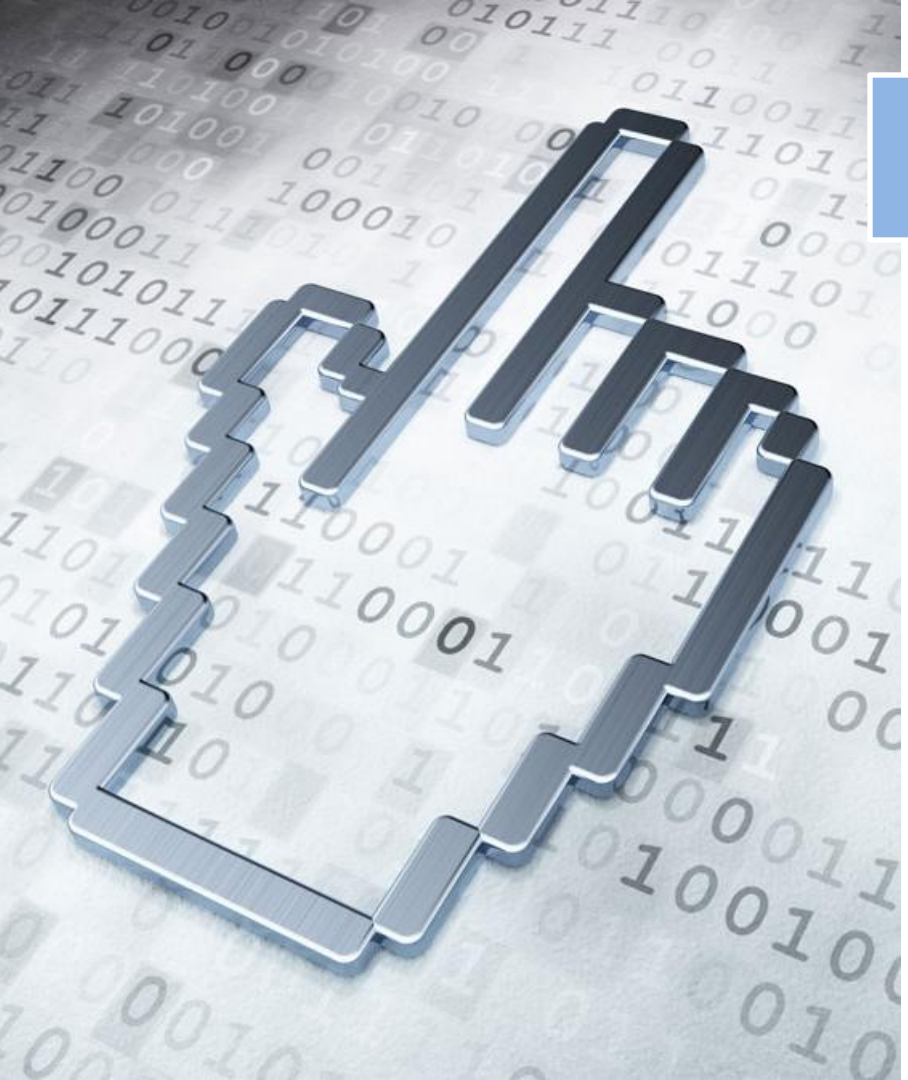
编译系统

第三章
词法分析

哈尔滨工业大学 陈鄞 单丽莉



本章内容

3.1 单词的描述

3.2 单词的识别

3.3 词法分析阶段的错误处理

3.4 词法分析器生成工具Lex

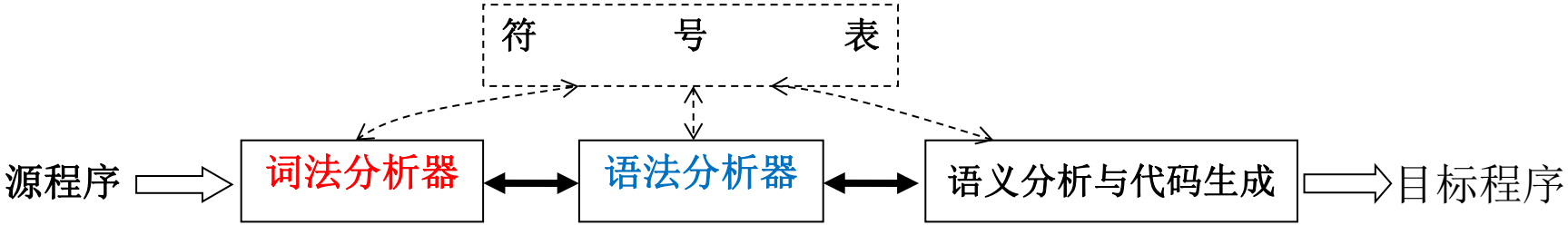
编译器的结构


字符流
词法分析器
词法单元流
语法分析器
语法树
语义分析器
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
目标代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

词法分析器(Scanner)

- 任务:
  - 从左到右扫描源程序，根据词法规则识别单词，并转换成统一表示的单词(token)串；同时，
    - 删掉空格字符和注释;
    - 对常数完成数字字符串到数值的转换;
    - 检查词法错误，记录行号报错;
    - 错误恢复;
- 输出：词法单元序列（token序列）
  - 词法单元的表示：二元组<符号名字，属性值>

词法分析器的位置



以语法分析器为中心

- 将词法分析器设计为单独的遍具有如下优点：
  - 简化编译器的设计：任务独立，语法分析更简易
  - 提高编译器的效率
  - 增强编译器的可移植性：只有词法分析与输入设备有关。

词法单元符号设计

➢ 词法单元 token：

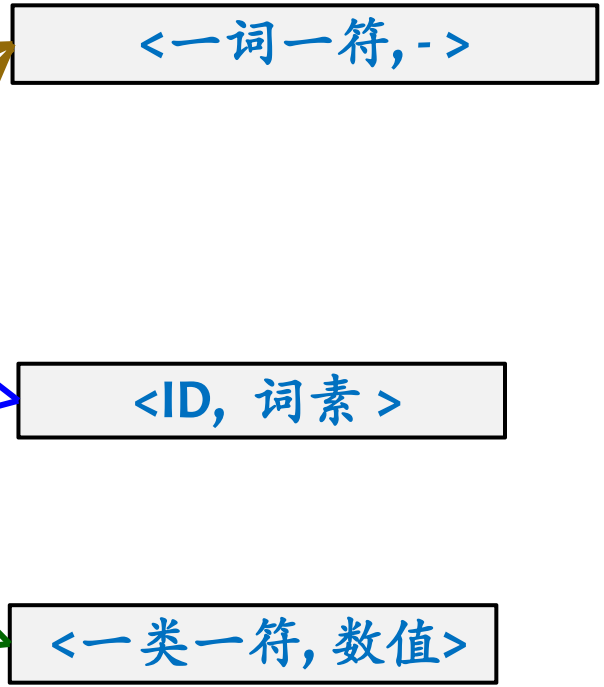
＜符号名字，属性值＞

	单词类型	种别	符号名字	属性值(词法值)
1	关键字	program、if、else、then、…	一词一符	无
2	数据类型	int、float、char、bool …	type	类型字符串(词素)
2	标识符	变量名、数组名、记录名、过程名、…	id	名字字符串(词素)
3	常量	整型、浮点型、字符型、布尔型、…	一类一符 INT、FLOAT…	数值
4	算法运算符	+ - * / ++ --	一词一符	无
5	关系运算符	> < == != >= <=	relop	运算符(词素)
6	逻辑运算符	&& || !	一词一符	无
7	界限符	. ; ( ) = { } [ ] …	一词一符	无

➢ 语法分析器依据文法识别token串，符号的集合即是文法的终结符集合。

例:语句if count>7 then result := 3.14;

if	<IF, - >
count	<ID, count>
>	<GT, - >
7	<INT, 7 >
then	<THEN, - >
result	<ID, result>
:=	<ASSIGN, - >
3.14	<FLOAT, 3.14>
;	<SEMIC, - >



3.1 单词的描述

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

    正则表达式 $r=a(a|b)^*(\varepsilon\mid(.|\_)(a|b)(a|b)^*)$
- 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
  - $L(a|b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$
  - $L((a|b)(a|b)) = L(a|b)\ L(a|b) = \{a, b\}\{a, b\} = \{\ aa, ab, ba, bb\ \}$
  - $L(a^*) = (L(a))^* = \{a\}^* = \{\ \varepsilon, a, aa, aaa, \ldots\ \}$
  - $L((a|b)^*) = (L(a|b))^* = \{a, b\}^* = \{\ \varepsilon, \underline{a, b}, \underline{aa, ab, ba, bb}, \underline{aaa, .}\ \ldots\}$
  - $L(a|a^*b) = L(a) \cup L(a^*b) = \{\ a, b, ab, aab, aaab, \ldots\}$

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(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(0|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*)

- 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 .

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

例: C标识符的右线性文法

① $S \to a \mid b \mid c \mid \ldots \mid z \mid \_$

② $S \to aT \mid bT \mid cT \mid dT \mid \ldots \mid zT \mid \_T$

③ $T \to a \mid b \mid c \mid d \mid \ldots \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9$

④ $T \to aT \mid bT \mid cT \mid \ldots \mid zT \mid \_T \mid 0T \mid 1T \mid 2T \mid 3T \mid \ldots \mid 9T$

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$\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$上的正则表达式

例1

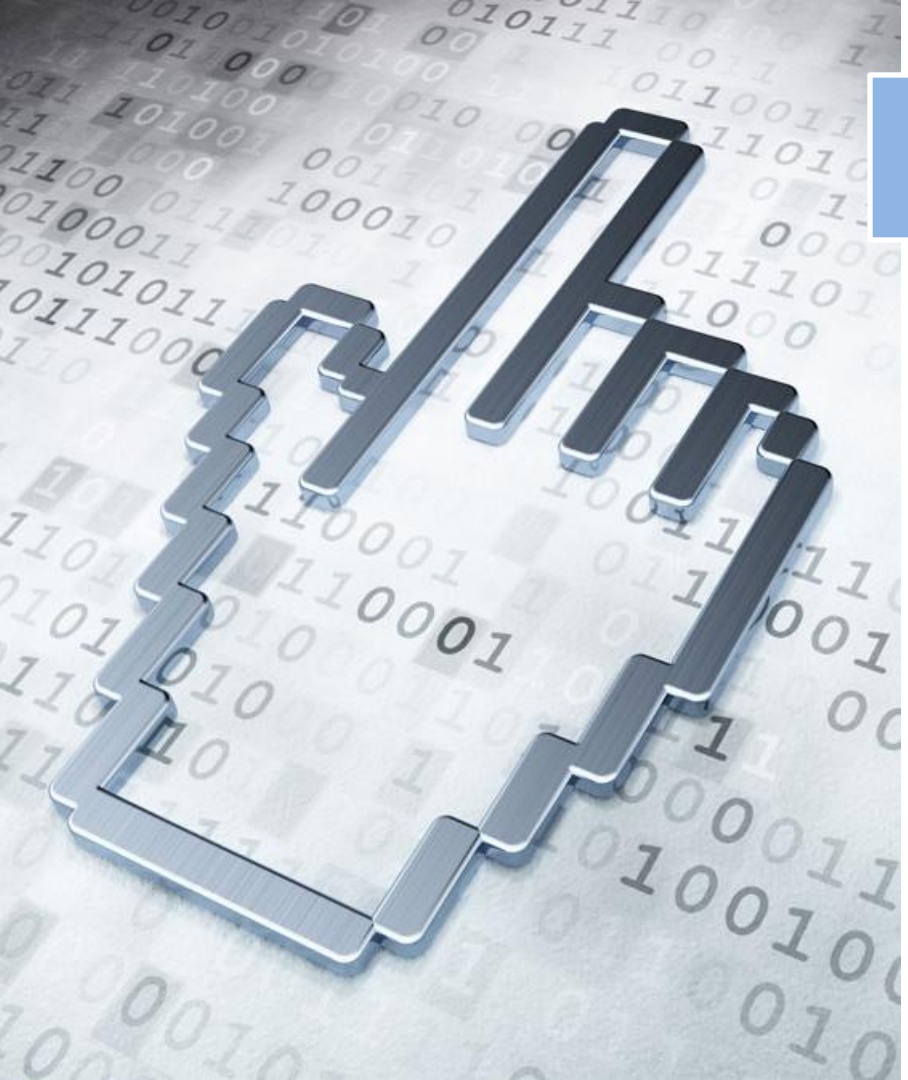
- C语言中标识符的正则定义
  - $digit \rightarrow 0|1|2|\ldots|9$
  - $letter\_ \rightarrow A|B|\ldots|Z|a|b|\ldots|z|\_$
  - $id \rightarrow letter\_(letter\_|digit)^*$

例2

- （整型或浮点型）无符号数的正则定义
  - *digit* → 0|1|2|…|9
  - *digits* → *digit digit**
  - *optionalFraction* → .*digits*|ε
  - *optionalExponent* → (*E*(+|-|ε)*digits*)|ε
  - *number* → *digits* *optionalFraction* *optionalExponent*

思考：假设整数部分不能以0开头，如何修改?

2	2.15	2.15*E*+3	2.15*E*-3	2.15*E*3	2*E*-3



提纲

3.1 单词的描述

3.2 单词的识别

3.3 词法分析阶段的错误处理

3.4 词法分析器生成工具Lex

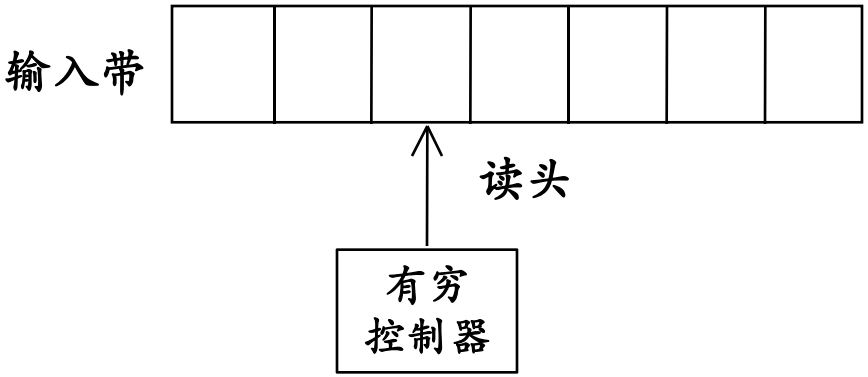
3.2 单词的识别

- 3.2.1 有穷自动机 (*Finite Automata*)
- 3.2.2 有穷自动机的分类
- 3.2.3 从正则表达式到有穷自动机
- 3.2.4 识别单词的*DFA*

3.2.1 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。

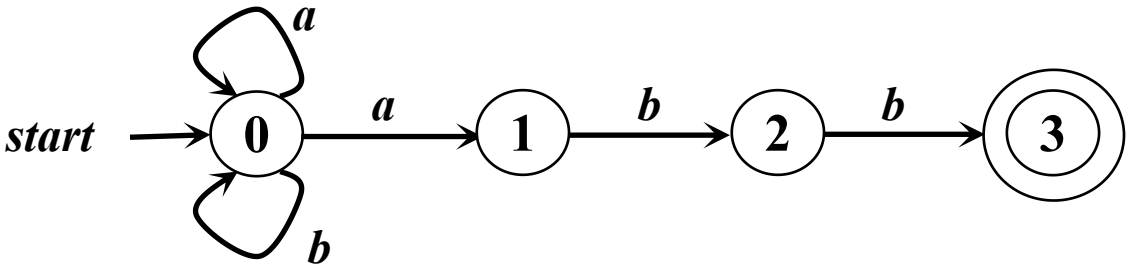
*FA*模型



- 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

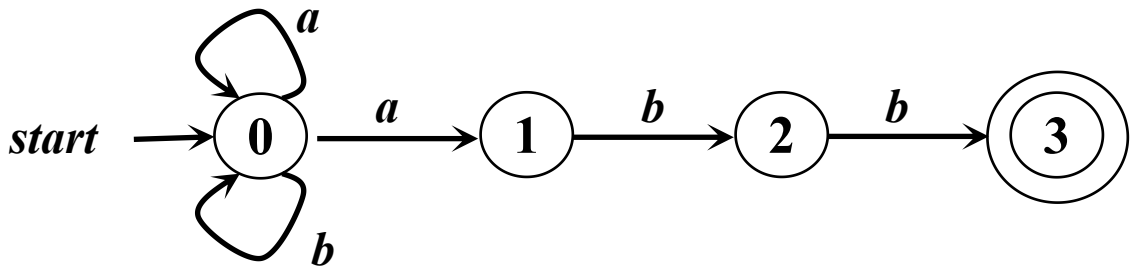
- **转换图** (*Transition Graph*)
  - 结点：*FA*的状态
    - 初始状态（开始状态）：只有一个，由*start*箭头指向
    - 终止状态（接收状态）：可以有多个，用双圈表示
  - 带标记的有向边：如果对于输入*a*，存在一个从状态*p*到状态*q*的转换，就在*p*、*q*之间画一条有向边，并标记上*a*



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，则称串x被该FA接收
- 由一个有穷自动机M接收的所有串构成的集合称为该FA定义（或接收）的语言，记为$L(M)$

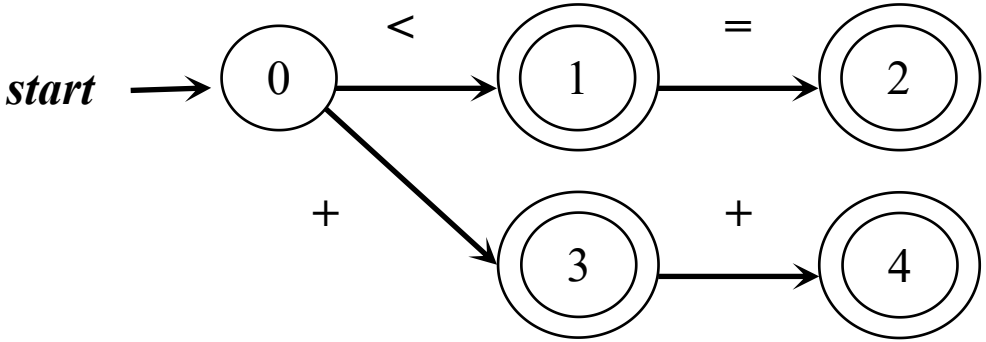
$r = (a|b)^*abb$



$L(M)$ 是 字母表$\{a, b\}$上所有以abb结尾的串的集合

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

3.2.2 *FA*的分类

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

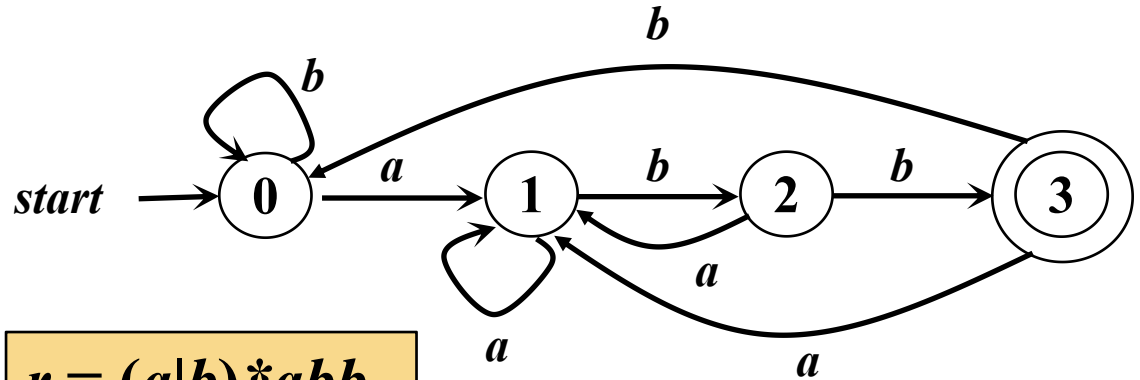
确定的有穷自动机 (*DFA*)

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

- 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*

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



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

转换表

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

DFA可以用转换图或转换表表示

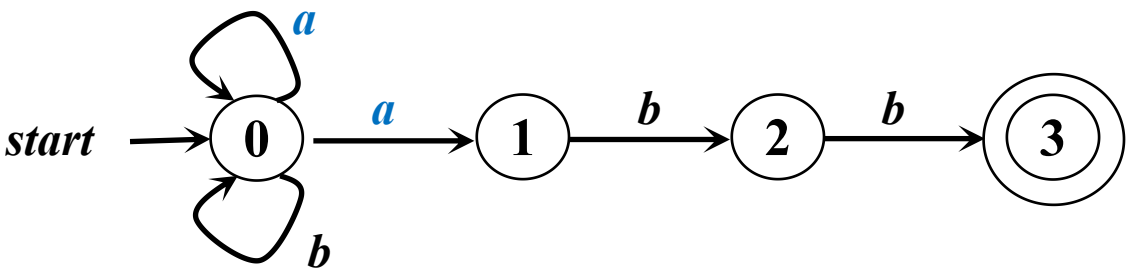
非确定的有穷自动机(*NFA*)

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

- 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

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



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

转换表

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

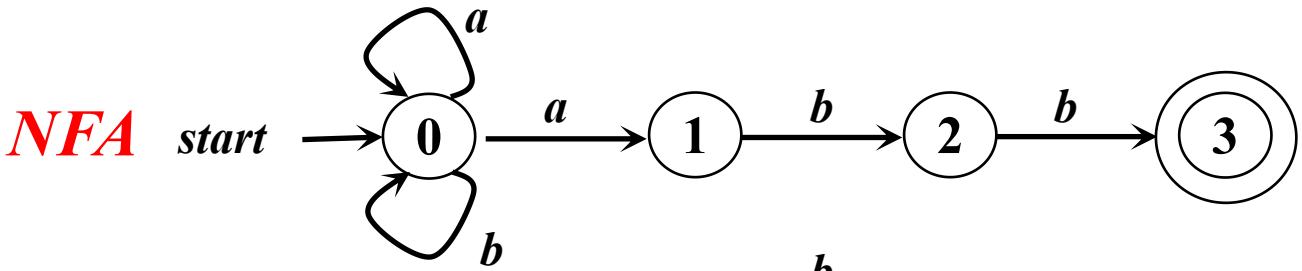
如果转换函数没有给出对应于某个状态-输入对的信息，就把Ø放入相应的表项中。

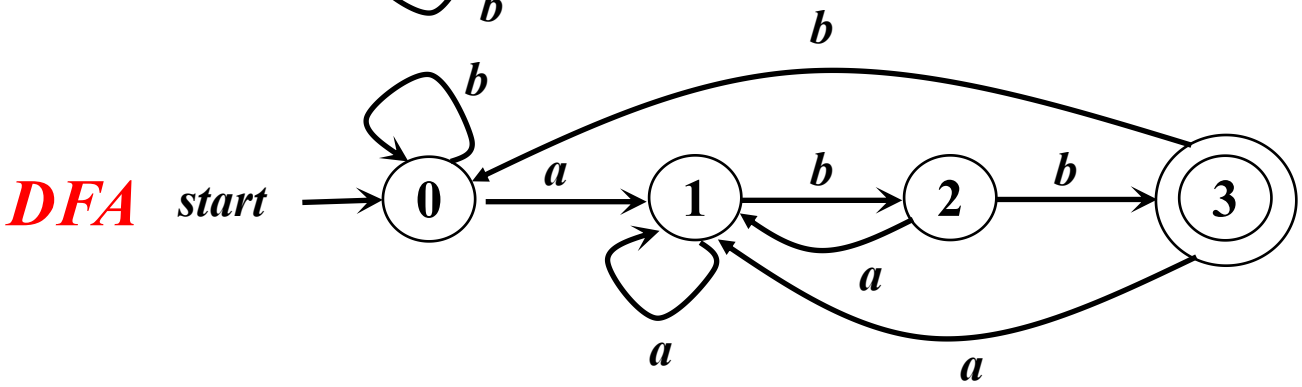
*DFA*和*NFA*的等价性

- 对任何*NFA N*，存在定义同一语言的*DFA D*
- 对任何*DFA D*，存在定义同一语言的*NFA N*

DFA和NFA的等价性

➢DFA和NFA可以识别相同的语言





状态1：串以a结尾

状态2：串以ab结尾

状态3：串以abb结尾

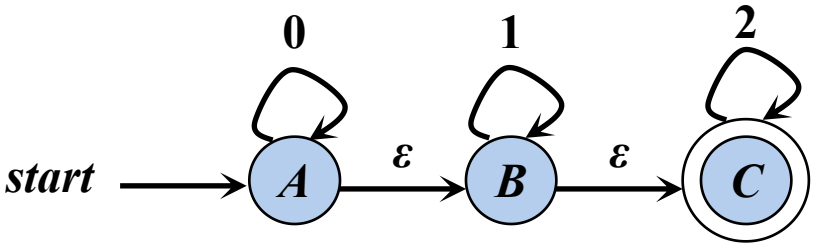
$r = (a|b)^*abb$

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，$F\subseteq S$

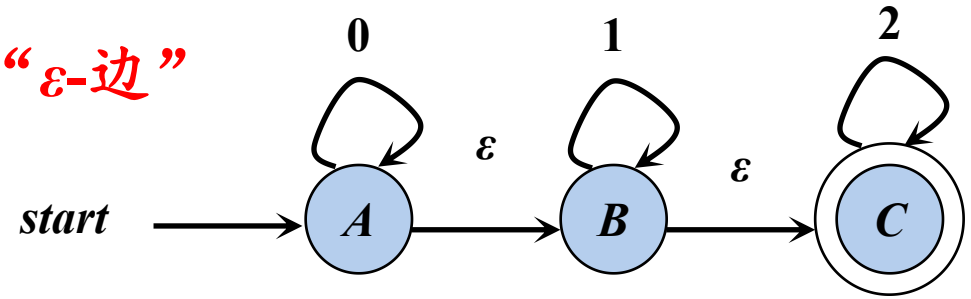


$r=0^*1^*2^*$

带有和不带有“ε-边”的*NFA*的等价性

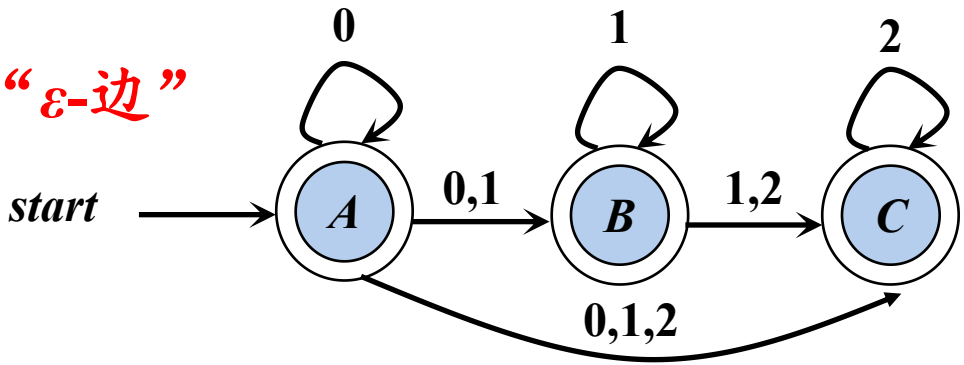
➢例

带“ε-边”



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

不带“ε-边”



状态A：0^*

状态B：0^*1^*

状态C：$0^*1^*2^*$

DFA的算法实现

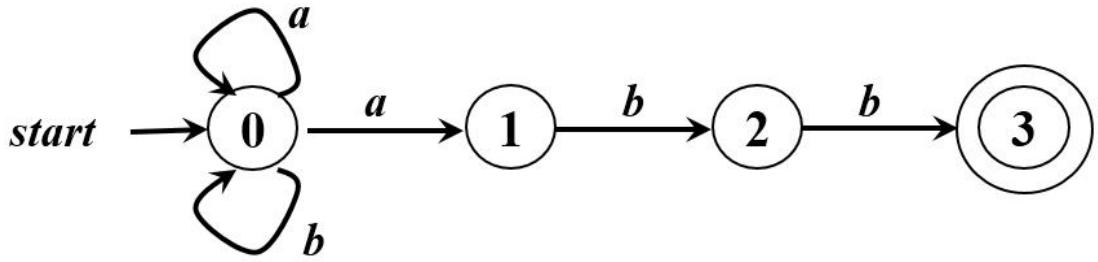
- 输入：以文件结束符eof结尾的字符串x。DFA D 的开始状态 s_0，接收状态集 F，转换函数$move$。
- 输出：如果D接收x，则回答“yes”，否则回答“no”。
- 方法：将下述算法应用于输入串 x。

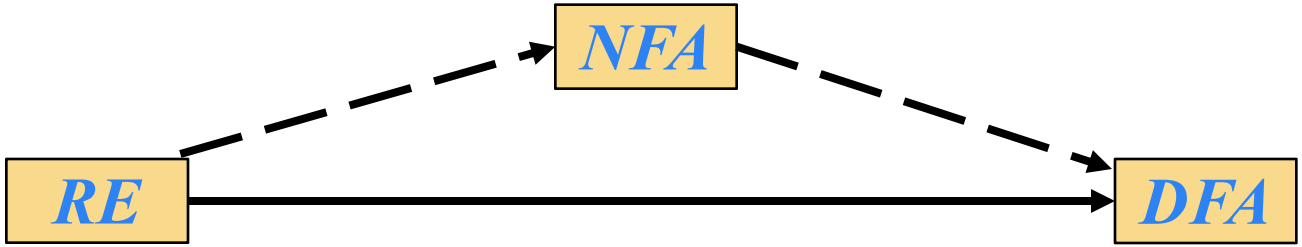
```
s = s0;
c = nextChar ();
while (c! = eof ) {
        s = move ( s , c ) ;
        c = nextChar ( ) ;
}
if (s在F中) return “yes” ;
else return “no” ;
```

- 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

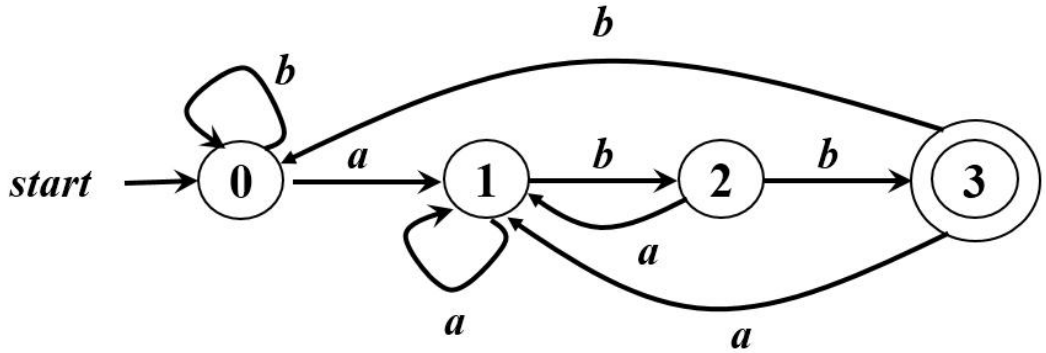
思考：move(s, c)函数如何实现?

3.2.3 从正则表达式到有穷自动机



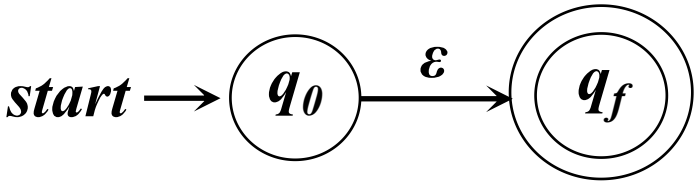


$r = (a|b)^*abb$

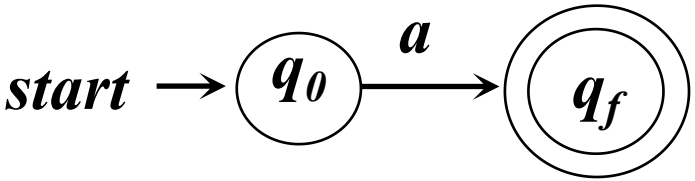


根据*RE*构造*NFA*

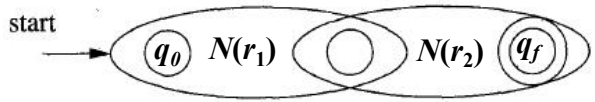
- *ε*对应的*NFA*



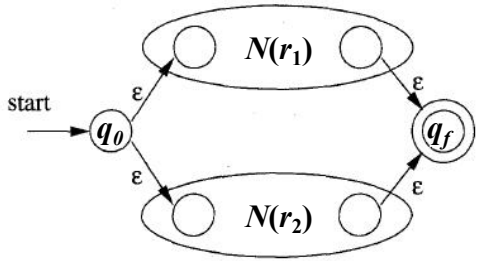
- 字母表*Σ*中符号*a*对应的*NFA*



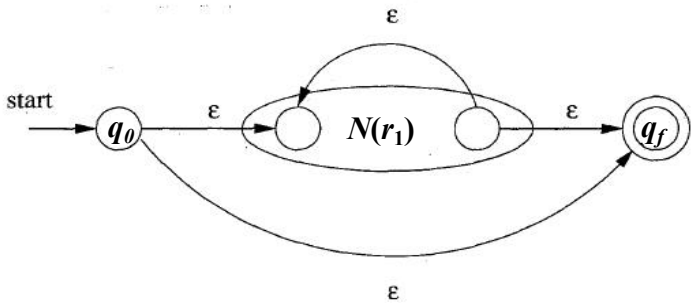
- 假设正则表达式r_1和r_2对应的NFA分别为$N(r_1)$和$N(r_2)$
  - $r = r_1 r_2$对应的NFA



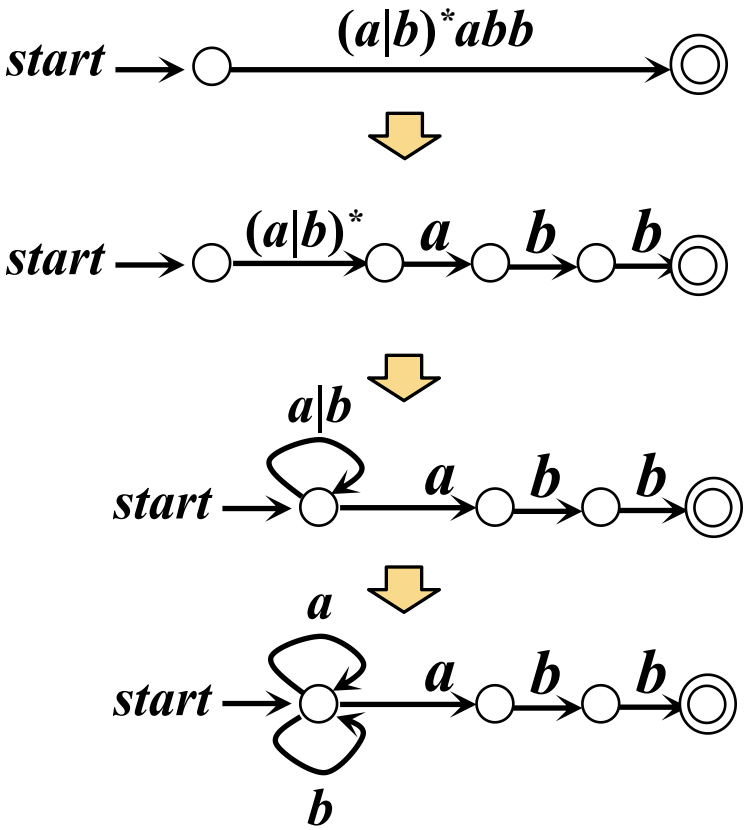
  - $r = r_1 | r_2$对应的NFA



  - $r = (r_1)^*$对应的NFA

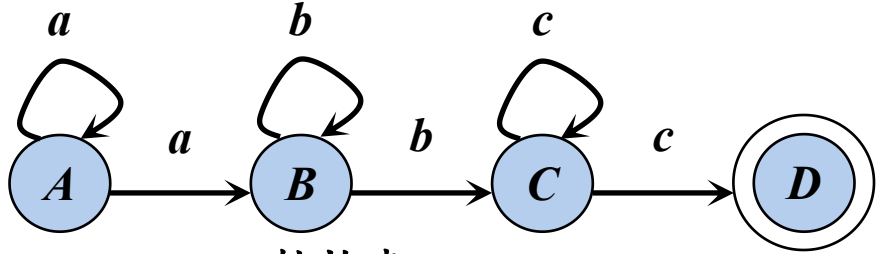


例：$r=(a|b)^*abb$ 对应的NFA



从NFA到DFA的转换

➢例1 NFA：



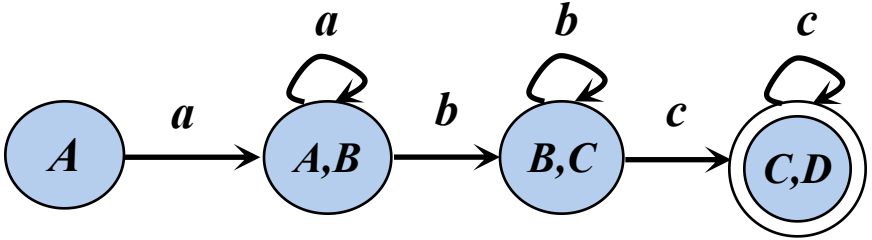
转换表

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

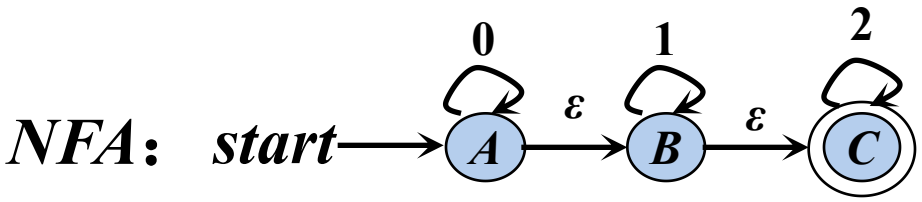
DFA的每个状态都是一个由NFA中的状态构成的集合,即NFA状态集合的一个子集

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

DFA：



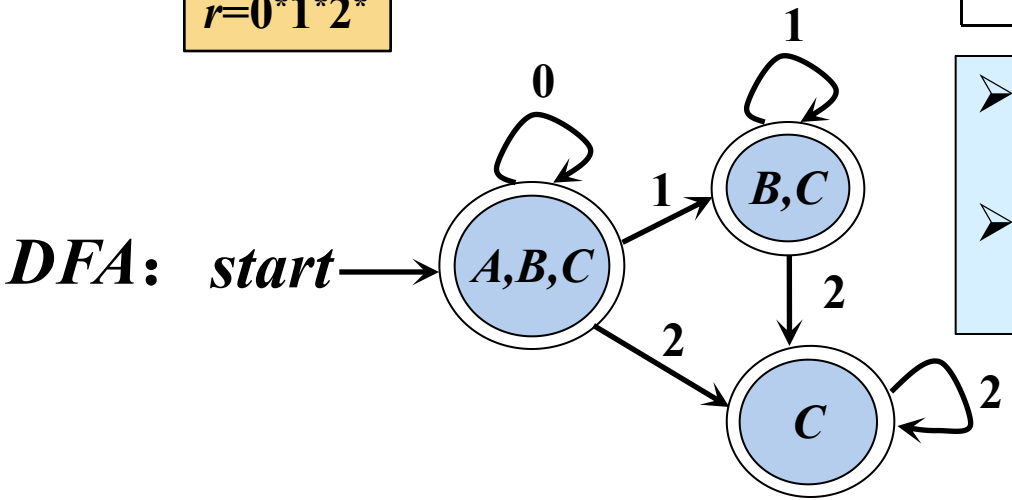
例2：从带有ε-边的NFA到DFA的转换



转换表

状态 \ 输入	0	1	2
A	{A,B,C}	{B,C}	{C}
B	Ø	{B,C}	{C}
C	Ø	Ø	{C}

r=0*1*2*



- 转换表：每个表项是状态集合的ε-可达闭包
- DFA的每个状态也是NFA子状态集合的ε-可达闭包

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;
             }
}
```

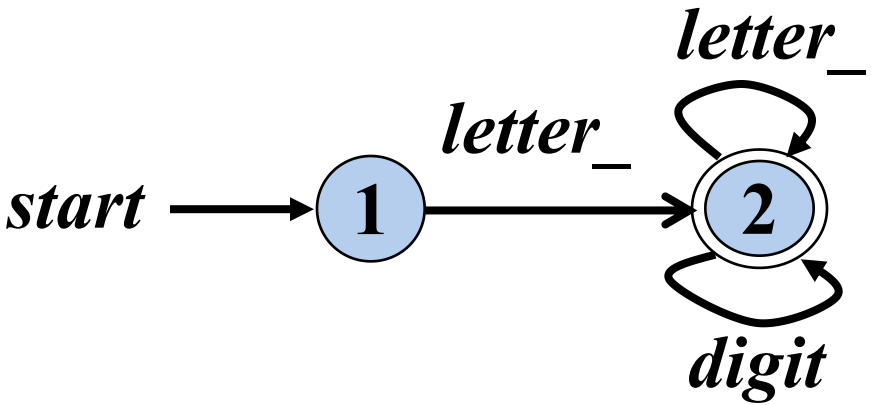
3.2.4 识别单词的*DFA*

➢识别标识符的*DFA*

$digit \rightarrow 0|1|2|\ldots|9$

$letter\_ \rightarrow A|B|\ldots|Z|a|b|\ldots|z|\_$

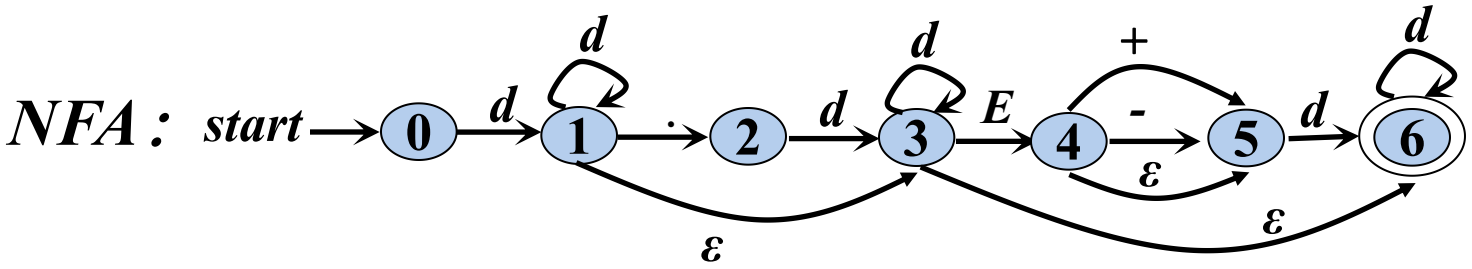
$id \rightarrow letter\_(letter\_|digit)^*$



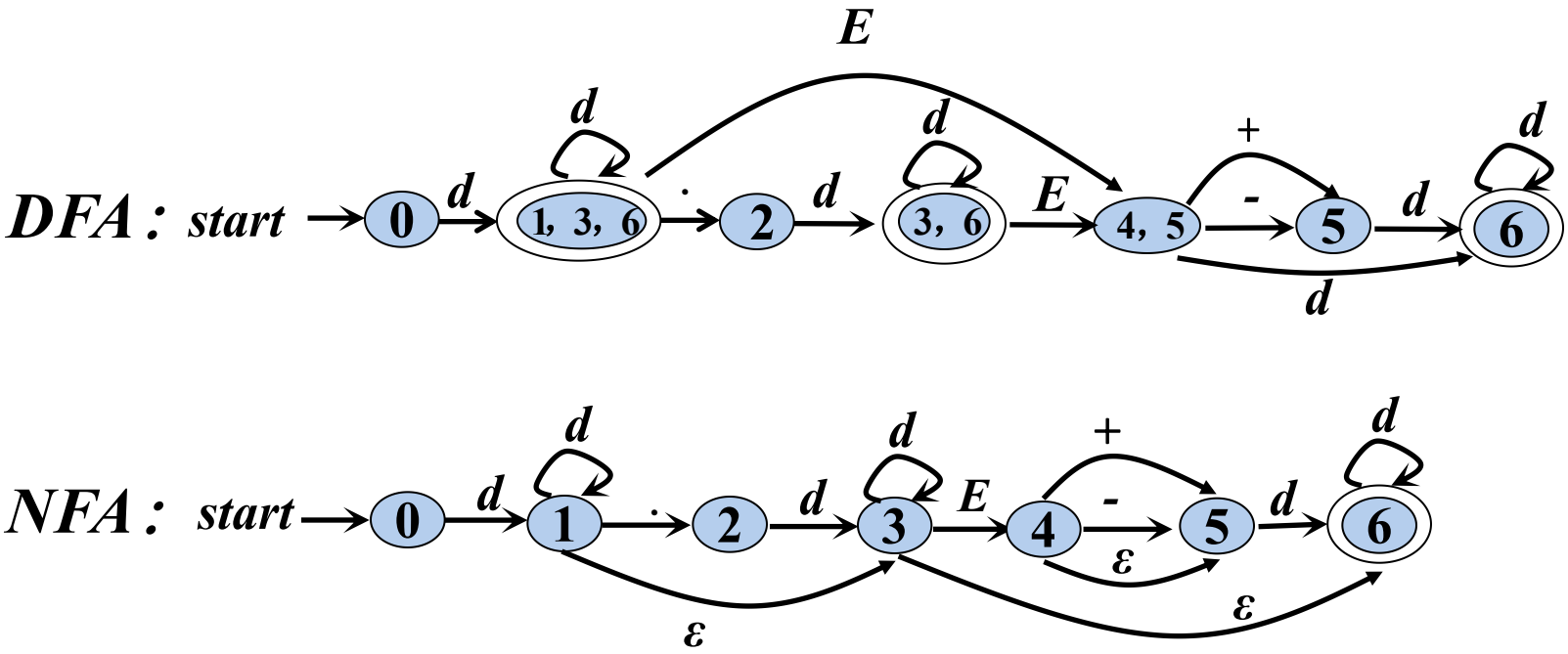
识别无符号数的DFA

- *digit* → 0|1|2|…|9
- *digits* → *digit digit**
- *optionalFraction* → .*digits*|ε
- *optionalExponent* → (*E*(+|-|ε)*digits*)|ε
- *number* → *digits optionalFraction optionalExponent*

思考：假设整数部分不能以0开头，如何修改？

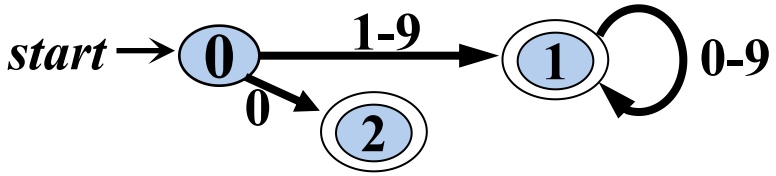


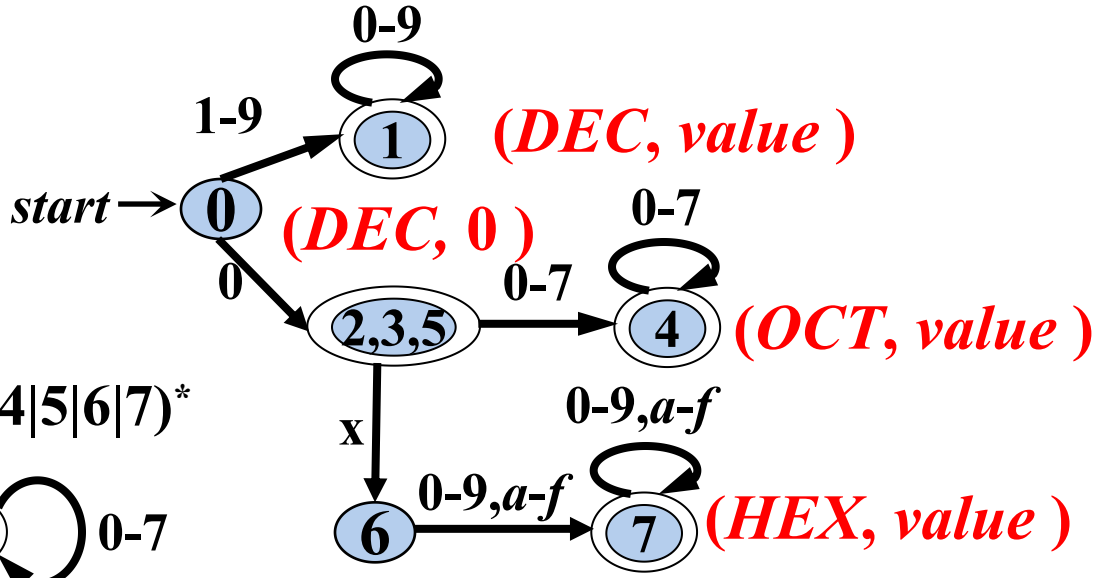
识别无符号数的DFA


DFA：start
0
d
1, 3, 6
d
.
2
d
3, 6
d
E
4, 5
+
-
5
d
6
d
E
d
d
d
NFA：start
0
d
1
d
.
2
d
3
d
E
4
+
-
5
d
6
d
ε
ε
ε

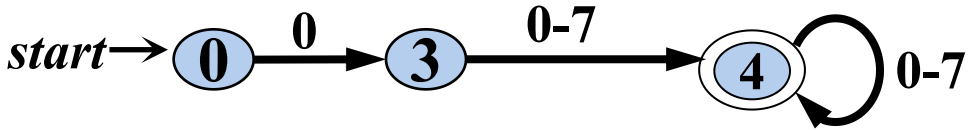
识别各进制无符号整数的*DFA*

DEC → (1|...|9)(0|...|9)* |0

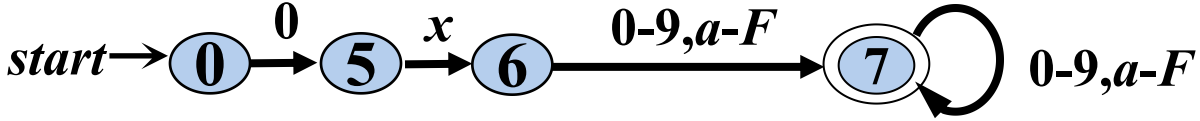




OCT → 0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)*

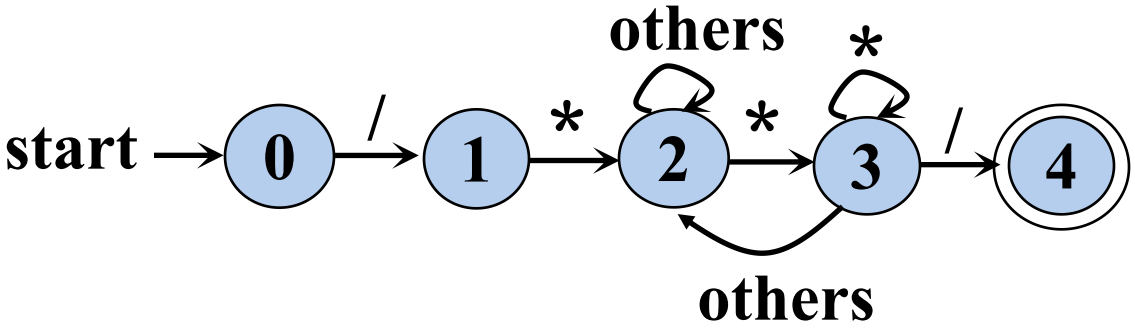


HEX → 0*x*(0|1|...|9|*a*|...|*f*|*A*|…|*F*)(0|...|9|*a*|...|*f* |*A*|…|*F*)*



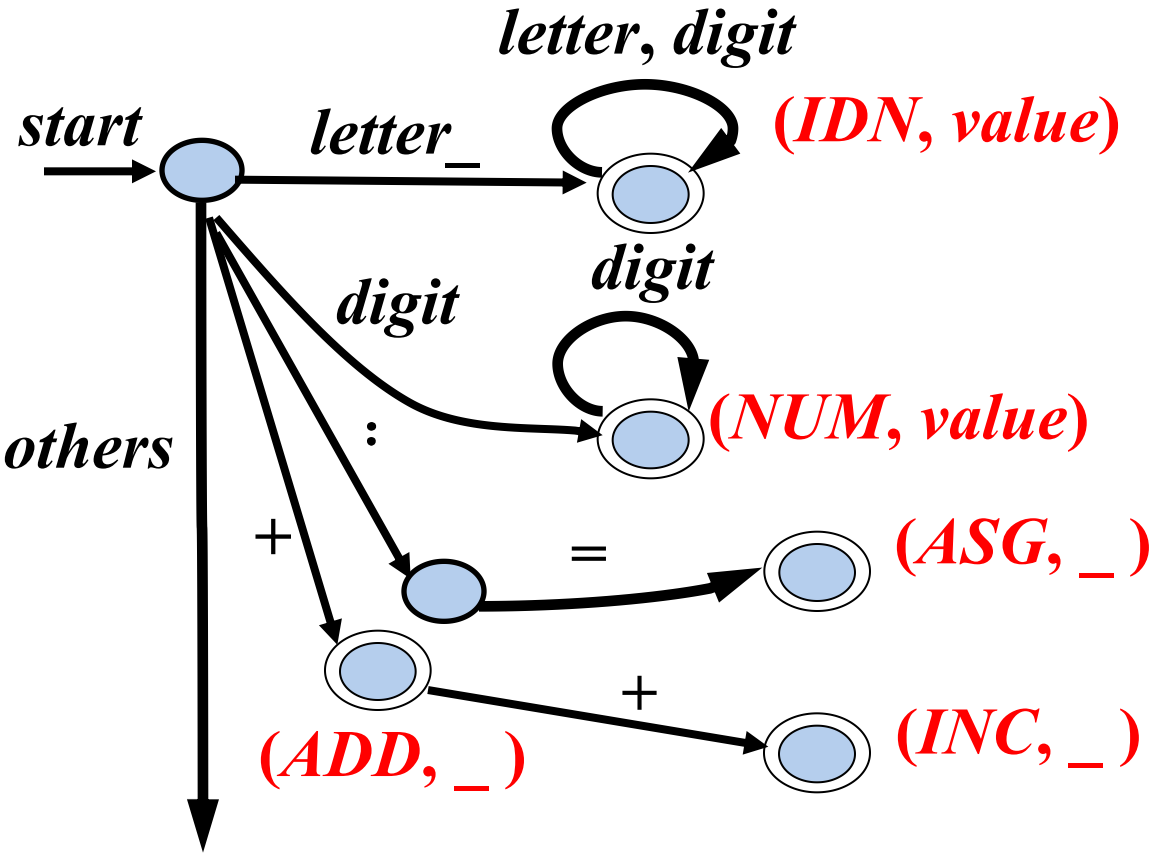
识别注释的*DFA*

others:除了*和 / 之外的其它字符



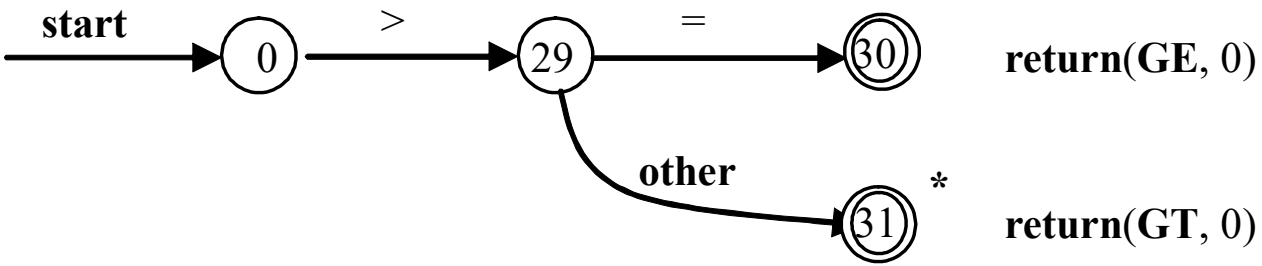
思考: 如果注释文本中允许出现*和 /，只要不出现“*/”子串，如何定义?

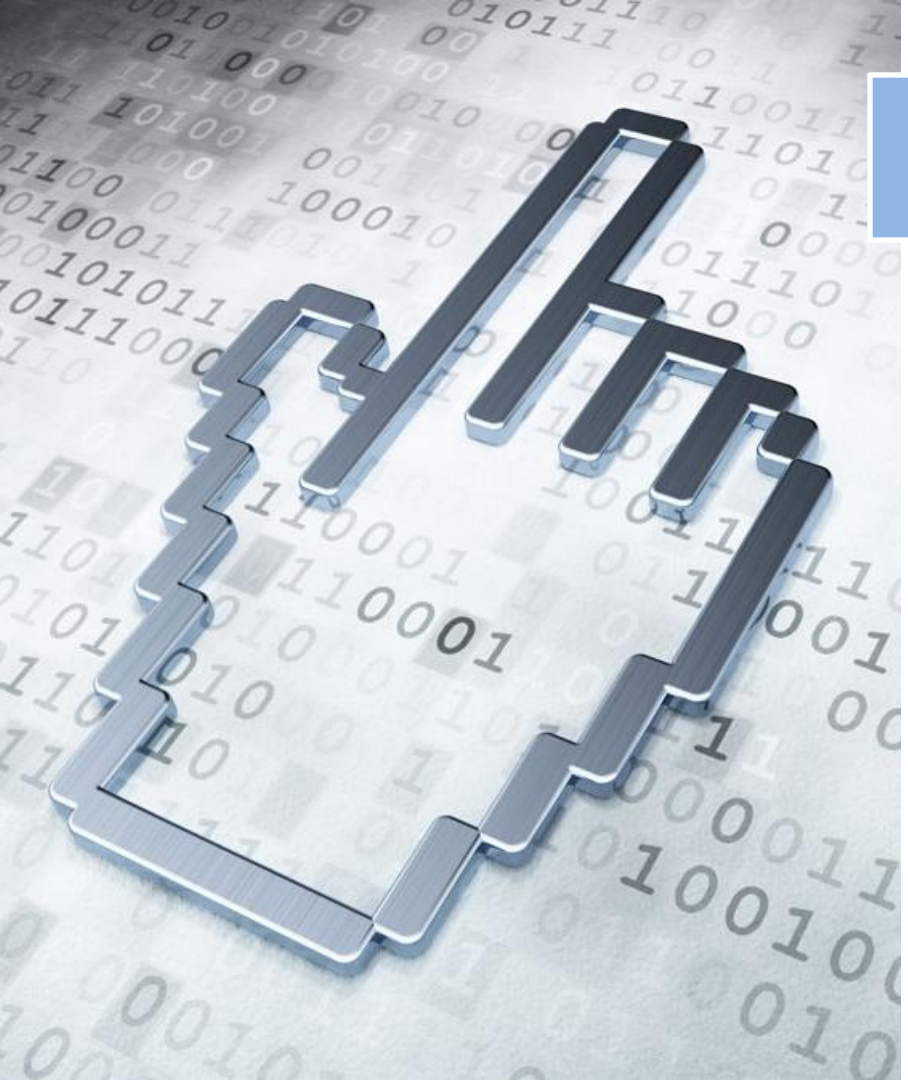
识别*Token*的*DFA*



识别*Token*的状态转换图

- **动作标记**：可以将识别单词后需要执行的**动作**标在*DFA*上构成状态转换图。
- **回退处理**：如果到达终止状态时，读入了一个与当前单词无关的字符，是下一个单词的开始符号，需要回退一个字符。
  - 状态上的*表示向前指针必须回退一个字符。





提纲

3.1 单词的描述

3.2 单词的识别

3.3 词法分析阶段的错误处理

3.4 词法分析器生成工具Lex

3.4 词法分析阶段的错误处理

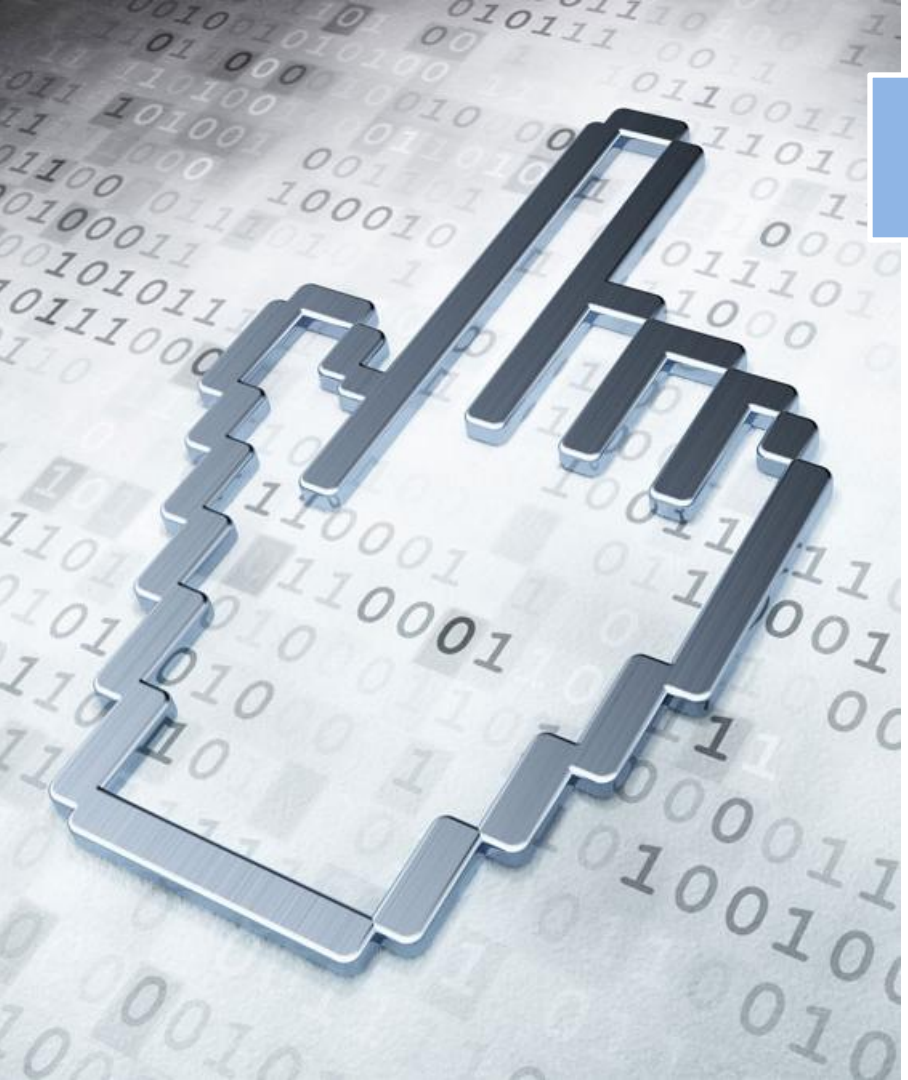
- 词法错误的类型
  - 非法字符
    - 例：~@
  - 单词拼写错误
    - 例：*int i* = 0*x*3*G*; *float j* =1.05*e*; //G和e识别为标识符
- 词法错误检测
  - 如果当前状态与当前输入符号在转换表对应项中的信息为空，而当前状态又不是终止状态，则调用错误处理程序

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
- 进行修补尝试
  - 删除一个多余的字符
  - 插入一个遗漏的字符
  - 用一个正确的字符代替一个不正确的字符；
  - 交换两个相邻的字符



提纲

3.1 单词的描述

3.2 单词的识别

3.3 词法分析阶段的错误处理

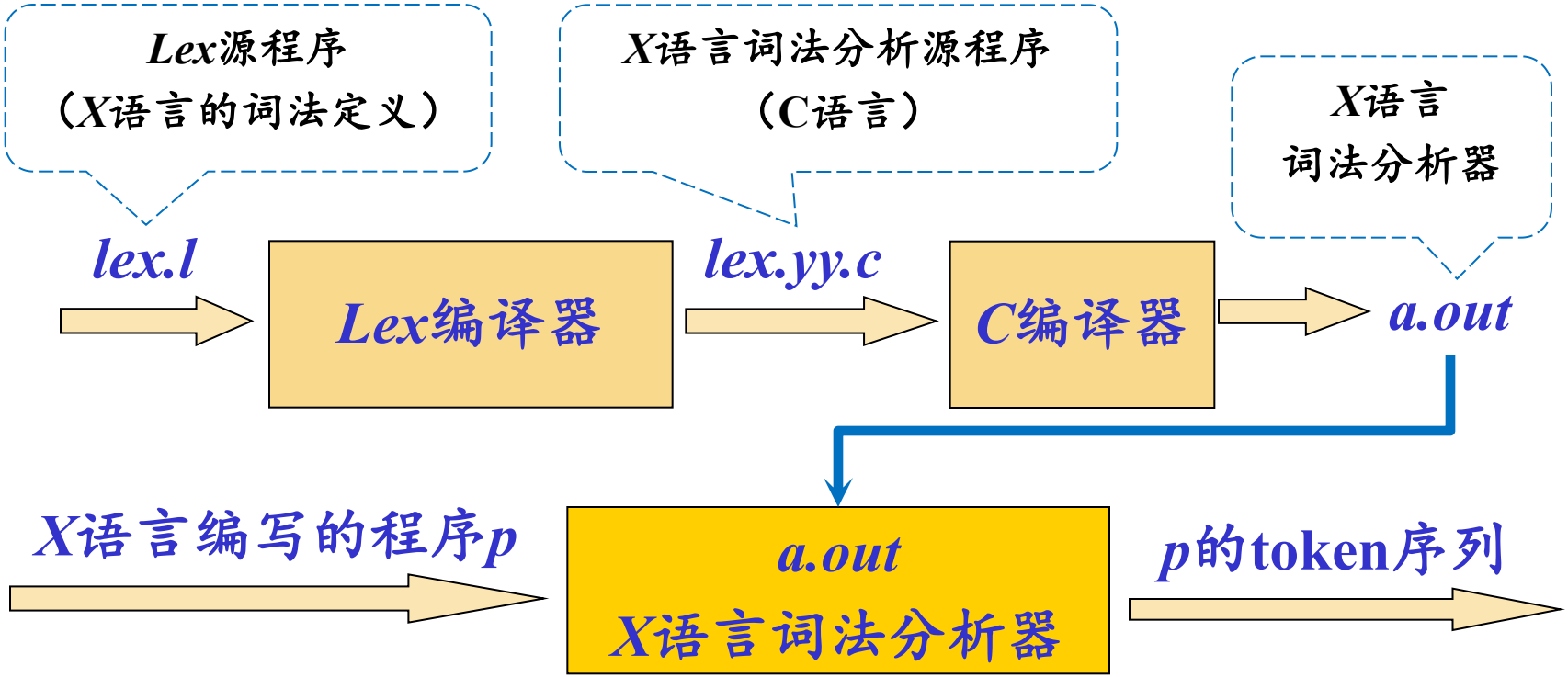
3.4 词法分析器生成工具Lex

3.4 词法分析器生成工具Lex

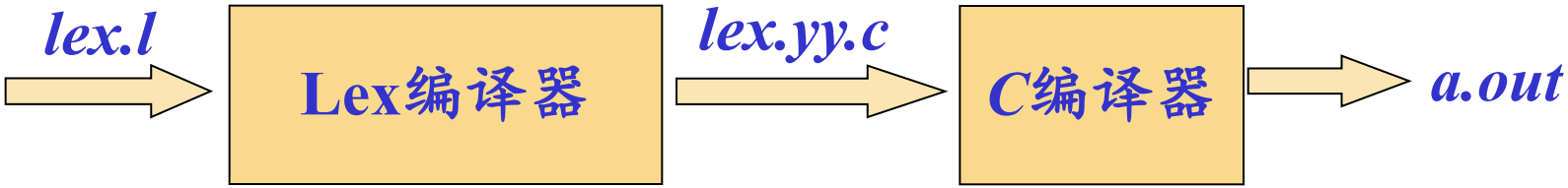
- Lex的构成
  - Lex语言
  - Lex编译器

Lesk, M.E. (October 1975). "Lex – A Lexical Analyzer Generator". *Comp. Sci. Tech. Rep. No. 39* (Murray Hill, New Jersey: Bell Laboratories).

Lex的使用


Lex源程序
（X语言的词法定义）
X语言词法分析源程序
（C语言）
X语言
词法分析器
lex.l
Lex编译器
lex.yy.c
C编译器
a.out
X语言编写的程序p
a.out
X语言词法分析器
p的token序列

Lex的使用



- Lex.yy.c 包含 函数 yylex()
  - 作用是识别输入文件中的词法单元
  - 可以添加规则，每识别一个词法单元就返回这个单词。

Flex程序的结构

声明部分（可选）

```
%{ /* 此处省略 #include部分 */
    int chars=0;
    int words = 0;
    int lines=0
%}
letter [a-zA-Z]
%%
{letter}+ {words++;  chars+= yyleng; }
\n        { chars++; lines
.         { chars++;}
%%
int main(int argc,char** argv)
if (argc>1){(argc >(!(yyin =
fopen(argv[1],"r")))}perror(argv[1]);return
1;}}yylex();printf("%8d%8d%8d\n",words,
chars);return 0;}
```

所有内容都被直接复制到文件lex.yy.c中

定义部分

正则定义

转换规则

格式：模式 {动作}

用户自定义代码

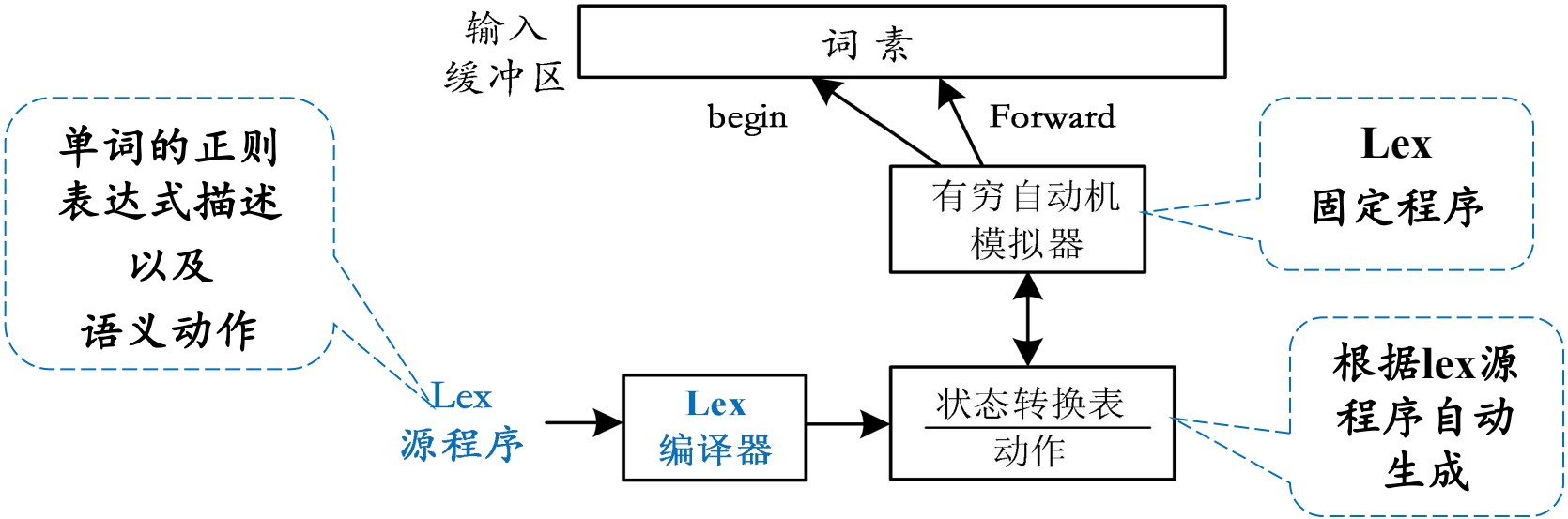
所有内容都被直接复制到文件lex.yy.c中

两组%%将代码分为三部分

Lex的实现原理

Lex的功能是根据Lex源程序构造一个词法分析程序，

该词法分析器实质上是一个有穷自动机。



Lex的功能是根据Lex源程序生成状态转换表

扫描器自动生成的意义

- Lex的构成加快了分析器的实现速度
  - 程序员只需在很高的模式层次上描述软件，就可以依赖自动生成工具来生成详细的代码
- 修改扫描器的工作变得更加简单
  - 只需修改那些受到影响的模式，无需改写整个程序

本章小结

- 单词的描述
  - 正则表达式
  - 正则定义
- 单词的识别
  - 有穷自动机
  - 有穷自动机的分类
  - 从正则表达式到有穷自动机
  - 识别单词的DFA
- 词法分析阶段的错误处理
- 词法分析器生成工具Lex

结束