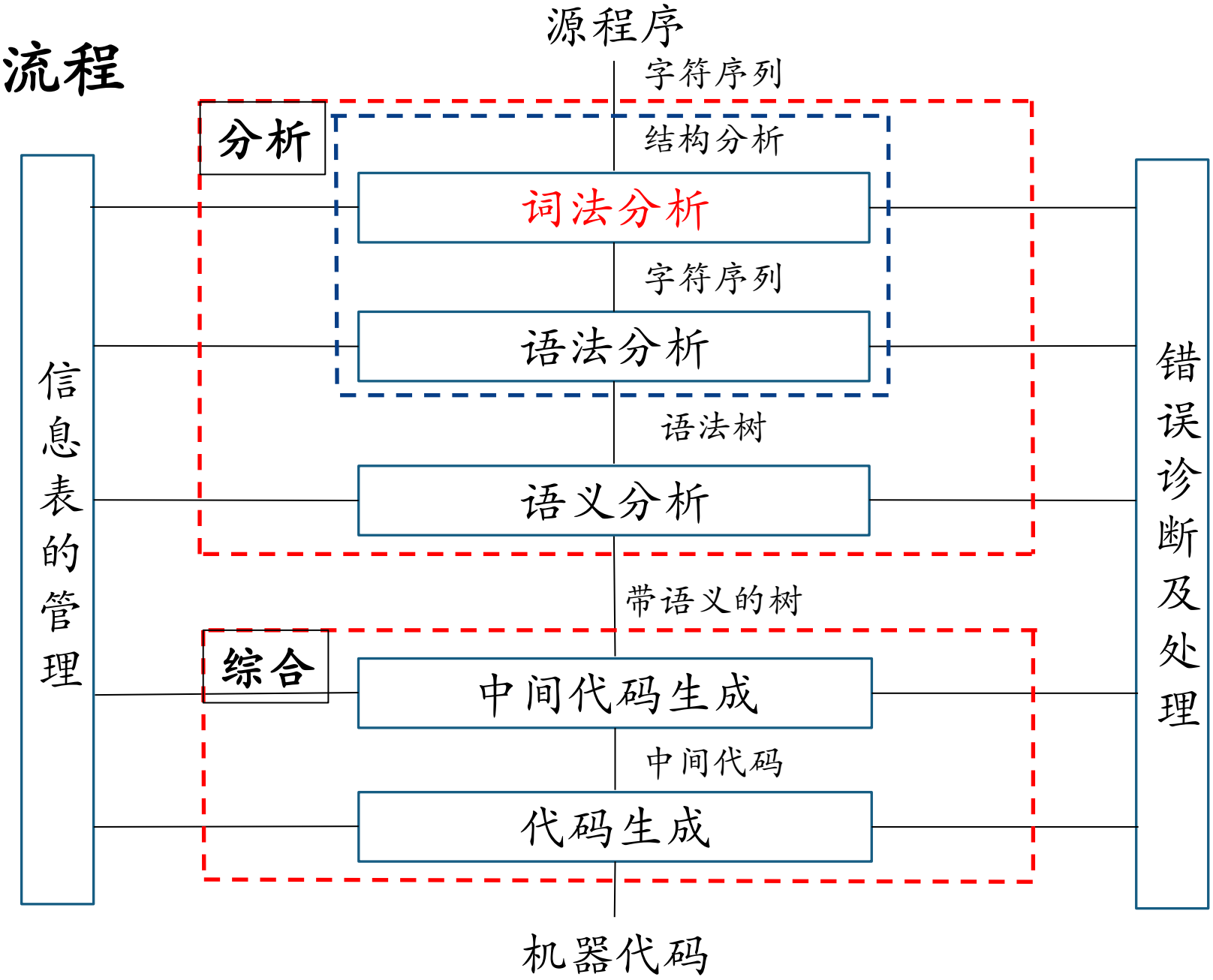
编译流程


源程序
字符序列
分析
结构分析
词法分析
字符序列
语法分析
语法树
语义分析
带语义的树
综合
中间代码生成
中间代码
代码生成
机器代码
信息表的管理
错误诊断及处理

第三章 词法分析器

信息与软件工程学院

邓伏虎

第三章 词法分析

词法分析的任务:

从左至右逐个字符地对源程序进行扫描，产生一个个单词符号。

词法分析器（Lexical Analyzer）:

又称扫描器，是执行词法分析的程序。

本章内容

3.1 词法分析器概述

3.2 单词的识别

3.3 状态转换图

3.4 正规表达式

3.5 有限状态转换机

3.6 词法分析器生成器LEX

3.1 词法分析器概述

* 词法分析器的功能：

  输入源程序，输出单词符号的集合。

* 单词符号的种类：
  - 基本字：如 if、else、case等
  - 标识符：表示各种名字，如变量名、数组名及过程名等
  - 常数：各种类型的常数
  - 运算符：+、-、*、/等
  - 界符：分号、花括号、空格、回车等

3.1 词法分析概述

* 输出的单词符号表示形式:
  （单词种别，单词自身的值）

* 单词种别通常用整数编码表示，称为**种别编码**、**类别编码**。
  - 若一个种别只有一个单词符号，则种别编码就代表该单词符号。通常假定基本字、运算符和界符都是**一符一种**。
  - 若一个种别有多个单词符号，则对每一个单词符号，分别给出种别编码和自身的值。

3.1 词法分析概述

表 3-1 单词符号编码表

单词符号	类别编码	助记符		单词符号	类别编码	助记符
标识符	1	$SYMBOL		<	14	$L
常数（整型）	2	$CONSTANT		<=	15	$LE
int	3	$INT		>	16	$G
if	4	$IF		>=	17	$GE
else	5	$ELSE		!=	18	$NE
while	6	$WHILE		==	19	$E
for	7	$FOR		=	20	$ASSIGN
read	8	$READ		(	21	$LPAR
write	9	$WRITE		)	22	$RPAR
+	10	$ADD		,	23	$COM
-	11	$SUB		;	24	$SEM
*	12	$MUL		space	25	$SPACE
/	13	$DIV		/t	26	$TAB

3.2 单词的识别

例：从以下源代码中识别单词符号

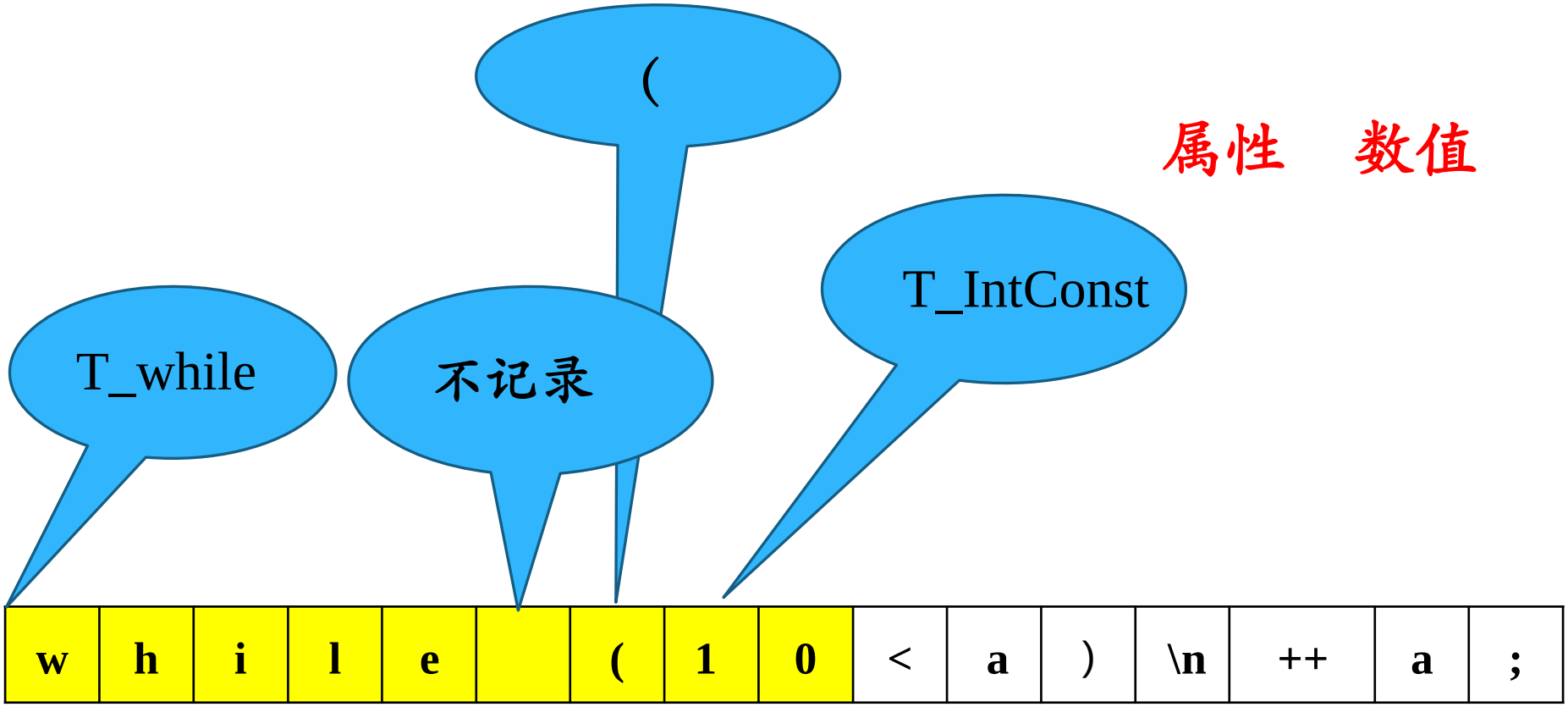
```
int student1_score=90,Class_score=500;
```

(int的编码（3），-)
(空格的编码（25），-)
(标识符的编码（1），标识符的名称“student1_score”）
(=的编码（20），-)
(常数的编码（2），0101 1010)
(，的编码(23)，－）
(标识符的编码（1），标识符的名称“Class_score”)
(＝的编码(20)，－)
(常数的编码(2)，0001 1111 0100)
(；的编码(24)，－)

Token

源程序中的一个独立单位

属性　数值



3.2 单词的识别

难点

C++中的嵌套的模板声明

vector< vector< int >> myVector

vector< vector< int > > myVector

PL/1中关键字可被用作标识符

IF THEN THEN THEN = ELSE; ELSE ELSE = IF

IF THEN THEN THEN = ELSE; ELSE ELSE = IF

思考

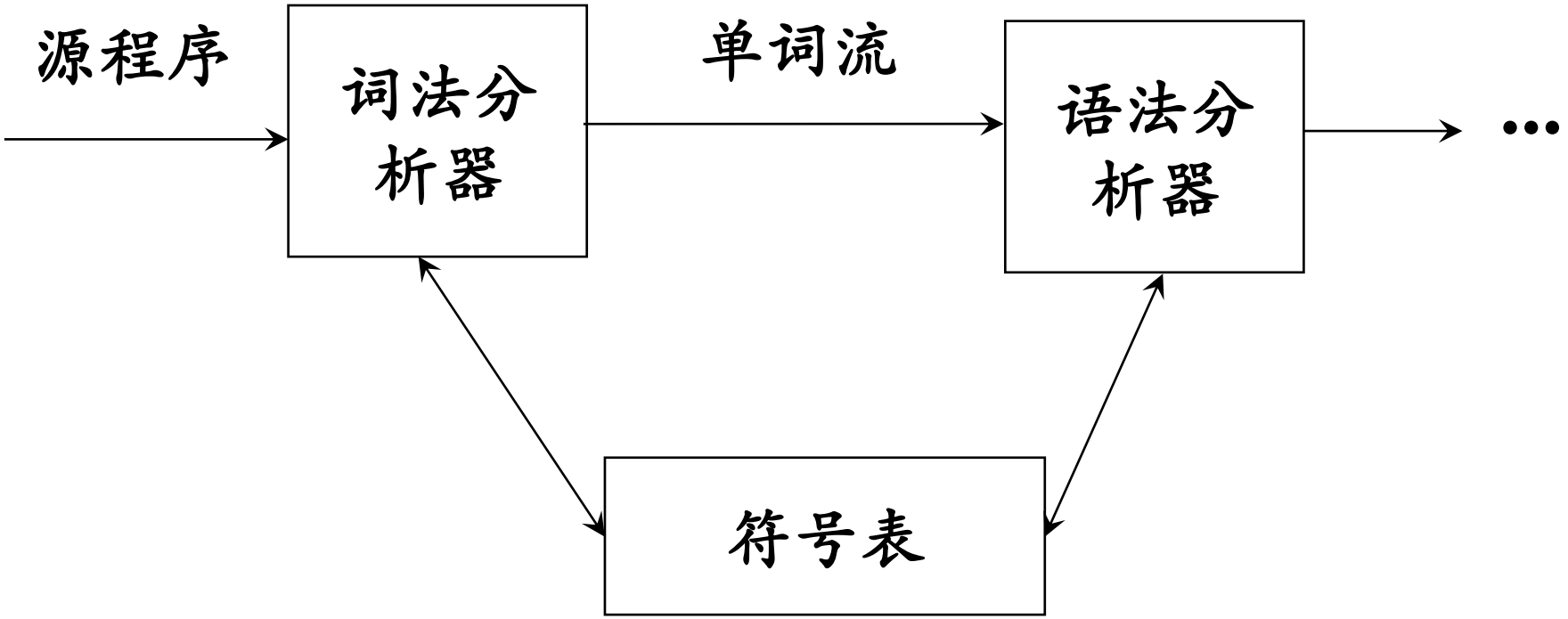
词法分析作为一个独立的阶段，是否应当将其处理为一遍呢？

思考

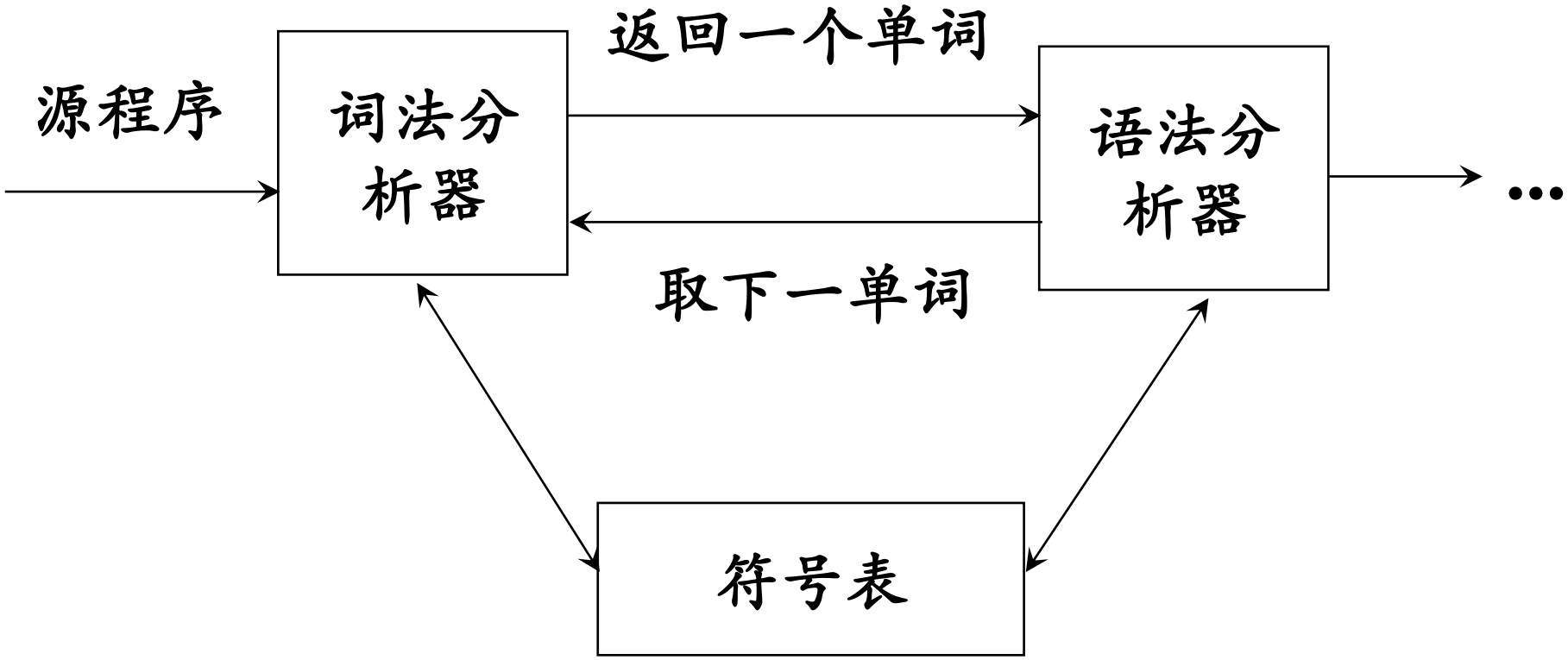
词法分析作为一个独立的阶段，是否应当将其处理为一遍呢?

- 作为独立阶段的优点：结构简洁、清晰和条理化，有利于集中考虑词法分析一些枝节问题。
- 不作为一遍，将其处理为一个子程序。

词法分析器与语法分析器



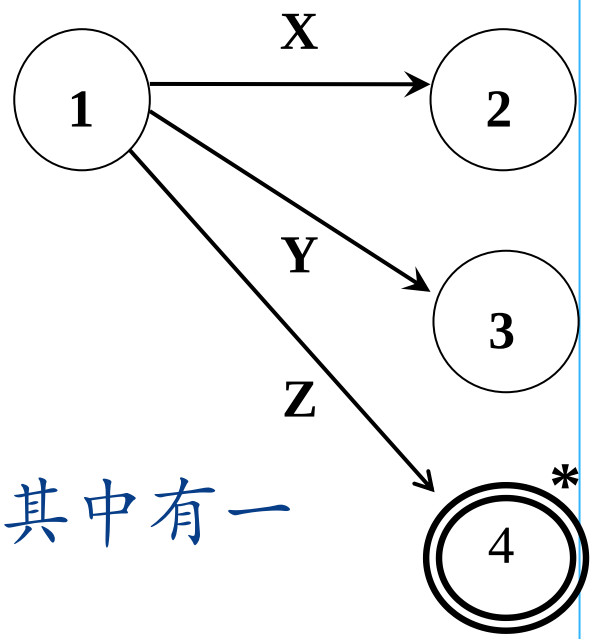
词法分析器与语法分析器



3.3 状态转换图

状态转换图是一张有限方向图。

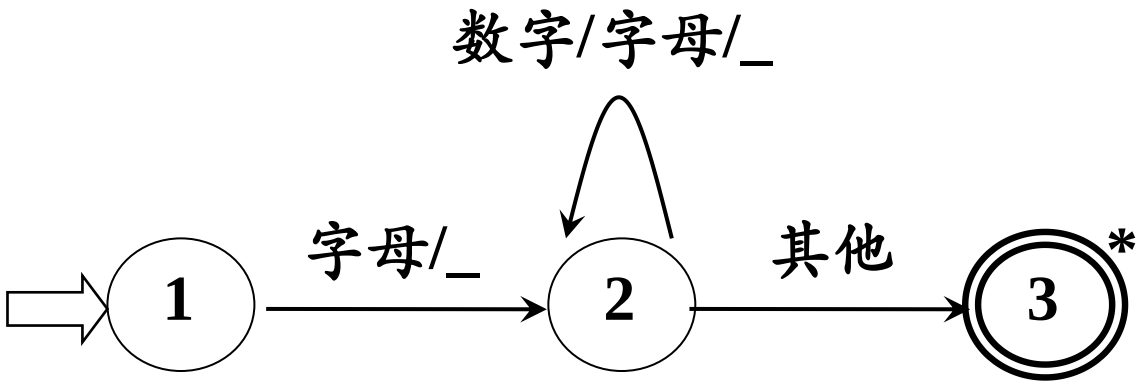
- 结点代表状态，用圆圈表示。
- 状态之间用箭弧连结，箭弧上的标记(字符)代表射出结状态下可能出现的输入字符或字符类。
- 一张转换图只包含有限个状态，其中有一个为初态，至少要有一个终态。



3.3 状态转换图

标识符格式：

字母/_　　数字/字母/_ …… 数字/字母/_　　其他字符

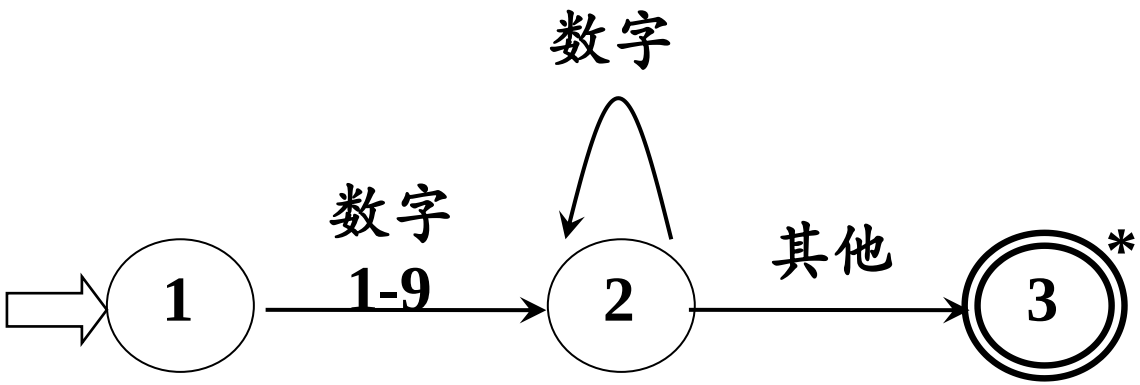


问题：如何识别保留字？

3.3 状态转换图

正整常数格式：

数字
1-9　　数字 数字 数字 …… 数字 其他字符

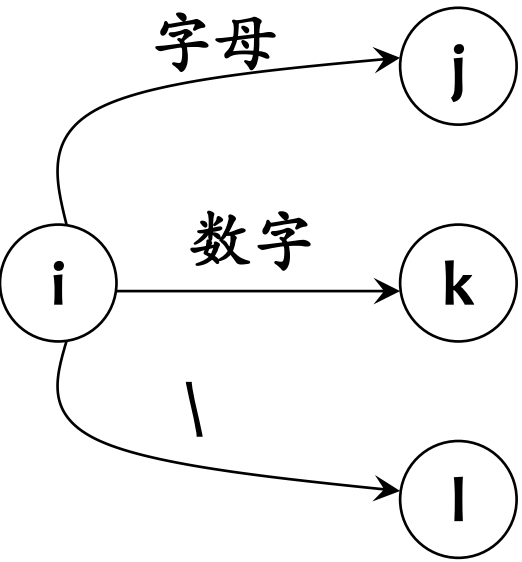


常数的状态转换图？

3.3 状态转换图

问题：状态转换图与程序的关系？

猜想：每一个状态结对应一小段程序。

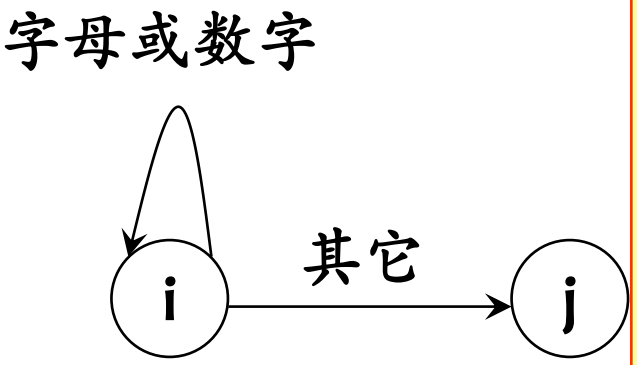


* 对不含回路的分叉结，可用一组IF-THEN-ELSE语句或一个CASE语句实

```
getChar( );
if (letter( )) {…状态j的对应程序段…;}
else if (digit( )) {…状态k的对应程序段…;}
else if (ch='\') {…状态l的对应程序段…;}
else {…错误处理…;}
```

3.3 状态转换图

对含有回路的状态结，对应的程序又是怎么样的呢?

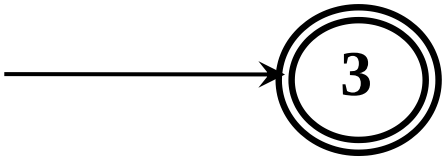



```
getChar( );
while (letter( ) or digit( )){
        getChar( );
        if(其他)
        …状态j的对应程序段…}
```

一段由WHILE结构和IF语句构成的程序

3.3 状态转换图

终态结点对应的程序

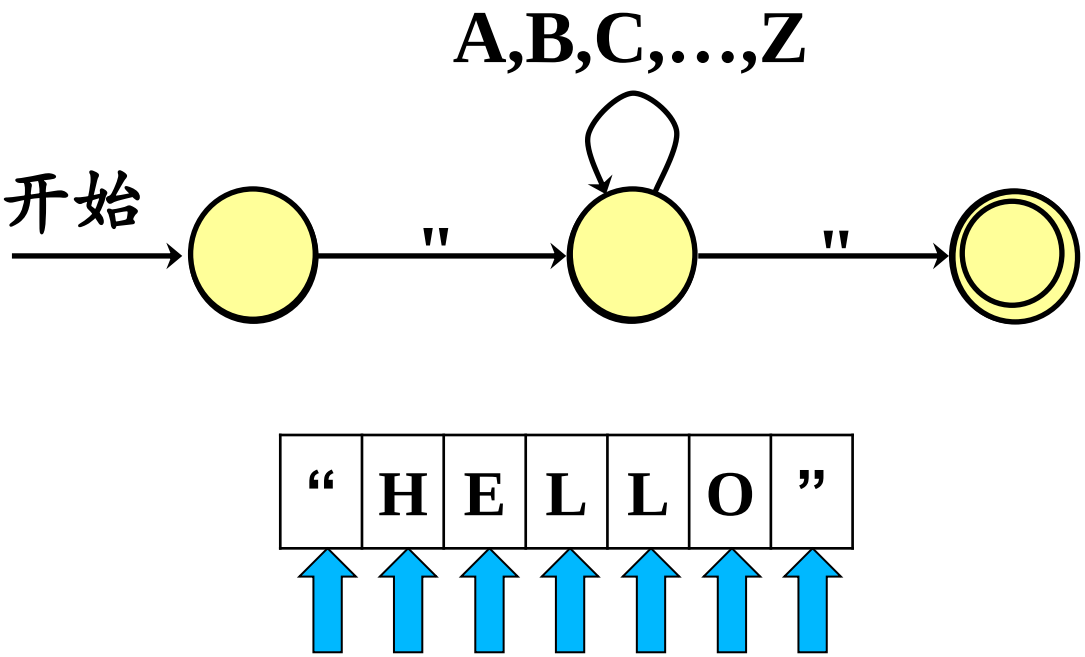


终态表示识别出某种单词符号，因此，对应的语句为

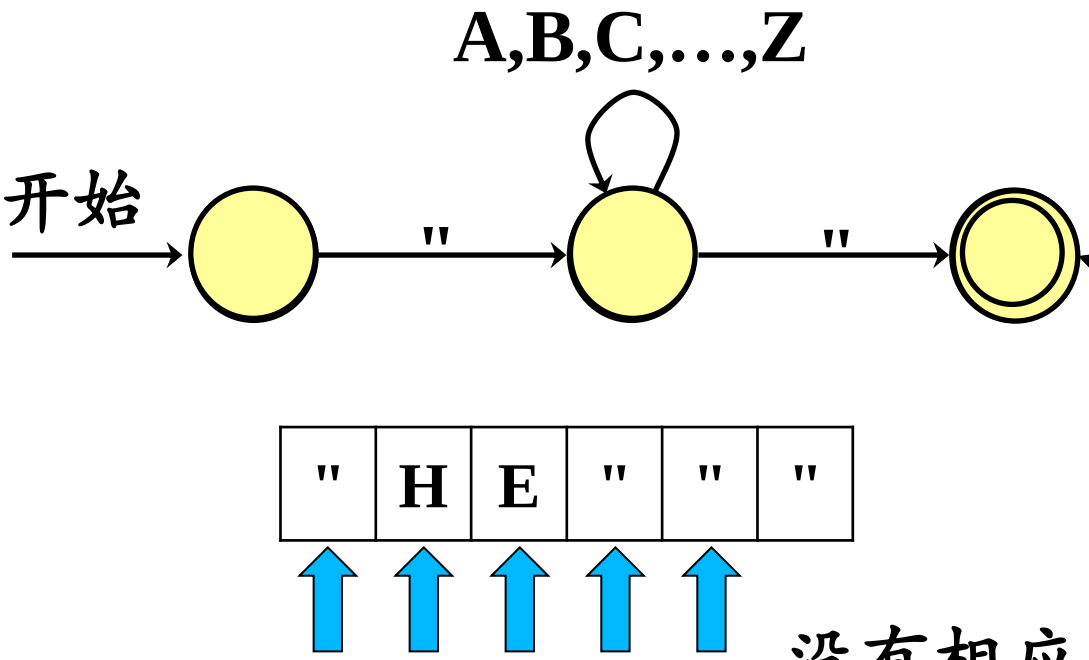
return (C，VAL);

其中，C为单词种别，VAL为单词属性值.

3.3 状态转换图



3.3 状态转换图

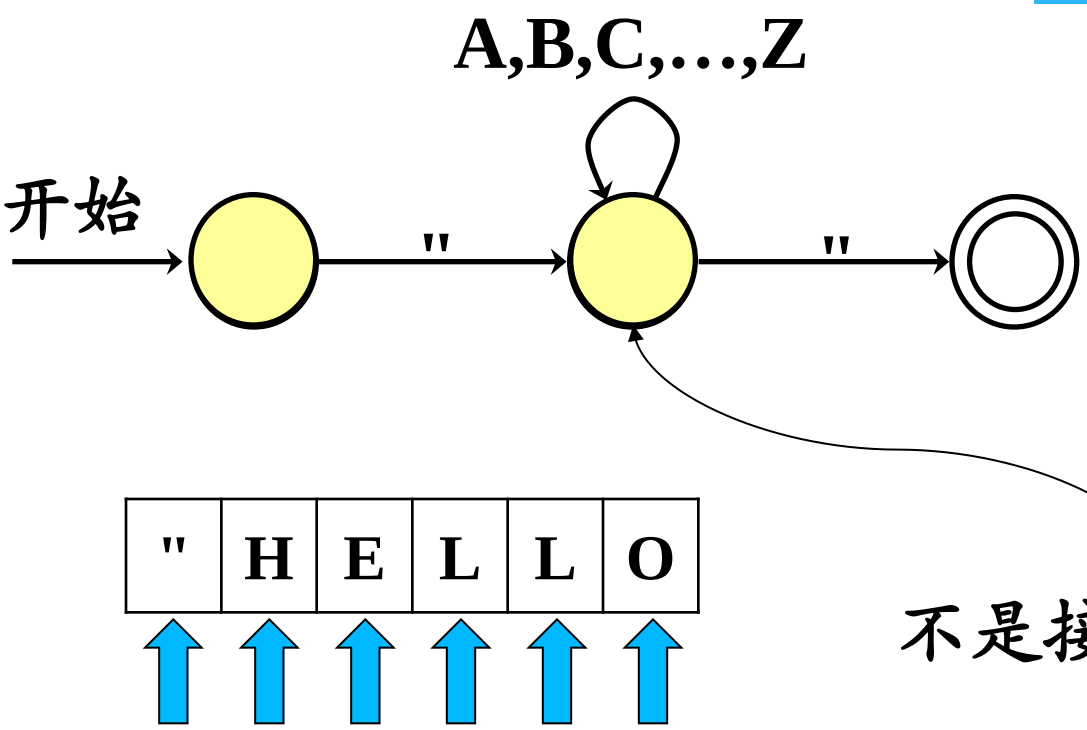


没有相应的转换状态，

停机

拒绝输入串

3.3 状态转换图



3.4 正则表达式

一种用来描述字符串集合的工具

几个概念：

字母表：一个有限的符号集合
串：字母表中符号的一个有穷序列
串s的长度，记作|s|，指s中出现的符号的次数
空串：长度为0的串，记作 ε
闭包：用a*来表示由零个或者多个a构成的串
正闭包：用a+来表示由一个或者多个a构成的串
语言：给定字母表上的一个任意的可数的串的集合

3.4 正则表达式

定义

假设Σ为一个有限字母表，在Σ上的正则表达式可递归地定义如下：

（1）ε和Ø是Σ上的正则表达式，它们表示的集合分别为{ε}和Ø。

（2）如果a是字母表Σ中的一个符号，那么a是Σ上的正则表达式，它表示的集合为{a}。

（3）如果r，s都是字母表Σ上的正则表达式，那么$r|s$、rs、r*也是字母表Σ上的正则表达式，它们表示的集合分别是：RUS，RS，R^*。

3.4 正则表达式

正则表达式	表示的集合
a	{a}
a|b	{a,b}
ab	{ab}
0(0|1)	{00,01}
a*	{ε, a, aa, aaa, …}
0^+1	{01, 001, 0001, …}
(a|b)*ba	{ba, aaba, ababa, baba,…}

3.4 正则表达式

描述	公理
并是可以交换的	$s\|t = t\|s$
并是可结合的	$s\|(t\|r) = (s\|t)\|r$
连接是可结合的	$(st)r = s(tr)$
连接对并可分配	$s(t\|r) = st\|sr$，　$(t\|r)s = ts\|rs$
ε是连接的恒等元素	$\varepsilon s = s$，$s\varepsilon = s$
闭包与ε间的关系	$s^* = (s\|\varepsilon)^*$
闭包是幂等的	$a^{**} = a^*$

3.4 正则表达式

化简规则：

（1）一元运算符 * 具有最高的优先级，并且是左结合的。

（2）. 连接具有次高的优先级，也是左结合的。

（3）| 的优先级最低，也是左结合的

例：(*a*)|((*b*)*(c))化简

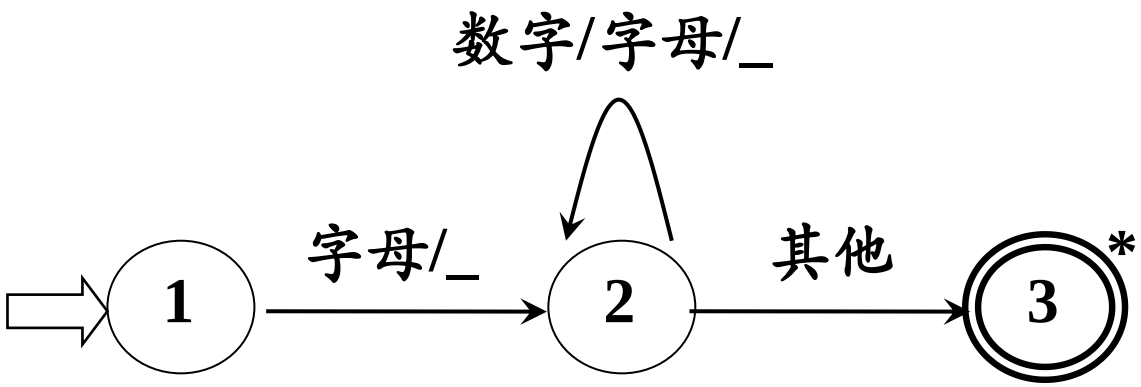
$$a|b*c$$

3.4 正则表达式

标识符的正规式

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

标识符的状态转换图



3.4 正则表达式

假设，Σ={0,1,2,…,9, a, b, c, …, z, A, B, C, …, Z}

数字: digit=0|1|2|…|9

字母: letter=a|b|c|…|z|A|B|C|…|Z

正整数: digit digit* 或digit$^+$

标识符: (letter|_) (letter|digit|_)*

3.5 有限状态自动机

有限状态自动机（Finite Automaton, FA）M是一个五元组：

$$M=(Q, \Sigma, \delta, q_0, F)$$

其中：

Q——状态的非空有限集合。$\forall q \in Q$, q称为M的一个状态。

Σ——输入字母表，输入字符串都是Σ上的字符串。

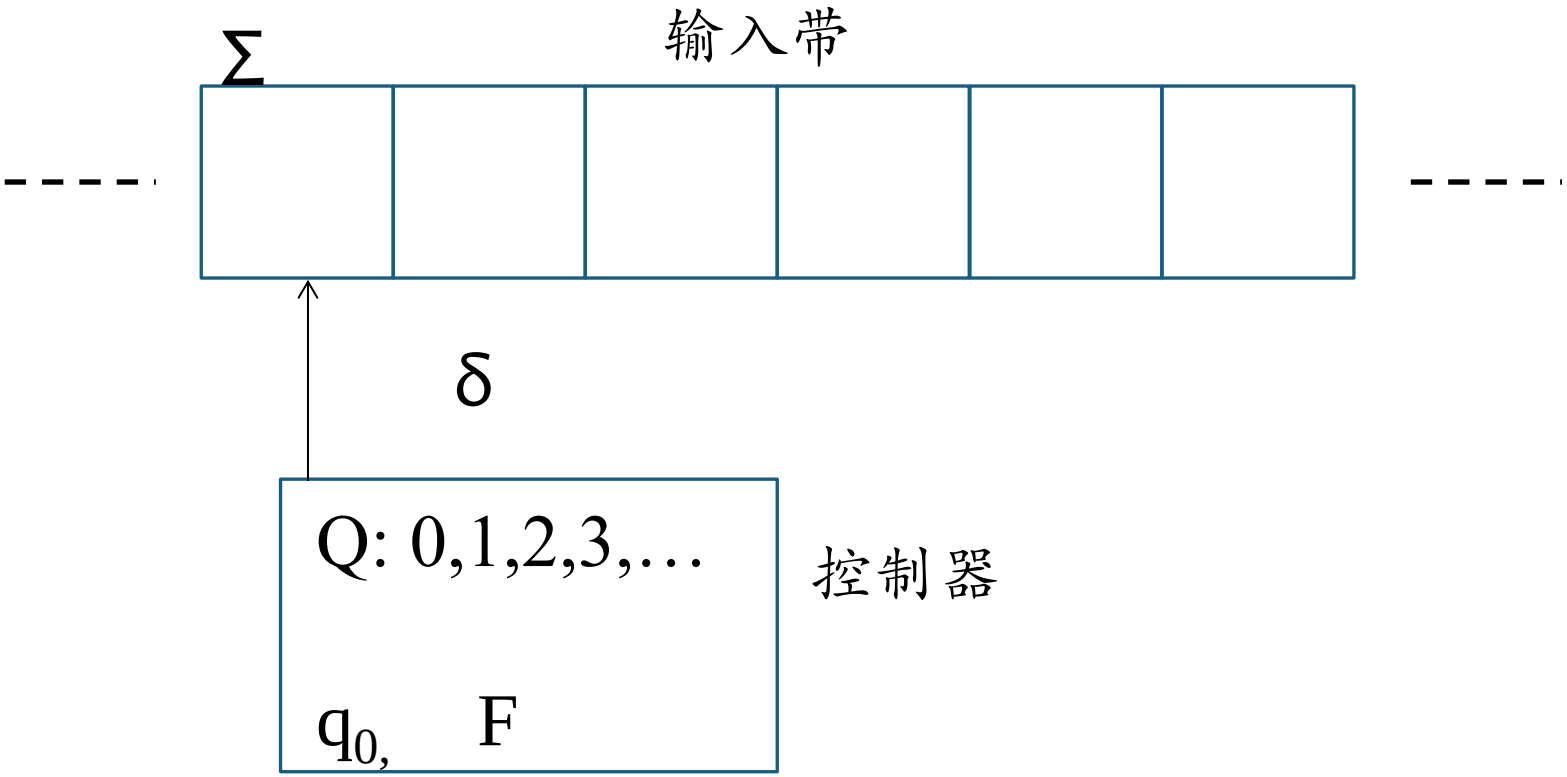
δ——状态转移函数，$\delta: Q \times \Sigma \to Q$，为单值映射。

$\delta(s,a)=s'$表示：在当前状态为s，输入为a的时候，状态将转移到下一状态s'。s'成为s的一个后继状态。

q_0——$q_0 \in Q$，是M的开始状态。

F——$F \subseteq Q$，是M的终止状态集合。$\forall q \in F$，q是M的终止状态。

3.5 有限状态自动机



3.5 有限状态自动机

非确定的有限状态自动机(Non-deterministic Finite Automation, NFA)

在状态q下,读取任意输入字符a,其状态将变成状态集合$\{p_1, p_2, \ldots, p_k\}$中的某一个状态，即 $\delta(q,a)=\{p_1, p_2, \ldots, p_k\}$。NFA中允许存在ε转换。初态可以是多个。

确定的有限状态自动机(Deterministic Finite Automation, DFA)

在状态q下,读取任意输入字符a,其状态将变成唯一确定的状态p,即$\delta(q,a)=p$。初态只能是一个。

3.5 有限状态自动机

M=$(Q, \Sigma, \delta, q_0, F)$，　其中：　δ定义如下：

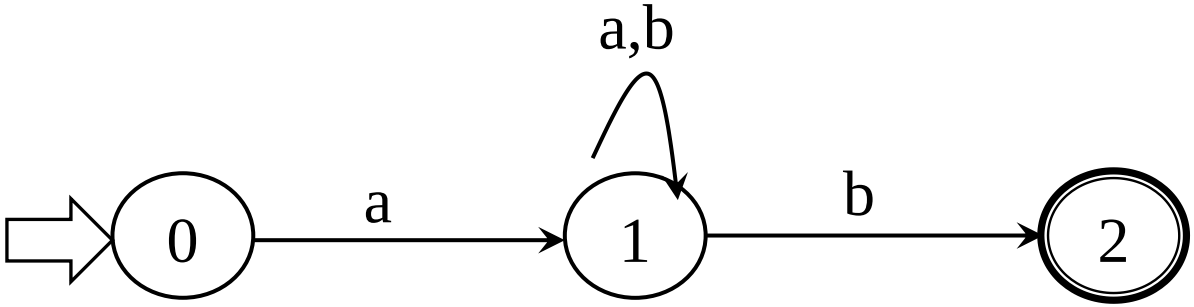
$\delta(0,a)=1$	$\delta(0,b)=$
$\delta(1,a)=1$	$\delta(1,b)=1,2$
$\delta(2,a)=$	$\delta(2,b)=$

状态	输入字符		终态/非终态
	a	***b***	
=>0	1		非终态
1	1	1，2	非终态
2			终态

3.5 有限状态自动机

M=$(Q, \Sigma, \delta, q_0, F)$，　其中：　δ定义如下：

$\delta(0,a)=1$	$\delta(0,b)=$
$\delta(1,a)=1$	$\delta(1,b)=1,2$
$\delta(2,a)=$	$\delta(2,b)=$



状态转换图

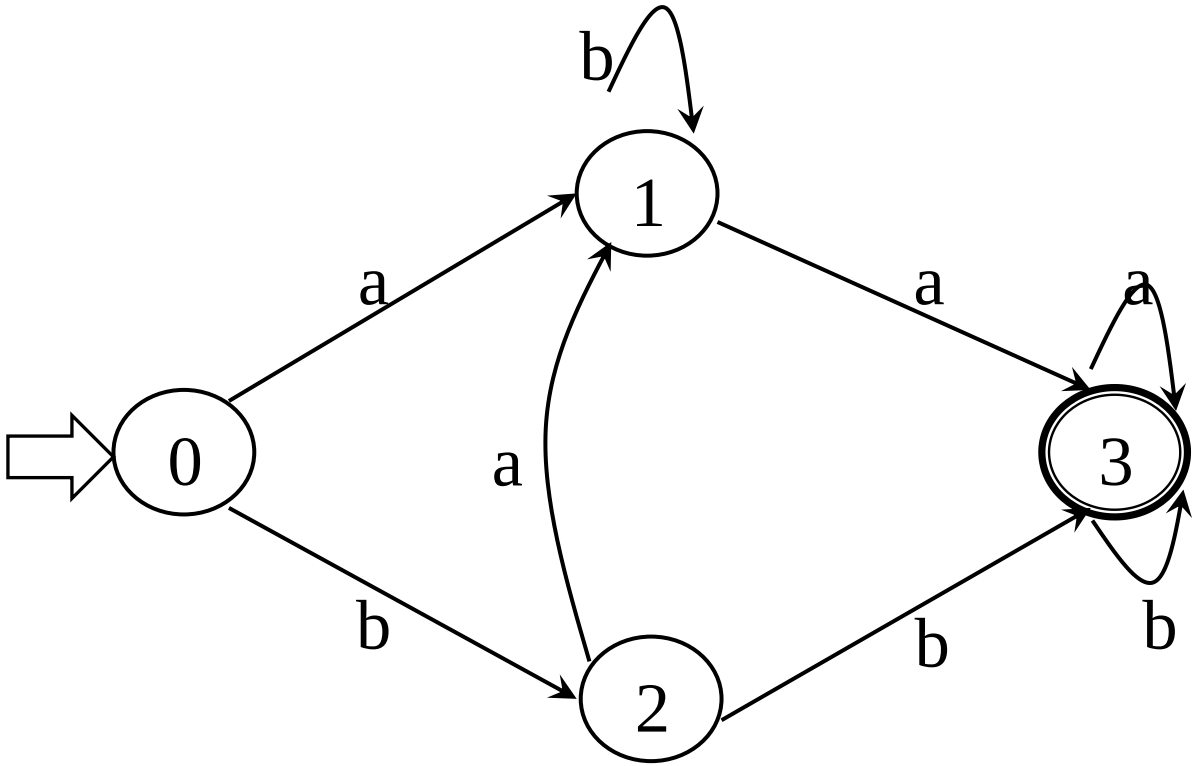
3.5 有限状态自动机

M=({0，1，2，3}，{a，b}，δ，{0}，{3})，其中：δ定义如下：

$\delta(0,a)=1$	$\delta(0,b)=2$
$\delta(1,a)=3$	$\delta(1,b)=1$
$\delta(2,a)=1$	$\delta(2,b)=3$
$\delta(3,a)=3$	$\delta(3,b)=3$

δ	a	b
=>0	1	2
1	3	1
2	1	3
3	3	3

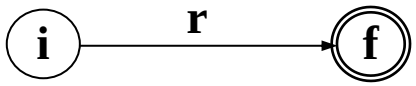
3.5 有限状态自动机



状态转换图

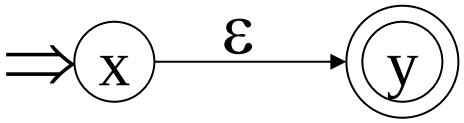
3.5 有限状态自动机

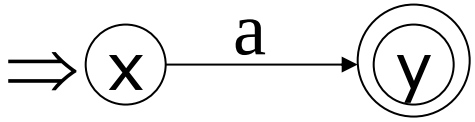
对于每一个正则表达式，可以通过添加一个初态及一个终态

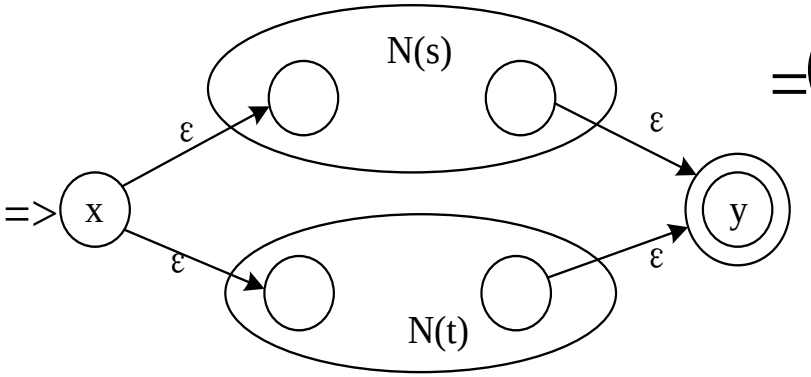


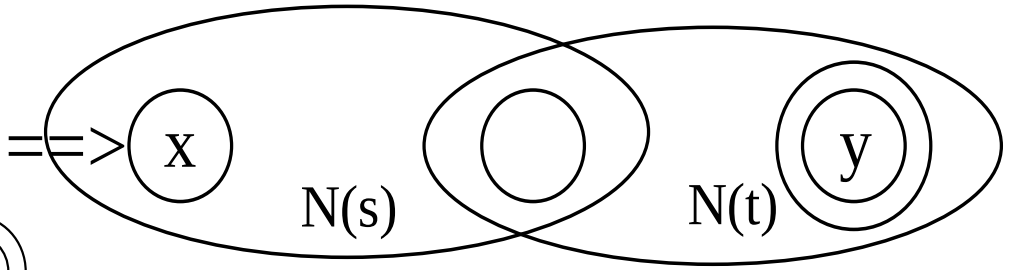
然后对这个正则表达式进行拆分，直到所有的边上都只剩下终结符以及可能的ε

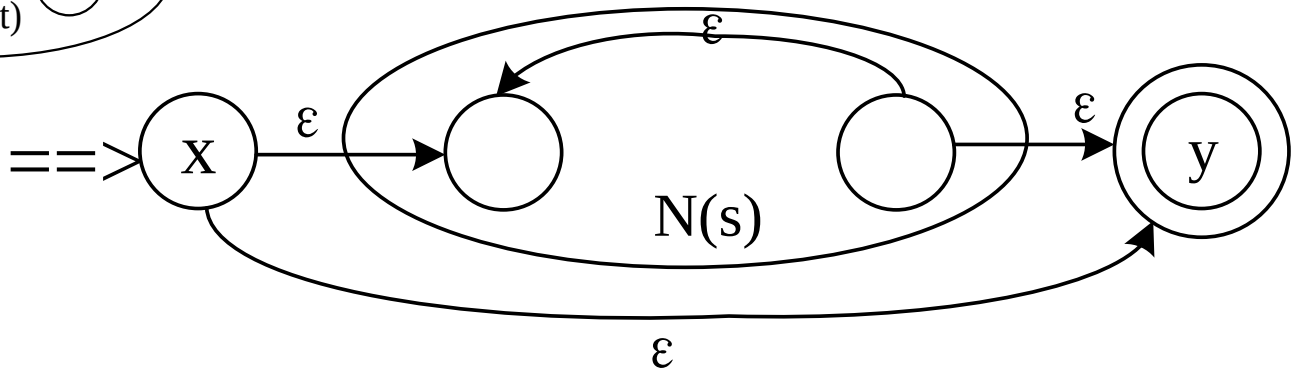
3.5 有限状态自动机









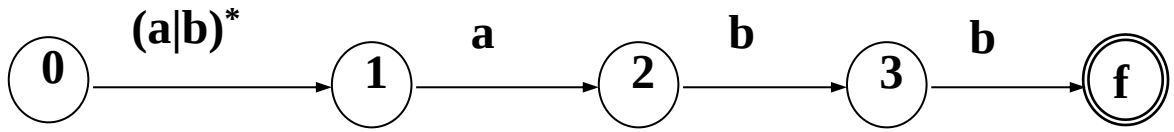


3.5 有限状态自动机

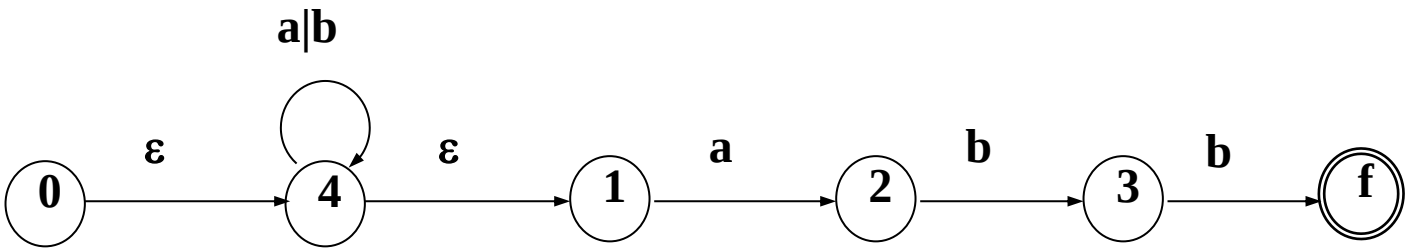
请构造一个识别(a|b)*abb的NFA

(a)

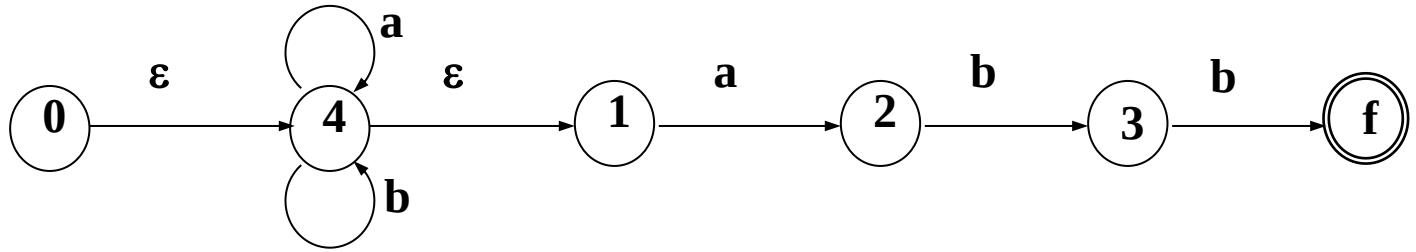
(b)



(c)

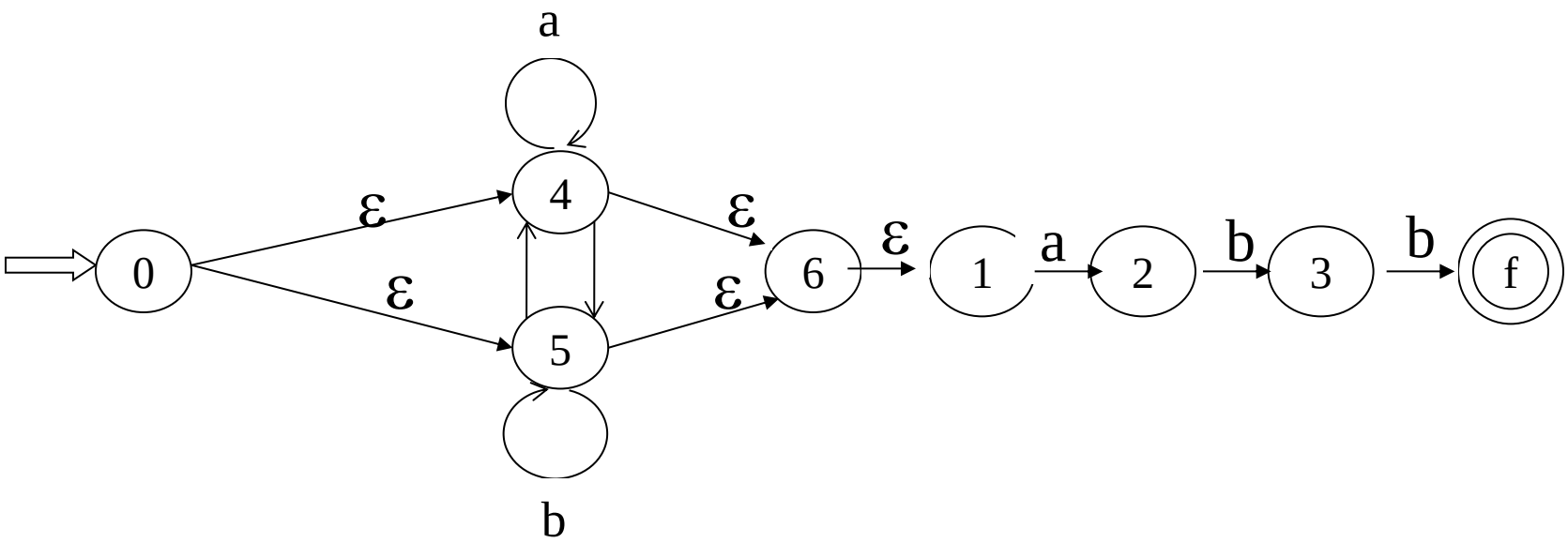


(d)



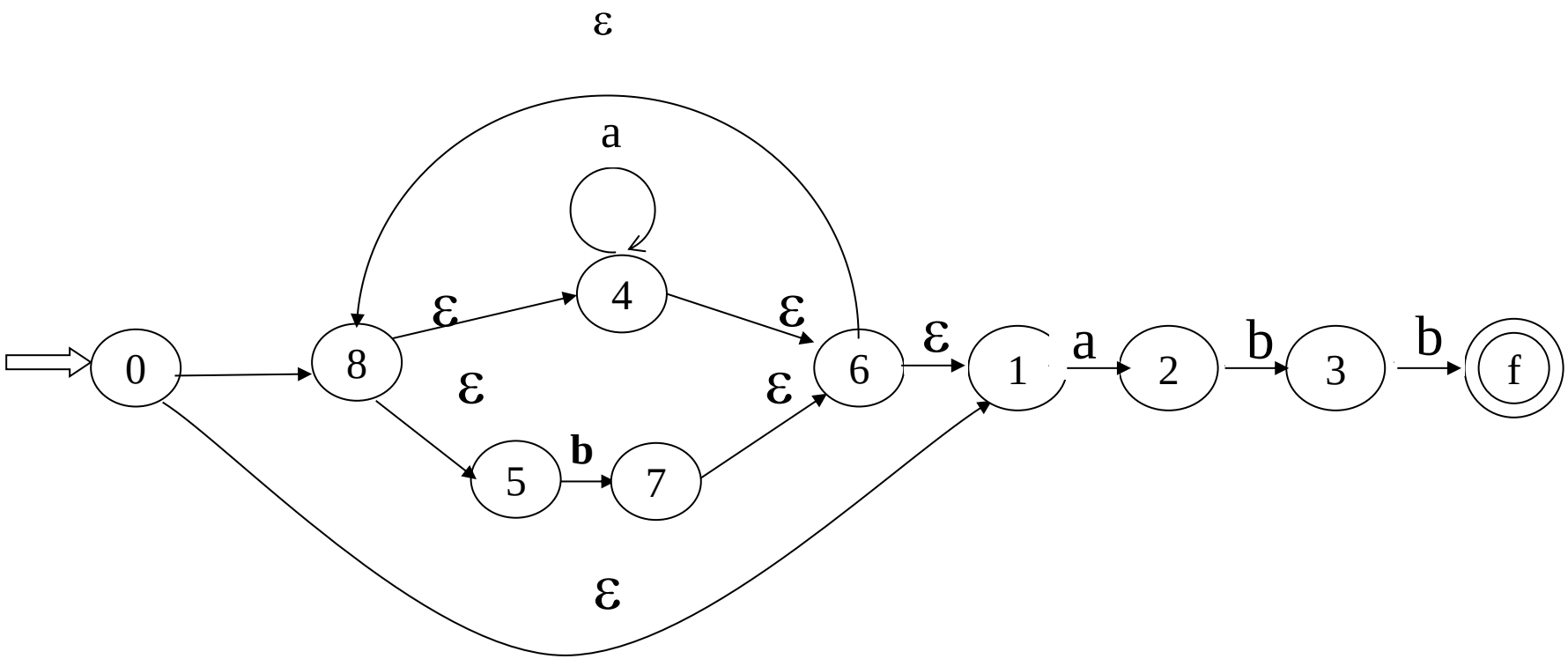
3.5 有限状态自动机

请构造一个识别(a|b)*abb的NFA



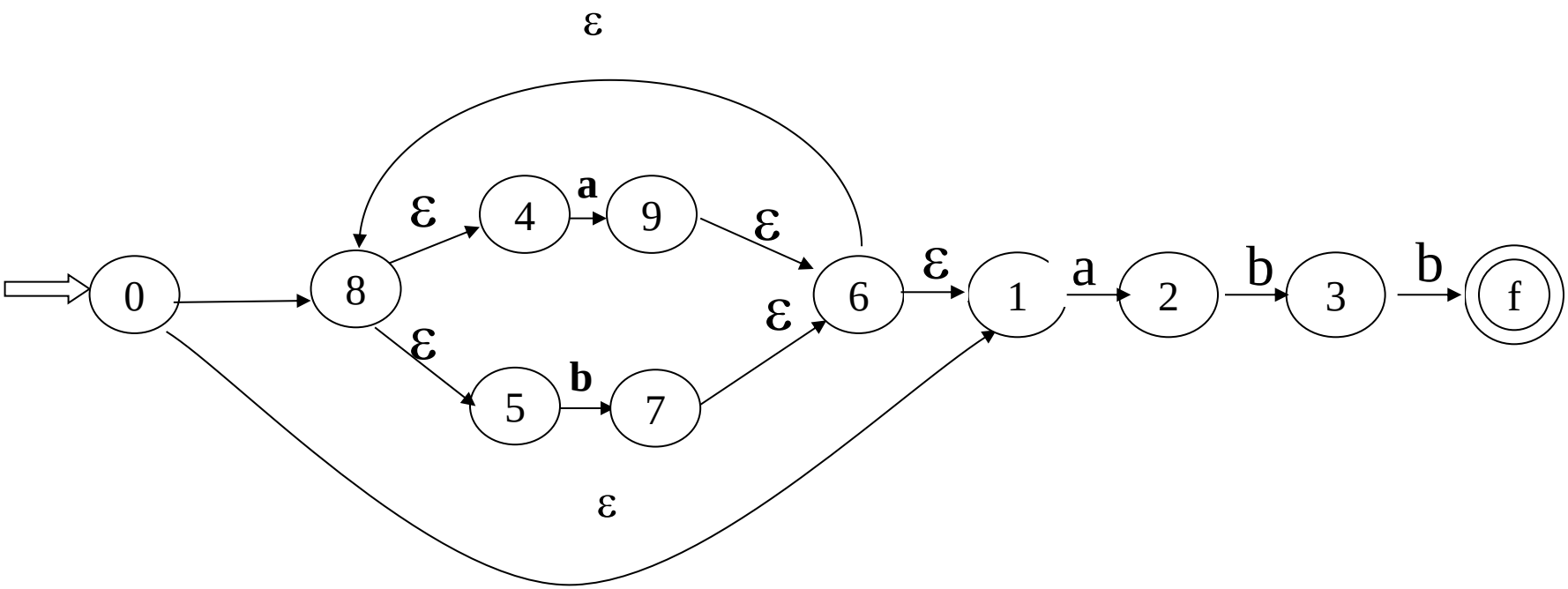
3.5 有限状态自动机

请构造一个识别(a|b)*abb的NFA



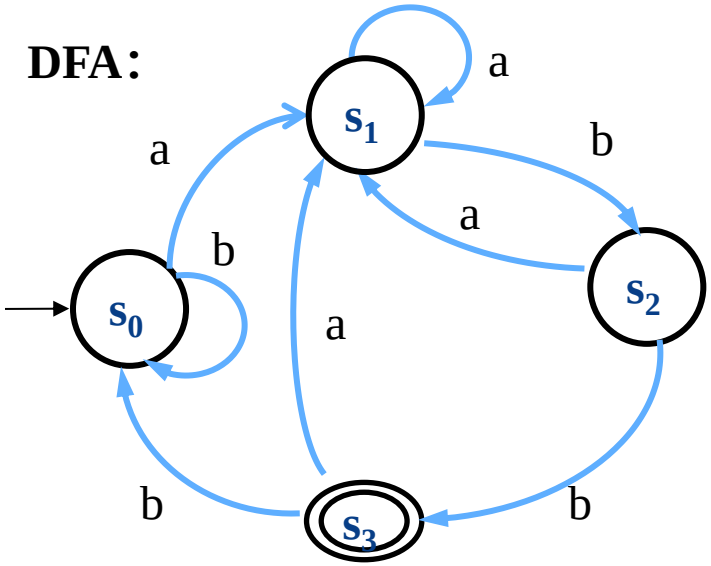
3.5 有限状态自动机

请构造一个识别(a|b)*abb的NFA

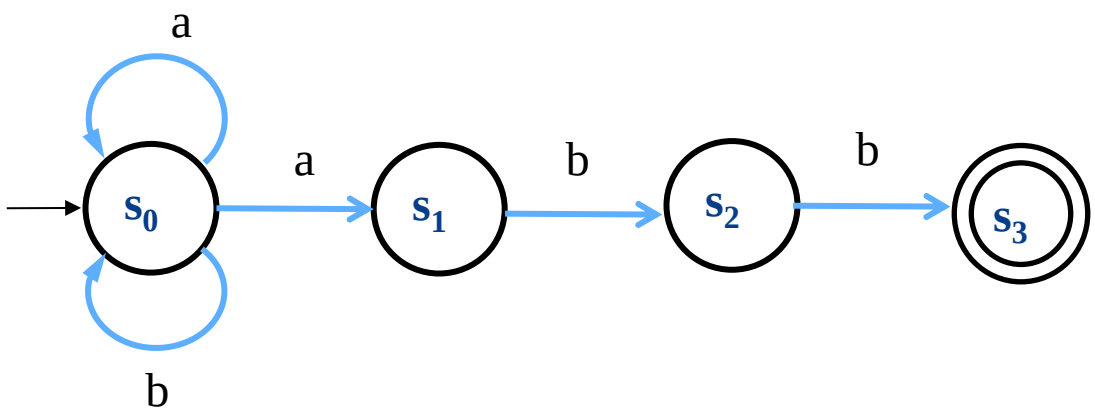


3.5 有限状态自动机

DFA:



NFA:



识别abb　aabb　~~abab~~

接受的语言：(a|b)*abb

对于Σ*中的任何字符串t, 若存在一条从初态到终态的路径（该路径上的输入符依次连接起来就是字符串t），则t为DFA M所接受，或者说t是该DFA可识别的。

DFA M所能接受的字符串的全体称为该自动机识别的语言，记为L(M)

若FA M 和FA M’所识别的语言L(M)=L(M’),则M与M’ 等价。

思考：如何找到与NFA等价的DFA?

是否存在？

如何找到？

3.5 有限状态自动机

* **算法思想**：NFA和DFA的本质区别在于—在某一状态下如果输入某个字符后，NFA可以转换到多个后继状态（是一个后继状态集），而DFA只能转换到一个状态，所以可以让DFA的一个状态来对应NFA的这个后继状态集，这样构造出来的DFA便与NFA等价。

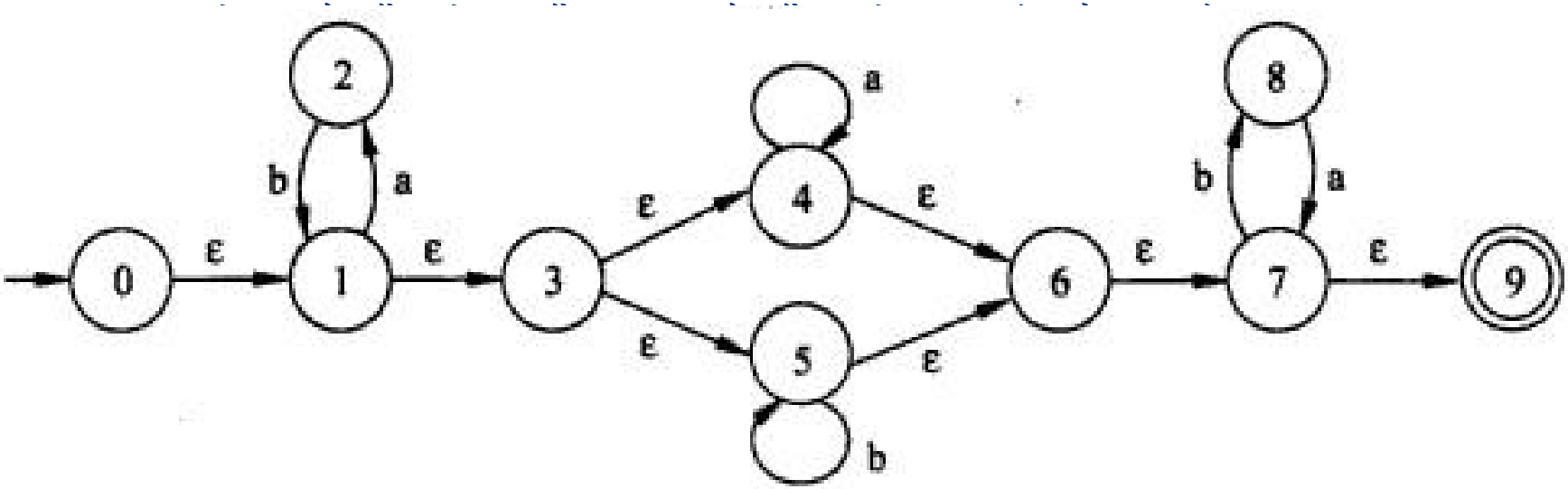
$$K\times(\sum\cup\{\varepsilon\})\rightarrow K\text{子集}\Rightarrow K\times\sum\rightarrow K$$

输入：一个NFA N。

输出：一个接收相同语言的DFA D。

方法：为D构造一个转换表 Dtran

3.5 有限状态自动机

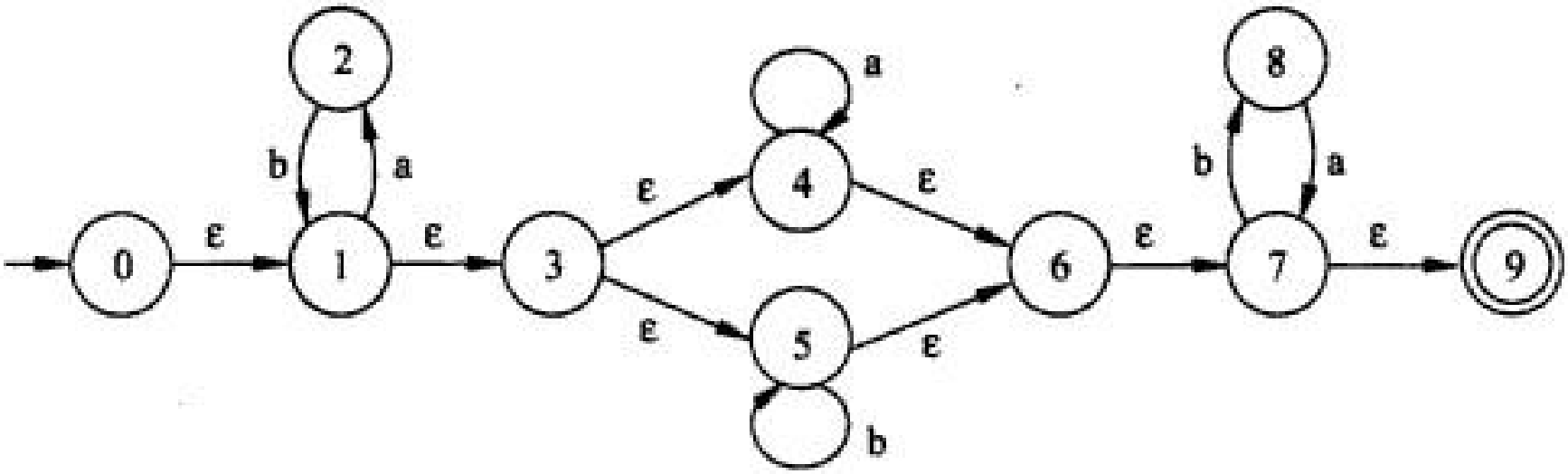


ε-closure(3) = {3, 4, 5, 6, 7, 9}

ε-closure(0, 1, 3) = ε-closure(0) ∪ ε-closure(1) ∪ ε-closure(3)

={0, 1, 3, 4, 5, 6, 7, 9}

3.5 有限状态自动机



move(5, b) = {5}

move([1,2,3,4,5], b) = {1, 5}

3.5 有限状态自动机

必须找出当 NFA *N* 读入了某个输入串之后可能位于的所有状态集合。

在读入第一个输入符号之前

N 可以位于集合ε-*closure*(*s*)中的任何状态上。s为开始状态。

3.5 有限状态自动机

假定 N 在读入输入串 x 之后可以位于集合 T 中的状态上。如果下一个输入符号是a，N的下一状态是?

N可以立即移动到集合$move\ (T, a)$中的任何状态。

由于N 可以在读入 a 后再执行几个 ε 转换，因此。。。

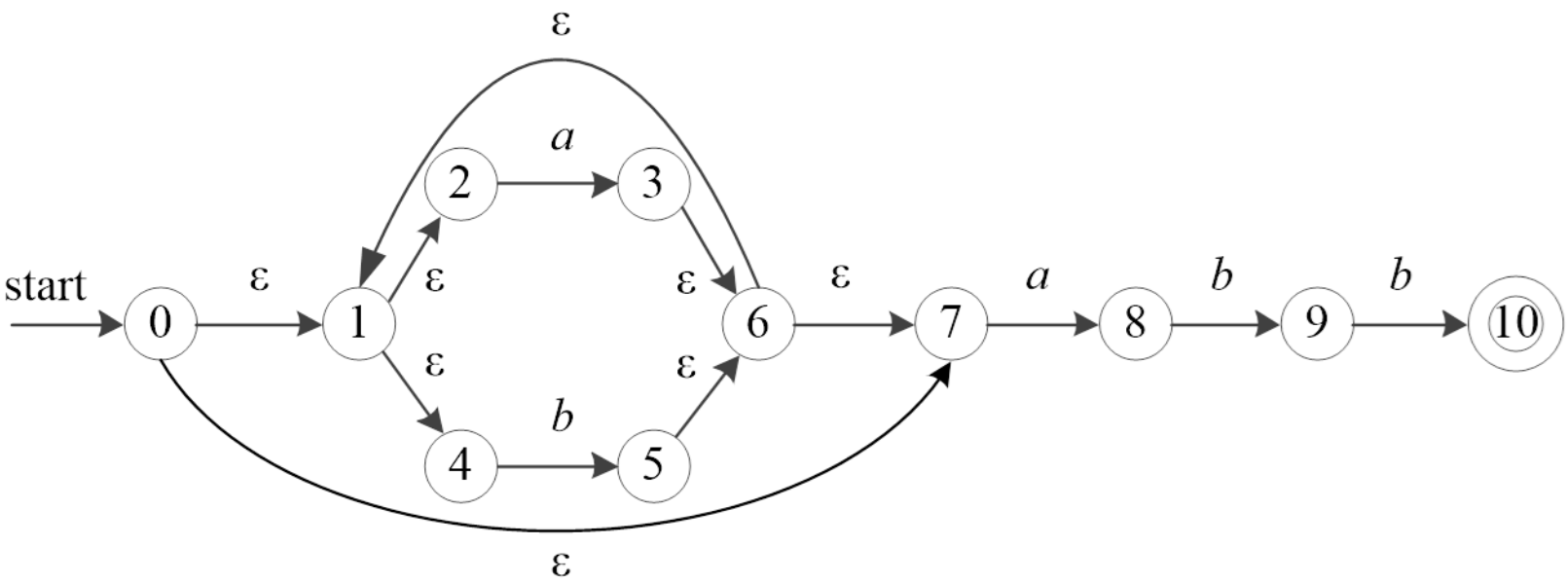
N在读入xa之后可位于$\varepsilon\text{-}closure(move\ (T, a))$中的任何状态上。

$$Dtran[T,a]= \varepsilon\text{-}closure(move\ (T, a))$$

根据这些构想，可以得到构造*D*的状态集合*Dstates*和*D*的转换函数*Dtran* 的算法：

一开始，ε-*closure*(s_0)是D*states*中的唯一状态，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 = ε-*closure*(*move* (*T*, *a*));
        if (*U*不在D*states*中)
            将*U*加入到D*states*中，且不加标记;
        Dtran[*T*, *a*] = *U*;
    }
}

例，一个接受语言（a|b）*abb的NFA 如下图所示



请给出接收相同语言的DFA

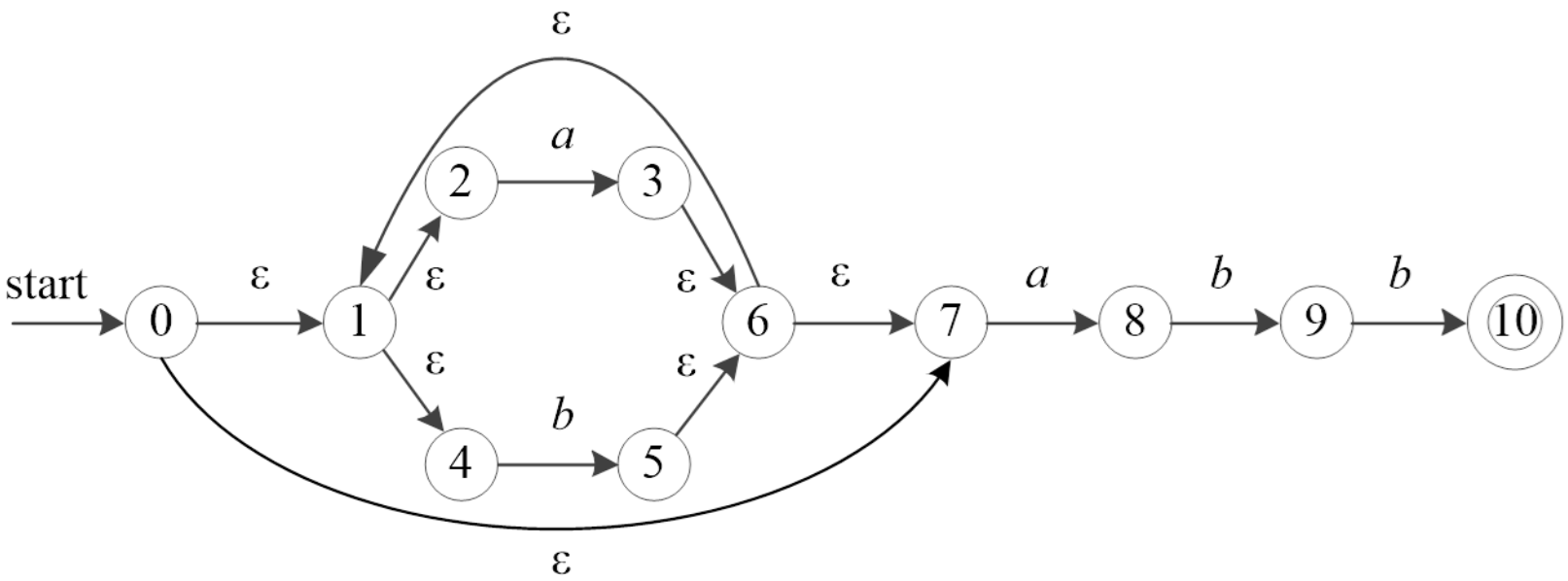
开始状态是?

A=ε-closure(0)

0 1 7 2 4

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

状态	输入符号	
	a	b
A		



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

Dtran[A,a]= B

状态	输入符号	
	a	b
A	B	

Dtran[0,a]= 空

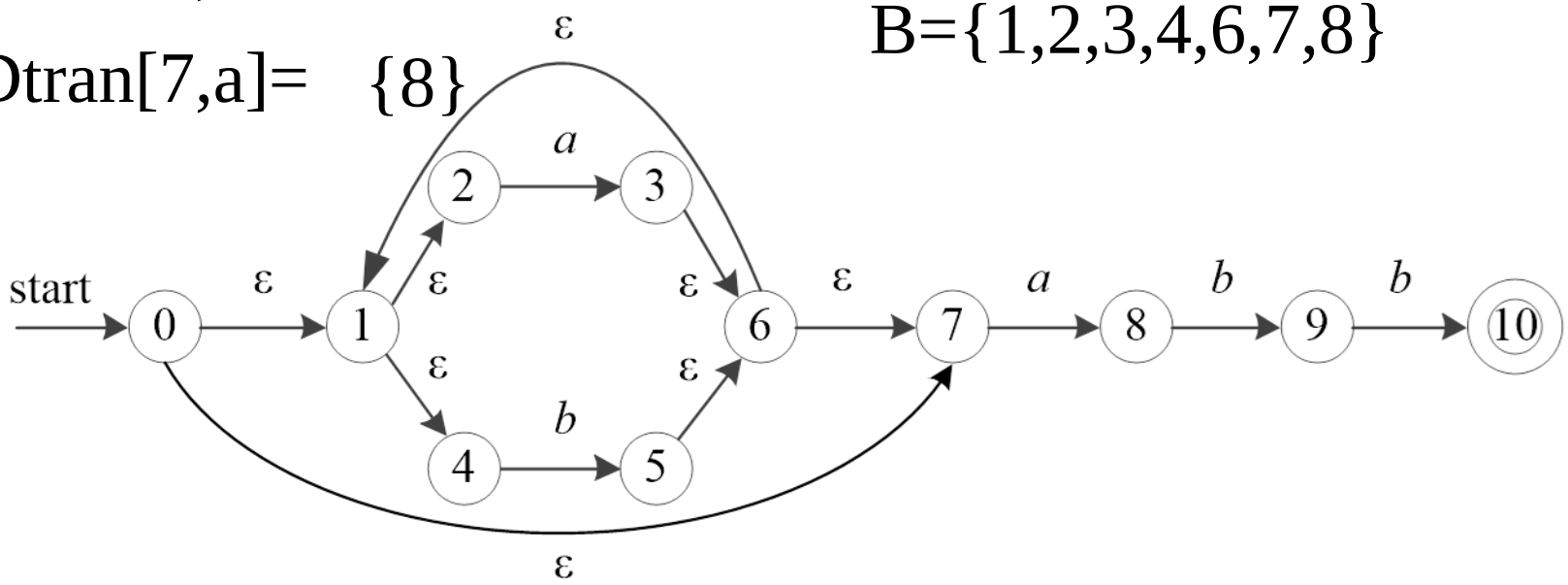
Dtran[1,a]= 空

Dtran[2,a]= {3,6,1,2,4,7}

Dtran[4,a]= 空

Dtran[7,a]= {8}

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



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

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

Dtran[0,b]= 空

Dtran[1,b]= 空

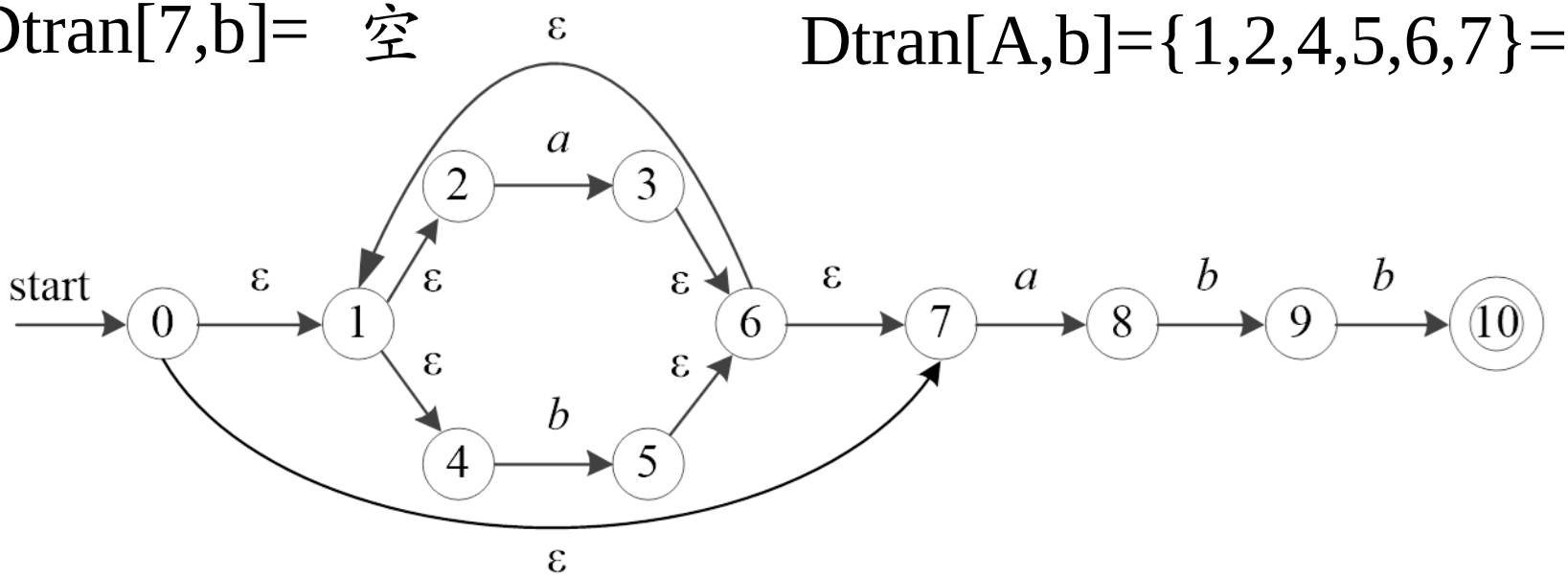
Dtran[2,b]= 空

Dtran[4,b]= {5,6,7,1,2,4}

Dtran[7,b]= 空

状态	输入符号	
	a	b
A	B	C

Dtran[A,b]={1,2,4,5,6,7}=C



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

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

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

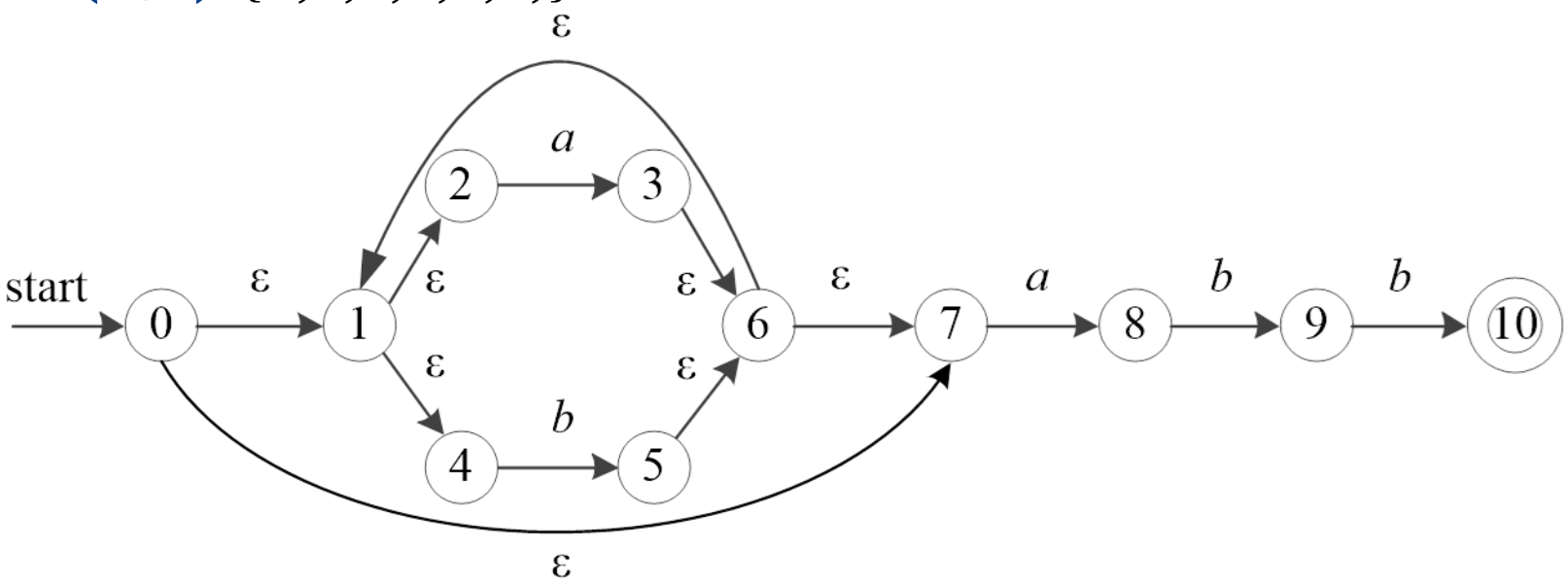
Dtran(B,a)={3,6,1,2,4,7,8} =B

Dtran(B,b)={5,6,1,2,4,7,9} =D

Dtran(C,a)={3,6,1,2,4,7,8} =B

Dtran(C,b)={5,6,1,2,4,7,} =C

状态	输入符号	
	a	b
A	B	C
B		
C		



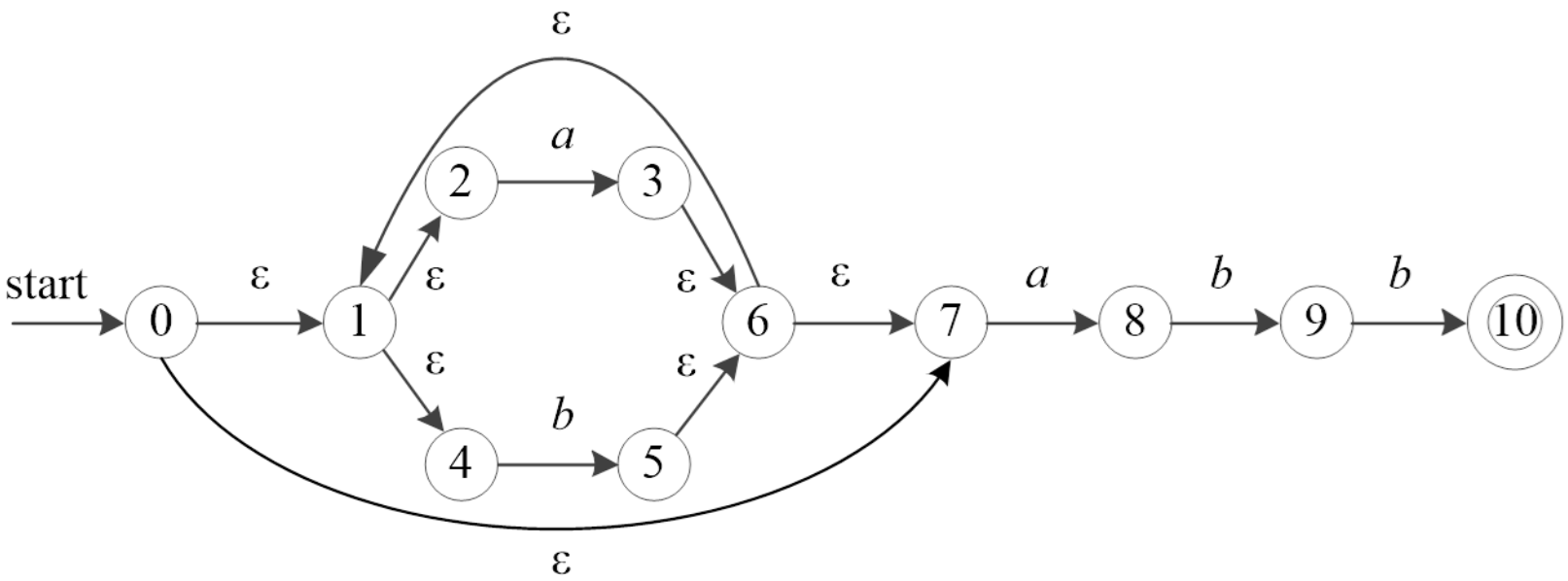
A={0,1,2,4,7}

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

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

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

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C



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

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

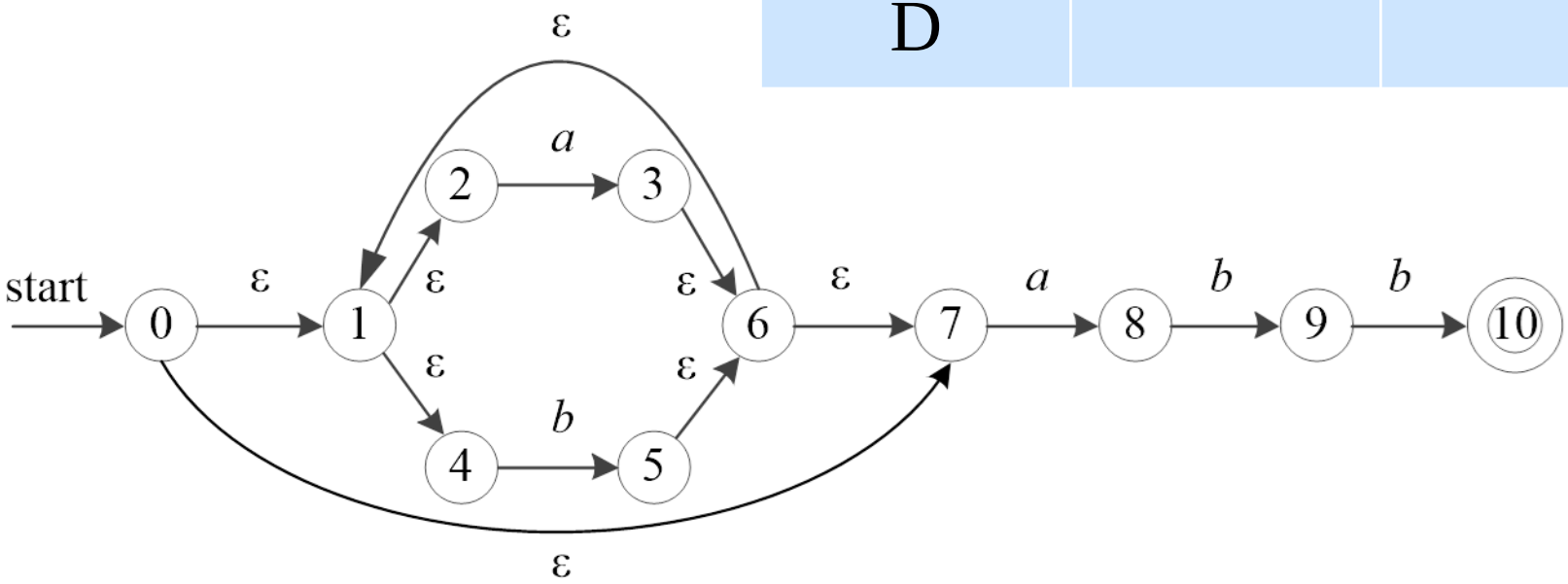
C={1,2,4,5,6,7}

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

Dtran(D,a)={3,6,1,2,4,7,8}=B

Dtran(D,b)={5,6,1,2,4,7,10}=E

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D		



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

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

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

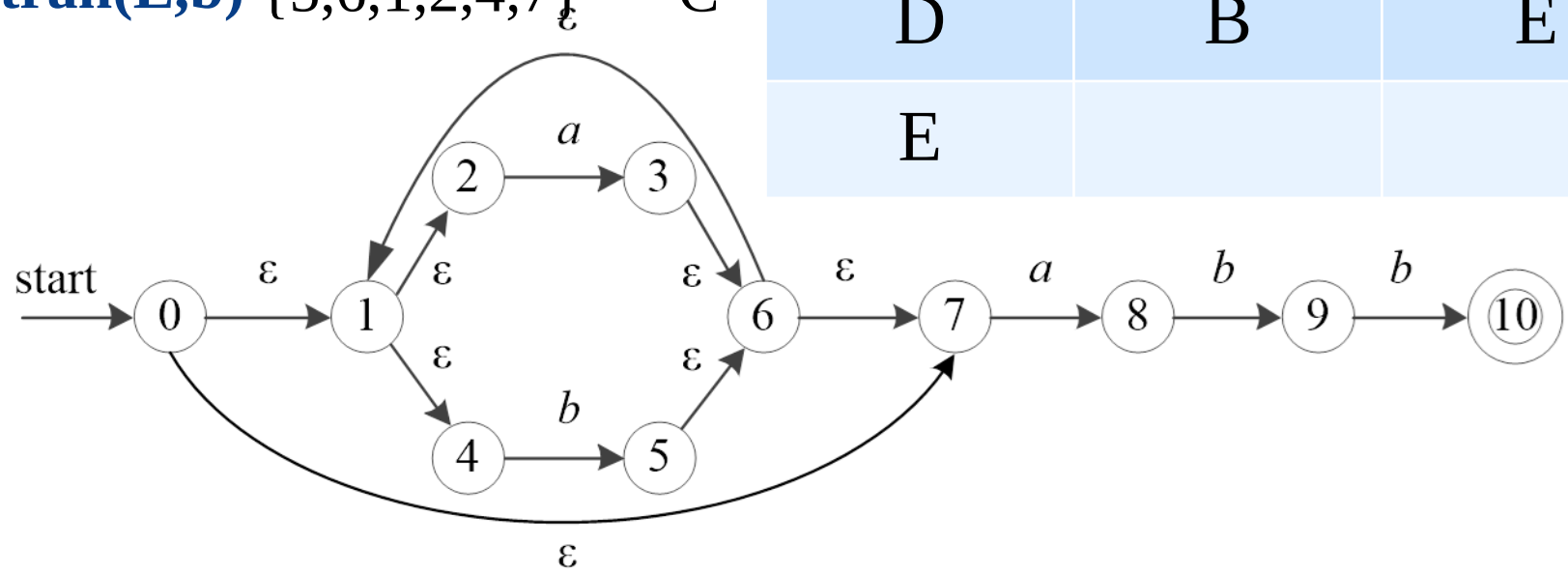
D={1,2,4,5,6,7,9}

E={1,2,,4,5,6,10}

Dtran(E,a)={3,6,1,2,4,7,8} =B

Dtran(E,b)={5,6,1,2,4,7} =C

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E		



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

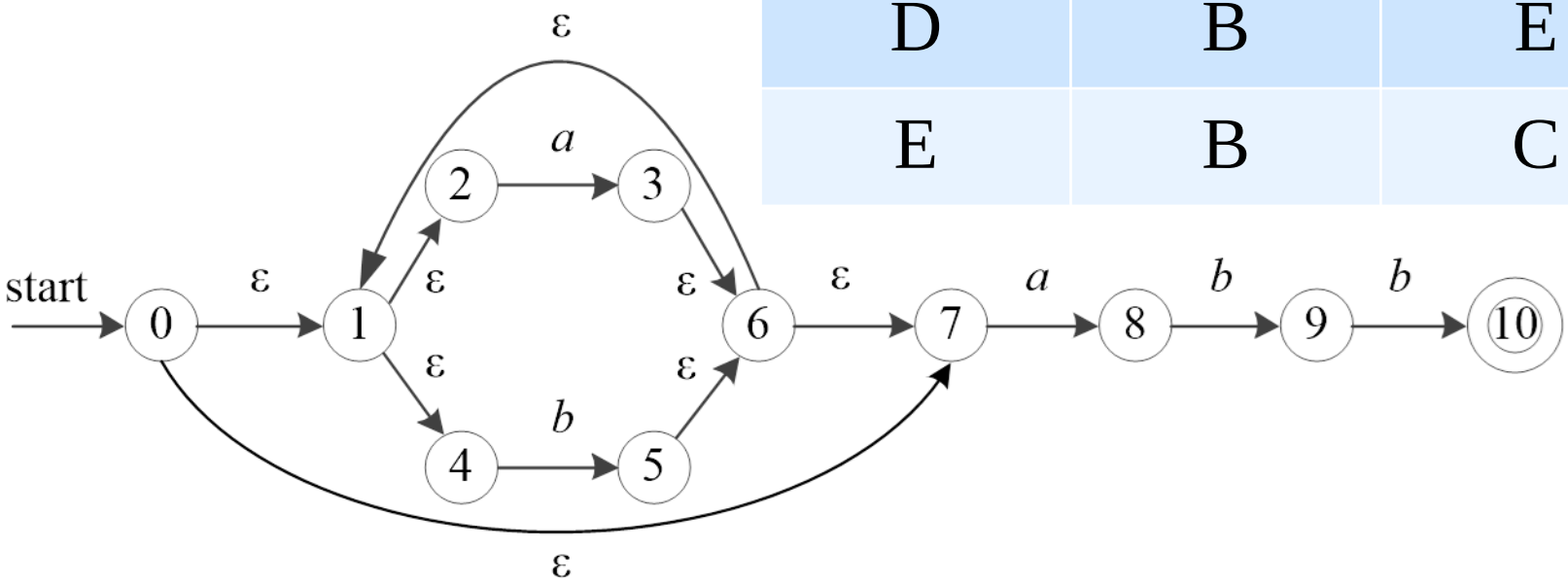
B={1,2,3,4,6,7,8}

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

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

E={1,2,4,5,6,7,10}

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Dtran(A,a)=B

Dtran(A,b)=C

Dtran(B,a)=B

Dtran(B,b)=D

Dtran(C,a)=B

Dtran(C,b)=C

Dtran(D,a)=B

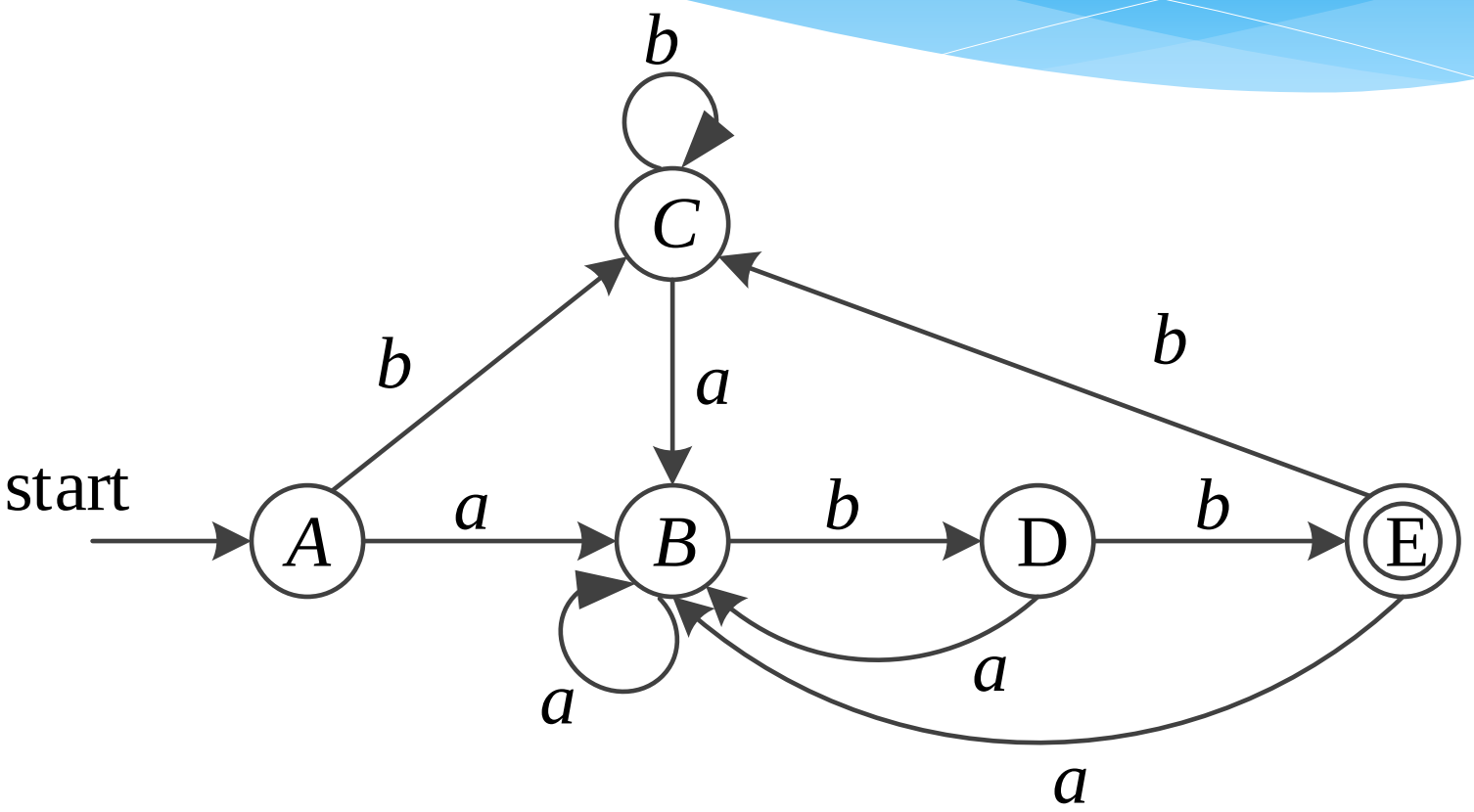
Dtran(D,b)=E

Dtran(E,a)=B

Dtran(E,b)=C

状态	输入状态	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

状态*A*是*DFA*的开始状态；包含原终止状态10的*E*状态是*DFA*的接受状态(终态)。


b
C
b
a
b
start
A
a
B
b
D
b
E
a
a
a

3.5 有限状态自动机

* 多余状态

  多余状态是指那些从开始状态出发，任何输入串都无法到达的状态

* 等价状态指满足以下两个条件的两个或者多个状态

  1）一致性条件：这些状态或同为可接受状态（终态），或同为不可接受状态（非终态）

  2）蔓延性条件：这些状态对所有输入状态，都必须转到等价的状态。

对一个DFA M最少化的基本思想:

M的状态集划分为一些不相交的子集，使得:

- ✓ 任何两个不同子集的状态是可区别的;
- ✓ 同一子集的任何两个状态是等价的。
- ✓ 最后，让每个子集选出一个代表，同时消去其他状态。

3.5 有限状态自动机

首先，把S划分为终态和非终态两个子集，形成基本划分Π。

假定某个时候，Π已含m个子集，记为：$\Pi=\{I^{(1)}, I^{(2)}, \ldots, I^{(m)}\}$，检查Π中的每个子集能否进一步划分：

对某个$I^{(i)}=\{s_1,s_2, \ldots,s_k\}$，选择某个输入字符*a*；

检查$I^{(i)}$在*a*上的转换，如果这些转换到达的状态落入当前划分的两个或多个组中，就将分割成为多个分组，使得：

s_i和s_j在同一组中当且仅当它们在*a*上的转换都到达同一组状态。

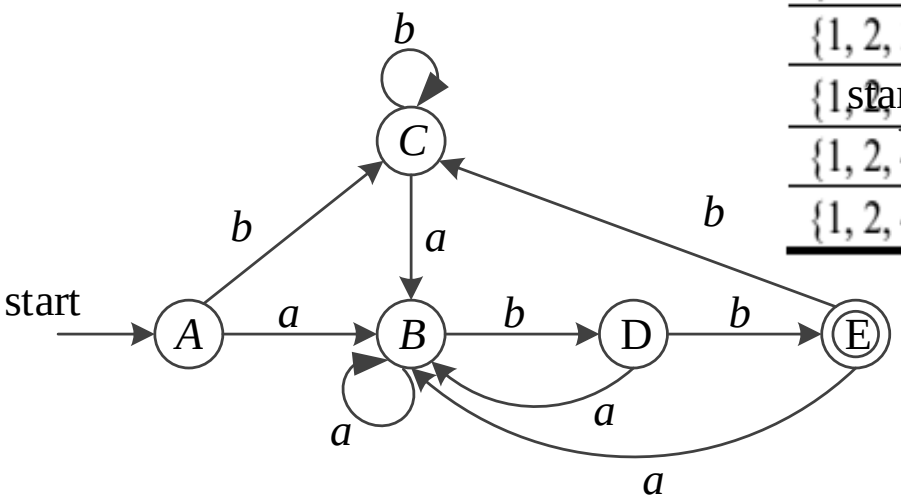
3.5 有限状态自动机

重复上述过程，直到Π所含子集数不再增长

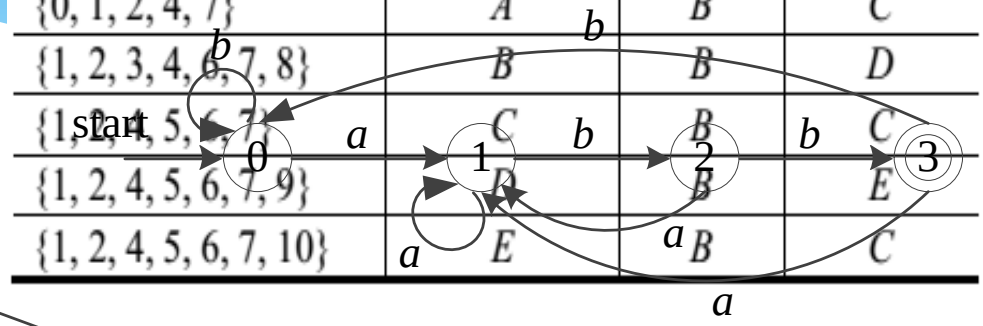
对于上述最后划分Π中的每个子集，选取每个子集中的一个状态代表其他状态，则可得到化简后的DFA M’。

若I含有原来的初态，则其代表为新的初态，若I含有原来的终态，则其代表为新的终态

(a | b) * abb



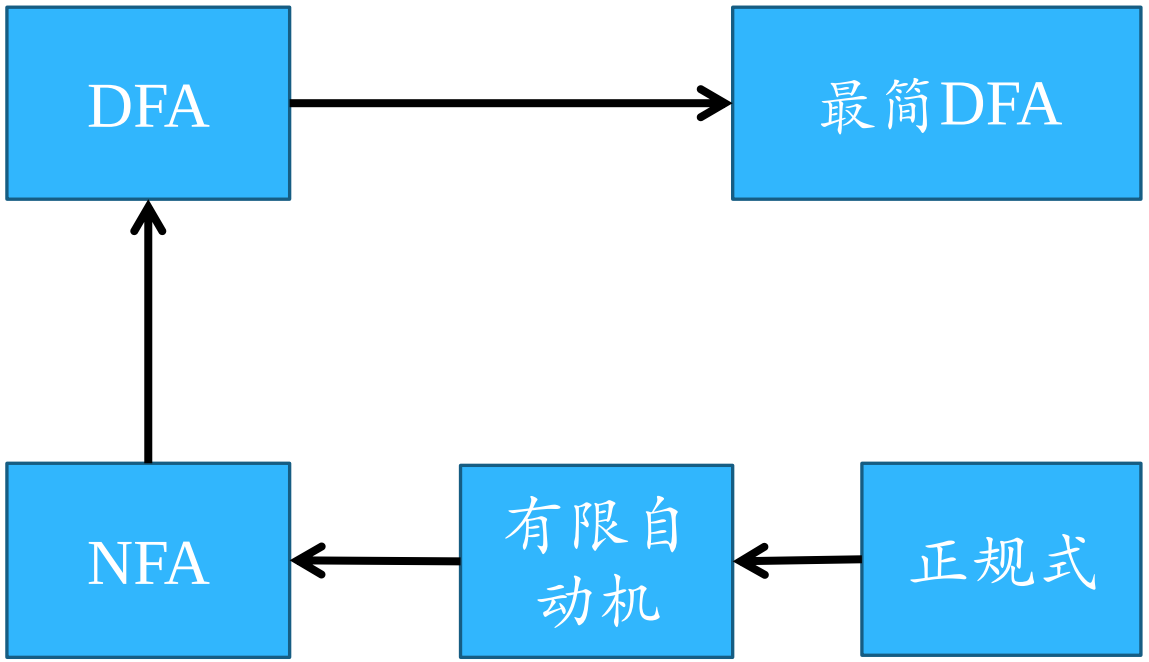
NFA 状态	DFA 状态	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E	B	C



- {A, B, C, D} {E}
- {A, B, C} {D} {E}
- {A, C} {B} {D} {E}

状态	a	b
A	B	A
B	B	D
D	B	E
E	B	A

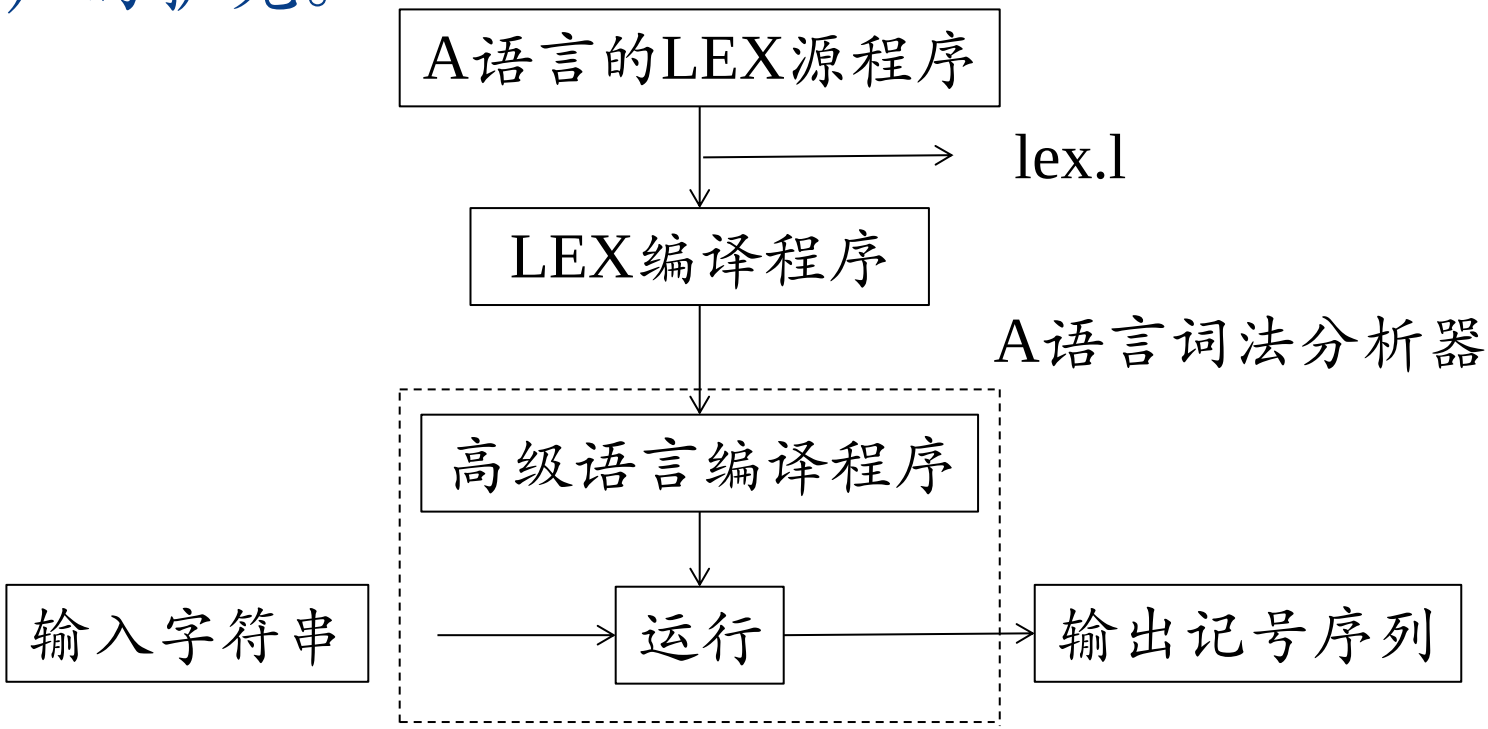
3.5 有限状态自动机



3.6 词法分析程序生成器LEX

Lex是一个词法分析器的自动产生系统。LEX并不是一种完整的语言，而是某种高级语言（宿主语言，如C语言）的扩充。

词法分析程序生成流程



3.6 词法分析程序生成器LEX

LEX源程序的一般格式:

{辅助定义部分}
%%
{翻译规则部分}
%%
{用户子程序部分}

```
/*辅助定义部分*/
%{
/*显明常量
LT,LE,EQ,NE,GT,GE,IF,THEN,ELSE,ID,NUMBER,RELOP的定义*/
%}
# include<studio.h>
char    *       yylval;
delim           [ \t\n]
ws              {delim}+
digit           [0-9]
letter          [A-Za-z]
id              {letter}({letter}|{digit})*
extern          yylval, yytext, yyleng;
%}
```

```
%%              /*翻译规则部分*/
{ws}            {}/*没有动作，也不返回，作用是跳过所有空字符*/
if              {return(IF);}
{id}            {yylval=install_id();return(ID);}
{number}        {yylval=install_number();return(NUMBER);}
“<”             {yylval=LT;return(RELOP);
 “>”            {yylval=GT;return(RELOP);}
…
%%              /*用户子程序部分*/
install_id()
{/*把单词装入符号表并返回指针。
yytext指向该单词的第一个字符，yyleng给出它的长度*/
}
Install_number()
{/*类似函数install_id()的动作*/}
```

3.6 词法分析程序生成器LEX

根据LEX源程序的规则P_i构造相应的NFA M_i;

将所有的NFA M_i合并成总的NFA M;

根据子集构造法转换成DFA;

化简DFA;

给出控制执行程序。

练习

构造一个最简DFA，使其能够识别所有由偶数个0和偶数个1所组成串。

提示：

正则表达式

NFA

DFA

最简化DFA