) 中){

 将 u 加入到 ε -closure (T) 中 ;

 将 u 压入栈中;

 }

 }

NFA到DFA的转换: 子集构造法

整个算法实际是一个搜索过程

– $Dstates$ 中的一个状态未加标记表示还没有搜索过它的 各个后继

- 输入: $NFA\ N$, 输出: 接收同样语言的 $DFA\ D$
- 方法: $\varepsilon\text{-closure}(s_0)$ 是 $Dstates$ 中的唯一状态, 且它未加标记;
while (在 $Dstates$ 中有一个未标记状态 T) {
 给 T 加上标记;
 for (每个输入符号 a) {
 $U = \varepsilon\text{-closure}(\text{move}(T, a))$;
 if (U 不在 $Dstates$ 中)
 将 U 加入到 $Dstates$ 中, 且不加标记;
 $Dtran[T, a] = U$;
 }
}

操作	描述
$\varepsilon\text{-closure}(s)$	从 NFA 的状态 s 开始, 只通过 ε 转换到达的状态集合
$\varepsilon\text{-closure}(T)$	从 T 中的某个 NFA 状态 s 开始, 只通过 ε 转换到达的状态集合, 即 $U_{s \in T} \varepsilon\text{-closure}(s)$
$\text{move}(T, a)$	从 T 中的某个状态 s 出发, 通过标号为 a 的转换到达的状态集合