第一章　　词法分析

本章主要掌握下面一些内容。

1．词法分析器的作用和接口，用高级语言编写词法分析器等内容，它们与词法分析器的实现有关。（我们没有安排这方面的习题，因为大部分教材上都有这方面的例子）。

2．掌握下面涉及的一些概念，它们之间转换的技巧、方法或算法。

- 非形式描述的语言 $\leftrightarrow$ 正规式（$\leftrightarrow$表示两个方向的转换都要掌握）
- 正规式 $\rightarrow$ NFA （非确定的有限自动机）
- 非形式描述的语言 $\leftrightarrow$ NFA
- NFA $\rightarrow$ DFA （确定的有限自动机）
- DFA $\rightarrow$ 最简 DFA
- 非形式描述的语言 $\leftrightarrow$ DFA（或最简 DFA）

1.1　叙述正规式$(00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$描述的语言。

答案　该正规式所描述的语言是，所有由偶数个 0 和偶数个 1 构成的串。另外，和该正规式等价的正规式有$(00 \mid 11 \mid ((01 \mid 10)(00 \mid 11)^*(01 \mid 10)))^*$。

分析　叙述正规式描述的语言并没有一种统一的办法，只能是通过对正规式的具体分析去总结。

该正规式的一个重要特点是，它是两个字符一组来考虑的。正规式$(00 \mid 11)^*$表示的串的长度是偶数，每两个字符一组的话，不是 00 就是 11。再看正规式$(01 \mid 10)(00 \mid 11)^*(01 \mid 10)$，它表示的串由 01 或 10 开始，中间有若干组 00 或 11，最后出现 01 或 10。这样的串仍然由偶数个 0 和偶数个 1 构成，只不过第一组是 01 或 10 的话，那么一定还要有一组 01 或 10 才能保证它们的偶数性。显然，正规式$(01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*$表示的串也仍然是由偶数个 0 和偶数个 1 构成。这样，可以判断题目所给的正规式表示的语言的每个句子都是由偶数个 0 和偶数个 1 构成。

反过来还需要考虑，任何由偶数个 0 和偶数个 1 构成的串是否都在这个语言中。这实际上是问，每个这样的串，其结构是否都符合正规式$(00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$所做的刻画。我们可以这样叙述由偶数个 0 和偶数个 1 构成的串，从左向右，每两个字符一组地考察，它

1．由若干个（强调一下，可以是零个） 00 或 11 开始（这由正规式$(00 \mid 11)^*$描述）；

2．一旦出现一个 01 或 10，那么经过若干个 00 或 11 后，一定会出现一个 01 或 10。这第二个 01 或 10 的后面可能还有若干个 00 或 11，一直到串的结束，或者到再次出现 01 或 10 为止。如果串没有结束的话，就是重复出现这里所描述的结构（所以，这由（$(01 \mid 10)(00 \mid 11)^*$

$(01 \mid 10)(00 \mid 11)^*)^*$描述)。

因此正规式$(00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$描述的是偶数个 0 和偶数个 1 构成的串。

可能会提出一个问题，这样的串是否能用更简单的观点来看待，也就是该语言是否能用更简洁的正规式描述。这是可能的。我们写出这样的正规式，

$$(00 \mid 11 \mid ((01 \mid 10)(00 \mid 11)^*(01 \mid 10)))^*$$

它是基于这样的考虑，满足要求的最简单的串有三种形式（空串除外）：

1. 00
2. 11
3. $(01 \mid 10)(00 \mid 11)^*(01 \mid 10)$

它们任意多次的重复构成的串仍然满足要求。

1.2 写出语言“由偶数个 0 和奇数个 1 构成的所有 0 和 1 的串”的正规定义。

答案 $even_0_even_1 \to (00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$

$even_0_odd_1 \to 1\ even_0_even_1 \mid 0(00 \mid 11)^*(01 \mid 10)\ even_0_even_1$

分析 有了上一题的结果，这个问题应该容易解决。首先给上一题的正规式起个名字：

$$even_0_even_1 \to (00 \mid 11)^*((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$$

对于偶数个 0 和奇数个 1 构成的串，其第一个字符可能是 0 或 1。

1. 如果是 1，那么剩下的部分一定是偶数个 0 和偶数个 1。
2. 如果是 0，那么经过若干个 00 或 11，一定会出现一个 01 或 10，才能保证 0 的个数是偶数，1 的个数是奇数。若串还没有结束，剩余部分一定是偶数个 0 和偶数个 1。

这样，正确的正规定义是：

$$even_0_odd_1 \to 1\ even_0_even_1 \mid 0(00 \mid 11)^*(01 \mid 10)\ even_0_even_1$$

1.3 写出语言“所有相邻数字都不相同的非空数字串”的正规定义。

答案

$no_0\text{-}8 \to 9$

$no_0\text{-}7 \to (8 \mid no_0\text{-}8\ 8)(no_0\text{-}8\ 8)^*(no_0\text{-}8 \mid \varepsilon) \mid no_0\text{-}8$

$no_0\text{-}6 \to (7 \mid no_0\text{-}7\ 7)(no_0\text{-}7\ 7)^*(no_0\text{-}7 \mid \varepsilon) \mid no_0\text{-}7$

$no_0\text{-}5 \to (6 \mid no_0\text{-}6\ 6)(no_0\text{-}6\ 6)^*(no_0\text{-}6 \mid \varepsilon) \mid no_0\text{-}6$

$no_0\text{-}4 \to (5 \mid no_0\text{-}5\ 5)(no_0\text{-}5\ 5)^*(no_0\text{-}5 \mid \varepsilon) \mid no_0\text{-}5$

$no_0\text{-}3 \to (4 \mid no_0\text{-}4\ 4)(no_0\text{-}4\ 4)^*(no_0\text{-}4 \mid \varepsilon) \mid no_0\text{-}4$

$no_0\text{-}2 \to (3 \mid no_0\text{-}3\ 3)(no_0\text{-}3\ 3)^*(no_0\text{-}3 \mid \varepsilon) \mid no_0\text{-}3$

$no_0\text{-}1 \to (2 \mid no_0\text{-}2\ 2)(no_0\text{-}2\ 2)^*(no_0\text{-}2 \mid \varepsilon) \mid no_0\text{-}2$

$no_0 \to (1 \mid no_0\text{-}1\ 1)(no_0\text{-}1\ 1)^*(no_0\text{-}1 \mid \varepsilon) \mid no_0\text{-}1$

$answer \to (0 \mid no_0\ 0)(no_0\ 0)^*(no_0 \mid \varepsilon) \mid no_0$

分析 刚拿到这个问题，一定不知从哪儿下手。其实和上面一样，关键是找到一种合适的看待这种句子结构的观点。我们的观点是这样，每个这样的句子由若干个 0 把它分成若干段，如

12303135710667803559O123

可以看成

123, 0, 313571, 0, 6678, 0, 3559, 0, 123

由 0 隔开的每一段，如 313571，它不含 0，并且又可以看成是由若干个 1 把它分成若干段。如此下去，就能找到该语言的正规定义。

按这个思路，上面的正规定义应该逆序看。

$$answer \rightarrow (0 \mid no_0\,0)\,(no_0\,0)^*\,(no_0 \mid \varepsilon) \mid no_0$$

表示一个句子由若干个 0 分成若干段，特殊情况是整个句子不含 0。在这个正规定义中，所引用的 *no_0* 表示不含 0 的串，它的定义和这个定义的形式一样，因为串的形式是一样的，只不过没有数字 0。所以有

$$no_0 \rightarrow (1 \mid no_0\text{-}1\,1)\,(no_0\text{-}1\,1)^*\,(no_0\text{-}1 \mid \varepsilon) \mid no_0\text{-}1$$

其中 *no_0-1* 表示不含 0和 1 的串。依此类推，最后 *no_0-8* 是表示不含 0, … , 8 的没有重复数字的串，它只可能是单个 9。

1.4 构造一个 DFA，它接受 $\Sigma = \{0, 1\}$ 上 0和 1 的个数都是偶数的字符串。

答案 见图 1.1。

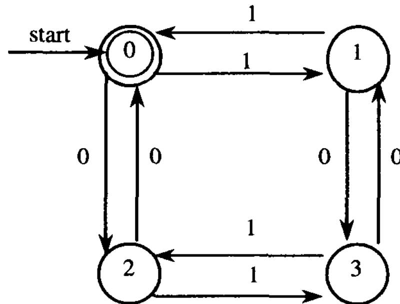

图 1.1 接受偶数个 0 和偶数个 1 的的 DFA

分析 对于这样的问题，不要急于去尝试画 DFA，先把问题分析一下，这里要接受的是偶数个 0 和偶数个 1 的串，和偶数相对的是奇数，因此，对于任意一个 0 和 1 的串，不论其 0和 1 的个数有多少，总归不是偶数个就是奇数个。因此任意一个串属于下面四种情况之一。

0: 偶数个 0和偶数个 1；
1: 偶数个 0 和奇数个 1；
2: 奇数个 0和偶数个 1；
3: 奇数个 0 和奇数个 1。

并且不管一个串是处于上面哪一种情况，该串再添加一个 0 或 1 后，总是处于上面另一种情况。由此分析可以知道，DFA 只需四个状态就够了，并且状态转换也很容易画出来。答案中的四个状态对应到这儿的四种情况。空串是属于偶数个 0 和偶数个 1 的情况，因此 0 状态是开始状态。因为我们接受偶数个 0 和偶数个 1 的串，因此它也是接受状态。

此为试读,需要完整PDF请访问: www.ertongbook.com

1.5　构造一个 DFA，它接受 $\Sigma = \{0, 1\}$上能被 5整除的二进制数。

答案　见图 1.2。

分析　由上题我们知道，构造 DFA 之前，首先搞请楚问题的状态空间。即想明白应该有多少个状态，状态之间的转换条件，以及针对该问题的开始状态和接受状态。对于本题目，任意一个二进制数除以 5 时，只有余数为 0（即整除），1，2，3 和 4 五种情况。图中的五个状态也是这样起名字的。一个二进制数的后面添上一个 0 意味着其值变成原来的两倍，而后面添上一个 1 意味着其值变成原来的两倍再加 1。不管是哪一种情况，都很容易从原来的余数决定值变化后的余数。这样，我们很快可以得出所有的状态转换。例如，我们考虑状态 4。任何一个余 4 的数，两倍后一定余 3，两倍再加 1 后一定还是余 4。所以，状态 4 的 0 转换到状态 3，而 1 转换到本身。显然，状态 0 既是开始状态又是接受状态。

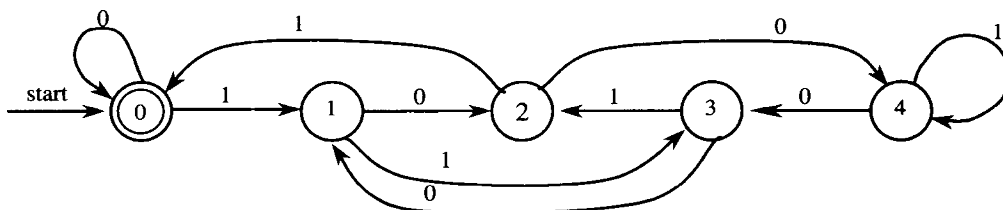

图 1.2 接受能被 5整除的二进制数的 DFA

需要注意的是，考虑状态空间时，还要检查我们是否取的是最简情况（即状态数极小）。例如，对于本题目，假如我们从这样的观点出发，每个二进制数都可以转换成一个十进制数。十进制数的末位有 0 到 9 十种情况，其中末位为 0 和 5 是能被 5 整除的情况。这样我们很可能会构造十个状态的 DFA，接受状态有两个。这也是一种解，但它不是最简的 DFA。

1.6　处于/* 和 */之间的串构成注解,注解中间没有*/。画出接受这种注解的 DFA 的状态转换图。

答案　见图 1.3。标记为 others 的边是指字符集中未被别的边指定的任意其它字符。

分析　这个 DFA 的状态数及含义并不难确定，见下面的五个状态说明。

状态 1：注释开始状态。

状态 2：进入注释体前的中间状态。

状态 3：表明目前正在注释体中的状态。

状态 4：离开注释前的中间状态。

状态 5：注释结束状态，即接受状态。

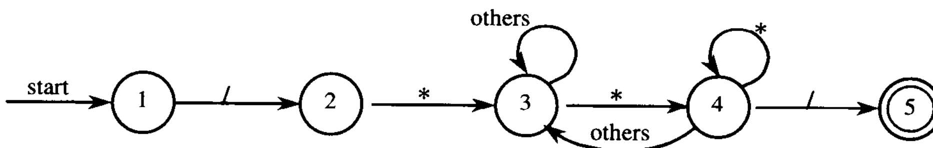

图 1.3　接受注释的 DFA

在这个 DFA 中，最容易忽略的是状态 4 到本身的‘*’转换。这个边的含义是：在离开注释

前的中间状态，若下一个字符是‘*’，那么把刚才读过的‘*’看成是注释中的一个字符，而把这下一个字符看成可能是结束注释的第一个字符。若没有这个边，那么像

/**** This is a comment **** /

这样的注释就被拒绝。

另外，上面的状态转换图并不完整。例如，对于状态 1，没有指明遇到其它字符怎么办。要把状态转换图画完整，还需引入一个死状态 6，进入这个状态就再也出不去了。因为它不是接受状态，因此进入这个状态的串肯定不被接受。完整的状态转换图见下面图 1.4，其中 all 表示任意字符。在能够说清问题时，通常我们省略死状态和所有到它的边。

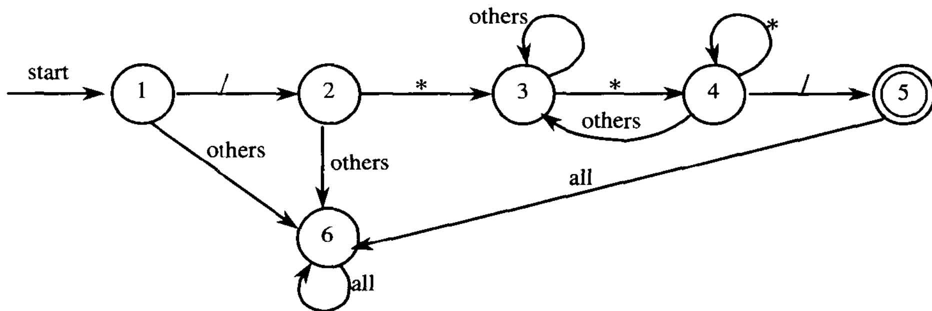

图 1.4 接受注释的完整 DFA

1.7 某操作系统下合法的文件名为

device:name.extension

其中第一部分（device:）和第三部分（.extension）可缺省，若 device, name 和 extension 都是字母串，长度不限，但至少为 1，画出识别这种文件名的 DFA。

答案 见图 1.5，图中的标记 d 表示任意字母。

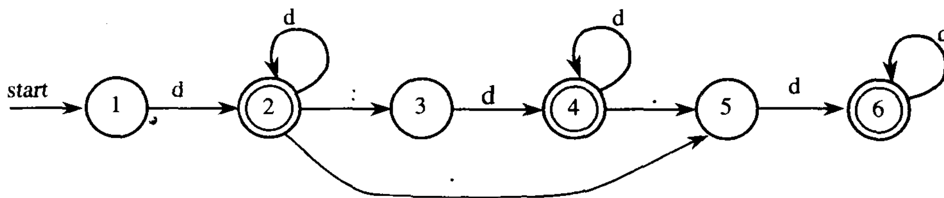

图 1.5 接受文件名的 DFA

分析 这个 DFA 和一些教材上接受无符号数的 DFA 有类似的地方。我们首先考虑 device: 和 .extension 全都出现的情况。这时的 DFA 比较容易构造，见图 1.6。

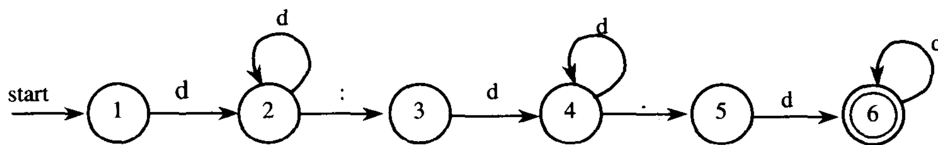

图 1.6 文件名的三部分都出现的 DFA

然后考虑缺省情况。因为 .extension 可缺省，因此把状态 4 也作为接受状态。因为 name 和 device 一样，都是字母序列，因此在 device:缺省时，把到状态 2 为止得到的字母序列看成是 name，所以从状态 2 画一条转换边到状态 5，标记为 '.'。(如果构成 name 和 device 的字符完全不一样，那么可以从状态 1 到状态 4 画一条边，其标记同状态 3 到状态 4 的标记一样。)由于 device:和 .extension 都可缺省，因此把状态 2 也作为接受状态。

1.8 为正规式(a | b)* a (a | b) (a | b)构造 NFA。
答 案 该 NFA 的状态转换图见图 1.7。

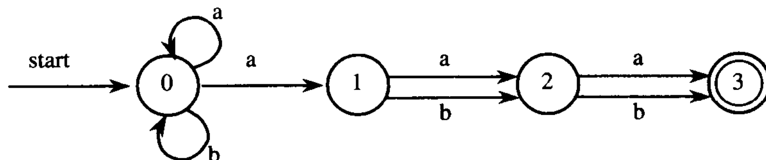

图 1.7 接受正规式(a | b)* a (a | b) (a | b)的 NFA

分析 各种教材在介绍有限状态自动机和正规表达式的等价时，都给出了从正规表达式构造等价的 NFA 的算法。不同书上的构造算法虽然不一样，但有一个共同的特点，或多或少引入了 ε 转换，使状态转换图变得复杂。尤其是，如果题目还要求你画出 DFA，那么状态数的增多，使得手工完成 NFA 确定化为 DFA 的过程变得更容易出错。因此，我们既要会用书上的算法构造 NFA，也要会手工构造更简洁一些的 NFA，尽量避免在 NFA 中出现 ε 转换。这在大多数情况下是可以做到的，本题就是一个例证。

1.9 用状态转换图表示接收(a | b)* aa 的 DFA。
答案 状态转换图见图 1.8。

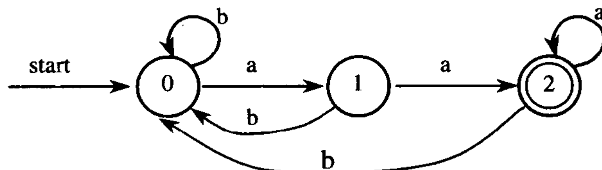

图 1.8 接收(a | b)* aa 的 DFA

分析 和上一题不同的是，现在是直接构造 DFA。我们仍然坚持这一点，大家既要会按教材上的算法从 NFA 的确定化得到 DFA，也要会手工直接构造 DFA。我们通过本题和下一题来说明，手工直接构造 DFA 也并不困难。

该正规式表示的语言是，字母表 Σ= {a, b}上最后两个字符都是 a 的串的集合。抓住这个特点，我们首先画出构造过程中的第一步，见图 1.9。它表明最简单的句子是 aa。

然后，因为在第一个 a 前可以有若干个 b，因此状态 0 有到自身的 b 转换。在最后两个字符都是 a 的串的末尾添加若干个 a，能够保持串的这个性质，因此状态 2 有到自身的 a 转换。这样我们有图 1.10。

最后，在状态 1 和状态 2 碰到 b 时，前面刚读过的 a，不管连续有多少个，都不可能作为

句子结尾的那两个字符 a，因此状态 1 和状态 2 的 b 转换回到状态 0。所有状态的 a 转换和 b 转换都已给出，这就得到最后结果。

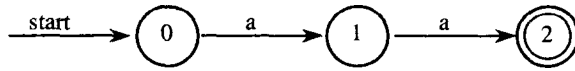

图 1.9 构造过程中的第一步

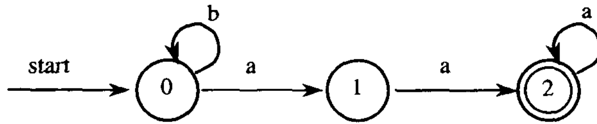

图 1.10 构造过程中的第二步

1.10 用状态转换图表示接收(a | b)* a (a | b) (a | b)的 DFA。

答案 状态转换图见图 1.11。

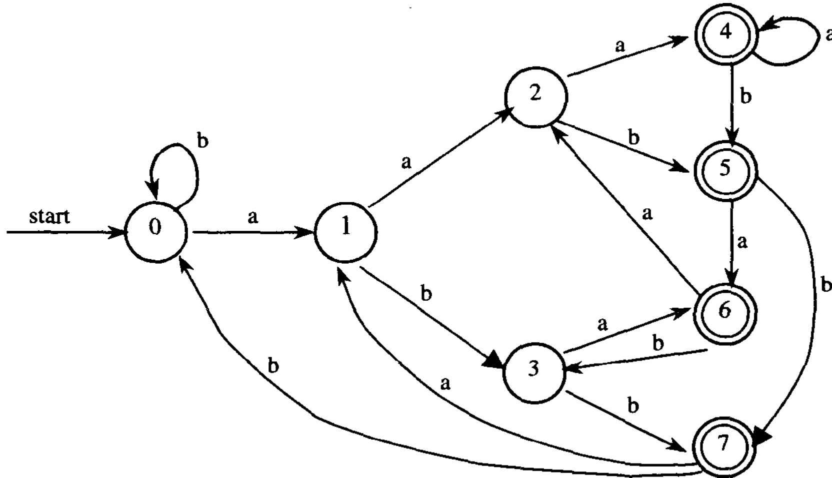

图 1.11 接收(a | b)* a (a | b) (a | b)的 DFA

分析 该正规式表示的语言是，字母表 $\Sigma=\{a, b\}$ 上倒数第三个字符是 a 的串的集合。根据上题的经验，我们首先画出图 1.12。因为最后两个字符任意，因此有这样的分杈，并有四个接受状态。

现在考虑这四个接受状态上的转换。

1. 状态 4 该状态表示最后三个字符是 aaa，若再添加一个 a，最后三个字符仍是 aaa，因此状态 4 的 a 转换到本身。若添加的是 b，那么最后三个字符是 aab，而状态 5 表示最后三个字符是 aab，因此状态 4 的 b 转换到状态 5。

2. 状态 5 该状态表示最后三个字符是 aab，若再添加一个 a，最后三个字符成了 aba，而状态 6 表示最后三个字符是 aba，因此状态 5 的 a 转换到状态 6。若添加的是 b，那么最后三个字符是 abb，而状态 7 表示最后三个字符是 abb 因此状态 5 的 b 转换到状态 7。

3. 状态 6 该状态表示最后三个字符是 aba，若再添加一个 a，最后三个字符成了 baa，其由 a 开始的后缀是 aa，因此状态 6 的 a 转换到状态 2（因为从状态 0 出发经 aa 是到状态 2）。

此为试读，需要完整PDF请访问：www.ertongbook.com

若添加的是 b，那么最后三个字符是 bab，其由 a 开始的后缀是 ab，因此状态 6 的 b 转换到状态 3.

4. 状态 7　该状态表示最后三个字符是 abb，若再添加一个 a，最后三个字符成了 bba，其由 a 开始的后缀是 a，因此状态 7 的 a 转换到状态 1。若添加的是 b，那么最后三个字符是 bbb，不存在由 a 开始的后缀，因此状态 7 的 b 转换到状态 0。

这样，所有状态的 a 转换和 b 转换都已给出，也就得到了最后结果。

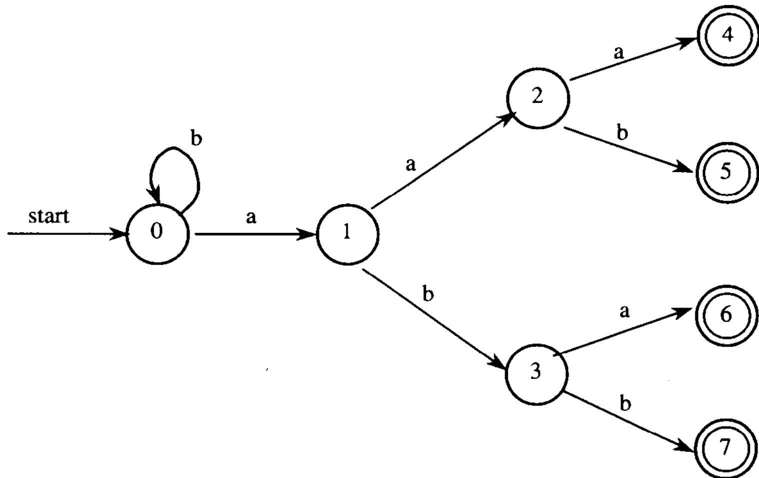

图 1.12　构造过程的第一步

1.11　将 1.8 得到的 NFA 变换成 DFA。

答案　所求的 DFA 就是 1.10 题的结果。

分析　我们之所以选这个题目，是为了比较一下，从正规式到 NFA，再把 NFA 确定化，这样得的结果同 1.10 题直接构造 DFA 的结果是否一样。

按照教材上的子集构造法，作为结果的 DFA 并不难得到。另外由于没有 ε 转换，构造过程相对简单了很多。

NFA 的开始状态是 0，因此首先从 NFA 的状态集合{0}开始，它是 DFA 的开始状态，起名叫状态 0′。它的 a 转换和 b 转换所得到的 NFA 的状态集合见下面第一行。根据子集构造法所得的 DFA 的所有状态和它们的转换函数都列在下面。

状态 0′:{0}	*move* ({0}, a) = {0, 1}	*move* ({0}, b) = {0}
状态 1′:{0, 1}	*move* ({0, 1}, a) = {0, 1, 2}	*move* ({0, 1}, b) = {0, 2}
状态 2′:{0, 1, 2}	*move* ({0, 1, 2}, a) = {0, 1, 2, 3}	*move* ({0, 1, 2}, b) = {0, 2, 3}
状态 3′:{0, 2}	*move* ({0, 2}, a) = {0, 1, 3}	*move* ({0, 2}, b) = {0, 3}
状态 4′:{0, 1, 2, 3}	*move* ({0, 1, 2, 3}, a) = {0, 1, 2, 3}	*move* ({0, 1, 2, 3}, b) = {0, 2, 3}
状态 5′:{0, 2, 3}	*move* ({0, 2, 3}, a) = {0, 1, 3}	*move* ({0, 2, 3}, b) = {0, 3}
状态 6′:{0, 1, 3}	*move* ({0, 1, 3}, a) = {0, 1, 2}	*move* ({0, 1, 3}, b) = {0, 2}
状态 7′:{0, 3}	*move* ({0, 3}, a) = {0, 1}	*move* ({0, 3}, b) = {0}

状态 4′, 5′, 6′和 7′中都含原 NFA 的接受状态 3，因此它们都是 DFA 的接受状态。不难看出所得的 DFA 和 1.10 题的结果是同构的，仅状态名不一样。

1.12　将图 1.13 的 DFA 极小化。

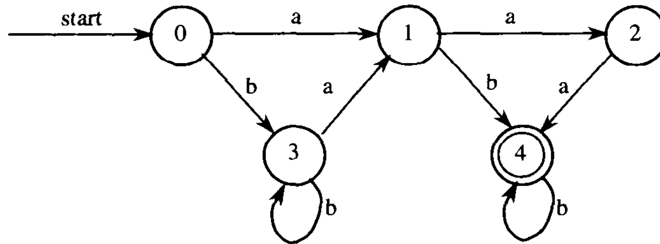

图 1.13　一个 DFA

答案　最简 DFA 见图 1.14。

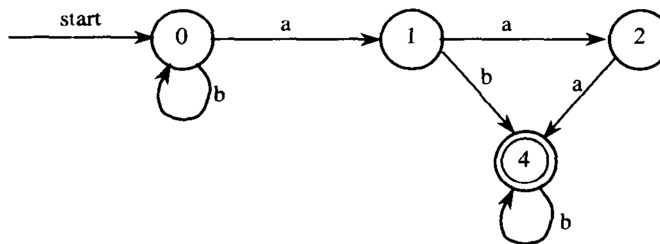

图 1.14　最简 DFA

分析　本题要注意的是，在使用极小化算法前，一定要检查一下，看状态转换函数是否为全函数，即每个状态对每个输入符号都有转换。若不是全函数，需加入死状态，然后再用极小化算法。有些教材上没有强调这一点，有的习题解上的示例甚至忽略了这一点，本题将告诉你，这一点是重要的。本题加入死状态 5 后的状态转换图见图 1.15。

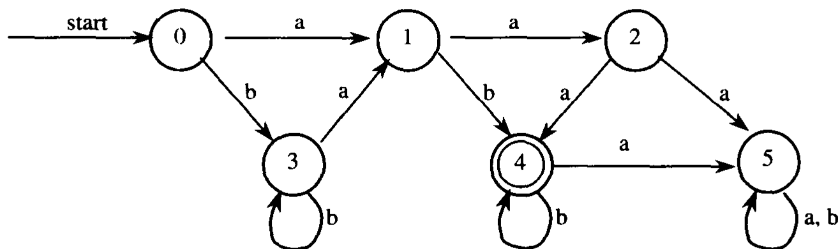

图 1.15　加入死状态后的 DFA

使用极小化算法，先把状态集分成非接受状态集{0, 1, 2, 3, 5}和接受状态集{4}这两个子集。

1. 集合{4}不能再分解，我们看集合{0, 1, 2, 3, 5}。

$move$ ({0, 1, 2, 3, 5}, a) = {1, 2, 5}　　　　$move$ ({0, 1, 2, 3, 5}, b) = {3, 4, 5}

由于 b 转换的结果{3, 4, 5}不是最初划分的某个集合的子集，因此{0, 1, 2, 3, 5}需要再分，由于状态 1 和状态 2 的 b 转换都到状态 4。因此状态集合的进一步划分是：

{1, 2}, {0, 3, 5} 和{4}

2. 由于

$move(\{1,2\}, a) = \{2, 5\}$　　$move(\{1,2\}, b) = \{4\}$

$move(\{0,3,5\}, a) = \{1, 5\}$　　$move(\{0,3,5\}, b) = \{3, 5\}$

显然 {1, 2} 和 {0, 3, 5}需要再分，分别分成：

{1}和{2} 以及{0, 3}和{5}

3. 由于

$move(\{0,3\}, a) = \{1\}$　　$move(\{0,3\}, b) = \{3\}$

因此不需要再分。这样状态 0 和状态 3 合并成一个状态，我们取 0 为代表，再删去死状态 5，就得到该题的结果。

如果不加死状态，我们来看一下极小化算法的结果。最初的划分是{0, 1, 2, 3} 和 {4}。

1. 状态集合的进一步划分是：

{1, 2}，{0, 3}和 {4}

2. 忽略了死状态的影响，会认为它们都不需要再分，此时得到的 DFA 如图 1.16。显然，它和原来的 DFA 不等价。

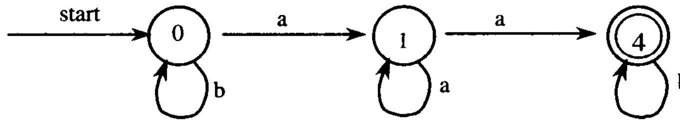

图 1.16　一个不正确的结果

1.13　将习题 1.10 结果的 DFA 极小化。

答案　化简的结果仍是习题 1.10 结果的 DFA，即该 DFA 已是最简 DFA。这说明手工构造最简 DFA 是完全可能的。

分析　我们简要说明执行过程。

初始时将状态集分成两组：{0, 1, 2, 3}和 {4, 5, 6, 7}。

1. 由于

$move(\{0,1,2,3\}, a) = \{1, 2, 4, 6\}$　　$move(\{0,1,2,3\}, b) = \{0, 3, 5, 7\}$

$move(\{4,5,6,7\}, a) = \{1, 2, 4, 6\}$　　$move(\{4,5,6,7\}, b) = \{0, 3, 5, 7\}$

因此进一步分成 {0, 1}、{2, 3}、{4, 5}和 {6, 7} 四组。

2. 由于

$move(\{0,1\}, a) = \{1, 2\}$　　$move(\{0,1\}, b) = \{0, 3\}$

因此{0, 1}还要进一步分，其它几组也是这样。因此最终分成了每个集合中都只有一个状态。

3. 所以原来的 DFA 已经是最简形式了。

1.14　若 L 是正规语言，证明下面 L′语言也是正规语言。L′语言的定义是

$$L' = \{x \mid x^R \in L\}$$

x^R表示 x 的逆。

答案　若我们能定义出接受语言 L′的一个 NFA，那么 L′是正规语言。因为 L 是正规语言，那么一定存在接受 L 的 DFA *M*，我们就基于 *M* 来构造接受 L′的 NFA *M′*，并且我们基于 *M* 的状态转换图来叙述。

1. 将状态转换图上的所有边改变方向，边上的标记不变。

2. 将原来的开始状态改成接受状态。

3. 增添一个状态作为开始状态，从它有 ε 转换到原来的每一个接受状态，并把原来的这些接受状态都改成普通的状态。

所得到的新状态转换图就是 M' 的状态转换图。

分析 简单说，判断一个串是否为 L' 的句子，只要在原来的状态转换图上逆行就可以了。由于 M 可能有多个接受状态，而不管是 DFA 还是 NFA 都只有唯一的开始状态，因此有上面的第 3 步 。

注意经过上面的改造后，得到的可能不是一个 DFA 的状态转换图。除了上面第 3 步有可能引入不确定性外，第 1 步也可能引入不确定性。

本题介绍的证明语言的正规性的方法在下一题表现得更加充分，叙述也比本题严格。

1.15 若 L 是正规语言，证明下面 $\frac{1}{2}L$ 语言也是正规语言。$\frac{1}{2}L$ 语言的定义是

$$\frac{1}{2}L = \{x \mid \exists y.\ xy \in L \ \&\ |x| = |y|\}$$

答案 因为 L 是正规语言，那么存在一个 DFA $M = (S, \Sigma, \delta, s, F)$，它接受语言 L。接受语言 $\frac{1}{2}L$ 的 DFA $M' = (S', \Sigma, \delta', s', F')$ 可如下构造。

1. S' 的每个状态是一个二元组 $\langle s_1, S_1\rangle$，其 中 $s_1 \in S$，$S_1 \subseteq S$。

2. $\delta'(\langle s_1, S_1\rangle, a) = \langle s_2, S_2\rangle$，其 中

   $s_2 = \delta(s_1, a)$

   $S_2 = \{ s_2 \mid \exists s_1 \in S_1.\exists b \in \Sigma.\delta(s_2, b) = s_1\}$

3. $s' = \langle s, F\rangle$

4. $F' = \{\langle s_1, S_1\rangle \mid s_1 \in S_1\}$

可以证明这是接受语言 $\frac{1}{2}L$ 的 DFA，因此 $\frac{1}{2}L$ 是正规语言（证明比较复杂，我们在此略去）。

分析 这个题目超出了编译课程的范围。但是如果理解了这儿的解答，那就掌握了证明正规语言的一种很重要的技巧。下面我们说明，为什么这样定义 DFA M'。

长度为 n 的串 w，若它是语言 L 的某个句子的前缀，那么从 M 的开始状态 s，经 n 步转换，到达某个状态 s_n。该串是否属于 $\frac{1}{2}L$，取决于是否存在从 s_n 开始到某个接受状态的路径，其长度为 n。

怎样判断是否存在这样的到某个接受状态的路径呢？我们可以在 M 的状态转换图上按下面的步骤办。

1. 首先找出从任意接受状态逆向任意走一步能到达的所有状态，把这个状态集合称作 R_1。

2. 再找出从 R_1 的任意状态逆向任意走一步能到达的所有状态，把这个状态集合称作 R_2。

3. 这样逆向操作 n 步后，得到状态集合 R_n。

此为试读，需要完整PDF请访问：www.ertongbook.com

4．若 s_n 在 R_n 中，那么串 w 属于 $\frac{1}{2}L$，否则不是。

可以看出，上面定义的 M' 是在 M 上同时跟踪从开始状态出发的正向状态转换和从接受状态集合出发的逆向搜索所有状态。我们再看一下 M' 的定义。

1．S' 的每个状态之所以是一个二元组 $\langle s_1, S_1\rangle$，是因为需要用 s_1 来表示正向的状态转换，用 S_1 来表示逆向搜索。

2．再看状态转换的定义 $\delta'(\langle s_1, S_1\rangle, a) = \langle s_2, S_2\rangle$，其中 $s_2 = \delta(s_1, a)$ 表示了 M 上正向一步转换到达的状态，$S_2 = \{ s_2 \mid \exists s_1 \in S_1 . \exists b \in \Sigma . \delta(s_2, b) = s_1\}$ 表示了 M 上逆向一步搜索到达的状态集合。

3．开始状态 $s' = \langle s, F\rangle$ 表示在 M 上的跟踪是从开始状态和接受状态集合这两端启动。

4．接受状态 $F' = \{\langle s_1, S_1\rangle \mid s_1 \in S_1\}$ 表达的意思是，若在 M 上从开始状态经 n 转换到 s_1，那么一定存在长度为 n 的从 s_1 到某个接受状态的路径。

下面我们以 1.4 题的正规语言为例。不难理解，该语言的 $\frac{1}{2}$ 语言应该是字母表 $\Sigma=\{0, 1\}$ 上的所有串。我们用上面的方法来构造接受它的 DFA。该 DFA 的开始状态 $s_0 = \langle 0, \{0\}\rangle$。

状态 s_0: $\langle 0, \{0\}\rangle$　　$\delta(s_0, 0) = \langle 2, \{1, 2\}\rangle$　　$\delta(s_0, 1) = \langle 1, \{1, 2\}\rangle$

状态 s_1: $\langle 2, \{1, 2\}\rangle$　　$\delta(s_1, 0) = \langle 0, \{0, 3\}\rangle$　　$\delta(s_1, 1) = \langle 3, \{0, 3\}\rangle$

状态 s_2: $\langle 1, \{1, 2\}\rangle$　　$\delta(s_2, 0) = \langle 3, \{0, 3\}\rangle$　　$\delta(s_2, 1) = \langle 0, \{0, 3\}\rangle$

状态 s_3: $\langle 0, \{0, 3\}\rangle$　　$\delta(s_3, 0) = \langle 2, \{1, 2\}\rangle$　　$\delta(s_3, 1) = \langle 1, \{1, 2\}\rangle$

状态 s_4: $\langle 3, \{0, 3\}\rangle$　　$\delta(s_4, 0) = \langle 1, \{1, 2\}\rangle$　　$\delta(s_4, 1) = \langle 2, \{1, 2\}\rangle$

从 s_0 到 s_4 的这五个状态，每个状态的第一元都在第二元的集合中，因此这五个状态都是接受状态。这个 DFA 的图形表示见图 1.17（图中用状态 0, 1, 2, 3 和 4 代替了这里对应的状态）。显然，它能接受字母表 $\Sigma=\{0, 1\}$ 上所有的串，化简这个 DFA 可以得到只有一个状态的 DFA。

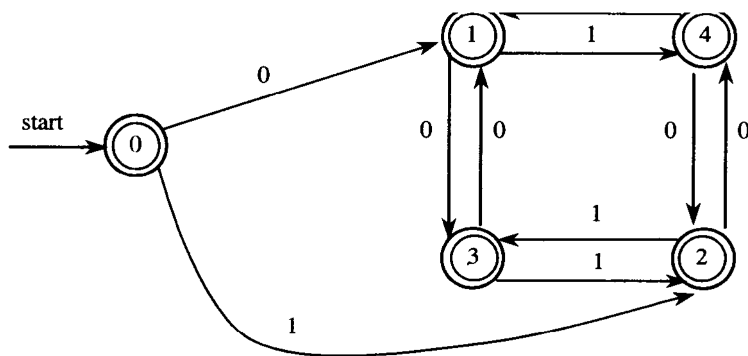

图 1.17　接受 $\Sigma=\{0, 1\}$ 上所有串的一个 DFA

1.16　保留字、关键字和标准标识符之间的区别是什么？

答案　保留字是语言预先确定了含义的单词，程序员不可以对这样的单词重新声明它的含义。如 Pascal 语言的 type 和 procedure 等。词法分析器读到一个符合标识符拼写的单词时，都要

和保留字进行比较，以确定该单词是标识符呢，还是哪个保留字。

很多语言的关键字是保留的，因此和上面的保留字没有区别，如 C 语言和 Java 语言。但是 FORTRAN 语言的关键字不保留，如 IF，当它作为语句的第一个单词时，很可能是关键字，但也不排除是用户定义的标识符。这给词法分析带来很大困难，因为识别单词和该单词所处的上下文有关。现在的语言都不会像 FORTRAN 这样来定义关键字。

标准标识符是预先确定了含义的单词，但是程序员可以重新声明它的含义。在这个声明的作用域内，程序员声明的含义起作用，而预先确定的含义消失，如 Pascal 语言的 integer 和 true 等。词法分析器对标准标识符没有什么特别的处理，由符号表管理来解决这件事。

1.17　词法分析器能查出源程序中什么样的错误？举例说明。

答案　词法分析器对源程序采取非常局部的观点，因此像 C 语言的语句

```
fi (a == f (x) ) ···
```

中，词法分析器把 fi 当作一个普通的标识符交给编译的后续阶段，而不会把它看成是关键字 if 的拼写错。

Pascal 语言要求作为实型常量的小数点后面必须有数字，如果程序中出现小数点后面没有数字情况，它由词法分析器报错。

1.18　一个 C 语言编译器编译下面的函数时，报告 parse error before ‘else’。这是因为 else 的前面少了一个分号。但是如果第一个注释

```
/* then part */
```

误写成

```
/* then part
```

那么该编译器发现不了遗漏分号的错误。这是为什么？

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        /* then part */
        return q
    else
        /* else part */
        return gcd(q, p%q);
}
```

答案　此时编译器认为

```
    /* then part
    return q
else
    /* else part */
```

是程序的注释，因此它不可能再发现 else 前面的语法错误。

分析 这是注释用配对括号表示时的一个问题。注释在词法分析时是忽略的，而词法分析器对程序采取非常局部的观点。当进入第一个注释后，词法分析器忽略输入符号，一直到出现注释的右括号为止，由于第一个注释缺少右括号，所以词法分析器在读到第二个注释的右括号时，才认为第一个注释处理结束。

为克服这个问题，后来的语言一般都不用配对括号来表示注释。例如 Ada 语言的注释始于双连字符（--），随行的结束而终止。如果用 Ada 语言的注释格式，那么上面函数应写成

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        --  then part
        return q
    else
        -- else part
        return gcd(q, p%q);
}
```

第二章　　语法分析

本章主要掌握下面一些内容。

1. 文法和语言的基本知识。有些教材把这方面内容单独作为一章，但是把它和语法分析方法放在一起学习，可以了解得更深刻。

2. 自上而下的分析方法：预测分析器（递归下降分析方法），非递归的预测分析器（分析表方法），LL(1)文法。

3. 算符优先分析方法。现在的编译器已很少使用这种方法，因此有些教材已不介绍这种方法。

4. 自下而上的分析方法：SLR(1)方法，规范 LR(1)方法和 LALR(1)方法。在这三种方法中，重点是规范 LR(1)方法，规范 LR(1) 方法了解透彻了，其它两种就不难了。

5. LR 方法如何用于二义文法。

6. 语法分析的错误恢复方法。

2.1　文法

$$S \rightarrow a\,S\,b\,S \mid b\,S\,a\,S \mid \varepsilon$$

产生的语言是什么？该文法是否二义？

答案　该文法产生的语言是 a 的个数和 b 的个数相等的串的集合。该文法二义，例如句子 abab 有两个不同的最左推导。

$$S \Rightarrow a\,S\,b\,S \Rightarrow a\,b\,S \Rightarrow aba\,S\,b\,S \Rightarrow abab\,S \Rightarrow abab$$

$$S \Rightarrow a\,S\,b\,S \Rightarrow ab\,S\,a\,S\,b\,S \Rightarrow aba\,S\,b\,S \Rightarrow abab\,S \Rightarrow abab$$

分析　为什么能判断该文法产生的语言是 a的个数和 b的个数相等的串的集合？因为三个产生式的右部，在除去非终结符后，都正好是a 的个数和 b 的个数相等的情况，而右部的非终结符又正好就是左部的非终结符。因此可以按推导的步数进行归纳，来证明该文法推导出的句子一定是 a 的个数和 b 的个数相等的串。另一方面还需要判断，每个这样的串，是否都能由这个文法产生。我们以 a 开始的串为例，它一定能分解成 aw_1bw_2 的形式，其中 w_1 中 a 的个数和 b 的个数相等，w_2 也是这样。如果找不到这样的分解，那么该串中 a 的个数和 b 的个数就一定不相等。这样可以按串长进行归纳，得出所有这样的串都由该文法产生。后面有一个习题专门介绍怎样证明一个文法是某个语言的一个文法。

怎样判断该文法二义，并能找一个最简单的句子来说明问题？以 a 开始的串为例，我们要找尽可能简单的句子，使得 aw_1bw_2 形式的分解不唯一，那么该句子一定有两棵不同的语法树。空串和串 ab 都不可能。那么再看串长为 4 的几种情况：aabb，abab 和 abba，只有 abab 有可能：$w_1 = \varepsilon$且 $w_2 = ab$或者 $w_1 = ba$且 $w_2 = \varepsilon$。因此 abab一定有两种不同形式的最左推导，我们在上面的答案中已经给出。

2.2 下面的二义文法描述命题演算公式，为它写一个等价的非二义文法。

S → S **and** S | S **or** S | **not** S | p | q | (S)

答案 非二义文法的产生式如下：

E → E **or** T | T

T → T **and** F | F

F → **not** F | (E) | p | q

分析 和教材上算术表达式的二义文法一样，由于该文法中没有体现算符 and、or和 not 的优先次序和结合规则，因此该文法二义。如 p **and** q **or** p 可以分解成求两个子公式 p **and** q 和 p 进行 or 运算，或者分解成求 p 和 q **or** p 进行 and运算，见图 2.1 的两棵不同的语法树。

图 2.1 p **and** q **or** p 的两棵不同语法树

要使得一个公式（下面我们称表达式）的分解唯一，就必须体现算符的优先次序和结合规则。我们把一个表达式看成是若干个项进行 or 运算，并且 or 运算左结合，每个项看成是若干个因子进行 and 运算，and 运算也左结合，因子有四种情况：p、q、加括号的表达式，或者加 not 前缀的因子。按这种观点，一个表达式的结构分解只有一种可能性，因而不会二义。上面的文法就是按这种观点写的。

一个易犯的困惑是，究竟用产生式 F → **not** E 还是用 F → **not** F？在这种由算符优先级体现的层次结构中，若高层结构（如 E）直接作为低层算符（如 not，运算的优先级高）的运算对象，很可能会使文法产生二义。在上面的文法中，如果使用 F → **not** E 代替 F → **not** F，则 **not** p **and** q 有两棵不同的语法树，见图 2.2 （为节约版面，部分地方用虚线表示省略）。

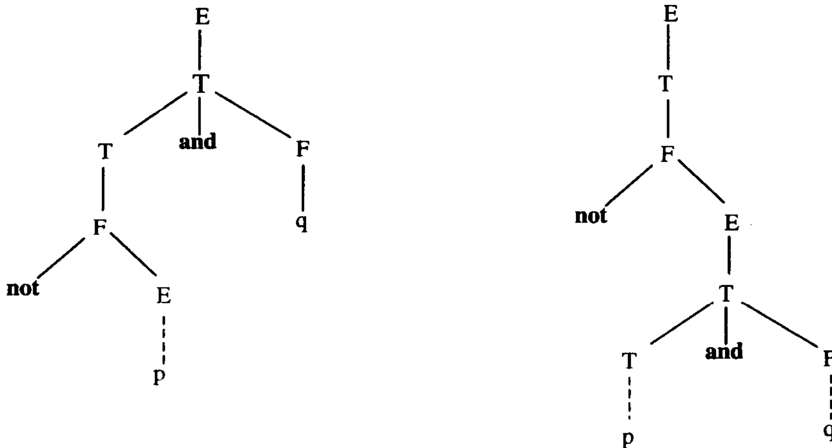

图 2.2 not p and q 两棵不同语法树

2.3 **文法**

R → R '|' R | RR | R* | (R) | a | b

产生字母表{a, b}上所有不含ε的正规式。注意：第一条竖线是正规式的符号"或"，而不是文法产生式右部各选择之间的分隔符；另外'*'在这儿是一个普通的终结符。该文法是二义的。为该文法写一个等价的非二义文法。

答案 有了上一题的经验，读者应该自己能写出答案。根据正规表达式的约定，'*'的优先级最高，并置（无算符）次之，'|'的优先级最低。非二义文法如下。

E → E '|' T | T

T → T F | F

F → F* | (E) | a | b

分析 也许你会担心，上一题的 **not** 是右结合的算符，而本题的'*'是左结合算符，设计这两个文法时是否要考虑这一点区别。这个问题在设计语言时已考虑了，右结合的一元算符放在运算对象的前面，而左结合的一元算符放在运算对象的后面。我们在图 2.3 中给出上题 **not not** q 和本题 a**的语法树如下，它们分别体现 **not** 的右结合和'*'的左结合。

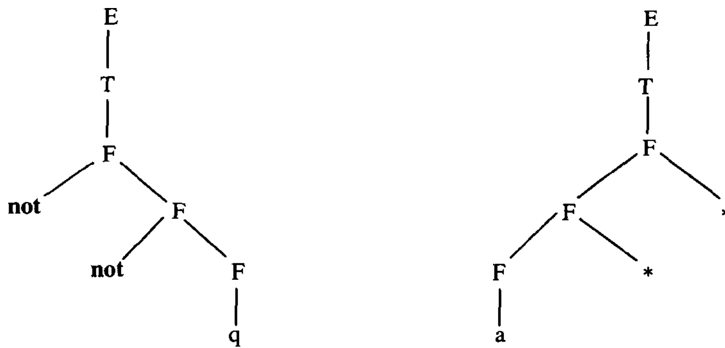

图 2.3 **not not** q 和 a**的语法树

2.4 **下面的条件语句文法**

stmt → **if** *expr* **then** *stmt* | *matched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *stmt* | **other**

试图消除悬空 else 的二义性。请你证明该文法仍然是二义的。

答案 句型 **if** *expr* **then if** *expr* **then** *matched_stmt* **else if** *expr* **then** *matched_stmt* **else** *stmt* 有两个不同的最左推导（见下面），因此该文法仍然是二义的。

stmt ⇒ *matched stmt*

⇒ **if** *expr* **then** *matched_stmt* **else** *stmt*

⇒ **if** *expr* **then if** *expr* **then** *matched_stmt* **else** *stmt* **else** *stmt*

⇒ **if** *expr* **then if** *expr* **then** *matched_stmt* **else if** *expr* **then** *stmt* **else** *stmt*

⇒ **if** *expr* **then if** *expr* **then** *matched_stmt* **else if** *expr* **then** *matched_stmt* **else** *stmt*

或者

此为试读，需要完整PDF请访问：www.ertongbook.com