第三章 词法分析

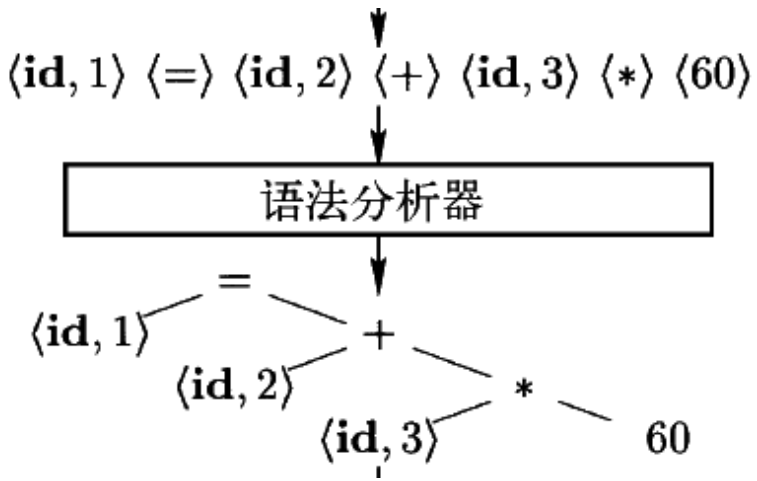
陈 林

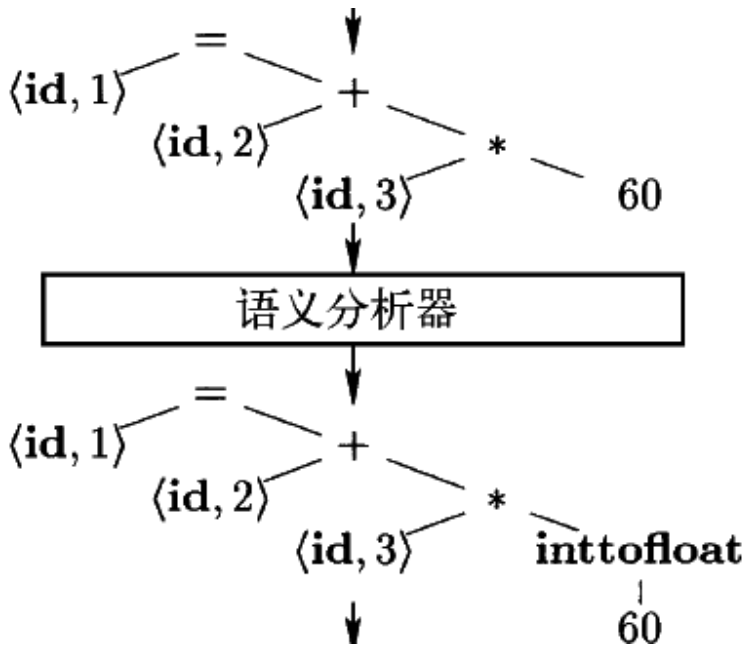
一个实例

- position = initial + rate * 60

词法分析

<id, 1> <=, > <id, 2> <+, > <id, 3> <*, > <number, 4>





提纲

- 词法分析的作用
- 词法单元的规约（正则表达式）
- 词法单元的识别（状态转换图/有穷自动机）
- 词法分析器生成工具及设计

词法分析器作用

- 词法分析是读入源程序的输入字符、将它们组成词素，生成并输出一个词法单元序列，每个词法单元对应于一个词素
- 常见的做法
  - 由语法分析器调用，需要的时候不断读取、生成词法单元
  - 可以避免额外的输入输出
- 在识别出词法单元之外，还会完成一些不需要生成词法单元的简单处理，比如删除注释、将多个连续的空白字符压缩成一个字符等

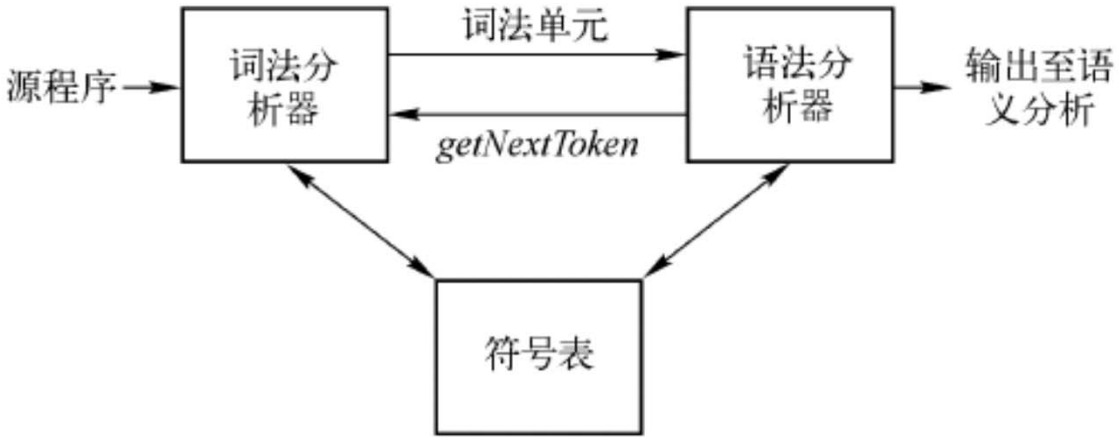


图 3-1　词法分析器与语法分析器之间的交互

词法分析相关概念

- 词素（Lexeme）
  - 源程序中的字符序列，它和某类词法单元的模式匹配，被词法分析器识别为该词法单元的实例。

- 词法单元（Token）：
  - 包含单元名（Token-name）和可选的属性值(attribute-value)
  - 单元名是表示某种词法单位抽象符号。语法分析器通过单元名即可确定词法单元序列的结构。

词法单元示例

词法单元	非正式描述	词素示例
if	字符 i, f	if
else	字符 e, l, s, e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=, !=
id	字母开头的字母 / 数字串	Pi, score, D2
number	任何数字常量	3.14159, 0, 6.02e23
literal	在两个 "之间，除" 以外的任何字符	"core dumped"

图 3-2　词法单元的例子

```
printf("Total = % d \n",sclore);
```

词法单元的属性

- 一个模式匹配多个词素时，必须通过属性来传递附加的信息。属性值将被用于语义分析、代码生成等阶段。
- 不同的目的需要不同的属性。因此，属性值通常是一个结构化数据。
- 词法单元id的属性
  - 词素、类型、第一次出现的位置、…

词法单元示例（名和属性值）

```
E = M * C ** 2
```

<**id**, 指向符号表中 E 的条目的指针 >
<**assign_op**>
<**id**, 指向符号表中 M 的条目的指针 >
<**mult_op**>
<**id**, 指向符号表中 C的条目的指针>
<**exp_op**>
<**number**, 整数值 2>

specification

词素

第一步：如何描述词素？

词法单元

模式（Pattern）

词法单元对应的词素可能具有的形式

可以用正则表达式来表示

提纲

- 词法分析的作用
- 词法单元的规约（正则表达式）
- 词法单元的识别（有穷自动机）
- 词法分析器生成工具及设计

相关概念

- 字母表：一个有限的符号集合
  - 二进制{0，1}
  - ASCII
  - Unicode
  - 典型的字母表包括字母、数位和标点符号
- 串：字母表中符号组成的一个有穷序列
  - 串s的长度 $|s|$
  - 空串ε，长度为0的串
- 语言：给定字母表上一个任意的可数的串的集合
  - 语法正确的C程序的集合，英语，汉语

相关概念

- 和串有关的术语（banana）
  - 前缀：从串的尾部删除0个或多个符号后得到的串（ban、banana、 ε）
  - 后缀：从串的开始处删除0个或多个符号后得到的串（nana、banana、ε）
  - 子串：删除串的某个前缀和某个后缀得到的串（banana、nan、 ε）
  - 真前缀、真后缀、真子串：既不等于原串，也不等于空串的前缀、后缀、子串
  - 子序列：从原串中删除0个或者多个符号后得到的串（baan）

相关概念

- 串的运算
  - 连接(concatenation)：x和y的连接时把y附加到x的后面形成的串，记作xy。
    - x=dog，y=house，xy=doghouse
  - 指数运算（幂运算）：$s^0=\varepsilon$，$s^1=s$，$s^i=s^{i-1}s$；
    - x=dog，$x^0=\varepsilon$，$x^1=dog$，$x^3=dogdogdog$

相关概念

- 语言上的运算

运算	定义和表示
L和M 的并	$L \cup M = \{s \mid s$ 属于L 或者s 属于 $M\}$
L和M 的连接	$LM = \{st \mid s$ 属于 L且 t 属于 $M\}$
L的 Kleene 闭包	$L^* = \cup_{i=0}^{\infty} L^i$
L的正闭包	$L^* = \cup_{i=1}^{\infty} L^i$

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

语言：串的集合

语言的一些实例

L = {A, B, ……, Z, a, b, ……, z}

D = {0, 1, ……, 9}

L U D : {A, B, ……, Z, a, b, ……, z, 0, 1, ……, 9}

LD ：520个长度为2的串的集合

L^4 ：所有由四个字母构成的串的集合

L* ：所有字母构成的集合，包括ε。

L(L U D) * :

D^+:

正则表达式

重要

- 正则表达式
  - 一种描述词素模式的重要表示方法

- 正则表达式可以高效、简洁地描述处理词法单元时用到的模式类型
- 可以描述所有通过对某个字母表上的符号应用运算符而得到的语言。其定义的集合叫做正则集合(regular set)
- 每个正则表达式r可以描述一个语言L(r)，也即其定义的正则集合
- 例如，C语言标识符的语言，可以用如下正则表达式来表示：letter_(letter_|digit)*

正则表达式

- 正则表达式可以由较小的正则表达式递归构建（字母表Σ上的正则表达式的定义）
  - 基本部分
    - ε 是一个正则表达式，L(ε)={ε}
    - 如果a是Σ上的一个符号，那么**a**是正则表达式，L(**a**)={a}
  - 归纳步骤：
    - 选择：(r) | (s)，L((r) | (s))=L(r) U L(s)；
    - 连接：(r)(s)，L((r)(s))=L(r)L(s) ；
    - 闭包：(r)*，L((r)*)=(L(r))*；
    - 括号：(r)，L((r))=L(r)
- 运算的优先级：* ＞ 连接符 ＞ |

  (a)|((b)*(c))可以改写为 a|b*c

正则表达式实例

- $\Sigma=\{a,b\}$
- $L(a|b) = \{a,b\}$
- $L((a|b)(a|b)) = \{aa,ab,ba,bb\}$
- $L(a^*) = \{\varepsilon,a,aa,aaa,aaaa,\ldots\ldots\}$
- $L((a|b)^*) = \{\varepsilon,a,b,aa,ab,ba,bb, aaa,aab,\ldots\ldots\}$
- $L(a|a^*b) = \{a,b,ab,aab,aaab,\ldots\}$
- P78, 练习3.3.2
  - $((\varepsilon|a)b^*)^*$

正则表达式的性质

- 等价性
  - 如果两个正则表达式r和s表示同样的语言，则$r=s$
- 代数定律

定律	描述
$r\|s = s\|r$	|是可以交换的
$r\|(s\|t) = (r\|s)\|t$	|是可结合的
$r(st) = (rs)t$	连接是可结合的
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	连接对|是可分配的
$\epsilon r = r\epsilon = r$	ϵ是连接的单位元
$r^* = (r\|\epsilon)^*$	闭包中一定包含ϵ
$r^{**} = r^*$	* 具有幂等性

正则定义

- 对正则表达式命名，使表示简洁

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

- 各个d_i不在字母表Σ中，且名字都不同
- 每个r_i都是$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$上的正则表达式

正则定义

- 各个d_i在Σ上的正则表达式如下：
  - d_1的正则表达式即r_1
  - 将r_2中的d_1替换为r_1，得到d_2的正则表达式
  - … … … …
  - 将r_i中的$d_1,d_2,\ldots,d_{i-1}$替换为各自的正则表达式，得到d_i的正则表达式

- 注意：替换的时候不能破坏替换进去的d_i的完整性

正则定义实例

- C语言的标识符集合

letter $\rightarrow$ A | B | ... | Z | a | b | ... | z | _

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

id $\rightarrow$ letter(letter|digit)*

正则定义实例

- Pascal无符号数集合，例如：1946, 11.28, 63.6E8, 1.99E-6

digit → 0 | 1 | ... | 9

digits → digit digit*

optional_fraction → .digits|ε

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

num → digits optional_fraction optional_exponent

正则表达式的扩展

- 基本运算符：并　连接　闭包
- 扩展运算符
  - 一个或多个：r^+，等价于rr*
  - 零个或一个：　r?，等价于ε|r
  - 字符类　[abc]等价于a|b|c，[a-z]等价于a|b|...|z
- 前面两个例子的简化表示

$$
\begin{aligned}
\mathit{letter\_} &\rightarrow [\texttt{A-Za-z\_}] \\
\mathit{digit} &\rightarrow [\texttt{0-9}] \\
\mathit{id} &\rightarrow \mathit{letter\_}\ (\ \mathit{letter\_} \mid \mathit{digit}\ )^*
\end{aligned}
$$

$$
\begin{aligned}
\mathit{digit} &\rightarrow [\texttt{0-9}] \\
\mathit{digits} &\rightarrow \mathit{digit}^+ \\
\mathit{number} &\rightarrow \mathit{digits}\ (\ .\ \ \mathit{digits})?\ \ (\ \texttt{E}\ [\texttt{+-}]?\ \ \mathit{digits}\ )?
\end{aligned}
$$

表达式	匹配	例子
c	单个非运算符字符 c	a
\c	字符 c 的字面值	*
“s”	串 s 的字面值	“**”
.	除换行以外的任何字符	a.*b
^	一行的开始	^abc
$	行的结尾	abc$
[s]	字符串 s 中的任何一个字符	[abc]
[^s]	不在串 s 中的任何一个字符	[^abc]
r*	由和 r 匹配的零个或多个串连接成的串	a*
r^+	由和 r 匹配的一个或多个串连接成的串	a+
r?	零个或一个 r	a?
r{m,n}	最少 m 个，最多 n 个 r 的连接	a{1,5}
r_1r_2	r_1后加上r_2	ab
$r_1 \mid r_2$	r_1或r_2	a|b
(r)	与 r 相同	(a|b)
r_1/r_2	后面跟有r_2时的r_1	abc/123

附注

- 正则表达式是一种描述手段，通常用来描述程序语言的词法符号。
- 很多编辑器支持正则表达式
- 工具GREP

有关实验

- 如何定义你的语言？

Tokens

INT	→	/* A sequence of digits without spaces[1] */
FLOAT	→	/* A real number consisting of digits and one decimal point. The decimal point must be surrounded by at least one digit[2] */
ID	→	/* A character string consisting of 52 upper and lower case alphabetic characters, the 10 digits and the underscore. In addition, identifiers must start with an alphabetic character[3] */
SEMI	→	;
COMMA	→	,
ASSIGNOP	→	=
RELOP	→	> | < | >= | <= | == | !=

正则表达式可以由较小的正则表达式递归构建

implementation

词素

第二步：如何识别词法单元？

词法单元

状态转换图、有限自动机

识别一个串是否属于某个正则集

提纲

- 词法分析的作用
- 词法单元的规约（正则表达式）
- 词法单元的识别（状态转换图/有穷自动机）
- 词法分析器生成工具及设计

词法单元的识别

- 词法分析器要求能够检查输入字符串，在前缀中找出和某个模式匹配的词素
  - 首先通过正则定义来描述各种词法单元的模式
  - 定义ws→(blank | tab | newline)+来消除空白
    - 词法分析器识别到这个模式时，不返回词法单元，继续识别其它模式

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

图 3-11　例 3.8 中词法单元的模式

词素	词法单元名字	属性值
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	指向符号表条目的指针
Any *number*	**number**	指向符号表条目的指针
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

图 3-12　词法单元、它们的模式以及属性值

状态转换图

- 状态转换图是词法分析器的重要组件之一
- 可以将正则表达式转换成状态转换图
- 状态转换图(transition diagram)
  - 状态(state)：表示在识别词素的过程中可能出现的情况
    - 状态看作是已处理部分的总结
    - 某些状态为接受状态或最终状态，表明已经找到词素
    - 加上*的接受状态表示最后读入的符号不在词素中
    - 开始状态（初始状态）：用start边表示
  - 边(edge)：从一个状态指向另一个状态；边的标号是一个或者多个符号
    - 如果当前符号为s，下一个输入符号为a，就沿着从s离开，标号为a的边到达下一个状态

状态转换图的例子

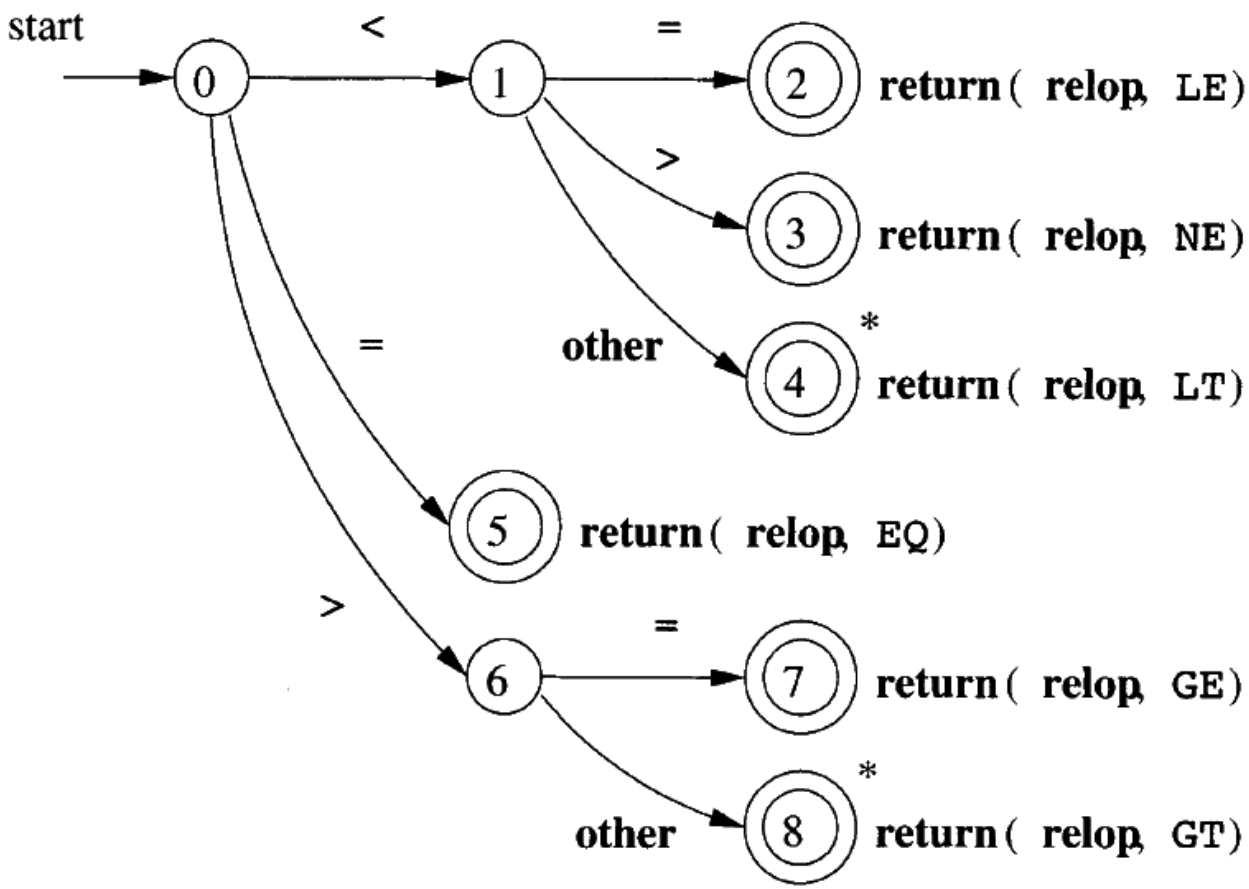


图 3-13　词法单元 **relop** 的状态转换图

保留字和标识符的识别

- 在很多程序设计语言中，保留字也符合标识符的模式，识别标识符的状态转换图也会识别 保留字
- 解决方法
  - 在符号表中预先填写保留字，并指明它们不是普通标识符。
  - 为关键字/保留字建立单独的状态转换图。并设定保留字的优先级高于标识符

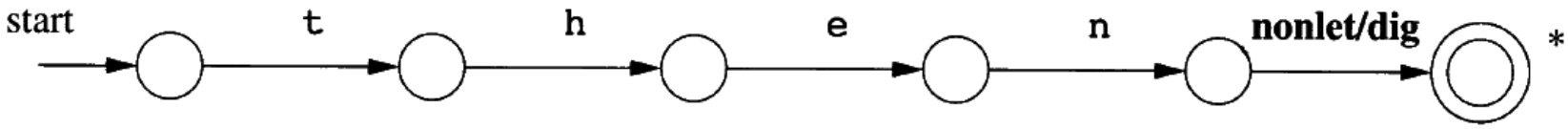


图 3-15　假想的关键字 then 的状态转换图

其它的状态转换图

$$
\begin{aligned}
digit &\rightarrow [0\text{-}9] \\
digits &\rightarrow digit^{+} \\
number &\rightarrow digits\ (.\ digits)?\ (\ \mathtt{E}\ [+-]?\ digits\ )?
\end{aligned}
$$

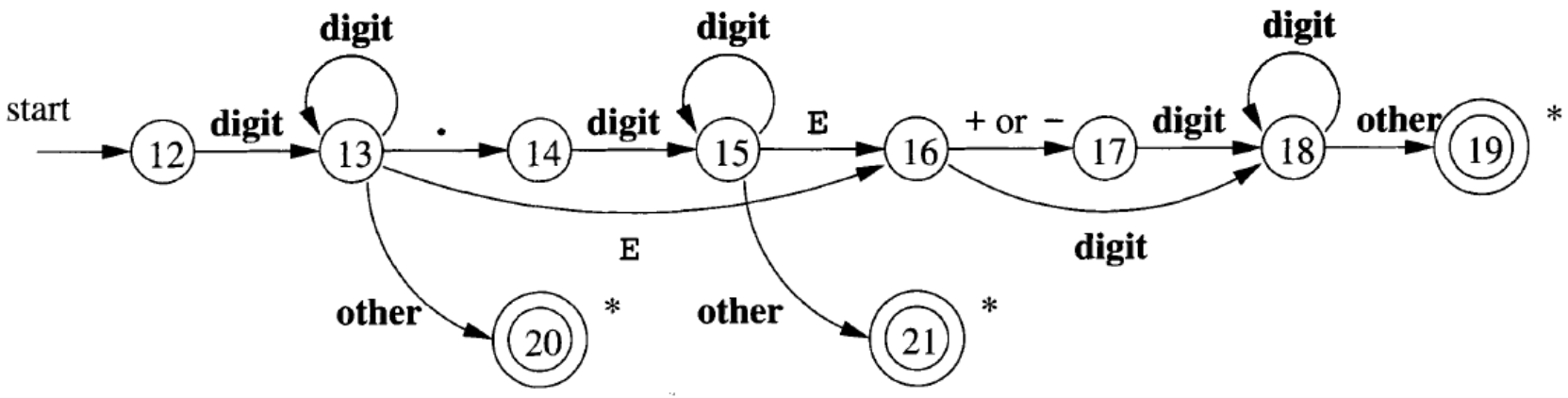


图 3-16　无符号数字的状态转换图

其它的状态转换图

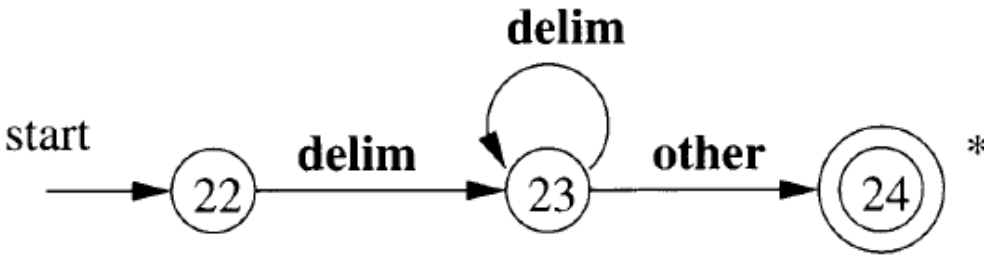


图 3-17　空白符的状态转换图

提纲

- 词法分析的作用
- 词法单元的规约（正则表达式）
- 词法单元的识别（状态转换图/有穷自动机）
- 词法分析器生成工具及设计

Finite-state machines

- Finite-state machines can model a large number of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines are sometimes used to describe neurological systems, and in linguistics they can be used to describe the grammars of natural languages.

有穷自动机

重要

- 本质上等价于状态转换图
- 区别在于：
  - 自动机是识别器，对每个输入串回答yes or no
- 分为两类
  - 不确定的有穷自动机（Nondeterministic Finite Automate，NFA）
  - 确定的有穷状态自动机（Deterministic Finite Automate，DFA）

不确定 vs. 确定

- 不同：
  - NFA：一个符号标记离开同一状态的多条边
  - DFA：对于每个状态和字母表中的每个字符，有且仅有一条离开该状态、以该符合为标号的边

  - NFA：可以有边的标号是ε
  - DFA：没有标记为ε的边

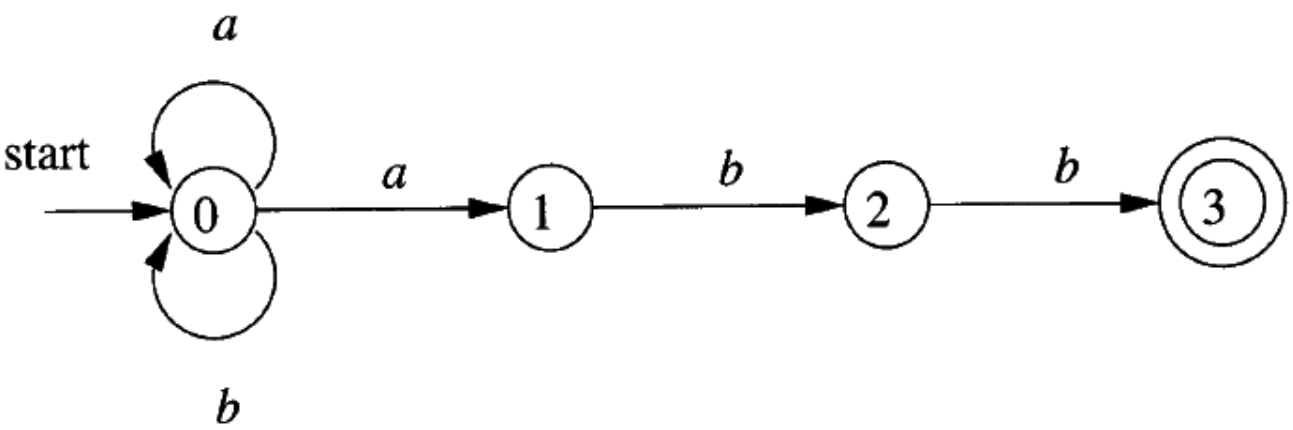
- 相同：都可以识别正则语言，两者之间存在等价性

不确定的有穷自动机（NFA）

- NFA由以下几部分组成
  - 一个有穷的状态集合S
  - 一个输入符号集合Σ（input alphabet）
  - 转换函数（transition function）对于每个状态和$\Sigma \cup \{\varepsilon\}$中的符号，给出相应的后继状态集合
  - 一个状态S_0被指定为开始状态/初始状态
  - S的一个子集F被指定为接受状态

NFA的例子

- 状态集合S={0, 1, 2, 3}
- 开始状态0
- 接受状态集合{3}
- 转换函数：
  - (0, a)→{0, 1}　　(0, b)→{0}　　(1, b)→2
    (2, b)→3



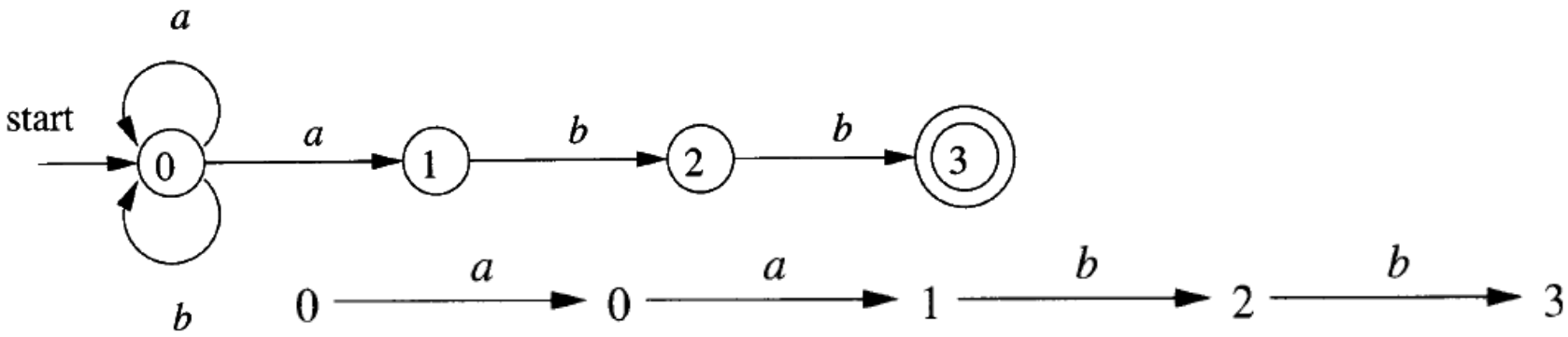
转换表

- NFA可以表示为一个转换表
  - 表的各行对应于状态
  - 各列对应于输入符号和ε
  - 表中的元素表示给定状态在给定输入下的后继状态

状态	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

自动机对输入字符串的接受

- 一个NFA接受输入字符串x，当且仅当对应的转换图中存在一条从开始状态到某个接受状态的路径，使得该路径中各条边上的标号组成符号串x （路径中可能包含ε 边）
- 下图对应的NFA能够接受aabb

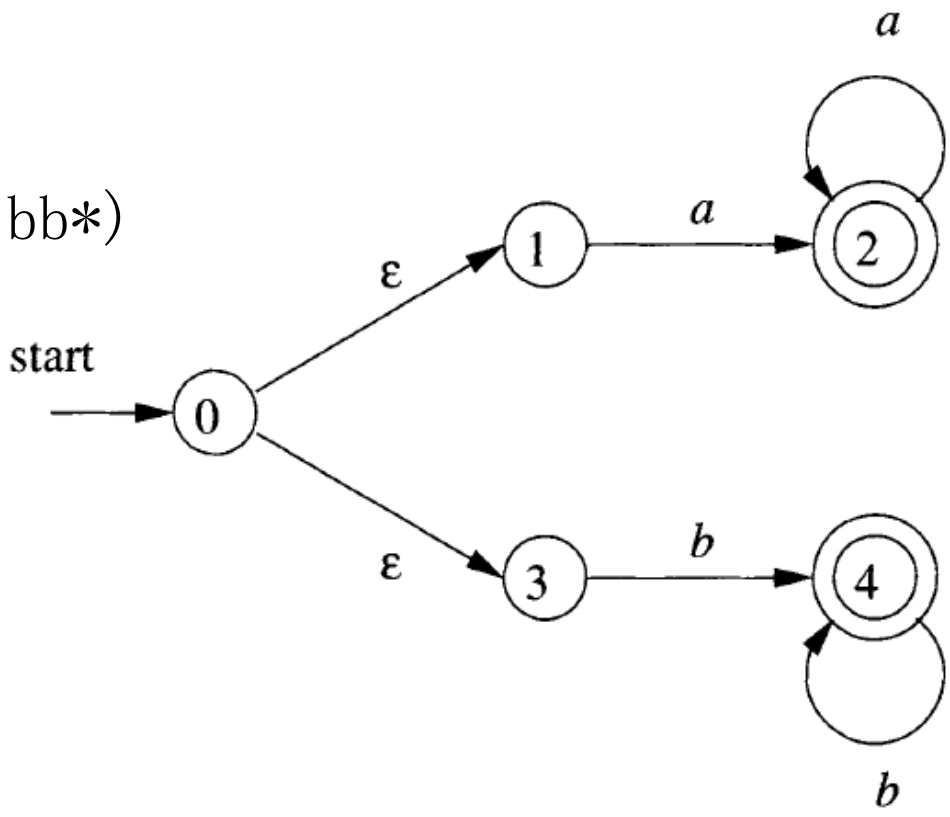


- 注意：只要存在从开始状态到接受状态的路径，符号串就认为被NFA接受

自动机与语言

- 由一个NFA *A*定义（接受）的语言是从开始状态到某个接受状态的所有路径上的符号串集合，称为*L(A)*

相应的语言：L(aa*|bb*)



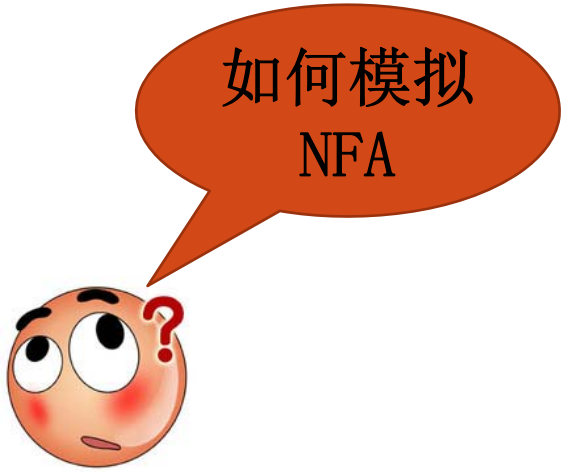
确定有穷自动机（DFA）

- 一个NFA被称为DFA，如果
  - 没有ε之上的转换动作
  - 对于每个状态s和每个输入符号a，有且只有一条标号为a的边
- 可以高效判断一个串能否被一个DFA接受
- 每个NFA都有一个等价的DFA，即它们接受同样的语言

DFA的模拟

- 假设输入符号就是字符串中的符号；
- Nextchar读入下一个字符（符号）
- move给出了离开s，标号为c的边的目标状态

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s在F中 ) return "yes";
else return "no";
```



DFA的例子

- 假设输入为ababb，那么进入的状态序列为0, 1, 2, 1, 2, 3，返回yes

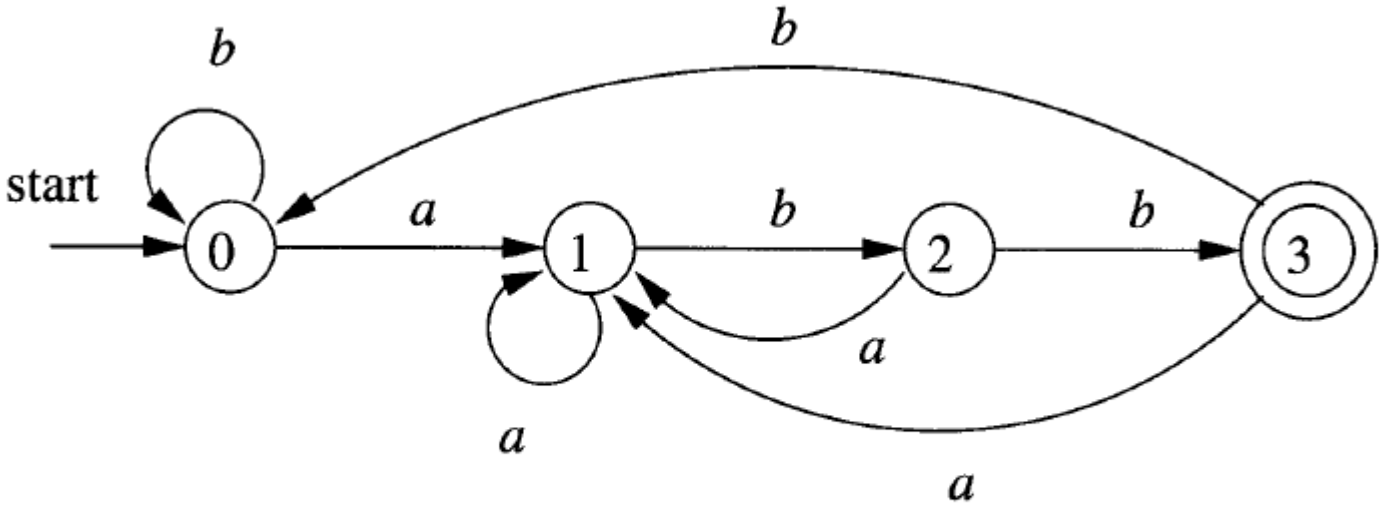


图 3-28　接受(**a**|**b**)＊**abb** 的 DFA

词素

词法单元

我们已有的工具：

1. RE
2. DFA
3. NFA

正则表达式到自动机

- 正则表达式可以简洁、精确地描述词法单元的模式但是在进行模式匹配时需要模拟DFA的执行
- 因此，需要将正则表达式转换为DFA
- 步骤：
  - 正则表达式到NFA
  - NFA到DFA

NFA转换成DFA – 子集构造法

- 对NFA的模拟往往不如对DFA的模拟直接，除非转换花费更多的时间
- 基本思想： DFA每个状态 ←→ NFA一个状态集
  - “并行地模拟” NFA在遇到一个给定输入串时可能执行的所有动作
  - 构造得到的DFA的每个状态和NFA的状态子集对应
  - DFA读入$a_1, a_2, \cdots, a_n$后到达的状态对应于从NFA开始状态出发沿着$a_1, a_2, \cdots, a_n$可能到达的状态集合
- 理论上，最坏情况下DFA的状态个数会是NFA状态个数的指数多个。但是对于大部分应用，NFA和相应的DFA的状态数量大致相同

NFA转换成DFA – 子集构造法

- 输入：一个NFA N
- 输出：一个接受相同语言的DFA D

s表示N中的单个状态，T代表N的一个状态集

操作	描述
ϵ-$closure(s)$	能够从NFA的状态 s 开始只通过ϵ转换到达的 NFA状态集合
ϵ-$closure(T)$	能够从T中某个NFA状态 s 开始只通过ϵ 转换到达的NFA状态集合，即$\cup_{s\epsilon T}\ \epsilon$-$closure(s)$.
$move(T,a)$	能够从 T 中某个状态 s 出发通过标号为a的转换到达的NFA状态的集合

图 3-31　NFA 状态集上的操作

NFA转换成DFA – 子集构造法

- D的开始状态是ε-closure(s_0)，D的接受状态是所有至少包含了N的一个接受状态的状态集合。

```
一开始, ε-closure(s0)是 Dstates 中的唯一状态，且它未加标记；
while（在 Dstates 中有一个未标记状态T）{
        给T加上标记;
        for（每个输入符号 a）{
                U = ε-closure(move(T, a));
                if（U 不在Dstates中）
                        将U 加入到Dstates中，且不加标记；
                Dtran[T, a] = U;
        }
}
```

图 3-32　子集构造法

NFA转换成DFA – 子集构造法

- 对NFA的任何状态集合ε-closure(T)的计算
- 一个图搜索过程

```
将T的所有状态压入stack中；
将 ϵ-closure(T) 初始化为T;
while ( stack非空) {
        将栈顶元素t弹出栈中;
        for (每个满足如下条件的u：从t出发有一个标号为ϵ的转换到达状态u)
                if ( u不在 ϵ-closure(T)中) {
                        将u加入到ϵ-closure(T)中;
                        将u压入栈中;
                }
}
```

图 3-33　计算 ϵ-*closure*(T)

NFA到DFA转换的示例

- A：=ε-closure(0)={0,1,2,4,7}
- B：Dtran[A,a]= ε -closure(move(A,a))= ε - losure({3,8})={1,2,3,4,6,7,8}
- C：Dtran[A,b]= ε -closure(move(A,b))= ε -closure({5})={1,2,4,5,6,7}
- D：Dtran[B,b]=ε -closure(move(B,b))= {1,2,4,5,6,7,9}
- …

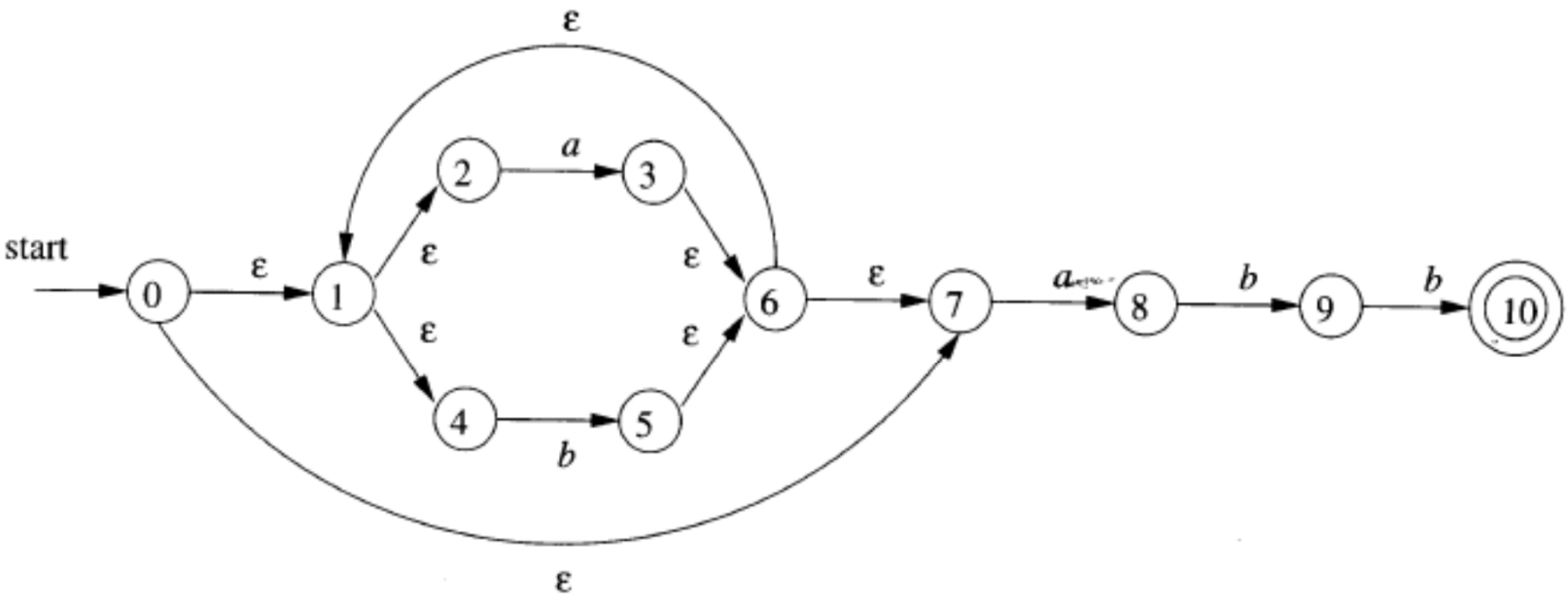


图 3-34 （a|b）* abb 对应的 NFA *N*

NFA到DFA转换的示例

- 开始状态：A
- 接受状态：E

NFA 状态	DFA 状态	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C

图 3-35　DFA D 的转换表 $Dtran$

对NFA的运行进行模拟:子集构造法

- 输入：一个以文件结束符eof结尾的输入串x，一个NFA N，其开始状态是S_0，接受状态集为F，转换函数为$move$
- 输出：如果N接受x，返回yes，否则返回no

```
S = ε-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ε-closure(move(S, c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

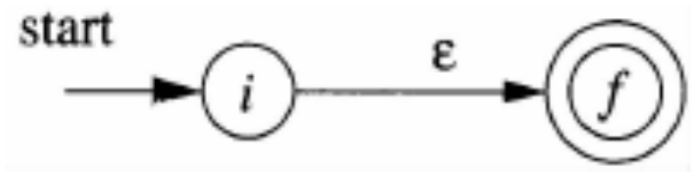
图 3-37　模拟的一个 NFA

正则表达式到NFA

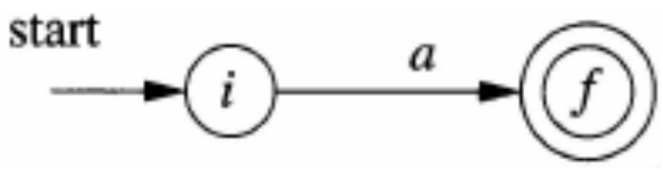
- 输入：字母表Σ上的一个正则表达式r
- 输出：一个接受$L(r)$的NFA N
- 基本思想
  - 根据正则表达式的递归定义，按照正则表达式的结构递归地构造出相应的NFA
  - 算法分成两个部分：
    - 基本规则处理ε和单符号的情况
    - 对于每个正则表达式的运算，建立构造相应NFA的方法

转换算法(1)

- 基本规则：
  - 表达式ε，



  - 表达式a，



转换算法(2)

- 归纳规则
  - 正则表达式s和t的NFA分别是 $N(s)$和$N(t)$
  - $r=s|r$， r的NFA $N(r)$

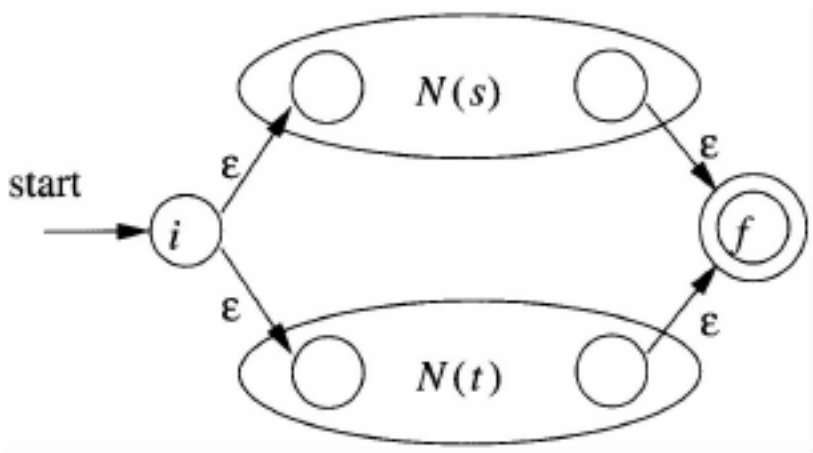


图 3-40 两个正则表达式的并的 NFA

转换算法(2)

- 归纳规则
  - 正则表达式$r=st$， $N(r)$

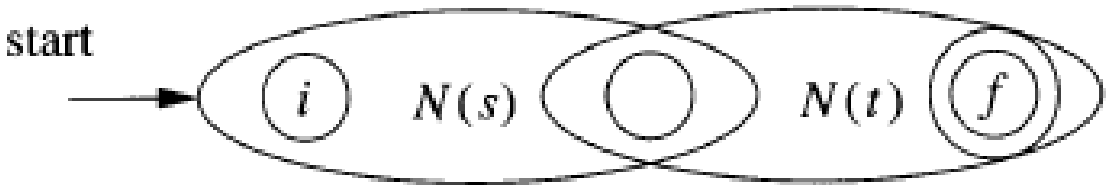


图 3-41　两个正则表达式的连接的 NFA

转换算法(3)

- 归纳规则
  - 正则表达式 $r=s*$, $N(r)$

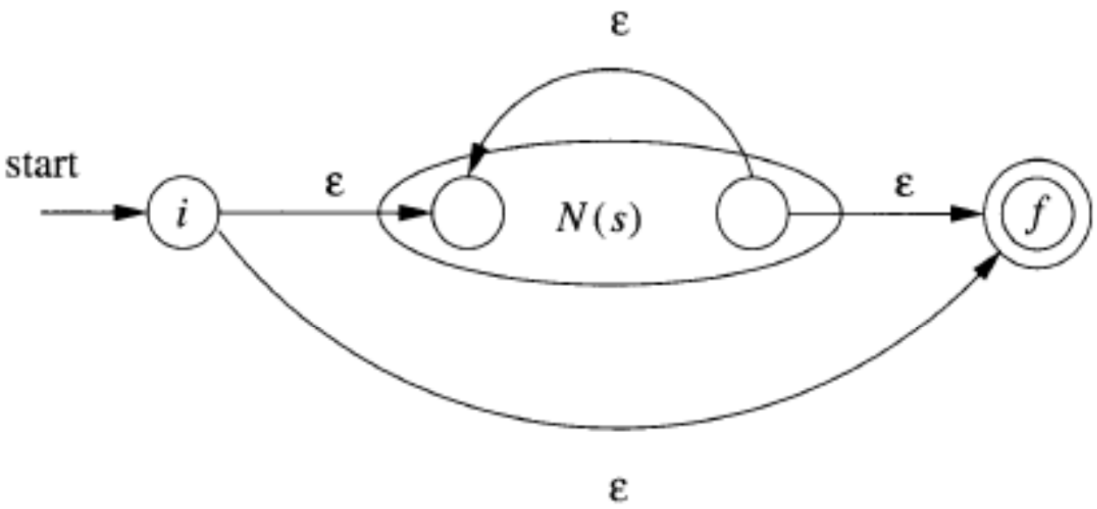
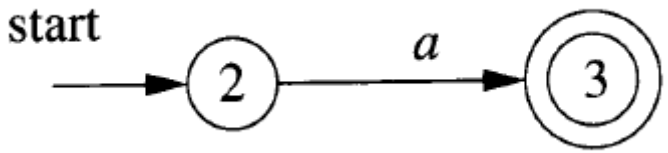


图 3-42　一个正则表达式的闭包的 NFA

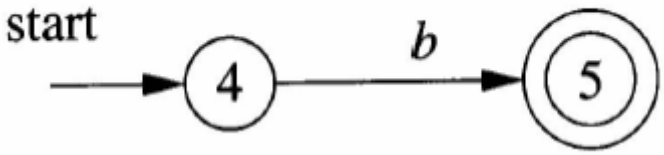
  - $r=(s)$, $N(r)=N(s)$

正则表达式到NFA的例子（1）

- 正则表达式(a|b)*abb
- 第一个a对应的NFA

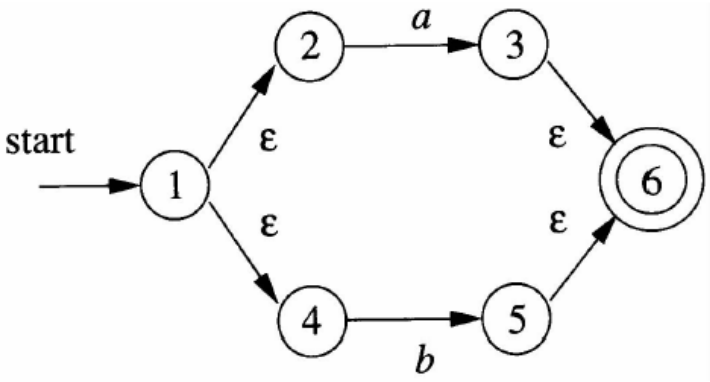


- 第一个b对应的NFA

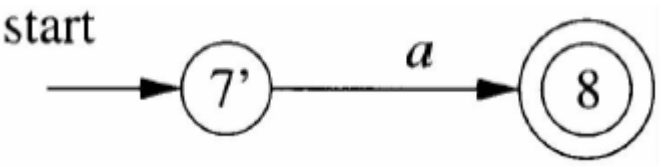


正则表达式到NFA的例子（2）

- (a|b)的NFA

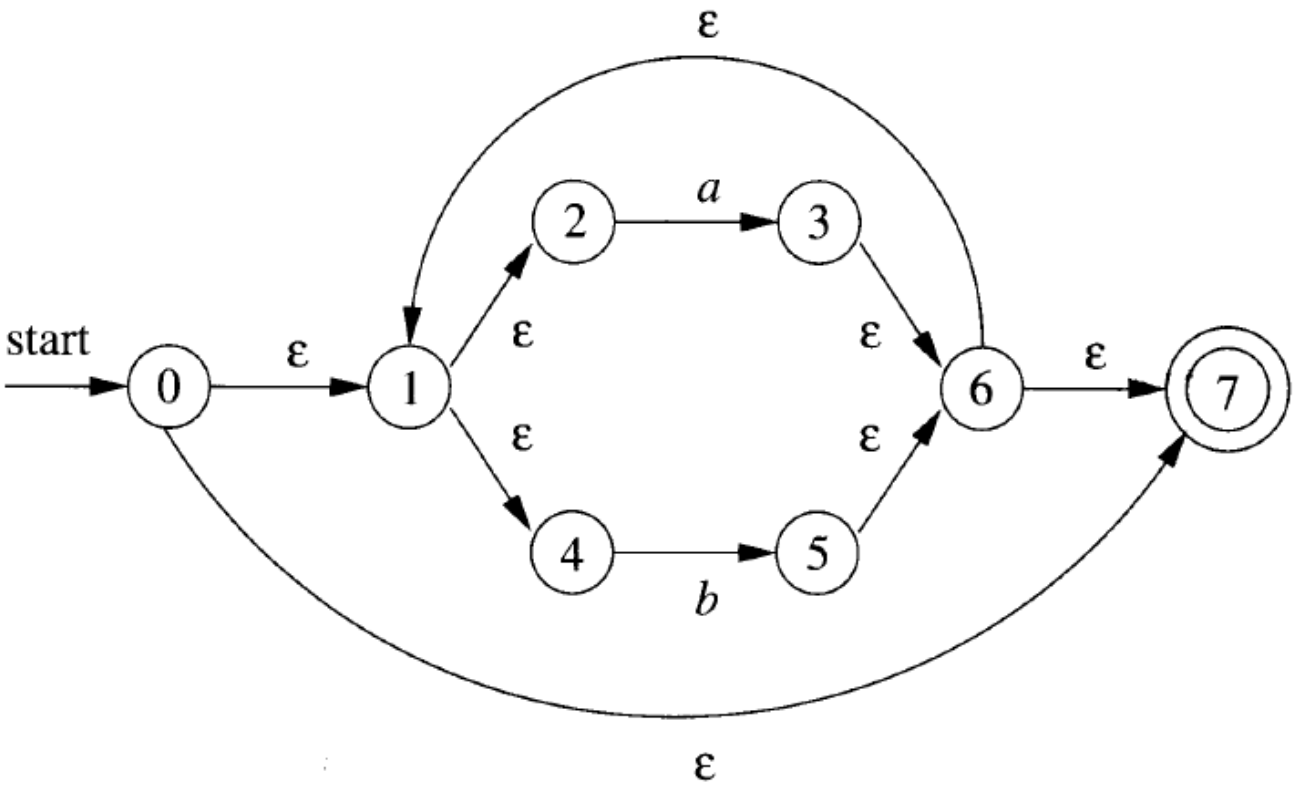


- 第二个a的NFA



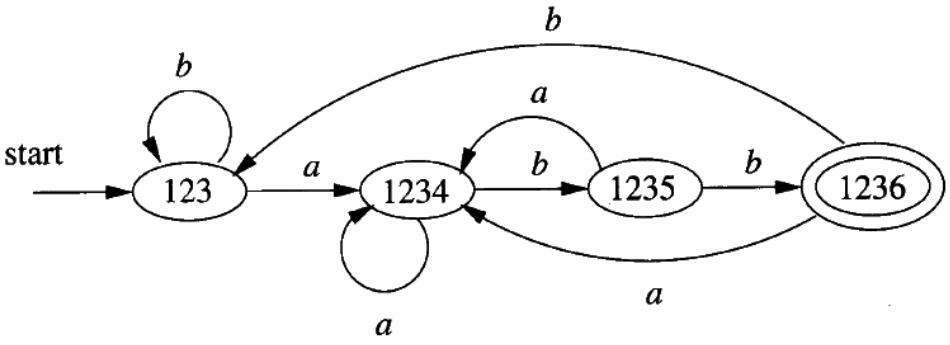
正则表达式到NFA的例子（3）

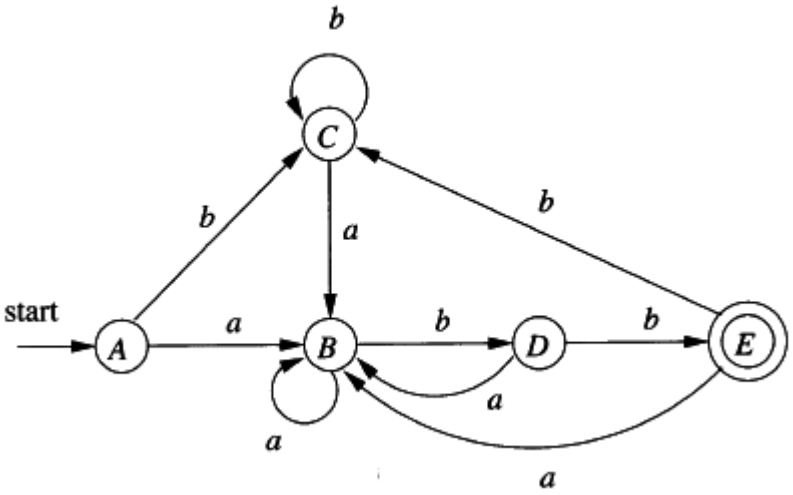
- (a|b)*的NFA



基于DFA的模式匹配器的优化

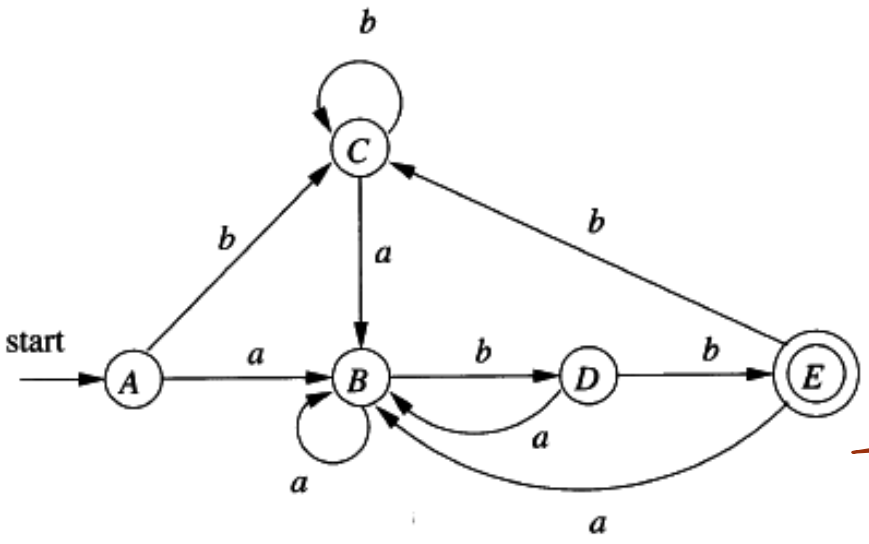
- DFA化简：状态数最小化
- 等价的DFA可能具有不同的状态个数
- 任何正则语言都有一个唯一的（不计同构）状态数目最少的DFA





DFA状态最小化

- 原理：将一个DFA的状态集合分划成多个组，每个组中的各状态之间相互不可区分，然后将每个组中的状态合并成状态最少DFA的一个状态
- 可区分的定义：
  - 如果分别从状态s和状态t出发，沿着标号为x的路径到达的两个状态只有一个是接受状态，称为x区分状态s和t
  - 如果存在能够区分s和t的串，那么它们就是可区分的



空串区分了E和其它状态
bb区分了A和B

A和C是可区分的吗？

最小化算法（分划部分）

1. 设置初始分划Π={S-F,F}
2. 迭代，不断分划：

   for（Π中的每个元素G){

   细分G，使得G中的s、t仍然在同一组中 iff

   对任意a，s,t都到达Π中的同一组；

   Π_{new}=将Π中的G替换为细分得到的小组；

   }
3. 如果Π_{new}==Π，令Π_{final}==Π，转步骤4；否则Π==Π_{new}，转步骤2；

最小化算法（构造部分）

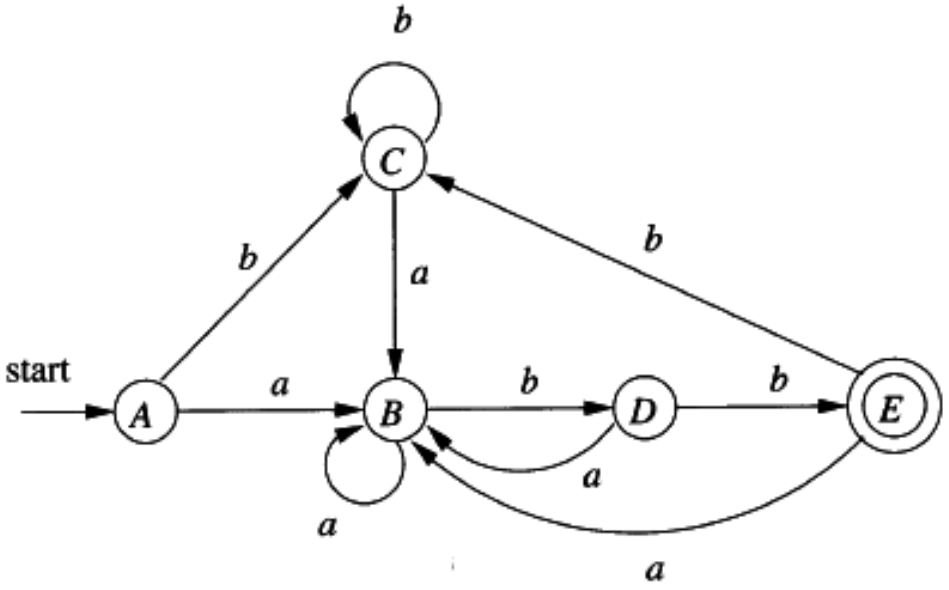
4. 在Π_{final}的每个组中选择一个状态作代表，作为最小DFA的状态
   - 开始状态就是中包含原开始状态的组的代表
   - 接受状态就是包含了原接受状态的组的代表
   - 转化关系构造如下：
     - 如果s是中G的代表，而s在a上的转换到达t，而t所在组的代表为r，那么最小DFA中有从s到r的、在a上的转换

DFA最小化的正确性证明

- 位于Π_{final}的同一组中的状态不可能被任意串区分
- Π_{final}的不同组的状态之间是可区分的

DFA最小化的例子

- 初始分划：{A,B,C,D}　　{E}
- 处理{A,B,C,D}，b把它细分为{A,B,C}　{D}
- 处理{A,B,C}，b把它细分为{A,C}　{B}
- 分划完毕。选取A,B,D和E作为代表，构造得到最小DFA



状态	a	b
A	B	A
B	B	D
D	B	E
E	B	A

图 3-65　状态最少DFA 的转换表

提纲

- 词法分析的作用
- 词法单元的规约（正则表达式）
- 词法单元的识别（状态转换图/有穷自动机）
- 词法分析器生成工具及设计

词法分析器的构造实现

- 两种方法
  - 基于词法单元的词法结构图或其它描述，手工编写代码扫描输入中的每个词素，并返回识别到的词法单元信息
  - 使用词法分析器生成工具（如lex /flex），给出描述词素的模式，利用工具编译为具有词法分析器功能的代码，高效且简单
- 正则表达式

基于状态转换图的词法分析器

- 从转换图构造词法分析器的方法
  - 变量state记录当前状态
  - 一个switch根据state的值转到相应的代码
  - 每个状态对应于一段代码
    - 这段代码根据读入的符号，确定下一个状态
    - 如果找不到相应的边，则调用fail()进行错误恢复
  - 进入某个接受状态时，返回相应的词法单元
    - 注意状态有*标记时，需要回退forward指针

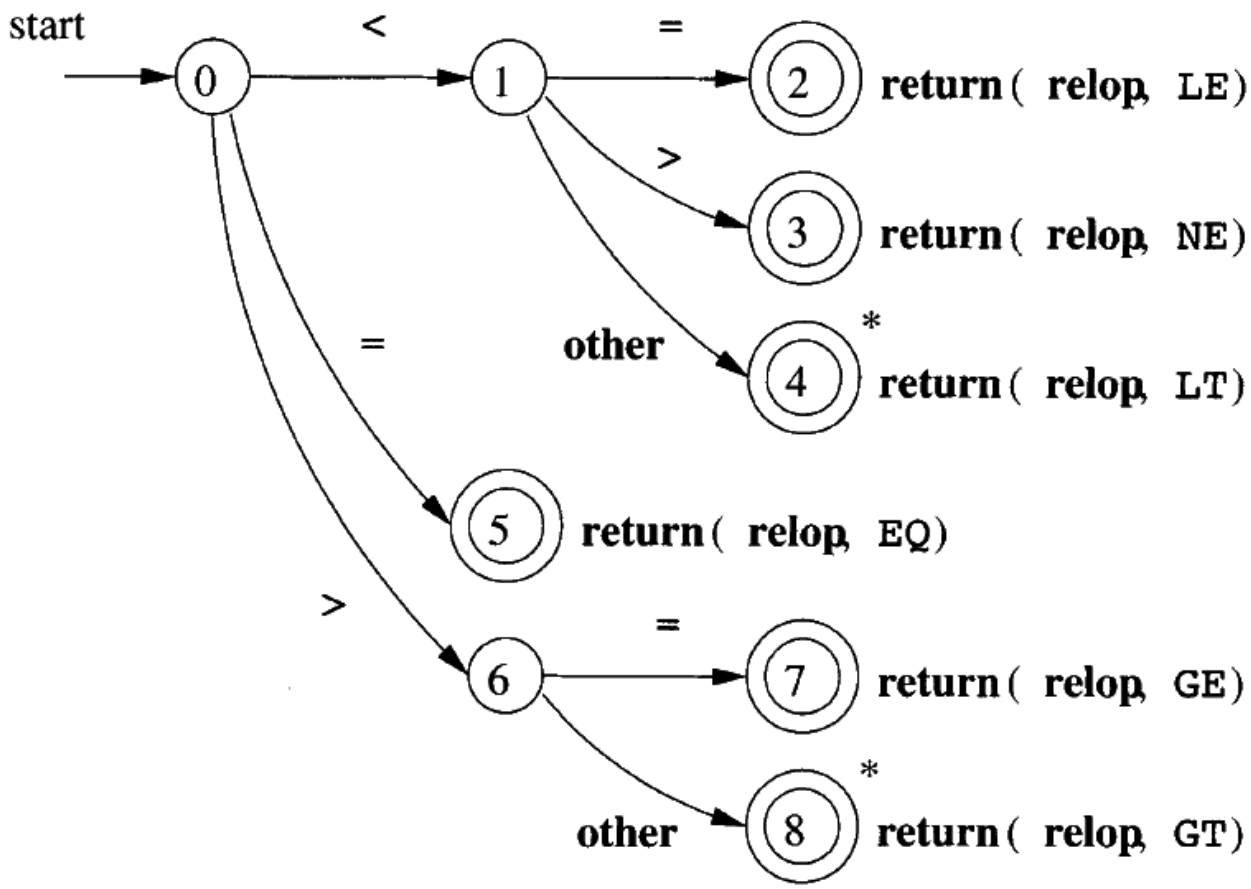


图 3-13　词法单元 **relop** 的状态转换图

Relop对应的代码概要

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

多个模式集成到词法分析器

- 方法1：词法分析器需要匹配多个模式（有多个状态转换图）
  - 顺序的尝试各个词法单元的状态转换图，如果引发fail，回退并启动下一个状态转换图
    - 次序问题
    - 优先级
- 方法2：并行的运行各个状态转换图
  - 一个图已经匹配到词素，另一个仍在继续读入
  - 取最长的和某个模式匹配的输入前缀（词法单元）
- 方法3：所有的状态转换图合并为一个图
  - 选择策略同方法2
  - 书中例子简单，因为没有两类词法单元以相同的字符开头

状态图的合并

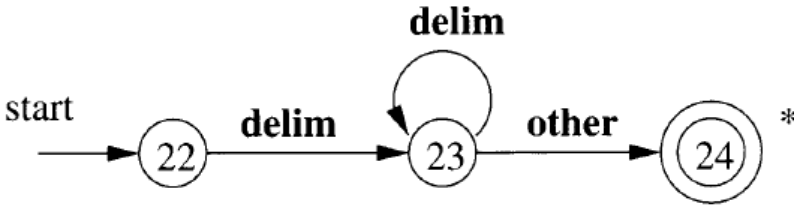


图 3-17　空白符的状态转换图

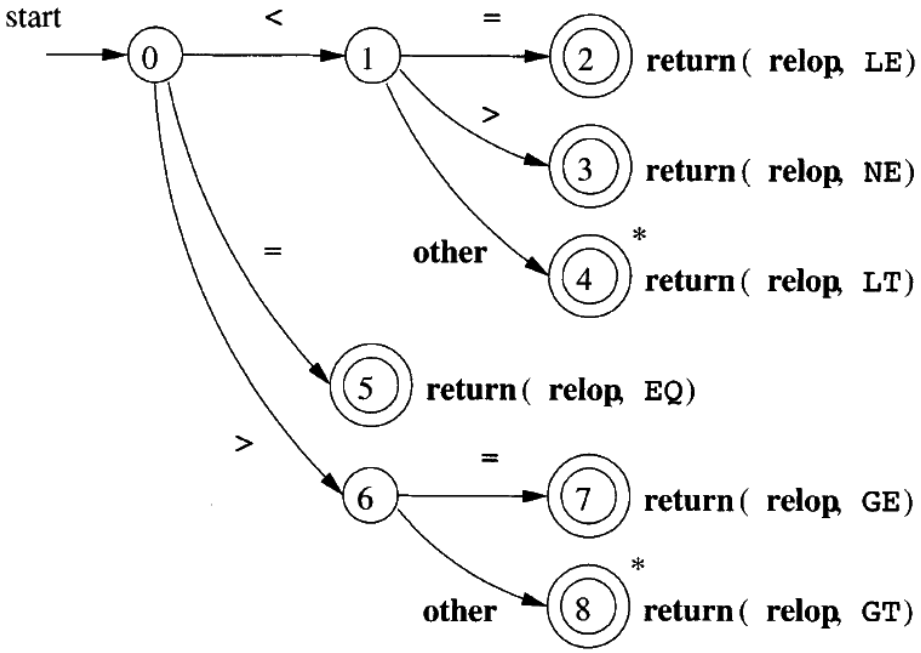


图 3-13　词法单元 **relop** 的状态转换图

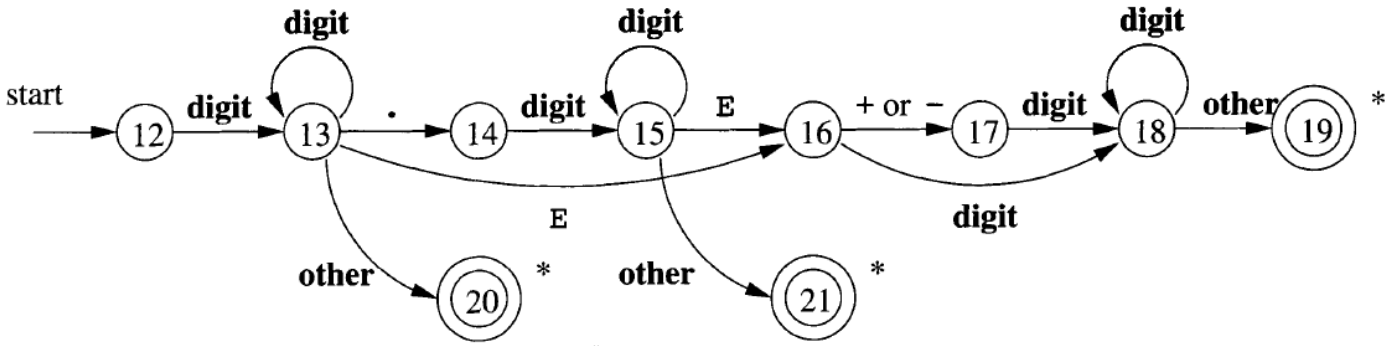


图 3-16　无符号数字的状态转换图

词法分析器生成工具的设计

- 词法分析器生成工具的体系结构

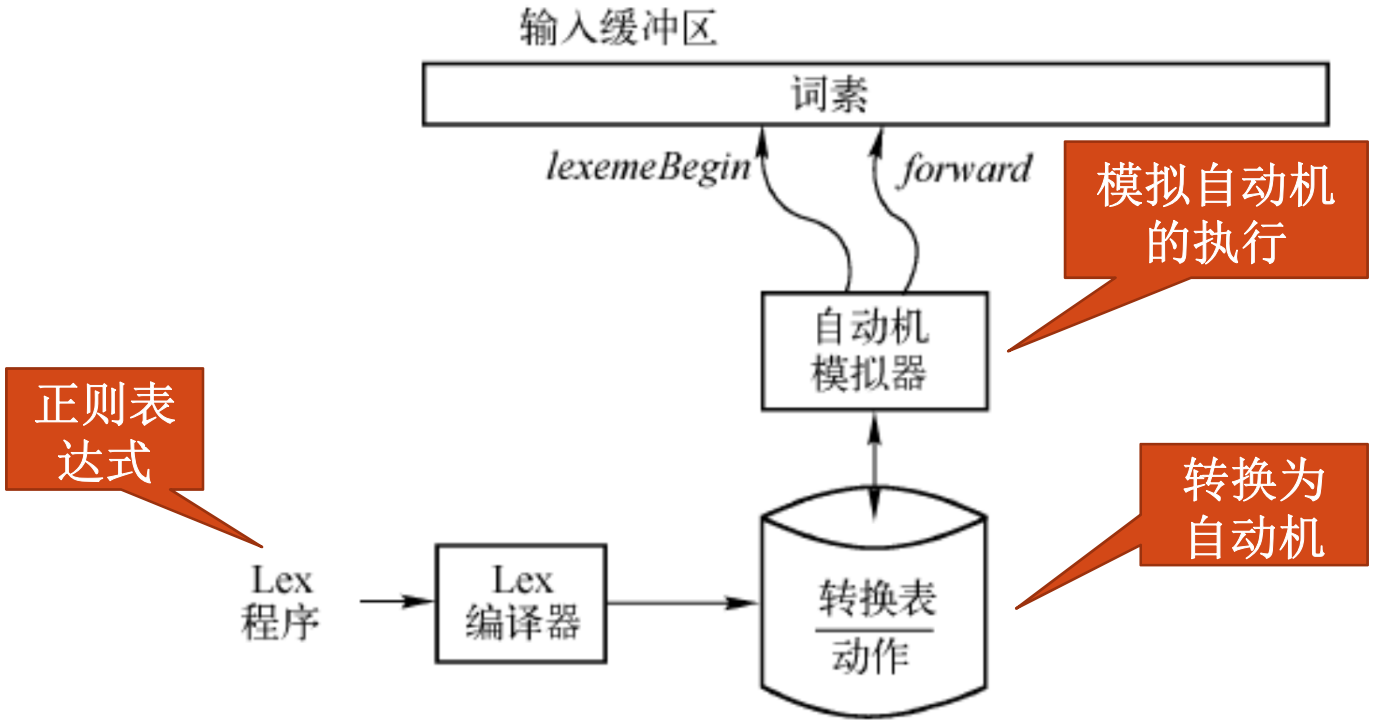


图 3-49　一个 Lex 程序被转变成由有限自动机模拟器使用的转换表和动作

自动机构建

- 把Lex程序中的每个正则表达式转换成NFA
- 由于自动机需要识别所有与Lex程序中的模式相匹配的词素，因此我们需要将这些NFA合并为一个NFA
- 引入一个新的开始状态，从这个新状态到各个对应于模式p_i的NFA N_i的开始状态各有一个ε转换

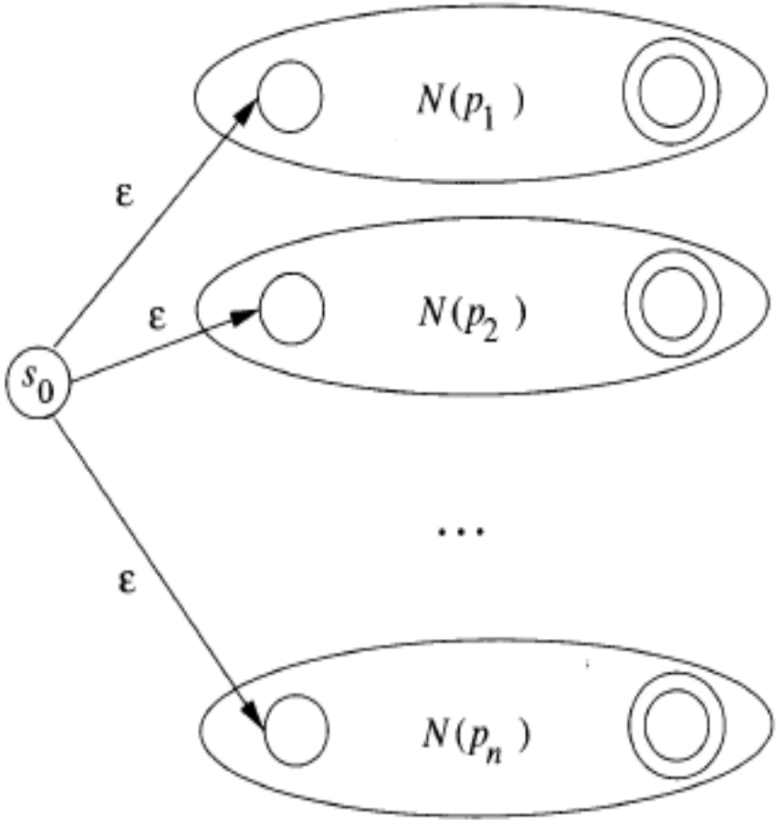


图 3-50　根据 Lex 程序构造得到的一个 NFA

示例

- 冲突问题

a	{模式 p_1 的动作 A_1}
abb	{模式 p_2 的动作 A_2}
$\mathbf{a} * \mathbf{b}^+$	{模式 p_3 的动作 A_3}

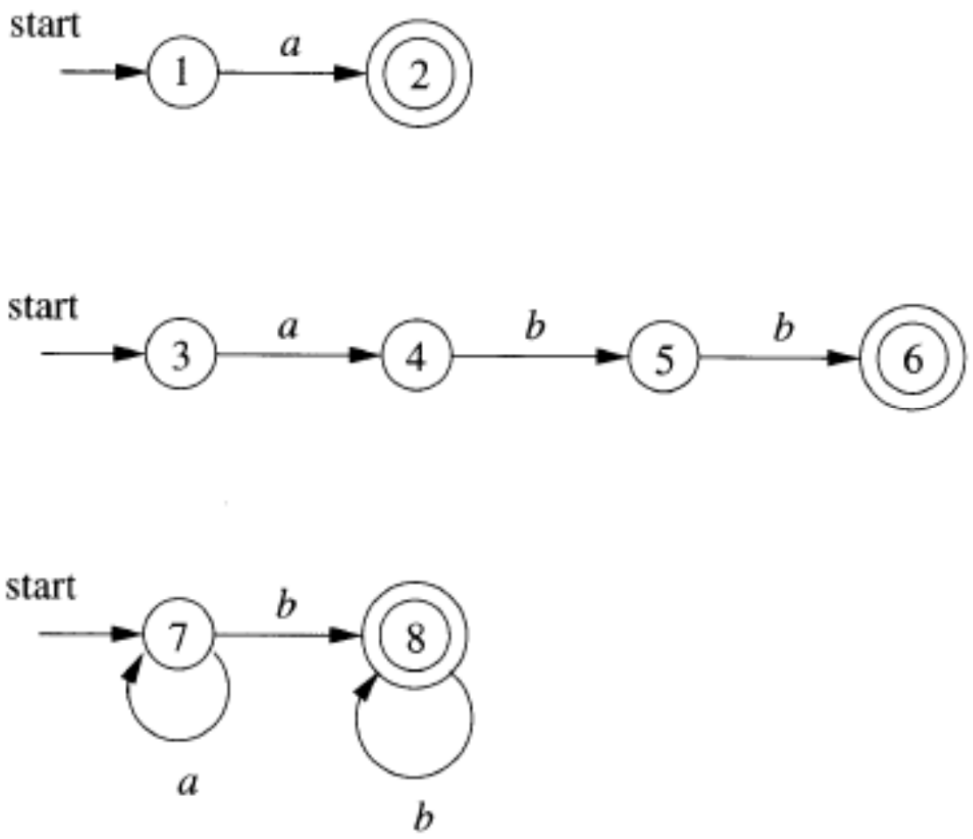
示例（续）

- 上述三种模式对应的NFA

a

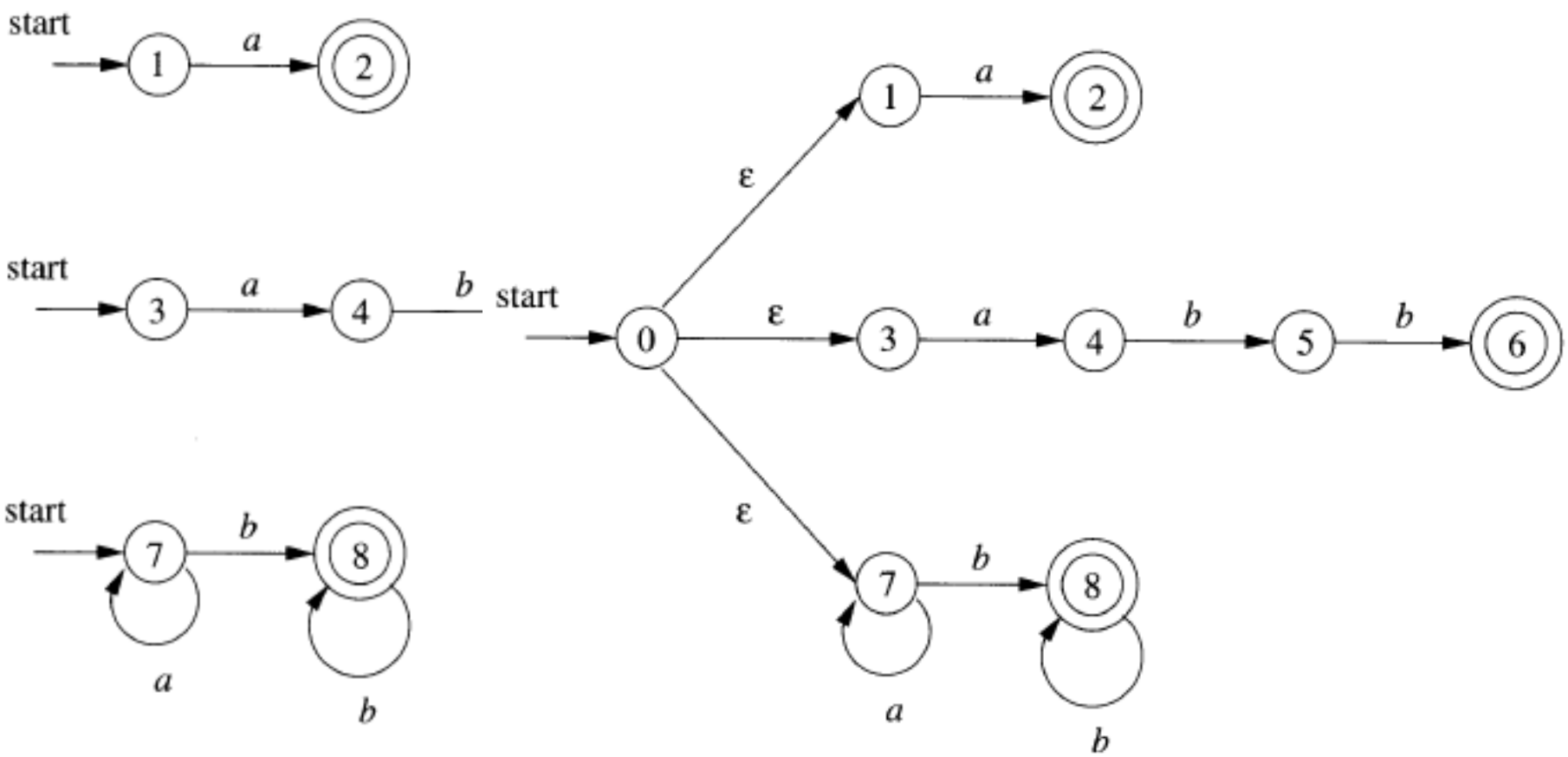
abb

a * b +



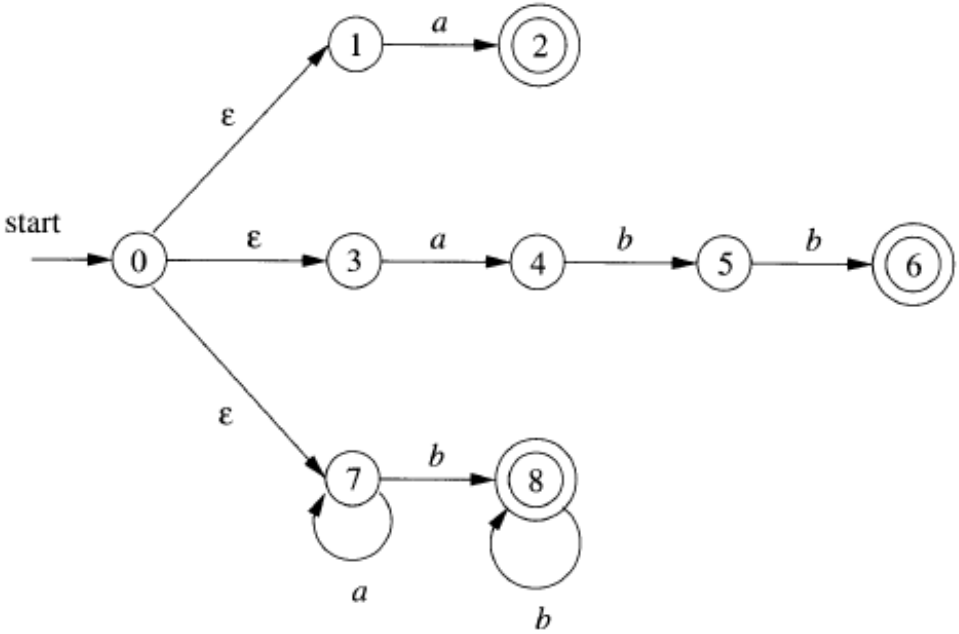
示例（续）

- 合并后的NFA


start
1
a
2
start
3
a
4
b
start
7
b
8
a
b
start
0
ε
1
a
2
ε
3
a
4
b
5
b
6
ε
7
b
8
a
b

基于NFA的模式

- 词法分析器模拟NFA的i
  态的输入点
- 沿着状态集顺序回找，
  接受状态的集合为止。
  我们就选择和在Lex程序
  接受状态p_i，执行相应对应A_i



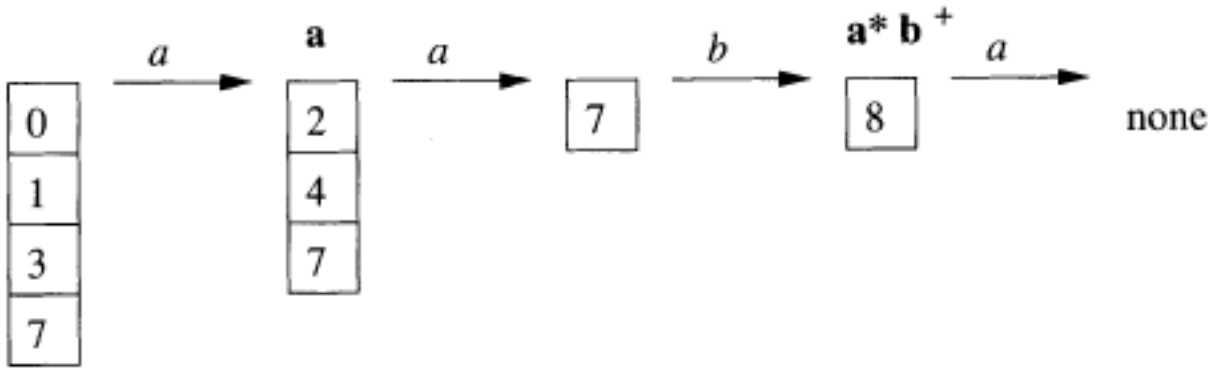


图 3-53　在处理输入 *aaba* 时进入的状态集的序列

使用DFA的词法分析器

- 将NFA转换成DFA之后，由词法分析器模拟DFA的运行
- 如果一个DFA的状态含有一个或多个NFA的接受状态，那么就要确定哪些模式的接受状态出现在此DFA的状态中，并找出第一个这样的模式，再给出该模式的输出
- 输入abba，找到模式p2=abb
- 输入aaba，找到模式 ？

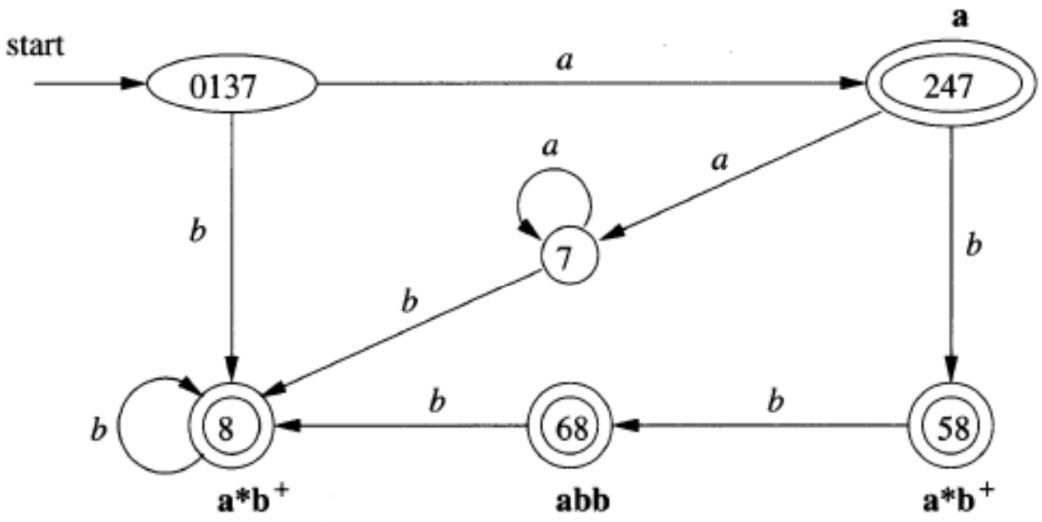


图 3-54　处理模式 **a**、**abb** 和 **a** * **b**$^+$ 的 DFA 的转换图

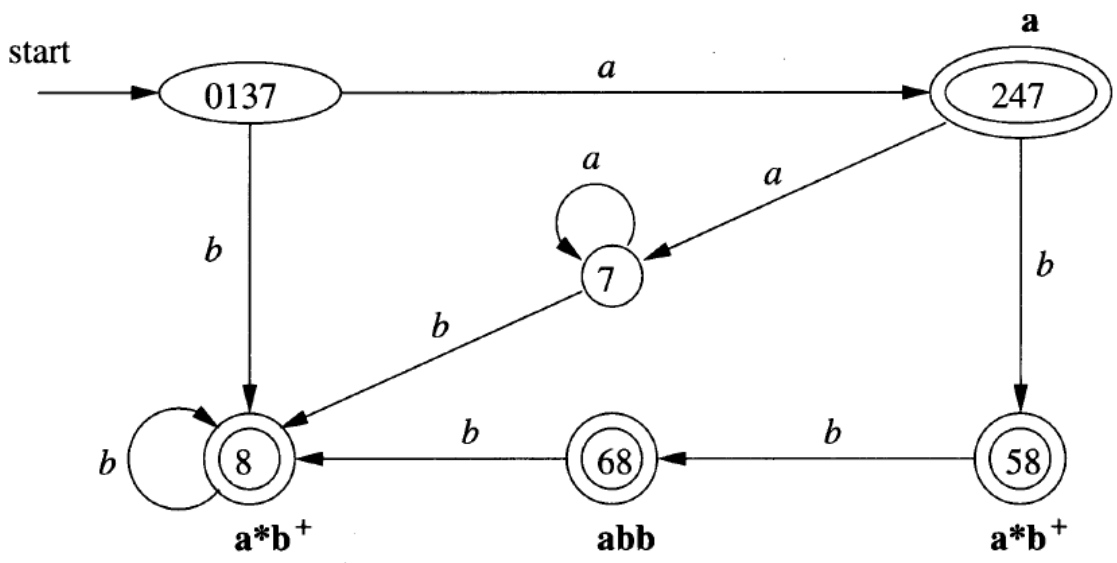
词法分析器的状态最小化

- 词法分析器中的不同接受状态对应于不同的模式，因此需要有不同于DFA化简的初始划分

- 初始分划为：所有非接受状态集合+对应于各个模式的接受状态集合

例子

- 初始划分增加死状态Φ，用作词法分析的DFA可以丢掉Φ
- 初始分划：{0137, 7} {247} {8, 58} {68} {Φ}



词法分析器工具Lex

- Lex/Flex:基于给定的用来描述词法单元模式的正则表达式，生成词法分析器
- 通常和Yacc一起使用，生成编译器的前端

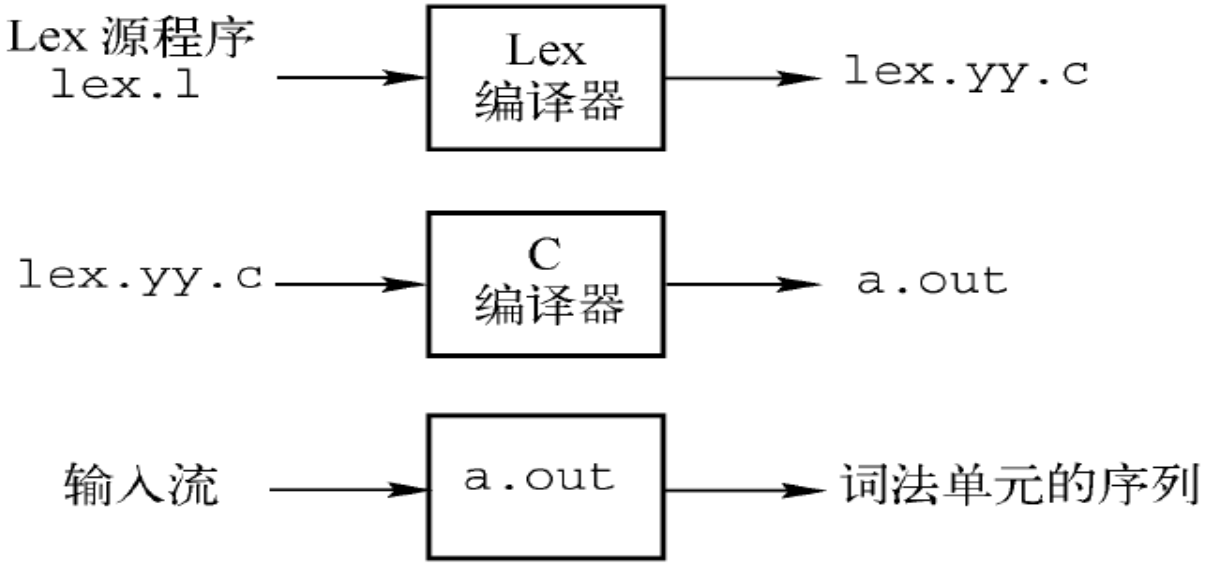


图 3-22　用 Lex 创建一个词法分析器

Lex程序结构

- 声明部分
  - 明示常量：表示常数的标识符
  - 正则定义
- 转换规则
  - 模式 {动作}
  - 模式是一个正则表达式或者正则定义
  - 动作通常是C语言代码，表示匹配该表达式后，应该执行的代码。
- 辅助函数
  - 动作中需要使用的函数

声明部分
%%
转换规则
%%
辅助函数

Lex程序的形式

Lex变量

- 当id被匹配时，会用到三个变量
  - yylval：token的值
  - yytext：token的lexeme
  - yyleng：lexeme的长度

词法分析器的工作方式（与语法分析器协同工作）

- 被调用时，不断读入余下的输入
- 直到发现最长的、与某个模式匹配的前缀，调用相应的动作；
  - 该动作进行相关处理，并把控制返回；
  - 如果不返回，则词法分析器继续寻找其它词素

Lex程序的例子（1）

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

%{和}%之间的内容一般被直接拷贝到lex.yy.c中；
这里的内容就是一段注释；
LT，LE等的值在yacc源程序中定义

正则定义

分隔声明部分和转换规则部分

Lex程序的例子

没有返回，表示继续识别其他词法单元

```
%%

{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNum(); 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程序的例子（3）

- 辅助函数被直接拷贝到lex.yy.c中
- 可在转换规则中直接调用

```
%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Lex中的冲突解决方法

- 当输入的多个前缀与一个或者多个模式匹配时，Lex按照如下规则解决冲突
  - 总是选择最长的前缀
    - 保证词法分析器把<=当作一个词法单元识别
  - 长度相等时，选择在Lex程序中首先被列出的模式
    - 如果保留字对应的规则在标识符的规则之前，词法分析器将识别出保留字