



第三章 词法分析

- 重点：**词法分析器的输入、输出，
用于识别符号的状态转移图的构造，
根据状态转移图实现词法分析器。
- 难点：**词法的正规文法表示、正规表达式表示、
状态转移图表示，它们之间的转换。



第3章 词法分析

3.1 词法分析器的功能

3.2 单词的描述

3.3 单词的识别

3.4 词法分析程序的自动生成

3.5 本章小结



3.1 词法分析器的功能

- **功能：输入源程序，输出单词符号。即：把构成源程序的字符串转换成“等价的”单词序列**
 - **根据词法规则识别及组合单词，进行词法检查**
 - **对数字常数完成数字字符串到二进制数值的转换**
 - **删去空格和注释等不影响程序语义的字符**

3.1.1 单词的八类与丰二

& 3.1.2 词法分

关键字、运算符和分界符都是程序设计语言预先定义的，数量固定，标识符和常数的数量不定

一、单词的种类

1. **关键字:**也称基本字，多用来作为语句的标识，如begin、end、for、do...
2. **标识符:**由用户定义，表示各种名字，如变量名等
3. **常数:**整常数、实常数、布尔常数、字符串常数等
4. **运算符:**算术运算符+、-、*、/等；逻辑运算符not、or与and等；关系运算符=、<>、>=、<=、>和<等
5. **分界符:** , 、 ; 、 (、) ...

二、单词的内部形式

表示单词的种类，可用整数编码或记忆符表示

不同的单词不同的值

二元组

| | |
|----|-----|
| 种别 | 属性值 |
|----|-----|

两种常用的单词内部表示形式：

- 1、按单词种类分类
- 2、固定数量单词采用一符一类



1、按单词种类分类

| 单词名称 | 类别编码 | 单词值 |
|--------|------|----------|
| 标识符 | 1 | 内部字符串 |
| 无符号整数 | 2 | 整数值 |
| 无符号浮点数 | 3 | 数值 |
| 布尔常数 | 4 | 0 或 1 |
| 字符串常数 | 5 | 内部字符串 |
| 关键字 | 6 | 关键字或内部编码 |
| 分界符 | 7 | 分界符或内部编码 |
| 运算符 | 8 | 运算符或内部编码 |

2、固定数量单词采用一符一类

| | 单词名称 | 类别编码 | 单词值 |
|----------------------------------|----------|-------|-------|
| 标识符 和变量 仍然采用 单词种类 分类 | 标识符 | 1 | 内部字符串 |
| | 无符号常数(整) | 2 | 整数值 |
| | 无符号浮点数 | 3 | 数值 |
| | 布尔常数 | 4 | 0 或 1 |
| | 字符串常数 | 5 | 内部字符串 |
| 关键字 运算符 和分界符 一符一类 | BEGIN | 6 | - |
| | END | 7 | - |
| | FOR | 8 | |
| | DO | 9 | |
| | | | |
| | : | 20 | |
| | + | 21 | |
| | * | 22 | - |
| | , | 23 | - |
| | (| | - |
| | | | |

可采用宏定义形式
给出单词的种别码,
如教材P66表3.1

采用一
符一类
的单词,
其二元
组单词
值为空

二、单词的内部形式

■ 对于标识符和常量，按单词种类分类

- 属于同一类的不同单词，具有相同的种别码，通过不同的属性值来区分

■ 问题：如何存储标识符和常量的属性值

- 方法1：用标识符和常量本身的价值表示
- 方法2：用指针表示

考虑各自的
优缺点

例3.1 语句if count>7 then result := 3.14; 的单词符号序列

(IF, 0)

(ID, 指向count 的符号表入口)

(GT, 0)

(INT, 7)

(THEN, 0)

(ID, 指向result的符号表入口)

(ASSIGN, 0)

(REAL, 3.14)

(SEMIC, 0)

IF: if的宏

ID: 标识符的宏

INT: 整数的宏

GT: >的宏

THEN: then的宏

ASSIGN: := 的宏

REAL: 实数的宏

SEMIC: ;的宏



3.1.3 源程序的输入缓冲与预处理

- 源程序以**字符流形式**存储于外部介质
- 为正确识别单词，编译程序需要一系列相关处理



3.1.3 源程序的输入缓冲与预处理

■ 超前搜索和回退

- 标识符的识别，或双字符运算符 ($**$, \leq , $\lt \gt$)
- 回退操作修正超前搜索

■ 缓冲区

- 假定源程序存储在磁盘上，这样每读一个字符就需要访问一次磁盘，效率显然是很低的。
- 一次性从磁盘读取给定大小的部分源程序

■ 空白字符的剔除

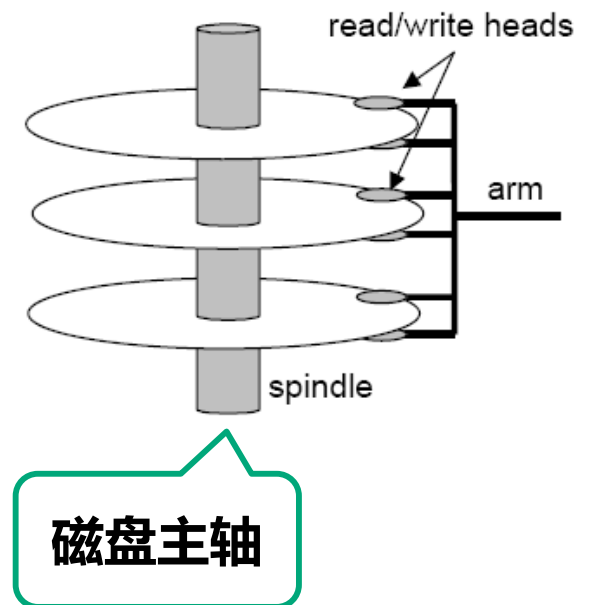
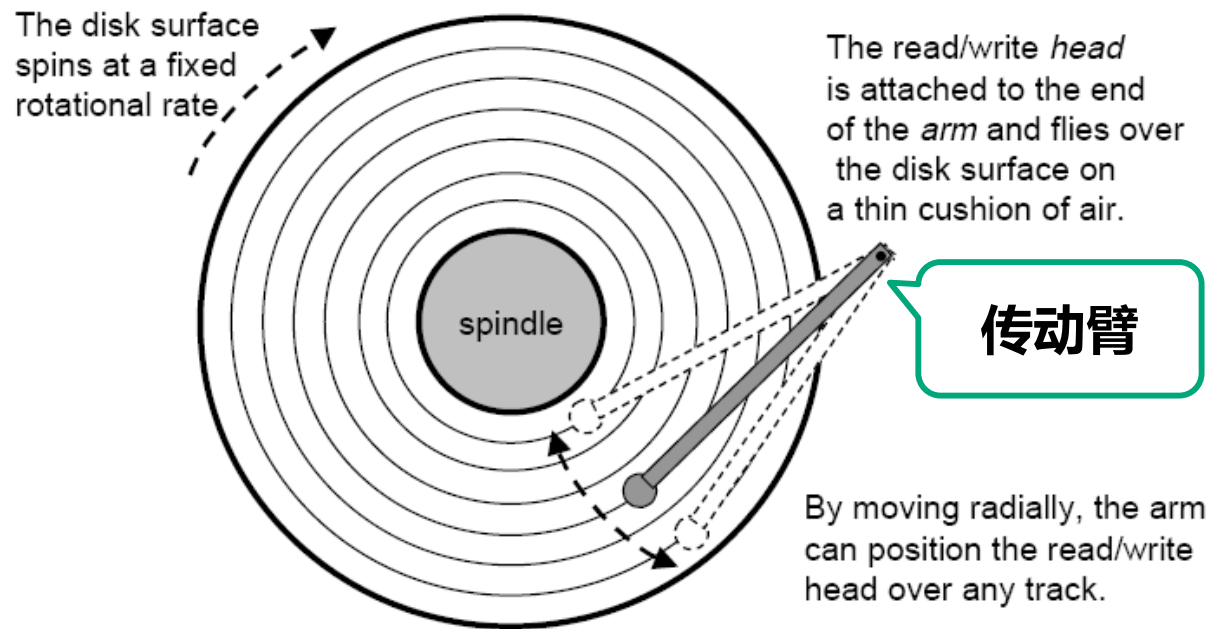
- 剔除源程序中的无用符号、空格、换行、注释等



磁盘结构

- **磁盘组合 (disk assembly)**: 由一个或多个圆形盘片组成, 围绕一根中心主轴旋转。上下表面覆盖了一层薄薄的磁性材料, 用于存储二进制信息。
- **磁头组合(head assembly)**: 承载着磁头, 每个盘面有一个磁头
- 盘片的旋转速度通常是5400-15000转每分钟
- 每英寸大约100,000个磁道

磁盘存取特性



磁盘存取特性

- 一个例子：考虑有如下参数的磁盘

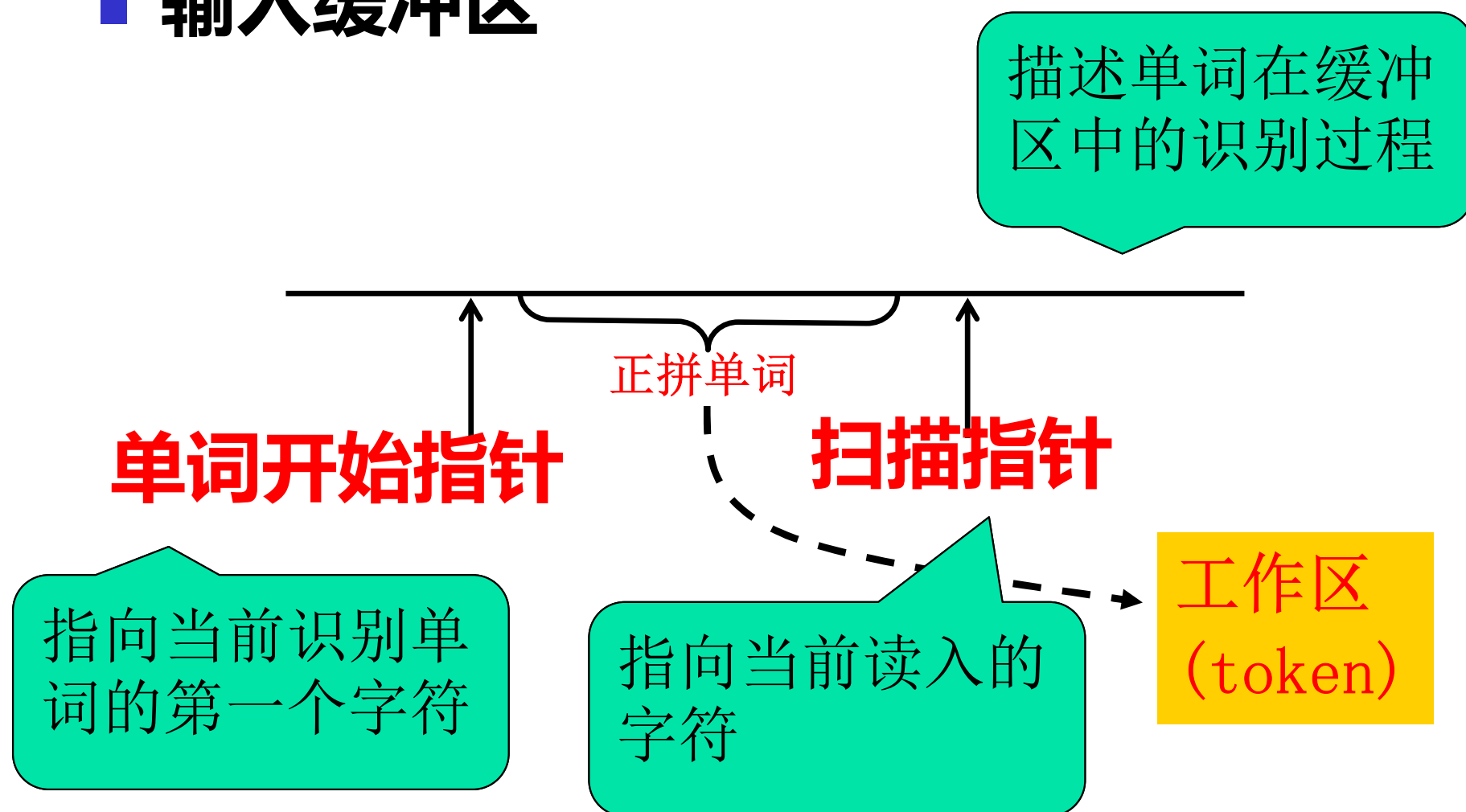
| Parameter | Value |
|-------------------------|-----------|
| Rotational rate | 7,200 RPM |
| $T_{avg\ seek}$ | 9 ms |
| Average # sectors/track | 400 |

- 估计的访问时间

$$\begin{aligned}T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\&= 9\ ms + 4\ ms + 0.02\ ms \\&= 13.02\ ms.\end{aligned}$$

3.1.3 源程序的输入缓冲与预处理(续)

■ 输入缓冲区



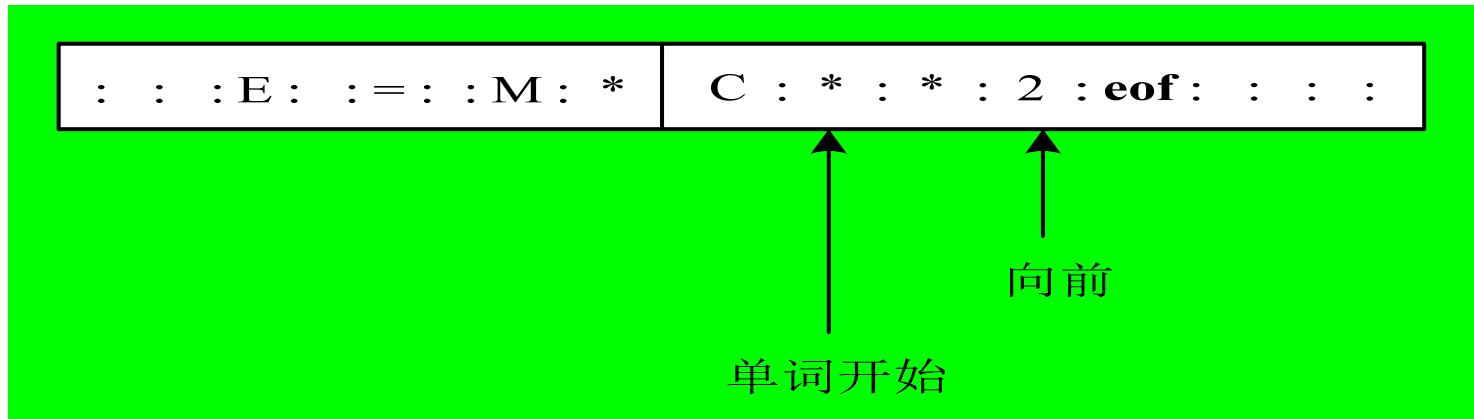


3.1.3 源程序的输入缓冲与预处理(续)

- **如果采用单个缓冲区，存在以下几个问题**
 - **缓冲区内容用完后，需要等待新的输入，应该避免类似的等待**
 - **缓冲区尾部可能只包含单词的一部分，载入下一部分程序时，当前缓冲区的内容被覆盖，最坏情况下可识别的单词长度只能为1，而且无法执行超前搜索**

3.1.3 源程序的输入缓冲与预处理(续)

双缓冲区



if *forward*在缓冲区第一部分末尾 then
重装缓冲区第二部分;
forward := *forward* + 1

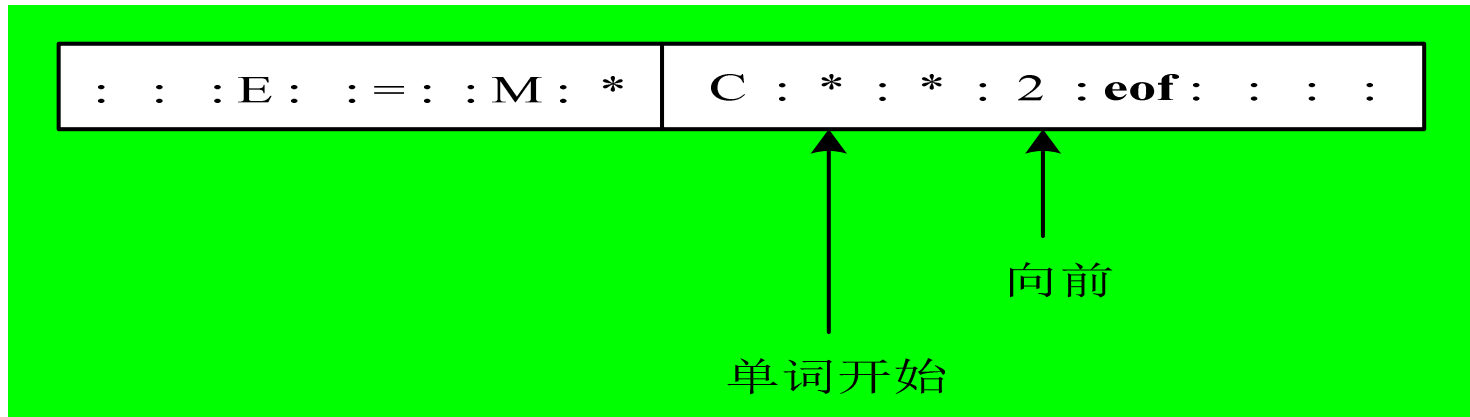
if *forward*在缓冲区第二部分末尾 then
重装缓冲区第一部分;
将*forward*移到缓冲区第一部分开始

其他情况且当前字符不是EOF *forward* := *forward* + 1;

令缓冲区大小为 $2N$ ，则
双缓冲区的每一个大小
是 N ，双缓冲技术将可识
别单词的长度扩展到 N

3.1.3 源程序的输入缓冲与预处理(续)

双缓冲区



if *forward*在缓冲区第一部分末尾 then
重装缓冲区第二部分;
forward := *forward* + 1

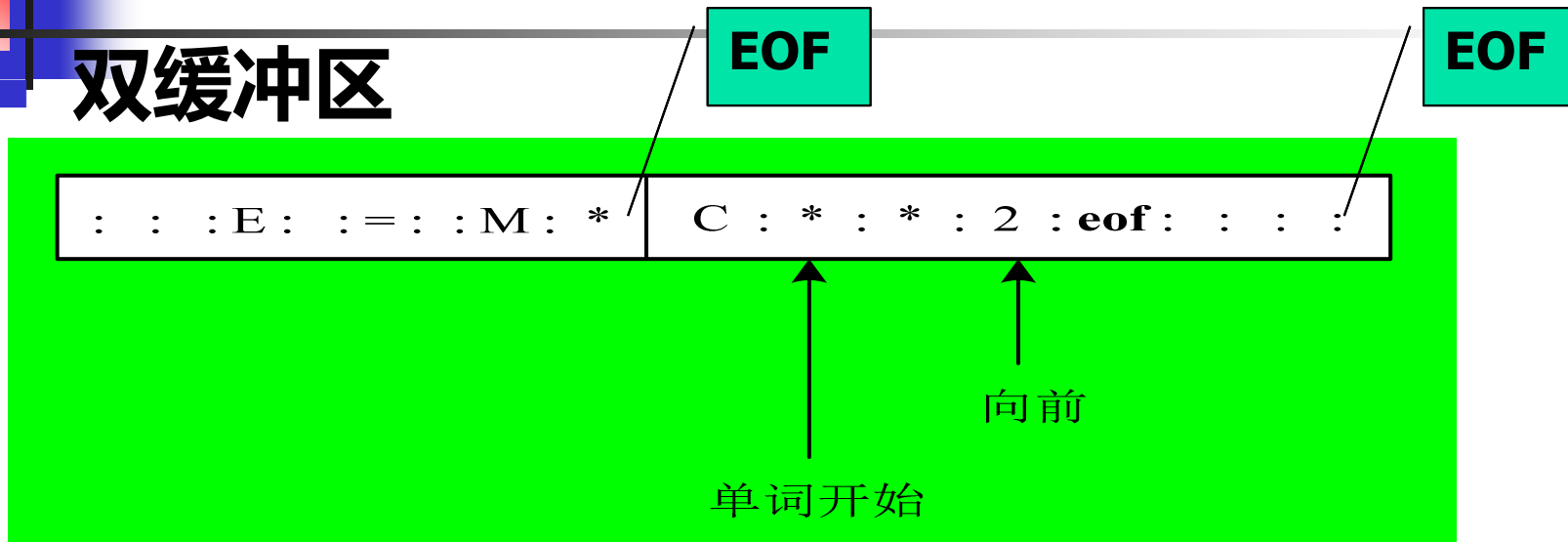
if *forward*在缓冲区第二部分末尾 then
重装缓冲区第一部分;
将*forward*移到缓冲区第一部分开始

其他情况且当前字符不是EOF *forward* := *forward* + 1;

每次移动向前指针都需要做两次测试:
1) 是否到缓冲区末尾
2) 当前字符是否是EOF

3.1.3 源程序的输入缓冲与预处理(续)

双缓冲区



if *forward*在缓冲区第一部分末尾 then
重装缓冲区第二部分;
forward := *forward* + 1

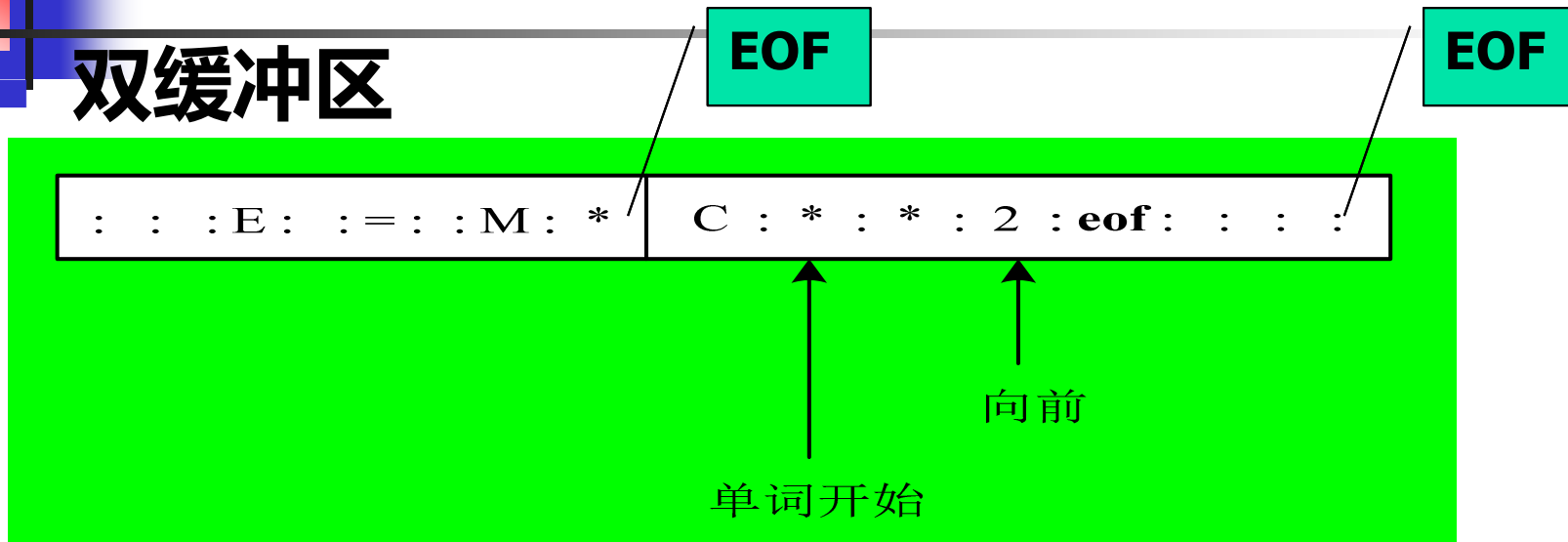
if *forward*在缓冲区第二部分末尾 then
重装缓冲区第一部分;
将*forward*移到缓冲区第一部分开始

其他情况且当前字符不是EOF *forward* := *forward* + 1;

每次移动向前指针都需要做两次测试
修正方法：采用带标记缓冲区，即两个缓冲区的末尾处各设置一个“EOF”标志

3.1.3 源程序的输入缓冲与预处理(续)

双缓冲区



if *forward*在缓冲区第一部分末尾 then
重装缓冲区第二部分;
forward := *forward* + 1

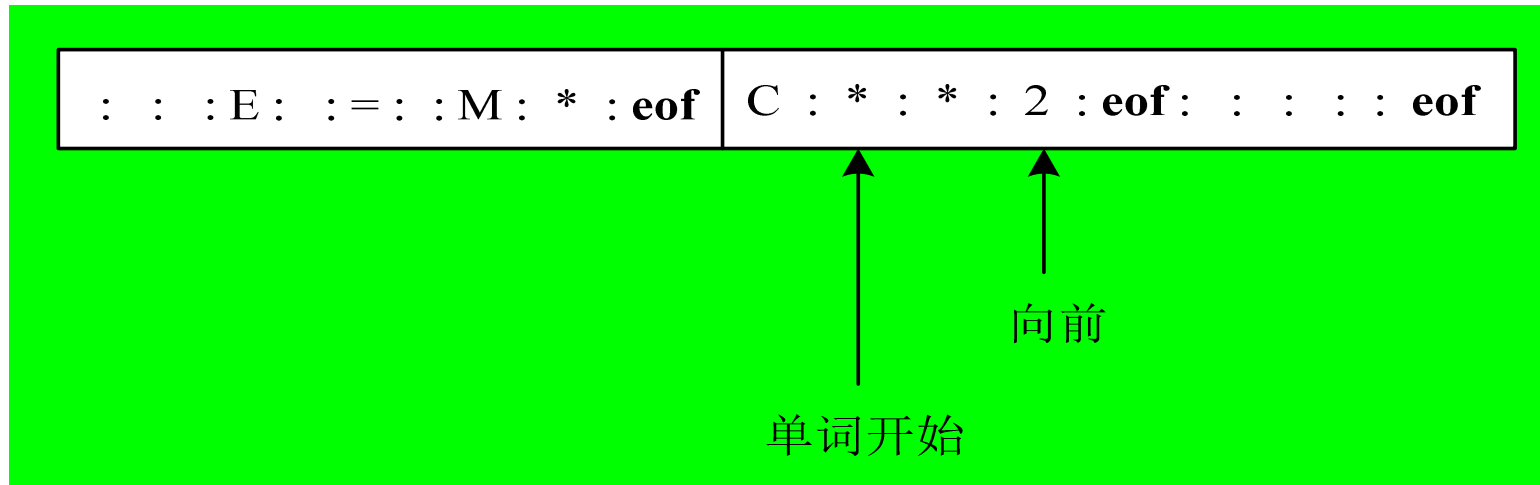
if *forward*在缓冲区第二部分末尾 then
重装缓冲区第一部分;
将*forward*移到缓冲区第一部分开始

其他情况且当前字符不是EOF *forward* := *forward* + 1;

采用带标记缓冲区，如果当前字符是“EOF”，就再判断是否到达缓冲区末尾，将移动向前指针需要的两次测试减少到 $(N+1)/N$

3.1.3 源程序的输入缓冲与预处理(续)

双缓冲区



大小问题 $128\text{Byte} \times 2$ | $1024\text{Byte} \times 2$ | $4096\text{Byte} \times 2$



3.1.4 词法分析阶段的错误处理

- 1. 非法字符**
- 2. 单词拼写错误**
- 3. 注释或字符串常量不封闭**
- 4. 变量重复说明**



3.1.4 词法分析阶段的错误处理

1. 非法字符检查

- 维护一个合法字符集合，对于每一个输入字符，判断该字符是否属于该字符集合



3.1.4 词法分析阶段的错误处理

2. 单词拼写错误

- 关键字拼写词法分析阶段无法检测，待语法分析阶段发现错误
- 标识符拼写错误，如3b78，处理方法有两种
 - 识别出整数3、标识符b78
 - 错误的标识符



3.1.4 词法分析阶段的错误处理

3. 不封闭错误检查

- 影响正常程序分析
- 对注释或字符串长度加以限制，如注释长度不超过1行，字符串长度最大是256



3.1.4 词法分析阶段的错误处理

4. 重复声明检查

- 兼顾符号表的查填工作

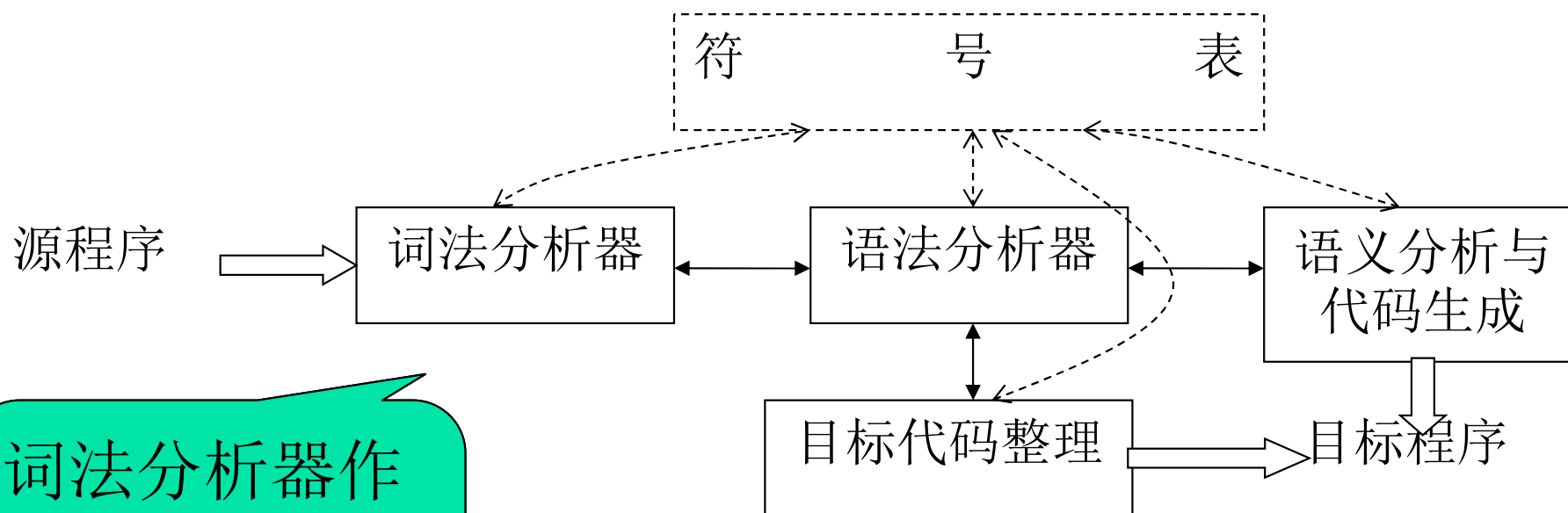


3.1.4 词法分析阶段的错误处理

5. 错误恢复与续编译

- 词法分析阶段的错误使得编译无法继续进行，需要采取措施使得编译继续下去
- 方法
 - 错误校正：极其困难
 - 紧急方式恢复(panic-mode recovery)：反复删掉剩余输入最前面的字符，直到词法分析器能发现一个正确的单词为止。

3.1.5 词法分析器的位置(1)



词法分析器作为语法分析器的独立子程序

图3.4 以语法分析器为中心



3.1.5 词法分析器的位置(2)

- **将词法分析作为一个单独阶段，将单词序列以中间文件形式存储，作为语法分析的输入**
- **优点：**
 - **简化编译器的设计。**
 - **提高编译器的效率。**
 - **增强编译器的可移植性。**



3.2 单词的描述

- **单词**是程序设计语言的**基本语法单位**
- 如果每类单词都看作一种语言，则大多数单词词法可以用**正则文法**来描述



3.2.1 正则文法

- 正则文法 $G = (V, T, P, S)$ 中, 对 $\forall \alpha \rightarrow \beta \in P$, $\alpha \rightarrow \beta$ 均具有形式 $A \rightarrow w$ 或 $A \rightarrow wB$ ($A \rightarrow w$ 或 $A \rightarrow Bw$), 其中 $A, B \in V, w \in T^+$ 。
- 正则文法描述 T 上的正则语言



3.2.1 正则文法

■ 例3.2 标识符的文法

- sum, result, a1, b2
- $\langle \text{id} \rangle \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$
- $\langle \text{id} \rangle \rightarrow \langle \text{id} \rangle A \mid \langle \text{id} \rangle B \mid \dots \mid \langle \text{id} \rangle Z$
- $\langle \text{id} \rangle \rightarrow \langle \text{id} \rangle a \mid \langle \text{id} \rangle b \mid \dots \mid \langle \text{id} \rangle z$
- $\langle \text{id} \rangle \rightarrow \langle \text{id} \rangle 0 \mid \langle \text{id} \rangle 1 \mid \dots \mid \langle \text{id} \rangle 9$



3.2.1 正则文法

■ 例3.2 标识符的文法

- 约定：用<digit>表示数字：0,1,2,...,9;
用<letter>表示字母：A,B,...,Z,a,b,...,z
- $\langle id \rangle \rightarrow \langle letter \rangle \mid \langle id \rangle \langle digit \rangle \mid \langle id \rangle \langle letter \rangle$
- $\langle letter \rangle \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$
- $\langle digit \rangle \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$



3.2.2 正则表达式

- **除正则语法外，正则表达式也可以描述单词**
- **正则语法和正则表达式的能力相同，可以互相转化**
- **正则表达式比正则语法更直观，有时首选正则表达式来表示正则语言**

3.2.2 正则表达式

- 例3.2：标识符的另一种表示
 - $\text{letter}(\text{letter}|\text{digit})^*$
 - | 表示“或”
 - * 表示Kleene闭包
 - + 表示正闭包
 - ? 表示“0或1个”
 - letter 和 $(\text{letter}|\text{digit})^*$ 的并列表示两者的连接
- 正则表达式 r 的相应的正则语言记为 $L(r)$
- 一个正则表达式通常称为一个**模式**，用来描述一系列符合某个**句法规则**的字符串



3.2.2 正则表达式一定义

定义3.1 设 Σ 是一个字母表，则 Σ 上的正则表达式及其所表示的正则语言可递归地定义如下：

- (1) \emptyset 是 Σ 上的一个正则表达式，它表示空集；**
- (2) ε 是 Σ 上的一个正则表达式，它表示语言 $\{\varepsilon\}$ ；**
- (3) 对于 $\forall a(a \in \Sigma)$ ， a 是 Σ 上的一个正则表达式，它表示的正则语言 $L(a)$ 是 $\{a\}$ ；**



3.2.2 正则表达式—定义

- (4) 假设 r 和 s 都是 Σ 上的正则表达式，它们表示的语言分别为 $L(r)$ 和 $L(s)$ ，则：
- ① (r) 是 Σ 上的正则表达式，它表示的语言为 $L(r)$ ；
 - ② $(r|s)$ 是 Σ 上的正则表达式，它表示的语言为 $L(r) \cup L(s)$ ；（并操作）
 - ③ $(r \bullet s)$ 是 Σ 上的正则表达式，它表示的语言为 $L(r)L(s)$ ；（连接操作）
 - ④ (r^*) 是 Σ 上的正则表达式，它表示的语言为 $(L(r))^*$ ；（克林闭包操作）
- (5) 使用上述规则构造的表达式是 Σ 上的正则表达式。



3.2.2 正则表达式—定义

定义3.2 如果正则表达式 r 与 s **表示的语言相同**，即
 $L(r)=L(s)$ ，则称 r 与 s **等价**，也称 r 与 s 相等，记作 $r=s$ ；



正则表达式中的运算优先级

运算优先级和结合性:

- $*$ > “连接” > $|$
- $|$ 具有交换律、结合律
- “连接” 具有结合律、和对 $|$ 的分配律
- $()$ 指定优先关系
- 意义清楚时, 括号可以省略
- $((a) | ((b)^*(c)))$ 等价于 $a|b^*c$



正则表达式中的运算优先级

例：

1. $L(a|b) = \{a, b\}$

2. $L((a|b)(a|b)) = \{aa, ab, ba, bb\}$

3. $L((a|b)^*) = \{x | x \text{ 是 } a \text{ 和 } b \text{ 构成的符号串, 包括 } \varepsilon\}$

4. $L(a|a^*b) = \{a, b, ab, aab, aaab, aaaab, \dots\}$



3.2.3 正则表达式与正则文法的等价性

■ 正则表达式与正则文法等价

- 对任意一个正则文法，存在一个定义同一语言的正则表达式
- 对任意一个正则表达式，存在一个定义同一语言的正则文法
- 这部分介绍正则表达式和正则文法之间的转换方法



1. 根据正则文法构造等价的正则表达式

- **问题：给定正则文法 G ，构造一个正则表达式 r ，使得 $L(r) = L(G)$**
- **基本思路**
 - 为正则文法的每个**产生式**构造一个**正则表达式方程式**，从而得到一个联立方程组。
 - 这些**方程式中的变量**是文法 G 中的**语法变量**，各变量的系数是正则表达式。
 - 用**代入消元法**消去联立方程组中**除开始符号外的其他变量**，最后得到关于开始符号 S 的解： $S = r$ ， r 即为所求的正则表达式。

1. 根据正则文法构造等价的正则表达式

■ 具体步骤

(1) 根据正则文法 G 构造正则表达式联立方程组。

假设正则文法 G 是右线性的，其每个产生式的右部只含有一个终结符，则有如下方程式构造规则：

① 对形如 $A \rightarrow a_1 | a_2 | \dots | a_m$ 的产生式，构造方程式 $A = a_1 | a_2 | \dots | a_m$ ；
对形如 $A \rightarrow \varepsilon$ 的产生式；

注意克林闭包符号，注意 $A=A$

② 对形如 $A \rightarrow a_1 A | a_2 A | \dots | a_m A$ 的产生式，构造方程式 $A = (a_1 | a_2 | \dots | a_m)^* A$ ；

③ 对形如 $A \rightarrow a_1 B | a_2 B | \dots | a_m B$ 的产生式，构造方程式 $A = (a_1 | a_2 | \dots | a_m) B$ ，其中 $B \neq A$ 。



1. 根据正则文法构造等价的正则表达式

(2)解联立方程组，求等价的正则表达式 r

用代入消元法逐个消去方程组中除开始符号 S 外的其他变量，最后即可得到关于开始符号 S 的解。

代入消元规则如下：

- ① 如果有 $A=r_1B|r_2B|\dots|r_nB$ ，则用 $A=(r_1|r_2|\dots|r_n)B$ 替换之，其中 $B\neq A$ ；
- ② 如果有 $A=t_1A|t_2A|\dots|t_mA$ ，则用 $A=(t_1|t_2|\dots|t_m)^*A$ 替换之；

1. 根据正则文法构造等价的正则表达式

③ 如果有 $A=(r_1|r_2|\dots|r_n)B$, $B=(t_1|t_2|\dots|t_m)C$, 则用 $A=(r_1|r_2|\dots|r_n)(t_1|t_2|\dots|t_m)C$ 替换之, 其中 $B \neq A$;

如果有 $A=(r_1|r_2|\dots|r_n)B$, $B=(t_1|t_2|\dots|t_m)$, 则用 $A=(r_1|r_2|\dots|r_n)(t_1|t_2|\dots|t_m)$ 替换之, 其中 $B \neq A$;

④ 对 $A=(t_1|t_2|\dots|t_m)^*A$ 且 $A=(r_1|r_2|\dots|r_n)B$, 其中 $B \neq A$, 则用 $A=(t_1|t_2|\dots|t_m)^*(r_1|r_2|\dots|r_n)B$ 替换之;

对 $A=(t_1|t_2|\dots|t_m)^*A$ 且 $A=r_1|r_2|\dots|r_n$ 则用 $A=(t_1|t_2|\dots|t_m)^*(r_1|r_2|\dots|r_n)$ 替换之;

1. 根据正则文法构造等价的正则表达式

⑤ 如果有 $A=\beta_1$ 、 $A=\beta_2$ 、...、 $A=\beta_h$ ，则用 $A=\beta_1|\beta_2|\dots|\beta_h$ 代替之。

如果最后得到的关于 S 的方程式为如下形式，

$$S=\alpha_1|\alpha_2|\dots|\alpha_n$$

则将方程式右边所有其中**仍然含有语法变量的 $\alpha_i(1\leq i\leq n)$ 删除**，剩下的结果就是与 G 等价的正则表达式

如果任意的 $\alpha_i(1\leq i\leq n)$ 均含有语法变量，则 \emptyset 就是与 G 等价的正则表达式。

1. 根据正则文法构造等价的正则表达式

■ 例3.6 将如下文法 G 转换成相应的正则表达式

$$S \rightarrow aS|aB$$

$$B \rightarrow bB|bC|aB|bS$$

$$C \rightarrow cC|c$$

1. 列方程组

$$\blacksquare S = a^*S \quad S = aB$$

$$\blacksquare B = (a|b)^*B \quad B = bC \quad B = bS$$

$$\blacksquare C = c^*C \quad C = c$$

1. 根据正则文法构造等价的正则表达式

1. 列方程组

$$\begin{aligned} \blacksquare S &= a^* S & S &= aB & B &= (a|b)^* B & B &= bC \\ \blacksquare B &= bS & C &= c^* C & C &= c \end{aligned}$$

2. 代入法解方程组

$$\begin{aligned} \blacksquare C &= c^* c & (\text{由 } C &= c^* C \text{ 和 } C = c) \\ \blacksquare B &= bc^* c & (\text{由 } B &= bC \text{ 和 } C = c^* c) \\ \blacksquare B &= (a|b)^* (bc^* c) & (\text{由 } B &= (a|b)^* B \text{ 和 } B = bc^* c) \\ \blacksquare B &= (a|b)^* bS & (\text{由 } B &= (a|b)^* B \text{ 和 } B = bS) \\ \blacksquare S &= a^* aB & (\text{由 } S &= a^* S \text{ 和 } S = aB) \\ \blacksquare S &= (a^* a(a|b)^* bS) | (a^* a(a|b)^* (bc^* c)) \\ \blacksquare S &= (a^* a(a|b)^* b)^* a^* a(a|b)^* (bc^* c) \\ \blacksquare \text{如果用正闭包表示, 则为 } & (a^+(a|b)^* b)^* a^+(a|b)^* (bc^+) \end{aligned}$$

2. 将正则表达式转换成等价的正则文法

- **问题：给定 Σ 上的一个正则表达式 r ，根据 r 构造正则文法 G ，使得 $L(G)=L(r)$**
- **定义3.3 设字母表为 Σ ， $\{A、B、...\}$ 为语法变量集合**
 - **对于 Σ 上的任意正则表达式 r ，形如 $A \rightarrow r$ 的式子称为正则定义式（即：正规定义式）；**
 - **如果 r 是 Σ 中的字母和用正则定义式定义的变量组成的正则表达式，则形如 $A \rightarrow r$ 的式子也称为正则定义式（即：正规定义式）。**

2. 将正则表达式转换成等价的正则文法

■ 例子：标识符的正规定义式

- $\langle \text{letter} \rangle \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$
- $\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$
- $\langle \text{id} \rangle \rightarrow \langle \text{letter} \rangle (\langle \text{digit} \rangle \mid \langle \text{letter} \rangle)^*$

用正则定义式定义的变量组成的正则表达式



2. 将正则表达式转换成等价的正则文法

- 给定正则表达式 r ，按如下方法构造正则定义式，并逐步将其转换成正则文法
- 引入开始符号 S ，从如下正则定义式开始
 - $S \rightarrow r$
- 按如下规则将 $S \rightarrow r$ 分解为新的正则定义式，在分解过程中根据需要引入新的语法变量

2. 将正则表达式转换成等价的正则文法

■ $A \rightarrow r$ 是正则定义式，则对 $A \rightarrow r$ 的分解规则如下：

(1) 如果 $r = r_1 r_2$ ，则将 $A \rightarrow r$ 分解为 $A \rightarrow r_1 B$, $B \rightarrow r_2$, $B \in V$;

(2) 如果 $r = r_1^* r_2$ ，则将 $A \rightarrow r$ 分解为 $A \rightarrow r_1 A$, $A \rightarrow r_2$;

(3) 如果 $r = r_1 | r_2$ ，则将 $A \rightarrow r$ 分解为 $A \rightarrow r_1$, $A \rightarrow r_2$ 。

不断应用分解规则(1)到(3)对各个正则定义式进行分解，直到每个正则定义式右端只含一个语法变量(即符合正则文法产生式的形式)为止。



2. 将正则表达式转换成等价的正则文法

■ 例子:

- 将正则表达式 $a(a|b)^*$ 转换为相应的正则文法
- 将正则表达式 $a(a|b)^*(\varepsilon|((\cdot|_) (a|b)(a|b)^*))$ 转换成相应的正则文法



例 3.10 标识符定义的转换

- 引入 S

$$S \rightarrow \langle \text{letter} \rangle (\langle \text{letter} \rangle | \langle \text{digit} \rangle)^*$$

- 分解为

$$S \rightarrow \langle \text{letter} \rangle A$$

$$A \rightarrow (\langle \text{letter} \rangle | \langle \text{digit} \rangle) A | \varepsilon$$

- 执行连接对|的分配律

$$S \rightarrow \langle \text{letter} \rangle A$$

$$A \rightarrow \langle \text{letter} \rangle A | \langle \text{digit} \rangle A | \varepsilon$$



高级语言词法的简单描述

■ 词法

- 单词符号的文法，用来描述高级语言中的：
标识符、常数、运算符、分界符、关键字
- 参考教材P73-77，了解如何定义高级语言中的整数、实数……等的相应正则文法。

例 3.7 某简易语言的词法

——正则定义式

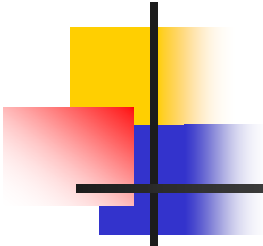
词法规则

单词种别 属性

$\langle \text{标识符} \rangle \rightarrow \langle \text{字母} \rangle (\langle \text{字母} \rangle | \langle \text{数字} \rangle)^*$ *IDN* 符号表入口

$\langle \text{无符号整数} \rangle \rightarrow \langle \text{数字} \rangle (\langle \text{数字} \rangle)^*$ *NUM* 数值

$\langle \text{赋值符} \rangle \rightarrow :=$ *ASG* 无



变换为正规文法

<标识符> \rightarrow letter<标识符尾>

<标识符尾> $\rightarrow \varepsilon$ | letter<标识符尾> | digit<标识符尾>

<整数> \rightarrow digit <整数尾>

<整数尾> $\rightarrow \varepsilon$ | digit<整数尾>

<赋值号> \rightarrow :=

<加号> \rightarrow +

<等号> \rightarrow =

...

(其它：实数、算术运算符、关系运算符、分号、括号等)



3.2.4 有穷状态自动机

- **正则语言的另一种等价描述**
- **从语言识别的角度实现对相应语言的刻画**

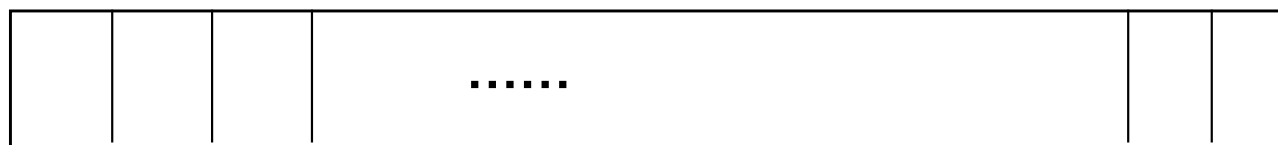


3.2.4 有穷状态自动机

- 具有离散输入输出的系统的数学模型
- 具有**有穷个内部状态**，不同状态代表不同意义
- 系统只需根据当前所处的**状态**和面临的**输入**就能确定**后继**的行为
- 处理完当前输入后系统的状态将发生变化
- 具有**初始状态**，系统在该状态下开始进行某个句子的处理
- **终止状态**集合，该状态表示到目前为止所读入的字符构成的字符串是语言的一个句子

有穷自动机的物理模型

输入带



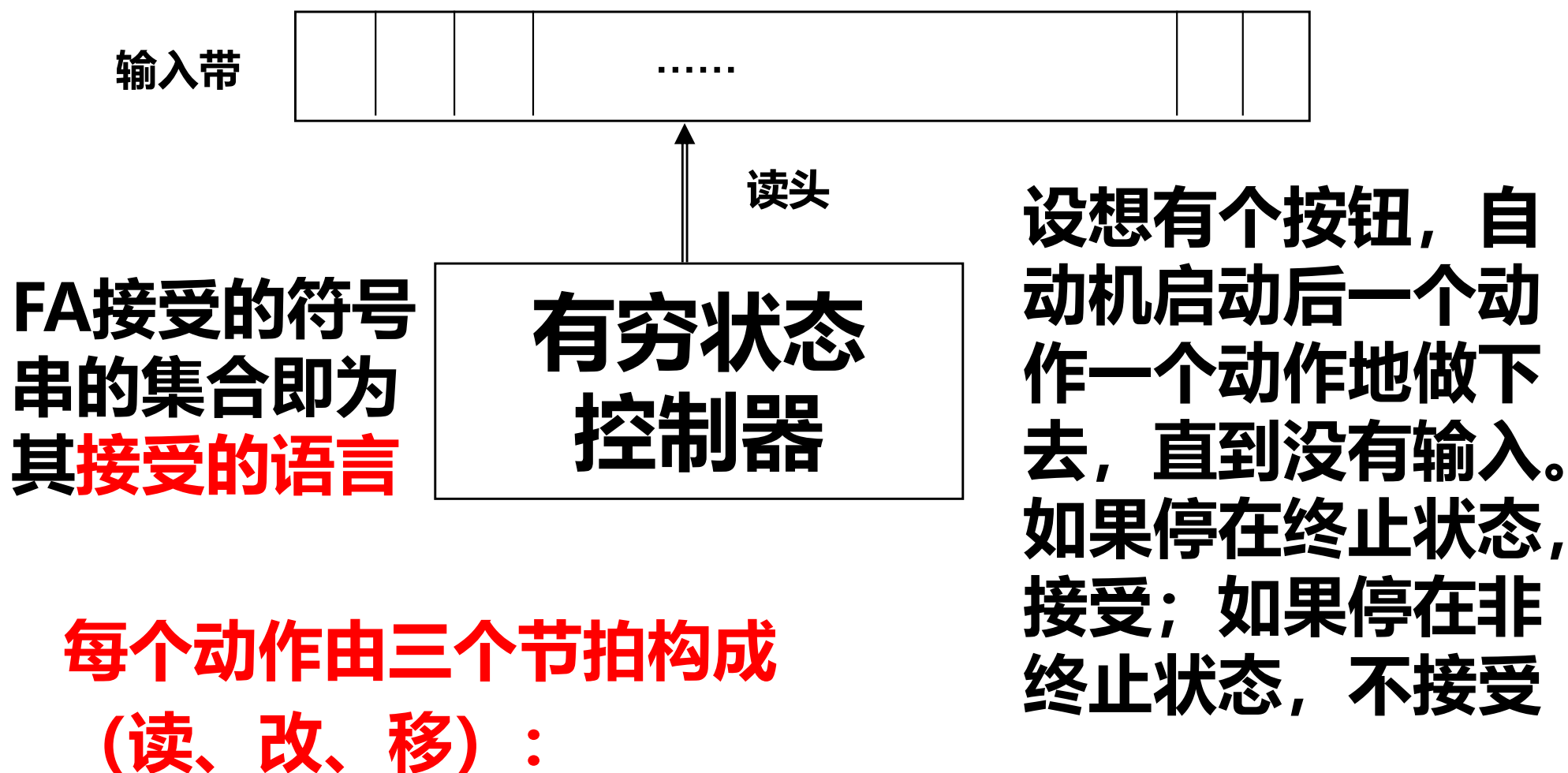
读头

**有穷状态
控制器**

该控制器的状态只有有穷多个，控制一个读头，从输入带上读取字符。每读取一个字符，就将读头指向下一个待读取的字符

该模型有一个**输入带**，输入带上有一系列的“**带方格**”，每个带方格可以**存放一个字符**，输入串从输入带的左端开始存放，输入带右端则是无穷的

有穷自动机的物理模型



$[p, a] \rightarrow q$, 读头前进一格

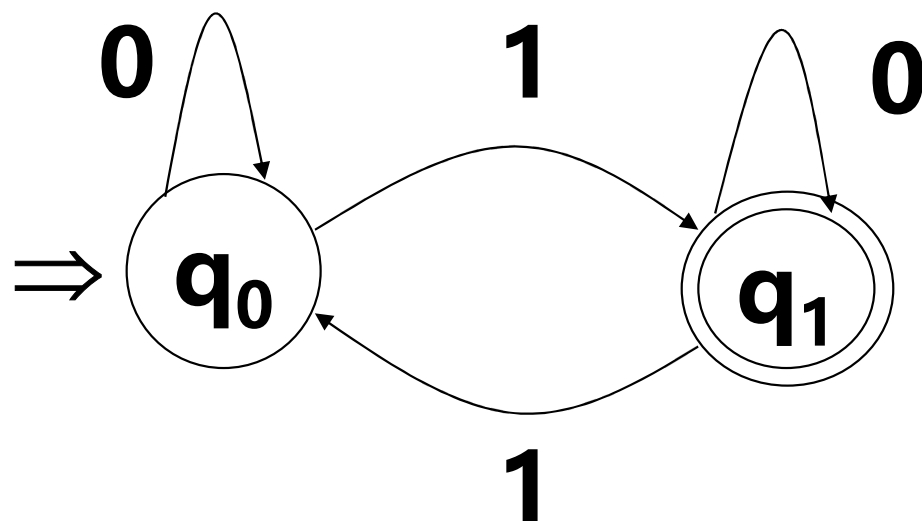


有穷自动机的用处

- **有穷自动机是许多重要类型的硬件和软件
的有用模型**
 - **数字电路的设计和检查软件**
 - **典型编译器的词法分析器**
 - **扫描大量文本来发现单词、短语或其他模式的出现的软件**
 - **所有只有有穷个不同状态的系统（如通信协议或安全交换信息的协议）的验证软件**

例：一个奇偶校验器

测试输入中1的个数的奇偶性，并且只接收含有奇数个1的那些输入串。



注意：状态有记忆功能，记住输入串的部分特征。

问题：有穷自动机的形式描述？

关键是如何描述动作？

确定的有穷自动机的形式定义

定义3.4 一个**确定的有穷自动机** M (记作DFA M) 是一个五元组 $M = (Q, \Sigma, \delta, q_0, F)$, 其中

- ① Q 是一个有穷状态集合。
- ② Σ 是一个字母表, 它的每个元素称为输入符号。
- ③ $q_0 \in Q$, q_0 称为初始状态。
- ④ $F \subseteq Q$, F 称为终止状态集合。
- ⑤ δ (状态转移函数) 是一个从 $Q \times \Sigma$ 到 Q 的单值映射

$$\delta(p, a) = q \quad (p, q \in Q, a \in \Sigma)$$

表示当前状态为 p , 输入符号为 a 时, 自动机将转换到下一个状态 q , q 称为 p 的一个**后继**。

确定的有穷自动机的形式定义

定义3.4 一个**确定的有穷自动机** M (记作DEFA M) 是一

个

基本执行过程

1. 给定一个输入字符串，自动机 M 从开始状态读入该串的第1个字符出发
2. 每处理完一个字符，就进入下一个状态，并在此新状态下读入下一个字符
3. 按照这个过程，直到整个字符串被处理完毕为止。
4. 此时，如果 M 位于终止状态，则接受该输入，否则不接受该输入

表示当前状态为 p ，输入符号为 a 时，自动机将转换到下一个状态 q ， q 称为 p 的一个**后继**。

DFA的表示

例 设DFA $M = (\{0, 1, 2, 3\}, \{a, b\}, \delta, 0, \{3\})$

其中:

$$\begin{aligned}\delta(0, a) &= 1, & \delta(1, a) &= 3 \\ \delta(2, a) &= 1, & \delta(3, a) &= 3 \\ \delta(0, b) &= 2, & \delta(1, b) &= 2 \\ \delta(2, b) &= 3, & \delta(3, b) &= 3\end{aligned}$$

一个DFA有三种表示:

- (1) 转移函数;
- (2) 转移矩阵;
- (3) 状态转换图。

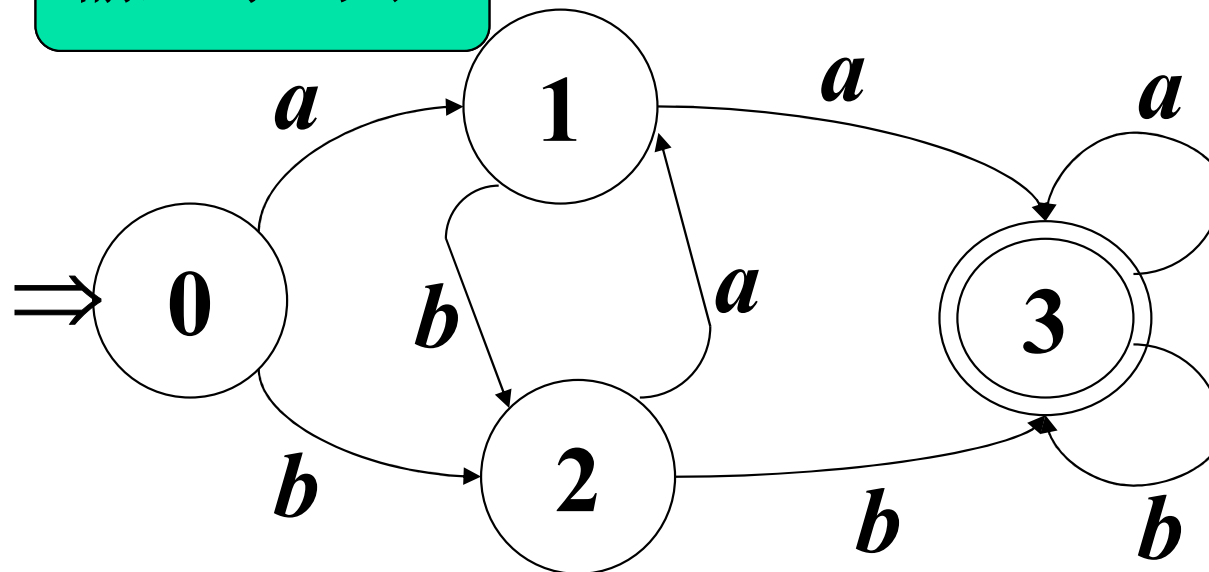
这就是转移函数表示

转移矩阵

状态转换图

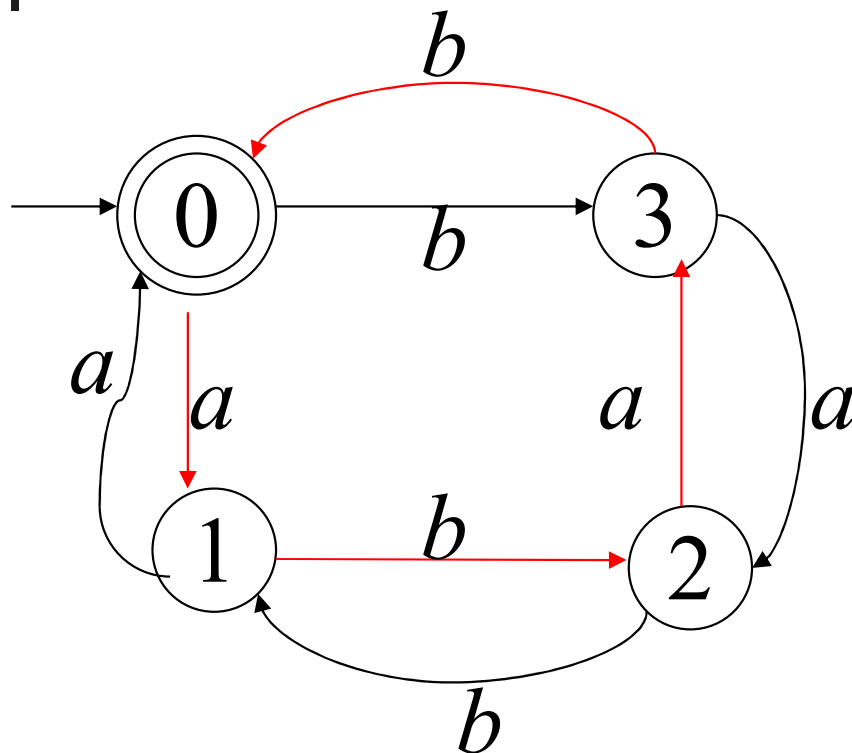
| | <i>a</i> | <i>b</i> |
|---|----------|----------|
| 0 | 1 | 2 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 3 | 3 | 3 |

输入字母表



状态集合

DFA M 接受的语言



从状态转换图看，从初始状态出发，沿任一条路径到达接受状态，这条路径的弧上的标记符号连接起来构成的符号串被接受。

如： $abab$

问题：如何描述DFA接受的语言？

DFA M 接受的语言

对所有 $w \in \Sigma^*$, $a \in \Sigma$, $q \in Q$ 以下述方式递归地扩展 δ 的定义,

$$(1) \hat{\delta}(q, \varepsilon) = q,$$

$$(2) \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a),$$

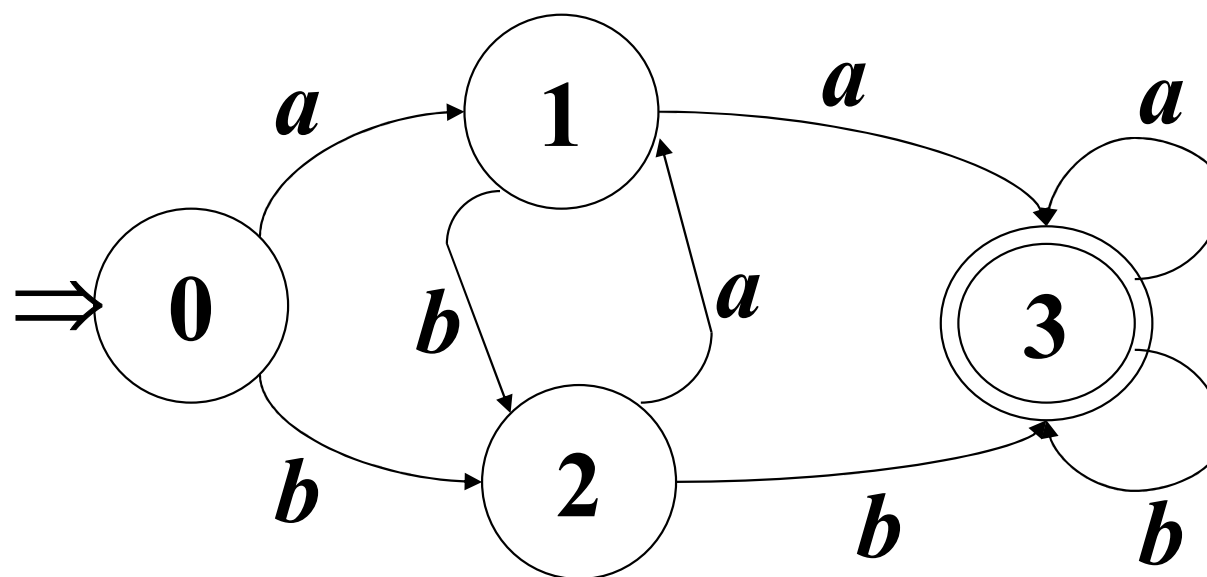
也可用 δ 来代替 $\hat{\delta}$

从原本针对单个输入字符的转移函数, 变化为针对零个或者多个输入字符的转移函数

转移矩阵

| | a | b |
|---|-----|-----|
| 0 | 1 | 2 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 3 | 3 | 3 |

状态转换图



例DFA M 和 $w=baa$,

$$\delta(0, baa) = \delta(2, aa) = \delta(1, a) = 3$$



DFA M 接受的语言

- **定义：** 给定DFA M , M 所接收的语言为：
$$L(M) = \{w \mid w \in \Sigma^*, \text{ 且 } \delta(q_0, w) \in F\}$$
- **定义：** 假设 M_1 和 M_2 都是DFA, 如果
 $L(M_1) = L(M_2)$, 则称 M_1 和 M_2 等价



非确定的有穷自动机NFA M

定义3.6 非确定的有穷自动机M是一个五元组

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中 Q, Σ, q_0, F 的意义和DFA的定义一样,
而 δ 是一个从 $Q \times (\Sigma \cup \{\varepsilon\})$ 到 Q 的子集的映射,
即 $\delta: Q \times S \rightarrow 2^Q$, 其中 $S = \Sigma \cup \{\varepsilon\}$ 。

类似于DFA, NFA M亦可用状态转换图表示,
同样也可以定义NFA M接受的语言。



NFA和DFA的等价性

- 每一台非确定型有穷自动机都等价于某一台确定型有穷自动机
- 基本思路：设一个语言被一台NFA识别，那么必证明还存在一台DFA也识别这个语言
- 设 k 是NFA的状态数，则它有 2^k 个状态子集，每一个子集对应模拟这台NFA的DFA必须记住的一种可能性，所以这台DFA会有 2^k 个状态

DFA M的模拟算法

输入：以eof结尾的串 x , DFA $M = (Q, \Sigma, \delta, q_0, F)$;

输出：如果 M 接受 x 则输出 “yes”, 如果 M 不接受 x 则输出 “no”

步骤:

1 $s = q_0$;

2 $c = \text{getchar}(x)$;

3 while ($c \neq \text{eof}$) {

4 $s = \text{move}(s, c)$;

5 $c = \text{getchar}(x)$;

6 }

7 if $s \in F$ return “yes”

8 else return “no”;

根据转移函数的转换

读取下一个字符



例：构造有穷状态自动机

令 $\Sigma = \{0, 1\}$, $L = \{x \mid x \in \Sigma^*, x \text{ 中 } 0 \text{ 的个数和 } 1 \text{ 的个数都是偶数}\}$, 试构造一个 DFA M , 使得 $L(M) = L$ 。

例：构造有穷状态自动机

令 $\Sigma = \{0, 1\}$, $L = \{x \mid x \in \Sigma^*, x \text{ 中 } 0 \text{ 的个数和 } 1 \text{ 的个数都是偶数}\}$, 试构造一个 DFA M , 使得 $L(M) = L$ 。

考虑状态之间的转移

q_{00}

偶数个0
偶数个1

q_{01}

偶数个0
奇数个1

q_{10}

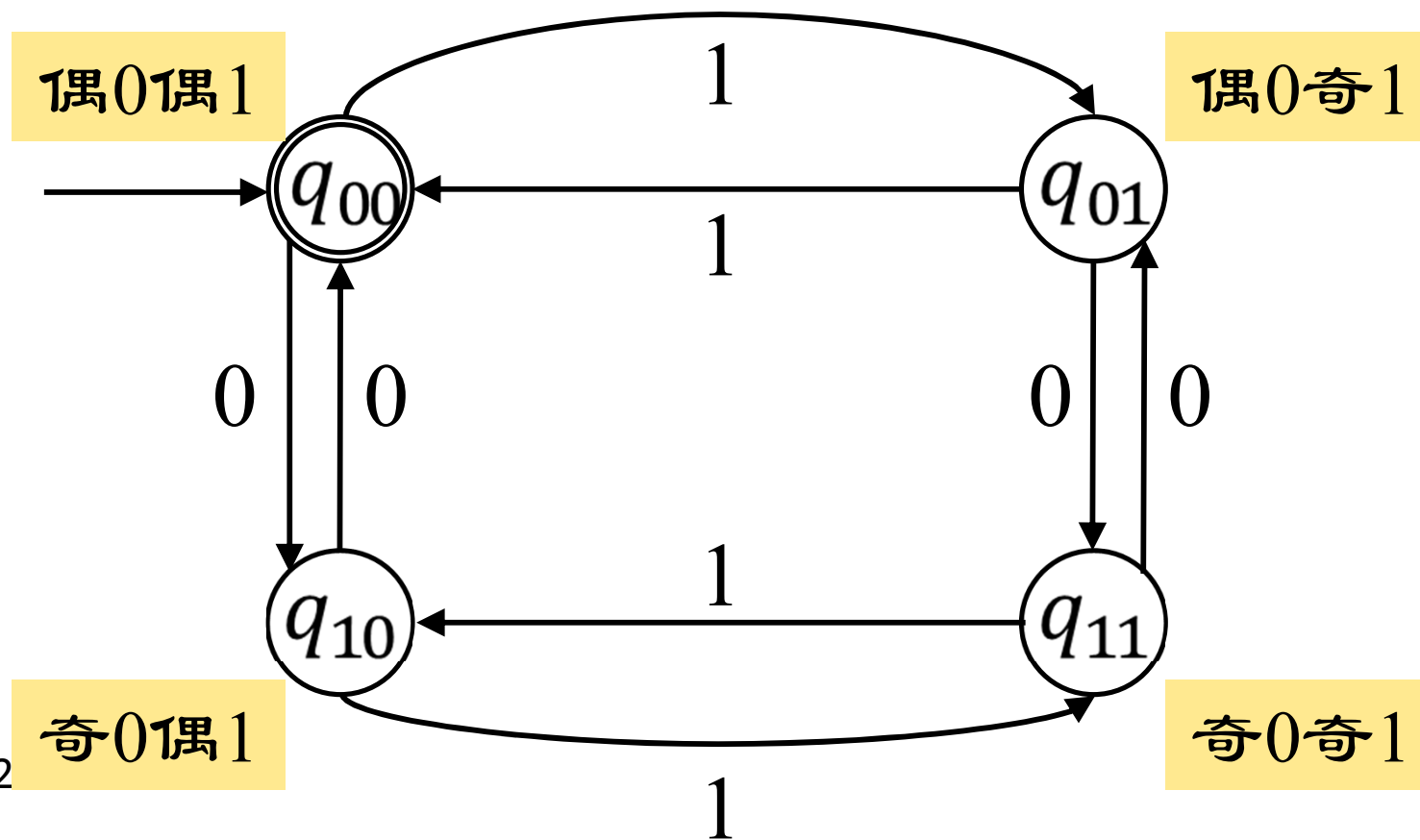
奇数个0
偶数个1

q_{11}

奇数个0
奇数个1

例：构造有穷状态自动机

令 $\Sigma = \{0, 1\}$, $L = \{x \mid x \in \Sigma^*, x \text{ 中 } 0 \text{ 的个数和 } 1 \text{ 的个数都是偶数}\}$, 试构造一个 DFA M , 使得 $L(M) = L$ 。



3.2.5 状态转换图

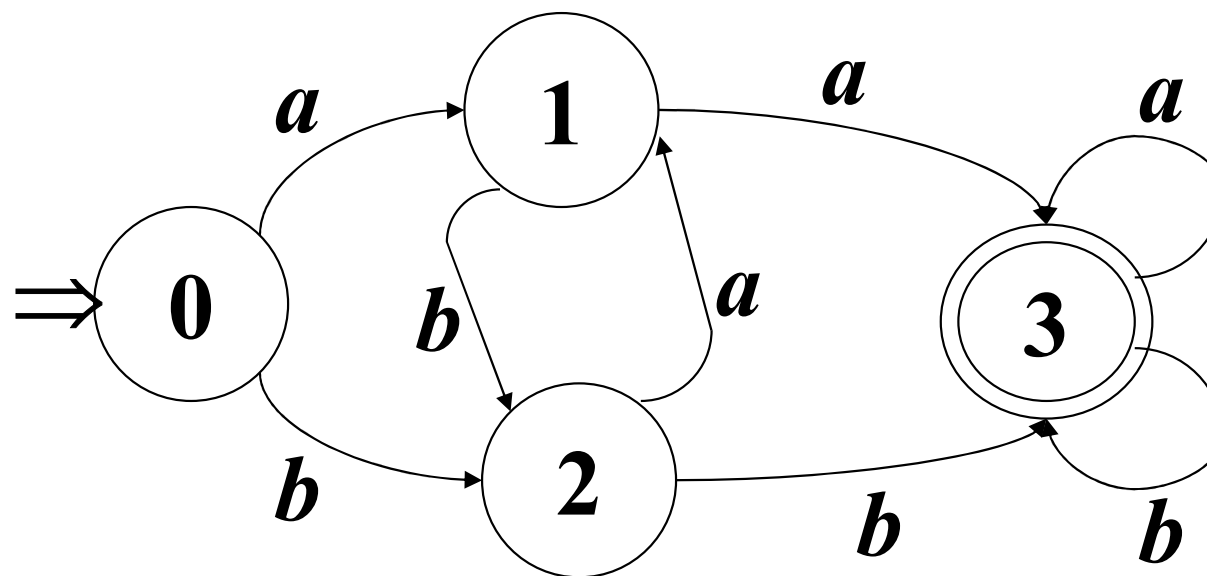
- 定义3.7 设 $M=(Q, \Sigma, \delta, q_0, F)$ 为一个有穷状态自动机，满足如下条件的有向图被称为 **M 的状态转换图**(transition diagram):
 - (1) $q \in Q \Leftrightarrow q$ 是该有向图中的一个顶点;
 - (2) $\delta(q, a)=p \Leftrightarrow$ 图中有一条从顶点 q 到顶点 p 的标记为 a 的弧;
 - (3) $q \in F \Leftrightarrow$ 标记为 q 的顶点被用双层圈标出;
 - (4) 用标有start的箭头指出 M 的开始状态。

3.2.5 状态转换图

转移矩阵

| | a | b |
|---|-----|-----|
| 0 | 1 | 2 |
| 1 | 3 | 2 |
| 2 | 1 | 3 |
| 3 | 3 | 3 |

状态转换图





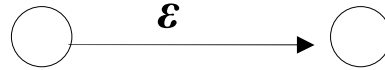
3.2.6 正则表达式转换为状态转换图

- 已知，有穷状态自动机也是正则语言的一种描述，即它和正则表达式是等价的
- 正则表达式和状态转换图之间可以互相转换
- 下面给出正则表达式到状态转换图的基本转换规则
- **注意：自动机看作对正则表达式的识别过程**

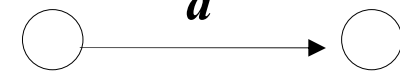
3.2.6 正则表达式转换为状态转换图



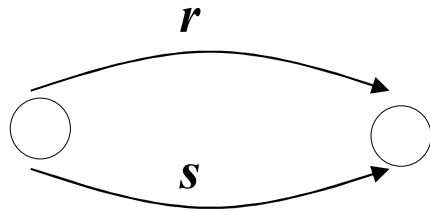
(a) \emptyset 对应的状态转换图



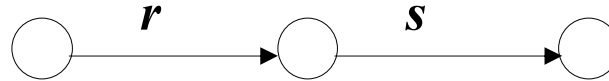
(b) ϵ 对应的状态转换图



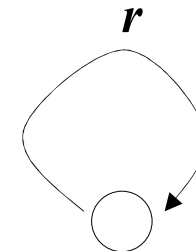
(c) a 对应的状态转换图



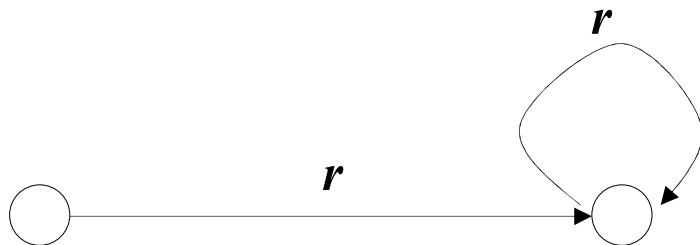
(d) $r \mid s$ 对应的状态转换图



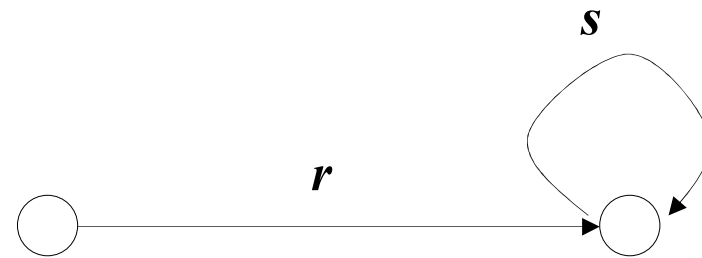
(e) rs 对应的状态转换图



(f) r^* 对应的状态转换图



(g) r^+ 对应的状态转换图



(h) rs^* 对应的状态转换图

图3.8 典型正则表达式对应的状态转换图

3.2.6 正则表达式转换为状态转换图

■ 转换过程如下：

- 设置一个开始状态和一个终止状态，从开始状态到终止状态引一条**标记为待转换正则表达式的边**；
- 检查图中边的标记，如果相应的标记**不是字符、 \emptyset 、 ϵ 或用“|”连接的字符和 ϵ** ，则根据规则(a)-(h)进行替换，直到图中不存在不满足要求的边。
- 按照习惯，如果一条边上标记的是 \emptyset ，这个边就不用画出来。

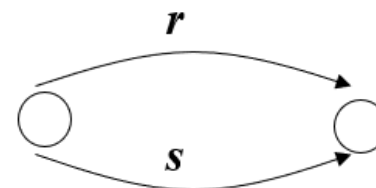
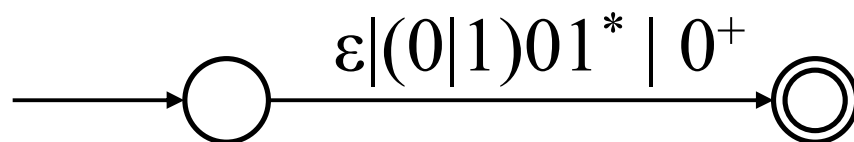


3.2.6 正则表达式转换为状态转换图

- 例子：构造 $\varepsilon|(0|1)01^*|0^+$ 的状态转换图

3.2.6 正则表达式转换为状态转换图

- 例子：构造 $\varepsilon|(0|1)01^*|0^+$ 的状态转换图



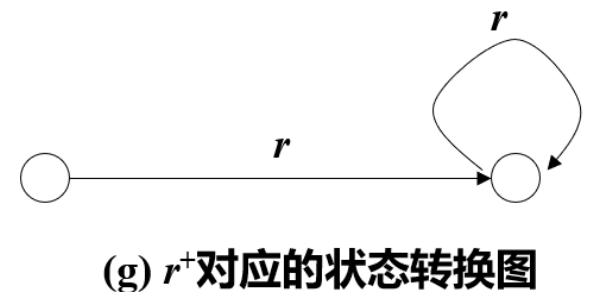
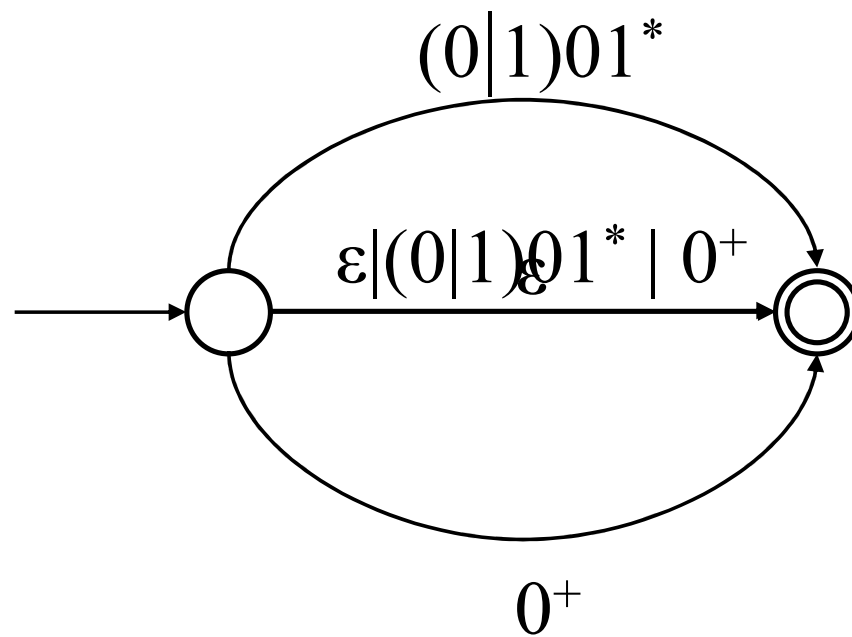
(d) $r|s$ 对应的状态转换图

第二步：

检查图中边的标记，如果相应的标记不是字符、 ε 或用“|”连接的字符和 ε ，则根据规则(a)-(h)进行替换，直到图中不再存在不满足要求的边。

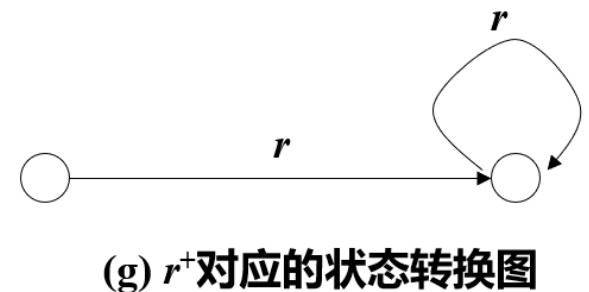
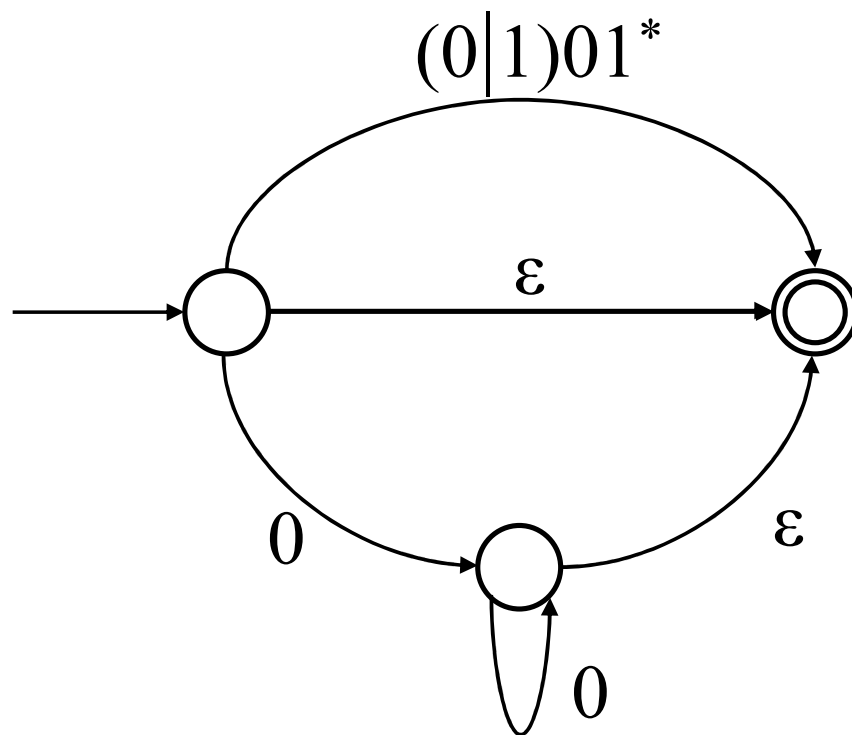
3.2.6 正则表达式转换为状态转换图

- 例子：构造 $\epsilon|(0|1)01^*|0^+$ 的状态转换图



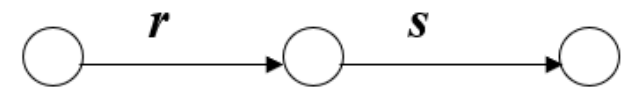
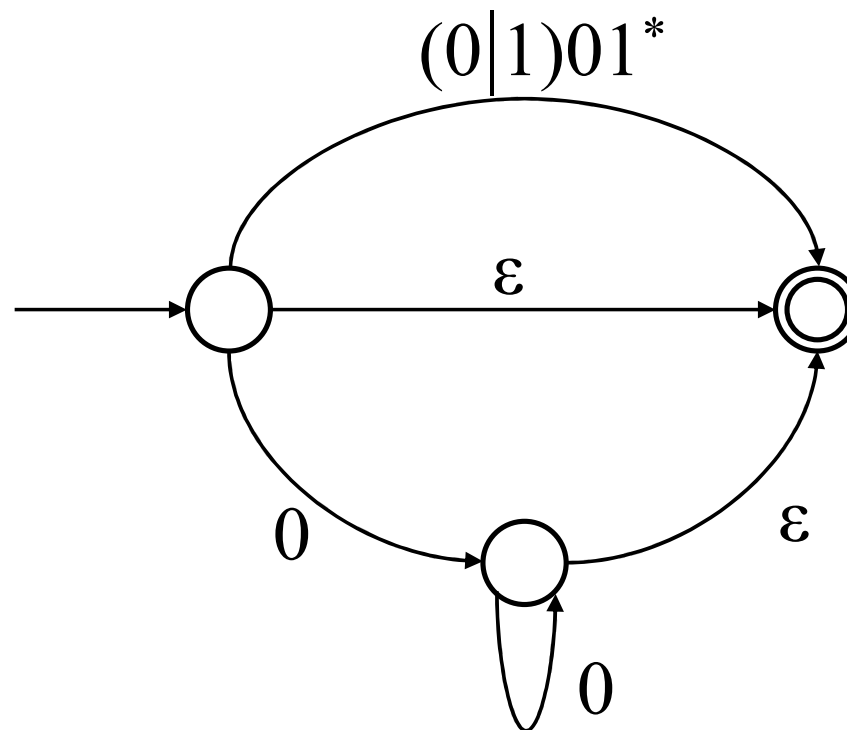
3.2.6 正则表达式转换为状态转换图

- 例子：构造 $\varepsilon|(0|1)01^*|0^+$ 的状态转换图



3.2.6 正则表达式转换为状态转换图

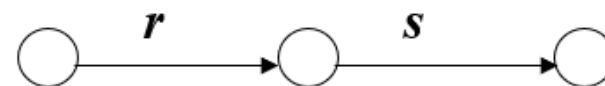
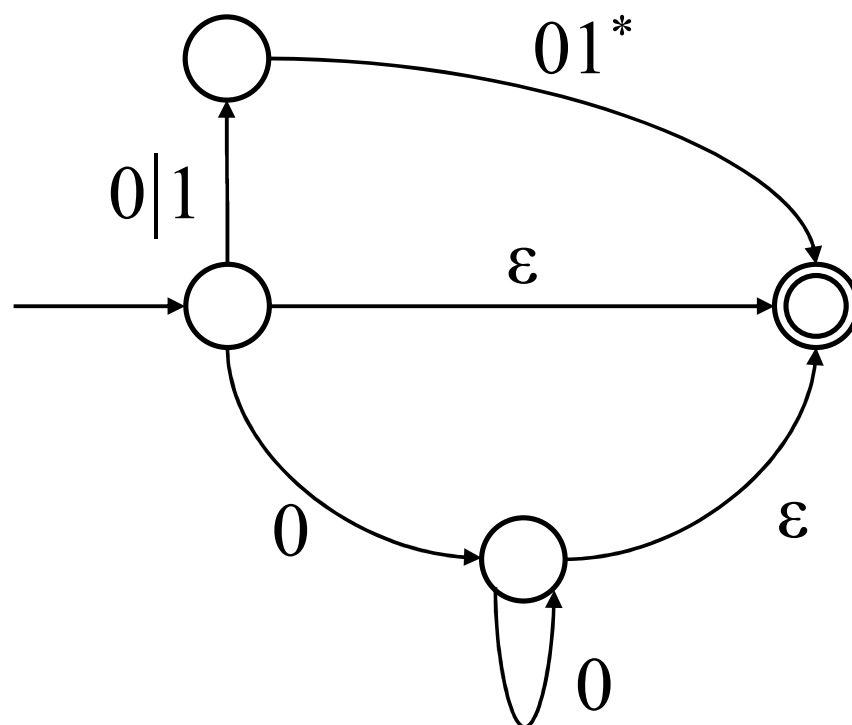
- 例子：构造 $\epsilon|(0|1)01^*|0^+$ 的状态转换图



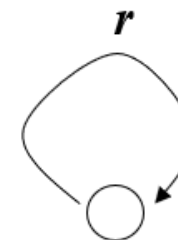
(e) rs 对应的状态转换图

3.2.6 正则表达式转换为状态转换图

- 例子：构造 $\varepsilon|(0|1)01^*|0^+$ 的状态转换图



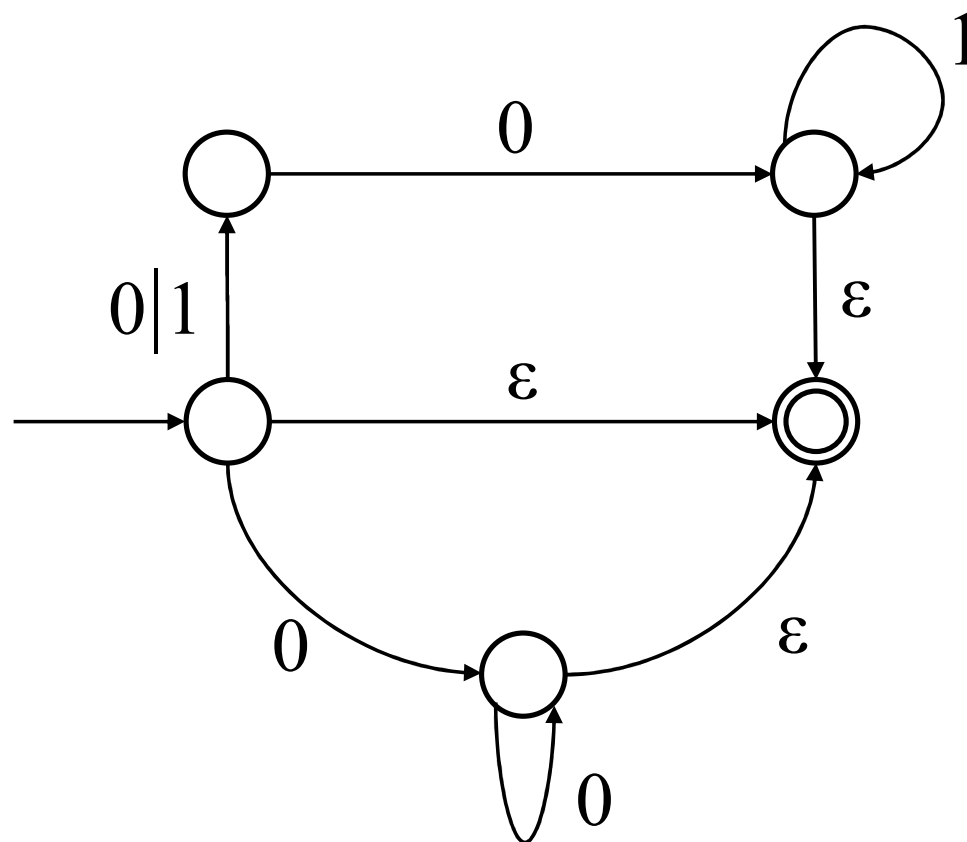
(e) rs 对应的状态转换图



(f) r^* 对应的状态转换图

3.2.6 正则表达式转换为状态转换图

- 例子：构造 $\varepsilon|(0|1)01^*|0^+$ 的状态转换图



3.3 单词的识别

3.3.1 有穷状态自动机与单词识别的关系

- **有穷状态自动机和正则文法等价，考虑到状态转换图的直观性，我们从状态转换图出发来考虑词法分析器的设计。**

3.3 单词的识别

3.3.1 有穷状态自动机与单词识别的关系

- 单词的识别过程相当于单词的**拼接过程**，一个字符一个字符地逐步进行
- 单词的识别总可以在有限的步骤内完成
- 单词的识别过程可以看成有限个状态的变换，每个状态反映的是某种**识别程度**
- 有一个初始状态和若干个终止状态，初始状态表示识别的开始，终止状态表示识别的结束

3.3 单词的识别

3.3.1 有穷状态自动机与单词识别的关系

- 允许在状态转换图的边上标记像 **digit**、**letter** 这样意义明确的符号
- 一个离开状态 r 的边上标记 **other**，表示除了离开 r 的其他边上标记的字符之外的任何字符



3.3.1 有穷状态自动机与单词识别的关系

- 考虑到在识别单词的过程中需要执行一些动作，将这些**动作标记**标在基本的状态转换图上。
- 状态上的*表示向前指针必须回退一个字符（和超前搜索对应）。



3.3.1 有穷状态自动机与单词识别的关系

- 关系运算符 \geq 和 $>$ 的识别
- 标识符和关键字的识别



例 3.14 不同进制无符号整数的识别

八进制数: (OCT, 值)

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

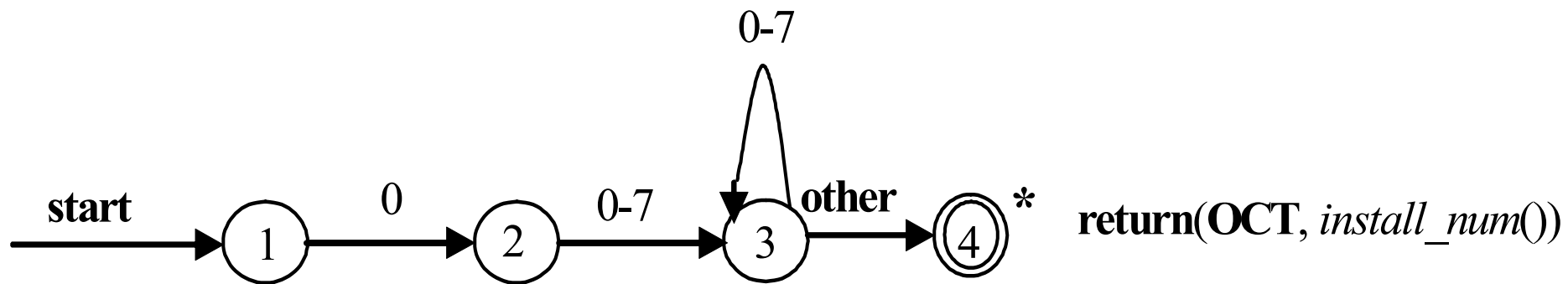
十进制数: (DEC, 值)

- $\text{dec} \rightarrow (1|\dots|9)(0|\dots|9)^* | 0$

十六进制数: (HEX, 值)

- $\text{hex} \rightarrow 0\text{x}(0|1|\dots|9|\text{a}|\dots|\text{f})(0|\dots|9|\text{a}|\dots|\text{f})^*$

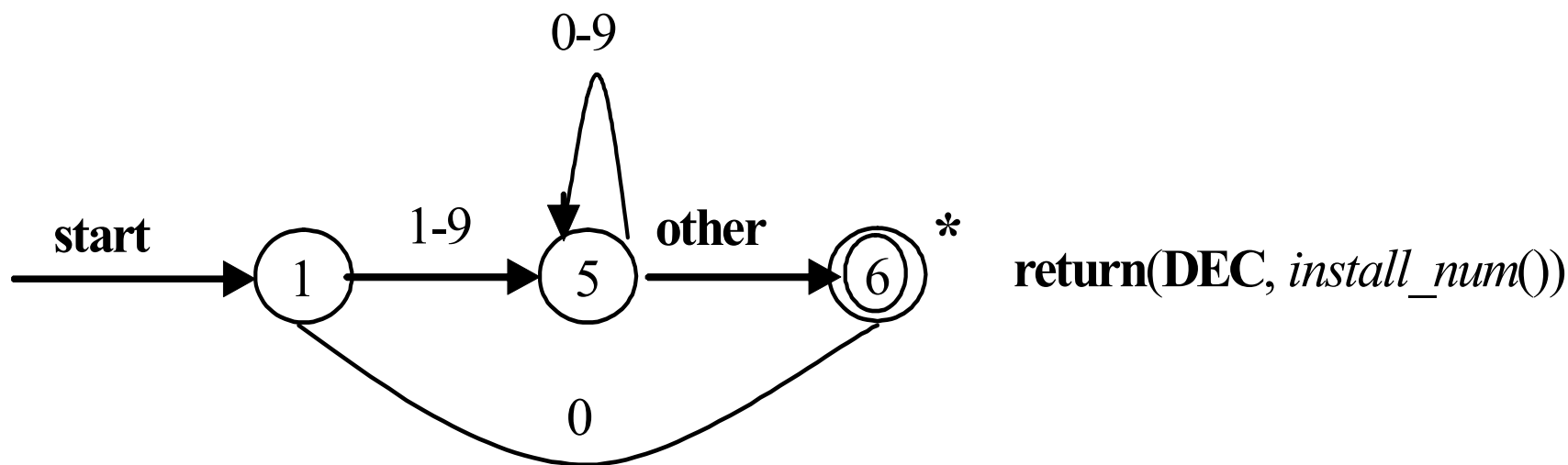
识别不同进制数的状态图



八进制数: $\text{oct} \rightarrow 0(0|1|2|3|4|5|6|7)(0|1|2|3|4|5|6|7)^*$

图3.11 识别八进制无符号整数的状态转换图

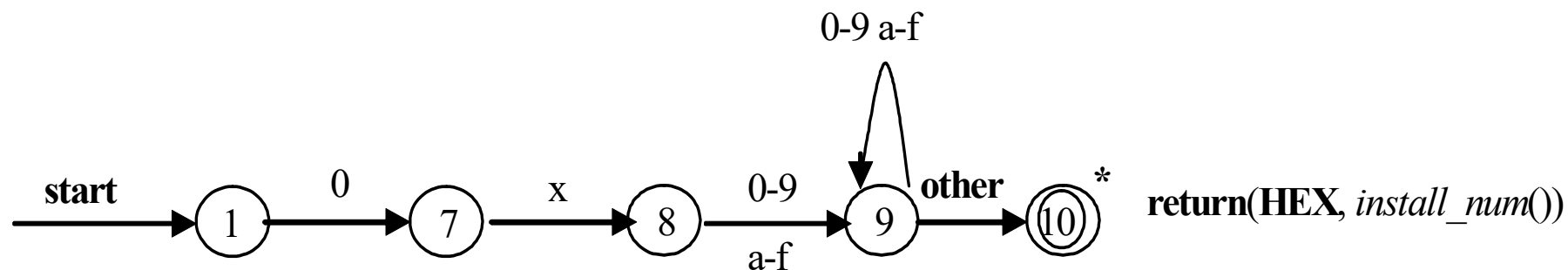
识别不同进制数的状态图



十进制数: $\text{dec} \rightarrow (1|...|9)(0|...|9)^* | 0$

图3.12 识别十进制无符号整数的状态转换图

识别不同进制数的状态图

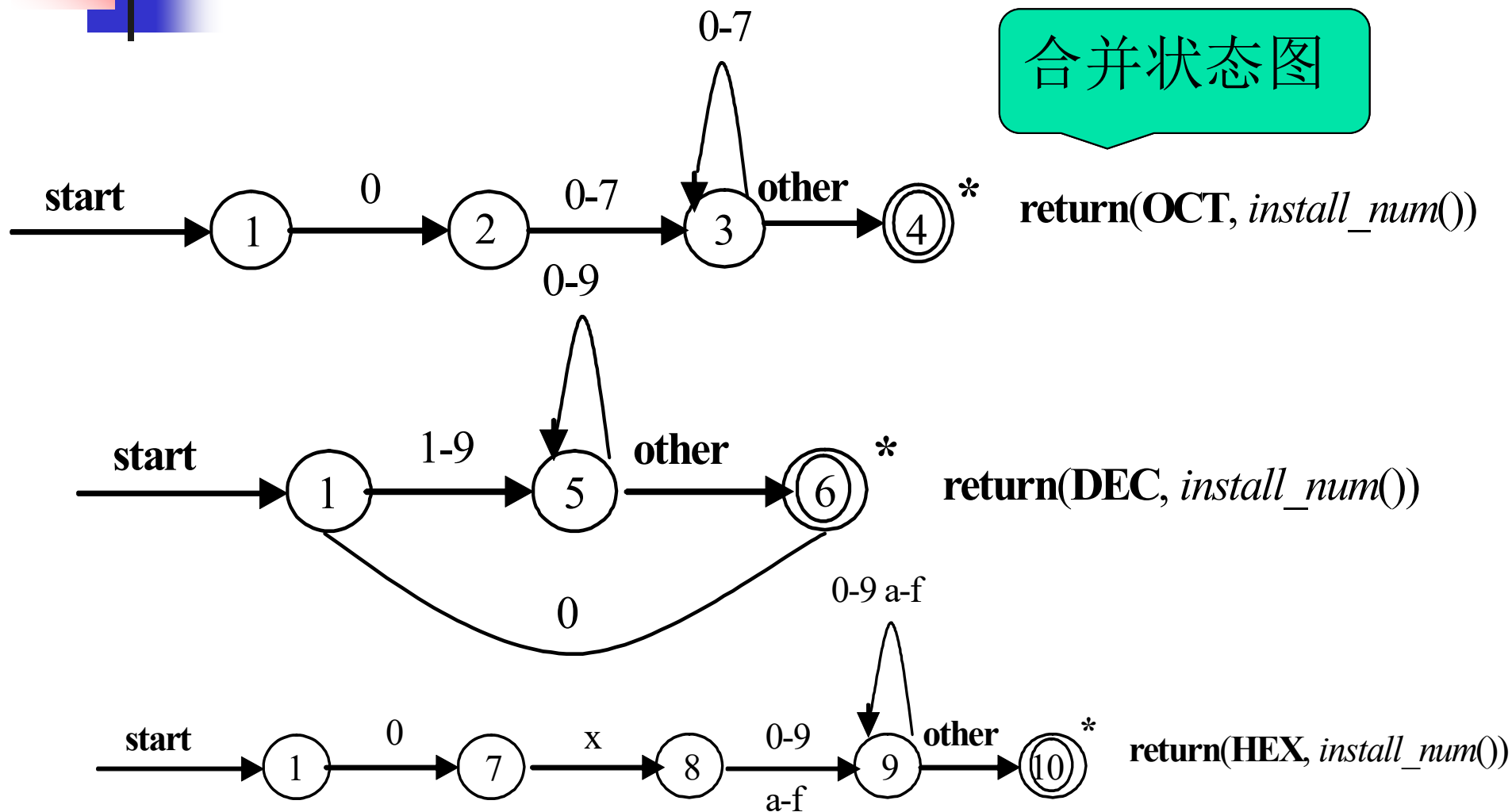


十六进制数: $\text{hex} \rightarrow 0\text{x}(0|1|\dots|9|\text{a}|\dots|\text{f})(0|\dots|9|\text{a}|\dots|\text{f})^*$

图3.13 识别十六进制无符号整数的状态转换图

识别不同进制数的状态图

合并状态图



识别不同进制数的状态图

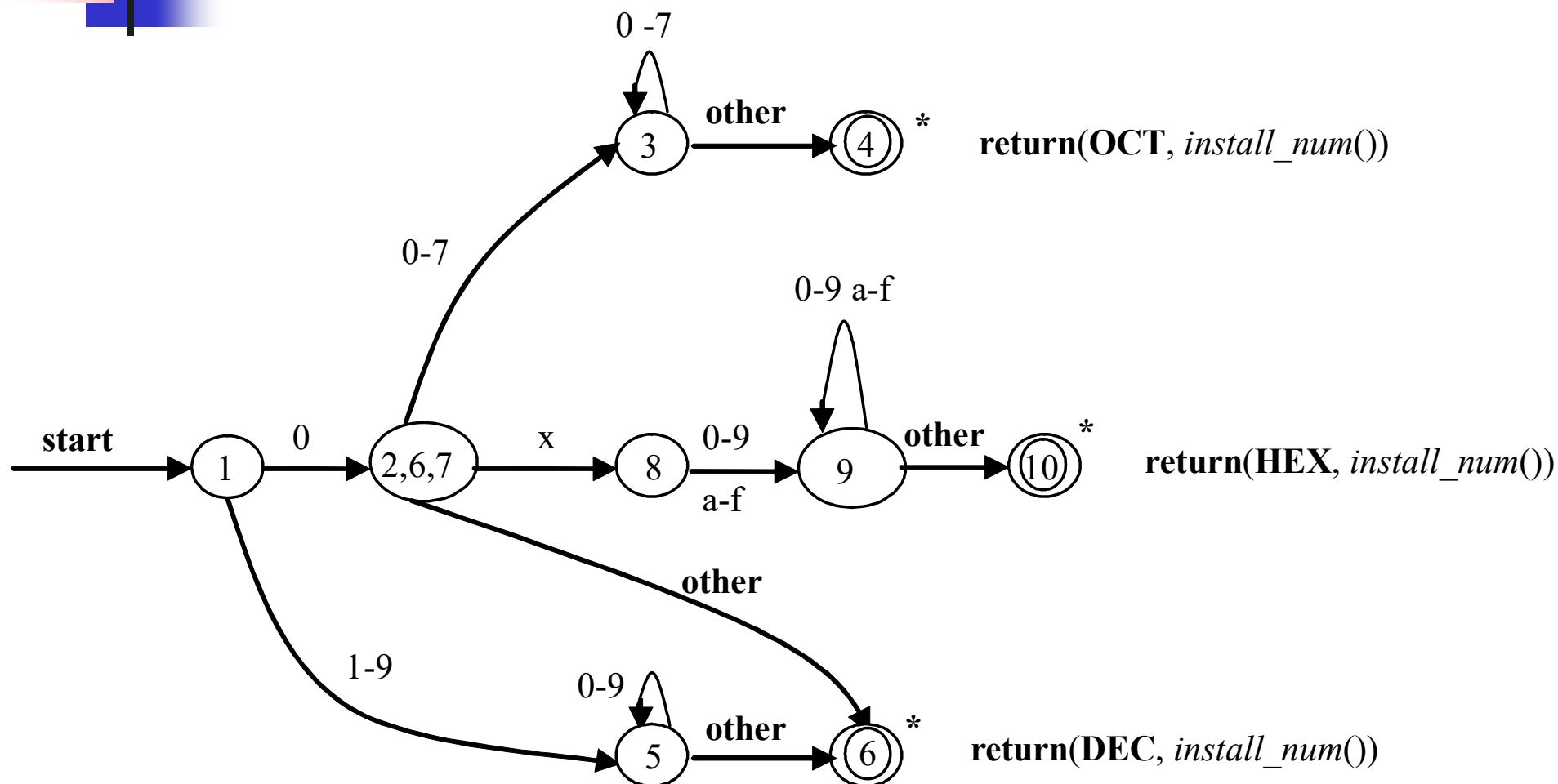
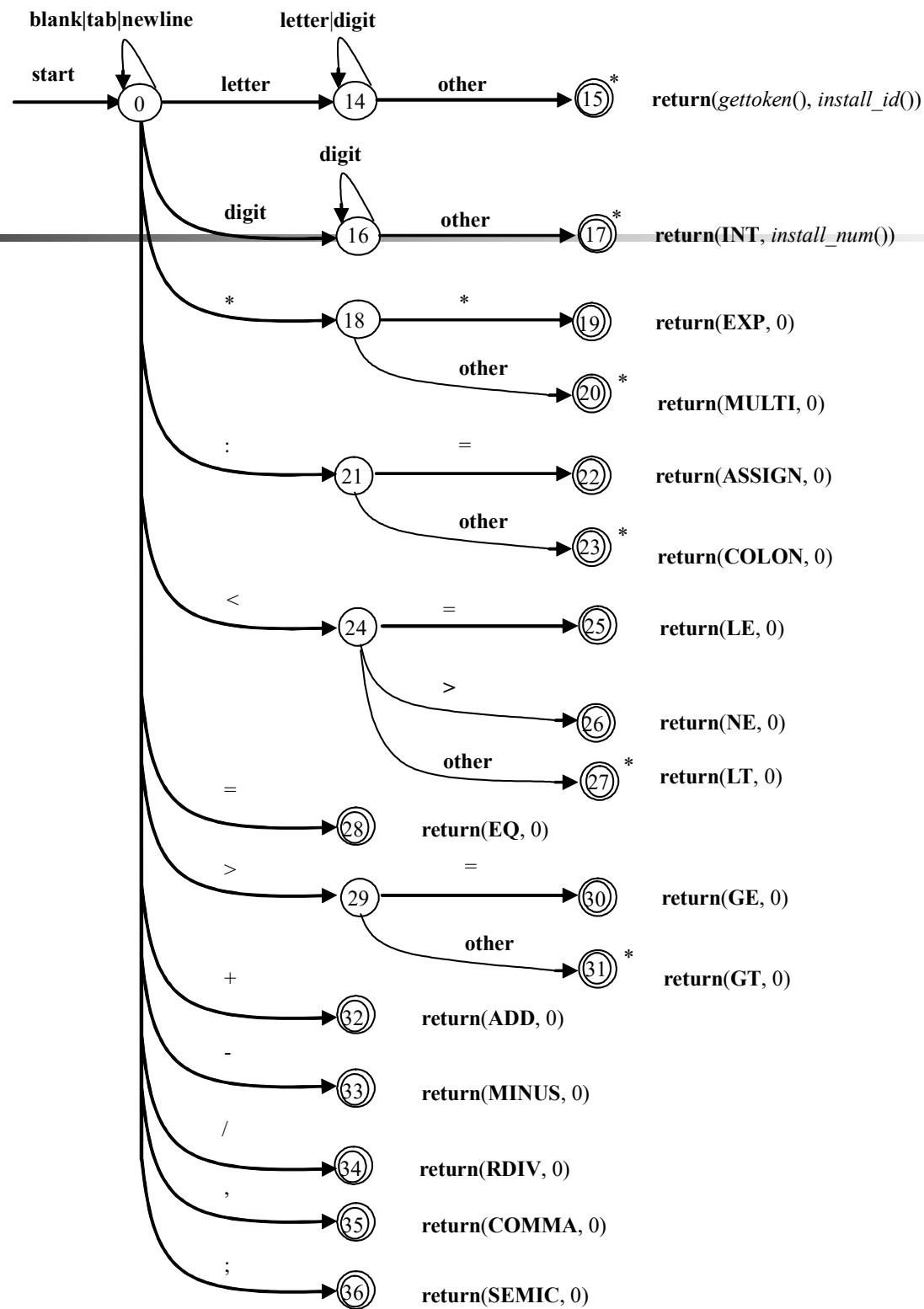
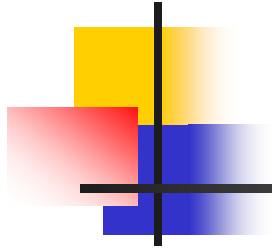


图3.14 识别C语言不同进制无符号整数的状态转换图

3.3.2 单词识别的状态转换图表示

- $\langle \text{id} \rangle \rightarrow \text{letter } \langle \text{id_left} \rangle$
- $\langle \text{id_left} \rangle \rightarrow \epsilon \mid \text{letter } \langle \text{id_left} \rangle \mid \text{digit } \langle \text{id_left} \rangle$
- $\langle \text{int} \rangle \rightarrow \text{digit } \langle \text{int_left} \rangle$
- $\langle \text{int_left} \rangle \rightarrow \epsilon \mid \text{digit } \langle \text{int_left} \rangle$
- $\langle \text{assignment} \rangle \rightarrow :=$
- $\langle \text{relop} \rangle \rightarrow < \mid <= \mid = \mid < > \mid > \mid >=$
- $\langle \text{op} \rangle \rightarrow + \mid - \mid * \mid / \mid **$
- $\langle \text{delimiter} \rangle \rightarrow : \mid , \mid ;$





利用状态转换图识别单词

- (1) 从初始状态出发;**
- (2) 读入一个字符;**
- (3) 按当前字符转入下一状态;**
- (4) 重复 (2)-(3) 直到无法继续转移为止。**

利用状态转换图识别单词

- 在遇到读入的字符是单词的分界符时，若当前状态是终止状态，说明读入的字符组成了一个单词；否则，说明输入字符串 w 不符合语法规则。
- 如果从状态转换图的初始状态出发，分别沿着所有可能的路径到达终止状态，并将每条路径上的标记依次连接起来，就得到了该状态转换图能够识别的所有单词。

状态可以看作当前的识别程度，例如开始状态和终止状态
- 读入字符 a 时从状态 A 转换到状态 B 正好对应着一步推导过程，即 $A \Rightarrow aB$ ，边与产生式($A \rightarrow aB$)相对应

利用状态转换图识别单词

- 在遇到读入的字符是单词的分界符时，若当前状态是终止状态，说明读入的字符组成了一个单词；否则，说明输入字符串 w 不符合语法规则。
- 如果从状态转换图的初始状态出发，分别沿着所有可能的路径到达终止状态，并将沿途的字符依次连接成字符串，则可以得到该图能够识别的所有单词，即该图能够识别的语言。

所以，状态转换图和正则文法是等价的。可以由正则文法来构造状态转换图，从而有效实现词法分析程序
- 读入字符 a 时从状态 A 转换到状态 B ，正好对应着一步推导过程，即 $A \Rightarrow aB$ ，边与产生式($A \rightarrow aB$)相对应



由正则文法构造状态转换图

- (1) 以每个语法变量(或其编号)为状态结点, 开始符号对应初始状态 S ;
- (2) 增设一个终止状态 T ;
- (3) 对 G 中每个形如 $A \rightarrow aB$ 的产生式, 从状态 A 到状态 B 画一条有向弧, 并标记为 a ;
- (4) 对 G 中每个形如 $A \rightarrow a$ 的产生式, 从状态 A 到终止状态 T 画一条标记为 a 的有向弧;
- (5) 对 G 中每个形如 $A \rightarrow \varepsilon$ 的产生式, 从状态 A 到终止状态 T 画一条标记为 any 的有向弧, any 表示 T 中任何符号。



由正则文法构造状态转换图

■ 无符号数的正则文法

$\langle \text{num} \rangle \rightarrow \text{digit} \langle \text{num_left} \rangle \mid . \langle \text{optional_fraction} \rangle \mid \text{digit}$

$\langle \text{num_left} \rangle \rightarrow \text{digit} \langle \text{num_left} \rangle \mid . \langle \text{dec_fraction} \rangle \mid \text{E}$
 $\langle \text{exponent_fraction} \rangle \mid . \mid \text{digit}$

$\langle \text{dec_fraction} \rangle \rightarrow \text{E} \langle \text{exponent_fraction} \rangle \mid \text{digit} \langle \text{dec_fraction} \rangle \mid$
 digit

$\langle \text{optional_fraction} \rangle \rightarrow \text{digit} \langle \text{dec_fraction} \rangle \mid \text{digit}$

$\langle \text{exponent_fraction} \rangle \rightarrow \text{digit} \langle \text{int_exponent_left} \rangle \mid +$
 $\langle \text{int_exponent} \rangle \mid - \langle \text{int_exponent} \rangle \mid \text{digit}$

$\langle \text{int_exponent} \rangle \rightarrow \text{digit} \langle \text{int_exponent_left} \rangle \mid \text{digit}$

$\langle \text{int_exponent_left} \rangle \rightarrow \text{digit} \langle \text{int_exponent_left} \rangle \mid \text{digit}$



由正则文法构造状态转换图

■ 无符号数的正则文法

- $\langle \text{num} \rangle : [0]$, $\langle \text{num_left} \rangle : [1]$, $\langle \text{dec_fraction} \rangle : [2]$,
 $\langle \text{optional_fraction} \rangle : [3]$, $\langle \text{exponent_fraction} \rangle : [4]$,
 $\langle \text{int_exponent} \rangle : [5]$, $\langle \text{int_exponent_left} \rangle : [6]$

$[0] \rightarrow \text{digit } [1] \mid . [3] \mid \text{digit}$

$[1] \rightarrow \text{digit } [1] \mid . [2] \mid \text{E } [4] \mid . \mid \text{digit}$

$[2] \rightarrow \text{E } [4] \mid \text{digit } [2] \mid \text{digit}$

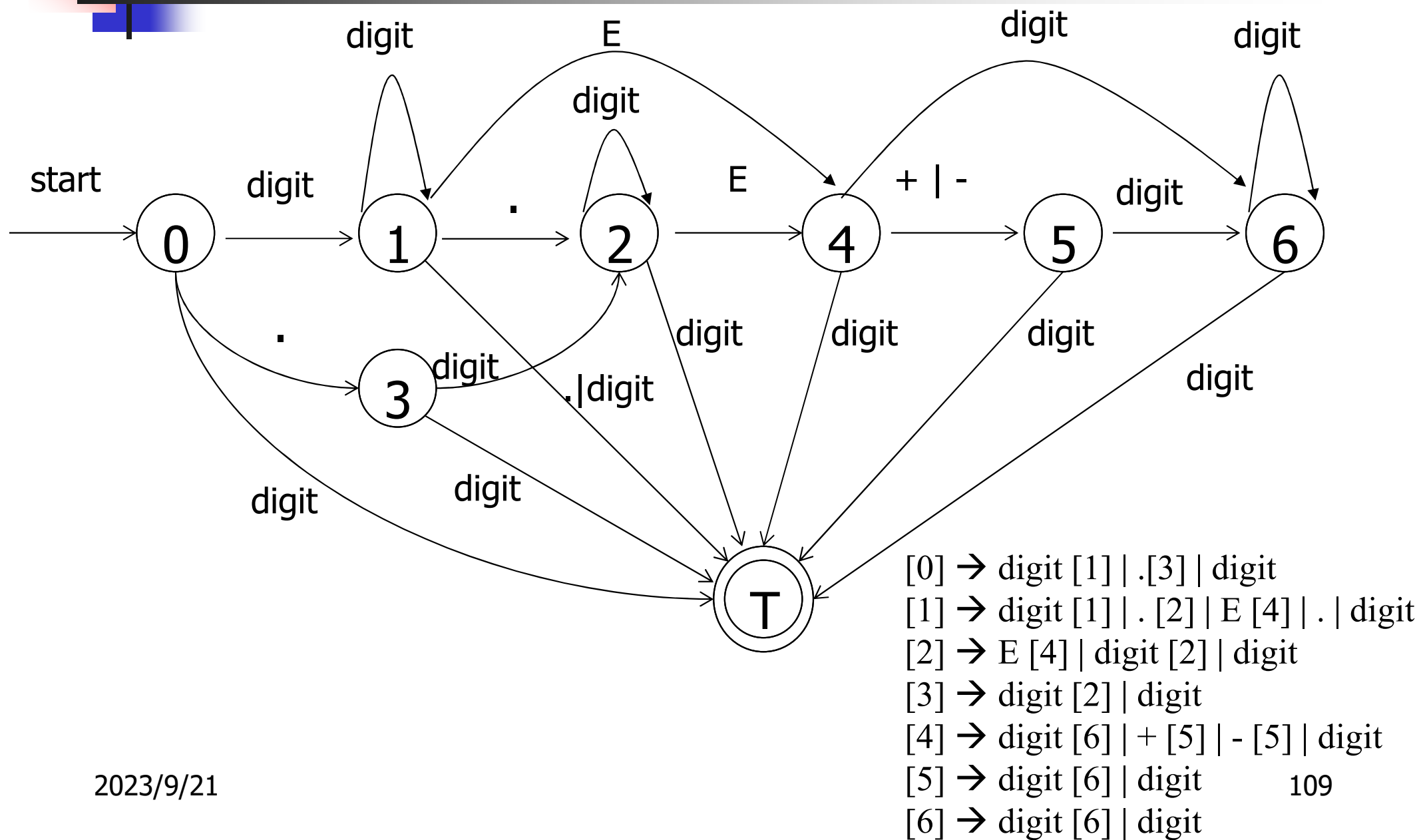
$[3] \rightarrow \text{digit } [2] \mid \text{digit}$

$[4] \rightarrow \text{digit } [6] \mid + [5] \mid - [5] \mid \text{digit}$

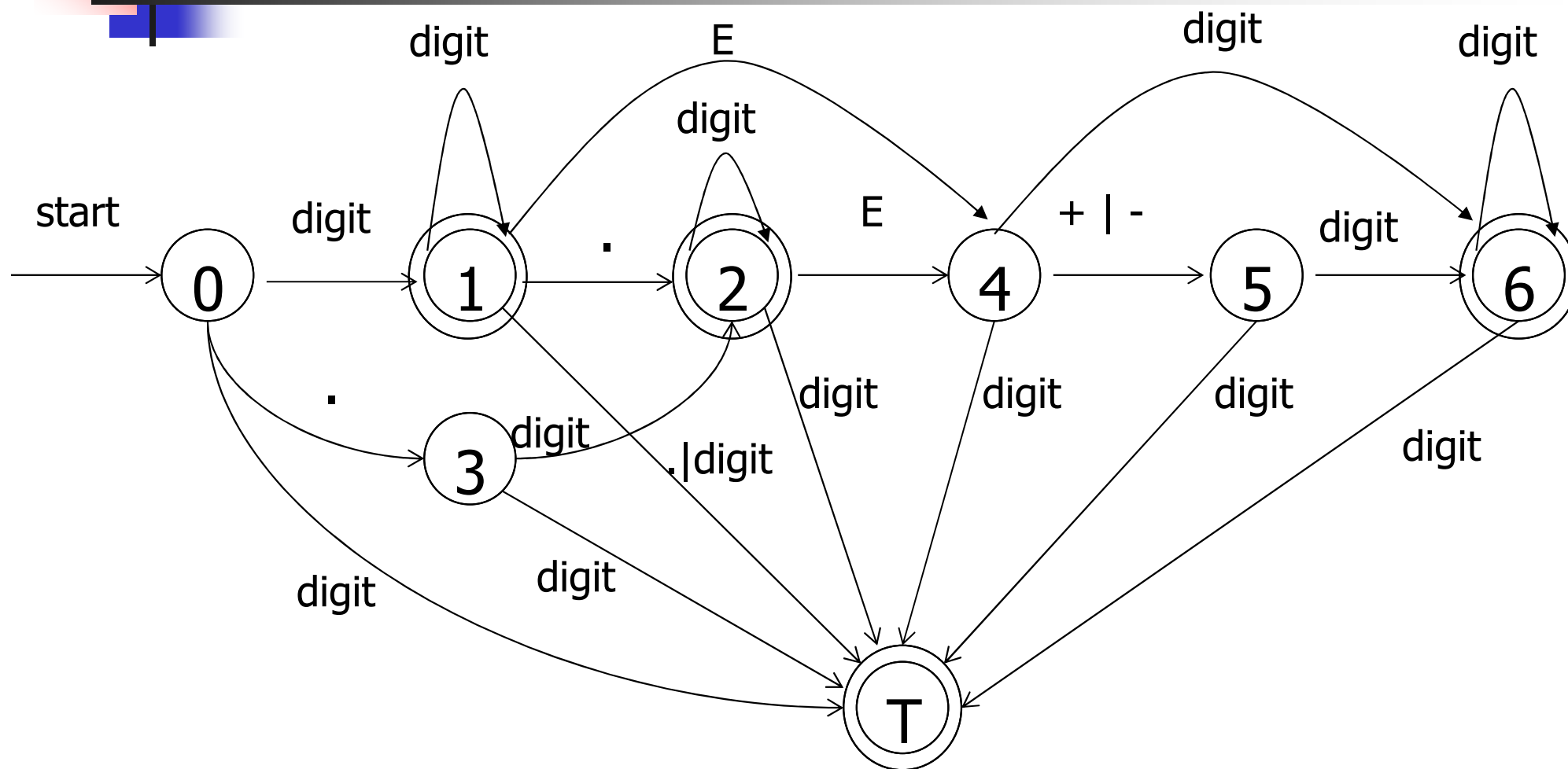
$[5] \rightarrow \text{digit } [6] \mid \text{digit}$

$[6] \rightarrow \text{digit } [6] \mid \text{digit}$

由正则文法构造状态转换图



由正则文法构造状态转换图





3.3.3 几种典型的单词识别问题

- **标识符的识别**
- **关键字的识别**
- **常数的识别**
- **算符和分界符的识别**
- **回退**



3.3.3 几种典型的单词识别问题

- **标识符的识别**

- **标识符的最大长度**
- **关键字是否作为保留字（禁止关键字作为标识符或标识符前缀）**
例如DO100I:表示关键字DO，整数100和标识符I
- **为方便，部分语言不将关键字作为保留字**

3.3.3 几种典型的单词识别问题

- 关键字的识别(以如下Fortran语句为例)

- - 1) DO100I=1,5
 - 2) IF(5.EQ.M)X=5
 - 3) DO100I=1.5
 - 4) IF(5)=10

超前扫描技术

3.3.3 几种典型的单词识别问题

■ 常数的识别

- 常数(直接量)
- 算术常数
- 逻辑常数
- 字符串常数

常数具有规定的值并在程序运行

十进制、十六进制、八进制和二进制的
数
不变
一个被
常数

真和假

数和复数

有效字符组成的任意字符串

3.3.3 几种典型的单词识别问题

■ 算符和分界符的识别

■ <、<=、<>、>、>=

■ /、÷、(、)

两个单词具有相同的前缀时，其词法分析器考虑超前搜索技术



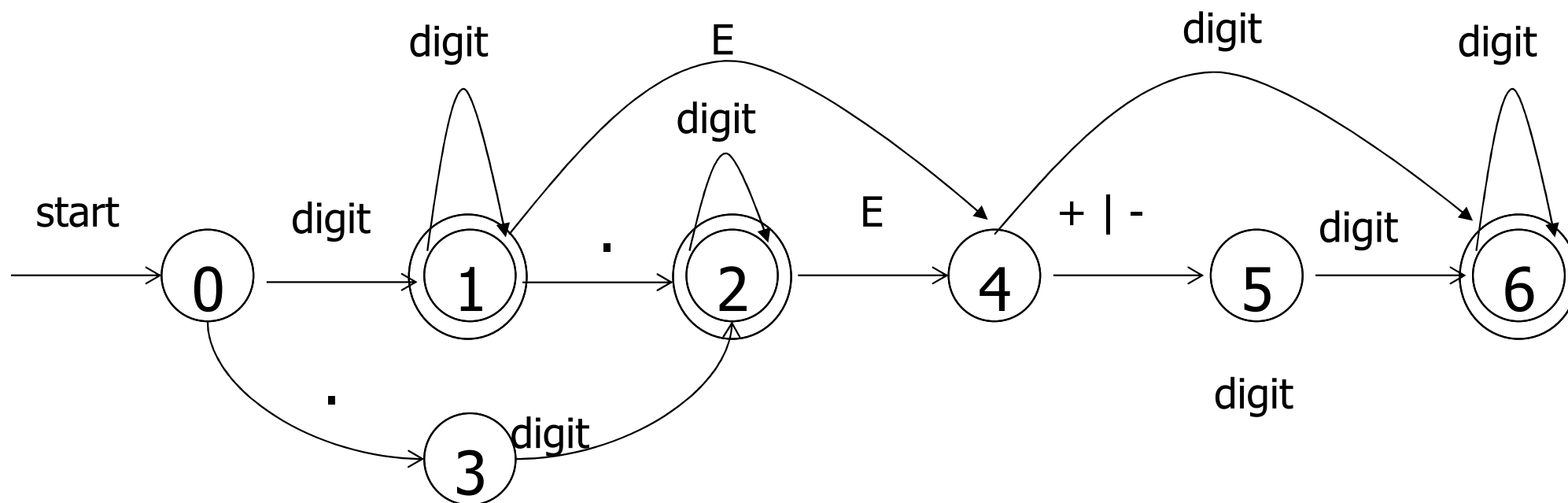
3.3.3 几种典型的单词识别问题

- **回退**

- **超前搜索一个或多个字符**
- **利用栈或队列来实现回退操作**

3.3.4 状态转换图的实现

- 本节考虑如何将状态转换图变换成识别单词的程序
- 状态转换图的实现也就是词法分析程序的实现





3.3.4 状态转换图的实现

- 如果将状态转换图看成是单词的**识别规则库**，则单词识别程序从当前状态(最初为初始状态)出发，读入一个输入字符后，将首先查询该规则库。

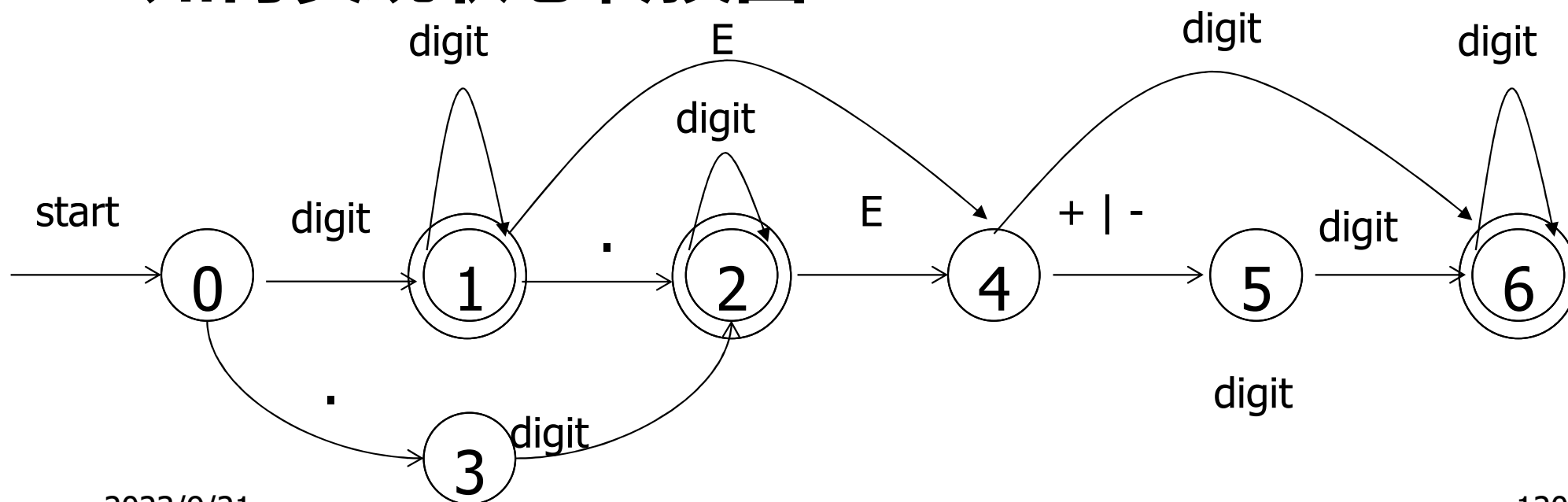
3.3.4 状态转换图的实现

■ 重复以下过程，直至到达某个终止状态。

- 如果从当前状态出发有一条边上标记了刚刚读入的输入字符，则单词识别程序将转入这条边所指向的那个状态，并再读入一个输入字符；
- 否则调用出错处理程序；
- 将从初始状态到该终止状态所经历的路径上的字符所组成的字符串作为一个单词输出；
- 并将当前状态重新置为开始状态，以便进行下一个单词的识别；
- 如果读完输入字符流后仍未进入某个终止状态则调用出错处理程序。

3.3.4 状态转换图的实现

- 状态转换图存储为计算机内部表示，单词识别程序设计成一个驱动程序，自动识别单词
- 状态转换图：一个边和顶点带有标记的有向图
- 如何实现状态转换图

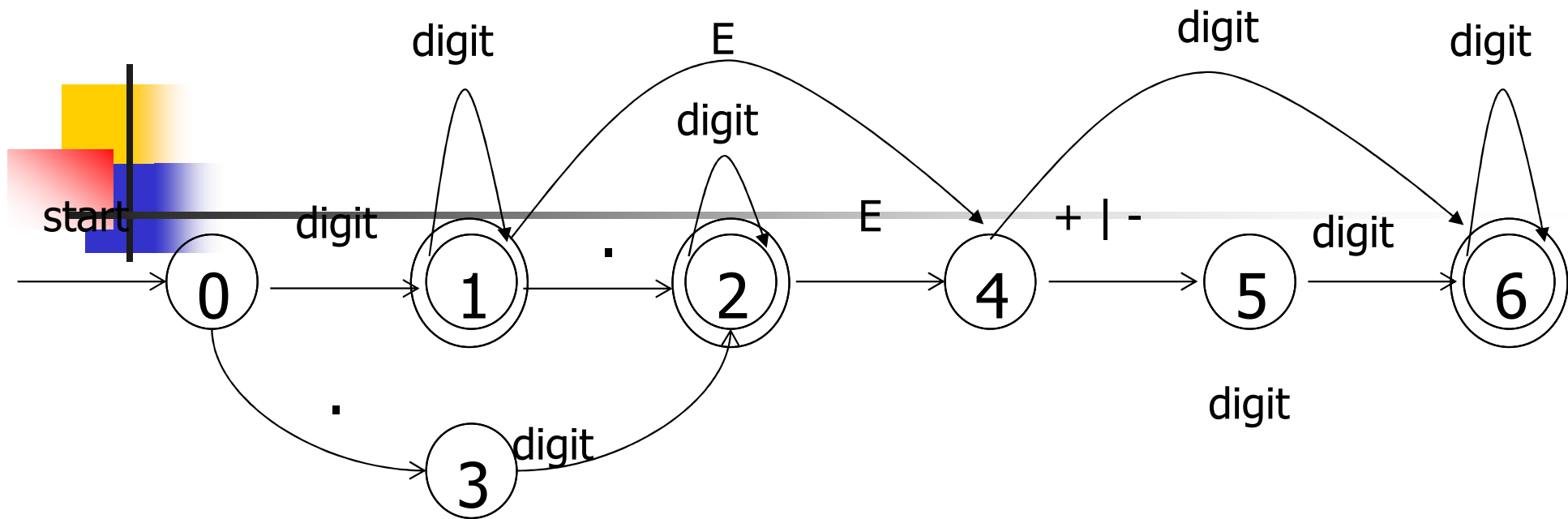




3.3.4 状态转换图的实现

■ 方法1：状态矩阵B

- 以状态转换图的各个状态为行
- 以可能的输入符号为列
- 给定状态 s_i ，输入符号 a_j ，那么 $B[i,j]$ 表示状态 s_i 读到字符 a_j 时转向的下一个状态（还包括此时应完成的语义动作）
- 优势：查询速度快
- 劣势：占用空间大，无用位置太多



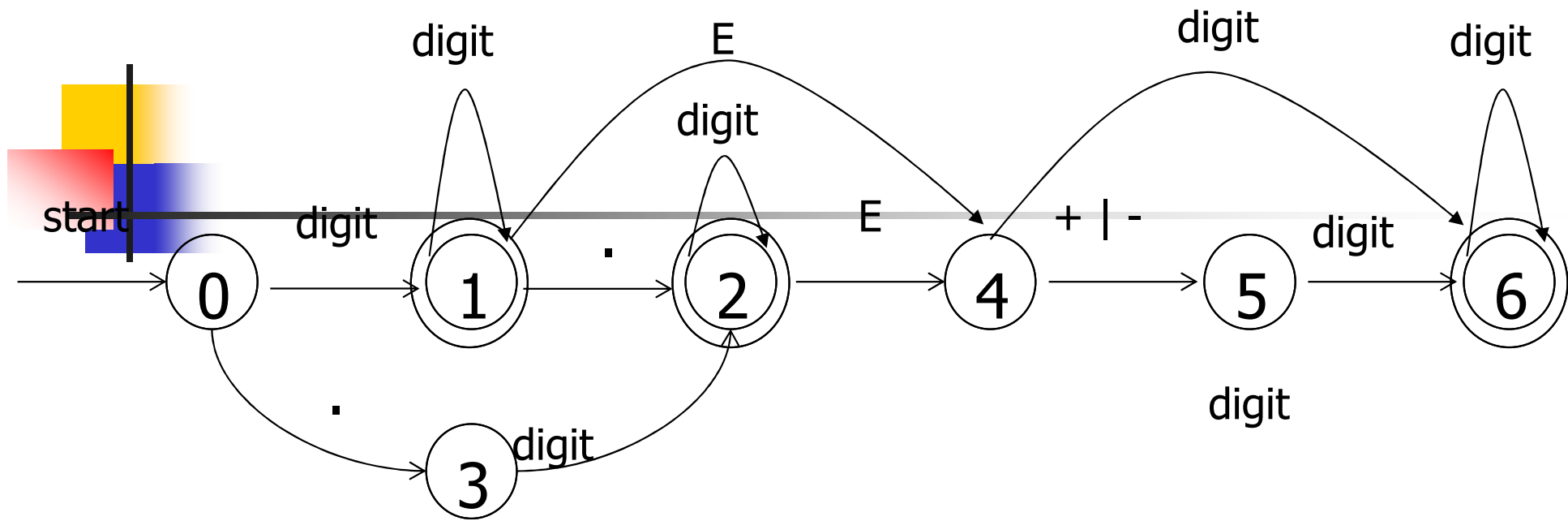
| | Digit | . | E | + | - |
|---|-------|------|------|------|------|
| 0 | 1 | 3 | NULL | NULL | NULL |
| 1 | 1 | 2 | 4 | NULL | NULL |
| 2 | 2 | NULL | 4 | NULL | NULL |
| 3 | 2 | NULL | NULL | NULL | NULL |
| 4 | 6 | NULL | NULL | 5 | 5 |
| 5 | 6 | NULL | NULL | NULL | NULL |
| 6 | 6 | NULL | NULL | NULL | NULL |



3.3.4 状态转换图的实现

■ 方法2：邻接表

- 以状态转换图的各个状态为头指针，维护一个元素的链表
- 每个元素包括两个域，第一个域是可能扫描到的符号，第二个域是当前状态转向的状态及需要执行的语义动作
- 第 i 个链表的元素 $[a/j]$ 表示在状态 i 读入符号 a 时转入到状态 j
- 优势：较节省空间
- 劣势：访问速度较慢



| | | | |
|---|-----------|-------|-------|
| 0 | Digit / 1 | ./ 3 | |
| 1 | Digit / 1 | ./ 2 | E / 4 |
| 2 | Digit / 2 | E / 4 | |
| 3 | Digit / 2 | | |
| 4 | Digit / 6 | + / 5 | - / 5 |
| 5 | Digit / 6 | | |
| 6 | Digit / 6 | | |



3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 状态矩阵方法和邻接链表方法各有其优点和缺点
- 一种更精妙的实现，整合状态矩阵和邻接链表的优势，而克服其劣势，即不但访问速度快，而且占用空间小

0

1

2

3

4

| | Digit | . | E | + | - |
|---|-------|------|------|------|------|
| 0 | 1 | 3 | NULL | NULL | NULL |
| 1 | 1 | 2 | 4 | NULL | NULL |
| 2 | 2 | NULL | 4 | NULL | NULL |
| 3 | 2 | NULL | NULL | NULL | NULL |
| 4 | 6 | NULL | NULL | 5 | 5 |
| 5 | 6 | NULL | NULL | NULL | NULL |
| 6 | 6 | NULL | NULL | NULL | NULL |

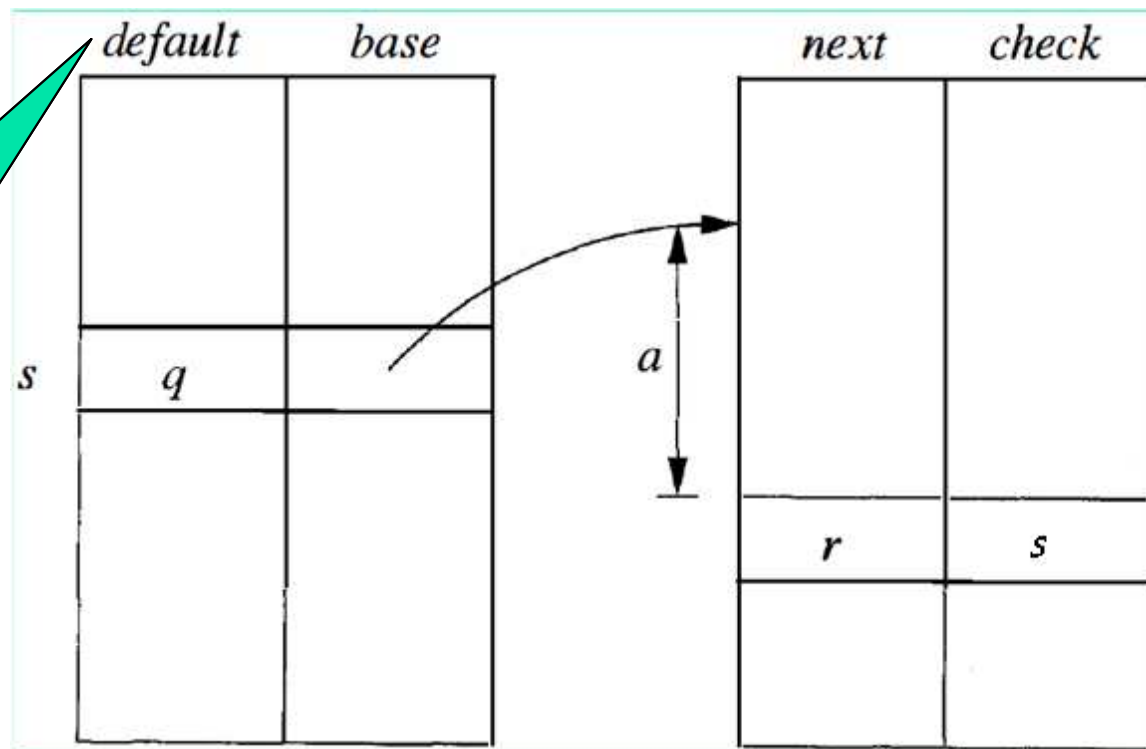
| | | | |
|---|-----------|-------|-------|
| 0 | Digit / 1 | ./ 3 | |
| 1 | Digit / 1 | ./ 2 | E / 4 |
| 2 | Digit / 2 | E / 4 | |
| 3 | Digit / 2 | | |
| 4 | Digit / 6 | + / 5 | - / 5 |
| 5 | Digit / 6 | | |
| 6 | Digit / 6 | | |

3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

以状态号为索引
每个状态在
default和base数
组中占有一个表项

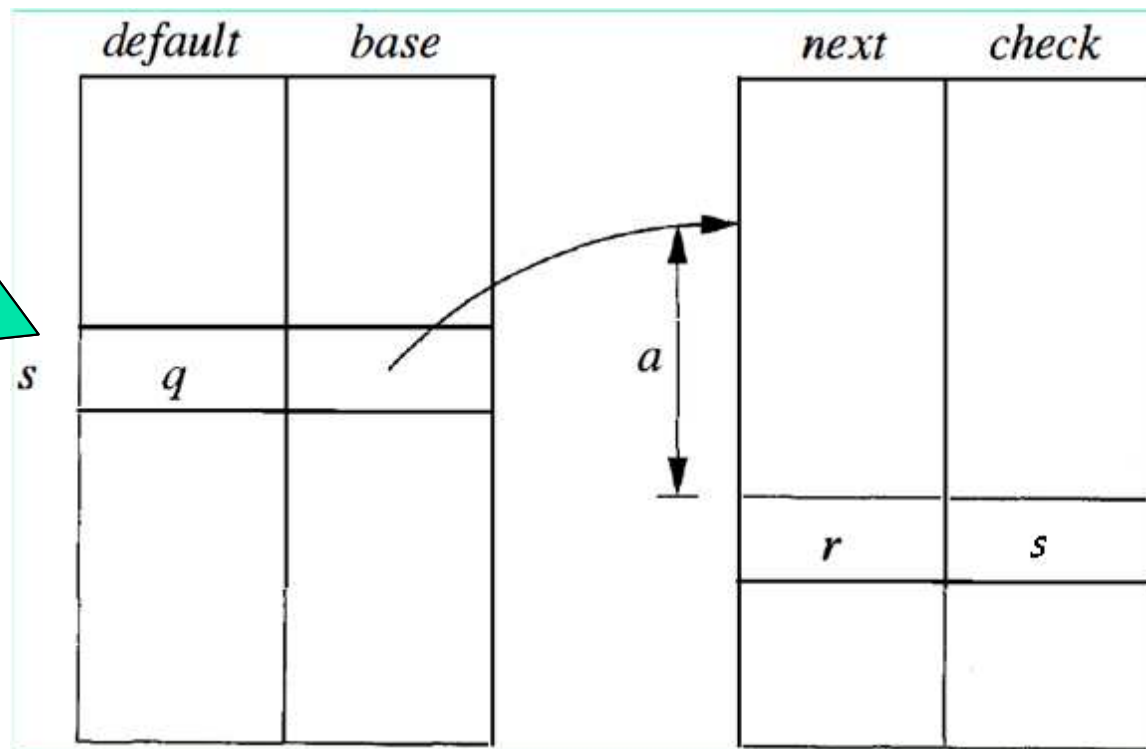


3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

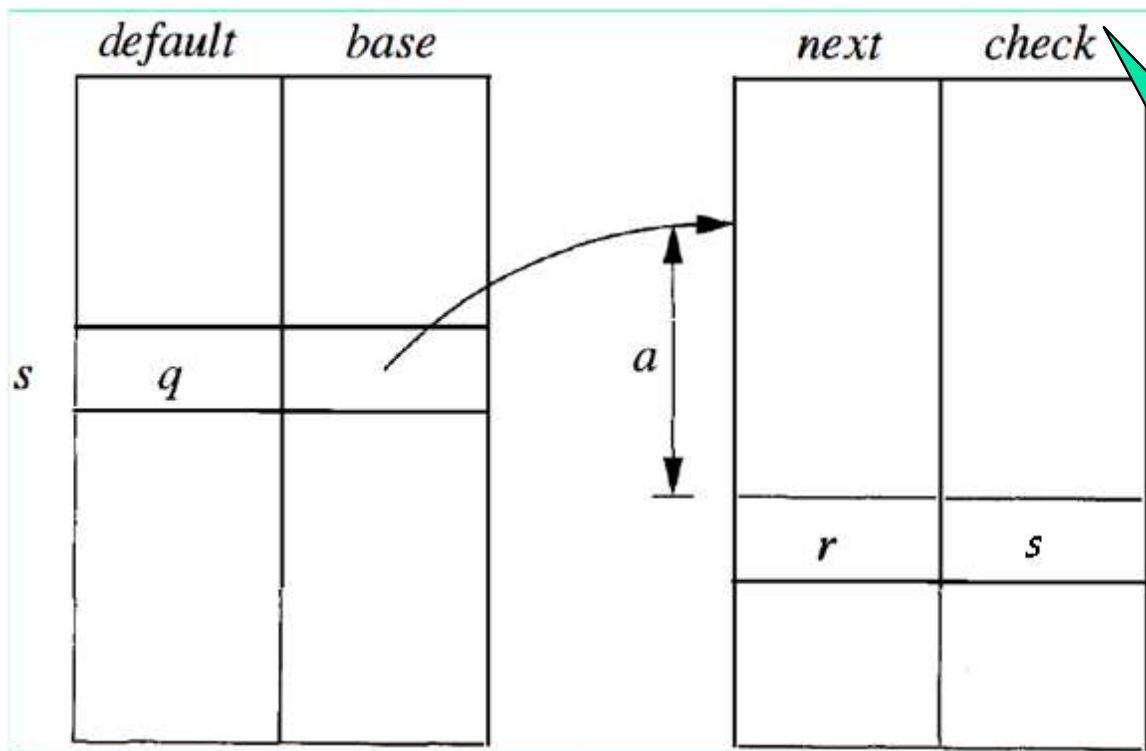
default[s]表示状态s的默认转向
base[s]表示状态s对应的转移向量在next和check数组中的起始位置



3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

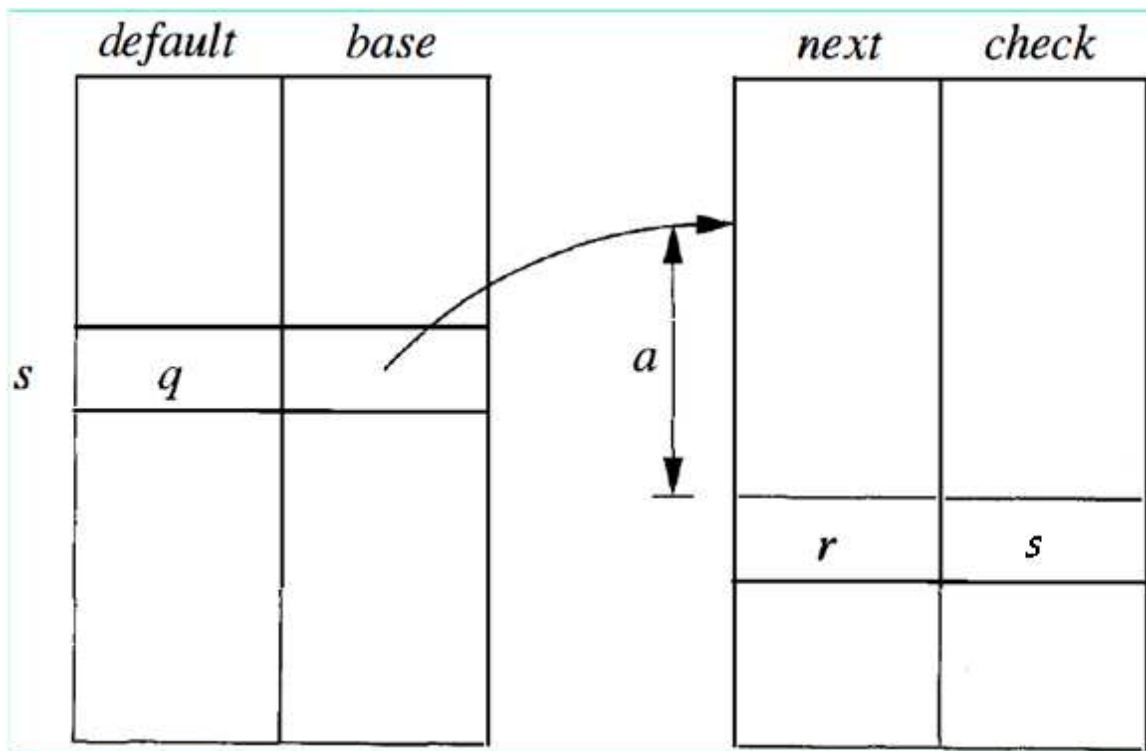


next和check数组以整数为索引，其大小与状态转换图里的边数有关

3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

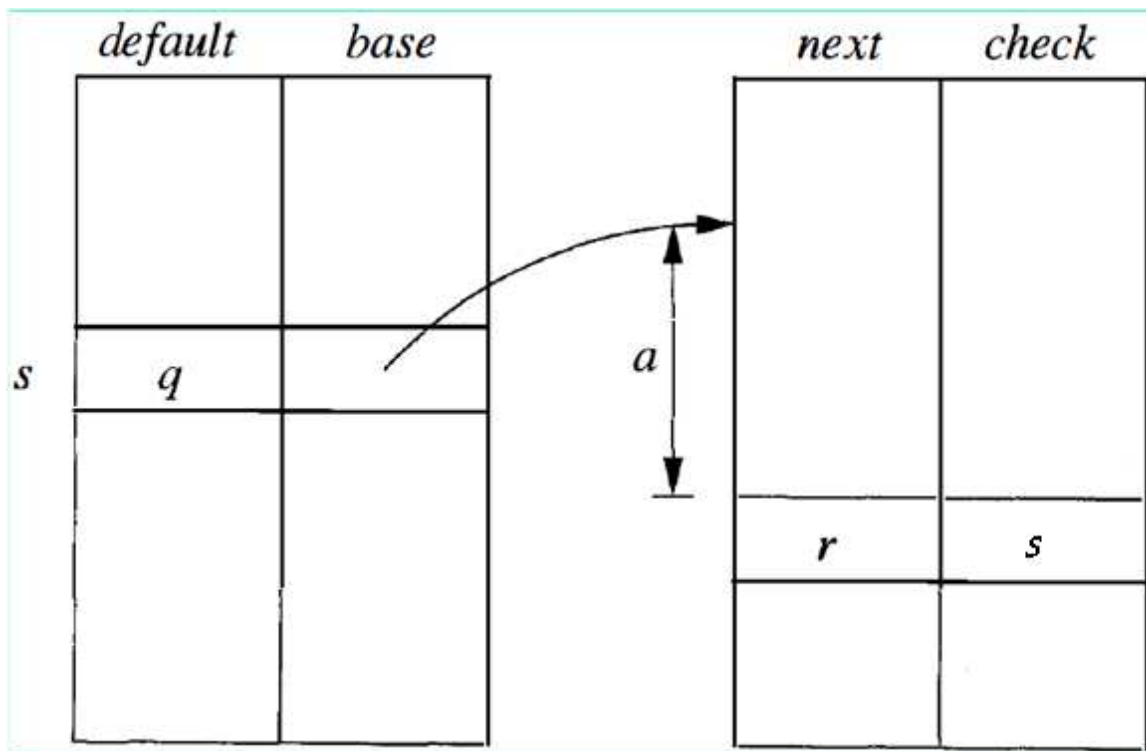


*next*用于保存每个状态*s*对应的转移向量，它们在*next*数组中从某个起始位置开始，以输入符号的整数编号为偏移量存放

3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

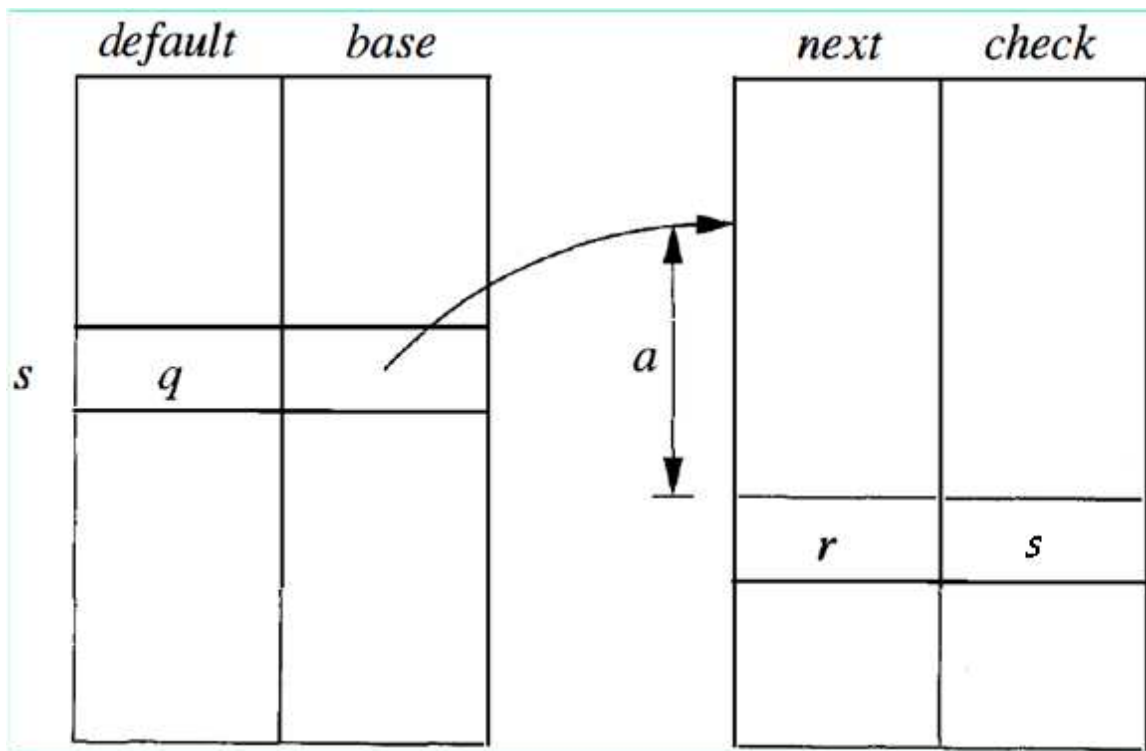


状态*s*在可能在某些输入符号上无转移，对应*next*单元为空，空单元可用于存放从其他状态转出的状态

3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

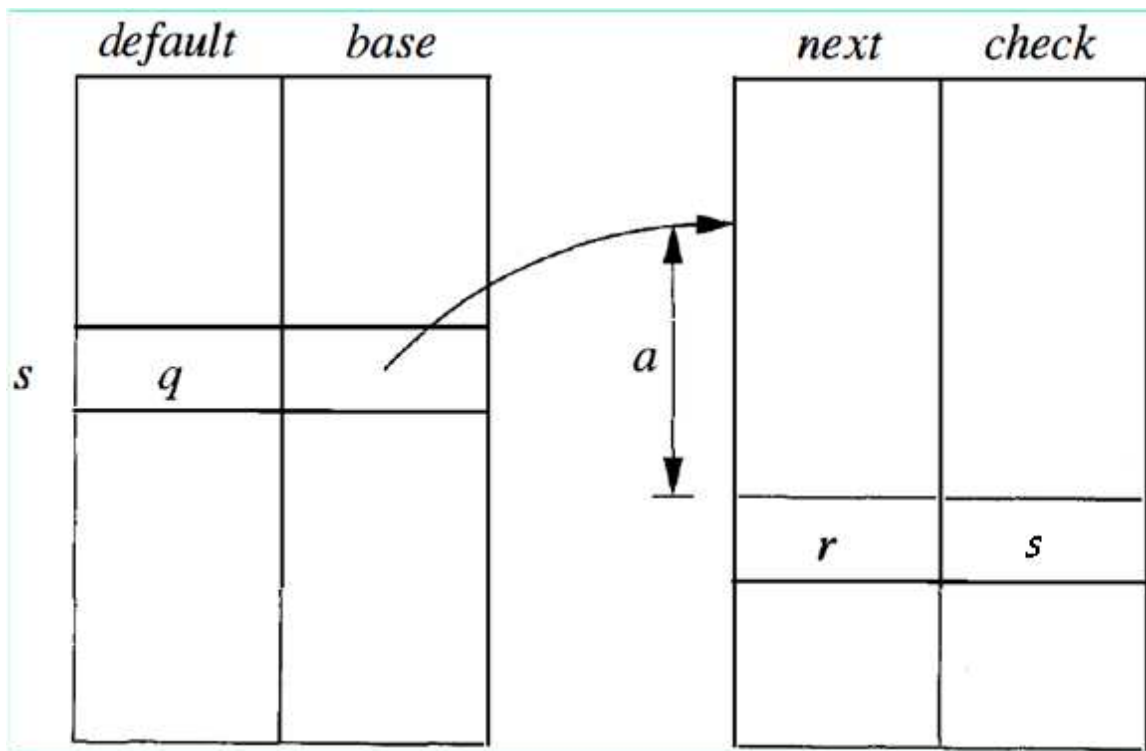


check数组标记
next中每个单元的
拥有者

3.3.4 状态转换图的实现

■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

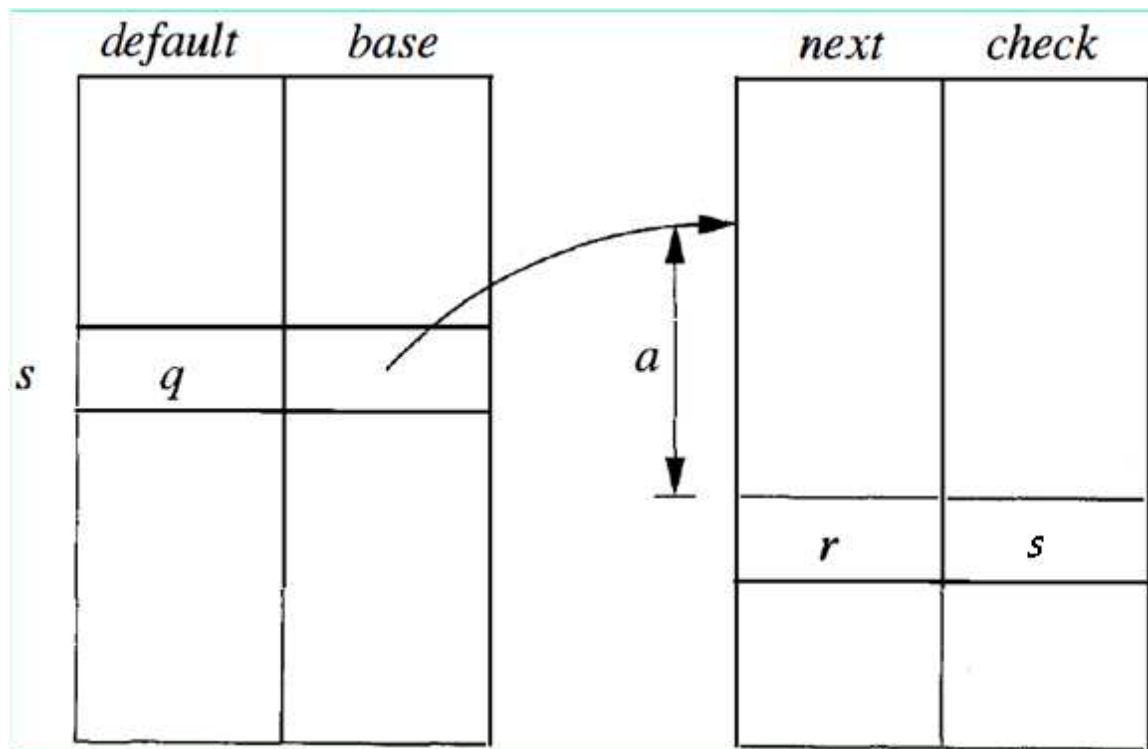


在状态*s*读入符号*a*时转入的状态为*r*，则
 $\text{next}[\text{base}[s] + a] = r$
 $\text{check}[\text{base}[s] + a] = s$

3.3.4 状态转换图的实现

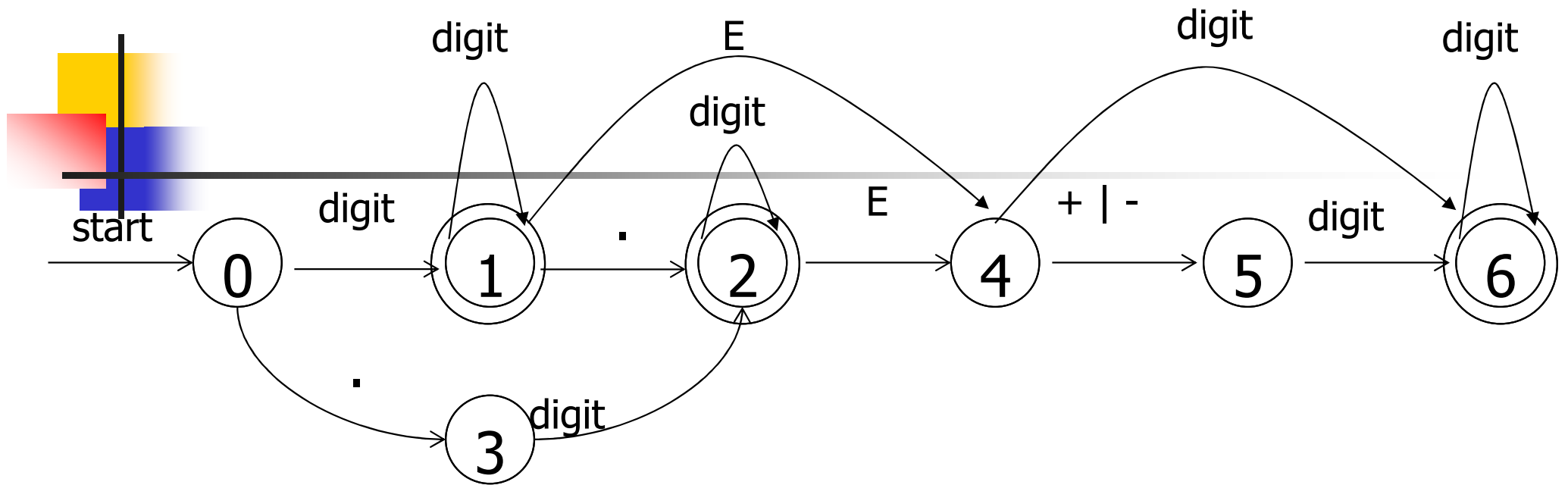
■ 如何实现状态转换图

- 四个数组组成的结构，既可以实现数据压缩存储，又可以实现元素的快速访问

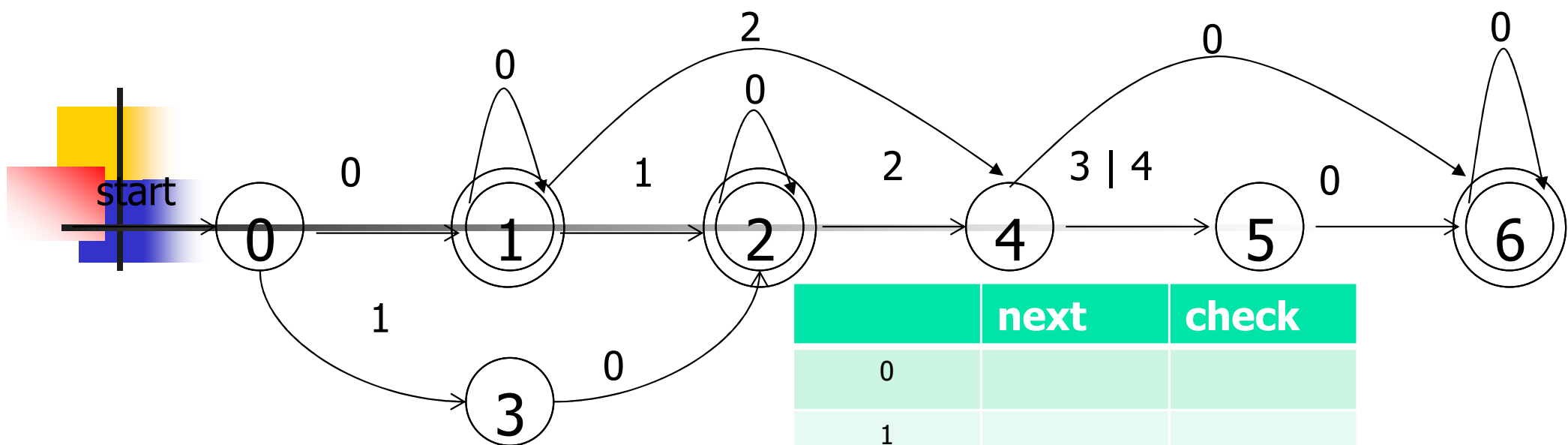


查找状态*s*在遇到字符*a*后进入的状态nextstate(*s*,*a*)

```
If(check[base[s]+a]==s)
    return next[base[s]+a]
Else
    进入默认转向
```



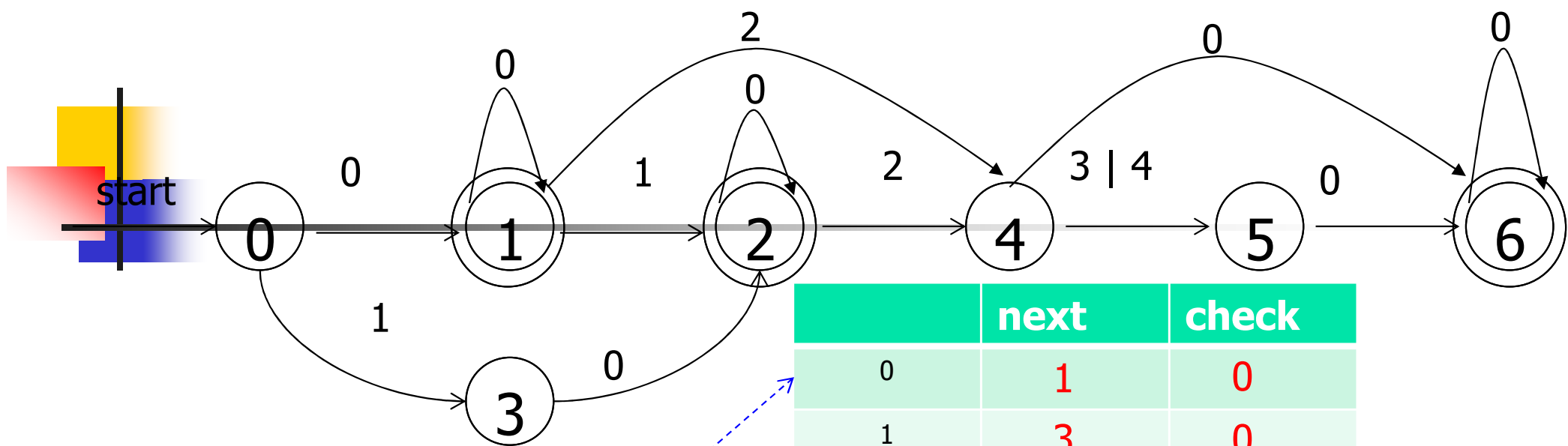
| | | |
|-------|---|---|
| digit | : | 0 |
| . | : | 1 |
| E | : | 2 |
| + | : | 3 |
| - | : | 4 |



| | next | check |
|----|------|-------|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | | |
| 12 | | |

| | default | base |
|---|---------|------|
| 0 | 0 | |
| 1 | 0 | |
| 2 | 0 | |
| 3 | 0 | |
| 4 | 0 | |
| 5 | 0 | |
| 6 | 0 | |

将**base**设置为使得新插入的特殊表项不会与已有表项发生冲突的最小值



| | default | base |
|---|---------|------|
| 0 | 0 | 0 |
| 1 | 0 | 2 |
| 2 | 0 | 5 |
| 3 | 0 | 6 |
| 4 | 0 | 8 |
| 5 | 0 | 9 |
| 6 | 0 | 10 |

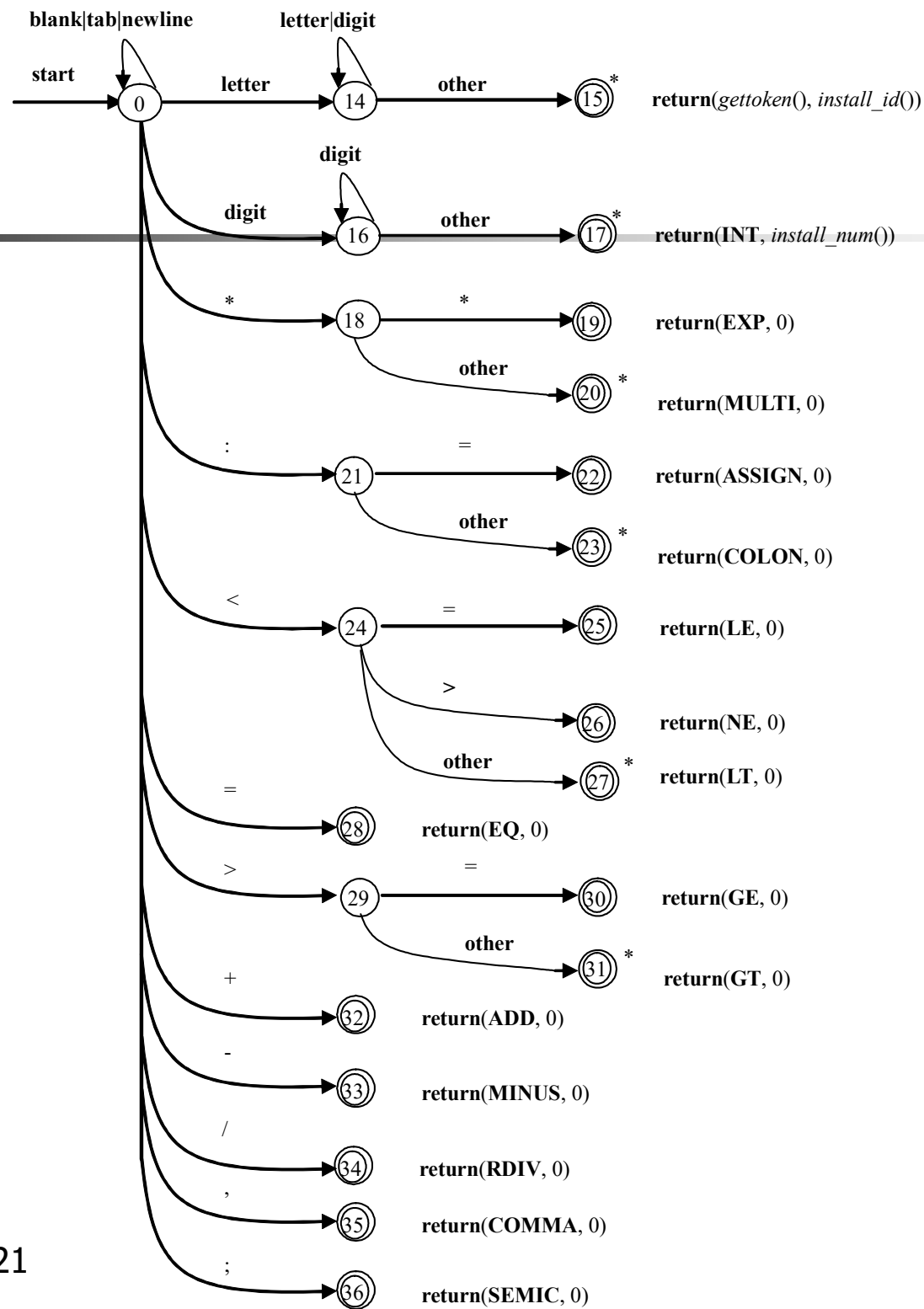
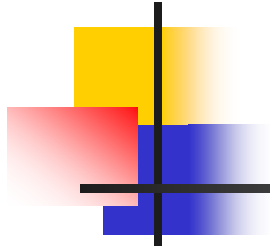
| | next | check |
|----|------|-------|
| 0 | 1 | 0 |
| 1 | 3 | 0 |
| 2 | 1 | 1 |
| 3 | 2 | 1 |
| 4 | 4 | 1 |
| 5 | 2 | 2 |
| 6 | 2 | 3 |
| 7 | 4 | 2 |
| 8 | 6 | 4 |
| 9 | 6 | 5 |
| 10 | 6 | 6 |
| 11 | 5 | 4 |
| 12 | 5 | 4 |

2023/9/21



3.3.5 词法分析程序的编写

- 状态转移图——教材P93图3.15
- 状态转移图的实现——教材P105图3.22
- 词法分析程序 *token_scan()*
 - 输入：字符流
 - 输出：
 - *symbol*.单词种别
 - *attr*.属性（全局变量）



数据结构与子例程

■ 数据结构

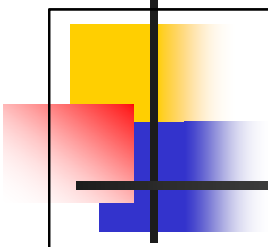
- *ch* 字符变量, 存放当前读入的输入字符
- *token* 字符串变量, 存放构成单词的字符串
- *symbol* 单词种别 (词法分析子程序的返回值)
- *attr* 属性 (全局变量)

■ 子例程

- *install_id(token)*: 将*token*存入符号表, 返回入口指针
- *getchar()*: 从输入缓冲区中读入一个字符放入*ch*
- *retract()*: 将向前指针回退一个字符, 将*ch*置为空白符
- *copytoken()*: 返回输入缓冲区中从开始指针
*lexeme_beginning*到向前指针*forward*之间的字符串
- *isLetter()* *isalpha()* *isalnum()*

图3.15的状态转换图的实现算法

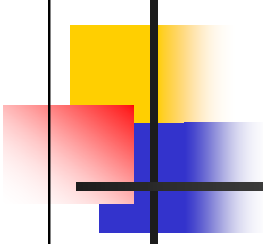
```
■ token token_scan()  
■ { char ch;  
■ char* token;  
■ ch = getchar();  
■ while (ch == blank || ch == tab || ch == newline) {  
■ ch = getchar();  
■   lexeme_beginning++;  
■ }  
■ if (isalpha(ch)) {ch = getchar();  
■   while (isalnum(ch))  
■     ch = getchar();  
■   retract(1);  
■   token = copytoken();  
■   return(gettoken(token), install_id(token));}
```



- **else**
- **if (*isdigit(ch)*) {**
- *ch = getchar();*
- **while (*isdigit(ch)*)**
- *ch = getchar();*
- *retract(1);*
- *token = copytoken();*
- **return(INT, *install_num(token)*);**
- **}**

- **else**
- **switch(*ch*)**
- **{**

- **case ‘*’: *ch = getchar();***
- **if(*ch* == ‘*’) return(EXP, 0);**
- **else {**
- *retract(1);*
- **return(MULTI, 0);**
- **} break;**



- **case ‘:’: *ch* = *getchar*();**
- **if(*ch* == ‘=’) return(ASSIGN, 0);**
- **else { *retract*(1); return(COLON, 0);**
- **} break;**

- **case ‘<’: *ch* = *getchar*();**
- **if(*ch* == ‘=’) return(LE, 0);**
- **else if(*ch* == ‘>’) return(NE, 0);**
- **else { *retract*(1); return(LT, 0);**
- **} break;**

- **case ‘=’: return(EQ, 0); break;**

- **case ‘>’: *ch* = *getchar*();**
- **if(*ch* == ‘=’) return(GE, 0);**
- **else { *retract*(1); return(GT, 0);**
- **} break;**

- **case ‘+’: return(PLUS, 0); break;**
- **case ‘-’: return(MINUS, 0); break;**
- **case ‘/’: return(RDIV, 0); break;**
- **case ‘,’: return(COMMA, 0); break;**
- **case ‘;’: return(SEMIC, 0); break;**

- **default: *error_handle*(); break;}return;}**



需要说明的问题

- **缓冲区预处理，超前搜索**
- **关键字的处理，符号表的实现**
- **Install_id(): 查找效率，算法的优化实现**
- **词法错误处理**
- **由于高级语言的词组成的集合为3型语言，所以，这里讨论的词法分析技术可以用于处理所有的3型语言。如：信息检索系统的查询语言、命令语言等**

3.4 词法分析程序的自动生成

Lex - A Lexical Analyzer Generator

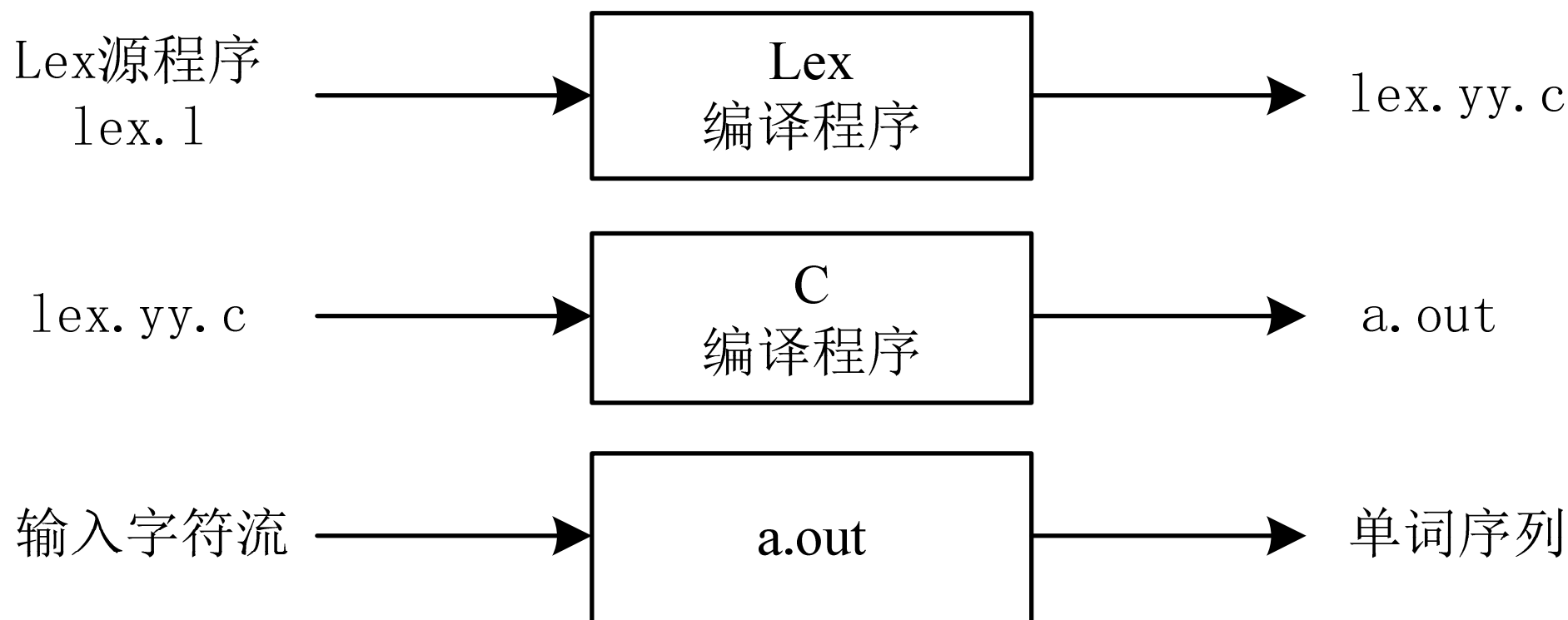


图3.23 利用Lex建立词法分析程序的过程

3.4.1 Lex源程序

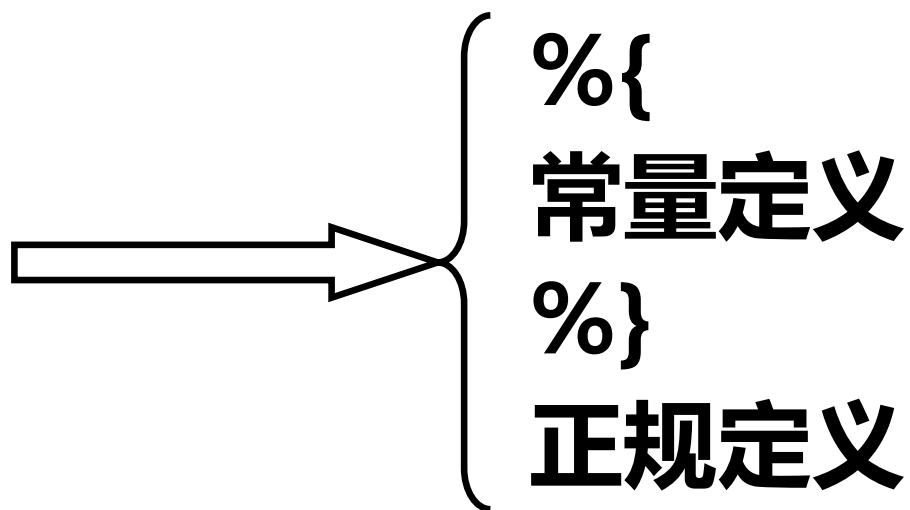
声明部分
(正规定义式)

%%

识别规则部分
(识别规则)

%%

辅助过程部分





3.4.1 Lex源程序

1、正规定义式

$\text{letter} \rightarrow A|B|C|\dots|Z|a|b|c|\dots|z$

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

$\text{identifier} \rightarrow \text{letter}(\text{letter}|\text{digit})^*$

$\text{integer} \rightarrow \text{digit}(\text{digit})^*$

2、识别规则

| 正规式 | 动作描述 |
|-----|------|
|-----|------|

| | |
|------------------|-----------------------|
| token_1 | $\{\text{action}_1\}$ |
|------------------|-----------------------|

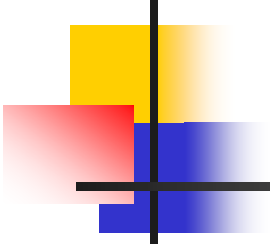
| | |
|------------------|-----------------------|
| token_2 | $\{\text{action}_2\}$ |
|------------------|-----------------------|

.....

| | |
|------------------|-----------------------|
| token_n | $\{\text{action}_n\}$ |
|------------------|-----------------------|



| | | | |
|-----------------------------------|------------------|-------------------|--|
| • %{ | | • delim | [\t\n] |
| • #include <stdio.h> | | • ws | [delim]+ |
| • #include "y.tab.h" | | • letter | [a-zA-Z] |
| • #define ID | 1 | • digit | [0-9] |
| • #define INT | 2 | • id | {letter}({letter} {digit})* |
| • #define EXP | 3 | • number | {digit}+ |
| • #define MULTI | 4 | • %% | |
| • #define COLON | 5 | • {ws} | ; |
| • #define EQ | 6 | • begin | return(BEGIN); |
| • #define NE | 7 | • end | return(END); |
| • #define LE | 8 | • if | return(IF); |
| • #define GE | 9 | • then | return(THEN); |
| • #define LT | 10 | • else | return(ELSE); |
| • #define GT | 11 | • do | return(DO); |
| • #define PLUS | 12 | • program | return(PROGRAM); |
| • #define MINUS | 13 | • {id} | {yyval = install_id(); return(ID);} |
| • #define RDIV | 14 | • {number} | {yyval = install_num(); |
| • #define COMMA | 15 | | return(INT);} |
| • #define SEMIC | 16 | | |
| • #define RELOP | 17 | | |
| • #define ASSGIN | 18 | | |
| • int line_no = 1; %{ | 2023/9/21 | | |



```

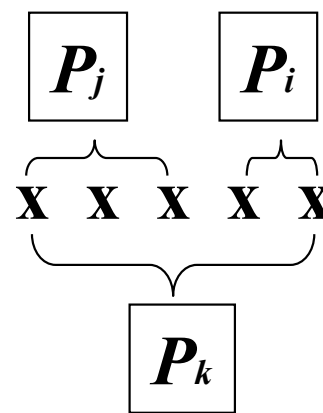
■ "<"           {yyval =LT; return(RELOP);}
■ "<="         {yyval =LE; return(RELOP);}
■ "=="          {yyval =EQ; return(RELOP);}
■ ">"           {yyval =GT; return(RELOP);}
■ ">="         {yyval =GE; return(RELOP);}
■ "<>"          {yyval =NE; return(RELOP);}
■ "+"           return(PLUS);
■ "-"           return(MINUS);
■ "*"           return(MULTI);
■ "/"           return(RDIV);
■ "**"           return(EXP);
■ ":"           return(COLON);
■ ":="          return(ASSGIN);
■ ","           return(COMMA);
■ ";"           return(SEMIC);
■ "\n"          line_no++;
■ "."           { fprintf (stderr, "'%c' (0%o): illegal charcter at
line
                %d\n", yytext[0], yytext[0], line_no); }
■ "%%"
■ install_id()
■ {.....}
■ install_num()
■ {.....}

```

LEX二义性问题的两条原则

1. 最长匹配原则

在识别单词过程中，有一字符串
根据最长匹配原则，应识别为这是
一个符合 P_k 规则的单词，
而不是 P_j 和 P_i 规则的单词。



2. 优先匹配原则

如果有一字符串有两条规则可以同时匹配时，那么用规则
序列中位于前面的规则相匹配，所以排列在最前面的规则
优先权最高。

3.4.2 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。

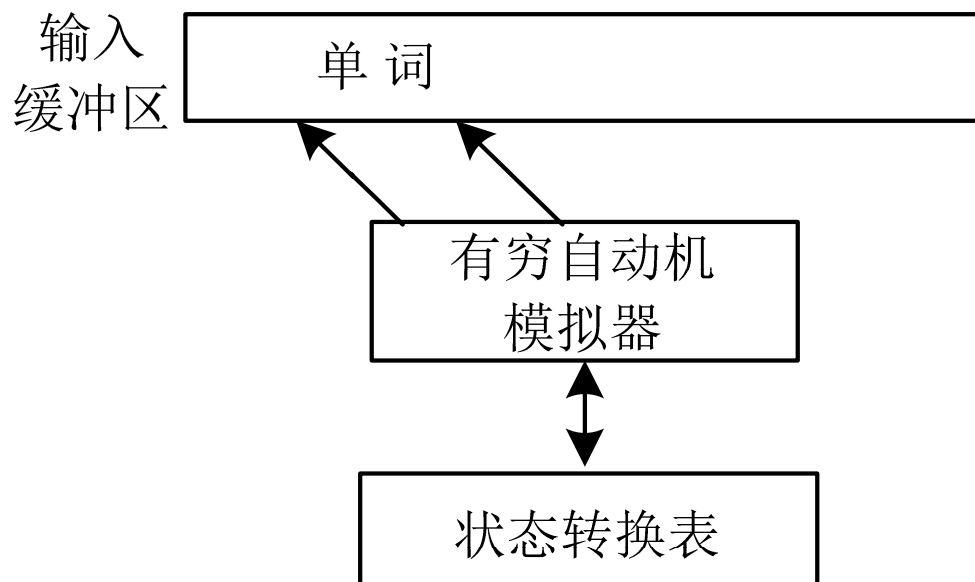


图 3.24 Lex生成的词法分析器结构

Lex的功能是根据Lex源程序生成状态转换矩阵和控制程序



三点说明

- 1) 以上是Lex的构造原理，虽然是原理性的，但据此就不难将Lex构造出来。**
- 2) 所构造出来的Lex是一个通用的工具，用它可以生成各种语言的词法分析程序，只需要根据不同的语言书写不同的LEX源文件就可以了。**
- 3) Lex不但能自动生成词法分析器，而且也可以产生多种模式识别器及文本编辑程序等**



本章小结

- **词法分析器接收表示源程序的“平滑字符流”，输出与之等价的单词序列；**
- **单词被分成多个种类，并被表示成(种别，属性值)的二元组形式；**
- **为了提高效率，词法分析器使用缓冲技术，而且在将字符流读入缓冲区时，是经过剔除注解、无用空白符等预处理后的结果；**



本章小结

- **单词的识别相当于正则语言的识别；**
- **词法的等价描述形式有正则文法、有穷状态自动机、正则表达式，其中有穷状态自动机可以用状态转换图表示；**
- **实现词法分析器时，状态转换图是一个很好的设计工具，根据该图，容易构造出相应的分析程序；**
- **使用恰当的形式化描述，可以实现词法分析器的自动生成，Lex就是一种自动生成工具。**