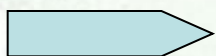


第2章 词法分析 (scanner)

(1) 扫描程序 (scanner)

源程序的字符 → 记号 (token)

记号 (token) = 自然语言如英语的单词



扫描 = 拼写

例如

$a[index] = 4 + 2$

记号:

a 标识符

[左括号

index 标识符

] 右括号

= 赋值

4 数字

+ 加号

2 数字

主控代码

```
switch(ch)
{
    case 字母:
    case 数字:
    case '[':
    case '=':
    case ']':
    case '+':
    ...
}
```

注意以下类似的问题

- C++源程序中以 / 开头的符号有：
- // 、 /=、 /*..... */ 、 /

主控代码

```
switch(ch)
```

```
{
```

```
...
```

```
case '/':
```

```
...
```

```
}
```


字母开头的单词

```
switch(ch)
```

```
{
```

```
    case 字母:
```

```
    ...
```

```
}
```

- 关键字与标识符
- 如何区分?

查找方法

- 二分查找
- 二叉排序树
- 字典树
- 散列 (Hash)

散列查找方法

- 散列函数
- 冲突解决方法

散列函数的构建方法

- 字母ASCII
- 字母序号

词法分析的理论

如何需要经常构造词法分析程序？

工具

自动生成工具 → 机器？

输入 → 自动生成工具 → 词法分析程序

反映单词组成的表达式

输入如何表示？

→即如何反映单词的组成形式

正则表达式

→ 用数学抽象的方法来表示单词串的组成

正则表达式的缺点：

正则表达式太抽象，不利于理解，也不利于代码的编制。

→ 有穷状态机器或称有穷自动机

即正则表达式所表达单词组成的识别算法。

2.2 正则表达式

- 正则表达式表示字符串的组成。
- 正则表达式 r 由它所匹配的串集来定义。
- 语言 $L(r)$ ——“串的集合”，
- 串的组成部分——字母表—— Σ 。

2.2.1 正则表达式的定义

1) 基本正则表达式

- 字母表中的单个字符且自身匹配。
- a 是字母表 Σ 中的任一字符，则正则表达式 a 通过书写 $L(a) = \{a\}$ 来匹配 a 字符。
- 空串就是不包含任何字符的串。
- 空串用 ε (epsilon) 来表示

2) 正则表达式运算

- ① 选择，用元字符| (竖线)表示。
- ② 连结，由并置表示(不用元字符)。
- ③ 重复或“闭包”，由元字符*表示。

3) 选择运算

- $r|s$ ——即可匹配被 r 或 s 匹配的任意串。
- $L(a|\varepsilon) = \{ a, \varepsilon \}$ 。
- $L(a|b|c|d) = \{ a, b, c, d \}$ 。
- $a|b|\dots|z$ ——表示匹配 $a\sim z$ 的任何小写字母。

4) 连结运算

- 正则表达式 ab —— 只匹配 ab ,
- 正则表达式 $(a|b)c$ —— 则匹配串 ac 和 bc 。
- $\mathcal{S}_1 = \{aa, b\}, \mathcal{S}_2 = \{a, bb\}$, 则
 $\mathcal{S}_1 \mathcal{S}_2 = \{aaa, aabb, ba, bbb\}$
- $L((a|b)c) = L(a|b)L(c) = \{a, b\}\{c\} = \{ac, bc\}$ 。

5) 重复——Kleene闭包 (Kleene closure)

- r^* —— 其中 r 是一个正则表达式。
- 正则表达式 r^* —— 匹配串的任意有穷连结，每个连结均匹配 r 。
- a^* —— ε 、 a 、 aa 、 $aaa \dots$
- $(a|bb)^*$ —— ε 、 a 、 bb 、 aa 、 abb 、 bba 、 $bbbb$ 、 aaa 、 $aabb$ ，.....

6) 运算符优先级和括号的使用

- 正则表达式 $a|b^*$ 如何理解?
- $(a|b)^*$? 还是 $a|(b^*)$?
- 运算优先级: *优先权最高, 连结其次, | 最末。
- 因此, $a|bc^*$ 就可解释为 $a|(b(c^*))$,

如何组织单词对应的正则表达式

- 例1:

一个或多个数字序列对应的一个正则表达式?

$(0|1|2|\dots|9)(0|1|2|\dots|9)^*$

缺点:书写太长,重复量大

解决方法:命名

digit = $0|1|2|\dots|9$

*digit digit**

• 例2.1 在仅由字母表中的3个字符组成的简单字母表 $\Sigma=\{a, b, c\}$ 中，考虑在这个字母表上的仅包括一个 b 的所有串的集合

• 解答：即 b 的两边可以有任意多个其它符号

$$(a|c)^*b(a|c)^*$$

- 例2.2 在与上面相同的字母表中，如果集合是包括了最多一个***b***的所有串，

- 解答：

- (1) 可理解为没有***b***或只有一个***b***。

$$(a|c)^*|(a|c)^*b(a|c)^*$$

- (2) 也可理解为：允许***b***又允许空串在重复的***a***或***c***之间出现，于是有另一个解：

$$(a|c)^*(b|\epsilon)(a|c)^*$$

- 注意：不同的正则表达式可生成相同的语言。

- 例2.3 在字母表 $\Sigma=\{a,b\}$ 上的串 S 的集合是由一个 b 及在其前后有相同数目的 a 组成:
- $S = \{ b, aba, aabaa, aaabaaa, \dots \} = \{ a^n b a^n \mid n \geq 0 \}$
- 正则表达式并不能描述这个集合

• 标识符的正则表达式

- 标识符必须由一个字母开头且只包含字母和数字。
- 正则表达式为：

letter = a | b | ... z | A | B | ... Z

digit = 0 | 1 | 2 | ... | 9

identifier = *letter* (*letter* | *digit*)*

缺点：书写繁琐，是否可以简化

• 整数的正则表达式

- 整数可以正整数（可带正号或不带）、负整数。
- $\text{digit} = 0|1|2|\dots|9$
- $\text{natural} = \text{digit digit}^*$
- $\text{signedNatural} = \text{natural} | +\text{natural} | -\text{natural}$

缺点:重复量大,是否可以简化

• 正则表达式的简化

- 缺点:书写繁琐

- →导致这个问题的核心是什么?

- 提供的运算太少了→扩充运算符号

1. 正闭包: 一个或多个重复

- r^* 表示允许 r 被重复 0 次或更多次。
- r^+ 表明 r 的一个或多个重复。
- 如: $(0|1)(0|1)^*$ 简化: $(0|1)^+$
- 如: $\text{natural} = \text{digit digit}^*$
简化: $\text{natural} = \text{digit}^+$

2. 字符范围表示运算符

- `a | b | ... | z` 来表示小写字母
- `0 | 1 | ... | 9` 来表示数字 复杂
- 用连字符——`[a-z]`是指所有小写字母，`[0-9]`则指数字。
- `a | b | c`可写成`[abc]`
- `[a-zA-Z]`代表所有的大小写字母。

3.表示可选的运算符

- 例如，整数的正则表达式为

$natural = [0-9]^+$

$signedNatural = natural|+natural|-natural$

复杂

- 引入元字符？
- $r?$ ——表示由 r 匹配的串是可选的
- 整数则可简化为：

$natural = [0-9]^+$

$signedNatural = (+|-)?natural$

4.表示任意字符的运算符

- 句号 “.” 表示任意字符匹配的典型元字符
- 至少有一个 b 的串对应正则表达式为：

$.^*b.^*$

5. 不在给定集合中的任意字符

- “ \sim ”，那么表示字母表中非 a 字符的正则表达式就是 $\sim a$ 。
- 非 a 、 b 及 c 表示为：
 $\sim(a|b|c)$

- 十进制数的正则表达式

- 十进制数可以是整数、浮点数、或带有指数的数(由 e 或 E 表示)的序列。

- 如: 123 3.14 -4.5 2.71E-2

- 正则式:

$\text{nat} = [0-9]^+$

$\text{signedNat} = (+|-)?\text{nat}$

$\text{number} = \text{signedNat}(".\text{nat})?(E \text{ signedNat})?$

• 注释的正则表达式

- 注释可有若干个不同的格式。

- 例如：

`{ this is a Pascal comment }`

`/* this is a C comment */`

- 或行注解，如：

`; this is a Scheme comment`

`-- this is an Ada comment`

单个分隔符的注释

如pascal注解: { this is a Pascal comment }

正则表达式: { (~ }) * }

两个分隔符的注释

C语言的注解: `/* this is a C comment */`

- `/*...(*/*/不同时期出现的任意长度串)...*/`
- `/* (~(*/*/))* */` **错**
- 缺陷: 正则表达式表达能力不强

• 二义性问题

- 例如：

- $<=$ \rightarrow 两种理解：

(小于号、等号) 和 小于等于号

- 解决方法：

- 最长子串原理 (principle of longest substring)

C++单词的二义性

- / // /* */ /= /
- < << <= <
- > >> >= >
- = == =
- * *= *
- - - -= -- ->
- + ++ += +
- 0 0 0x 0X
-

正则表达式的缺点

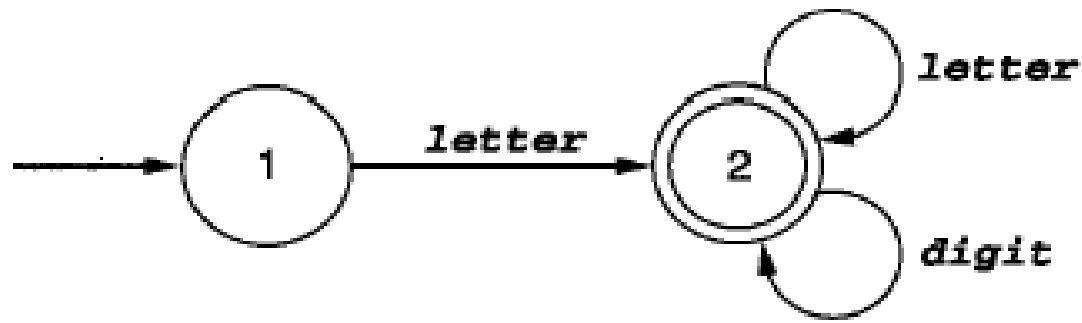
正则表达式太抽象，不利于理解，也不利于代码的编制。

→ 有穷状态机器或称有穷自动机

即正则表达式所表达单词组成的识别算法。

• 2.3 有穷自动机

- ——有穷状态的机器
 - ——是描述(或“机器”)特定类型算法的数学方法。
 - ——可用作描述在输入串中识别模式的过程
 - ——可用作构造扫描程序。
-
- 有穷自动机与正则表达式之间关系?
 - 例: `identifier=letter(letter|digit)*` 有穷自动机?



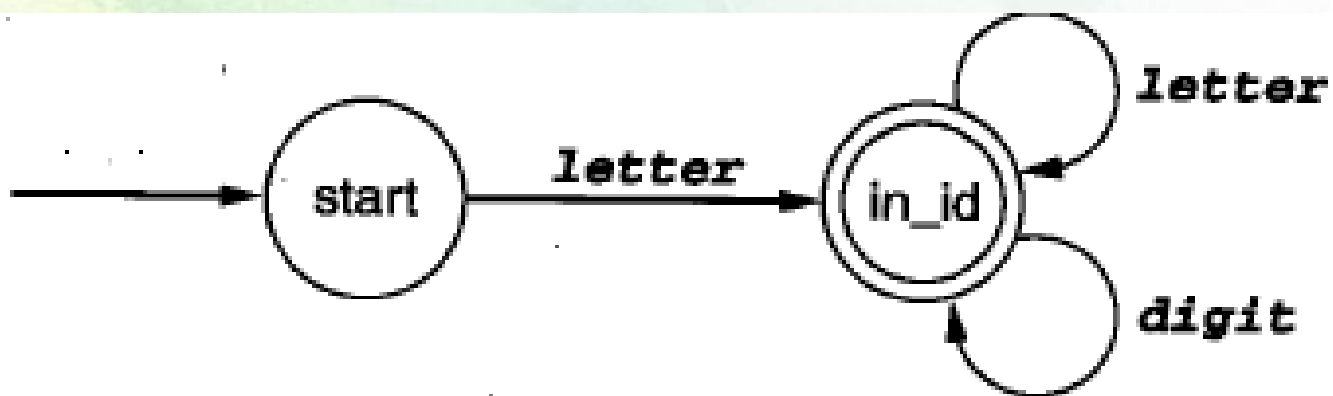
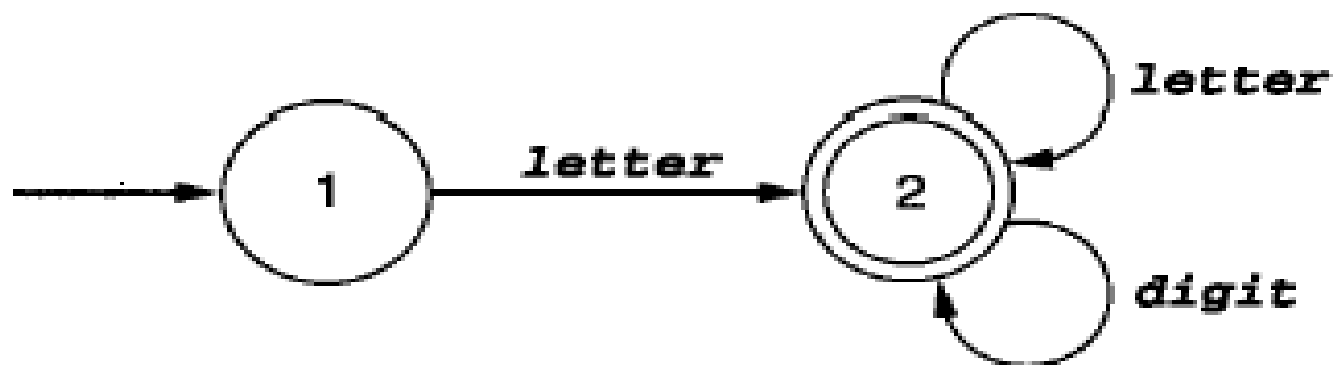
- 状态 (state)
- 初始状态 (start state)
- 接受状态 (accepting state)
- 转换 (transition)
- 识别过程:

如xtemp:

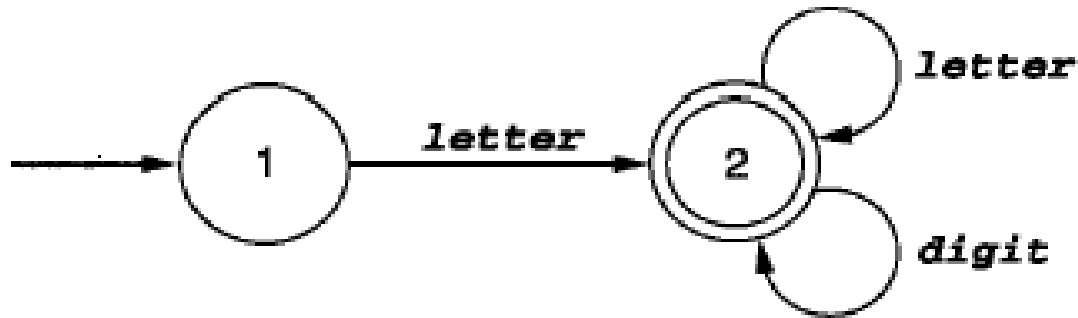
→1→2→2→2→2→2

- 方便地展示出算法过程，因此它对于有穷自动机的描述很有用处。

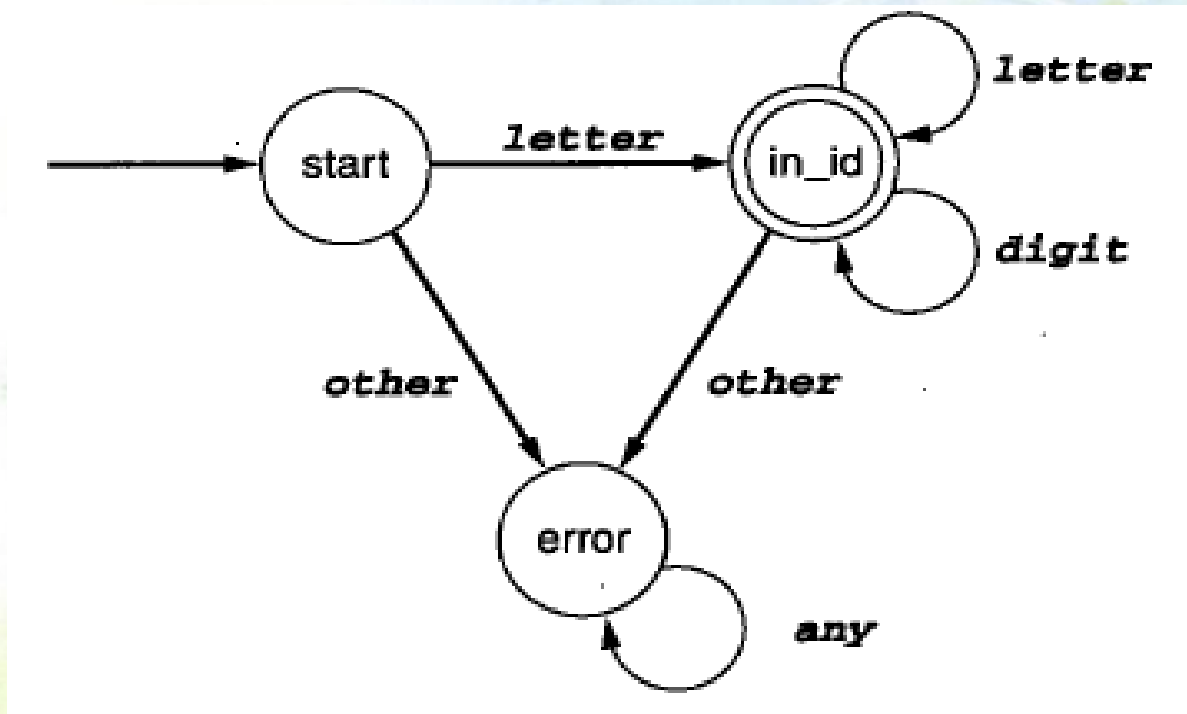
- 等价图



DFA图的疑惑

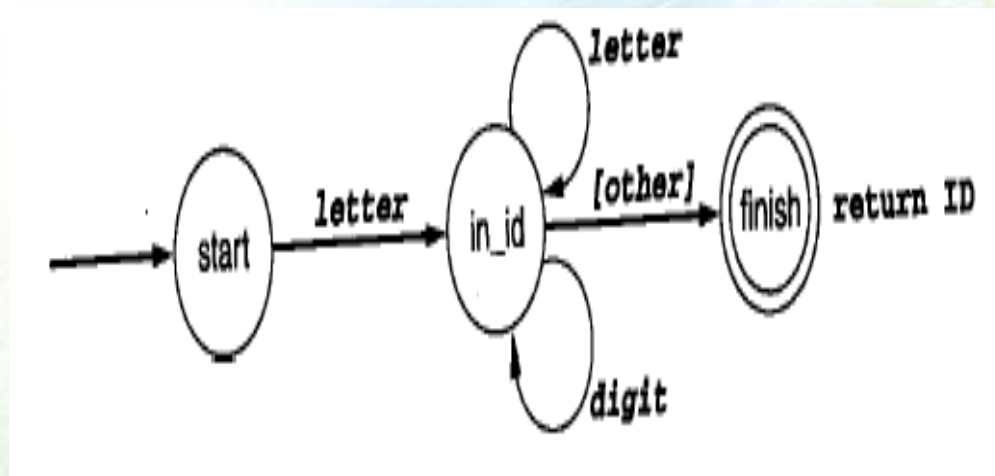


- 输入为: 1xtemp
- 出错转换(error transition)
 - 没画而假设它总是存在



又有新的疑惑:

如果输入的串为 `xtemp=1`; 则.....



- **解决方法：带有方括号**，表示应先行考虑分隔字符，也就是：应先将其返回到输入串并且不能丢掉。
- **——最长子串原理**

- 2.3.1 确定性有穷自动机的定义——DFA
- 即：下一个状态由当前状态和当前输入字符唯一给出的自动机。
- **定义：** DFA M 由字母表 Σ 、状态集合 S 、转换函数 $T: S \times \Sigma \rightarrow S$ 、初始状态 $s_0 \in S$ 及接受状态集合 $A \subset S$ 组成。

- 正则表达式与DFA之间的关系

- 相互转换

- 例1:正则表达式 ab

- 例2:正则表达式 $a|b$

- 例3:正则表达式 a^*

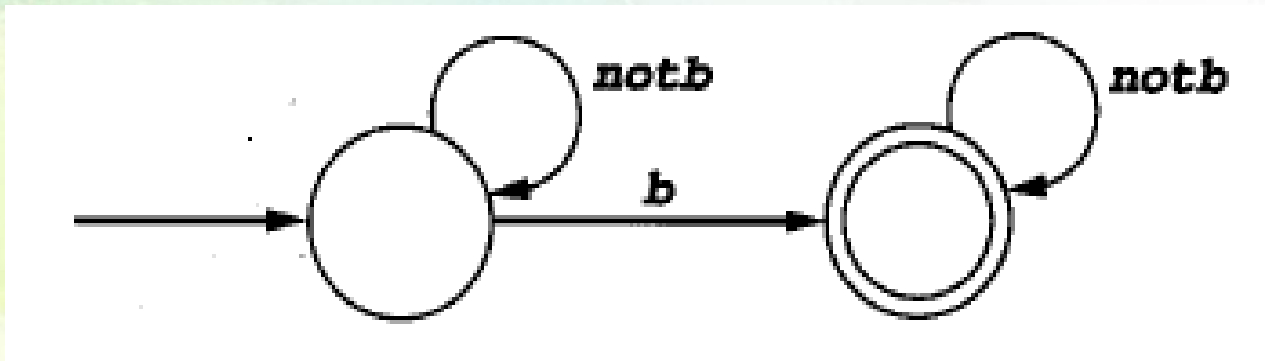
- 例4:正则表达式 $(ab)^*$

- 例5:正则表达式 $(a|b)^*$

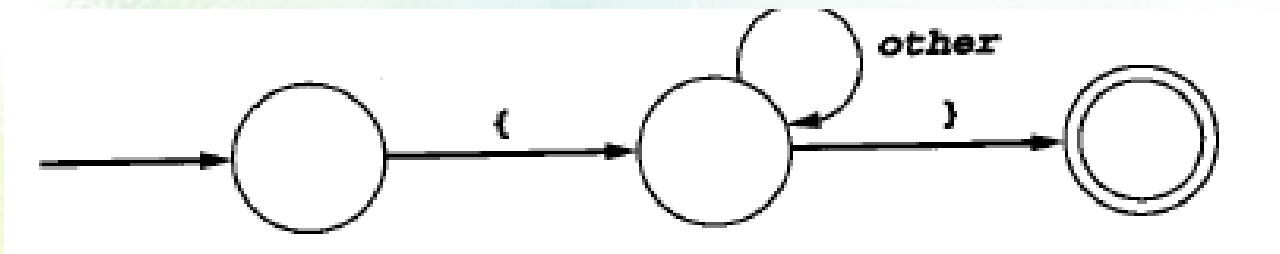
- 例6:正则表达式 $ab|ac$

- 例2.6: 串中仅有一个***b***的集合的正则表达式为:
 $(\text{not } b)^* b (\text{not } b)^*$

- 其对应的DFA为:



- 例2.9 非嵌套注释的DFA描述。
- 例如，Pascal注释 $\{ (\sim)^* \}$ 对应的DFA为：

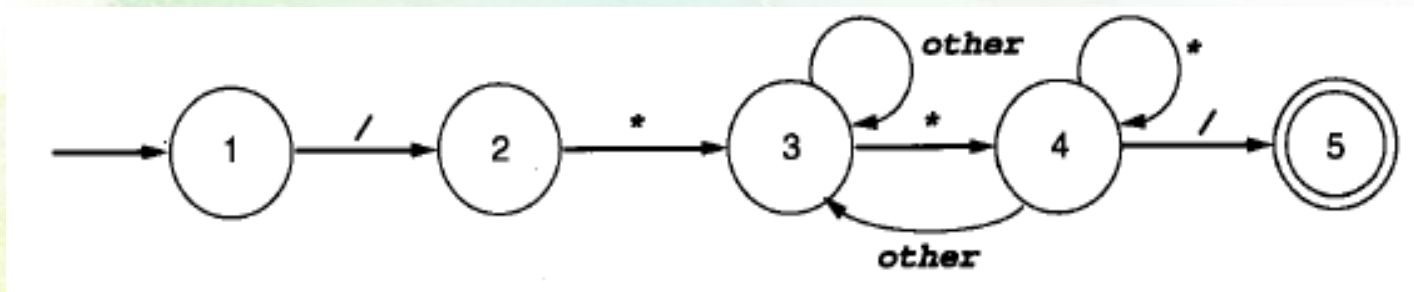


- 注意：other意味着除了右边花括号外的所有字符。

- C注释 —— 正则表达式困难！！

`/* ... (/* 不同时出现) ... */`

- DFA表示则非常简单：



- 结论：DFA的表达 ability 比正则表达式强！！

- 例2.8 科学表示法的数字常量的正则表达式为:

$nat = [0-9]^+$

$signedNat = (+|-)? nat$

$number = signedNat ("." nat)? (E signedNat)?$

- 如何画对应的DFA? 分解

- 第一步: 引入名字进行简化

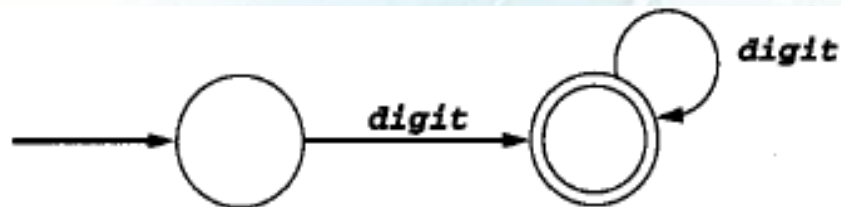
$digit = [0-9]$

$nat = digit^+$

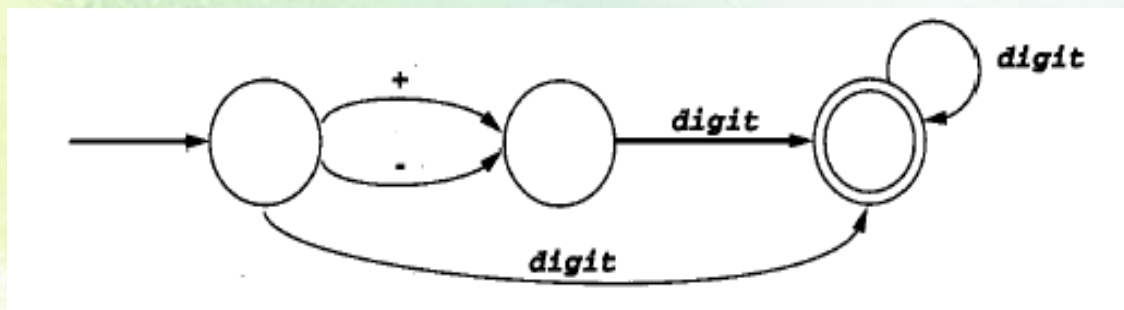
$signedNat = (+|-)? Nat$

$number = signedNat ("." nat)? (E signedNat)?$

- 第二步: 画出nat 的DFA:

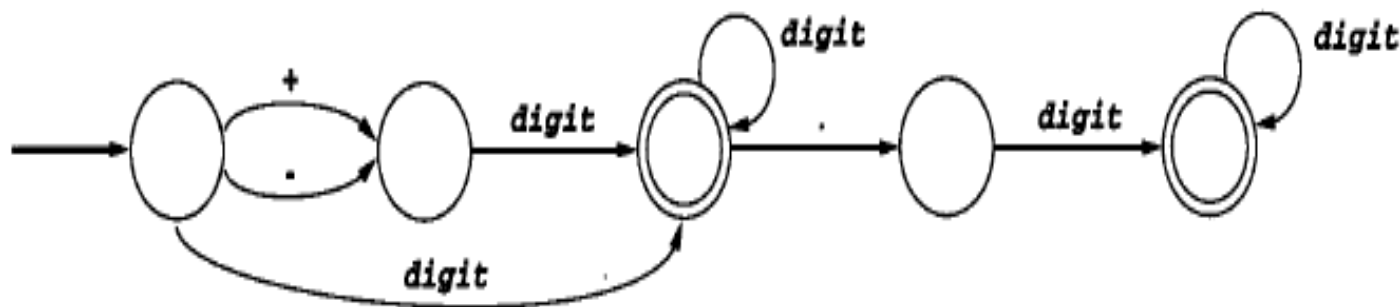


- 注意: 请记住 $a^+ = aa^*$ 对任意的 a 均成立。
- 第三步: 画出signedNat = (+|-)? Nat 的 DFA:
 - 添加可选的+|-



- 第四步:画出signedNat(“.” Nat) 的DFA:

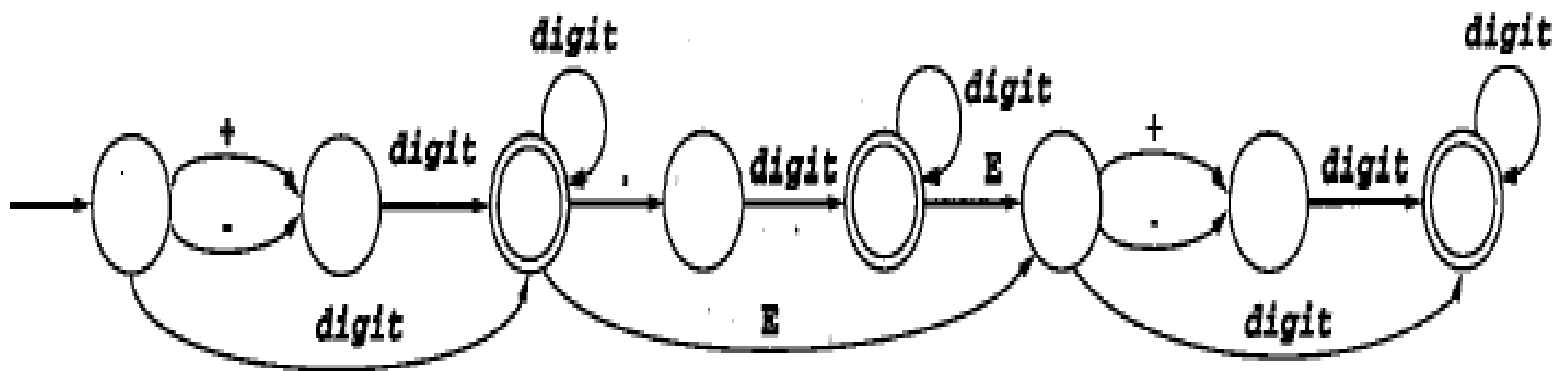
- 添加小数部分



- 注意: 它有两个接受状态, 它们表示小数部分是可选的。

第五步: $\text{signedNat}(\text{"." nat})?(E \text{ signedNat})?$ 的 DFA

- 添加可选的指数部分



- **结论:** 直接将正则表达式转换为 DFA 不容易, 规律不容易找!! 怎么办??

转换困难的原因分析

- 状态身份重叠

→ 粘合不容易

- 状态身份的分拆

- 正则表达式与DFA之间的关系

- 相互转换

- 例1:正则表达式 ab

- 例2:正则表达式 $a|b$

- 例3:正则表达式 a^*

- 例4:正则表达式 $(ab)^*$

- 例5:正则表达式 $(a|b)^*$

- 例6:正则表达式 $ab|ac$

- 例2.8 科学表示法的数字常量的正则表达式为:

nat=[0-9]+

signedNat=(+|-)?*nat*

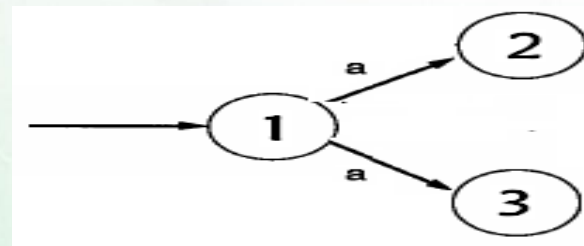
number=*signedNat*("." *nat*)?(E *signedNat*)?

转换困难的原因分析

- 状态身份重叠

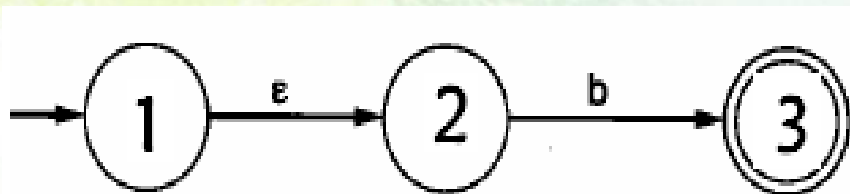
→ 粘合不容易

- 状态身份的分拆



- (1) 允许一对多的转换

- (2) 允许 ϵ -转换



- 引入新的有穷自动机

- 非确定性有穷自动机
(nondeterministic finite automaton)
- 简称为NFA。
- NFA与DFA不同之处：
 - (1) 允许 ε -转换
 - (2) 允许一对多的转换

NFA(nondeterministic finite automaton)

- **定义**: NFA M 由字母表 Σ 、状态的集合 S 、转换函数 $T: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$ 、 S 的初始状态 s_0 ，以及 S 的接受状态 A 的集合组成。

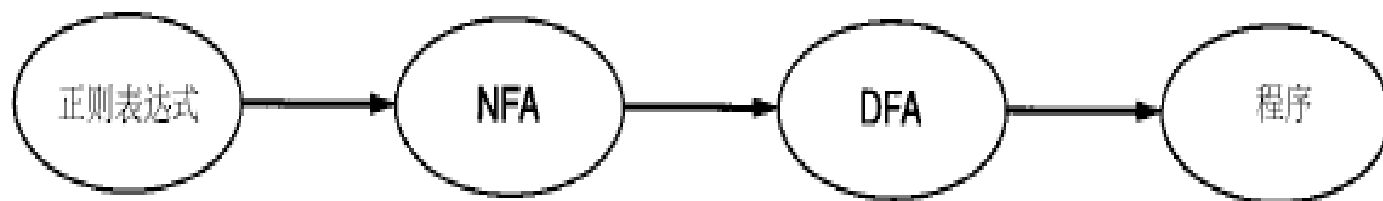
2.4 正则表达式 \rightarrow DFA

- 方法1: 直接转换 \rightarrow 困难, 复杂, 难以找到规律
- 方法2: 引入中间模型 \rightarrow NFA \rightarrow 简单, 有规律

- 方法2的介绍

- (1)将正则表达式 \rightarrow NFA

- (2)将NFA \rightarrow DFA



2.4.1 正则表达式 \rightarrow NFA

- Thompson方法 (Thompson construction)
- Thompson方法利用 ϵ -转换将正则表达式的机器片段“粘在一起”以构成与整个表达式相对应的机器。
- 归纳方法 \rightarrow 即从正则表达式定义出发，
 - 首先为每个基本正则表达式画出一个NFA，
 - 接着根据正则表达式的运算符特点将各个NFA连接起来。

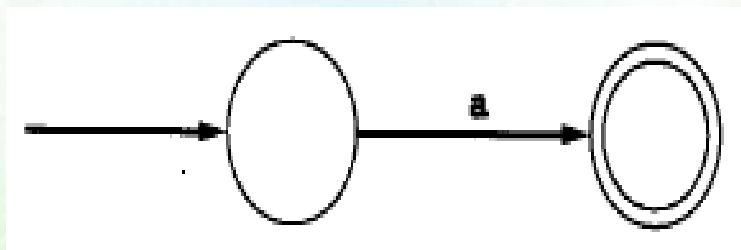
如何掌握好该转换方法

- (1) 弄清楚一个基本正则表达式对应的NFA是如何画出来的。
- (2) 弄清楚各种运算符号在NFA图中的连接方法。

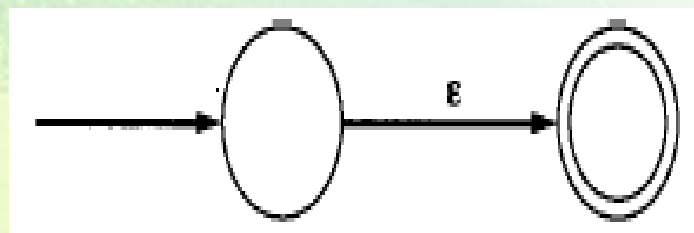
或运算 |
并置运算
重复运算 *

- 1) 基本正则表达式

- 与正则表达式 a 等同的NFA:

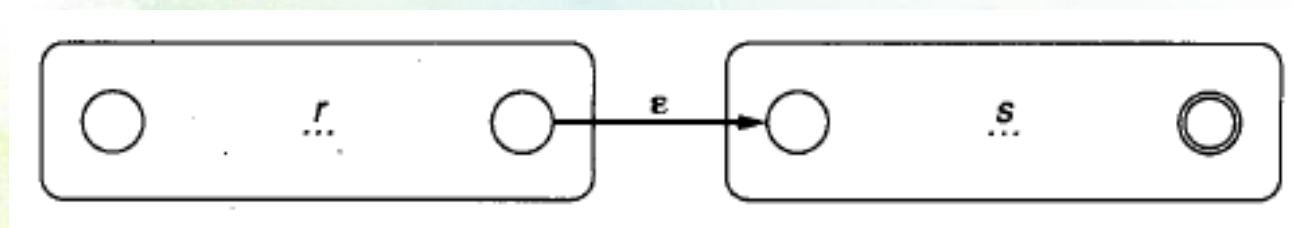


- 与 ϵ 等同的NFA:



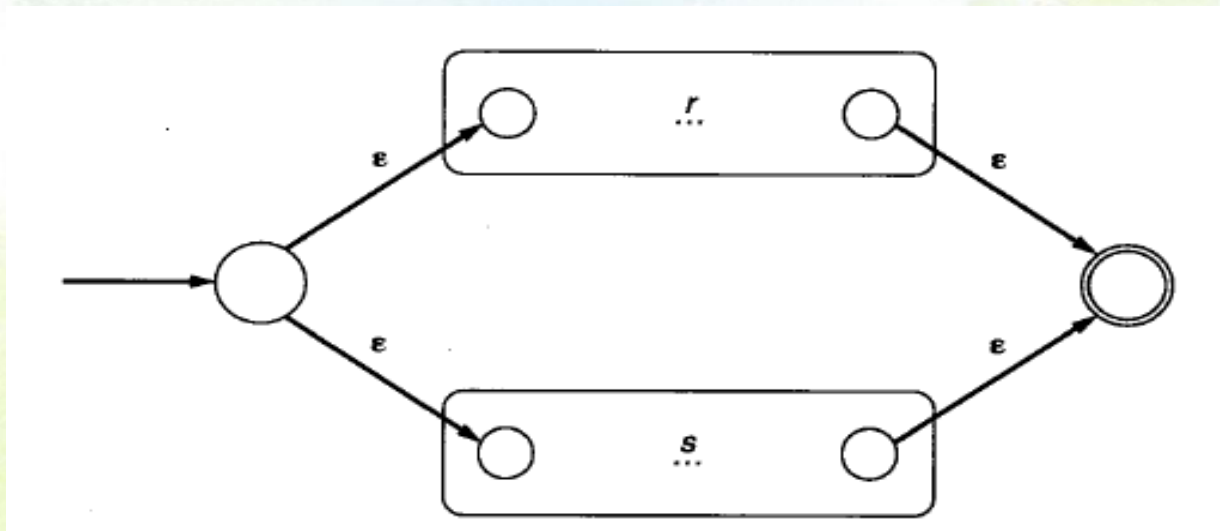
2) 并置

与 rs 对应的 NFA:



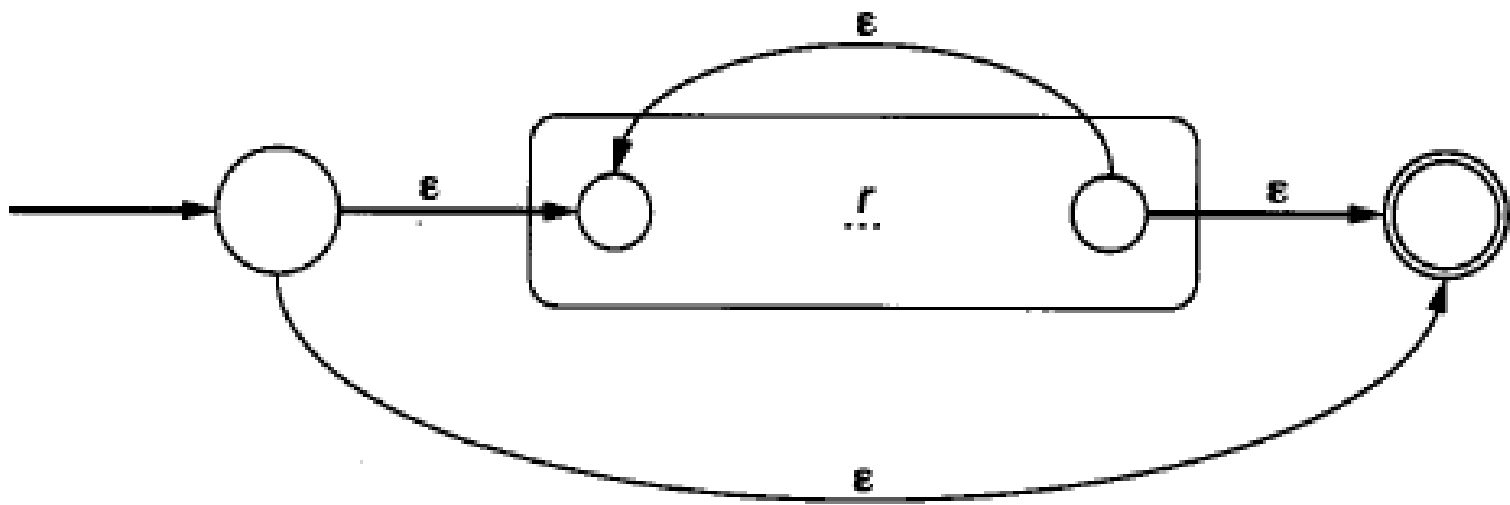
- 3) 在各选项中选择

与 $r|s$ 相对应的NFA:

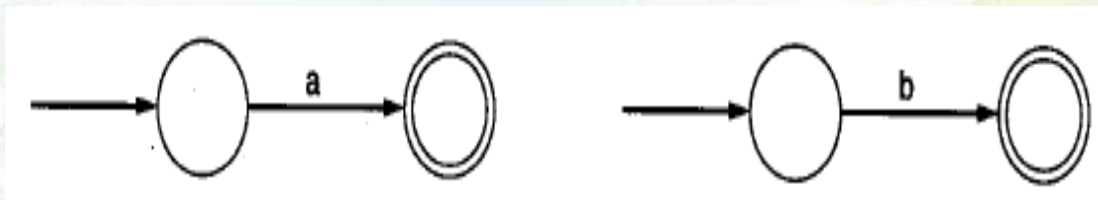


- 4) 重复

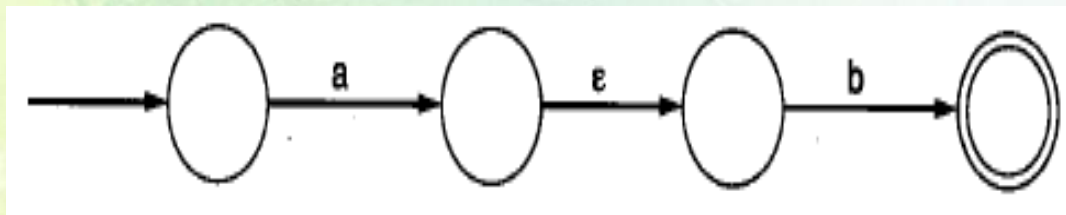
- r^* 相对应的NFA:



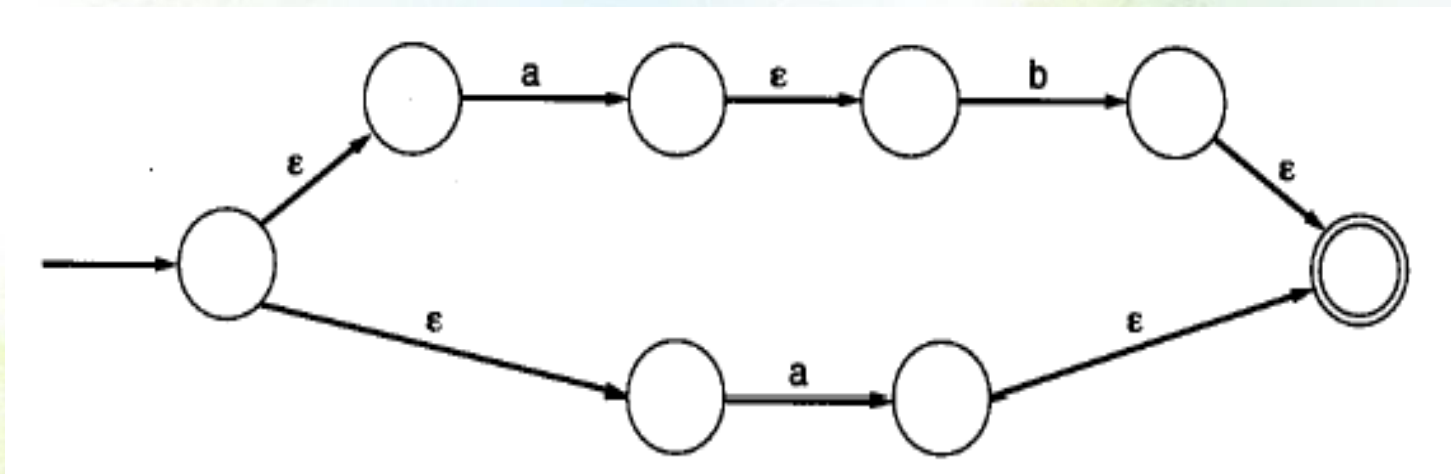
- 例1 将正则表达式 $ab|a$ 转换为NFA。
- 首先为正则表达式 a 和 b 分别构造机器：



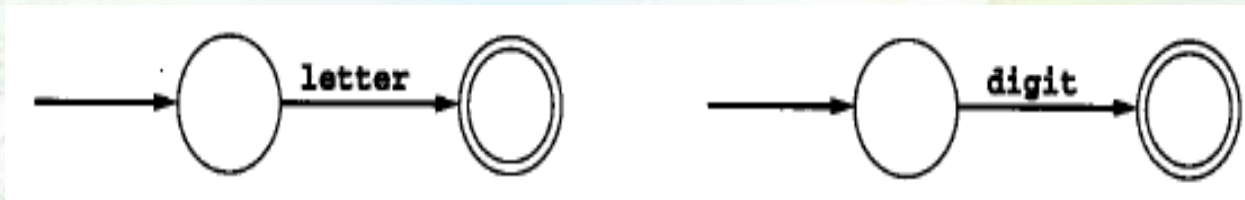
- 接着再为并置 ab 构造机器：



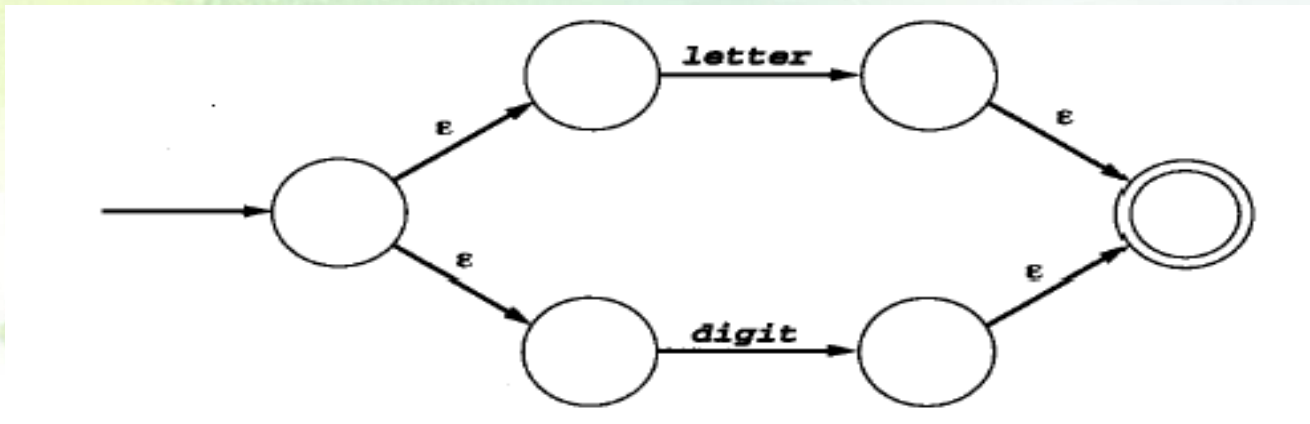
- $ab|a$ 对应的NFA:



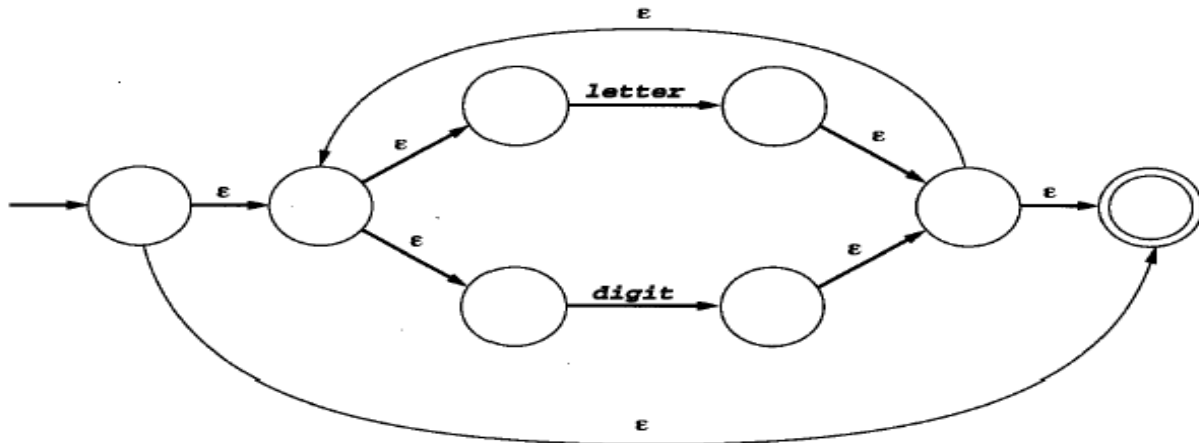
- 例2 正则表达式 $letter(letter|digit)^*$ 对应的NFA。
- 首先分别为正则表达式 $letter$ 和 $digit$ 构建机器：



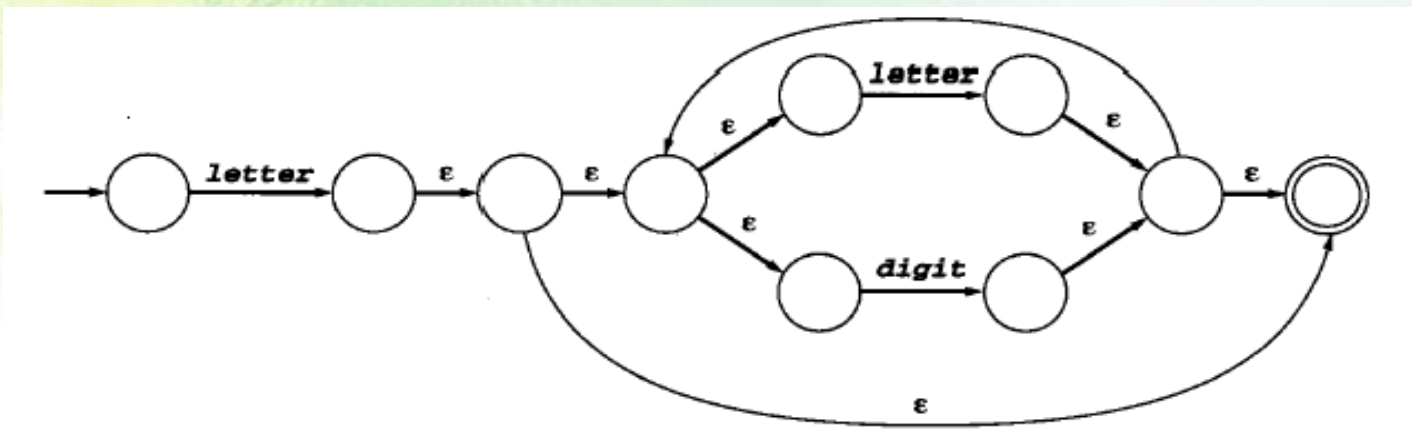
- 接着再为选择 $letter|digit$ 构造机器：



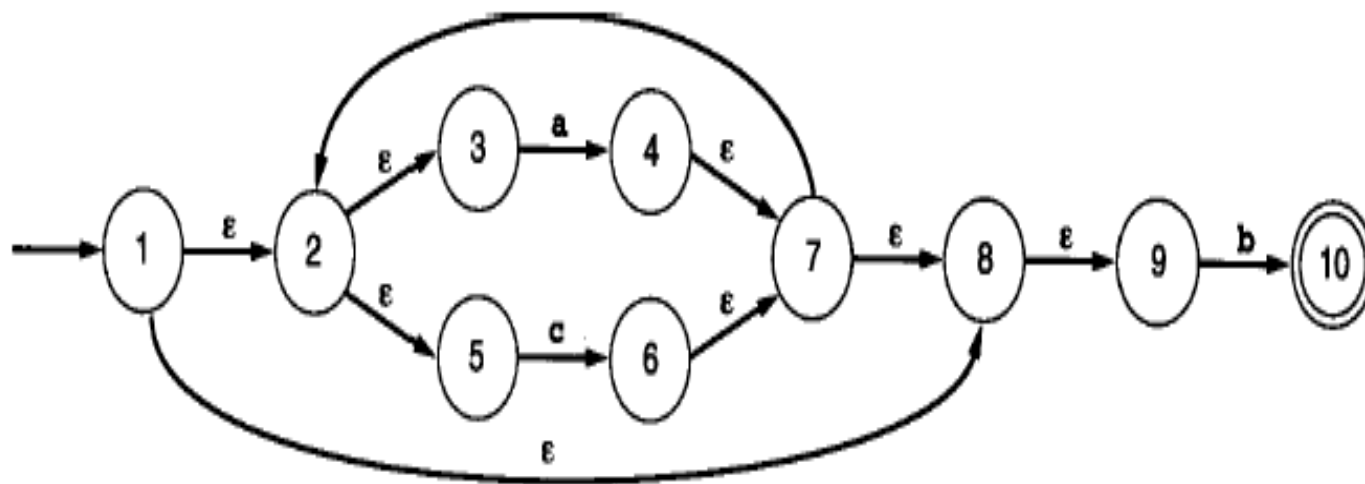
- 重复 $(letter|digit)^*$ 构造NFA，如下所示：



- 最后，将 $letter$ 和 $(letter|digit)^*$ 并置在一起，并构造该并置的机器以得到完整的NFA：



- 例：正则表达式 $(a|c)^*b$ 按 Thompson 方法构造所得 NFA 如图：



2.4.2 NFA \rightarrow DFA

- 合并等价状态
- 分析NFA与DFA有何不同：
- (1) ε -转换
- (2) 多重转换

- (1) 消除 ϵ -转换

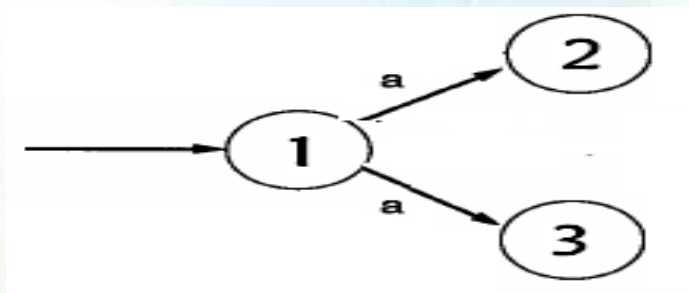


由于 ϵ 是空串，因此，可以1,2状态理解为一类

ϵ -闭包(ϵ -closure)是可由 ϵ -转换从某状态或某些状态达到的所有状态集合。

即状态1的 **ϵ -闭包(ϵ -closure)**为： **$\{1, 2\}$**

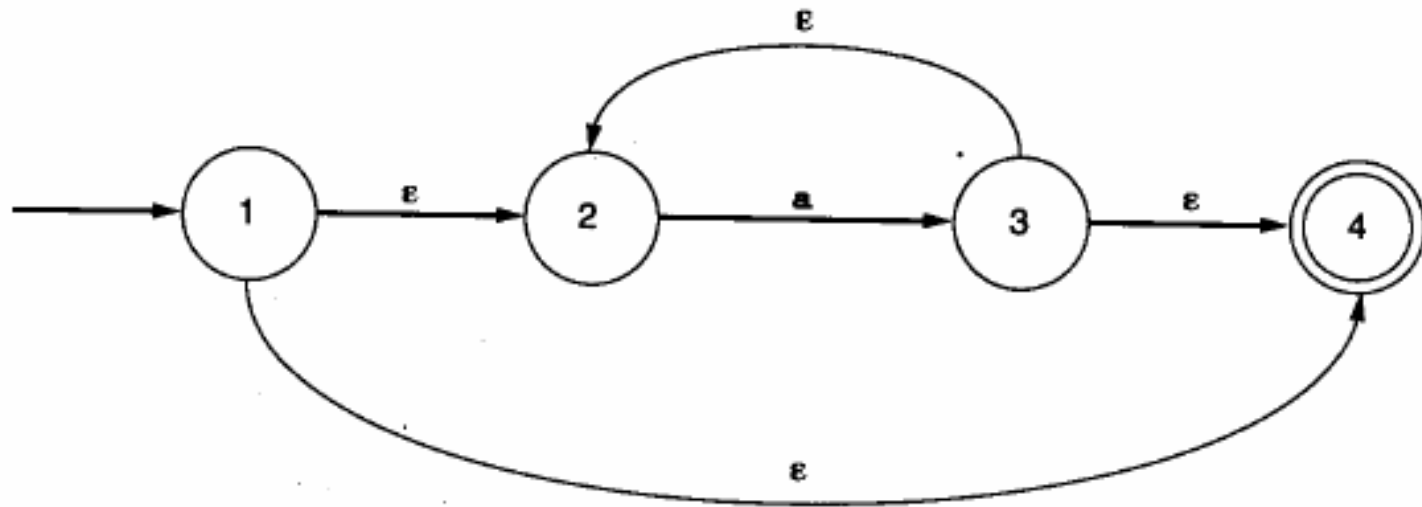
- (2)消除多重转换



- 即 $F(1,a)=\{2,3\}$

- **结论**：因为这两个过程得到的结果均是状态集合而不是单个状态，因此称这个算法为**子集构造**(subset construction)。

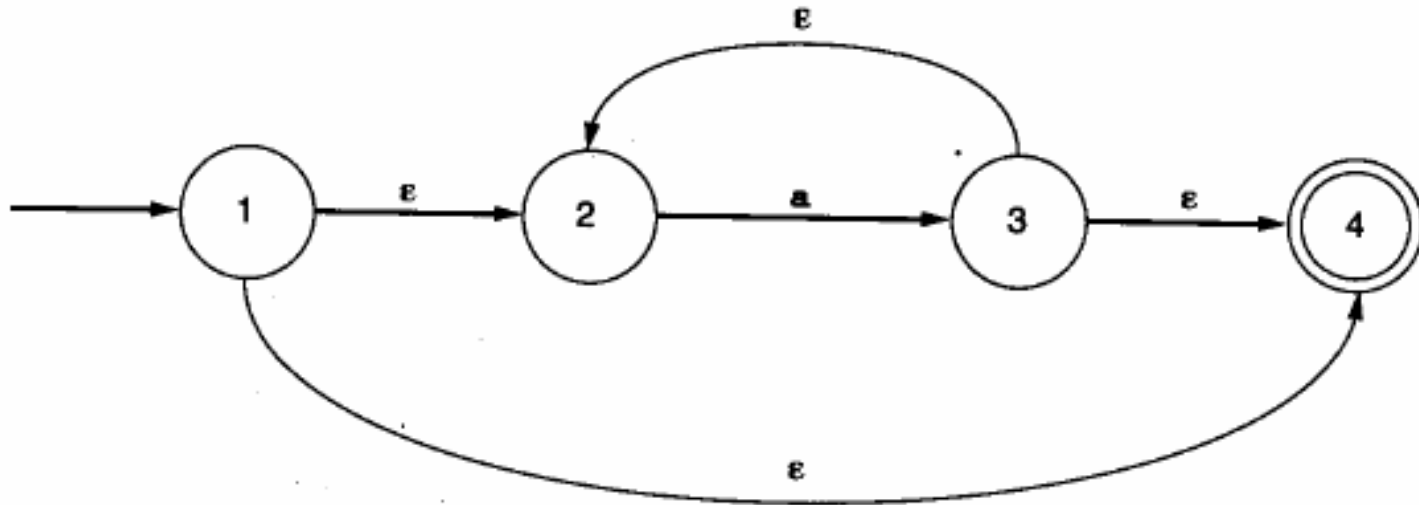
如何实现转换



如何实现转换

- 从目标出发找出解决方法：
 - 1.DFA初态只有一个。
 - 意味着要把NFA中的初态进行等价合并。
 - 2.DFA中存在的都是非 ϵ 转换。
 - 意味着从初态开始进行非 ϵ 转换，如果得到新状态就要进行新一轮的处理。

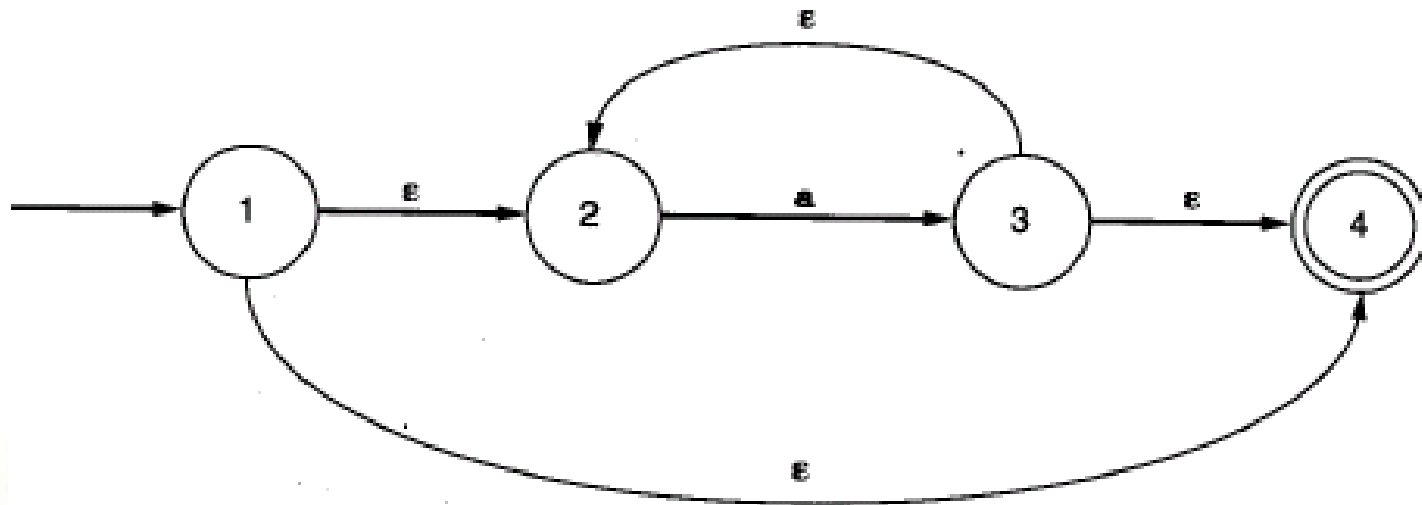
例1 求出正则表达式 a^* 对应NFA初态1的等价状态：



初态1的等价状态有： $\{1, 2, 4\}$ ，书写为： $\bar{1} = \{1, 2, 4\}$

\bar{s} 称为状态 s 的 ϵ -闭包：即由一系列的零个或多个 ϵ -转换所能达到的状态集合。

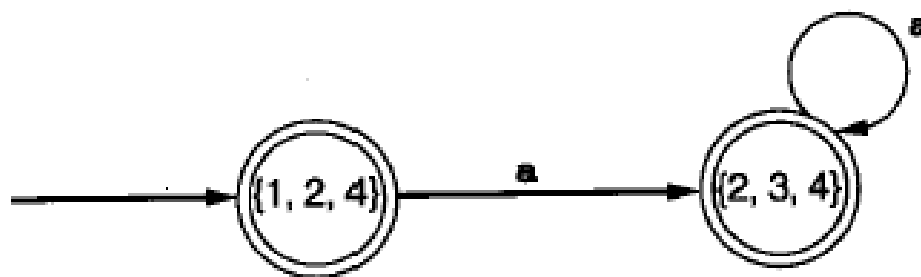
例：求初态集合 $\{1, 2, 4\}$ 的非 ϵ 转换。

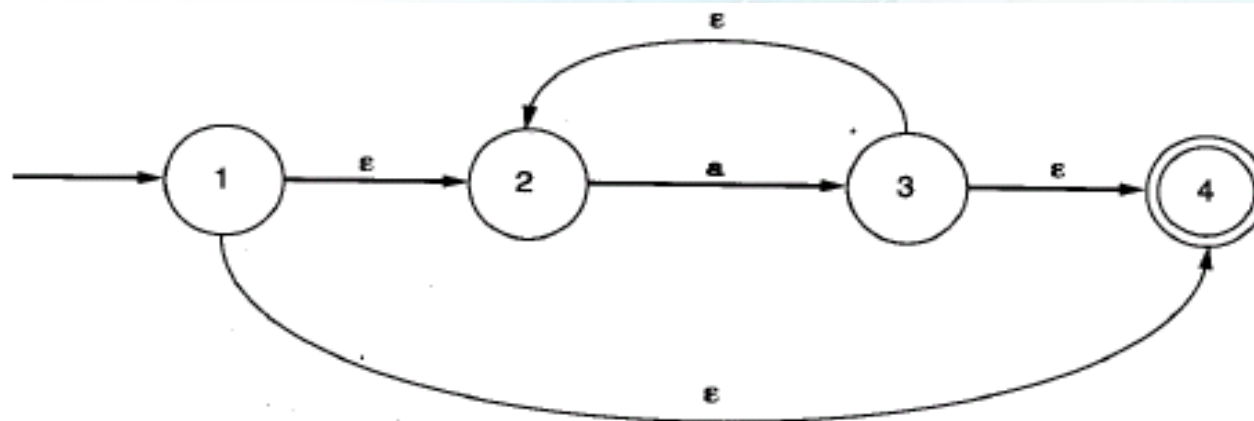


NFA初始状态1的 ϵ -闭包 $=\{1, 2, 4\}$ ，而转换只有a上的转换。

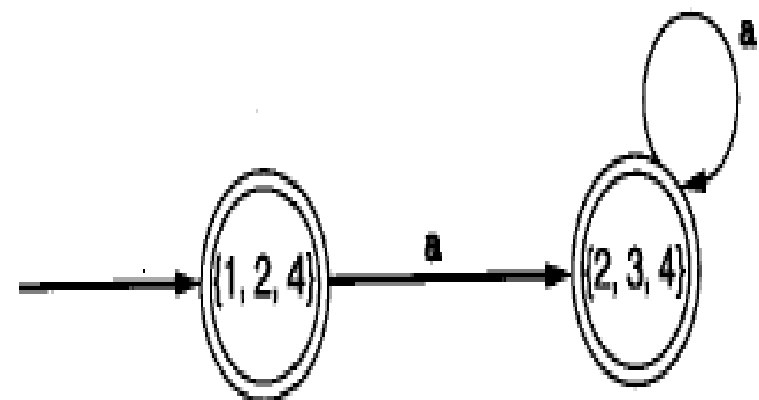
例2.15 请考虑例2.14中的NFA：与它相对应的DFA的初始状态是 $\bar{1} = \{1,2,4\}$ ，且存在着在字符 a 上的由状态2向状态3的转换，而在 a 上则没有来自状态1或状态4的转换，因此在 a 上就有从 $\{1,2,4\}$ 到 $\overline{\{1,2,4\}_a} = \overline{\{3\}} = \{2,3,4\}$ 的转换。由于再也没有来自一个字符上的1、2或4状态的转换了，因此就可将注意力转向新状态 $\{2,3,4\}$ 。此时在 a 上有从状态2到状态3的转换，且也没有来自3或4状态的 a - 转换，因此就有从 $\{2,3,4\}$ 到 $\overline{\{2,3,4\}_a} = \overline{\{3\}} = \{2,3,4\}$ 的转换，因而也就有从 $\{2,3,4\}$ 到它本身的 a - 转换。我们已将所有的状态都考虑完了，所以也构造出了整个DFA。唯一需要读者注意的是NFA的状态4是接受的，这是因为 $\{1,2,4\}$ 和 $\{2,3,4\}$ 都包含了状态4，它们都是相应的DFA的接受状态。将构造出的DFA画出来，其中用状态各自的子集命名状态：

于是转换得到的DFA为：

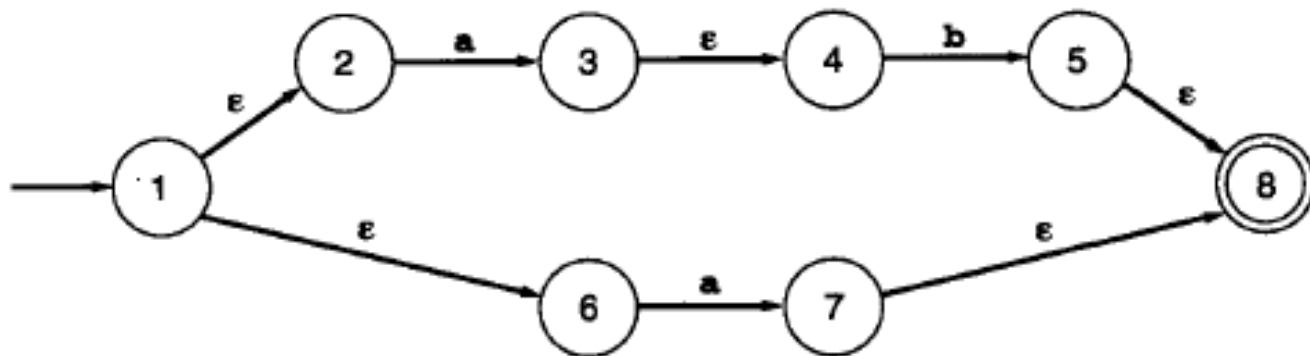




<div> <div></div> <div>字符</div> </div>	a
状态集合	
$\{1, 2, 4\}$	$\{2, 3, 4\}$
$\{2, 3, 4\}$	$\{2, 3, 4\}$



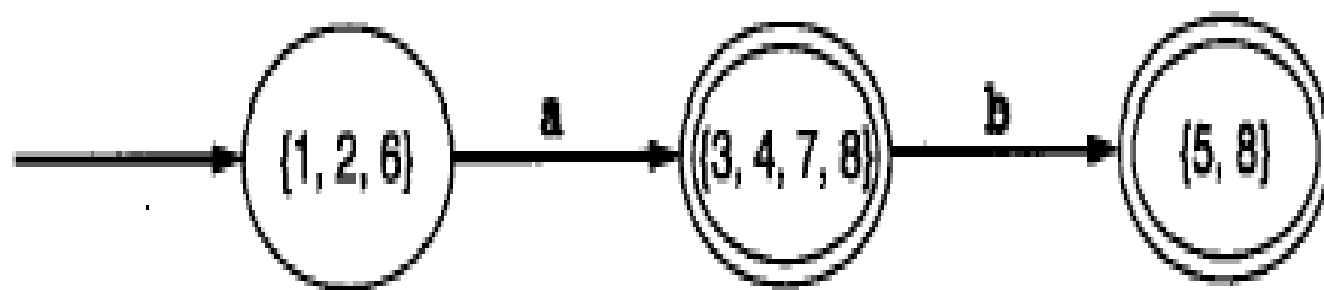
- 例2 将下面NFA转换为DFA.



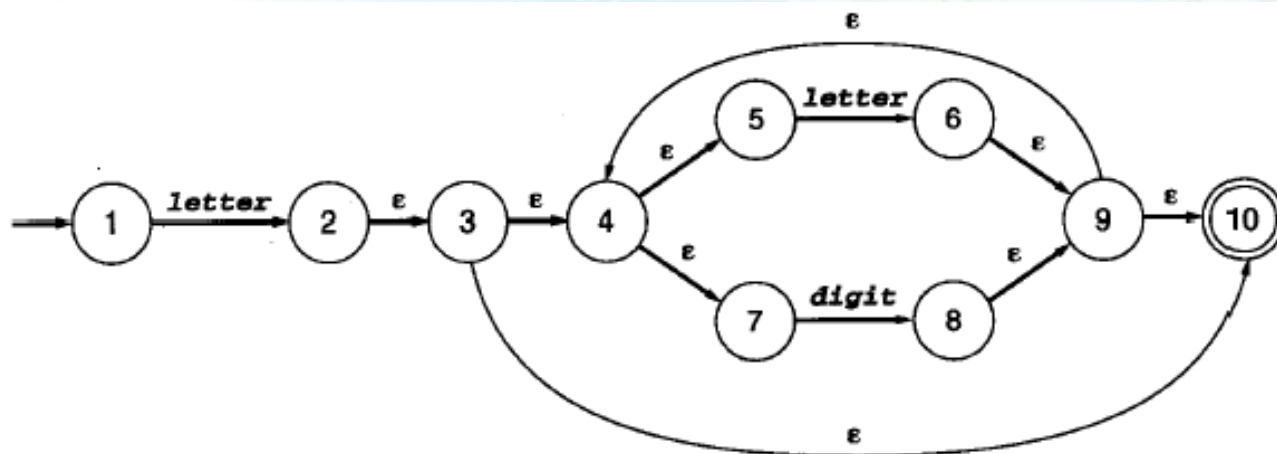
则子集构造过程用下表表示:

状态集合 \ 字符	a	b
{ 1, 2, 6 }	{ 3, 4, 7, 8 }	
{ 3, 4, 7, 8 }		{ 5, 8 }
{ 5, 8 }		

因此所得的DFA为：



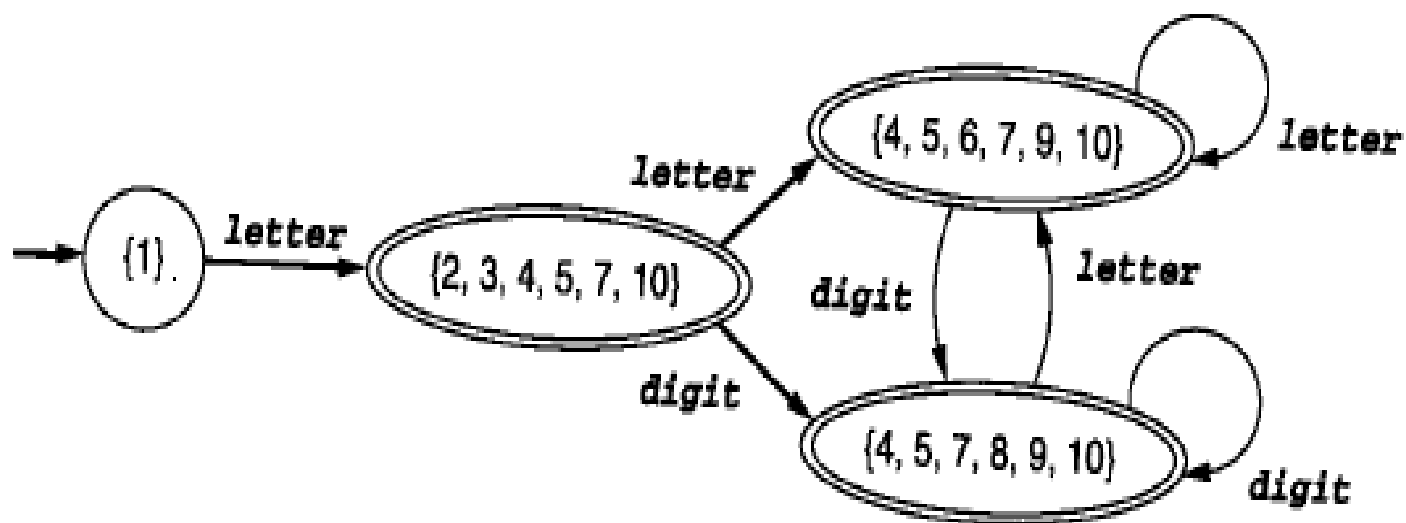
- 例3.将下面NFA转换为DFA。



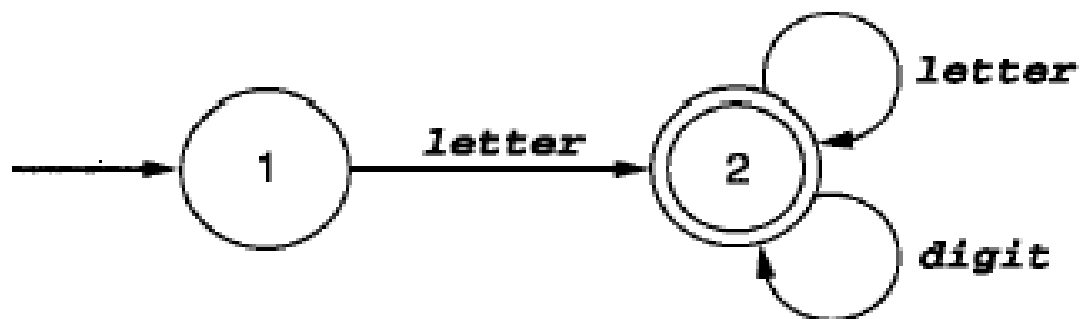
- 子集构造过程用下表表示：

字符 \ 状态集合	letter	digit
{ 1 }	{ 2, 3, 4, 5, 7, 10 }	
{ 2, 3, 4, 5, 7, 10 }	{ 6, 9, 4, 7, 10, 5 }	{ 8, 9, 4, 7, 5, 10 }
{ 6, 9, 4, 7, 10, 5 }	{ 6, 9, 4, 7, 10, 5 }	{ 8, 9, 4, 7, 5, 10 }
{ 8, 9, 4, 7, 5, 10 }	{ 6, 9, 4, 7, 10, 5 }	{ 8, 9, 4, 7, 5, 10 }

因此，得到的DFA如下图：



缺陷：转换得到的DFA太复杂，状态数太多。



如何最小化——方法一

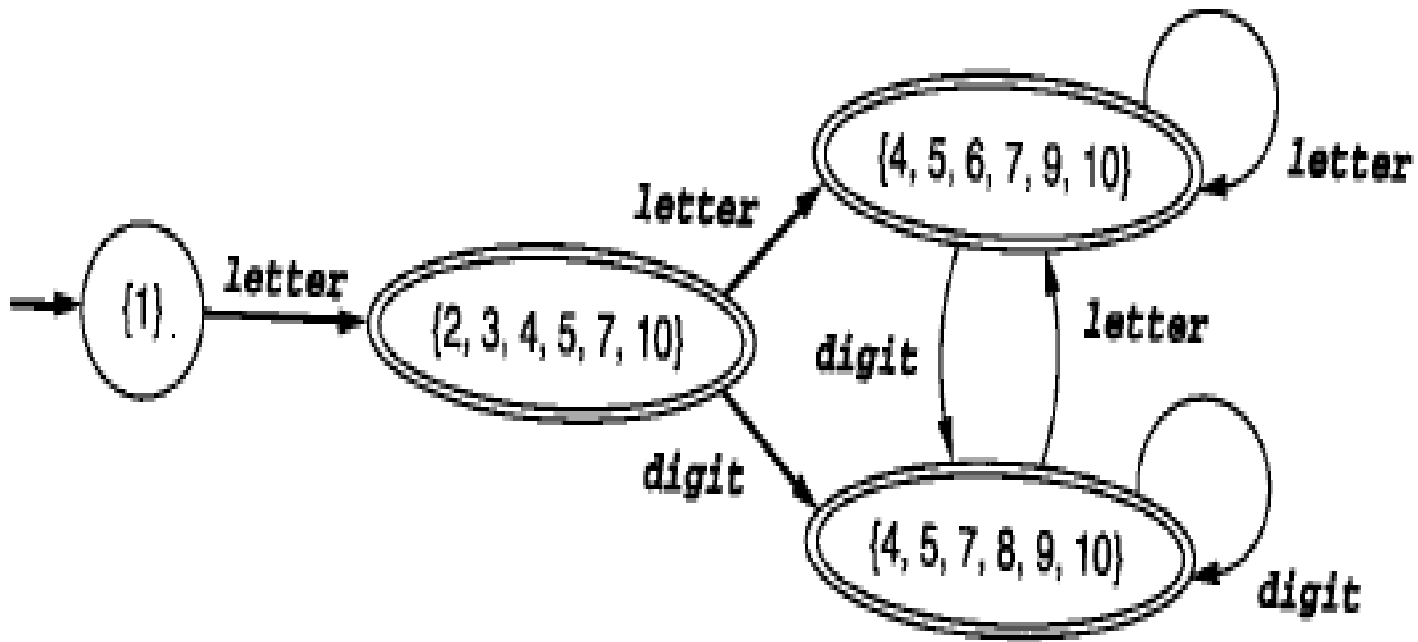
- 状态数最少

- 即意味着把多余的、等价的状态进行合并

- 等价：目标一致即可

- 如何进行等价合并呢？

- 逐个状态逐个状态进行分析比较。



- 缺点：逐个找状态进行等价判断以确定是否进行合并，这样比较次数太多。

如何最小化——方法二

- 状态数最少

→ 反向思考

→ 从DFA拥有的最少状态（终态和非终态）
开始进行分析

→ 根据目标是否一致来确定是否需要分拆

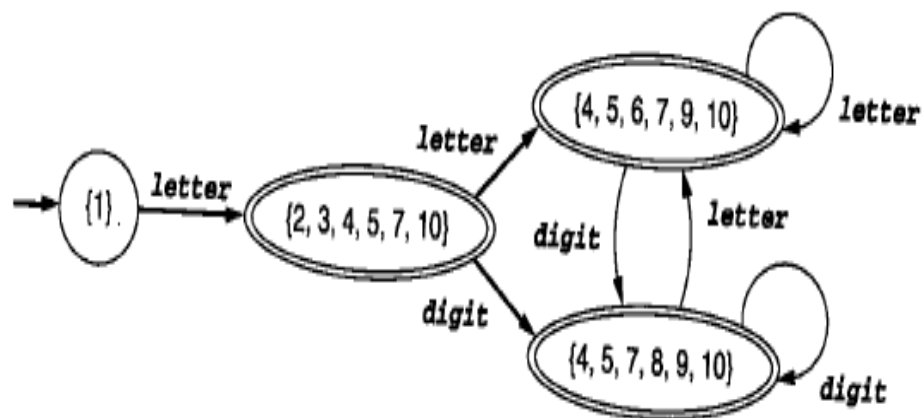
首先,令

$A = \{ 1 \}$

$B = \{ 2, 3, 4, 5, 7, 10 \}$

$C = \{ 6, 9, 4, 7, 10, 5 \}$

$D = \{ 8, 9, 4, 7, 5, 10 \}$

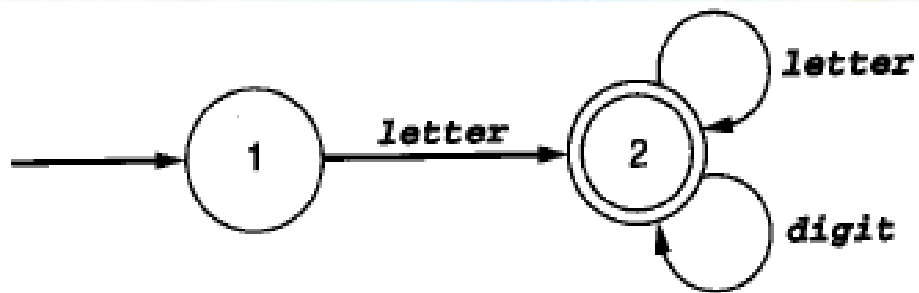


(1) 将DFA中的状态划分为非终态集合 $s_1 = \{A\}$, 和终态集合 $s_2 = \{B, C, D\}$

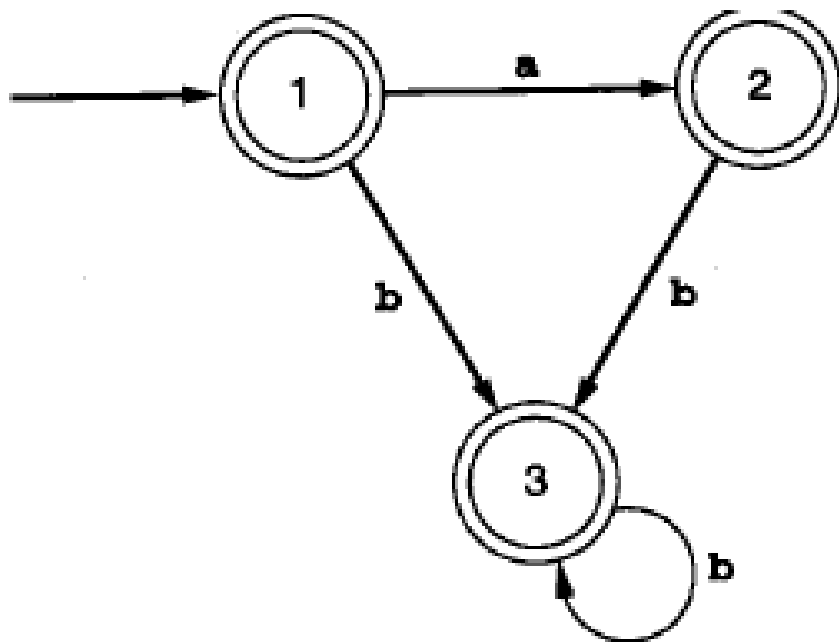
符号 \ 状态集合	letter	digit
A	$\{ B \} \subset S_2$	
B	$\{ C \} \subset S_2$	$\{ D \} \subset S_2$
C	$\{ C \} \subset S_2$	$\{ D \} \subset S_2$
D	$\{ C \} \subset S_2$	$\{ D \} \subset S_2$

(2) 由于 s_1, s_2 集合中的各状态经符号 letter 和 digit 转换, 得到的集合均属于同一集合。

- 最小状态DFA为：



- 例2.19 将下面与正则表达式 $(a|\epsilon)b^*$ 对应的DFA进行最小化

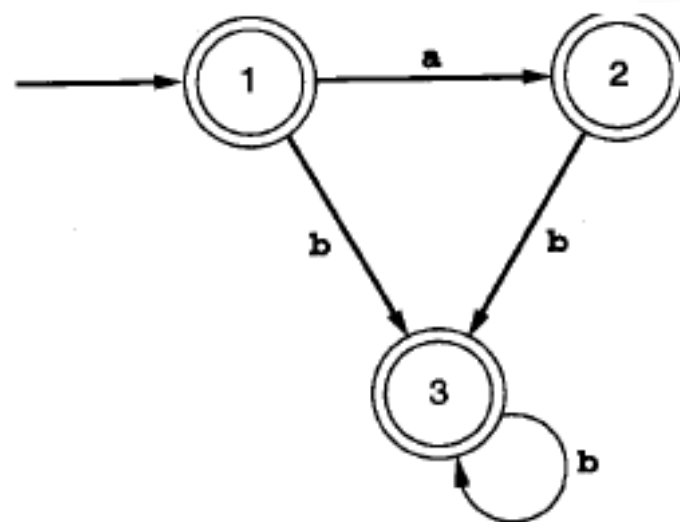
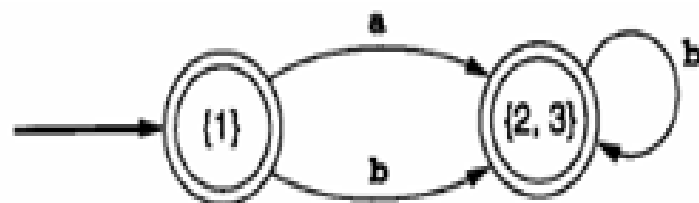


(1) 将DFA中的状态划分为终态集合 $s1=\{1, 2, 3\}$ ，和非终态集合 $s2=\{\}$

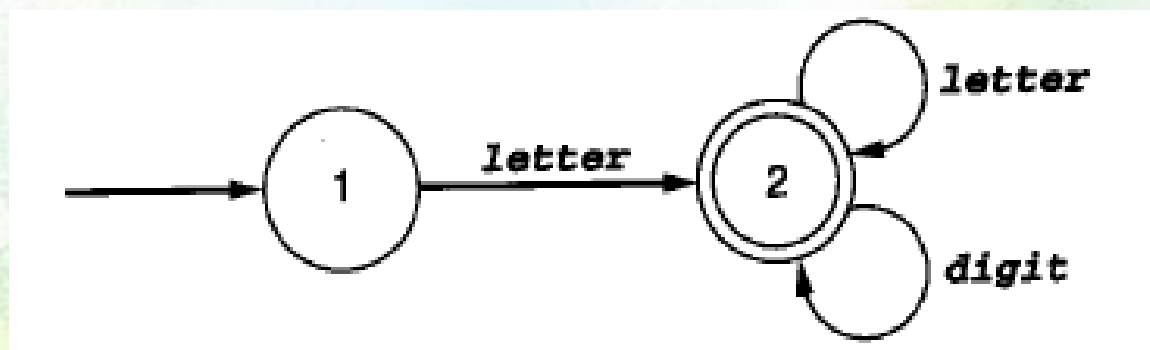
符号 \ 状态集合	a	b
{1}	{2} $\subset S1$	{3} $\subset S1$
{2}		{3} $\subset S1$
{3}		{3} $\subset S1$

(2) 可见 {2} 与 {3} 状态经a或b转换得到的状态集合是一样，但与 {1} 得到的状态集合不一样。因此 {2} {3} 合并作为一个状态处理。{1} 则作为另一个状态。

因此，就得到了最小状态的DFA：



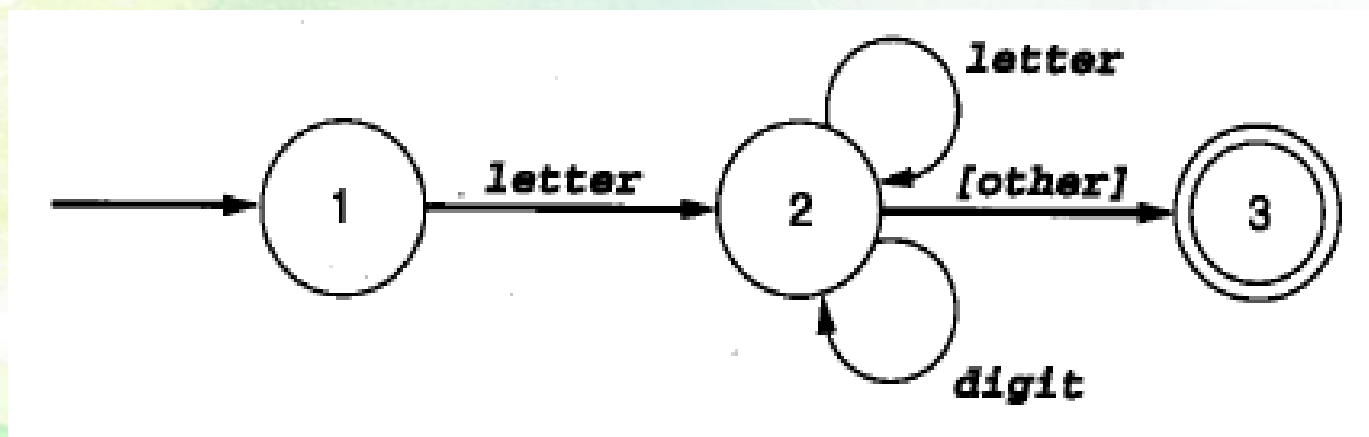
- 2.3.3 用代码实现有穷自动机



- 2.3.3 用代码实现有穷自动机

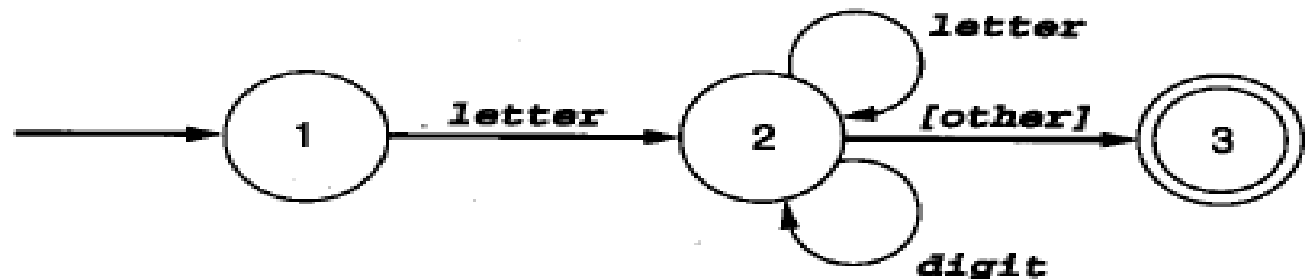
既然DFA可以反映单词串的识别过程，那么**如何将DFA转换为程序代码？**

- 首先让我们标识符中合了**先行和最长子串原理**的DFA图：



代码为:

```
{ starting in state 1 }  
if the next character is a letter then  
    advance the input;  
    { now in state 2 }  
    while the next character is a letter or a digit do  
        advance the input; { stay in state 2 }  
    end while;  
    { go to state 3 without advancing the input }  
    accept ;  
else  
    { error or other cases }  
end if;
```



- **特点：**状态少且DFA中的循环较小，那么就比较合适了。
- **缺点：**
 - 首先它是特殊的，即必须用略微不同的方法处理各个DFA，而且规定一个用这种方法将每个DFA翻译为代码的算法较难。
 - 其次：当状态增多或更明确时，且当相异的状态与任意路径增多时，代码会变得非常复杂。

{ state 1 }

if the next character is “ / ” then

advance the input: { state 2 }

if the next character is “*” then

advance the input; { state 3 }

done := false;

while not done do

while the next input character is not “*” do

advance the input; { stay in state 3 }

end while;

advance the input; { state 4 }

while the next input character is “*” do

advance the input; { stay in state 4 }

end while;

if the next input character is “ / ” then

done := true ;

end if;

advance the input;

end while;

accept; { state 5 }

else

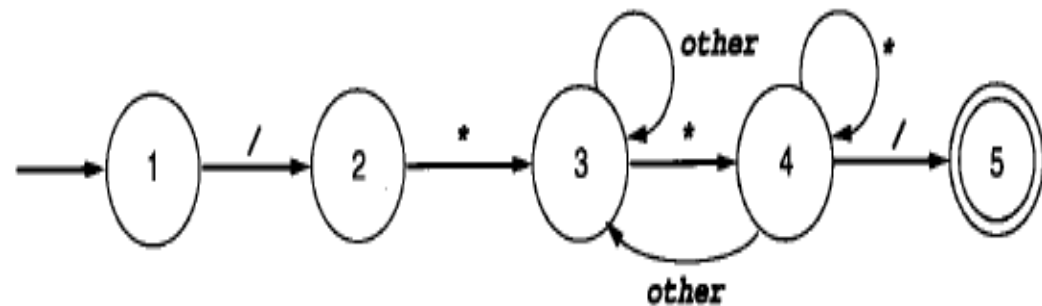
{ other processing }

end if;

else

{ other processing }

end if;



较好的解决方法1

- ——状态转换方法
- 即利用一个变量保持当前的状态，并将转换写成一个双层嵌套的case语句而不是一个循环。
- 其中第1个case语句测试当前的状态，嵌套着的第2层测试输入字符及所给状态。

- **新代码:**

state := 1; { start }

while *state = 1 or 2* **do**

case *state* **of**

1: **case** *input character* **of**

letter : *advance the input*;

state := 2;

else *state := ... { error or other }*;

end case;

2: **case** *input character* **of**

letter, digit: *advance the input*;

state := 2; { actually unnecessary }

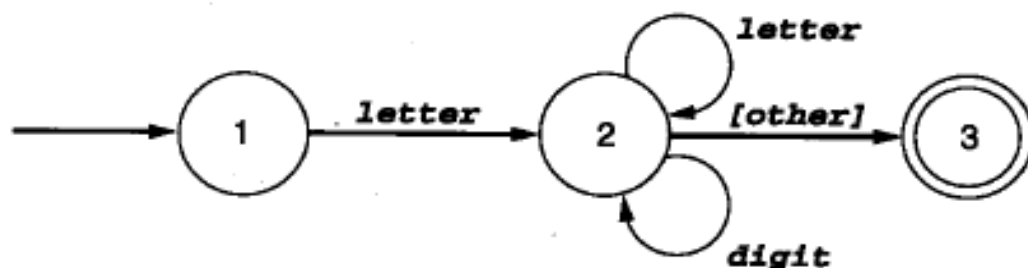
else *state := 3*;

end case;

end case;

end while;

if *state = 3* **then** *accept* **else** *error* ;



state := 1; { start }

while *state = 1, 2, 3 or 4 do*

case *state of*

1: **case** *input character of*

"/" : advance the input;

state := 2;

else *state := . . . { error or other };*

end case;

2: **case** *input character of*

"": advance the input;*

state := 3;

else *state := . . . { error or other };*

end case;

3: **case** *input character of*

"": advance the input;*

state := 4;

else *advance the input { and stay in state 3 };*

end case;

4: **case** *input character of*

"/" advance the input;

state := 5;

"": advance the input; { and stay in state 4 }*

else *advance the input;*

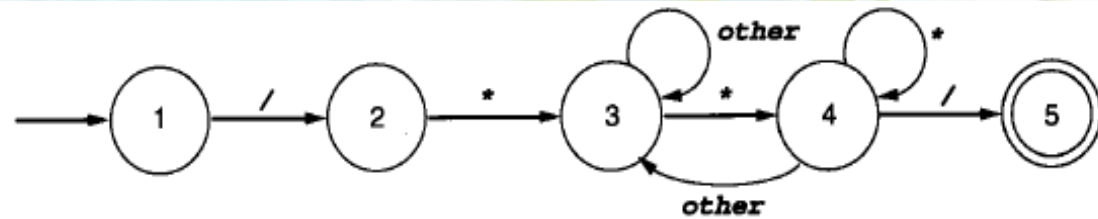
state := 3;

end case;

end case;

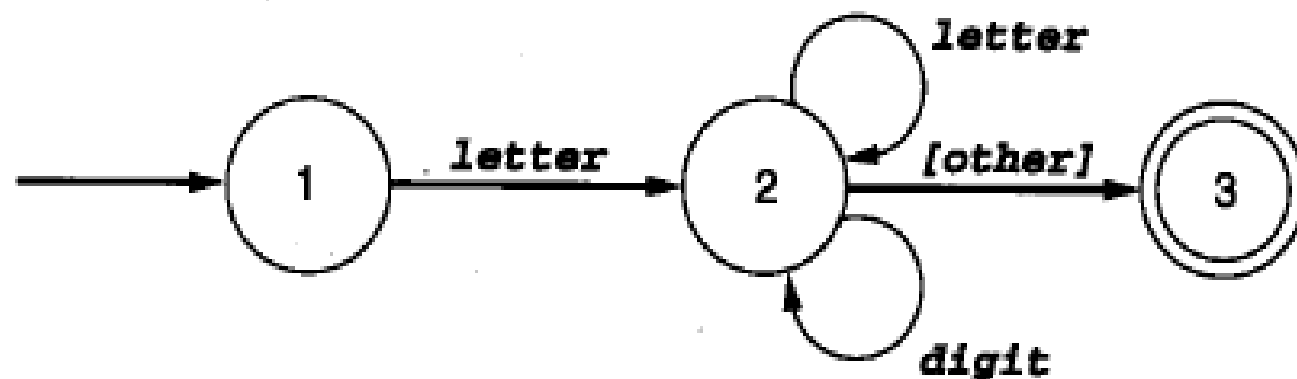
end while;

if *state = 5 then accept else error ;*



解决方法2

- **转换表**(transition table)——二维数组
- 通过表示转换函数 T 值的状态和输入字符来索引



	字母表C中的字符
状态S	经转换T(S,c)所达到的状态

例如：标识符的DFA可表示为如下的转换表：

输入 \ 状态	字母	数字	其他
1	2		
2	2	2	3
3			

设计TINY语言的词法分析程序

TINY语言的词法分析

Tiny语言的词法

TINY 样本语言的单词分类

(1). 关键词:

If then else end repeat until read write 不区分大小写

(2). 专用符号: + - * / < = { } ; :=

(3). identifier和number, 则定义:

identifier: 字母开头, 后面可跟若干个字母, 且不区分大小写

number: 数字符号开头, 后面可跟若干个数字

也可以抽象地表示为:

identifier = letter letter*

number = digit digit*

letter = a|..|z|A|..|Z

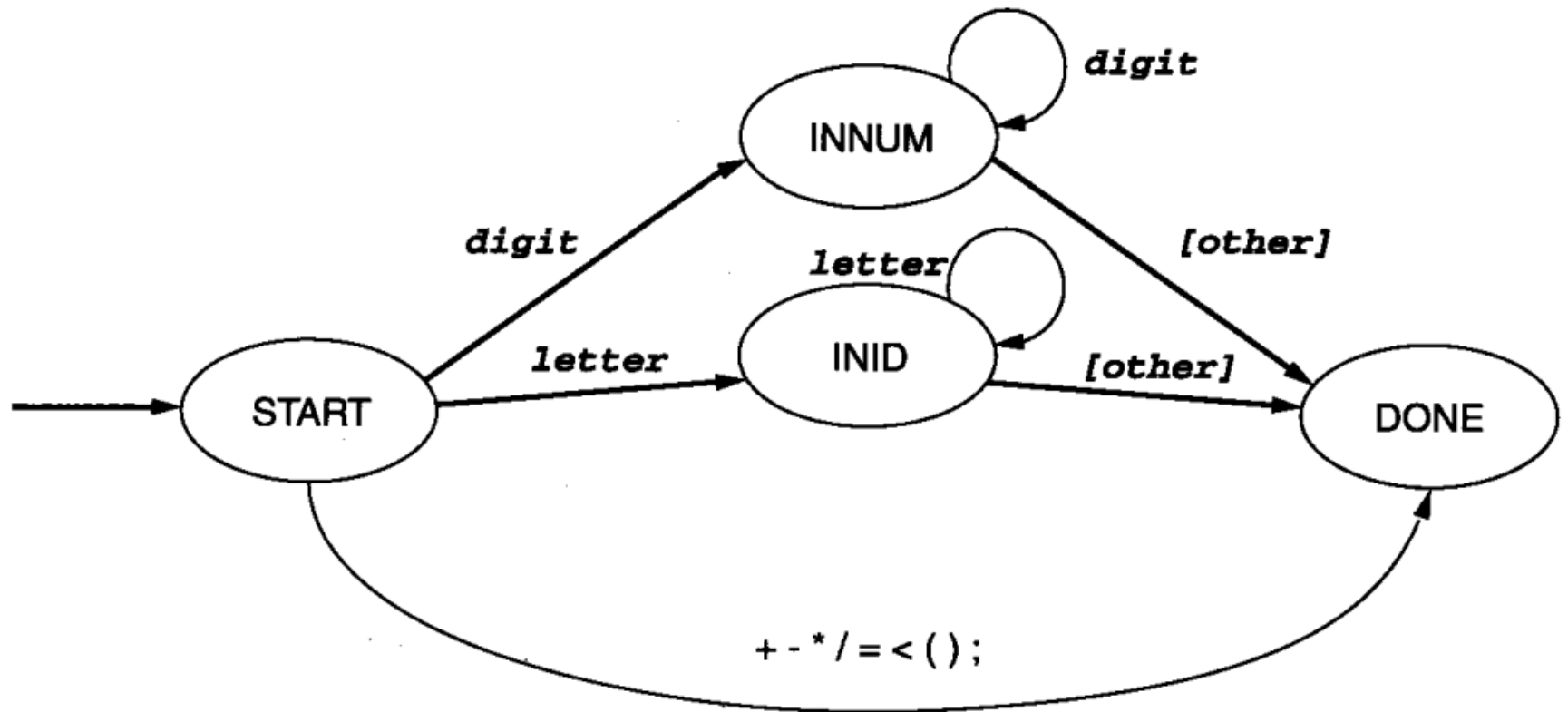
digit = 0|..|9

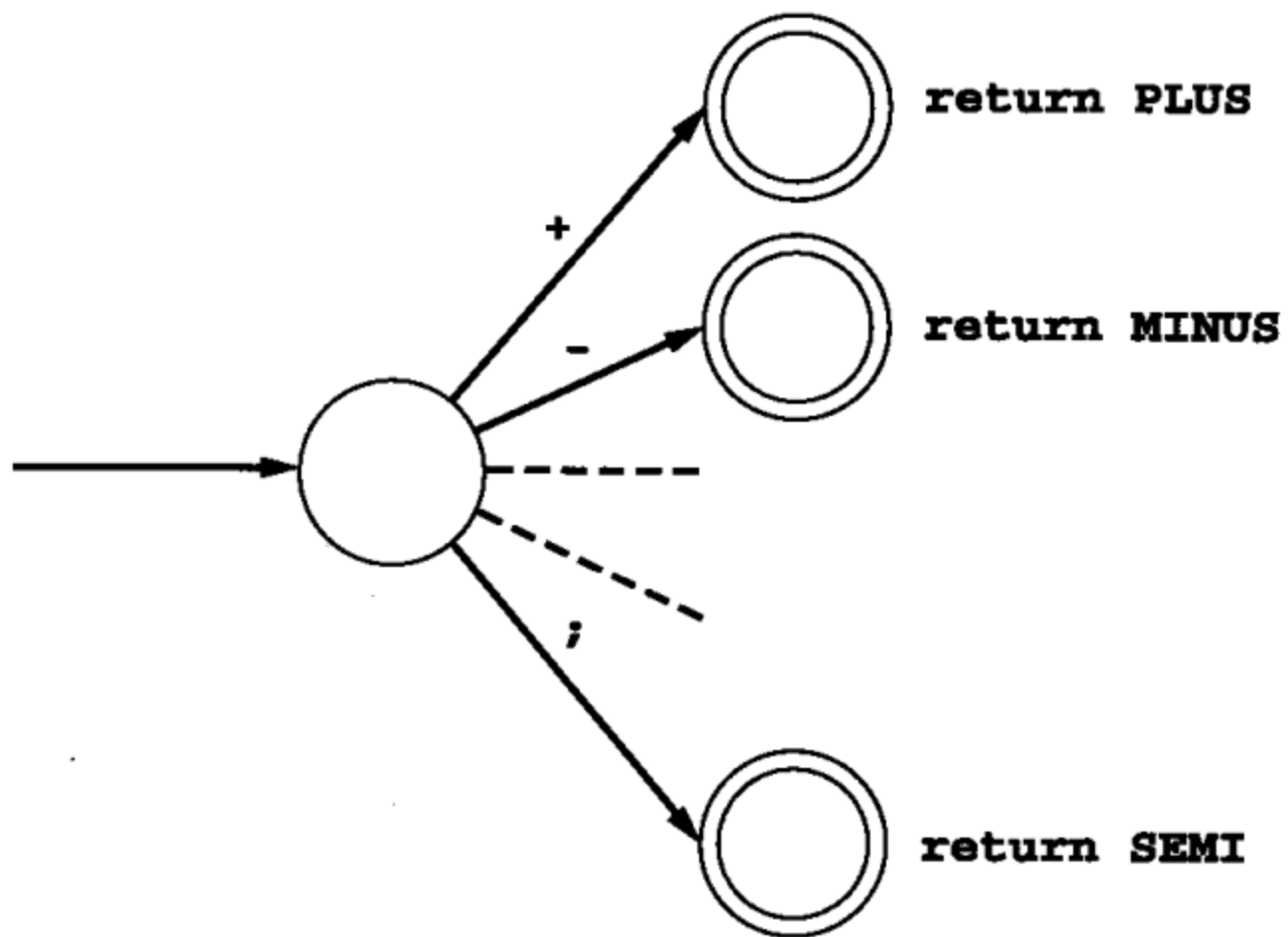
(4). 空格由空白、换行符和制表符组成。

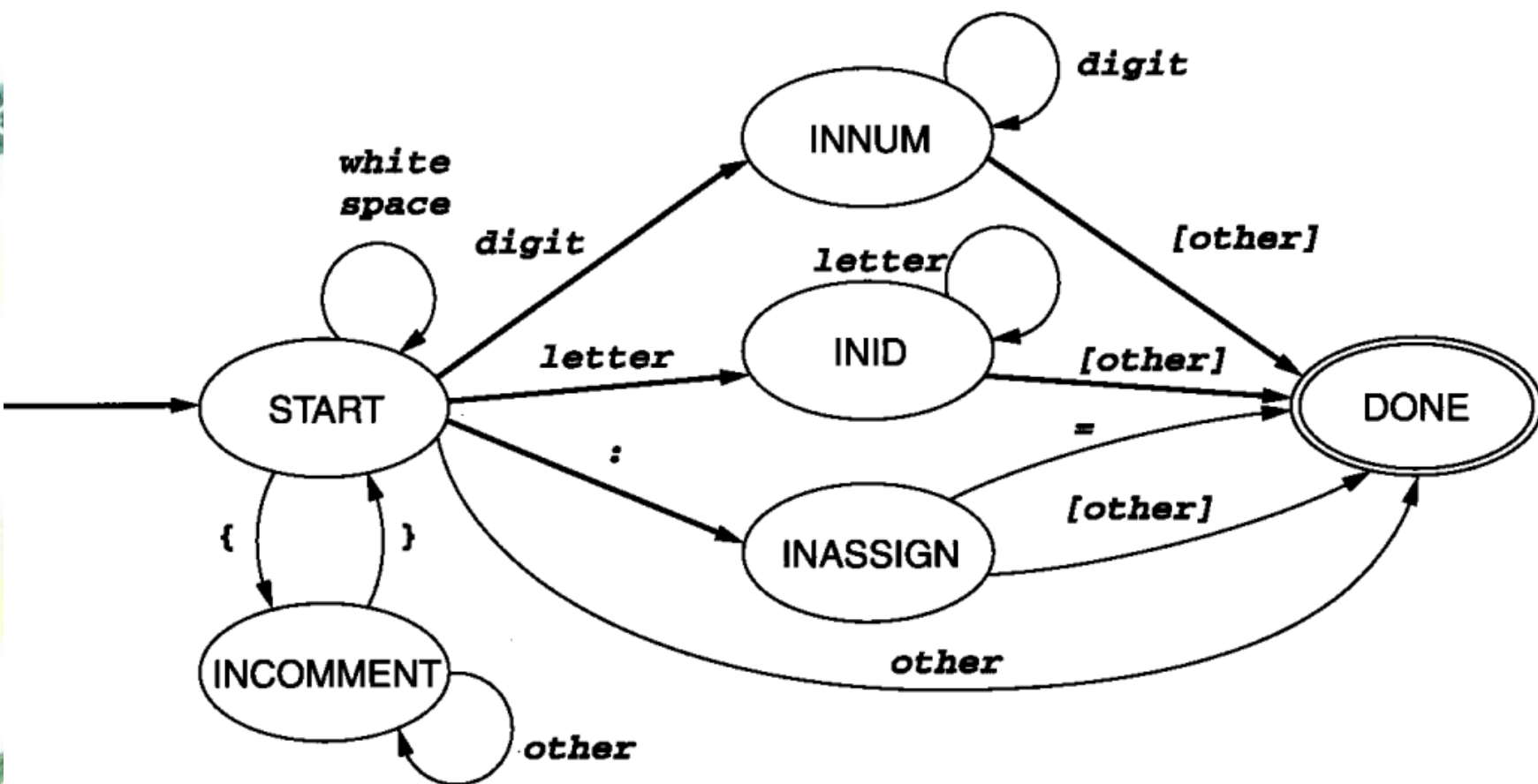
(5). 用表示{ }注释, 注释可以放在任何空白出现的位置(即注释不能放在标记内)上, 且可以超过一行。注释不能嵌套。

TINY语言的记号

保 留 字	特 殊 符 号	其 他
if	+	数
then	-	(1个或更多的数字)
else	*	
end	/	
repeat	=	
until	<	标识符
read	((1个或更多的字母)
write)	
	;	
	:=	





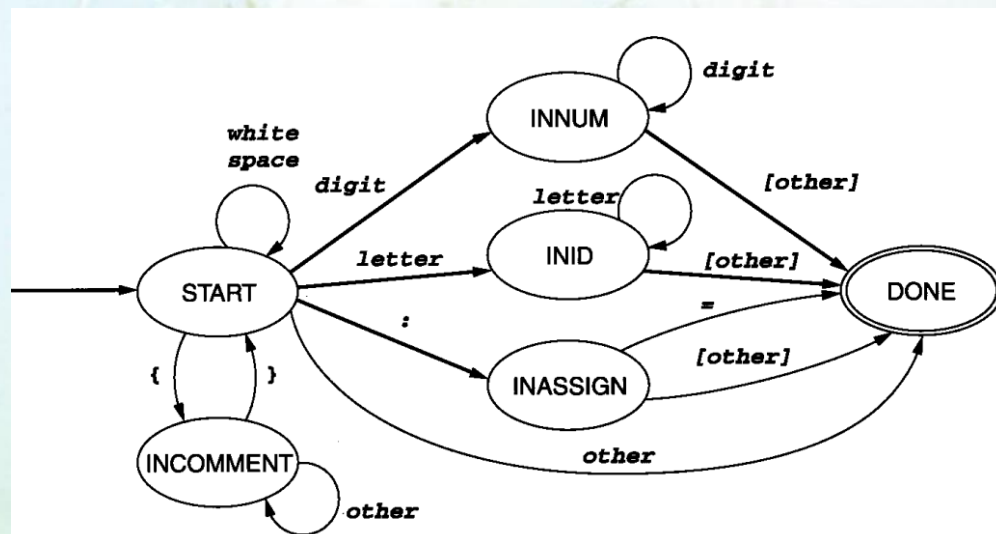


TINY扫描程序的DFA

设计TINY语言的词法分析程序

scan.c

```
TokenType getToken(void)
{
    int tokenStringIndex = 0;
    TokenType currentToken;
    StateType state = START;
    int save;
    while (state != DONE)
    { int c = getNextChar();
      save = TRUE;
      switch (state)
      { case START:
        if (isdigit(c))
            state = INNUM;
        else if (isalpha(c))
            state = INID;
        else if (c == ':')
            state = INASSIGN;
        else if ((c == ' ') || (c == '\t') || (c == '\n'))
            save = FALSE;
        else
            state = INCOMMENT;
        break;
        case INCOMMENT:
            if (c == '{')
                state = START;
            else
                state = INCOMMENT;
            break;
        case INNUM:
            if (isdigit(c))
                state = INNUM;
            else
                state = DONE;
            break;
        case INID:
            if (isalpha(c))
                state = INID;
            else
                state = DONE;
            break;
        case INASSIGN:
            if (c == '=')
                state = DONE;
            else
                state = DONE;
            break;
        case DONE:
            break;
      }
    }
    return currentToken;
}
```

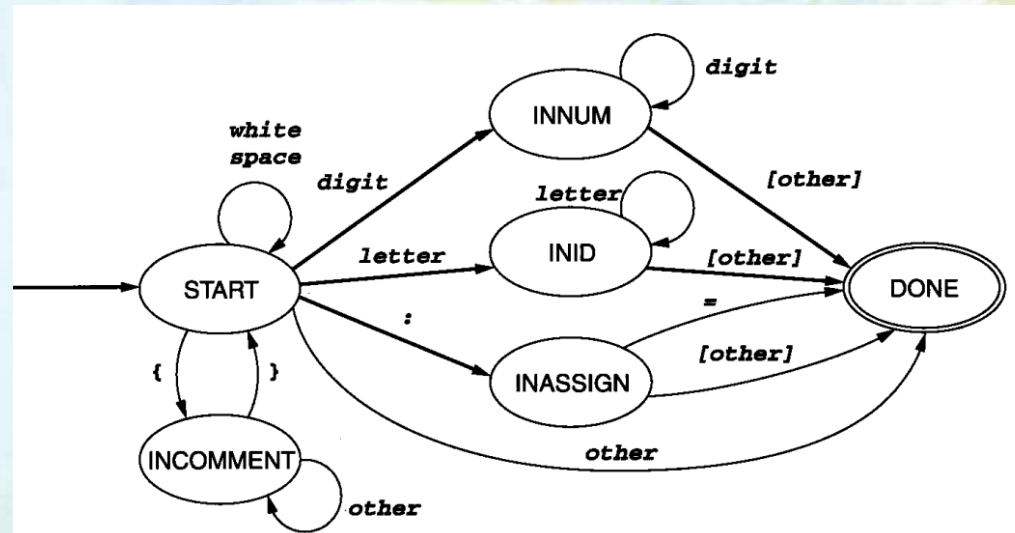


TINY扫描程序的DFA

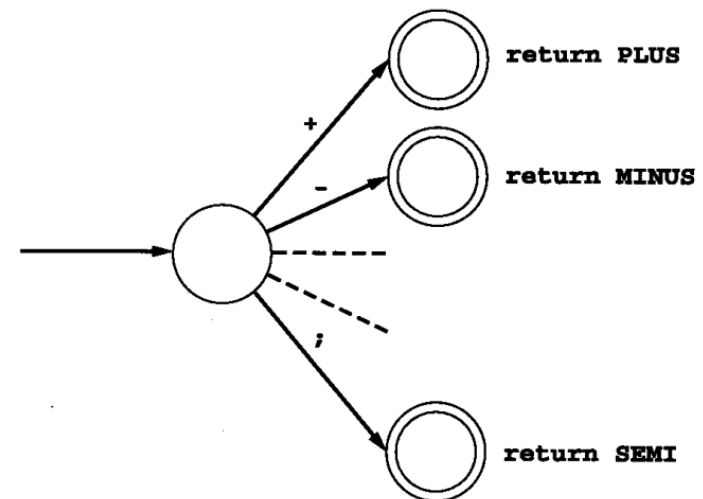
```

else if (c == '{')
{ save = FALSE;
  state = INCOMMENT;
}
else
{ state = DONE;
  switch (c)
  { case EOF:
    save = FALSE;
    currentToken = ENDFILE;
    break;
    case '=':
    currentToken = EQ;
    break;
    case '<':
    currentToken = LT;
    break;
    case '+':
    currentToken = PLUS;
    break;
    case '-':
    currentToken = MINUS;
    break;

```



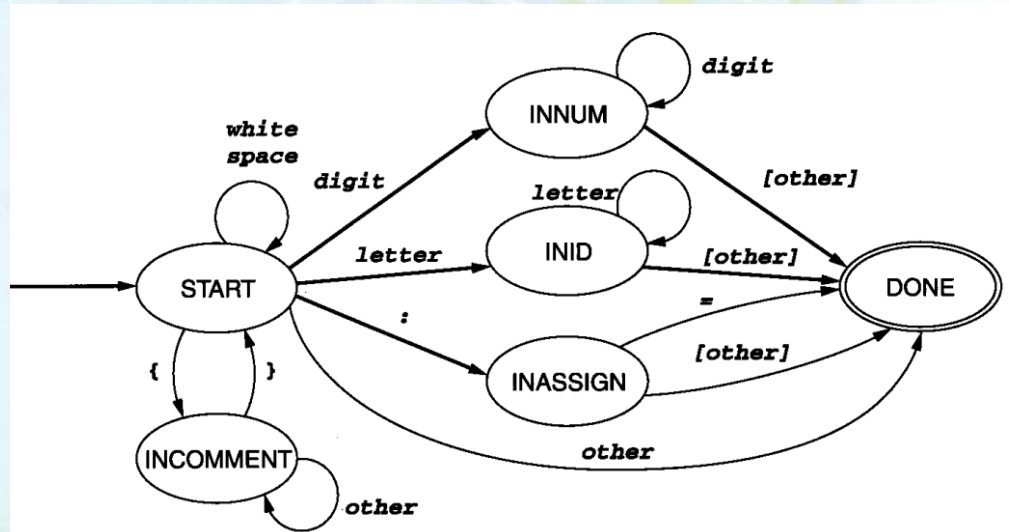
TINY扫描程序的DFA



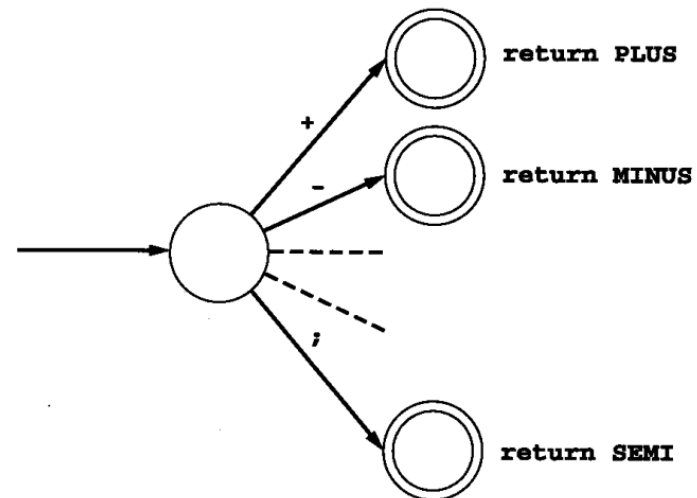
```

case '*':
    currentToken = TIMES;
    break;
case '/':
    currentToken = OVER;
    break;
case '(':
    currentToken = LPAREN;
    break;
case ')':
    currentToken = RPAREN;
    break;
case ';':
    currentToken = SEMI;
    break;
default:
    currentToken = ERROR;
    break;
}
}
break;

```



TINY扫描程序的DFA



- 利用Lex自动生成扫描程序

学习总结

1.正则表达式

2.有穷自动机

3.正则表达式→有穷自动机

正则表达式→NFA→DFA→DFA最小化

4.有穷自动机→程序

5.TINY语言词法分析程序的实现过程。