

---

# Lecture 3: Lexical Analysis ( Part II )

---

Xiaoyuan Xie 谢晓园

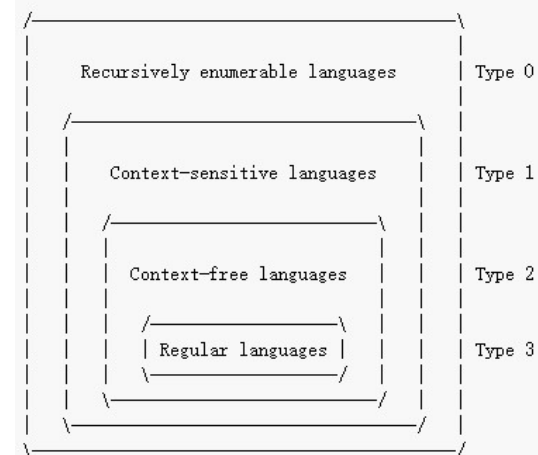
[xxie@whu.edu.cn](mailto:xxie@whu.edu.cn)

计算机学院E301

## 回顾

- 编译器：把高级语言翻译成目标(机器)语言（几步）
- 如何定义语言？
  - 语言定义在字母表上 $L(\Sigma)$ ，字母表 $\Sigma$ 定义了语言中允许出现的全部符号(有穷集合)。
  - $L(\Sigma)$ 规定了词法(三型文法)、语法(二型文法)、语义

Chomsky hierarchy:



## 回顾

### ■ 如何定义文法

词法、语法都是这样定义的

■ 文法(G, Grammar): 四元组  $G = (V_N, V_T, S, P)$  , 其中

- $V_N$  : 一个非空有限的非终结符号集合, 它的每个元素称为非终结符, 一般用大写字母表示, 它是可以被取代的符号;
- $V_T$  : 一个非空有限的终结符号集合, 它的每个元素称为终结符, 一般用小写字母表示, 是一个语言不可再分的基本符号 --- 来自于  $\Sigma$
- $S$  : 一个特殊的非终结符号, 称为文法的开始符号或识别符号,  $S \in V_N$ 。开始符号  $S$  必须至少在某个产生式的左部出现一次;
- $P$  : 产生式的有限集合。所谓的产生式, 也称为产生规则或简称为规则, 是按照一定格式书写的定义语法范畴的文法规则。
- 设  $V$  是文法  $G$  的符号集, 则有:  $V = V_N \cup V_T$  ,  $V_N \cap V_T = \emptyset$

$G_1 = (\{S\}, \{a, b\}, S, P)$ ,  
(0)  $S \rightarrow aS$  (左线性, 递归)  
(1)  $S \rightarrow a|b$  (一定要有递归出口)

## 回顾

### ■ 词法分析

词素

e.g. valu#e

Comply with

RE

e.g. id的RE为`letter(letter|digit)*`

valu#e = initial + rate \* 60

RE等价于正规文法（三型），用 $L(RE)$ 表示

正规文法的能力足以描述词法规定

当前语言对词法的规定，回忆一下，C语言对变量名的规定，对函数名的规定



## 回顾

### ■ 熟悉字符串的操作符

- 括号( $r$ )：不改变 $r$ 表示，主要是用于确定运算优先关系
- 或运算 $|$ ：表示“或”关系
- 连接运算 $\cdot$ ：表示连接，经常省略，如 $r \cdot s$ 也可表示为 $rs$
- $*$ 运算： $r^*$ 表示对 $r$ 所描述的文本进行0到若干次循环连接
- $[]$ ——任选符号： $[ab]$ 等价于 $(a|b)$

## 回顾

### ■ 定义RE( $\Sigma$ 为字母表)

- 原子正则表达式(atomic regular expressions)
  - $\epsilon$ 和 $\emptyset$  是 $\Sigma$ 上的正则表达式，它们所表示的正则集分别为 $L(\epsilon)=\{\epsilon\}$ ， $L(\emptyset)=\{\}$ .
  - 对任何 $a \in \Sigma$ ,  $a$  是 $\Sigma$ 上的正则表达式，它所表示的正则集 $L(a)=\{a\}$ ;
- 归纳步骤：若 $r$ 和 $s$ 都是 $\Sigma$ 上的正则表达式，它们所表示的正则集分别为 $L(r)$ 和 $L(s)$ , 则
  - $(r)$ 也是 $\Sigma$ 上的正则表达式 --- ( ) 在这里是操作符!!! 识别RE- ✓
  - $r|s$ 也是 $\Sigma$ 上的正则表达式
  - $r \cdot s$ 也是 $\Sigma$ 上的正则表达式
  - $r^*$ 也是 $\Sigma$ 上的正则表达式
  - 有限次使用上述3条规则构成的表达式称为 $\Sigma$ 上的正则表达式 --- 上述操作可以满足三型文法  
(证明：[http://tutorialspoint.howtolib.com/automata\\_theory/regular\\_sets.htm](http://tutorialspoint.howtolib.com/automata_theory/regular_sets.htm))

怎么自己写一个RE?

## 回顾

- 练习：设字母表 $\Sigma = \{0, 1\}$ , 试写正则表达式
  - 所有 $\Sigma$ 上定义的串
    - $[01]^*$ , 或  $(0|1)^*$
  - 表示二进制数
    - 特点：以0开头后面不接任何数，以1开头后面可接任何数
    - $0|1[10]^*$ , 或  $0|1(0|1)^*$
  - 能被2整除的二进制数
    - 特点：以0结尾
    - $0|1(0|1)^*0$

## 回顾

### ■ 练习：为自然语言构造RE

- All strings of lowercase letters that contain the five vowels in order.
  - $S \rightarrow \text{other}^* a (\text{other}|a)^* e (\text{other}|e)^* i (\text{other}|i)^* o (\text{other}|o)^* u (\text{other}|u)^* \text{other} \rightarrow [\text{bcdfghjklmnpqrstvwxyz}]$

自己写一个RE ✓

给定RE，怎么描述出它定义的字符串？



## 回顾

### ■ 练习：下列正则表达式定义了什么语言

- $a(a|b)^*a$

- 由a, b组成的，并由a开头和结尾的字符串

- $((\varepsilon|a)b^*)^*$

- $(\varepsilon b^*|ab^*)^* \rightarrow (b^*|ab^*)^*$  : 空串或所有由a, b组成的字符串

- $((\underline{(\varepsilon|a)b})^*)^*$

- $((\underline{(\varepsilon|a)b})^* \rightarrow (\varepsilon b|ab)^* \rightarrow (b|ab)^*$  : b/ab b/ab b/ab ..... b/ab

- 空串 或 任意个b组成的字符串，两个b之间隔着0-1个a

## 回顾

### ■ 练习：下列正则表达式定义了什么语言

- $b^*(ab^*ab^*)^*$ 
  - 空串 或 由偶数个a和任意个b组成的字符串
- $c^*a(a|c)^*b(a|b|c)^* | c^*b(b|c)^* a(a|b|c)^*$ 
  - 由a,b,c组成，至少包含一个a和一个b的串

给定RE，描述出它定义的字符串 ✓

## 回顾

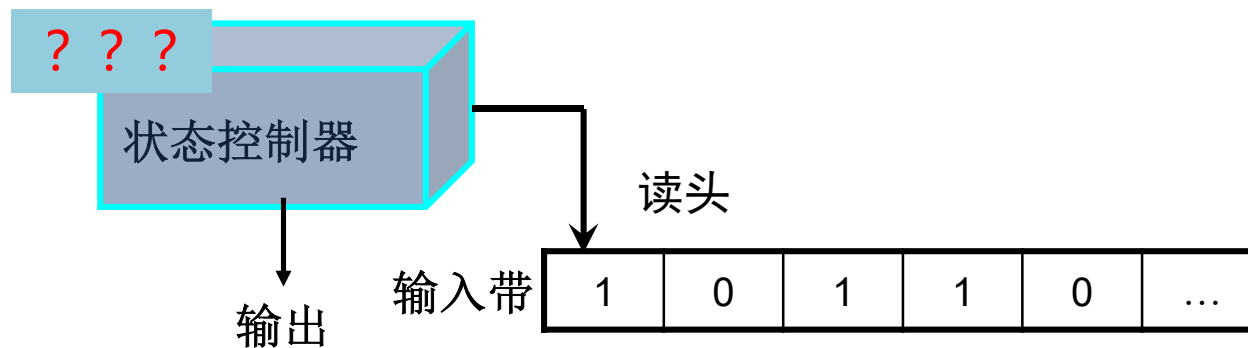
### ■ 产生式

- 词法分析只考虑三型文法，而三型文法一般用RE表示就足够了，一般不需要用与RE等价的产生式来表示
- 暂时不做回顾，到上下文无关文法时再回顾

## 回顾

### ■ 词法分析

- 如何实现匹配过程？ --- FM



## 回顾

### ■ FA : 转换图 v.s. 转换矩阵

DFA  $M = (\{S_0, S_1, S_2, S_3\}, \{a, b\}, f, S_0, \{S_3\})$ , 其中  $f$  定义为 :

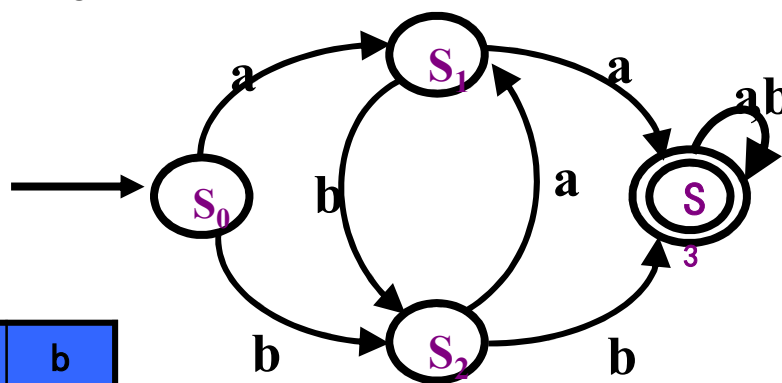
$f(S_0, a) = S_1$  ,  $f(S_2, a) = S_1$

$f(S_0, b) = S_2$  ,  $f(S_2, b) = S_3$

$f(S_1, a) = S_3$  ,  $f(S_3, a) = S_3$

$f(S_1, b) = S_2$  ,  $f(S_3, b) = S_3$

	a	b
0 <sup>+</sup>	1	2
1	3	2
2	1	3
3 <sup>-</sup>	3	3

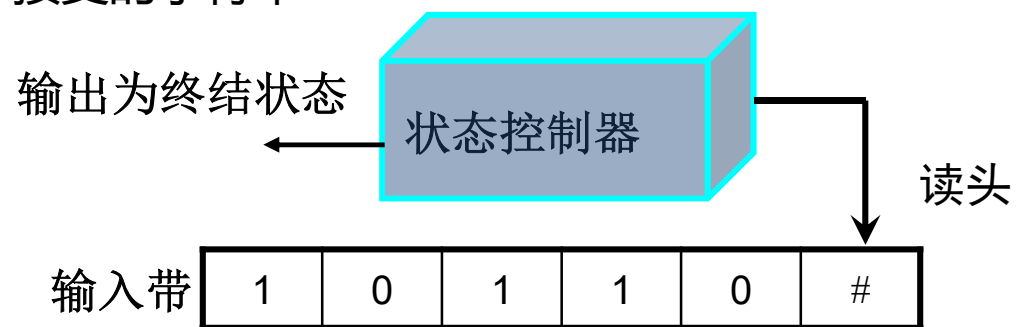




## 回顾

### ■ 词法分析

- FA接受的字符串：

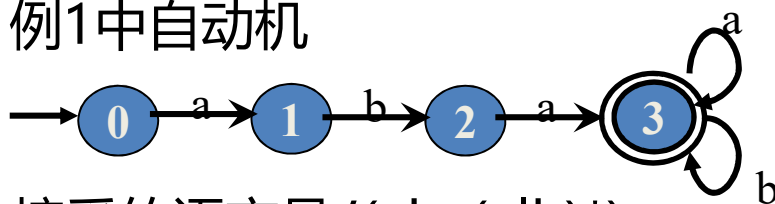


FA 和 RE 的关系??

## 回顾

### ■ $L(RE) \leftrightarrow FA$

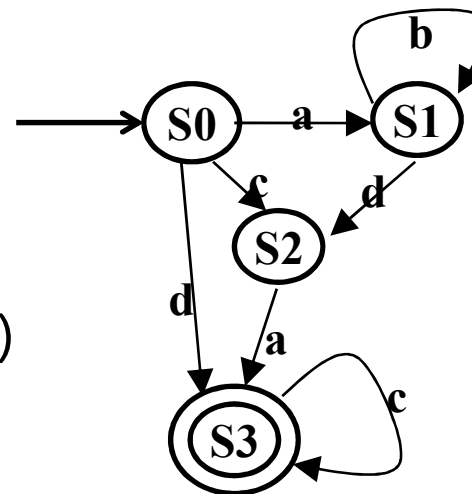
#### ■ 例1中自动机



接受的语言是  $L(aba(a|b)^*)$

#### ■ 例2中自动机

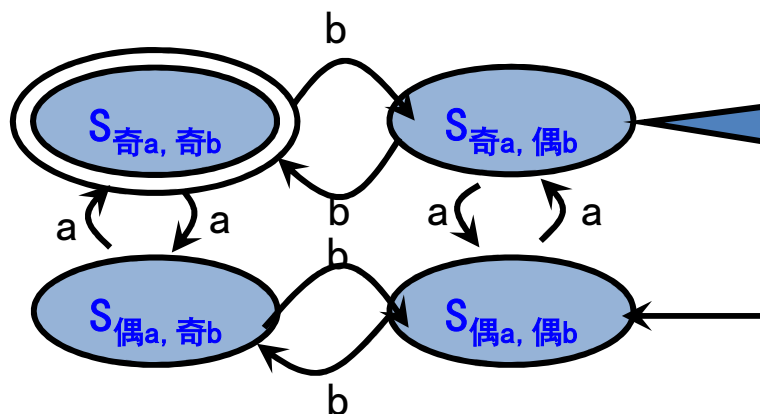
接受的语言是  $L((ab^*da \mid ca \mid d)c^*)$



## 回顾

### ■ $L(RE) \leftrightarrow FA$

- 自动机的设计是一个创造过程，没有固定的算法和过程（语法设计也如此）
- 例1： $\Sigma = \{a, b\}$ , 构造自动机识别由所有奇数个a和奇数个b组成的字符串。



关键：不需要记住所看到的整个字符串，只需记住至此所看到的a和b的个数是奇数还是偶数

## 回顾

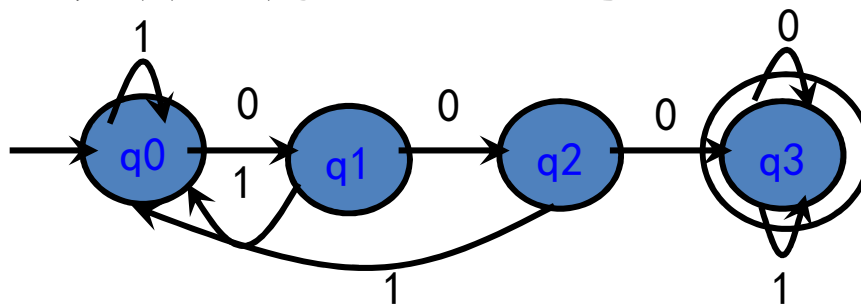
### ■ $L(RE) \leftrightarrow FA$

- 例2：设计有限自动机M，识别 $\{0,1\}$ 上的语言

$$L = \{x000y \mid x,y \in \{0,1\}^*\}$$

分析：该语言的特点是每个串都包含连续3个0的子串。

自动机的任务就是识别/检查 000 的子串。

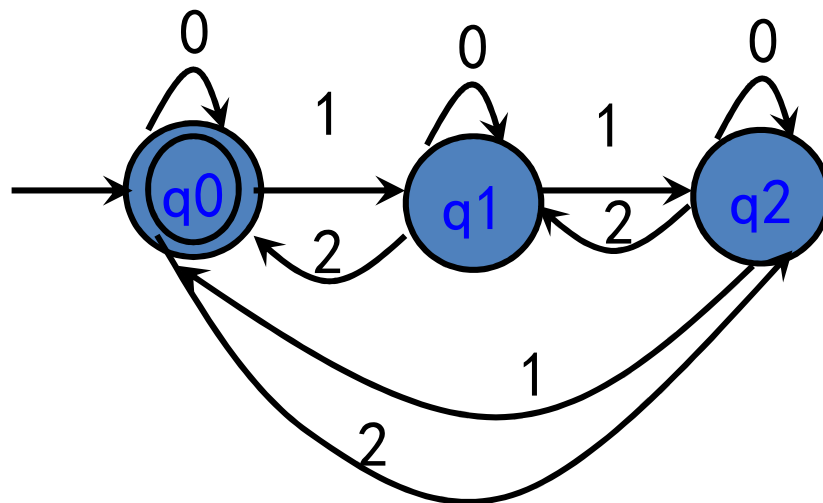


## 回顾

### ■ $L(RE) \leftrightarrow FA$

- 例3：设计有限自动机M，识别 $\{0,1,2\}$ 上的语言，每个字符串代表的数字能整除3。

分析: (1) 一个十进制数除以3，余数只能是0,1,2；(2) 被3整除的十进制数的特点：  
十进制数的所有位数字的和能整除3。





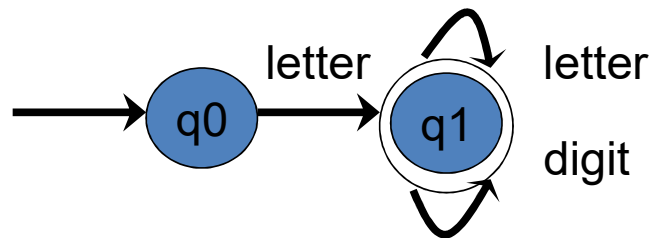
## 回顾

### ■ $L(RE) \leftrightarrow FA$

■ 例5：使用DFA定义程序设计语言的标识符

x, Xy, x123, xYz 接受

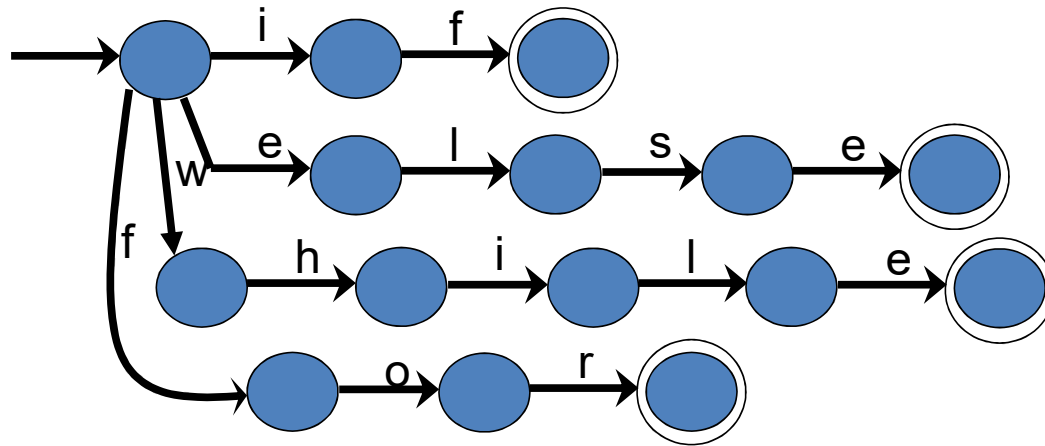
23x, 12\_x, \_x 拒绝



## 回顾

### ■ $L(RE) \leftrightarrow FA$

- 例6：使用DFA定义程序设计语言的保留字  
{if, else, while, for}



## 回顾

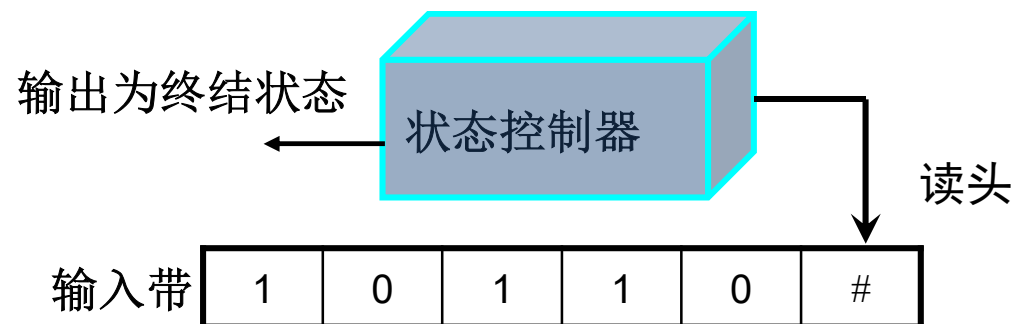
### ■ FA: DFA v.s. NFA

	DFA	NFA
初始状态	一个初始状态	初始状态集合
$\epsilon$ 边	不允许	允许
$\delta(S, a)$	$S'$ or $\perp$	$\{S_1, \dots, S_n\}$ or $\perp$
实现	容易	有不不确定性

## 回顾

### ■ FA: DFA v.s. NFA

- DFA/NFA接受的字符串(可以等价)：



**$\epsilon$ 意味着读头不动，但是状态依旧发生转换**

## 回顾

### ■ NFA $\rightarrow$ DFA (子集法)

- 输入：一个NFA  $N = \{\Sigma, SS, SS^0, \delta, TS\}$
- 输出：一个接受同样语言的DFA  $D = \{\Sigma, SS', S^0, \delta', TS'\}$
- 方法：为D构造一个转换表Dtran，D的每个状态是一个NFA状态集合，构造Dtran使得D可以模拟N在遇到一个给定输入串时可能执行的所有动作

消除不确定性：合并所有 $\epsilon$ 转换的状态；合并所有相同字符转换的状态

Tip: 把N中的状态集合 看做 D中的一个状态



## 回顾

### ■ NFA $\rightarrow$ DFA (子集法)

#### ■ 一些基本操作

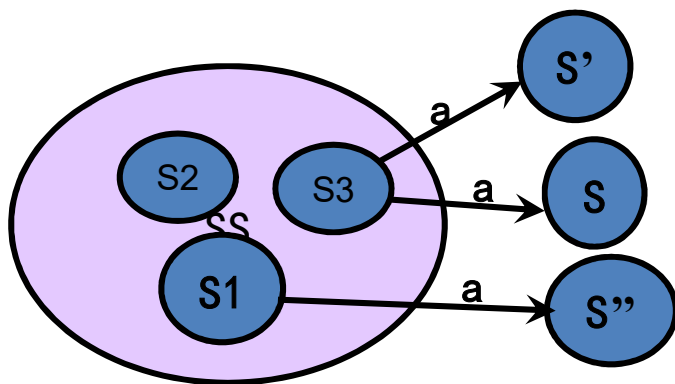
OPERATION	DESCRIPTION
$\epsilon\text{-closure}(s)$	Set of NFA states reachable from NFA state $s$ on $\epsilon$ -transitions alone.
$\epsilon\text{-closure}(T)$	Set of NFA states reachable from some NFA state $s$ in set $T$ on $\epsilon$ -transitions alone; $= \bigcup_{s \in T} \epsilon\text{-closure}(s)$ .
$\text{move}(T, a)$	Set of NFA states to which there is a transition on input symbol $a$ from some state $s$ in $T$ .

#### ■ 核心思想：找出当N读入某个输入串之后可能位于的所有状态集合

## 回顾

### ■ NFA $\rightarrow$ DFA (子集法)

- 对于NFA  $M$ 中的给定状态集合 $T$ 和符号  $a$ ,  $\text{Move}(T, a) = \{s \mid \text{对于状态集} T \text{中的一个状态} s_i, \text{如果} A \text{中存在一条从} s_i \text{到} s \text{的} a \text{转换边}\}$



$$\text{Move}(\{S1, S2, S3\}, a) = \{S, S', S''\}$$

# 回顾

## ■ 由NFA构造DFA (子集法)

### ■ 构造Dtran

我们需要找出当N读入了某个输入串之后可能位于的所有状态集合。

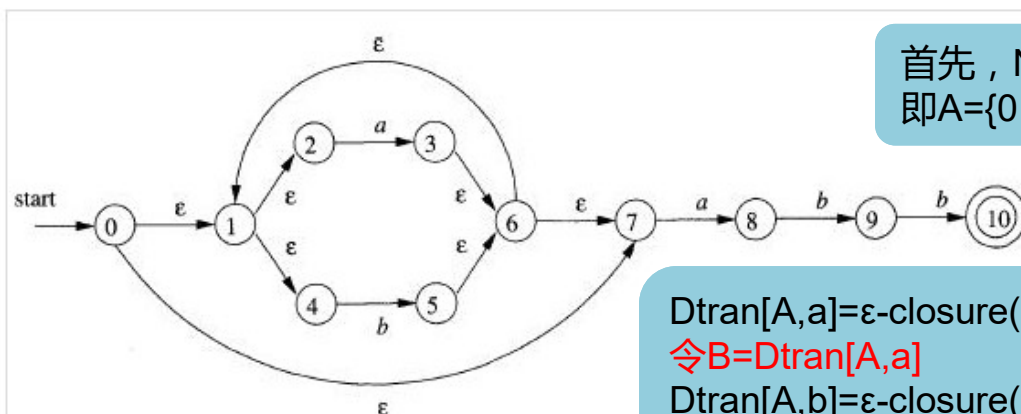
```
initially,  $\epsilon$ -closure( $s_0$ ) is the only state in  $Dstates$ , and it is unmarked;  
while ( there is an unmarked state  $T$  in  $Dstates$  ) {  
    mark  $T$ ;  
    for ( each input symbol  $a$  ) {  
         $U = \epsilon$ -closure( $move(T, a)$ );  
        if (  $U$  is not in  $Dstates$  )  
            add  $U$  as an unmarked state to  $Dstates$ ;  
         $Dtran[T, a] = U$ ;  
    }  
}
```

首先，在读入第一个输入符号之前，N可以位于集合 $\epsilon$ -closure( $s_0$ )中的任何状态上，其中 $s_0$ 是N的开始状态。下面进行归纳定义，

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 $a$ ，那么N可以立即移动到 $move(T, a)$ 中的任何状态。然而，N可以在读入 $a$ 后再执行几个 $\epsilon$ 转换，因此N在读入 $a$ 之后可位于 $\epsilon$ -closure( $move(T, a)$ )中的任何状态上。

## 回顾

### ■ 由NFA构造DFA (子集法)例1 : $r=(a|b)^*abb$ 的NFA to DFA



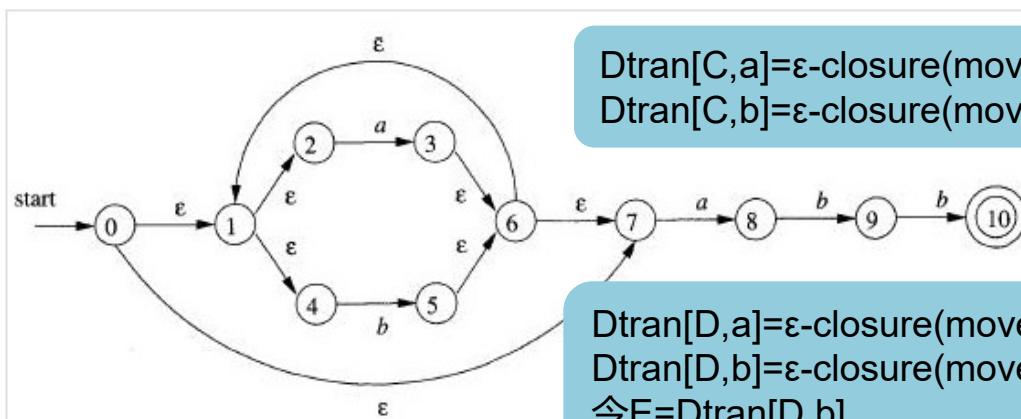
首先，NFA的开始状态A是 $\epsilon$ -closure(0)，  
即 $A=\{0, 1, 2, 4, 7\}$ ，NFA的输入字母表是 $\{a,b\}$

$Dtran[A,a]=\epsilon\text{-closure}(\text{move}(A,a))=\epsilon\text{-closure}(\{3,8\})=\{1,2,3,4,6,7,8\}$ ,  
 $\text{令 } B=Dtran[A,a]$   
 $Dtran[A,b]=\epsilon\text{-closure}(\text{move}(A,b))=\epsilon\text{-closure}(\{5\})=\{1,2,4,6,7\}$ ,  
 $\text{令 } C=Dtran[A,b]$

$Dtran[B,a]=\epsilon\text{-closure}(\text{move}(B,a))=\epsilon\text{-closure}(\{3,8\})=\{1,2,3,4,6,7,8\}=B$   
 $Dtran[B,b]=\epsilon\text{-closure}(\text{move}(B,b))=\epsilon\text{-closure}(\{5,9\})=\{1,2,4,5,6,7,9\}$ ,  
 $\text{令 } D=Dtran[B,b]$

## 回顾

### ■ 由NFA构造DFA (子集法) 例1 : $r=(a|b)^*abb$ 的NFA to DFA



$Dtran[C,a]=\epsilon\text{-closure}(\text{move}(C,a))=\epsilon\text{-closure}(\{3,8\})=\{1,2,3,4,6,7,8\}=B$   
 $Dtran[C,b]=\epsilon\text{-closure}(\text{move}(C,b))=\epsilon\text{-closure}(\{5\})=\{1,2,4,6,7\}=C$

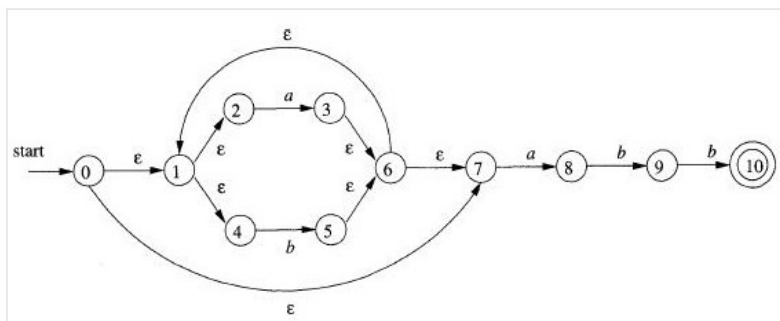
$Dtran[D,a]=\epsilon\text{-closure}(\text{move}(D,a))=\epsilon\text{-closure}(\{3,8\})=\{1,2,3,4,6,7,8\}=B$   
 $Dtran[D,b]=\epsilon\text{-closure}(\text{move}(D,b))=\epsilon\text{-closure}(\{5,10\})=\{1,2,4,5,6,7,10\}$ ,  
 令  $E=Dtran[D,b]$

$Dtran[E,a]=\epsilon\text{-closure}(\text{move}(E,a))=\epsilon\text{-closure}(\{3,8\})=\{1,2,3,4,6,7,8\}=B$   
 $Dtran[E,b]=\epsilon\text{-closure}(\text{move}(E,b))=\epsilon\text{-closure}(\{5\})=\{1,2,4,6,7\}=C$

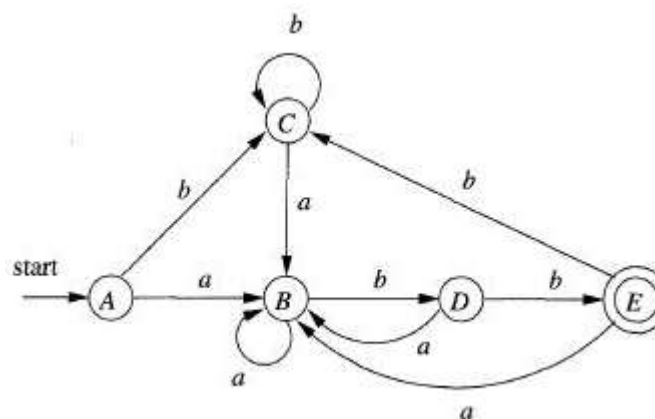


# 回顾

## ■ 由NFA构造DFA (子集法) 例1 : $r=(a|b)^*abb$ 的NFA to DFA



NFA STATE	DFA STATE	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, 3, 5, 6, 7, 10}	E	B	C



## 回顾

### ■ 作业

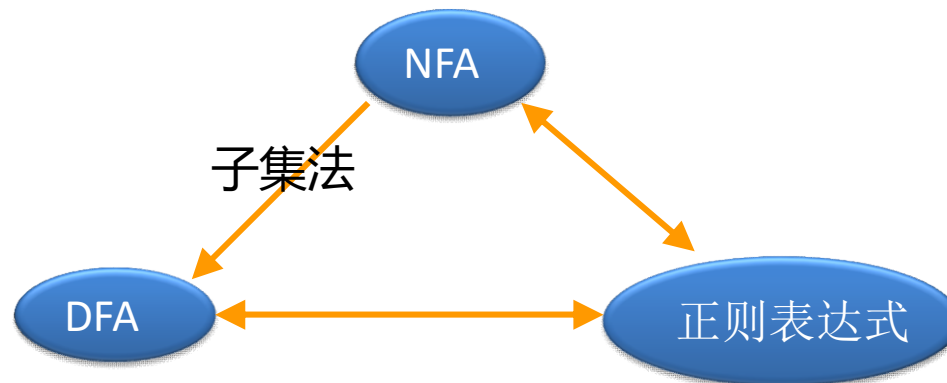
- 教材P78 : 3.3.2 , 3.3.5 ( 有一定难度 )
- 教材P86 : 3.4.1 , 3.4.2 ( 基础题 , not covered )
- 教材P96 : 3.6.3 , 3.6.4 , 3.6.5 ( 基础题 )
- 教材P105 : 3.7.1 ( 基础题 )



## 3.7 RE v.s. NFA/DFA

### 3.7 RE v.s. NFA/DFA

- 对 $\Sigma$ 上的每一个正则表达 $R$ ，存在一个 $\Sigma$ 上的非确定有限自动机 $N$ ，使得  $L(N) = L(R)$
- $N$ 可以通过子集法得到与之等价的确有限自动机 $D$



## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法一：RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  最小DFA (\*\*\*)
- 方法二：RE  $\rightarrow$  DFA  $\rightarrow$  最小DFA (\*)
- 方法三：RE  $\rightarrow$  NFA, 然后直接模拟 (模拟算法见教材P99，算法3.22)

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法一：RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  最小DFA
- 方法二：RE  $\rightarrow$  DFA  $\rightarrow$  最小DFA
- 方法三：RE  $\rightarrow$  NFA（然后直接模拟，模拟算法见教材P99，算法3.22）

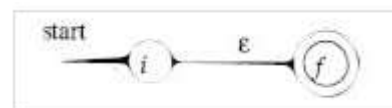
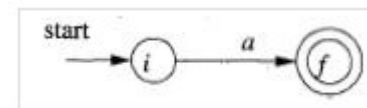
## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法一：由RE构造NFA (Thompson算法)

- 输入：字母表 $\Sigma$ 上的一个正则表达式 $r$
- 输出：一个接受 $L(r)$ 的NFA  $N$
- 基本规则：

- 对于字母表中的原子表达式 $a$ , 构造下面的NFA
- 对于表达式 $\varepsilon$ , 构造右边的NFA



又是递归：  
 $r$ 由sub- $r$ 递归构成  
 $N(r)$ 也由 $N(\text{sub-}r)$ 递归构成



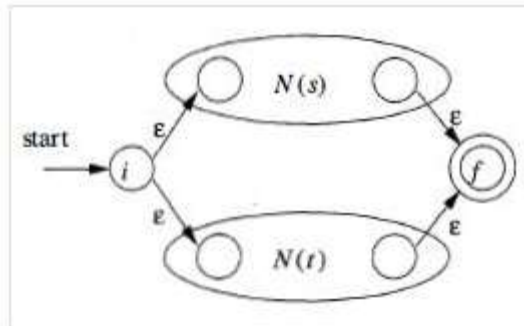
## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法一：由RE构造NFA (Thompson算法)

##### ■ 归纳规则：假设正则表达式s和t的NFA分别为N(s)和N(t)

- 对于  $r = s|t$ ，构造如下NFA，这里i和f都是新状态，分别是N(r)的开始状态和接受状态，从i到N(s)和N(t)的开始状态各有一个 $\epsilon$ 转换，从N(s)和N(t)到接受状态f也各有一个 $\epsilon$ 转换



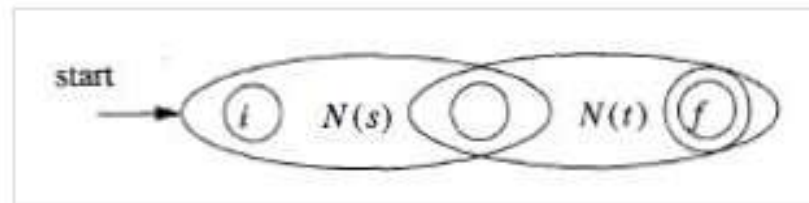
## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法一：由RE构造NFA (Thompson算法)

##### ■ 归纳规则：假设正则表达式s和t的NFA分别为N(s)和N(t)

- 对于 $r=st$ ，构造下面的NFA，N(s)的开始状态变成了N(r)的开始状态；N(t)的接受状态成为N(r)的唯一接受状态；N(s)的接受状态和N(t)的开始状态合并为一个状态，合并后的状态拥有原来进入和离开合并前的两个状态的全部转换



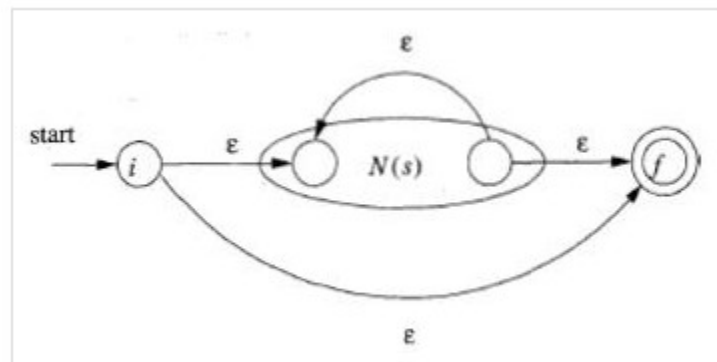
## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法一：由RE构造NFA (Thompson算法)

##### ■ 归纳规则：假设正则表达式s和t的NFA分别为N(s)和N(t)

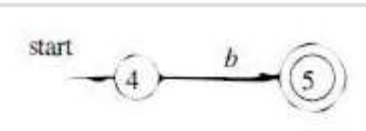
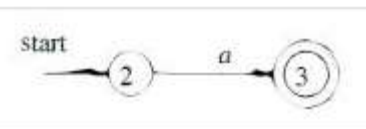
- 对于 $r=s^*$ ，构造下面的NFA，i和f是两个新状态，分别是N(r)的开始状态和唯一接受状态；要从i到达f，我们可以沿着新引入的标号为 $\epsilon$ 的路径前进，该路径对应于L(s)的一个串；我们也可以达到N(s)的开始状态，然后经过该NFA，零次或多次从它的接受状态回到它的开始状态并重复上述过程



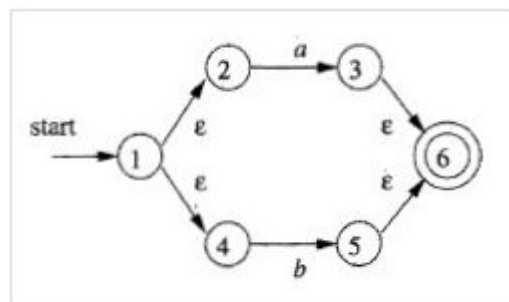
## 3.7 RE v.s. NFA/DFA

- 由RE构造NFA (Thompson算法), 为 $r=(a|b)^*abb$ 构造NFA

- 为表达式 $r_1 = a$ ,  $r_2 = b$ 构造NFA



- 为表达式 $r_3 = r_1|r_2$ 构造NFA

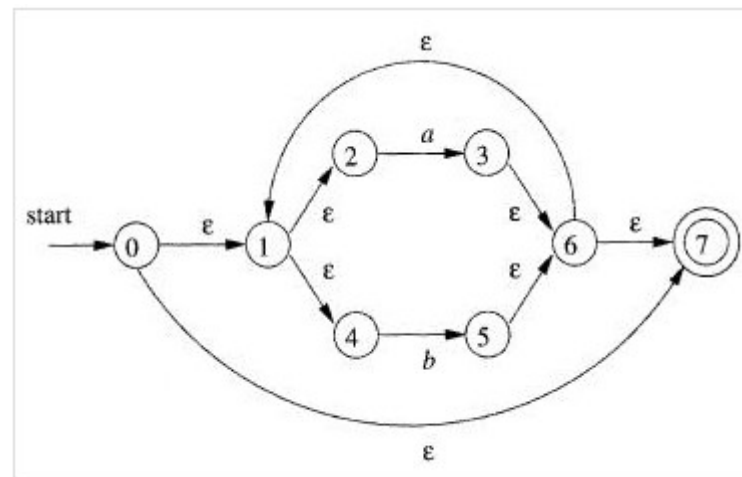
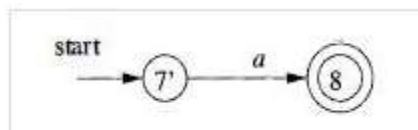


## 3.7 RE v.s. NFA/DFA

- 由RE构造NFA (Thompson算法), 为 $r=(a|b)^*abb$ 构造NFA

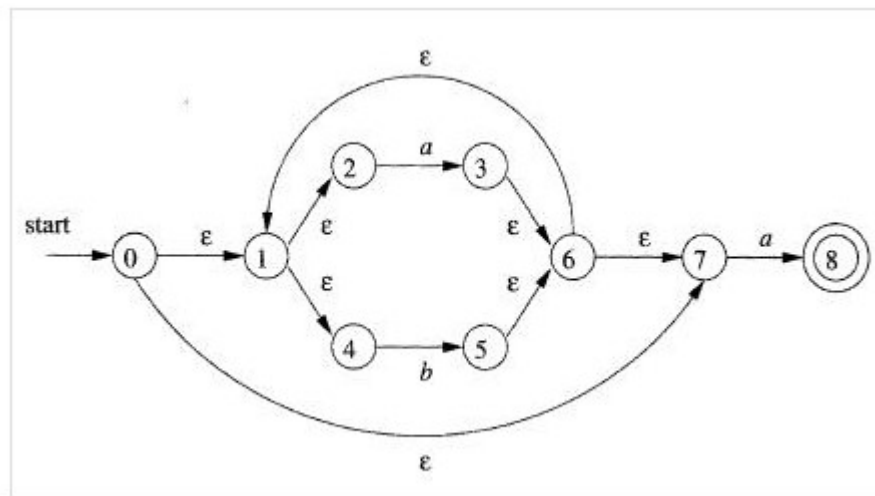
- 为表达式 $r_5 = r_3^*$ 构造NFA

- 为表达式 $r_6 = a$ 构造NFA



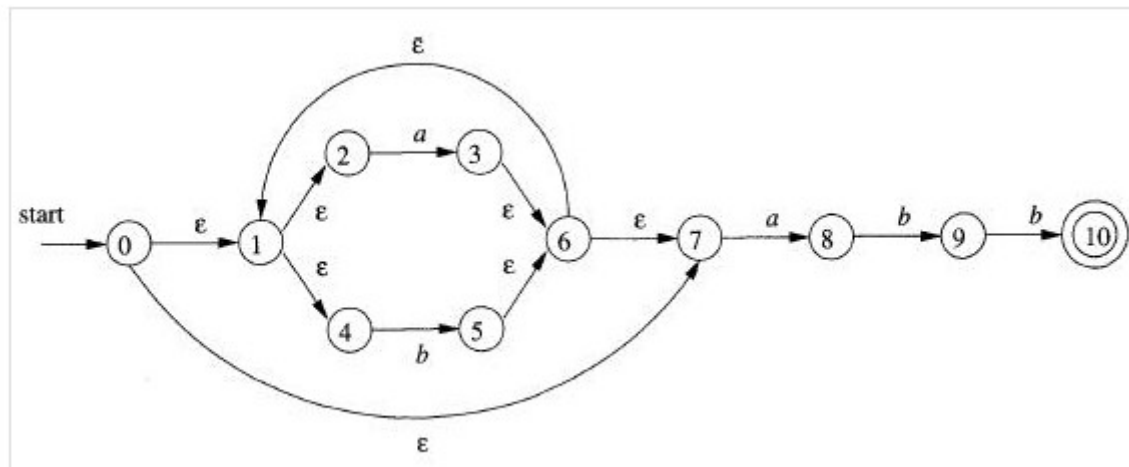
## 3.7 RE v.s. NFA/DFA

- 由RE构造NFA (Thompson算法), 为 $r=(a|b)^*abb$ 构造NFA
  - 为表达式 $r_7 = r_5r_6$ 构造NFA



## 3.7 RE v.s. NFA/DFA

- 由RE构造NFA (Thompson算法), 为 $r=(a|b)^*abb$ 构造NFA
  - 同理, 最终得到  $(a|b)^*abb$  的NFA





## 3.7 RE v.s. NFA/DFA

- **从RE生成FA，用来模拟RE的实现**
  - 方法一：生成NFA后，继续使用子集法构造与NFA等价的DFA
  - 然后最小化DFA ( to be discussed later )

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法一：RE  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  最小DFA
- 方法二：RE  $\rightarrow$  DFA  $\rightarrow$  最小DFA
- 方法三：RE  $\rightarrow$  NFA（然后直接模拟，模拟算法见教材P99，算法3.22）

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法二：由RE直接构造DFA

- 首先构造语法分析树，并标记位置
- $(a|b)^*abb \rightarrow (a|b)^*abb\#$

增广正则表达式  $(r)\#$

图 3-56 是一个正则表达式的抽象语法树。其中的小圆圈表示 cat 结点。

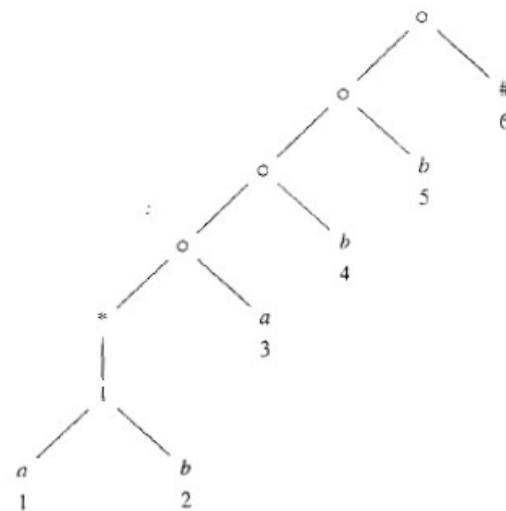


图 3-56  $(a|b)^*abb\#$ 的抽象语法树

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法二：由RE直接构造DFA，
  - NFA的重要状态：NFA状态有一个标号非 $\epsilon$ 的离开转换，则称该状态为重要状态(important state) --- 子集法在计算 $\text{move}(T, a)$ 的时候，只使用了重要状态
  - 计算四个函数 $\text{nullalbe}$ ,  $\text{firstpos}$ ,  $\text{lastpos}$ ,  $\text{followpos}$

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法二：由RE直接构造DFA，

- **nullable(n) returns true or false**：表示以n为根结点推导出的句子集合是否包括空串，“是”则nullable(n)=true；“否”则nullable(n)=false
- **firstpos(n)**定义了以结点n为根推导出的某个句子的第一个符号的**位置集合**
- **lastpos(n)**定义了以结点n为根推导出的某个句子的最后一个符号的**位置集合**---规则在本质上和计算firstpos的规则相同，但是在针对cat结点的规则中，左右子树的角色要对调

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

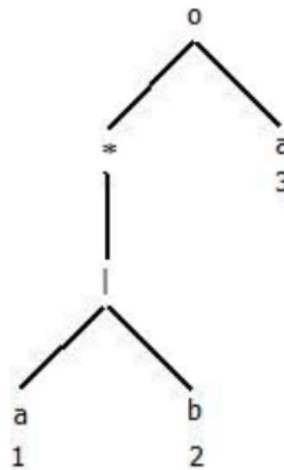
- 方法二：由RE直接构造DFA

- 小例子

$\text{nullable}(n) = \text{false}$

$\text{firstpos}(n) = \{1, 2, 3\}$

$\text{lastpos}(n) = \{3\}$



## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法二：由RE直接构造DFA，
  - 计算nullable, firstpos, lastpos

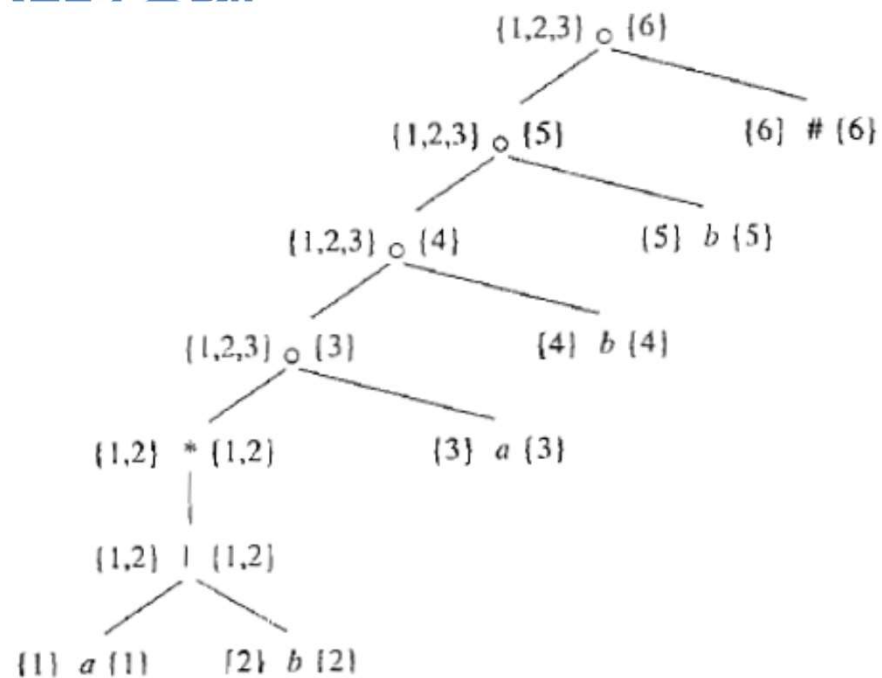
node n	nullable(n)	firstpos(n)	lastpos(n)
A leaf labeled $\epsilon$	true	$\emptyset$	$\emptyset$
A leaf with position i	false	{i}	{i}
An or-node $n=c_1 c_2$	nullable( $c_1$ ) or nullable( $c_2$ )	firstpos( $c_1$ ) $\cup$ firstpos( $c_2$ )	lastpos( $c_1$ ) $\cup$ lastpos( $c_2$ )
A cat-node $n=c_1c_2$	nullable( $c_1$ ) and nullable( $c_2$ )	if (nullable( $c_1$ )) firstpos( $c_1$ ) $\cup$ firstpos( $c_2$ ) else firstpos( $c_1$ )	if (nullable( $c_2$ )) lastpos( $c_2$ ) $\cup$ lastpos( $c_1$ ) else lastpos( $c_2$ )
A star-node $n=c_1^*$	true	firstpos( $c_1$ )	lastpos( $c_1$ )



## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法二：由RE直接构造DFA
- firstpos, lastpos  
for  $(a|b)^*abb \rightarrow (a|b)^*abb\#$



## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法二：由RE直接构造DFA，

##### ■ $\text{followpos}(p)$ 定义了一个和位置 $p$ 相关的、抽象语法树中某些位置的集合：

positions  $q$  is in  $\text{followpos}(p)$  iff 存在 $L((r)\#)$  中的某个串 $x=a_1a_2\dots a_n$ ，是的我们在解释为什么 $x$ 属于 $L((r)\#)$  可以将 $x$ 中某个 $a_i$ 和抽象语法树中的位置 $p$ 匹配，并将位置 $a_{i+1}$ 和位置 $q$ 匹配

语法树上  $pq$  两个位置的值 就是  $a_i a_{i+1}$ ， $a_i a_{i+1}$ 是RE可以接受字符串的一个子串

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法二：由RE直接构造DFA，
  - 计算followpos，只有下面两种情况会使得RE中一个位置跟在另一个位置之后
    - 当n是一个cat结点，且其左右子树分别为c1、c2，那么对于lastpos(c1)中的所有位置i，firstpos(c2)中的所有位置都在followpos(i)中。
    - 当n是一个star结点，且i是lastpos(n)中的一个位置，那么firstpos(n)中的所有位置都在followpos(i)中。

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法二：由RE直接构造DFA

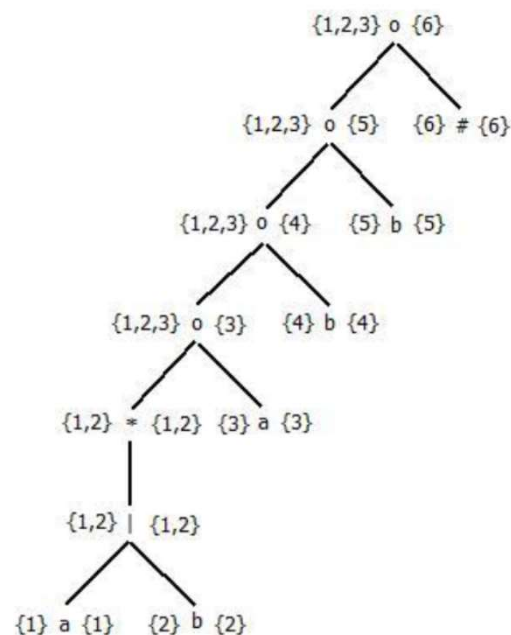
##### ■ followpos(p)

##### • Applying rule 1

- followpos(1) incl. {3}
- followpos(2) incl. {3}
- followpos(3) incl. {4}
- followpos(4) incl. {5}
- followpos(5) incl. {6}

##### • Applying rule 2

- followpos(1) incl. {1,2}
- followpos(2) incl. {1,2}



## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

- 方法二：由RE直接构造DFA
  - $\text{followpos}(p)$

位置 $n$	$\text{followpos}(n)$
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	$\emptyset$

图 3-60 函数  $\text{followpos}$

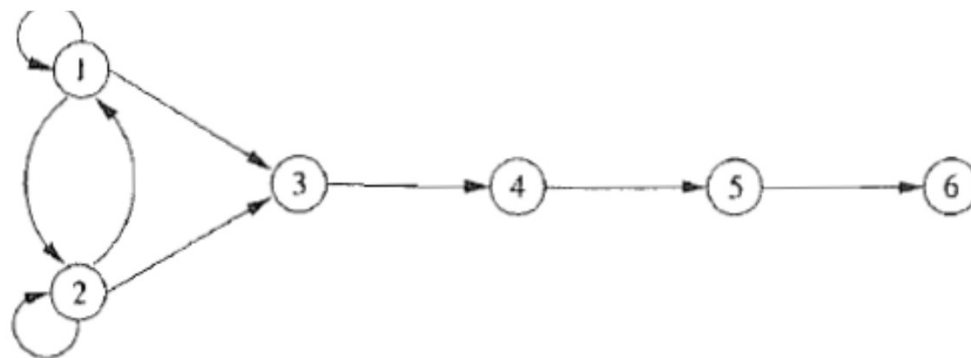


图 3-61 表示函数  $\text{followpos}$  的有向图

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法二：由RE直接构造DFA

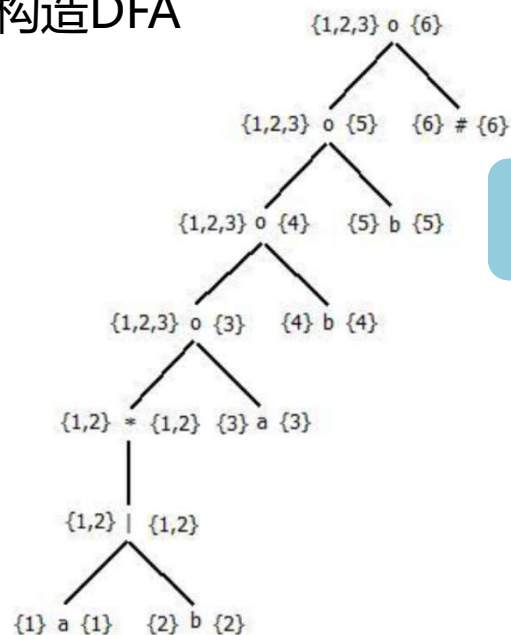
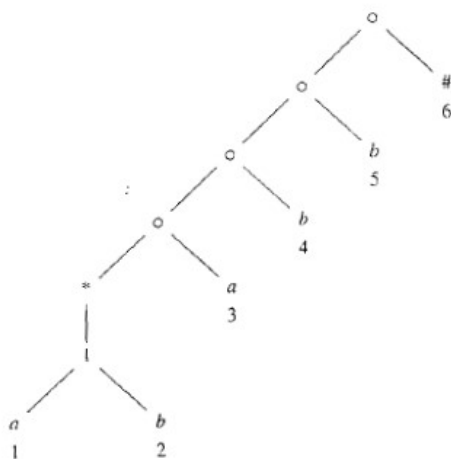
```
初始化  $Dstates$ , 使之只包含未标记的状态  $firstpos(n_0)$ ,  
    其中  $n_0$  是  $(r)^\#$  的抽象语法树的根结点;  
while (  $Dstates$  中存在未标记的状态  $S$  ) {  
    标记  $S$ ;  
    for ( 每个输入符号  $a$  ) {  
        令  $U$  为  $S$  中和  $a$  对应的所有位置  $p$  的  $followpos(p)$  的并集;  
        if (  $U$  不在  $Dstates$  中 )  
            将  $U$  作为未标记的状态加入到  $Dstates$  中;  
         $Dtran[S, a] = U$ ;  
    }  
}
```

图 3-62 从一个正则表达式直接构造一个 DFA

## 3.7 RE v.s. NFA/DFA

### ■ 从RE生成FA，用来模拟RE的实现

#### ■ 方法二：由RE直接构造DFA

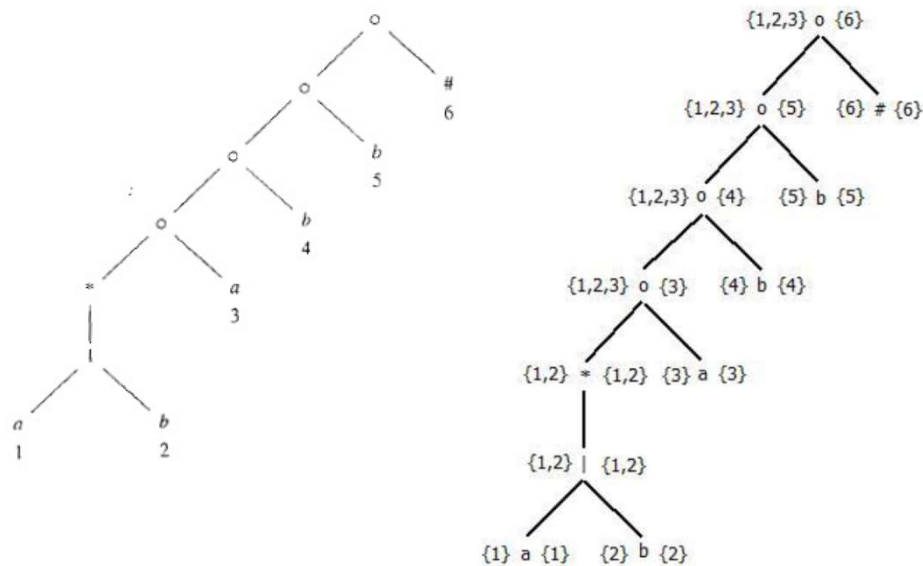


首先，DFA的开始状态定义为根节点 $n_0$ 的  $\text{firstpos}(n_0)=\{1,2,3\}$ ，标记为A

位置 $n$	$\text{followpos}(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\emptyset$

图 3-60 函数  $\text{followpos}$



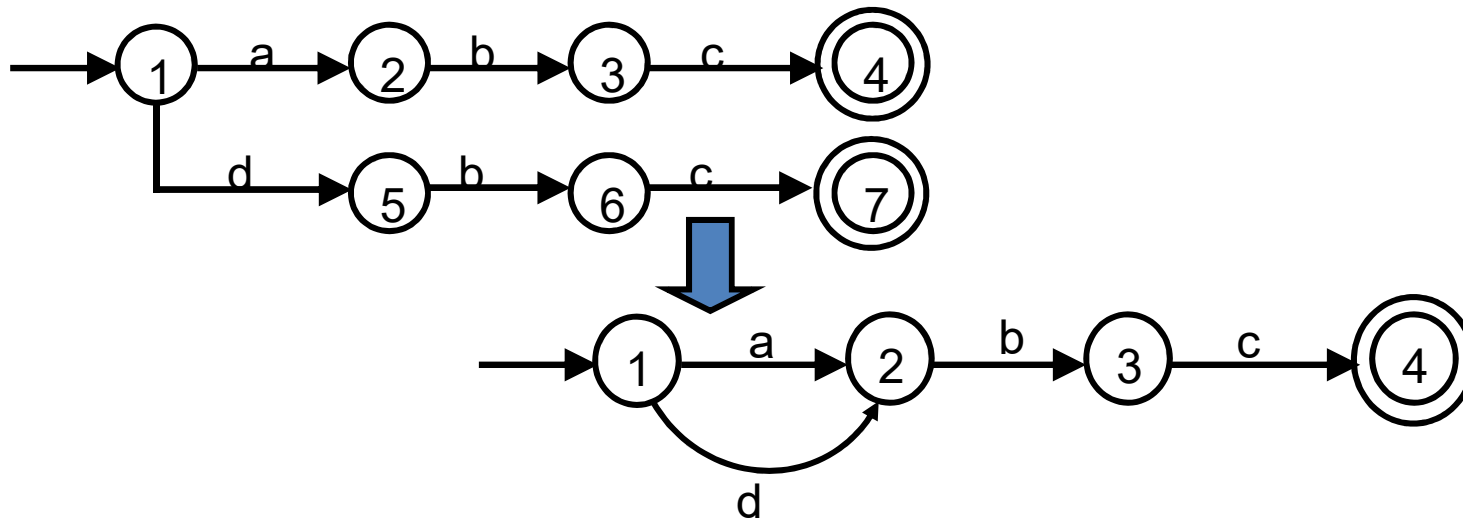
图 3-60 函数 *followpos*

```
Dtran[B,a]=followpos(1) U  
followpos(3)=B  
Dtran[B,b]=followpos(2) U  
followpos(4)={1,2,3,5}=C
```

## 3.7 RE v.s. NFA/DFA

### ■ DFA的最小化

- NFA转换成的DFA，有时候会有一些等价状态，这些等价状态会使分析效率降低，因此应合并。

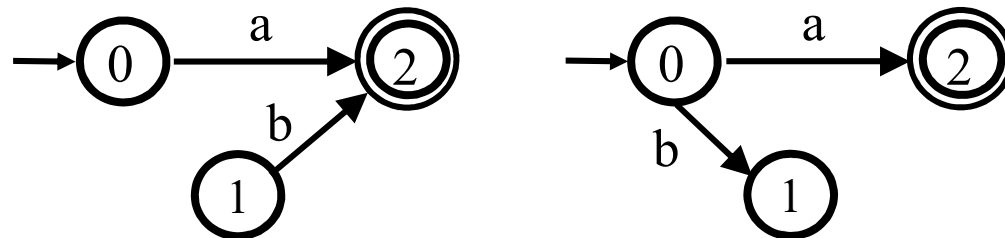


## 3.7 RE v.s. NFA/DFA

### ■ 最小DFA定义

- 如果DFA  $M$  没有无关状态，也没有等价状态，则称 $M$ 为最小(最简)自动机

- 无关状态：从开始状态没有到 $S$ 的通路，或 $S$ 到任意终止状态无通路，称 $S$ 为 $M$ 的无关状态



- 等价状态：对DFA中的两个状态 $S1$ 和 $S2$ ，如果将它们看作是初始状态，所接受的符号串相同，则定义 $S1$ 和 $S2$ 是等价的。

## 3.7 RE v.s. NFA/DFA

- **所以自动机最小化就是两个问题**
  - 一个是合并可以合并的等价状态（比较麻烦）
  - 一个是删去无用的无关状态（直接删除）

## 3.7 RE v.s. NFA/DFA

- **两个状态S1和S2等价的条件**
  - 一致性条件：S1和S2同时为可接受状态或不可接受状态。
  - 蔓延性条件：S1和S2对所有输入符号必须都要转换到等价状态里。
- **DFA终止状态和非终止状态是不等价的。**

## 3.7 RE v.s. NFA/DFA

### ■ DFA化简的两种方式

- 合并等价状态; (状态合并法)
- 分离不等价状态; (状态分离法)



## 3.7 RE v.s. NFA/DFA

### ■ 状态分离法化简DFA

- 输入：一个DFA  $D$ ，其状态集合为 $S$ ，输入字母表为 $\Sigma$ ，开始状态为 $s_0$ ，接受状态为 $F$ 。
- 输出：一个DFA  $D'$ ，它和 $D$ 接受同样的语言，且状态数最少。
- Notation:  $\Pi$  即DFA状态的一个划分 $\{S_1, S_2, \dots\}$ 和 $S - \{S_1, S_2, \dots\}$
- 方法：
  - 1)首先构造包含两个组 $F$ 和 $S-F$ 的初始划分 $\Pi$ ，这两个组分别是 $D$ 的接受状态组和非接受状态组。
  - 2)用下面的构造新的分划 $\Pi_{\text{new}}$

分割原则：分离出不等价状态

```
initially, let  $\Pi_{\text{new}} = \Pi$ ;  
for ( each group  $G$  of  $\Pi$  ) {  
    partition  $G$  into subgroups such that two states  $s$  and  $t$   
        are in the same subgroup if and only if for all  
        input symbols  $a$ , states  $s$  and  $t$  have transitions on  $a$   
        to states in the same group of  $\Pi$ ;  
    /* at worst, a state will be in a subgroup by itself */  
    replace  $G$  in  $\Pi_{\text{new}}$  by the set of all subgroups formed;  
}
```

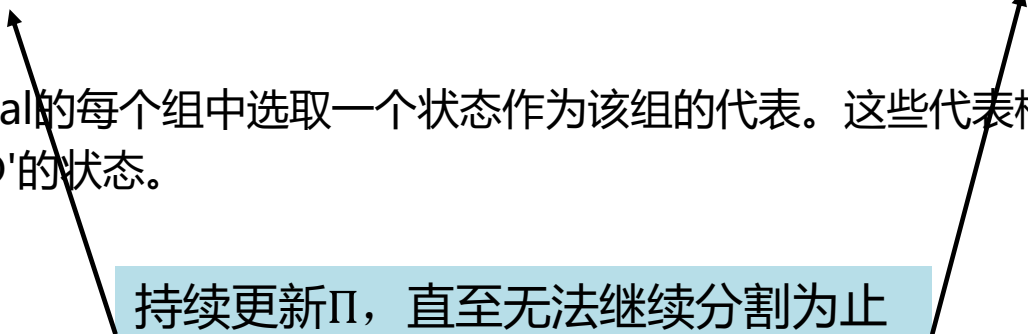
## 3.7 RE v.s. NFA/DFA

### ■ 状态分离法化简DFA

#### ■ 方法：

- 3)如果 $\Pi_{\text{new}} = \Pi$ ,令 $\Pi_{\text{final}} = \Pi$ 并接着执行步骤 4), 否则, 用 $\Pi_{\text{new}}$ 替换 $\Pi$ 并重复步骤2)。
- 4)在分划 $\Pi_{\text{final}}$ 的每个组中选取一个状态作为该组的代表。这些代表构成了状态最少DFA  $D'$ 的状态。

持续更新 $\Pi$ , 直至无法继续分割为止  
(即, 该状态子集中的状态都为等价)



## 3.7 RE v.s. NFA/DFA

- DFA  $D = (\{0,1,2,3,4,5\}, \{a,b\}, \delta, 0, \{0,1\})$ , 其中  $\delta$  见表

状态	a	b
0	1	2
1	1	4
2	1	3
3	3	2
4	0	5
5	5	4

Step 1: 根据一致性条件  
 $A = \{0,1\}$  ;  $B = \{2,3,4,5\}$ 。



状态	类别	a	b
0	A	1(A)	2(B)
1	A	1(A)	4(B)
2	B	1(A)	3(B)
3	B	3(B)	2(B)
4	B	0(A)	5(B)
5	B	5(B)	4(B)

## 3.7 RE v.s. NFA/DFA

- DFA  $D = (\{0,1,2,3,4,5\}, \{a,b\}, \delta, 0, \{0,1\})$ , 其中  $\delta$  见表

状态	类别	a	b
0	A	1(A)	2(B)
1	A	1(A)	4(B)
2	B	1(A)	3(B)
3	B	3(B)	2(B)
4	B	0(A)	5(B)
5	B	5(B)	4(B)

Step2: 根据蔓延性条件，  
对状态进行再分类

状态	a	b
0	1	2
1	1	4
2	1	3
3	3	2
4	0	5
5	5	4

状态	类别	a	b
0	A	1(A)	2(B)
1	A	1(A)	4(B)
2	B	1(A)	3(C)
3	C	3(C)	2(B)
4	B	0(A)	5(C)
5	C	5(C)	4(B)

不可再分

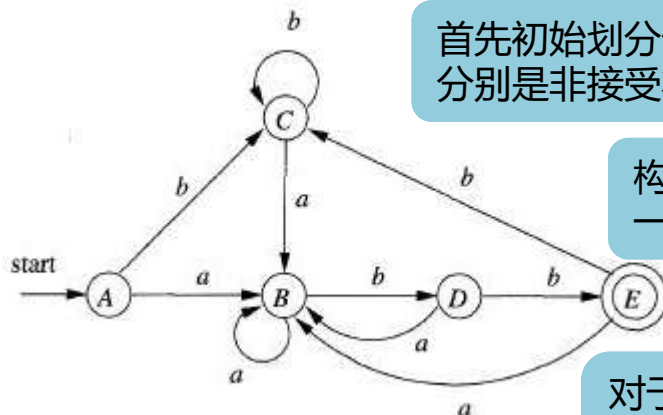
## 3.7 RE v.s. NFA/DFA

- DFA  $D = (\{0,1,2,3,4,5\}, \{a,b\}, \delta, 0, \{0,1\})$  最小化为 :  
DFA  $D' = (\{A,B,C\}, \{a,b\}, \delta, A, \{A\})$  , 其中  $\delta$  见表

状态	a	b
A	A	B
B	A	C
C	C	B

## 3.7 RE v.s. NFA/DFA

### ■ 对 $r=(a|b)^*abb$ 的 DFA化简



首先初始划分包括两个组 $\{A,B,C,D\}, \{E\}$ , 分别是非接受状态组和接受状态组。

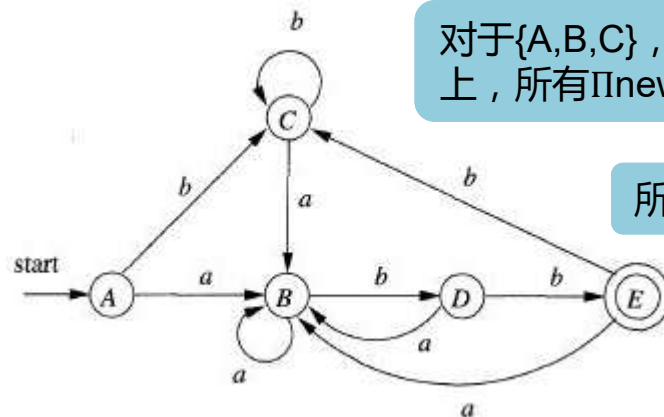
构造 $\Pi_{new}$ 时, 考虑这两个组和输入符号 $a$ 和 $b$ , 因为组 $\{E\}$ 只包含一个状态, 不能再被分割, 所以 $\{E\}$ 被原封不动的保留在 $\Pi_{new}$ 中。

对于 $\{A,B,C,D\}$ , 是可以被分割的, 因此我们必须考虑各个输入符号的作用, 在输入 $a$ 上, 这些状态中的每一个都转到 $B$ , 因此使用以 $a$ 开头的串无法区分这些状态。

但对于输入 $b$ , 状态 $A, B, C$ 都转换到 $\{A,B,C,D\}$ 的某个成员上, 而 $D$ 转到另一组中的成员 $E$ 上, 因此在 $\Pi_{new}$ 中,  $\{A,B,C,D\}$ 被分割成 $\{A,B,C\}, \{D\}$ , 现在 $\Pi_{new}$ 中有 $\{A,B,C\}, \{D\}, \{E\}$ 。

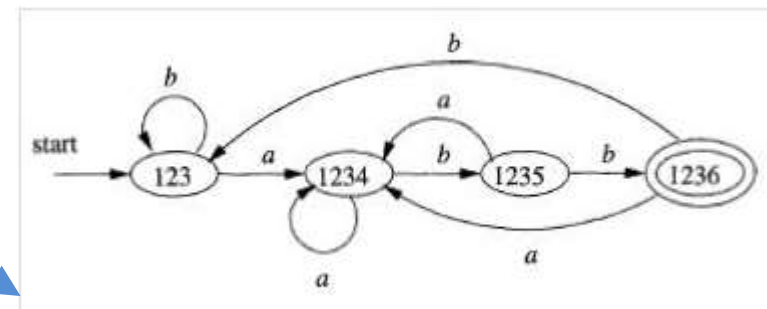
## 3.7 RE v.s. NFA/DFA

### ■ 对 $r=(a|b)^*abb$ 的 DFA化简



对于 $\{A,B,C\}$ ，在输入 $b$ 上， $A$ 和 $C$ 都到达 $\{A,B,C\}$ 中的元素， $B$ 却到达 $D$ 上，所有 $\Pi_{new}$ 有 $\{A,C\},\{B\},\{D\},\{E\}$ ，对于 $\{A,C\}$ 无法在分割。

所有最后有 $\{A,C\},\{B\},\{D\},\{E\}$ ，构造如下的DFA





## 3.7 RE v.s. NFA/DFA

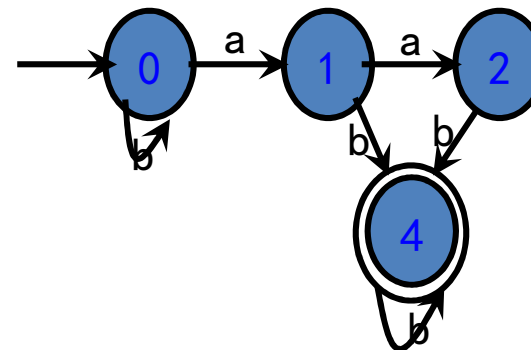
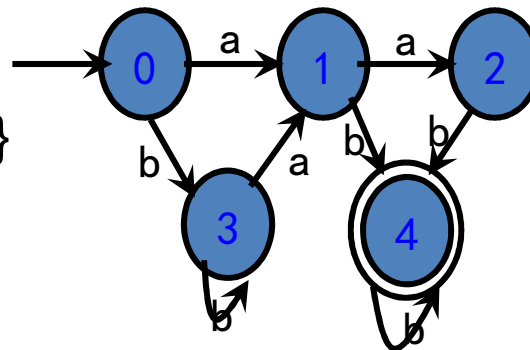
### ■ 状态分离法化简DFA

- D'的其他部分按如下步骤构建：
  - (a) D'的开始状态是包含了D的开始状态的组的代表。
  - (b) D'的接受状态是那些包含了D的接受状态的组的代表。
  - (c) 令s是 $\Pi_{\text{final}}$ 中某个组G的代表，并令DFA D中在输入a上离开s的转换到达状态t。令r为t所在组H的代表，那么在D'中存在一个从s到r在输入a上的转换。

## 3.7 RE v.s. NFA/DFA

### ■ 例1:

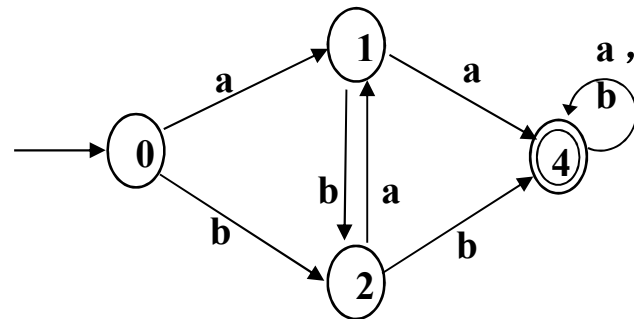
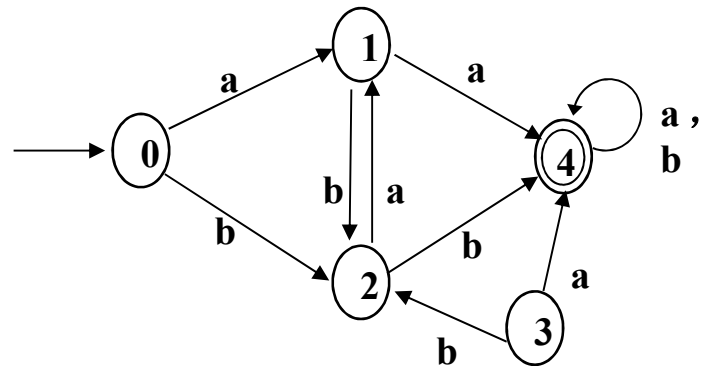
- $\{0, 1, 2, 3\}$ 和 $\{4\}$
- $\{0, 1, 3\}$ ,  $\{2\}$ 和 $\{4\}$
- $\{0, 3\}$ ,  $\{1\}$ ,  $\{2\}$ 和 $\{4\}$



## 3.7 RE v.s. NFA/DFA

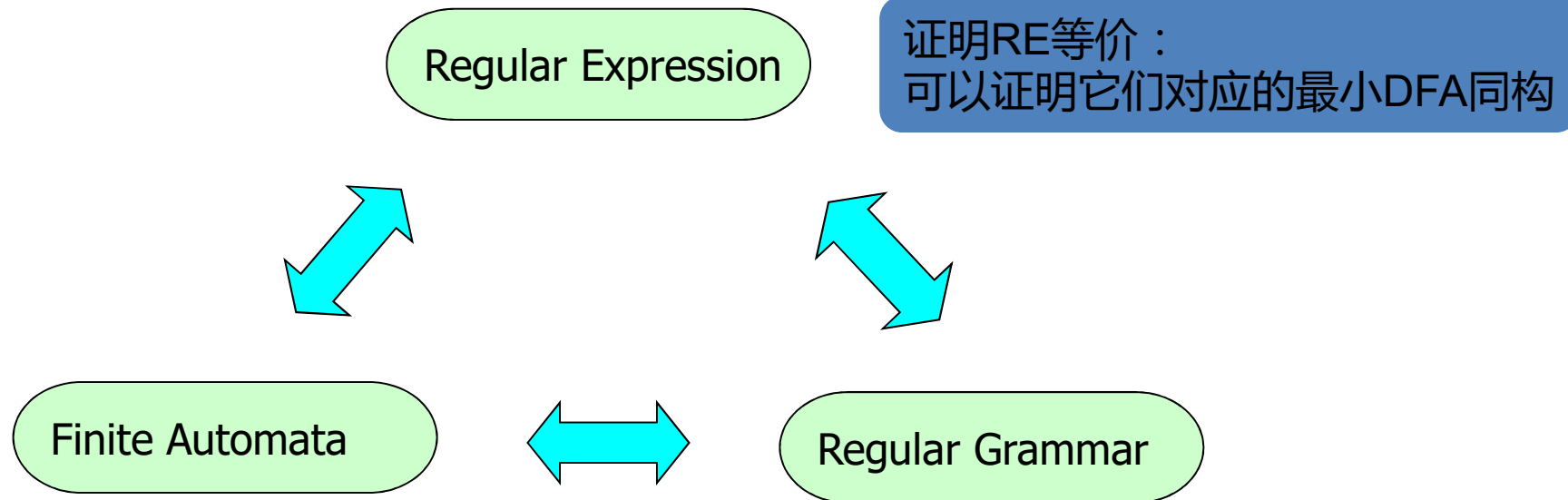
### ■ 例2:

- $\{0, 1, 2, 3\}, \{4\}$
- $\{0, 2\}, \{1, 3\}, \{4\}$
- $\{0\}, \{2\}, \{1, 3\}, \{4\}$



## 3.7 RE v.s. NFA/DFA

### ■ RE , DFA(NFA) , L(RE)三者等价



## 作业

- 教材P105 : 3.7.3(4)
- 教材P109 : 3.8.1

## 附加思考题

- 教材P118 : 3.9.3 (用算法3.36构造) , 3.9.4

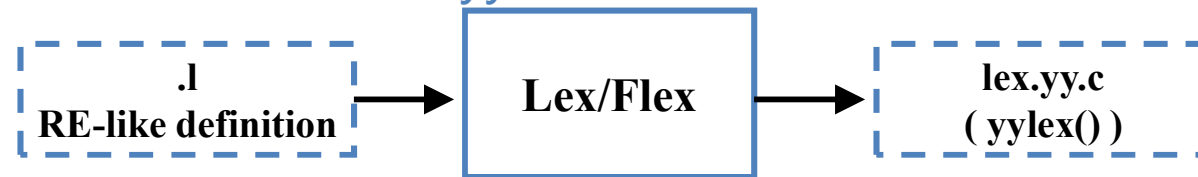


## 3.8 词法分析器的开发

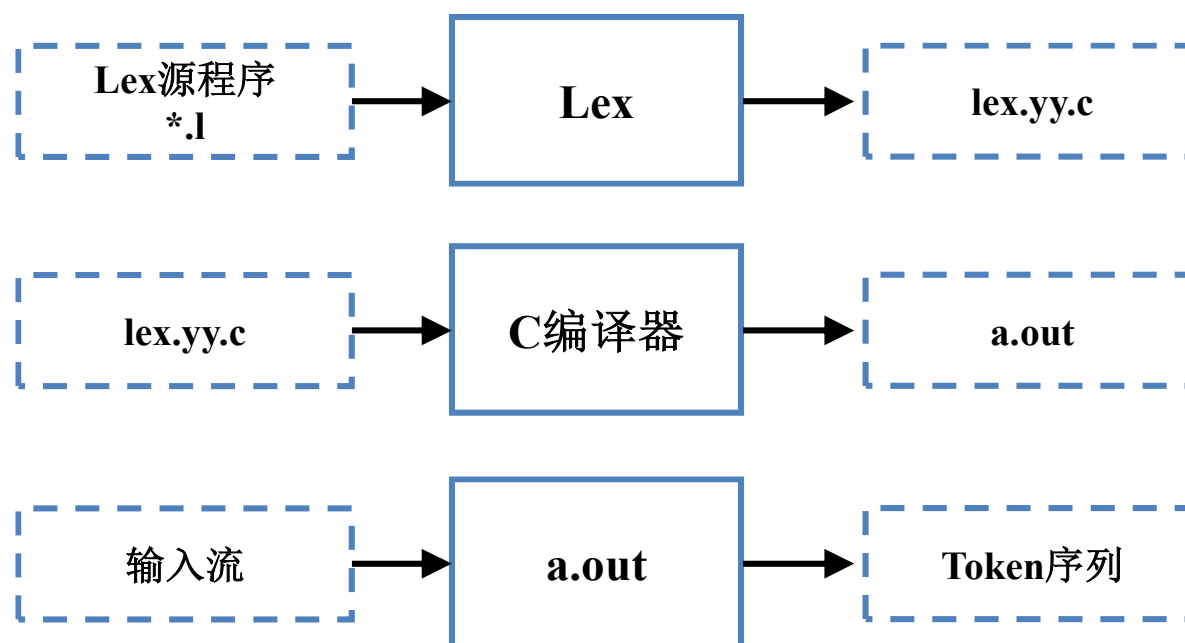


## 3.8 词法分析器的开发

- LEX ( Lexical Analyzer Generator ) 即词法分析器的生成器，是由贝尔实验室于1972年在UNIX操作系统上首次开发的。最新版本是FLEX(Fast Lexical Analyzer Genrator)
- 工作原理：LEX通过对Lex源文件(.l文件)的扫描，经过宏替换将规则部分的正则表达式转换成与之等价的DFA，并产生DFA的状态转换矩阵；利用该矩阵和Lex源文件中的C代码一起产生一个名为yylex()的词法分析函数，并将yylex()函数拷贝到输出文件lex.yy.c中。



## 3.8 词法分析器的开发



用Lex创建一个词法分析器的过程

## 3.8 词法分析器的开发

### ■ Lex源文件

声明部分

%%

转换规则

%%

辅助函数

动作中需要使用的函数

```
int Change()
```

```
{ /*将字符串形式的常数  
   转换成整数形式*/
```

```
}
```

%{ 常量

%

正则定义

模式 {动作}:

- 模式是一个正则表达式或者正则定义
- 动作通常是C语言代码，表示匹配该表达式后应该执行的代码。

%{ ID,NUM,IF,ADD

%

letter [A-Za-z]

digit [0-9]

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

num {digit}+

if {return (IF);}

+ {return(ADD);}

{id} {yyval = strcpy(yytext,  
 yylength); return(ID); }

{num} {yyval = Change();  
 return(NUM);}

yyval: token的值

yytext: token的lexeme

yylen: lexeme的长度

## 3.8 词法分析器的开发

### ■ Lex例子

词素	词法单元名字	属性值
Any ws	-	-
if	if	-
then	then	-
else	else	-
Any id	id	指向符号表条目的指针
Any number	number	指向符号表条目的指针
<	relop	LT
<=	relop	LE
=	relop	EQ
>	relop	NE
>	relop	GT
>=	relop	GE

图 3-12 词法单元、它们的模式以及属性值

```
%{
/* 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}     { yyval = (int) installID(); return(ID); }
{number} { yyval = (int) installNum(); return(NUMBER); }
"<"     { yyval = LT; return(RELOP); }
"<="    { yyval = LE; return(RELOP); }
"="      { yyval = EQ; return(RELOP); }
">"     { yyval = NE; return(RELOP); }
">"     { yyval = GT; return(RELOP); }
">="    { yyval = 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 numer-
                   ical constants into a separate table */
}
```

图 3-23 识别图 3-12 中的词法单元的 Lex 程序

## 3.8 词法分析器的开发

### ■ 冲突解决

当输入与长度不同的多个模式匹配时，Lex选择长模式进行匹配

当输入与长度相同的多个模式匹配时，Lex选择列于前面的模式进行匹配

%%

program printf(“%s\n”,yytext);/\*模式1\*/

procedure printf(“%s\n”,yytext);/\*模式2\*/

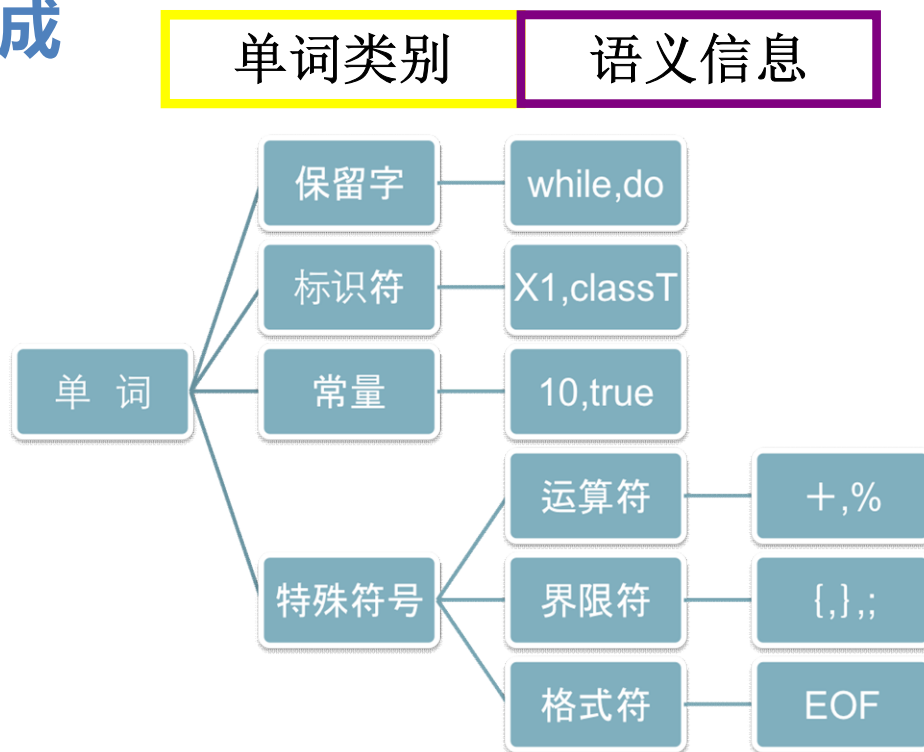
[a-z][a-z0-9]\* printf(“%s\n”,yytext);/\*模式3\*/

当输入串为 “programming”时，模式1（匹配 “program”）和模式3（ “programming”）都匹配，但会选择匹配串长的模式3。

当输入串为 “program”时，因为模式1和模式3匹配的串长度相等故会选择模式1。

## 3.8 词法分析器的开发

### 需要定义的词法组成 TOKEN结构图





*Thank you!*