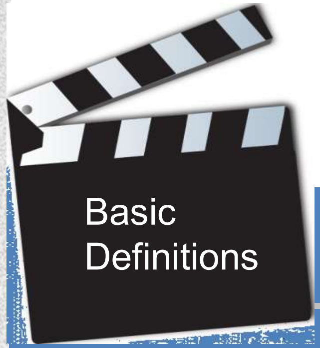
Lecture 3: Lexical Analysis（Part I）

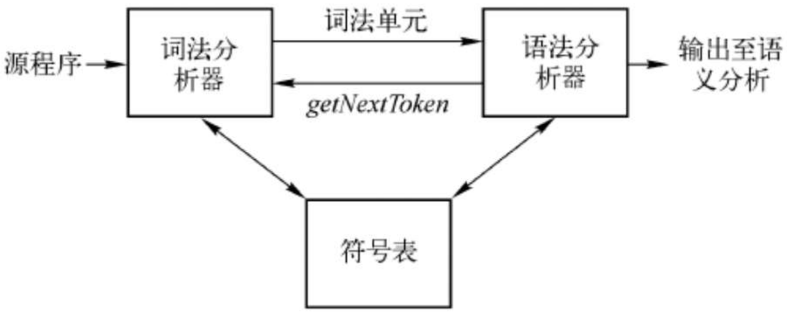
Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301



3.1 基本概念

3.1 基本概念

- **词法分析(lexical analysis)，也称scanning,**
  - 编译程序的第一阶段,其作用是识别单词(程序意义上)并找出词法错误.
- **读入源程序的输入字符、将它们拆分成词素，生成并输出一个词法单元序列，每个词法单元对应于一个词素**
- **常见的做法**
  - 由语法分析器调用，
    需要的时候不断读取、生成词法单元
  - 可以避免额外的输入输出



- **在识别出词法单元之外，还会完成一些不需要生成词法单元的简单处理，比如删除注释、将多个连续的空白字符压缩成一个字符等**

3.1 基本概念

- **词素（Lexeme）**
  - 源程序中的字符序列，它和某类词法单元的模式匹配，被词法分析器识别为该词法单元的实例。
- **词法单元 (Token) <词法单元名、属性值(可选)>**
  - 单元名是表示词法单位种类的抽象符号，语法分析器通过单元名即可确定词法单元序列的结构, 是有意义的最小的程序单位
  - 属性值通常用于语义分析之后的阶段
    - 例1: x+12, token有3个：x, +, 12
    - 例2：x12 + y, token有3个：x12, +, y
    - 例3：“This is a test.”，token有1个：“This is a test.”
    - 例4：

```
if (x==y)
    z=0;
else
    z=1;
```

token有15个：

标识符(Identifier)、
关键字(Keyword)、
数(Integer,float)、
格式符(Whitespace)
运算符(Operator)
其他符号({,},[,],;,:,)

3.1 基本概念

- **词性划分: 根据作用对程序子串进行分类**
  - x12 + y → id + id
- **词法分析的输出是单词(token)的序列**
  - (id, x12) (+,-) (id, y)
- **Token序列将作为语法分析程序的输入**

3.1 基本概念

- **词法分析流程**
  - 输入：字符串 (ASCII)

    E.g \tif (x==j)\n\t\tz=0;\n\telse\n\t\tz=1;
  - Tokenize

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

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

↓

(if,-),((,-),(id,x),(==,-),(id,j),(),-),(id,z),(=,-),
(num,0),(;,-),(else,-),(id,z),(=,-),(num,1),(;,-)

3.1 基本概念

Program (character stream):

position = initial + rate * 60

Lexical Analyzer (Scanner)

Token Stream:

⟨id，指向符号表中position条目的指针⟩, ⟨assign _ op⟩, ⟨id，指向符号表中initial条目的指针⟩, ⟨add_op⟩, ⟨id，指向符号表中rate条目的指针⟩, ⟨mul_ op⟩, ⟨number，整数值60⟩

报错：18..23 + val#ue

Variable names cannot have ‘#’ character

Not a number

3.1 基本概念

- ## 更多例子

2*3.14*r
(num, 2), (*,-), (num,3.14), (*,-), (id, r)

while (1) {x = 0;}
(while, -), ((,-), (num,1), (),-), ({,-), (id,x),
(=,-), (num, 0), (;,-), (},-)

“This is a test program.”
(string, “This is a test program”)

3.1 基本概念

- 词法分析程序除识别单词外，还要完成词法错误检查。
- 词法错误：词法分析只能检查很有限的错误。
  - 非法字符：出现语言字母表以外的字符，@
  - 不封闭的字符、字符串、注释等：‘A， “This ，/*This is a test

Regular Languages

3.2 正则语言

3.2 正则语言

- **A language to define lexical structure**
  - describes how to generate tokens of a particular type
  - a regular expression == a set of strings

3.2 正则语言

- **概念回顾**
  - 字母表：是元素的非空有穷集合，记为“Σ”，Σ中的元素可称为字母、符号、字符。
  - 定义在字母表Σ上的语言：是从Σ中抽取的字符构成的一些字符串的集合，记为$L(\Sigma)$---定义在同一个字母表上的语言有很多!
  - Σ定义了语言中允许出现的全部符号。
    - 例1： Σ = 英文字母, $L(\Sigma)$是英文句子
    - 例2： Σ = ASCII, $L(\Sigma)$是C语言程序

不是所有的英文字母串是英文句子，ASCII字母表还可以定义Java程序

3.2 正则语言

- $L(\Sigma)$是字符串，但不是Σ上任意的字符串，是满足某种**规则**的字符串 --- 如何定义规则？
- **三型文法的表达能力足以满足这种规则**

3.2 正则语言

- **Chomsky 3型文法:正规文法**
  - P中产生式具有形式$A\rightarrow \alpha B$， $A\rightarrow \alpha$ (左线性)，或者$A\rightarrow B\alpha$，$A\rightarrow \alpha$(右线性)， 其中A，$B\in V_N$，$\alpha\in V_T^*$。
  - 也称为正规文法RG、线性文法：若所有产生式均是左线性，则称为左线性文法；若所有产生式均是右线性，则称为右线性文法。
  - 产生式要么均是右线性产生式，要么是左线性产生式，不能既有左线性产生式，又有右线性产生式。

3.2 正则语言

- **用字符串来定义语言**

设三型文法G_1=({S},{a,b}, S,P)，其中P为:

(0) $S \rightarrow aS$

(1) $S \rightarrow a$

(2) $S \rightarrow b$　　$L(G_1) = \{a^i (a \mid b) \mid i >= 0\}$

- **正则表达式(regular expressions, RE)**
  - 定义正则语言(regular languages, RL)的标准工具
  - 其定义的集合叫做正则集合(regular set)
  - 是词法单元的规约
- **识别3型语言的自动机称为有限状态自动机(FA)。**

3.2 正则语言

- **首先定义正则表达式定义中的四种运算的作用**
  - 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 · ：表示连接，经常省略，如r·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

3.2 正则语言

- **定义正则表达式**（Σ 为字母表）
  - 原子正则表达式(atomic regular expressions)
    - ε和∅ 是∑上的正则表达式，它们所表示的正则集分别为L(ε)={ε}， L(∅)={}.
    - 对任何a∈∑,a 是∑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∑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∑上的正则表达式，表示的正则集L((r))= L(r) --- （ ）在这里是操作符！！！
    - r|s也是∑上的正则表达式，表示的正则集L(r|s)= L(r) ∪ L(s)
    - r ·s也是∑上的正则表达式，表示的正则集L(r · s)= L(r)L(s)
    - r^*也是∑上的正则表达式，表示的正则集$L(r^*)$= (L(r))*
    - 有限次使用上述3条规则构成的表达式称为∑上的正则表达式，表示的字符串集合称为∑上的正则集或正规集。--- 上述操作可以满足三型文法，但并不意味着符合具体某种词法的规定

3.2 正则语言

- **首先定义正则表达式定义中的四种运算的作用**
  - 括号(r)：主要是用于确定运算优先关系
  - 或运算 |：把复杂问题分成几个种情况依次定义正则表达式，然后把这些正则表达式用或运算连接起来描述整个问题。
  - 连接运算 · ：把一个大问题分成前后关联的几个部分依次定义正则表达式，然后把各部分正则表达式按先后顺序用连接运算连接起来描述问题
- **实际应用中会扩充很多正则表达式的运算，如：**
  - r^+也是∑上的正则表达式，表示的正则集$L(r^+) = (L(r))^+$
  - 运算的优先级：* > 连接符 > ， (a)|((b)*(c)) 可写为 a|b*c

3.2 正则语言

- **正规式的例子，例1：∑ = {a, b}**
  - a | b, ab, a^*,b^*, ab*, a(a|b)* 也是∑上的正则表达式
  - L(a|b) = {a, b}
  - L(ab) = {ab}
  - L(a^*) = {ε,a,aa,aaa,...}
  - L(b^*) = {ε, b, bb, bbb,...}
  - L(aa | ab | ba | bb) = {aa, ab, ba, bb}
  - L((a | b) (a | b)) = {aa, ab, ba, bb}
  - L($(a | b)^*$) = 由a和b构成的所有串集
  - L(ab*) = {a,ab,abb,…}
  - L(a(a|b)*) = {a,aa,ab,aaa, aab,aba,abb,aaaa, aaab,…}

3.2 正则语言

- **例2：**
  - （ 00 | 11 | （(01 | 10) (00 | 11)* (01 | 10)））*
  - 句子：0100110100000100000101111001

3.2 正则语言

- **例3：程序设计语言的单词Σ = ASCII,**
  - 整数 ：非空的数字序列 (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)$^+$

    （正闭包运算符：+ 　$A^+ = AA^*$ ）

正则定义：给正则表达式起个名字,避免过长的RE定义

Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Integer = Digit$^+$

3.2 正则语言

- 例4：程序设计语言的单词Σ = ASCII, 保留字：else , if, while, int, float, ……
  - ELSE ： e • l • s • e
  - IF : i • f
  - WHILE : w • h • i • l • e
  - INT : i • n • t
  - FLOAT : f • l • o • a • t
  - ……
- Keywords = ELSE | IF | WHILE | INT | FLOAT | ……

3.2 正则语言

- **例5：程序设计语言的单词∑ = ASCII, 标识符：以字母开始的，由字母和数字构成的串 (a | b | … | z | A | B | …| Z)(a | b | …|z|A|B|…|Z|0|1|…|9)***
  - Letter = a | b | … | z | A | B | …| Z
  - Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - Identifier = Letter (Letter|Digit)*

    注：Letter (Letter|Digit)* 等价于 Letter Letter* | Letter Digit*

3.2 正则语言

- **例6：程序设计语言的单词∑ = ASCII, 运算符：+, - ,*,/**
  - ADD：+
  - SUB：-
  - MUL：*
  - DIV：/
  - Operator = ADD | SUB | MUL | DIV

3.2 正则语言

- **例7：程序设计语言的单词Σ = ASCII, 空白：空格，换行，Tab**
  - ‘ ’
  - \n
  - \t
  - Whitespace = (‘ ’ | \n | \t)$^+$
  - Words = Integer | Keywords | Identifier | Operator | Whitespace

3.2 正则语言

- **正则表达式随处可见！电话号码、身份证号、学号、Email地址等。**
- **例9：Email地址的例子,**
  - xiaoyang2011@163.com 或xiaoyang2011@jlu.edu.cn
  - Σ = {字母，数字，@,.}
  - Names = (Letter | Digit) $^{+}$
  - Address = Names@(Names.| ε) Names.Names

3.2 正则语言

- **正则表达式性质：Letter (Letter|Digit)* 等价于 Letter Letter* | Letter Digit***
  - 如果两个正则表达式r和s表示同样的语言，则称r和s等价，记作 r = s.
  - A | B = B | A　　　　| 的可交换性
  - A | (B | C)=(A | B)| C　　　　| 的可结合性
  - A (B C) =(A B)C　　　　连接的可结合性
  - A (B | C) =A B | A C　　　　连接的可分配性
  - (A | B) C =A C | B C　　　　连接的可分配性
  - $A^{**} = A^{*}$　　　　幂的等价性
  - A ε = ε A= A　　　　ε是连接的恒等元素

3.2 正则语言

- 练习：设字母表Σ={0，1},试写正则表达式
  1. 所有Σ上定义的串
  2. 表示二进制数
  3. 能被2整除的二进制

3.2 正则语言

- **正则语言(regular language, RL)：可用一个正则表达式定义的语言叫做正则语言**
- **一般地，程序设计语言的单词是正则语言，可用RE定义。**
- **正则表达式局限性：RE 不能定义具有下列结构的语言**
  - 对称结构：
    - 例如 $A = \{a^n\ b\ a^n \mid n > 0\}$,
    - A不能用RE定义，因为a^+ba^+不能保证b两侧a的个数相等
  - 嵌套结构：
    - 例如简单算术表达式的定义

      1) $n \in AE$ 2) $(AE) \in AE$ 3) $AE+AE \in AE$

3.2 正则语言

- **正则表达式和正则文法等价，可以互相转化**

设文法G_1＝({S},{a,b}, S,P)，其中P为:

(0) $S \to aS$

(1) $S \to a$

(2) $S \to b$ $\quad L(G_1) = \{a^i (a \mid b) \mid i >= 0\}$

- 所以，上述例子中RE对应的产生式都是什么？
- 思考：如何证明正则表达式满足三型文法规定？

3.3 有限状态自动机

3.3 有限状态自动机

- **正则表达式 – specification(便于书写理解)；有限自动机 – Implementation(便于计算机执行)**
  - 有限自动机是描述有限状态系统的数学模型。
- **有限状态系统：**
  - 状态：是将事物区分开的一种标识.
  - 具有离散状态的系统：如数字电路(0,1);电灯开关(on,off);十字路口的红绿灯；其状态数是有限的。
  - 具有连续状态的系统：水库的水位、室内的温度等可以连续发生变化；可以有无穷个状态.
  - 有限状态系统是离散状态系统。
- **在很多领域，如网络协议分析、形式验证、代码安全、排版系统等有重要应用。**

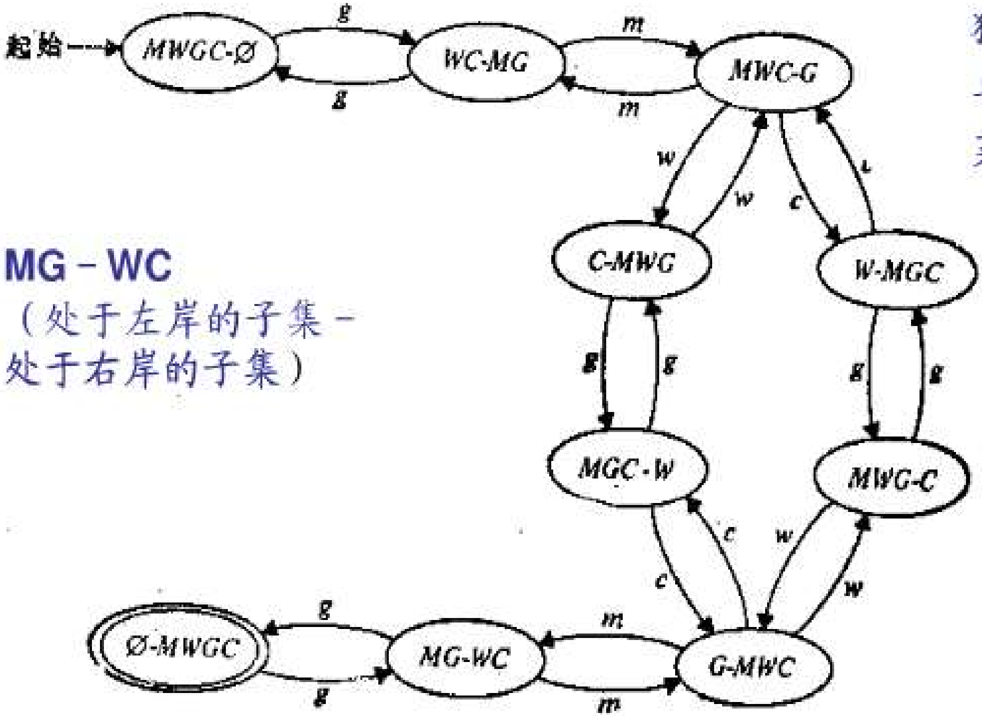
3.3 有限状态自动机

- **有限自动机的例子-经典的过河问题**
  - 一个人带着一头狼，一头羊，以及一棵白菜处于河的左岸。人和他的伴随品都希望渡到河的右岸。有一条小船，每摆渡一次，只能携带人和其余三者之一。如果单独留下狼和羊，狼会吃羊；如果单独留下羊和白菜，羊会吃菜。怎样才能渡河，而羊和白菜不会被吃掉呢？

3.3 有限状态自动机

- 过河问题模型化

人(M)
狼(W)
羊(G)
菜(C)

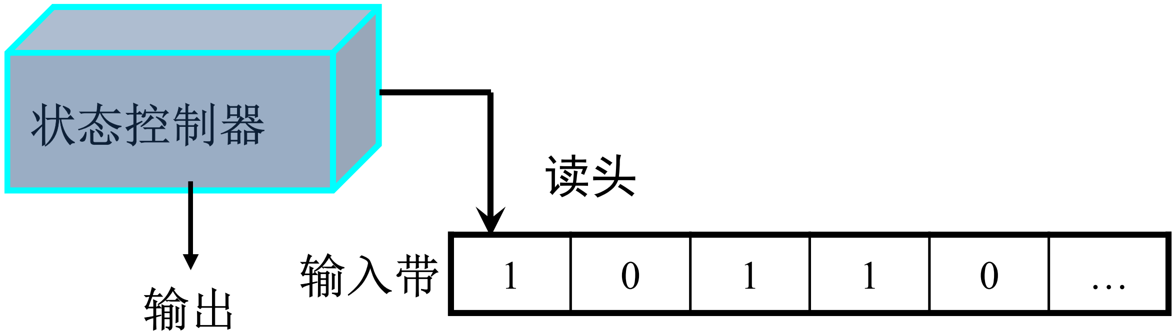


MG－WC
（处于左岸的子集－
处于右岸的子集）

3.3 有限状态自动机

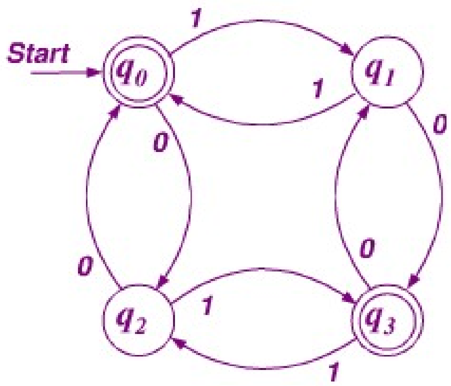
■ 有限自动机FA可以理解成状态控制器

- FA有有限个状态，其中有初始状态，终止状态
- 起始：处于初始状态，读头位于输入带开头
- 中间：从左到右依次读取字符，发生状态迁移
- 结束：读头到达输入带末尾，状态到达终态



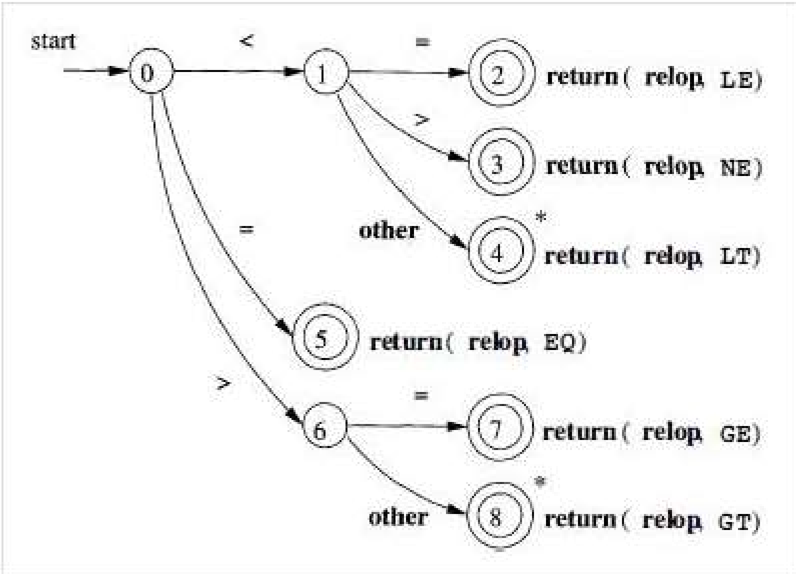
3.3 有限状态自动机

- **有限自动机的五要素**
  - 有限状态集 SS --- 结点
  - 有限输入符号集 Σ
  - 转移函数 δ (s,a) = t
  - 一个开始状态s0
  - 一个终止状态集 TS
  - 输入：字符串
  - 输出：若输入字符串结束，且到达终止状态，则接受，否则拒绝
- **例如：“101” 输出拒绝，“1010”输出接受。**



3.3 有限状态自动机

- 词法分析器在扫描输入串过程中，寻找和某个模式匹配的次数，转换图中每个状态代表一个可能在这个过程中出现的情况
  - relop-> < | > | <= | >= | == | <>
  - 有穷自动机是识别器，
    只能对每个可能的输入串回答：
    是 或 否



3.4 确定有限自动机

3.4 确定的有限状态自动机

- **确定有限自动机DFA是一个五元组 $M = (SS, \Sigma, \delta, S_0, TS)$，**
  - SS ：有限的状态集合 $\{S_0，S_1，S_2，…\}$
  - Σ ：有限的输入字符表
  - δ ：状态转换函数，$SS \times \Sigma \rightarrow SS \cup \{\perp\}$

    δ是单值全映射函数；
  - S_0 ：初始状态， $S_0 \in SS$
  - TS ：终止状态集，$TS \subseteq SS$

3.4 确定的有限状态自动机

- 例1：DFA M=({0,1,2,3,4},{a,b},δ,{0},{3})，其中δ为：

$\delta(0, a) = 1 \quad \delta(0, b) = 4$

$\delta(1, a) = 4 \quad \delta(1, b) = 2$

$\delta(2, a) = 3 \quad \delta(2, b) = 4$

$\delta(3, a) = 3 \quad \delta(3, b) = 3$

$\delta(4, a) = 4 \quad \delta(4, b) = 4$

3.4 确定的有限状态自动机

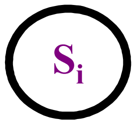
- **DFA的两种表示方式**
  - 状态转换图 ：用有向图表示自动机，比较直观，易于理解；
  - 状态转换矩阵：用二维数组描述自动机，易于程序的自动实现；

3.4 确定的有限状态自动机

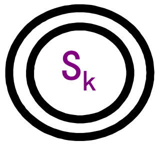
- 状态转换图：用有向图表示自动机

结点：表示状态：

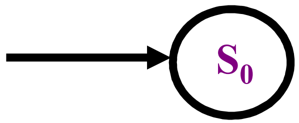
非终止状态：



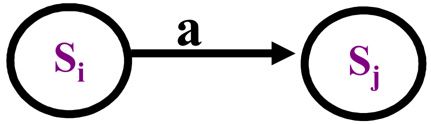
终止状态：



开始状态：



边：表示状态转换函数： $f(S_i,a)=S_j$



3.4 确定的有限状态自动机

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

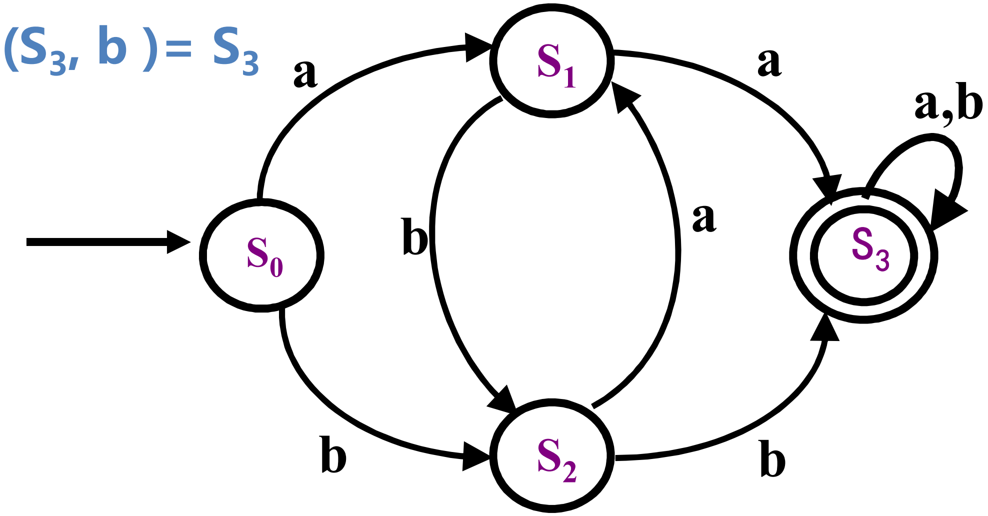
  f (S_0, a)=S_1　　f (S_2, a)=S_1

  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

状态转换图



3.4 确定的有限状态自动机

- **状态转换矩阵：用二维数组描述DFA**
  - 行：表示所有的状态;
    - 初始状态：一般约定，第一行表示开始状态，或在右上角标注“+”；
    - 终止状态：右上角标有“*”或“-”；
  - 列：表示Σ上的所有输入字符；
  - 矩阵元素：表示状态转换函数

3.4 确定的有限状态自动机

状态 \ Σ	a	b
S_0^+	S_1	S_2
S_1	S_3	S_2
S_2	S_1	S_3
S_3^-	S_3	S_3

3.4 确定的有限状态自动机

- 例：DFA M=({0,1,2,3,4},{a,b}, δ,{0},{3})
- 其中δ为：

$\delta(0, a) = 1$　　$\delta(0, b) = 4$

$\delta(1, a) = 4$　　$\delta(1, b) = 2$

$\delta(2, a) = 3$　　$\delta(2, b) = 4$

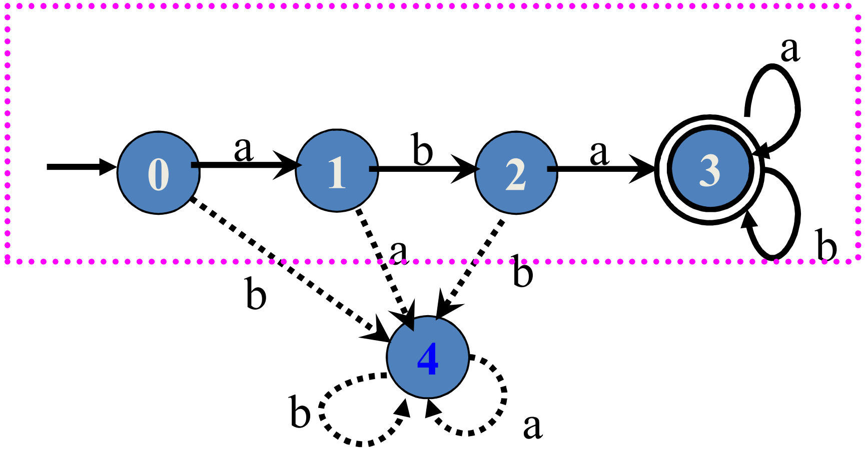
$\delta(3, a) = 3$　　$\delta(3, b) = 3$

$\delta(4, a) = 4$　　$\delta(4, b) = 4$

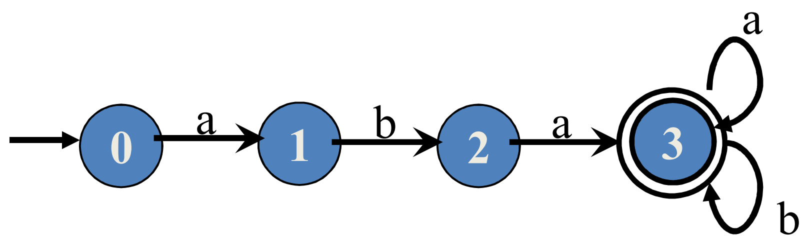
	a	b
0^+	1	4
1	4	2
2	3	4
3^-	3	3
4	4	4

3.4 确定的有限状态自动机

	a	b
0+	1	4
1	4	2
2	3	4
3⁻	3	3
4	4	4



3.4 确定的有限状态自动机



	a	b
0+	1	⊥
1	⊥	2
2	3	⊥
3^-	3	3

↔

	a	b
0+	1	
1		2
2	3	
3^-	3	3

3.4 确定的有限状态自动机

- **DFA例2**

∑: {a, b, c, d}

SS: {S0, S1, S2, S3}

开始状态: **S0**

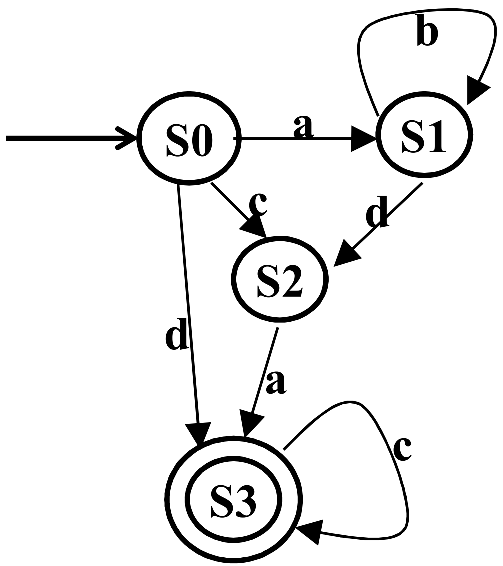
终止状态集: **{S3}**

f: {(S0,a)→ S1, (S0,c)→S2,

(S0,d)→S3, (S1,b)→S1,

(S1,d)→S2, (S2,a)→S3,

(S3, c)→S3}



3.4 确定的有限状态自动机

- ## DFA的确定性
  - 形式定义
    - 初始状态唯一：S0
    - 转换函数是单值函数，即对任一状态和输入符号，唯一地确定了下一个状态
    - 没有输入为ε空边，即不接受没有任何输入就进行状态转换的情况。
  - 转换表上的体现
    - 初始状态唯一：第一行
    - 表元素唯一
  - 转换图上的体现
    - 初始状态唯一:
    - 每个状态最多发出n条边，n是字母表中字母的个数，且发出的任意两条边上标的字母都不同

3.4 确定的有限状态自动机

DFA接受的字符串

- 如果M是一个DFA, $a_1 a_2 \dots a_n$是一个字符串，如果存在一个状态序列 (S_0, S_1, … ,S_n),满足

$$S_0 \xrightarrow{a1} S_1 \;,\; S_1 \xrightarrow{a2} S_2 \;,\; \dots\dots ,\; S_{n-1} \xrightarrow{an} S_n$$

其中 S_0 是开始状态，S_n 是接受状态之一，则串$a_1 a_2 \dots a_n$ 被DFA M接受.

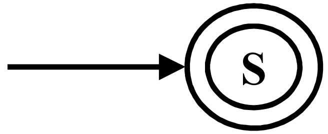
DFA定义的串的集合(DFA接受的语言)

- DFA M接受的所有串的集合，称为M定义的语言，记为 $L(M)$

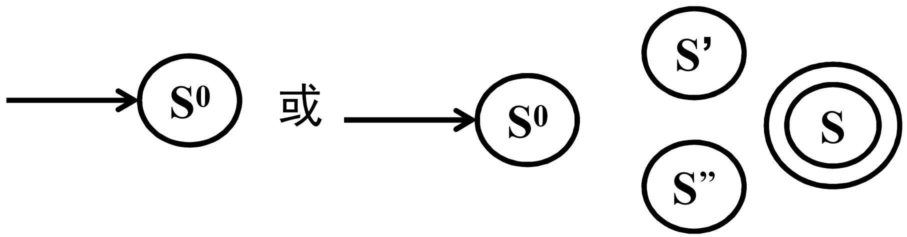
3.4 确定的有限状态自动机

DFA接受的语言

- 若DFA *M*只有一个状态，既是开始状态又是终止状态，则DFA *M*定义的串集是$L(\varepsilon) = \{\varepsilon\}$



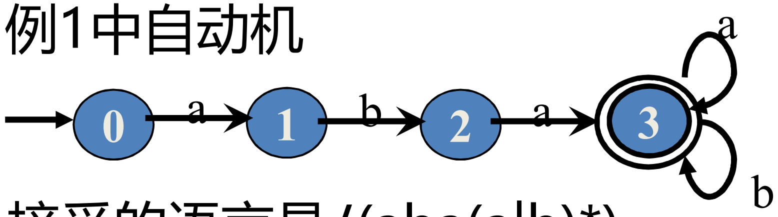
- 若DFA M只有一个状态，并且是开始状态或DFA M有若干个状态，但开始状态到终止状态之间没有通路，则DFA M定义的串集是空集Φ



3.4 确定的有限状态自动机

DFA接受的语言

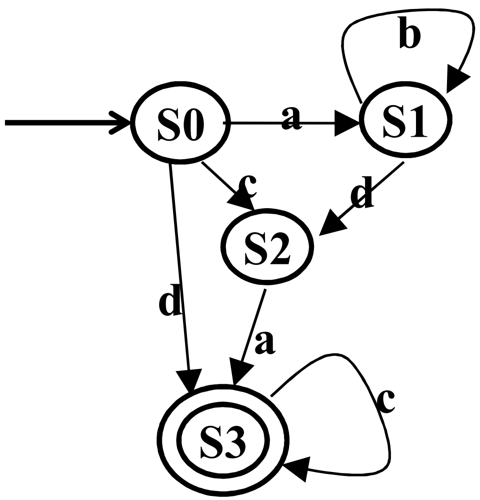
- 例1中自动机



接受的语言是L(aba(a|b)*)

- 例2中自动机

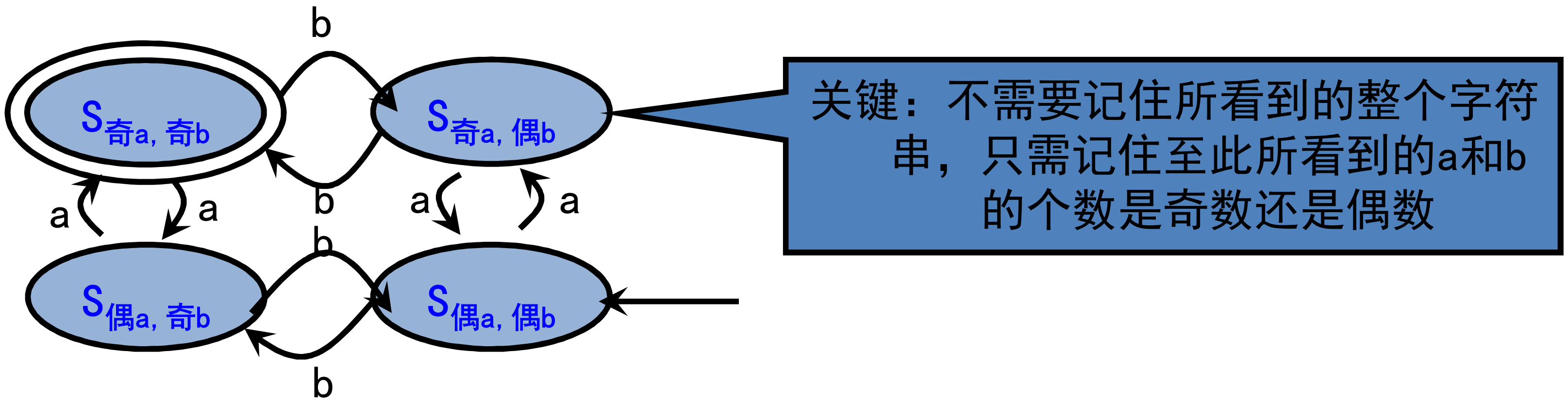
接受的语言是L((ab*da | ca | d)c*)



3.4 确定的有限状态自动机

自动机的设计

- 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：Σ = {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。



3.4 确定的有限状态自动机

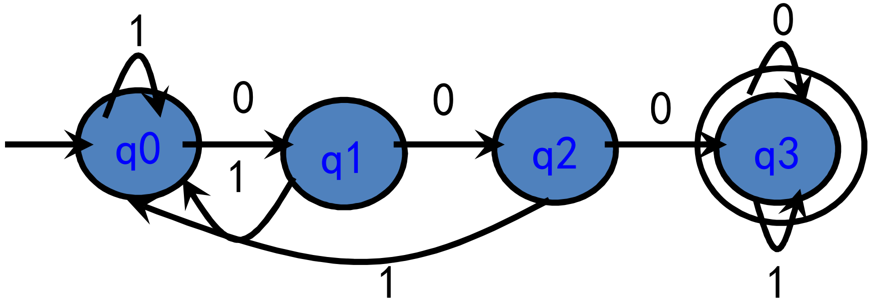
自动机的设计

- 例2：设计有限自动机M，识别{0,1}上的语言

  L = {x000y| x,y∈{0|1}*}　　思考：what if L = {x000x| x∈{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。

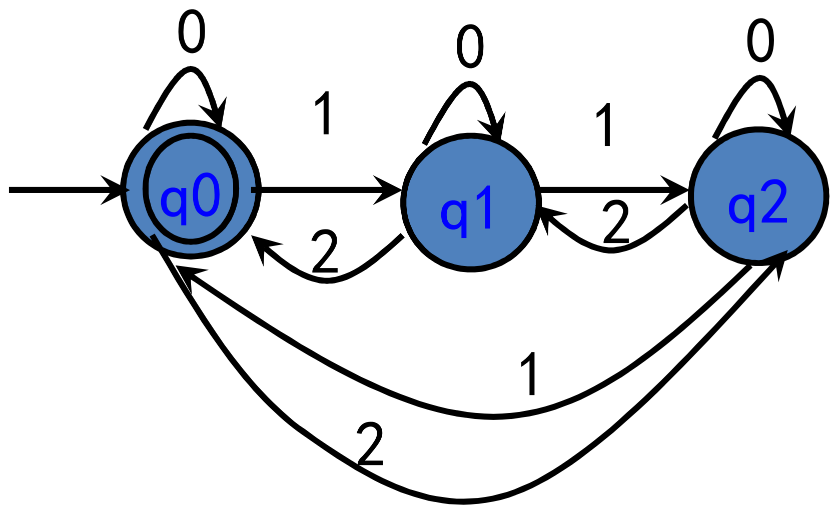


3.4 确定的有限状态自动机

自动机的设计

- 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。



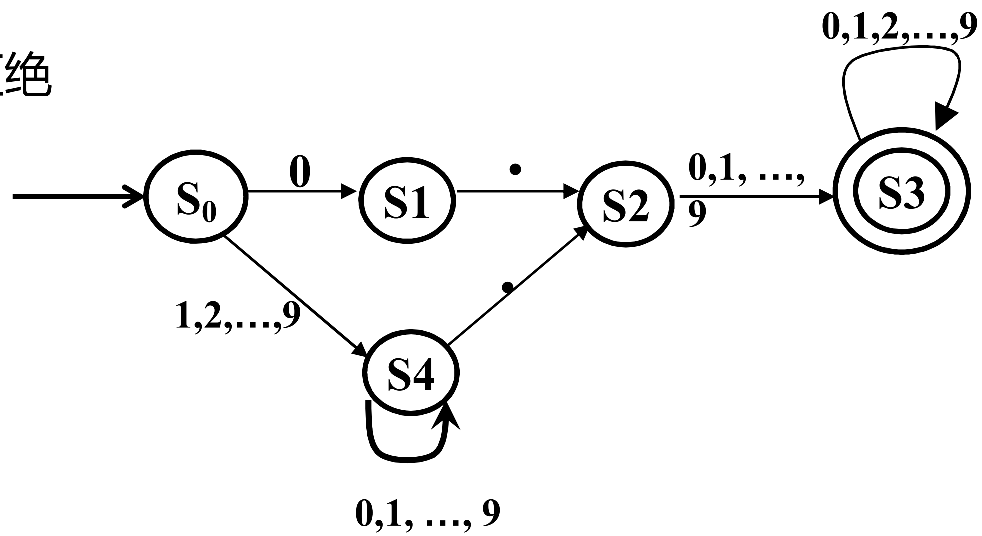
3.4 确定的有限状态自动机

自动机的设计

- 例4：使用DFA定义程序设计语言的无符号实数

0.12, 34.15 接受

00.12, 00., ., 33. 拒绝



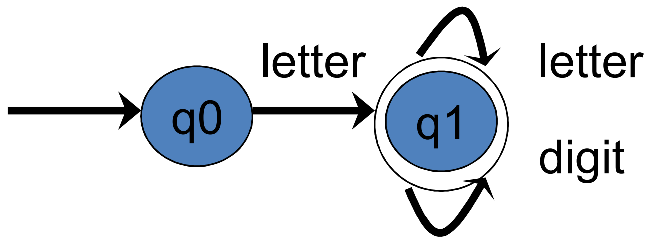
3.4 确定的有限状态自动机

自动机的设计

- 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

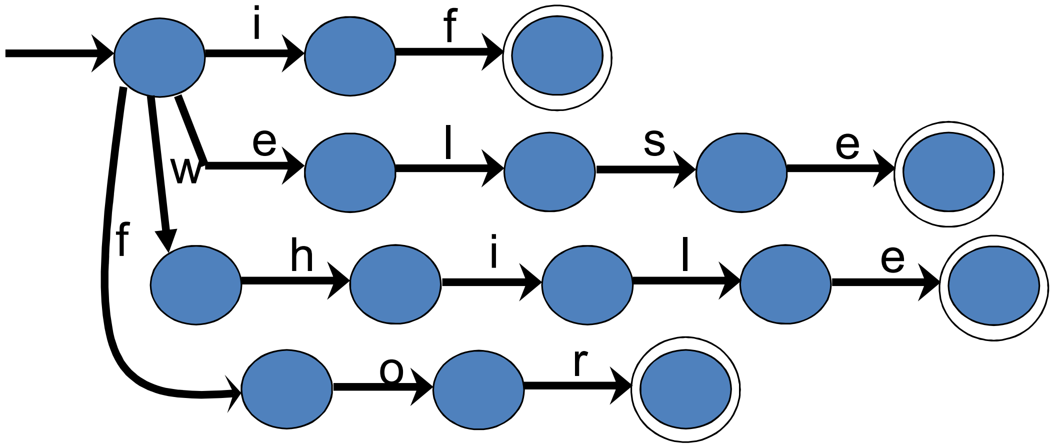


3.4 确定的有限状态自动机

■ 自动机的设计

- 例6：使用DFA定义程序设计语言的保留字

{if, else, while, for}



3.4 确定的有限状态自动机

- **DFA的实现**
  - 目的
    - 给定一个DFA M定义了一个串集
    - 编写一个程序，检查给定的串是否被DFA M所识别或接受
  - 两种途径
    - 基于转换表
    - 基于转换图

3.4 确定的有限状态自动机

- **基于转换表的DFA实现 主要思想**
  - 输入：一个字符串α,以’#’结尾.
  - 输出： 如果接受则输出*true* 否则输出 *false*
  - 数据结构：
    - 转换表 (二维数组 *T*)
  - 两个变量
    - *State*: 记录当前状态;
    - *CurrentChar*: 记录串α中当前正在被读的字符;

3.4 确定的有限状态自动机

■ 算法主要思想

```
1. State = InitState;
2. Read(CurrentChar);
3. while T(State, CurrentChar) <> error && CurrentChar <>  '#'
   do
     begin  State = T(State, CurrentChar);
            Read(CurrentChar);
     end;
 4. if CurrentChar =  '#'  && State∈ FinalStates， return true;
     otherwise, return false.
```

3.4 确定的有限状态自动机

	a	b	c	d
S0+	S1	⊥	S2	S3
S1	⊥	S1	⊥	S2
S2	S3	⊥	⊥	⊥
S3-	⊥	⊥	S3	⊥

- $S0 \rightarrow_{(c)} S2 \rightarrow_{(a)} S3 \rightarrow_{(b)} \perp$

1）cab 接受 or 拒绝？

拒绝

- $S0 \rightarrow_{(a)} S1 \rightarrow_{(b)} S1 \rightarrow_{(d)} S2 \rightarrow_{(a)} S3 \rightarrow_{(c)} S3 \rightarrow_{(c)} S3$

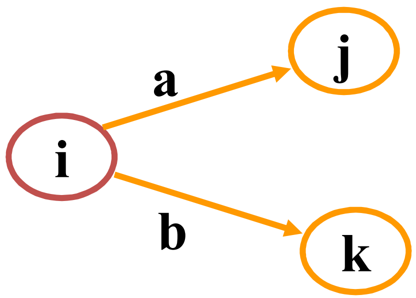
2）abdacc 接受 or 拒绝？

接受

3.4 确定的有限状态自动机

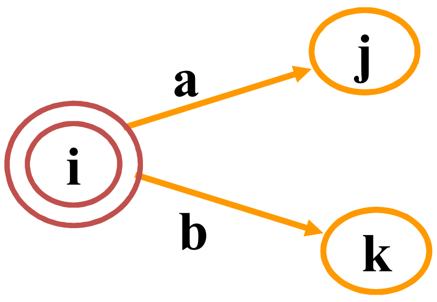
基于转换图的DFA实现

- 每个状态对应一个带标号的*case* 语句
- 每条边对应一个 *goto* 语句
- 对于每个终止状态，增加一个分支，如果当前字符是字符串的结束符#，则接受;



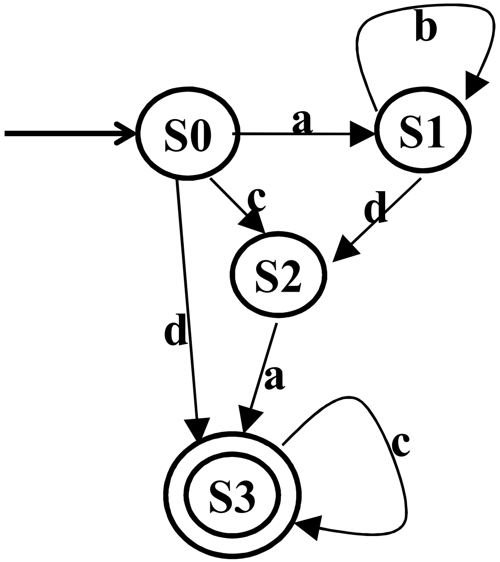

```
Li: case CurrentChar of
        a      :  goto Lj
        b      :  goto Lk
        other  :  Error( )
```

3.4 确定的有限状态自动机




```
Li: case CurrentChar of
        a     :  goto Lj
        b     :  goto Lk
        #     :  return true;
        other :  return false;
```

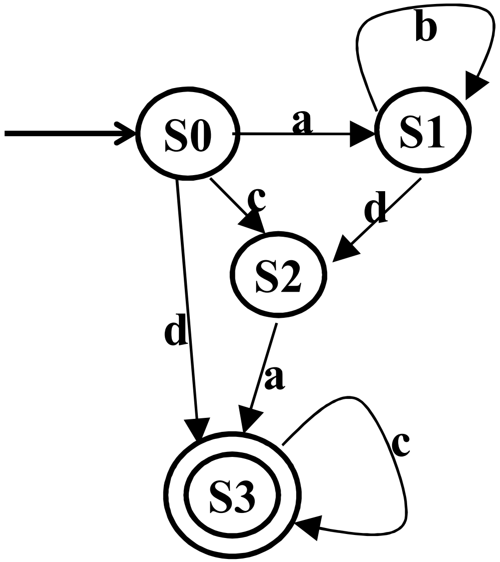
3.4 确定的有限状态自动机




```
LS0: Read(CurrentChar);
     switch (CurrentChar) {
      case  a : goto LS1;
      case  c : goto LS2:
      case  d : goto LS3;
      other   : return false; }
```

```
LS1: Read(CurrentChar);
     switch (CurrentChar) {
     case  b : goto LS1;
     case  d : goto LS2:
     other   : return false; }
```

3.4 确定的有限状态自动机




```
LS2: Read(CurrentChar);
        switch (CurrentChar) {
          case  a : goto LS3;
            other : return false;   }

LS3: Read(CurrentChar);
        switch (CurrentChar) {
               case  c : goto LS3:
            case  # : return true;
              other : return false;   }
```

3.4 确定的有限状态自动机

- **比较**
  - 转换表方式：
    - 是通用的算法，不同的语言，只需改变输入的转换表，识别程序不需改变
  - 转换图方式：
    - 不需要存储转换表(通常转换表是很大的)，但当语言改变即自动机的结构改变时，整个识别程序都需要改变。

3.5 非确定有限自动机

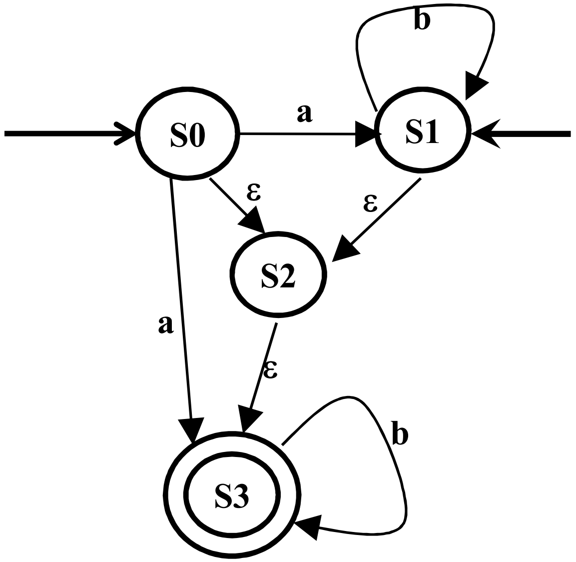
3.5 非确定的有限状态自动机

- **确定有限自动机DFA是五元组(SS,∑,δ,s0,TS)**
  - 确定性体现：初始状态唯一、转换函数是单值函数
- **非确定有限自动机NFA也是五元组(SS,∑,δ,S0,TS)**
  SS={S0,S1,…Sn}是状态集
  - ∑是字母表
  - $S0 \subseteq SS$是初始状态集,不能为空
  - δ是转换函数，但不要求是单值的
  - $\delta : SS \times (\Sigma \cup \varepsilon) \to 2^{SS}$
  - $TS \subseteq SS$是终止状态集

3.5 非确定的有限状态自动机

- $\sum = \{a, b\}$ ，SS: {S0, S1, S2, S3}
  - 初始状态集: {S0 , S1}
  - 终止状态集: {S3}
  - δ : {(S0,a)→ {S1, S3},(S0,ε)→ {S2},
    (S1,b)→{S1}, (S1, ε)→{S2},
    (S2, ε)→{S3},
    (S3, b)→{S3}}
- **NFA也可以用状态转换图或状态转换矩阵表示**

3.5 非确定的有限状态自动机



	a	b	ε
$S0^+$	{S1,S3}		{S2}
$S1^+$		{S1}	{S2}
S2			{S3}
$S3^-$		{S3}	

状态集合

3.5 非确定的有限状态自动机

NFA接受的字符串

- 如果M是一个NFA, ***a_1 a_2... an*** 是一个字符串，如果存在一个状态序列(S0, S1 , ... ,Sn),满足

  $$S0 \xrightarrow{a1} S1 \quad , S1 \xrightarrow{a2} S2 \quad , \ldots\ldots , \quad Sn\text{-}1 \xrightarrow{an} Sn$$

  其中 S0 是开始状态之一，Sn 是接受状态之一，则串a_1 a_2... an 被NFA M接受.

> 转换路径中的ε将被忽略，因为空串不会影响构建得到的字符串
> NFA所能接受的串与DFA是相同的，但NFA实现起来很困难

NFA定义的串的集合(NFA接受的语言)

- NFA M接受的所有串的集合，称为M定义的语言，记为 ***L(M)***

3.6 NFA v.s. DFA

3.6 NFA v.s. DFA

	DFA	NFA
初始状态	一个初始状态	初始状态集合
ε边	不允许	允许
$\delta(S, a)$	S’ or ⊥	{S1, …, Sn} or ⊥
实现	容易	有不确定性

- 对于确定的输入串，DFA只有一条路径接受它
- NFA则可能需要在多条路径中进行选择！

3.6 NFA v.s. DFA

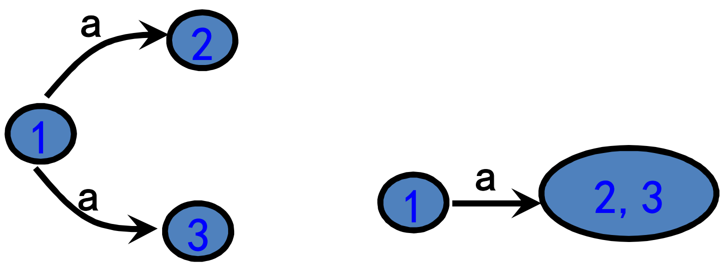
- **由NFA模拟RE，还是由DFA模拟RE？**
  - 延伸阅读：由NFA模拟---Section 3.7.2&3.7.3(教材P99)
  - 思考：Pros and cons：how about the efficiency?
- **Solution: NFA和DFA都识别RE，NFA可转换成DFA。**

3.6 NFA v.s. DFA

■ 由NFA构造DFA

- 对任意NFA, 都存在一个DFA与之等价, 转换的思想-消除不确定性
- 合并初始状态集成一个状态
- 消除ε边

- 消除多重定义的边。



3.6 NFA v.s. DFA

- **由NFA构造DFA (子集法) --- 基本思路**
  - 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

3.6 NFA v.s. DFA

- **由NFA构造DFA (子集法)**
  - 一些基本操作

OPERATION	DESCRIPTION
ϵ-$closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-$closure(T)$	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\, \epsilon\text{-}closure(s)$.
$move(T, a)$	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

3.6 NFA v.s. DFA

由NFA构造DFA (子集法)

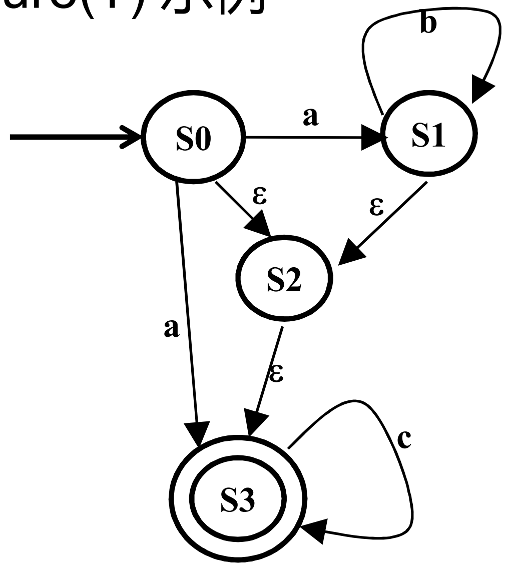
- ε-closure(T): 对于给定的 NFA *A*, 和它的一个状态集合 T, T的空闭包计算如下：

第一步：令ε-closure(T) = T;
第二步：如果在状态集T中存在状态s,
s到状态s'存在一条ε边，
并且$s' \notin$ε-closure(T),
则将s'加入T的空闭包ε-closure(T);
重复第二步,直到再没有状态可加入ε-closure(T)．

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ε )
                if ( u is not in ε-closure(T) ) {
                        add u to ε-closure(T);
                        push u onto stack;
                }
}
```

3.6 NFA v.s. DFA

- 由NFA构造DFA (子集法)
  - ε-closure(T) 示例



ε-closure({S0, S1}) =

{S0, S1 }

{S0, S1, S2}

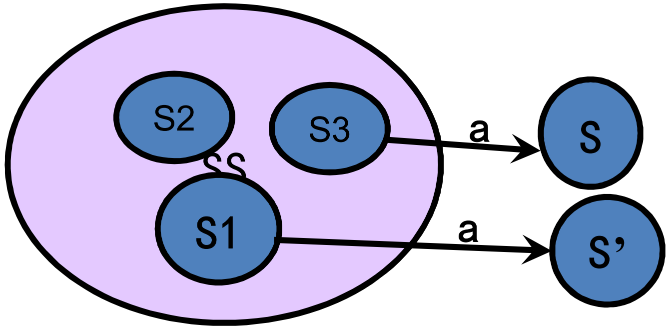
{S0, S1, S2}

{S0, S1, S2,S3}

3.6 NFA v.s. DFA

由NFA构造DFA (子集法)

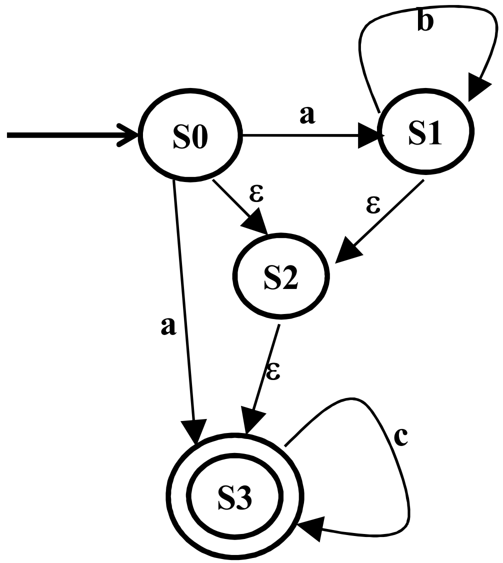
- 对于NFA *N*中的给定状态集合T和符号 *a*, Move(T, a) = {s | 对于状态集T中的一个状态s1, 如果A中存在一条从s1到s的a转换边}



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

3.6 NFA v.s. DFA

- **由NFA构造DFA (子集法)**
  - Move(T, a) 示例



Move({S0, S1}, a) = {S1, S3}

Move({S0, S1}, b) = {S1}

3.6 NFA v.s. DFA

▪ 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。

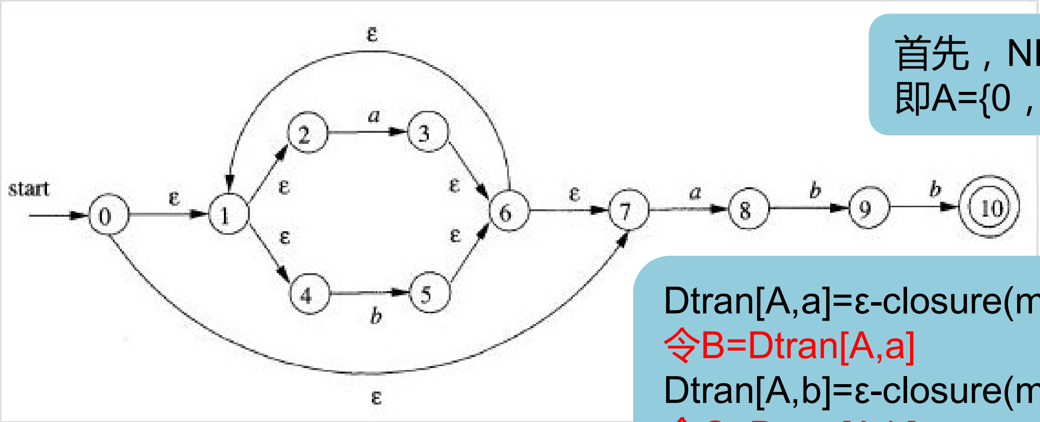
```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U 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ε-closure(s0)中的任何状态上，其中s0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ε转换，因此N在读入a之后可位于ε-closure(move(T,a))中的任何状态上。

3.6 NFA v.s. DFA

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



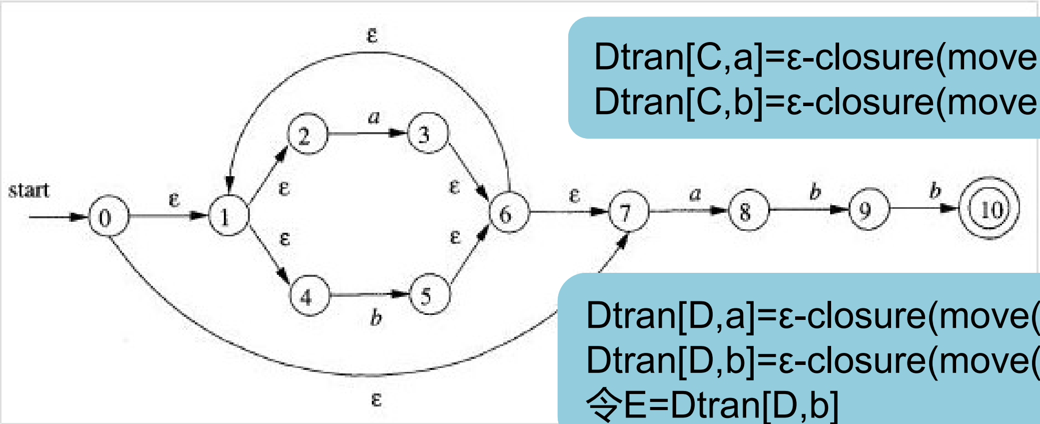
首先，NFA的开始状态A是ε-closure(0)，
即A={0，1，2，4，7}，NFA的输入字母表是{a,b}

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

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

3.6 NFA v.s. DFA

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



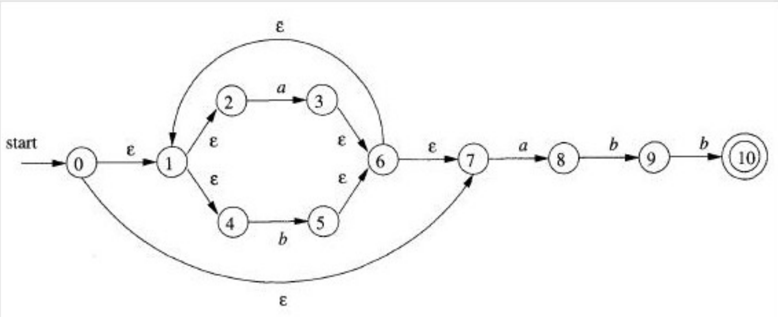
Dtran[C,a]=ε-closure(move(C,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[C,b]=ε-closure(move(C,b))=ε-closure({5})={1,2,4,6,7}=C

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

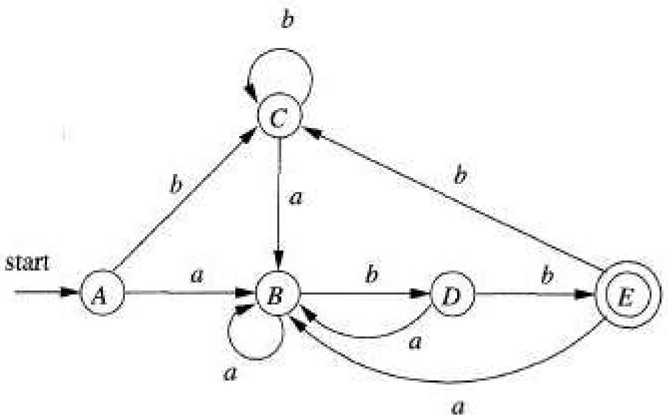
Dtran[E,a]=ε-closure(move(E,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[E,b]=ε-closure(move(E,b))=ε-closure({5})={1,2,4,6,7}=C

3.6 NFA v.s. DFA

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



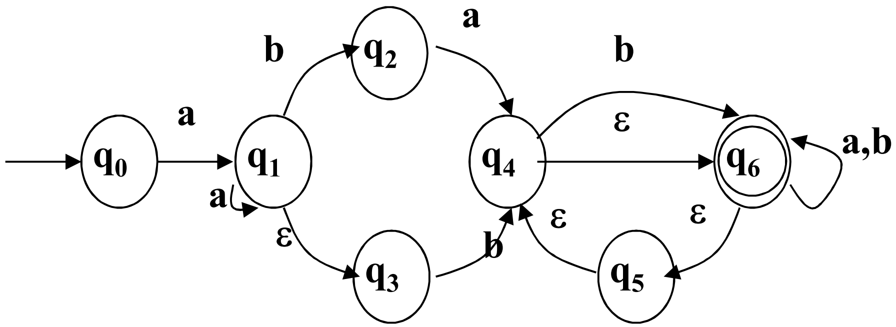
NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C



3.6 NFA v.s. DFA

- **由NFA构造DFA (子集法)** 例2:

	a	b
{q0}+	{q1,q3}	⊥
{q1,q3}	{q1,q3}	{q2,q4,q6,q5}
{q2,q4,q6,q5}-	{q4,q6,q5}	{q6,q5,q4}
{q4,q6,q5}-	{q6,q5,q4}	{q6,q5,q4}



作 业

- 教材P78：3.3.2， 3.3.5
- 教材P86：3.4.1，3.4.2
- 教材P96：3.6.3，3.6.4，3.6.5
- 教材P105：3.7.1

Thank you!