第三章　词法分析

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

3.1 词法分析器的角色

- 读入源程序字符流、组成词素，输出词法单元序列。
- 过滤空白、换行、制表符、注释等。
- 将词素添加到符号表中。

基本术语

词法单元

- 源代码字符串集的分类
- <identifier, count>, <number,80>

模式

- 描述“字符串集如何分类为单词”的规则
- 正则表达式，[A-Z]*.*

词素

- 程序中实际出现的字符串，与模式匹配，分类为单词
- i，count，name，60…

基本术语（续）

词法单元	词素实例	非正式描述
else	else	字符 e, l, s, e
if	if	字符 i, f
comparison	<, <=, =, < >, >, >=	< 或 <=或=或< >或>=或>
id	pi, count, D2	字母开头的字母或数字
number	3.1416, 0, 6.02E23	任何数字常量
literal	“core dumped”	在两个“之间，除”以外的任何字符

大部分词法单元的类别

- 每个关键字有一个词法单元。一个关键字的模式就是该关键字本身
- 表示运算符的词法单元。可以表示单个运算符，也可以表示一类运算符
- 表示所有标识符的词法单元
- 一个或多个表示常量的词法单元，比如数字和字面值字符串
- 每一个标点符号有一个词法单元，比如左右括号、逗号、分号

词法单元的属性

- 词素的更多信息
- 词法单元的名字——影响语法分析
- 词法单元的属性——影响翻译
- 用二元组<记号，属性值>表示；属性一般用符号表的指针来表示

词法单元的属性

- 例如，position := initial + rate * 60
  - ＜id，指向符号表中position条目的指针>
  - ＜assign _ op ＞
  - ＜id，指向符号表中initial条目的指针>
  - ＜add_op>
  - ＜id，指向符号表中rate条目的指针>
  - ＜mul_ op>
  - ＜num，整数值60>

词法错误

- 较少：词法分析是对源程序极为局部化的视角
- fi (a == f(x)) …——词法分析无法发现
- 什么情况下发生？——剩余输入的前缀无法与任何一个模式相匹配
- 可能的错误修复方法
  - 删除、插入字符
  - 替换、交换字符
  - 最短编辑距离

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

3.2 缓冲技术

加快程序读入速度的方法

若干个字符 ⟹ 正确的词素，

譬如： = 或< 可能是 == 或 <=

至少向前看一个字符

单字符I/O＋预读和回退——效率低下

磁盘 —块I/O→ 缓冲区 —单字符读取→ 词法分析器

3.2 缓冲技术

- 三种实现方式
  1. 自动生成工具——Lex，生成工具提供读取输入和缓冲的函数
  2. 高级语言手工编码，利用高级语言提供的I/O函数
  3. 汇编语言编程，直接访问磁盘
- 1 → 3，性能↗，实现难度↗
- 唯一读取文件的阶段，值得优化

方案：

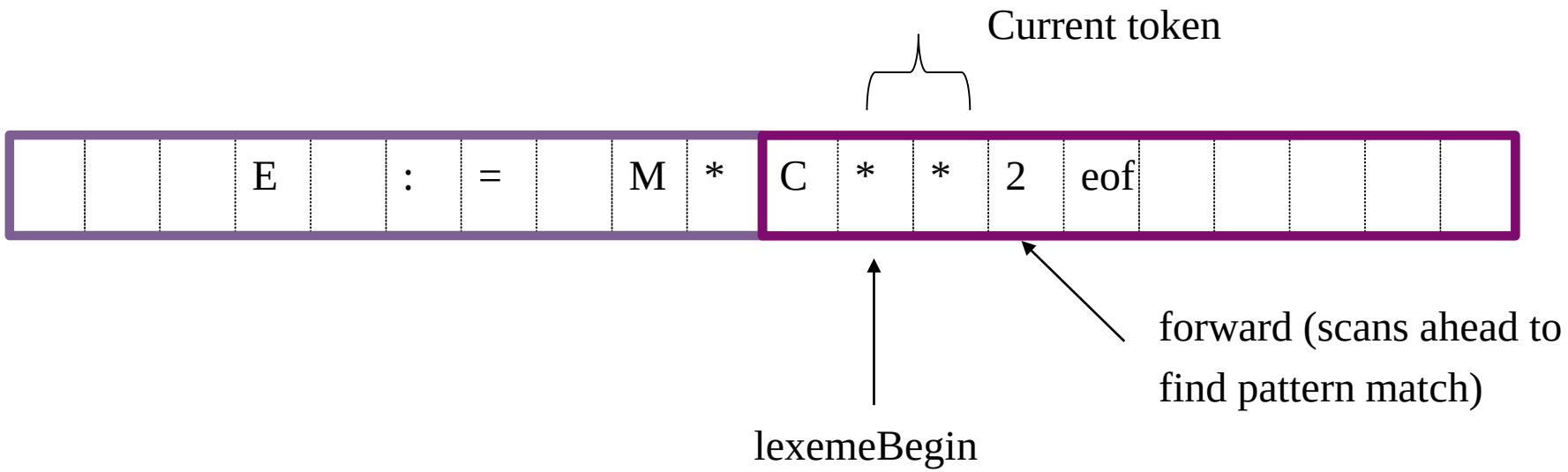
1. 双缓冲区方案
2. 哨兵标记

3.2.1 缓冲区对

双缓冲技术

- 缓冲区分成两个部分，N个字符（N = 1024 / 4096）
- 每次读N个字符至缓冲区，不足N个字符eof.
- 指针lexemeBegin：指向当前词素的开始处
- 指针forward：一直向前扫描直至匹配某个模式

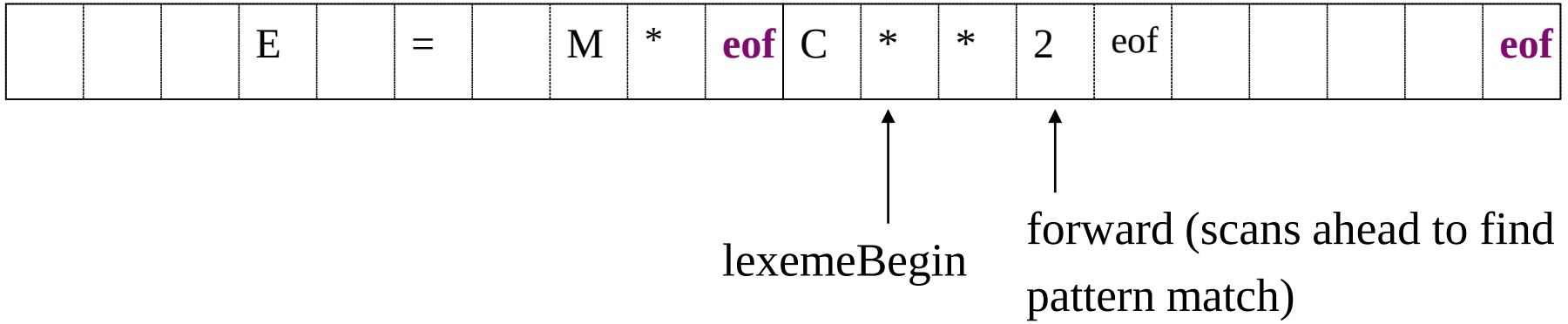
双缓冲技术图示


Current token
E
:
=
M
*
C
*
*
2
eof
forward (scans ahead to find pattern match)
lexemeBegin

双缓冲技术伪代码

```
if forward 位于第一半区的末端 then begin
        装载第二个半区 ;
        forward := forward + 1
end
else if forward 位于第二个半区的末端 then begin
        装载第一个半区 ;
        forward 移动到第一个半区的开始
end
else forward := forward + 1
```

3.2.2 哨兵标记



每个缓冲区末端添加标记

——哨兵(sentinel): eof

减少条件判断

哨兵技术的伪代码

```
forward : = forward + 1 ;
if forward↑ = eof then begin
   if forward 位于第一半区的末端 then begin
        装载第二个半区 ;
        forward : = forward + 1
   end
   else if forward 位于第二个半区的末端 then begin
        装载第一个半区 ;
        forward 移动到第一个半区的开始
   end
   else / * 缓冲区内部的eof意味着输入的结束 * /
      结束词法分析
end
```

← Block I/O

← Block I/O

2nd eof ⇒ 输入结束！

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

3.3 词法单元的描述

正则表达式（正规式）：

描述词素模式的重要表示方法

高效地描述在处理词法单元时要用到的模式类型

正则表达式 (regular expression)

C语言标识符： **letter _(letter_ | digit)***

正则表达式r

——表示语言L(r)

——利用一组规则（运算）来构造

- 指明如何用符号表中符号构成特定符号串集合
- 基本、简单的正则表达式如何递归地构成复杂的正则表达式

正则表达式定义规则

归纳基础

字母表Σ上的正则表达式r的定义规则，以及r所表示的语言L(r)定义如下：

1. ε是正则表达式，表示语言{ε}
2. 若a∈Σ，则a是正则表达式，表示语言{a}

正则表达式定义规则

归纳步骤

3. r, s为正则表达式，表示语言L(r)和L(s)，则

优先级降低

a) (r) | (s)是正则表达式，表示语言$L(r) \cup L(s)$
b) (r)(s)是正则表达式，表示语言L(r)L(s)
c) $(r)^*$是正则表达式，表示语言$(L(r))^*$
d) (r) 是正则表达式，表示语言L(r)

a | b* c = (a)|((b)*(c))

例

- Σ={a, b}

1. a | b　　{ a, b}
2. (a | b)(a | b)　　{ aa, ab, ba, bb }
3. a^*　　{ε, a, aa, aaa, … }
4. $(a \mid b)^*$　　{ 空串及所有由a、b组成的符号串 }
5. $a \mid a^*b$　　{ a, b, 所有以多个a开头，后跟一个b的符号串 }

- 正则集合：正则表达式定义的语言
- 正则表达式等价（equivalent）：r = s ⟷ 表示的语言相同，L(r) = L(s)

$(a \mid b)^* = (a^* b^*)^*$　{ε, a, b, aa, ab, ba, bb, …}

正则表达式的代数定律

定律	描述
$r \mid s = s \mid r$	| 满足交换率
$r \mid (s \mid t) = (r \mid s) \mid t$	| 满足结合率
$(r\ s)\ t = r\ (s\ t)$	连接满足结合率
$r\ (s \mid t) = r\ s \mid r\ t$ $(s \mid t)\ r = s\ r \mid t\ r$	连接和 | 满足分配率
$\varepsilon\ r = r$ $r\ \varepsilon = r$	ε是连接运算的单位元
$r^* = (r \mid \varepsilon)^*$	* 和ε间的关系
$r^{**} = r^*$	* 是幂等的

3.3.4 正则定义

为正则表达式指定名字

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

d_i是不同的名字，并且不在Σ 中

r_i是$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$（即基本符号与前面定义的名字）上的正则表达式

例：C 语言标识符

(_ | A | … | Z | a | … | z)(_ | A | … | Z | a | … | z | 0 | … | 9)*

letter_ → A | B | C | … | Z | a | b | … | z | _
digit → 0 | 1 | 2 | … | 9
id → letter_(letter_ | digit)*

例：C语言无符号数

例如：5280，0.012，6.33E4， 1.8E-4

digit → 0 | 1 | 2 | … | 9

digits → digit digit*

optional_fraction → . digits | ε

optional_exponent → (E (+ | - | ε) digits) | ε

num → digits optional_fraction optional_exponent

3.3.5 符号简写（扩展）

R^+

- one or more strings from L(R): R(R*)

R?

- optional R: (R|ε)

[abce]

- one of the listed characters: (a|b|c|e)

[a-z]

- one character from this range:(a|b|c|d|e|…|y|z)

[^ab]

- anything but one of the listed chars

[^a-z]

- one character not from this range

例：用简写

C语言的标识符集合

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

$$digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$$

$$id \rightarrow letter\_ \; (\; letter\_ \mid digit)^*$$

简化为：

$$letter\_ \rightarrow [A\text{-}Za\text{-}z\_]$$

$$digit \rightarrow [0\text{-}9]$$

$$id \rightarrow letter\_ \; (\; letter\_ \mid digit)^*$$

例：用简写

- C语言无符号数的集合

  $digit \rightarrow 0 \mid 1 \mid \ldots \mid 9$

  $digits \rightarrow digit\ digit^*$

  $optionalFraction \rightarrow .digits \mid \varepsilon$

  $optionalExponent \rightarrow (\mathbf{E}\ (\ + \mid - \mid \varepsilon\ )\ digits\ ) \mid \varepsilon$

  $number \rightarrow digits\ optionalFraction\ optionalExponent$

  简化为：

  $digit \rightarrow [0\text{-}9]$

  $digits \rightarrow digit^+$

  $number \rightarrow\ digits\ (.digits)?\ (\mathbf{E}(+|-)?\ digits)?$

正则表达式练习题

描述正则表达式表示的语言

- 0*10*10*10*：

- ((ε | 0) 1*)*：

设计语言的正则表达式

- 能被5整除的10进制整数

- 不包含连续的0的01串

练习题

3. r, s为正规式，表示语言L(r)和L(s)，则

优先级降低

a) (r) | (s)是正规式，表示语言L(r) ∪ L(s)
b) (r)(s)是正规式，表示语言L(r)L(s)
c) $(r)^*$是正规式，表示语言$(L(r))^*$
d) (r) 是正规式，表示语言L(r)

描述正则表达式表示的语言

- 0*10*10*10*：
- ((ε | 0) 1*)*：
- (ε 1*| 01*)*
- (1*| 01*)*

包含3个1的01串

所有01串

$r(s\mid t)=rs\mid rt$ $(s\mid t)r=sr\mid tr$	连接和 | 满足分配率
$\varepsilon r=r$ $r\varepsilon=r$	ε是连接运算的单位元

练习题

设计语言的正则表达式

- 能被5整除的10进制整数
- 0，5，10，15，20，25，…，105，1000，...

[1-9][0-9]*(0 | 5) | 0 | 5

- 不包含连续的0的01串
- ε，0，1，01，10，11，010，101，110，111，…

$(1|0)^*$

$(1|01)^*(\varepsilon|0)$

$(1|01)^*0?$

$(\varepsilon\ 1|01)^*0?$

$((\varepsilon|0)1)^*0?$

$(0?1)^*0?$

非正则表达式集

正则表达式无法描述的语言

- {wcw | w是a、b组成的符号串}
- 正规式无法描述平衡或嵌套的结构

正则表达式只能表示

- 有限的重复
- 一个给定结构的无限重复

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

3.4 词法单元的识别

根据需要识别的词法单元的模式来构造出一段代码

检查输入字符串，在输入的前缀中找出一个和某个模式匹配的词素

过滤空白符

空白符也可写成正则表达式

delim → **blank | tab | newline**

ws → *delim*$^{+}$

全部正则表达式定义

Regular Expression	Token	Attribute-Value
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
< >	relop	NE
>	relop	GT
>=	relop	GE

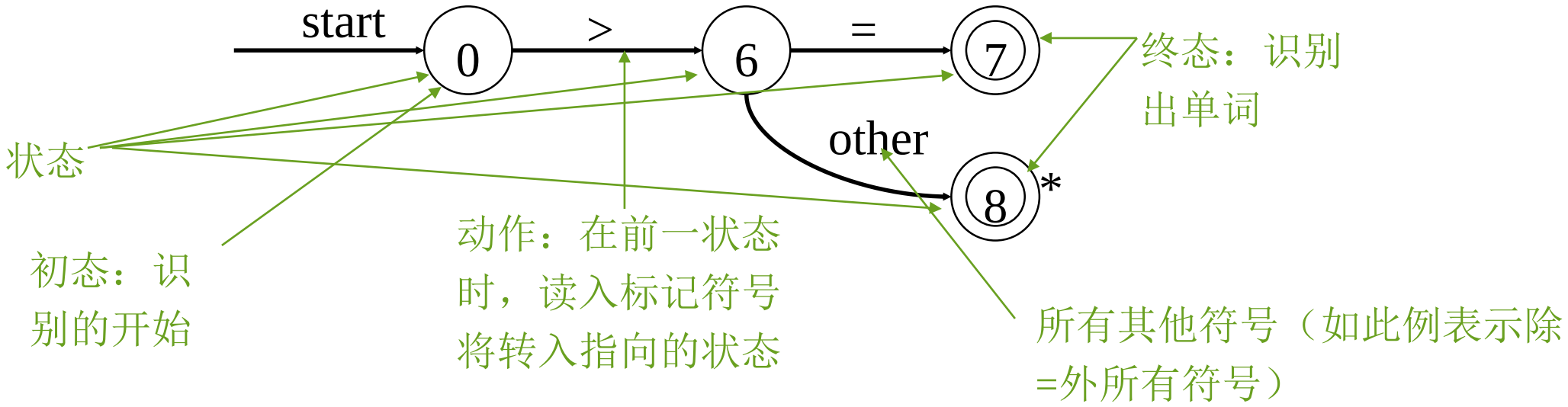
状态转换图(transition diagram, TD)

状态转换图是正则表达式的一种表示

- 状态转换图的组成
  - 状态States: 圆圈表示
    - 初始状态：识别的开始，没有出发结点的，标号为‘start’的边的表明
    - 终止状态：识别的结束，双层的圆圈表示
  - 动作：由状态间的箭头表示
  - 回退：接受状态处加‘*’，表示将forward指针回退一个位置

例：

transition diagram，识别单词



确定的：一个状态发出的不同的边不可能标记相同的符号

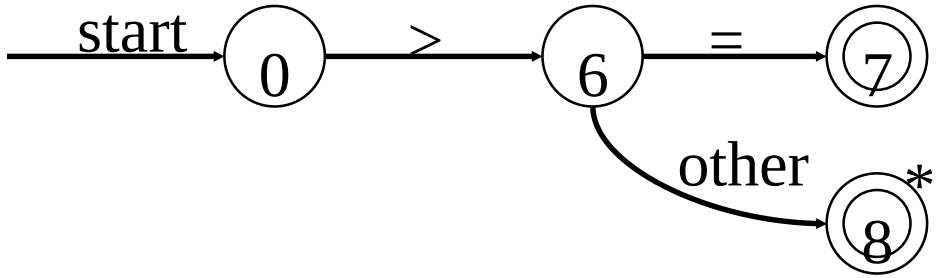
如何识别单词？

词法分析器（算法）

- 已读入符号串（前缀）+ 未读入符号串——
  与模式进行匹配

状态转换图——一种词法分析算法描述

- TD ←→模式
- 状态←→已读入符号串
- 边←→下一符号、应采取的动作

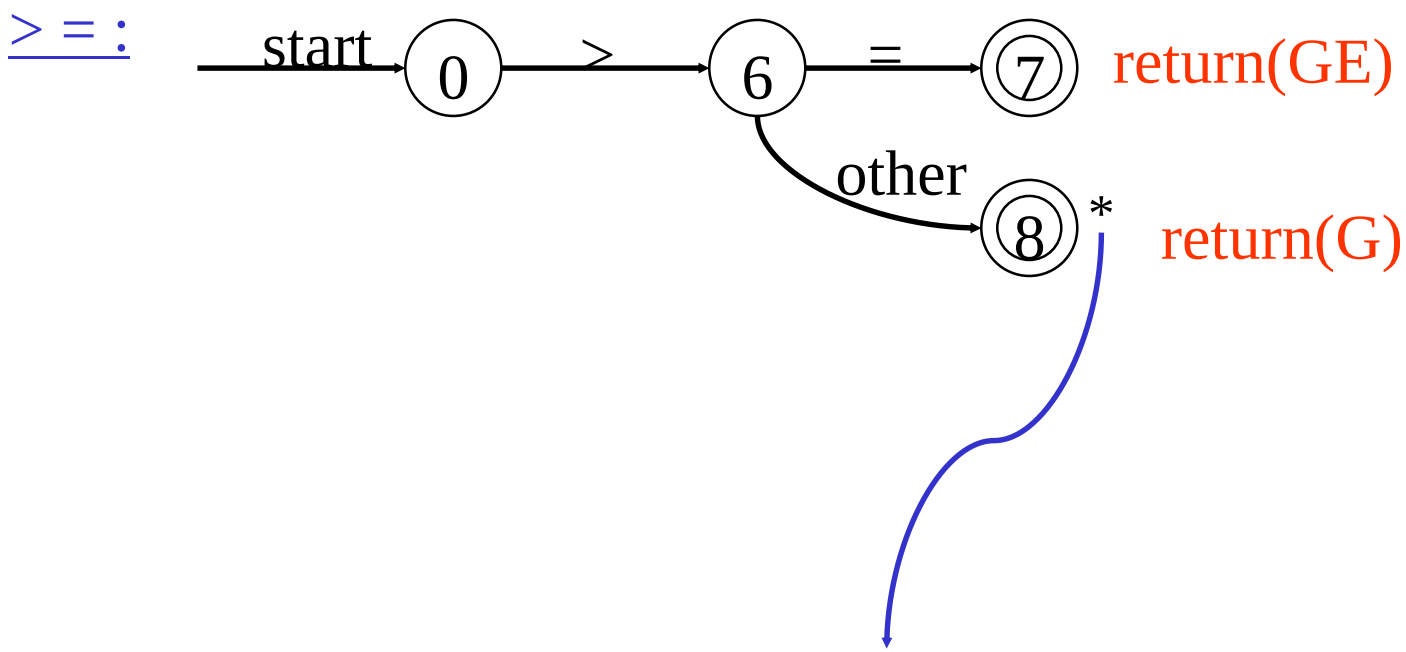


如何识别单词？

TD工作方式

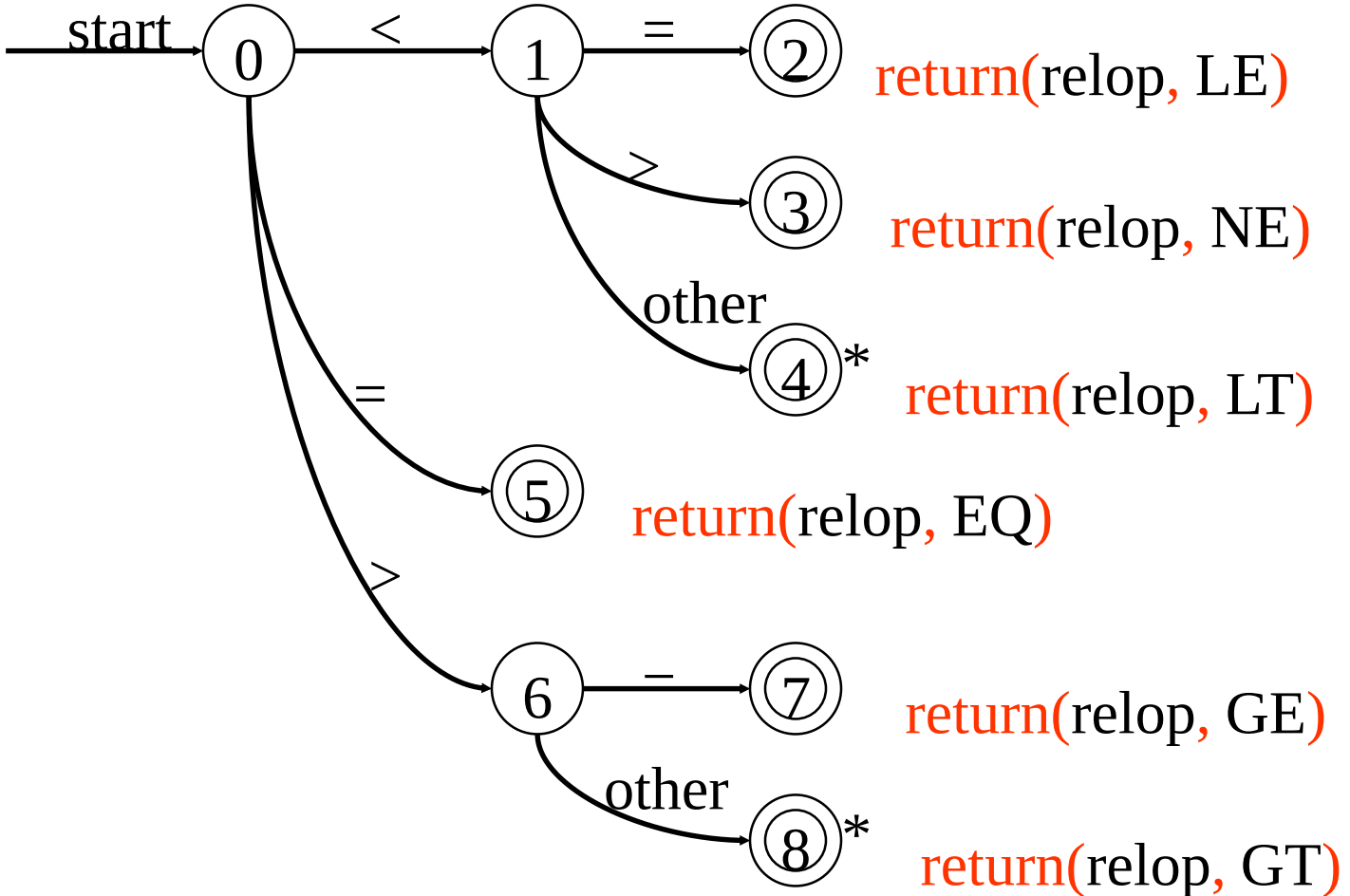
- 开始识别，初态
- 读入符号，转换状态
- 终态，接受！；无法转换，失败！

例



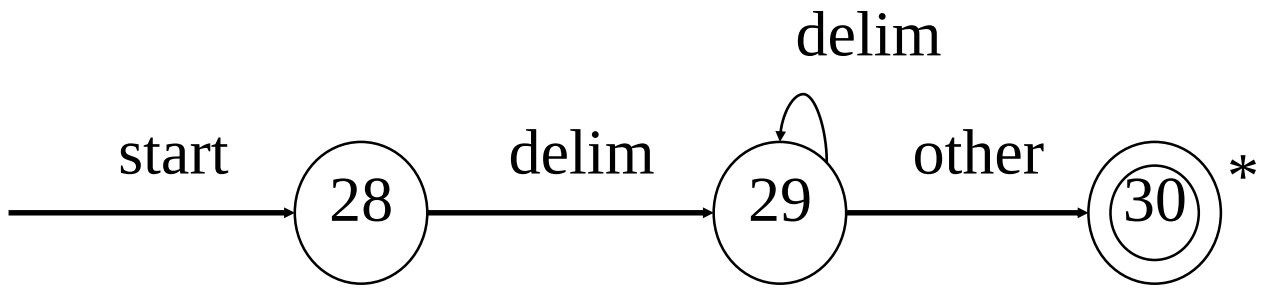
已经接受“>”，且已经多读取一个其他符号，需退回这个符号

例：所有关系运算符


start
0
<
1
=
2
return(relop, LE)
>
3
return(relop, NE)
other
4*
return(relop, LT)
=
5
return(relop, EQ)
>
6
=
7
return(relop, GE)
other
8*
return(relop, GT)

例：空白符

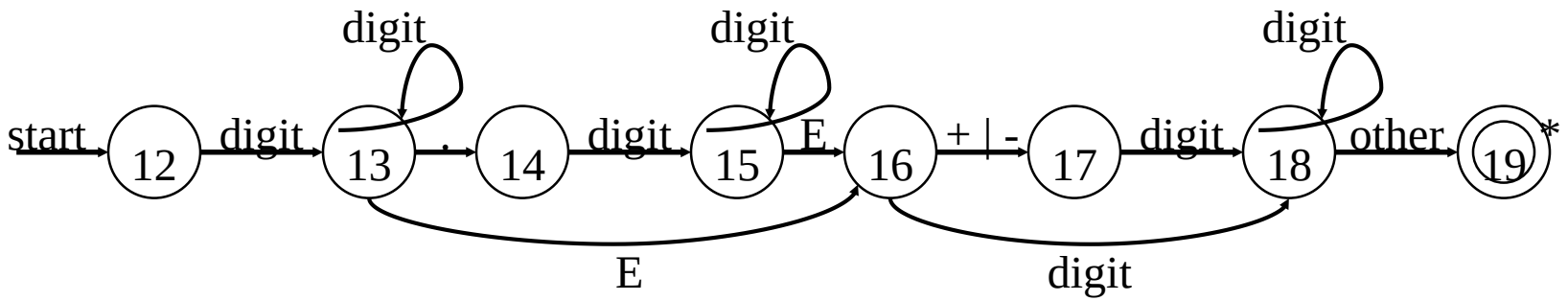
delim :



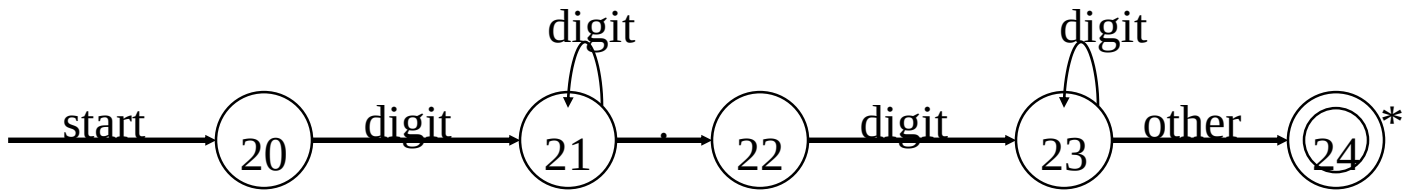
例：无符号数

$$num \rightarrow digit^{+} (.digit^{+})? (E(+|-)? digit^{+})?$$

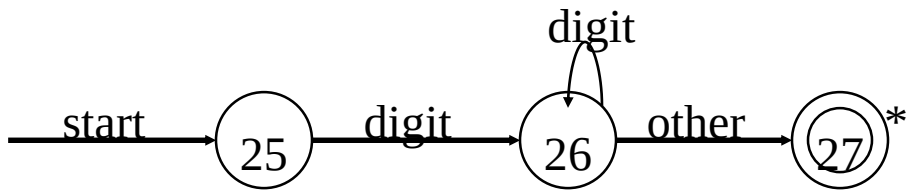
12.3E4



12.3



12



保留字的处理

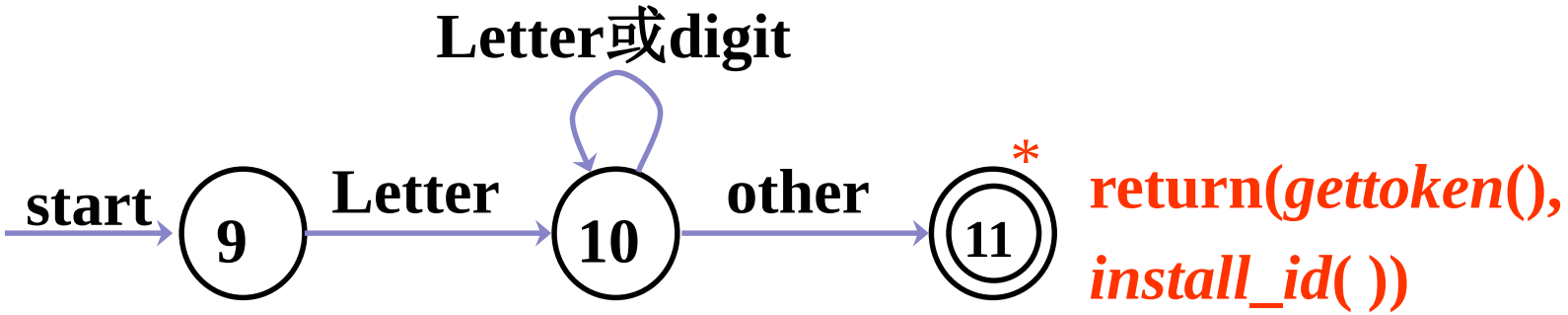
关键字/保留字与标识符一样进行匹配

- 保存在符号表或一个特殊的关键字表中，保存关键字的符号串和单词值（一般不需要）
- 当识别出标识符/关键字，查询表
- 若与某个关键字匹配，返回对应的单词，和单词值（若有的话）
- 若与任何关键字都不匹配，则认为是标识符，进行相应处理

标识符和保留字的转换图

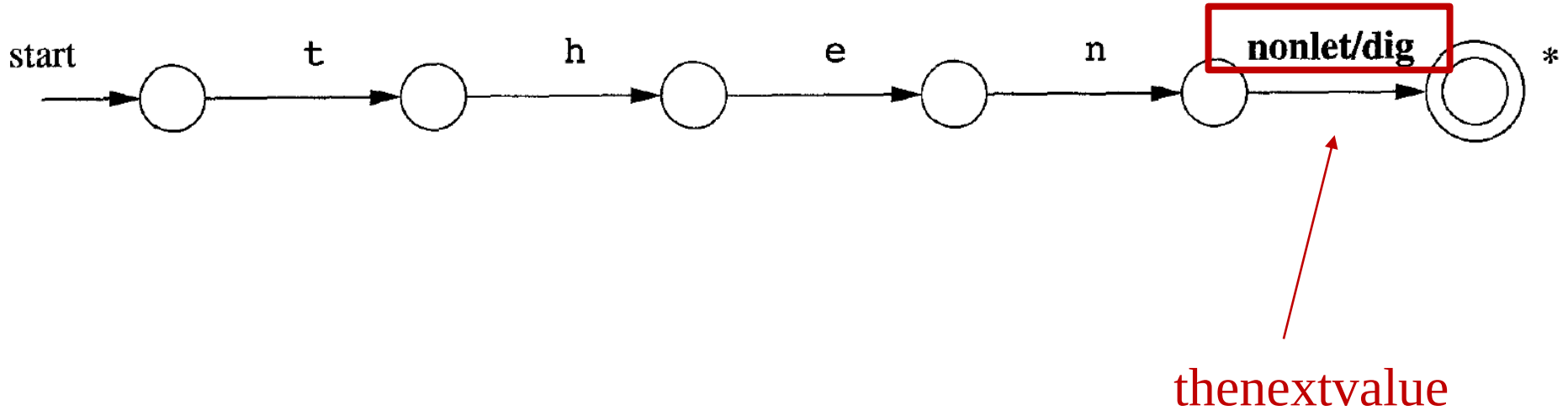
return(*gettoken*(), *install_id*( ))返回记号和属性值*lexical_value*；

- *install_id*( )首先得到该词素，再对符号表进行操作（查表及填表）；
- *gettoken*()在符号表中查找单词，若是关键字，则返回相应的*token*，否则返回*token*类型为id。



关键字的处理的两种方法

1. 初始化时将各个保留字填入符号表中(查表)
2. 为每个关键字建立单独的状态转换图



状态转换图的实现

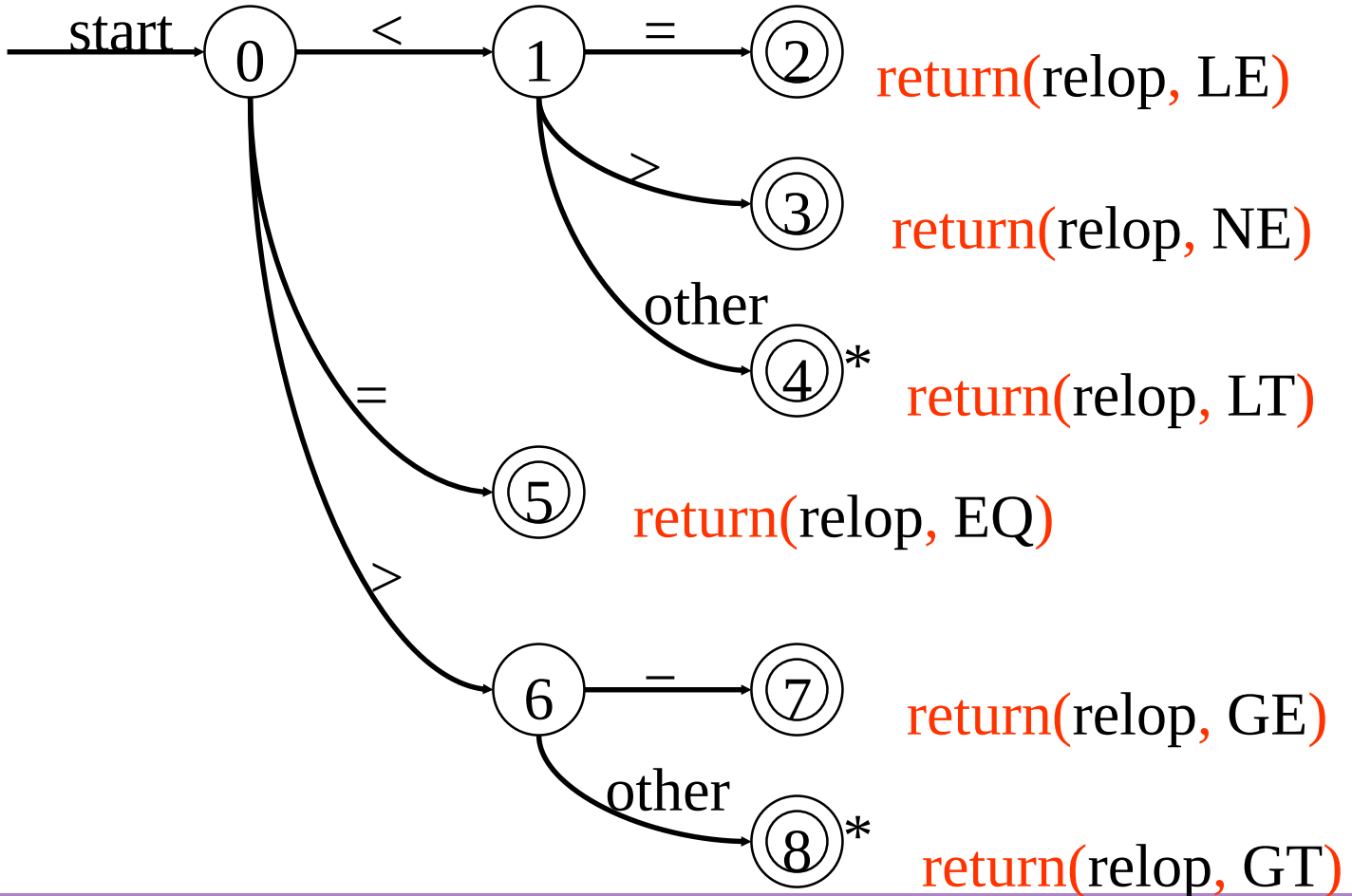
为每个状态构造一段代码

- 普通状态
  - 读取字符
  - 每条出射边的处理：根据字符转换状态
  - 其他情况，错误
- 终态
  - 返回单词，单词值

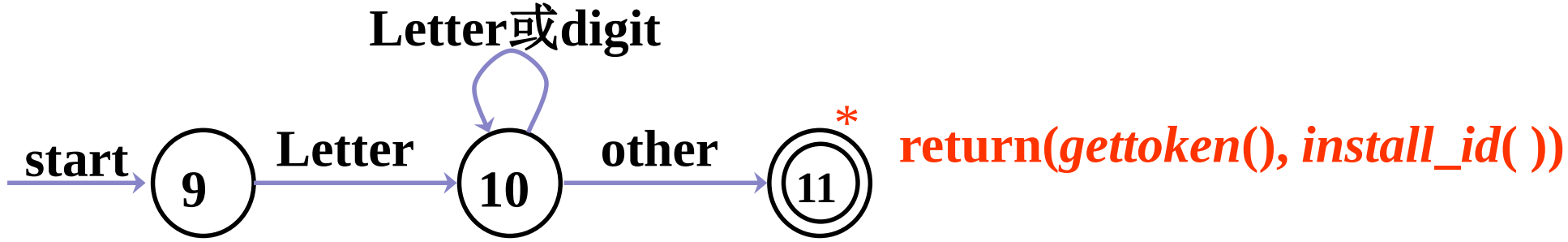
错误处理

- 尝试其他状态转换图
- 尝试完毕，与任何单词都不匹配，词法错误，错误恢复

例：所有状态转换图

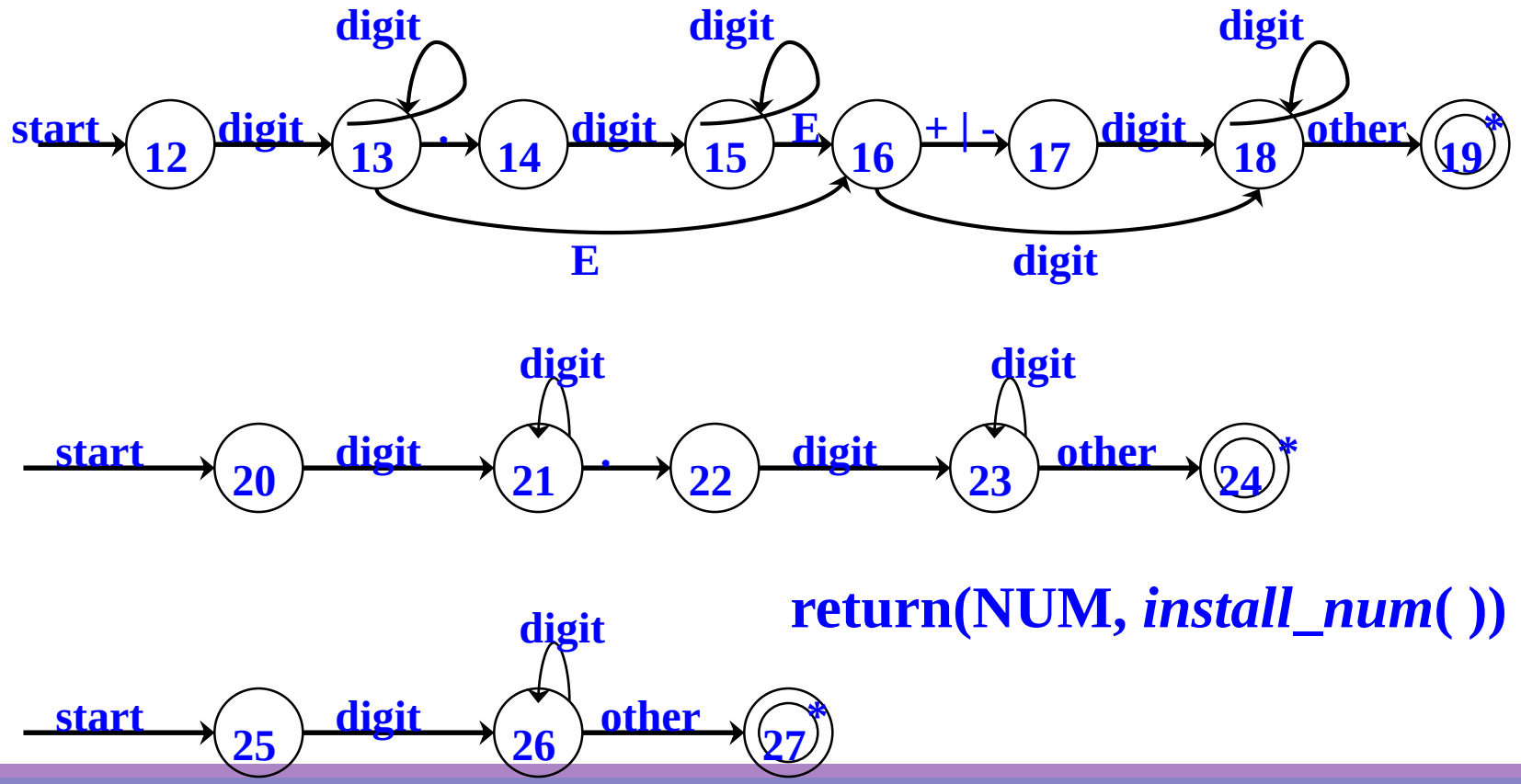

start
0
<
1
=
2
return(relop, LE)
>
3
return(relop, NE)
other
4
*
return(relop, LT)
=
5
return(relop, EQ)
>
6
=
7
return(relop, GE)
other
8
*
return(relop, GT)

例：所有状态转换图


Letter或digit
start
9
Letter
10
other
*
11
return(gettoken(), install_id( ))

例：所有状态转换图

num → digit^{+} (.digit^{+})? (E (+ | -)? digit^{+})?



实现代码

```
state = 0;
token nexttoken()
{
 while(1) {
        switch (state) {
    case 0:
         c = nextchar();
```

实现代码

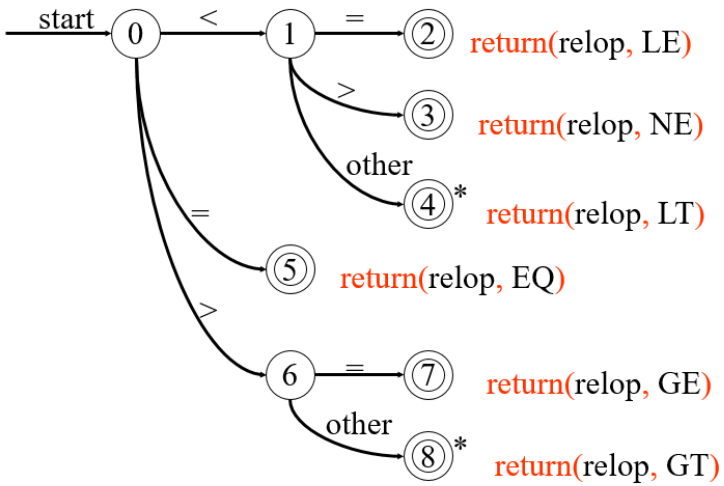
```
        /* c is lookahead character */
if (c== blank || c==tab || c== newline) {
  state = 0;
  lexeme_beginning++;
  /* advance beginning of lexeme */
}
```

作用?

实现代码

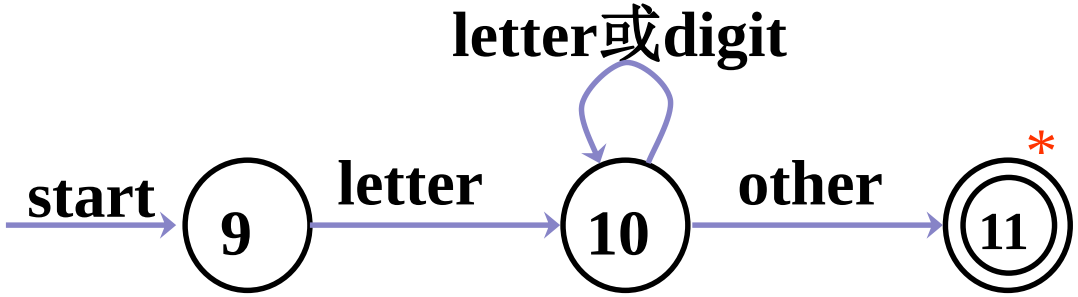
```
else if (c == '<') state = 1;
else if (c == '=') state = 5;
else if (c == '>') state = 6;
else state = fail();
break;
… /* cases 1-8 here */
```

不是比较运算符，怎么办？



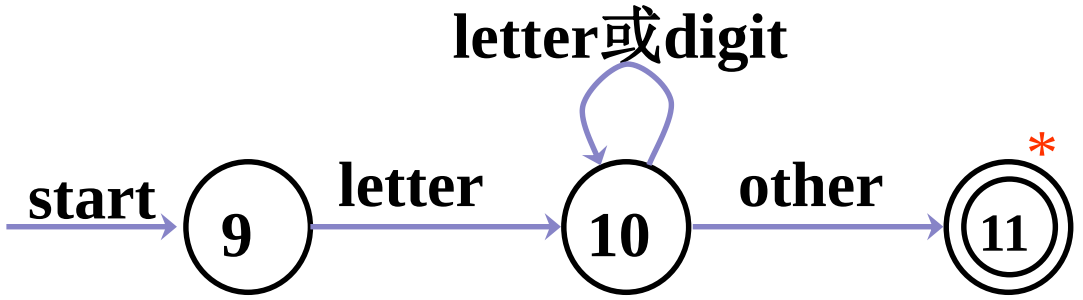
实现代码

```
case 9:
        c = nextchar();
 if (isletter(c)) state = 10;
 else state = fail();
 break;
```



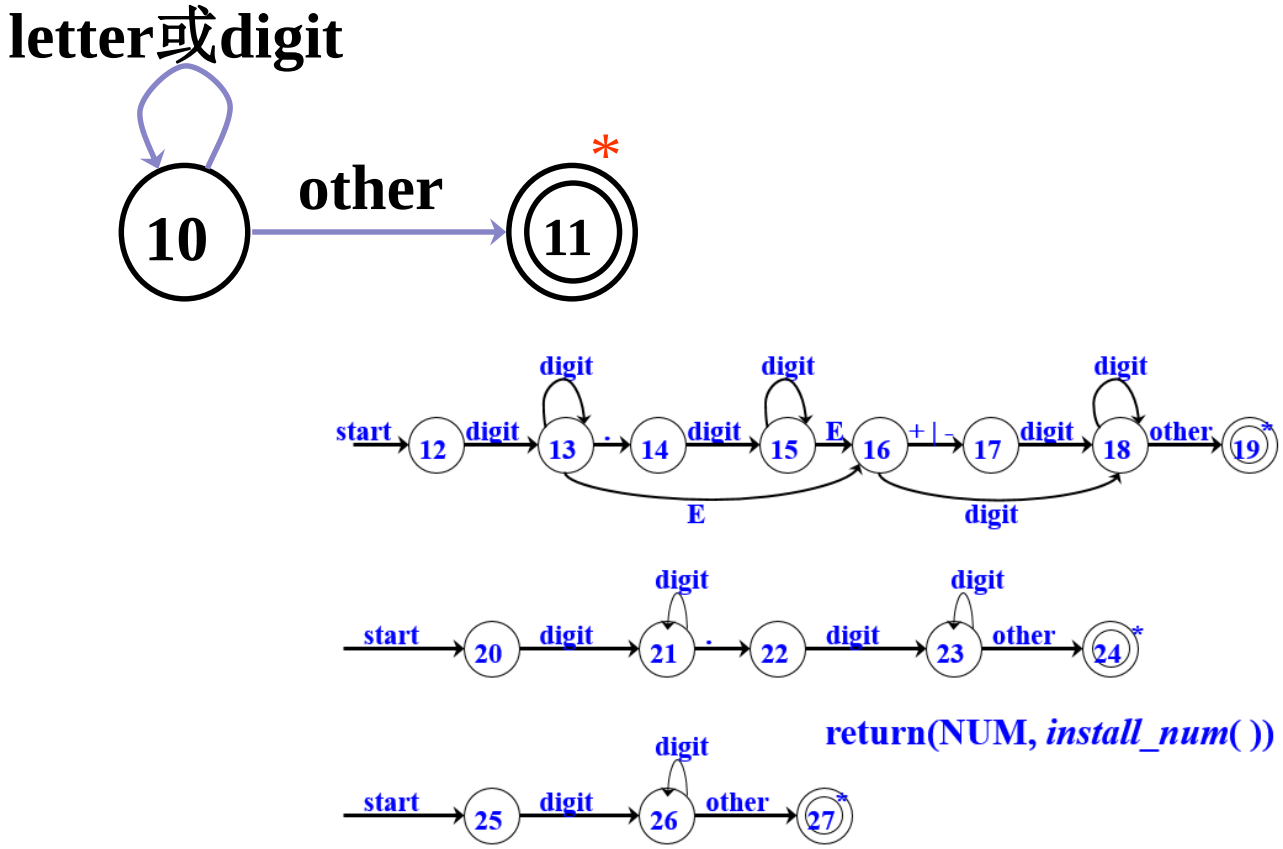
实现代码

```
case 10:
        c = nextchar();
 if (isletter(c)) state = 10;
 else if (isdigit(c)) state = 10;
 else state = 11;
 break:
```



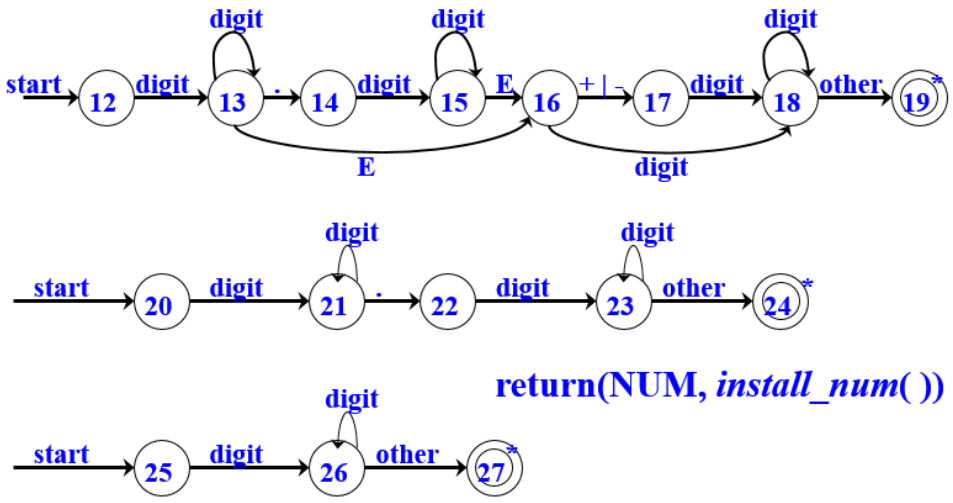
实现代码

```
case 11;
        retract(1); install_id();
 return ( gettoken() );
 … /* cases 12-24 here */
case 25:
        c = nextchar();
 if (isdigit(c)) state = 26;
 else state = fail();
 break;
```

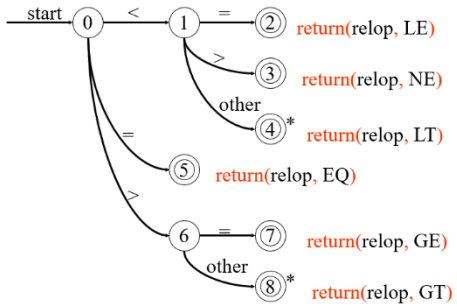


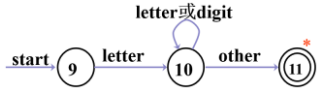
实现代码

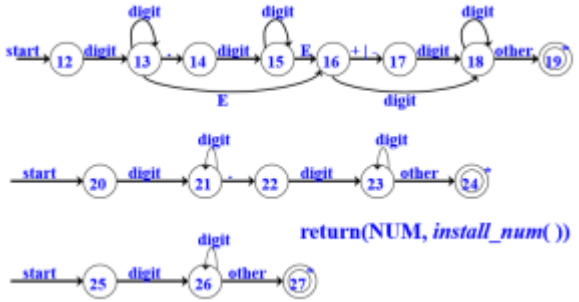
```
    case 26:
            c = nextchar();
     if (isdigit(c)) state = 26;
     else state = 27;
     break;
    case 27:
            retract(1); install_num();
     return ( NUM );
   }
  }
}
```



实现代码








```
int state = 0,  start = 0;
int lexical_value;    /* “返回” 词法值 */
int fail()
{                forward = lexeme beginning;
    switch (start) {
       case 0:   start = 9;  break;
       case 9:   start = 12; break;
       case 12:  start = 20; break;
       case 20:  start = 25; break;
       case 25:  recover();  break;
       default:   /* compiler error */
    }
    return start;
}
```

不是比较运算符，是不是标识符呢？

也不是标识符，是不是数值常量呢？

什么都不是，那就是词法错误了。

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

3.5 词法分析器的构造

有限自动机

从正则表达式到自动机

编写词法分析程序

优化词法分析程序

有限自动机

如何转换？

正则表达式 ⟶ 识别程序（recognizer）

有限自动机（finite automata）

- 不确定有限自动机

  Non-Deterministic Finite Automata (NFAs)

  一个状态对同一个输入符号，有多个可能的动作

- 确定有限自动机

  Deterministic Finite Automata (DFAs)

  一个状态对一个输入符号，至多有一个动作

NFA和DFA

都可识别正则表达式，时一空折衷

NFA

- 正则表达式 —简单→ NFA
- “不精确”，占用内存少，速度慢

DFA

- 正则表达式 —复杂→ DFA
- “精确”，占用内存大，速度快

NFA → DFA，DFA → NFA

不确定有限自动机

数学模型，表示为五元组

$M = \{ S, \Sigma, \delta, s_0, F \}$，其中

1. S：有限状态集
2. Σ：有穷字母表，其元素为输入符号
3. δ：S×Σ到S的子集的映射，即

   $\delta: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$，状态转换函数。
4. $s_0 \in S$ 是唯一的初态
5. $F \subseteq S$ 是一个终态集

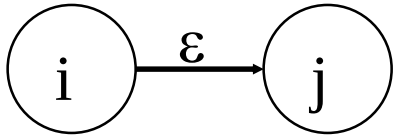
表示方式

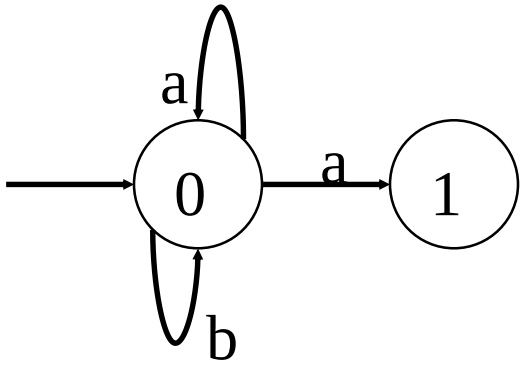
五元组

- δ用函数形式表示，δ(s, a) = S’意味着，若当前状态为s，输入符号为a时，下一个状态可以是集合S’中任意一个

转换图

- 状态（圆圈）、边、终态…　　允许ε (null)边





转换状态，但不读入符号

1. 允许ε (null)边
2. 同一个符号可以标记从同一状态出发到多个状态的多条边

转换矩阵（转换表）

- 行—状态，列—符号，表项—转换状态集
- 适合计算机实现

例：(a|b)*abb

A 五元组

$S = \{ 0, 1, 2, 3 \}$

$s_0 = 0$

$F = \{ 3 \}$

$\Sigma = \{ a, b \}$

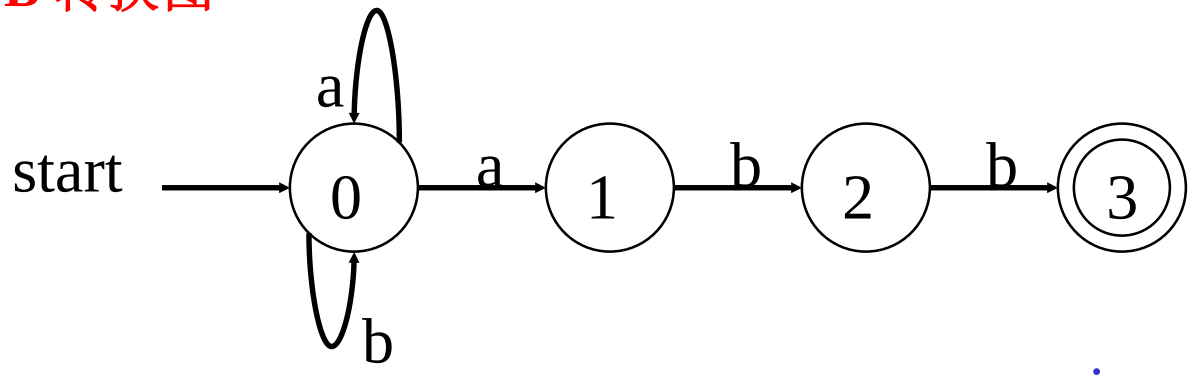
$\delta(0,a) = \{ 0, 1 \}$

$\delta(0,b) = \{ 0 \}$

$\delta(1,b) = \{ 2 \}$

$\delta(2,b) = \{ 3 \}$

B 转换图



C 转换表

state \ input	a	b
0	{ 0, 1}	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

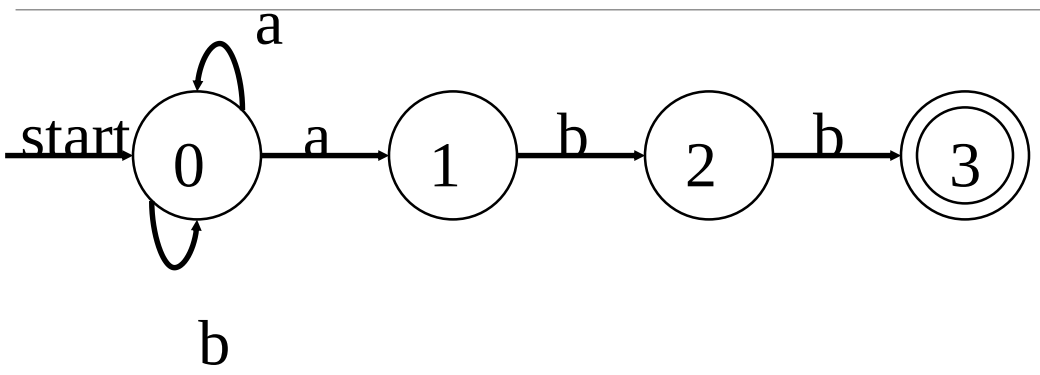
NFA如何工作？

NFA接受（accept）符号串x

↔存在一条从初态到终态的路径，路径上的符号组成x，路径中的ε将被忽略

NFA M定义的语言：M接受的符号串集合，记为L(M)

NFA如何工作？（续）



- 对给定符号串，跟踪状态转换
- 若符号读取完，且处于终态，接受

EXAMPLE: Input: ababb

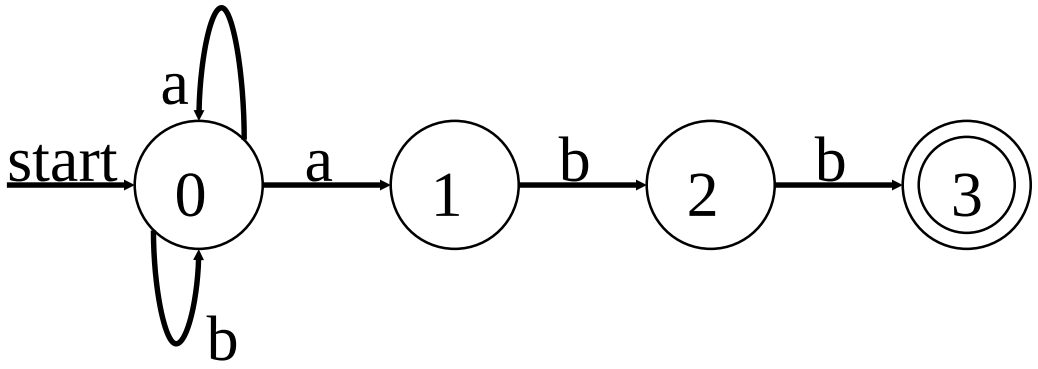
move(0, a) = 1
move(1, b) = 2
move(2, a) = ? (undefined)

REJECT !

-OR-

move(0, a) = 0
move(0, b) = 0
move(0, a) = 1
move(1, b) = 2
move(2, b) = 3
ACCEPT !

NFA如何工作？（续）

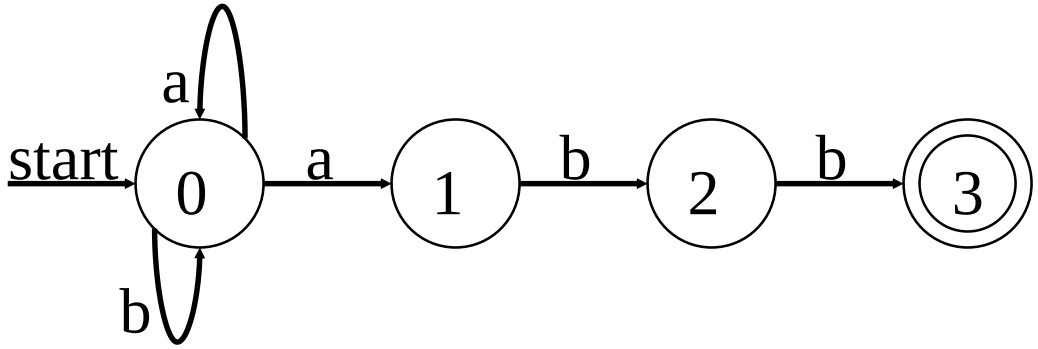


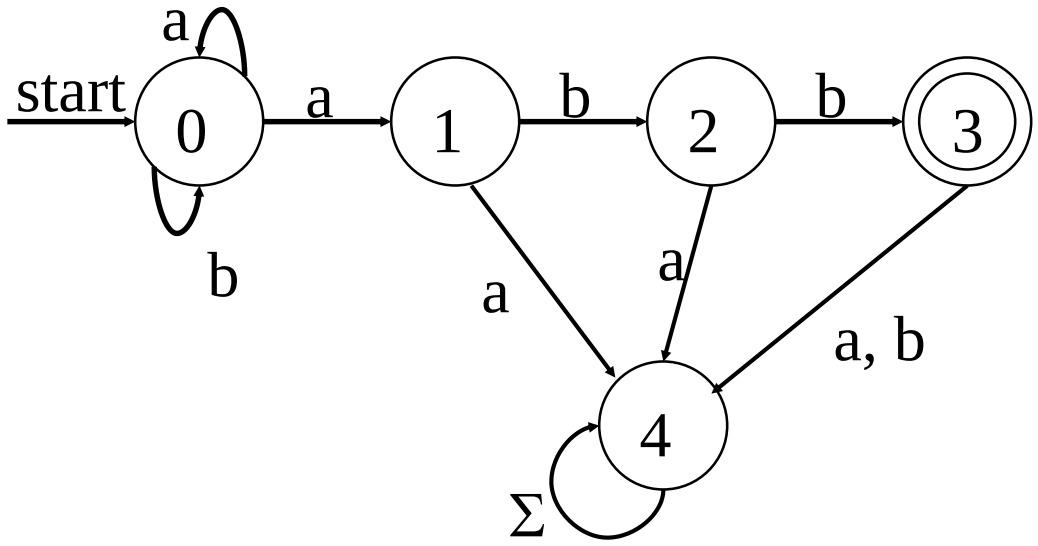
不是所有路径都表示接受

aabb

- 路径**0 → 0 → 1 → 2 → 3**表示接受
- 而**0 → 0 → 0 → 0 → 0**表示不接受

处理未定义状态转换





定义一个额外的“死状态”

所有未定义的状态转换都指向它

它自身的状态转换都指向自身

NFA与正则表达式/编译器的关系

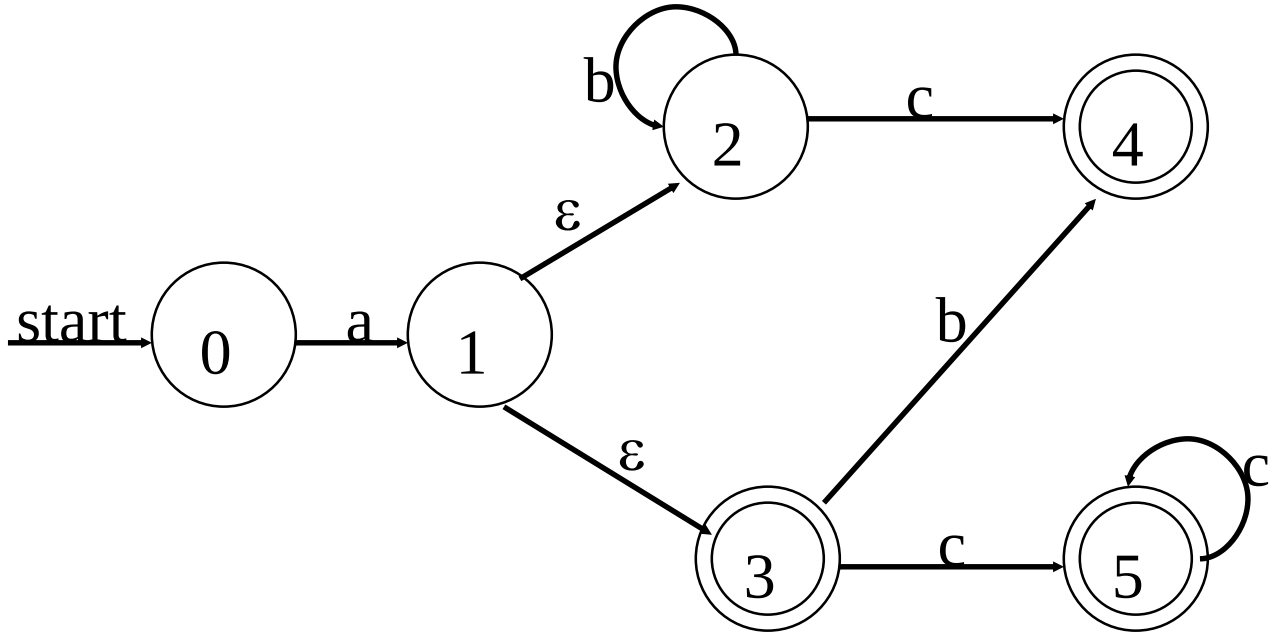
单词←→ 模式←→ 正则表达式←→ NFA ←?→ 词法分析程序

单词是词法分析的基本构件

词法分析器可用一组NFA来描述，每个NFA表示一个单词

NFA，例

(a (b*c)) | (a (b | c^+)?)

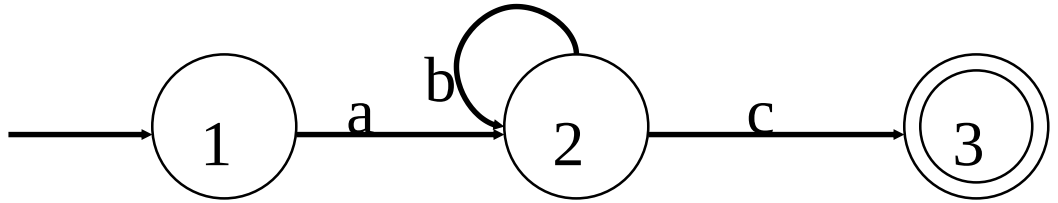


- 可接受abbc

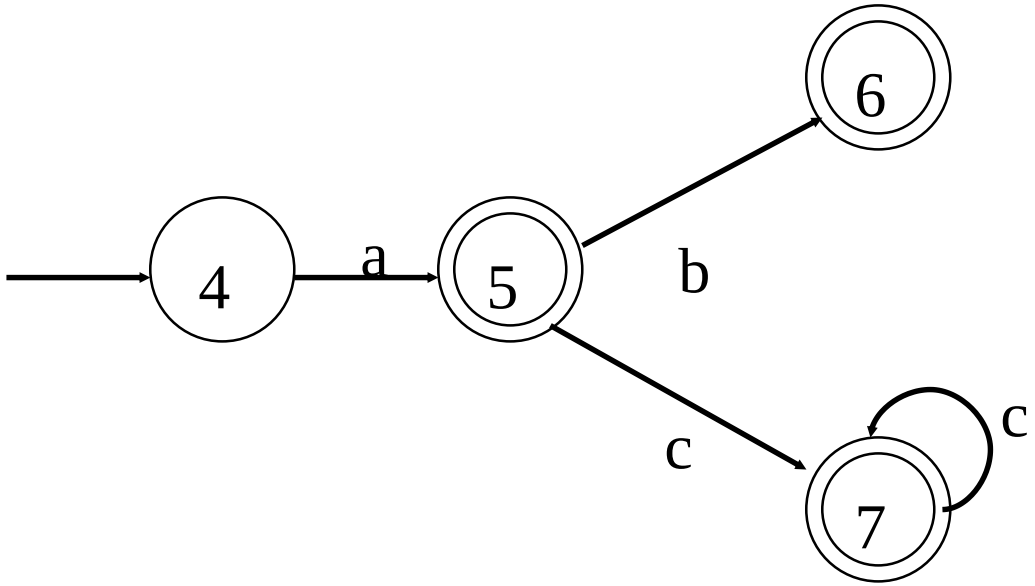
另一解法

(a (b*c)) | (a (b | c^{+})?)

a (b*c)

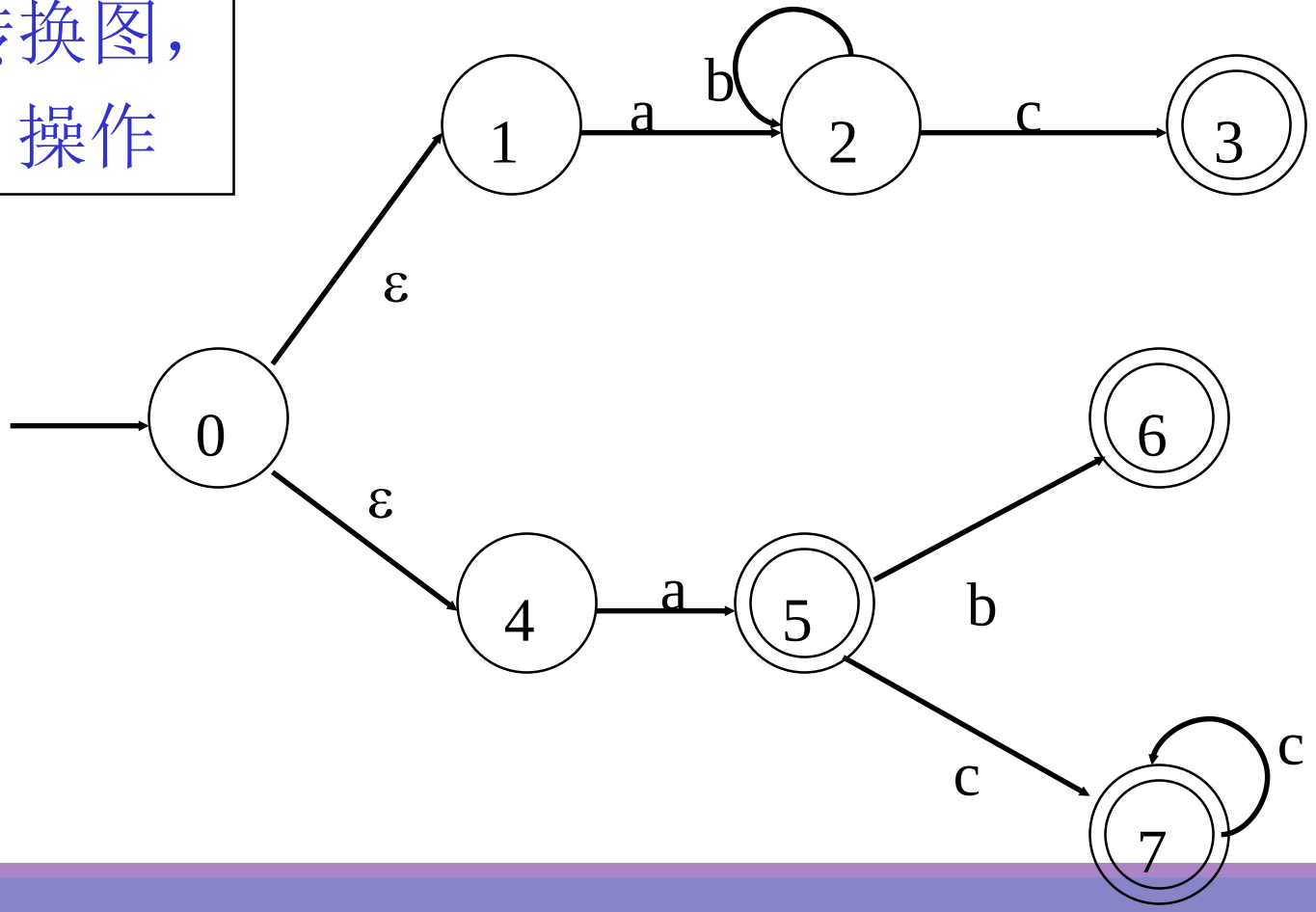


a (b | c+)?



另一解法

两个状态转换图，
进行“并”操作



确定有限自动机 DFA

五元组定义

$M = \{ S, \Sigma, d, s_0, F \}$，其中

1. S：有限状态集
2. Σ：有穷字母表
3. δ：S×Σ到S的单值映射，即

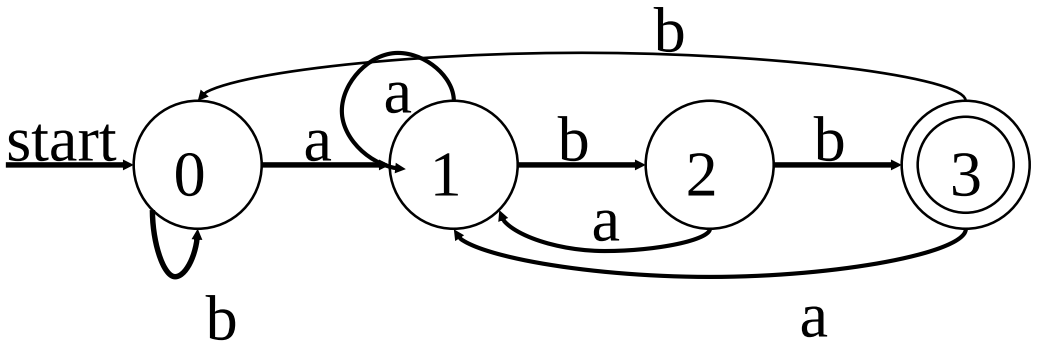
   $\delta: S \times \Sigma \to S$
4. $s_0 \in S$是唯一的初态
5. $F \subseteq S$ 是一个终态集

没有ε

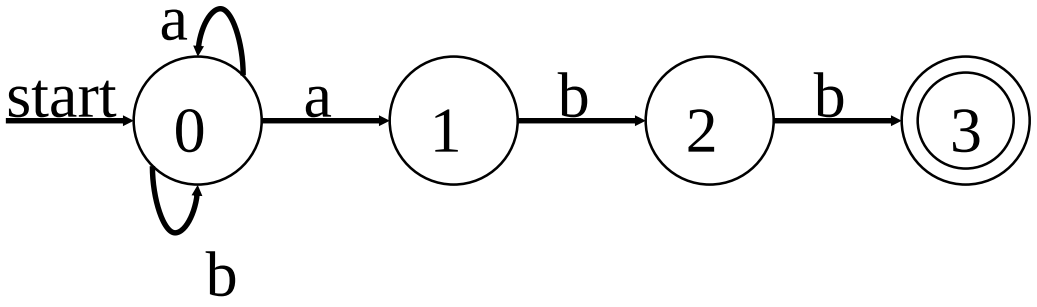
值域为S，不是S的子集的集合

例：(a|b)*abb

- DFA



对比NFA



算法： 模拟DFA

输入：以eof结束的输入串x，初态s_0，终态集F的DFA D

输出：若D接受x，返回“yes”，否则返回“no”

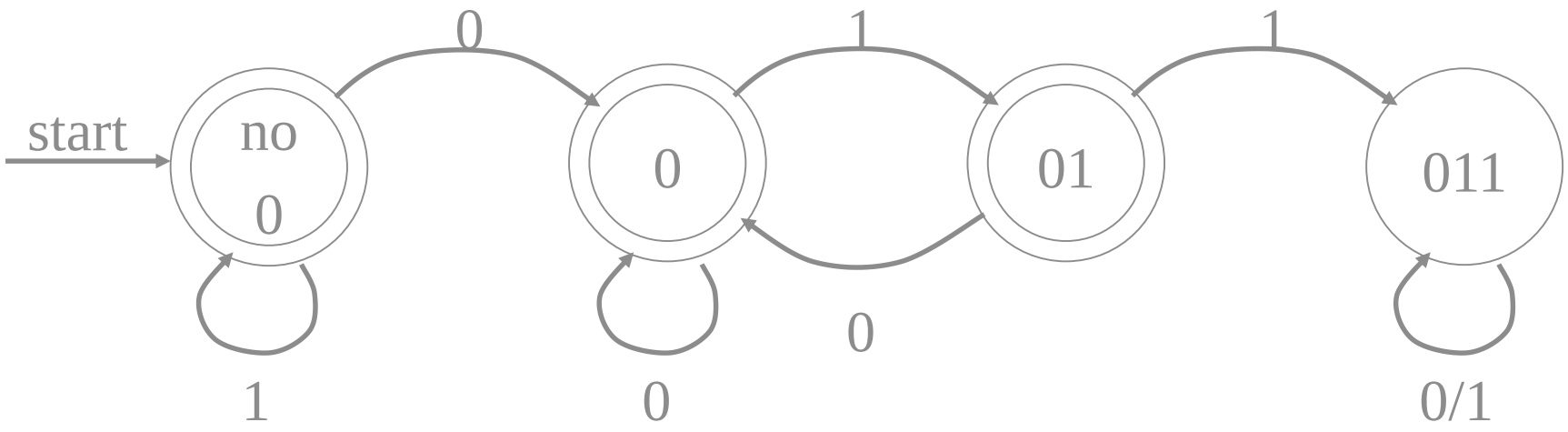
方法：利用nextchar读取符号，利用转换函数d进行状态转换

```
s ← s0
c ← nextchar;
while c ← eof do
  s ← d(s,c);
  c ← nextchar;
end;
if s is in F then return “yes”
            else return “no”
```

设计自动机

一种思路：动态观点——考虑自动机运转方式，状态的变化

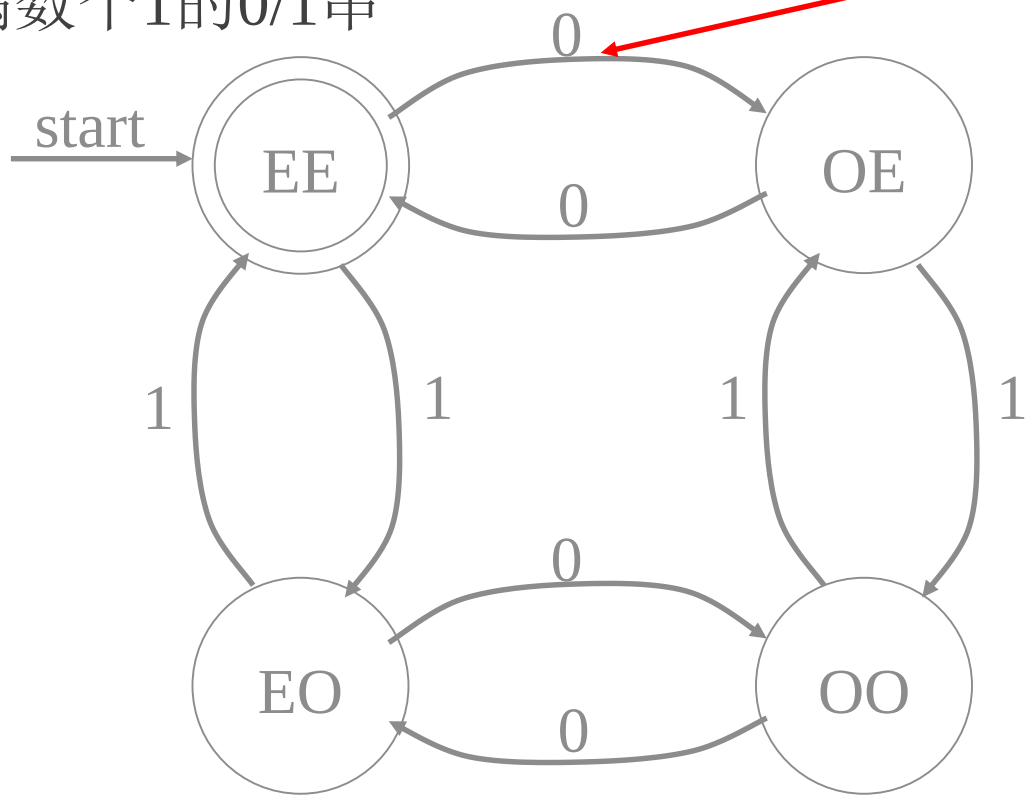
不包含子串011的0、1串



设计自动机

另一种思路：状态——符号串集合

偶数个0，偶数个1的0/1串



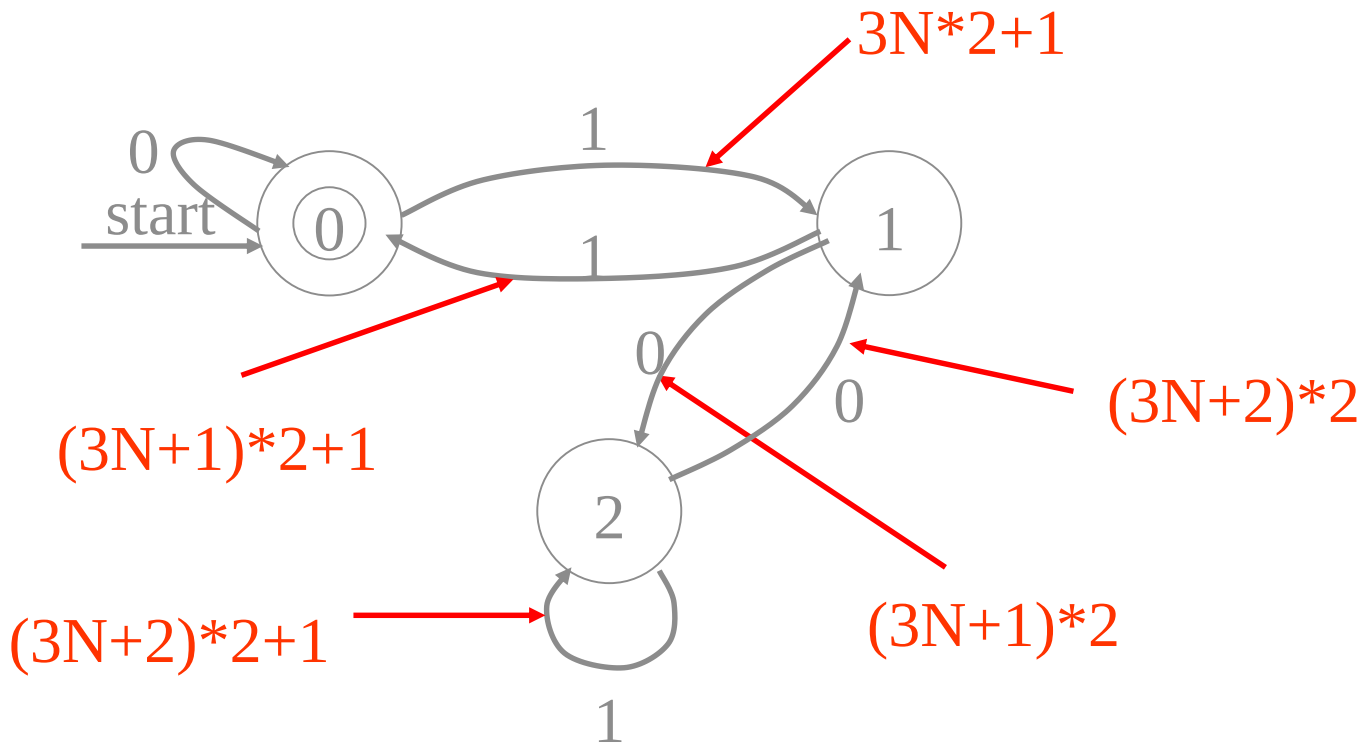
偶数个0偶数个1

拼接1个0 →

奇数个0偶数个1

设计自动机

能被3整除的二进制串



0	0
1	01
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

学习内容

- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

编写词法分析器

词法分析器的构造

- 正则表达式→构造NFA
- NFA →转换DFA
- 模拟DFA → 词法分析器

由正则表达式构造NFA

目的

- 正则表达式（描述单词）

  →NFA（定义语言）

  → DFA（适于计算机实现）

算法描述

根据正则表达式的结构转换为NFA

- 基本符号、 ε → 简单NFA
- 子正则表达式通过各种操作（定义规则）：|、连接、闭包构造复杂正则表达式
  →相应的子NFA组合为复杂NFA

从正则表达式到自动机

算法：Thompson构造法

输入：字母表Σ上的一个正则表达式r

输出：一个NFA N，L(N)=L(r)

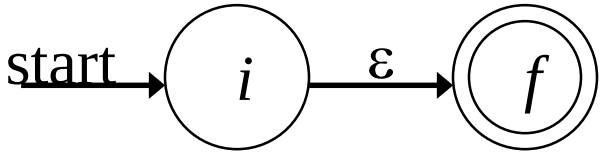
Thompson构造法

1. 将r分解为子正则表达式
2. 对其中每个基本符号（字母表中符号和ε），按下面给出的规则（1）、（2）构造NFA。注意：同一符号在不同位置需构造不同NFA
3. 按照r的语法结构，对每个正则表达式操作，按下面规则（3）的方法，将操作对象——子正则表达式对应的子NFA组合成更大的NFA，直至形成完整正则表达式r对应的最终的NFA

算法：正则表达式→ NFA

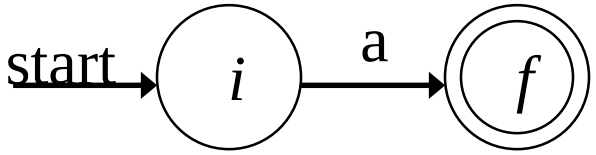
构造规则

1. 对于ε，构造NFA



L(ε)

2. 对于a ∈Σ，构造NFA

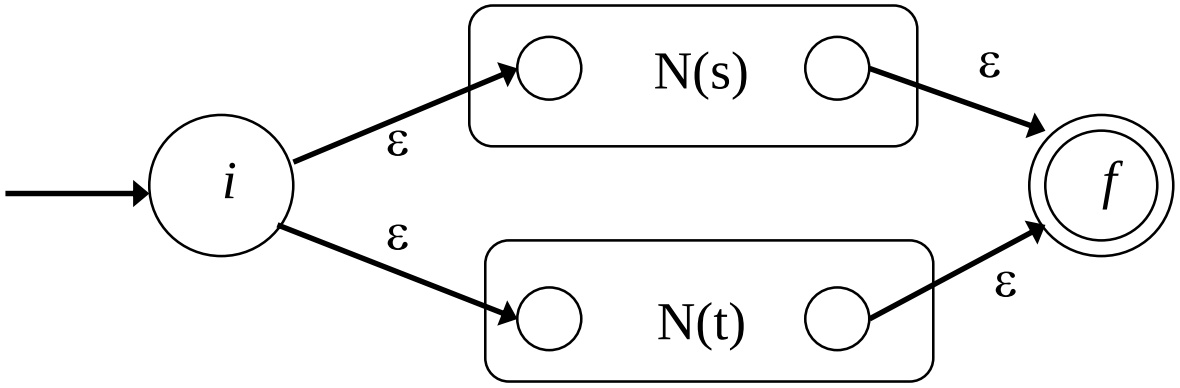


L(a)

算法：正则表达式→ NFA

3. 假定N(s), N(t)是正则表达式s, t对应的NFA

a) 对正则表达式s|t，构造如下组合NFA N(s|t)



$L(s) \cup L(t)$

两个新状态：*i*——新初态　　*f*——新终态

原初态和终态不再是组合NFA的初态和终态

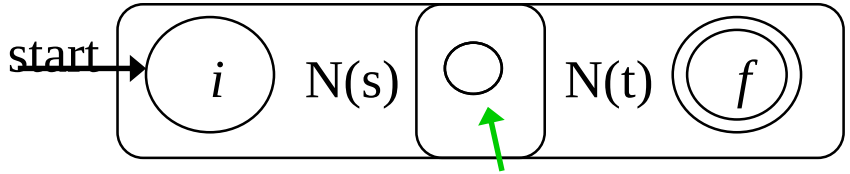
四条新e边：

i → N(s)初态　*i* → N(t)初态

N(s)终态→ *f*　N(s)终态→ *f*

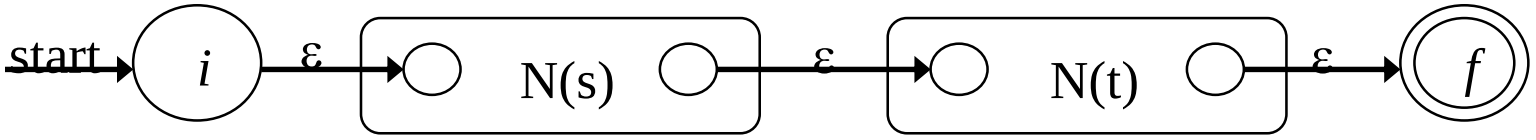
算法：正则表达式→ NFA

b) 对正则表达式st，构造如下组合NFA N(st)



L(s) L(t)

方法二:

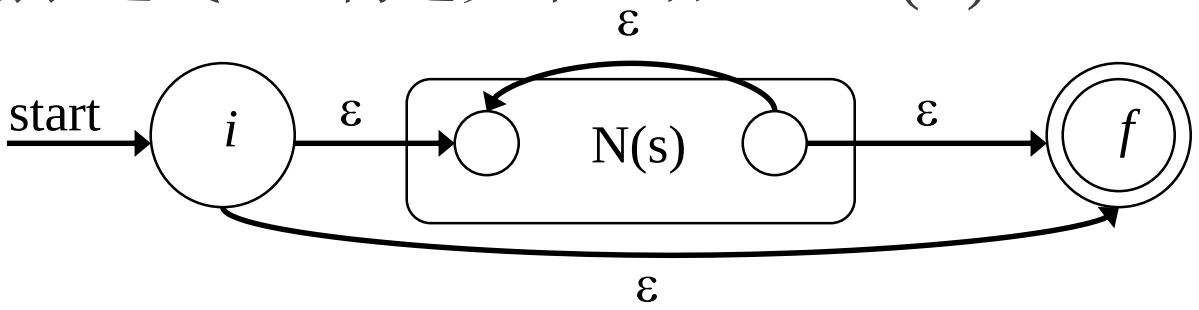


方法一：新初态——N(s)初态，
新终态——N(t)终态
N(s)终态与N(t)初态合并

方法二：
新初态、终态——*i*、*f*
三条新ε边：*i*→N(s)初态　　N(t)终态→ *f*
N(s)终态→ N(t)初态

算法：正则表达式→ NFA

c) 对正则表达式s^*，构造如下组合NFA $N(s^*)$



新初态、终态——*i*、*f*

四条新ε边： *i* → *f*

i → N(s)初态　　N(s)终态→ *f*

N(s)终态→ N(s)初态

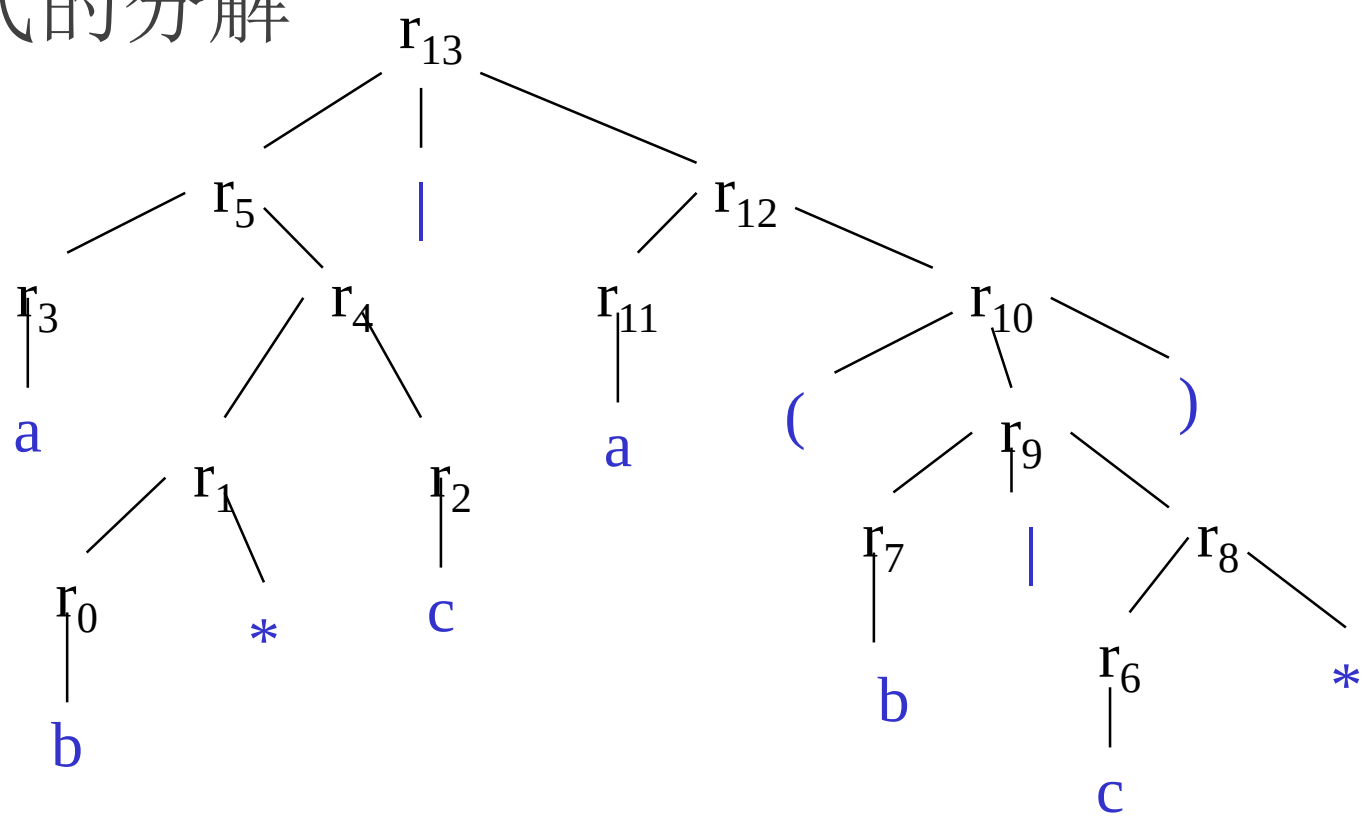
d) 对正则表达式(s)，N((s)) = N(s)

算法特性

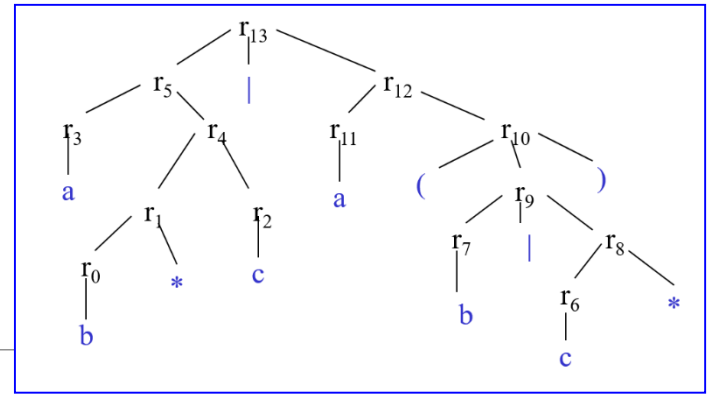
1. 每个步骤最多增加两个新状态➔
   N(r)状态数 <= 2 × (r的符号数+操作符数)
2. N(r)有且只有一个初态和一个终态
3. N(r)的每个状态，或者有一条标记为某个$a \in \Sigma$的输出边，或者至多有两条ε输出边
4. 为状态取名要小心

例：(ab*c) | (a(b|c*))

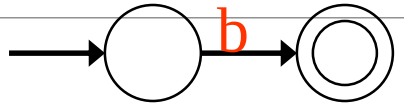
正则表达式的分解



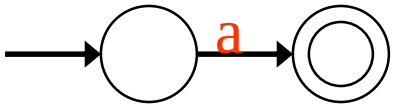
例：(ab*c) | (a(b|c*))



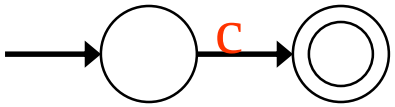
r_0:



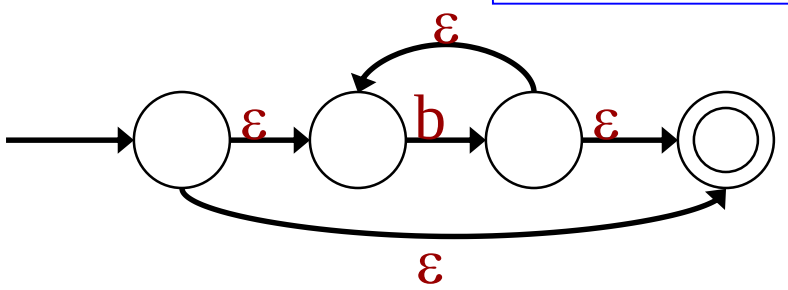
r_3:



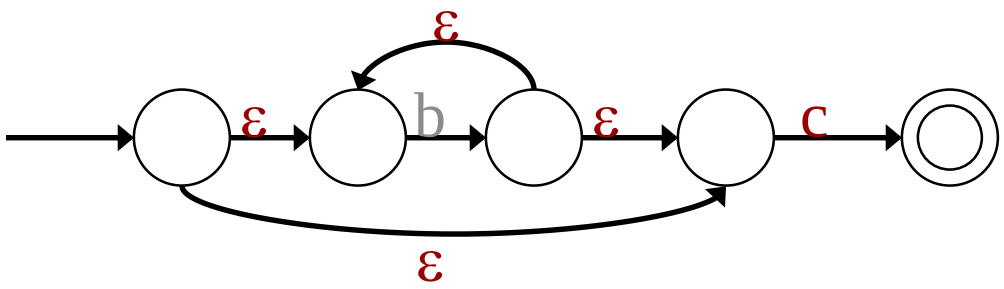
r_2:



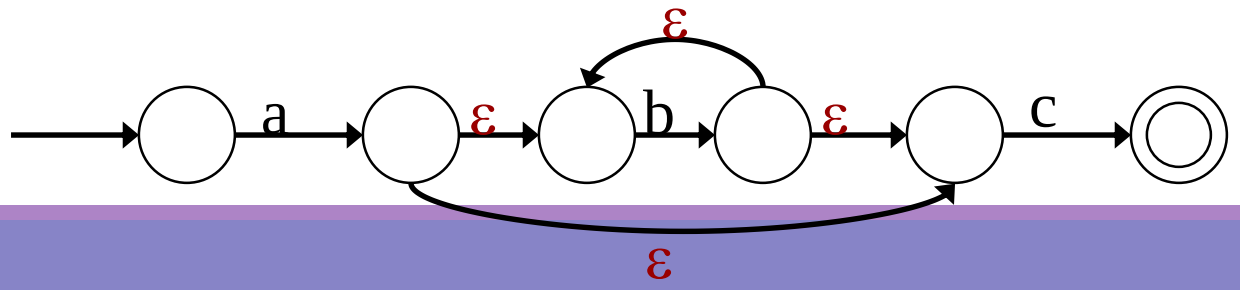
$r_1 : r_0^*$

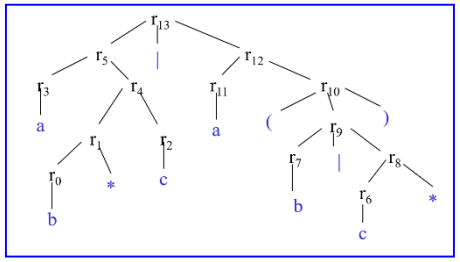


$r_4 : r_1\ r_2$

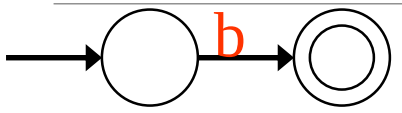


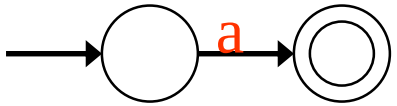
$r_5 : r_3\ r_4$

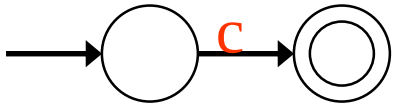


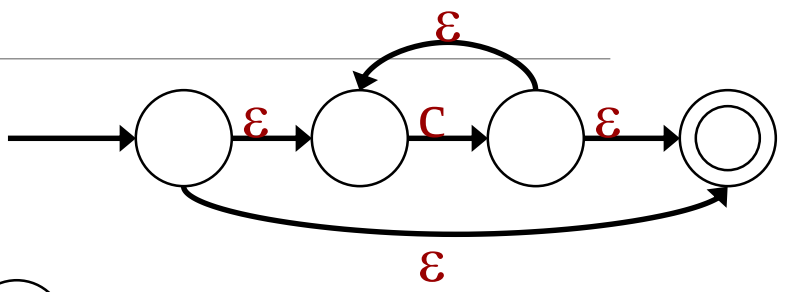


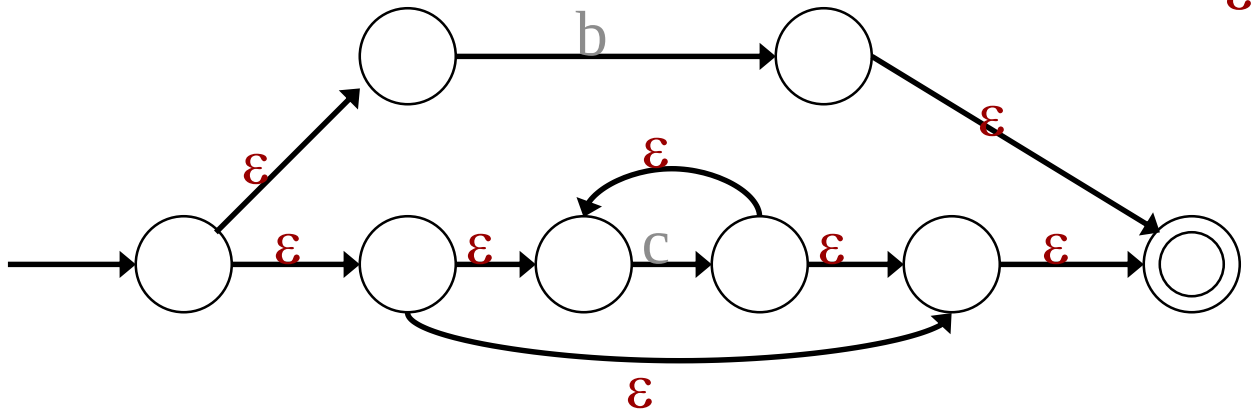
例：(ab*c) | (a(b|c*))

r_7: 

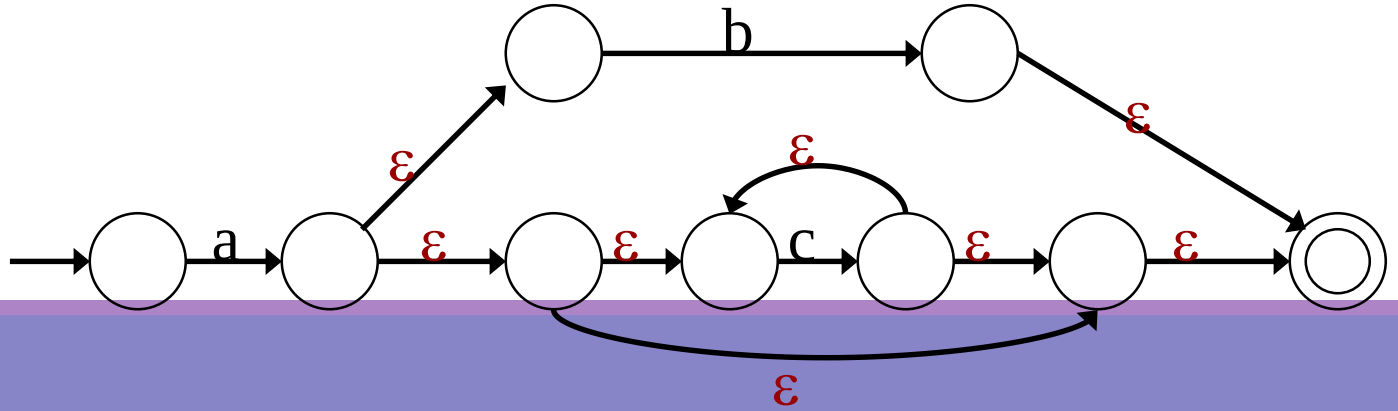
r_{11}: 

r_6: 

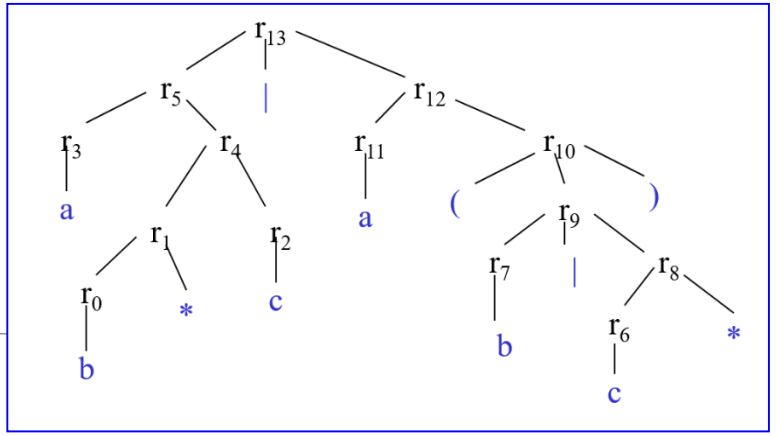
r_8:r_6* 

$r_9 : r_7 \mid r_8$ 

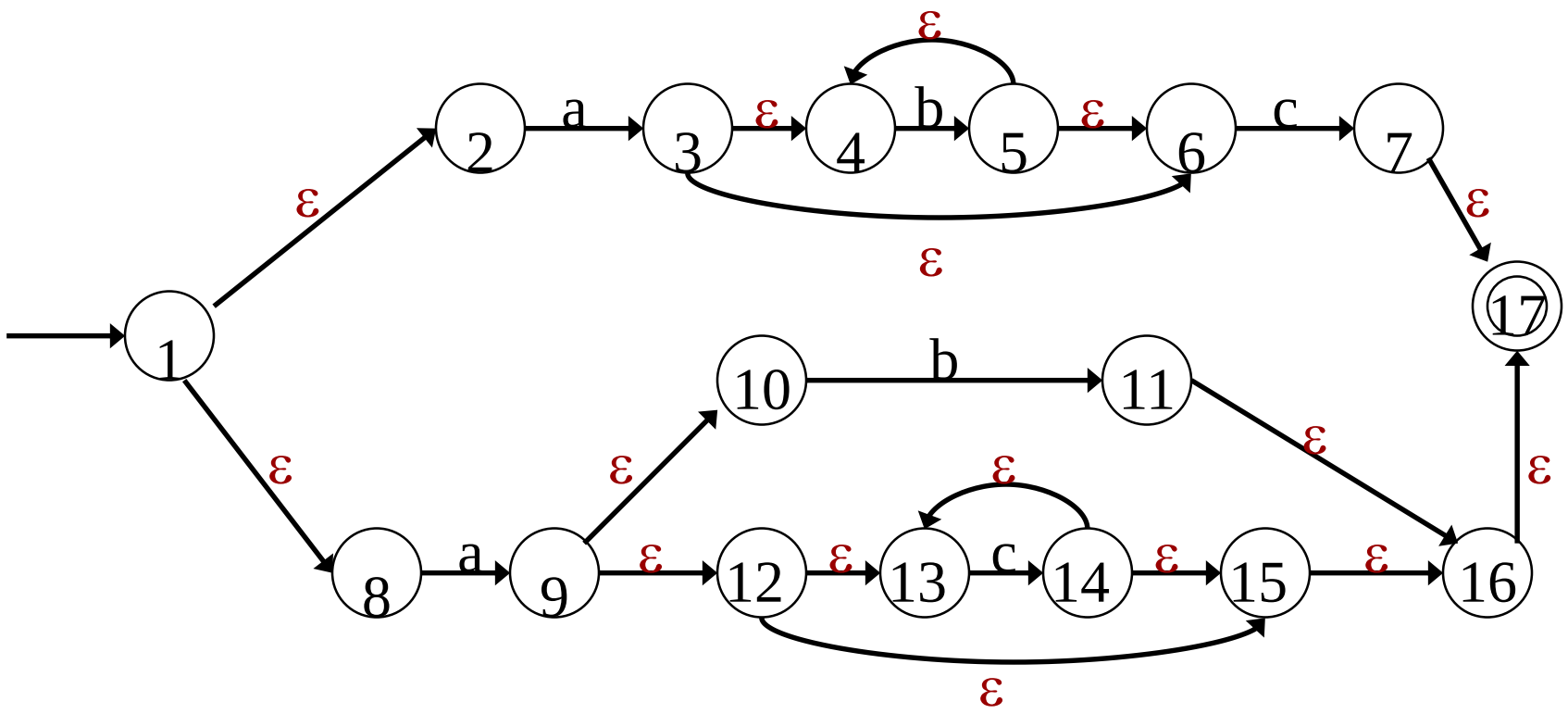
$r_{10} : (r_9)$

$r_{12} : r_{11}\ \ r_{10}$ 

例：(ab*c) | (a(b|c*))

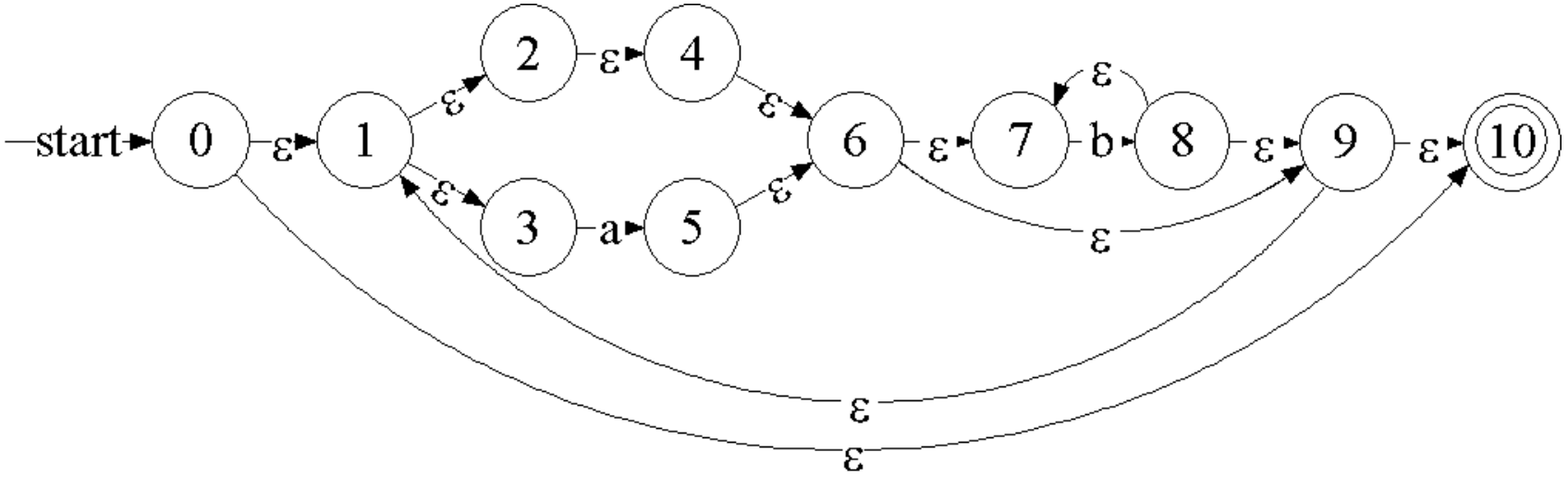


$r_{13} : r_5 \mid r_{12}$



正则表达式→ NFA练习

((ε | a)b*)*



NFA到DFA的转换

原因

- 正则表达式 $\xrightarrow{\text{简单}}$ NFA
- 但NFA存在缺点

  同一符号/ ε和其它符号→多义性

  难以实现

算法：NFA转换为DFA

输入：一个NFA N

输出：一个DFA D，L(D)=L(M)

算法中用到下列操作，其中s表示NFA的一个状态，T表示NFA的一个状态集

操作	描述
ε-closure(s) （ε闭包）	s以及从s出发仅通过ε边可到达的所有状态的集合
ε-closure(T)	∪ε-closure(s), s∈T
d(T, a)	∪d(s, a), s∈T

子集构造法

subset construction

- 什么是NFA？输入字符串，输出状态集
  什么是DFA？输入字符串，输出状态
- NFA、DFA等价是什么？输入任意符号串，输出相同结果
- DFA状态←→ NFA状态集！
- DFA状态s与NFA状态集T有对应关系，那么
  输入$a_1a_2…a_n$ →到达DFA状态s，一定有
  输入$a_1a_2…a_n$ → NFA所有可到达的状态集合T

算法基本思想

平凡算法：穷举所有可能的符号串

- 每个符号串输入NFA，状态集合→ DFA状态
- 不可行！

递推方法

- 最简单的符号串ε：NFA状态集合←→ DFA状态
- 长度为1的串a=εa，在自动机中可达的状态为：从ε对应的状态经过标记为a的边可达的状态
- 长度为2的串…

算法基本思想

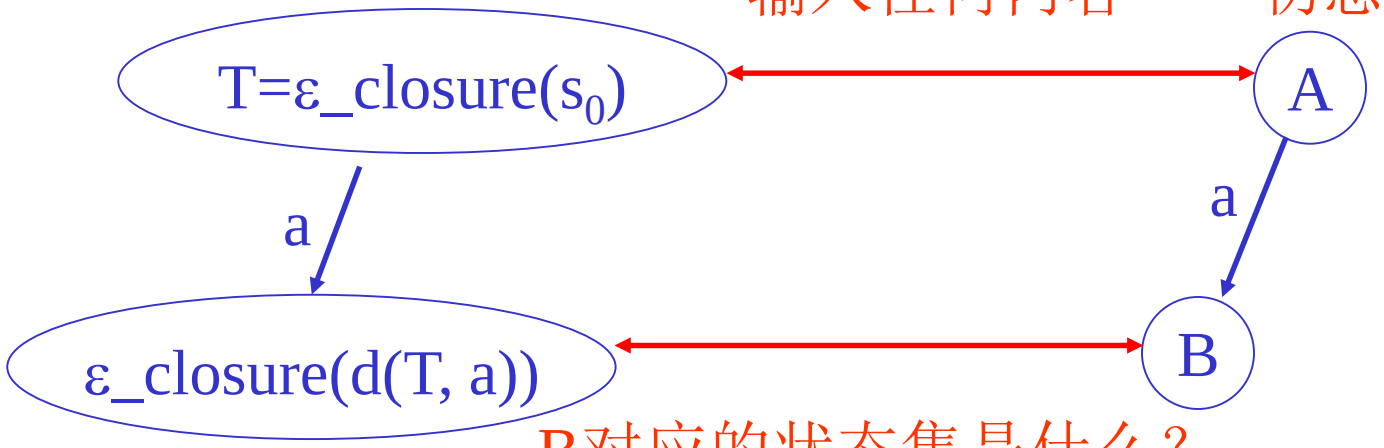
最简单的符号串ε输入NFA和DFA应得到相同结果——未输入任何内容——初态

A ←→ T

B：A读入a后转到的状态

与B对应的应该是：T中所有状态读入a后转到的状态（d(T,a)），

注意：还需一次闭包计算



B对应的状态集是什么？

算法正确性

如何保证“对任何符号串，输入NFA和DFA得到相同的结果”？

对ε，显然正确：T ⟷ A

对a、b、c…——ε连接一个符号——T、A通过一条边到达的状态，显然也正确

数学归纳法

算法：NFA转换为DFA

计算ε -closure(T)的算法

```
T中所有状态压栈;
ε -closure(T) = T;
while (栈不空) {
 pop t;
 for (每个状态u，t到u有一条ε边) {
        if (u不在ε -closure(T)中) {
                将u加入ε -closure(T);
                u压栈;
        }
 }
}
```

算法：NFA转换为DFA

NFA → DFA的算法

初始， ε -closure(s_0)是Dstates（DFA状态集）中唯一状态，且未标记;

while (Dstates中存在未标记状态T) {

 标记T;

 for (每个输入符号a) {

 U = ε -closure(δ(T, a));

 if (U不在Dstates中)

 将U加入Dstates，且设为未标记;

 Dtran[T, a] = U; //DFA状态转换矩阵

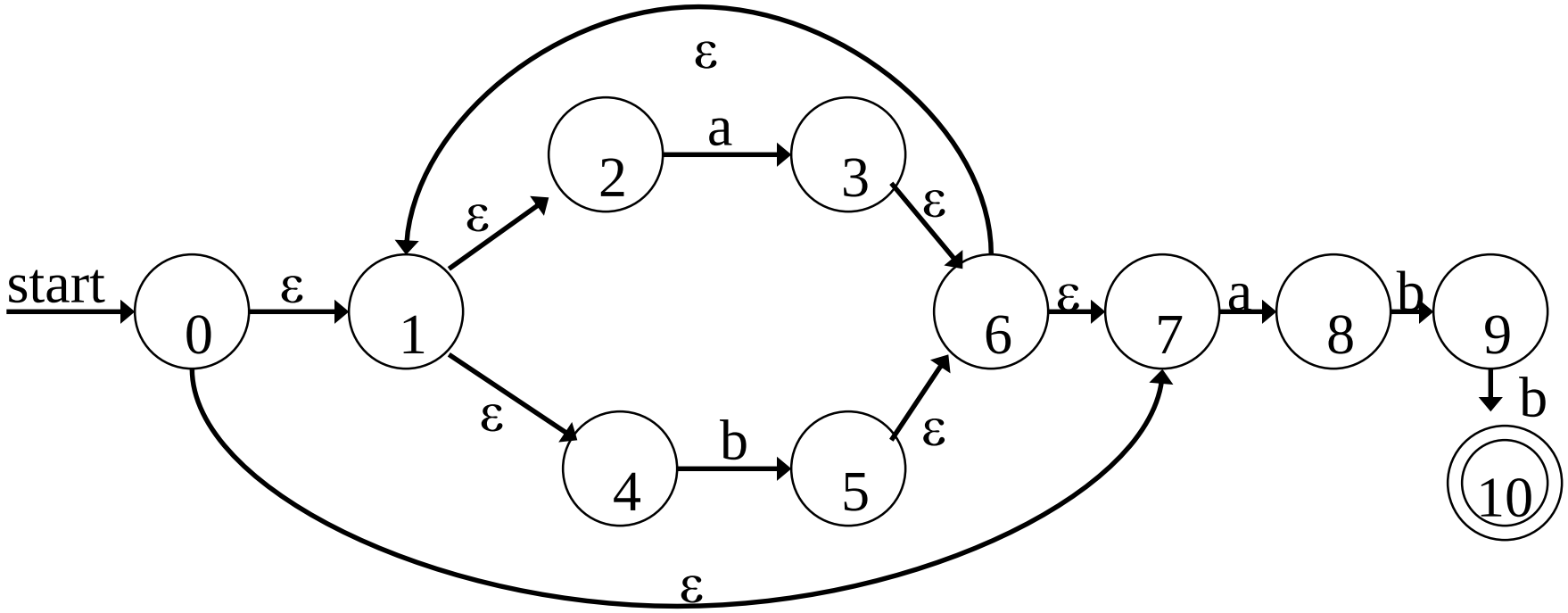
 }

}

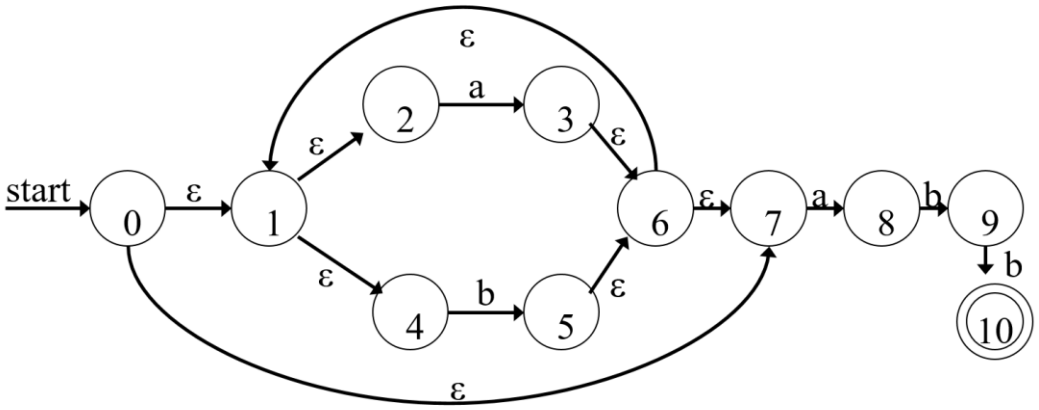
ε -closure(s_0)为初态，包含NFA终态的DFA状态为DFA终态

例：(a | b)*abb

NFA → DFA



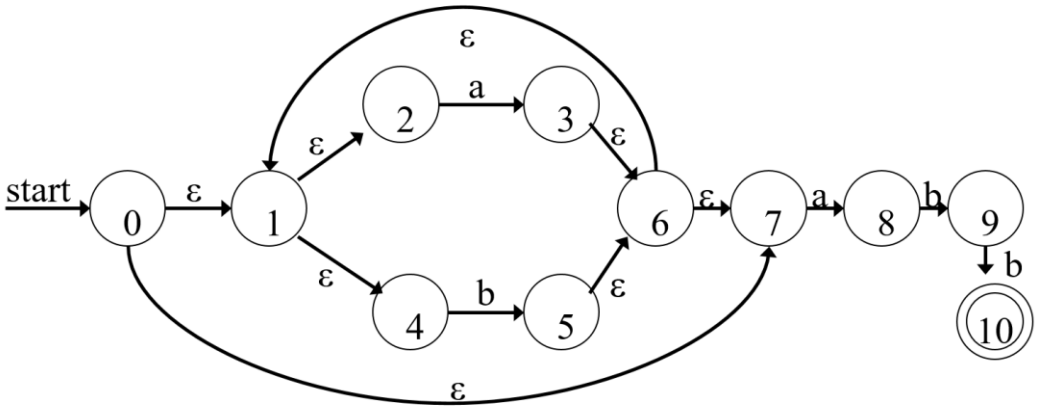
例：(a | b)*abb



第一步

ε-closure(0) = {0,1,2,4,7} =A

例：(a | b)*abb



第一步

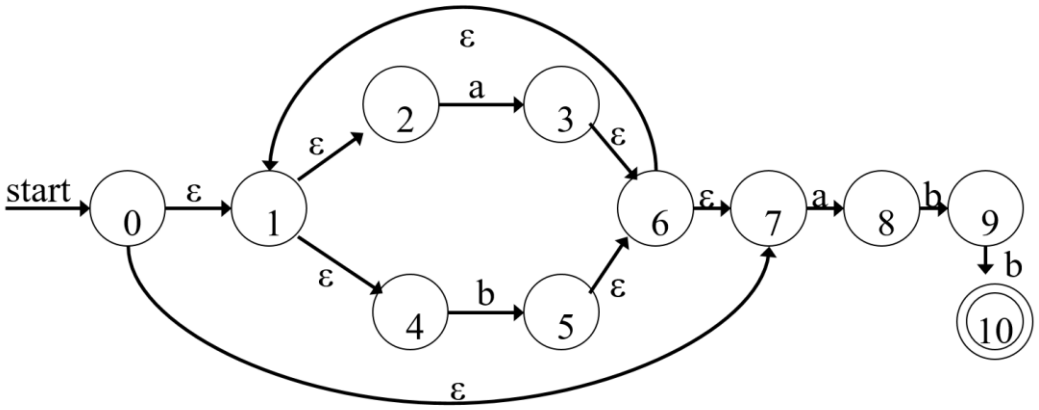
ε-closure(0) = {0,1,2,4,7} =A

第二步

a: ε -closure(δ(A,a)) = ε -closure(δ({0,1,2,4,7},a)) = ε -closure({3,8}) = {1,2,3,4,6,7,8} = B

Dtran[A,a] = B

例：(a | b)*abb



第一步

ε-closure(0)={0,1,2,4,7} =A

第二步

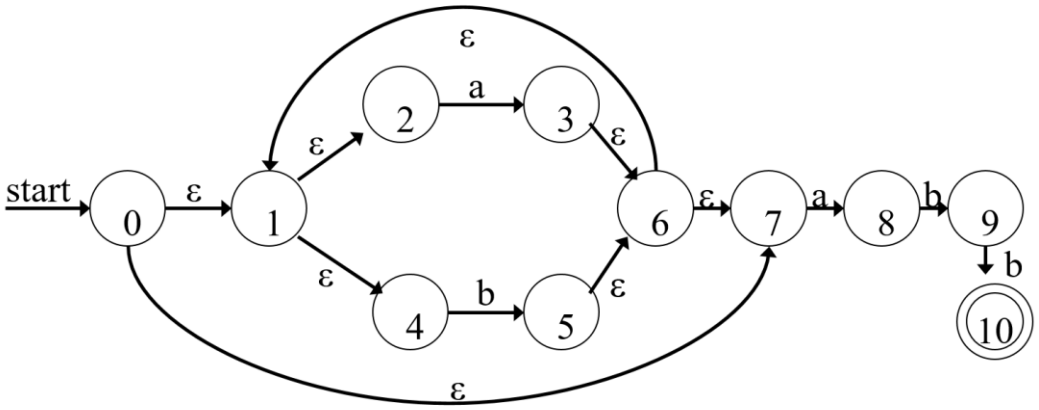
a: ε-closure(δ(A,a)) = ε-closure(δ({0,1,2,4,7},a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B

Dtran[A,a] = B

b : ε-closure(δ(A,b)) = ε-closure(δ({0,1,2,4,7},b)) = ε-closure({5}) = {1,2,4,5,6,7} = C

Dtran[A,b] = C

例：(a | b)*abb



第三步

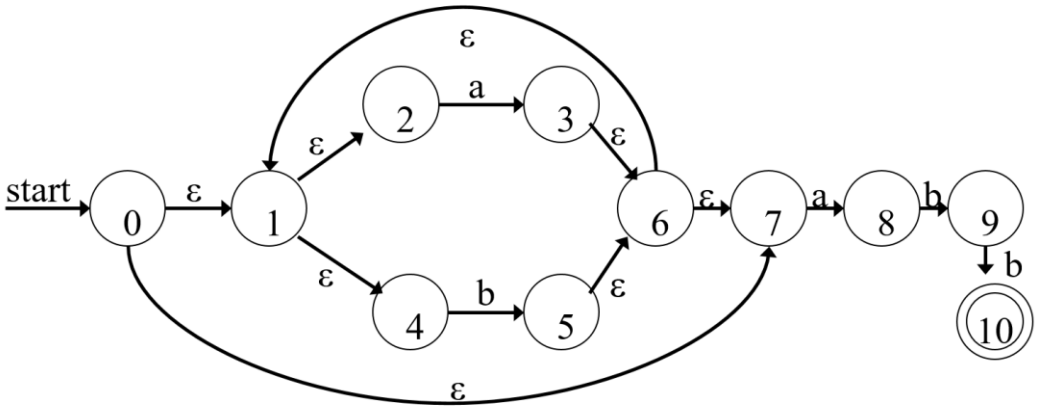
a : ε -closure(δ(B,a)) = ε -closure(δ({1,2,3,4,6,7,8},a))} = {1,2,3,4,6,7,8} = B

Dtran[B,a] = B

b : ε -closure(δ(B,b)) = ε -closure(δ({1,2,3,4,6,7,8},b))} = {1,2,4,5,6,7,9} = D

Dtran[B,b] = D

例：(a | b)*abb



第四步

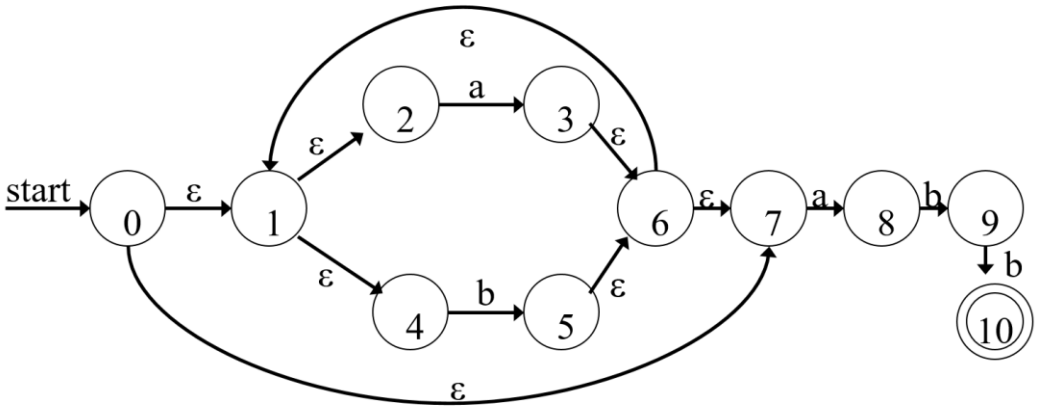
a : ε -closure(δ(C,a)) = ε -closure(δ({1,2,4,5,6,7},a))}= {1,2,3,4,6,7,8} = B

Dtran[C,a] = B

b : ε -closure(δ(C,b)) = ε -closure(δ({1,2,4,5,6,7},b))}= {1,2,4,5,6,7} = C

Dtran[C,b] = C

例：(a | b)*abb



第五步

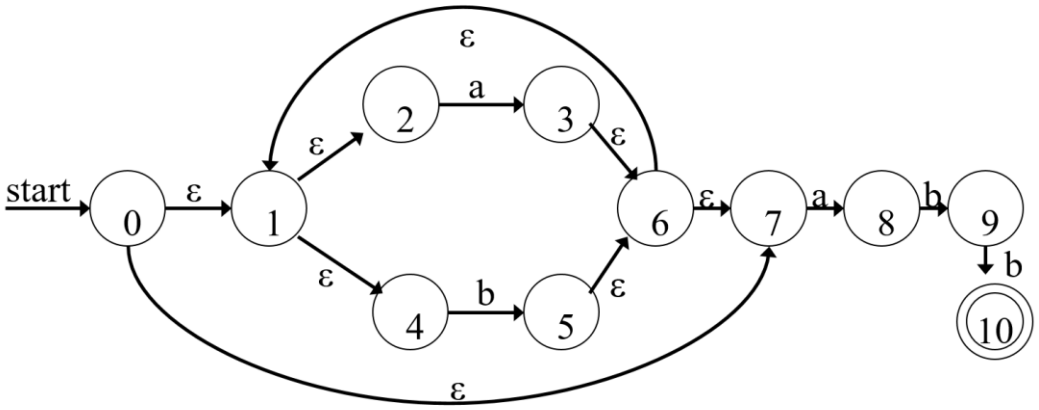
a : ε -closure(δ(D,a)) = ε -closure(δ({1,2,4,5,6,7,9},a))}= {1,2,3,4,6,7,8} = B

Dtran[D,a] = B

b : ε -closure(δ(D,b)) = ε -closure(δ({1,2,4,5,6,7,9},b))}= ε -closure({5,10}) = {1,2,4,5,6,7,10} = E

Dtran[D,b] = E

例：(a | b)*abb



第六步

a : ε -closure(δ(E,a)) = ε -closure(δ({1,2,4,5,6,7,10},a))}= {1,2,3,4,6,7,8} = B

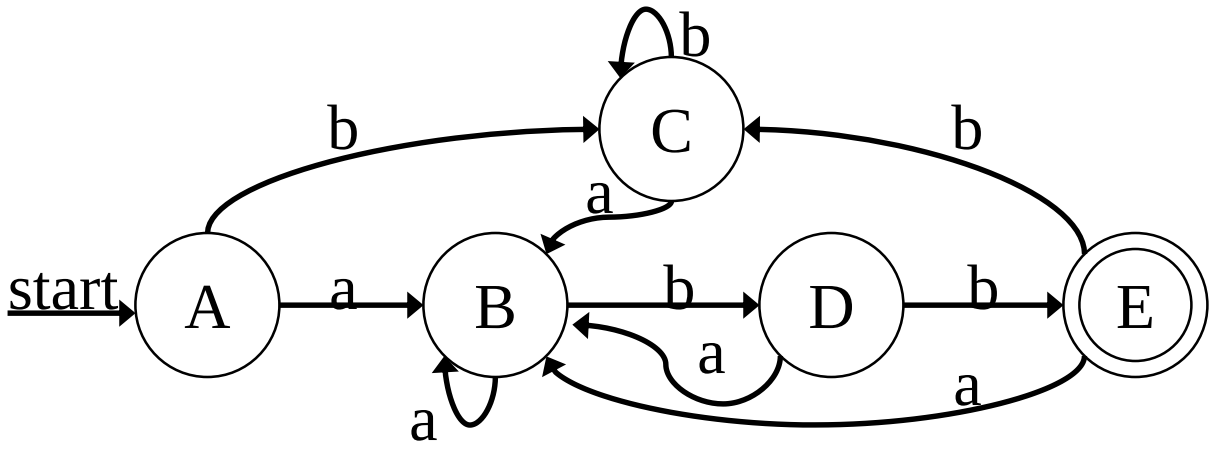
Dtran[E,a] = B

b : ε -closure(δ(E,b)) = ε -closure(δ({1,2,4,5,6,7,10},b))}= {1,2,4,5,6,7} = C

Dtran[E,b] = C

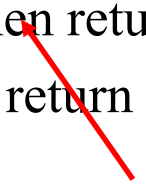
例：(a | b)*abb

State	Input Symbol a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



模拟DFA

```
s ← s0
c ← nextchar;
while c ≠ eof do
  s ← d(s,c);
  c ← nextchar;
end;
if s is in F then return “yes”
             else return “no”
```



对照DFA模拟算法

模拟NFA

S ← ε -closure($\{s_0\}$)
c ← nextchar;
while c ≠ eof do
  S ← ε -closure(d(s,c));
  c ← nextchar;
end;
if S ∩ F ≠∅ then return “yes”
        else return “no”

s ← s_0
c ← nextchar;
while c ≠ eof do
  s ← d(s,c);
  c ← nextchar;
end;
if s is in F then return “yes”
        else return “no”

对照DFA模拟算法

NFA与DFA的比较

NFA模拟算法

- 两个栈：当前状态集/下一状态集
- 时间与|N|*|x|成比例，|N|——状态数，
  |x|——输入串长度

NFA实现正则表达式/DFA实现正则表达式

—— 时—空的折衷

一种解决方法

“lazy transition evaluation”

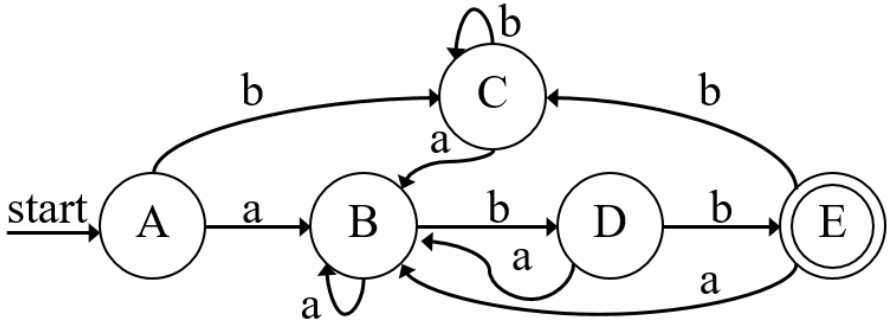
- 使用DFA进行词法分析
- 但不预先构造完整的DFA，而是翻译（词法分析）过程中用到哪些状态和转换关系才即时计算，并利用缓存机制

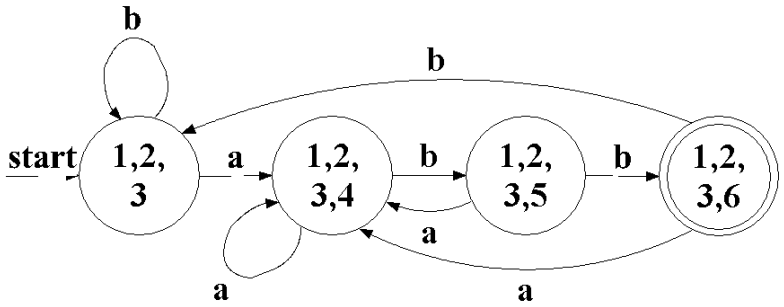
优化词法分析器

DFA状态最小

最小化DFA的状态数

$(a|b)^*abb$





对于一个正则表达式，识别它的DFA中，存在唯一一个状态数最少的DFA

最小化DFA的状态数

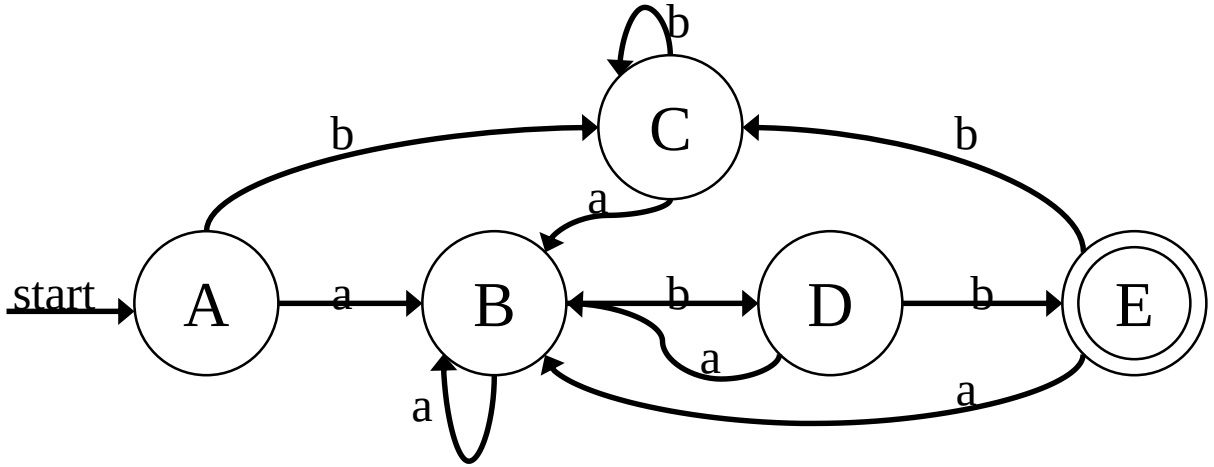
问题：DFA M，状态集S，字母表Σ，化简之，使状态数最少

分组合并等价状态

符号串w区分(distinguish)状态s、t

- 分别从s、t开始，读入w、转换状态，读取完毕后，一个到达终态，另一个到达非终态

“区分”



bb即可区分A、B

任何符号串均无法区分A、C

化简方法

寻找所有可被区分的状态组

- 不可区分→合并

实际算法是不断划分而不是合并，划分状态组过程中

- 同组：尚未区分状态
- 不同组：已区分状态

化简方法（续）

初始，两个组：终态与非终态

划分方法，对于状态组$A=\{s_1,s_2,\ldots,s_k\}$

- 对符号a，得到其转换状态$t_1,t_2,\ldots,t_k$
- 若$t_1,t_2,\ldots,t_k$属于不同状态组，则需将A对应划分为若干组

算法：最小化DFA

划分算法

for (Π中每个状态组G) {

将G划分为若干分组，使得

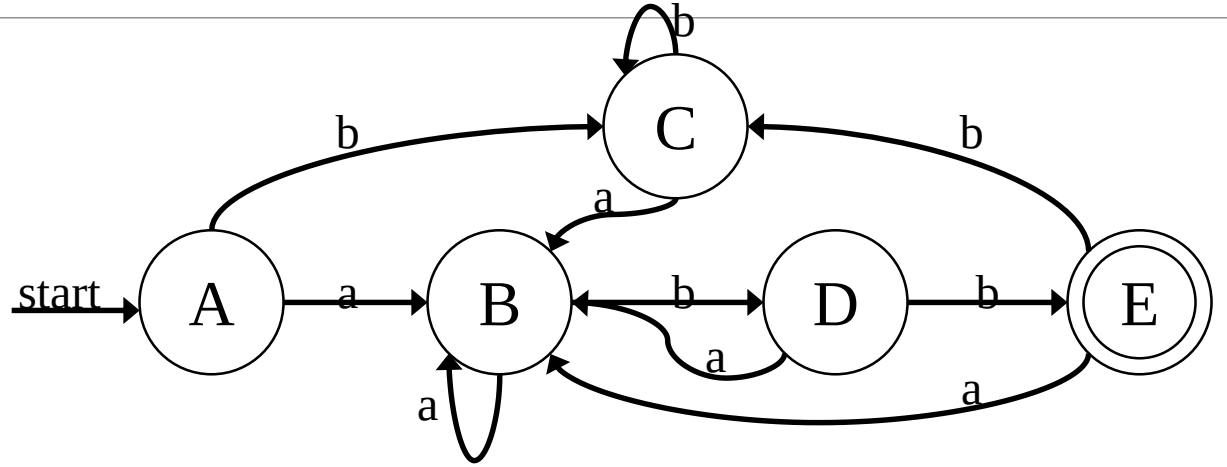
G中两个状态s、t在同一分组中的充要条件为

对所有符号a，s、t转换后的状态处于Π中的同一个状态组中;

在Π_{new}中，用所有分组替换G

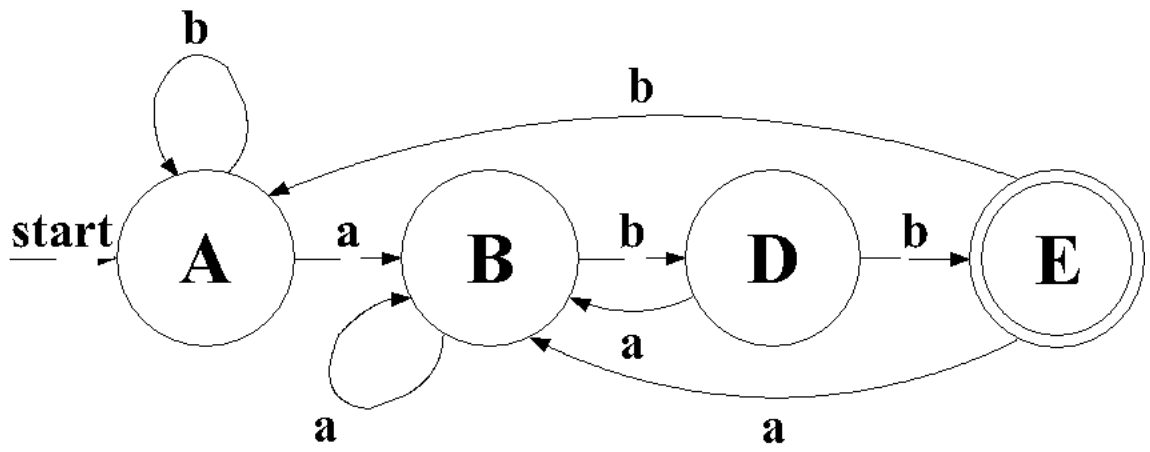
}

例：



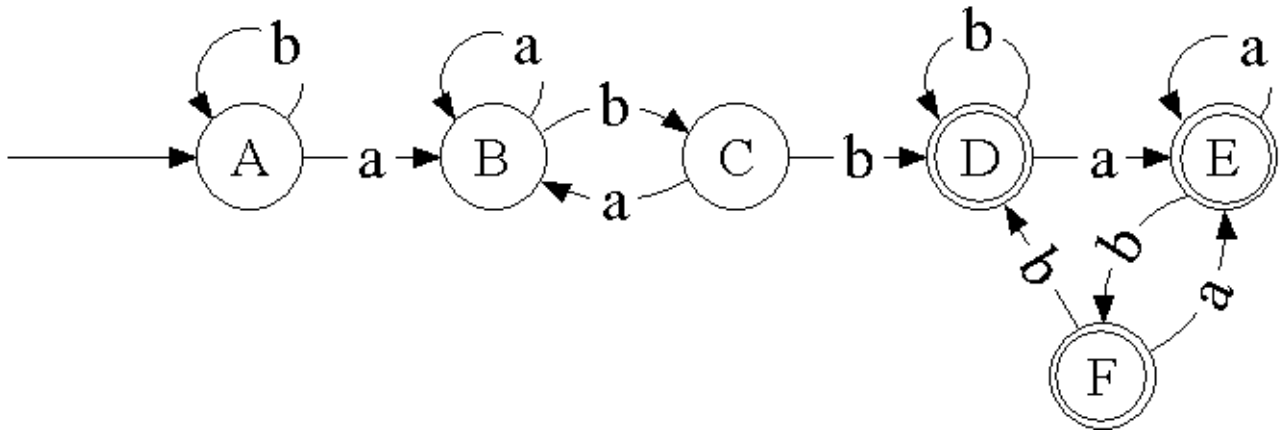
1. {E}, {A, B, C, D}
2. {A, B, C, D} $\xrightarrow{b}$ {C, D, C, E} → {E}, {D}, {A, B, C}
3. {A, B, C} $\xrightarrow{b}$ {C, D, C} → {E}, {D}, {B}, {A, C}
4. Π_{final}: {A, C}, {B}, {D}, {E}

例：（续）



状态	符号	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

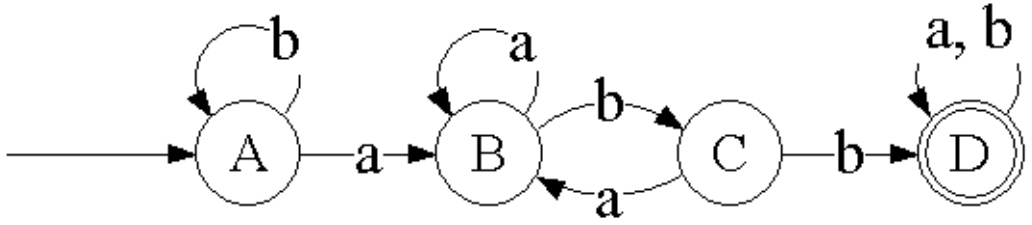
最小化DFA练习



初始划分：{A, B, C}, {D, E, F}

{A, B, C} —b→ {A, B}, {C}

{A, B} —b→ {A}, {B}



词法分析器

- 正则表达式 + 状态转换图
  - 正则表达式→状态转换图
  - 状态转换图的实现→词法分析器
- 正则表达式 + 自动机
  - 正则表达式→构造NFA
  - NFA → DFA
  - 最小化DFA
  - 模拟DFA →词法分析器

练习题

(a|b)*aab(a|b)*

设计正规式，接受除以4余3的八进制数