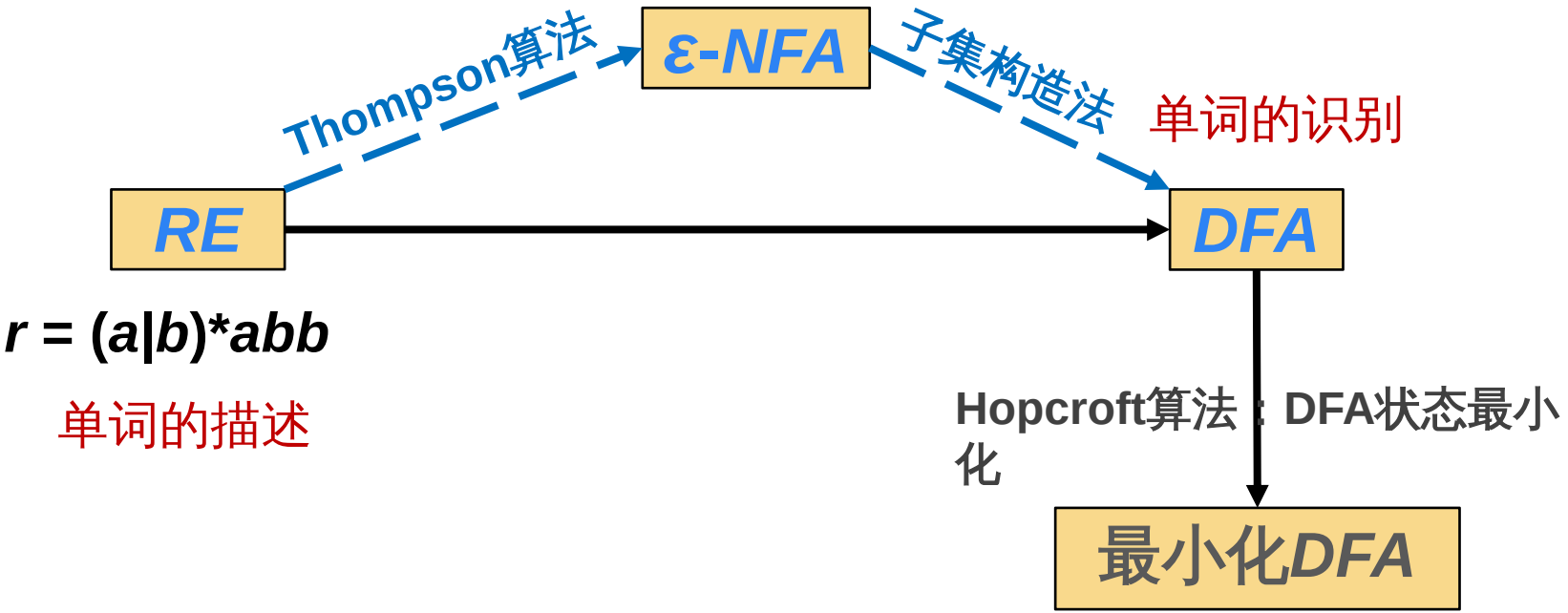
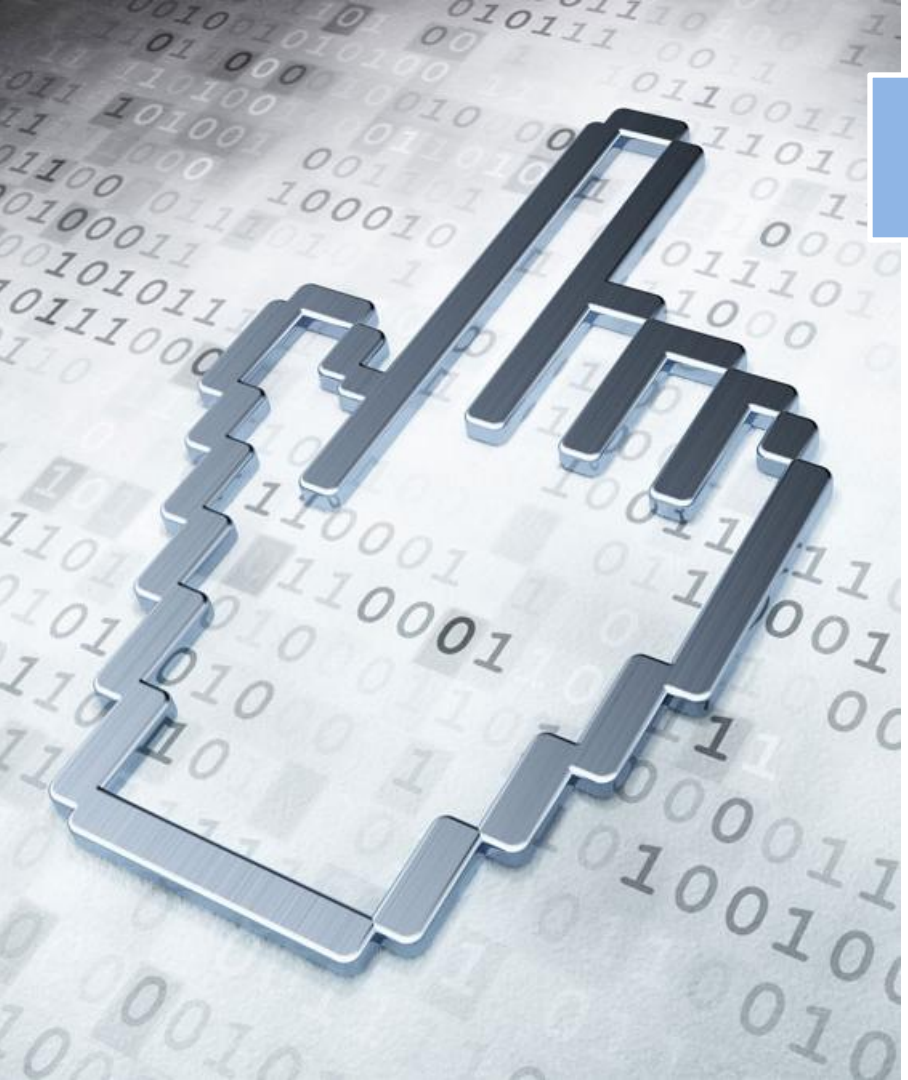
编译系统

第三章 词法分析

哈尔滨工业大学　陈鄞　单丽莉

3.2.3 从正则表达式到有穷自动机





本章内容

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，以便继续进行词法分析；
- 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

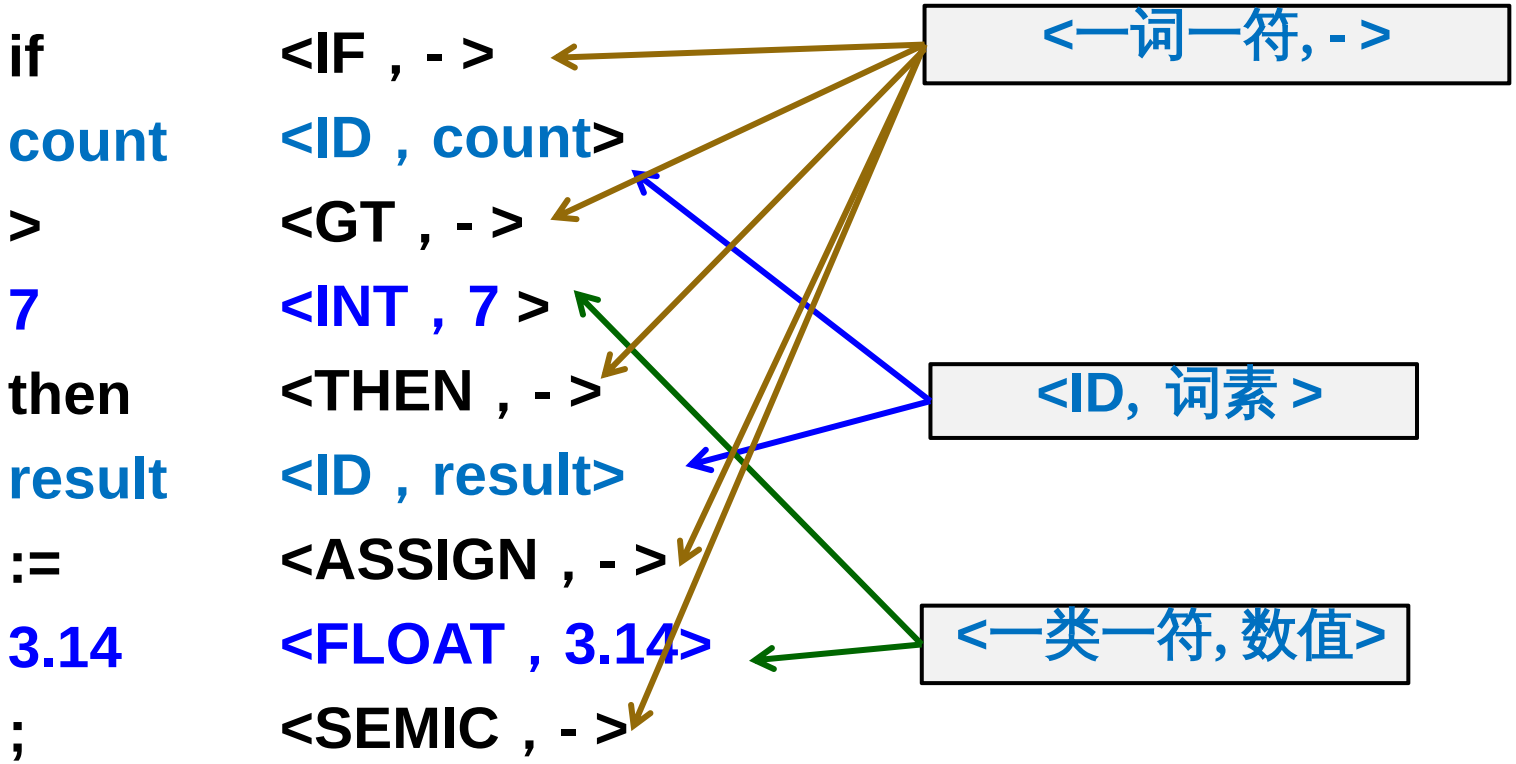
- 词法单元 token：

< 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	算术运算符	+ - * / ++ --	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分析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;



使用正则文法即可定义每类单词构成的语言，是正则语言

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 $r = a(a|b)^*(\varepsilon | (.| \_)(a|b)(a|b)^*)$
- 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*(ε) = {ε}
- 如果 *a*∈∑，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*)∪*L*(*s*)
  - *rs* 是一个*RE*，*L*(*rs*) = *L*(*r*) *L*(*s*)
  - r^* 是一个*RE*，*L*(r^*) = (*L*(*r*))*
  - (*r*) 是

运算的优先级：*、连接、|

例

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

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
$r (s\ t) = (r\ s) t$	连接是可结合的
$r (s \mid t) = r\ s \mid r\ t$; $(s \mid t) r = s r \mid t\ r$	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 $S \rightarrow a \mid b \mid c \mid \ldots \mid z \mid \_$

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

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

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

正则定义（*Regular Definition*）

- 正则定义是具有如下形式的定义序列：

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

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

例1

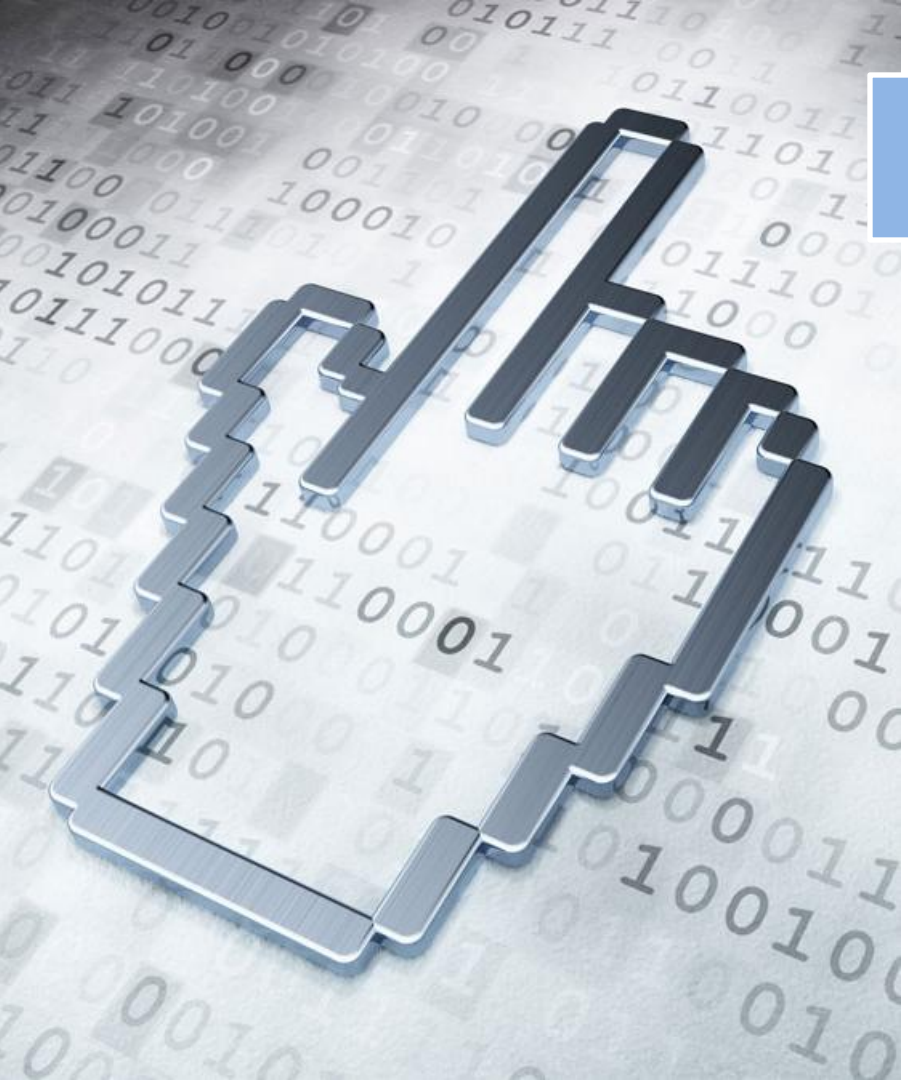
- C语言中标识符的正则定义
  - *digit* → 0|1|2|…|9
  - *letter*_ → *A*|*B*|…|*Z*|*a*|*b*|…|*z*|_
  - *id* → *letter*_(*letter*_|*digit*)*

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*

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

3.2 单词的识别

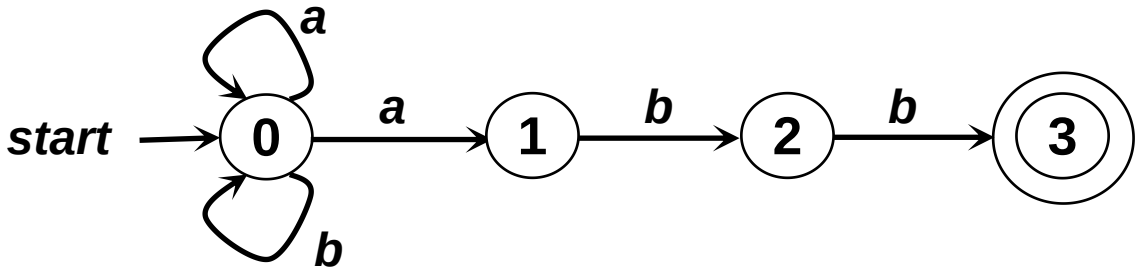
- 3.2.1 有穷自动机 (*Finite Automata*)
- 3.2.2 有穷自动机的分类
- 3.2.3 从正则表达式到有穷自动机
- 3.2.4 识别单词的*DFA*

3.2.1 有穷自动机

- 有穷自动机 (*Finite Automata*，*FA*)是能够识别正则语言的数学模型。
- 词法分析器通过模拟有穷自动机的执行来识别单词。
- 有穷自动机包含一个状态的集合和一些从一个状态通向另一个状态的边，每条边上标记有一个符号；状态中有一个称为初始状态 ，某些称为接收状态 。
- 从初始状态经过若干个边能够到达某个接收状态，将这些边上标记的符号按顺序串连起来，就是*FA*识别的句子，所这样的句子的集合，就是*FA*识别的语言。

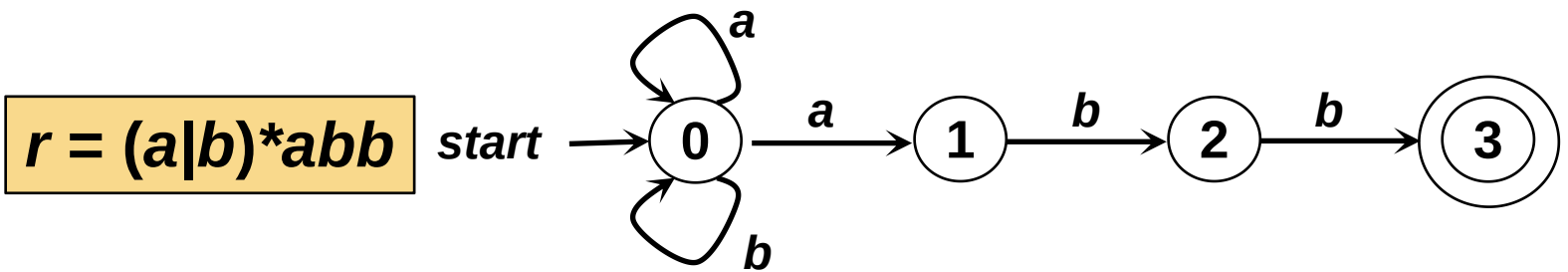
*FA*的表示

- **转换图 (*Transition Graph*)**
  - 结点：*FA*的状态
    - 初始状态：只有一个，由*start*箭头指向
    - 接收状态：可以有多个，用双圈表示
  - 带标记的有向边（转换函数）：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

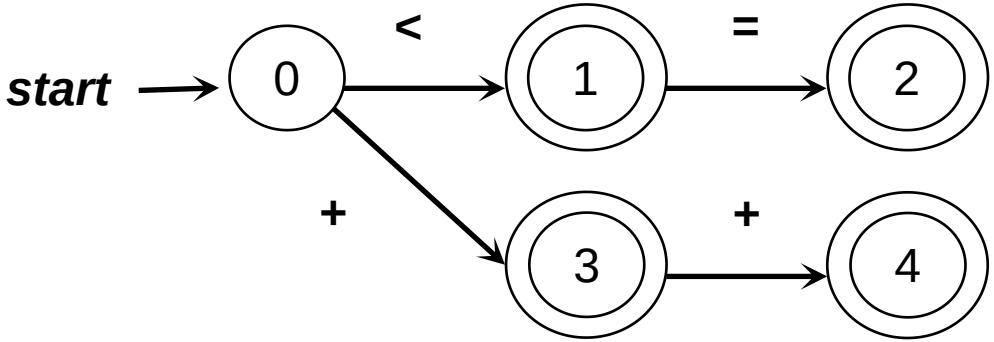
- 给定输入串*x*，如果存在一个对应于串*x*的从初始状态到某个接收状态的转换序列，则称串*x*被该*FA*接收
- 由一个有穷自动机*M*接收的所有串构成的集合称为该*FA*定义（或接收）的语言，记为*L*(*M*)



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*)

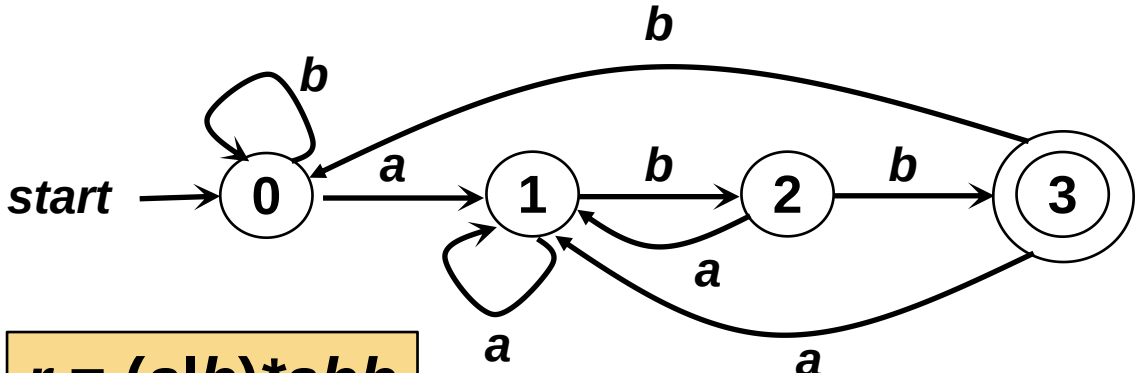
确定的有穷自动机 (*DFA*)

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

- 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$S\times\Sigma \rightarrow S$的转换函数。 $s\in S$, $a\in\Sigma$, $\delta(s, a)$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

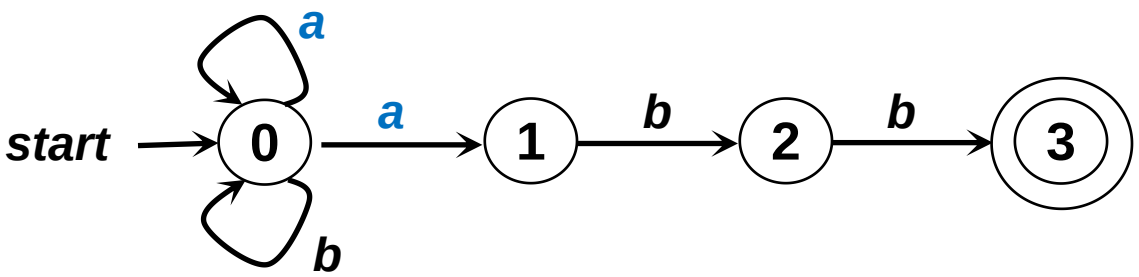
非确定的有穷自动机(*NFA*)

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

- 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$S\times\Sigma \rightarrow 2^S$的转换函数。 $s\in S$, $a\in\Sigma$, $\delta(s, a)$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

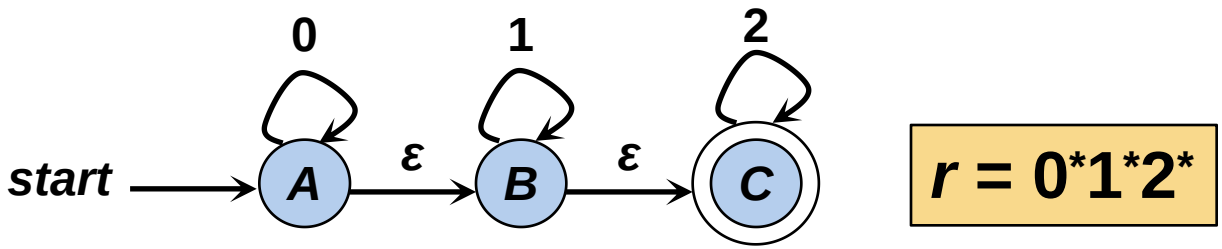
状态 \ 输入	a	b
0	{0, 1}	{0}
1	Ø	{2}
2	Ø	{3}
3●	Ø	Ø

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。

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$S\times(\Sigma\cup\{\varepsilon\})\rightarrow 2^S$的转换函数。 $s\in S$, $a\in\Sigma\cup\{\varepsilon\}$, $\delta(s,a)$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，$F\subseteq S$

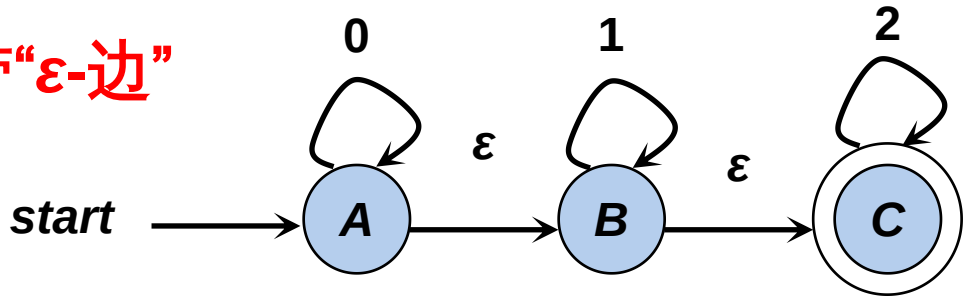


$r = 0^*1^*2^*$

带有和不带有“ε-边”的*NFA*的等价性

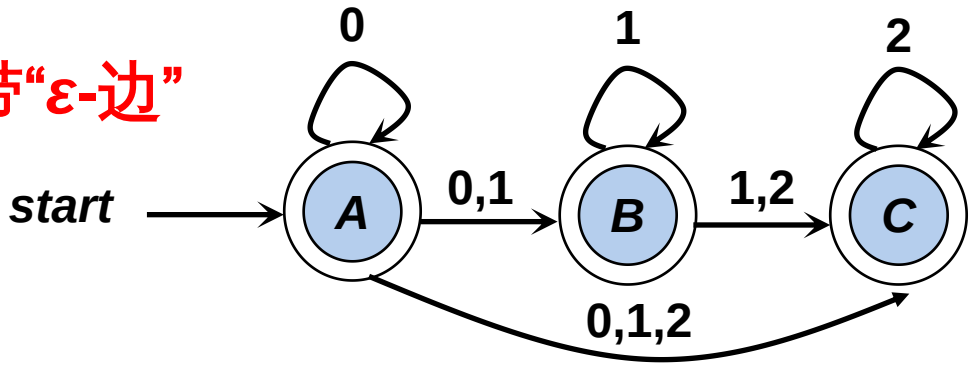
- 例

带“ε-边”



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

不带“ε-边”



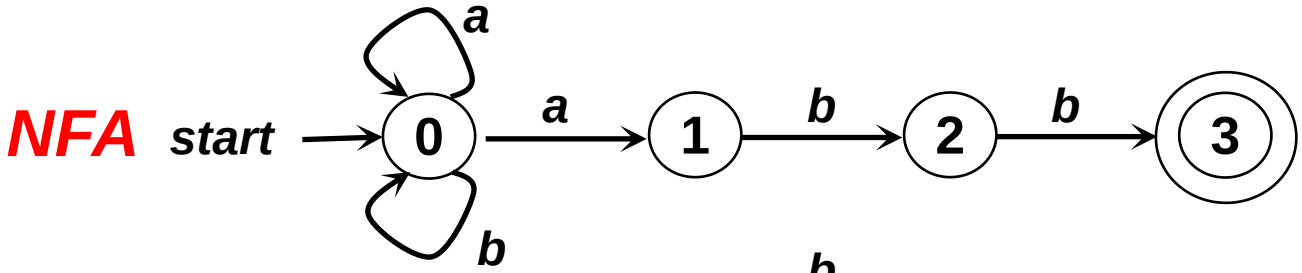
状态*A*：0^*

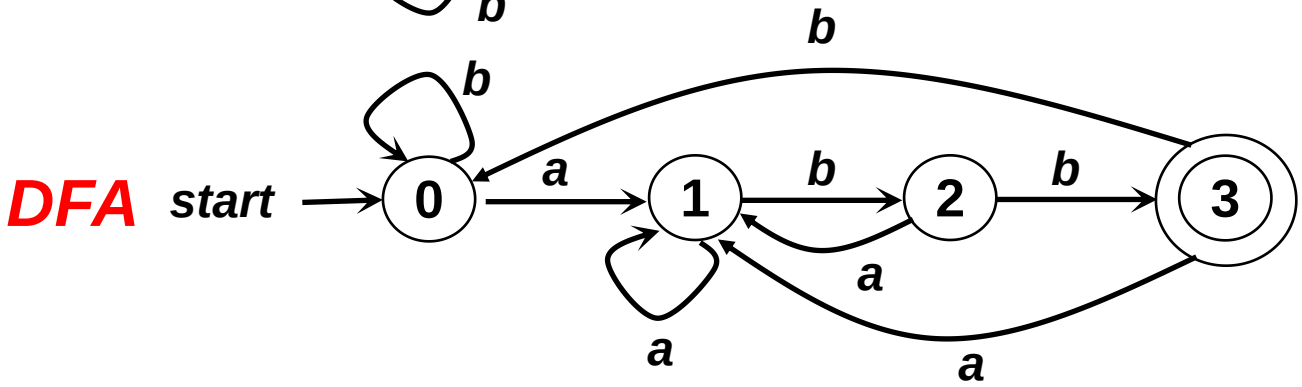
状态*B*：0^*1^*

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

*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*

- *NFA*在识别串时，从一个状态出发可能有多条标有相同符号的边，导致算法在实现时需要回溯，影响效率
- *DFA*在识别串时，给定当前状态和输入符号，每一步都有确定的转移状态，识别效率高。

DFA的算法实现（DFA模拟器）

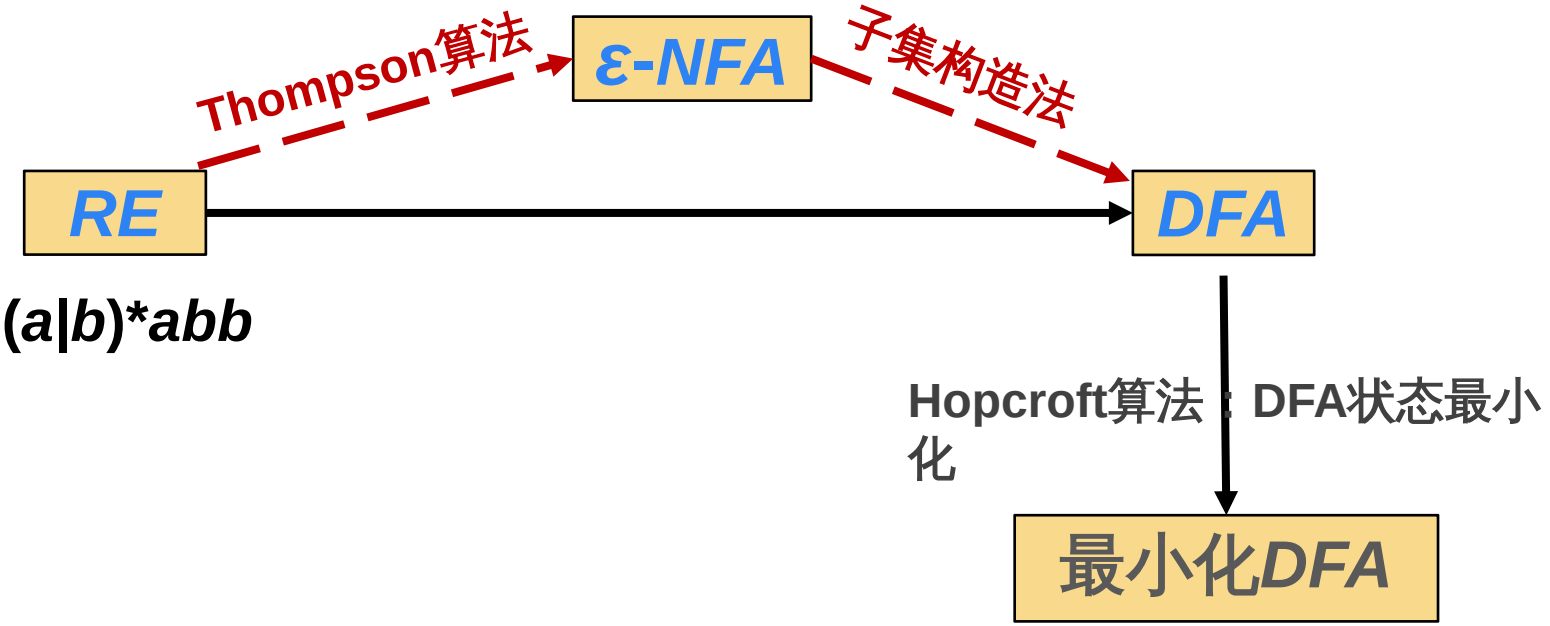
- 输入：以文件结束符eof结尾的字符串*x*。*DFA D* 的开始状态 s_0，接收状态集 *F*，转换函数*move*。
- 输出：如果 *D*接收 *x*，则回答“*yes*”，否则回答“*no*”。
- 方法：将下述算法应用于输入串 *x*。

```
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：此模拟器代码是否依赖正则表达式的定义？*move*(*s*, *c*)函数需访问DFA转换表？如何生成DFA转换表？

3.2.3 从正则表达式到有穷自动机



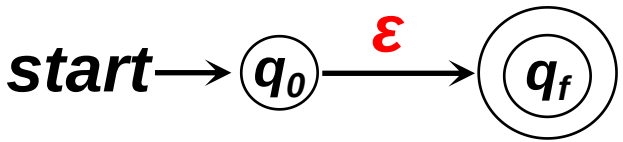
$r = (a|b)^*abb$

根据*RE* 构造*ε-NFA*（Thompson算法）

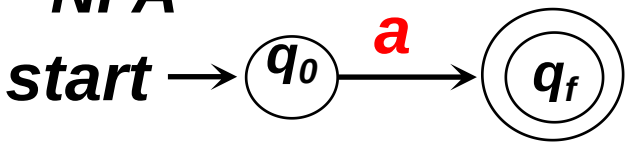
- **算法3.25** Thompson算法：
  - 对于ε、符号*a* 等基本符号，直接构造其*ε-NFA*
  - 对于复合正则表达式（连接、或和闭包），使用归纳规则构造其*ε-NFA*

基本规则：基本符号的*ε-NFA*

- 对于***ε***，有*ϵ-NFA*



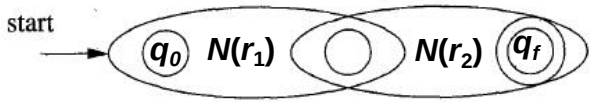
- ***a*∈*Σ*, 对于*a*，有*ε-NFA***



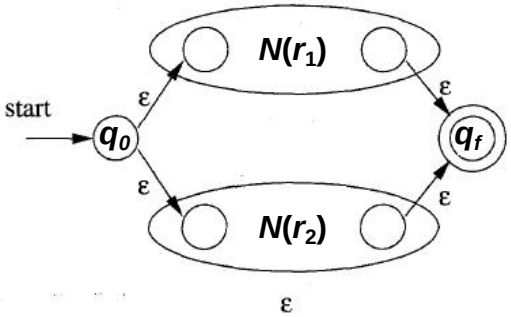
根据*RE* 构造*ε-NFA*（Thompson算法）

归纳规则：假设正则表达式r_1和r_2对应的*NFA*分别为$N(r_1)$和$N(r_2)$

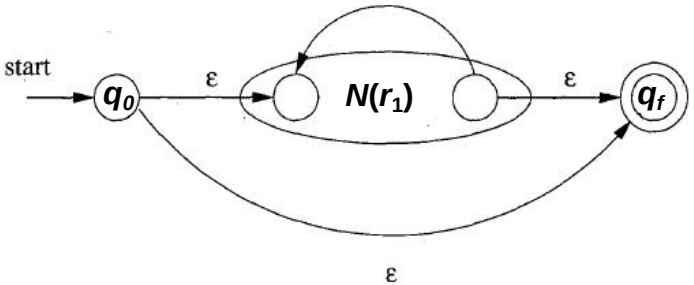
- $r = r_1r_2$对应的*NFA*



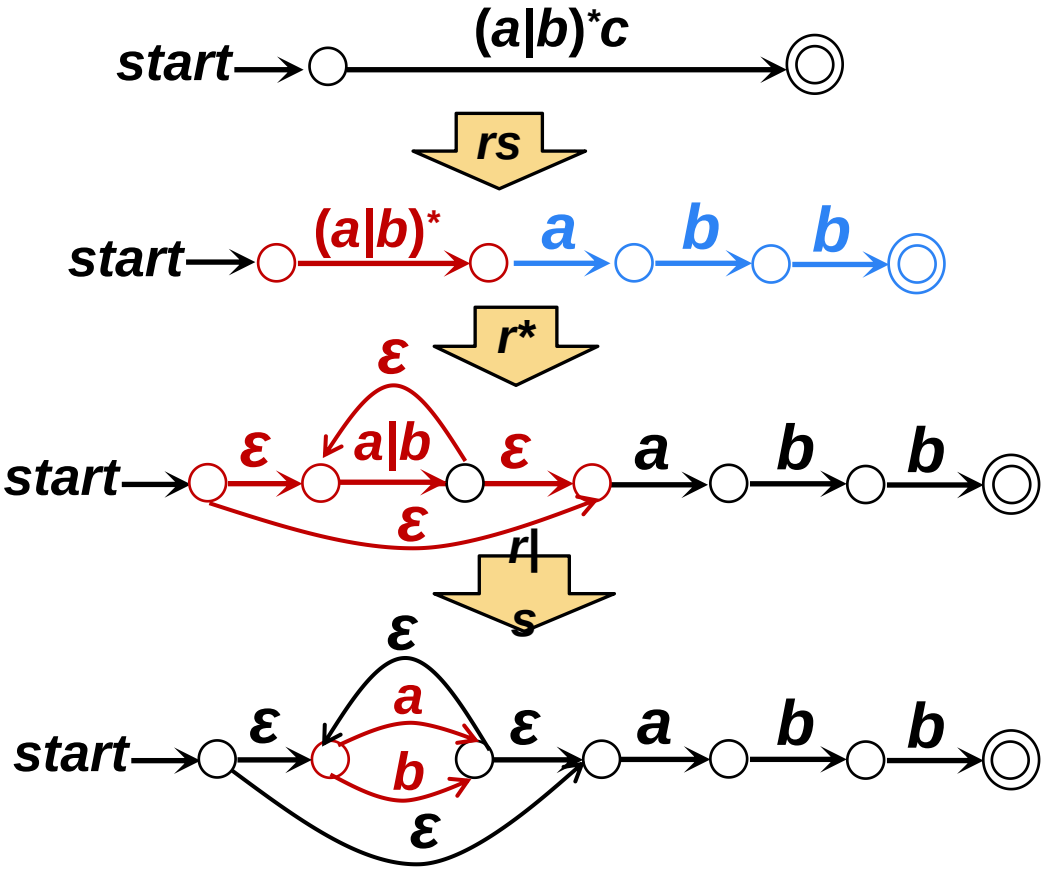
- $r = r_1|r_2$对应的*NFA*



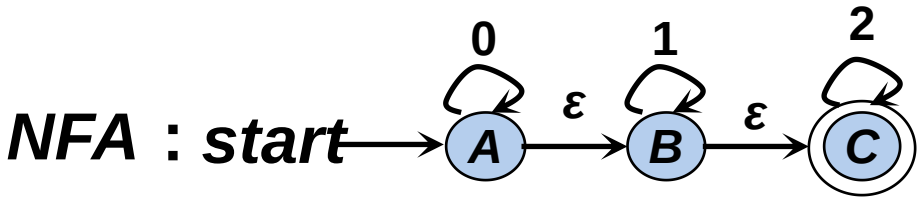
- $r = (r_1)^*$对应的*NFA*



例:$r=(a|b)^*abb$ 对应的NFA



例：从带有ε-NFA到DFA的转换

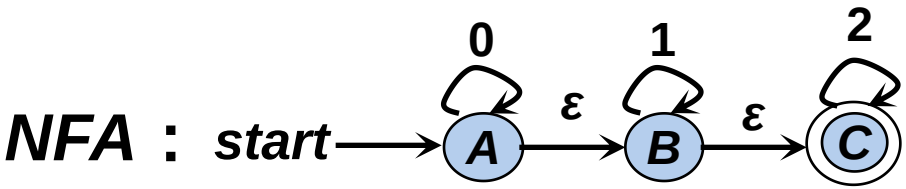


- 基础定义： ε-闭包

定义	表示	描述
状态 *s* 的ε-闭包	*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
状态集 *T* 的ε-闭包	*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*)

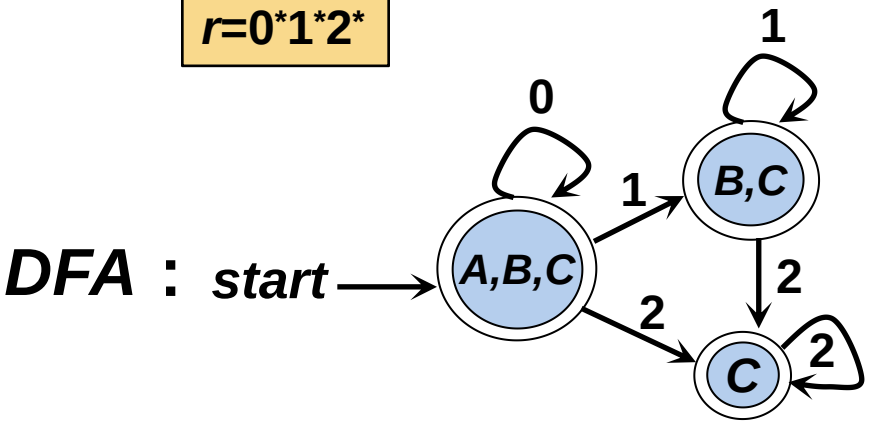
例：从带有ε-NFA到DFA的转换

NFA：



r=0*1*2*

DFA：



$move(T, a) = U_{s \in T}\ \delta_{NFA}(s, a)$

DFA转换表

NFA状态集	DFA状态	0	1	2
{A,B,C}●	A’	{ A,B,C}	{B,C}	{C}
{ B,C} ●	B’	Ø	{B,C}	{C}
{ C } ●	C’	Ø	Ø	{C}

- DFA开始状态：是NFA开始状态的ϵ-闭 包
- DFA状态s是一个NFA状态集T，则 $\delta_{DFA}(s, a) = \varepsilon\text{-}closure(move(T, a))$,
- 包含NFA接受状态的DFA状态是DFA接受状态

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*）

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；

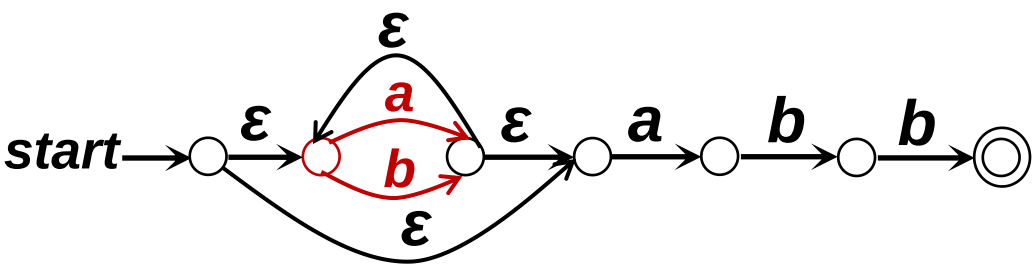
将*u*压入栈中;

}

}

课后练习

- 将NFA转换为DFA，画出转换表或转换图。



3.2.4 识别单词的*DFA*

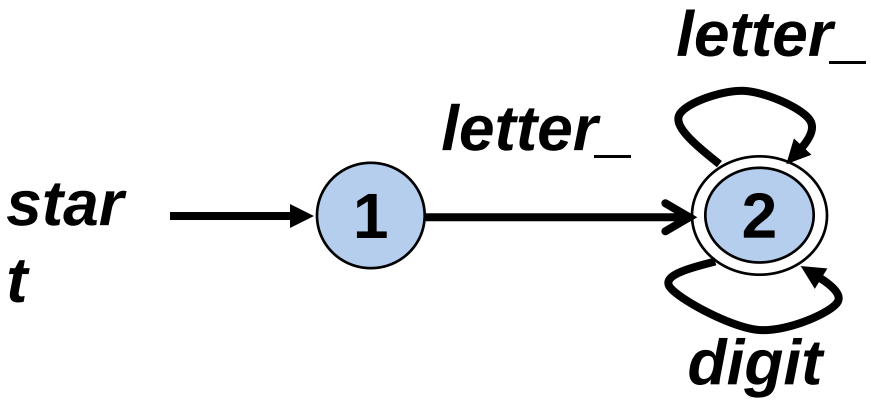
- 识别标识符的*DFA*

标识符的词法规则：字母或下划线开头，字母数字、下划线组成的符号串

digit → 0|1|2|…|9

*letter_* → *A*|*B*|…|*Z*|*a*|*b*|…|*z*|_

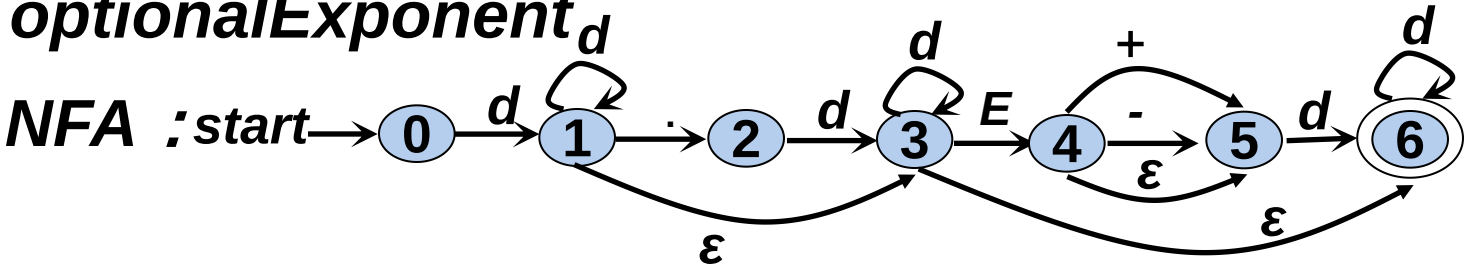
id → *letter_*(*letter_*|*digit*)*



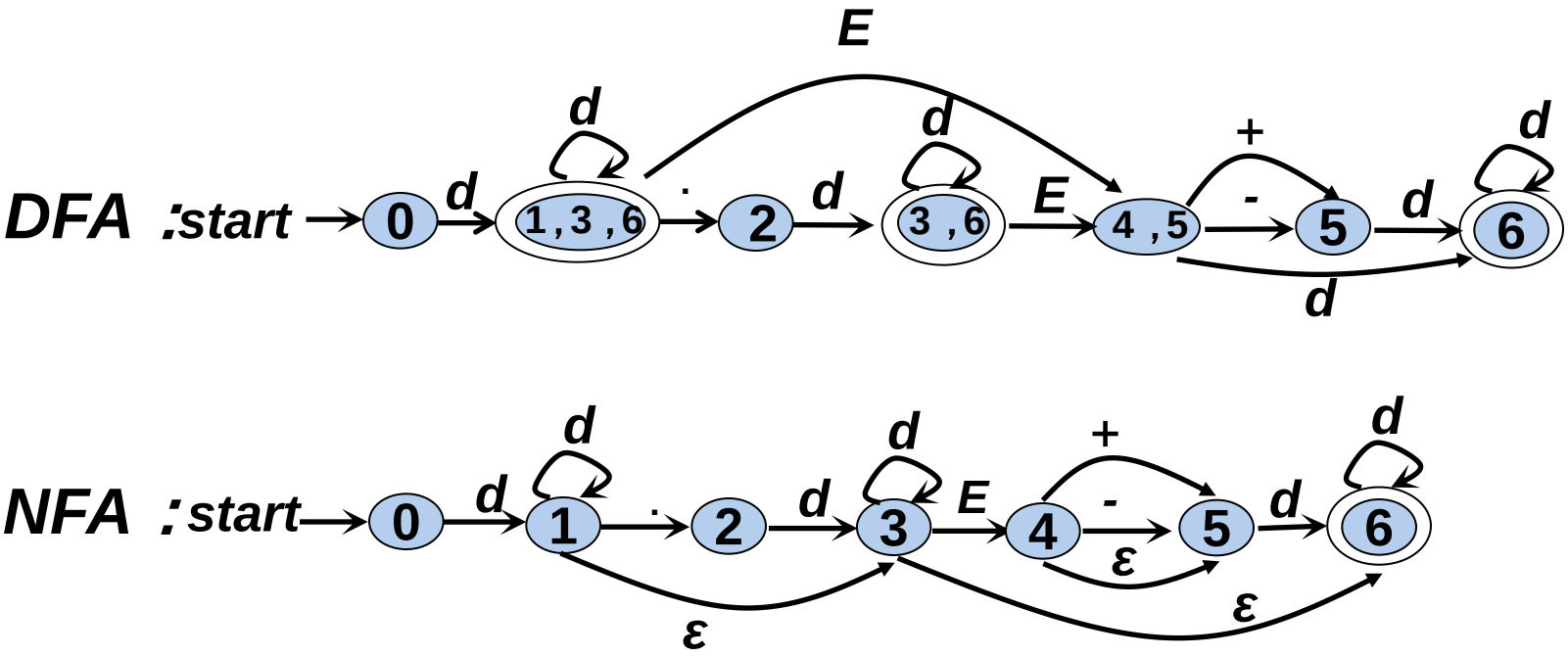
识别无符号数的*DFA*

- ***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？

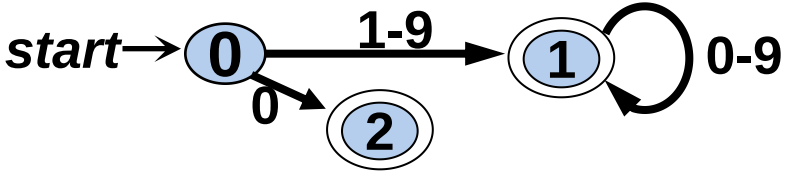


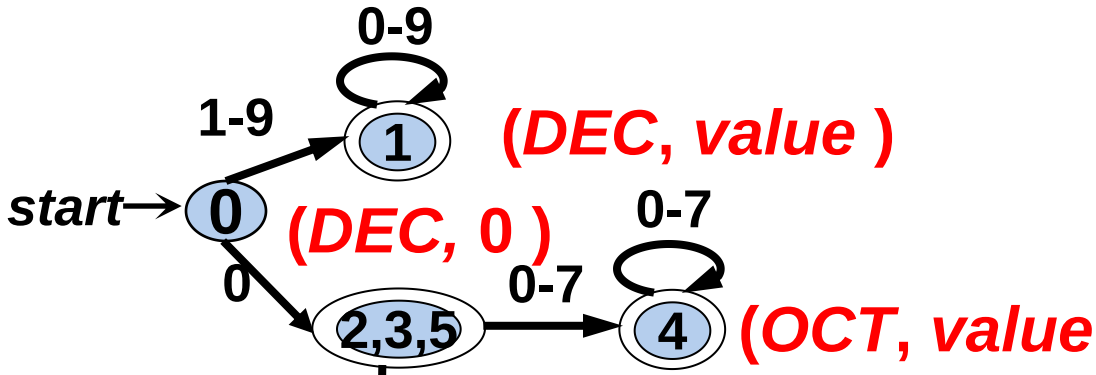
识别无符号数的DFA


DFA :
start
0
d
1,3,6
d
.
2
d
3,6
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

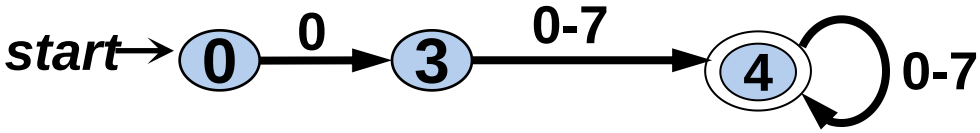
识别各进制无符号整数的*DFA*

DEC → (1|...|9)(0|...|9)* |0

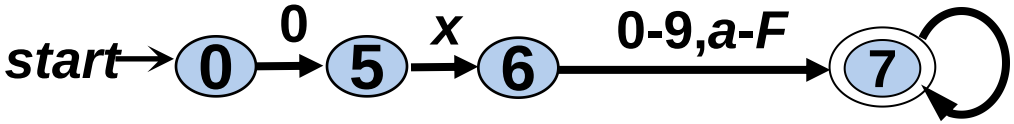




OCT → 0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)*

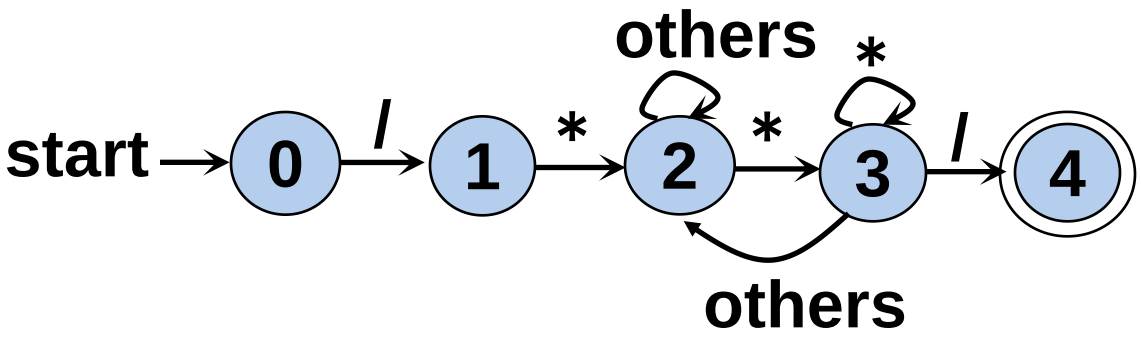


HEX → 0*x*(0|1|...|9|*a*|...|*f*|*A*|…|*F*)(0|...|9|*a*|...|*f* |*A*|…|*F*)*



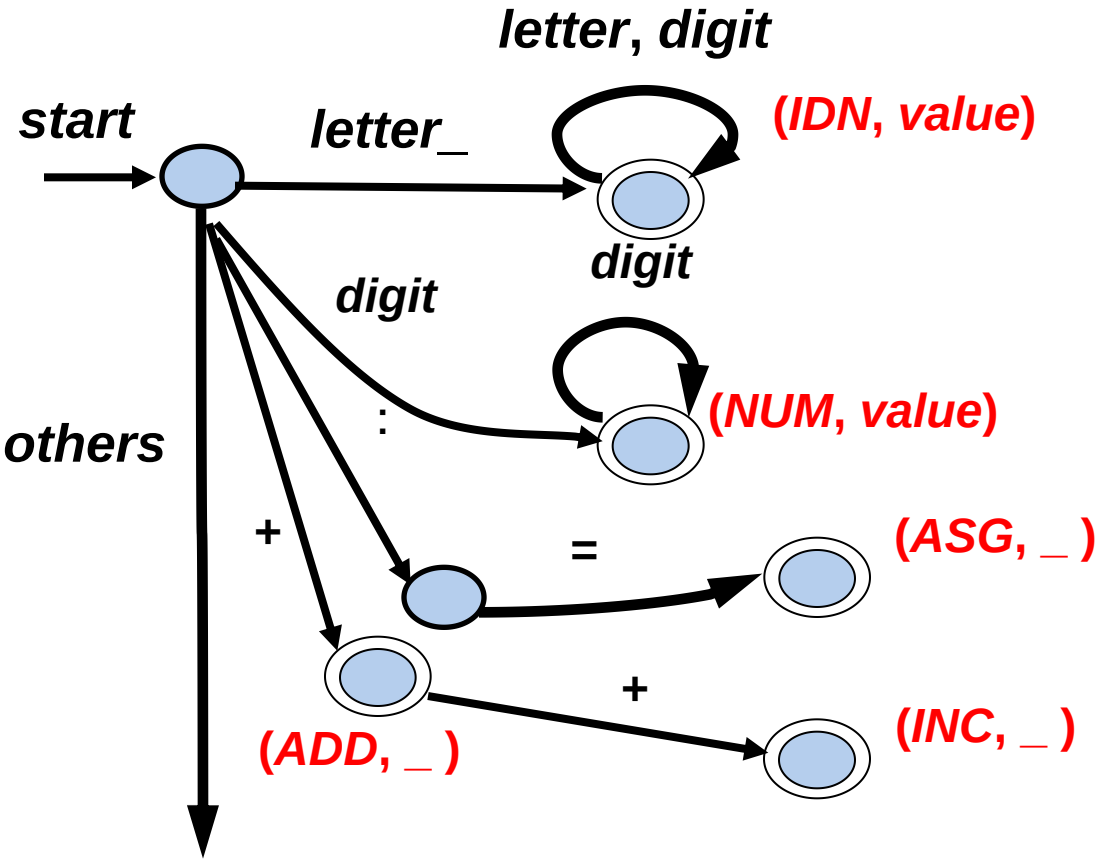
识别注释的*DFA*

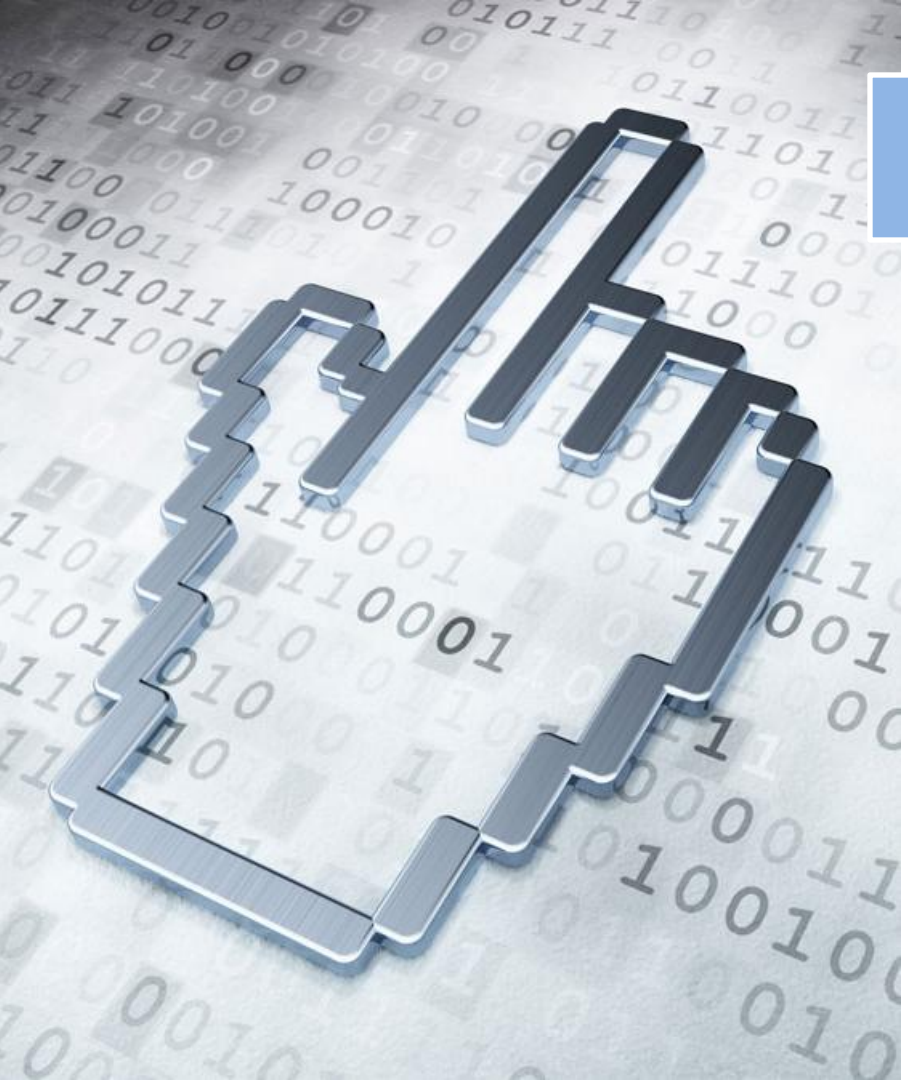
***others*:除了*和 / 之外的其它字符**



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*





提纲

3.4 词法分析阶段的错误处理

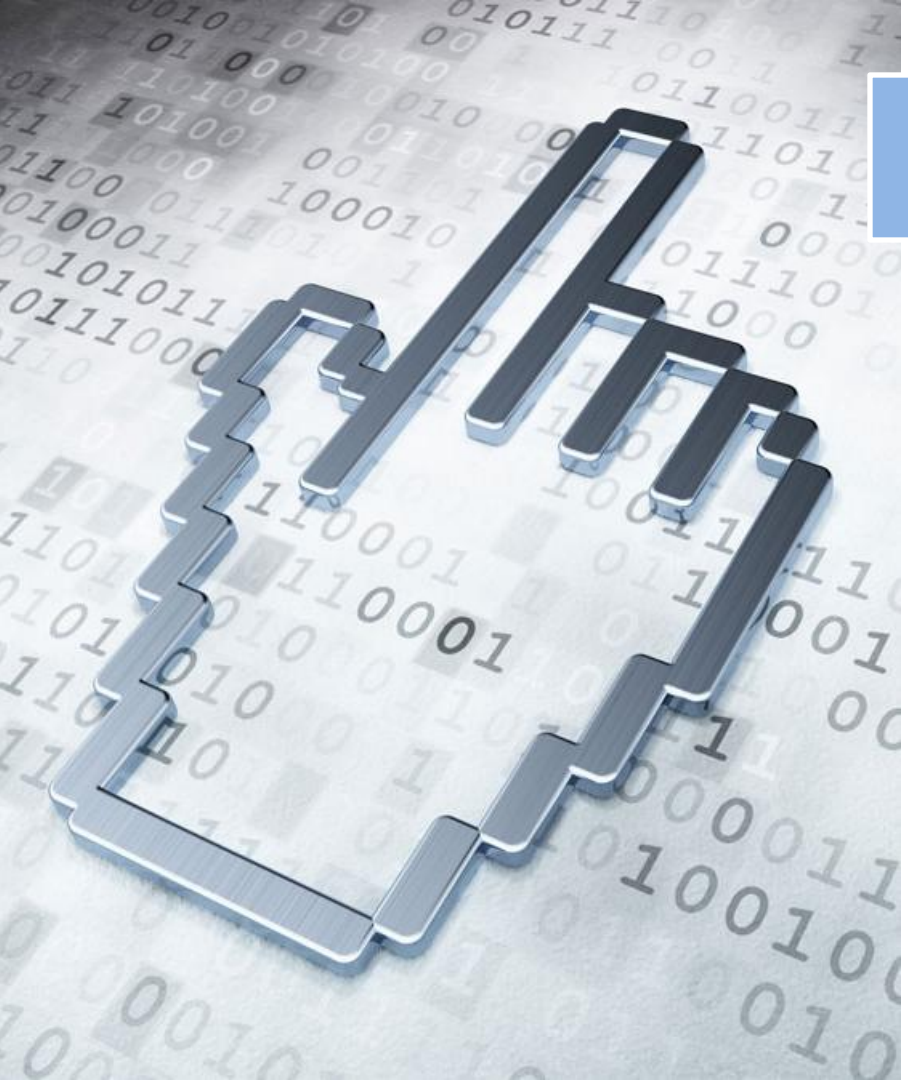
- 词法错误的类型
  - 非法字符
    - 例：~ @
  - 单词拼写错误
    - 例：*int i* = 0*x*3*G*; *float j* =1.05*e*; *G*和e识别为标识符
- 词法错误检测
  - 如果当前状态与当前输入符号在转换表对应项中的信息为空，而当前状态又不是终止状态，则调用错误处理程序

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

自动生成词法分析器的基本思想

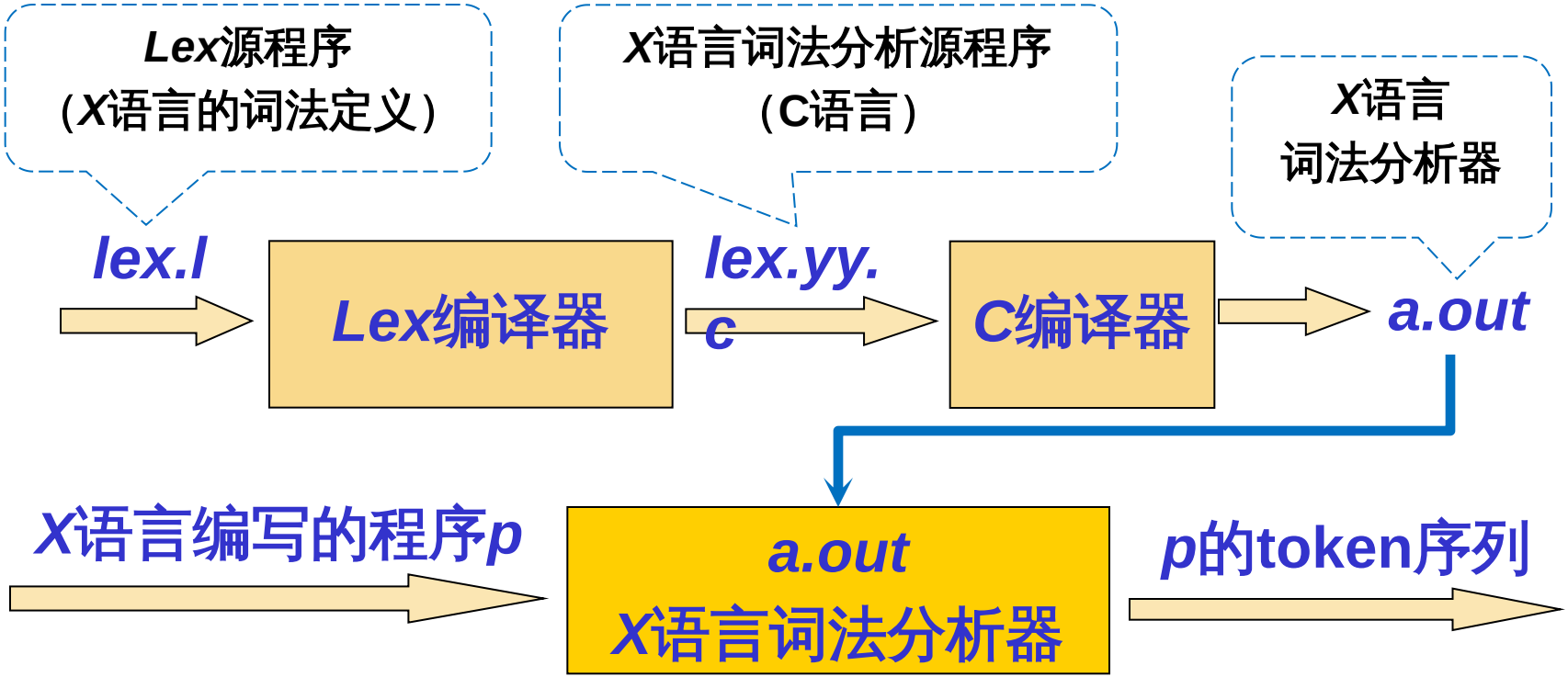
- 接收一组自定义RE 和 识别单词后的动作代码
  - 自动调用算法转换为*DFA*转换表
  - 将固定的*DFA*模拟器与此*DFA*转换表集成，即可生成词法分析器程序

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

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++; linesars++;}
.           { 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

模式(RE) {动作}

用户自定义代码

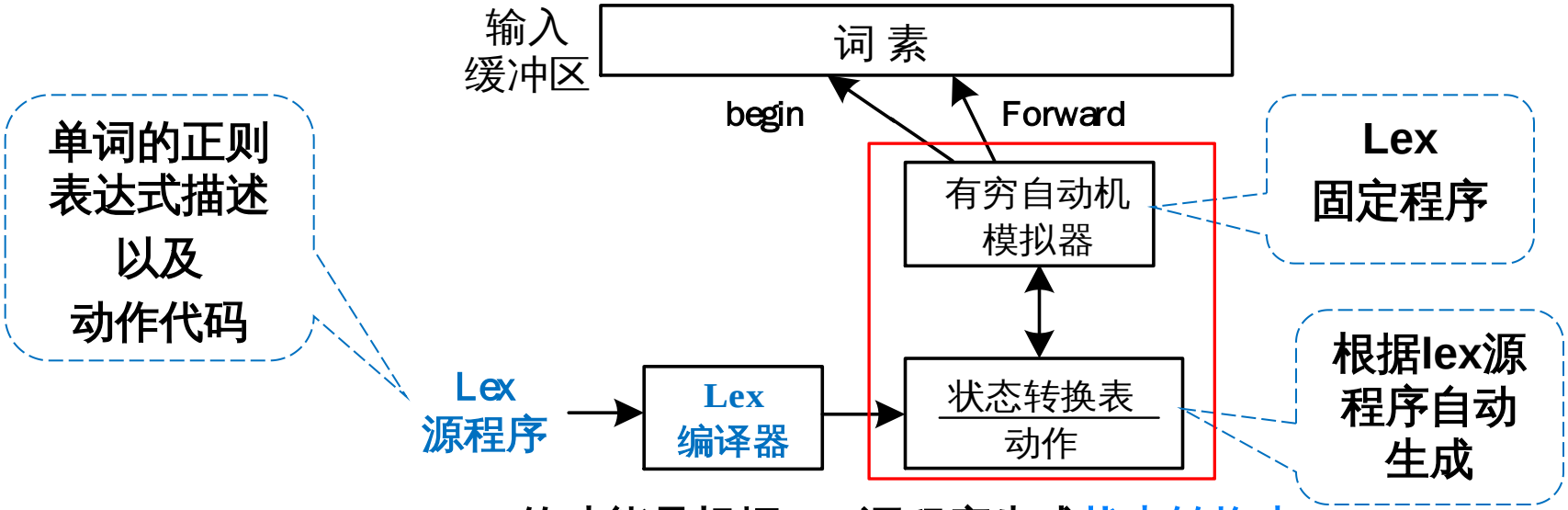
所有内容都被直接复制到文件lex.yy.c中

两组%%将代码分为三部分

Lex的实现原理

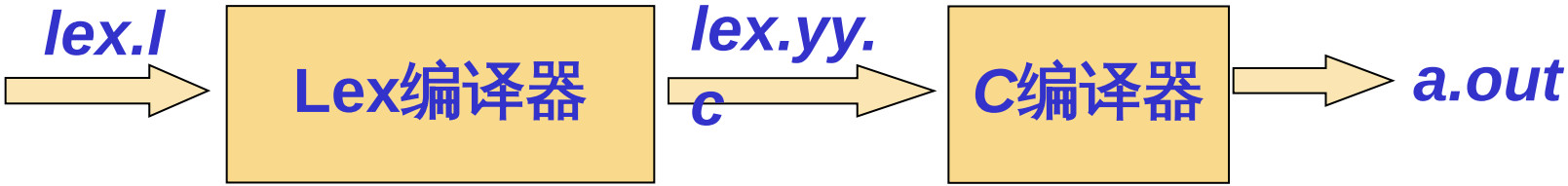
Lex的功能是根据Lex源程序构造一个词法分析程序，

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

生成的词法分析器Lex.yy.c的结构



- Lex.yy.c，其核心是函数 yylex() ，包含
  - DFA转换表（根据正则表达式自动生成）
  - 固定的模拟DFA的程序（事先编制）
  - lex源代码中的C函数和动作（从lex源代码复制）
  - 根据动作代码定义可以返回词法单元

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

实验一词法分析与语法分析

- 要求：实验指导书-词法分析与语法分析
- 利用Flex实现词法分析器
  - 补全C--的词法规则
  - 学习Flex使用，编写Flex源程序并编译
  - 编写main函数调用词法分析函数yylex对输入c--源代码文件进行词法分析，并输出词法错误。（后面与语法分析集成）
  - 对词法分析器进行测试
    - 实践参考书中的代码和设计测试程序代码
- 必须通过1.1.6（必做内容）

实验提示——系统思维

- 三个实验任务是递近式，最终要构建一个完整的编译系统。
- 注意系统地设计代码结构，将程序功能模块化，并优化模块关系和接口。
- 功能模块要尽量松耦合，因为在某些实验中的输出或实现的任务（例如输出语法分析树），在后续实验中不再需要，要考虑模块易于剥离和维护。

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

课程主要内容

1．绪论	（2学时）
2. 语言及其文法	（2学时）
3．词法分析	（3学时）
4．语法分析	（9学时）
5．语法制导翻译	（6学时）
6．中间代码生成	（7学时）
7．运行时的存贮组织	（3学时）
8．代码优化	（6学时）
9．代码生成	（2学时）

结束