第11章 词法分析程序的自动生成技术

1. 词法分析的功能
2. 词法分析程序的设计与实现
   - 状态图
3. 词法分析程序的自动生成
   - 有穷自动机

11.1 正则表达式与有穷自动机

➢ 正则表达式和正则集合的递归定义

有字母表∑，定义在∑上的正则表达式和正则集合递归定义如下：

1. ε和ϕ都是∑上的正则表达式，它们所表示的正则集合分别为：{ε}和ϕ；
2. 任何a∈∑，a是∑上的正则表达式，它所表示的正则集合为：{a}；
3. 假定U和V都是∑上的正则表达式，它们所表示的正则集合分别记为L(U)和L(V)，那么U|V，U•V和U^*也都是∑上的正则表达式，它们所表示的正则集合分别为L(U)∪L(V)、L(U)•L(V)和$L(U^*)$；
4. 任何∑上的正则表达式和正则集合均由1、2和3产生。

正则表达式中的运算符：

| ——或（选择）　　• ——连接

* 或 { }——重复　　( ) ——括号

与集合的闭包运算有区别
这里a*表示由任意个a组成的串，
而{a, b}* = {ε，a，b，aa，ab，ba，bb，.....}

运算符的优先级：

先*，　后•，　最后 |

- 在正则表达式中可以省略。

正则表达式相等⇔这两个正则表达式表示的语言相等。

例：设 ∑ = { a, b }，下面是定义在∑上的正则表达式和正则集合

正则表达式	正则集
ba*	∑上所有以b为首，后跟任意多个a的字。
a(a|b)*	∑上所有以a为首的字。
(a|b)*(aa|bb)(a|b)*	∑上所有含有两个相继的a或两个相继的b的字。

正则表达式的性质：

设e_1, e_2和e_3均是某字母表上的正则表达式，则有：

单位正则表达式：ε　　$\varepsilon e = e\varepsilon = e$

交换律：　$e_1|e_2 = e_2|e_1$

结合律：　$e_1|(e_2|e_3) = (e_1|e_2)|e_3$

$e_1(e_2e_3) = (e_1e_2)e_3$

分配律：　$e_1(e_2|e_3) = e_1e_2|e_1e_3$

$(e_1|e_2)e_3 = e_1e_3|e_2e_3$

此外：　$r^* = (r|\varepsilon)^*$　　$r^{**} = r^*$

$(r|s)^* = (r^*s^*)^*$

正则表达式与3型文法等价

例如:

正则表达式:	ba*	a(a|b)*
3型文法:	Z ::= Za|b	Z::=Za|Zb|a

例:

3型文法	正则表达式	
S::= aS|aB	aS|aba*a	a*aba*a
B::= bC	ba*a	
C::= aC|a	a*a	

11.2.1 确定有穷自动机（DFA）— 前面介绍状态图的形式化

一个确定的有穷自动机（DFA）M是一个五元式:

$$M = (S, \Sigma, \delta, s_0, Z)$$

其中:

1. S — 有穷状态集
2. Σ — 输入字母表
3. δ — 映射函数(也称为状态转换函数)

   $S \times \Sigma \rightarrow S$

   $\delta(s, a) = s'$　　$s, s' \in S$,　$a \in \Sigma$

   s'叫做s的后继状态
4. s_0 — 初始状态　　$s_0 \in S$
5. Z — 终止状态集　　$Z \subseteq S$

例如: M: ({ 0, 1, 2, 3 }, { a, b }, δ, 0, {3})

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

状态转移函数δ可用一矩阵来表示:

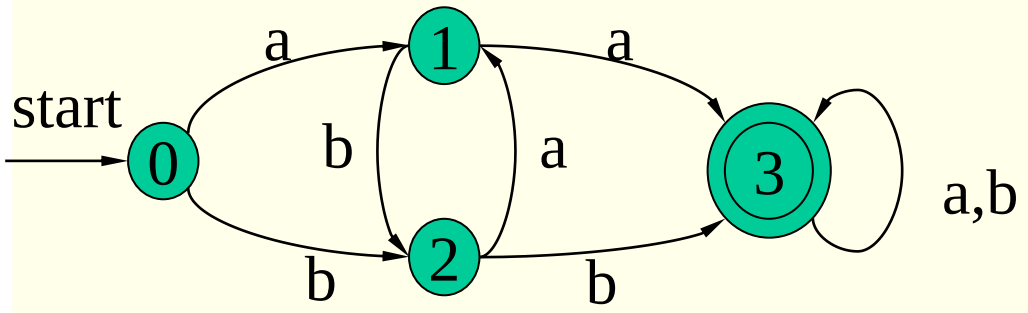
状态 \ 输入字符	a	b
0	1	2
1	3	2
2	1	3
3	3	3

所谓确定的状态机，其确定性表现在状态转移函数是单值函数!

一个DFA也可以用一个状态转换图表示:

状态 \ 输入字符	a	b
0	1	2
1	3	2
2	1	3
3	3	3

DFA的状态图表示:



DFA M所接受的符号串：

令$\alpha = a_1 a_2 \cdots a_n$，$\alpha \in \Sigma^*$，

若$\delta($ s_{n-1} $, a_n) = s_n$, 且$s_n \in Z$,

则可以写成$\delta(s_0, \alpha) = s_n$, 我们称$\alpha$可为M所接受。

换言之：若存在一条从初始状态到某一终止状态的路径，且这条路径上所有弧的标记符连接成符号串α，则称α为DFA M(接受)识别。

DFA M所接受的语言为： $L(M) = \{\alpha \mid \delta(s_0, \alpha) = s_n,\ s_n \in Z\}$

11.2.2 非确定的有穷自动机(NFA)

若δ是一个多值函数，且输入可允许为ε，则有穷自动机是不确定的。即在某个状态下，对于某个输入字符存在多个后继状态。

NFA的形式定义为:

一个非确定的有穷自动机NFA M'是一个五元式:

NFA M' = (S, Σ∪{ε}, δ, S_0, Z)

其中 S — 有穷状态集

Σ∪{ε}—输入符号加上ε，即自动机的每个结点所射出的弧可以是Σ中的一个字符或是ε。

S_0 — 初态集　　Z — 终态集

δ— 转换函数 $S \times \Sigma \cup \{\varepsilon\} \rightarrow 2^S$

(2^S： S的幂集—S的子集构成的集合）

NFA M'所接受的语言为:

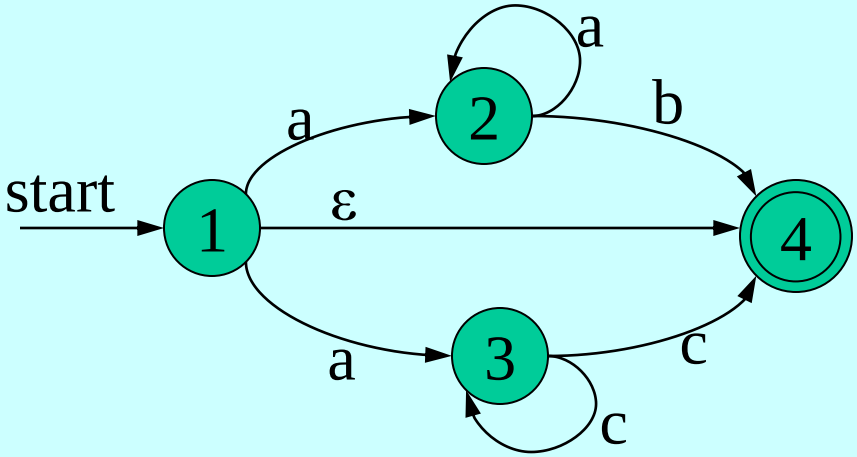
$$L(M') = \{\alpha \mid \delta(S_0, \alpha) = S' \quad S' \cap Z \neq \Phi\}$$

例: NFA $M' = (\{1, 2, 3, 4\}, \{\varepsilon, a, b, c\}, \delta, \{1\}, \{4\})$

状态 \ 符号	ε	a	b	c
1	{4}	{2, 3}	Φ	Φ
2	Φ	{2}	{4}	Φ
3	Φ	Φ	Φ	{3, 4}
4	Φ	Φ	Φ	Φ

状态 \ 符号	ε	a	b	c
1	{4}	{2, 3}	Φ	Φ
2	Φ	{2}	{4}	Φ
3	Φ	Φ	Φ	{3, 4}
4	Φ	Φ	Φ	Φ

上例题相应的状态图为:



M' 所接受的语言（用正则表达式） $R=aa^* b|ac^*c|\varepsilon$

11.2.3 NFA的确定化

已证明：非确定的有穷自动机与确定的有穷自动机从功能上来说是等价的。也就是说，我们能够从：

NFA M' —构造成一个→ DFA M

使得 L(M) = L(M')

为了使得NFA确定化，我们首先给出两个定义：

定义1、集合 I 的 ε-闭包:

令 I 是一个状态集的子集，定义ε- closure(I)为:

1）若s∈I，则s∈ε- closure(I);

2）若s∈I，则从 s 出发经过任意条ε弧能够到达的任何状态都属于ε- closure(I)。

状态集ε- closure(I)称为 I 的ε-闭包。

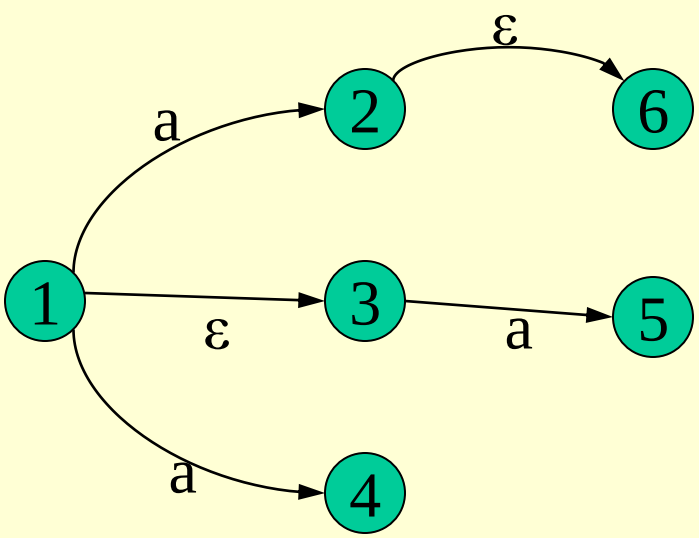
我们可以通过一例子来说明状态子集的 ε –闭包的构造方法。

例:

如图所示的状态图:

令I={1},

求 ε - closure(I)=?



根据定义:

ε - closure(I)={1, 3}

定义2：令 I 是NFA M'的状态集的一个子集，$a \in \Sigma$

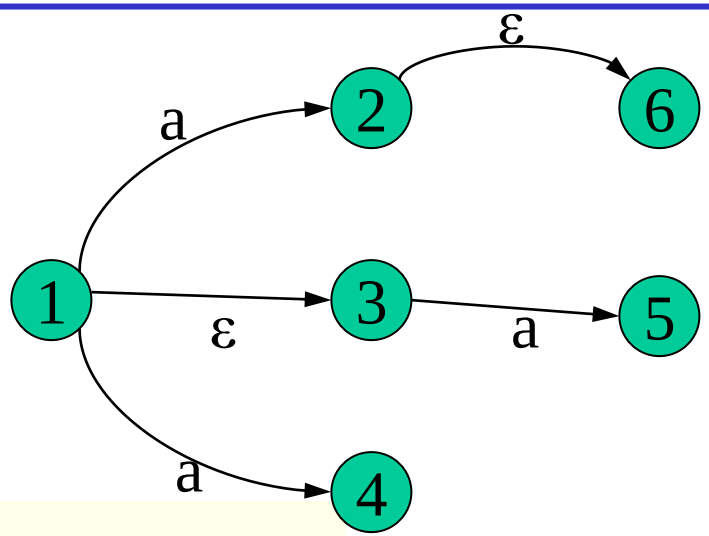
定义：$I_a = \varepsilon\text{- closure}(J)$

其中 $J = \bigcup_{S \in I} \delta(s, a)$

—— J 是从状态子集 I 中的每个状态出发，经过标记为 a 的弧而达到的状态集合。

—— I_a是状态子集，其元素为 J 中的状态，加上从 J 中每一个状态出发通过 ε 弧到达的状态。

我们同样可以通过一例子来说明上述定义，仍采用前面给定状态图为例。



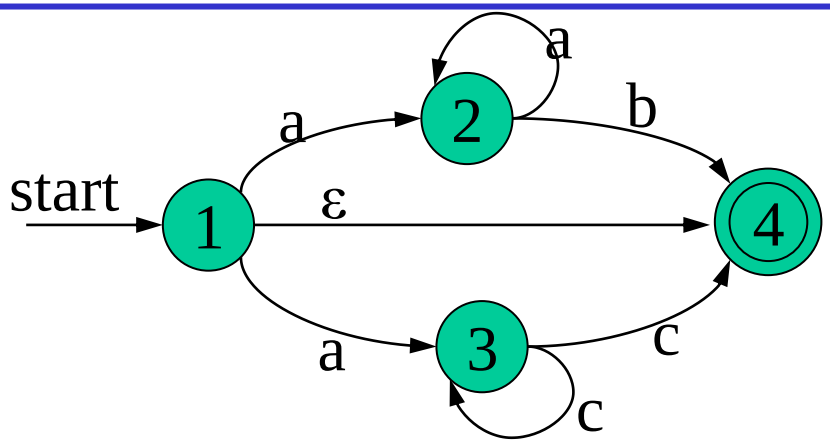
例：令 $I=\{1\}$

$I_a=\varepsilon\text{-closure}(J)$

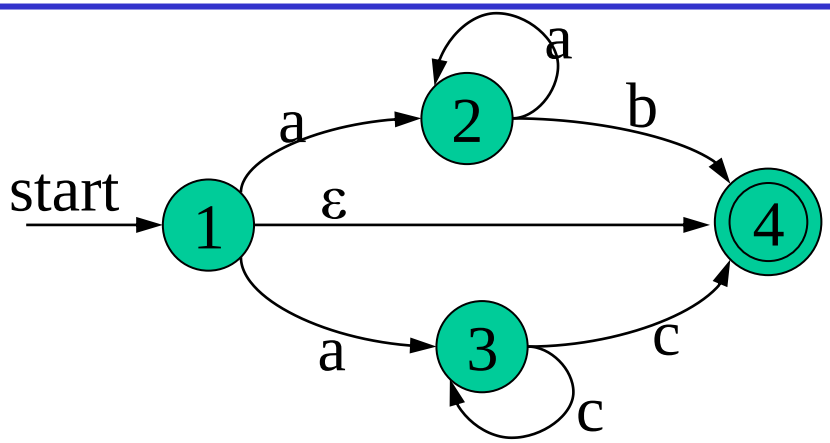
$=\varepsilon\text{-closure}(\delta(1,a))$

$=\varepsilon\text{-closure}(\{2,4\})=\{2,4,6\}$

根据定义1、2，可以将上述的M'确定化（即可构造出状态的转换矩阵）。



I	I_a	I_b	I_c
{1,4}	{2,3}	φ	φ
{2,3}	{2}	{4}	{3,4}
{2}	{2}	{4}	φ
{4}	φ	φ	φ
{3,4}	φ	φ	{3,4}



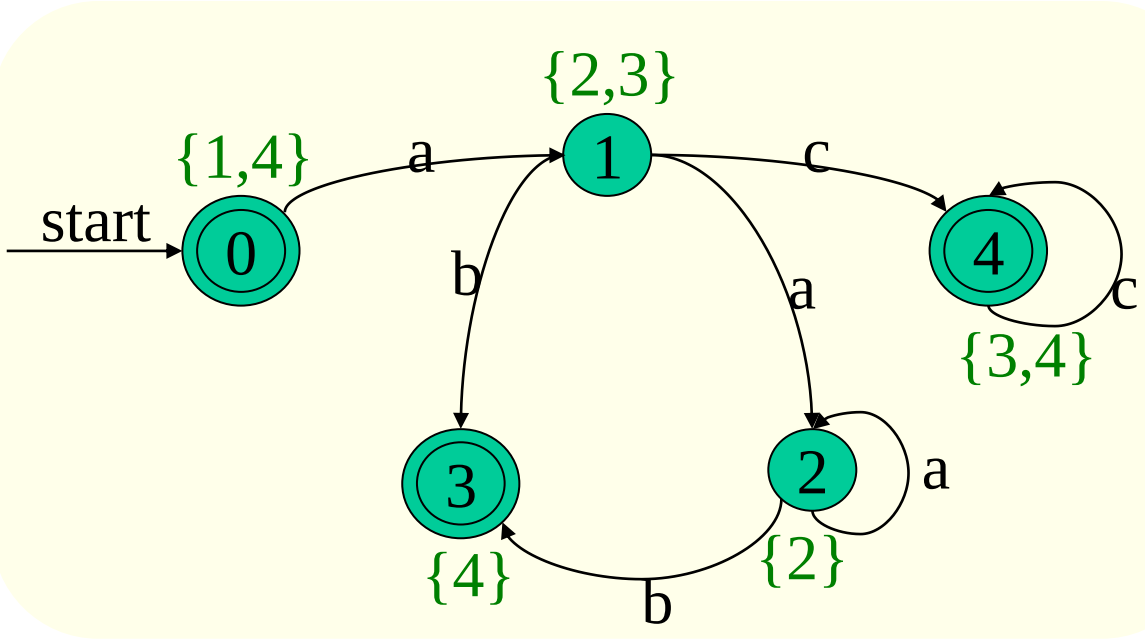
I	I_a	I_b	I_c
A	B	φ	φ
B	C	D	E
C	C	D	φ
D	φ	φ	φ
E	φ	φ	E

将求得的状态转换矩阵重新编号

DFA M状态转换矩阵:

状态 \ 符号	a	b	c
0	1	–	–
1	2	3	4
2	2	3	–
3	–	–	–
4	–	–	4

DFA M的状态图:



★ 注意: 原初始状态 1 的 ε-closure 为DFA M的初态;
包含原终止状态 4 的状态子集为DFA M的终态。

11.2.4 正则表达式与DFA的等价性

定理：在Σ上的一个字集V（$V \subseteq \Sigma^*$）是正则集合，当且仅当存在一个DFA M，使得V = L(M)。

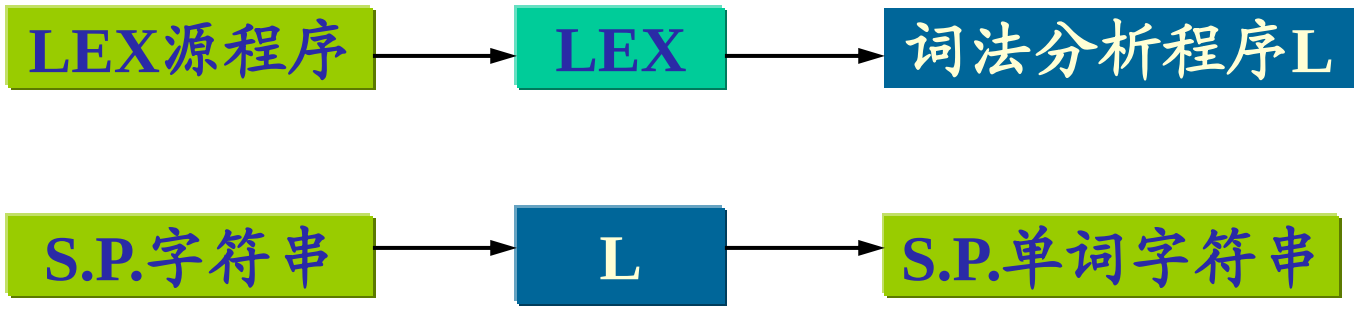
V是正则集合，
R是与其相对应的正则表达式 ⇔ DFA M
V=L(R) L(M)=L(R)

所以，正则表达式R ⇒ NFA M’ ⇒ DFA M
L(R) = L(M’) = L(M)

证明：根据定义.

11.3 词法分析程序的自动生成器—LEX（LEXICAL）

LEX的功能：



下面我们介绍LEX源程序：

11.3.1 LEX源程序

一个LEX源程序主要由三个部分组成:

1. 规则定义段
2. 识别规则段
3. 用户代码段

各部分之间用%%隔开。

规则定义段是如下形式的LEX语句：

格式：

name　definitions

D_1　R_1

D_2　R_2

⋮

D_n　R_n

其中：

D_1, D_2，……，D_n 为正则表达式名字，称简名。

R_1, R_2，……，R_n 为正则表达式。

例如某种语言标识符:

letter A|B|……|Z

digit 0|1| ……|9

iden letter(letter|digit)*

带符号整数:

integer digit (digit)*

sign +| - |ε

signinteger sign integer

识别规则：是一串如下形式的LEX语句：

格式：pattern　action

P_1　$\{A_1\}$

P_2　$\{A_2\}$

⋮

P_m　$\{A_m\}$

P_i：定义在$\Sigma \cup \{D_1, D_2, \cdots D_n\}$上的正则表达式，也称模式。

$\{A_i\}$：A_i为语句序列。它指出在识别出词形为P_i的单词以后，词法分析器所应作的动作。其基本动作是返回单词的类别编码和单词值。

下面是识别某语言单词符号的LEX源程序：

例：LEX 源程序

```
AUXILIARY    I
    letter → A|B|·
    digit  →  0|1|·
%%
RECOGNITION
```

```
1.BEGIN          {RETURN(1,—) }
2.END            {RETURN(2,—) }
3.FOR            {RETURN(3,—) }
```

RETURN是LEX过程，该过程将单词传给语法分析程序

RETURN（C，LEXVAL）

其中C为单词类别编码

LEXVAL:

标识符：TOKEN（字符数组）

整常数：DTB（数值转换函数，将TOKEN中的数字串转换二进制）

其他单词：无定义。

4.DO	{RETURN(4,–) }
5.IF	{RETURN(5,–) }
6.THEN	{RETURN(6,–) }
7.ELSE	{RETURN(7,–) }
8.letter(letter |digit)*	{RETURN(8,TOKEN) }
9.digit(digit)*	{RETURN(9,DTB }
10. :	{RETURN(10,–) }
11. +	{RETURN(11,–) }
12. “*”	{RETURN(12,–) }

13. ,	{RETURN(13,—) }
14. “（”	{RETURN(14,—) }
15. “）”	{RETURN(15,—) }
16. :=	{RETURN(16,—) }
17. =	{RETURN(17,—) }

```
%{
        /* definitions of manifest constants
        LT, LE, EQ, NE, GT, GE
        IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}     {/* no action and no return */}
if       {return{IF};}
then     {return{THEN};}
else     {return{ELSE};}
{id}     {yylval = (int) installID();return(ID);}
{number} {yylval = (int) installNum();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);}
%%

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

int installNum() {/* similar to installID, but puts numerical
                   constants into a separate table */
}
```

规则定义式

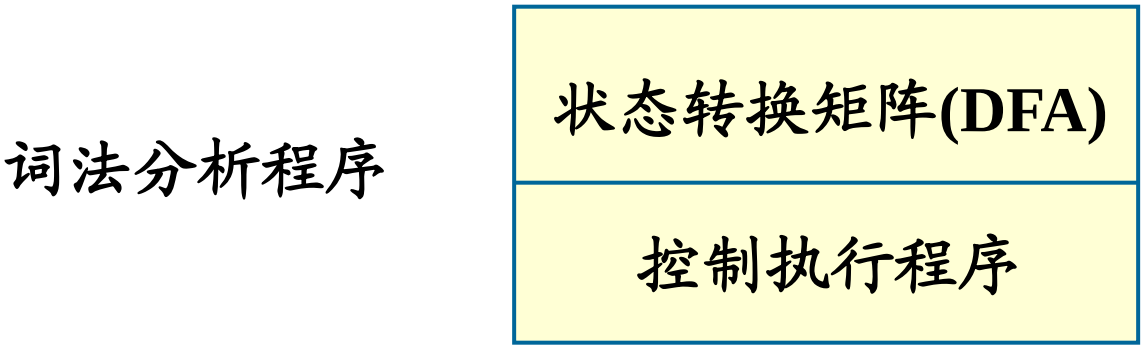
识别规则

用户子程序

11.3.2 LEX的实现

LEX的功能是根据LEX源程序构造一个词法分析程序，该词法分析器实质上是一个有穷自动机。

LEX生成的词法分析程序由两部分组成:

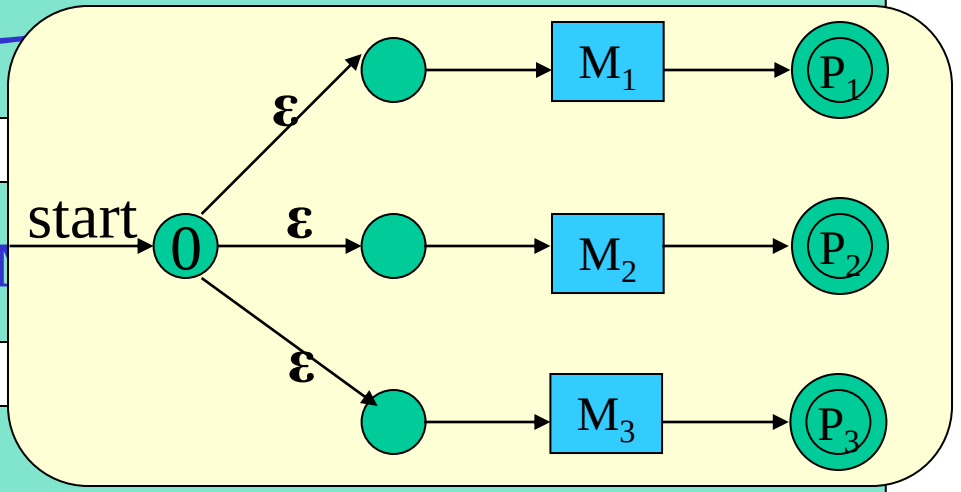


∴LEX的功能是根据LEX源程序生成状态转换矩阵和控制程序

LEX的处理过程:

- 扫描每条识别规则P_i构造一

..将各条规则的有穷自动机



∴确定化　　NFA⇒DFA

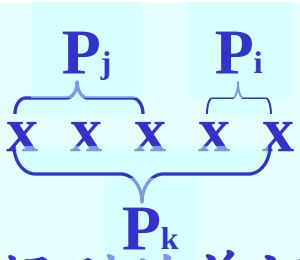
∵生成该DFA的状态转换矩阵和控制执行程序

如“BEGIN”是关键字还是标识符？
“: =”是冒号还是赋值号？

LEX二义性问题的两条原则：

1. 最长匹配原则

在识别单词过程中，有一字符串



根据最长匹配原则，应识别为这是一个符合P_k规则单词，而不是P_j和P_i规则的单词。

2. 最优匹配原则

如有一字符串，有两条规则可以同时匹配时，那么用规则序列中位于前面的规则相匹配，所以排列在前面的规则优先权最高。

例：字符串···BEGIN···

P_1

P_8

根据最优匹配原则，应该识别为关键字begin，所以在写LEX源程序时应注意规则的排列顺序。另要注意的是，优先匹配原则是在符合最长匹配的前提下执行的。

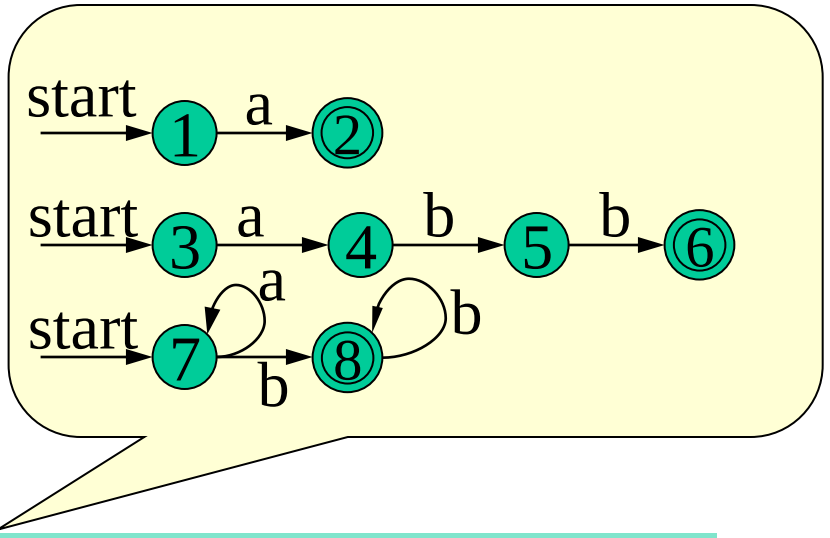
我们可以通过一个例子来说明这些问题。

例： LEX源程序，

a { }

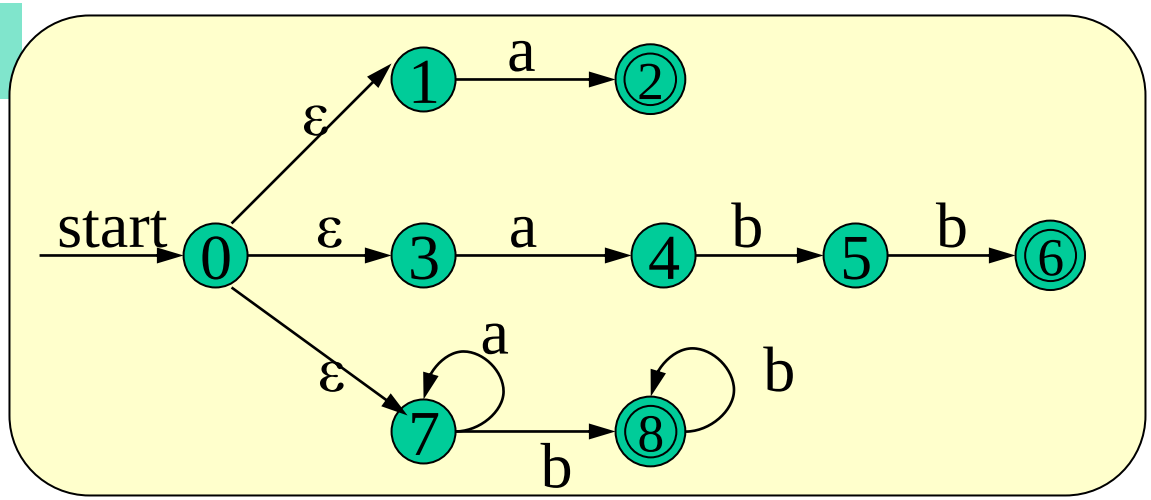
abb { }

a^*bb^* { }

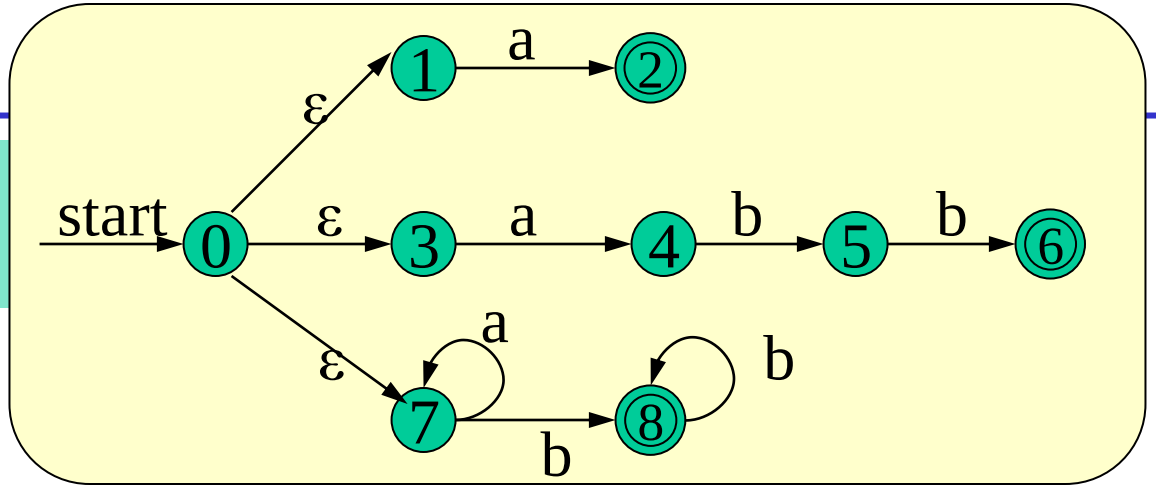


一、读LEX源程序，分别生成NFA，用状态图表示为：

二、合并成一个NFA:



三、确定化
给出状态转换的矩阵



状态	a	b	到达终态所识别的单词
初态 {0,1,3,7}	{2,4,7}	{8}	
终态 {2,4,7}	{7}	{5,8}	a
终态 {8}	φ	{8}	a^*bb^*
{7}	{7}	{8}	
终态 {5,8}	φ	{6,8}	a^*bb^*
终态 {6,8}	φ	{8}	abb或a

{6, 8} 集合存在二义性！解决方法：多个终态规则式谁在先就算谁！

在此DFA M中　初态为{0,1,3,7}

终态为{2,4,7},{8},{5,8},{6,8}

词法分析程序的分析过程

令输入字符串为aba…

(1) 吃进字符ab

(2) 按反序检查状态子集
检查前一次状态是否含有原NFA的终止状态

读入字符	读入字符
开始	{0,1,3,7}
a	{2,4,7}
b	{5,8}
a	无后继状态(退掉输入字符a)

- 即检查{5,8}。含有终态8，因此断定所识别的单词ab是属于 a*bb*中的一个。
- 若在状态子集中无NFA的终态，则要从识别的单词再退掉一个字符(b)，然后再检查上一个状态子集。
- 若一旦吃进的字符都退完，则识别失败，调用出错程序，一般是跳过一个字符然后重新分析。(应打印出错信息)

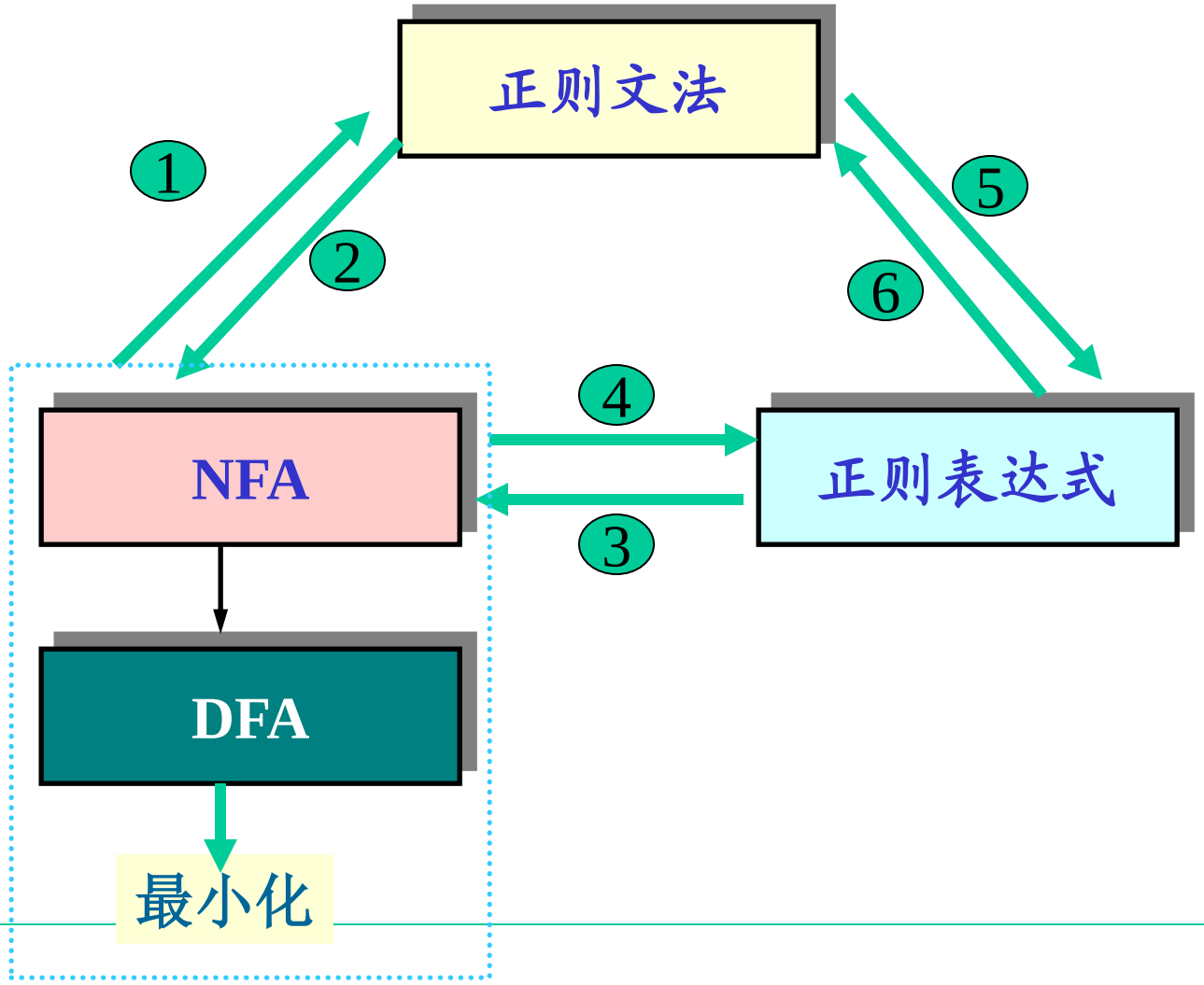
三点说明:

1)以上是LEX的构造原理，虽然是原理性，但据此就不难将LEX构造出来。

2)所构造出来的LEX是一个通用的工具，用它可以生成各种语言的语法分析程序，只需要根据不同的语言书写不同的LEX源文件就可以了。

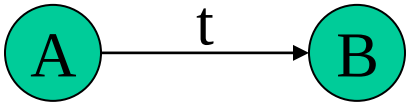
3)LEX不但能自动生成词法分析器，而且也可以产生多种模式识别器及文本编辑程序等。

补充



（1）有穷自动机 ⇒ 正则文法

算法：

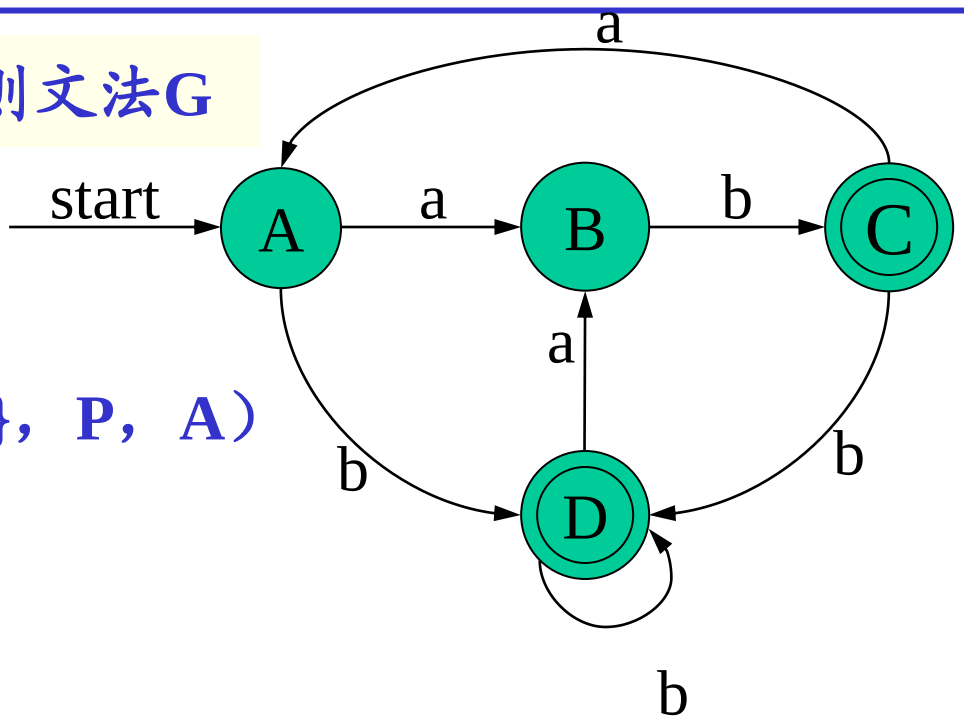


1. 对转换函数f (A, t) = B，可写成一个产生式:A→tB

2. 对可接受状态Z，增加一个产生式：Z→ε

3. 有穷自动机的初态对应于文法的开始符号，
   有穷自动机的字母表为文法的终结符号集。

例：给出如图NFA等价的正则文法G



G=（{A，B，C，D}，{a，b}，P，A）

其中P: $A \rightarrow aB$

$A \rightarrow bD$

$B \rightarrow bC$

$C \rightarrow aA$

$C \rightarrow bD$

$C \rightarrow \varepsilon$

$D \rightarrow aB$

$D \rightarrow bD$

$D \rightarrow \varepsilon$

（2）正则文法 ⇒ 有穷自动机M

算法:

1. 字母表与G的终结符号相同。

2. 为G中的每个非终结符生成M的一个状态，G的开始符号S是开始状态S。

3. 增加一个新状态Z，作为NFA的终态。

4. 对G中的形如A→tB，其中t为终结符或ε, A和B为非终结符的产生式，构造M的一个转换函数f (A, t) = B。

5. 对G中的形如A→t的产生式，构造M的一个转换函数 f (A, t) = Z。

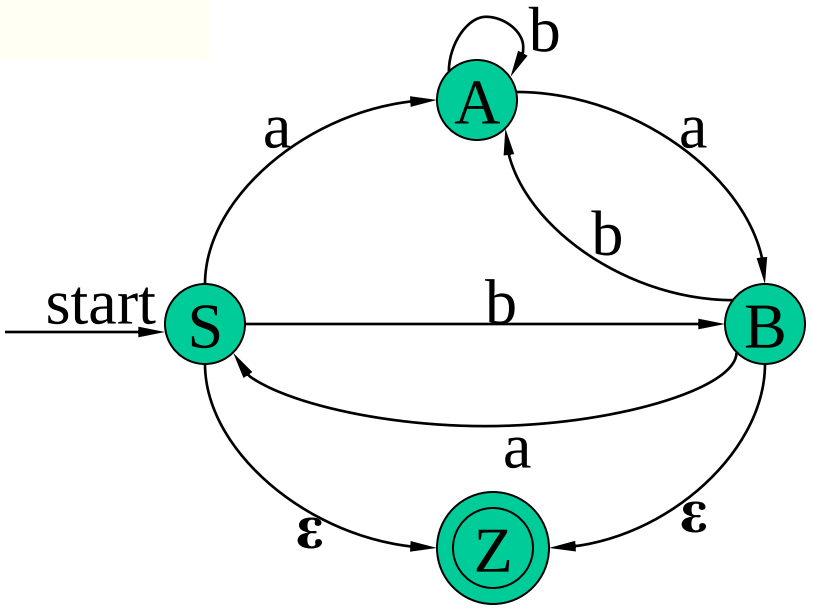
例:求与文法G[S]等价的NFA

G[S]: S→aA | bB |ε

A→aB | bA

B→aS | bA |ε

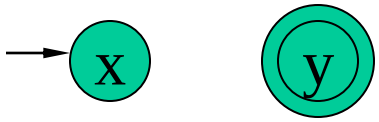
求得:



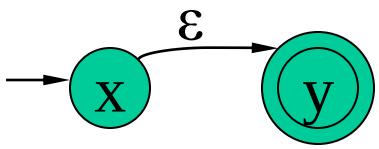
（3）正则式⇒有穷自动机

语法制导方法

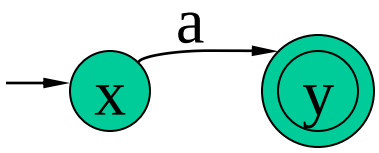
1.(a)对于正则式φ，所构造NFA:



(b)对于正则式ε，所构造NFA:

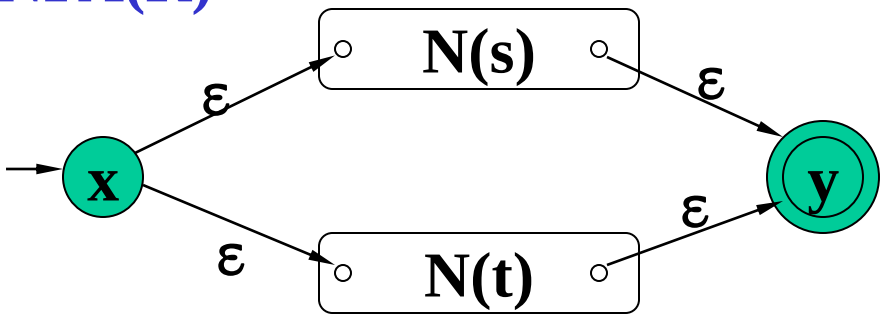


(c)对于正则式a，a∈Σ,则 NFA:

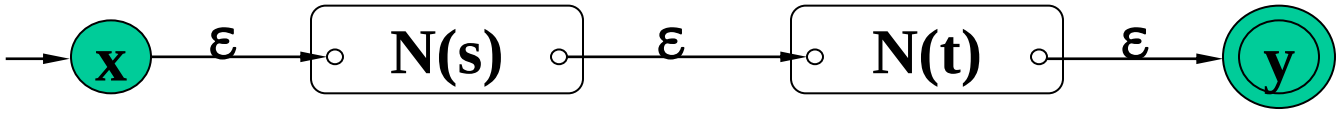


2. 若s, t为Σ上的正则式，相应的NFA分别为N(s)和N(t);

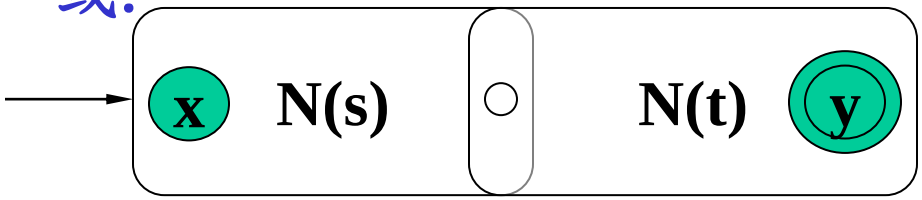
(a)对于正则式R = s | t， NFA(R)



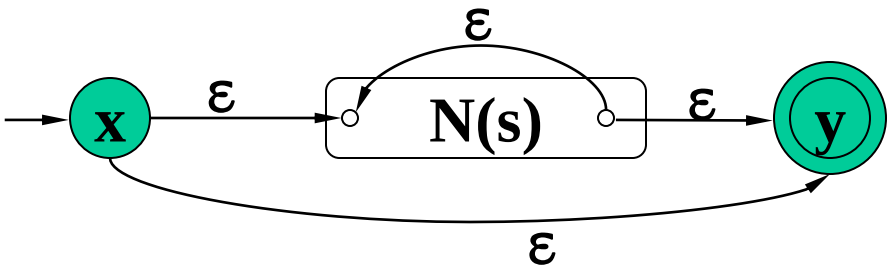
(b)对正则式R = s t， NFA(R)



或:



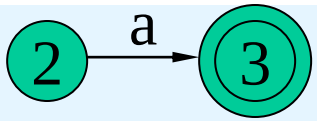
(c)对于正则式R = s^*， NFA(R)



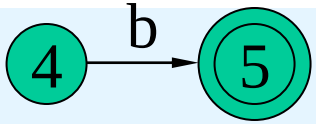
(d)对R = (s)，与R = S的NFA一样。

例:为R=(a|b)* abb构造NFA，使得L(N)=L(R)

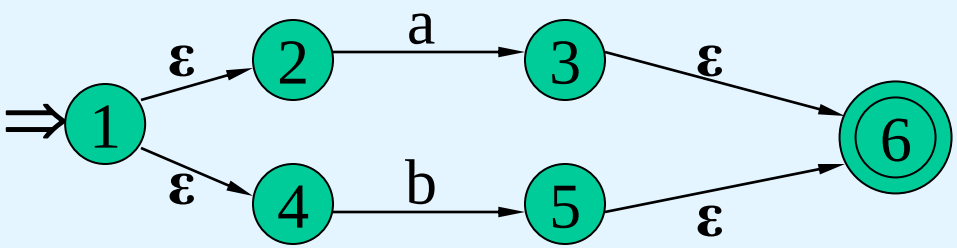
从左到右分解R,令r_1=a,第一个a,则有⇒



令r_2=b,则有⇒

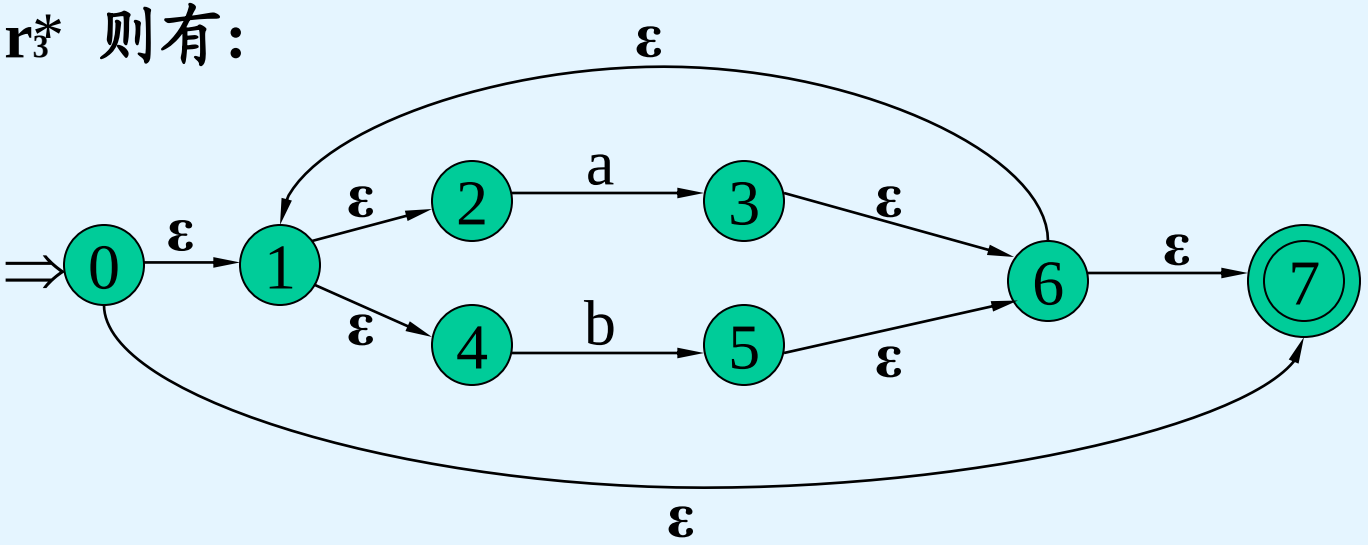


令r_3= r_1|r_2,则有

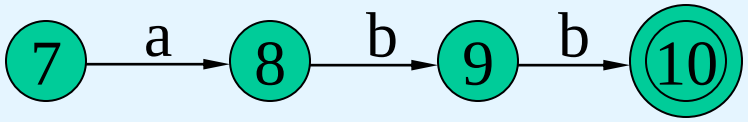


R=(a|b)*abb

令$r_4= r_3^*$ 则有:

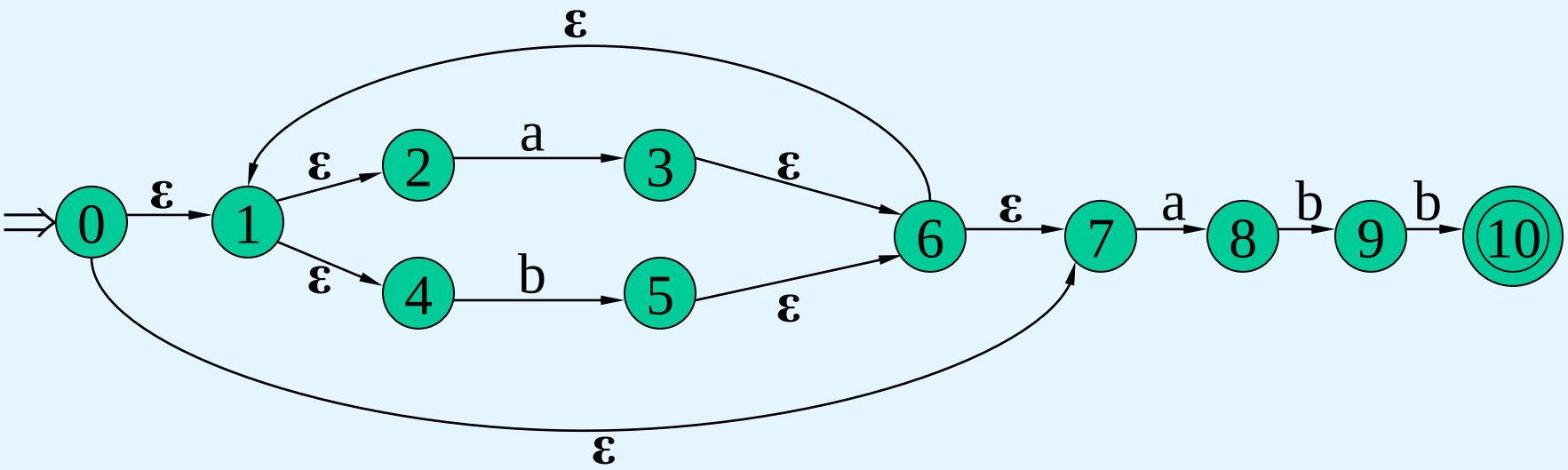


令r_5=a,
令r_6=b
令r_7=b
令$r_8= r_5 r_6$
令$r_9= r_8 r_7$ 则有 ⇒



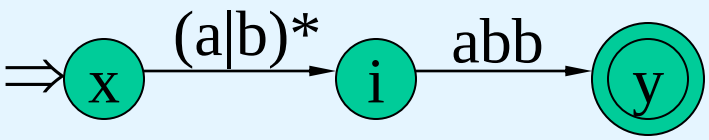
R=(a|b)*abb

令$r_{10}= r_4 r_9$ 则最终得到NFA N:

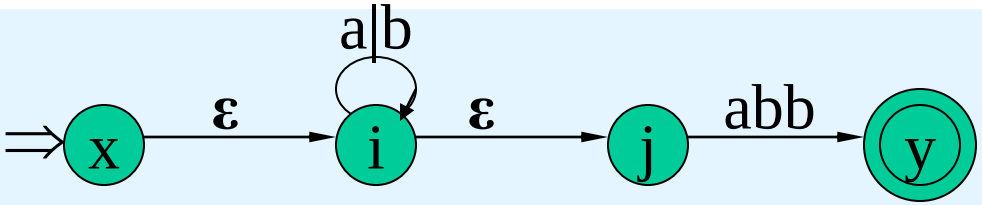


分解R的方法有很多种。下面我们看看另一种分解方式和所构成的NFA。

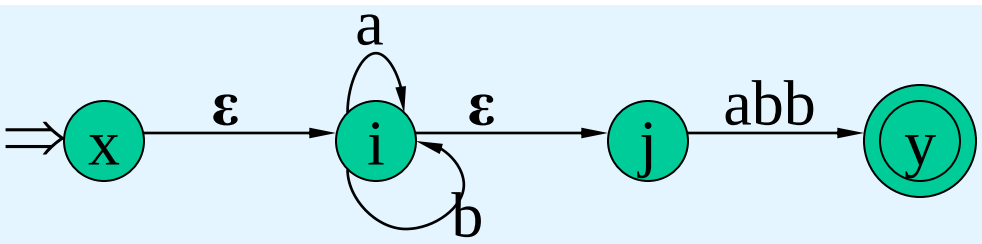
R=(a|b)*abb



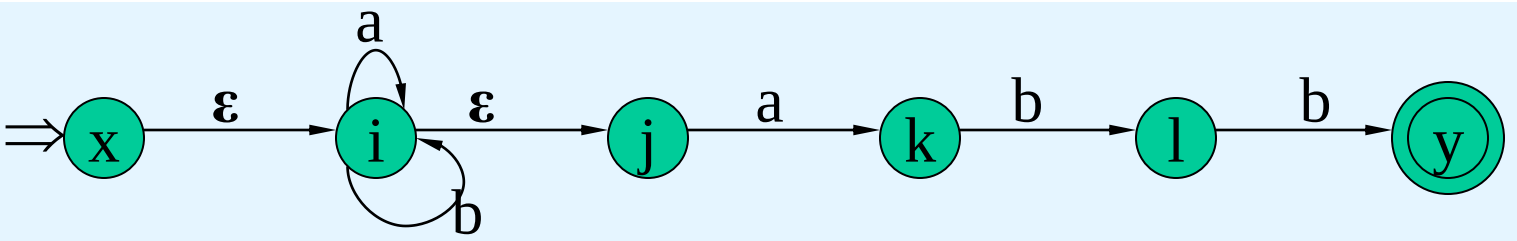
（a）



（b）



（c）



（d）

（4）有穷自动机⇒正则式R

算法:

1) 在M上加两个结点x, y。从x用ε弧连接到M的所有初态结点，从M的所有终态结点用ε弧连接到y，形成与M等价的M'。M'只有一个初态x和一个终态y。

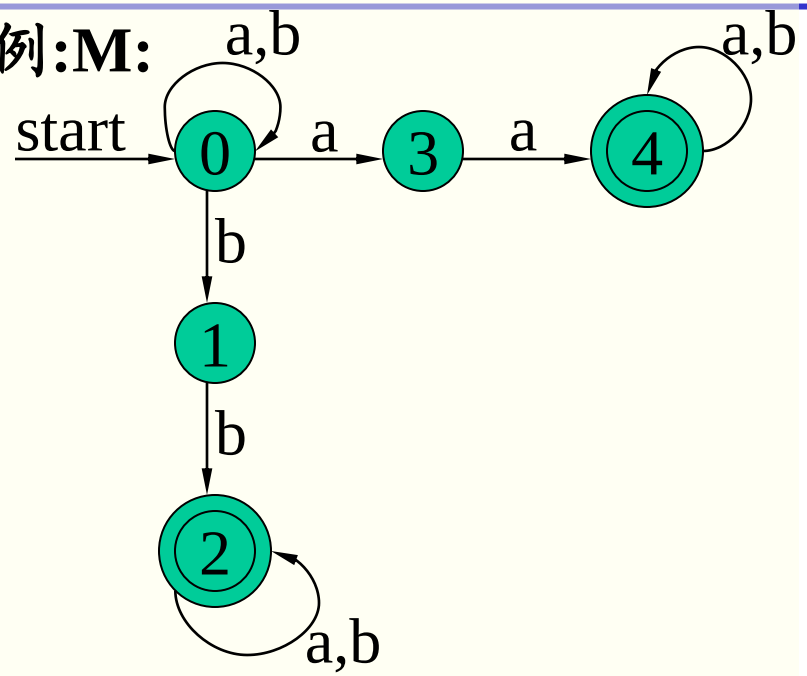
2) 逐步消去M'中的所有结点，直至剩下x和y为止。在消除结点的过程中，逐步用正则式来标记箭弧。其消结规则如下:

1.对于 

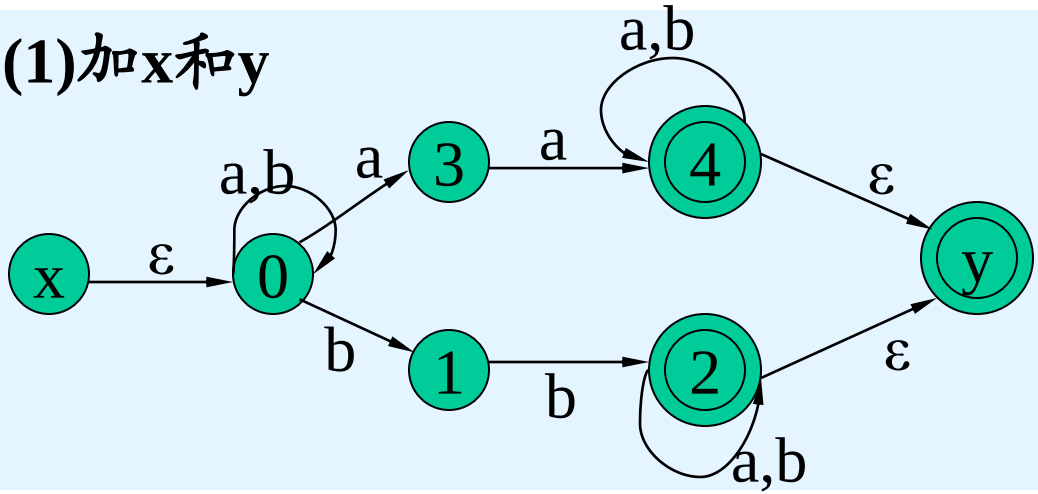
2.对于 

3.对于 

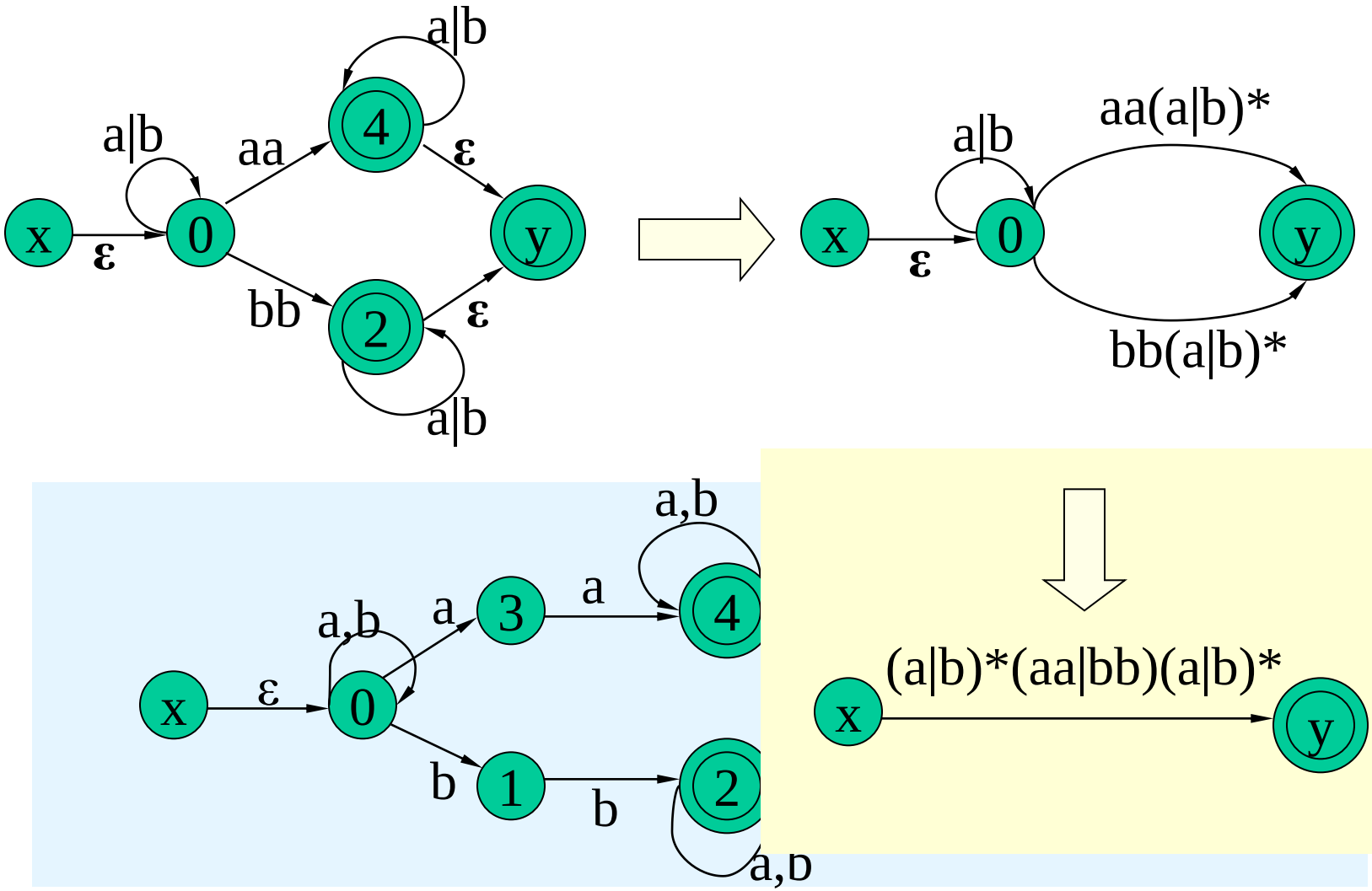
例:M:



解: (1)加x和y



(2) 消除M中的所有结点


a|b
a|b
aa
4
ε
x
ε
0
y
bb
2
ε
a|b
aa(a|b)*
a|b
x
ε
0
y
bb(a|b)*
a,b
a
3
a
4
a,b
x
ε
0
b
1
b
2
a,b
(a|b)*(aa|bb)(a|b)*
x
y

（5）正则文法 ⇒ 正则式

利用以下转换规则，直至只剩下一个开始符号定义的产生式，并且产生式的右部不含非终结符。

规则	文法产生式	正则式
规则1	A→xB, B→y	A=xy
规则2	A→xA | y	A=x*y
规则3	A→x, A→y	A=x|y

例：有文法G [s]

S→a A | a

A→a A | d A | a | d

于是：S = a A | a

A= (a A | d A) | (a | d)

⇒ A = (a | d) A | (a | d)

由规则二：A= (a | d)* (a | d)

代入：S = a ((a | d)* (a | d)) | a

于是：S = a ((a | d)* (a | d) |ε)

（6）正则式 ⇒ 正则文法

算法:

1) 对任何正则式 r，选择一个非终结符S作为识别符号，并产生产生式 S→r

2) 若x, y是正则式，对形为A→x y的产生式，重写为A→x B和B→y，其中B为新的非终结符，$B \in V_N$

同样，对于 A→x* y ⇒ A→x A A→y

A→x | y ⇒ A→x A→y

例: 将S = a (a | d)* 转换成相应的正则文法

解: 1) S→a(a|d)*

2) S→aA
A→(a|d)*

3) S→aA
A→(a|d)A
A→ε

4) S→aA
A→aA|dA
A→ε

例：若将S = a (a | d)* 转换成左线性的正则文法

算法：

若x, y是正则式，对形为A→x y的产生式，重写为

A→B y 和B→x ，其中B为新的非终结符，$B \in V_N$

同样，对于 A→ x y * ⇒ A→A y A→x

A→x | y ⇒ A→x A→y

解：1) S→a(a|d)*

2) S→S(a|d)

S→a

3) S→Sa

S→Sd

S→a

但一般都不转化为左线性文法，而是转化为右线性的！

——原因是左线性文法将递归死循环。

补充：DFA的简化（极小化）

“对于任一个DFA，存在一个唯一的状态最少的等价的DFA”

一个有穷自动机是化简的 ⇔ 它没有多余状态并且它的状态中没有两个是互相等价的。

一个有穷自动机可以通过消除多余状态和合并等价状态而转换成一个最小的与之等价的有穷自动机。

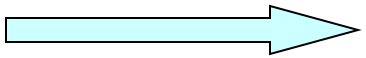
定义:

(1) 有穷自动机的多余状态: 从该自动机的开始状态出发，任何输入串也不能到达那个状态。

例:

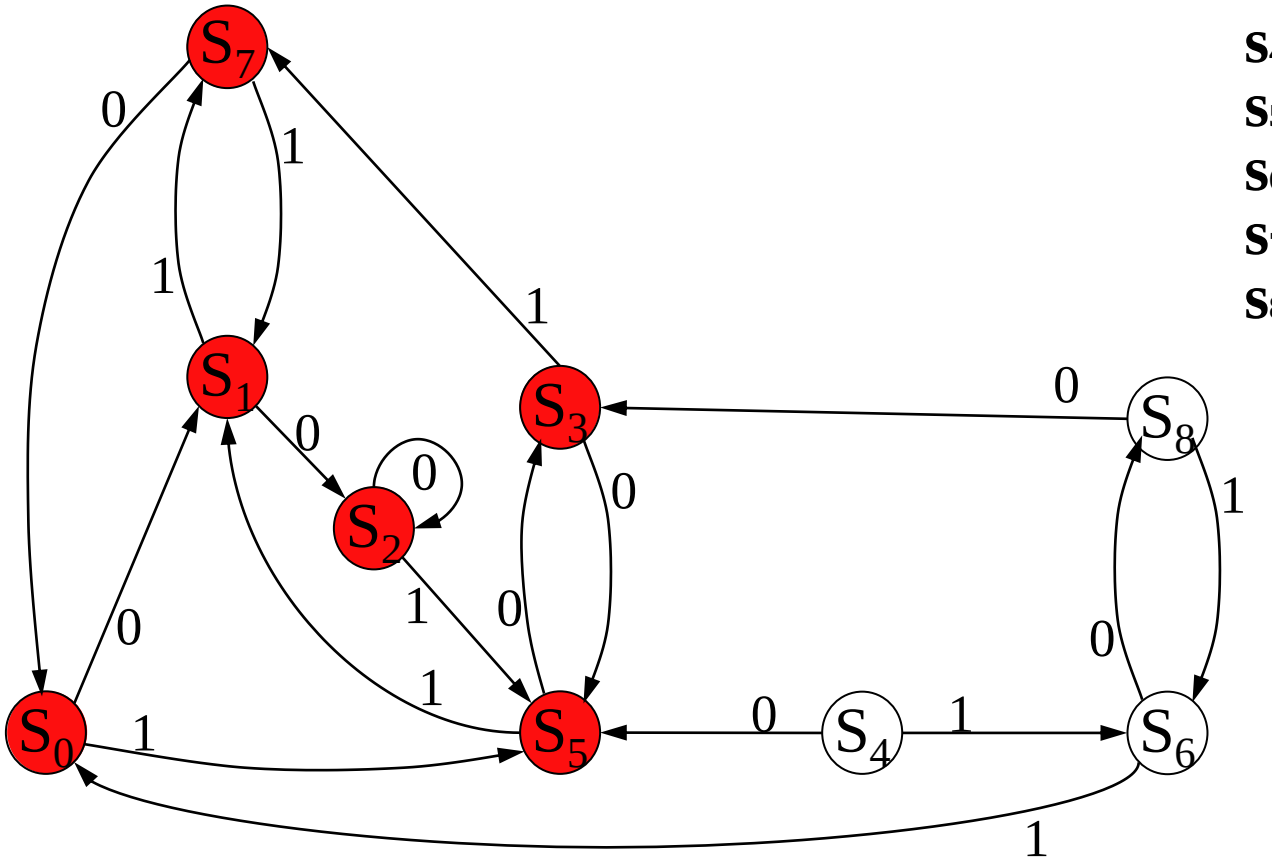
	0	1
S_0	S_1	S_5
S_1	S_2	S_7
S_2	S_2	S_5
S_3	S_5	S_7
S_4	S_5	S_6
S_5	S_3	S_1
S_6	S_8	S_0
S_7	S_0	S_1
S_8	S_3	S_6

画状态图可以看出S_4, S_6, S_8为不可达状态，应该消除。



	0	1
S_0	S_1	S_5
S_1	S_2	S_1
S_2	S_2	S_5
S_3	S_5	S_1
S_5	S_3	S_1
S_7	S_0	S_1

	0	1
S_0	S_1	S_5
S_1	S_2	S_7
S_2	S_2	S_5
S_3	S_5	S_7
S_4	S_5	S_6
S_5	S_3	S_1
S_6	S_8	S_0
S_7	S_0	S_1
S_8	S_3	S_6



(2) 等价状态 $\Longleftrightarrow$ 状态 s 和 t 的等价条件是:

1) 一致性条件: 状态 s 和 t 必须同时为可接受状态或不接受状态。

2) 蔓延性条件: 对于所有输入符号，状态 s 和 t 必须转换到等价的状态里。

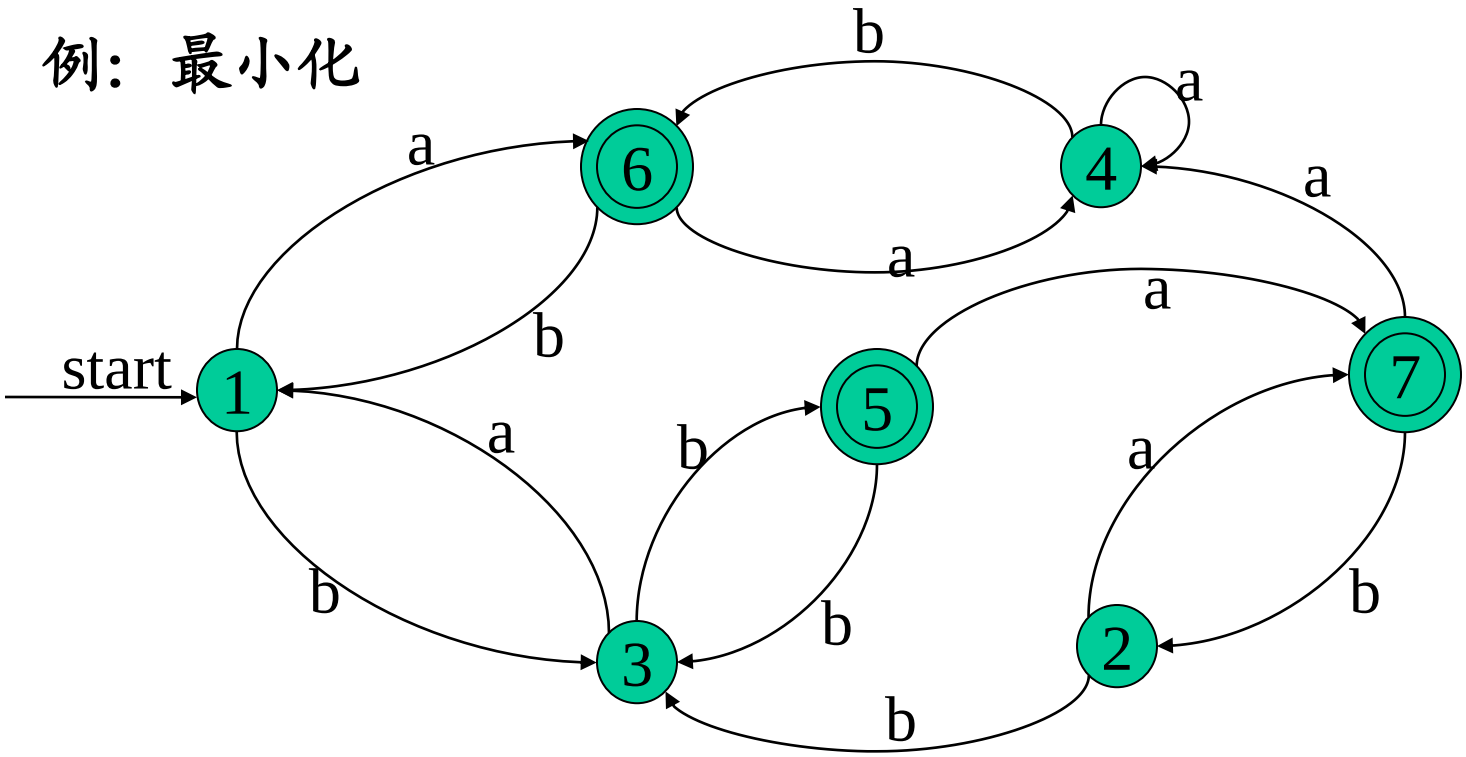
对于所有输入符号c，$I_c(s) = I_c(t)$，即状态 s、t 对于c具有相同的后继，则称 s，t 是等价的。

（任何有后继的状态和任何无后继的状态一定不等价！）

若有穷自动机的状态 s 和 t 不等价，则称这两个状态是可区别的。

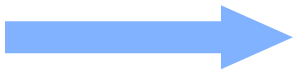
“分割法”：把一个DFA（不含多余状态）的状态分割成一些不相关的子集，使得任何不同的两个子集状态都是可区别的，而同一个子集中的任何状态都是等价的。

例：最小化

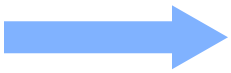


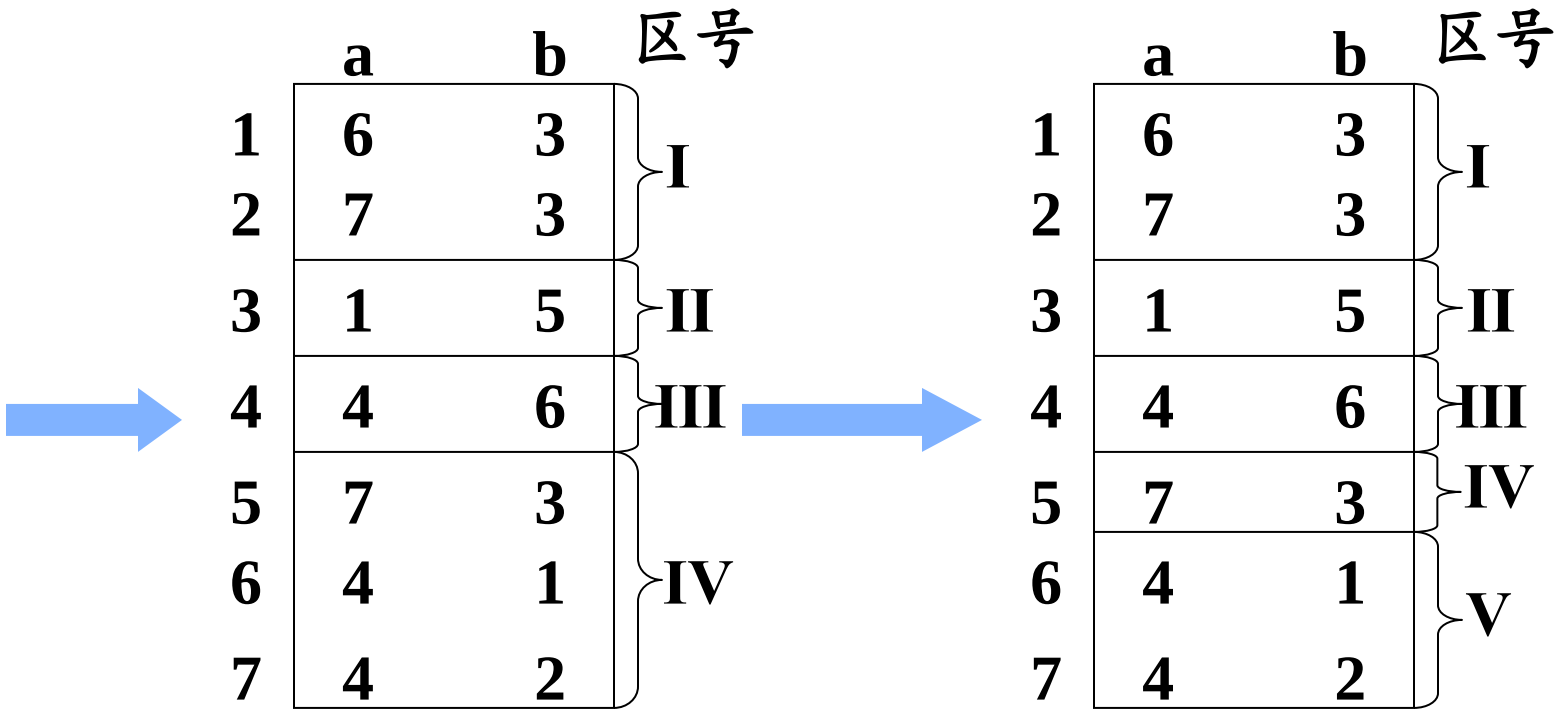
解：(一) 区分终态与非终态

	a	b	区号
1	II	I	I
2	II	I	
3	I	II	
4	I	II	
5	7	3	II
6	4	1	
7	4	2	



	a	b	区号
1	6	3	I
2	7	3	
3	1	5	II
4	4	6	
5	7	3	III
6	4	1	
7	4	2	




a
b
区号
1 6 3 I
2 7 3
3 1 5 II
4 4 6 III
5 7 3 IV
6 4 1
7 4 2
a
b
区号
1 6 3 I
2 7 3
3 1 5 II
4 4 6 III
5 7 3 IV
6 4 1 V
7 4 2

将区号代替状态号得:

	a	b
1	5	2
2	1	4
3	3	5
4	5	2
5	3	1