第三章：词法分析

提纲

词法分析流程

- 逐个读入源程序字符并按照**构词规则**切分成一系列**单词(token)**。
- 单词是语言中具有独立意义的最小单位，包括保留关键字、标识符、常量、运算符、标点符号、分界符等。

例

```
int main()
{
        int i,j,t,a[11];
        printf("请输入10个数：\n");
        for(i=1;i<11;i++)
            scanf("%d",&a[i]);
          for(i=1;i<=9;i++)
              for (j=i+1;j<=10;j++)
                  if(a[i]>a[j]) {
                      t=a[i];
                      a[i]=a[j];
                      a[j]=t;
                  }
        return 0;
}
```

token序列

< INT， - >

< IDN， main >

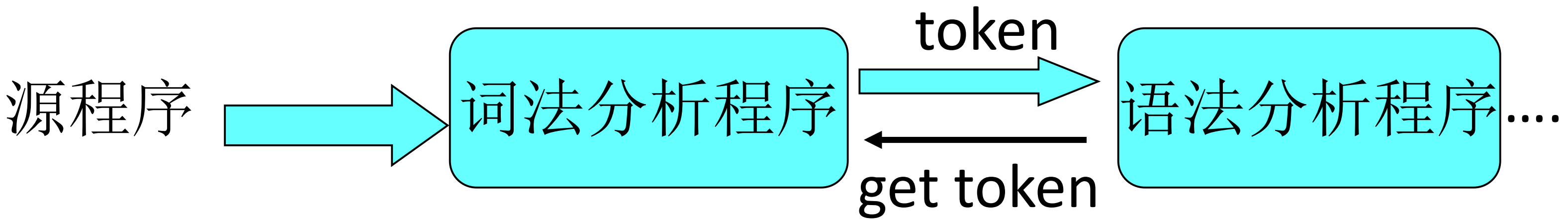
< SLP， - >

< SRP， - >

……

词法分析程序和语法分析程序的接口方式

- 词法分析是编译过程中的一个阶段，在语法分析前进行。也可和语法分析结合在一起作为一遍，由语法分析程序调用词法分析程序来获得当前单词供语法分析使用。



词法分析程序的主要任务及输出

- 读源程序，产生用二元组表示的单词符号：

  <单词种别，单词自身的值>

- 滤掉空格，跳过注释、换行符
- 记录源程序的行号，以便出错处理程序准确定位源程序的错误
- 宏展开等……

提纲

3.1 词法分析程序的设计
3.2 正则表达式
3.3 正则定义
3.4 有穷自动机 (***Finite Automata***)
3.5 有穷自动机的分类
3.6 从正则表达式到有穷自动机
3.7 从NFA到DFA的转换
3.8 识别单词的DFA

正则表达式

语言$L=\{a\}\{a,b\}^*(\{\varepsilon\}\cup(\{.,_\}\{a,b\}\{a,b\}^*))$

- 正则表达式(*Regular Expression*，*RE*)是一种用来描述正则语言的更紧凑的表示方法
  - 例：$r = a(a|b)^*(\varepsilon\,|\,(.|\_)(a|b)(a|b)^*)$
- 正则表达式可以由较小的正则表达式按照特定规则递归地构建。每个正则表达式r定义（表示）一个语言，记为$L(r)$。这个语言也是根据r的子表达式所表示的语言递归定义的

正则表达式的定义

- ε是一个RE，$L(\varepsilon) = \{\varepsilon\}$
- 如果 $a\in\sum$，则a是一个RE，$L(a) = \{a\}$
- 假设 r和 s都是 RE，表示的语言分别是 $L(r)$和$L(s)$，则
  - $r|s$ 是一个RE，$L(\, r|s\,) = L(r) \cup L(s)$
  - rs 是一个RE，$L(\, rs\,) = L(r)\, L(s)$
  - r^* 是一个RE，$L(\, r^*\,)= (L(r))^*$
  - (r) 是一个RE，$L(\, (r)\,) = L(r)$

运算的优先级：*、连接、|

例

- 令 $\sum = \{a, b\}$，则
  - $L(a|b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$
  - $L((a|b)(a|b)) = L(a|b)\ L(a|b) = \{a, b\}\{a, b\} = \{ aa, ab, ba, bb \}$
  - $L(a^*) = (L(a))^* = \{a\}^* = \{ \varepsilon, a, aa, aaa, \ldots \}$
  - $L((a|b)^*) = (L(a|b))^* = \{a, b\}^* = \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$
  - $L(a|a^*b) = \{ a, b, ab, aab, aaab, \ldots\}$

例：*C*语言无符号整数的*RE*

- 十进制整数的*RE*
  - $(1|...|9)(0|...|9)^*|0$
- 八进制整数的*RE*
  - $0(1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)^*$
- 十六进制整数的*RE*
  - $0x(1|...|9|a|...|f|A|...|F)(0|...|9|a|...|f|A|...|F)^*$

正则语言

- 可以用*RE*定义的语言叫做

  正则语言(*regular language*)或**正则集合**(*regular set*)

RE的代数定律

定律	描述
$r \mid s = s \mid r$	$\mid$是可以交换的
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$是可结合的
$r(st) = (rs)t$	连接是可结合的
$r(s \mid t) = rs \mid rt$； $(s \mid t)r = sr \mid tr$	连接对$\mid$是可分配的
$\varepsilon r = r\varepsilon = r$	ε是连接的单位元
$r^{*} = (r \mid \varepsilon)^{*}$	闭包中一定包含ε
$r^{**} = r^{*}$	*具有幂等性

正则文法与正则表达式等价

- 对任何正则文法 G，存在定义同一语言的正则表达式 r
- 对任何正则表达式 r，存在生成同一语言的正则文法 G

正则语言可以由正则文法定义

也可以由正则表达式定义

正则文法与正则表达式等价转换

- 正则文法(*Regular Grammar, RG*)
  - 右线性(*Right Linear*)文法： $A\to wB$ 或 $A\to w$
  - 左线性(*Left Linear*) 文法： $A\to Bw$ 或 $A\to w$

正则表达式转换成正则文法

将Σ上的一个正则表达式*r*转换成文法$G=(V_N,V_T,S,P)$，令$V_T=\Sigma$，然后确定P和V_N。选定一个非终结 符S定为开始符号，生成规则$S\to r$：

- 对于正则表达式xy：定义一个规则$A\to xy$，然后改写成：$A\to xB$, $B\to y$，$A,B\in V_N$
- 对于正则表达式x*y：定义一个规则$A\to x^*y$，然后改写成：，$A\to xB|y$, $B\to xB|y$，$A,B\in V_N$
- 对于正则表达式x|y：定义一个规则$A\to x|y$, $A\in V_N$

例：分别将正规式ab，a|b，a*, a (a|b)*， 其中$a,b\in V_T$转换成正则文法

正则文法与正则表达式的等价性

➢ 正则文法转换成正则表达式

将Σ上的正则文法G= (V_N,V_T, S, P)转换成正则表达式r，令Σ = V_T ,

r=S；

- 对于规则A→xB, B→y：定义一个正则表达式A=xy,
- 对于规则A→xA|y：定义一个正则表达式A=x*y
- 对于规则A→x|y：定义一个正则表达式A=x|y

例:对于文法G[S]

S→aA|a

A→aA|dA|a|d 求对应的正则表达式

提纲

正则定义（*Regular Definition*）

- 正则定义是具有如下形式的定义序列：

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

> 给一些*RE*命名，并在之后的*RE*中像使用字母表中的符号一样使用这些名字

其中：

- 每个d_i都是一个新符号，它们都不在字母表Σ中，而且各不相同
- 每个r_i是字母表$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$上的正则表达式

例1

- ***C*语言中标识符的正则定义**
  - $digit \rightarrow \mathbf{0|1|2|\ldots|9}$
  - $letter\_ \rightarrow A|B|\ldots|Z|a|b|\ldots|z|\_$
  - $id \rightarrow letter\_(letter\_|digit)^*$

例2

- （整型或浮点型）无符号数的正则定义
  - $digit \rightarrow 0|1|2|\ldots|9$
  - $digits \rightarrow digit\ digit^*$
  - $optionalFraction \rightarrow .digits|\varepsilon$
  - $optionalExponent \rightarrow (\ E(+|-|\varepsilon)digits\ )|\varepsilon$
  - $number \rightarrow digits\ \ optionalFraction\ \ optionalExponent$

2	2.15	2.15E+3	2.15E-3	2.15E3	2E-3

提纲

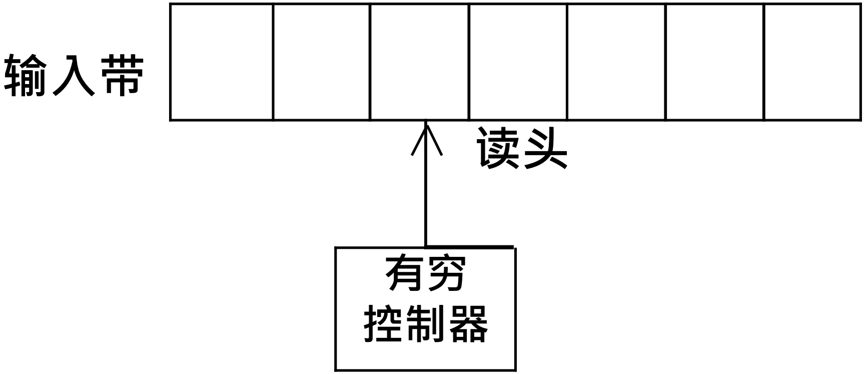
有穷自动机

- 有穷自动机（*Finite Automata*，*FA*）由两位神经物理学家*MeCuloch*和*Pitts*于1948年首先提出，是对一类处理系统建立的数学模型
- 这类系统具有一系列离散的输入输出信息和有穷数目的内部状态（状态：概括了对过去输入信息处理的状况）
- 系统只需要根据当前所处的状态和当前面临的输入信息就可以决定系统的后继行为。每当系统处理了当前的输入后，系统的内部状态也将发生改变

FA的典型例子

- 电梯控制装置
  - 输入：顾客的乘梯需求（所要到达的层号）
  - 状态：电梯所处的层数+运动方向
  - 电梯控制装置并不需要记住先前全部的服务要求，只需要知道电梯当前所处的状态以及还没有满足的所有服务请求

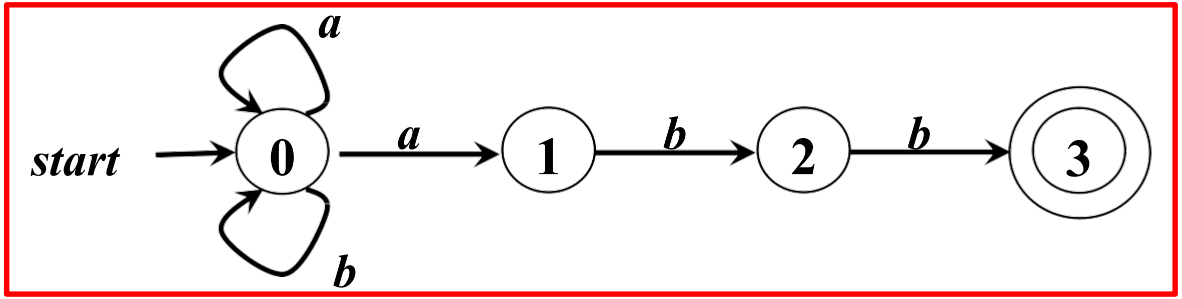
FA模型



- **输入带**(*input tape*)：用来存放输入符号串
- **读头**(*head*)：从左向右逐个读取输入符号，不能修改（只读）、不能往返移动
- **有穷控制器**(*finite control*)：具有有穷个状态数，根据当前的状态和当前输入符号控制转入下一状态

FA的表示

- 转换图 (*Transition Graph*)
  - 结点：*FA*的状态
    - 初始状态（开始状态）：只有一个，由*start*箭头指向
    - 终止状态（接收状态）：可以有多个，用双圈表示
  - 带标记的有向边：如果对于输入*a*，存在一个从状态*p*到状态*q*的转换，就在*p*、*q*之间画一条有向边，并标记上*a*

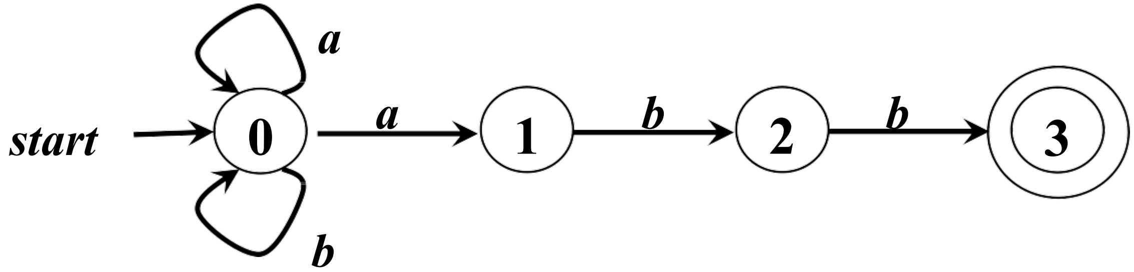


FA定义（接收）的语言

➢ 给定输入串x，如果存在一个对应于串x的从初始状态到某个终止状态的转换序列，则称符号串x被该*FA*接收

➢ 由一个有穷自动机M接收的所有符号串构成的集合称为是该*FA*定义（或接收）的语言，记为$L(M)$

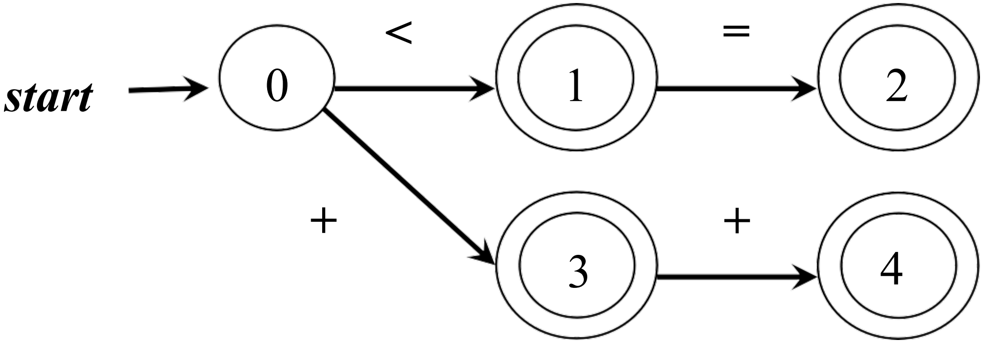
abbaabb



$L(M)$ =所有以*abb*结尾的字母表$\{a, b\}$上的符号串的集合

最长子串匹配原则(*Longest String Matching Principle*)

➢当输入串的多个前缀与一个或多个模式匹配时，总是选择最长的前缀进行匹配



➢在到达某个终态之后，只要输入带上还有符号，*DFA*就继续前进，以便寻找尽可能长的匹配

提纲

*FA*的分类

- 确定的*FA* (*Deterministic finite automata, DFA*)
- 非确定的*FA* (*Nondeterministic finite automata, NFA*)

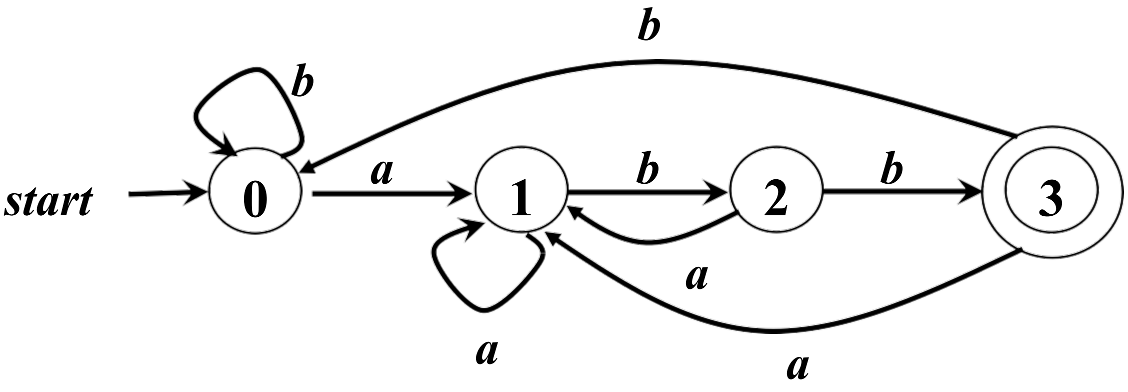
确定的有穷自动机（*DFA*）

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

- S：有穷状态集
- Σ：输入字母表，即输入符号集合。假设ε不是Σ中的元素
- δ：将$S\times\Sigma$映射到S的转换函数。$\forall s\in S, a\in\Sigma, \delta(s,a)$表示从状态$s$出发，沿着标记为$a$的边所能到达的状态。
- s_0：开始状态（或初始状态），$s_0\in S$
- F：接收状态（或终止状态）集合，$F\subseteq S$

例：一个*DFA*

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



转换表

状态 \ 输入	*a*	*b*
0	1	0
1	1	2
2	1	3
3	1	0

可以用转换表表示*DFA*

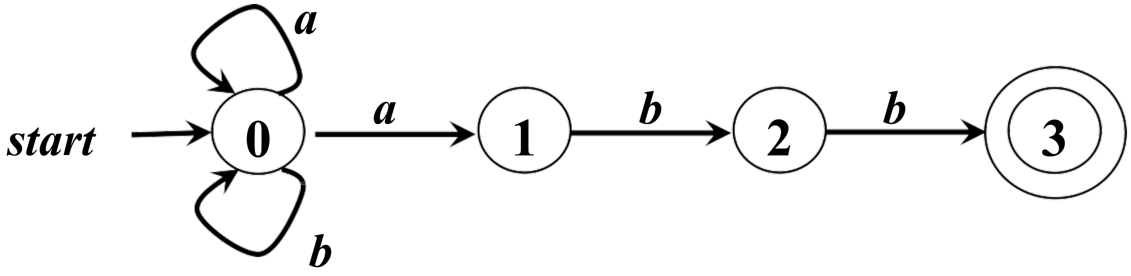
非确定的有穷自动机(*NFA*)

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

- S：有穷状态集
- Σ：输入符号集合，即输入字母表。假设ε不是Σ中的元素
- δ：将$S\times\Sigma$映射到2^S的转换函数。$\forall s\in S, a\in\Sigma$, $\delta(s,a)$表示从状态s出发，沿着标记为a的边所能到达的状态集合
- s_0：开始状态 (或初始状态)，$s_0\in S$
- F：接收状态 (或终止状态) 集合，$F\subseteq S$

例：一个 *NFA*

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



转换表

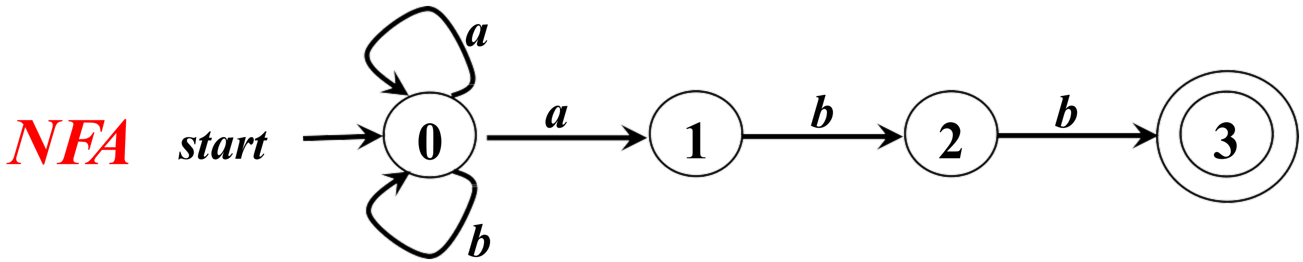
状态 \ 输入	*a*	*b*
0	{ 0,1 }	{ 0 }
1	Ø	{ 2 }
2	Ø	{ 3 }
3 •	Ø	Ø

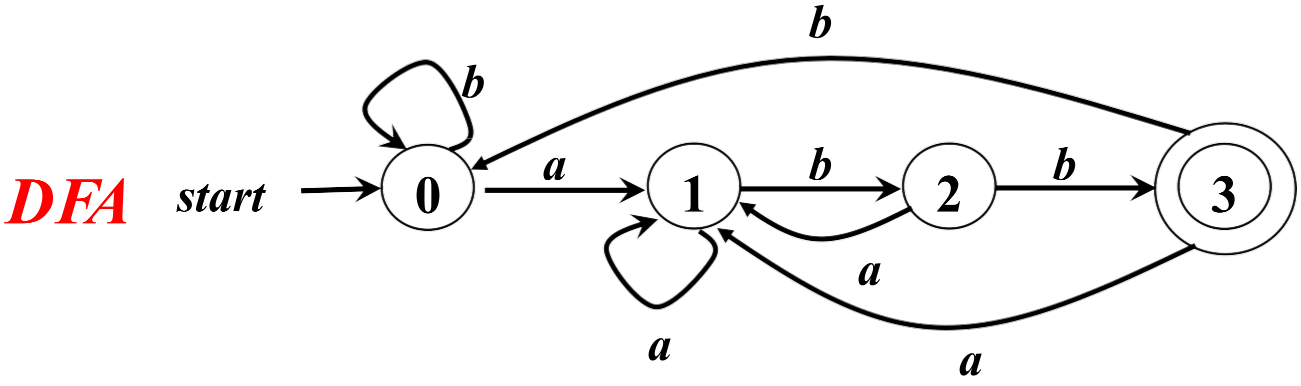
如果转换函数没有给出对应于某个状态-输入对的信息，就把Ø放入相应的表项中

DFA和NFA的等价性

- 对任何NFA N，存在识别同一语言的DFA D
- 对任何DFA D，存在识别同一语言的NFA N

例





状态1：串以a结尾

状态2：串以ab结尾

状态3：串以abb结尾

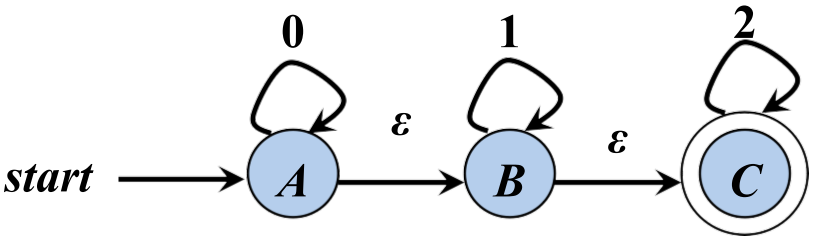
$r = (a|b)^{*}abb$

正则文法⇔正则表达式 ⇔ FA

带有“ε-边”的NFA

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

- ➢ S：有穷状态集
- ➢ Σ：输入符号集合，即输入字母表。假设ε不是Σ中的元素
- ➢ δ：将$S\times(\Sigma\cup\{\varepsilon\})$映射到$2^S$的转换函数。$\forall s\in S, a\in\Sigma\cup\{\varepsilon\}, \delta(s,a)$表示从状态$s$出发，沿着标记为$a$的边所能到达的状态集合
- ➢ s_0：开始状态(或初始状态)，$s_0\in S$
- ➢ F：接收状态(或终止状态)集合，$F\subseteq S$

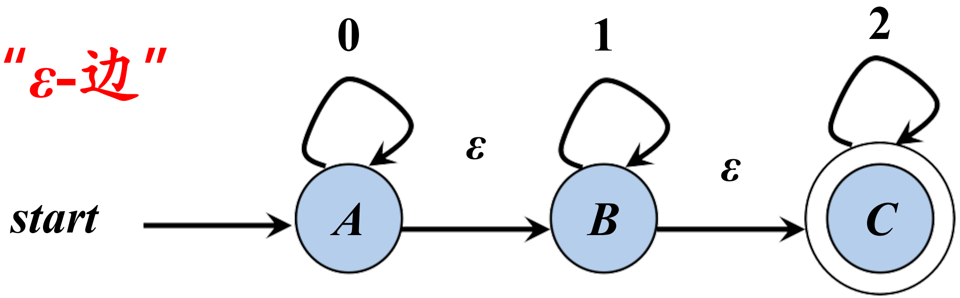


$r = 0^*1^*2^*$

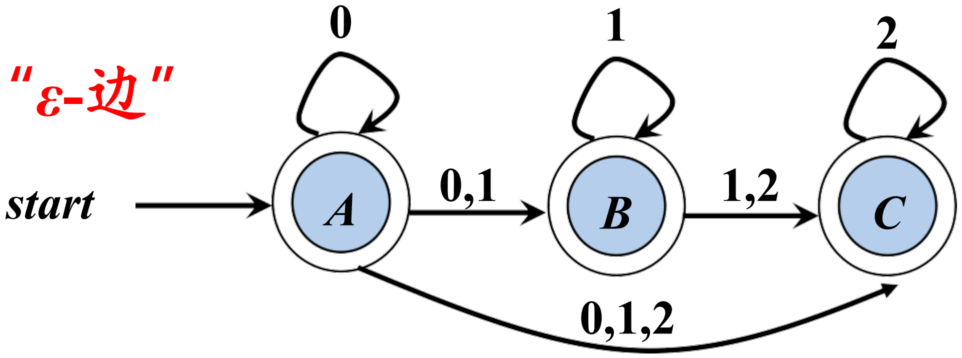
带有和不带有“ε-边”的NFA的等价性

➢例

带“ε-边”



不带“ε-边”



$r = 0^*1^*2^*$

状态A：0^*

状态B：0^*1^*

状态C：$0^*1^*2^*$

*DFA*的算法实现

- 输入：以文件结束符eof结尾的字符串*x*。*DFA D* 的开始状态s_0，接收状态集 *F*，转换函数*move*。
- 输出：如果 *D*接收 *x*，则回答"*yes*"，否则回答"*no*"。
- 方法：将下述算法应用于输入串 *x*。

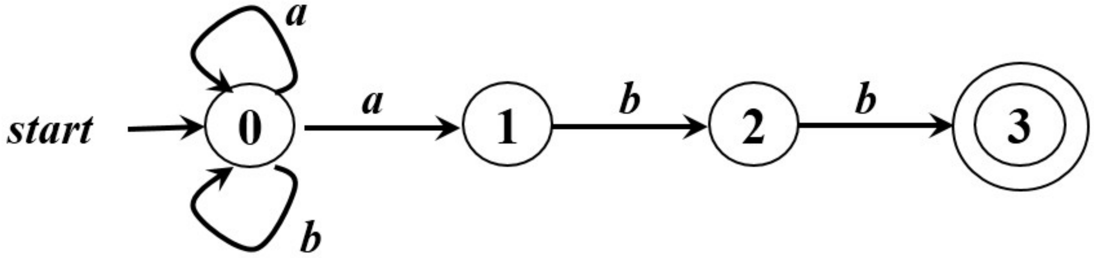
```
s = s0 ;
c = nextChar ( ) ;
while ( c! = eof  ) {
        s = move ( s , c ) ;
        c = nextChar ( ) ;
 }
if (s在F中) return "yes" ;
else  return  "  no  "  ;
```

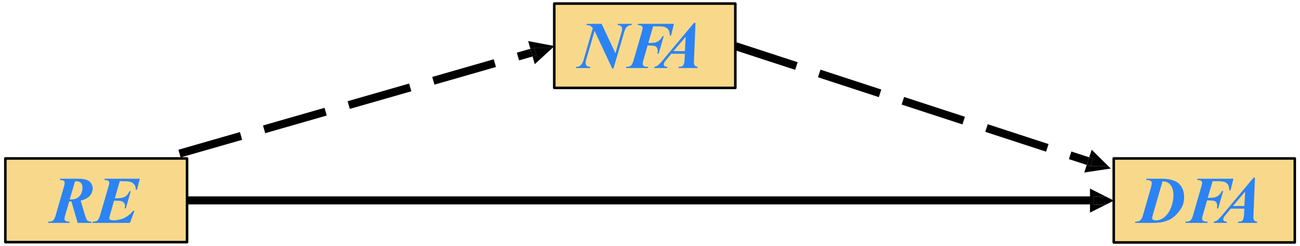
- 函数*nextChar*( )返回输入串*x*的下一个符号
- 函数move(s, c)表示从状态s出发，沿着标记为c的边所能到达的状态

提纲

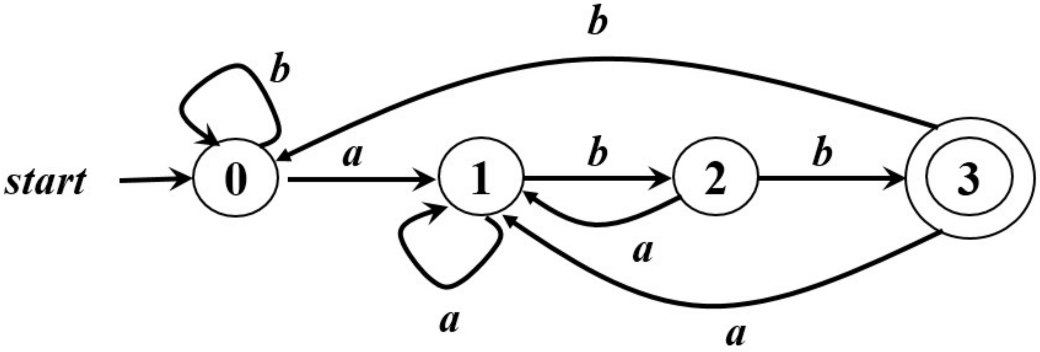
3.1 词法分析程序的设计
3.2 正则表达式
3.3 正则定义
3.4 有穷自动机 (***Finite Automata***)
3.5 有穷自动机的分类
3.6 从正则表达式到有穷自动机
3.7 从NFA到DFA的转换
3.8 识别单词的DFA

从正则表达式到有穷自动机



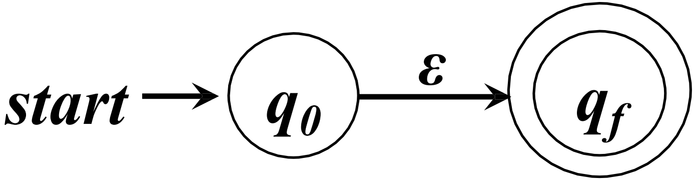


$r = (a|b)^{*}abb$

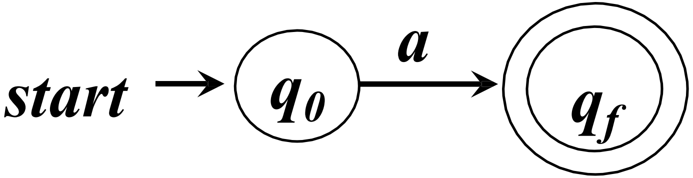


根据*RE*构造*NFA*

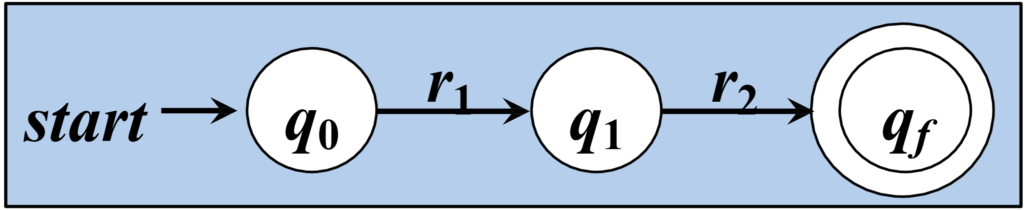
- ε对应的*NFA*



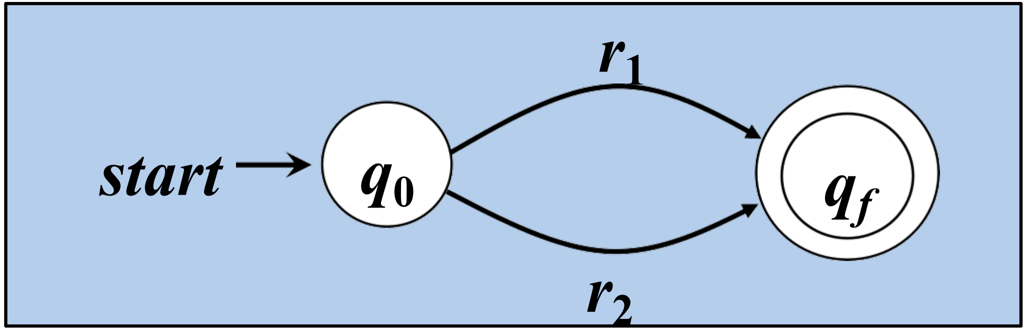
- 字母表Σ中符号a对应的*NFA*



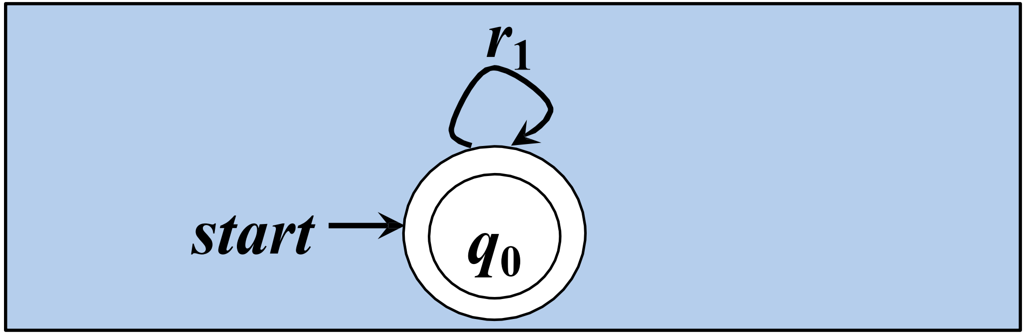
- $r = r_1 r_2$对应的NFA



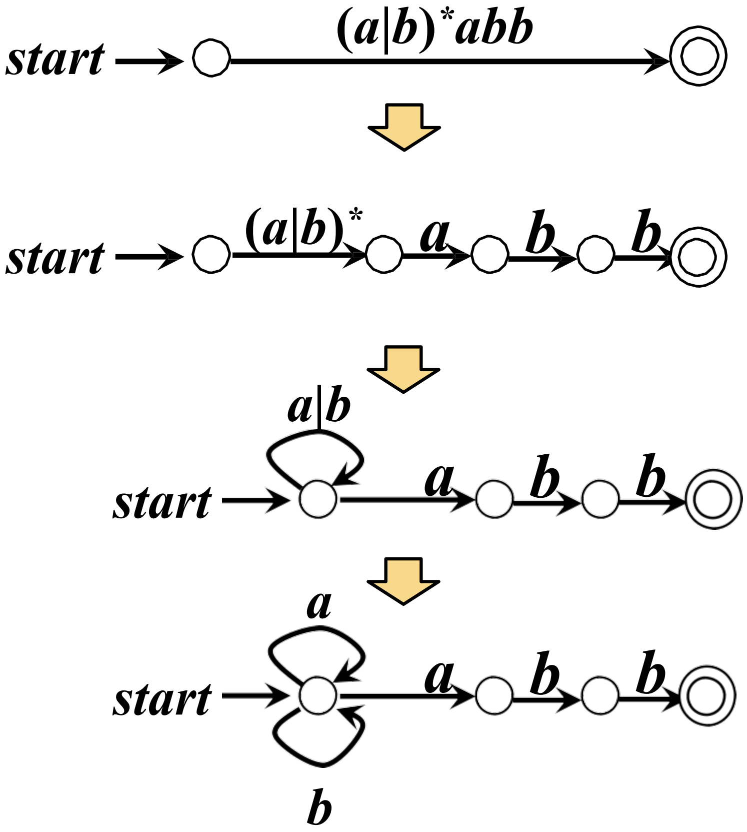
- $r = r_1 | r_2$对应的NFA



- $r = (r_1)^*$对应的NFA



例：$r=(a|b)^*abb$ 对应的NFA



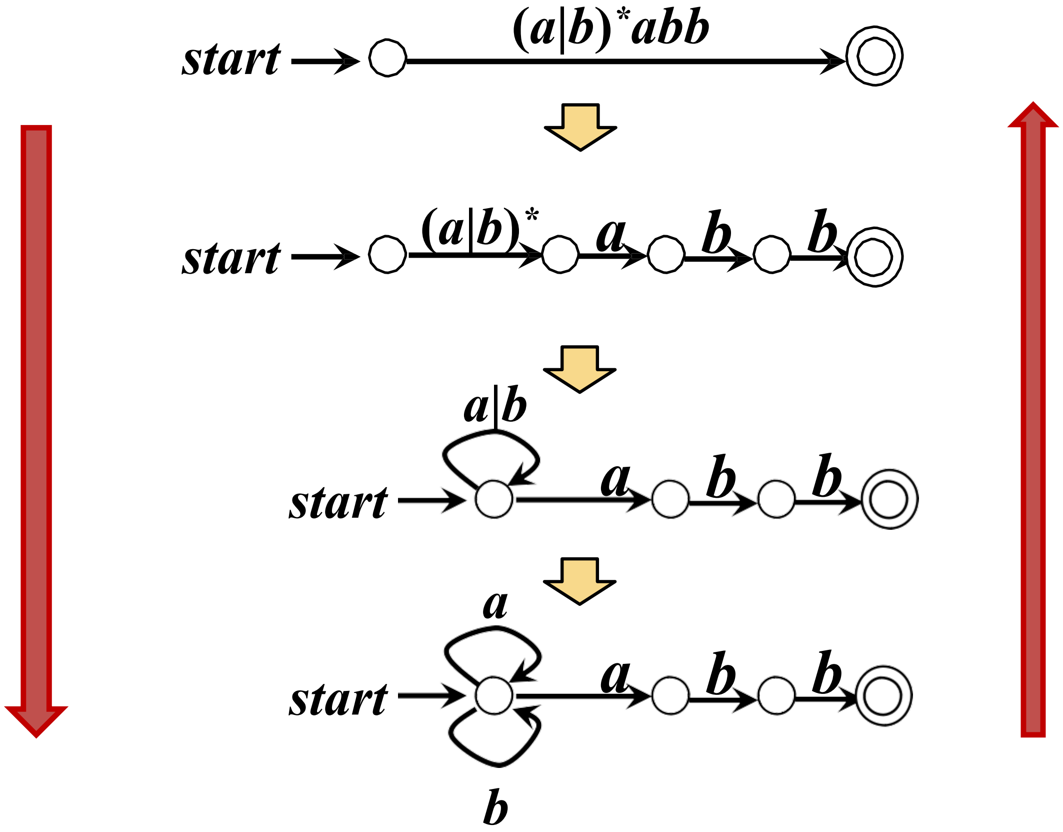
正则表达式与有穷自动机的等价性

- 对于∑上的一个*NFA* M，可以构造一个∑上的正则表达式*r*,使得L(*r*)=L(M)。

- 对于∑上的一个正则表达式*r*，可以构造一个∑上的*NFA* M，使的L(M)=L(*r*)。

词法分析程序的自动构造基于*FA*和*RE*的等价性。

例：$r=(a|b)^*abb$ 对应的 NFA



课堂练习：
为下面正则表达式构造 NFA。
$(a|b)^*(aa|bb)(a|b)^*$

提纲

从*NFA*到*DFA*的转换

- ***DFA*是*NFA*的特例**

  对每个*NFA* **N**一定存在一个*DFA* **M**，使得*L*(*M*)=*L*(*N*)

- **将*NFA*转换成接受同样语言的*DFA*的方法--子集法**

  与某一*NFA*等价的*DFA*不唯一

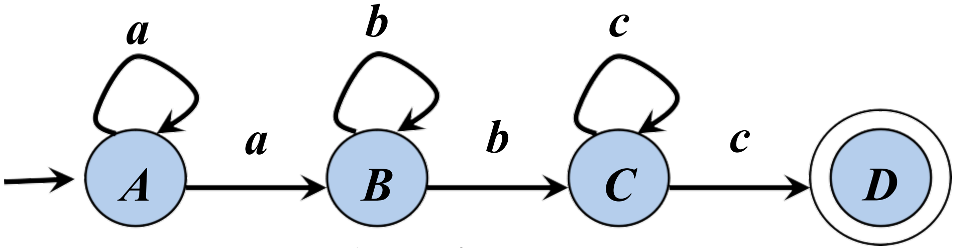
子集法

设$NFA\ N=(K, \sum, f, K_0, K_t)$，按如下办法构造一个$DFA\ \ M=(S, \sum, d, S_0, S_t)$，使得$L(M)=L(N)$：

- 状态集S：$\forall s \in S$，s由K的子集组成，用$[S_1\, S_2 \ldots S_j]$表示
- 输入字母$\sum$：与N的输入字母表相同
- 转换函数：$d([S_1\, S_2 \ldots S_j],a)= [R_1\, R_2 \ldots R_t]$ 其中$\{R_1\, R_2 \ldots R_t\}$ $=\ \varepsilon\text{-closure}(\text{move}(\{S_1\, S_2 \ldots S_j\},a))$
- 开始状态：$S_0=\varepsilon\text{-closure}(K_0)$
- 接收状态：$S_t=\{[S_i\, S_k \ldots S_e]$，其中$[S_i\ \ S_k \ldots S_e] \in S$且$\{S_i, S_{k,} \ldots S_e\} \cap K_t \neq \Phi\}$

从NFA到DFA的转换

➢例1 NFA：



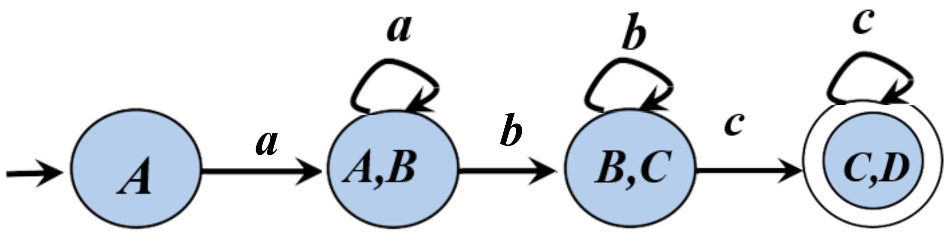
转换表

状态 \ 输入	a	b	c
A	$\{A,B\}$	Ø	Ø
B	Ø	$\{B,C\}$	Ø
C	Ø	Ø	$\{C,D\}$
D	Ø	Ø	Ø

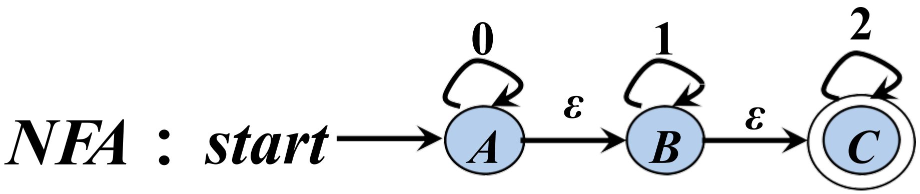
DFA的每个状态都是一个由NFA中的状态构成的集合,即NFA状态集合的一个子集

$r = aa^*bb^*cc^*$

DFA：



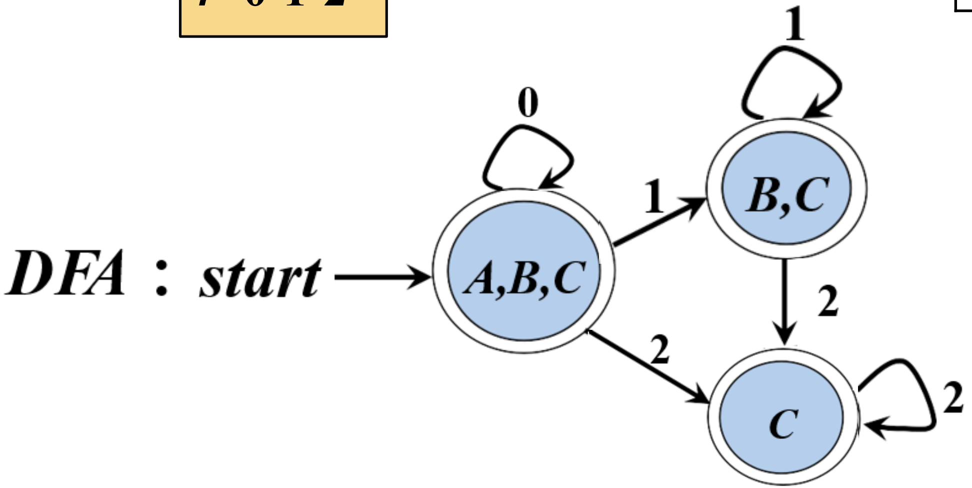
例2：从带有ε-边的NFA到DFA的转换



转换表

输入 / 状态	0	1	2
A	$\{A,B,C\}$	$\{B,C\}$	$\{C\}$
B	$\emptyset$	$\{B,C\}$	$\{C\}$
C	$\emptyset$	$\emptyset$	$\{C\}$

$r=0^*1^*2^*$



子集构造法（ *subset construction* ）

- 输入：*NFA* **N**
- 输出：接收同样语言的*DFA* **D**
- 方法：一开始，*ε-closure*(s_0)是*Dstates* 中的唯一状态，且它未加标记;

```
while（在Dstates中有一个未标记状态T）{
    给T加上标记；
    for（每个输入符号a）{
        U = ε-closure(move(T, a)); if (
        U不在Dstates中)
            将U加入到Dstates中，且不加标记；
        Dtran[T, a]=U；
} }
```

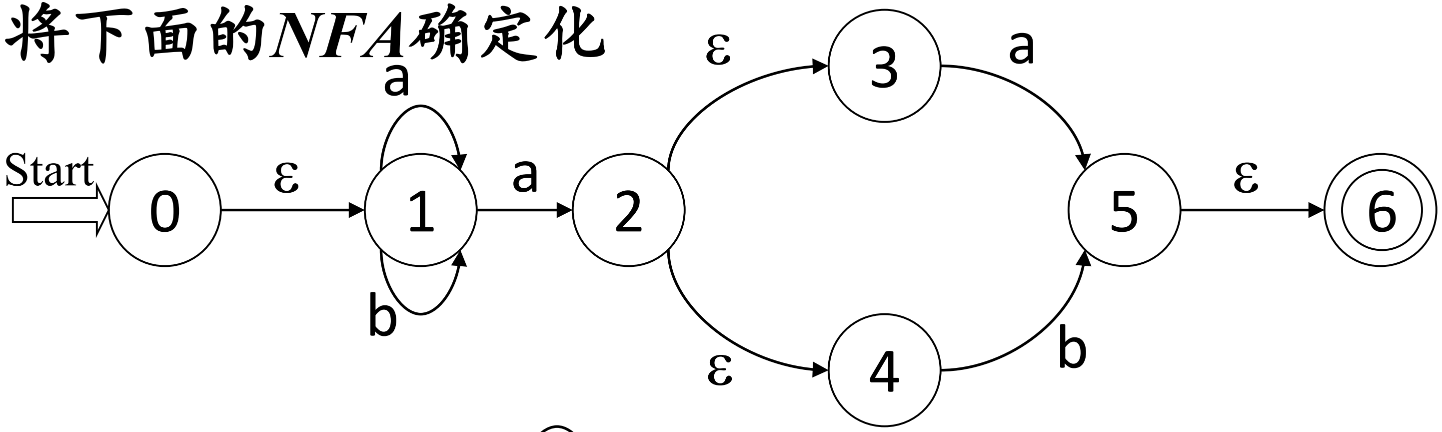
操作	描述
ε-closure (*s*)	能够从NFA的状态s开始只通过ε转换到达的NFA状态集合
ε-closure (*T*)	能够从T 中的某个NFA状态 s开始只通过ε转换到达的NFA状态集合, 即 $U_{s\in T}$ *ε-closure* (*s*)
move(*T* , *a*)	能够从T 中的某个状态 s出发通过标号为a的转换到达的NFA状态的集合

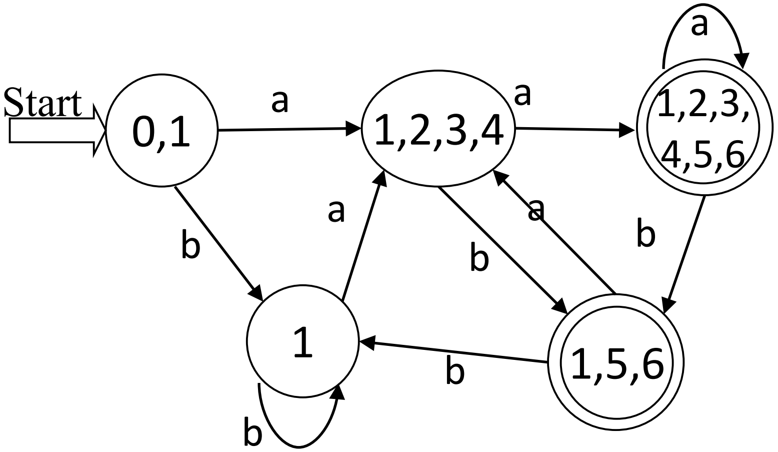
计算 ε-*closure* (T)

```
将T的所有状态压入stack中；
将ε-closure( T )初始化为T；
while ( stack非空 ){
        将栈顶元素 t 给弹出栈中；
        for ( 每个满足如下条件的u ：从t出发有一个标号为ε的转换到达状态u)
                if ( u不在ε-closure( T )中 ){
                        将u加入到ε-closure( T )中；
                        将u压入栈中;
                }
}
```

课堂练习

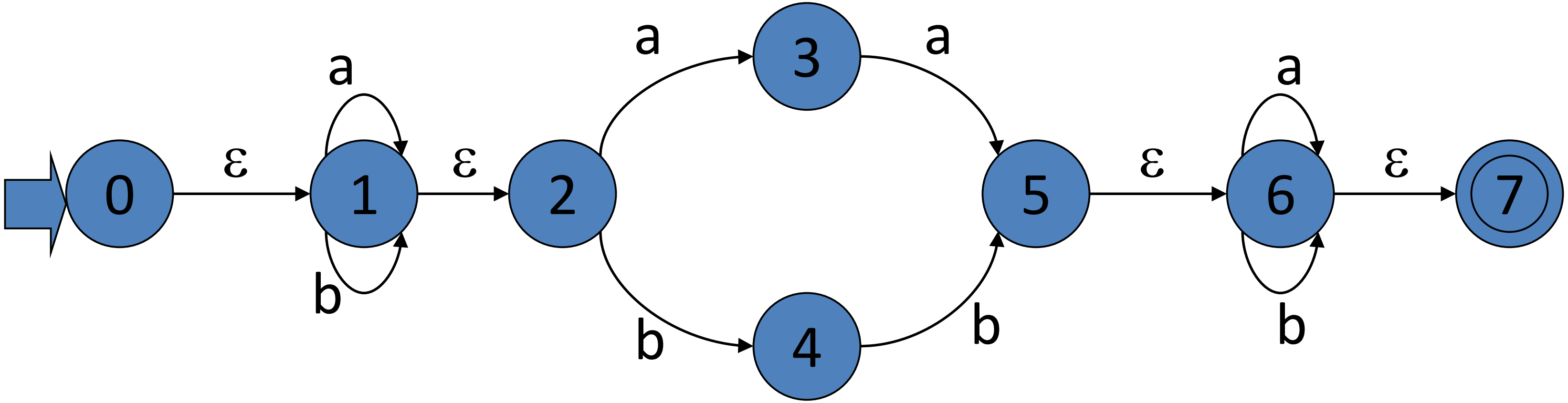
将下面的*NFA*确定化





作业

将下面的NFA确定化



NFA 和*DFA*的区别

	NFA	*DFA*
初始状态	不唯一	唯一
弧上的标记	字符、ε	字符
转换关系	不确定	确定

*DFA*的化简

*DFA*化简：通过消除无用状态和合并等价状态而转换成一个最小的与之等价的有穷自动机。

- 多余状态：从开始状态出发，任何输入串也不能到达的那个状态，或者从这个状态没有通路到达终态。

例

输入 状态	0	1
s0	*s1*	*s5*
s1	*s2*	*s7*
s2	*s2*	*s5*
s3	*s5*	*s7*
s4	*s5*	*s6*
s5	*s3*	*s1*
s6	*s8*	*s0*
s7	*s0*	*s1*
s8	*s3*	*s6*

输入 状态	0	1
s0	*s1*	*s5*
s1	*s2*	*s7*
s2	*s2*	*s5*
s3	*s5*	*s7*
s5	*s3*	*s1*
s6	*s8*	*s0*
s7	*s0*	*s1*
s8	*s3*	*s6*

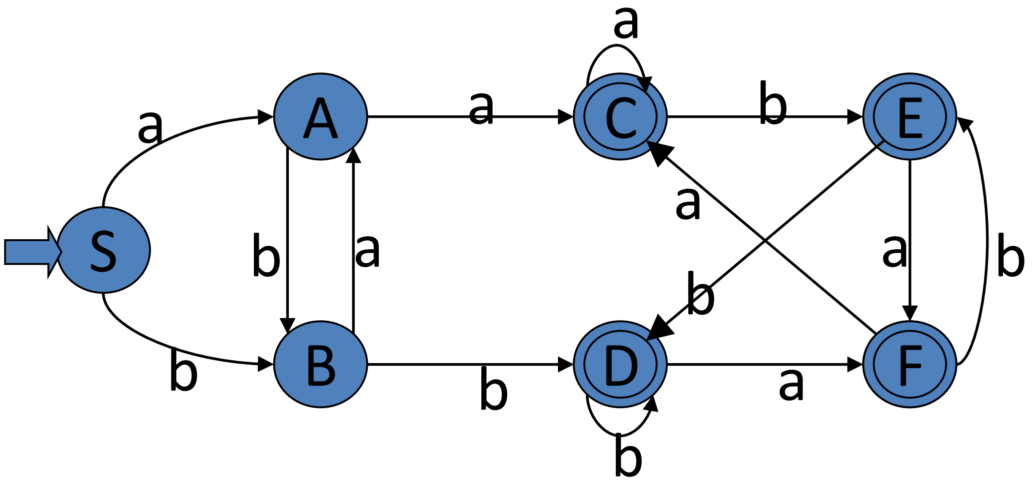
输入 状态	0	1
s0	*s1*	*s5*
s1	*s2*	*s7*
s2	*s2*	*s5*
s3	*s5*	*s7*
s5	*s3*	*s1*
s7	*s0*	*s1*

DFA的化简

*DFA*化简：通过消除无用状态和合并等价状态而转换成一个最小的与之等价的有穷自动机。

- 多余状态：从开始状态出发，任何输入串也不能到达的那个状态，或者从这个状态没有通路到达终态。
- 等价状态：*T1*和*T2*同是终态或同是非终态，且*T1*出发对任意一个读入符号$a(a \in \Sigma)$和从*T2*出发读入a到达的状态等价

例

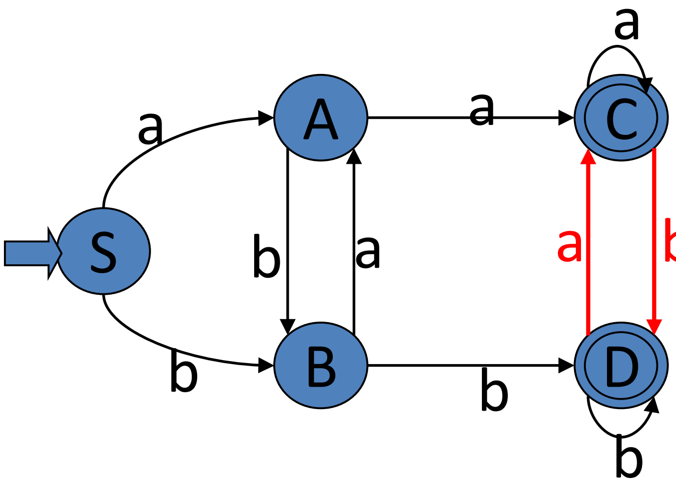


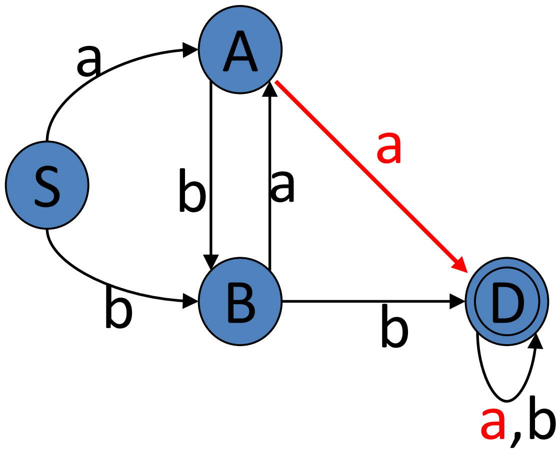
{S,A,B} {C,D,E,F}

{S} {A} {B} {C,D,E,F}

{S} {A} {B} {C,D}

{S} {A} {B} {D}

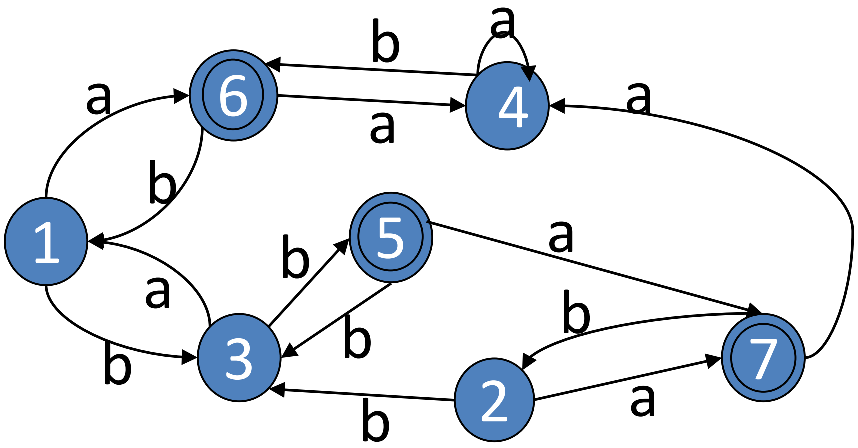




分割法

分割法：把一个*DFA*（不含多余状态）的状态分成一些不相交的子集，使得任何不同的两个子集的状态都是可区别的，而同一子集中的任何两个状态都是等价的。

例

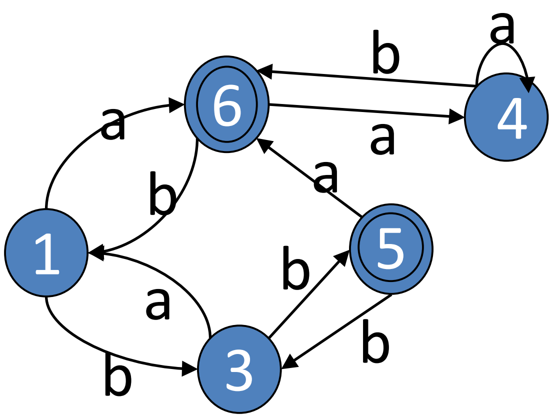


(1)初始化分P=({1,2,3,4} {5,6,7})

(2) P1=({1,2} {3,4} {5,6,7})

(3) P1=({1,2} {3} {4} {5,6,7})

(4) P1=({1,2} {3} {4} {5} {6,7})



最小状态*DFA*

最小状态DFA的定义：

- 没有多余状态(死状态)
- 没有等价状态(不可区别)

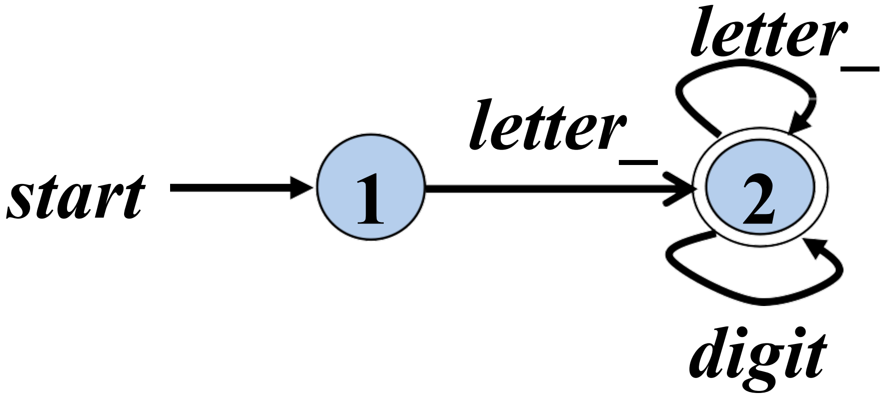
对于一个*DFA* $M=(K,\sum,f,k_0,k_t)$，存在一个最小状态*DFA* $M'=(K',\sum,f',K_0',K_t')$，使$L(M')=L(M)$

接受L的最小状态有穷自动机不计同构是唯一的。

提纲

识别标识符的*DFA*

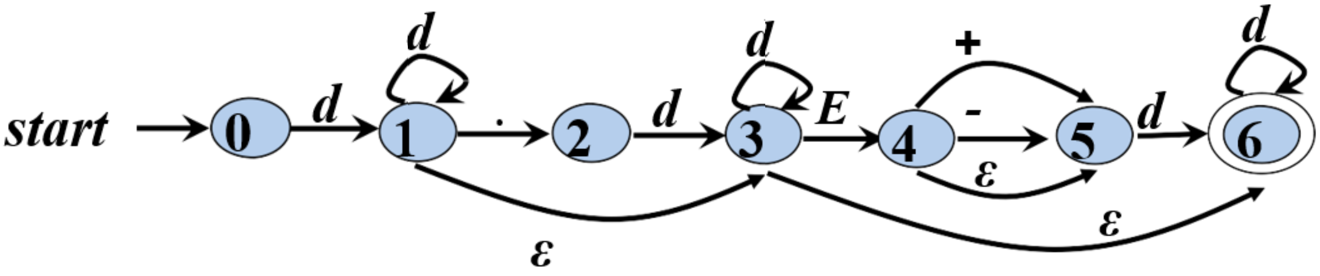
- 标识符的正则定义
  - $digit \rightarrow 0|1|2|\ldots|9$
  - $letter\_ \rightarrow A|B|\ldots|Z|a|b|\ldots|z|\_$
  - $id \rightarrow letter\_(letter\_|digit)^*$



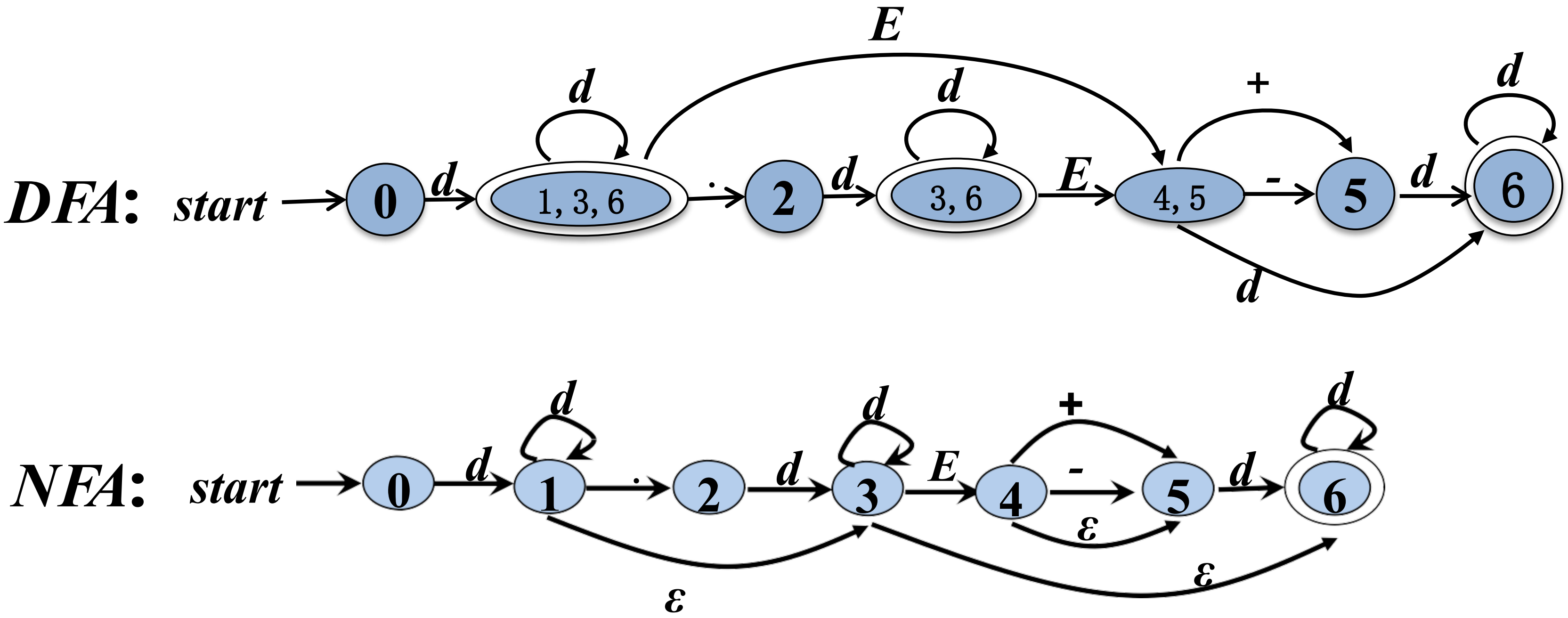
识别无符号数的*DFA*

- $digit \rightarrow 0|1|2|\ldots|9$
- $digits \rightarrow digit\ digit^*$
- $optionalFraction \rightarrow .digits|\varepsilon$
- $optionalExponent \rightarrow (\ E(+|-|\varepsilon)digits\ )|\varepsilon$
- $number \rightarrow digits\ \ optionalFraction\ \ optionalExponent$

NFA:

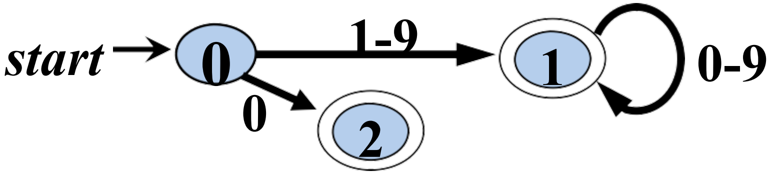


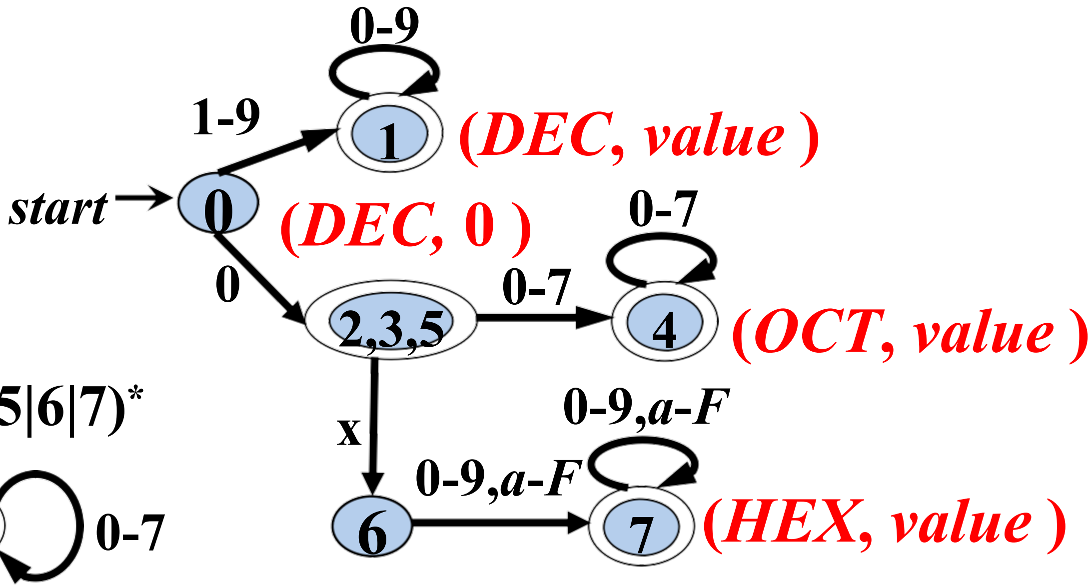
识别无符号数的*DFA*



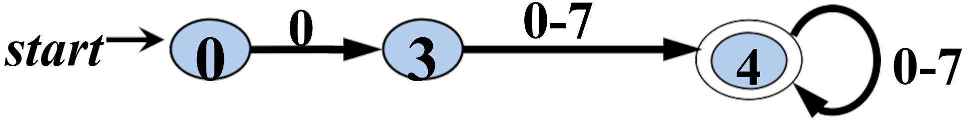
识别各进制无符号整数的*DFA*

$DEC \rightarrow (1|...|9)(0|...|9)^{*}\,|0$

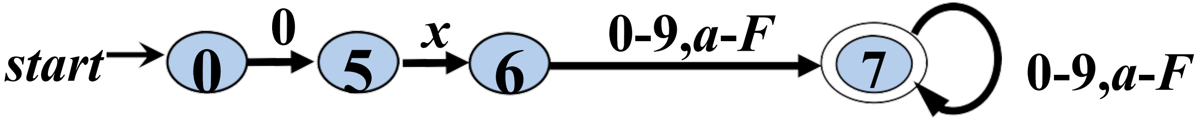




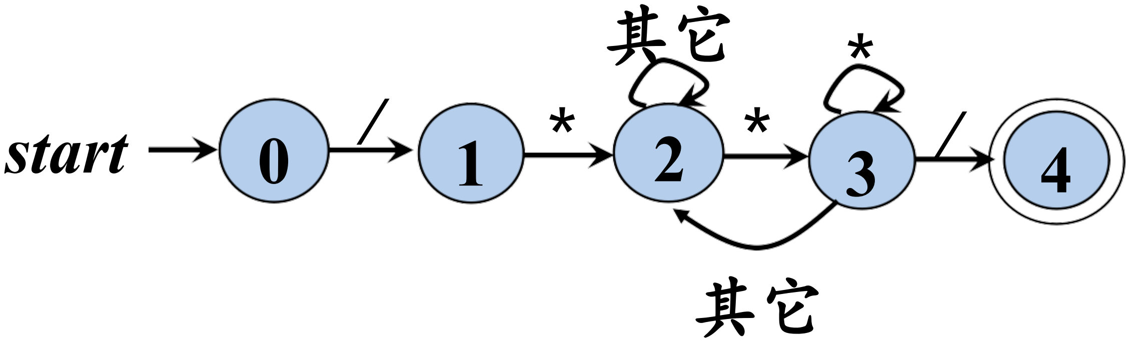
$OCT \rightarrow 0(1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)^{*}$



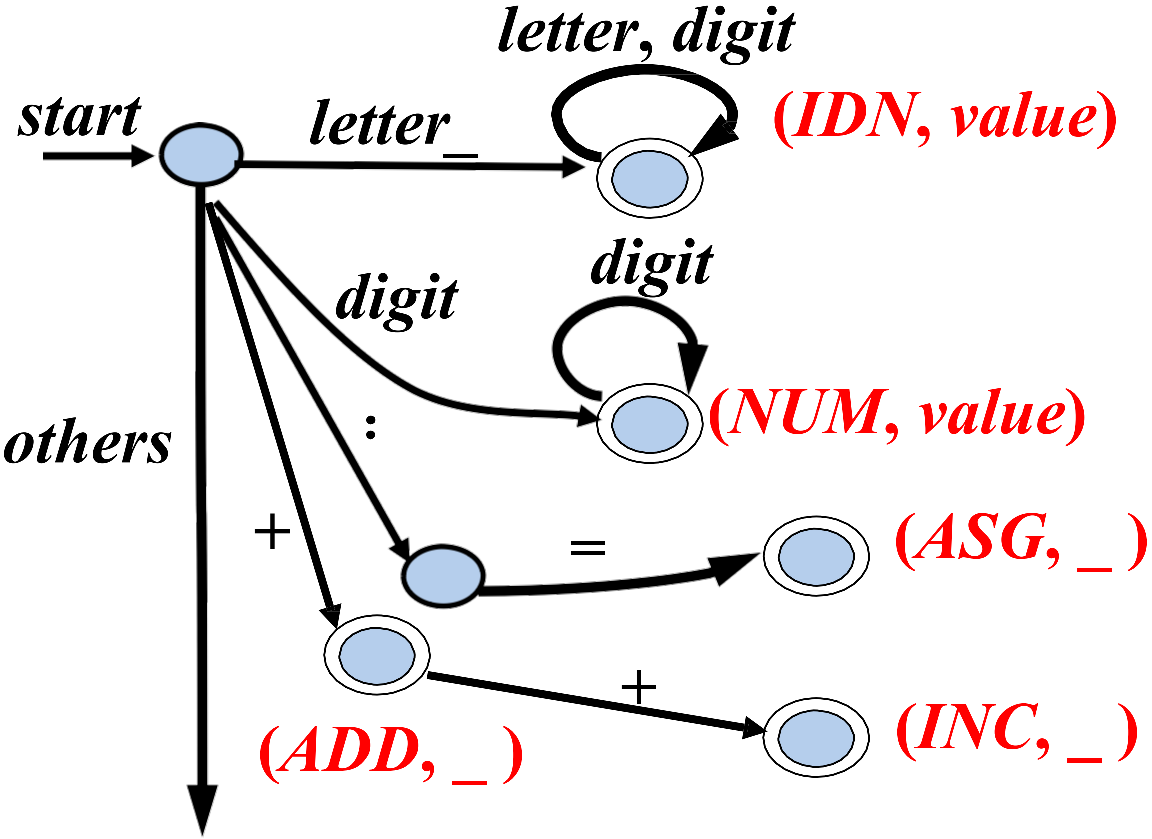
$HEX \rightarrow 0x(1|...|9|a|...|f|A|\ldots|F)(0|...|9|a|...|f\,|A|\ldots|F)^{*}$



识别注释的 *DFA*



识别 *Token*的*DFA*



词法分析阶段的错误处理

- 词法分析阶段可检测错误的类型
  - 单词拼写错误
    - 例：*int i* = 0*x*3*G*; *float j* =1.05*e*;
  - 非法字符
    - 例：~ @
- 词法错误检测
  - 如果当前状态与当前输入符号在转换表对应项中的信息为空，而当前状态又不是终止状态，则调用错误处理程序

错误处理

- 查找已扫描字符串中最后一个对应于某终态的字符
  - 如果找到了，将该字符与其前面的字符识别成一个单词。然后将输入指针退回到该字符，扫描器重新回到初始状态，继续识别下一个单词
  - 如果没找到，则确定出错，采用错误恢复策略

错误恢复策略

- 最简单的错误恢复策略：“恐慌模式(*panic mode*)”恢 复
  - 从剩余的输入中不断删除字符，直到词法分析器能够在剩余输入的开头发现一个正确的字符为止

本章小结

正则文法RG、正则表达式RE、有穷自动机FA

正则文法⇔正则表达式 ⇔ 有穷自动机

正规集、正则表达式、DFA

单词符号	种别编码	助忆符	内码值
DIM	1	$DIM	-
IF	2	$IF	-
DO	3	$DO	-
STOP	4	$STOP	-

```
curState = 初态
GetChar();
while( stateTrans[curState][ch]有定义){
    //存在后继状态，读入、拼接
    Concat();
    //转换入下一状态，读入下一字符
    curState= stateTrans[curState][ch];
    if curState是终态 then 返回strToken中的单词
    GetChar( );
}
```

DIM,IF, DO,STOP,END
number, name, age
125, 2169
…

构成

程序语言的单词集合—正规集

等价

正则定义—正则表达式

描述

DIM
IF
DO
STOP
END
letter(letter|digit)*
digit(digit)*

等价

FA

DFA

化简

DFA

等价

NFA

易于人工设计

词法分析器自动生成过程

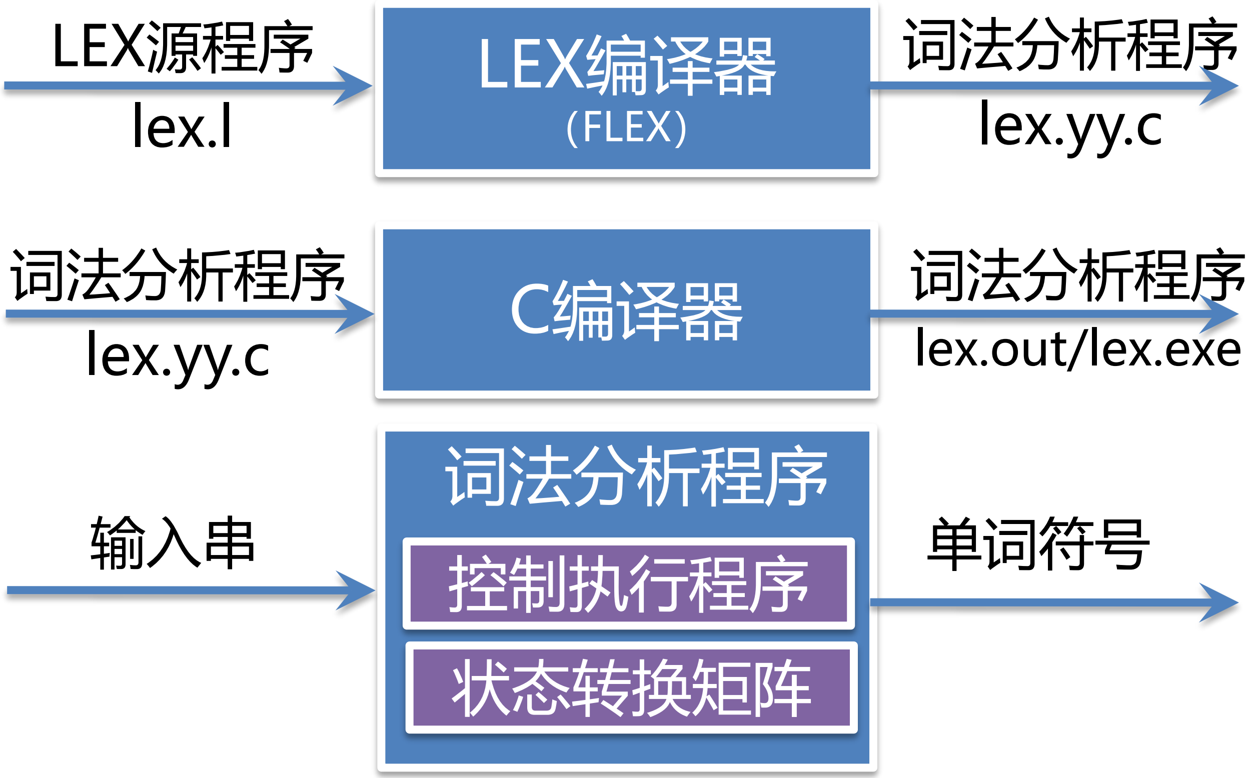
1.正则表达式→NFA（语法制导的构造算法）

2.NFA→DFA (子集构造法)

3.DFA化简 (分割法)

4.根据DFA构造词法分析器源码

词法分析器的自动产生-LEX



词法分析器的自动产生-LEX

AUXILIARY DEFINITION

letter→A|B|…|Z

digit→0|1|…|9

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
…
$d_n \rightarrow r_n$

给一些*RE*命名，并在之后的*RE*中像使用字母表中的符号一样使用这些名字

RECOGNITION RULES

1	DIM	{ RETURN (1,-) }
2	IF	{ RETURN (2,-) }
3	DO	{ RETURN (3,-) }
4	STOP	{ RETURN (4,-) }
5	END	{ RETURN (5,-) }
6	letter(letter|digit)*	{ RETURN (6, TOKEN) }
7	digit(digit)*	{ RETURN (7, DTB) }
8	=	{ RETURN (8, -) }
9	+	{ RETURN (9,-) }
10	*	{ RETURN (10,-) }
11	**	{ RETURN (11,-) }
12	,	{ RETURN (12,-) }
13	(	{ RETURN (13,-) }
14	)	{ RETURN (14,-) }

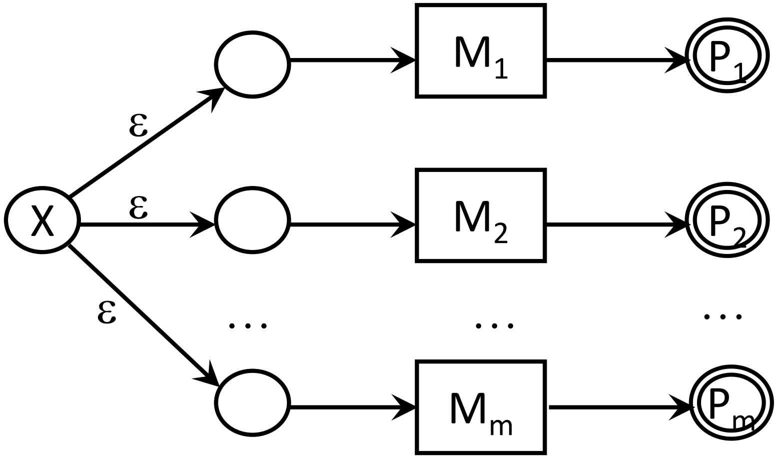
词法分析器的自动产生-LEX

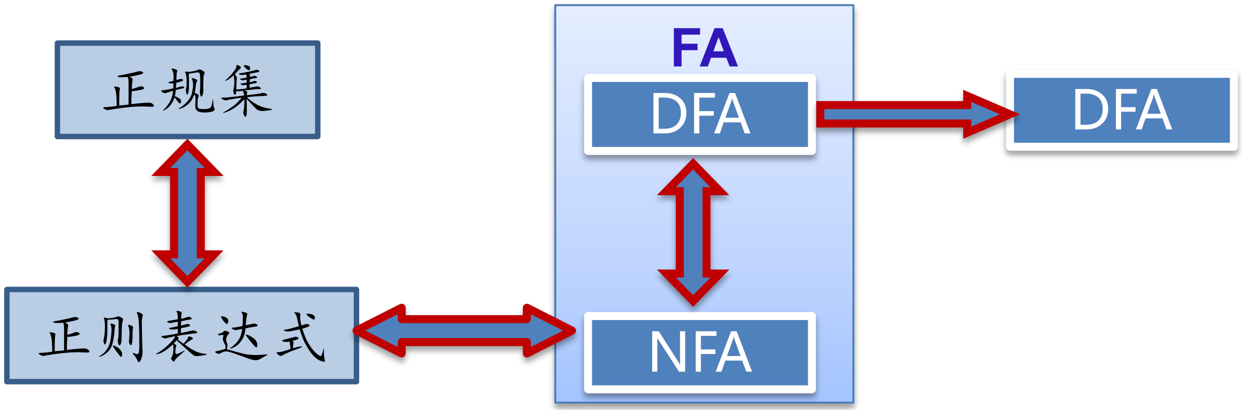
LEX的工作过程

对每条识别规则P_i构造一个相应的非确定有限自动机M_i；

引进一个新初态X，通过ε弧，将这些自动机连接成一个新的NFA；

把M确定化、最小化，生成该DFA的状态转换表和控制执行程序





写在最后

我听到的会忘掉，

我看到的能记住，

我做过的才真正明白。