



编译原理

2020年3月



上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

编译原理

第三章

上海交通大学

张冬莱

Email: zhang-dm@cs.sjtu.edu.cn

2020年3月



第3章 词法分析



词法分析器从输入串中识别单词，编译程序对源程序的分析由此开始。单词构词规则可由状态转换图表示，获得状态转换图，便可产生高效的词法分析器。

本章的重点：如何由正规式表示构词规则并转换成识别单词的有限自动机，从而得到了识别单词的状态转换图。

3.1 词法分析器的作用



词法分析是编译的第一阶段。词法分析器的主要任务是读入输入字符，产生记号序列，提交给语法分析使用。这种交互（图3-1中）通常可以通过使词法分析器作为语法分析器的子程序或协作程序来实现。当词法分析器收到语法分析器发出的“取下一个记号”的命令时，词法分析器读入输入字符，直到识别出下一个记号。

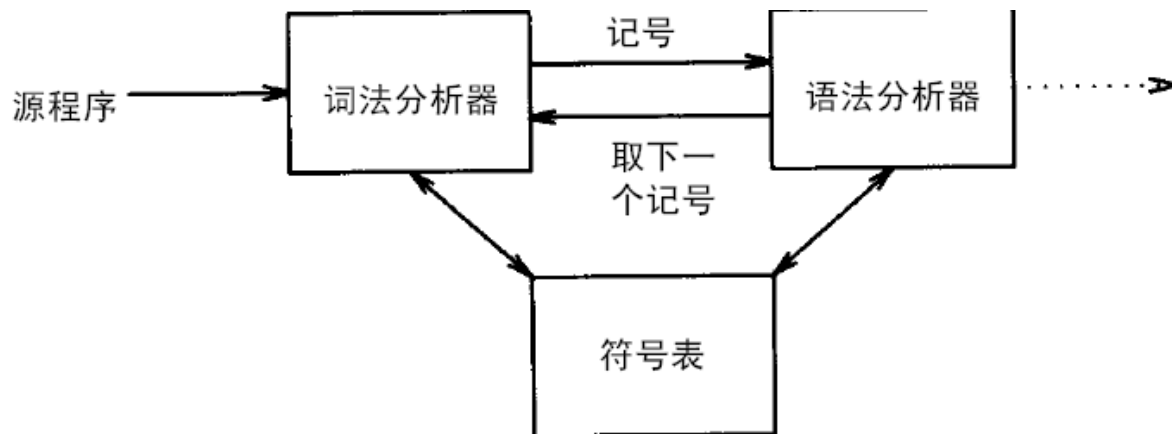


图3-1 词法分析器与语法分析器之间的交互

3.1.1 词法分析及语法分析



- 把编译过程的分析阶段划分为词法分析和语法分析的原因如下：
- 1. 简化编译器的设计可能是最重要的考虑。
- 2. 提高编译器的效率。
- 3. 增强编译器的可移植性。

3.1.2 词法单元、模式、词素



- 在词法分析的讨论中，我们使用术语“词法单元（也称记号）”、“模式”、“词素”表示特定的含义。

在多数程序设计语言中

- 记号：关键字、操作符、标识符、常量、文字串和标点符号（如括号、逗号和分号）
- 词素：源程序的字符序列，由一个记号的模式来匹配。
- 模式：描述源程序中表示特定记号的词素集合的规则。

例3.1



图3-2是使用它们的例子。通常，在输入中有一组字符串会产生相同的记号（作为输出），这个字符串构成的集合由一个与该记号相关联的称为模式的规则来描述。这个模式被说成匹配该集合中的每个字符串。

记号	词素示例	模式的非形式描述
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	<或<=或=或<>或>或>=
id	pi, count, D2	字母打头的字母数字串
num	3.1416, 0, 6.02E23	任何数字常数
literal	"core dumped"	在"与"之间除"以外的任何字符

图3-2 记号的例子

3.1.3 词法单元的属性



词法分析器把与词法单元（记号）有关的信息收集到记号的属性中。

记号影响语法分析，而属性影响记号的翻译。

记号及其属性可以用一个二元组来表示：

(记号，记号的属性值)

在实际实现时，记号通常只有一个属性，即指向符号表中一个表项的指针，与记号有关的信息保存在这个对应的表项中。为了诊断错误，我们不仅要知道匹配标识符的词素，而且还需要知道这个词素第一次出现的行号。这些信息都可以存储在符号表中该标识符对应的表项内。



我们可以根据一个记号的模式是否只匹配一个词素，将词素分为两类：

- **有限类：**

由于基本字，运算符，界符的数目有限，一般可以每个词素本身就是记号，就代表自身的性质，这时，它的第二元就没有识别意义了。

- **无限类：**

由于常数和标识符类的词素，对应的词素个数是无限的，在这种情况下，常数或标识符将由词素的属性值来区别是哪一个词素。通常记号通常只有一个属性，即指向符号表中一个表项的指针，与记号有关的信息保存在这个对应的表项中。

例3.2



Fortran语句： $E = M * C ** 2$ 中的记号和它们的属性值的
二元组序列表示如下：

- $\langle \text{id}, \text{指向符号表中与E相关的表项的指针} \rangle$
- $\langle \text{assign_op}, \rangle$
- $\langle \text{id}, \text{指向符号表中与M相关的表项的指针} \rangle$
- $\langle \text{mult_op}, \rangle$
- $\langle \text{id}, \text{指向符号表中与C相关的表项的指针} \rangle$
- $\langle \text{exp_op}, \rangle$
- $\langle \text{num}, \text{整数值}2 \rangle$

注意：某些二元组不需要属性值，它的第一个分量足以标识词素。在上述例子中，记号num的属性是一个整数值。当然，编译器也可以把形成数的字符串存入符号表中，并让记号num的属性是指向符号表中相应表项的指针。

3.1.4 词法错误



因为词法分析器不能从全局的角度考察源程序，所以能在词法层发现的错误是有限的。原则上词法分析器能够发现的错误是词法规则不能识别的

词素，如：

- 非法字符；
- 非法常数；

除此之外没有什么词法错误是在词法分析时发现的，往往要推迟到语法分析时才发现，

例如：把if写成fi，则词法分析作为标识符处理。le ngth，中间多了空白，词法分析将le和ngth作两个标识符处理，由此而造成的后果，将在其他分析过程中才反映出来。

3.2 词法单元的规约



正规表达式(正规式)是表示模式的一种重要方法。

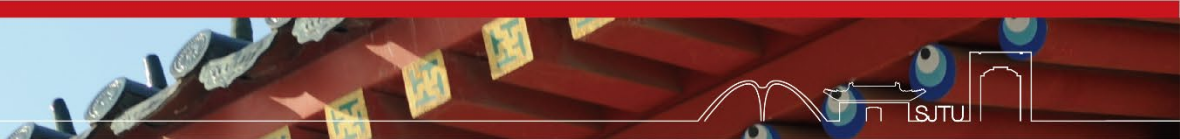
每个模式匹配一个字符串集。因此正规表达式将作为字符串集的名字。

3.2.1 串和语言



术语：

- 字母表：一个非空有限符号的集合。例如： $\{a,b,c,0,1,2\}$ 。
- 字符串：字母表中符号的有穷序列。在语言理论中，“句子”和“字”常作为“字符串”的同义词。字符串 s 的长度是出现在 s 中的符号的个数，通常记作 $|s|$ 。
- 空字符串：是长度为0的特殊字符串，简称空串，用 ϵ 表示。



下图概括了用于表示字符串各部分的常用术语。

术 语	定 义
s 的前缀	去掉串 s 尾部的 0 个或多个符号后得到的字符串。例如， ban 是 banana 的前缀
s 的后缀	去掉串 s 头部的 0 个或多个符号后得到的字符串。例如， nana 是 banana 的后缀
s 的子字符串	去掉 s 的一个前缀和一个后缀后得到的字符串。例如， nan 是 banana 的一个子串。 s 的每个前缀和后缀都是 s 的一个子串，但子串并不总是 s 的前缀或后缀。对于每个字符串 s ， s 和 ϵ 是 s 的前缀、后缀和子串
s 的真前缀 (真后缀、真子串)	如果非空串 x 是串 s 的前缀 (后缀、子串)，而且 $s \neq x$ ， 则称 x 是 s 的真前缀 (真后缀、真子串)
s 的子序列	从串 s 中删除 0 个或多个符号后得到的串 (这些被删除的 符号可以不相邻)。例如，baaa 是 banana 的子序列

图3-7 字符串的各部分的术语



- 语言：是给定字母表上的任意一个字符串集合。这个定义是广义的。
像空集和仅包含空符号串的集合 $\{\epsilon\}$ 这样的抽象语言也符合这个定义。
- 字符串的连接：如果 x 和 y 是字符串，那么 x 和 y 的连接（记作 xy ）是把 y 连接到 x 后面所形成的符号串。
例如：如果 $x=\text{dog}$ 且 $y=\text{house}$ ，那么 $xy=\text{doghouse}$ 。
对连接运算而言，空串是一个单位元，也就是说， $s\epsilon = \epsilon s = s$ 。
- 字符串的“指数”：如果把两个符号串的连接看成是这两个串的“乘积”，我们可以定义符号串的“指数”如下：定义 s^0 为 ϵ ，对于 $i > 0$ ， s^i 为 $s^{i-1}s$ ，因为 s 就是 s 本身，所以 $s^1=s$ ， $s^2=ss$ ， $s^3=sss$ ，依此类推。

3.2.2 语言上的运算



设字母表为 Σ ，对于 Σ 的两个语言 L 和 M ，可以定义图3-8所示的四种语言之间的运算。

运 算	定 义
L 和 M 的并(记作 $L \cup M$)	$L \cup M = \{s \mid s \text{ 属于 } L \text{ 或 } s \text{ 属于 } M\}$
L 和 M 的连接(记作 LM)	$LM = \{st \mid s \text{ 属于 } L \text{ 且 } t \text{ 属于 } M\}$
L 的克林 (Kleene) 闭包(记作 L^*)	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* 表示 0个或多个 L 的连接
L 的正闭包 (记作 L^+)	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ 表示 1个或多个 L 的连接

图3-8 语言上的运算的定义

例3.3



- 令 L 表示集合 $\{A, B, \dots, Z, a, b, \dots, z\}$, D 表示集合 $\{0, 1, \dots, 9\}$ 。我们可以将 L 看成是由大、小写字母组成的字母表, 将 D 看成是10个数字组成的字母表。同时, 由于单个符号也可以看成长度为1的符号串, 我们可以把 L 和 D 分别看成是有穷的语言集。下面是将图3-8中定义的运算作用于 L 和 D 得到新语言:
 - 1. $L \cup D$ 是字母和数字的集合。
 - 2. LD 是一个字母后随一个数字的符号串的集合。
 - 3. L^4 是由四个字母构成的符号串的集合。
 - 4. L^* 是所有字母构成的串 (包括) 的集合。
 - 5. $L(L \cup D)^*$ 是所有以字母开头的字母数字串的集合。
 - 6. D^+ 是由一个或多个数字构成的数字串的集合。

3.2.3 正规表达式



下面是定义字母表 Σ 上的正规表达式的规则，每一条规则后带有所定义的正规表达式所表示的语言的一个说明：

- 1. ϵ 是正规表达式，它表示 $\{\epsilon\}$ ，即包含空串的集合。
- 2. 如果 a 是 Σ 上的符号，那么 a 是正规表达式，表示 $\{a\}$ ，也就是包含符号串 a 的集合。
- 3. 假定 r 和 s 都是正规表达式，分别表示语言 $L(r)$ 和 $L(s)$ ，则：
 - a) $(r) \mid (s)$ 是正规表达式，表示 $L(r) \cup L(s)$ 。
 - b) $(r) (s)$ 是正规表达式，表示 $L(r)L(s)$ 。
 - c) $(r)^*$ 是正规表达式，表示 $(L(r))^*$ 。
 - d) (r) 是正规表达式，表示 $L(r)$ 。




- 正规表达式表示的语言叫做正规集。
- 采用如下约定：
 - 1. 一元运算符 $*$ 具有最高的优先级，并且是左结合的。
 - 2. 连接的优先级次之，也是左结合的。
 - 3. $|$ 的优先级最低，同样是左结合的。

例3.4



设: $\Sigma = \{a, b\}$, 则:

- 1. 正规表达式 $a|b$ 表示集合 $\{a, b\}$ 。 
- 2. 正规表达式 $(a|b)(a|b)$ 表示 $\{aa, ab, ba, bb\}$, 即由 a 和 b 组成的长度为2的符号串集合。表示同样集合的另一正规表达式是 $aa / ab / ba / bb$ 。
- 3. 正规表达式 a^* 表示由零个或多个 a 组成的所有串的集合 $\{, a, aa, aaa, \dots\}$ 。
- 4. 正规表达式 $(a|b)^*$ 表示由零个或多个 a 或 b 构成的符号串集合, 即由 a 和 b 构成的所有符号串的集合。这个集合也可用另一个正规表达式 $(a^*b^*)^*$ 来表示。
- 5. 正规表达式 $a|a^*b$ 表示包含串 a 和零个或多个 a 后跟随一个 b 构成的符号串集合。



- 两个正规表达式：如果两个正规表达式 r 和 s 表示同样的语言，则称 r 和 s 等价，记作 $r = s$ 。例如， $(a|b) = (b|a)$ 。
- 正规表达式的代数定律：它们可以用于正规表达式的等价变换，图3-9是正规表达式 r 、 s 和 t 遵循的代数定律。

公 理	描 述
$r s = s r$	$ $ 是可交换的
$r (s t) = (r s) t$	$ $ 是可结合的
$(rs)t = r(st)$	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
$\epsilon r = r$ $r\epsilon = r$	ϵ 是连接的单位元
$r^* = (r \epsilon)^*$	$*$ 和 ϵ 间的关系
$r^{**} = r^*$	$*$ 是幂等的

图3-9 正规表达式的代数性质

3.2.4 正规定义



- 为表示方便，我们可能希望为正规表达式命名，并用这些名字来定义正规表达式，就如同它们也是符号一样。如果S是基本符号的字母表，那么正规定义是如下形式的定义序列：

$$\begin{aligned}d_1 &\rightarrow r_1 \\d_2 &\rightarrow r_2 \\&\dots \\d_n &\rightarrow r_n\end{aligned}$$



- 其中：每个 d_i 都是一个名字，并且它们各不相同，每个 r_i 是 $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ （即基本符号和前面定义的名字）中符号上的正规表达式。由于限制了每个 r_i 中只含有 Σ 中的符号和在它之前定义的名字，所以我们可以反复地用名字所代表的正规表达式替代该名字的方法为任何一个 r_i 构造 Σ 上的正规表达式。如果 r_i 用到了 d_j ，并且 $j \geq i$ ，则 r_i 是递归定义的，而且这个替换过程不会中止。
- 为了区别名字和符号，用黑体字表示正规定义中的名字。

例3.5



- Pascal语言的标识符集合是以字母开头的字母数字串的集合,

则这个集合的正规定义是:

letter \rightarrow **A** | **B** | \dots | **Z** | **a** | **b** | \dots | **z**

digit \rightarrow **0** | **1** | \dots | **9**

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

例3.6



- Pascal语言中的无符号数是形如5280、39.37、6.336E4或1.894E-4这样的符号串。下面的**正规定义**给出了这类符号串的精确说明：

digit $\rightarrow 0 \mid 1 \mid \dots \mid 9$
digits $\rightarrow \text{digit digit}^*$
optional_fraction $\rightarrow . \text{ digits } \mid \epsilon$
optional_exponent $\rightarrow (E (+ \mid - \mid \epsilon) \text{ digits }) \mid \epsilon$
num $\rightarrow \text{digits optional_fraction optional_exponent}$

3.3 词法单元的识别



- 考虑下述文法片断：

$\text{stmt} \rightarrow \text{if expr then stmt} \mid \text{if expr then stmt else stmt} \mid \epsilon$

$\text{expr} \rightarrow \text{term relop term} \mid \text{term}$

$\text{term} \rightarrow \text{id} \mid \text{num}$

其中：终结符if、then、else、relop、id和num产生由以下正规定义给出的串的集合：

if \rightarrow **if**
then \rightarrow **then**
else \rightarrow **else**
relop \rightarrow **<** **|** **<=** **|** **=** **|** **<>** **|** **>** **|** **>=**
id \rightarrow **letter** (**letter** **|** **digit**)^{*}
num \rightarrow **digit**⁺ (**.** **digit**⁺)? (**E**(**+** **|** **-**)? **digit**⁺)?



- 其中，letter和digit的定义与前面相同。
- 对这个给定的语言，词法分析器将识别关键字if、then、else和由relop（关系操作符）、id（标识符）和num（数）表示的词素。为简单起见，我们假定关键字是保留的，也就是说，它们不能作为标识符使用。类似于例3.5，这里的num表示Pascal中的无符号整数和实数。
- 此外，我们还假定词素由空白符分隔。空白符是空格、制表符、换行符组成的非空序列。词法分析器还要完成去掉空白符的任务。这个任务通过把输入串与如下的ws 正规定义相比较来完成：

ws -> delim

delim -> blank | tab | newline

3.3 词法单元的识别



- 如果发现了与ws匹配的字符串，则词法分析器不返回记号给语法分析器，继续识别空白符后面的记号，然后把它返回给语法分析器。
- 我们的目标是构造一个词法分析器，这个词法分析器能利用图3-10给出的映射表在输入缓冲区中识别出下一个记号的词素，产生该词素相应的记号和属性值的二元组。关系操作符的

正规表达式	记 号	属 性 值
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	指向符号表表项的指针
num	num	指向符号表表项的指针
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

图3-10 记号的正规表达式模式

属性值由符号常量LT、LE、EQ、NE、GT和GE给出。

3.3.1 状态转换图



- 状态转换图：是模式转换成特定意义的流图，这种流图可以手工构造，也可以自动构造。
- 状态转换图的结点用圆圈表示，叫做状态。状态间由箭头连接，称为边。由状态 s 到状态 r 的边上标记的字符表示使状态 s 转换到状态 r 的输入字符。标记other表示任意一个未被离开状态 s 的边所标定字符。本节假定状态转换图是确定的，即没有一个符号可以同时与离开一个状态的两条以上的边的标记匹配。如图3-11所示：

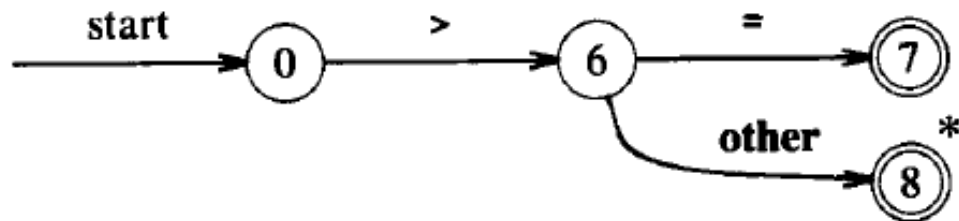


图3-11 $>=$ 和 $>$ 的状态转换图

例3.7



- 图3-12给出了记号relop的状态转换图。注意，图3-11中的状态转换图只是这个更复杂的状态转换图的一部分。

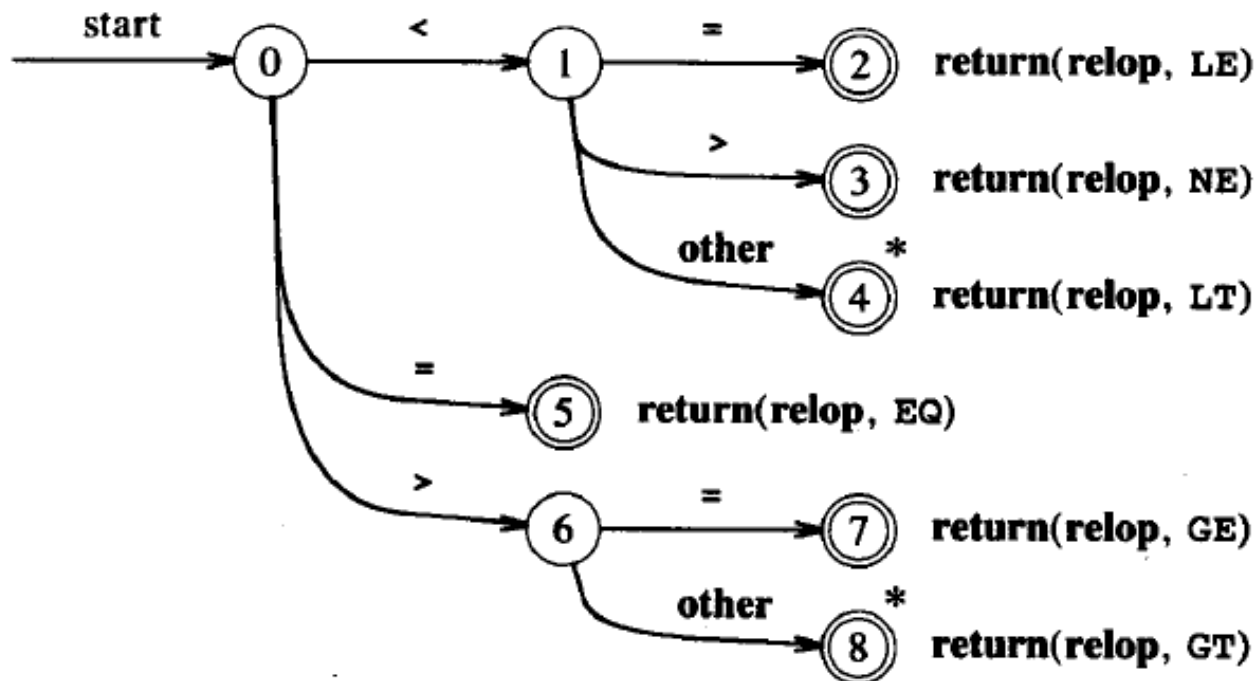


图3-12 关系操作符的状态转换图

3.3.2 关键字和标识符的识别



因为关键字是字母序列，所以它们也符合标识符的规则，即由字母开头的字母和数字的序列。一般来说，我们不为关键字单独构造状态转换图，而是把关键字看成特殊的标识符。当到达图3-13的接受状态时，执行一段代码，以确定这次识别的词素是关键字还是标识符。其中：

- `gettoken()`：返回的记号。
- `install_id()`：返回的记号的属性值。若记号在关键字表中，`install_id()`返回0，当它是程序变量时，`install_id()`返回指向相应符号表表项的指针。如果在符号表中没有找到该词素，则把该词素作为变量填入符号表中，并返回指向新建表项的指针。

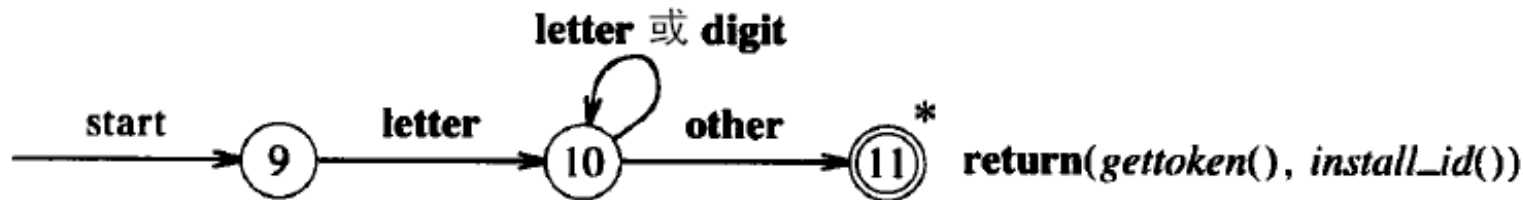


图3-13 标识符和关键字的状态转换图

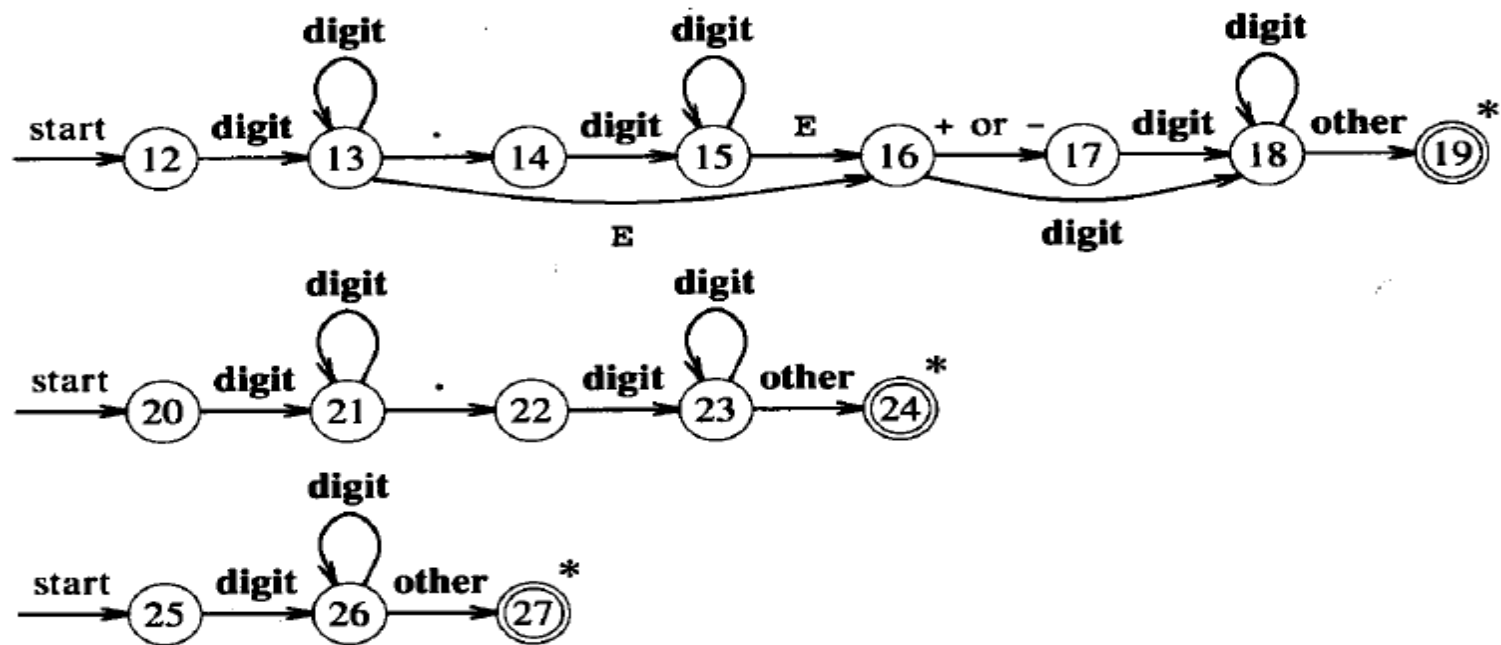


图3-14 Pascal中无符号数的状态转换图

我们可以使用多种方法避免图3-14中的多余匹配。如将这些状态转换图合并成一张图，一般来说这个任务比较艰巨。。

3.3.4 状态转换图的实现



设：图3-15是一个
小语言的词法分析
的状态转换图

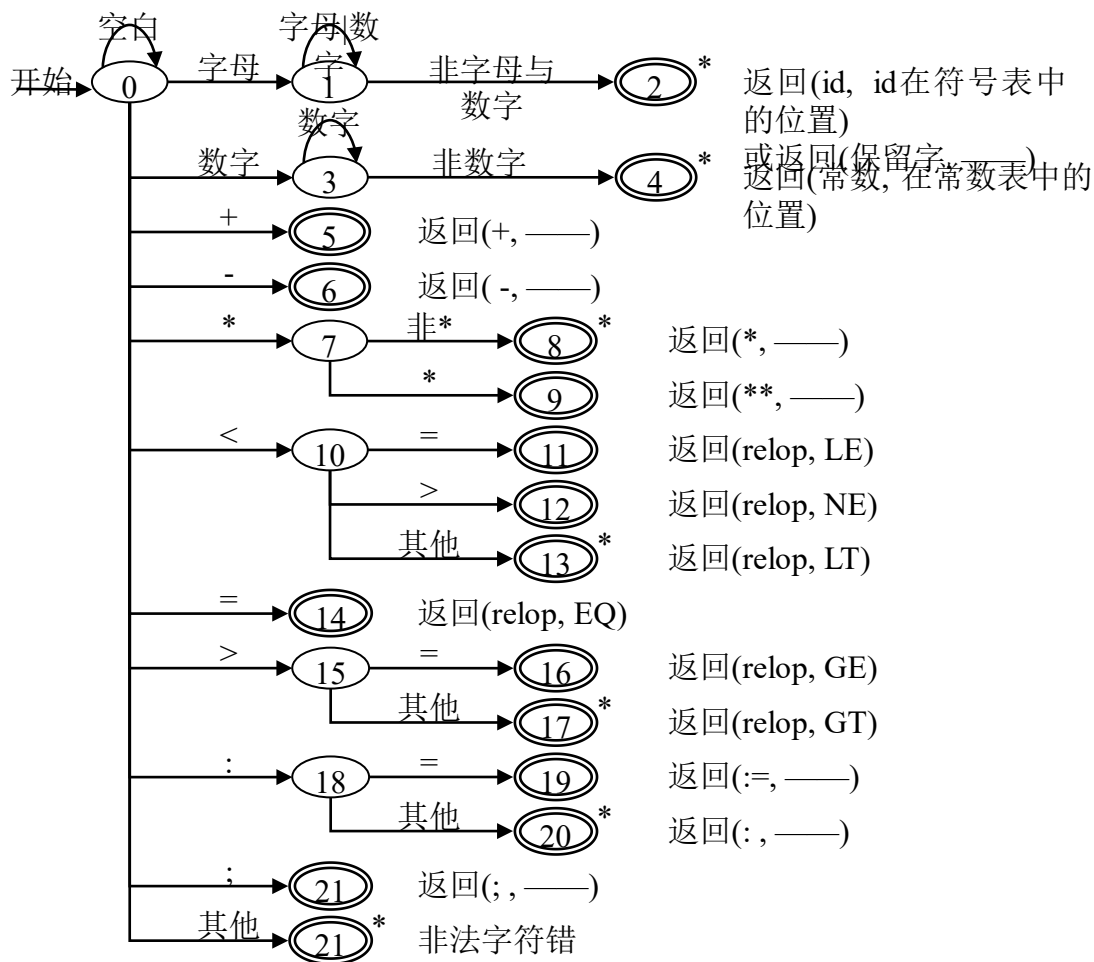


图3.-15 词法分析的状态转换图

状态转换图的核心C程序如下：



```
token=""; /*置token为空串*/。
s=getchar(); getbe();
switch(s)
{case 'a': /*字母开头*/
  case 'b':...
  case 'z':
while(letter()||digit())
{concatenation();
getchar(); }
retract();
c=reserve();
if(c==0){buildlist();
return (id, 指向id的符号表入口指针) ;
}
else {return (保留字码, null) }
break;
```

```
case '0': /*数字开头*/
case '1':
...
case '9':
while(digit())
{concatenation();
getchar();
}
retract();
buildlist();
return(num, num的常数表入口指针);
break;
```



```
case '+': /*算术运算符*/  
return(plus-op, null);  
break;  
case '-':  
return(mimus-op, null);  
break;  
case '*':  
getchar();  
if(character=='*')  
{return(power-op, null); }  
retract();  
return(star, null);  
break;
```

```
case '<': /*关系运算符*/  
getchar();  
if(character=='<')  
    {return(rel-op, LE); }  
else if(character=='>')  
    {return(rel-op, NE); }  
retract();  
return(rel-op, LT);  
break;  
case '=': return(rel-op, EQ);  
break;  
case '>': getchar();  
if(character=='>')  
    {return(rel-op, GE); }  
retract();  
return(rel-op, GT);  
break;
```

```
case ':': /*界符*/  
getchar();  
if(character==':')  
{  
return(assign-op, null);  
}  
retract();  
return(colon, null);  
break;  
case ';':  
return(semicolon, null);  
}
```



其中引进的变量和过程为：

- token：字符数组，存放单词符号的符号串。
- getchar:将下一输入字符读入character的过程，将向前指针移向下一字符。
- getbe：若character中的字符为空白，则调用getchar，直至character为非空白时止。开始指针移到向前指针位置。
- concatenation：将token中的字符串与character中字符连接，作为token中的新的字符串。

3.4 词法分析器描述语言



- 目前有很多基于正规表达式从特定表示法构建词法分析器的工具。
- 本节将介绍一个叫做Lex的工具。Lex已经广泛地应用于各种语言的词法分析器的描述。我们称这种工具为Lex编译器，而且Lex编译器的输入称为Lex语言。讨论现有的工具的目的旨在说明如何把正规表达式描述的模式与行为（如在符号表中创建新表项，这是词法分析器需要做的动作）结合起来。

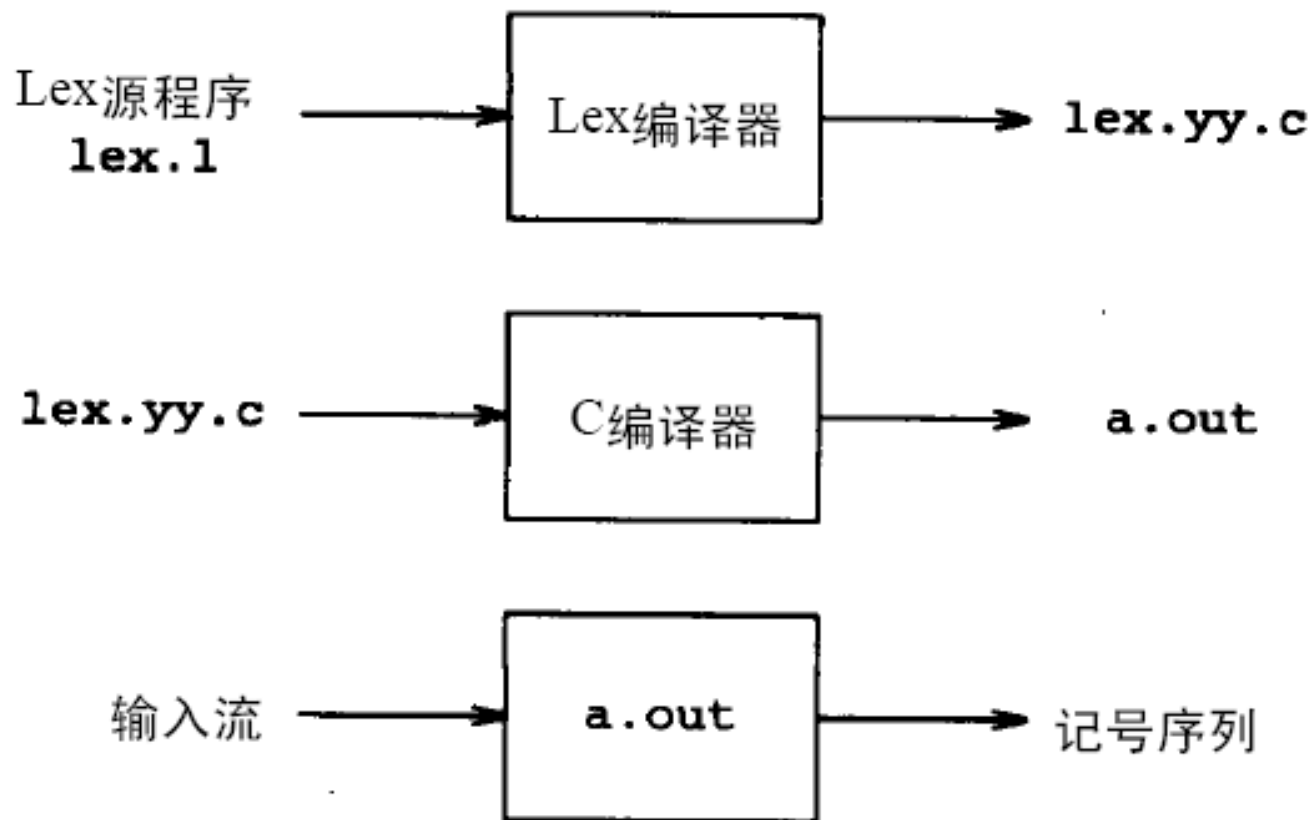


图3-17 用 Lex 建立一个词法分析器

3.4.1 Lex 说明



- 一个Lex程序由如下三部分组成：

(1) 声明部分：声明部分包括变量声明、符号常量声明和正规定义。（符号常量是被声明来表示常数的标识符。）

(2) %%

转换规则：如：

%%

p_1	{ <i>action</i> ₁ }
p_2	{ <i>action</i> ₂ }
...	...
p_n	{ <i>action</i> _n }

(3) 辅助过程：包含*action*所需要的辅助过程。这些过程可以单独编译，并与词法分析器一起装载。

例3.8



```
%{
    /* 符号常量定义
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
}%

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

%%

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

%%

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

install_num() {
    /* 与填词素的过程类似，只不过词素是一个数 */
}
```

图3-18 关于图3-10中记号的 Lex 程序

元语言符号：在定义语言时用到的符号。有：

- `[]`:表示在一定范围。
- `()`:表示包括。
- `|`: 表示并。
- `?`: 表示出现过0次或一次。
- `\`: 表示转义字符。
- 第一个`%%`和第二个`%%`之间为转换规则。

3.4.2 超前扫描操作



- 在Lex中我们可以把模式写成 r_1/r_2 的形式，其中， r_1 和 r_2 都是正规表达式。它的意思是当一个字符串与 r_1 匹配时，还需其后的字符串与 r_2 匹配，这样才算该字符串与 r_1 匹配成功。在超前扫描操作符/后面的正规表达式 r_2 表示需要进一步匹配的内容，这里它只是匹配模式的一个限制，而不是匹配的一部分。



3.5 有穷自动机



- 语言的识别器是一个程序，它以字符串 x 作为输入，
当 x 是语言的句子时，回答“是”，否则回答“不是”。
- 有穷自动机把正规表达式编译成识别器。
- 有穷自动机是更一般化的状态转换图。
- 有穷自动机它可以是确定的或不确定的。
- 确定和不确定的有穷自动机都能而且仅能识别正规集。
- 确定的有穷自动机导出的识别器比不确定的有穷自动机导出的识别器快得多，
- 确定的有穷自动机可能比与之等价的不确定的有穷自动机大

3.5.1 不确定的有穷自动机



不确定的有穷自动机（简称为NFA）是一个由以下几部分组成的数学模型：

- 1. 一个状态的有穷集合 S 。
- 2. 一个输入符号集合 Σ ，即输入符号字母表。
- 3. 一个转换函数 $move$ ，它把由状态和符号组成的二元组映射到状态集合。
- 4. 状态 s_0 是惟一的开始或初始状态。
- 5. 状态集合 F 是接受（或终止）状态集合。

- 转换图：带标记的有向图，
其中：节点是状态，有标记的边表示转换函数。
- NFA：同一个字符可以标记始于同一个状态的两个或多个转换，边可以是输入字符符号，也可以是特殊符号 ϵ 标记。
- 图3-19给出了识别语言 $(a|b)^*abb$ 的NFA的转换图。
这个NFA的：状态集合是 $\{0,1,2,3\}$ ，
输入符号表是 $\{a,b\}$ ，
状态0是开始状态，
状态3是接受状态，用双圈表示。

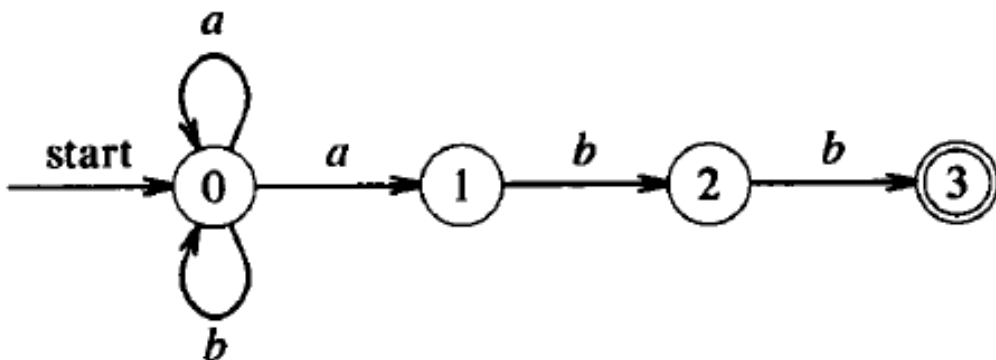


图3.-19识别语言 $(a|b)^*abb$ 的NFA的状态转换图

3.5.2 转换表



- 描述NFA时，最简单的办法是使用转换表。转换表的每个状态占一行，每个输入符号占一列。如果必要，符号 ϵ 也占一列。表中第 i 行 a 列对应的表项是当输入为 a 时从状态 i 所能到达的状态的集合。
- 图3-20是与图3-19的NFA 对应的转换表。

状 态	输入符号	
	a	b
0	{0, 1}	{0}
1	—	{2}
2	—	{3}

图3-20 与图3-19的NFA对应的转换表

3.5.3 自动机中输入字符串的接受



图3-21是接受 $aa^*|bb^*$ 的NFA。

字符串 aaa 经过状态0、1、2、2、2的路径被接受。

这些边上的标记分别为 ϵ ， a ， a 和 a ，连接成 aaa 。

ϵ 在连接中“消失”。

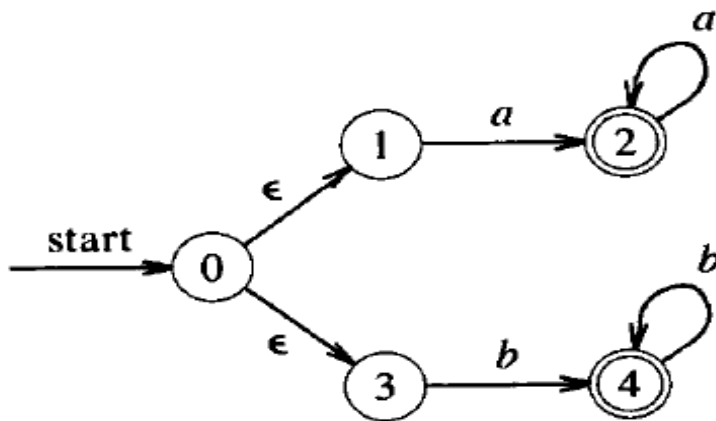


图3-21 接受 $aa^*|bb^*$ 的NFA

3.5.4 确定的有穷自动机



- 确定的有穷自动机（简称DFA）是不确定的有穷自动机的特例，其中：
 - 没有一个状态具有转换，即在输入上的转换。
 - 对每个状态 s 和输入符号 a ，最多只有一条标记为 a 的边离开 s 。

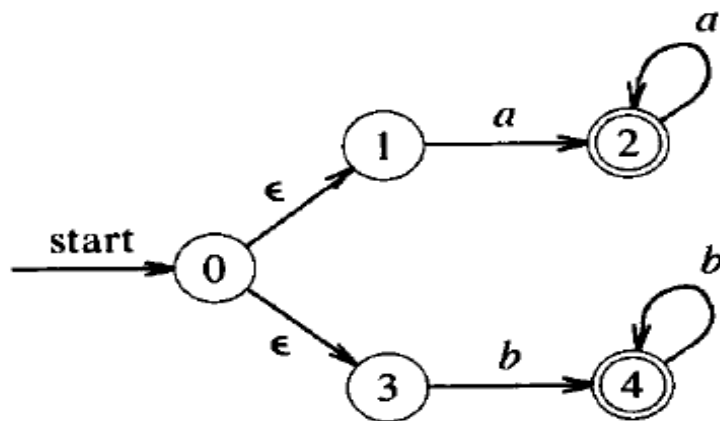
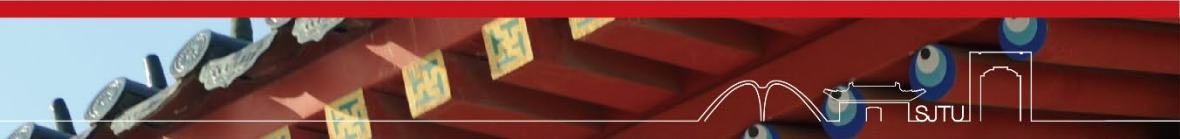


图3-21 接受 $aa^*|bb^*$ 的NFA



- 算法3.1 模拟DFA。
- 输入：输入以文件结束符eof结尾的串 x ；一个DFA D ，其开始状态为 s_0 ，接受状态集合为 F 。
- 输出：如果 D 接受 x ，则回答“yes”，否则回答“no”。
- 方法：把图3-22的算法应用于输入字符串 x 。函数 $move(s, c)$ 给出在状态 s 上遇到输入字符 c 时应该转换到的下一个状态。函数 $nextchar$ 返回输入串 x 的下一个字符。

```
 $s := s_0;$   
 $c := nextchar;$   
while  $c \neq eof$  do  
     $s := move(s, c);$   
     $c := nextchar$   
end;  
if  $s$  is in  $F$  then  
    return “yes”  
else return “no”;
```

图3-22 模拟DFA

- 图3-23是与图3-19的NFA接受同一语言 $(a|b)^*abb$ 的DFA的转换图。对这个DFA和输入串 $ababb$ ，算法3.1沿着状态序列0、1、2、1、2、3移动，并返回“yes”。

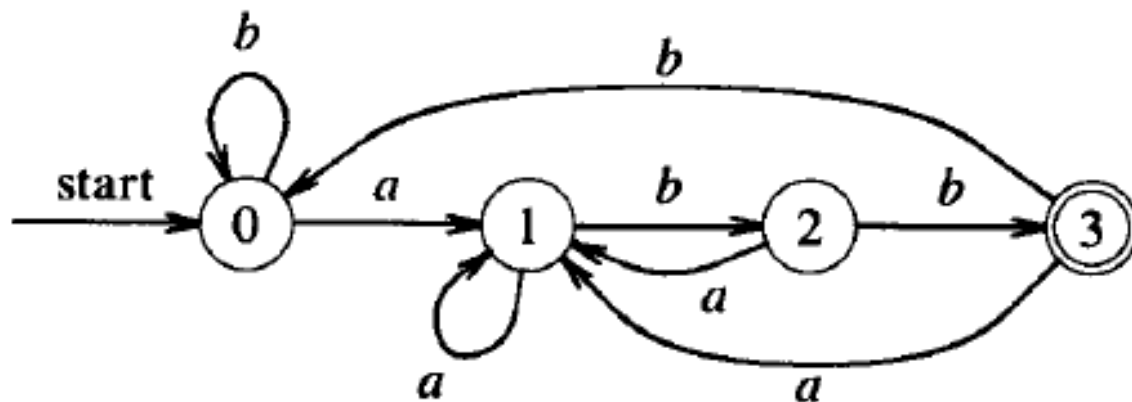


图3-23 接受 $(a|b)^*abb$ 的DFA

3.6 从正规表达式到自动机



- 有很多从正规表达式建立其识别器的策略，本节我们将可以介绍子集构造法把NFA 变成DFA。

3.6.1 从NFA到DFA的变换



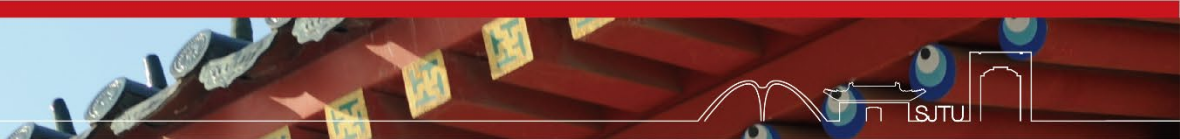
- 在NFA的转换表中，每个表项是一个状态集；而在DFA的转换表中，每个表项只有一个状态。从NFA变换到DFA的基本思想是让DFA的每个状态对应NFA的一个状态集。这个DFA用它的状态去记住NFA在读输入符号后到达的所有状态。也就是说，在读了输入 $a_1a_2\cdots a_n$ 后，DFA到达一个代表NFA的状态子集 T 的状态。这个子集 T 是从NFA的开始状态沿着那些标有 $a_1a_2\cdots a_n$ 的路径能到达的所有状态的集合。DFA的状态数有可能是NFA状态数的指数。但实际上，这种最坏的情况很少发生。

算法3.2（子集构造算法）从NFA构造DFA。

- **输入：** 一个NFA N 。
- **输出：** 一个接受同样语言的DFA D 。
- **方法：** 为 D 构造转换表 $Dtran$ ，DFA的每个状态是NFA的状态集， D 将“并行”地模拟 N 对输入串的所有可能的移动。

操 作	描 述
ϵ -closure(s)	从NFA状态 s 只经过 ϵ 转换可以到达的NFA状态集
ϵ -closure(T)	从 T 中的状态只经过 ϵ 转换可以到达的NFA状态集
$move(T, a)$	从 T 中的状态 s 经过输入符号 a 上的转换可以到达的NFA状态集

图3-24 NFA状态上的操作



- 在读第一个输入符号前， N 可以处于集合 $\epsilon\text{-closure}(s_0)$ 中的任何状态上，其中 s_0 是 N 的开始状态。假定从 s_0 出发经过输入字符串上的一系列移动， N 到达集合 T 中的状态。令 a 是下一个输入符号。遇到 a 时， N 可以移动到集合 $\text{move}(T, a)$ 中的任何状态。由于允许转换，遇到 a 以后， N 可以处于 $\epsilon\text{-closure}(\text{move}(T, a))$ 中的任何状态。

```
初始时,  $\epsilon$ -closure( $s_0$ )是 $Dstates$ 中惟一的状态且未被标记;  
while  $Dstates$ 中存在一个未标记的状态  $T$  do begin  
    标记 $T$ ;  
    for 每个输入符号  $a$  do begin  
         $U := \epsilon$ -closure(move( $T, a$ ));  
        if  $U$  没在 $Dstates$ 中 then  
            将 $U$  作为一个未标记的状态添加到  $Dstates$  中;  
             $Dtran[T, a] := U$   
        end  
    end  
end
```

图3-25 子集构造法

```
将 $T$ 中所有的状态压入栈  $stack$  中;  
将 $\epsilon$ -closure( $T$ )初始化为  $T$ ;  
while 栈  $stack$  不空 do begin  
    将栈顶元素  $t$  弹出栈;  
    for 每个这样的状态  $u$ : 从  $t$  到  $u$  有一条标记为  $\epsilon$  的边 do  
        if  $u$  不在  $\epsilon$ -closure( $T$ )中 do begin  
            将 $u$ 添加到  $\epsilon$ -closure( $T$ );  
            将 $u$ 压入栈 $stack$ 中  
        end  
    end  
end
```

图3-26 ϵ -closure 的计算

例3.9



- 图3-27给出了接受语言 $(a|b)^*abb$ 的另一个NFA N （它是下一节中从正规表达式开始一步一步地构造出来的NFA）。我们现在把算法3.2运用到 N 。等价的DFA的开始状态是 ϵ -closure(0)，即 $A = \{0, 1, 2, 4, 7\}$ ，其中的每个状态都是从状态0出发经过每条边都由标记的路径能到达的状态。注意，由于路径可以没有边，所以0也是经这样的路径从0能到达的状态。

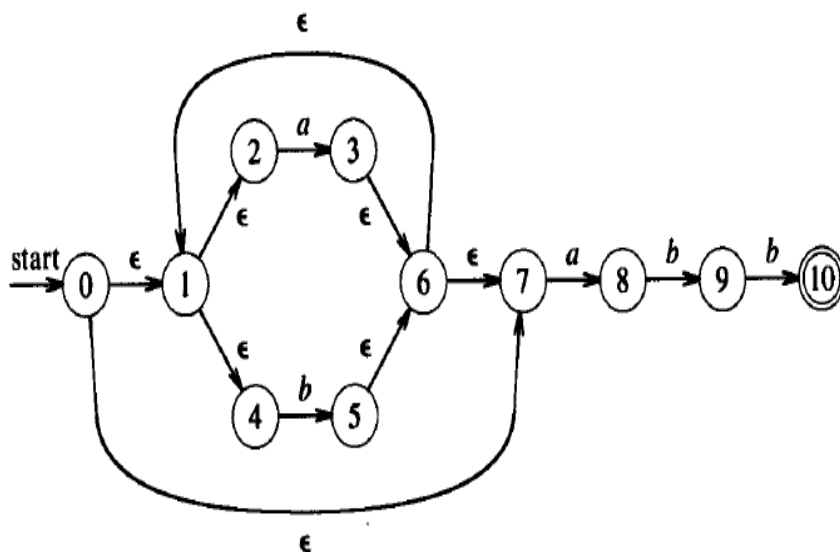
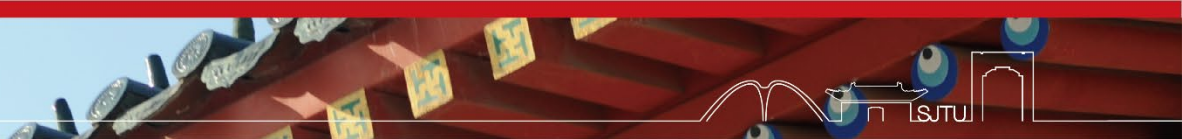


图3-27 $(a|b)^*abb$ 的NFA N



这里的输入符号表是 $\{a, b\}$ 。按图3-25中给出的算法：

(1) 先标记 $A = \{0, 1, 2, 4, 7\}$,

(2) 计算 $\epsilon\text{-closure}(\text{move}(A, a))$ 。让我们首先计算 $\text{move}(A, a)$ ，即对输入 a 从 A 状态可以转换到的 N 的状态集。在状态 $0, 1, 2, 4$ 和 7 中只有 2 和 7 有 a 上的转换，分别到达状态 3 和 8 ,

所以 $\epsilon\text{-closure}(\text{move}(\{0, 1, 2, 4, 7\}, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$

让我们称这个集合为 B 。于是， $D\text{tran}[A, a] = B$ 。

(3) 计算 $\epsilon\text{-closure}(\text{move}(B, a))$ 。在 A 中只有状态 4 对输入 b 有一个转换（转换到状态 5 ），所以DFA对输入 b 有一个从 A 到 C 的转换，

其中： $C = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$ 。因此 $D\text{tran}[A, b] = C$ 。

(4) 对新的没标记过的集合 B 和 C 继续这个过程，最终会使得所有的集合(即DFA的状态)都已标记过。因为包含 11 个状态的集合其不同子集“只有” 2^{11} 个，而且一个集合一旦被标记就永远是标记的，所以这个过程肯定能终止。最终，实际构造出的 5 个不同的状态集合是：



- 图3-27的DFA确定过程:

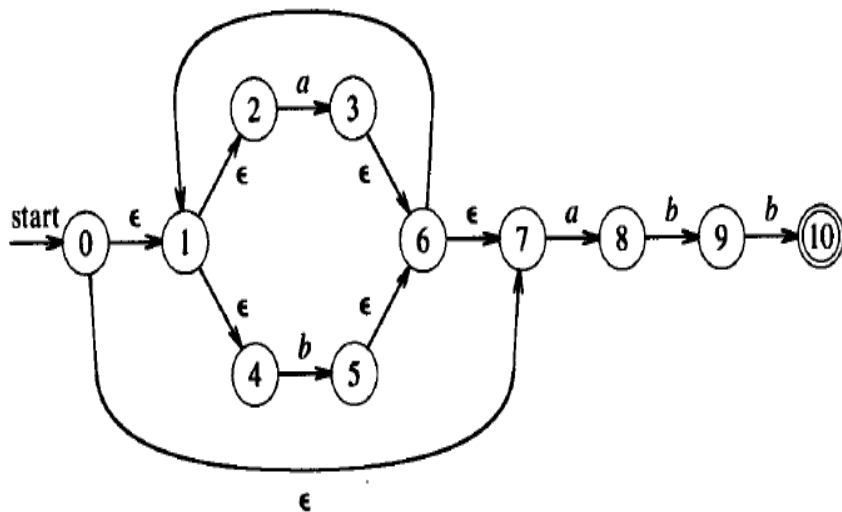


图3-27 $(a|b)^*abb$ 的NFAN

$$\begin{aligned} A &= \{0, 1, 2, 4, 7\} & D &= \{1, 2, 4, 5, 6, 7, 9\} \\ B &= \{1, 2, 3, 4, 6, 7, 8\} & E &= \{1, 2, 4, 5, 6, 7, 10\} \\ C &= \{1, 2, 4, 5, 6, 7\} \end{aligned}$$

状态	输入符号	
	<i>a</i>	<i>b</i>
<i>A</i>	<i>B</i>	<i>C</i>
<i>B</i>	<i>B</i>	<i>D</i>
<i>C</i>	<i>B</i>	<i>C</i>
<i>D</i>	<i>B</i>	<i>E</i>
<i>E</i>	<i>B</i>	<i>C</i>

图3-28 DFA的转换表 *Dtran*

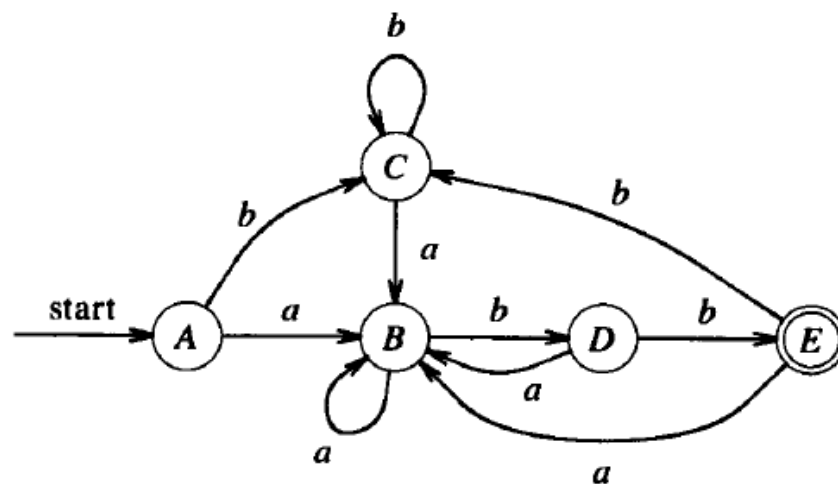


图3-29 对图3-27应用子集构造法得到的结果

3.6.2 最小化DFA的状态数



- 理论上的一个重要结论是：每一个正规集都可以由一个状态数最少的DFA识别，这个DFA是惟一的（状态名不同的同构情况除外）
- 我们说字符串 w 区别状态 s 和 t ，如果：DFA M 从状态 s 出发，对输入串 w 进行状态转换，最后停在某个接受状态；从 t 出发，对输入串 w 进行状态转换，停在一个非接受状态；反之亦然。

算法3.3 最小化DFA的状态数。

- **输入:** DFA M (其状态集合为 S) , 输入符号集为 Σ , 转换函数为 $f: S \times \Sigma \rightarrow S$, 开始状态为 s_0 , 接受状态集为 F 。
- **输出:** 一个DFA M' , 它和 M 接受同样的语言, 且状态数最少。
- **方法:**
 - 1. 构造具有两个组的状态集合的初始划分 Π : 接受状态组 F , 非接受状态组 $S - F$ 。
 - 2. 对 Π 采用图3-45所述的过程来构造新的划分 Π_{new} 。
 - 3. 如果 $\Pi_{\text{new}} = \Pi$, 令 $\Pi_{\text{final}} = \Pi$, 再执行步骤4; 否则, 令 $\Pi := \Pi_{\text{new}}$, 重复步骤2。
 - 4. 在划分 Π_{final} 的每个状态组中选一个状态作为该组的代表。这些代表构成了简化后的DFA M' 的状态。令 s 是一个代表状态, 而且假设: 在DFA M 中, 在输入 a 上有从 s 到 t 的转换。令 t 所在组的代表是 r (r 可能就是 t), 那么在 M' 中有一个从 s 到 r 的 a 上的转换。令包含 s_0 的状态组的代表是 M' 的开始状态, 并令 M' 的接受状态是那些属于 F 集的状态所在组的代表。注意, Π_{final} 的每个组或者仅含 F 中的状态, 或者不含 F 中的状态。

- 5. 如果 M 含有死状态（即一个对所有输入符号都有到自身的转换的非接受状态 d ），则从 M 中去掉它；删除从开始状态不可到达的状态；取消从任何其他状态到死状态的转换定义。

for Π 中的每个组 G **do begin**

当且仅当对任意输入符号 a ，状态 s 和 t 在 a 上的转换到达 Π 的同一组中的状态时，才把 G 划分成小组，以便 G 的两个状态 s 和 t 在同一小组中；

/* 最坏情况下，一个状态就可能成为一个组 */

用所有新形成的小组集代替 Π_{new} 中的 G ；

end

图3-45 Π_{new} 的构造



让我们重新考虑图3-29中给出的DFA。

- (1) 初始划分 Π 包括两个组：接受状态组(E)和非接受状态组($ABCD$)。
- (2) 构造 Π_{new} 时，首先考虑(E)。因为这个组只包含一个状态，它不能再划分，所以把(E)仍放回 Π_{new} 中。然后，算法考虑($ABCD$)。对于输入 a ，这些状态都转换到 B ，因此分组($ABCD$)不变；但对于输入 b ， A 、 B 和 C 都转换到状态组($ABCD$)的一个成员，而 D 转换到另一组的成员 E 。于是，在 Π_{new} 中，状态组($ABCD$)必须分裂成两个新组(ABC)和(D)。因而 Π_{new} 成了(ABC)(D)(E)。
- (3) 再执行一遍图3-45中的算法时，在输入 a 上仍然没有分裂，但对输入 b ，(ABC)还要划分，因为 A 和 C 都转到 C ，但 B 转到 D ，而 C 和 D 不在一个组中，于是 Π_{new} 的下一个值是(AC)(B)(D)(E)。

- (4) 再执行一遍图3-45中的算法时，只有 (AC) 有划分的可能。但是对于输入 a ， A 和 C 都转换到 B ，对输入 b ，它们都转换到状态 C ，因而不必再划分。这次循环结束时， $\Pi_{\text{new}} = \Pi$ 。于是， Π_{final} 是 $(AC)(B)(D)(E)$ 。
- (5) 如果我们选择 A 作为 (AC) 的代表，选择 B 、 D 和 E 作为其他单状态组的代表，最后可以得到简化的自动机。它的转换表如图3-46所示，状态 A 是开始状态，状态 E 是惟一的接受状态。

状 态	输 入 符 号	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

图3-46 简化的DFA的转换表

例：图3-29中给出的DFA 的最小化过程

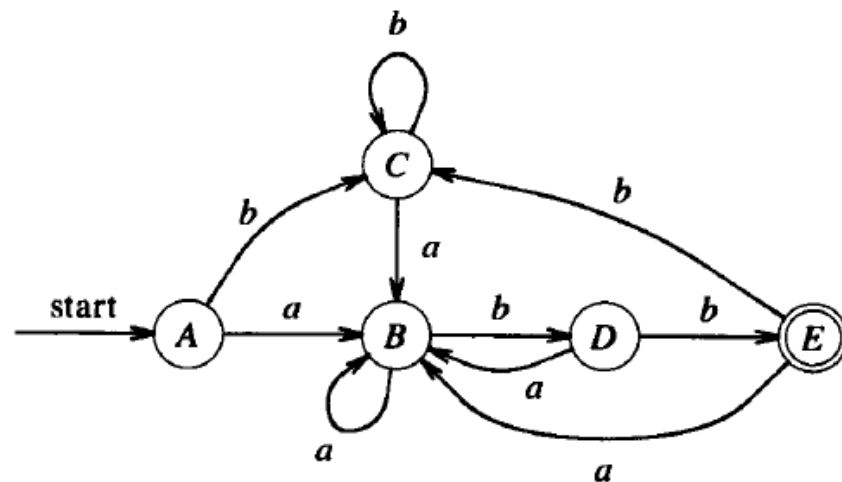


图3-29 对图3-27应用子集构造法得到的结果

3.6.3 从正规表达式构造NFA



算法3.4 (从正规表达式构造NFA。

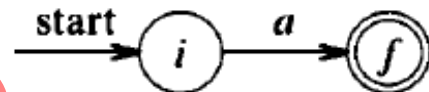
- 输入：字母表 Σ 上的一个正规表达式 r 。
- 输出：接受 $L(r)$ 的NFA N 。

- 方法: 首先, 分析 r 并将其分解成最基本的子表达式, 然后使用下面的规则1和规则2为 r 中的每个基本符号(或字母表中的符号)构造NFA。基本符号对应正规表达式定义的1和2两部分。请注意, 如果符号 a 在 r 中出现多次, 则要为它的每次出现构造一个NFA。

- 规则 1: 对 ϵ , 构造NFA: 

其中, i 是新的开始状态, f 是新的接受状态。很明显这个NFA识别 $\{\epsilon\}$ 。

- 规则 2: 对于 S 中的每个符号 a , 构造NFA:

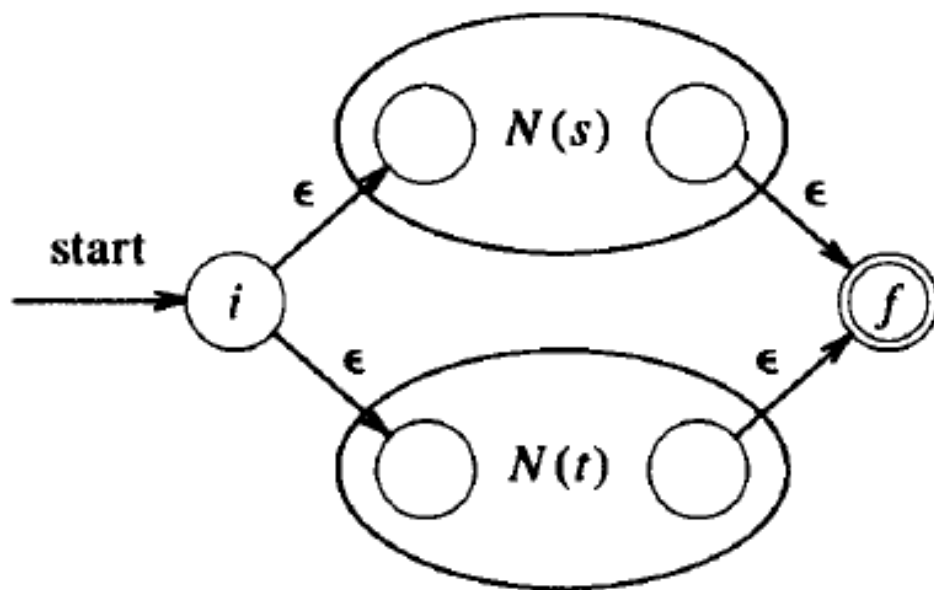


同样, i 是新的开始状态, f 是新的接受状态。这个NFA识别 $\{a\}$ 。

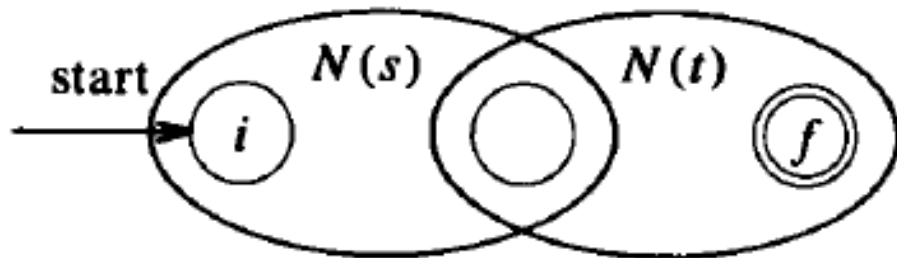
然后，由正规表达式 r 的运算规则，用下面的规则3逐步地组合前面构造的NFA，直到获得整个正规表达式的NFA为止。在构造过程中所产生的中间NFA（与 r 的子表达式对应）

有几个重要的性质：只有一个终态；开始状态无入边，终态无出边。

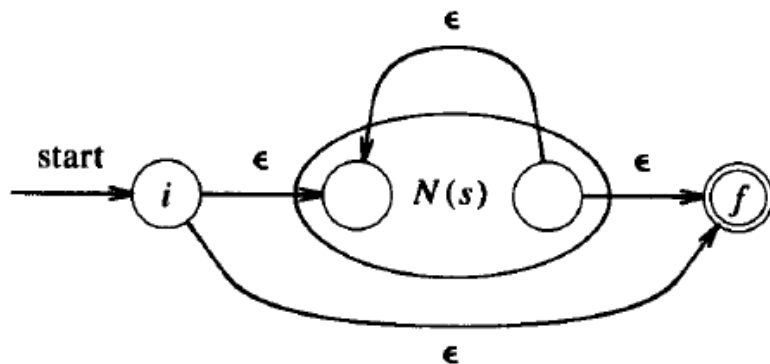
- 规则3.如果 $N(s)$ 和 $N(t)$ 是正规表达式 s 和 t 的NFA，则：
(a) 对于正规表达式 $s|t$ ，可构造复合的NFA $N(s|t)$ 如下：



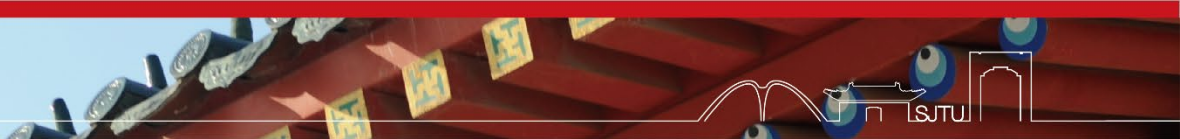
- (b) 对于正规表达式 st , 可构造复合的NFA $N(st)$ 如下:



- (c) 对于正规表达式 s^* , 可构造复合的NFA $N(s^*)$ 如下:



- (d) 对于括起来的正规表达式 (s) , 使用 $N(s)$ 本身作为它的NFA。



- 可以验证，算法3.4构造的每一步都产生识别对应语言的NFA。此外，产生的NFA具有下列性质：
- 1. $N(r)$ 的状态数最多是 r 中符号和运算符个数的两倍。因为构造的每步最多引入两个新状态。
- 2. $N(r)$ 只有一个开始状态和一个接受状态，开始状态没有入边，接受状态没有出边。
- 3. $N(r)$ 除接受状态外的每个状态或者有一个用 Σ 中的符号标记的出边，或者至多有两个标记为 ϵ 的出边。

- 我们用算法3.4构造正规表达式 $r = (a|b)^*abb$ 的 $N(r)$ 。
图3-30是 r 的分析树。这个分析树类似于2.2节中为算术表达式构造的分析树。

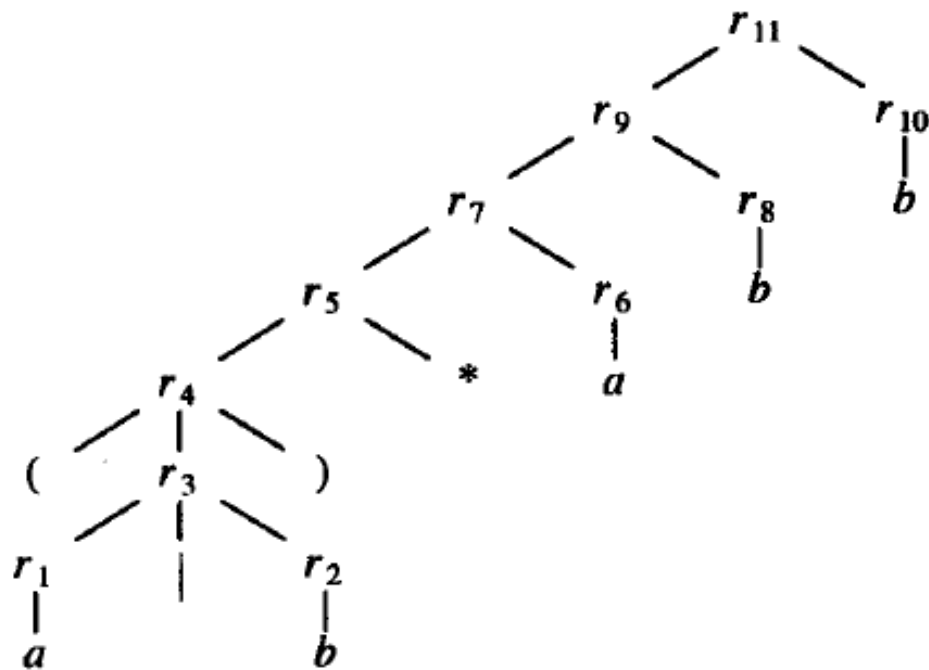
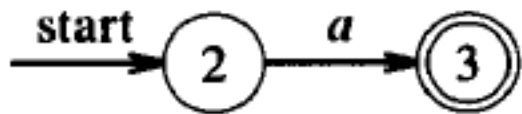


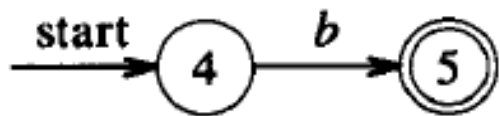
图3-30 $(a|b)^*abb$ 的分解



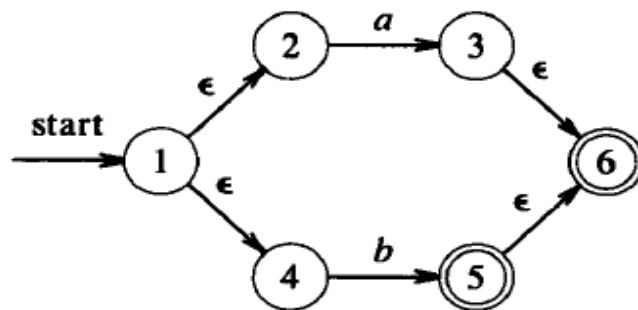
- 对成份 r_1 (即第一个 a)，构造它的NFA如下：



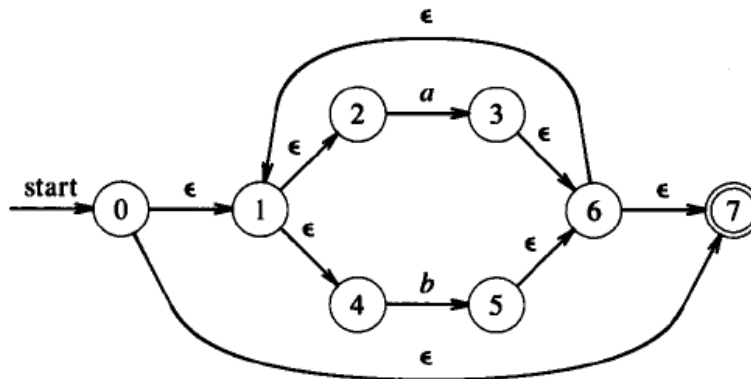
- 对 r_2 ，构造它的NFA如下：



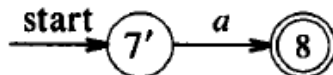
- 再用并规则组合 $N(r_1)$ 和 $N(r_2)$ ，得到 $r_3 = r_1 | r_2$ 的NFA如下：



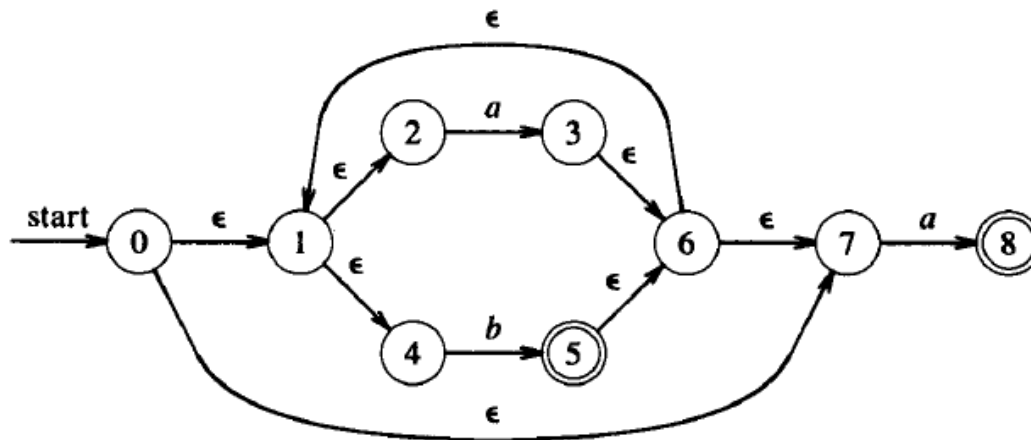
- (r_3) 的NFA和 r_3 的一样。 $(r_3)^*$ 的NFA构造如下:



- $r_6 = a$ 的NFA如下:



- 我们合并状态7和7' (称其为状态7) 得到 $r_5 r_6$ 的自动机, 该自动机如下:



- 这样依次做下去，最后得到 $r_{11}=(a|b)^*abb$ 的NFA，如图3-27所示。

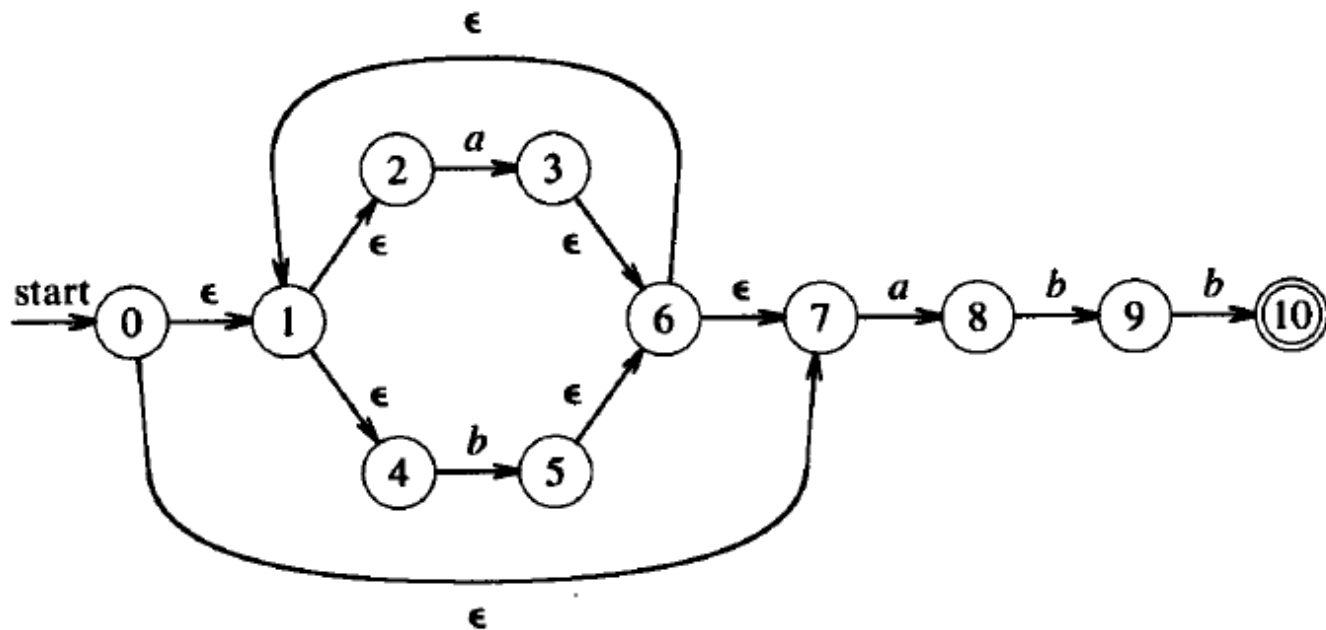


图3-27 $(a|b)^*abb$ 的NFA

3.6.4 从正规表达式构造NFA的另外一种规则



有很多从正规表达式变成NFA的策略，本节我们将可以介绍另外一种规则构造NFA:

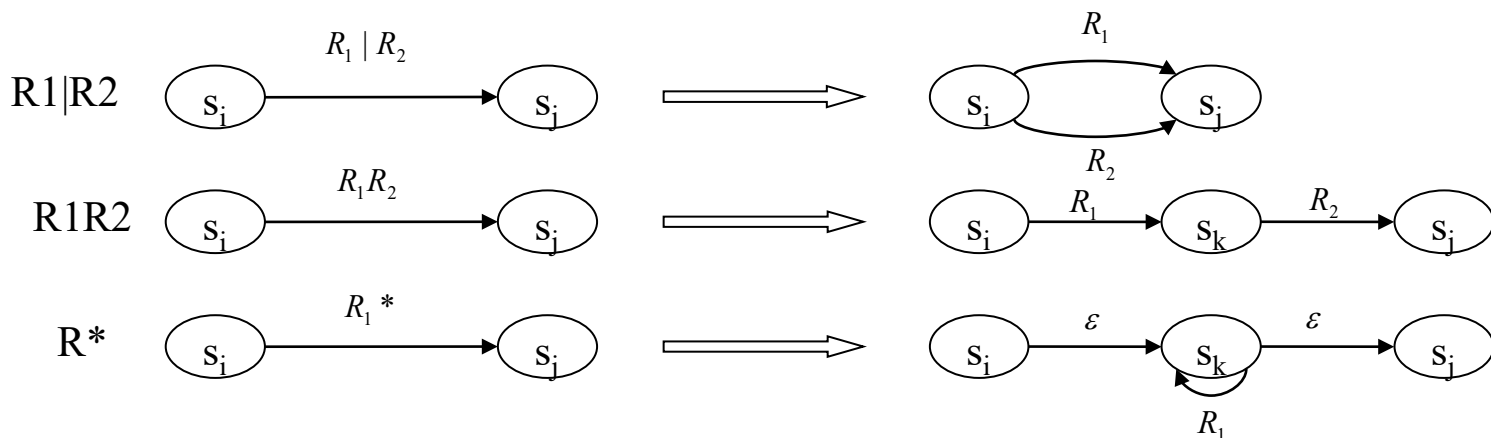


图3.28 转换规则

例：接受语言 $(a|b)^*abb$ 的最小化的完整过程



第一步：从正规表达式构造NFA。

例：接受语言 $(a|b)^*abb$ 的最小化的完整过程



第二步：从NFA构造DFA。

例：接受语言 $(a|b)^*abb$ 的最小化的完整过程



第三步：将DFA最小化。

第三章 总结与作业



主要内容：

- 程序语言的词法定义方法：正规表达式
- 程序语言的词法分析器的自动生成方法：有限自动机

课后要求：

- 完成课后作业：

下列 $\Sigma = \{a, b\}$ 上的正规式的等价的状态最少的DFA是什么？

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

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

谢谢!



上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

上海交通大学

