第2章 词法分析

本章要点

- 扫描处理
- 正则表达式
- 有穷自动机
- 从正则表达式到DFA
- TINY扫描程序的实现
- 利用Lex自动生成扫描程序

编译器的扫描或词法分析（lexical analysis）阶段可将源程序读作字符文件并将其分为若干个记号。记号与自然语言中的单词类似：每一个记号都是表示源程序中信息单元的字符序列。典型的有：关键字（keyword），例如**if**和**while**，它们是字母的固定串；标识符（identifier）是由用户定义的串，它们通常由字母和数字组成并由一个字母开头；特殊符号（special symbol）如算术符号+和*、一些多字符符号，如> = 和< >。在各种情况中，记号都表示由扫描程序从剩余的输入字符的开头识别或匹配的某种字符格式。

由于扫描程序的任务是格式匹配的一种特殊情况，所以需要研究在扫描过程中的格式说明和识别方法，其中最主要的是正则表达式（regular expression）和有穷自动机（finite automata）。但是，扫描程序也是处理源代码输入的编译器部分，而且由于这个输入经常需要非常多的额外时间，扫描程序的操作也就必须尽可能地高效了。因此还需十分注意扫描程序结构的实际细节。

扫描程序问题的研究可分为以下几个部分：首先，给出扫描程序操作的一个概貌以及所涉及到的结构和概念。其次是学习正则表达式，它是用于表示构成程序设计语言的词法结构的串格式的标准表示法。接着是有穷状态机器或称有穷自动机，它是对由正则表达式给出的串格式的识别算法。此外还研究用正则表达式构造有穷自动机的过程。之后再讨论当有穷自动机表示识别过程时，如何实际编写执行该过程的程序。另外还有 TINY语言扫描程序的一个完整的实现过程。最后再看到自动产生扫描器生成器的过程和方法，并使用 Lex再次实现TINY的扫描程序，它是适用于Unix和其他系统的标准扫描生成器。

2.1 扫描处理

扫描程序的任务是从源代码中读取字符并形成由编译器的以后部分（通常是分析程序）处理的逻辑单元。由扫描程序生成的逻辑单元称作记号（ token），将字符组合成记号与在一个英语句子中将字母构成单词并确定单词的含义很相像。此时它的任务很像拼写。

记号通常定义为枚举类型的逻辑项。例如，记号在C中可被定义为⊖：

⊖ L在一种没有列举类型的语言中，则只能将记号直接定义为符号数值。因此在老版本的 C中有时可看到：

```
#define  IF  256
#define  THEN  257
#define  ELSE  258
...
```

(这些数之所以是从256开始是为了避免与ASCII的数值混淆。)

```
typedef enum
    { IF, THEN, ELSE, PLUS, MINUS, NUM, ID, ...}
    TokenType;
```

记号有若干种类型，这其中包括了保留字（reserved word），如**IF**和**THEN**，它们表示字符串“ if ”和“ then ”；第2类是特殊符号（ special symbol），如算术符号加（ PLUS）和减（MINUS），它们表示字符“ + ”和“ - ”。第3类是表示多字符串的记号，如 **NUM**和**ID**，它们分别表示数字和标识符。

作为逻辑项的记号必须与它们所表示的字符串完全区分开来。例如：保留字记号 **IF**须与它表示的两个字符“ if ”的串相区别。为了使这个区别更明显，由记号表示的字符串有时称作它的串值（string value）或它的词义（lexeme）。某些记号只有一个词义：保留字就具有这个特性。但记号还可能表示无限多个语义。例如标识符全由单个记号 **ID**表示，然而标识符有许多不同的串值来表示它们的单个名字。因为编译器必须掌握它们在符号表中的情况，所以不能忽略这些名字。因此，扫描程序也需用至少一些记号来构造串值。任何与记号相关的值都是记号的属性（attribute），而串值就是属性的示例。记号还可有其他的属性。例如， **NUM**记号可有一个诸如“ 32767 ”的串值属性，它是由5个数字字符组成，但它还会有一个由其值计算所得的真实值32767组成的数字值属性。在诸如**PLUS**这样的特殊符号记号中，不仅有串值“ + ”还有与之相关的真实算术操作 +。实际上，记号符号本身就可看作是简单的其他属性，而记号就是它所有属性的总和。

为了后面的处理，扫描程序要求至少具有与记号所需相等的属性。例如要计算 **NUM**记号的串值，但由于从它的串值就可计算，因此也就无需立刻计算它的数字值了。另一方面，如果计算它的数字值，就会丢弃掉它的串值。有时扫描程序本身会完成在恰当位置记录属性所需的操作，或直接将属性传到编译器后面的阶段。例如，扫描程序能利用标识符的串值将其输入到符号表中，或在后面传送它。

由于扫描程序必须计算每一个记号的若干属性，所以将所有的属性收集到一个单独构造的数据类型中是很有用的，这种数据类型称作记号记录（ token record）。可在C中将这样的记录说明为：

```
typedef struct
  {  TokenType tokenval;
     char * stringval;
     int numval;
  }  TokenRecord;
```

或可能作为一个联合

```
typedef struct
     { TokenType tokenval;
       union
       { char * stringval;
         int numval;
       } attribute;
     } TokenRecord;
```

（以上假设只有标识符需要串值属性，只有数字需要数值属性）。对于扫描程序一个更普通的安排是只返回记号值并将变量的其他属性放在编译器的其他部分访问得到的地方。

尽管扫描程序的任务是将整个源程序转换为记号序列，但扫描程序却很少一次性地完成它。实际上，扫描程序是在分析程序的控制下进行操作的，它通过函数从输入中返回有关命令的下

一个记号，该函数有与C说明相似的说明：

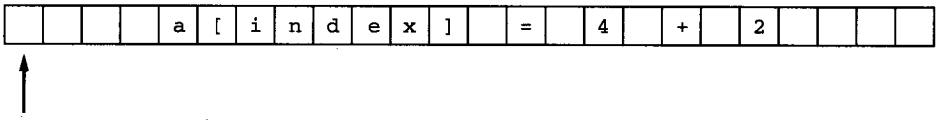
```
TokenType getToken( void );
```

这个方式中声明的**getToken**函数在调用时从输入中返回下一个记号，并计算诸如记号串值这样的附加属性。输入字符串通常并不给这个函数提供参数，但参数却被保存在缓冲区中或由系统输入设备提供。

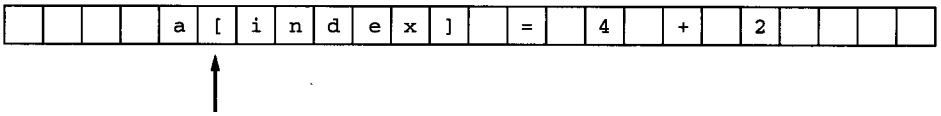
请考虑在**getToken**的操作示例中以下的C源代码行，在第1章中已用到过它：

```
a [ index ] = 4 + 2
```

假定将这个代码行放在一个输入缓冲区中，它的下一个输入字符由箭头指出，如下所示：



一个对**getToken**的调用现在需要跳过下面的4个空格，识别由单个字符*a* 组成的串“ a ”，并返回记号值**ID**作为下个记号，此时的输入缓冲区如下所示：



因此，**getToken**随后的调用将再次从左括号字符开始识别过程。

现在开始研究在字符串中定义和识别格式的方法。

2.2 正则表达式

正则表达式表示字符串的格式。正则表达式 r完全由它所匹配的串集来定义。这个集合称为由正则表达式生成的语言（language generated by the regular expression），写作L（r）。此处的语言只表示“ 串的集合 ”，它与程序设计语言并无特殊关系（至少在此处是这样的）。该语言首先依赖于适用的字符集，它一般是 ASCII字符的集合或它的某个子集。有时该集比 ASCII字符的集合更普通一些，此处集合的元素称作符号（ symbol）。这个正规符号的集合称作字母表（alphabet）并且常写作希腊符号∑（sigma）。

正则表达式r还包括字母表中的字符，但这些字符具有不同的含义：在正则表达式中，所有的符号指的都是模式。在本章中，所有模式都是用黑体写出以与作为模式的字符区分开来。因此，**a** 就是一个作为模式的字符a。

最后，正则表达式r还可包括有特殊含义的字符。这样的字符称作元字符（ metacharacter）或元符号（metasymbol）。它们并不是字母表中的正规字符，否则当其作为元字符时就与作为字母表中的字符时很难区分了。但是通常不可能要求这种排斥，而且也必须使用一个惯例来区分元字符的这两种用途。在很多情况下，它是通过使用可“ 关掉 ”元字符的特殊意义的转义字符（escape character）做到的。源代码层一般是反斜杠和引号。如果转义字符是字母表中的正规字符，则请注意它们本身也就是元字符了。

2.2.1 正则表达式的定义

现在通过讲解每个模式所生成的不同语言来描述正则表达式的含义。首先讲一下基本正则

表达式的集合（它是由单个符号组成），之后再描述从已有的正则表达式生成一个新的正则表达式的运算。它同构造算术表达式的方法类似：基本的算术表达式是由数字组成，如43和2.5。算术运算的加法和乘法等可用来从已有的表达式中产生新的表达式，如在43 * 2.5和43 * 2.5 + 1.4中。

从它们只包括了最基本的运算和元符号这一方面而言，这里所讲到的一组正则表达式都是最小的，以后还会讲得更深一些。

1) **基本正则表达式** 它们是字母表中的单个字符且自身匹配。假设a是字母表∑中的任一字符，则指定正则表达式**a** 通过书写L(**a**) = {a}来匹配a字符。而特殊情况还要用到另外两个字符。有时需要指出空串（empty string）的匹配，空串就是不包含任何字符的串。空串用 ε (epsilon)来表示，元符号ε （黑体ε）是通过设定L(ε) = {ε}来定义的。偶尔还需要写出一个与任何串都不匹配的符号，它的语言为空集（empty set），写作{}。我们用符号ϕ来表示，并写作L(ϕ) = {}。请注意{}和{ε}的区别：{}集不包括任何串，而{ε}则包含一个没有任何字符的串。

2) **正则表达式运算** 在正则表达式中有3种基本运算：① 从各选择对象中选择，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。后面通过为匹配了的串的语言提供相应的集合结构依次讨论以上 3种基本运算。

3) **从各选择对象中选择** 如果r 和s 是正则表达式，那么正则表达式r | s 可匹配被r 或s 匹配的任意串。从语言方面来看， r | s 语言是r 语言和s 语言的联合（union），或L (r | s) = L (r) ∪ L (s)。以下是一个简单例子：正则表达式**a** | **b**匹配了a 或b 中的任一字符，即L (**a** | **b**) = L (**a**) ∪ L (**b**) = {a} ∪ {b} = {a, b}。又例如表达式**a** | ε 匹配单个字符a 或空串（不包括任何字符），也就是L (**a** | ε) = {a, ε}。

还可在多个选择对象中选择，因此L (**a** | **b** | **c** | **d**) = {a, b, c, d} 也成立。有时还用点号表示选择的一个长序列，如**a** | **b** |...| **z**，它表示匹配a~z 的任何小写字母。

4) **连结** 正则表达式r 和正则表达式s 的连结可写作rs，它匹配两串连结的任何一个串，其中第1个匹配r，第2个匹配s。例如：正则表达式**ab**只匹配ab，而正则表达式(**a** | **b**) **c** 则匹配串ac 和bc（下面将简要介绍括号在这个正则表达式中作为元字符的作用）。

可通过由定义串的两个集合的连结所生成的语言来讲解连结的功能。假设有串 S_1 和S_2 的两个集合，串S_1S_2 的连结集合是将串S_2 完全附加到串S_1 上的集合。例如若S_1 = {aa, b}, S_2 = {a, bb}，则S_1S_2 = {aaa, $aabb$, ba, bbb}。现在可将正则表达式的连结运算定义如下：L (rs) = L (r) L (s)，因此（利用前面的示例），L ((**a** | **b**) **c**) = L (**a** | **b**) L (**c**) = {a, b} {c} = {ac, bc}。

连结还可扩展到两个以上的正则表达式：L (r_1r_2... r_n) = L (r_1) L (r_2) ... L (r_n) = 由将每一个L (r_1), ..., L (r_n) 串连结而成的串集合。

5) **重复** 正则表达式的重复有时称为Kleene闭包（(Kleene) closure），写作**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。例如**a***匹配串ε、a、aa、aaa ...（它与ε 匹配是因为ε 是与**a**匹配的非串连结）。通过为串集合定义一个相似运算 *，就可用生成的语言定义重复运算了。假设有一个串的集合S，则

$$S^* = \{\varepsilon\} \quad S \quad SS \quad SSS \quad \ldots$$

这是一个无穷集的联合，但是其中的每一个元素都是S中串的有穷连结。有时集合S*可写作：

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

其中$S^n = S \ldots S$，它是S 的n 次连结（S^0 = {ε}）。

现在可如下定义正则表达式的重复运算：

$$L(r^*) = L(r)^*$$

例：在正则表达式(**a** | **bb**) * (括号作为元字符的原因将在后面解释）中，该正则表达式与以下串任意匹配：ε、*a*、*bb*、*aa*、*abb*、*bba*、*bbbb*、*aaa*、*aabb* 等等。在语言方面，*L* ((**a** | **bb**)*) = *L* (**a** | **bb**)* = {*a, bb*}* = {ε , *a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa,* . . . }。

6) **运算的优先和括号的使用** 前面的内容忽略了选择、连结和重复运算的优先问题。例如对于正则表达式**a** | **b***，是将它解释为(**a** | **b**)* 还是**a** | (**b***) 呢（这里有一个很大的差别，因为*L* ((**a** | **b**)*) = { ε, *a, b, aa, ab, ba, bb,* . . . }，但*L* (**a** | (**b***)) = {ε, *a, b, bb, bbb,* . . . })？标准惯例是重复的优先权较高，所以第 2个解释是正确的。实际上，在这 3个运算中，*优先权最高，连结其次，| 最末。因此，**a** | **bc*** 就可解释为**a** | (**b** (**c***))，而**ab** | **c*d** 却解释为(**ab**) | ((**c***) **d**)。

当需指出与上述不同的优先顺序时，就必须使用括号。这就是为什么用 (**a** | **b**) **c** 能表示选择比连结有更高的优先权的原因。而**a** | **bc** 则被解释为与*a* 或*bc* 匹配。类似地，没有括号的(**a** | **bb**)* 应解释为**a** | **bb***，它匹配*a*、*b*、*bb*、*bbb*. . . 。此处括号的用法与其在算术中类似：(3 + 4) * 5 =35，而3 + 4 * 5 = 23，这是因为* 的优先权比+ 的高。

7) **正则表达式的名字** 为较长的正则表达式提供一个简化了的名字很有用处，这样就不再需要在每次使用正则表达式时书写长长的表达式本身了。例如：如要为一个或多个数字序列写一个正则表达式，则可写作：

```
(0|1|2|...|9)(0|1|2|...|9)*
```

或写作

```
digit digit*
```

其中

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

就是名字***digit***的正则定义（regular definition）了。

正则定义的使用带来了巨大的方便，但是它却增加了复杂性，它的名字本身也变成一个元符号，而且必须要找到一个方法能将这个名字与它的字符连结相区分开。在我们的例子中是用斜体来表示它的名字。请注意，在名字的定义中不能使用名字（也就是递归地）——必须能够用它代表的正则表达式替换它们。

在考虑用一些例子来详细描述正则表达式的定义之前，先将有关正则表达式定义的所有信息收集在一起。

定义 正则表达式（regular expression）是以下的一种：

1. 基本（basic）正则表达式由一个单字符**a**（其中*a* 在正规字符的字母表∑中），以及元字符ε或元字符ϕ 组成。在第1种情况下，*L* (**a**) = {*a*}；在第2种情况下，*L* (ε) = {ε}；在第3种情况下，*L* (ϕ) = {}。
2. **r** | **s** 格式的表达式：其中*r* 和*s* 均是正则表达式。在这种情况下，*L* (**r** | **s**) = *L* (*r*) *L* (*s*)。
3. **rs** 格式的表达式：其中*r* 和*s* 是正则表达式。在这种情况下，*L* (*rs*) = *L* (*r*) *L* (*s*)。
4. *r** 格式的表达式：其中*r* 是正则表达式。在这种情况下，*L* (*r**) = *L* (*r*)*。
5. (*r*) 格式的表达式：其中*r* 是正则表达式。在这种情况下，*L* ((*r*)) = *L* (*r*)，因此，括号并不改变语言，它们只调整运算的优先权。

我们注意到在上面这个定义中，（2）、（3）和（4）的优先权与它们所列的顺序相反，也就

是：|的优先权最低，连结次之，*则最高。另外还注意到在这个定义中，6个符号——ϕ、ε、|、*、(和)都有了元字符的含义。

本节后面将给出一些示例来详述上述定义，但它们并不经常作为记号描述在程序设计语言中出现。2.2.3节将讨论一些经常作为记号在程序设计语言中出现的常用正则表达式。

在下面的示例中，被匹配的串通常是英语描述，其任务是将描述翻译为正则表达式。包含了记号描述的语言手册是编译器的程序员最常见的。偶尔还需要变一下，也就是将正则表达式翻译为英语描述，我们也有一些此类的练习。

例2.1 在仅由字母表中的3个字符组成的简单字母表∑ = {*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，这个集合由正则表达式

```
(a|c)*b(a|c)*
```

产生。尽管**b**出现在正则表达式的正中间，但字母*b* 却无需位于被匹配的串的正中间。实际上，在*b* 之前或之后的*a* 或*c* 的重复会发生不同的次数。因此，串*b*、*abc*、*abaca*、*baaaac*、*ccbaca*和*ccccccb* 都与上面的正则表达式匹配。

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，则这个集合的正则表达式可通过将上例的解作为一个解（与那些仅为一个 *b* 的串匹配），而正则表达式(**a** | **c**) *则作为另一个解（与*b* 根本不匹配）来获取。因此有以下解：

```
(a|c)*|(a|c)*b(a|c)*
```

下面是一个既允许*b* 又允许空串在重复的*a* 或*c* 之间出现的另一个解：

```
(a|c)*(b|ε)(a|c)*
```

本例引出了正则表达式的一个重要问题：不同的正则表达式可生成相同的语言。虽然在实际中从未尝试着证实已找到了“最简单的”，例如最短的，正则表达式，但通常总是试图用尽可能简单的正则表达式来描述串的集合。这里有两个原因：首先在现实中极少有标准的“最短的”解；其次，在研究用作识别正则表达式的方法时，那儿的算法无需首先将正则表达式简化就可将识别过程简化了。

例2.3 在字母表∑= {*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：

$$S = \{\ b,\ aba,\ aabaa,\ aaabaaa,\ \ldots\ \} = \{\ a^n\ b\ a^n \mid n \neq 0\ \}$$

正则表达式并不能描述这个集合，其原因在于重复运算只有闭包运算 ***一种，它允许有任意次的重复。因此如要写出表达式**a*ba**（尽可能接近地得到*S*的正则表达式），就不能保证在*b* 前后的*a* 的数量是否相等了，它通常表示为“不能计算的正则表达式”。但若要给出一个数学论证，则需使用有关正则表达式的称作 Pumping引理（Pumping lemma）的著名定理，这将在自动机理论中学到，现在就不谈了。

很明显，并非用简单术语描述的所有串都可由正则表达式产生，因此为了与其他集合相区分，作为正则表达式语言的串集合称作正则集合（regular set）。非正则集合偶尔也作为串出现在需要由扫描程序识别的程序设计语言中，通常是在出现时才处理它们，我们也将其放在扫描程序一节中讨论。

例2.4 在字母表∑= {*a*, *b*, *c*}上的串*S*中，任意两个*b* 都不相连，所以在任何两个 *b* 之间都至少有一个*a* 或*c*。可分几步来构造这个正则表达式，首先是在每一个 *b* 后都有一个*a* 或*c*，它写作：

```
(b(a|c))*
```

将其与表达式**(a|c)***合并，**(a|c)***是与完全不包含*b*的串匹配，则写作：

```
((a|c)*|(b(a|c))*)*
```

或考虑到**(r*|s*)***与**(r|s)***所匹配的串相同，则：

```
((a|c)|(b(a|c)))*
```

或

```
(a|c|ba|bc)*
```

（警告！这还不是正确答案）。

这个正则表达式产生的语言实际上具有了所需的特性，即：没有两个相连的*b*（但这还不正确）。有时需要证明一下上面的这个说法，也就是证明在*L*(**(a|c|ba|bc)***)中的所有串都不包括两个相连的*b*。该证明是通过对串长度（即串中字符数）的归纳实现的。很显然，它对于所有长度为0、1和2的串是正确的：这些串实际是串ε、*a*、*c*、*aa*、*ac*、*ca*、*cc*、*ba* 和*bc*。现在假设对于在长度$i < n$ 的语言中的所有串也为真，并令*s* 是长度为$n > 2$ 的语言中的一个串，那么*s* 包含了至少一个上面所列的非ε 的串，所以$s = s_1s_2$，其中s_1 和s_2 也是语言中的非ε 串。通过假设归纳，证明了s_1 和s_2 都没有两个相连的*b*。因此要使*s* 本身包括两个相连的*b* 的唯一方法是使s_1 以一个*b* 结尾，而s_2 以一个*b* 开头。但又因为语言中的串都不以*b* 结尾，所以这是不可能的。

论证中的最后一个事实，即由上面的正则表达式所产生的串都不以 *b* 结尾，也显示了我们的解还不太正确：它不产生串 *b*、*ab*和*cb*，这3个都不包括两个相连的 *b*。可以通过为其添加一个可选的结尾*b*来修改它，如下所示：

```
(a|c|ba|bc)*(b|ε)
```

这个正则表达式的镜像也生成了指定的语言：

```
(b|ε)(a|c|ab|cb)*
```

以下也可生成相同的语言：

```
(notb|b notb)*(b|ε)
```

其中***notb*** **= a|c**。这是一个使用了下标表达式名字的例子。由于无需将原表达式变得更复杂就可使***notb***的定义调整为包括了除*b* 以外的所有字符，因此实际是在字母表较大时使用这个解。

例2.5　本例给出了一个正则表达式，要求用英语简要地描述它生成的语言。若有字母表 ∑= {*a*, *b*, *c*}，则正则表达式：

```
((b|c)*a(b|c)*a)*(b|c)*
```

生成了所有包括偶数个*a* 的串的语言。为了看清它，可考虑外层左重复之中的表达式：

```
(b|c)*a(b|c)*a
```

它生成的串是以*a* 结尾且包含了两个*a*（在这两个*a* 之前或之间可有任意个*b* 和*c*）。重复这些串则得到所有以*a* 结尾的串，且*a* 的个数是2的倍数（即偶数）。在最后附加重复**(b|c)***（如前例所示）则得到所需结果。

这个正则表达式还可写作：

```
(nota* a nota* a)* nota*
```

2.2.2 正则表达式的扩展

前面已给出了正则表达式的一个定义，这个正则表达式使用了在所有应用程序中都常见到

运算的最小集合，而且使上面的示例仅限于使用3种基本运算（包括括号）。但是从以上这些示例中可看出仅利用这些运算符来编写正则表达式有时显得很笨拙，如果可用一个更有表达力的运算集合，那么创建的正则表达式就会更复杂一些。例如，使任意字符的匹配具有一个表示法很有用（我们现在须在一个长长的解中列出字母表中的每个字符）。除此之外，拥有字符范围的正则表达式和除单个字符以外所有字符的正则表达式都十分有效。

下面几段文字将描述前面所讨论的标准正则表达式的一些扩展情况，以及与它及类似情况相对应的新元符号。在大多数情况下并不存在通用术语，所以使用的是与在扫描程序生成器Lex中用到的类似的表示法，本章后面将会讲到 Lex。实际上，以后要谈到的很多情况都会在对Lex的讨论中再次提到。并非所有使用正则表达式的应用程序都包括这些运算，即使是这样，也要用到不同的表示法。

下面是新运算的列表。

(1) 一个或多个重复

假若有一个正则表达式*r*，*r* 的重复是通过使用标准的闭包运算来描述，并写作*r**。它允许 *r* 被重复0次或更多次。0次并非是最典型的情况，一次或多次才是，这就要求至少有一个串匹配*r*，但空串ε却不行。例如在自然数中需要有一个数字序列，且至少要出现一个数字。如要匹配二进制数，就写作**(0|1)***，它同样也可匹配不是一个数的空串。当然也可写作

```
(0|1)(0|1)*
```

但是这种情况只出现在用**+**代替*****的这个相关的标准表示法被开发之前：**r+**表明*r* 的一个或多个重复。因此，前面的二进制数的正则表达式可写作：

```
(0|1)+
```

(2) 任意字符

为字母表中的任意字符进行匹配需要一个通常状况：无需特别运算，它只要求字母表中的每个字符都列在一个解中。句号“ .”表示任意字符匹配的典型元字符，它不要求真正将字母表写出来。利用这个元字符就可为所有包含了至少一个 *b* 的串写出一个正则表达式，如下所示：

```
.*b.*
```

(3) 字符范围

我们经常需要写出字符的范围，例如所有的小写字母或所有的数字。直到现在都是在用表示法**a|b|...|z** 来表示小写字母，用**0|1|...|9**来表示数字。还可针对这种情况使用一个特殊表示法，但常见的表示法是利用方括号和一个连字符，如 **[a-z]**是指所有小写字母，**[0-9]**则指数字。这种表示法还可用作表示单个的解，因此 **a|b|c**可写成**[abc]**，它还可用于多个范围，如**[a-zA-Z]**代表所有的大小写字母。这种普遍表示法称为字符类（character class）。例如，**[A-Z]**是假设位于*A*和*Z*之间的字符*B*、*C*等（一个可能的假设）且必须只能是*A*和*Z*之间的大写字母（ASCII字符集也可）。但**[A-z]**则与**[A-Za-z]**中的字符不匹配，甚至与ASCII字符集中的字符也不匹配。

(4) 不在给定集合中的任意字符

正如前面所见的，能够使要匹配的字符集中不包括单个字符很有用，这点可由用元字符表示“非”或解集合的互补运算来做到。例如，在逻辑中表示“非”的标准字符是波形符“ ~”，那么表示字母表中非*a* 字符的正则表达式就是**~a**。非*a*、*b* 及*c* 表示为：

```
~(a|b|c)
```

在Lex中使用的表示法是在连结中使用插入符“^”和上面所提的字符类来表示互补。例如，

任何非*a* 的字符可写作**[^a]**，任何非*a*、*b* 及*c* 的字符则写作：

```
[^abc]
```

(5) 可选的子表达式

有关串的最后一个常见的情况是在特定的串中包括既可能出现又可能不出现的可选部分。例如，数字前既可有一个诸如**+**或**-**的先行符也可以没有。这可用解来表示，同在正则定义中是一样的：

```
natural= [0-9]+
signedNatural= natural| + natural| - natural
```

但这会很快变得麻烦起来，现在引入问号元字符*r*?来表示由*r* 匹配的串是可选的（或显示*r* 的0个或1个拷贝）。因此上面那个先行符号的例子可写成：

```
natural= [0-9]+
signedNatural= (+|-)?natural
```

2.2.3 程序设计语言记号的正则表达式

在众多不同的程序设计语言中，程序设计语言记号可分为若干个相当标准的有限种类。第1类是保留字的，有时称为关键字（keyword），它们是语言中具有特殊含意的字母表字符的固定串。例如：在Pascal、C和Ada语言中的**if**、**while**和**do**。另一个范围由特殊符号组成，它包括算术运算符、赋值和等式。它们可以是一单个字符，如 **=**，也可是多个字符如：**=**或**++**。第3种由标识符（identifier）组成，它们通常被定义为以字母开头的字母和数字序列。最后一种包括了文字（literal）或常量（constant），如数字常量42和3.14159，如串文字“hello, world, ”，及字符“ *a* ”和“ *b* ”。在这里仅讨论一下它们的典型正则表达式以及与记号识别相关的问题，本章后面还会更详细地谈到实际识别问题。

1) **数** 数可以仅是数字（自然数）、十进制数、或带有指数的数（由*e* 或*E* 表示）的序列。例如：2.71E-2表示数.0271。可用正则定义将这些数表示如下：

```
nat = [0-9]+
signedNat= (+|-)?nat
number= signedNat("." nat ) ? (E signedNat)?
```

此处，在引号中用了一个十进制的点来强调它应直接匹配且不可被解释为一个元字符。

2) **保留字和标识符** 正则表达式中最简单的就是保留字了，它们由字符的固定序列表示。如果要将所有的保留字收集到一个定义中，就可写成：

```
reserved= if | while | do | ...
```

相反地，标识符是不固定的字符串。通常，标识符必须由一个字母开头且只包含字母和数字。可用以下的正则定义表示：

```
letter= [ a - zA - Z ]
digit = [ 0 - 9 ]
identifier= letter(letter| digit)*
```

3) **注释** 注释在扫描过程中一般是被忽略的㊀。然而扫描程序必须识别注释并舍弃它们。因此尽管扫描程序可能没有清晰的常量记号（可将其称为“伪记号 pseudotoken ”），仍需要给注释编写出正则表达式。注释可有若干个不同的格式。通常，它们可以是前后为分隔符的自由

㊀ 它们有时可包括编译目录。

格式，例如：

```
{ this is a Pascal comment }
/* this is a C comment */
```

或由一个或多个特殊字符开头并直到该行的结尾，如在

```
; this is a Scheme comment
-- this is an Ada comment
```

中。

为有单个字符的分隔符的注释(如 Pascal 注释)编写正则表达式并不难，且为那些从行的特殊字符到行尾编写正则表达式也不难。例如 Pascal 注释可写作：

```
{(~})*}
```

其中，用**~}**表示“非**}**”，并假设字符**}**作为元字符没有意义（在Lex中的表示与之不同，本章后面将会提到）。类似地，一个Ada注释可被正则表达式

```
--(~newline)*
```

匹配。其中，假设newline匹配一行的末尾（在许多系统中可写作**\n**），“**-**”字符作为元字符没有意义，该行的结尾并未包括在注释本身之中（2.6节将谈到如何在Lex中书写它）。

为同C注释一样，其中的分隔符如多于一个字符时，则编写正则表达式就要困难许多。例如串集合*ba*. . .（没有*ab*）. . . *ab*（用*ba*. . . *ab* 代替C的分隔符/*...*/，这是因为星号，有时还有前斜杠要求特殊处理的元字符）。不能简单地写作：

```
ba(~(ab))*ab
```

由于“非”运算通常限制为只能是单个字符而不能使用字符串。可尝试用**~a**、**~b**和**~(a|b)**为**~(ab)**写出一个定义来，但这却没有多大用处。其中的一个解是：

```
b*(a*~(a|b)b*)*a*
```

然而这很难读取（且难证明是正确的）。因此，C注释的正则表达式是如此之难以至于在实际中几乎从未编写过。实际上，这种情况在真正的扫描程序中经常是通过特殊办法解决的，本章后面将会提到它。

最后，在识别注释时会遇到的另一个复杂的问题是：在一些程序设计语言中，注释可被嵌套。例如Modula-2允许格式注释：

```
(* this is (*a Modula-2 *) comment *)
```

在这种嵌套注释中，注释分隔符必须成对出现，故以下注释在 Modula-2中是不正确的：

```
(* this is ( * illegal in Modula-2 *)
```

注释的嵌套要求扫描程序计算分隔符的数量，但我们又注意到在例 2.3（2.2.1节）中，正则表达式不能表示计数操作。实际上，一个简单的计算器配置可作为这个问题的特殊解（参见练习）。

4) **二义性、空白格和先行** 在程序设计语言记号使用正则表达式的描述中，有一些串经常可被不同的正则表达式匹配。例如：诸如 **if**和**while**的串可以既是标识符又可以是关键字。类似地，串 < >可解释为表示两个记号（“小于号”和“大于号”）或一单个符号（“不等于”）。程序设计语言定义必须规定出应遵守哪个解释，但正则表达式本身却无法做到它。相反地，语言定义必须给出无二义性规则（ disambiguating rules），由其回答每一种情况下的含义。

下面给出处理示例的两个典型规则。首先当串既可以是标识符也可以是关键字时，则通常认为它是关键字。这暗示着使用术语保留字（ reserved word），其含义只是关键字不能同时也

是标识符。其次，当串可以是单个记号也可以是若干记号的序列时，则通常解释为单个记号。这常常被称作最长子串原理（principle of longest substring）：可组成单个记号的字符的最长串在任何时候都是假设为代表下一个记号[⊖]。

在使用最长子串原理时会出现记号分隔符（token delimiter）的问题，即表示那些在某时不能代表记号的长串的字符。分隔符应是肯定为其他记号一部分的字符。例如在串**xtemp=ytemp**中，等号将标识符**xtemp**分开，这是因为=不能作为标识符的部分出现。通常也认为空格、新行和制表位是记号分隔符：因此**while x..**就解释为包含了两个记号——保留字**while**和带有名字**x**的标识符，这是因为一个空格将两个字符串分开。在这种情况下，定义空白格伪记号非常有用0它与注释伪记号相类似，但注释伪记号仅仅是在扫描程序内部区分其他记号。实际上，注释本身也经常作为分隔符，因此例如C代码片段：

```
do / ** / if
```

表示的就是两个保留字**do**和**if**，而不是标识符**doif**。

程序设计语言中的空白格伪记号的典型定义是：

```
whitespace= (newline| blank | tab | comment)+
```

其中，等号右边的标识符代表恰当的字符或串。请注意：空白格通常不是作为记号分隔符，而是被忽略掉。指定这个行为的语言叫作自由格式语言（free format）。除自由格式语言之外还可以是一些诸如FORTRAN的固定格式语言，以及各种使用缩排格式的语言，例如越位规则（offside rule）（参见“注意与参考”一节）。自由格式语言的扫描程序必须在检查任意记号分隔功能之后舍弃掉空白格。

分隔符结束记号串，但它们并不是记号本身的一部分。因此，扫描程序必须处理先行（lookahead）问题：当它遇到一个分隔符时，它必须作出安排分隔符不会从输入的其他部分中删除，方法是将分隔符返回到输入串（“备份”）或在将字符从输入中删除之前先行。在大多数情况下，只有单个字符才需要这样做（“单个字符先行”）。例如在串**xtemp=ytemp**中，当遇到**=**时，就可找到标识符**xtemp**的结尾，且**=**必须保留在输入中，这是因为它表示要识别下一个记号。还应注意，在识别记号时可能不需要使用先行。例如，等号可能是以**=**开头的唯一字符，此时无需考虑下一个字符就可立即识别出它了。

有时语言可能要求不仅仅是单个字符先行，且扫描程序必须准备好可以备份任意多个字符。在这种情况下，输入字符的缓冲和为追踪而标出位置就给扫描程序的设计带来了问题（本章后面将会讨论到其中的一些问题）。

FORTRAN是一个并不遵守上面所谈的诸多原则的典型语言。它是固定格式语言，它的空白格在翻译开始之前已由预处理器删除了。因此，下面的FORTRAN行

```
I F ( X 2 . EQ. 0 ) THE N
```

在编译器中就是

```
IF(X2.EQ.0)THEN
```

所以空白格再也不是分隔符了。FORTRAN中也没有保留字，故所有的关键字也可以是标识符，输入每行中字符串的位置对于确定将要识别的记号十分重要。例如，下面代码行在FORTRAN中是完全正确的：

```
IF(IF.EQ.0 )THENTHEN=1.0
```

⊖ 有时这也称作“最大咀嚼”定理。

第1个**IF**和**THEN**都是关键字，而第 2个**IF**和**THEN**则是表示变量的标识符。这样的结果是FORTRAN的扫描程序必须能够追踪代码行中的任意位置。例如：

```
DO99I=1,10
```

它引起循环体为第99行代码的循环。在Pascal中，则表示为**for i := 1 to 10**。另一方面，若将逗号改为句号：

```
DO99I=1.10
```

就将代码的含义完全改变了：它将值 1.1赋给了名字为**DO99I**的变量。因此，扫描程序只有到它接触到逗号（句号）时才能得出起始的 **DO**。在这种情况下，它可能只得追踪到行的开头并且由此开始。

2.3 有穷自动机

有穷自动机，或有穷状态的机器，是描述（或“机器”）特定类型算法的数学方法。特别地，有穷自动机可用作描述在输入串中识别模式的过程，因此也能用作构造扫描程序。当然有穷自动机与正则表达式之间有着很密切的关系，在下一节中大家将会看到如何从正则表达式中构造有穷自动机。但在学习有穷自动机之前，先看一个说明的示例。

通过下面的正则表达式可在程序设计语言中给出标识符模式的一般定义（假设已定义了**letter**和**digit**）：

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

它代表以一个字母开头且其后为任意字母和 / 或数字序列的串。识别这样的一个串的过程可表示为图2-1。在此图中，标明数字 1和2的圆圈表示的是状态（state），它们表示其中记录已被发现的模式的数量在识别过程中的位置。带有箭头的线代表由记录一个状态向另一个状态的转换（transition），该转换依赖于所标字符的匹配。在较简单的图示中，状态 1是初始状态（start state）或识别过程开始的状态。

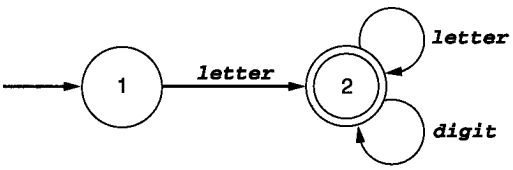


图2-1 标识符的有穷自动机

按照惯例，初始状态表示为一个“不来自任何地方”且指向它的未作标识的箭头线。状态 2代表有一单个字母已被匹配的点（表示为从状态 1到状态2的转换，且其上标有**letter**）。一旦位于状态2中，就可看到任何数量的字母和 / 或数字，它们的匹配又使我们回到了状态2。代表识别过程结束的状态称作接受状态（accepting state），当位于其中时就可说明成功了，在图中它表示为在状态的边界画出双线。它们可能不只一个。在上面的例图中，状态 2就是一个接受状态，它表示在看到一个字母之后，随后的任何字母和数字序列（也包括根本没有）都表示一个正规的标识符。

将真实字符串识别为标识符的过程现在可通过列出在识别过程中所用到的状态和转换的序列来表示。例如，将**xtemp**识别为标识符的过程可表示为：

$$\longrightarrow 1 \xrightarrow{x} 2 \xrightarrow{t} 2 \xrightarrow{e} 2 \xrightarrow{m} 2 \xrightarrow{p} 2$$

在此图中，用在每一步中匹配的字母标出了每一个转换。

2.3.1 确定性有穷自动机的定义

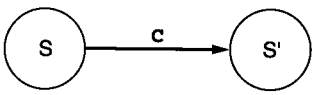
因为上面所示的例图很方便地展示出算法过程，所以它对于有穷自动机的描述很有用处。

但是偶尔还需使用有穷自动机的更正式的描述，现在就给出一个数学定义。但绝大数情况并不需要这么抽象，且在大多数示例中也只使用示意图。有穷自动机还有其他的描述，尤其是表格，表格对于将算法转换成运行代码很有用途。在需要它的时候我们将会谈到它。

另外读者还需注意：我们一直在讨论的是确定性的（deterministic）有穷自动机，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。非确定性的有穷自动机是由它产生的。本节稍后将谈到它。

定义：DFA（确定性有穷自动机）M由字母表$\sum$、状态集合S、转换函数$T: S\times\sum\to S$、初始状态s_0 S以及接受状态的集合A S组成。由M接受的且写作$L(M)$被定义为字符$c_1c_2\ldots c_n$串的集合，其中每个c_i $\sum$，存在状态$s_1 = T(s_0, c_1), s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$，其中$s_n$是$A$（即一个接受状态）的一个元素。

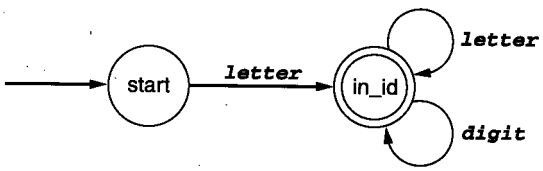
有关这个定义请注意以下几点。$S\times\sum$指的是S和$\sum$的笛卡尔积或叉积：集合对（s, c），其中s S且c $\sum$。如果有一个标为c的由状态s到状态s'的转换，则函数T记录转换：$T(s, c) = s'$。与M相应的示图片段如下：



当接受状态序列$s_1 = T(s_0, c_1), s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$存在，且$s_n$是一个接受状态时，它表示如下所示的意思：

$$\to s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-1} \xrightarrow{c_n} s_n$$

在DFA的定义和标识符示例的图示之间有许多区别。首先，在标识符图中的状态使用了数字，而定义并未用数字对状态集合作出限制。实际上可以为状态使用任何标识系统，这其中也包括了名字。例如：下图与图2-1完全一样：



在这里就称作状态**start**（因为它是初始状态）和**in_id**（因为我们发现了一个字母并识别其后的任何字母和数字后面的标识符）。这个图示中的状态集合现在变成了 **{start, in_id}**，而不是**{1,2}**了。

图示与定义的第2个区别在于转换不是用字符标出而是表示为字符集合的名字。例如，名字***letter***表示根据以下正则定义的字母表中的任意字母：

```
letter = [ a - zA - Z ]
```

因为如要为每个小写字母和每个大写字母画出总共52个单独的转换非常麻烦，所以这是定义的一个便利的扩展。本章后面还会用到这个定义的扩展。

图示与定义的第3个区别更为重要：定义将转换表示为一个函数 $T: S\times\sum\to S$。这意味着$T(s, c)$必须使每个s和c都有一个值。但在图中，只有当c是字母时，才定义T(start, c)；而也只有当c是字母或数字时，才定义T(in_id, c)。那么，所丢失的转换跑到哪里去了呢？答案是它们表示了出错——即在识别标识符时，我们不能接受除来自初始状态之外的任何字符以及

这之后的字母或数字⊖。按照惯例，这些出错转换（error transition）在图中并没有画出来而只是假设它总是存在着。如果要画出它们，则标识符的图示应如图2-2所示：

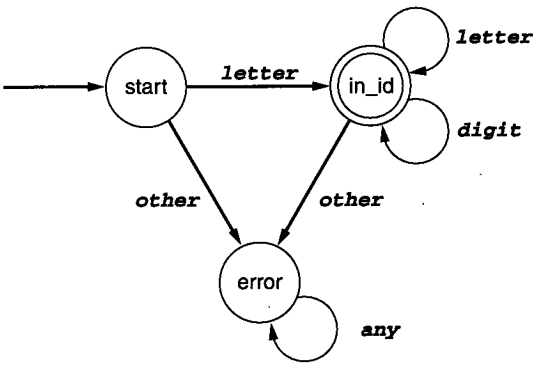


图2-2 带有出错转换的标识符的有穷自动机

在该图中，我们将新状态 error标出来了（这是因为它表示出错的发生），而且还标出了出错转换**other**。按照惯例，**other**表示并未出现在来自于它所起源的状态的任何其他转换中的任意字符，因此**other**的定义来自于初始状态为：

```
other = ~letter
```

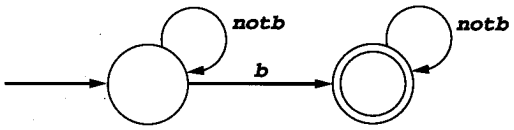
来自in_id状态的**other**的定义是：

```
other = ~(letter|digit)
```

请注意，来自出错状态的所有转换都要回到其本身（这些转换用 **any** 标出以表示在这个转换中得出的任何字符）。另外，出错状态是非接受的，因此一旦发生一个出错，则无法从出错状态中逃避开来，而且再也不能接受串了。

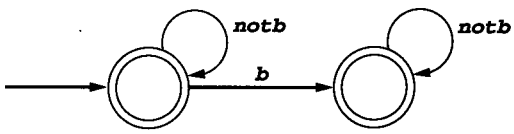
下面是DFA的一些示例，其中也有一些是在上一节中提到过的。

例2.6 串中仅有一个*b* 的集合被下示的DFA接受：



请注意，在这里并未标出状态。当无需用名字来指出状态时就忽略标签。

例2.7 包含最多一个*b* 的串的集合被下示的DFA接受：



请注意这个DFA是如何修改前例中的DFA，它将初始状态变成另一个的接受状态。

例2.8 前一节给出了科学表示法中数字常量的正则表达式，如下所示：

```
nat = [0-9]+
signedNat= (+|-)?nat
number = signedNat("." nat)? (E signedNat)?
```

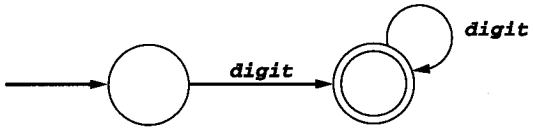
我们想为由这些定义匹配的串写出DFA，但是先如下重写它会更为有用：

```
digit = [0-9]
nat = digit+
```

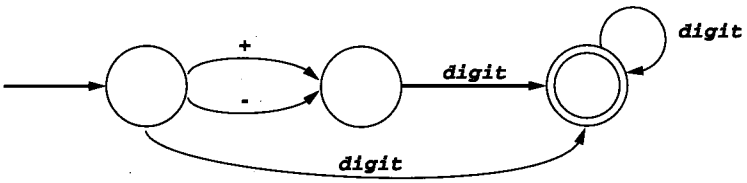
⊖ 在实际情况中，这些非文字数字的字符意味着根本就没有标识符 (如果是在初始状态中)或遇到了一个结束标识符的识别的分隔符(如果是在接受状态中)。本节后面将介绍如何处理这些情况。

```
signedNat= (+|-)?nat
number= signedNat("." nat)? (E signedNat)?
```

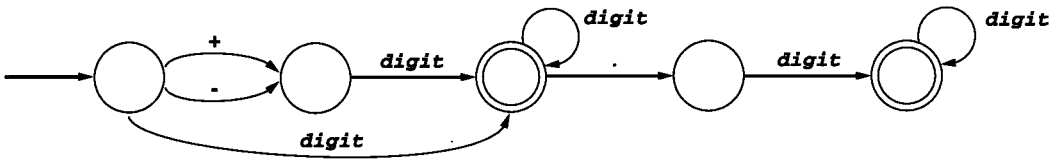
如下为**nat**写出一个DFA是非常简单（请记住**a+ = aa***对任意的**a**均成立）的：



由于可选标记，**signedNat**要略难一些，但是可注意到**signedNat**是以一个数字或一个标记与数字开头，并写作以下的DFA：



在它上面添加可选的小数部分也很简单，如下所示：



请注意，它有两个接受状态，它们表示小数部分是可选的。

最后需要添加可选的指数部分。要做到它，就要指出指数部分必须是以字母 *E*开头，且只能发生在前面的某个接受状态之后，图2-3是最终的图。

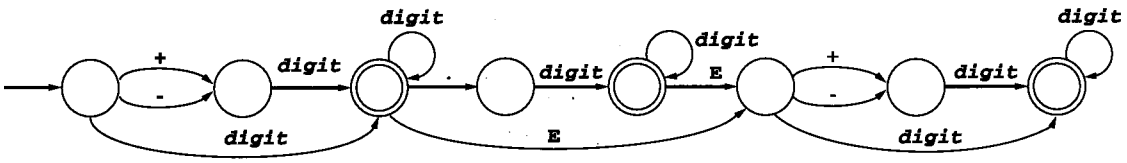
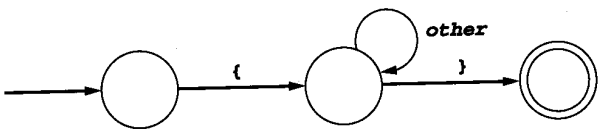


图2-3 浮点数的有穷自动机

例2.9 使用DFA可描述非嵌套注释。例如，前后有花括号的注释可由以下的 DFA接受：



在这种情况下，**other**意味着除了右边花括号外的所有字符。这个 DFA与第2.2.4节中所写的正则表达式**{(~})*}**相对应。

我们注意到在2.2.4中，为被两个字符的序列分隔开的注释编写一个正则表达式很难， C注释的格式**/*...(no*/s).../** 就是这样的。编写接受这个注释的DFA比编写它的正则表达式实际上要简单许多，图2-4中的DFA就是这样的C注释。

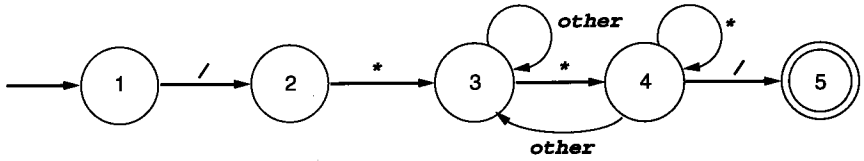


图2-4 有C风格注释的有穷自动机

在该图中，由状态3到其自身的**other**转换表示除“ * ”之外的所有字符，但由状态4到状态3的**other**转换则表示除“ * ”和“ / ”之外的所有字符。为了简便起见，还给状态编了号，但仍能为状态赋予更多有意义的名字，如下所示（在括号中带有其相应的编号）：start (1)、entering_comment (2)、in_comment (3)、exiting_comment (4）和finish（5)。

2.3.2 先行、回溯和非确定性自动机

作为根据模式接受字符串的表示算法的一种方法，我们已经学习了 DFA。正如同读者可能早已猜到的一样，模式的正则表达式与根据模式接受串的 DFA之间有很密切的关系，下一节我们将探讨这个关系。但首先需要更仔细地学习 DFA表示的精确算法，这是因为希望最终能将这些算法变成扫描程序的代码。

我们早已注意到 DFA的图表并不能表示出 DFA所需的所有东西而仅仅是给出其运算的要点。实际上，我们发现数学定义意味着 DFA必须使每个状态和字符都具有一个转换，而且这些导致出错的转换通常是不在 DFA的图表中。但即使是数学定义也不能描述出 DFA算法行为的所有方面。例如在出错时，它并不指出错误是什么。在程序将要到达接受状态时或甚至是在转换中匹配字符时，它也不指出该行为。

进行转换时发生的典型动作是：将字符从输入串中移到属于单个记号（记号串值或记号词）累积字符的字符串中。在达到某个接受状态时的典型的动作则是将刚被识别的记号及相关属性返回。遇到出错状态的典型动作则是在输入中备份（回溯）或生成错误记号。

在关于标识符最早的示例中有许多这里将要描述的行为，所以我们再次回到图 2-4中。由于某些原因，该图中的DFA并没有如希望的那样来自扫描程序的动作。首先，出错状态根本就不是一个真正的错误，而是表示标识符将不被识别（如来自于初始状态）或是已看到的一个分隔符，且现在应该接受并生成标识符记号。我们暂时假设（实际这是正确的操作）有其他的转换可表示来自初始状态的非字母转换。接着指出可看到来自状态 in_id的分隔符，以及应被生成的一个标识符记号，如图2-5所示。

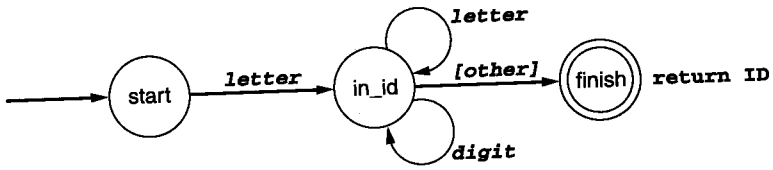
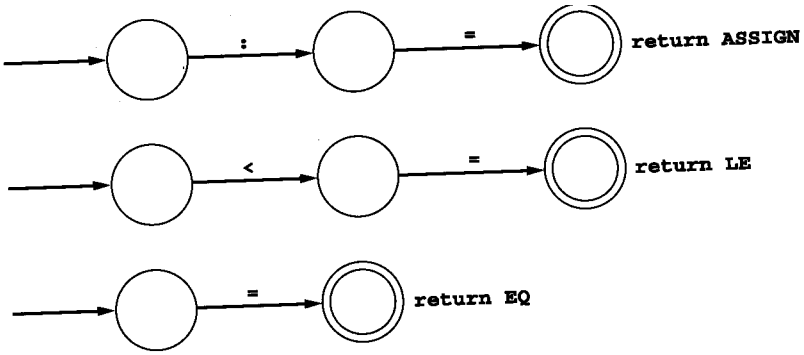


图2-5 有分隔符和返回值的标识符的有穷自动机

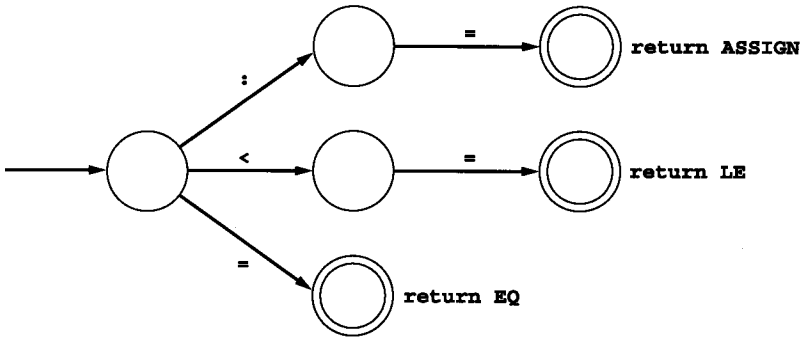
在该图中，**other**转换前后都带有方括号，它表示了应先行考虑分隔字符，也就是：应先将其返回到输入串并且不能丢掉。此外在该图中，出错状态已变成接受状态，且没有离开接受状态的转换。因为扫描程序应一次识别一个记号并在每一个记号识别之后再一次从它的初始状态开始，所以这正是所需要的。

这个新的图示还表述了在2.2.4节中谈到的最长子串原理：DFA将一直（在状态**in_id**中）匹配字母和数字直到找到一个分隔符。与在读取标识符串时允许DFA在任何地方接受的旧图相反，我们确实不希望发生某些事情。

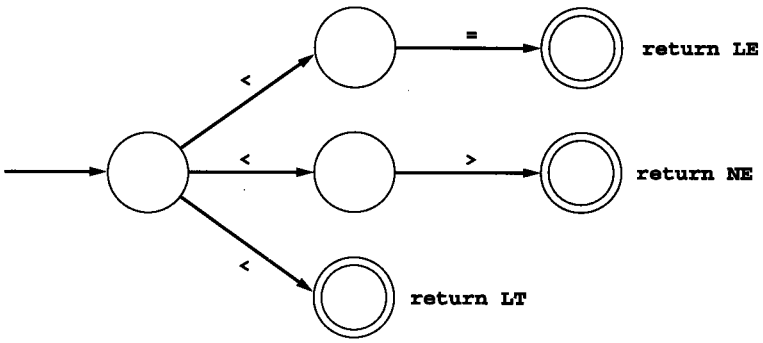
现在将注意力转向如何在一开始就到达初始状态的问题上。典型的程序设计语言中都有许多记号，且每一个记号都能被其自己的DFA识别出来。如果这每一个记号都以不同的字符开头，则只需通过将其所有的初始状态统一到一个单独的初始状态上，就能很便利地将它们放在一起了。例如，考虑串：=、<=和=给出的记号。其中每一个都是一个固定串，它们的DFA可写作：



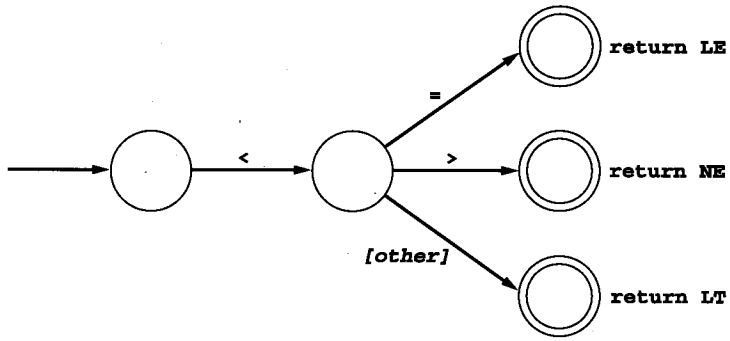
因为每一个记号都是以不同的字符开始的，故只需通过标出它们的初始状态就可得出以下的DFA：



但是假设有若干个以相同字符开头的记号，例如<、<=和<>，就不能简单地将其表示为如下的图表了。这是因为它不是DFA（给出一个状态和字符，则通常肯定会有一个指向单个的新状态的唯一转换）：



相反地，我们必须做出安排，以便在每一个状态中都有一个唯一的转换。例如下图：

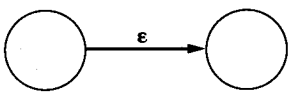


在理论上是应该能够将所有的记号都合并为具有这种风格的一个巨大的 DFA，但是它非常复杂，在使用一种非系统性的方法时尤为如此。

解决这个问题的一个方法是将有穷自动机的定义扩展到包括了对某一特定字符一个状态存在有多个转换的情况，并同时为系统地将这些新生成的有穷自动机转换成 **NFA**开发一个算法。这里会讲解到这些生成的自动机，但有关转换算法的内容要在下一节才能提到。

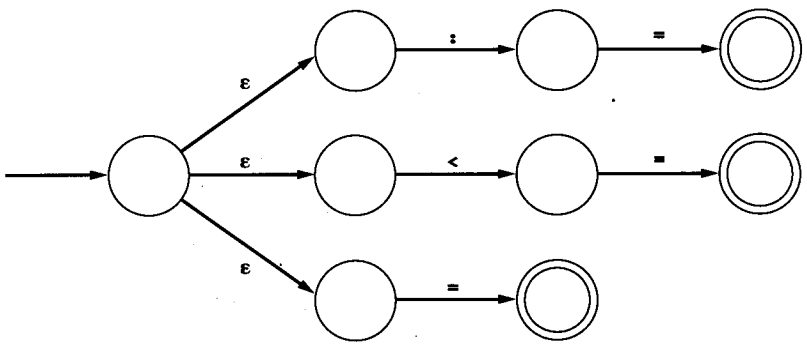
新的有穷自动机称作非确定性有穷自动机（ nondeterministic finite automaton）或简称为 NFA。在对它下定义之前，还需要为在扫描程序中应用有穷自动机再给出一个概括的讲法： ε-转换的概念。

ε-转换（ε-transition）是无需考虑输入串（且无需消耗任何字符）就有可能发生的转换。它可看作是一个空串的“ 匹配 ”，空串在前面已讲过是写作 ε 的。ε-转换在图中的表示就好像 ε 是一个真正的字符：



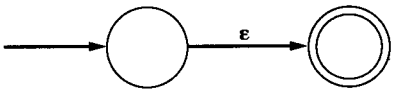
这不应同与在输入中的字符 ε 的匹配相混淆：如果字母表包括了这样一个字符，则必须与使用ε作为表示ε-转换的元字符相区别。

ε-转换与直觉有些相矛盾，这是因为它们可以“ 同时 ”发生，换言之，就是无需先行和改变到输入串，但它们在两方面有用：首先，它们可以不用合并状态就表述另一个选择。例如：记号:=、<=和=的选择可表述为：为每一个记号合并自动机，如下所示：

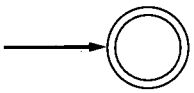


这对于使最初的自动机保持完整并只添加一个新的初始状态来链接它们很有好处。 ε-转换

的第2个好处在于它们可以清晰地描述出空串的匹配：

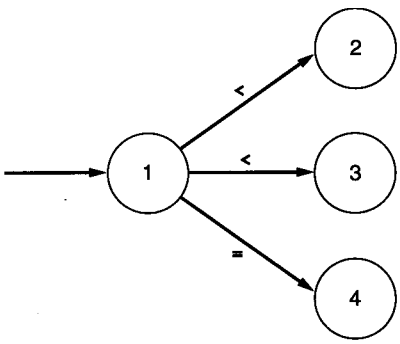


当然，这与下面的DFA等同，该DFA表示接受可在无任何字符匹配时发生：



但是具有前面清晰的表示法也是有用的。

现在来讲述非确定性自动机的定义。它与DFA的定义很相似，但有一点不同：根据上面所讨论的，需要将字母表∑扩展到包括了ε。将原来写作∑的地方（这假设ε最初并不属于∑）改写成∑ {ε}（即∑和ε的并集）。此外还需要扩展T（转换函数）的定义，这样每一个字符都可以导致多个状态，通过令T的值是状态的一个集合而不是一个单独的状态就可以做到它。例如下示的图表：



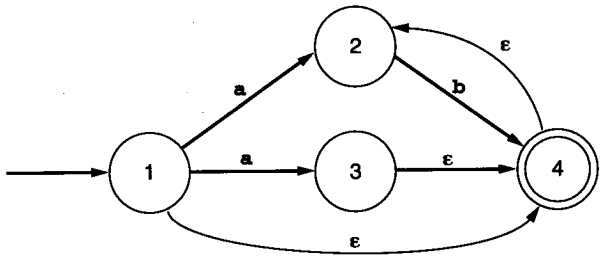
使T (1,<) = {2,3}，即：在输入字符<上，由状态1可到状态2或状态3，且T成为一个将状态/符号对映射到状态集合的函数。因此， T的范围是状态的S集合（S的所有子集的集合）的幂集（power set），写作$\wp(S)$（S的草写的p的集合）。下面给出定义。

定义：NFA（nondeterministic finite automaton）M由字母表∑、状态的集合S、转换函数T: $S\times$(∑ {ε})→$\wp(S)$、S的初始状态s_0，以及S的接受状态A的集合组成。由M接受的语言写作L (M)，它被定义为字符$c_1c_2\ldots c_n$，其中每一个c_i都属于∑ {ε}，且存在关系：s_1在T (s_0, c_1) 中、s_2在T (s_1, c_2) 中、...、s_n在T (s_{n-1}, c_n) 中，s_n是A中的元素。

有关这个定义还需注意以下几点。在$c_1c_2\ldots c_n$中的任何c_i都有可能是ε，而且真正被接受的串是删除了ε的串$c_1c_2\ldots c_n$（这是因为s和ε的联合就是s本身）。因此，串$c_1c_2\ldots c_n$中真正的字符数可能会少于n个。另外状态序列$s_1\ldots s_n$是从状态集合T (s_0, c_1) , . . . , T (s_{n-1}, c_n）选出的，这个选择并不总是唯一确定的。实际上这就是为什么称这些自动机是非确定性的原因：接受特定串的转换序列并不由状态和下一个输入字符在每一步中确定下来。实际上，任意个 ε 都可在任一点上被引入到串中，并与NFA中ε- 转换的数量相对应。因此， NFA并不表示算法，但是却可通过一个在每个非确定性选择中回溯的算法来模拟它，本节下面会谈到这一点。

首先看一些NFA的示例。

例2.10 考虑下面的NFA图。

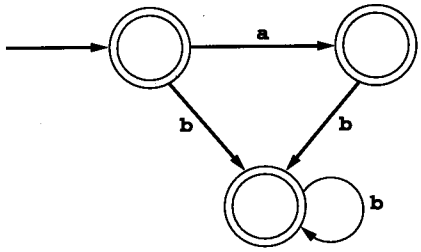


下面两个转换序列都可接受串**abb**：

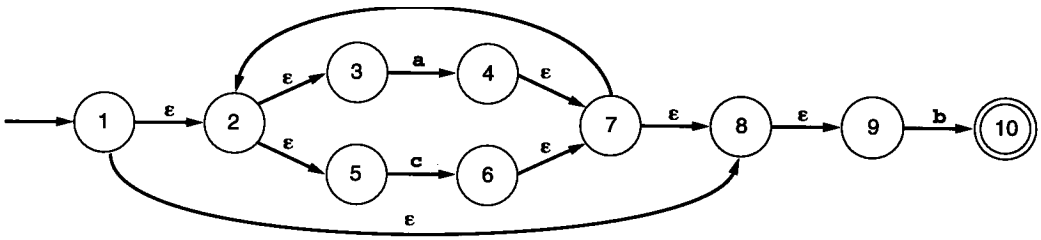
$$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

$$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

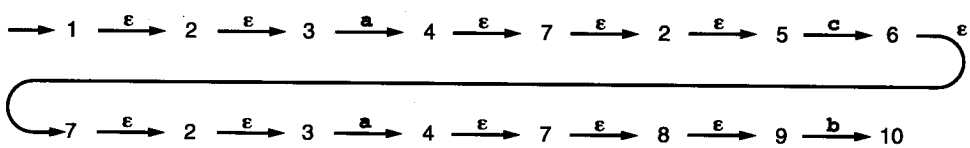
实际上，*a* 上的由状态1向状态2的转换与*b* 上的由状态2向状态4的转换均允许机器接受串*ab*，接着再使用由状态4向状态2的转换，所有的串与正则表达式**ab+**匹配。类似地，*a* 上的由状态1向状态3的转换，和ε上的由状态3向状态4的转换也允许接受与**ab***匹配的所有串。最后，由状态1向状态4的ε- 转换可接受与 **b*** 匹配的所有串。因此，这个 NFA接受与正则表达式**ab+|ab*|b***相同的语言。能够生成相同的语言的更为简单的正则表达式是 **(a|ε)b***。下面的DFA也接受这个语言：



例2.11 考虑下面的NFA：



它通过下面的转换接受串*acab*：

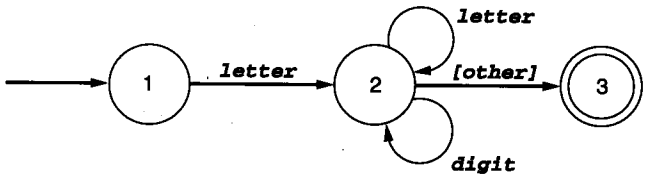


不难看出，这个NFA接受的语言实际上与由正则表达式**(a|c)*b**生成的语言相同。

2.3.3 用代码实现有穷自动机

将DFA或NFA翻译成代码有若干种方法，本节将会讲到它们。但并不是所有的方法对编译器的扫描程序都有用，本章的最后两节将更详细地讲到与扫描程序相关的编码问题。

请再想一下最开始那个接受由一个字母及一个字母和 /或数字序列组成的标识符的 DFA的示例，以及当它位于包含了先行和最长子串原理的修改格式（参见图 2-5）：



模拟这个DFA最早且最简单的方法是在下面的格式中编写代码：

```
{ 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;
```

这个代码使用代码中的位置（嵌套于测试中）来隐含状态，这与由注释所指出的一样。如果没有太多的状态（要求有许多嵌套层）且 DFA中的循环较小，那么就合适了。类似这样的代码已被用来编写小型扫描程序了。但这个方法有两个缺点：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办法将每个 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。现在来考虑一下在例2.9（图2-4）中接受注释的DFA，它可用以下的格式来实现：

```
{ 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 ；
      end while;
      advance the input ； { state 4 }
```

```
    while the next input character is "*" do
     advance the input;
    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;
```

我们注意到这样做复杂性大大增加了，并且还需要利用布尔变量 *done*来处理涉及到状态3和状态4的循环。

一个较之好得多的实现方法是：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。例如，前一个标识符的DFA可翻译为程序清单2-1的代码模式。

程序清单2-1　利用状态变量和嵌套的case测试实现标识符DFA

```
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 ;
```

请注意这个代码是如何直接反映DFA的：转换与对*state*变量新赋的状态相对应，并提前输入（除了由状态2到状态3的“非消耗”转换）。

现在C注释的DFA（图2-4）可被翻译成程序清单2-2中更可读的代码模式。除了这个结构之外，还可使外部case基于在输入字符之上，并使内部case基于在当前状态之上（参见练习）。

程序清单2-2　图2-4中DFA的实现

```
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 ;
```

在前面的示例中，DFA已正好被“硬连”进代码之中，此外还有可能将DFA表示为数据结构并写成实现来自该数据结构的行为的“类”代码。转换表（transition table），或二维数组就是符合这个目标的简单数据结构，它由表示转换函数*T*值的状态和输入字符来索引：

	字母表*C*中的字符
状态 *S*	代表转换*T* (*s*, *c*) 的状态

例如：标识符的DFA可表示为如下的转换表：

状态 \ 输入	字母	数字	其他
1	2		
2	2	2	3
3			

在这个表格中，空表项表示未在DFA图中显示的转换（即：它们表示到错误状态或其他过程的转换）。另外还假设列出的第1个状态是初始状态。但是，这个表格尚未指出哪些状态正在接受以及哪些转换不消耗它们的输入。这个信息可被保存在与表示表格相同的数据结构中，或是保存在另外的数据结构中。如果将这些信息添加到上面的转换表中（另用一列来指出接受状态并用括号指出“未消耗输入”的转换），就会得到下面这个表格：

状态 \ 输入	字母	数字	其他	接受
1	2			不
2	2	2	[3]	不
3				是

又如：下面是C注释的DFA表格（前述的第2个例子）：

状态 \ 输入	/	*	其他	接受
1	2			不
2		3		不
3	3	4	3	不
4	5	4	3	不
5				是

现在若给定了恰当的数据结构和表项，就可以在一个将会实现任何 DFA的格式中编写代码了。下面的代码图解假设了转换被保存在一个转换数组 *T*中，而*T*由状态和输入字符索引；先行输入的转换（即：那些在表格中未被括号标出的）是由布尔数组 *Advance*给出，它们也由状态和输入字符索引；而由布尔数组*Accept*给出的接受状态则由状态索引。下面就是代码图解：

```
state := 1;
ch : = next input character;
while notAccept[state] and not error (state) do
  newstate := T [state,ch];
  if Advance [state,ch] then ch := next input char;
  state := newstate;
end while;
if Accept [state] then accept;
```

类似于刚刚讨论过的算法方法被称作表驱动（ table driven），这是因为它们利用表格来引导算法的过程。表驱动方法有若干优点：代码的长度缩短了，相同的代码可以解决许多不同的问题，代码也较易改变（维护）了。但也有一些缺点：表格会变得非常大，使得程序要求使用的空间也变得非常大。实际上，我们刚描述过的数组中的许多空间都是浪费了的。因此，尽管表压缩经常会多耗费时间，但表驱动方法经常仍要依赖于诸如稀疏数组表示法的压缩方法，这是因为扫描程序的效率必须很高，因此尽管可能会在诸如 Lex的扫描程序生成器程序上用到它们，也是仍很少用到这些方法。在这里也就不再提它们了。

最后注意到可用与DFA相似的方法来实现NFA，但有一点除外——因为NFA是非确定性的，所以必须要尝试转换潜在的许多不同序列。因此模拟 NFA的程序必须储存还未尝试过的转换并回溯失败的转换。除了是由输入串引导搜索之外，这与在指示图中试图找到路径的算法相类似。由于此时进行回溯的算法有可能效率不高，而扫描程序对效率的要求又必须尽可能地高，所以也就不再谈这个算法了。相反地， NFA的模拟问题可以通过使用下一节将谈到的“ 将 NFA转换成DFA ”的方法解决，在这一节中将还会简要地再谈到 NFA的模拟问题。

2.4 从正则表达式到DFA

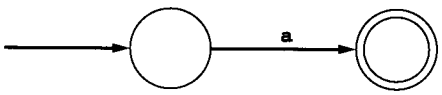
在本节中，我们将学到将正则表达式翻译成DFA的算法。由于也存在着将DFA翻译成正则表达式的算法，所以这两种概念是等同的。然而因为正则表达式的简洁性，它们趋向于将DFA当作记号来描述，而这样扫描程序的生成就通常是从正则表达式开始，并通过DFA的构造以达到最终的扫描程序。正是因为这一点，我们只是将兴趣放在完成该等同推导的算法之上。

将正则表达式翻译成DFA的最简单算法是通过中间构造，在它之中，正则表达式派生出一个NFA，接着就用该NFA构造一个同等的DFA。现在有一些算法可将正则表达式直译为DFA，但是它们很复杂，且对中间构造也有些影响。因此我们只关心两个算法：一个是将正则表达式翻译成NFA，另一个是将NFA翻译成DFA。再与将DFA翻译成前一节中描述的程序的一个算法合并，则构造一个扫描程序的自动过程可分为3步，如下所示：

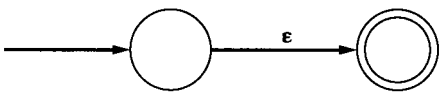
2.4.1 从正则表达式到NFA

下面将要谈到的结构是Thompson结构（Thompson construction），它以其发明者命名。Thompson结构利用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。因此该结构是归纳的，而且它依照了正则表达式定义的结构：为每个基本正则表达式展示一个NFA，接着再显示如何通过连接子表达式的NFA（假设这些是已经构造好的）得到每个正则表达式运算。

1) **基本正则表达式** 基本正则表达式格式**a**、ε或ϕ，其中**a**表示字母表中单个字符的匹配，ε表示空串的匹配，而ϕ则表示根本不是串的匹配。与正则表达式**a**等同的NFA（即在其语言中准确接受）的是：

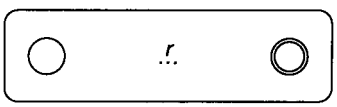


类似地，与ε等同的NFA是：



正则表达式ϕ的情况（它在实际的编译器中是不可能发生）将留在练习中。

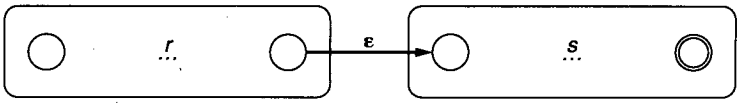
2) **并置** 我们希望构造一个与正则表达式*rs*等同的NFA，其中*r*和*s*都是正则表达式。假设已构造好了与*r*和*s*等同的NFA，并用NFA对应*r*且与*s*类似来写出它：



在该图中，圆角矩形的左边圆圈表示初始状态，右边的双圆表示接受状态，中间的3个点表示NFA中未显示出的状态和转换。这个图假设与*r*相应的NFA中只有一个接受状态。如果构造的每个NFA都有一个接受状态，那么这个假设就要调整一下。对于基本正则表达式的NFA，这是

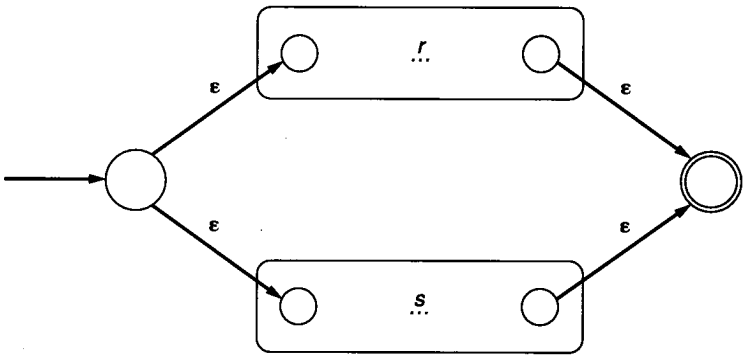
正确的；且对于下面每个结构，它也是正确的。

可将与*rs* 对应的NFA构造如下：



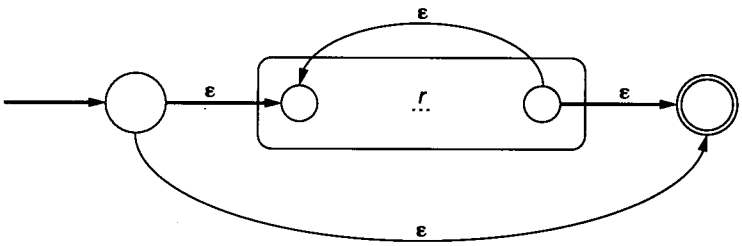
我们已将*r* 机的接受状态与*s* 机的接受状态通过一个ε-转换连接在一起。新机器将*r* 机的初始状态作为它的初始状态，并将 *s* 机的接受状态作为它的接受状态。很明显，该机可接受 *L* (*rs*) = *L* (*r*) *L* (*s*) 的关系，它也对应于正则表达式*rs*。

3) **在各选项中选择** 我们希望在与前面相同的假设下构造一个与 *r* | *s* 相对应的NFA。如下进行：



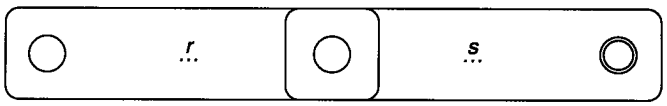
我们添加了一个新的初始状态和一个新的接受状态，并利用 ε-转换将它们连接在一起。很明显，该机接受语言*L* (*r* | *s*) = *L* (*r*) *L* (*s*)。

4) **重复** 我们需要构造与**r***相对应的机器，现假设已有一台与*r* 相对应的机器。那么就如下进行：



这里又添加了两个新的状态：一个初始状态和一个接受状态。该机中的重复由从 *r* 机的接受状态到它的初始状态的新的ε-转换提供。它允许*r* 机来回多次移动。为了保证也能接受空串（即*r* 的重复为零），就必须也画出一个由新初始状态到新接受状态的ε-转换。

这样就完成了Thompson 结构的描述。请读者注意这个构造并不唯一。特别是当将正则表达式运算翻译成NFA时，也可能有另一个结构。例如在表述并置 *rs* 时，就可以省略在*r* 机和*s* 机之间的ε-转换，相反却是将*r* 机的接受状态等同于*s* 机的初始状态，如下：

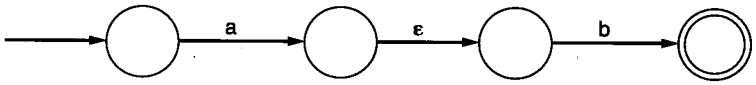


(但是这种简化却要取决于：在别的结构中，接受状态没有来自其他状态的转换，参见练习)。在其他情况下也会有别的简化。之所以像上面那样来表述转换是因为机器构造遵循的原则也非常简单。首先，每个状态具有最多两个来自它的状态，而且如果有两个转换，就必须都是 ε-转换。其次，不能在构造时删除状态，而且除了来自接受状态的其他转换之外，转换也不可更改。这些属性就将过程简化了。

用以下几个示例来结束对Thompson结构的讨论。

例2.12 根据Thompson 结构将正则表达式**ab|a** 翻译为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：



现在再为**a** 构造另一个机器复件，并利用它们组成**ab|a** 完整的NFA，如图2-6所示：

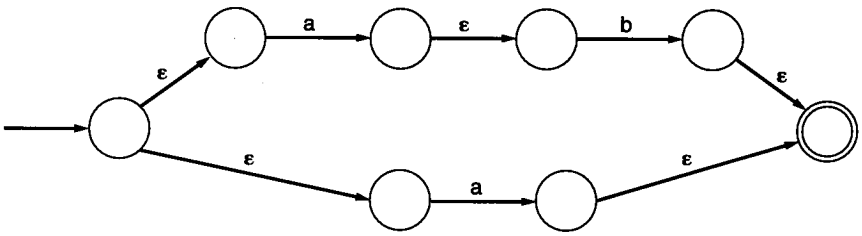
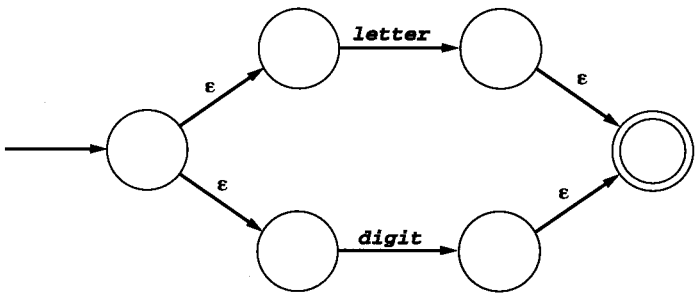


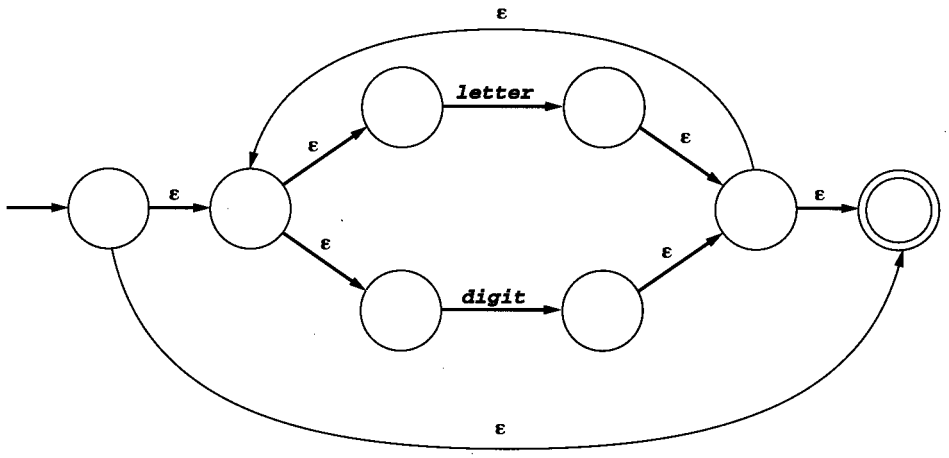
图2-6 利用Thompson结构完成的正则表达式**ab|a** 的NFA

例2.13 利用Thompson 结构完成正则表达式***letter(letter|digit)****的NFA。同前例一样，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：



现在为重复***(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，如图2-7所示：

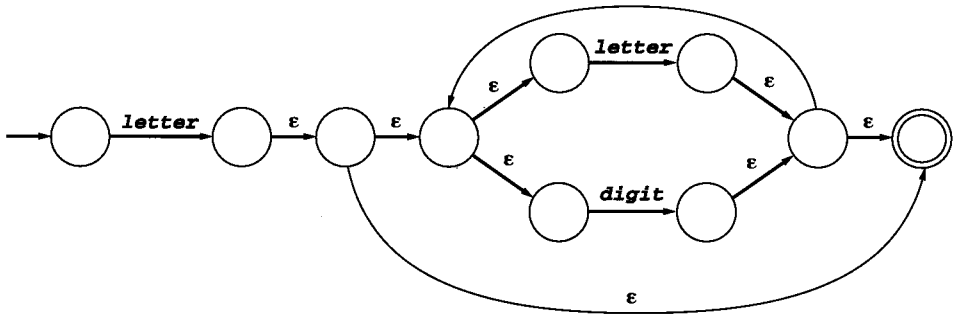


图2-7 利用Thompson结构得到正则表达式 **letter(letter|digit)*** 的NFA

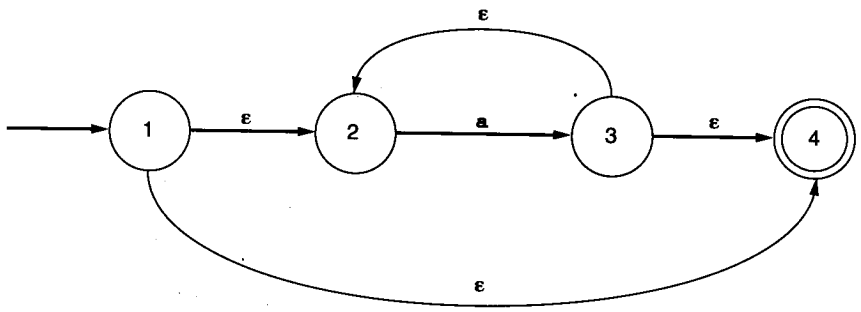
作为最后一个示例：我们注意到例 2.11（2.3.2节）与正则表达式**(a|c)*b**在Thompson结构下完全对应。

2.4.2 从NFA到DFA

若给定了一个任意的 NFA，现在来描述构造等价的 DFA（即：可准确接受相同串的 DFA）的算法。为了做到它，则需要可从单个输入字符的某个状态中去除 ε- 转换和多重转换的一些方法。消除ε- 转换涉及到了ε- 闭包的构造。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。消除在单个输入字符上的多重转换涉及跟踪可由匹配单个字符而达到的状态的集合。这两个过程都要求考虑状态的集合而不是单个状态，因此，当看到构建的 DFA 如同它的状态一样，也有原始NFA的状态集合时，就不会感到意外了。所以就将这个算法称作子集构造（subset construction）。我们首先较详细地讨论一下ε- 闭包，然后再描述子集合构造。

1) **状态集合的ε- 闭包** 我们将单个状态s 的ε- 闭包定义为可由一系列的零个或多个ε- 转换能达到的状态集合，并将这个集合写作 $\bar{s}$。该定义的更为数学化的语言将放在练习中，现在直接谈一个示例；但请大家应注意到一个状态的ε- 闭包总是包含着该状态本身。

例2.14 考虑在Thompson结构下，下面与正则表达式**a***相对应的NFA：



在这个NFA，有 $\overline{1}$ = {1,2,4}，$\overline{2}$ = {2}，$\overline{3}$ = {2,3,4}，$\overline{4}$ = {4}。

现在将状态的一个集合的ε-闭包定义为每个单独状态的ε- 闭包的和。若S是状态集，则用符号表示就是：

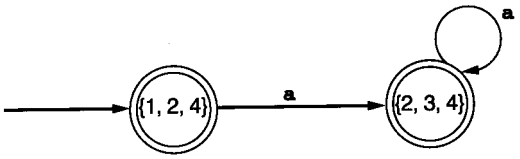
$$\overline{S} = \bigcup_{s \text{ in } S} \overline{s}$$

例如在例2.14的NFA中，$\overline{\{1,3\}} = \overline{1}$　$\overline{3}$ = {1,2,4}　{2,3,4} = {1,2,3,4}。

2) **子集构造**　现在来描述从一个给定的NFA——M来构造DFA的算法，并将其称作 $\overline{M}$。首先计算M初始状态的ε- 闭包，它就变成 $\overline{M}$ 的初始状态。对于这个集合以及随后的每个集合，计算a字符之上的转换如下所示：假设有状态的S集和字母表中的字符a，计算集合S'_a = {t | 对于S中的一些s，在a上有从s 到t 的转换}。接着计算 $\overline{S'_a}$，它是S'_a 的闭包。这就定义了子集构造中的一个新状态和一个新的转换 $S \xrightarrow{a} \overline{S'_a}$，继续这个过程直到不再产生新的状态或转换。当接受这些构造的状态时，按照包含了M的接受状态的方式作出记号。这就是DFA的 $\overline{M}$，它并不包括ε-转换，这是因为每个状态都被构造成了一个ε- 闭包。它至多包括了一个来自字母a 上的状态的转换，这又是因为每个新状态都由从单个字符 a 上的一个状态的转换构造为来自M的所有可接受到的状态。

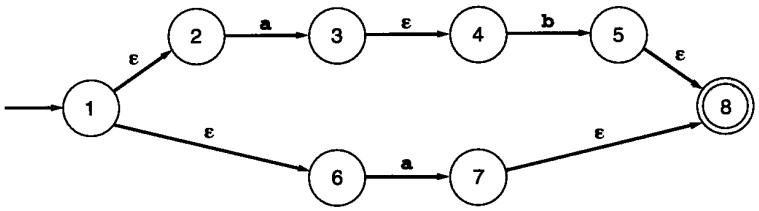
下面用若干示例来说明子集构造。

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 $\overline{1}$ = {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：

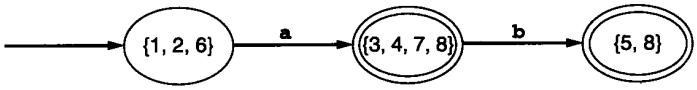


（一旦构造完成，则如果愿意就可将子集术语丢置一旁了）。

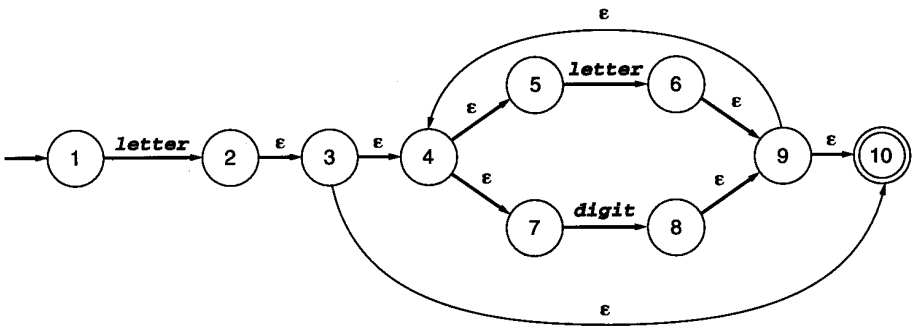
例2.16　考虑向图2-6中的NFA增添状态数：



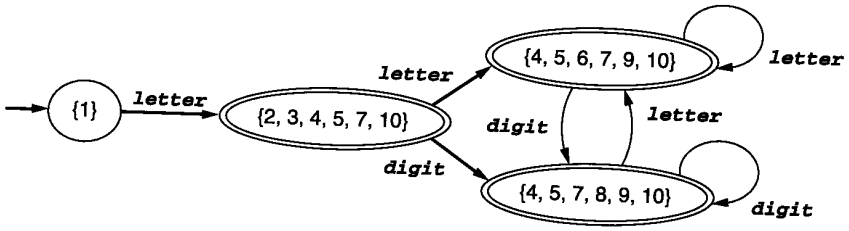
DFA子集结构与它的初始状态 $\overline{\{1\}} = \{1,2,6\}$ 相同。此时在 a 上有从状态2到状态3的转换，且有从状态6到状态7的转换，因此，$\overline{\{1,2,6\}}_a = \overline{\{3,7\}} = \{3,4,7,8\}$，且有 $\{1,2,6\} \xrightarrow{a} \{3,4,7,8\}$。由于再也没有来自1、2或6状态的其他字符的转换，则只需看{3,4,7,8}就行了。此时在 b 上有从状态4到状态5的转换，且 $\overline{\{3,4,7,8\}}_b = \overline{\{5\}} = \{5,8\}$，且有转换 $\{3,4,7,8\} \xrightarrow{b} \{5,8\}$。除此之外就再也没有别的转换了。因此所产生的以下DFA子集构造与前一个NFA等同：



例2.17　考虑图2-7中的NFA（正则表达式**letter(letter|digit)***的Thompson 结构）：



子集构造过程如下：它的初始状态是 $\overline{\{1\}} = \{1\}$，在***letter***上有到 $\overline{\{2\}} = \{2,3,4,5,7,10\}$ 的转换。在***letter***上还有一个从这个状态到 $\overline{\{6\}} = \{4,5,6,7,9,10\}$ 的转换以及在***digit***上有到 $\overline{\{8\}} = \{4,5,7,8,9,10\}$ 的转换。最后，所有这些状态都有在***letter***和***digit***上的转换，或是到其自身或是到另一个。完整的DFA在下图中给出：



2.4.3　利用子集构造模拟NFA

上一节简要地讨论了编写模拟NFA的程序的可能性，这是一个要求处理机器的非确定性或非算法本质的问题。模拟NFA的一种方法是使用子集构造，但并非是构造与 DFA相关的所有状态，而是在由下一个输入字符指出的每个点上只构造一个状态。因此，这样只构造了在给出的

输入串上被取用的DFA的路径中真正发生的状态集合。这样做的好处在于有可能就不再需要构造整个DFA了，它的缺点在于如果路径中包含了循环，则有可能会多次构造某个状态。

例如：在例2.16中，若使输入串只由单个字符a 组成，则构建初始状态{1,2,6}和第2个状态{3,4,7,8}，之后再移至这个状态并匹配a。由于随后再没有b 了，因此也就无需生成状态{5,8}了。

另一方面，在例2.17中，给定了输入串$r2d2$，就有下面的状态和转换序列：

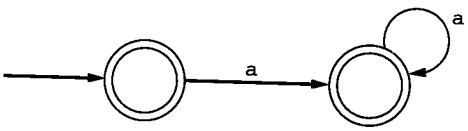
$$\{1\}\xrightarrow{r}\{2,3,4,5,7,10\}\xrightarrow{2}\{4,5,7,8,9,10\}$$

$$\xrightarrow{d}\{4,5,6,7,9,10\}\xrightarrow{2}\{4,5,7,8,9,10\}$$

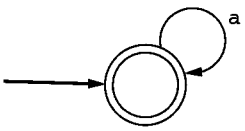
如果在转换发生时构建这些状态，则也构造了 DFA的所有状态，且构造两次状态{4,5,7,8,9,10}。因此，这个过程比第一次构造整个 DFA的效率要低一些。正是由于这个原因，在扫描程序中并不做NFA的模拟。但在编辑器和搜索程序中却保留了模式匹配的选项，在编辑器和搜索程序中，正则表达式可由用户动态地提供。

2.4.4 将DFA中的状态数最小化

我们上面所描述的由正则表达式利用算法派生出 DFA的过程有一个缺点：生成的DFA可能比需要的要复杂得多。例如在例2.15中派生出的DFA：



对于正则表达式**a***，下面的**DFA**同样是可以的：



因为在扫描程序中，效率是很重要的，如果可能的话，在某种意义上构造的 DFA应最小。实际上，自动机理论中有一个很重要的结果，即：对于任何给定的 DFA，都有一个含有最少量状态的等价的DFA，而且这个最小状态的 DFA是唯一的（重命名的状态除外）。人们有可能从任何指定的DFA中直接得到这个最小状态的DFA，本节将简要地描述这个算法，但我们不证明它确实构造了最小状态的等价的DFA（对于读者而言，通过阅读算法证明一下并不难）。

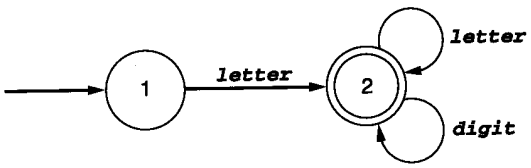
该算法通过创建统一到单个状态的状态集来进行。它以最乐观的假设开始：创建两个集合，其中之一包含了所有的接受状态，而另一个则由所有的非接受状态组成。假设这样来划分原始DFA的状态，还要考虑字母表中每个a 上的转换。如果所有的接受状态在a 上都有到接受状态的转换，那么这样就定义了一个由新接受状态（所有旧接受状态的集合）到其自身的a- 转换。类似地，如果所有的接受状态在a 上都有到非接受状态的转换，那么这也定义了由新接受状态到新的非接受状态（所有旧的非接受状态的集合）的a- 转换。另一方面，如果接受状态s 和t 在a上有转换且位于不同的集合，则这组状态不能定义任何a- 转换，此时就称作a 区分（distinguish）了状态s 和t。在这种情况下必须根据考虑中状态集合（即所有接受状态的集合）的a- 转换的位置而将它们分隔开。当然状态的每个其他集合都有类似的语句，而且一旦要考虑字母表中的所有字符时，就必须移到它们的位置之上。当然如果还要分隔别的集合，就得返回到开头并重复这一过程。我们继续将原始DFA的各部分状态集中到集合里，并一直持续到所有集合只有一个

元素（在这种情况下，就显示原始DFA为最小）或一直是到再没有集合可以分隔了。

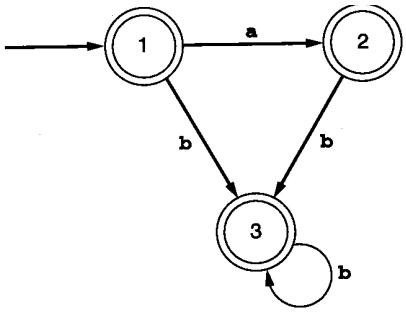
如要准确地完成上面所描述的过程，还必须掌握非接受的错误状态的错误转换。也就是：如果有两个接受状态 s 和t，其中s 有一个到其他接受状态的 a- 转换，而t 却根本没有a- 转换（即：错误转换），那么a 就将s 和t 区分开来了。类似地，如果非接受状态s 有到某个接受状态的a-转换，而另一个非接受状态t 却没有a- 转换，那么在这种情况下，a 也将s 和t 区分开来。

下面用几个示例来总结一下状态最小化的讨论。

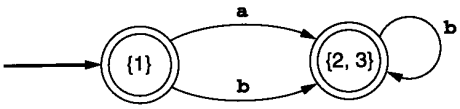
例2.18　考虑在前例中构造的DFA，其与正则表达式***letter(letter|digit)****相对应，它有4个状态：1个初始状态和3个接受状态，这3个接受状态在***letter***和***digit***上都有到其他接受状态的转换，且除此之外再也没有其他（非错的）转换了。因此，任何字符也不能区分开这3个接受状态，且最小化算法将 3个接受状态合并为一个接受状态，而剩下了下面的最小状态DFA（即在2.3节开头所看到的）：



例2.19　考虑下面的DFA，在例2.1（2.3.2节）中已指出它与正则表达式**(a|ε)b***相对应：



在这种情况下，所有的状态（除了错误状态之外）都在接受。现在考虑字符 b。每个接受状态都有到其他接受状态的b- 转换，因此b 不能区分任何状态。另一方面，状态1存在有到一个接受状态的a- 转换，但状态2和状态3却没有a- 转换（或说成是：在a 上到错误的非接受状态的错误转换，更合适一些）。所以，a可将状态1与状态2和3区分开来，我们必须将状态重新分配为集合{1}和{2,3}。然后再重复一遍。集合 {1}不能再分隔了，我们也就不再考虑它了。状态2和状态3不能由a 或b 区分开来。因此，就得到了最小状态的DFA：



2.5　TINY扫描程序的实现

现在开发扫描程序的真正代码以阐明本章前面所学到的概念。在这里将用到 TINY语言，它曾在第1章（1.7节）中被提到过，接着再讲一些由该扫描程序引出的实际问题。

2.5.1 为样本语言TINY实现一个扫描程序

第1章只是非常简要地介绍了一下TINY语言。现在的任务是完整地指出TINY的词法结构，也就是：定义记号和它们的特性。TINY的记号和记号类都列在表2-1中。

TINY的记号分为3个典型类型：保留字、特殊符号和“其他”记号。保留字一共有 8个，它们的含义类似（尽管直到很后面才需知道它们的语义）。特殊符号共有10种：分别是4种基本的整数运算符号、2种比较符号（等号和小于），以及括号、分号和赋值符号。除了赋值符号是两个字符的长度之外，其余均为一个字符。

表2-1 TINY语言的记号

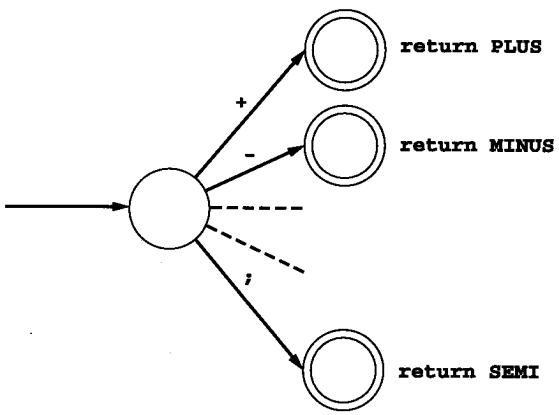
保 留 字	特 殊 符 号	其 他
if	+	**数**
then	-	（1个或更多的数字）
else	*	
end	/	
repeat	=	
until	<	**标识符**
read	(	(1个或更多的字母)
write	)	
	;	
	:=	

其他记号就是数了，它们是一个或多个数字以及标识符的序列，而标识符又是（为了简便）一个或多个字母的序列。

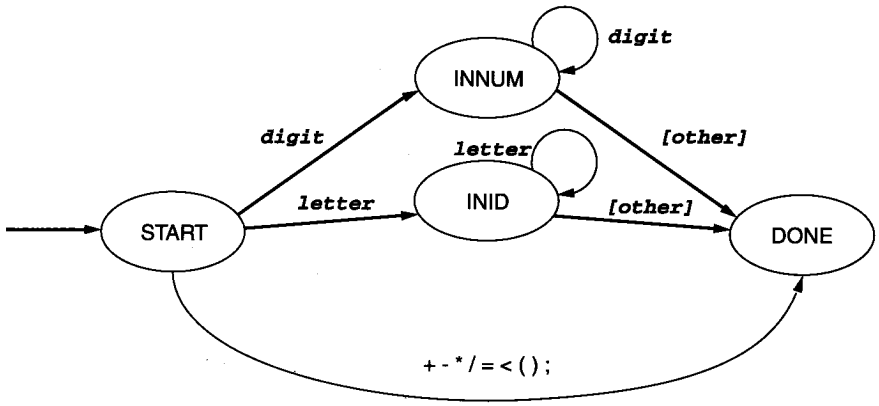
除了记号之外，TINY还要遵循以下的词法惯例：注释应放在花括号{...}中，且不可嵌套；代码应是自由格式；空白格由空格、制表位和新行组成；最长子串原则后须接识别记号。

在为该语言设计扫描程序时，可以从正则表达式开始并根据前一节中的算法来开发 NFA和DFA。实际上，前面已经给出了数、标识符和注释的正则表达式（TINY具有更为简单的版本）。其他记号的正则表达式都是固定串，因而均不重要。由于扫描程序的 DFA记号十分简单，所以无需按照这个例程就可直接开发这个DFA了。我们将按以下步骤进行。

首先要注意到除了赋值符号之外，其他所有的特殊符号都只有一个字符，这些符号的 DFA如下：



在该图中，不同的接受状态是由扫描程序返回的记号区分开来。如果在这个将要返回的记号（代码中的一个变量）中使用其他指示器，则所有接受状态都可集中为一个状态，称之为**DONE**。若将这个二状态的DFA与接受数和标识符的DFA合并在一起，就可得到下面的DFA：



请注意，利用方括号指出了不可被消耗的先行字符。

现在需要在这个DFA中添加注释、空白格和赋值。一个简单的从初始状态到其本身的循环要消耗空白格。注释要求一个额外的状态，它由花括号左边达到并在花括号右边返回到它。赋值也需要中间状态，它由分号上的初始状态达到。如果后面紧跟有一个等号，那么就会生成一个赋值记号。反之就不消耗下一个字符，且生成一个错误记号。实际上，未列在特殊符号中的所有单个字符既不是空白格或注释，也不是数字或字母，它们应被作为错误而接受，我们将它们与单个字符符号混合在一起。图2-8是为扫描程序给出的最后一个DFA。

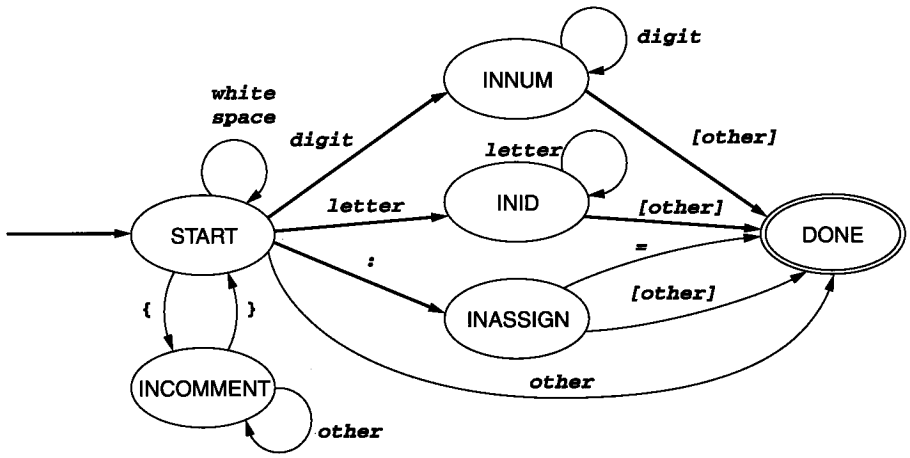


图2-8 TINY扫描程序的DFA

在上面的讨论或图2-8中的DFA都未包括保留字。这是因为根据DFA的观点，而认为保留字与标识符相同，以后再在接受后的保留字表格中寻找标识符是最简单的。当然，最长子串原则保证了扫描程序唯一需要改变的动作是被返回的记号。因而，仅在识别了标识符之后才考虑保留字。

现在再来讨论实现这个DFA的代码，它已被放在了**scan.h**文件和**scan.c**文件之中（参

见附录B)。其中最主要的过程是**getToken**（第674到第793行），它消耗输入字符并根据图2-8中的DFA返回下一个被识别的记号。这个实现利用了在2.3.3节中曾提到过的双重嵌套情况分析，以及一个有关状态的大型情况列表，在大列表中的是基于当前输入字符的单独列表。记号本身被定义成**globals.h**（第174行到第186行）中的枚举类型，它包括在表2-1中列出的所有记号以及内务记号**EOF**（当达到文件的末尾时）和**ERROR**（当遇到错误字符时）。扫描程序的状态也被定义为一个枚举类型，但它是位于扫描程序之中（第612行到第614行）。

扫描程序还需总地计算出每个记号的特性（如果有的话），并有时会采取其他动作（例如将标识符插入到符号表中）。在TINY扫描程序中，所要计算的唯一特性是词法或是被识别的记号的串值，它位于变量**tokenString**之中。这个变量同**getToken**一并是提供给编译器其他部分的唯一的两个服务，它们的定义已被收集在头文件 **scan.h**（第550行到第571行）。请读者注意声明了**tokenString**的长度固定为41，因此那个标识符也就不能超过40个字符（加上结尾的空字符）。后面还会提到这个限制。

扫描程序使用了3个全程变量：文件变量**source**和**listing**，在**globals.h**中声明且在**main.c**中被分配和初始化的整型变量**lineno**。

由**getToken**过程完成的额外的簿记如下所述：表**reservedWords**（第649行到第656行）和过程**reservedLookup**（第658行到第666行）完成位于由**getToken**的主要循环识别的标识符之后的保留字的查找，**currentToken**的值也随之改变。标志变量**save**被用作指示是否将一个字符增加到**tokenString**之上；由于需要包括空白格、注释和非消耗的先行，所以这些都是必要的。

到扫描程序的字符输入由**getNextChar**函数（第627行到第642行）提供，该函数将一个256-字符缓冲区内部的**lineBuf**中的字符取到扫描程序中。如果已经耗尽了这个缓冲区，且假设每一次都获取了一个新的源代码行（以及增加的 **lineno**），那么**getNextChar**就利用标准的C过程**fgets**从**source**文件更新该缓冲区。虽然这个假设允许了更简单的代码，但却不能正确地处理行的字数超过 255个字符的 TINY程序。在练习中，我们再探讨在这种情况下的**getNextChar**的行为（以及它更进一步的行为）。

最后，TINY中的数与标识符的识别要求从**INNUM**和**INID**到最终状态的转换都应是非消耗的（参见图2-8）。可以通过提供一个**ungetNextChar**过程（第644行到第647行）在输入缓冲区中反填一个字符来完成这一任务，但对于源行很长的程序而言，这也不是很好，练习将提到其他的方法。

作为TINY扫描程序行为的解释，读者可考虑一下程序清单2-3中TINY程序**sample.tny**（在第1章中已作为一个示例给出了）。程序清单2-4假设将这个程序作为输入，那么当**TraceScan**和**EchoSource**都是集合时，它列出了扫描程序的输出。

本节后面将详细讨论由这个扫描程序的实现所引出的一些问题。

程序清单2-3　TINY语言中的样本程序

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
```

```
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

程序清单2-4　当程序清单2-3中的TINY程序作为输入时，扫描程序的输出

```
TINY COMPILATION: sample.tny
    1: { Sample program
    2:   in TINY language -
    3:   computes factorial
    4: }
    5: read x; { input an integer }
        5: reserved word: read
        5: ID, name= x
        5: ;
    6: if 0 < x then { don't compute if x <= 0 }
        6: reserved word: if
        6: NUM, val= 0
        6: <
        6: ID, name= x
        6: reserved word: then
    7:   fact := 1;
        7: ID, name= fact
        7: :=
        7: NUM, val= 1
        7: ;
    8:   repeat
        8: reserved word: repeat
    9:     fact := fact * x;
        9: ID, name= fact
        9: :=
        9: ID, name= fact
        9: *
        9: ID, name= x
        9: ;
   10:     x := x - 1
       10: ID, name= x
       10: :=
       10: ID, name= x
       10: -
       10: NUM, val= 1
   11:   until x = 0;
       11: reserved word: until
       11: ID, name= x
       11: =
       11: NUM, val= 0
       11: ;
   12:   write fact { output factorial of x }
       12: reserved word: write
       12: ID, name= fact
   13: end
       13: reserved word: end
       14: EOF
```

2.5.2 保留字与标识符

TINY对保留字的识别是通过首先将它们看作是标识符，之后再在保留字表中查找它们来

完成的。这在扫描程序中很平常，但它却意味着扫描程序的效率须依赖于在保留字表中查找过程的效率。我们的扫描程序使用了一种非常简便的方法——线性搜索，即按顺序从开头到结尾搜索表格。这对于小型表格不成问题，例如TINY中的表格，它只有8个保留字，但对于真实语言而言，这却是不可接受的，因为它通常有30~60个保留字。这时就需要一个更快的查找，而这又要求使用更好的数据结构而不是线性列表。假若保留字列表是按字母表的顺序写出的，那么就可以使用二分搜索。另一种选择是使用杂凑表，此时我们希望利用一个冲突性很小的杂凑函数。由于保留字不会改变（至少不会很快地），所以可事先开发出这样一个杂凑函数，它们在表格中的位置对于编译器的每一步运行而言都是固定的。人们已经确定了各种语言的最小完善杂凑函数（minimal perfect hash function），也就是说能够区分出保留字且具有最小数值的函数，因此杂凑表可以不大于保留字的数目。例如，如果只有8个保留字，则最小完善杂凑函数总会生成一个0~7的值，且每个保留字也会生成不同的值（参见“注意与参考”一节）。

在处理保留字时，另一个选择是使用储存标识符的表格，即：符号表。在过程开始之前，将所有的保留字整个输入到该表中并且标上“保留”（因此不允许重新定义）。这样做的好处在于只要求一个查找表。但在TINY扫描程序中，直到扫描阶段之后才构造符号表，因此这个方法对于这种类型的设计并不合适。

2.5.3 为标识符分配空间

TINY扫描程序设计中的另一个缺点是记号串最长仅为40个字符。由于大多数的记号的大小都是固定的，所以对于它们而言这并不是问题；但是对于标识符来讲就麻烦了，这是因为程序设计语言经常要求程序中的标识符长度为任意值。更糟的是：如果为每一个标识符都分配一个40个字符长度的数组，那么就会浪费掉大多数的空间；这是因为绝大多数的标识符都很短。由于使用了实用程序函数**copyString**复制记号串，其中copyString函数动态地分配仅为所需的空间（如同将在第4章看到的一样），TINY编译器的代码就不会出现这个问题了。**TokenString**长度限制的解决办法与之类似：仅仅基于需要来分配，有可能使用**realloc**标准C函数。另一种办法是为所有的标识符分配最初的大型数组，接着再在该数组中按照自己做的方式进行存储器的分配（这是将在第7章要谈到的标准动态存储器管理计划的特殊情况）。

2.6 利用Lex自动生成扫描程序

本节将重复前一节完成的用于TINY语言的扫描程序的开发，但此次是利用Lex扫描程序生成器从作为正则表达式的TINY记号的描述中生成一个扫描程序。由于Lex存在着多个不同的版本，所以我们的讨论仅限于对于所有的或大多数版本均通用的特征。Lex最常见的版本是**flex**（Fast Lex），它是由Free Software Foundation创建的**Gnu compiler package**的一部分，可以在许多Internet站点上免费得到。

Lex是一个将包含了正则表达式的文本文件作为其输入的程序，此外还包括每一个表达式被匹配时所采取的动作。Lex生成一个包含了定义过程**yylex**的C源代码的输出文件，其中yylex是与输入文件相对应的DFA表驱动的实现，它的运算与**getToken**过程类似。接着编译通常称作**lex.yy.c**或**lexyy.c**的Lex输出文件，并将它们链接到一个主程序上以得到一个可运行的程序，这与在前一节中将**scan.c**文件与**tiny.c**文件链接相似。

下面将首先讨论用于编写正则表达式的Lex惯例和Lex输入文件的格式，之后还会谈到附录

B中给出的TINY扫描程序的Lex输入文件。

2.6.1 正则表达式的Lex约定

Lex 约定与2.2.3节中所谈到的十分相似，但它并不列出所有的 Lex元字符且不逐个地描述它们，我们将给出一个概述，之后再在一个表格中写出 Lex 约定。

Lex允许匹配单个字符或字符串，只需像前面各节中所做地一样按顺序写出字符即可。 Lex还允许通过将字符放在引号中而将元字符作为真正的字符来匹配。引号可用于并不是元字符的字符前后，但此时的引号却毫无意义。因此，在要被直接匹配的所有字符前后使用引号很有意义，而不论该字符是否为元字符。例如，可以用**if**或**"if"**来匹配一个**if**语句开始的保留字if。另一方面，如要匹配一个左括号，就必须写作**"("**，这是因为左括号是一个元字符。另一个方法是利用反斜杠元字符****，但它只有在单个元字符时才起作用：如要匹配字符序列**(***，就必须重复使用反斜杠，写作**\(***。很明显，**"(*"**更好一些。另外将反斜杠与正规字符一起使用就有了特殊意义。例如：**\n**匹配一新行，**\t**匹配一个制表位（这些都是典型的C约定，大多数这样的约定在Lex中也可行）。

Lex按通常的方法解释元字符*****、**+**、**(**、**)**和**|**。Lex 还利用问号作为元字符指示可选部分。为了说明前面所讲到的Lex表示法，可为*a* 串和*b* 串的集合写出正则表达式，其中这些串是以*aa*或*bb*开头，末尾则是一个可选的*c*：

```
(aa|bb)(a|b)*c?
```

或写作：

```
("aa"|"bb")("a"|"b")*"c"?
```

字符类的Lex 约定是将字符类写在方括号之中。例如**[abxz]**就表示*a*、*b*、*x* 或*z* 中的任意一个字符，此外还可在Lex 中将前面的正则表达式写作：

```
(aa|bb)[ab]*c?
```

在这个格式的使用中还可利用连字符表示出字符的范围。因此表达式**[0-9]**表示在Lex 中，任何一个从0~9的数字。句点也是一个表示字符集的元字符：它表示除了新行之外的任意字符。互补集合——也就是不包含某个字符的集合——也可使用这种表示法：将插入符 ^作为括号中的第1个字符，因此**[^0-9abc]**就表示不是任何数字且不是字母*a*、*b* 或*c* 中任何一个的其他任意字符。

例：为一个标有符号的数集写出正则表达式，这个集合可能包含了一个小数部分或一个以字母E开头的指数部分（在2.2.4节中，这个表达式的写法略有不同）：

```
("+"|"-")?[0-9]+("."[0-9]+)?(E("+"|"-")?[0-9]+)?
```

Lex有一个古怪的特征：在方括号（表示字符类）中，大多数的元字符都丧失了其特殊状况，且无需用引号引出。甚至如果可以首先将连字符列出来的话，则也可将其写作正则字符。因此，可将前一个数字的正则表达式**("+"|"-")**写作**[-+]**，（但不可写作**[+-]**，这是因为元字符“ - ”用于表示字符的一个范围）。又例如：**[."?]**表示了句号、引号和问号3个字符中的任一个字符（这3个字符在括号中都失去了它们的元字符含义）。但是一些字符即使是在方括号中也仍旧是元字符，因此为了得到真正的字符就必须在字符前加一个反斜杠（由于引号已失去了它们的元字符含义，所以不能用它），因此**[\^\\]**就表示了真正的字符^或\。

Lex中一个更为重要的元字符约定是用花括号指出正则表达式的名字。在前面已经提到过可以为正则表达式起名，而且只要没有递归引用，这些名字也可使用在其他的正则表达式中。

例如：将2.2.4节中的**signedNat**定义如下：

```
nat = [0-9] +
signedNat= ("+"|"-") ? nat
```

在本例和其他示例中，我们使用斜体字将名字和普通的字符序列区分开来。但是 Lex 文件是普通的文本文件，因此无需使用斜体字。相反地， Lex 却遵循将前面定义的名字放在花括号中的约定。因此，上一例在Lex 中就表示为（Lex 还在定义名字时与等号一起分配）：

```
nat [0-9] +
signedNat (+|-)? {nat}
```

请注意，在定义名字时并未出现花括号，它只在使用时出现。

表2-2是讨论过的Lex元字符约定的小结列表。Lex 中还有许多我们用不到的元字符，这里也就不讲了（参见本章末尾的“ 注意与参考 ”）。

表2-2 Lex中的元字符约定

格 式	含 义
a	字符*a*
"a"	即使*a* 是一个元字符，它仍是字符*a*
\a	当*a* 是一个元字符时，为字符*a*
a*	*a* 的零次或多次重复
a+	*a* 的一次或多次重复
a?	一个可选的*a*
a|b	*a* 或*b*
(a)	*a* 本身
[abc]	字符*a*、*b* 或*c* 中的任一个
[a-d]	字符*a*、*b*、*c* 或*d* 中的任一个
[^ab]	除了*a* 或*b* 外的任一个字符
.	除了新行之外的任一个字符
{xxx}	名字*xxx* 表示的正则表达式

2.6.2 Lex 输入文件的格式

Lex输入文件由 3个部分组成：定义（ definition）集、规则（ rule）集以及辅助程序（auxiliary routine）集或用户程序（user routine）集。这3个部分由位于新一行第 1列的双百分号分开，因此，Lex输入文件的格式如下所示：

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

为了正确理解Lex如何解释这样的输入文件，就必须记住该文件的一些部分是正则表达式信息，Lex利用这个信息指导构成它的C输出代码，而文件的另一部分则是提供给 Lex的真正的C代码，Lex会在适当的位置逐字地将它插入到输出代码中。在我们逐个解释完这 3个部分以及给出一些示例之后，将会告诉大家Lex 在这里所用的规则。

定义部分出现在第1个双百分号之前。它包括两样东西：第 1件是必须插入到应在这一部分

中分隔符“ **%{** ”和“ **%}** ”之间的任何函数外部的任意C代码（请注意这些字符的顺序）。第2件是正则表达式的名字也得在该部分定义。这个名字的定义写在另一行的第 1列，且其后（后面有一个或多个空格）是它所表示的正则表达式。

第2个部分包含着一些规则。它们由一连串带有C代码的正则表达式组成；当匹配相对应的正则表达式时，这些C代码就会被执行。

第3个部分包括着一些C代码，它们用于在第2个部分被调用且不在任何地方被定义的辅助程序。如果要将Lex输出作为独立程序来编译，则这一部分还会有一个主程序。当第 2个双百分号无需写出时，就不会出现这一部分（但总是需要写出第 1个百分号）。

下面给出一些示例来说明Lex输入文件的格式。

例2.20　以下的Lex输入指出一个给文本添加行号的扫描程序，它将其输出发送到屏幕上（如果被重定向，则是一个文件）：

```
%{
/* a Lex program that adds line numbers
   to lines of text, printing the new text
   to the standard output
*/
#include <stdio.h>
int lineno = 1;
%}
line *.\n
%%
{line} { printf ( "%5d %s", lineno++, yytext ); }
%%
main()
{ yylex(); return 0; }
```

例如，运行从这个输入文件本身的Lex中获取的程序会得到以下输出：

```
1 %{
2 /* a Lex program that adds line numbers
3    to lines of text, printing the new text
4    to the standard output
5 */
6 #include <stdio.h>
7 int lineno = 1;
8 %}
9 line .*\n
10 %%
11 { line } { printf ( "%5d %s", lineno++,yytext); }
12 %%
13 main ( )
14 { yylex( ) ; return 0; }
```

下面为这个使用了这些行号的Lex 输入文件作出解释。首先，第1行到第8行都是位于分隔符**%{**和**%}**之间，这样就使这些行可以直接插入到由Lex 产生的C代码中，而它是位于任何过程的外部。特别是从第 2行到第 5行的注释可以插入到程序开头的附近，还将从外部插入**#include**指示与第6行和第7行上的整型变量**lineno**的定义，因此**lineno**就变成了一个全程变量且在最初被赋值为1。出现在第1个**%%**之前的其他定义是名字**line**的定义，**line**被定义

为正则表达式**".*\n"**，它与零个或多个其后接有一新行的字符匹配（但不包括新行）。换而言之：由**line**定义的正则表达式与输入的每一行都匹配。在第10行的**%%**之后，第11行包括了Lex 输入文件的行为部分。此时每当匹配了一个 **line**时，都写下了一个要完成的行为（根据Lex 约定，**line**前后都用花括号以示与其作为一个名字相区别）。正则表达式之后是**action**，即每当匹配正则表达式都要被执行的 C代码。在这个示例中，该行为由包含在一个 C块的花括号中的C语句组成（请记住在名字line前后的花括号与构成下面行为中的C代码块的花括号有完全不同的作用）。这个C语句将打印行号（在一个有5个空格的范围内且右对齐）以及在它后面要增加**lineno**的串**yytext**。**yytext**的名字是Lex 赋予并由正则表达式匹配的串的内部名字，此时的正则表达式是由输入的每一行组成（包括新行）⊖ 。最后，当Lex 生成C代码结束时在第2个双百分号（第13行和第14行）之后插入C代码。在本例中，代码包括了一个调用函数 yylex 的**main**过程（**yylex**是由Lex构造的过程的名字，这个Lex 实现了与正则表达式和在输入文件的行为部分中与给出的行为相关的DFA）。

例2.21 考虑下面的Lex输入文件：

```
%{
/* a Lex program that changes all numbers
   from decimal to hexadecimal notation,
   printing a summary statistic to stderr
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit [0-9]
number {digit}+
%%
{number} { int n = atoi (yytext);
           printf ("%x", n);
           if (n > 9) count++;}
%%
main( )
{ yylex ( );
  fprintf ( stderr, "number of replacements = %d",
                    count);
  return 0 ;
}
```

它在结构上与前例类似，但**main**过程打印了在调用了**yylex**之后替换到**stderr**的次数。这个例子与前例的不同还在于它并没有匹配所有的文本。而实际上只在行为部分匹配了数字；在行为部分中，行为的C代码第1次将匹配串（**yytext**）转变成一个整型*n*，接着又将其打印为十六进制的格式**(printf("%x",...))**，最后如果这个数大于9则增加**count**（如果小于或等于9，则与在十六进制中没有分别）。因此，为串指定的唯一行为就是数字序列。 Lex还生成了一个可匹配所有非数字字符的程序，并将它传送到输出中。这是 Lex 的一个缺省行为（default action）的示例。如果字符或字符串与行为部分中的任何一个正则表达式都不匹配，则缺省地，

⊖ 本节最后将用一个表格列出这一节中所讨论过的Lex 内置名字。

Lex 将会匹配它并将它返回到输出中（Lex 还可被迫生成一个程序错误，但这里不讨论它了）。Lex 的内部定义宏**ECHO**也可特别指定缺省行为（下一个示例将会学到它）。

例2.22 考虑下面的Lex 输入文件：

```
%{
/* Selects only lines that end or
   begin with the letter 'a'.
   Deletes everything else.
*/
#include <stdio.h>
%}
ends_with_a .*a\n
begins_with_a a.*\n
%%
{ends_with_a} ECHO;
{begins_with_a} ECHO;
.*\n ;
%%
main( )
{  yylex( );  return 0;  }
```

这个Lex 输入将以字符*a* 开头或结尾的所有输入行均写到输出上，并消除其他行。行的消除是由**ECHO**规则下的规则引起的，在这个规则中，为 C行为代码编写一个分号就可为正则表达式**.*\n**指定“空”行为。

这个Lex 输入还有一个值得注意的特征：所列的规则具有二义性（ ambiguous），这是因为串可匹配多个规则。实际上，无论它是否是以*a* 开头或结尾的行的一部分，任何输入行都可与表达式**.*\n**匹配。Lex 有一个解决这种二义性的优先权系统。首先，Lex 总是匹配可能的最长子串（因此Lex 总是生成符合最长子串原则的扫描程序）。接着，如果最长子串仍与两个或更多的规则匹配，Lex 就选取在行为部分所列的第 1个规则。正是由于这个原因，上面的 Lex 输入文件就将**ECHO**行为放在第1个。如果已按下面的顺序列出行为：

```
.*\n;
{ends_with_a} ECHO;
{begins_with_a} ECHO;
```

则由Lex 生成的程序就不会再生成任何文件的输出，这是因为第 1个规则已匹配了输入的每一行了。

例2.23 在本例中，Lex 生成了将所有的大写字母转变成小写字母的程序，但这不包括 C- 风格注释中的字母（即：任何位于分隔符**/*...*/**之间的字母）：

```
%{
/*  Lex program to convert uppercase to
    lowercase except inside comments
*/
#include <stdio.h>
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
```

```
#define TRUE 1
#endif
%}
%%
[A-Z]  {putchar(tolower(yytext[0]));
       /* yytext[0] is the single
          uppercase char found */
       }
"/*"   { char c ;
         int done = FALSE;
         ECHO;
         do
         { while ((c=input())!='*')
             putchar(c);
           putchar(c);
           while((c=input())=='*')
             putchar(c);
           putchar(c);
           if (c == '/') done = TRUE;
         } while (!done);
       }
%%
void main(void )
{ yylex();}
```

这个示例显示如何编写代码以回避较难的正则表达式，并且像执行一个 Lex 行为一样直接执行一个小的DFA。读者可以回忆一下2.2.4节中用C注释的一个正则表达式极难编写，相反地，我们只为开始C注释的串编写了正则表达式即："/*"，之后还提供了搜索结束串"*/"的行为代码，同时为注释中的其他字符提供适当的行为（此时仅是返回它们而不是继续进行）。这是通过模拟例2.9中的DFA来完成的（参见图2-4）。一旦识别出串"/*"，则就是在状态3，因此代码就在这里找到了 DFA。首先做的事情是在字符中循环直到看到一个星号（与状态 3中的***other***循环相对应）为止，如下所示：

```
while ((c=input())!='*') putchar(c);
```

这里又使用了Lex 另一个内部过程**input**，该过程的使用——并不是利用**getchar**的一个直接输入——保证使用了Lex 输入缓冲区，而且还保留了这个输入串的内部结构（但请注意，我们确实使用了一个直接输出过程**putchar**，2.6.4节将谈到它）。

DFA代码的下一步与状态4相对应。再次循环直到看不到星号为止；之后如在字符前有一个前斜杠就退出；否则就返回到状态3中。

本节的最后是小结在各例中介绍到的Lex约定。

(1) 二义性的解决

Lex输出总是首先将可能的最长子串与规则相匹配。如果某个子串可与两个或更多的规则匹配，则Lex的输出将找出列在行为部分中的第1个规则。如果没有规则可与任何非空子串相匹配，则缺省行为将下一个字符复制到输出中并继续下去。

(2) C代码的插入

1) 任何写在定义部分**%{**和**%}**之间的文本将被直接复制到外置于任意过程的输出程序之中。2) 辅助过程中的任何文本都将被直接复制到Lex 代码末尾的输出程序中。3) 将任何跟在行为部

分（在第1个**%%**之后）的正则表达式之后（中间至少有一个空格）的代码插入到识别过程**yylex**的恰当位置，并在与对应的正则表达式匹配时执行它。代表一个行为的 C代码可以既是一个C语句，也可以是一个由任何说明及由位于花括号中的语句组成的复杂的C语句。

(3) 内部名字

表2-3列出了在本章中所提到过的Lex内部名字，大多数都已在前面的示例中讲过了。

表2-3 一些Lex内部名字

Lex 内部名字	含义/使用
lex.yy.c或**lexyy.c**	Lex 输出文件名
yylex	Lex 扫描例程
yytext	当前行为匹配的串
yyin	Lex 输入文件（缺省：**stdin**）
yyout	Lex 输出文件（缺省：**stdout**）
input	Lex 缓冲的输入例程
ECHO	Lex 缺省行为（将**yytext**打印到**yyout**）

上表中有一个前面未曾提到过的特征： Lex 为一些文件备有其自身的内部名字： **yyin**和**yyout**，Lex 从这些文件中获得输入并向它们发送输出。通过标准的 Lex 输入例程**input**就可自动地从文件**yyin**中得到输入。但是在前述的示例中，却回避了内部输出文件 **yyout**，而只通过**printf**和**putchar**写到标准输出中。一个允许将输出赋到任一文件中的更好的实现方法是用**fprintf(yyout,...)**和**putc(...,yyout)**取代它们。

2.6.3 使用Lex 的TINY扫描程序

附录B中有一个Lex 输入文件**tiny.l**的列表，**tiny.l**将生成TINY语言的扫描程序（2.5节已描述了TINY语言的记号，参见表2-1）。下面对这个输入文件（第3000行到第3072行）做一些说明。

首先，在定义部分中，直接插入到Lex 输出中的C代码是由3个**#include**指示(**globals.h**、**util.h**和**scan.h**）及**tokenString**特性组成的。在扫描程序和其他的 TINY编译器之间有必要提供一个界面。

定义部分更深的内容还包括了定义 TINY记号的正则表达式的名字的定义。请注意，**number**的定义利用了前面定义的名字 **digit**，而**identifier**的定义利用了前面定义的**letter**。由于新行会导致增加**lineno**，所以定义还区分了新行和其他的空白格（空格和制表位，以及第3019行和第3020行）。

Lex输入的行为部分由各种记号的列表和 return语句组成，其中 **return**语句返回在**globals.h**中定义的恰当记号。在这个Lex定义中，在标识符规则之前列出了保留字规则。假若首先列出标识符规则，Lex 的二义性解决规则就会总将保留字识别为标识符。我们还可以写出与前一节中的扫描程序中相同的代码，在这里只能识别出标识符，然后再在表中查找保留字。由于单独识别的保留字使得由Lex 生成的扫描程序代码中的表格变得很大（而且扫描程序使用的存储器也会因此变得很大），因此在真正的编译中倾向于使用它。

Lex 输入还有一个“怪僻”：即使TINY注释的正则表达式很容易书写，也必须编写识别注释的代码以确保正确地更新了**lineno**。正则表达式实际是：

```
"{"[^\}]*"}"
```

（请注意，方括号中的花括号的作用是删除右花括号的元字符含义——引号在这里不起作用）[⊖]。

我们还注意到：并未为在遇到输入文件末尾时返回**EOF**编写代码。Lex 过程**yylex**在遇到**EOF**时有一个缺省行为——它返回 0值。正是由于这个原因，在 globals.h中的TokenType定义（第179行）中首先写出记号**ENDFILE**，所以它有0值。

最后，**tiny.1**文件包括了辅助过程部分中的**getToken**过程的定义（第3056行到第3072行）。虽然这个代码包含了Lex内部代码（如**yyin**和**yyout**）的一些在主程序中能更好地直接完成的特殊初始化，它还是允许直接使用Lex生成的扫描程序，而无需改变TINY编译器中的任何其他文件。实际上在生成C扫描程序**lex.yy.c**（或**lexyy.c**）后，就可编译这个文件，并可将它与其他的TINY源文件链接以生成一个基于Lex版本的编译器了。但是这个版本的编译器却缺少了以前版本的一项服务，这是因为没有源代码回应提供的行号（参见练习 **2.35**）。

练习

2.1 为以下的字符集编写正则表达式；若没有正则表达式，则说明原因：

a. 以*a* 开头和结尾的所有小写字母串。

b. 以*a* 开头或/和结尾的所有小写字母串。

c. 第1个不为0的所有数字串。

d. 所有表示偶数的数字串。

e. 每个2均在每个9之前的所有数字串。

f. 所有的*a* 串和*b* 串，且不包含3个连续的*b*。

g. 包含单数个*a* 或/和单数个*b* 的所有*a* 串和*b* 串。

h. 包含偶数个*a* 或偶数个*b* 的所有*a* 串和*b* 串。

i. *a* 和*b* 数目相等的所有*a* 串和*b* 串。

2.2 为由以下正则表达式生成的语言写出英语描述：

a. `(a|b)*a(a|b|ε)`

b. `(A|B|...|Z)(a|b|...|z)*`

c. `(aa|b)*(a|bb)*`

d. `(0|1|...|9|A|B|C|D|E|F)+(x|X)`

2.3 **a.** 许多系统都有grep（global regular expression print）的一个版本，它是最早为Unix编写的正则表达式搜索程序[⊜]。寻找一个描述你的本地grep文档，并描述它的元符号约定。

b. 如果你的编辑器为它的串搜索接受某种正则表达式，则描述它的元符号约定。

2.4 在正则表达式的定义中，我们讲了一些运算的优先问题，但并未提到它们之间的联系。例如并未指出**a|b|c**表示的是**(a|b)|c**还是**a|(b|c)**以及与并置是否相似，为什么是这样的呢？

2.5 试证明对于任何正则表达式*r* 都有*L* (*r***) = *L* (*r**)。

2.6 在使用正则表达式描述的程序设计语言的记号中，并不需要有元符号 ϕ（空集）或ε（空串）。为什么？

⊖ Lex的一些版本有一个内部定义的变量yylineno，它可以自动更新。用这个变量代替Lineno就有可能省掉特殊代码了。

⊜ 大多数的Unix系统上实际有3个版本的grep：“ regular ” grep、egrep (extended grep)和fgrep (fast grep)。

2.7 画出与正则表达式ϕ相对应的DFA。

2.8 为练习**2.1**中**a**至**i**的每个字符集画出DFA，或说出为什么不存在DFA？

2.9 画出从C语言中接受以下4个保留字**case**、**char**、**const**和**continue**的DFA。

2.10 利用将输入字符作为外部情况测试及将状态作为内部情况测试，重写用于 C注释的DFA实现（2.3.3节）的伪代码。将你的伪代码与书中的作一比较。什么时候为代码实现DFA使用这个组织？

2.11 给NFA的状态集的闭包下一个数学定义。

2.12 **a.** 使用Thompson结构将正则表达式**(a|b)*a(a|b|**ε**)**转化成一个NFA。

b. 利用子集结构将**a**中的NFA转化成一个DFA。

2.13 **a.** 使用Thompson结构将正则表达式**(aa|b)*a(a|bb)***转化成一个NFA。

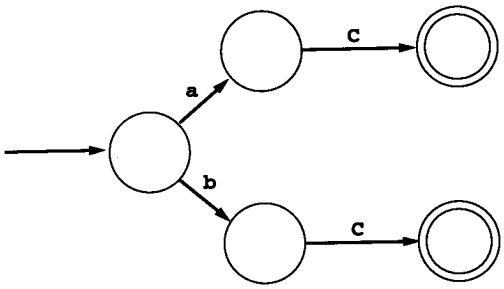
b. 利用子集结构将**a**中的NFA转化成一个DFA。

2.14 利用子集结构将例2.10（2.3.2节）中的NFA转化成一个DFA。

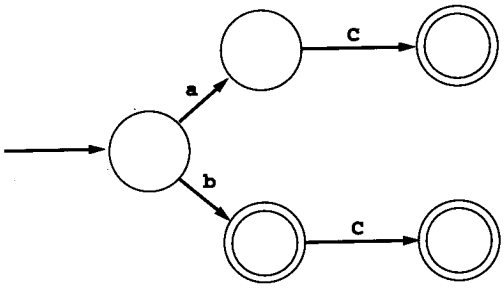
2.15 2.4.1节讲到了为了并置将Thompson结构简化，即：省略被并置的正则表达式的两个NFA之间的ε- 转换。另外还提到这种简化在结构的其他步骤中，除接受状态之外不能再有转换。请给出一个例子来说明这一点（提示：考虑用于重复的一个新 NFA结构，它省略了新初始状态和接受状态，再为*r***s**考虑NFA）。

2.16 为下面的DFA提供2.4.4节中用到的状态最小化算法：

a.



b.



2.17 Pascal注释中允许有两个不同的注释约定：花括号对**{...}**（当在TINY中）和括号-星号对**(*...*)**。试写出一个识别这两种风格的注释的DFA。

2.18 **a.** 为Lex表示法中的C注释写出一个正则表达式（提示：参见2.2.3节中的讨论）。

b. 试证明**a**中的答案是正确的。

2.19 下面的正则表达式已作为 C注释的一个 Lex定义给出（参见 Schreiner和Friedman [1985.p.25]）：

```
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
```

请说明这个表达式是不正确的（提示：考虑串`/**_/*/`）。

编程练习

2.20 编写将一个C程序中的所有注释字母均大写的程序。

2.21 编写一个程序，使之将一个C程序注释之外的所有保留字全部大写（在Kernighan和Ritchie [1988,p.192]中可找到一个C的保留字列表）。

2.22 编写一个Lex输入文件，使之可生成将C程序中所有注释的字母均大写的程序。

2.23 编写一个Lex输入文件，使之可生成将C程序注释之外的所有保留字均大写的程序。

2.24 编写一个Lex输入文件，使之生成可计算文本文件的字符、单词和行数且能报告该数字的程序。试定义一个单词是不带标点或空格的字母和/或数字的序列。标点和空白格不计算为单词。

2.25 通过能将注释中的行为与其他任何地方的行为区别开来的一个全程标志`inComment`，可缩短例2.23（2.6.2节中）的Lex代码。按上所述重写这个示例中的代码。

2.26 向例2.23中的Lex代码添加嵌套的C注释。

2.27 **a.** 重写TINY的扫描程序，使之能利用二分法搜索保留字。

b. 重写TINY的扫描程序，使之能利用杂凑表查找保留字。

2.28 通过为`tokenString`动态地分配空间来去除对于TINY扫描程序中的标识符不可多于40个字符的限制。

2.29 **a.** 当源程序行超过扫描程序中的缓冲区大小时，测试TINY扫描程序的行为，同时找出尽可能多的问题。

b. 重写TINY扫描程序以解决在**a**中发现的问题（或至少改善其行为）（此时要求重写`getNextChar`和`ungetNextChar`过程）。

2.30 如果要求在TINY扫描程序中不用`ungetNextChar`过程来完成非消费的转换，则也可使用布尔标志指出要消费的当前字符，这样就无需在输入中进行备份了。请按照这个方法重写TINY扫描程序，并将它与现存的代码相比较。

2.31 利用一个称为`nestLevel`的计数器将嵌套注释添加到TINY扫描程序中。

2.32 在TINY扫描程序中添加Ada风格的注释（Ada注释以两个连字符开头并一直到行结尾）。

2.33 向TINY的Lex扫描程序添加表格中的保留字的备份（可以像在手写的TINY扫描程序一样使用线性搜索，或使用练习**2.27**中建议的搜索方法）。

2.34 在TINY扫描程序的lex代码中添加Ada风格注释（Ada注释以两个连字符开头并一直到行结尾）。

2.35 在TINY扫描程序的Lex代码中添加源代码行回应（利用EchoSource标志），这样当设置了标志时，源代码的每一行及其行数都会打印到列表文件中（此时要求比本书所提到的更多的Lex内部知识）。

注意与参考

Hopcroft和Ullman [1979]详细讨论了正则表达式的数学理论与有穷自动机，在其中还能找到一些对该理论的历史发展的描述。特别地，这里还有一个关于对有穷自动机与正则表达式相

等价的证明（本章只谈到了这个等式的一个方面）。在这里还可找到关于pumping引理的讨论，以及对在描述格式中正则表达式的限制的推理。此外还能看到对状态最小化算法的更详细的描述，另外还包括了对这样的DFA在本质上是唯一的证明。在Aho、Hopcroft 和 Ullman[1986]中可看到从正则表达式到DFA的进一步构造（与这里所谈到的用两步来构造相反）的描述。此外这里还有将表格压缩成一个表驱动的扫描程序的方法。 Sedgewick [1990]中有使用与本章所提到的相当不同的NFA约定的Thompson结构的描述，此外在这里还能看到为了识别保留字，描述了二分搜索和杂凑法的算法（第6章还要讨论到杂凑）。Cichelli [1980]和Sager [1985]都提到了2.5.2节中的最小完善杂凑函数。一个称作**gperf**的实用程序可作为Gnu编译包的一部分来分发，它可以快速地为保留字更大的集合生成完善的杂凑函数。虽然它们并不是最小生成的，但在实际中仍很有用。Schmidt [1990]中有gperf的描述。

Lesk [1975]中有Lex扫描程序生成器的最早描述，它仍然对许多最近的版本有点作用。稍后的版本，尤其是Flex (Paxson [1990]）解决了一些很重大的问题，它可以与调整得已很好的手写的扫描程序相竞争（Jacobson[1987]）。在Schreiner和Friedman[1985]中可看到Lex的一个有用的描述，以及可完成各种格式匹配任务的简单 Lex程序的许多示例。在 Kernighan and Pike [1984]中可找到格式匹配的 grep家族（练习2.3）的简要描述，其更为深入的讨论可在 Aho [1979]中找到。Landin [1966]和Hutton [1992]讲到了在2.2.3节中所提到过的。