
Lecture 2: Lexical Analysis

Xiaoyuan Xie 谢晓园

xxie@whu.edu.cn

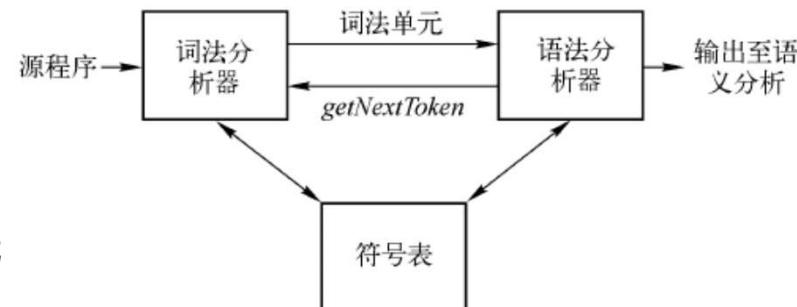
计算机学院E301



2.1 回顾

2.1 回顾

- 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



2.1 回顾

■ 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;

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

2.1 回顾

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

	单词类型	种别
1	关键字	program、if、else、then、...
2	标识符	变量名、数组名、记录名、过程名、...
3	常量	整型、浮点型、字符型、布尔型、...
4	运算符	算术 (+ - * / ++ --) 关系 (> < == != >= <=) 逻辑 (& ~)
5	界限符	; () = { } ...

2.1 回顾

■ 词法分析流程

- 输入：字符串 (ASCII)

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

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

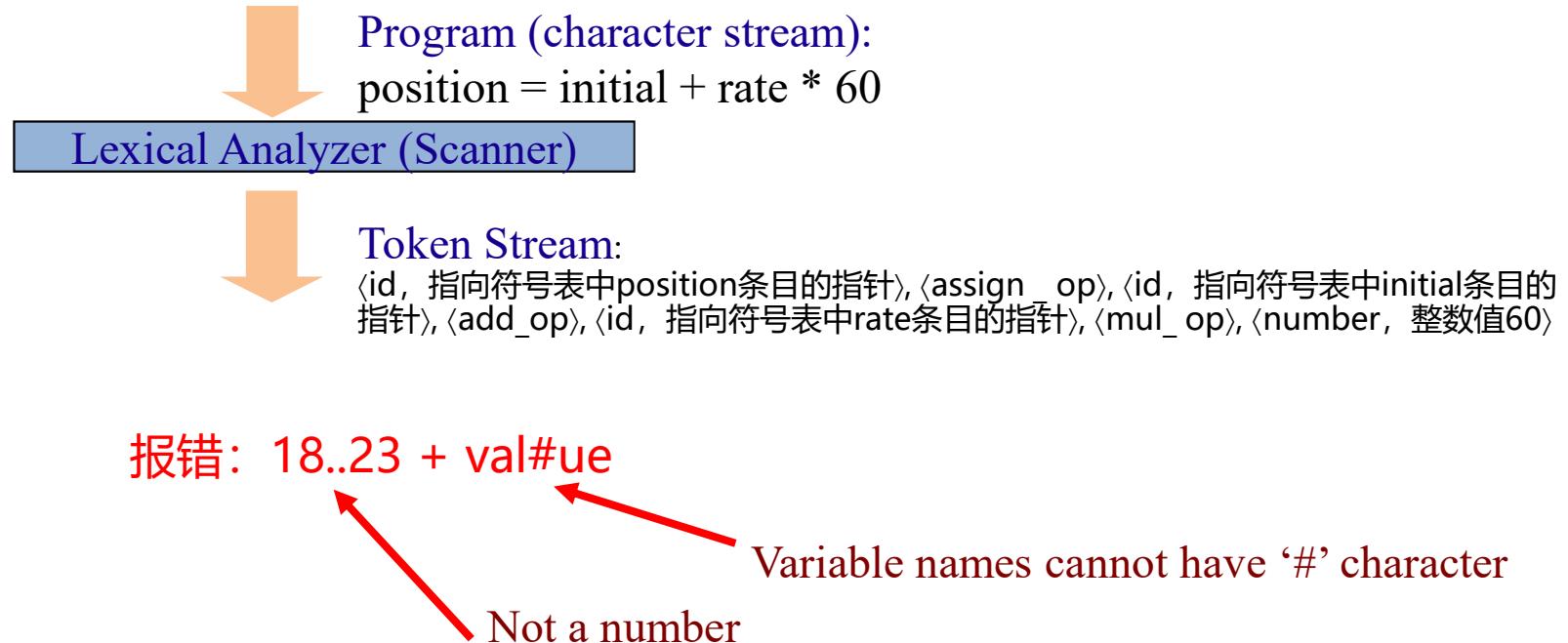
- Tokenize

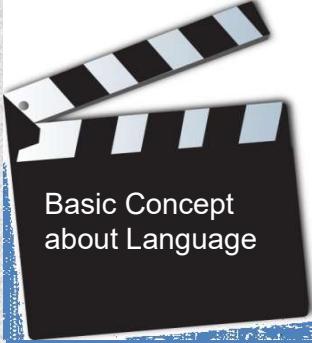
\tif (x==j)\n\t\z=0;\n\telse\n\t\z=1;



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

2.1 回顾





2.2 语言的定义

2.2 语言的定义

■ 什么是语言?

- 作用：沟通，交流，传递信息
- 自然语言：英语，中文，日语.....

■ 你如何定义语言?

- 无限的集合
- 抽象地来看：字符组成了单词；单词组成了句子；句子携带了信息

2.2 语言的定义

- 语言——形式化的内容提取
 - 语言 (Language): 满足一定条件的句子集合
 - 句子 (Sentence): 满足一定规则的单词序列
 - 单词 (Token): 满足一定规则的字符 (Character) 串
- 语言是字和组合字的规则
 - “doge” (it's not a word): reserved words and some rules for words (词法)
 - “Long time no see” (It's not a correct sentence): rules for sentences (语法)



2.2 语言的定义

- 程序设计语言——形式化的内容提取
 - 程序设计语言(Programming Language)：组成程序的所有语句的集合。
 - 程序(Program)：满足语法规则的语句序列。
 - 语句(Sentence)：满足语法规则的单词序列。
 - 单词(Token)：满足词法规则的字符串。
- 例：
 - 变量:=表达式
 - if 条件 then 语句
 - while 条件 do 语句
 - call 过程名(参数表)

2.2 语言的定义

- 语言是一个无限集合
 - How can **finite recipes** generate enough infinite sets of sentences?
 - If a sentence is just a sequence and has no structure and if the meaning of a sentence derives, among other things, from its structure, how can we **assess the meaning** of a sentence?
- 计算机视角-抽象，再抽象
- Grammars (文法) - a Generating Device
 - 有限的规则来描述的无限集

2.2 语言的定义

- 描述形式——文法
 - 语法——语句
 - 语句的组成规则
 - 描述方法: BNF范式、产生式
 - 词法——单词
 - 单词的组成规则
 - 描述方法: BNF范式、产生式、正规式

2.2 语言的定义

- 符号(Symbol/Character): 语言中不可再分的单位
- 字母表: 符号的非空有穷集合
 - Σ , V 或其它大写字母, 其中的元素可称为字母、符号、字符
 - 例如: $V_1 = \{a, b, c\}$, $V_2 = \{+, -, 0, 1, \dots, 9\}$, $\Sigma = \{x|x \in \text{ASCII字符}\}$
- 符号串(字符串): 某字母表上的符号的有穷序列
 - a, b, c, abc, bc, \dots : V_1 上的符号串; $1250, +2, -1835, \dots$: V_2 上的符号串
 - 空串 (ϵ): 不含任何符号的串

2.2 语言的定义

- 语句: 字母表上符合某种构成规则的符号串序列
 - He is a good student. Peanut eats monkey.
 - for(int i = 0; i<10; i++) {call_func(i);}
- 语言 L: 某字母表上的语句的集合

2.2 语言的定义

- 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 $L(\Sigma)$ ---注意：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： $\Sigma =$ 英文字母, $L(\Sigma)$ 是英文句子
- 例2： $\Sigma =$ ASCII, $L(\Sigma)$ 是C语言程序
- 例3：字母表{0, 1}上的语言

{0, 1} {00, 11} {0, 1, 00, 11} {0, 1, 00, 11, 01, 10}



2.3 正则文法

■ 符号串连接:

- x 和 y 的连接 xy 是把 y 的所有符号顺序地接在 x 的符号之后所得到的符号串

■ 符号串方幂:

- 设 x 是字母表 Σ 上的符号串，把 x 自身连接 n 次得到的符号串 z , 即 $z = xx\dots xx$ (n 个 x), 称作符号串 x 的 n 次幂，记作 $z = x^n$

■ 符号串前缀后缀:

- 设 x 、 y 、 z 是某一字母表上的符号串, $x = yz$, 则 y 是 x 的前缀, z 是 x 的后缀; $z \neq \epsilon$ 时 y 是 x 的真前缀, $y \neq \epsilon$ 时 z 是 x 的真后缀

2.3 正则文法

■ 符号串子串：

- 非空字符串 x , 删去它的一个前缀和一个后缀后所得到的字符串称为 x 的子字符串, 简称子串。
如果删去的前缀和后缀不同时为 ϵ , 则称该子串为真子串

■ 符号串子序列

- 从 s 中删去零或多于零个符号(这些符号不要求连续)

■ 符号串逆转（用SR表示）：

- 将 S 中的符号按相反次序写出而得到的字符串

■ 符号串长度：

- 是该字符串中的符号的数目。例如 $|aab|=3, |\epsilon|=0$ 。

2.3 正则文法

■ 符号串集合(语言)的积

- 设串集 $L=\{\alpha_1, \alpha_2, \dots\}$, $M=\{\beta_1, \beta_2, \dots\}$, 二者的笛卡尔积 $LM=\{\alpha\beta \mid \alpha \in L, \beta \in M\}$
- E.g. $L=\{ab, abb\}$, $M=\{ced, cd\}$, 那么 $LM = \{abcd, abcd, abced, abbcd\}$

■ 字符串集合(语言)的方幂

- $L^0=\{\epsilon\}$, $L^1=L$, $L^n=LL^{n-1}$
- 若 $|L|=m$, 那么, $|L^0|=1$, $|L^1|=m$, $|L^n|=m^n$

■ 字符串集合(语言)的Kleene闭包

- $L^*=L^0 \cup L^1 \cup L^2 \cup \dots$

语言就是其字母表上闭包的子集

■ 字符串集合(语言)的正闭包

- $L^+=L^1 \cup L^2 \cup \dots = L^* - \{\epsilon\}$

2.3 正则文法

例 3.3 令 L 表示字母的集合 $\{A, B, \dots, Z, a, b, \dots, z\}$, 令 D 表示数位的集合 $\{0, 1, \dots, 9\}$ 。我们可以用两种不同但等价的方式来考虑 L 和 D 。一种方法是将 L 看成是大、小写字母组成的字母表, 将 D 看成是 10 个数位组成的字母表。另一种方法是将 L 和 D 看作语言, 它们的所有串的长度都为一。下面是一些根据图 3-6 中的运算符从 L 和 D 构造得到的新语言:

- 1) $L \cup D$ 是字母和数位的集合——严格地讲, 这个语言包含 62 个长度为 1 的串, 每个串是一个字母或一个数位。
- 2) LD 是包含 520 个长度为 2 的串的集合, 每个串都是一个字母跟一个数位。
- 3) L^4 是所有由四个字母构成的串的集合。
- 4) L^* 是所有由字母构成的串的集合, 包括空串 ϵ 。
- 5) $L(L \cup D)^*$ 是所有以字母开头的, 由字母和数位组成的串的集合。
- 6) D^+ 是由一个或多个数位构成的串的集合。

2.3 正则文法

■例：

- $\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots\}$
- $\{a, b, c, d\}^+ = \{a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, \dots, aaa, aab, aac, aad, aba, abb, abc\dots\}$
- $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots\}$
- $\{a, b, c, d\}^* = \{\varepsilon, a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, \dots, aaa, aab, aac, aad, aba, abb, abc, \dots\}$

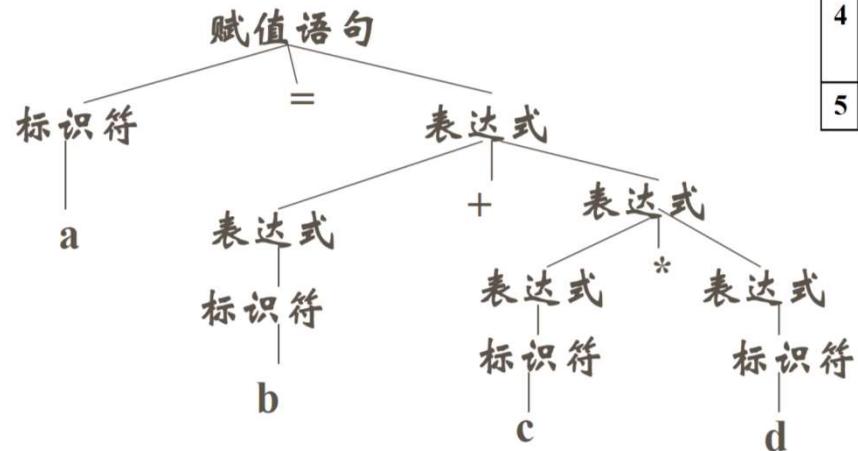
2.2 语言的定义

- $L(\Sigma)$ 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？

语言 $L(\Sigma)$ 就是其字母表（把字母表看作是由 $|\Sigma|$ 个长度为1的字符串组成的语言）上闭包的子集

2.2 语言的定义

■ 程序设计语言文法示例



	单词类型	种别
1	关键字	program、if、else、then、...
2	标识符	变量名、数组名、记录名、过程名、...
3	常量	整型、浮点型、字符型、布尔型、...
4	运算符	算术 (+ - * / ++ --) 关系 (> < == != >= <=) 逻辑 (& ~)
5	界限符	; () = { } ...

2.2 语言的定义

■ 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;
- 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;
- 设 V 是文法 G 的符号集, 则有: $V = V_N \cup V_T$, $V_N \cap V_T = \emptyset$
- 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。

词法、语法都是这样定义的

2.2 语言的定义

- 文法规则的递归定义
 - 非终结符的定义中包含了非终结符自身
 - 设 $\Sigma = \{0, 1\}$; <整数> \rightarrow <数字><整数>|<数字>; <数字> \rightarrow 0 | 1
- 使用递归定义时要谨慎，要有递归出口，否则可能永远产生不出句子

2.2 语言的定义

■ 自然语言文法示例

■ 产生式

<句子> → <主语><谓语><宾语>
<主语> → <形容词><名词>
<谓语> → <动词>
<宾语> → <形容词><名词>
<形容词> → young | pop
<名词> → men | music
<动词> → like

非终结符

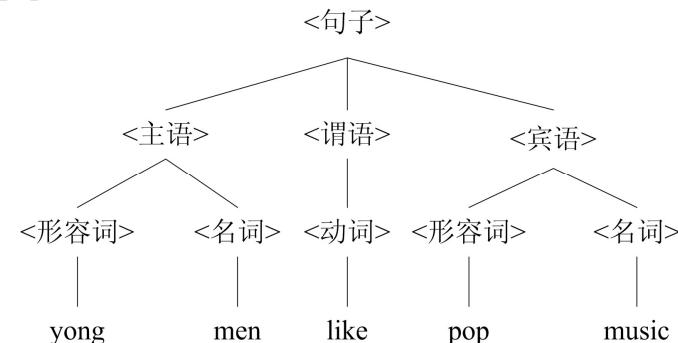
终结符

还能推导出什么句子?

<句子> → <主语><谓语><宾语>
→ <形容词><名词><谓语><宾语>
→ young<名词><谓语><宾语>
→ young men <谓语><宾语>
→ young men <动词><宾语>
→ young men like<宾语>
→ young men like <形容词><名词>
→ young men like pop<名词>
→ young men like pop music

最左推导

最右规约



2.2 语言的定义

■ 句型

- 从文法开始符号S开始，每步推导（包括0步推导）所得到的字符串 $\alpha: S \rightarrow \alpha$, 其中 $\alpha \in (V_N \cup V_T)^*$

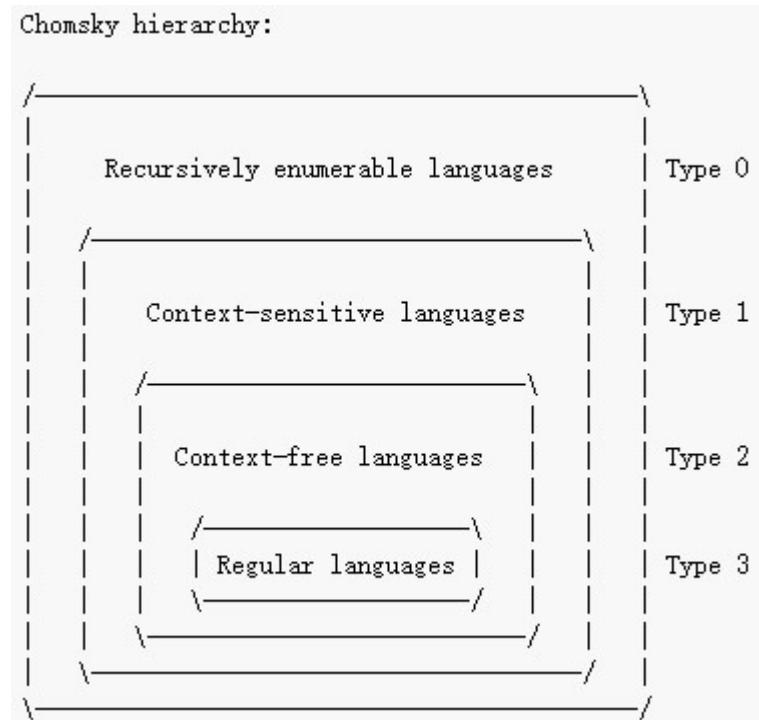
■ 句子

- 仅含终结符的句型

■ 语言

- 由S推导所得的句子的集合 $L(G) = \{\alpha | S \rightarrow \alpha, \text{ 且 } \alpha \in V_T^*\}$, G为文法

2.2 语言的定义



2.2 语言的定义

■ 扩充的BNF表示

- () —— 提因子: $U \rightarrow ax \mid ay \mid az$ 改写为 $U \rightarrow a(x \mid y \mid z)$
- {} —— 重复次数的指定: <标识符> \rightarrow <字母>{<字母>|<数字>}₀⁵
- [] —— 任选符号: <整数> \rightarrow [+|-]<数字>{<数字>}

2.2 语言的定义

■ 也可以用字符串定义语言

例：设文法 $G_2 = (\{S\}, \{a, b\}, P, S)$, 其中P为：

$$(0) S \rightarrow aSb$$

$$(1) S \rightarrow ab$$

等价于 $L(G_2) = \{a^n b^n | n \geq 1\}$



2.3 正则文法

2.3 正则文法

■ Chomsky 3型文法:正规 (正则) 文法

- P中产生式具有形式 $A \rightarrow \alpha B$, $A \rightarrow \alpha$ (右线性), 或者 $A \rightarrow B\alpha$, $A \rightarrow \alpha$ (左线性), 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。

A language to define lexical structure

Describes how to generate tokens of a particular type

2.3 正则文法

- 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) $S \rightarrow b$ $L(G_1) = \{a^i(a \mid b) \mid i \geq 0\}$
- 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*)。

2.3 正则文法

■ 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 \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

2.3 正则文法

■ 定义正则表达式(Σ 为字母表)

- 原子正则表达式(atomic regular expressions)
 - ϵ 和 \emptyset 是 Σ 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 Σ 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 Σ 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 Σ 上的正则表达式，表示的正则集 $L((r))= L(r)$
 - $r|s$ 也是 Σ 上的正则表达式，表示的正则集 $L(r|s)= L(r) \cup L(s)$
 - $r \cdot s$ 也是 Σ 上的正则表达式，表示的正则集 $L(r \cdot s)= L(r)L(s)$
 - r^* 也是 Σ 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 Σ 上的正则表达式，表示的字符串集合称为 Σ 上的正则集或正规集。

2.3 正则文法

- 实际应用中会扩充很多正则表达式的运算，如：

- r^+ 也是 Σ 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 $a|b^*c$

a regular expression == a set of strings (即，语言)

RE的作用和产生式等价

不同语言的正则表达式语法规定

<http://www.greenend.org.uk/rjk/tech/regexp.html>

2.3 正则文法

■ 正规式的例子，例1： $\Sigma = \{a, b\}$

- $a \mid b, ab, a^*, b^*, ab^*, a(a|b)^*$ 也是 Σ 上的正则表达式
- $L(a|b) = \{a, b\}$
- $L(ab) = \{ab\}$
- $L(a^*) = \{\varepsilon, a, aa, aaa, \dots\}$
- $L(b^*) = \{\varepsilon, b, bb, bbb, \dots\}$
- $L(aa \mid ab \mid ba \mid bb) = \{aa, ab, ba, bb\}$
- $L((a \mid b)(a \mid b)) = \{aa, ab, ba, bb\}$
- $L((a \mid b)^*) =$ 由a和b构成的所有串集
- $L(ab^*) = \{a, ab, abb, \dots\}$
- $L(a(a|b)^*) = \{a, aa, ab, aaa, aab, aba, abb, aaaa, aaab, \dots\}$

2.3 正则文法

■ 例2：

- $(00 \mid 11 \mid ((01 \mid 10)(00 \mid 11)^*(01 \mid 10)))^*$
- 句子: 01001101000010000010111001

2.3 正则文法

- 例3：程序设计语言的单词 $\Sigma = \text{ASCII}$,
- 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⁺

2.3 正则文法

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

2.3 正则文法

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

2.3 正则文法

■ 例6：程序设计语言的单词 $\Sigma = \text{ASCII}$, 运算符: +, -, *, /

- ADD: +
- SUB: -
- MUL: *
- DIV: /
- Operator = ADD | SUB | MUL | DIV

2.3 正则文法

■ 例7：程序设计语言的单词 $\Sigma = \text{ASCII}$, 空白：空格，换行，Tab

- ' '
- \n
- \t
- Whitespace = (' ' | \n | \t) $^+$
- Words = Integer | Keywords | Identifier | Operator | Whitespace

2.3 正则文法

- 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@whu.edu.cn
 - $\Sigma = \{\text{字母, 数字, @, .}\}$
 - Names = (Letter | Digit) $^+$
 - Address = Names@(Names)+Names

PS: 在使用regular识别工具时，.需要转义。另外，可以有一些简化写法，例如使用^和\$识别开头结尾

2.3 正则文法

■ 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
- $A \varepsilon = \varepsilon A = A$ ε 是连接的恒等元素

2.3 正则文法

- 练习：设字母表 $\Sigma = \{0, 1\}$, 试写正则表达式
 - 所有 Σ 上定义的串
 - $(0|1)^*$
 - 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
 - $0|1(0|1)^*$
 - 能被2整除的二进制数
 - 特点：以0结尾
 - $0|1(0|1)^*0$

2.3 正则文法

■ 练习：为自然语言构造RE

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

2.3 正则文法

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

2.3 正则文法

■ 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
- $c^*a(a|c)^*b(a|b|c)^* \mid c^*b(b|c)^* a(a|b|c)^*$
 - 由a,b,c组成，至少包含一个a和一个b的串

2.3 正则文法

- 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 $A = \{a^n b a^n \mid n > 0\}$, $S \rightarrow aBa$ $B \rightarrow aBa|b$
 - 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) $AE + AE \in AE$
 - FA的无辅助栈, 用来存储状态

2.3 正则文法

■ 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:

$$(0) S \rightarrow aS$$

$$(1) S \rightarrow a$$

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

■ 所以，上述例子中RE对应的产生式都是什么？

2.3 正则文法

■ 词法分析：使用RE/RL



valu#e = initial + rate * 60

RE <==> 正规文法 (三型) L(RE)表示

正规文法的能力足以描述词法规定

作业

- 教材P78: 3.3.2, 3.3.5 (1, 2, 3, 5, 6, 8, 9)



2.4 有限状态自动机

2.4 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。

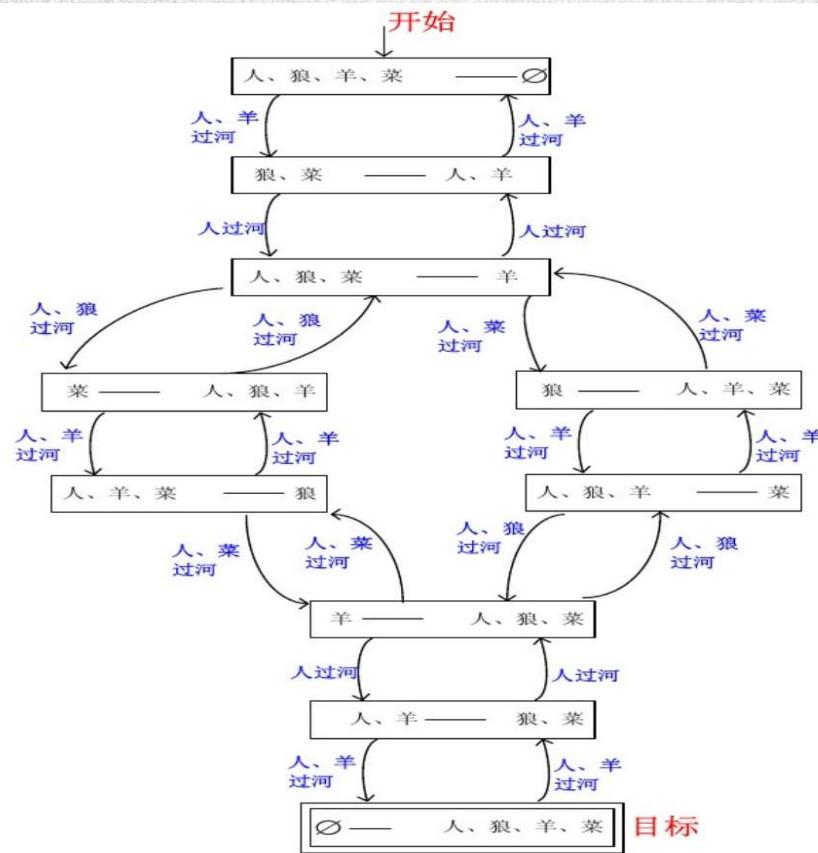
2.4 有限状态自动机

■ 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？

2.4 有限状态自动机

■ 过河问题

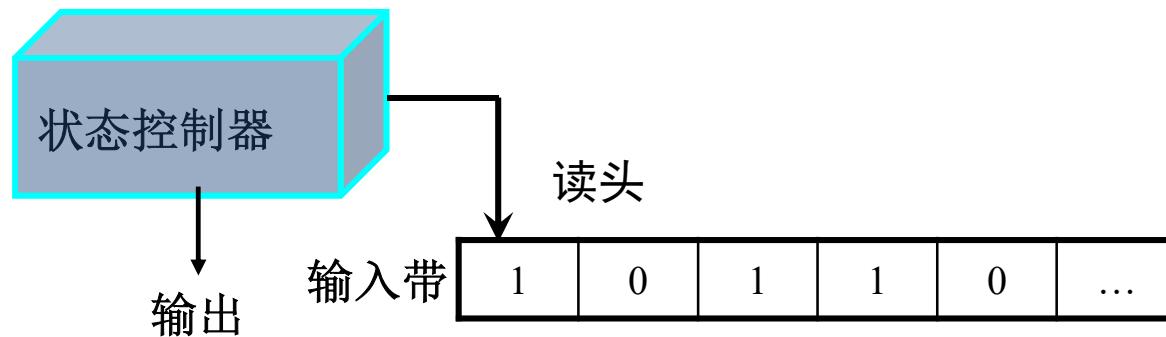


- 从这个图中找出从初始状态到目标状态的一条有向路，就是这个问题的一个解。

2.4 有限状态自动机

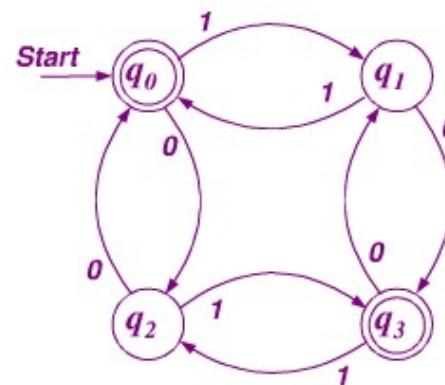
■ 有限自动机FA可以理解成状态控制器

- 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



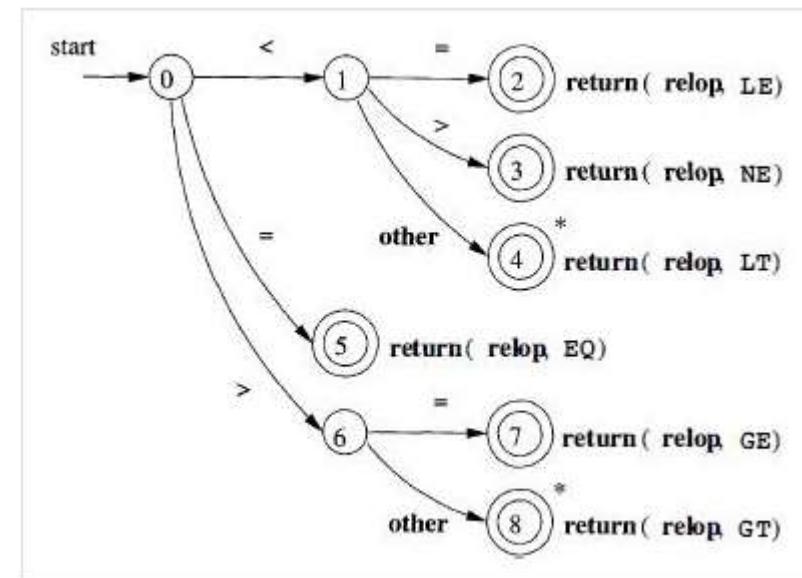
2.4 有限状态自动机

- **有限自动机的五要素**
 - 有限状态集 S_S --- 结点
 - 有限输入符号集 Σ
 - 转移函数 $\delta(s,a) = t$
 - 一个开始状态 s_0
 - 一个终止状态集 T_S
 - 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。**



2.4 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





2.5 确定有限自动机

2.5 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 $\{S_0, S_1, S_2, \dots\}$
- 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, $TS \subseteq SS$

2.5 确定的有限状态自动机

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

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

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

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

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

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

2.5 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；

2.5 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：



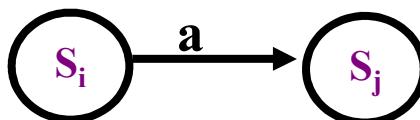
终止状态：



开始状态：



边：表示状态转换函数： $f(S_i, a) = S_j$



2.5 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:

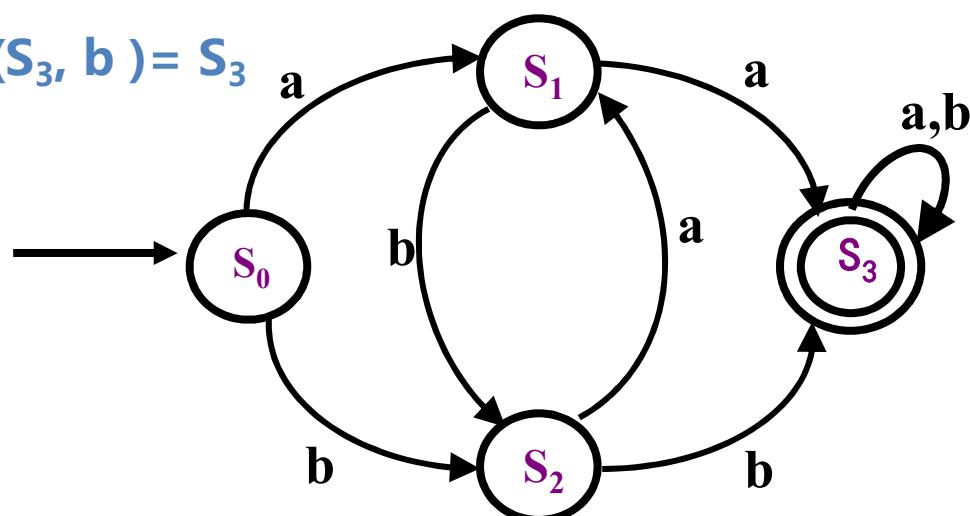
$$f(S_0, a) = S_1 \quad f(S_2, a) = S_1$$

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

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

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

状态转换图



2.5 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

2.5 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

2.5 确定的有限状态自动机

■ DFA: 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:

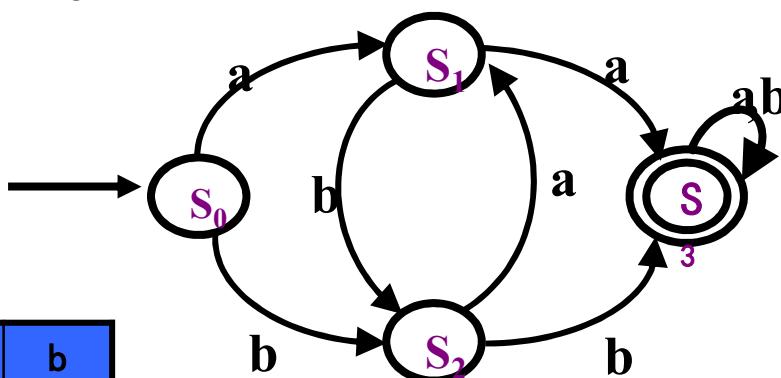
$$f(S_0, a) = S_1, f(S_2, a) = S_1$$

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

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

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

	a	b
0 ⁺	1	2
1	3	2
2	1	3
3 ⁻	3	3



2.5 确定的有限状态自动机

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

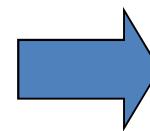
$$\delta(0, a) = 1 \quad \delta(0, b) = 4$$

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

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

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

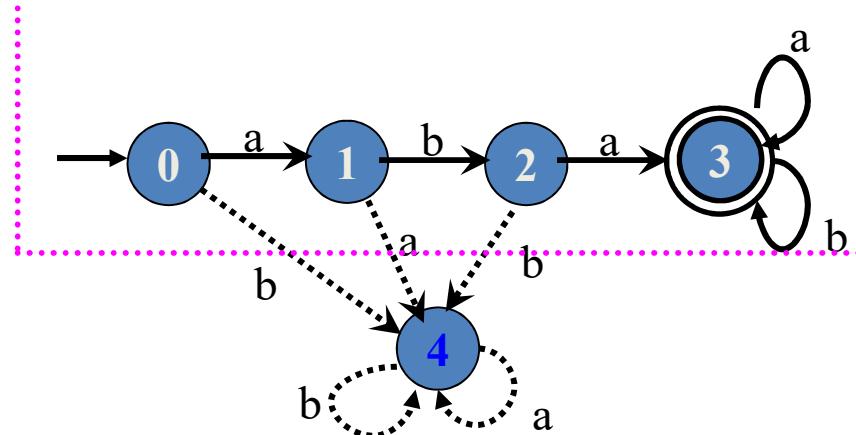
$$\delta(4, a) = 4 \quad \delta(4, b) = 4$$



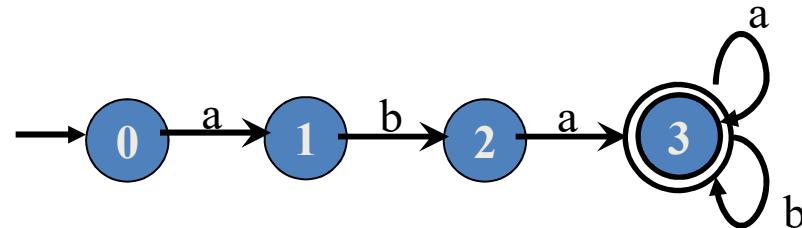
	a	b
0 ⁺	1	4
1	4	2
2	3	4
3 ⁻	3	3
4	4	4

2.5 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



2.5 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

2.5 确定的有限状态自动机

■ DFA例2

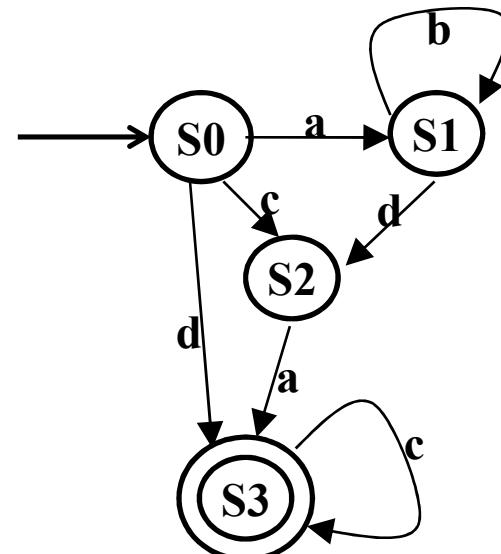
$\Sigma: \{a, b, c, d\}$

$SS: \{S0, S1, S2, S3\}$

开始状态: $S0$

终止状态集: $\{S3\}$

$f: \{(S0,a) \rightarrow S1, (S0,c) \rightarrow S2,$
 $(S0,d) \rightarrow S3, (S1,b) \rightarrow S1,$
 $(S1,d) \rightarrow S2, (S2,a) \rightarrow S3,$
 $(S3, c) \rightarrow S3\}$



2.5 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: s_0
 - 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

2.5 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, \dots, S_n) , 满足

$$S_0 \xrightarrow{a_1} S_1, S_1 \xrightarrow{a_2} S_2, \dots, S_{n-1} \xrightarrow{a_n} S_n$$

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.

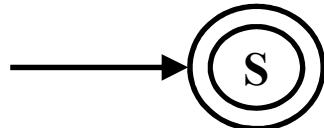
■ DFA定义的串的集合(DFA接受的语言)

- DFA M接受的所有串的集合, 称为M定义的语言, 记为 $L(M)$

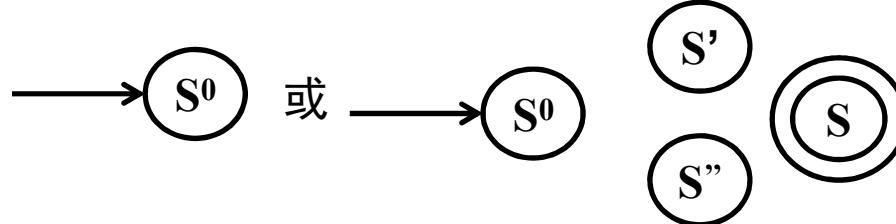
2.5 确定的有限状态自动机

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 $L(\varepsilon) = \{\varepsilon\}$

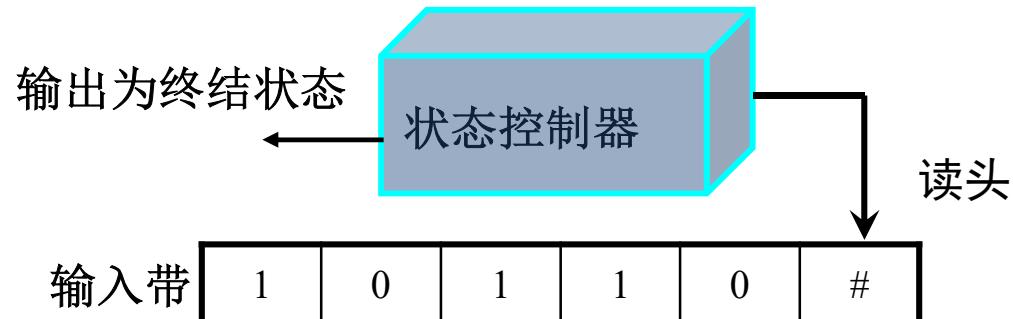


- 若DFA M 只有一个状态，并且是开始状态或DFA M 有若干个状态，但开始状态到终止状态之间没有通路，则DFA M 定义的串集是空集 \emptyset



2.5 确定的有限状态自动机

■ DFA接受的语言

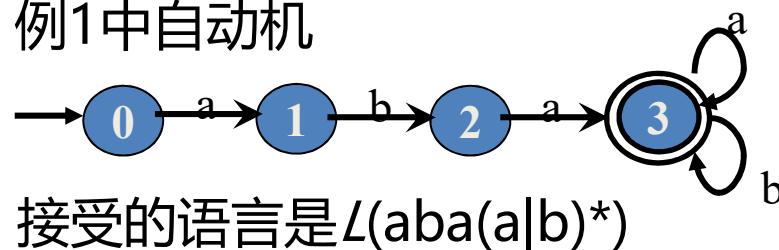


FA 和 RE 的关系? ?

2.5 确定的有限状态自动机

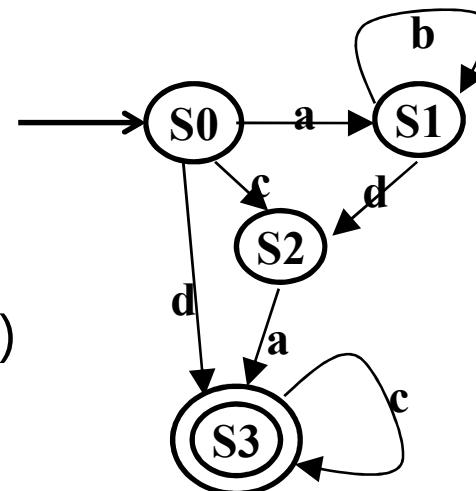
■ DFA接受的语言

■ 例1中自动机



■ 例2中自动机

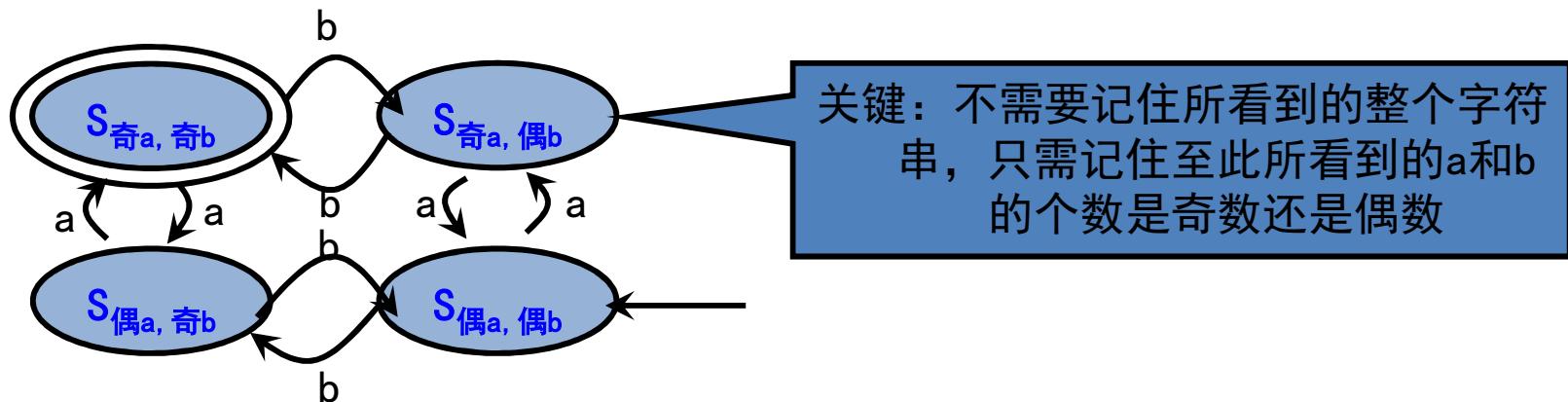
接受的语言是 $L((ab^*da \mid ca \mid d)c^*)$



2.5 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： $\Sigma = \{a,b\}$, 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。



2.5 确定的有限状态自动机

■ 自动机的设计

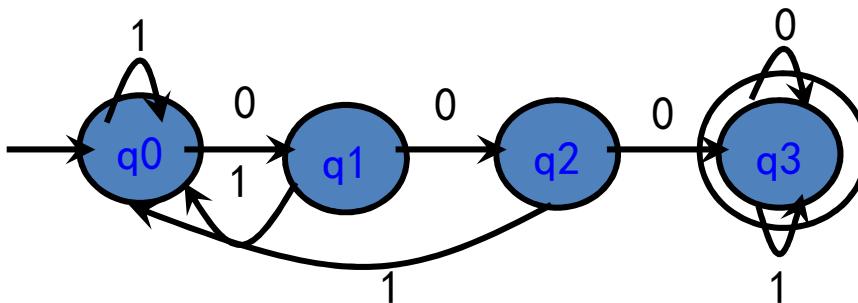
- 例2：设计有限自动机M，识别 $\{0,1\}$ 上的语言

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

思考：what if $L = \{x000x \mid x \in \{0,1\}^*\}???$

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。

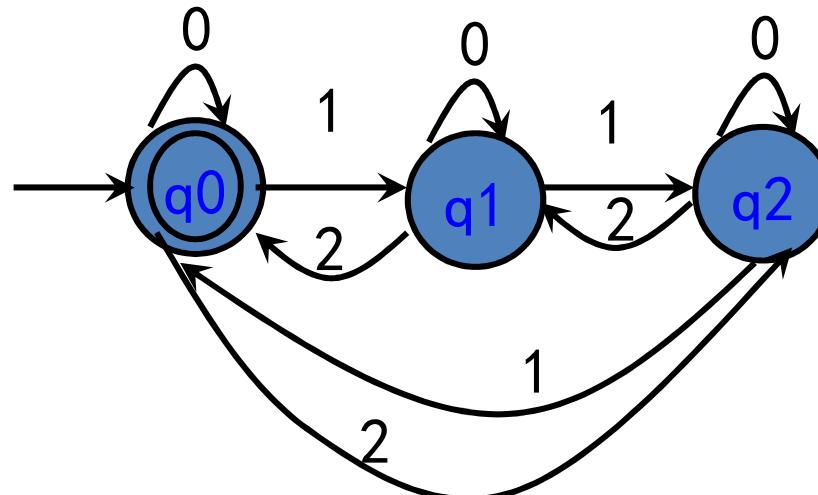


2.5 确定的有限状态自动机

■ 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。



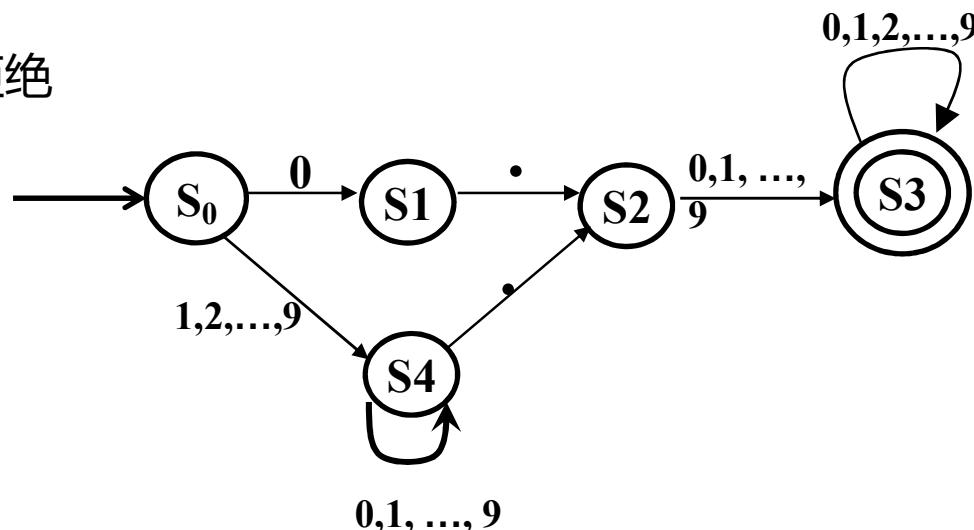
2.5 确定的有限状态自动机

■ 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



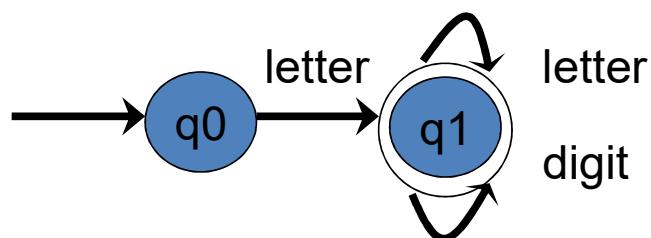
2.5 确定的有限状态自动机

■ 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

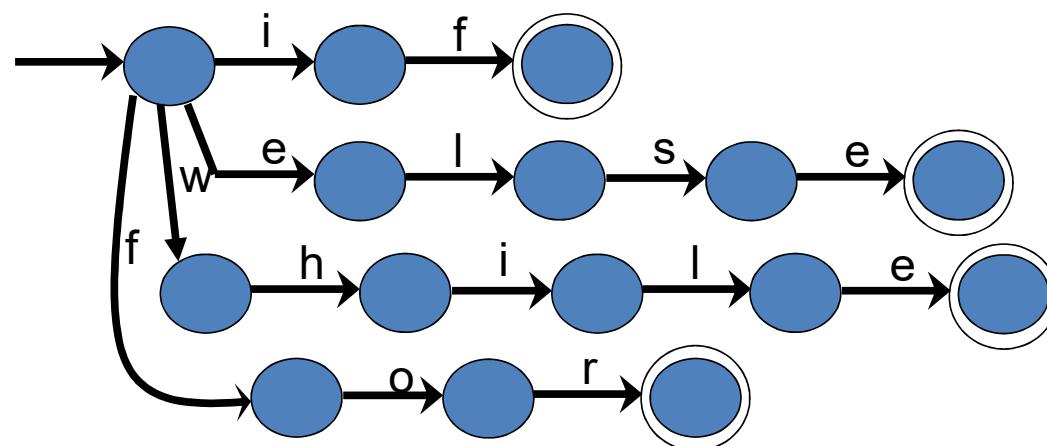
23x, 12_x, _x 拒绝



2.5 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}



2.5 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

2.5 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 中目前正在被读的字符;

2.5 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.

2.5 确定的有限状态自动机

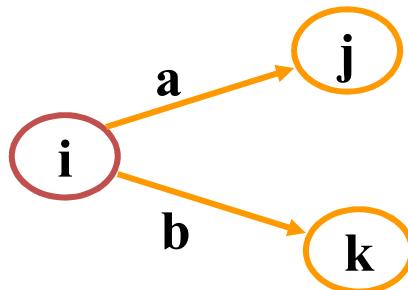
	a	b	c	d
S0+	S1	⊥	S2	S3
S1	⊥	S1	⊥	S2
S2	S3	⊥	⊥	⊥
S3-	⊥	⊥	S3	⊥

- $S0 \rightarrow_{(c)} S2 \rightarrow_{(a)} S3 \rightarrow_{(b)} \perp$
1) cab 接受 or 拒绝?
拒绝
- $S0 \rightarrow_{(a)} S1 \rightarrow_{(b)} S1 \rightarrow_{(d)} S2 \rightarrow_{(a)} S3 \rightarrow_{(c)} S3$
2) abdacc 接受 or 拒绝?
接受

2.5 确定的有限状态自动机

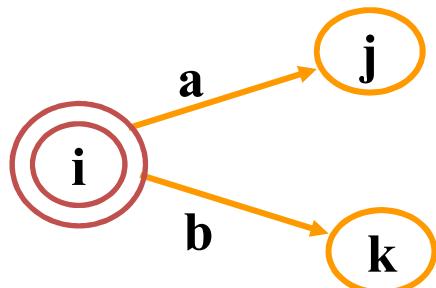
■ 基于转换图的DFA实现

- 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()
```

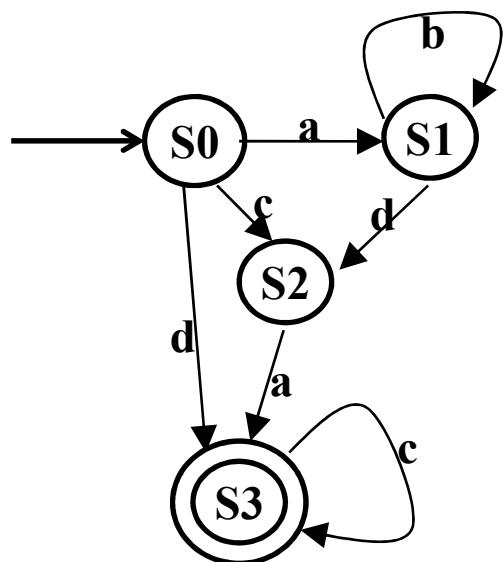
2.5 确定的有限状态自动机



Li: case CurrentChar of

a : goto Lj
b : goto Lk
: return true;
other : return false;

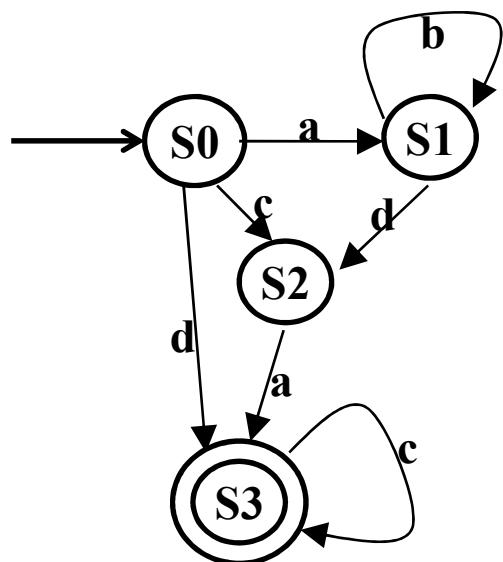
2.5 确定的有限状态自动机



```
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; }
```

2.5 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; }
```

2.5 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，识别程序不需要改变

■ 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。



2.6 非确定有限自动机

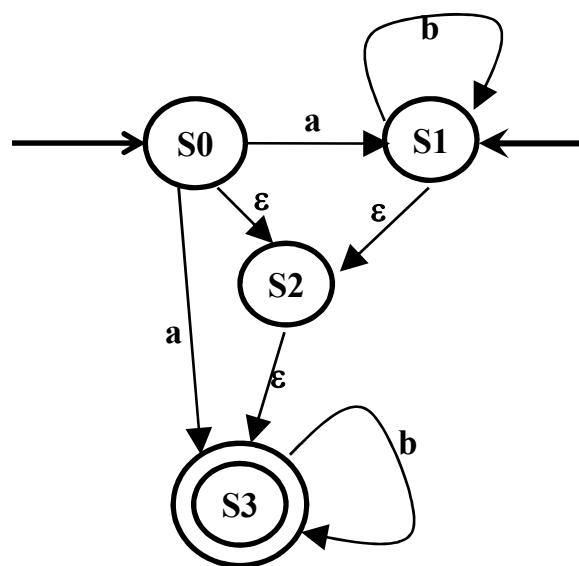
2.6 非确定的有限状态自动机

- 确定有限自动机DFA是五元组($SS, \Sigma, \delta, s0, TS$)
 - 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($SS, \Sigma, \delta, S0, TS$)
 $SS = \{S0, S1, \dots, Sn\}$ 是状态集
 - Σ 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
 - $\delta : SS \times (\Sigma \cup \epsilon) \rightarrow 2^{SS}$
 - $TS \subseteq SS$ 是终止状态集

2.6 非确定的有限状态自动机

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

2.6 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

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

$$S_0 \xrightarrow{a_1} S_1, S_1 \xrightarrow{a_2} S_2, \dots, S_{n-1} \xrightarrow{a_n} S_n$$

其中 S_0 是开始状态之一, S_n 是接受状态之一, 则串 $a_1 a_2 \dots a_n$ 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 $L(M)$

作业

- 教材P86: 3.4.1, 3.4.2 (1, 2, 3, 5, 6, 8, 9)
- 教材P96: 3.6.3, 3.6.4



2.7 NFA v.s. DFA

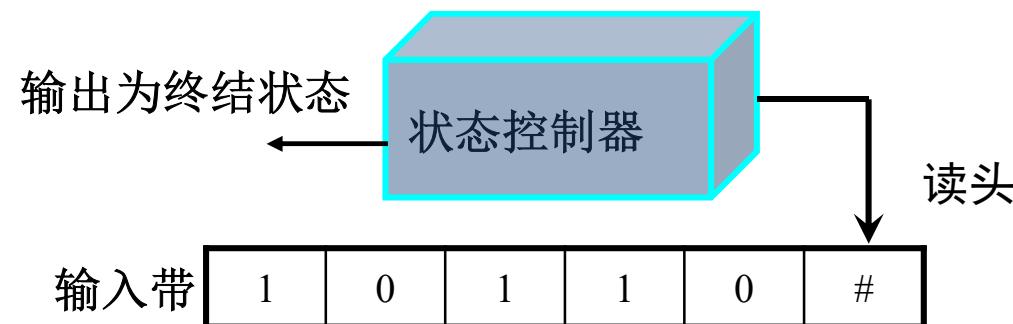
2.7 NFA v.s. DFA

	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 \perp	$\{S_1, \dots, S_n\}$ or \perp
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！

2.7 NFA v.s. DFA

- DFA/NFA接受的字符串(可以等价):



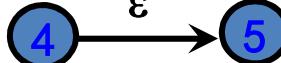
ϵ 意味着读头不动，但是状态依旧发生转换

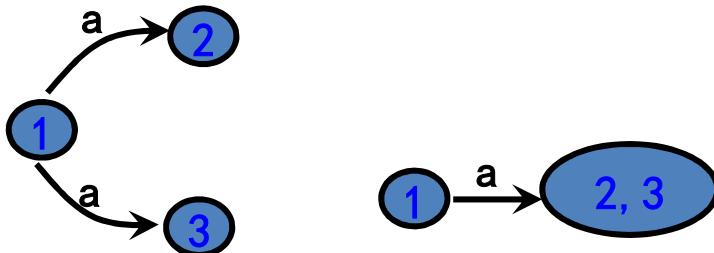
2.7 NFA v.s. DFA

- 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。

2.7 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。



2.7 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

2.7 NFA v.s. DFA

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

2.7 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 ;

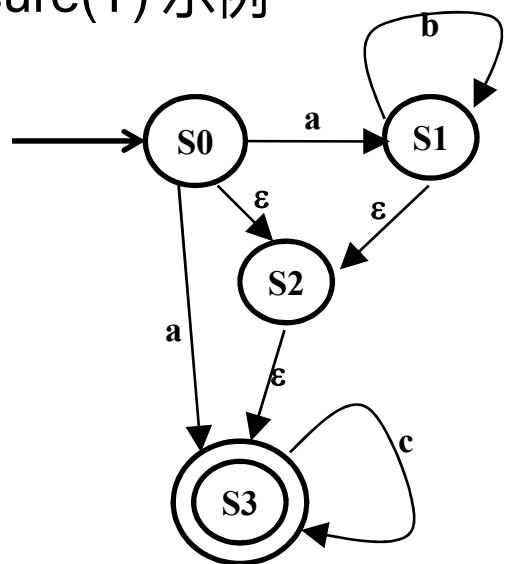
第二步: 如果在状态集 T 中存在状态 s ,
到状态 s' 存一条 ϵ 边,
并且 $s' \notin \epsilon$ -closure(T),
则将 s' 加入 T 的空闭包 ϵ -closure(T);
重复第二步, 直到再没有状态可加入 ϵ -closure(T).

```
push all states of  $T$  onto stack;  
initialize  $\epsilon$ -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  $\epsilon$  )  
        if (  $u$  is not in  $\epsilon$ -closure( $T$ ) ) {  
            add  $u$  to  $\epsilon$ -closure( $T$ );  
            push  $u$  onto stack;  
        }  
}
```

2.7 NFA v.s. DFA

■ 由NFA构造DFA (子集法)

■ ϵ -closure(T) 示例

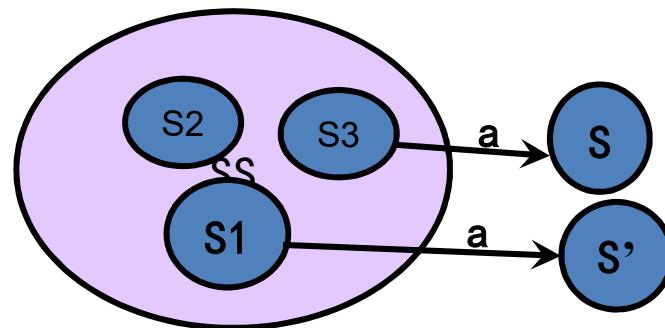


ϵ -closure({S0, S1}) =
{S0, S1 }
{S0, S1, S2}
{S0, S1, S2}
{S0, S1, S2,S3}

2.7 NFA v.s. DFA

■ 由NFA构造DFA (子集法)

- 对于NFA Λ 中的给定状态集合 T 和符号 a , $\text{Move}(T, a) = \{s \mid$ 对于状态集 T 中的一个状态 s_1 , 如果 Λ 中存在一条从 s_1 到 s 的 a 转换边 $\}$

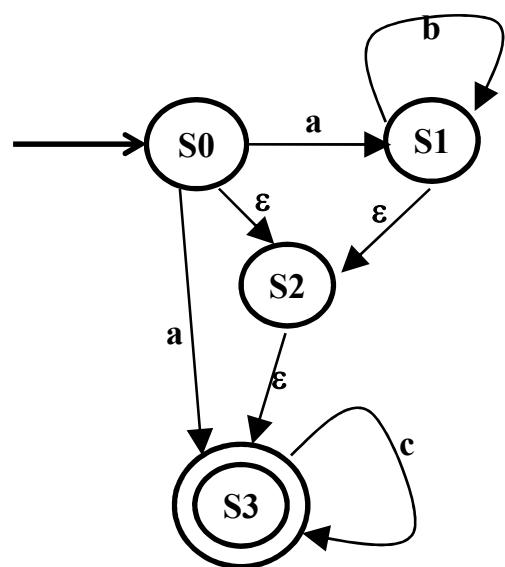


$$\text{Move } (\{S1, S2, S3\}, a) = \{S, S'\}$$

2.7 NFA v.s. DFA

■ 由NFA构造DFA (子集法)

■ Move(T, a) 示例



$\text{Move}(\{S0, S1\}, a) = \{S1, S3\}$

$\text{Move}(\{S0, S1\}, b) = \{S1\}$

2.7 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。

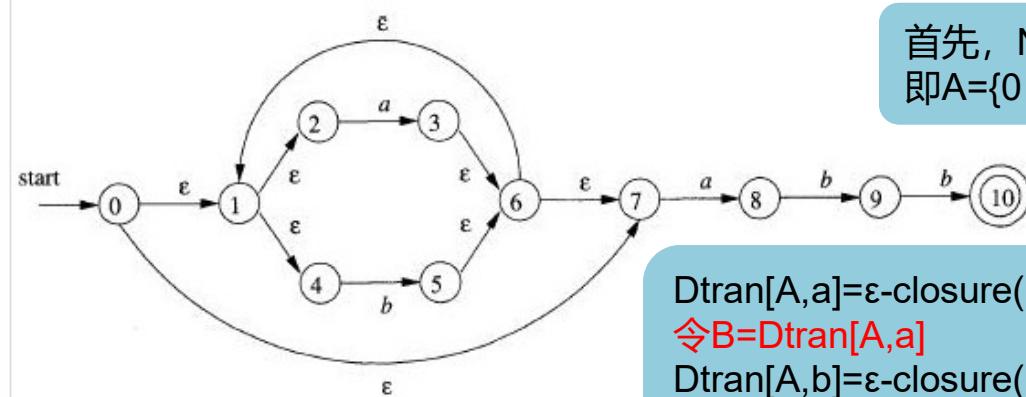
```
initially,  $\epsilon$ -closure( $s_0$ ) is the only state in Dstates, and it is unmarked;  
while ( there is an unmarked state T in Dstates ) {  
    mark T;  
    for ( each input symbol a ) {  
        U =  $\epsilon$ -closure(move(T, a));  
        if ( U is not in Dstates )  
            add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

2.7 NFA v.s. DFA

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



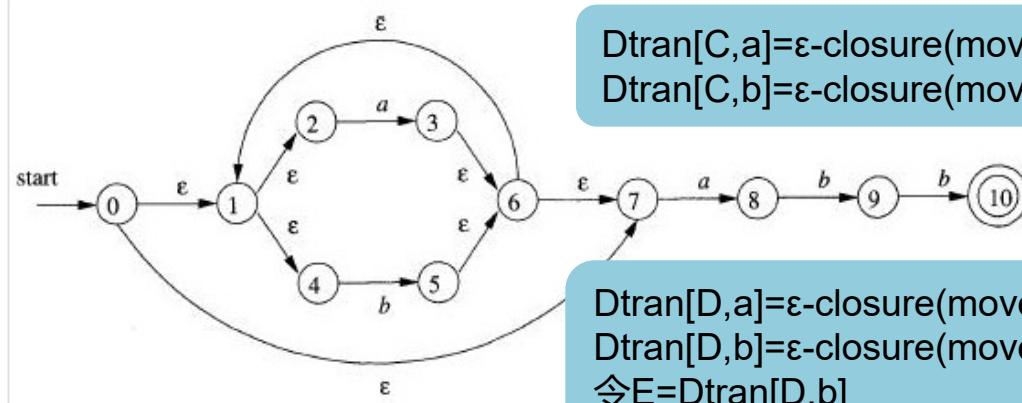
首先，NFA的开始状态A是 ϵ -closure(0)，
即A={0, 1, 2, 4, 7}，NFA的输入字母表是{a,b}

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]

2.7 NFA v.s. DFA

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



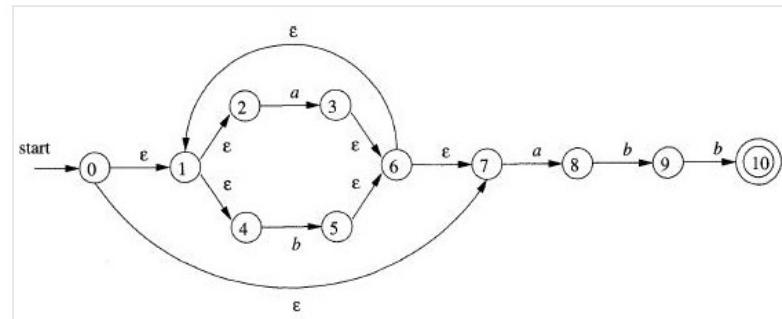
$D_{\text{tran}}[C, a] = \epsilon\text{-closure}(\text{move}(C, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$
 $D_{\text{tran}}[C, b] = \epsilon\text{-closure}(\text{move}(C, b)) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 6, 7\} = C$

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

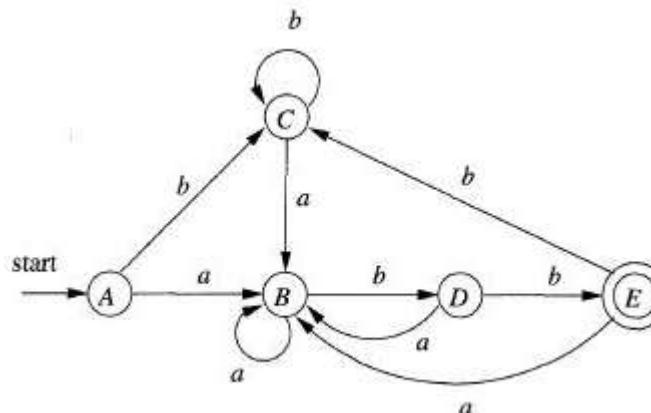
$D_{\text{tran}}[E, a] = \epsilon\text{-closure}(\text{move}(E, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$
 $D_{\text{tran}}[E, b] = \epsilon\text{-closure}(\text{move}(E, b)) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 6, 7\} = C$

2.7 NFA v.s. DFA

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



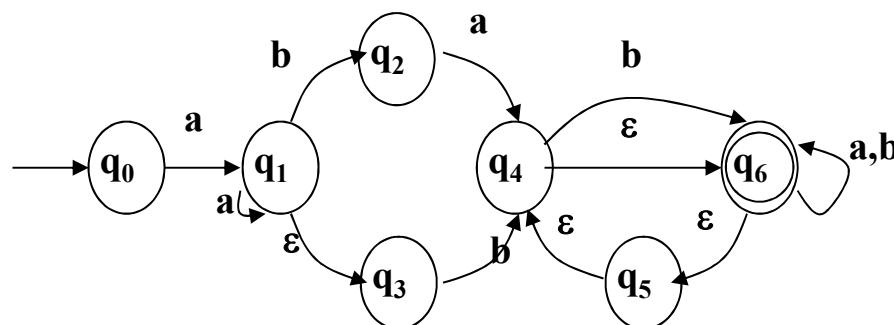
NFA STATE	DFA STATE	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 3, 5, 6, 7, 10}	E	B	C



2.7 NFA v.s. DFA

■ 由NFA构造DFA (子集法) 例2:

	a	b
$\{q_0\}^+$	$\{q_1, q_3\}$	\perp
$\{q_1, q_3\}$	$\{q_1, q_3\}$	$\{q_2, q_4, q_6, q_5\}$
$\{q_2, q_4, q_6, q_5\}^-$	$\{q_4, q_6, q_5\}$	$\{q_6, q_5, q_4\}$
$\{q_4, q_6, q_5\}^-$	$\{q_6, q_5, q_4\}$	$\{q_6, q_5, q_4\}$



作业

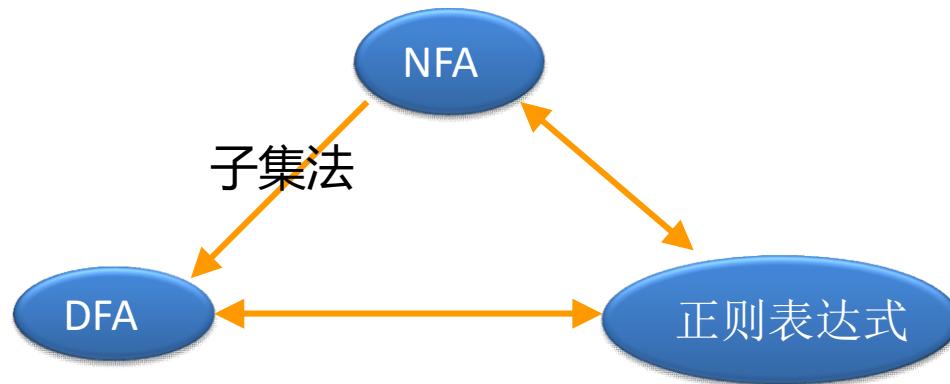
- 教材P96: 3.6.5
- 教材P105: 3.7.1, 3.7.3 (4)



2.8 RE v.s. NFA/DFA

2.8 RE v.s. NFA/DFA

- 对 Σ 上的每一个正则表达R，存在一个 Σ 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定有限自动机D



2.8 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)

2.8 RE v.s. NFA/DFA

■ 从RE生成FA，用来模拟RE的实现

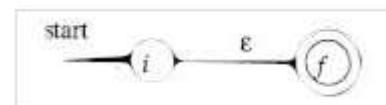
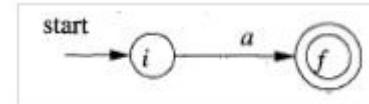
- 方法一：RE → NFA → DFA → 最小DFA
- 方法二：RE → DFA → 最小DFA
- 方法三：RE → NFA （然后直接模拟，模拟算法见教材P99，算法3.22）

2.8 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 Σ 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 ϵ , 构造右边的NFA

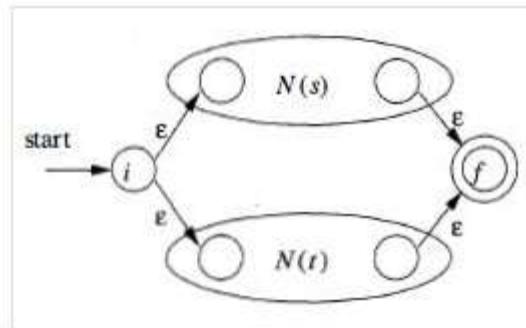
又是递归：
 r 由sub- r 递归构成
 $N(r)$ 也由 $N(\text{sub-}r)$ 递归构成



2.8 RE v.s. NFA/DFA

■ 从RE生成FA，用来模拟RE的实现

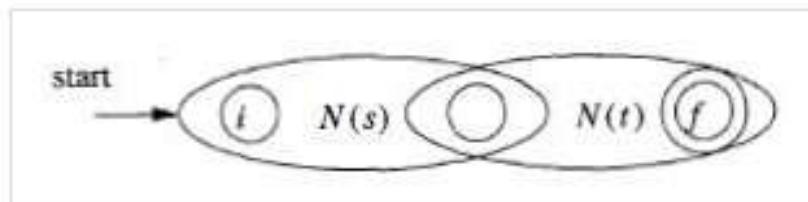
- 方法一：由RE构造NFA (Thompson算法)
 - 归纳规则：假设正则表达式s和t的NFA分别为 $N(s)$ 和 $N(t)$
 - 对于 $r = s|t$, 构造如下NFA, 这里i和f都是新状态, 分别是 $N(r)$ 的开始状态和接受状态, 从i到 $N(s)$ 和 $N(t)$ 的开始状态各有一个 ϵ 转换, 从 $N(s)$ 和 $N(t)$ 到接受状态f也各有一个 ϵ 转换



2.8 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



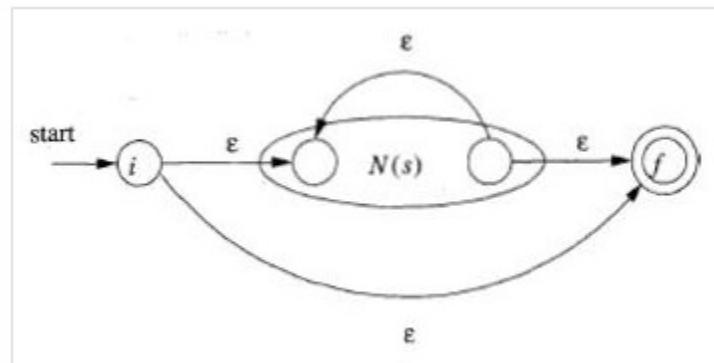
2.8 RE v.s. NFA/DFA

■ 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 $N(t)$

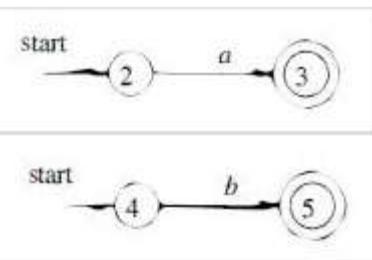
- 对于 $r=s^*$, 构造下面的NFA, i 和 f 是两个新状态, 分别是 $N(r)$ 的开始状态和唯一接受状态; 要从 i 到达 f , 我们可以沿着新引入的标号为 ϵ 的路径前进, 该路径对应于 $L(s)$ 的一个串; 我们也可以达到 $N(s)$ 的开始状态, 然后经过该NFA, 零次或多次从它的接受状态回到它的开始状态并重复上述过程



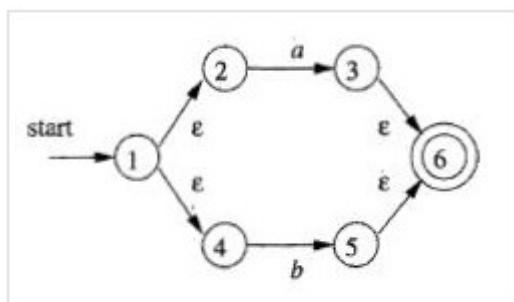
2.8 RE v.s. NFA/DFA

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

- 为表达式 $r1 = a$, $r2=b$ 构造NFA

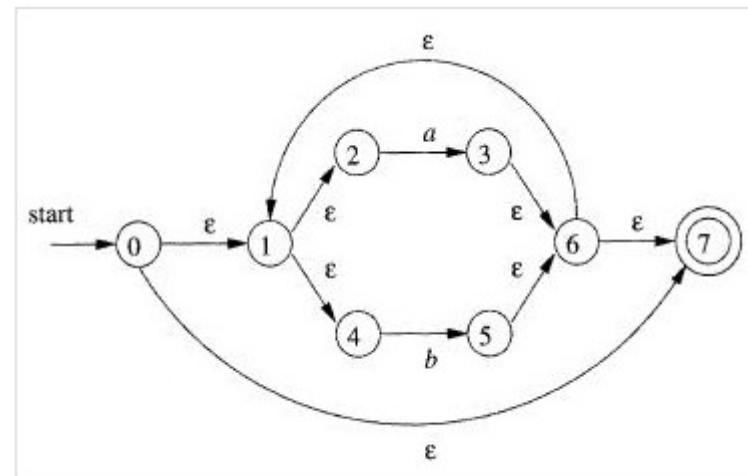
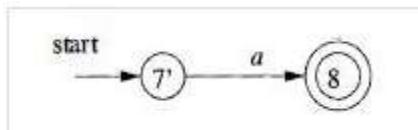


- 为表达式 $r3 = r1|r2$ 构造NFA



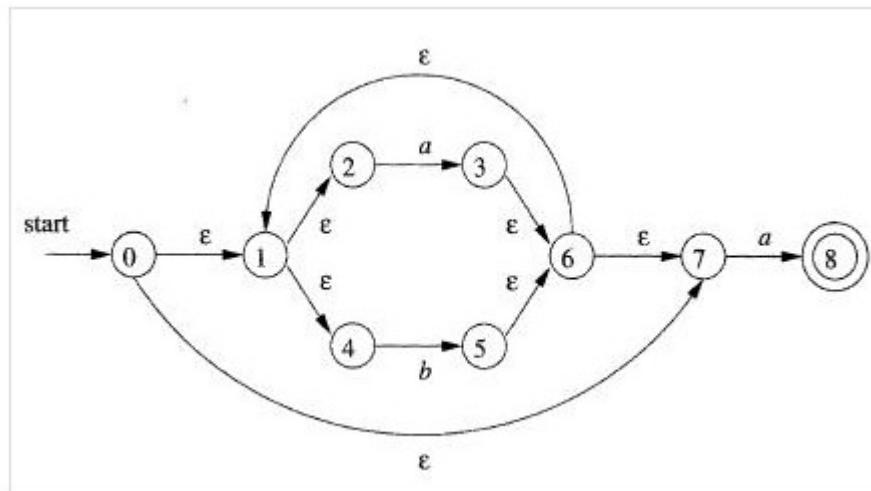
2.8 RE v.s. NFA/DFA

- 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



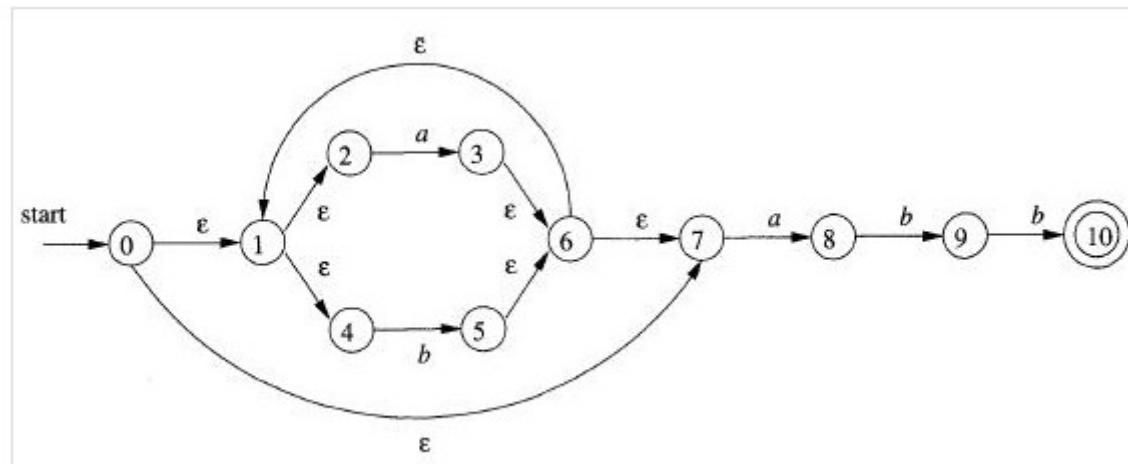
2.8 RE v.s. NFA/DFA

- 由RE构造NFA (Thompson算法), 为 $r=(a|b)^*abb$ 构造NFA
 - 为表达式 $r7 = r5r6$ 构造NFA



2.8 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



2.8 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)

2.8 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）

2.8 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。

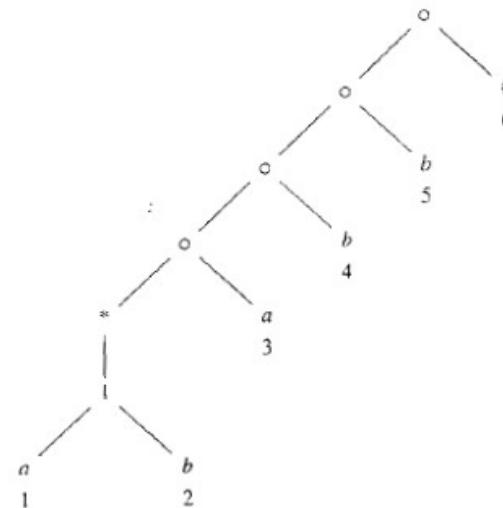


图 3-56 $(a|b)^*abb\#$ 的抽象语法树

2.8 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 ϵ 的离开转换，则称该状态为重要状态(important state) --- 子集法在计算move(T, a)的时候，只使用了重要状态
 - 计算四个函数nullalbe, firstpos, lastpos, followpos

2.8 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 $\text{nullable}(n)=\text{true}$ ；“否”则 $\text{nullable}(n)=\text{false}$
 - **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

2.8 RE v.s. NFA/DFA

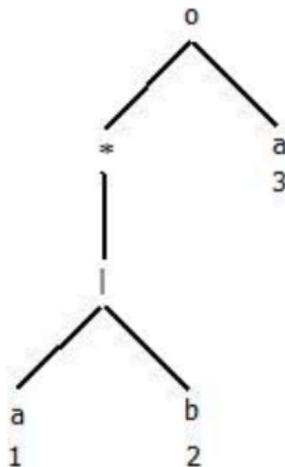
■ 从RE生成FA，用来模拟RE的实现

- 方法二：由RE直接构造DFA
- 小例子

nullable(n) = false

firstpos(n)={1,2,3}

lastpos(n) = {3}



2.8 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nullalbe, firstpos, lastpos

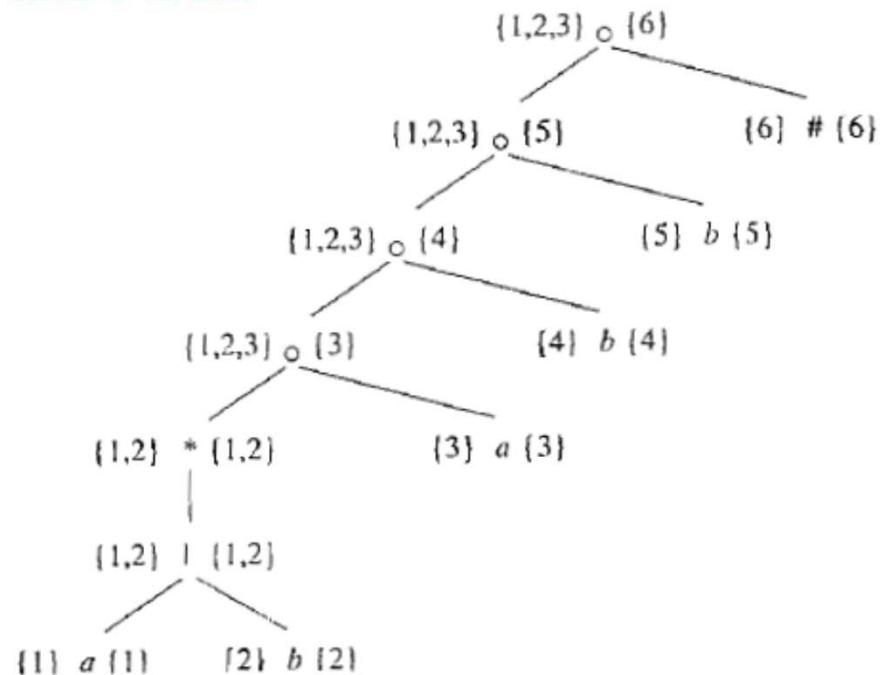
node n	nullable(n)	firstpos(n)	lastpos(n)
A leaf labeled ϵ	true	\emptyset	\emptyset
A leaf with position i	false	{i}	{i}
An or-node $n=c_1 c_2$	nullable(c_1) or nullable(c_2)	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$
A cat-node $n=c_1c_2$	nullable(c_1)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

2.8 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 $(a|b)^*abb \rightarrow (a|b)^*abb\#$



2.8 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 p, q 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

2.8 RE v.s. NFA/DFA

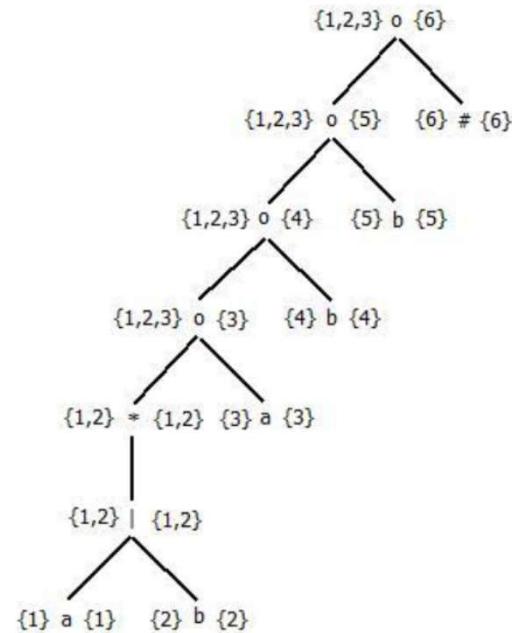
■ 从RE生成FA，用来模拟RE的实现

- 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。

2.8 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
 - $\text{followpos}(p)$
 - Applying rule 1
 - $\text{followpos}(1)$ incl. $\{3\}$
 - $\text{followpos}(2)$ incl. $\{3\}$
 - $\text{followpos}(3)$ incl. $\{4\}$
 - $\text{followpos}(4)$ incl. $\{5\}$
 - $\text{followpos}(5)$ incl. $\{6\}$
 - Applying rule 2
 - $\text{followpos}(1)$ incl. $\{1,2\}$
 - $\text{followpos}(2)$ incl. $\{1,2\}$



2.8 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)

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

图 3-60 函数 $followpos$

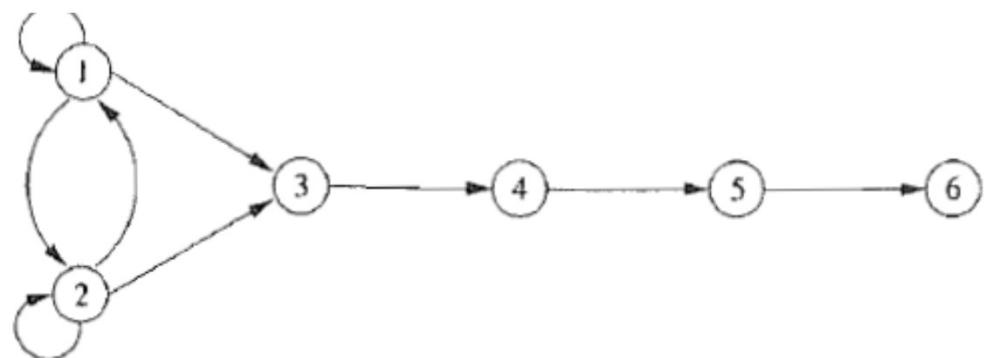


图 3-61 表示函数 $followpos$ 的有向图

2.8 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

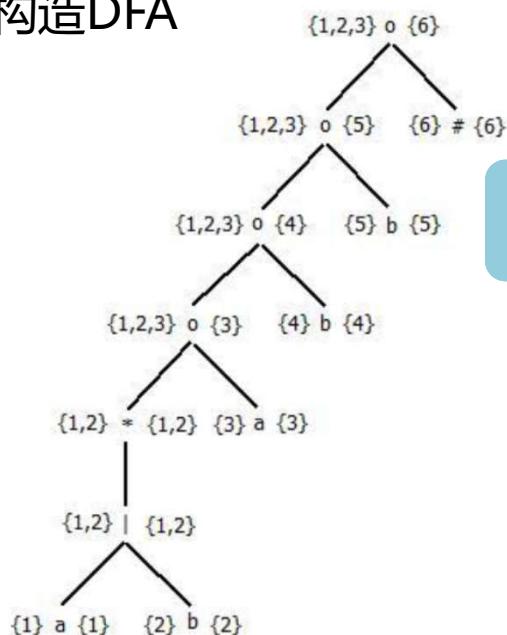
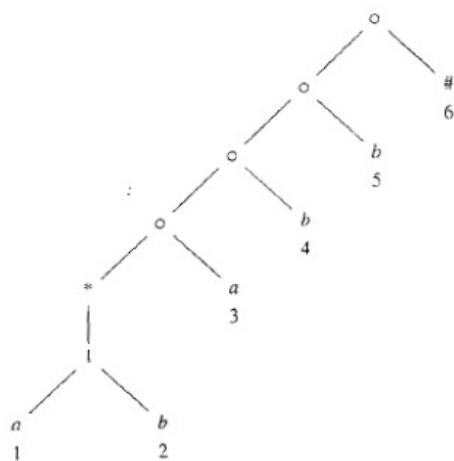
```
初始化  $Dstates$ , 使之只包含未标记的状态  $firstpos(n_0)$ ,  
其中  $n_0$  是  $(r)^\#$  的抽象语法树的根结点;  
while ( $Dstates$  中存在未标记的状态  $S$ ) {  
    标记  $S$ ;  
    for (每个输入符号  $a$ ) {  
        令  $U$  为  $S$  中和  $a$  对应的所有位置  $p$  的  $followpos(p)$  的并集;  
        if ( $U$  不在  $Dstates$  中)  
            将 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

2.8 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n0的
 $\text{firstpos}(n_0)=\{1,2,3\}$ ，标记为A

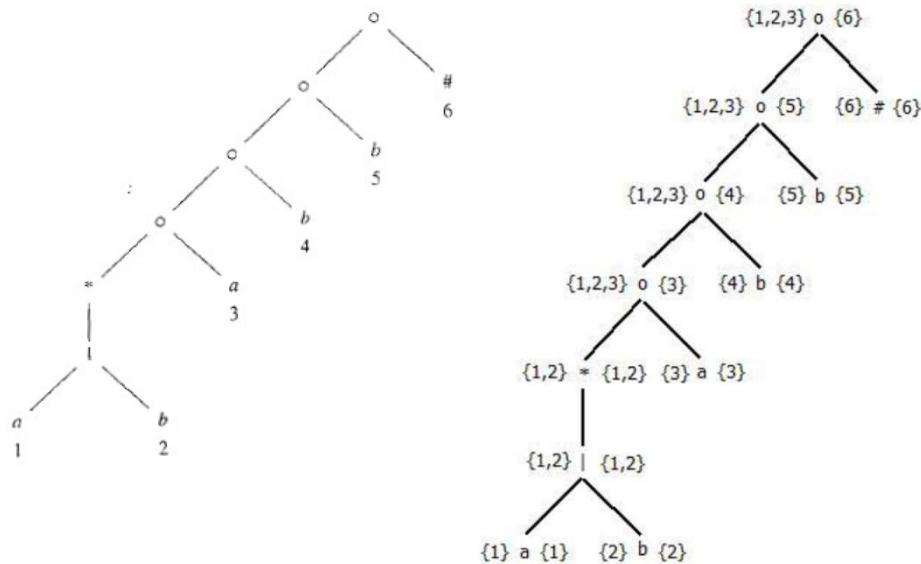
位置 n	$\text{followpos}(n)$
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	\emptyset

图 3-60 函数 followpos

2.8 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



从DFA的开始状态 $A = \text{firstpos}(n_0) = \{1, 2, 3\}$, 起 we have
 $D\text{tran}[A, a] = \text{followpos}(1) \cup \text{followpos}(3) = \{1, 2, 3, 4\} = B$
 $D\text{tran}[A, b] = \text{followpos}(2) = \{1, 2, 3\} = A$

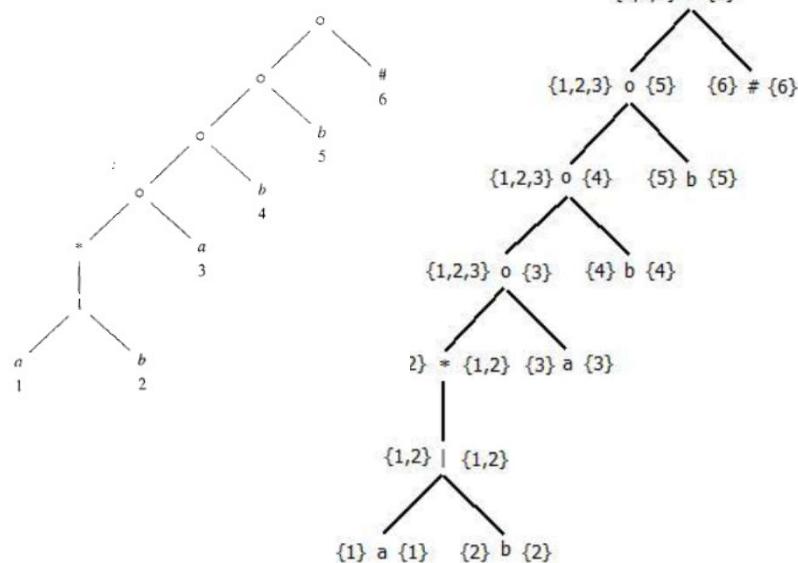
位置 n	$\text{followpos}(n)$
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	\emptyset

图 3-60 函数 followpos

2.8 RE v.s. NFA/DFA

■ 从RE生成FA，用来模拟RE的实现

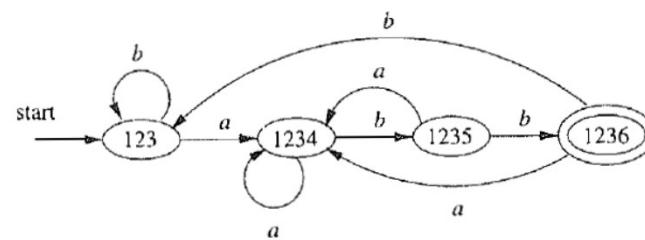
■ 方法二：由RE直接构造DFA



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

图 3-60 函数 $followpos$

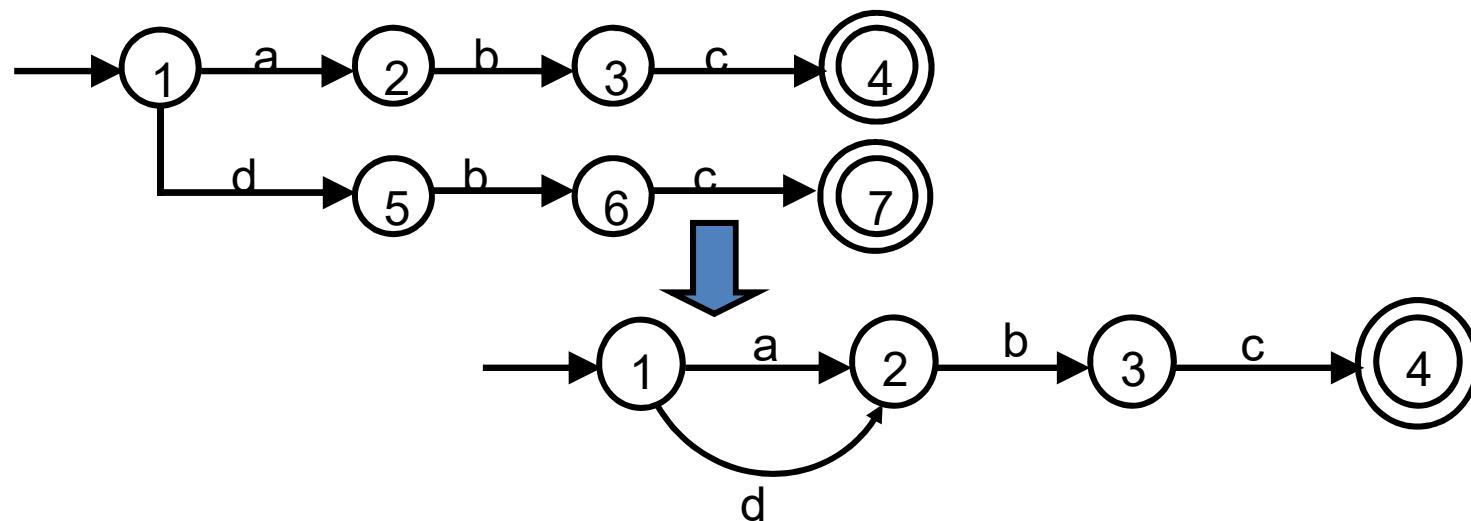
$Dtran[B,a] = followpos(1) \cup followpos(3) = B$
 $Dtran[B,b] = followpos(2) \cup followpos(4) = \{1,2,3,5\} = C$



2.8 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。

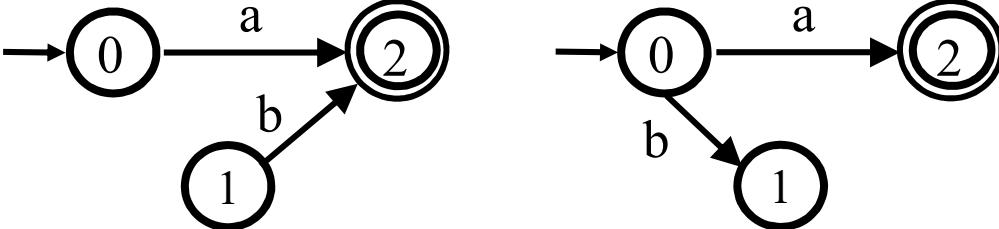


2.8 RE v.s. NFA/DFA

■ 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。

2.8 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）

2.8 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。

2.8 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)

2.8 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 Σ ，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: Π 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 Π ，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 Π_{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;
}
```

2.8 RE v.s. NFA/DFA

■ 状态分离法化简DFA

■ 方法:

- 3)如果 $\Pi_{new} = \Pi$,令 $\Pi_{final} = \Pi$ 并接着执行步骤 4), 否则, 用 Π_{new} 替换 Π 并重
复步骤2)。
- 4)在分划 Π_{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 Π , 直至无法继续分割为止
(即, 该状态子集中的状态都为等价)

2.8 RE v.s. NFA/DFA

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

2.8 RE v.s. NFA/DFA

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

不可再分

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)

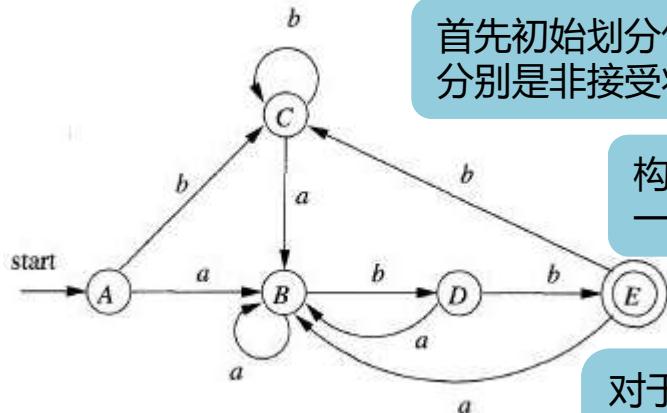
2.8 RE v.s. NFA/DFA

- 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 δ 见表

状态	a	b
A	A	B
B	A	C
C	C	B

2.8 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。

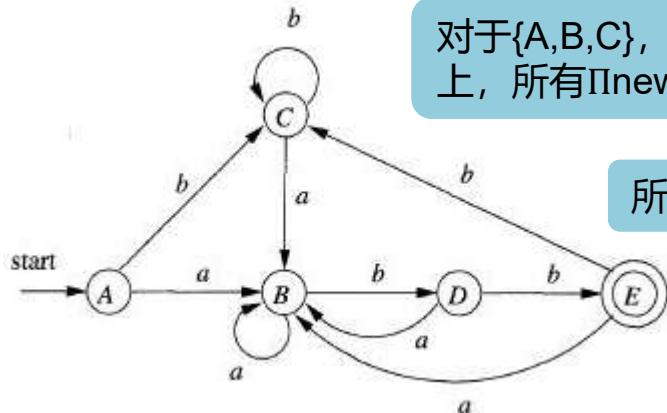
构造 Π_{new} 时，考虑这两个组和输入符号a和b，因为组{E}只包含一个状态，不能再被分割，所以{E}被原封不动的保留在 Π_{new} 中。

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 Π_{new} 中，{A,B,C,D}被分割成{A,B,C}, {D}, 现在 Π_{new} 中有{A,B,C}, {D}, {E}。

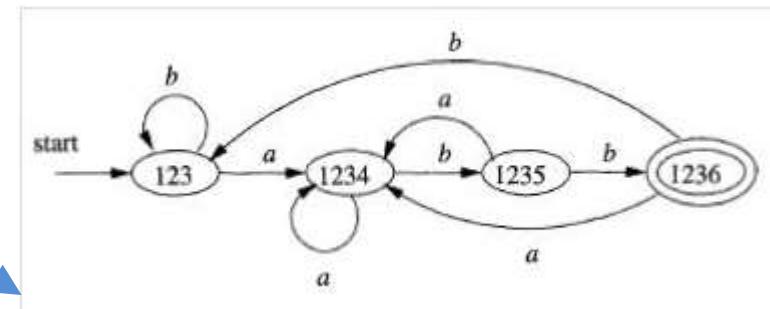
2.8 RE v.s. NFA/DFA

■ 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 Π_{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



2.8 RE v.s. NFA/DFA

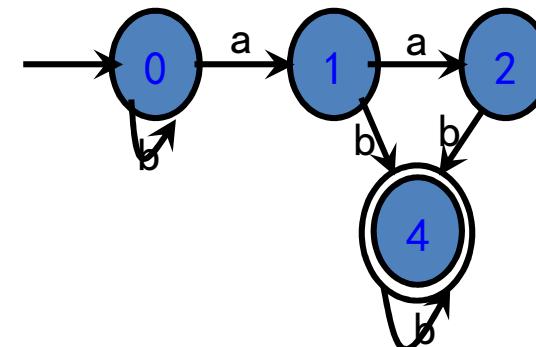
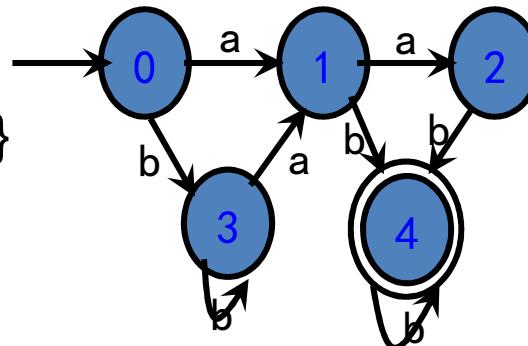
■ 状态分离法化简DFA

- 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 Π_{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。

2.8 RE v.s. NFA/DFA

例1:

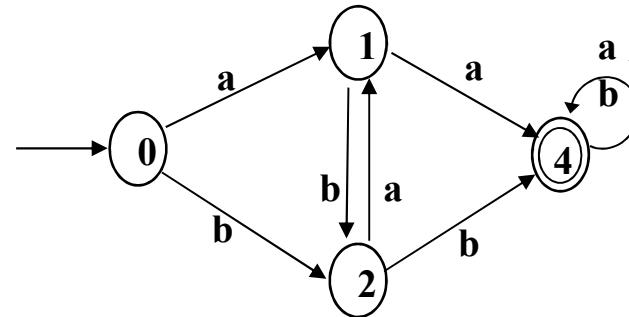
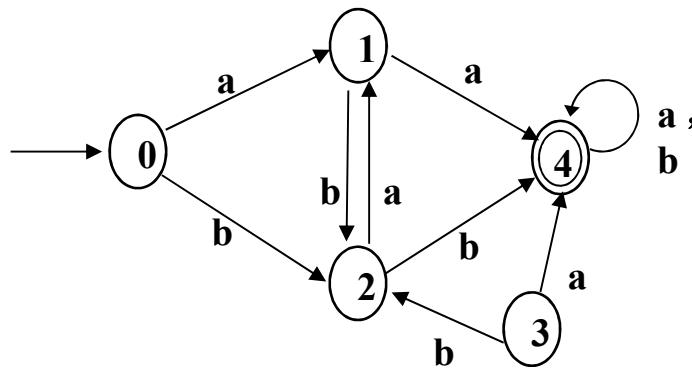
- $\{0, 1, 2, 3\}$ 和 $\{4\}$
- $\{0, 1, 3\}, \{2\}$ 和 $\{4\}$
- $\{0, 3\}, \{1\}, \{2\}$ 和 $\{4\}$



2.8 RE v.s. NFA/DFA

例2:

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



2.8 RE v.s. NFA/DFA

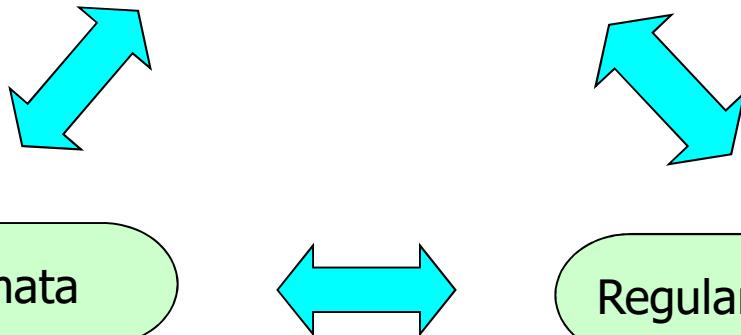
■ RE, DFA(NFA), L(RE)三者等价

Regular Expression

证明RE等价：
可以证明它们对应的最小DFA同构

Finite Automata

Regular Grammar



作业

- 教材P105: 2.7.3(4)
- 教材P109: 2.8.1

附加思考题

- 教材P118: 3.9.3 (用算法3.36构造), 3.9.4



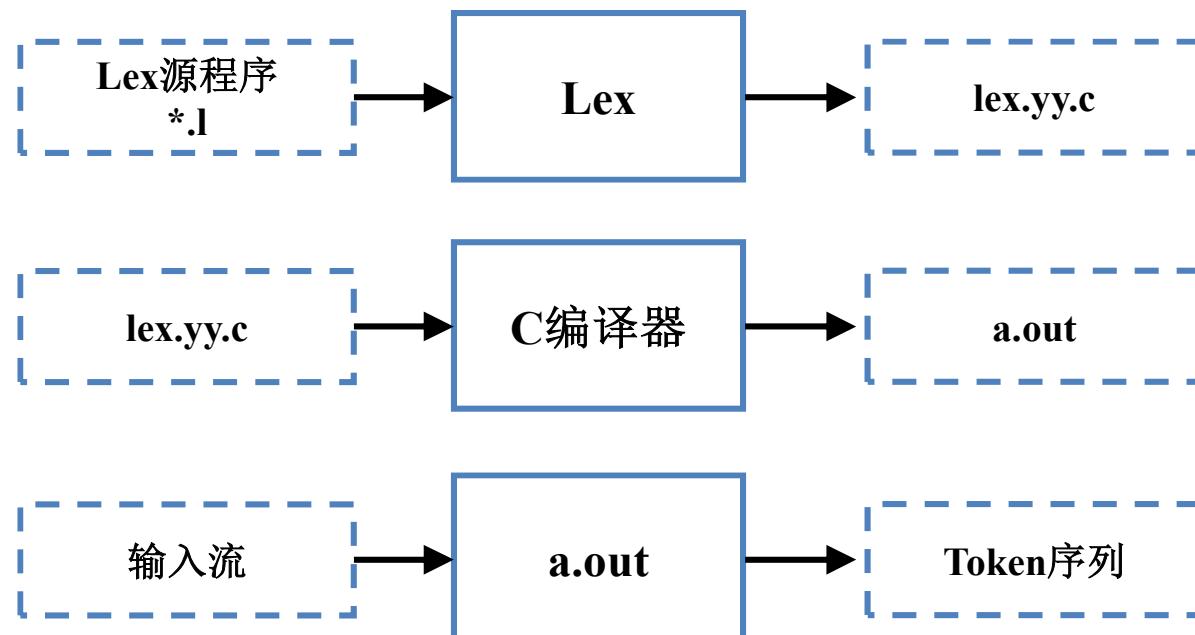
2.9 词法分析器的开发

2.9 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 Generator)
- 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。



2.9 词法分析器的开发



用Lex创建一个词法分析器的过程

2.9 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

yyleng: lexeme的长度

2.9 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

2.9 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-zA-Z][a-zA-Z0-9]* printf("%s\n",yytext);/*模式3*/
```

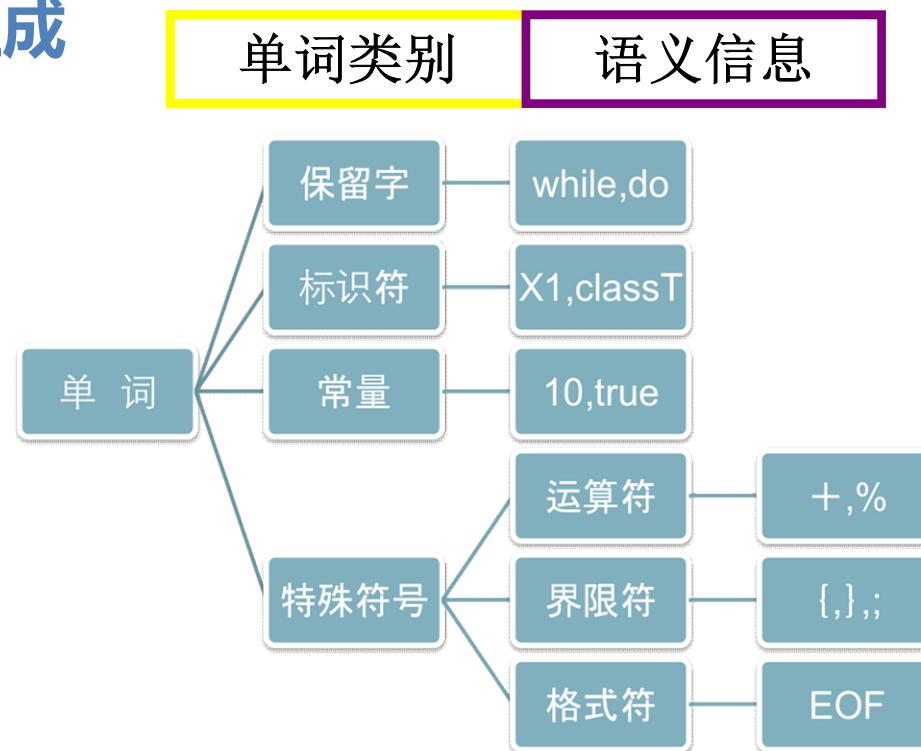
当输入串为“programming”时，模式1（匹配“program”）和模式3（“programming”）都匹配，但会选择匹配串长的模式3。

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。

2.9 词法分析器的开发

需要定义的词法组成

TOKEN结构图



Thank you!
