

第三章 词法分析

学习内容

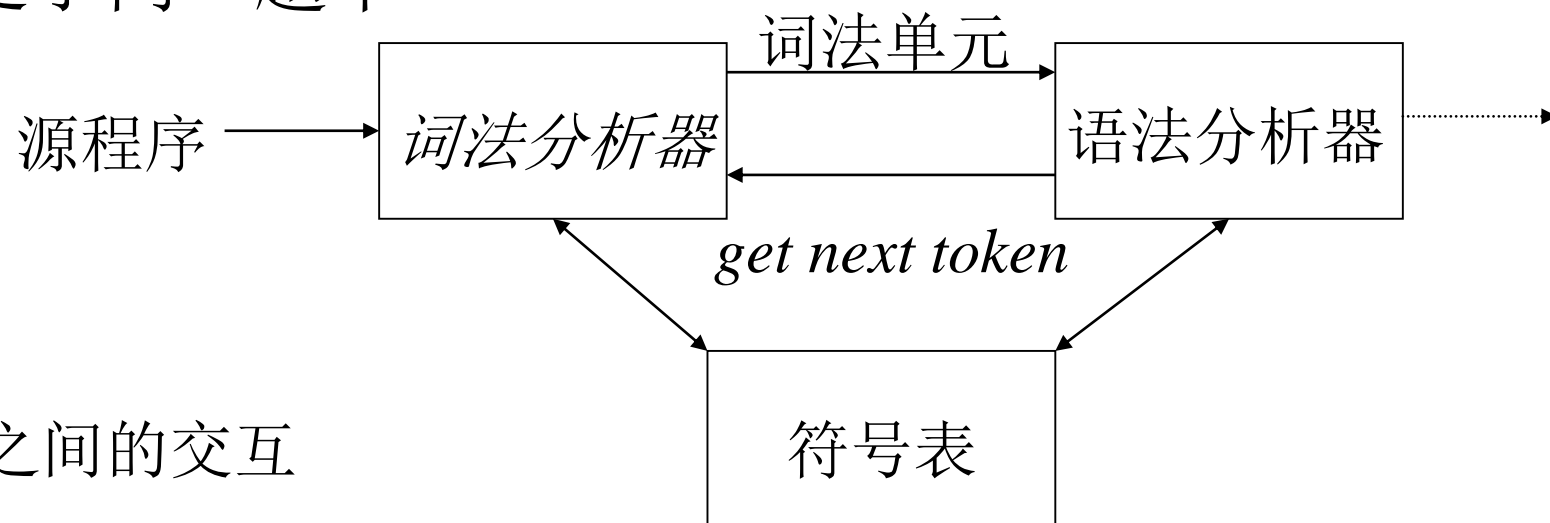
- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

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。
- 通常和语法分析器处于同一趟中



词法分析器与语法分析器之间的交互

3.1 词法分析器的角色

编译过程的分析部分：词法分析 + 语法分析

简化编译器的设计

词法分析器可以首先完成一些简单的处理工作

提高编译器的效率

相对于语法分析，词法分析过程简单，可高效实现增强编译器的可移植性（输入设备无关）

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

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

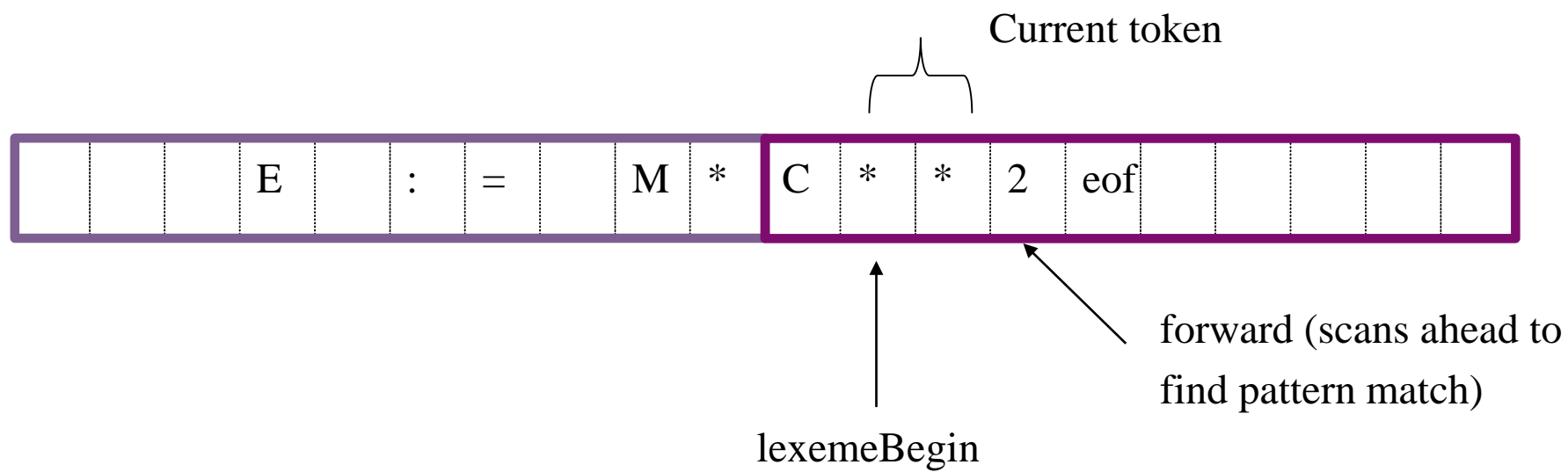
单字符I/O + 预读和回退——效率低下

磁盘 $\xrightarrow{\text{块I/O}}$ 缓冲区 $\xrightarrow{\text{单字符读取}}$ 词法分析器

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

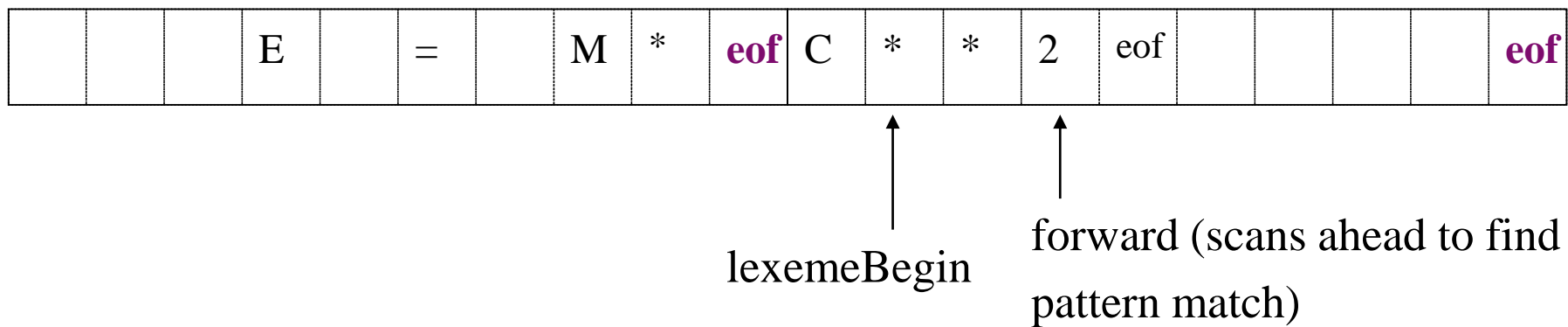
双缓冲技术图示



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 ; ← Block I/O  
    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 ; ← Block I/O  
    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
```

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 $\{\varepsilon\}$
2. 若 $a \in \Sigma$ ，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

○ $\Sigma = \{a, b\}$

1. $a \mid b$ $\{ a, b \}$

2. $(a \mid b)(a \mid b)$ $\{ aa, ab, ba, bb \}$

3. a^* $\{ \epsilon, a, aa, aaa, \dots \}$

4. $(a \mid b)^*$ $\{ \text{空串及所有由} a、b \text{组成的符号串} \}$

5. $a \mid a^*b$ $\{ a, b, \text{所有以多个} a \text{开头, 后跟一个} b \text{的符号串} \}$

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

$$(a \mid b)^* = (a^* b^*)^* \quad \{ \epsilon, a, b, aa, ab, ba, bb, \dots \}$$

正则表达式的代数定律

定律	描述
$r \mid s = s \mid r$	\mid 满足交换率
$r \mid (s \mid t) = (r \mid s) \mid t$	\mid 满足结合率
$(r \mid s) t = r (s t)$	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 \mid 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

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

...

$$d_n \rightarrow r_n$$

d_i 是不同的名字，并且不在 Σ 中

r_i 是 $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ （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

$(_ | \mathbf{A} | \dots | \mathbf{Z} | \mathbf{a} | \dots | \mathbf{z})(_ | \mathbf{A} | \dots | \mathbf{Z} | \mathbf{a} | \dots | \mathbf{z} | \mathbf{0} | \dots | \mathbf{9})^*$

$\text{letter_} \rightarrow \mathbf{A} | \mathbf{B} | \mathbf{C} | \dots | \mathbf{Z} | \mathbf{a} | \mathbf{b} | \dots | \mathbf{z} | _$

$\text{digit} \rightarrow \mathbf{0} | \mathbf{1} | \mathbf{2} | \dots | \mathbf{9}$

$\text{id} \rightarrow \text{letter_}(\text{letter_} | \text{digit})^*$

例：C语言无符号数

例如：5280, 0.012, 6.33E4, 1.8E-4

digit \rightarrow 0 | 1 | 2 | ... | 9

digits \rightarrow digit digit*

optional_fraction \rightarrow . digits | ϵ

optional_exponent \rightarrow (E (+ | - | ϵ) digits) | ϵ

num \rightarrow 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 : $(R|\epsilon)$

$[abce]$

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

$[a-z]$

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

$[^ab]$

- 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

$$letter_ \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _$$
$$digit \rightarrow 0 \mid 1 \mid \dots \mid 9$$
$$id \rightarrow letter_ (letter_ \mid digit)^*$$

简化为：

$$letter_ \rightarrow [A-Za-z_]$$
$$digit \rightarrow [0-9]$$
$$id \rightarrow letter_ (letter_ \mid digit)^*$$

例：用简写

- C语言无符号数的集合

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

$digits \rightarrow digit\ digit^*$

$optionalFraction \rightarrow .digits \mid \varepsilon$

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

$number \rightarrow digits\ optionalFraction\ optionalExponent$

简化为：

$digit \rightarrow [0-9]$

$digits \rightarrow digit^+$

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

正则表达式练习题

描述正则表达式表示的语言

- $0^*10^*10^*10^*$:
- $((\epsilon \mid 0) 1^*)^*$:

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) $(r) \mid (s)$ 是正规式，表示语言 $L(r) \cup L(s)$
- 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) $\neg(r)$ 是正规式，表示语言 $L(r)$

描述正则表达式表示的语言

- $0^*10^*10^*10^*$:
- $((\epsilon \mid 0)1^*)^*$:
- $(\epsilon 1^* \mid 01^*)^*$
- $(1^* \mid 01^*)^*$

包含3个1的01串

所有01串

$\begin{array}{l} r(s \mid t) = rs \mid rt \\ (s \mid t)r = sr \mid tr \end{array}$	连接和 \mid 满足分配率
-------------------------------------------------------------------------------------	------------------

$$\begin{array}{l} \epsilon r = r \\ r \epsilon = r \end{array}$$

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, ...

$(1|0)^*$

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

非正则表达式集

正则表达式无法描述的语言

- $\{wcw \mid w \text{ 是 } a、b \text{ 组成的符号串} \}$
- 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

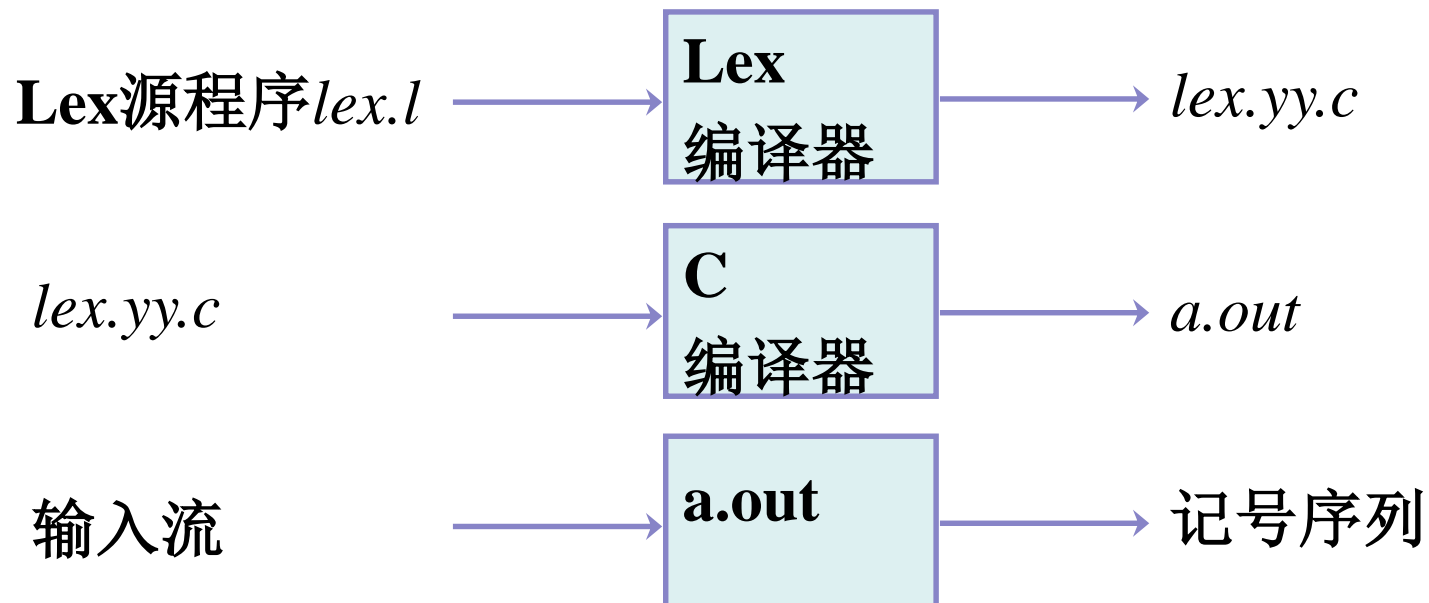
- 3.1 词法分析器的作用
- 3.2 输入缓冲
- 3.3 词法单元的描述
- 3.4 词法单元的识别
- 3.5 词法分析器的构造

词法分析器的自动产生

用Lex建立词法分析器的步骤

Lex 编程可以分为三步：

- 以 Lex 可以理解的格式指定模式相关的动作。
- 在这一文件上运行lex文件，生成扫描器的 C 代码。
- 编译和链接 C 代码，生成可执行的扫描器。



Lex程序结构

一个 Lex 程序分为三个部分：

- 第一部分是 C 和 Lex 的全局声明
- 第二部分包括模式（C 代码）
- 第三部分是补充的 C 函数。一般都有 main() 函数

这三个部分以%%来分界。

declarations

声明

%%

translation rules

转换规则

%%

auxiliary procedures

辅助过程

declarations 声明

可以提供 C 变量声明、常量定义、头文件包含等

```
%{
```

```
int wordCount = 0;
```

```
%}
```

标识(token)的正则表达式定义

```
chars      [A-Za-z]
```

```
words      {chars}+
```

translation rules 转换规则

Lex程序的转换规则是如下形式的语句:

p_1	{ action ₁ }
p_2	{ action ₂ }
...	...
p_n	{ action _n }

其中, p_i 是正则表达式, 每个动作**action**_{*i*}表示匹配该表达式成功后, 词法分析器要执行的程序段(用C语言编写) 应该执行的代码。这些代码都将放在函数**yylex()**中。

%%

{ words} { wordCount++; /* increase the word count by one*/ }

auxiliary procedures 辅助过程

```
%%  
void main()  
{  
    yylex();    /* start the analysis*/  
    printf("No of words:%d\n", wordCount);  
}
```

Lex词法分析器工作方式

- 词法分析器，从尚未扫描的输入字符串中读字符，每次读入一个字符，直到发现能与某个正规表达式 p_i 匹配的最长前缀。
- 词法分析器执行`actioni`。
- 词法分析器这种不断查找词素，直到以显示的`return`调用结束工作的方式，使其可以方便的处理空白符和注释。
- 词法分析器只返回记号给语法分析器，带有与词素相关信息的属性值是通过全局变量`yylval`传递的。

Lex举例

正则表达式	记号	属性值
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	指向符号表表项的指针
num	num	指向符号表表项的指针
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Lex 举例

```
% {  
    /* 符号常数定义  
    LT, LE, EQ, NE, GT, GE,  
    IF, THEN, ELSE, ID, NUMBER, RELOP */  
% }  
/* 正规定义 */  
delim          [ \t \n]  
ws              { delim } +  
letter          [ A-Za-z ]  
digit           [ 0-9 ]  
id              { letter } ( { letter } | { digit } ) *  
number          { digit } + ( \. { digit } + ) ? ( E [ + \- ] ) ? { digit } + ) ?  
% %
```

Lex 举例

```
% %  
  
{ws}          { /* 没有动作和返回值 */ }  
  
if             { return(IF); }  
then          { return(THEN); }  
else          { return(ELSE); }  
  
{id}          { yylval = install_id(); return(ID); }  
  
{number}      { yylval = install_num(); return(NUMBER); }  
  
“<”          { yylval = LT; return(RELOP); }  
  
“<=”         { yylval = LE; return(RELOP); }  
  
“=”          { yylval = EQ; return(RELOP); }  
  
“<>”         { yylval = NE; return(RELOP); }  
  
“>”          { yylval = GT; return(RELOP); }  
  
“>=”        { yylval = GE; return(RELOP); }  
  
% %
```

Lex举例

```
%%
```

```
install_id()
```

```
{
```

```
/*往符号表填入词素的过程，yytext指向词素的第一个字符，yyleng表示词素的长度。将词素填入符号表，返回指向该词素所在表项的指针*/
```

```
}
```

```
install_num()
```

```
{
```

```
/*与填写词素的过程类似，只不过词素是一个数。*/
```

```
}
```


Lex 举例

```
% {  
int wordCount = 0;  
% }
```

C和Lex的全局声明

为字数统计程序声明一个整型变量，保存统计得到的字数

```
chars      [A-Za-z]  
delim      [ \n\t]  
whitespace {delim}+  
words      {chars}+
```

模式匹配规则

使用 C 语句来定义标记

匹配后的动作

```
% %  
{ words }      { wordCount++; /* increase the word count by one*/ }
```

```
{ whitespace }  { /* do nothing*/ }  
\n|.             { /* gobble up */ }
```

```
% %  
void main()  
{  
    yylex();          /* start the analysis*/  
    printf("No of words:%d\n", wordCount);  
}
```

Lex变量和函数

Lex 有几个函数和变量提供了不同的信息，可以用来编译实现复杂函数的程序。下面列出了一些变量和函数，以及它们的使用。

Lex 变量

`yyin` `FILE*` 类型。指向 `lexer` 正在解析的当前文件。

`yyout` `FILE*` 类型。指向记录 `lexer` 输出的位置。

缺省情况下，`yyin` 和 `yyout` 都指向标准输入和输出。

`yytext` 匹配模式的文本存储在这一变量中（`char*`）。

`yyleng` 给出匹配模式的长度。

Lex的匹配策略

elsen = 0;

1	else	n	=	0
2	elsen	=	0	

当输入的多个前缀与一个或多个模式匹配时，lex利用如下规则选择正确的词素：

总是选择最长的前缀

如果最长的可能前缀与多个模式匹配，总是选择在Lex程序中先被列出的模式。

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

$delim \rightarrow \mathbf{blank} \mid \mathbf{tab} \mid \mathbf{newline}$

$ws \rightarrow delim^+$

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

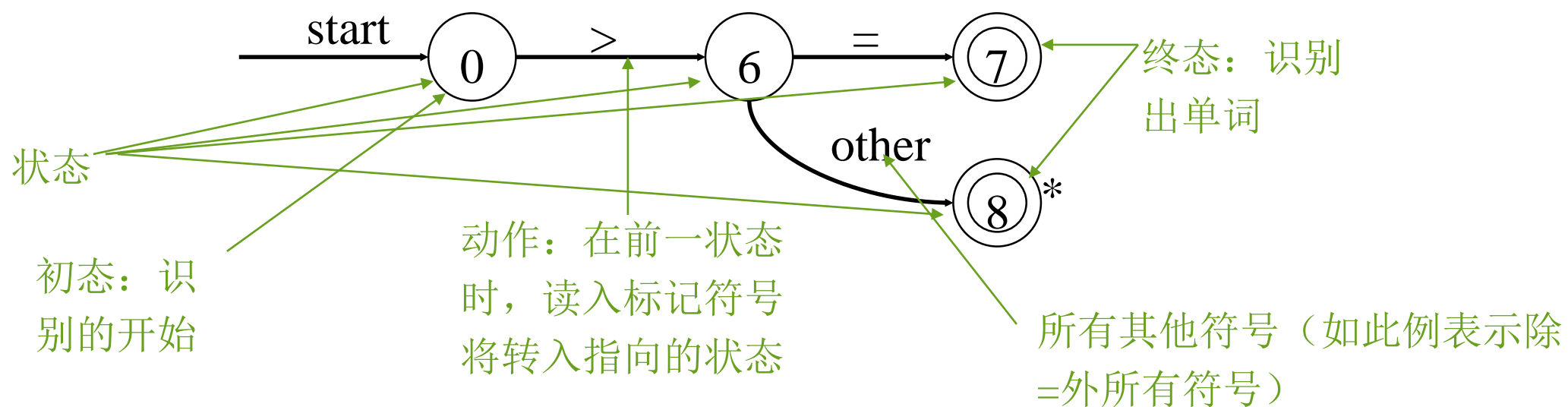
状态转换图(transition diagram, TD)

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

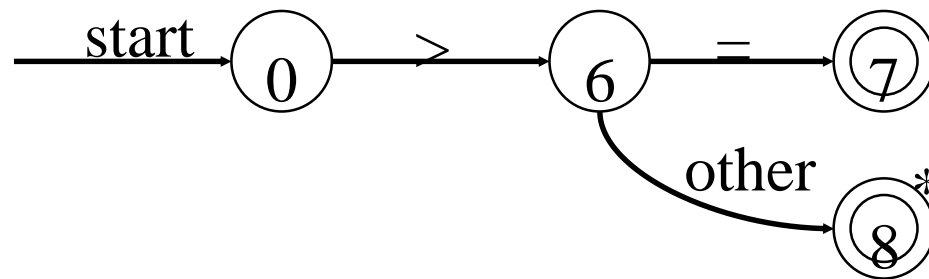
如何识别单词？

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 \leftrightarrow 模式
- 状态 \leftrightarrow 已读入符号串
- 边 \leftrightarrow 下一符号、应采取的动作

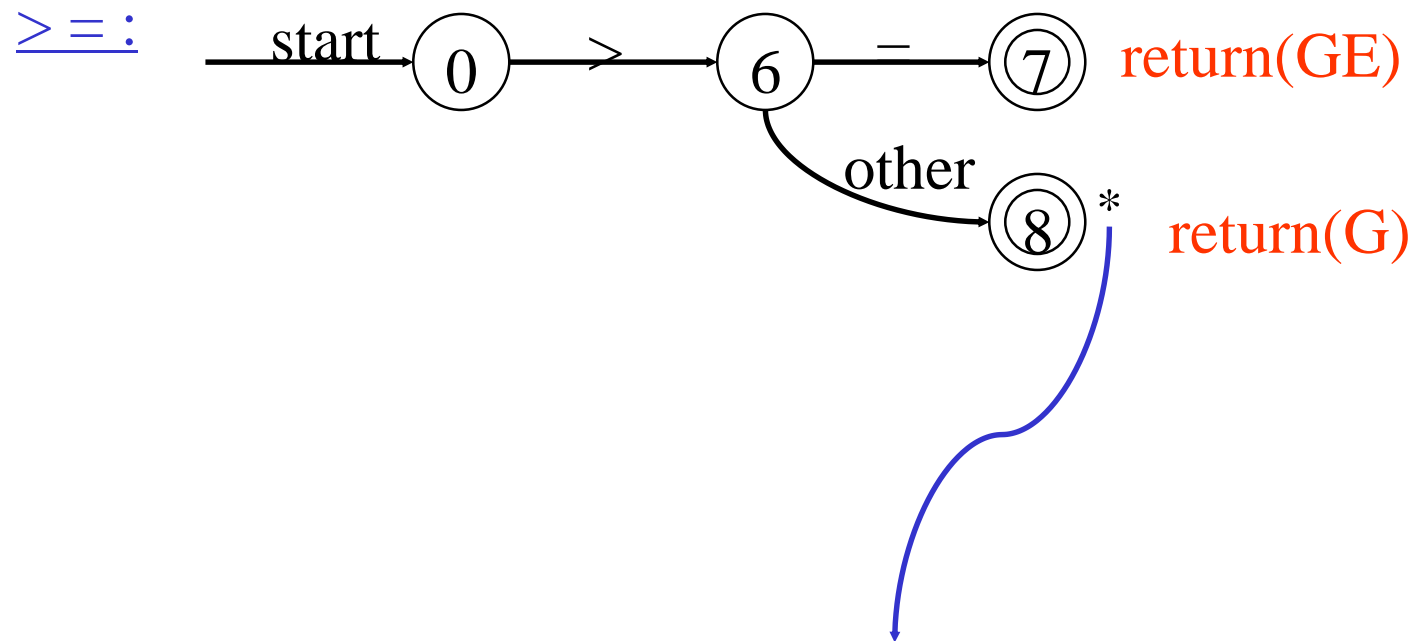


如何识别单词？

TD工作方式

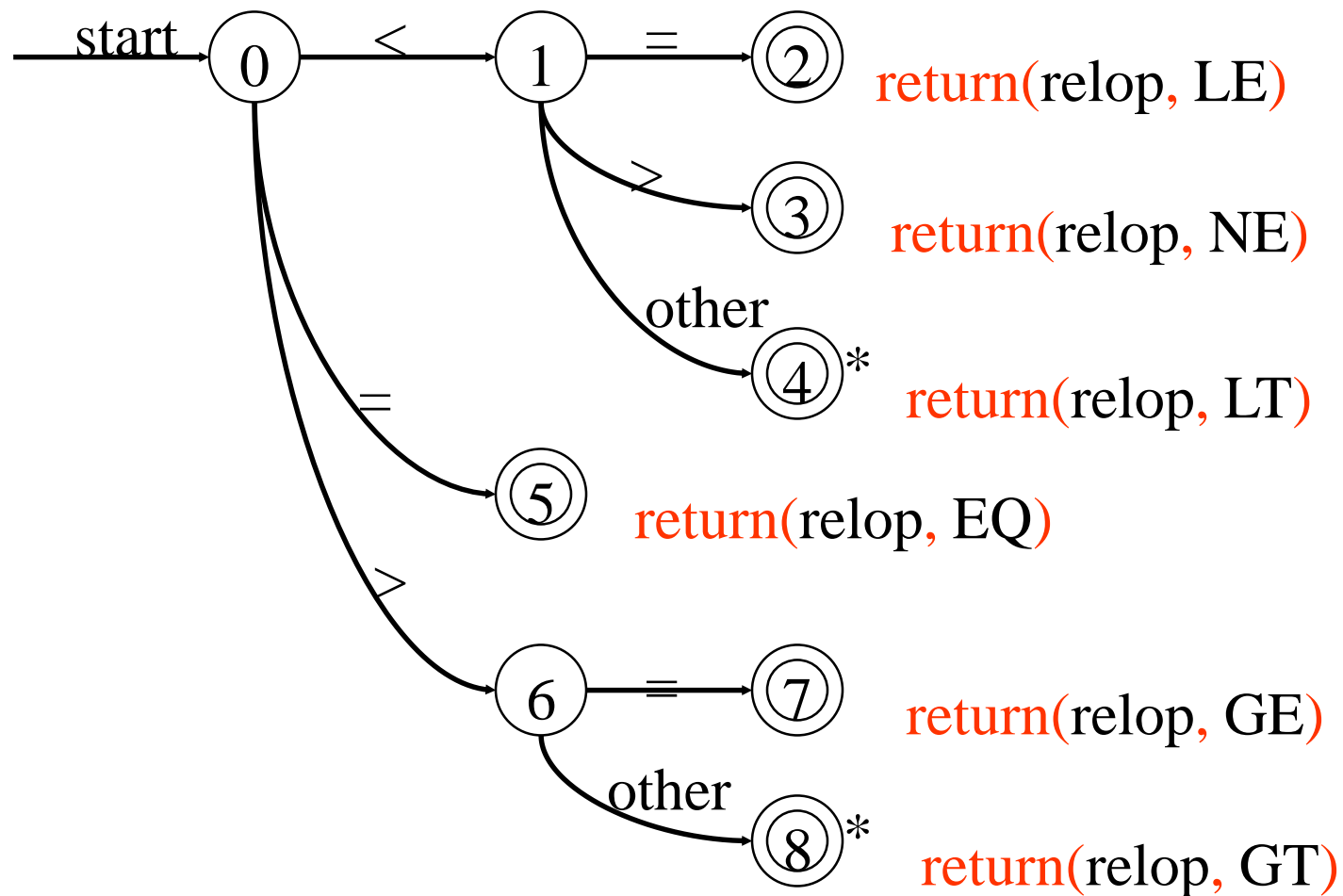
- 开始识别，初态
- 读入符号，转换状态
- 终态，接受！； 无法转换，失败！

例



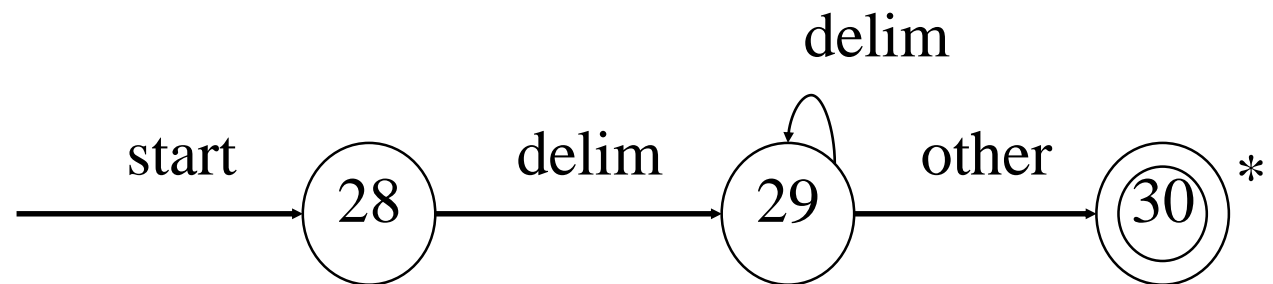
已经接受 “>”，且已经多读取一个其他符号，需退回这个符号

例：所有关系运算符



例：空白符

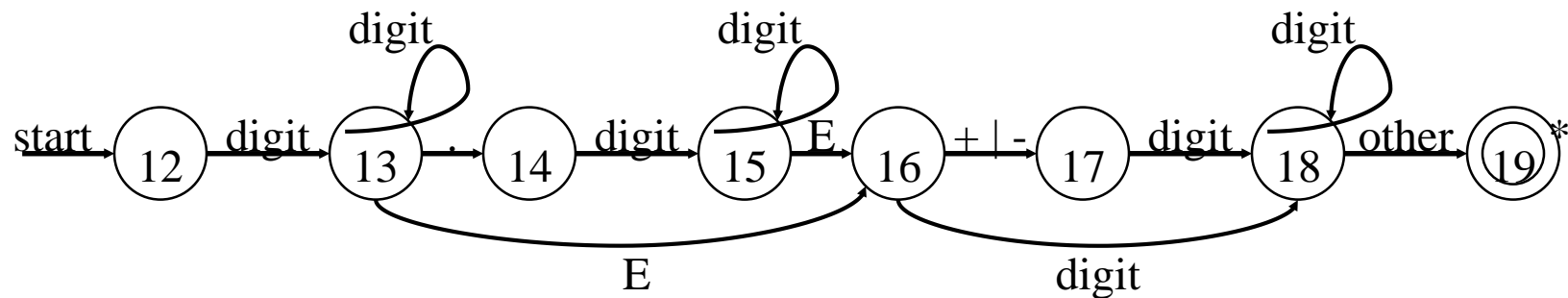
delim :



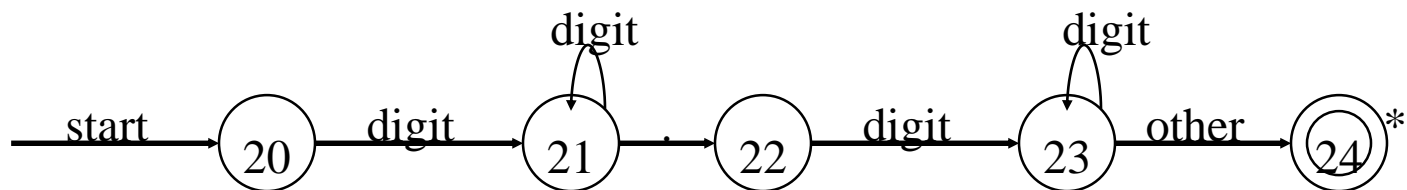
例：无符号数

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

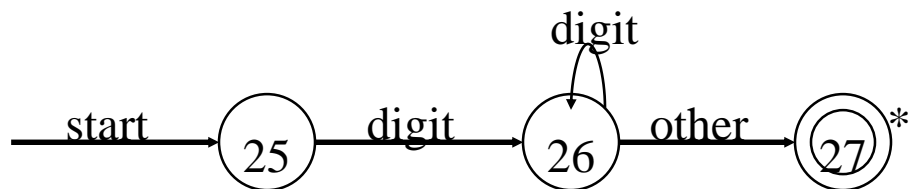
12.3E4



12.3



12



保留字的处理

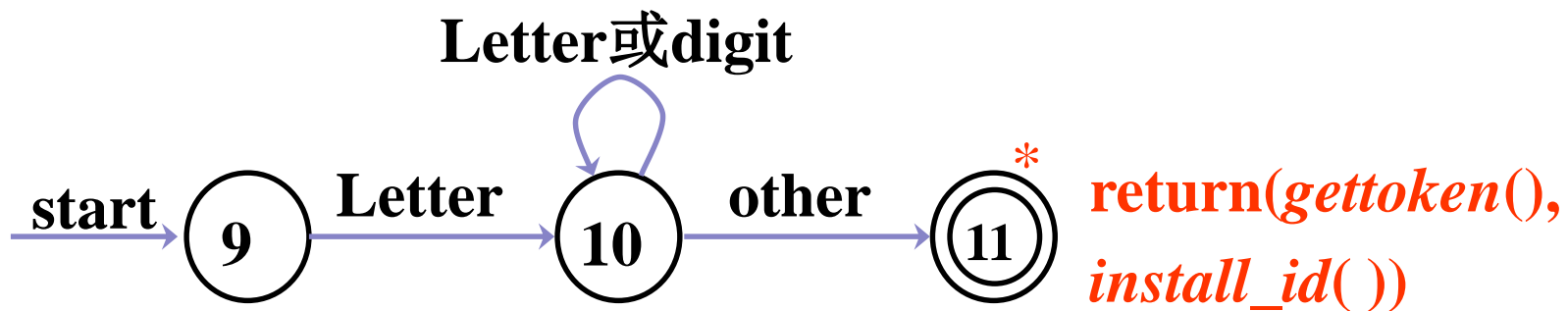
关键字/保留字与标识符一样进行匹配

- 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

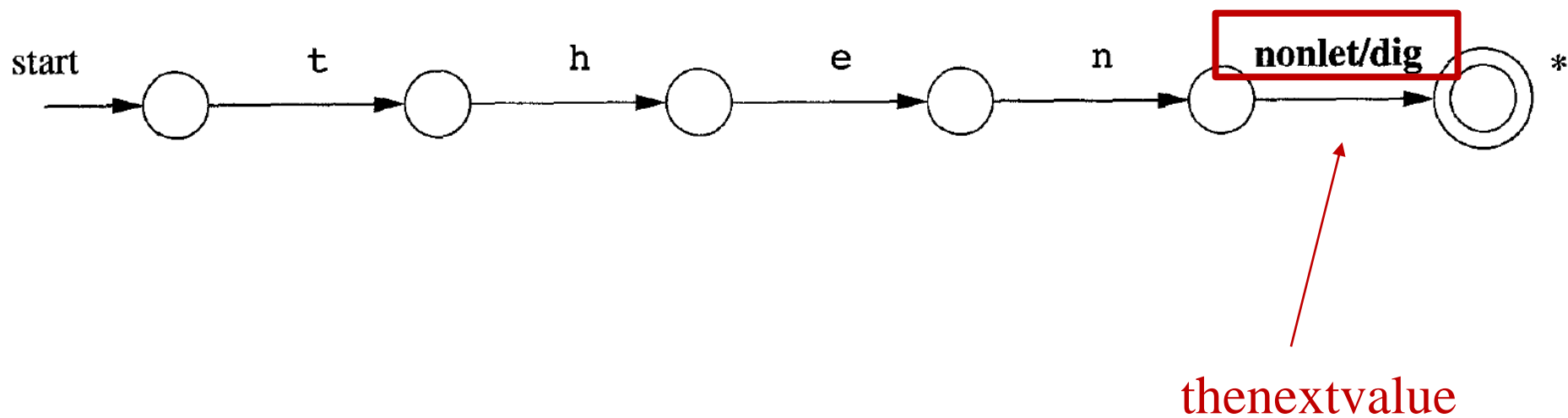
`return(gettoken(), install_id())`返回记号和属性值`lexical_value`;

- `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

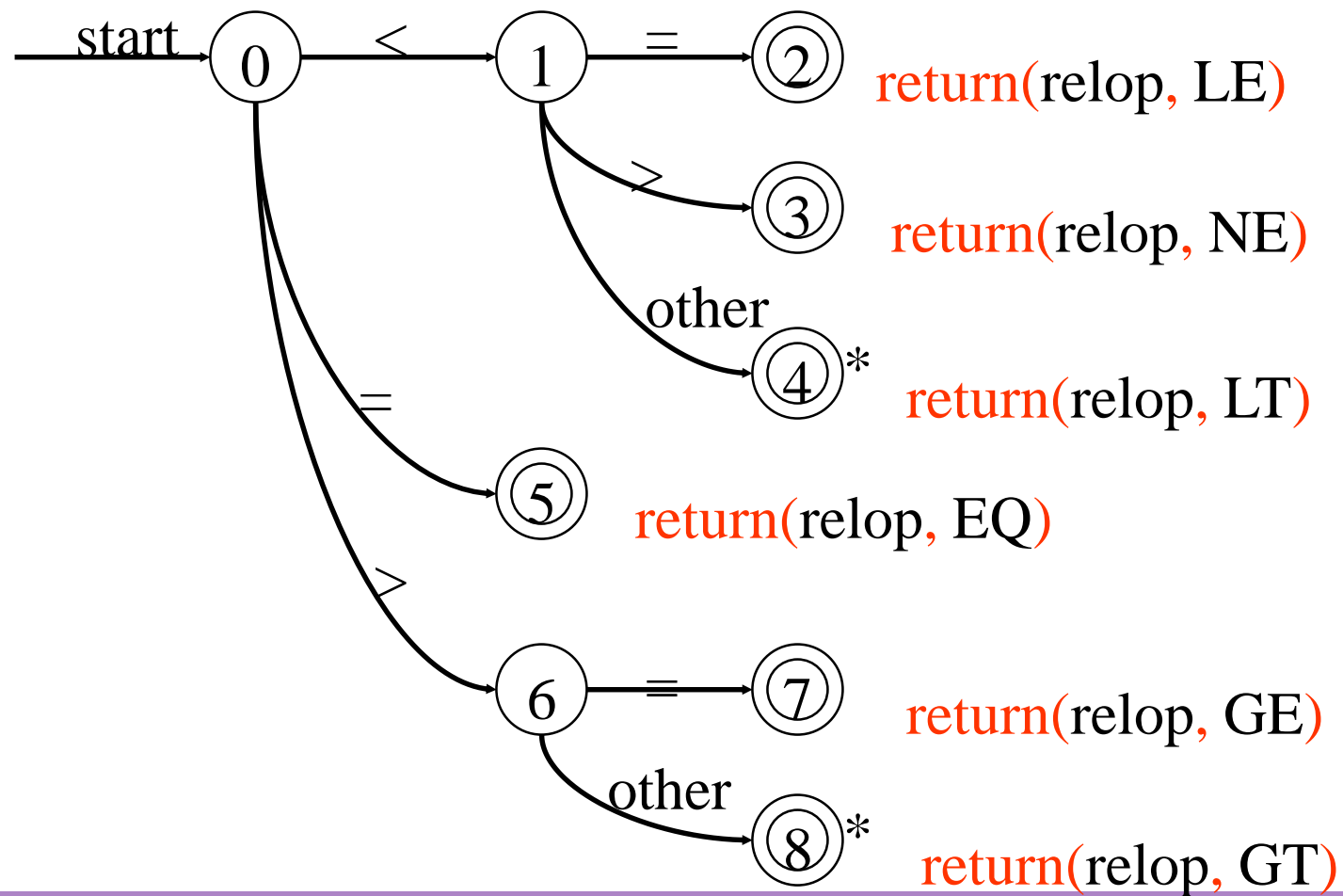
为每个状态构造一段代码

- 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

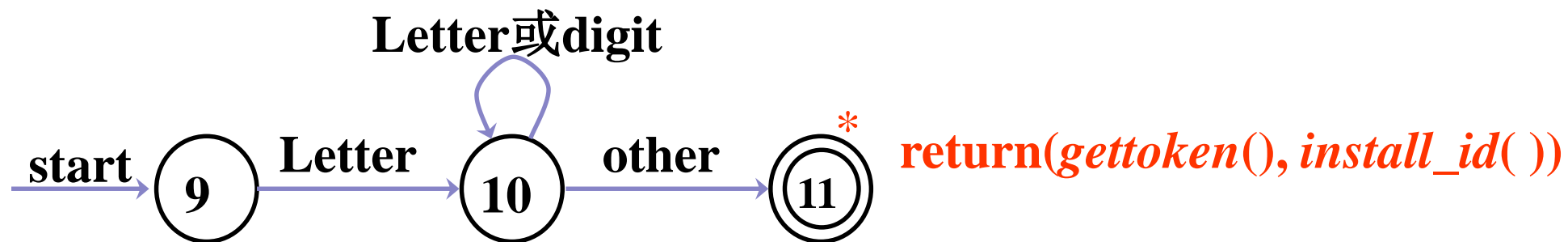
错误处理

- 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

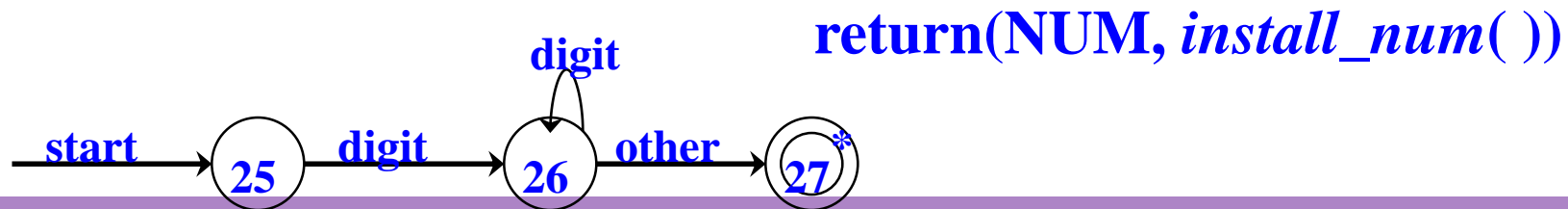
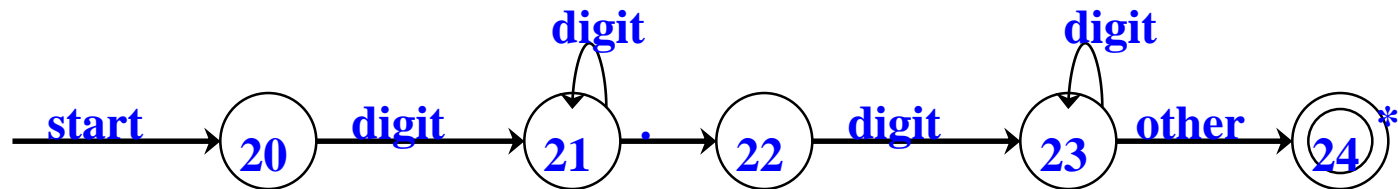
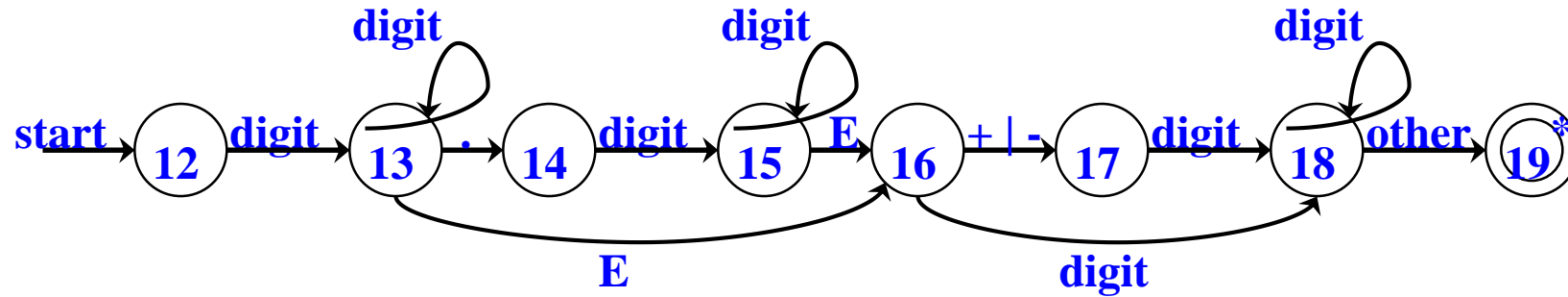


例：所有状态转换图



例：所有状态转换图


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



实现代码

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

实现代码

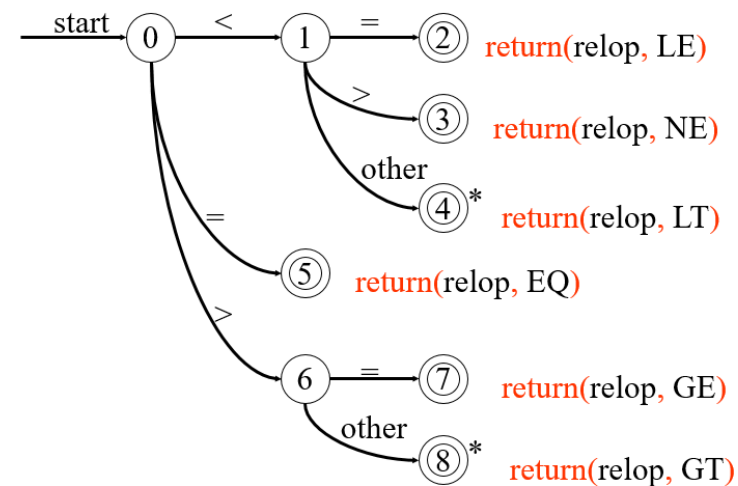
作用?  `/* c is lookahead character */
if (c== blank || c==tab || c== newline) {
 state = 0;
 lexeme_beginning++;
 /* advance beginning of lexeme */
}`

实现代码

不是比较运算符，
怎么办？



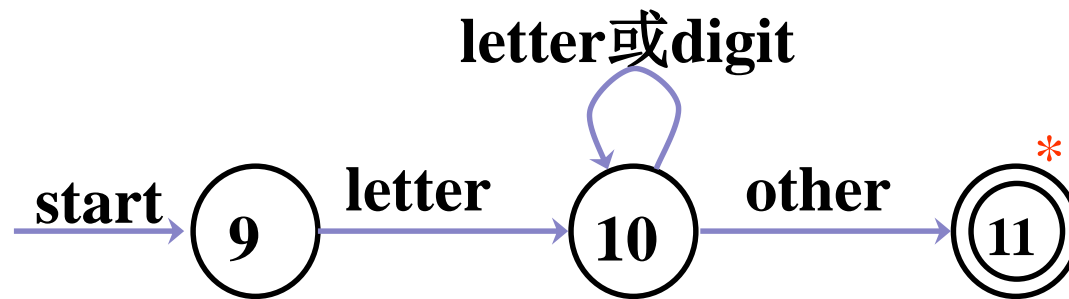
```
else if (c == '<') state = 1;  
else if (c == '=') state = 5;  
else if (c == '>') state = 6;  
else state = fail();  
break;  
... /* cases 1-8 here */
```



实现代码

case 9:

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



实现代码

case 10:

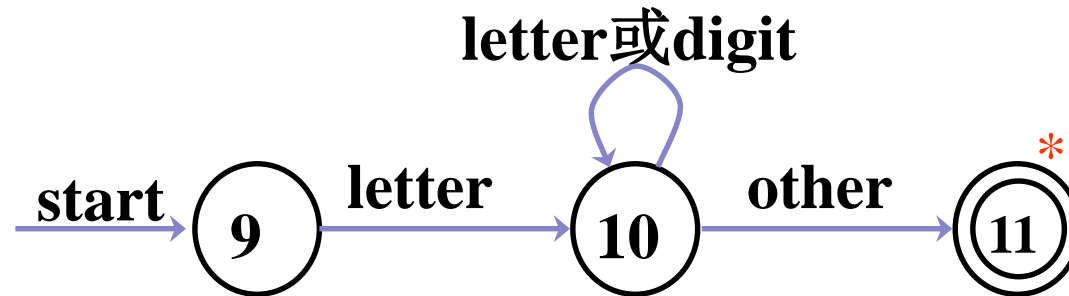
c = nextchar();

if (isletter(c)) state = 10;

else if (isdigit(c)) state = 10;

else state = 11;

break;



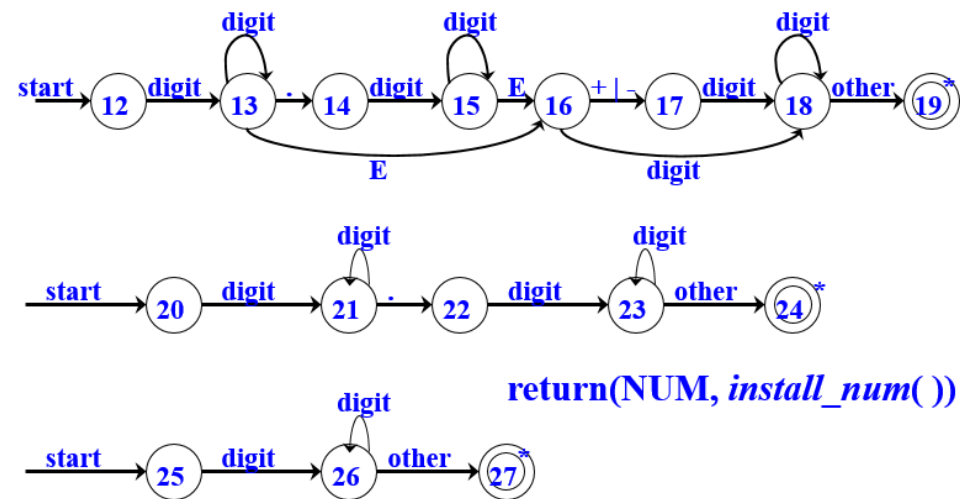
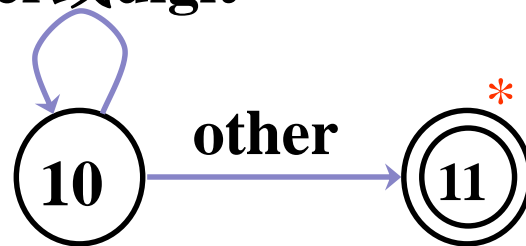
实现代码

```
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;
```

letter或digit



实现代码

case 26:

```
    c = nextchar();
```

```
    if (isdigit(c)) state = 26;
```

```
    else state = 27;
```

```
    break;
```

case 27:

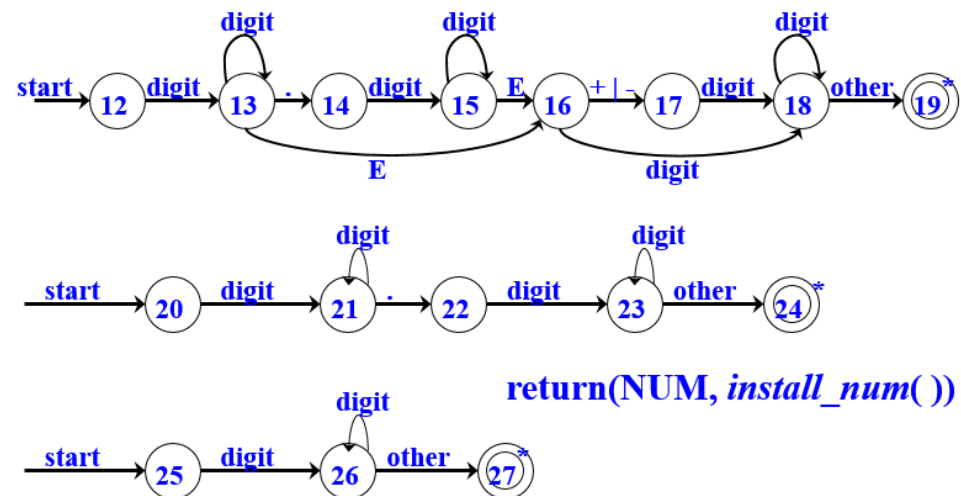
```
    retract(1); install_num();
```

```
    return ( NUM );
```

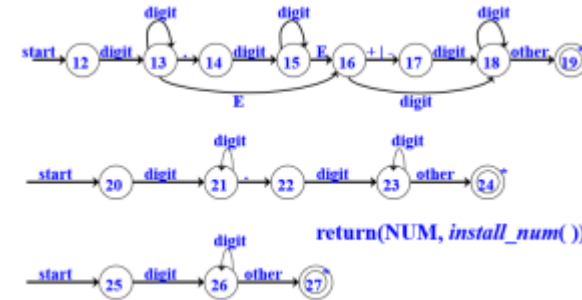
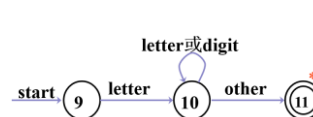
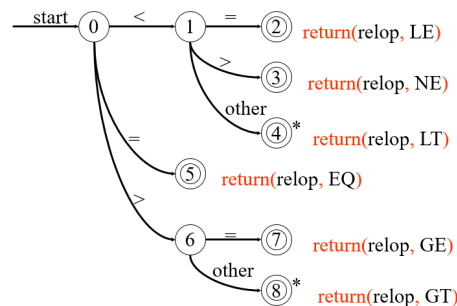
```
}
```

```
}
```

```
}
```



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 \longrightarrow 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 $\xrightarrow{\text{简单}}$ NFA
- “不精确”，占用内存少，速度慢

DFA

- 正则表达式 $\xrightarrow{\text{复杂}}$ DFA
- “精确”，占用内存大，速度快

NFA \rightarrow DFA, DFA \rightarrow 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 : $S \times \Sigma$ 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

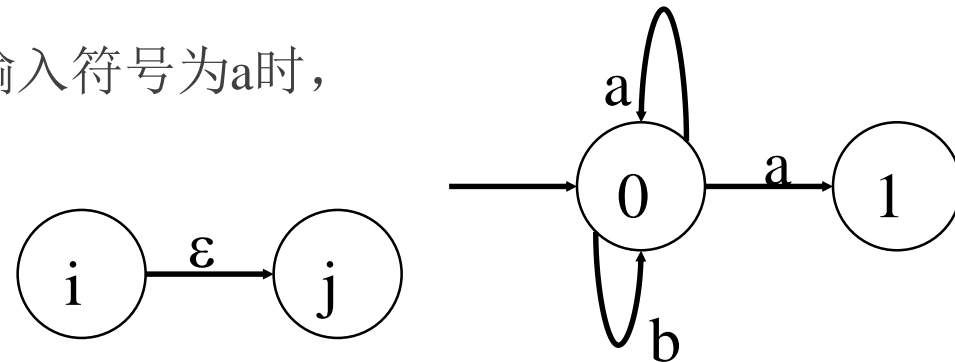
表示方式

五元组

- δ 用函数形式表示, $\delta(s, a) = S'$ 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

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

$s_0 = 0$

$F = \{ 3 \}$

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

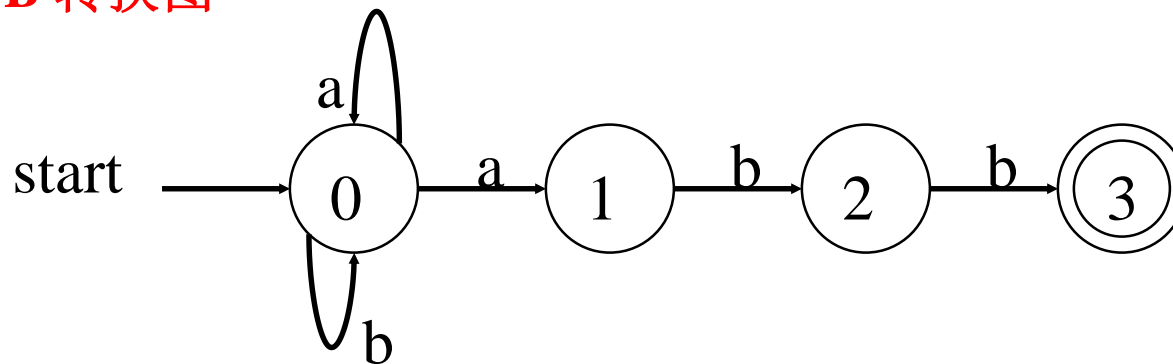
$\delta(0,a) = \{ 0, 1 \}$

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

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

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

B 转换图



C 转换表

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

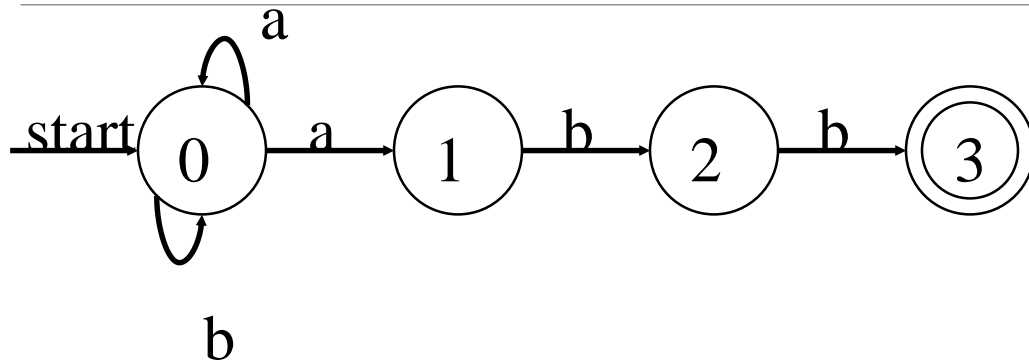
NFA如何工作？

NFA接受（**accept**）符号串 x

\leftrightarrow 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)

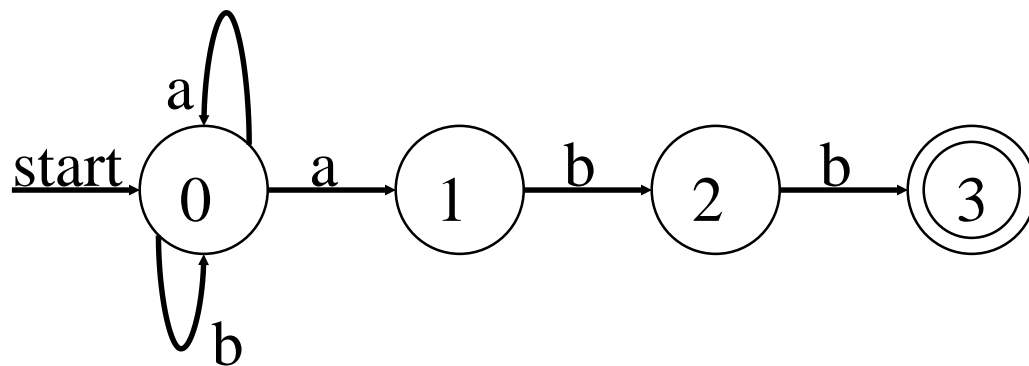
REJECT !

-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）

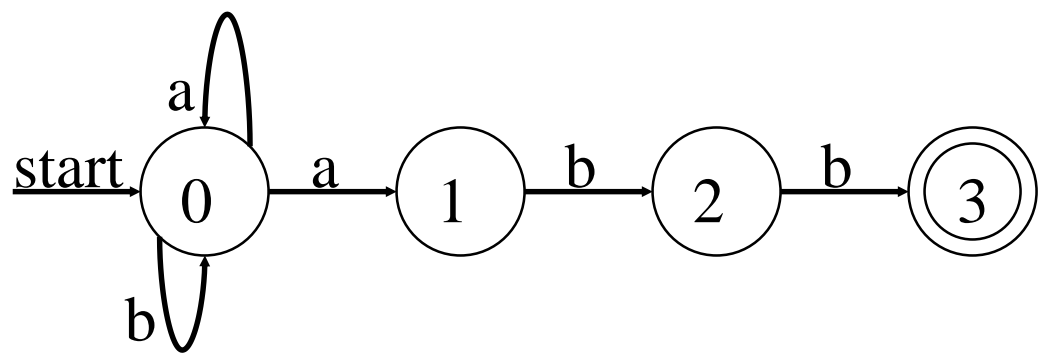


不是所有路径都表示接受

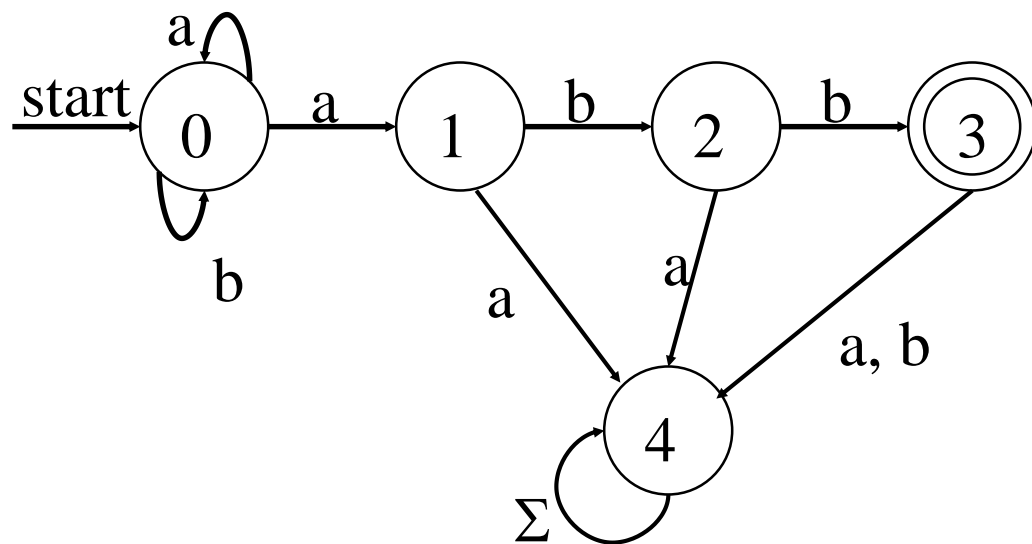
aabb

- 路径 $0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ 表示接受
- 而 $0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0$ 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

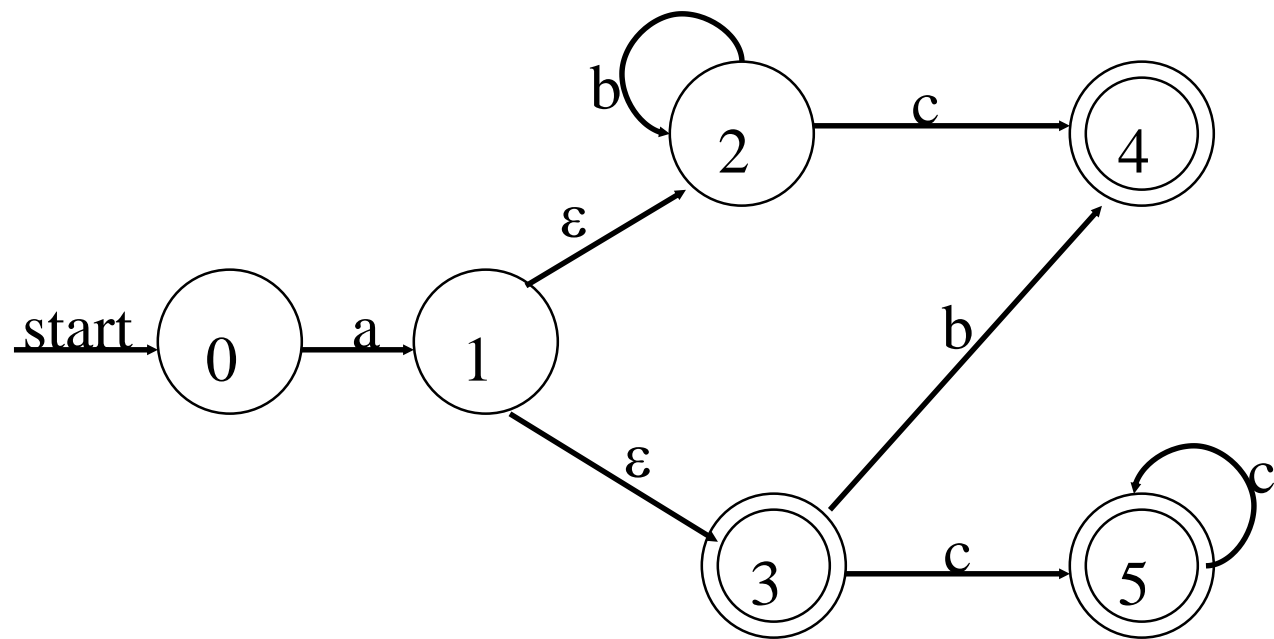
单词 \leftrightarrow 模式 \leftrightarrow 正则表达式 \leftrightarrow NFA $\leftarrow? \rightarrow$ 词法分析程序

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

$(a(b^*c)) \mid (a(b \mid c^+)?)$

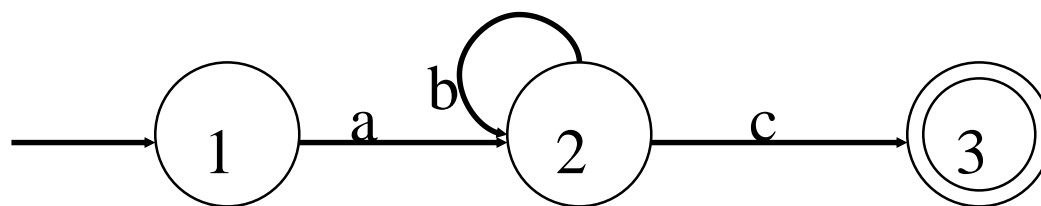


○ 可接受 **abbc**

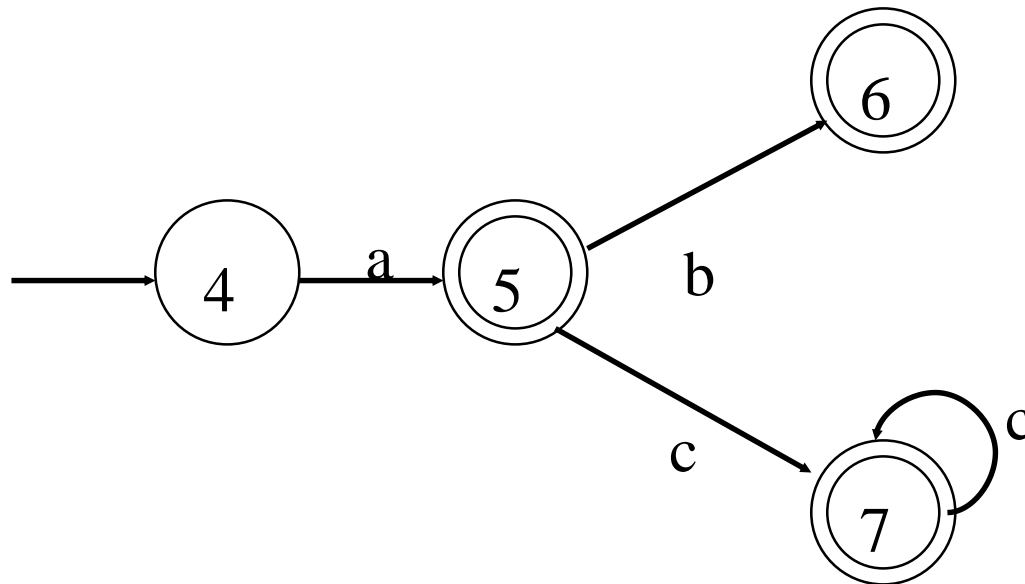
另一解法

$(a(b^*c)) \mid (a(b \mid c^+)?)$

$a(b^*c)$

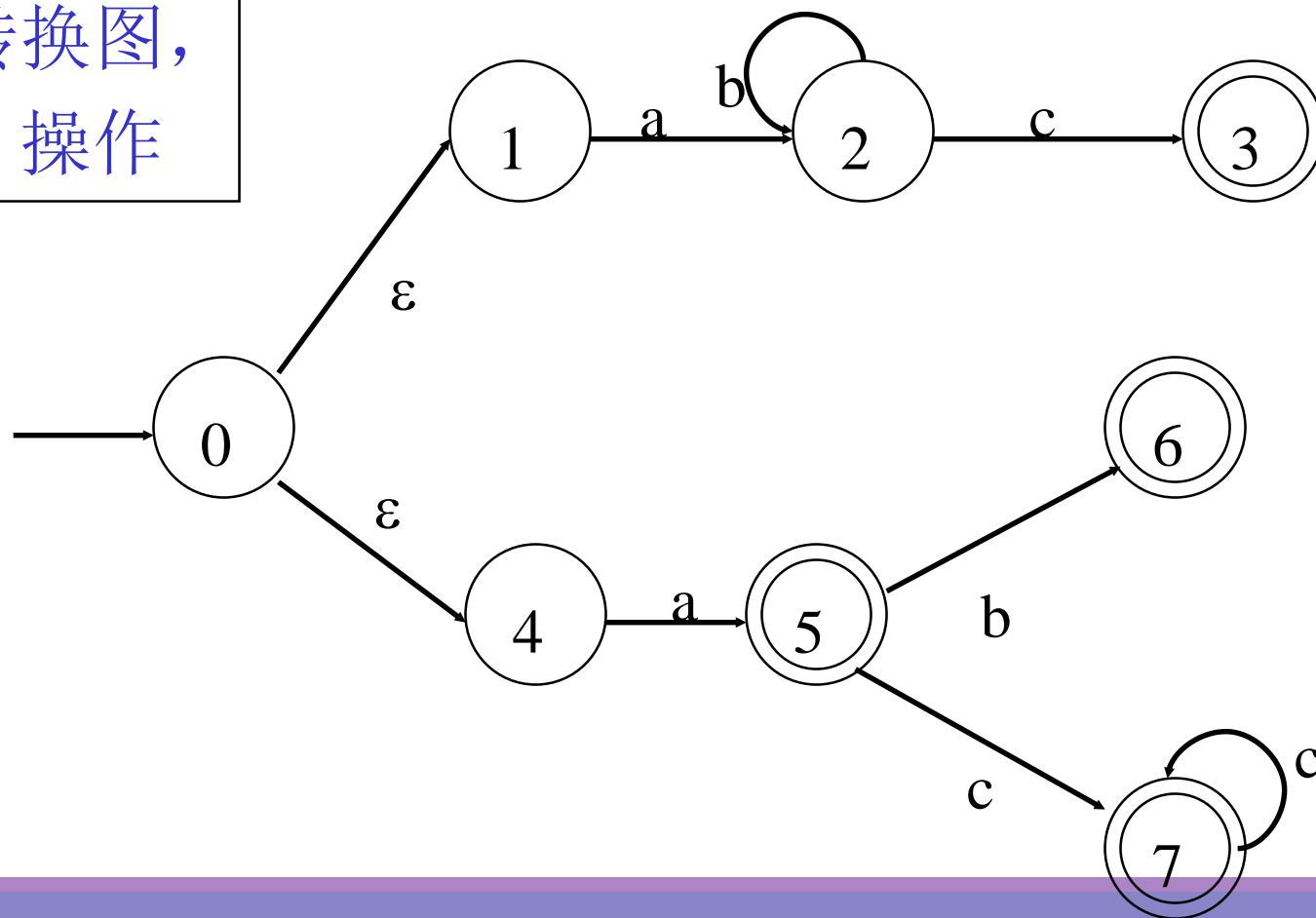


$a(b \mid c^+)?$



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

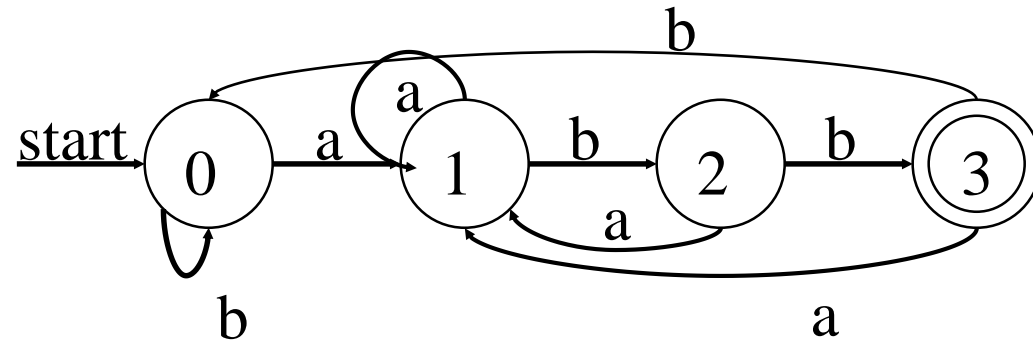
1. S : 有限状态集
2. Σ : 有穷字母表
3. $\delta: S \times \Sigma$ 到 S 的单值映射, 即
 $\delta: S \times \Sigma \rightarrow S$
4. $s_0 \in S$ 是唯一的初态
5. $F \subseteq S$ 是一个终态集

没有 ϵ

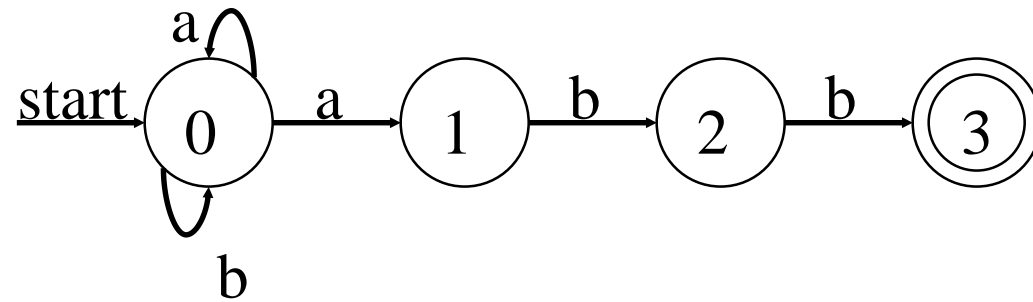
值域为 S , 不是 S
的子集的集合

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

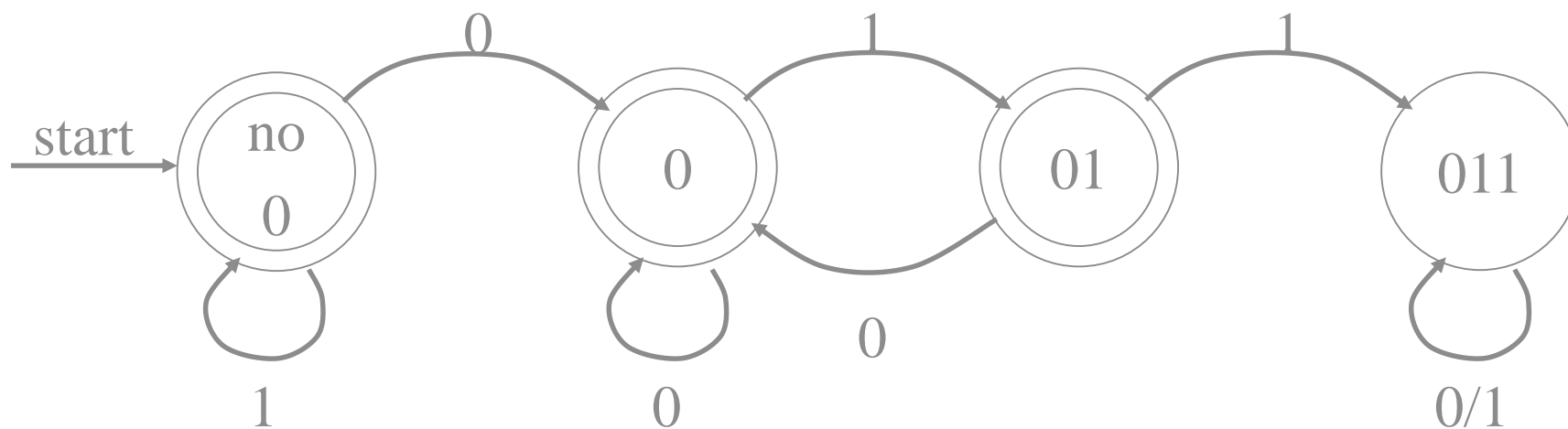
输出：若D接受x，返回“yes”，否则返回“no”

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”
```

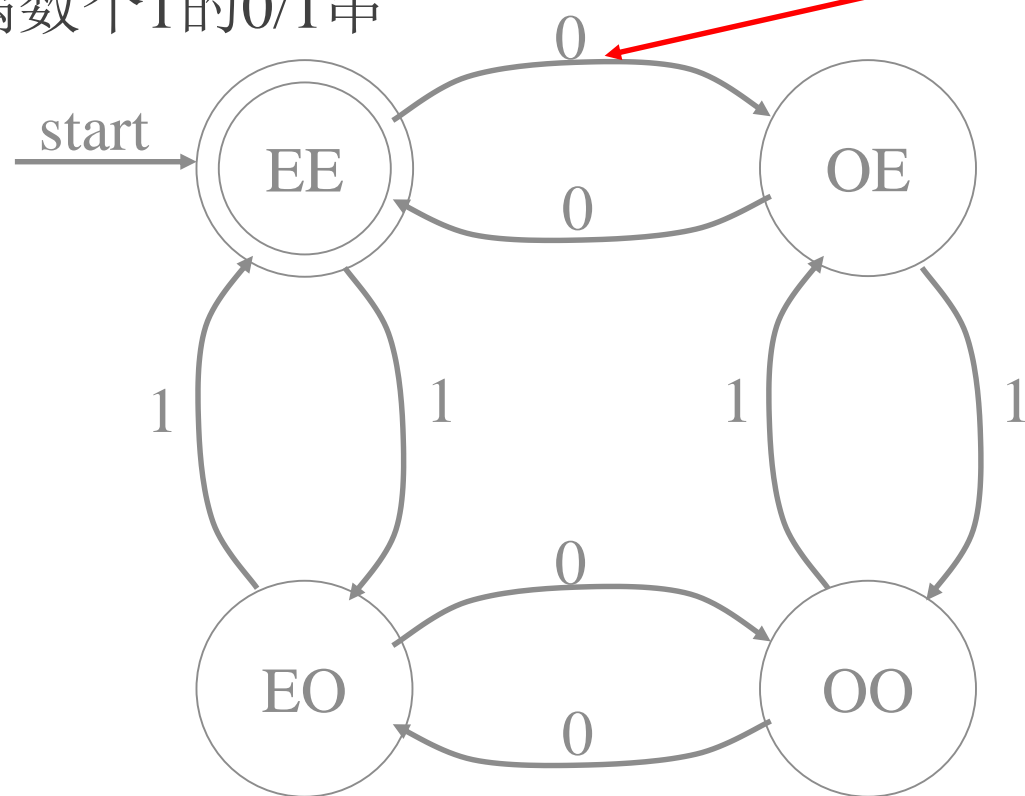

设计自动机

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

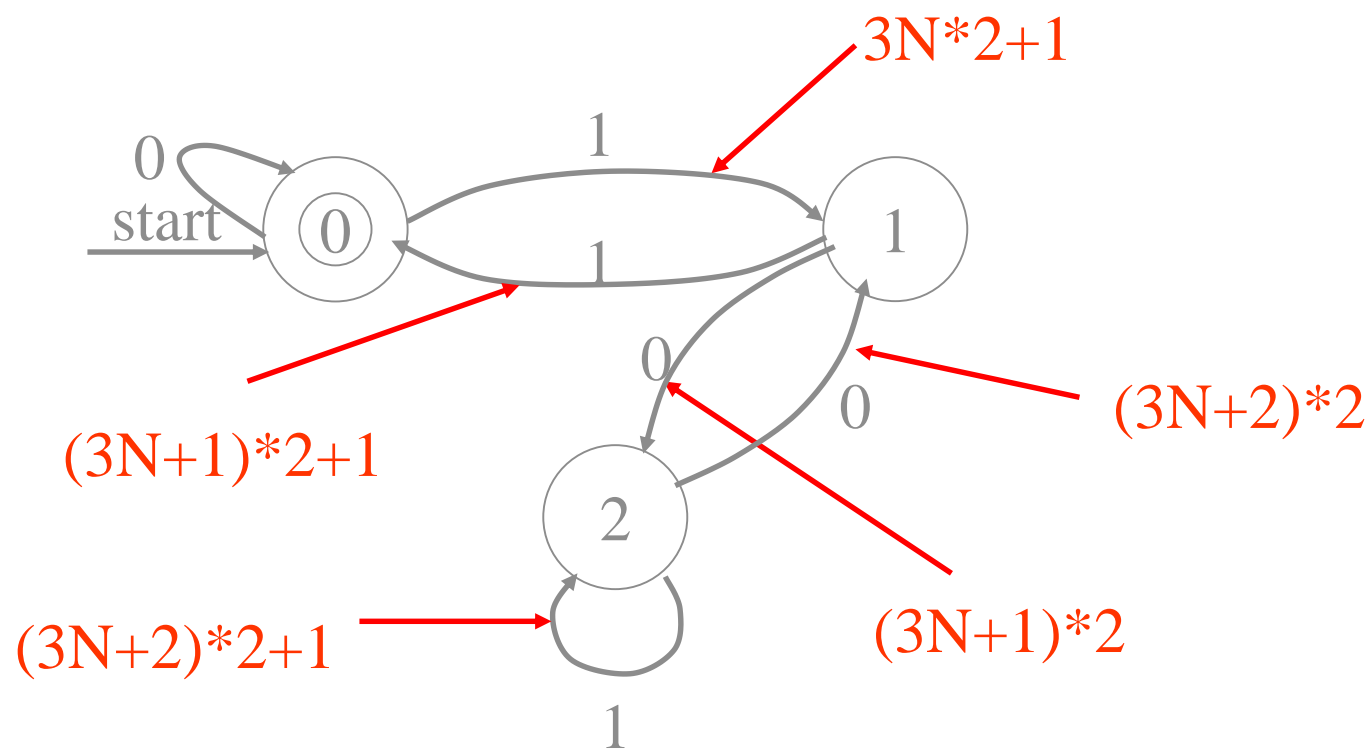
另一种思路：状态——符号串集合
偶数个0，偶数个1的0/1串



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、 $\epsilon \rightarrow$ 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
 \rightarrow 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 ， $L(N)=L(r)$

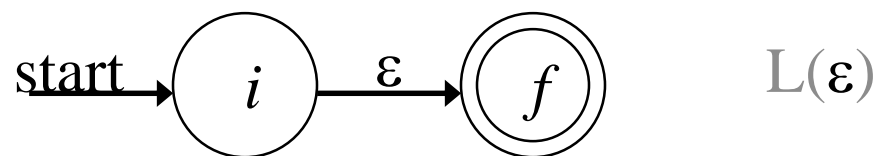
Thompson构造法

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

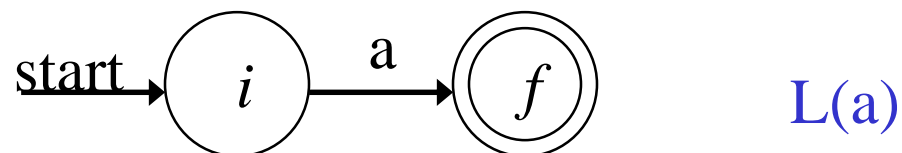
算法：正则表达式 \rightarrow NFA

构造规则

1. 对于 ε ，构造NFA



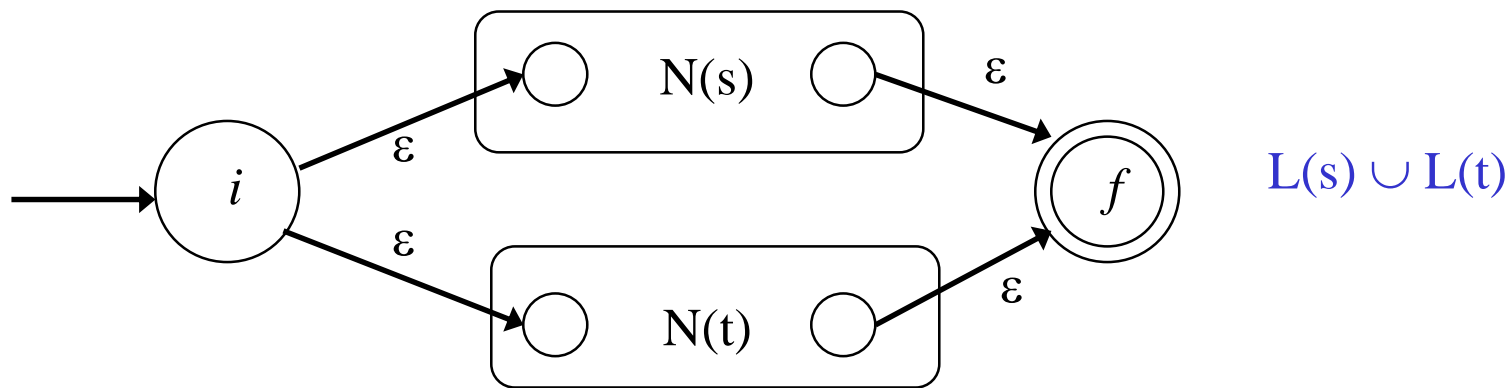
2. 对于 $a \in \Sigma$ ，构造NFA



算法：正则表达式 \rightarrow 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 $N(s|t)$



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

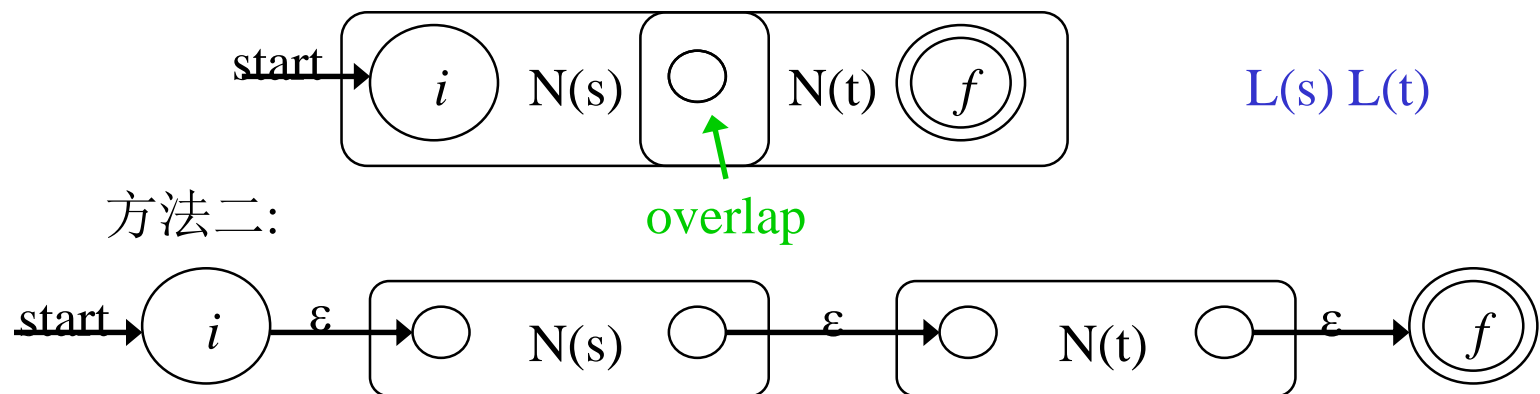
四条新 ϵ 边：

$i \rightarrow N(s)\text{初态}$ $i \rightarrow N(t)\text{初态}$

$N(s)\text{终态} \rightarrow f$ $N(t)\text{终态} \rightarrow f$

算法：正则表达式 \rightarrow NFA

b) 对正则表达式 st ，构造如下组合NFA $N(st)$

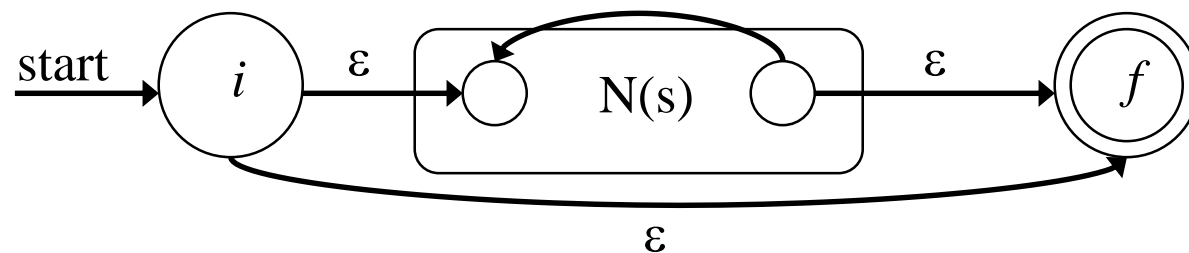


方法一：新初态—— $N(s)$ 初态，
新终态—— $N(t)$ 终态
 $N(s)$ 终态与 $N(t)$ 初态合并

方法二：
新初态、终态—— i 、 f
三条新 ϵ 边： $i \rightarrow N(s)$ 初态 $N(t)$ 终态 $\rightarrow f$
 $N(s)$ 终态 $\rightarrow N(t)$ 初态

算法：正则表达式 \rightarrow 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： $i \rightarrow f$

$i \rightarrow N(s)$ 初态 $N(s)$ 终态 $\rightarrow f$

$N(s)$ 终态 $\rightarrow N(s)$ 初态

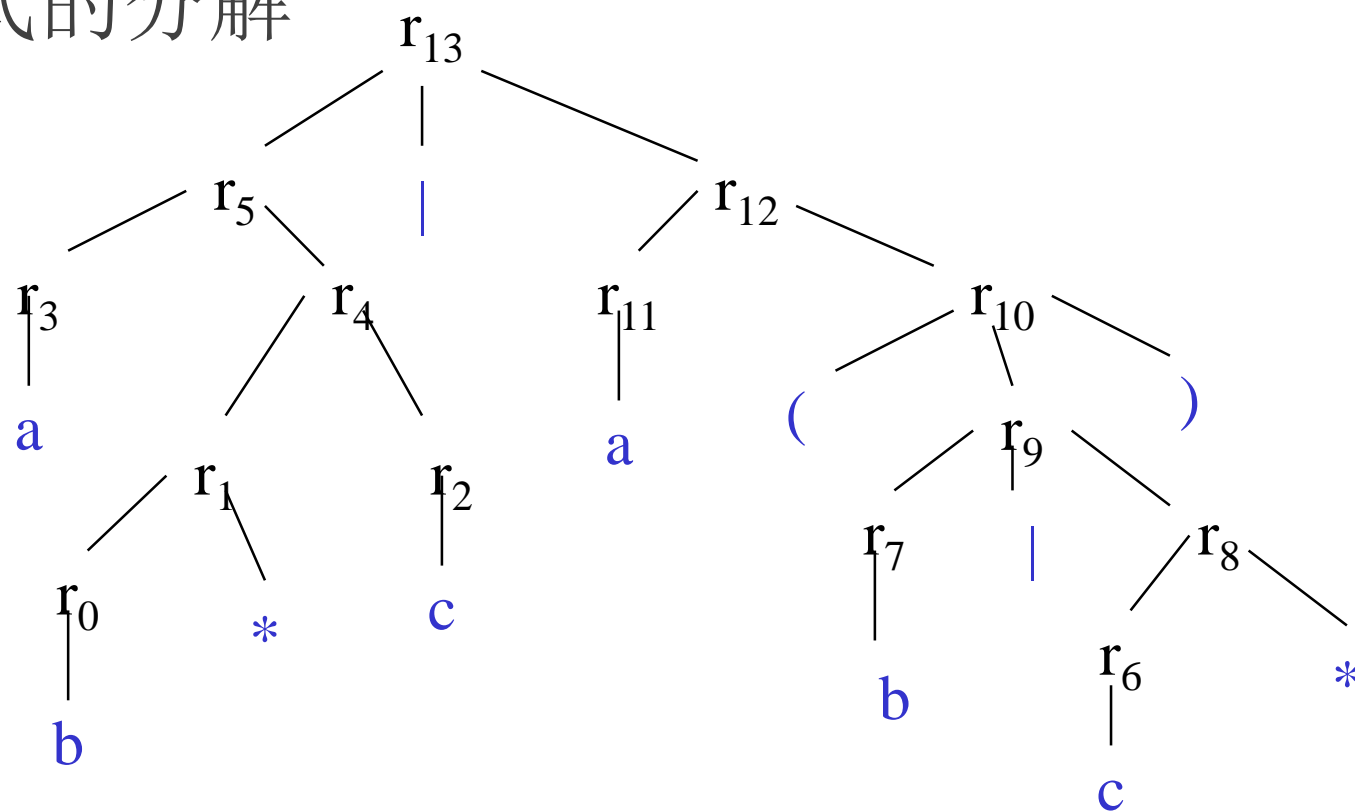
d) 对正则表达式 (s) ， $N((s)) = N(s)$

算法特性

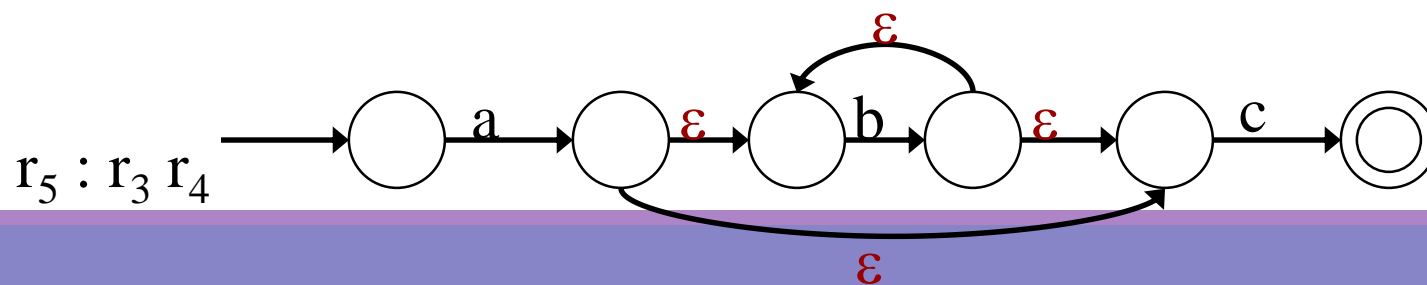
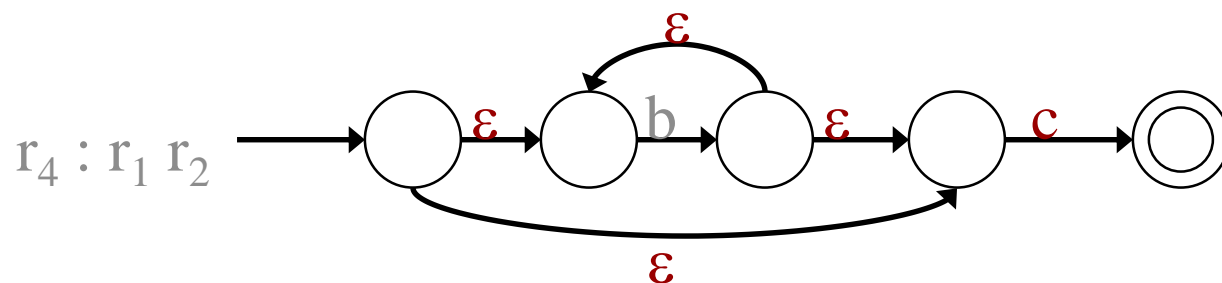
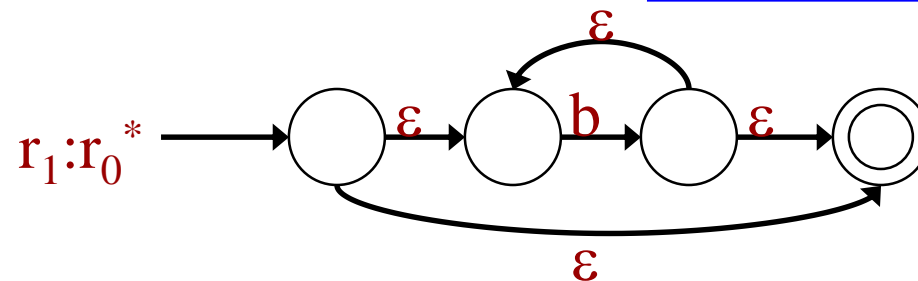
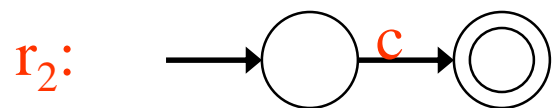
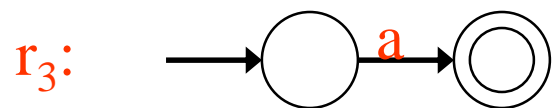
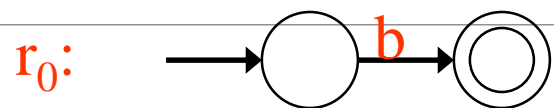
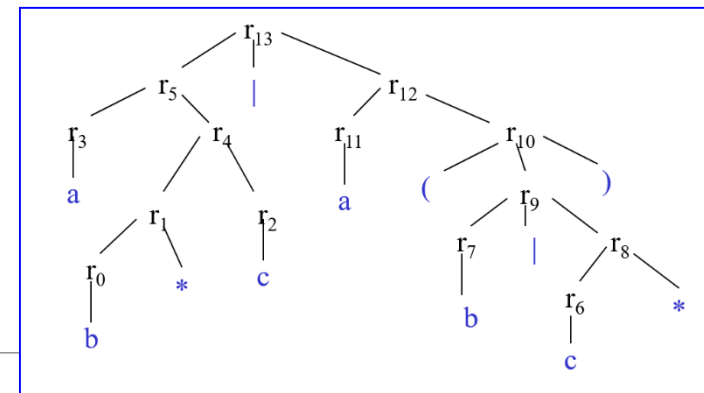
1. 每个步骤最多增加两个新状态→
 $N(r)$ 状态数 $\leq 2 \times (r \text{的符号数} + \text{操作符数})$
2. $N(r)$ 有且只有一个初态和一个终态
3. $N(r)$ 的每个状态，或者有一条标记为某个 $a \in \Sigma$ 的输出边，或者至多有两条 ϵ 输出边
4. 为状态取名要小心

例: $(ab^*c) \mid (a(b|c)^*)$

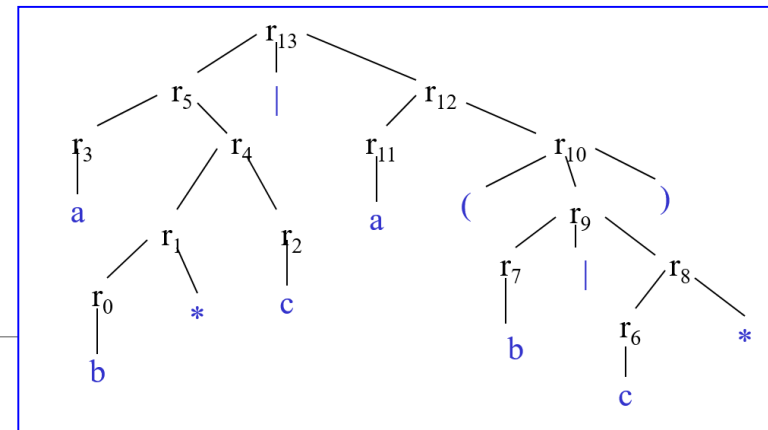
正则表达式的分解



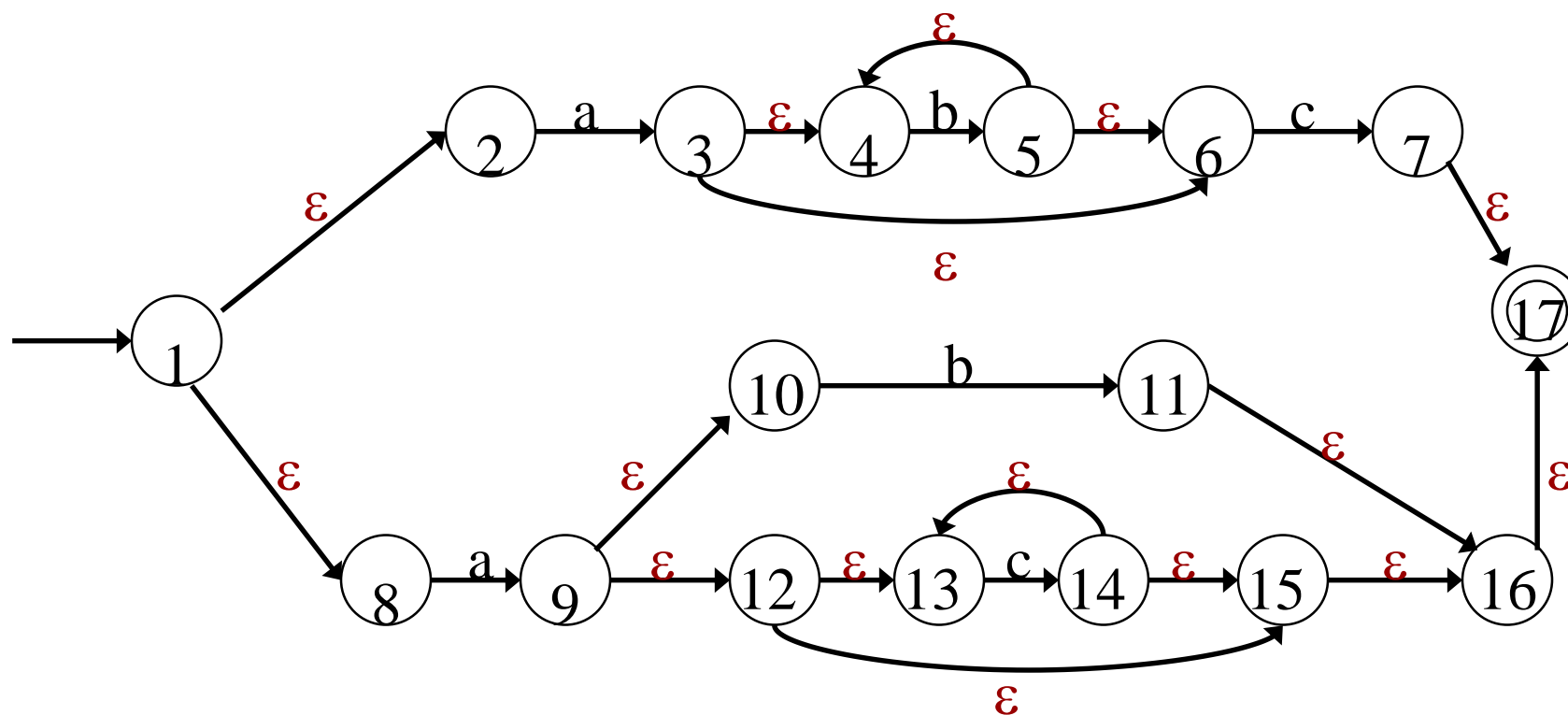
例: $(ab^*c) \mid (a(b|c)^*)$



例: $(ab^*c) \mid (a(b|c)^*)$

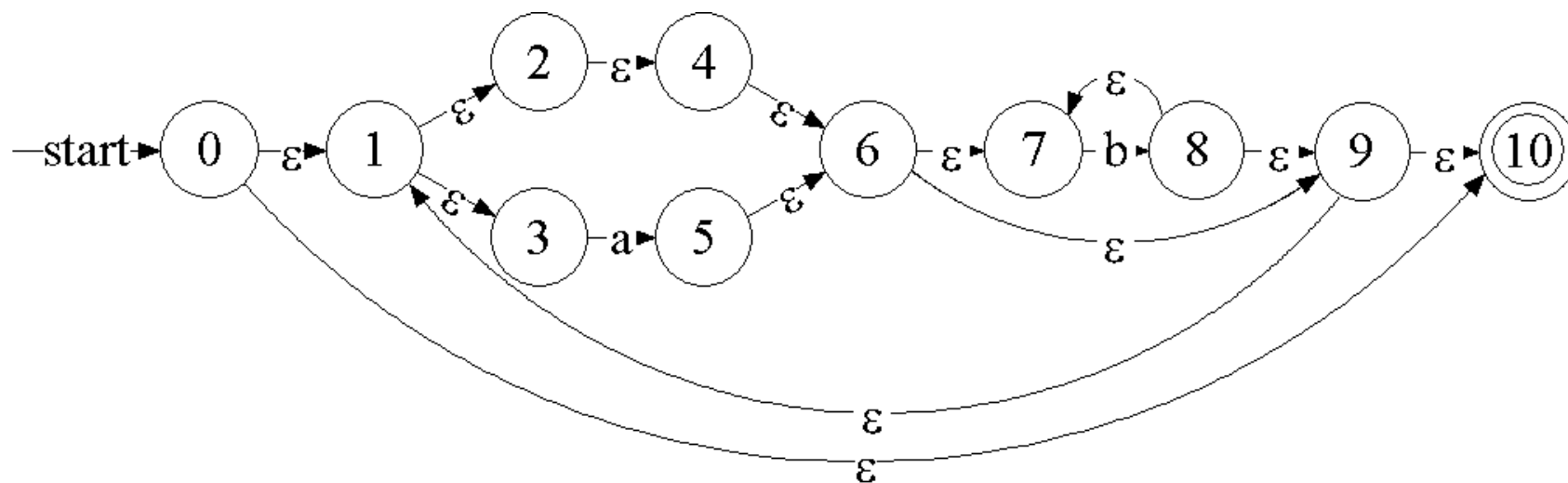


$r_{13} : r_5 \mid r_{12}$



正则表达式 \rightarrow NFA 练习

$((\epsilon \mid a)b^*)^*$



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 \rightarrow 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 , $L(D)=L(N)$

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
$\epsilon\text{-closure}(s)$ (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
$\epsilon\text{-closure}(T)$	$\cup \epsilon\text{-closure}(s), s \in T$
$d(T, a)$	$\cup d(s, a), s \in T$

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 \longleftrightarrow NFA状态集！
- DFA状态 s 与NFA状态集 T 有对应关系，那么
输入 $a_1a_2\dots a_n \rightarrow$ 到达DFA状态 s ，一定有
输入 $a_1a_2\dots a_n \rightarrow$ 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 \rightarrow DFA状态
- 不可行！

递推方法

- 最简单的符号串 ϵ ： NFA状态集合 \leftrightarrow DFA状态
- 长度为1的串 $a=\epsilon a$ ，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

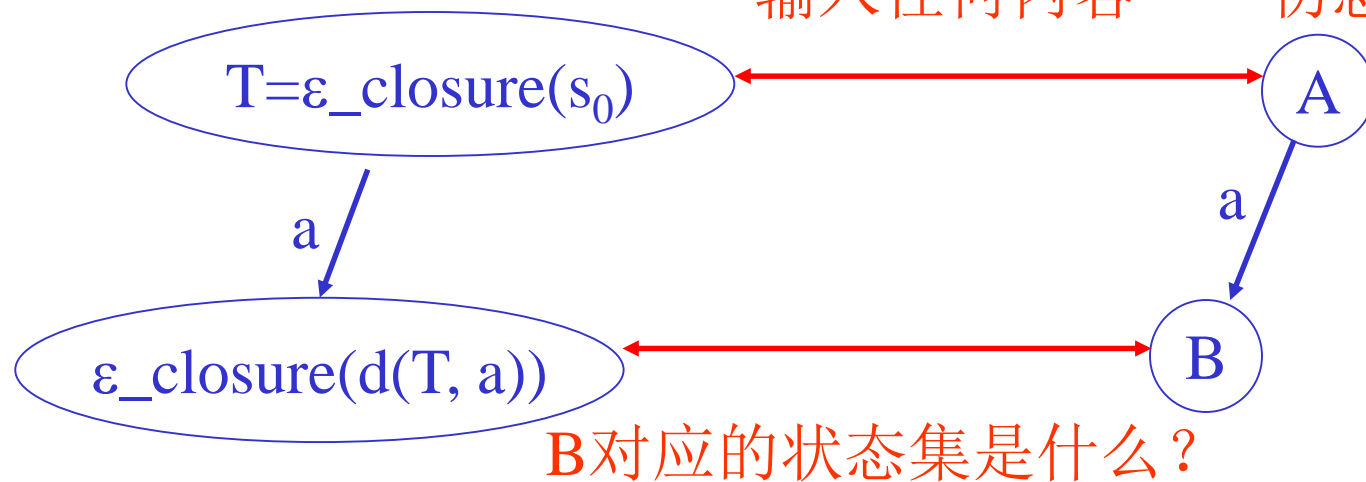
$A \leftrightarrow T$

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



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： $T \leftrightarrow A$

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 \rightarrow 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) {

$U = \epsilon$ -closure($\delta(T, a)$);

 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

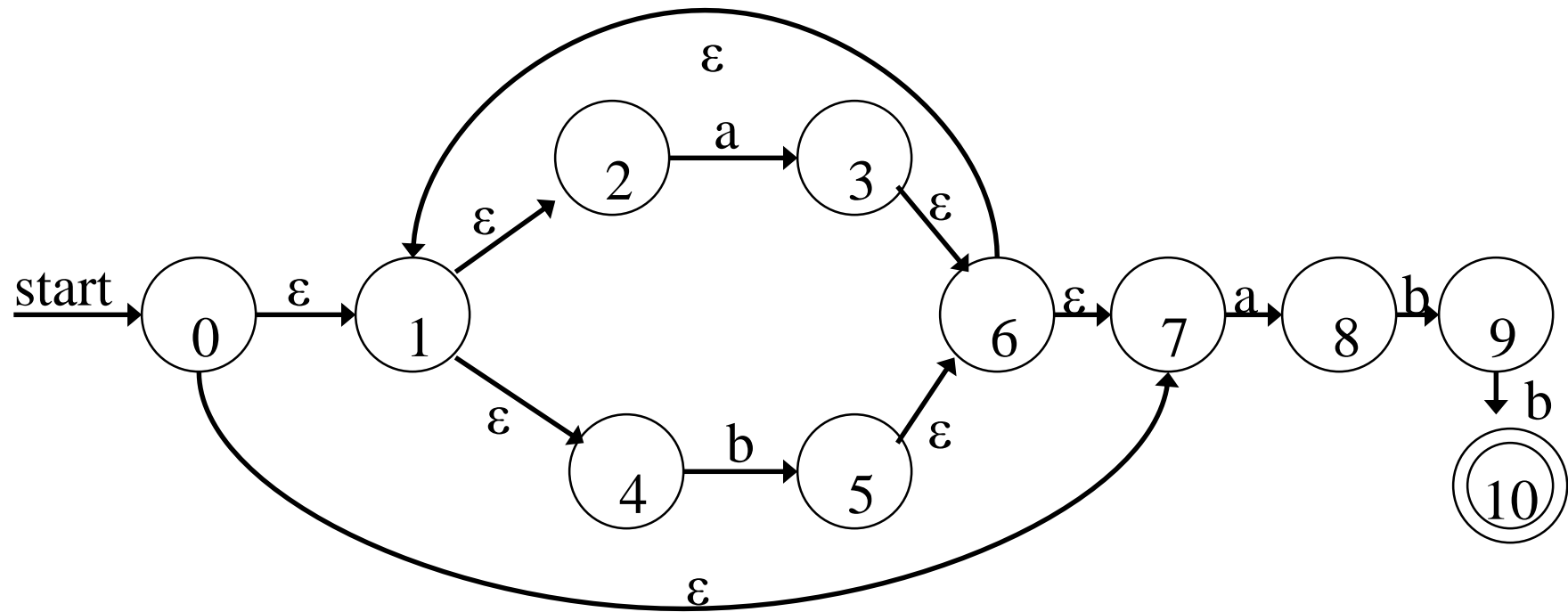
 }

}

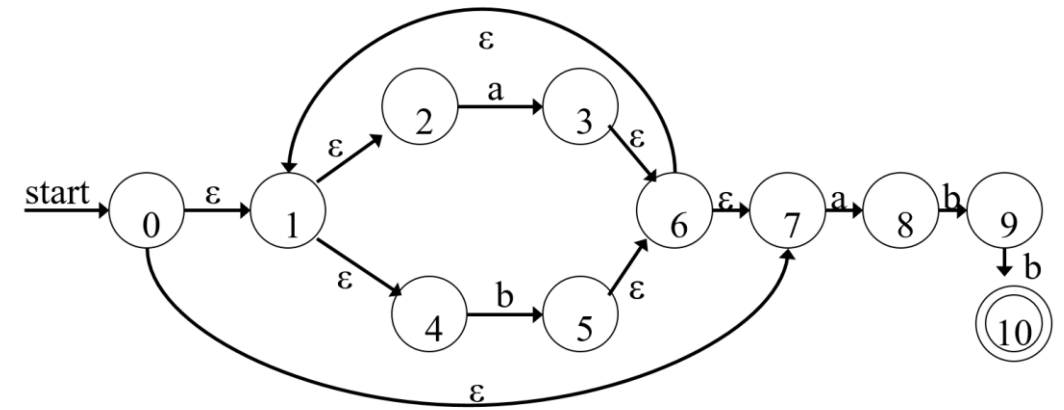
ϵ -closure(s_0)为初态，包含NFA终态的DFA状态为DFA终态

例: $(a \mid b)^*abb$

NFA \rightarrow DFA



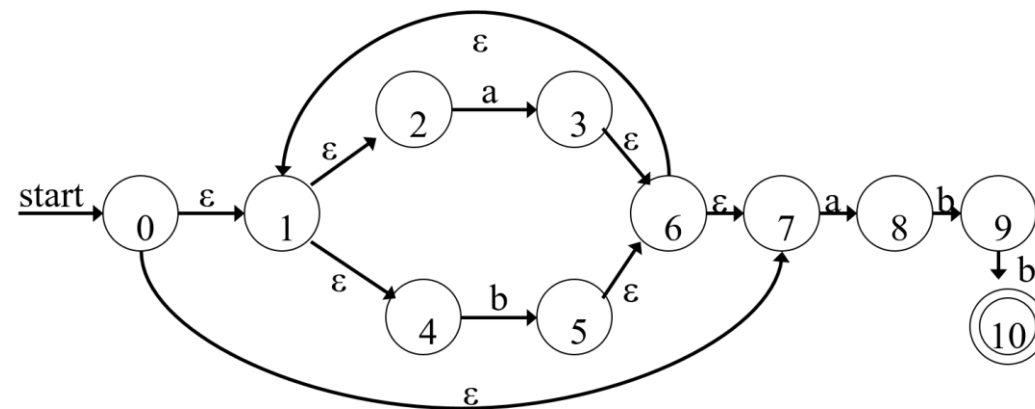
例: $(a \mid b)^*abb$



第一步

ϵ -closure(0) = {0,1,2,4,7} = **A**

例: $(a \mid b)^*abb$



第一步

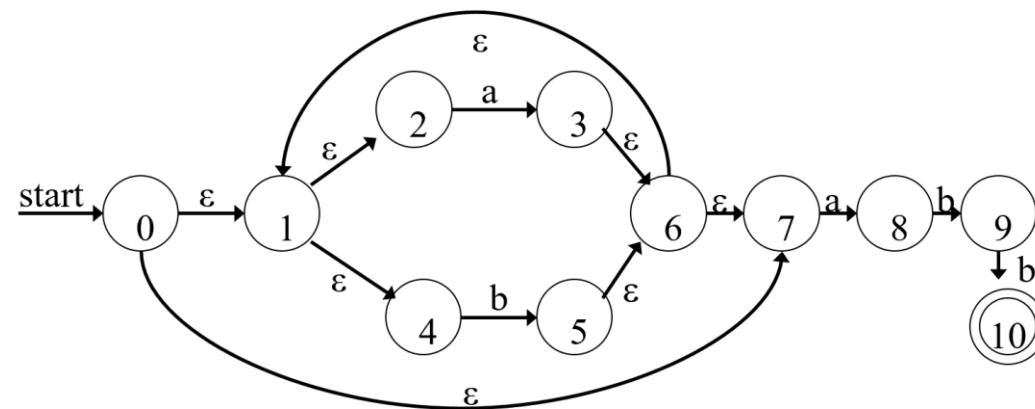
$$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\} = A$$

第二步

$$\underline{a}: \epsilon\text{-closure}(\delta(A, a)) = \epsilon\text{-closure}(\delta(\{0, 1, 2, 4, 7\}, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$D\text{tran}[A, a] = B$$

例: $(a \mid b)^*abb$



第一步

$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\} = A$

第二步

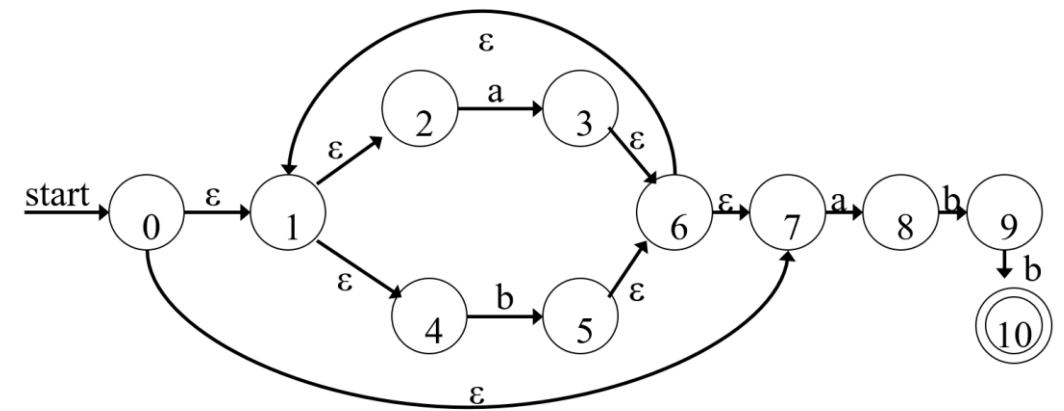
a: $\epsilon\text{-closure}(\delta(A, a)) = \epsilon\text{-closure}(\delta(\{0, 1, 2, 4, 7\}, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$

$D\text{tran}[A, a] = B$

b: $\epsilon\text{-closure}(\delta(A, b)) = \epsilon\text{-closure}(\delta(\{0, 1, 2, 4, 7\}, b)) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$

$D\text{tran}[A, b] = C$

例: $(a \mid b)^*abb$



第三步

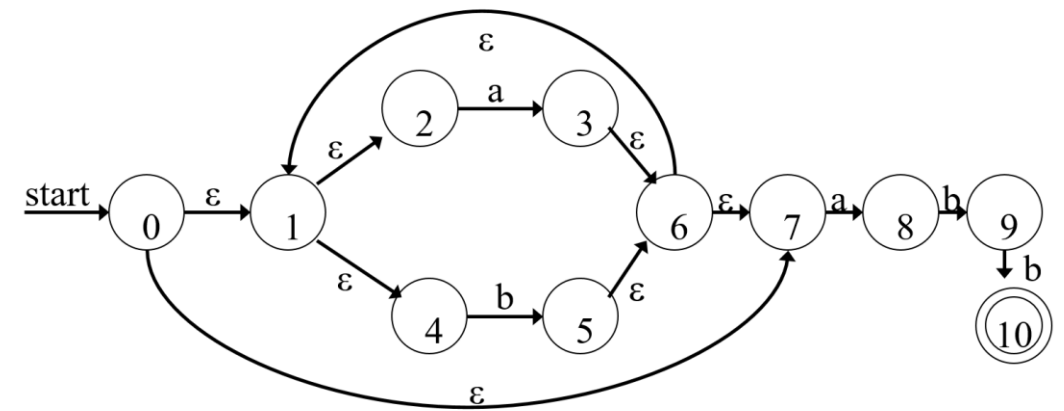
$\underline{a} : \epsilon\text{-closure}(\delta(B,a)) = \epsilon\text{-closure}(\delta(\{1,2,3,4,6,7,8\},a)) = \{1,2,3,4,6,7,8\} = B$

$D_{\text{tran}}[B,a] = B$

$\underline{b} : \epsilon\text{-closure}(\delta(B,b)) = \epsilon\text{-closure}(\delta(\{1,2,3,4,6,7,8\},b)) = \{1,2,4,5,6,7,9\} = D$

$D_{\text{tran}}[B,b] = D$

例: $(a \mid b)^*abb$



第四步

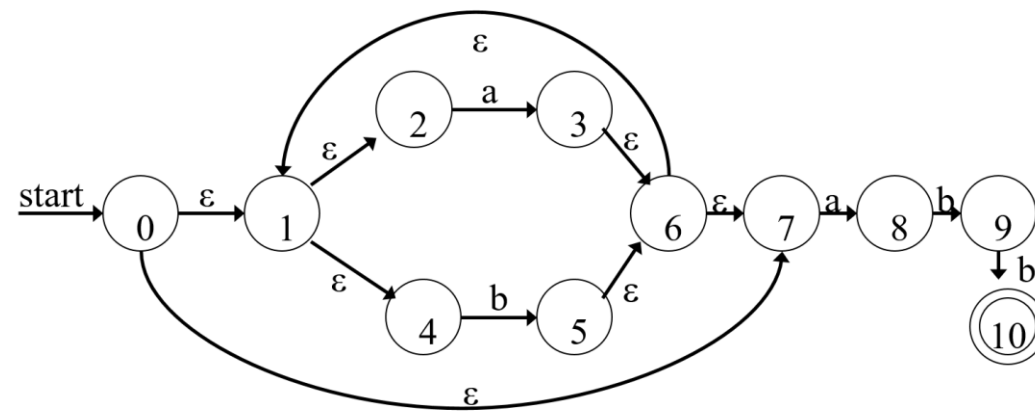
a : ϵ -closure($\delta(C, a)$) = ϵ -closure($\delta(\{1, 2, 4, 5, 6, 7\}, a)$) = $\{1, 2, 3, 4, 6, 7, 8\} = B$

$D_{\text{tran}}[C, a] = B$

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

$D_{\text{tran}}[C, b] = C$

例: $(a \mid b)^*abb$



第五步

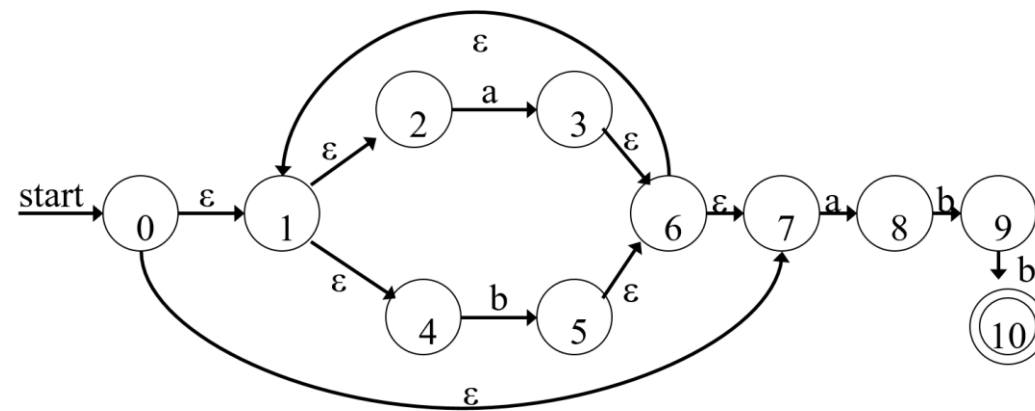
$\underline{a} : \varepsilon\text{-closure}(\delta(D,a)) = \varepsilon\text{-closure}(\delta(\{1,2,4,5,6,7,9\},a)) = \{1,2,3,4,6,7,8\} = B$

$D\text{tran}[D,a] = B$

$\underline{b} : \varepsilon\text{-closure}(\delta(D,b)) = \varepsilon\text{-closure}(\delta(\{1,2,4,5,6,7,9\},b)) = \varepsilon\text{-closure}(\{5,10\}) = \{1,2,4,5,6,7,10\} = E$

$D\text{tran}[D,b] = E$

例: $(a \mid b)^*abb$



第六步

$\underline{a} : \varepsilon\text{-closure}(\delta(E,a)) = \varepsilon\text{-closure}(\delta(\{1,2,4,5,6,7,10\},a)) = \{1,2,3,4,6,7,8\} = B$

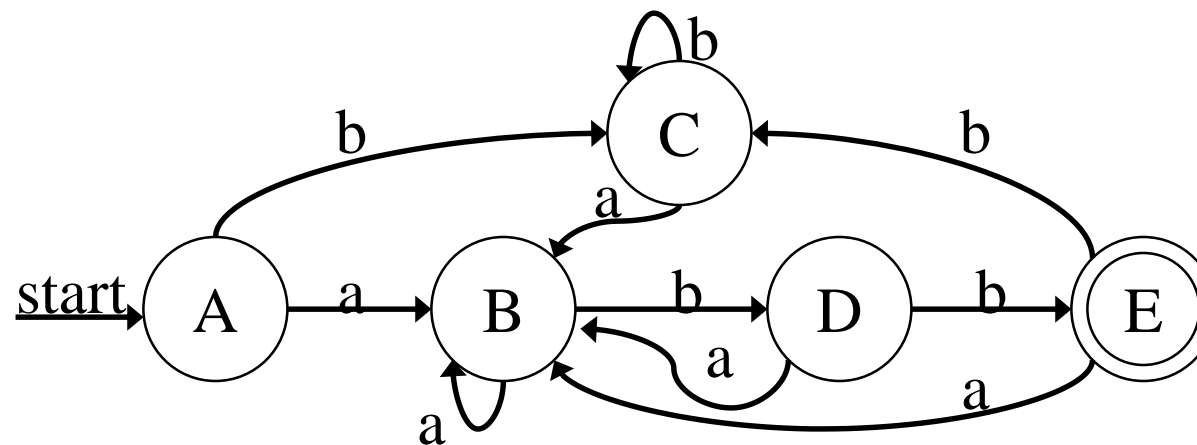
$D_{\text{tran}}[E,a] = B$

$\underline{b} : \varepsilon\text{-closure}(\delta(E,b)) = \varepsilon\text{-closure}(\delta(\{1,2,4,5,6,7,10\},b)) = \{1,2,4,5,6,7\} = C$

$D_{\text{tran}}[E,b] = C$

例: $(a \mid b)^*abb$

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 ←  $\epsilon$ -closure( $\{s_0\}$ )
c ← nextchar;
while c  $\neq$  eof do
    S ←  $\epsilon$ -closure(d(s,c));
    c ← nextchar;
end;
if  $S \cap F \neq \emptyset$  then return “yes”
else return “no”
```

```
s ←  $s_0$ 
c ← nextchar;
while c  $\neq$  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”

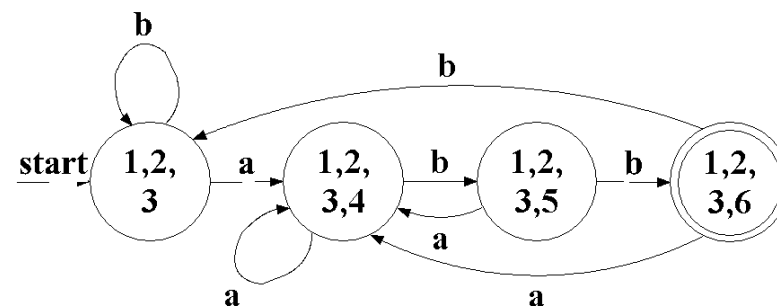
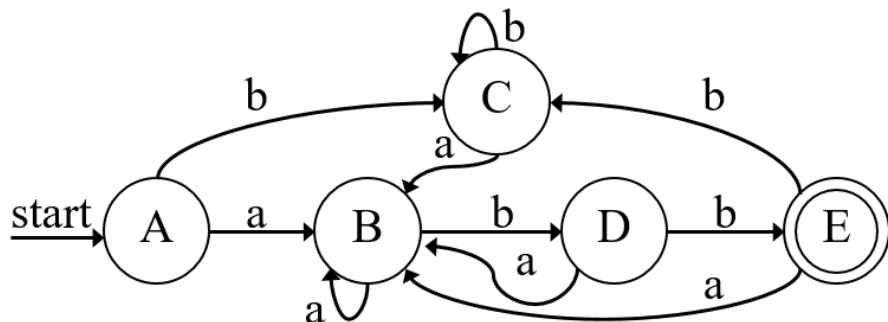
- 使用DFA进行词法分析
- 但不预先构造完整的DFA，而是翻译（词法分析）过程中用到哪些状态和转换关系才即时计算，并利用缓存机制

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

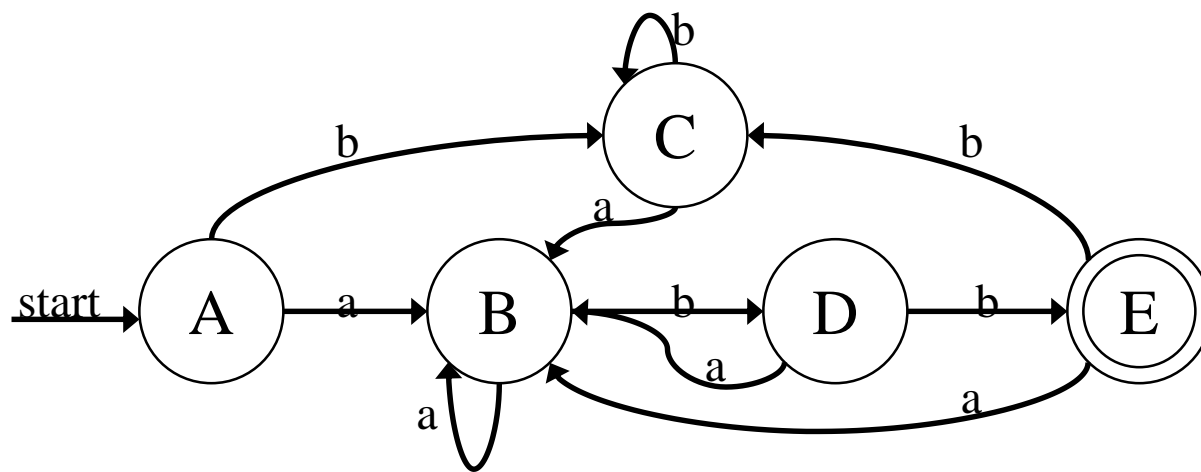
最小化DFA的状态数

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 $A = \{s_1, s_2, \dots, s_k\}$

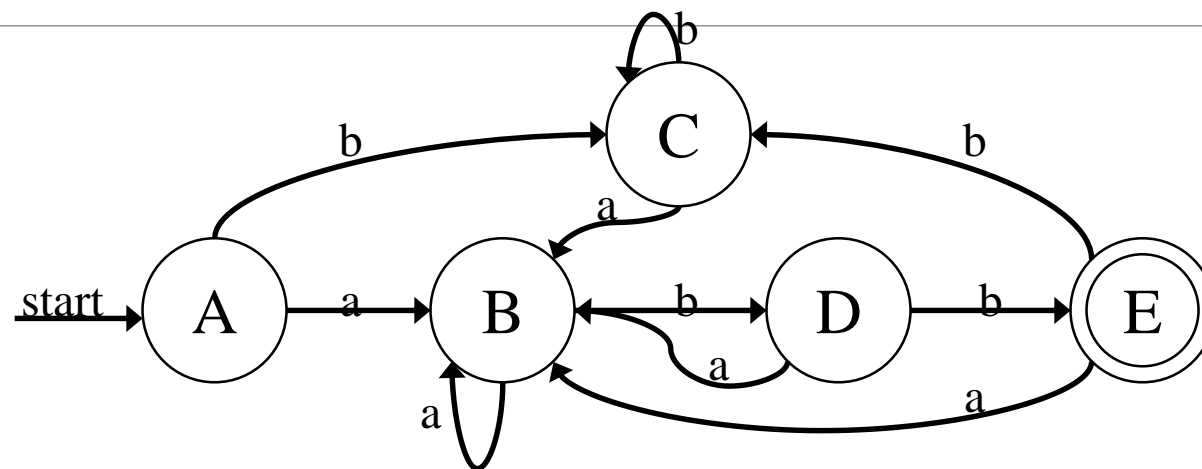
- 对符号 a ，得到其转换状态 t_1, t_2, \dots, t_k
- 若 t_1, t_2, \dots, t_k 属于不同状态组，则需将 A 对应划分为若干组

算法：最小化DFA

划分算法

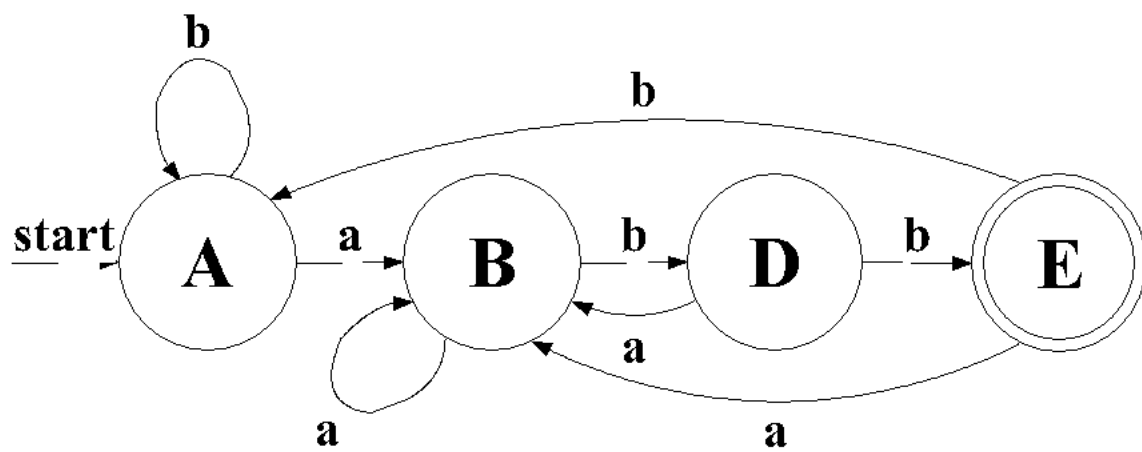
```
for ( $\Pi$ 中每个状态组 $G$ ) {  
    将 $G$ 划分为若干分组，使得  
     $G$ 中两个状态 $s$ 、 $t$ 在同一分组中的充要条件为  
    对所有符号 $a$ ， $s$ 、 $t$ 转换后的状态处于 $\Pi$ 中的同一个状态组中；  
    在 $\Pi_{\text{new}}$ 中，用所有分组替换 $G$   
}
```

例:



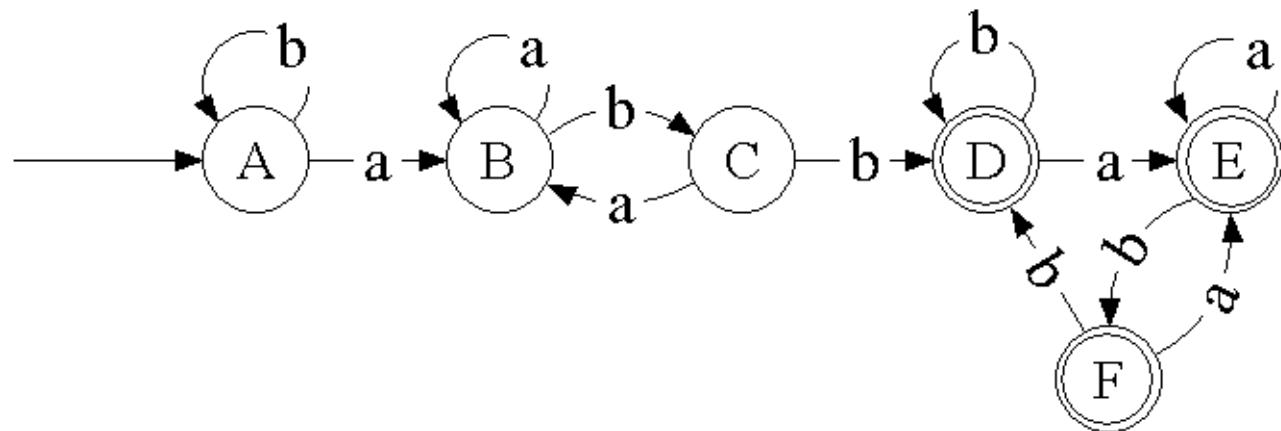
1. $\{E\}, \{A, B, C, D\}$
2. $\{A, B, C, D\} \xrightarrow{b} \{C, D, C, E\} \rightarrow \{E\}, \{D\}, \{A, B, C\}$
3. $\{A, B, C\} \xrightarrow{b} \{C, D, C\} \rightarrow \{E\}, \{D\}, \{B\}, \{A, C\}$
4. $\Pi_{\text{final}}: \{A, C\}, \{B\}, \{D\}, \{E\}$

例：（续）



状态	符号	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

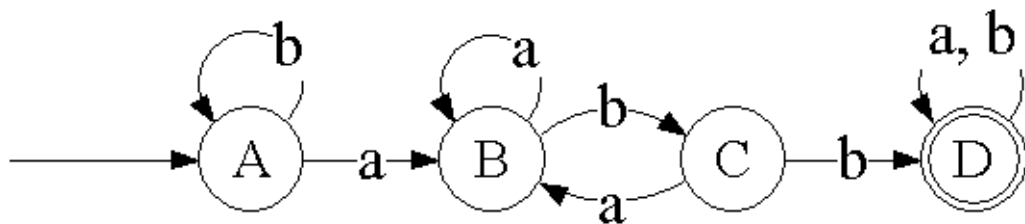
最小化DFA练习



初始划分: $\{A, B, C\}, \{D, E, F\}$

$\{A, B, C\} \xrightarrow{b} \{A, B\}, \{C\}$

$\{A, B\} \xrightarrow{b} \{A\}, \{B\}$



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