文法/正规式/有限自动机
（课程预备知识）

2024-9

形式语言与自动机理论在编译程序的设计原理和构造方法中扮演了极其重要的角色。特别是在词法分析、语法分析乃至语义分析阶段，上下文无关文法、正规表达式和有限状态自动机等模型的引入，使得相应的语言定义更加严格，语言的处理过程更加有系统性，并以此为基础产生了许多有用的自动构造工具。

这一讲的绝大部分内容来自<形式语言与自动机>课程，修过或正在修该课的同学不必再看，课堂上大部分内容也不会讲。没有修该课的同学可以依此文件自学。

1. 形式语言概念

本节首先引入关于语言的一些基本概念的记号。这里的语言是指形式语言，比自然语言更加抽象，具有严格的形式定义。

1.1 字母表和字

语言的载体是文字符号，如在自然语言中，汉语言的载体是有限个汉字，英语的载体是有限个英文字母。在一个（形式）语言中，我们用有限个形式符号来表达语言载体的文字，这个有限符号的集合称为该语言的**字母表**，字母表中的每一个形式符号称为**字母**。字母表常用符号 Σ 来表示。

例如，英文字母表 $\Sigma_1 = \{ a, b, \ldots, z, A, B, \ldots, Z \}$，汉字表 $\Sigma_2 = \{ \ldots, 编, \ldots, 译, \cdots \}$，化学元素表 $\Sigma_3 = \{ H, He, Li, \cdots, Une \}$，以及 $\Sigma_4 = \{任, 意, a, n, y\}$。

字母表 Σ 上的一个**字**，或称为**串**，为 Σ 中字母构成的一个有限序列。 不包含任何字母的字称为**空字**，或**空串**。常用 ε 表示任意字母表 Σ 上的空字。

设 $\Sigma = \{ a, b \}$，则 $\varepsilon, a, aaa, baba$ 等都是Σ 上的字。

字符串 w 的长度，记为 $|w|$ ，是包含在 w 中字母的个数。如，$|\varepsilon| = 0$，$|bbaba| = 5$。

1.2 关于字的运算和字母表上的运算

设 x，y 为字，且 $x = a_1 a_2 \ldots a_m$，$y = b_1 b_2 \ldots b_n$，则 x 与 y 的**连接**

$$x\,y = a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n$$

连接运算满足如下性质：

- $(x\,y)\,z = x(y\,z)$ （结合律）
- $\varepsilon x = x\varepsilon = x$ （ε 是连接运算的幺元）

- $|xy| = |x| + |y|$

设 Σ 为字母表，n 为任意自然数，则 Σ 的**幂** Σ^n 归纳定义如下：

- $\Sigma^0 = \{\varepsilon\}$；
- 若 $x \in \Sigma^{n-1}$，$a \in \Sigma$， 则 $ax \in \Sigma^n$；
- Σ^n 中的元素只能由以上步骤产生。

例如，设 $\Sigma = \{0, 1\}$，则 $\Sigma^0 = \{\varepsilon\}$， $\Sigma^1 = \{0, 1\}$，$\Sigma^2 = \{00, 01, 10, 11\}$，…

Σ 的**星闭包**（简称**闭包**）为 $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$。$\Sigma$ 的 **正闭包** $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$。对于 $\Sigma = \{0, 1\}$，我们有 $\Sigma^+ = \{0，1，00，01 ，10 ，11，\ldots\}$，及 $\Sigma^* = \{\varepsilon ，0，1，00，01 ，10 ，11，\ldots\}$。

显然，对于非空字母表 Σ，有 $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$， $\Sigma^+ = \Sigma^* - \{\varepsilon\}$。

1.3 语言

设 Σ 为字母表，则任何集合 $L \subseteq \Sigma^*$ 是 Σ 上的一个语言。

直观地讲，由 Σ 上的字构成的任何集合都是 Σ 上语言。例如：

- 英文单词集 $\{\ldots, English, \ldots, words, \ldots\}$ 是字母表 Σ_1 上的一个语言；
- 汉语四字成语集 $\{\ldots$, 语不惊人, …, 言之有物, …$\}$ 是字母表 Σ_2 上的一个语言；
- 部分化学分子式集 $\{\ldots, H_2O, \ldots, NaCl, \ldots\}$ 是字母表 $\Sigma = \{_i \mid _i$ 是下标形式的自然数$\} \cup \Sigma_3$ 上的一个语言；
- $\{any$, 任意$\}$ 是字母表 Σ_4 上的一个语言；

其中，Σ_1，Σ_2，Σ_3 和 Σ_4 见1.1节。

对任何字母表 Σ,不含任何字的集合 $\Phi \subseteq \Sigma^*$ 称为 Σ 上的**空语言**。注意 Φ 和语言 $\{\varepsilon\}$ 之间的差别，后者包含空字ε。

1.4 关于语言的运算

通过运算，由已知语言可以得到新的语言。在关于语言的运算中，最重要的有三个。它们是：并，连接和闭包。

设 Σ 为字母表，L 和 M 是Σ 上的两个语言，L 和 M 的**并**

$$L \cup M = \{w \mid w \in L \vee w \in M\} ,$$

L 和 M 的**连接**

$$LM = \{w_1 w_2 \mid w_1 \in L \wedge w_2 \in M\} ,$$

L 的**（星）闭包**（或称**闭包**）

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots = \cup_{i\geq 0} L^i \ ,$$

其中，$L^0 = \{\varepsilon\}$， $L^1 = L$， $L^2 = LL$， …，$L^n = L^{n-1}L$ 。

例如，设 $L = \{0 , 11\}$，$M = \{\varepsilon , 0 , 001\}$，则有 $L \cup M = \{\varepsilon , 0, 11, 001\}$，$LM = \{0$，00，11，110，0001，11001$\}$，$L^* = \{\varepsilon , 0 , 11 , 00 , 011 , 110 , 1111 , 000 , 0011 , 0110 , 01111 , 1100 , 11011 , 11110 , 111111 , \ldots \}$。

在上述定义中，L^n 称为 L 的**幂**，可递归定义如下：

- $L^0 = \{\varepsilon\}$；
- 若 $n>0$，则 $L^n = L^{n-1}L$ 。

根据幂运算，还可以定义 L 的**正闭包** $L^+ = L^1 \cup L^2 \cup \ldots = \cup_{i\geq 1} L^i$。

此外，由于语言是字的集合，因而任何关于集合的运算都可以作为语言的运算。

2. 正规语言及其描述

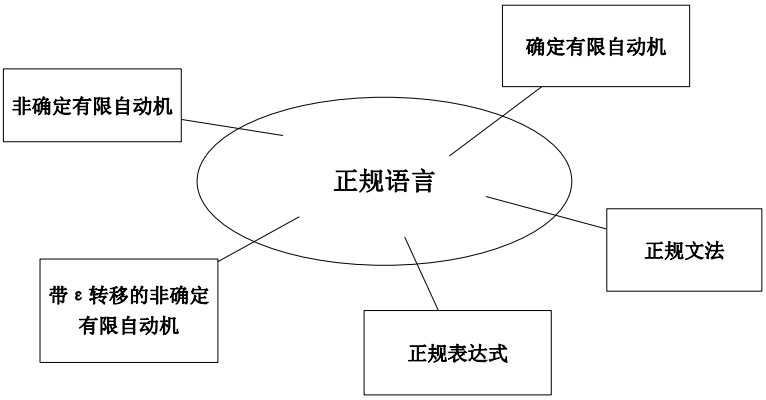


图 1　正规语言的不同表示模型

图1列举了本节将要介绍的正规语言表示模型：正规表达式，以及三类有限自动机。这些模型的计算能力是相互等价的，它们可以处理的语言都是正规语言。我们先从正规表达式以及通过正规表达式定义正规语言开始本节的内容。

2.1　正规表达式与正规语言

2.1.1　正规表达式

正规表达式是用来表示正规语言的一种代数表达式。最基本的正规表达式有三个基本运算符，分别对应作用于正规语言上的三种运算：并、连接、闭包。这三种运算的定义见前面一章，为方便，我们将其重新叙述如下：

设 Σ 为字母表，L 和 M 是 Σ 上的两个语言，则

- L 和 M 的并，$L \cup M = \{ w \mid w \in L \vee w \in M \}$。
- L 和 M 的连接，$LM = \{w_1 w_2 \mid w_1 \in L \wedge w_2 \in M\}$。
- L 的（星）闭包（或称闭包），$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots = \cup_{i \geq 0} L^i$ ，其中 $L^0 = \{\varepsilon\}$，$L^1 = L$， $L^2 = LL$， …，$L^n = L^{n-1}L$ 。

我们先给出在本课中使用的**正规表达式**的定义。

设 Σ 为字母表，则 Σ 上的正规表达式集合 $\Re$ 归纳定义如下:

（1）基础

- $\varepsilon \in \Re$ ；
- $\phi \in \Re$ ；
- 若 $a \in \sum$ ，则 $a \in \Re$

（2）归纳

- 若 $E \in \Re$ 和 $F \in \Re$ ，则 $E \mid F \in \Re$ ；
- 若 $E \in \Re$ 和 $F \in \Re$ ，则 $EF \in \Re$ ；
- 若 $E \in \Re$ ，则 $E^* \in \Re$ ；
- 若 $E \in \Re$ ，则 $(E) \in \Re$

以上定义给出了正规表达式的表示形式,即它的语法。下面我们给出正规表达式的语义，即它所代表的语言。

设 $\Re$ 为字母表 Σ 上的正规表达式集合。对于每个 $E \in \Re$，我们用 $L(E)$ 来表示 E 所代表的语言。$L(E)$ 可以归纳定义如下:

（1）基础

- $L(\varepsilon) = \{\varepsilon\}$ ；
- $L(\phi) = \Phi$ ， 这里，Φ 表示空语言；
- 若 $a \in \sum$ ，则 $L(a) = \{ a \}$

（2）归纳

- 若 $E \in \Re$ 和 $F \in \Re$ ，则 $L(E \mid F) = L(E) \cup L(F)$ ；
- 若 $E \in \Re$ 和 $F \in \Re$ ，则 $L(EF) = L(E)\,L(F)$ ；
- 若 $E \in \Re$ ，则 $L(E^*) = (L(E))^*$ ；
- 若 $E \in \Re$ ，则 $L((E)) = L(E)$

为了减少括号的数目，通常会规定这三种运算的优先级从高到低依次为：闭包、连接、并。

例 1 设计表示如下语言 L 的正规表达式：

$L = \{ w \mid w \in \{0,1\}^*$，且 w 由交替的 0 和 1 构成 $\}$

解 表示语言 L 的一个正规表达式为：

(01)* | (10)* | 0(10)* | 1(01)*

我们简单分析一下这样设计的思路。由 | 分开的 4 个子表达式所代表的子语言分别是：(01)* 表示 0 开头 1 结尾的串集合，并包含空串 ε；(10)* 表示 1 开头 0 结尾的串集合，并包含空串 ε；0(10)* 表示 0 开头 0 结尾的串集合，并包含串 0；1(01)* 表示 1 开头 1 结尾的串集合，并包含串 1。

一个语言的正规表达式并非唯一的，不同的设计思路会得看似出不同形式的表达式。如，例3.1中 L 的另外两个正规表达式为：

（1）(ε | 1) (01)* (ε | 0)

（2）(ε | 0) (10)* (ε | 1)

大家可以想一想这两个正规表达式的设计思路。

例 2 设计表示如下语言 L 的正规表达式：

$L = \{ w \mid w \in \{0,1\}^*$，$|w|>0$，且 w 的第 1 位，或第 2 位，…，或第 5 位中至少有 1 个 0 $\}$

解 表示语言 L 的一个正规表达式为：

(ε | 0 | 1) (ε | 0 | 1) (ε | 0 | 1) (ε | 0 | 1) 0 (0 | 1)*

简单分析一下这一解法：在4个 (ε | 0 | 1) 的连接之后的一个0，保证了 $|w|>0$ 并且第1~5位中至少有一个0。注意，(ε | 0 | 1) 有可选项 ε，所以保证了 0 可以出现在第1~5位中的任何一位。

虽然基本正规表达式的定义中只用到三种运算符，但为了方便，常常会允许附加别的运算符。如，我们将会在3.4节看到许多派生的正规表达式运算符。以下列出其它几个最常见的关于正规表达式的运算：

- 正闭包运算，$E^+ = EE^* = E^*E$
- 任选运算，$E? = \varepsilon \mid E$
- 幂运算，$E^n = EE^{n-1}$ (n>0)， $E^0 = \varepsilon$

比如，若引入幂运算，则例3.2解答中的正规表达式也可以写作：

$(\varepsilon \mid 0 \mid 1)^4\ 0\ (0 \mid 1)^*$

2.1.2 正规语言

我们可以借助正规表达式的概念来定义正规语言。

对于字母表 $\sum$ 上的语言 R，若存在 $\sum$ 上的正规表达式 E，满足 $L(E) = R$ ，则 R 是**正规语言**。

我们也可以直接给出正规语言的定义。

字母表 $\sum$ 上的正规语言归纳定义如下：

（1）基础

- $\{\varepsilon\}$ 是正规语言；
- Φ 是正规语言，这里，Φ 表示空语言；
- 若 $a \in \sum$ ，则 $\{a\}$ 是正规语言

（2）归纳

- 若 L 和 R 是正规语言，则 $L \cup R$ 也是正规语言；
- 若 L 和 R 是正规语言，则 LR 也是正规语言；
- 若 L 是正规语言，则 L^* 也是正规语言

由正规表达式的语义，我们不难看出这两种定义是等价的。

此外，我们将会介绍，有限自动机的计算能力等价于正规表达式，所以也可以借助于有限自动机给出正规语言的定义。

2.2 有限自动机

有限自动机是一种具有有限个状态的转移系统，是最常用的语言和计算模型之一。

有限自动机的表示有五个要素。图2 是一个有限自动机 A 的转移图表示，我们以此为例来说明这五个方面：

（1） 一个非空有限状态的集合。如，有限自动机 A 包含 q_0，q_1，q_2 和 q_3 等 4 个状态，图中用圆圈表示。

（2） 一个非空的有限输入符号的集合。这相当于所代表语言的字母表。如，有限自动机 A 的输入符号集合为 $\{0, 1\}$，这些符号出现在图中的转移边标记上。

（3） 一个转移函数。在转移图表示中，通过所有转移边来表示转移函数。如，有限自动机 A 有 8 条转移边。对转移函数定义的不同限制，将导致不同类型的有限自动机定义，我们随后讨论 3 种类型的有限自动机。

（4） 一个开始状态。在转移图表示中，我们用一个单箭头并标有 *Start* 来表示。如，有限自动机 A 的开始状态为 q_0。在有些文献和书籍中，有限自动机可以有不止一个开始状态，但在本书规定只能有唯一的开始状态。

（5） 一个终态的集合。在转移图表示中，我们用双圆圈表示终态。如，有限自动机 A 有两个终态，q_0 和 q_3。

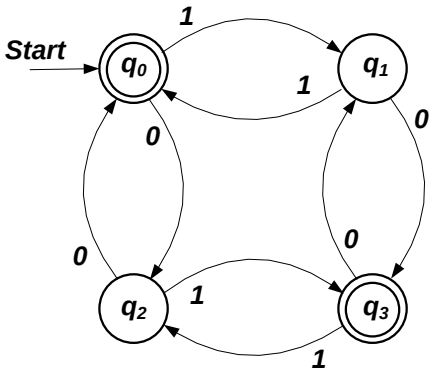


图 2 一个有限自动机的转移图

在随后的三种有限自动机的介绍中，我们将用五元组的形式来刻画者五个要素，从而给出这几种有限自动机的形式定义。

2.2.1 确定有限自动机

确定有限自动机 *DFA (deterministic finite automata)* 是一个五元组

$$A = (Q, \Sigma, \delta, q_0, F)$$

其中，A 是所讨论的 *DFA* 的名称，Q 是有限状态集合，Σ 是有限输入符号集合，δ 是转移函数，q_0 是开始状态，F 是终态的集合，并且满足：

- $Q \neq \Phi$ ；
- $\Sigma \neq \Phi$ ；
- $q_0 \in Q$ ；
- $F \subseteq Q$ ；
- $\delta: Q \times \Sigma \to Q$

应该注意到，δ 是从 $Q \times \Sigma$ 到 Q 的映射，所以，对于任何状态 $q \in Q$ 和 任何输入符号 $a \in \Sigma$，$\delta(q, a)$ 都是由定义的，并且 $\delta(q, a) \in Q$ 是唯一确定的。

也可以通过转移图和转移表的形式表示 *DFA*。图2中利用转移图表示的有限自动机 A，实际上就是一个 *DFA*。下面是这个 *DFA* 的五元组表示：

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0, q_1, q_2, q_3\}).$$

其中，转移函数 δ 定义为：

$$\delta(q_0, 0) = q_2, \quad \delta(q_0, 1) = q_1,$$
$$\delta(q_1, 0) = q_3, \quad \delta(q_1, 1) = q_0,$$
$$\delta(q_2, 0) = q_0, \quad \delta(q_2, 1) = q_3,$$
$$\delta(q_3, 0) = q_1, \quad \delta(q_3, 1) = q_2$$

图3是这个 *DFA* 的转移表。在表中，表的每一行对应一个状态，所有状态的名字在表

头的一列中给出；表的每一列对应一个输入符号，所有符号的名字在表头的一行中给出；开始状态用“→”标出；所有的终态都用“*”标出；表中的元素给出了转移函数的定义。如图3中，第 3 行第 2 列是 q_3，则表示 $\delta(q_2,1)=q_3$。

	0	1
$\rightarrow *q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
$*q_3$	q_1	q_2

图 3　一个有限自动机的转移表

现在我们来看一下 *DFA* 是如何接受输入符号串的，以便理解 *DFA* 的语言是什么？

以图2的 *DFA A* 为例。我们先来考虑输入符号串 100101。图4是 *A* 接受 100101 的过程。开始时，当前状态置为 q_0，剩余的输入符号串置为 100101。由于 $\delta(q_0,1)=q_1$，所以在读入符号 1 时， *A* 的状态转移至 q_1，剩余的输入符号串变为 00101。依次类推，直至步骤（5）时，当前状态为 q_2，剩余的输入符号串只剩下 1。结果，在读完输入符号串的所有符号后（即剩余的输入符号串变为空串 ε），到达状态 q_3。因为 q_3 是终态之一，所以 *A* 可以接受 100101。

步骤	当前状态	剩余的输入符号串
（0）	q_0	100101
（1）	q_1	00101
（2）	q_3	0101
（3）	q_1	101
（4）	q_0	01
（5）	q_2	1
（6）	q_3	ε

图 4　*DFA* 接受输入符号串的过程

若是在读完某个输入符号串后，到达的状态不是终态，那么这个 *DFA* 就不接受该符号串。试验证：图2的 *DFA A* 不接受输入符号串 10101。

一个 *DFA* 能够接受的全部输入符号串集合就是该 *DFA* 的语言。下面我们将给出 *DFA* 的语言的形式定义。为此，我们先将转移函数 $\delta: Q\times\Sigma\rightarrow Q$ 扩展为 $\delta': Q\times\Sigma^*\rightarrow Q$，使得 δ' 适合于考虑读入一个输入符号串后的状态转移。

设一个 *DFA* $A=(Q,\Sigma,\delta,q_0,F)$，其中 $\delta: Q\times\Sigma\rightarrow Q$ 为转移函数。我们定义扩展的转移函数 $\delta': Q\times\Sigma^*\rightarrow Q$ 为：对任何 $q\in Q$，

- $\delta'(q,\varepsilon)=q$ ；
- 若 $w=xa$，其中 $x\in\Sigma^*, a\in\Sigma$, 则有 $\delta'(q,w)=\delta(\delta'(q,x),a)$

例如，对于图2的 *DFA A*，我们有

$\delta'(q_0, \varepsilon) = q_0$,
$\delta'(q_0, 0) = \delta(\delta'(q_0, \varepsilon), 0) = \delta(q_0, 0) = q_2$,
$\delta'(q_0, 00) = \delta(\delta'(q_0, 0), 0) = \delta(q_2, 0) = q_0$,
$\delta'(q_0, 001) = \delta(\delta'(q_0, 00), 1) = \delta(q_0, 1) = q_1$,
$\delta'(q_0, 0010) = \delta(\delta'(q_0, 001), 0) = \delta(q_1, 0) = q_3$

在 δ' 定义的基础上，我们容易给出 ***DFA* 的语言**的定义。

设一个 *DFA* $A = (Q, \Sigma, \delta, q_0, F)$，定义 A 的语言

$$L(A) = \{ w \mid \delta'(q_0, w) \in F \}$$

例 3 设计表示如下语言 L 的一个 *DFA*：

$L = \{ w \mid w \in \{0, 1\}^*$，且 w 中 0 的个数和 1 的个数的奇偶性相同 $\}$

解 我们可以这样来考虑，所设计的 *DFA* 在读入任何一个 0、1 串后一定会到达如下 4 个状态之一：已读入偶数个 0 和偶数个的 1 的状态 q_0，已读入偶数个 0 和奇数个的 1 的状态 q_1，已读入奇数个 0 和偶数个的 1 的状态 q_1，已读入奇数个 0 和奇数个的 1 的状态 q_3。因此，我们可以设计拥有 4 个状态 q_0，q_1，q_2 和 q_3 的 *DFA*。开始时，没有读入任何符号，即已读入 0 个 0 和 0 个 1，因此可以设开始状态为 q_0。在状态 q_0 时，若读入 1，则应转移到 q_1；若读入 0，则应转移到 q_2。同样可以分析，当处于状态 q_1，q_2 和 q_3 时，分别读入 1 和 0 后应该转移到那个状态。根据题目要求，终态集合应该包括状态 q_0 和 q_3。最终，我们恰好得到一个如图2所示的 *DFA*。

所设计的 *DFA* 的语言是否恰好就是该语言呢？严格讲，应该给出形式证明。证明的方法一般要采用归纳法。但实践中所遇到的问题一般比较直观，因此，除非特别指明，我们通常都不要求给出证明。

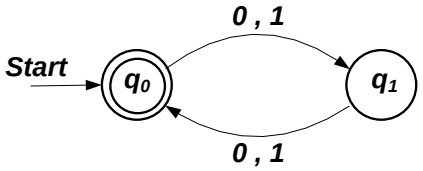


图 5　*DFA* 接受输入符号串的过程

对于同一个语言，所设计的 *DFA* 不一定唯一的，这取决于采取什么样的设计思路。例如，对于例3中的语言，我们可以从另一个角度来考虑：w 中 0 的个数和 1 的个数的奇偶性相同，实际上等价于说 w 的长度是偶数。读者可以想一下为什么？因此，我们可以设计出该语言的另一个 *DFA*，如图5所示。注意，我们这里使用了一种简化表示：若同一条转移边上的标记了 a_1，a_2，…，a_n，那么实际上是代表了分别标记了 a_1，a_2，…，a_n 的 n 条转移边。在以后的章节里我们都采用这个约定。

我们可以证明，图2和图5所表示的 *DFA* 是等价的，即它们接受同样的语言。实际上，我们将在后面的 *DFA* 的最小化一节中将会看到，可以将图2中的 *DFA* 最小化为图5中的 *DFA*。

2.2.2 非确定有限自动机

在给出非确定有限自动机的定义之前，我们先看图6中的两个非确定有限自动机的例子。可以注意到，它们与确定有限自动机的不同之处主要有以下两点：

（1）可能存在某些状态，在读入某个输入符号时，可能转移到多个不同的状态。如图6（*a*）中，在处于状态 *q* 时读入符号 1，下一个状态可能是 *q* 或 *r*。对于图6（*b*）中的状态 *p* 和输入符号 1，也是类似的情况。

（2）可能存在某些状态，当读入某个输入符号时，不能转移到任何状态。体现到转移图上，就是某些状态不存在以某个输入符号为标记的引出边。如图6（*a*）中的状态 *p* 和 *r*，以及图6（*b*）中的状态 *r*。

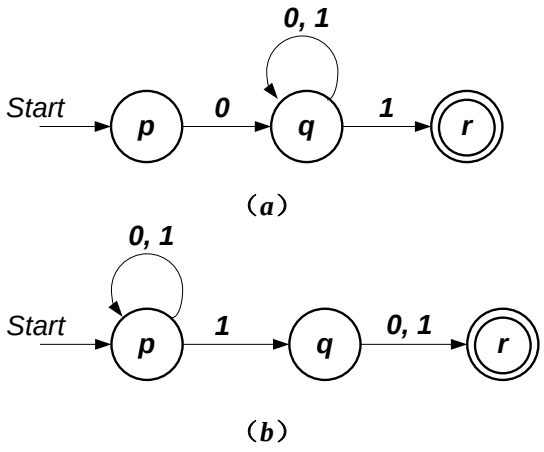


图 6　*NFA* 的例子

非确定有限自动机 *NFA (deterministic finite automata)* 是一个五元组

$$A = (Q, \Sigma, \delta, q_0, F)$$

其中，*A* 是所讨论的 *NFA* 的名称，*Q*，Σ，q_0，*F* 和 *DFA* 的定义中一样，需要满足的条件也一样，唯一的差别就是转移函数 δ 的定义：

- $\delta: Q \times \Sigma \to 2^Q$

其中，2^Q 表示 *Q* 的幂集合。这样，对于任何状态 $q \in Q$ 和 任何输入符号 $a \in \Sigma$，$\delta(q, a)$ 是 *Q* 的一个子集。当 $\delta(q, a)$ 中有多于一个状态时，在状态 *q* 读入输入符号 *a* 时的下一个状态就会有多种选择；而当 $\delta(q, a)$ 为空集合 Φ 时，在状态 *q* 读入输入符号 *a* 时不能转移到任何状态。

例如，对于图6（*a*）中的 *NFA*，我们有 $\delta(p, 0) = \{q\}$，$\delta(q, 0) = \{q\}$，$\delta(q, 1) = \{q, r\}$，以及对于其它状态和输入符号的组合，转移函数 δ 的值都为 Φ。

如同 *DFA*，*NFA* 除了可以表示成转移图的形式，也可以表示成转移表的形式。如，对于图6中的两个转移图表示的 *NFA*，图7 是相应它们的转移表形式。同样，在 *NFA* 的转移表中，每一行对应一个状态；每一列对应一个输入符号；开始状态用“→”标出；所有的终态都用“*”标出；表中的元素给出了转移函数的定义。和 *DFA* 的转移表唯一不同的是表中的元素是一个状态的集合，而不是单个状态。如图7（*a*）中，第 2 行第 2 列是 {*q, r*}，则表示 $\delta(q, 1) = \{q, r\}$。

	0	1
$\rightarrow p$	$\{q\}$	Φ
q	$\{q\}$	$\{q, r\}$
$*r$	Φ	Φ

（a）

	0	1
$\rightarrow p$	$\{p\}$	$\{p, q\}$
q	$\{r\}$	$\{r\}$
$*r$	Φ	Φ

（b）

图 7　*NFA* 的转移表

同样，为了理解 *NFA* 的语言是如何定义的，我们先来考察一下 *NFA* 是如何接受输入符号串的。

我们以状态图形式的 *NFA* 为例。一个 *NFA* 可以接受某个输入符号串，当且仅当存在从初态开始到某一个终态的路径，将该路径上每条边的所标记的符号依次连接起来恰好是这个输入符号串。例如，对于图6（b）中的 *NFA*，输入符号串为 01010。由于存在路径 $p \rightarrow p \rightarrow p \rightarrow p \rightarrow q \rightarrow r$，所以该 *NFA* 可以接受这个输入符号串

一个 *NFA* 能够接受的全部输入符号串集合就是该 *NFA* 的语言。为形式化定义 *NFA* 的语言，我们也先将转移函数 δ 进行扩展。

设一个 *NFA* $A = (Q, \Sigma, \delta, q_0, F)$，其中 $\delta: Q \times \Sigma \rightarrow 2^Q$ 为转移函数。我们定义扩展的转移函数 $\delta': Q \times \Sigma^* \rightarrow Q$ 为：对任何 $q \in Q$，

- $\delta'(q, \varepsilon) = \{q\}$；
- 若 $w = xa$，其中 $x \in \Sigma^*, a \in \Sigma$，并假设 $\delta'(q, x) = \{p_1, p_2, \ldots, p_k\}$，则有

$$\delta'(q, w) = \delta(p_1, a) \cup \delta(p_2, a) \cup \ldots \cup \delta(p_k, a)$$

例如，对于图6（b）的 *NFA* A，我们有

$\delta'(p, \varepsilon) = \{p\}$,
$\delta'(p, 0) = \delta(p, 0) = \{q\}$,
$\delta'(p, 01) = \delta(q, 1) = \{q, r\}$,
$\delta'(p, 010) = \delta(q, 0) \cup \delta(r, 0) = \{q\}$,
$\delta'(p, 0100) = \delta(q, 0) = \{q\}$,
$\delta'(p, 01001) = \delta(q, 1) = \{q, r\}$

在 δ' 定义的基础上，我们容易给出 ***NFA* 的语言**的定义。

设一个 *NFA* $A = (Q, \Sigma, \delta, q_0, F)$，定义 A 的语言

$$L(A) = \{ w \mid \delta'(q_0, w) \cap F \neq \Phi \}$$

对于图6（b）的 *NFA* A，因为 $\delta'(p, 01001) = \delta(q, 1) = \{q, r\}$，所以有 $01001 \in L(A)$。

例 4 设计一个可以识别十进制浮点数的 *NFA*，该十进制浮点数的形式为：

<正负号><整数部分><小数点><小数部分>

且满足如下说明：

（1）<正负号> 可以为‘+’或‘–’，也可以为空白；

（2）<整数部分>和<小数部分>是由数字‘0’，‘1’，…，和‘9’构成的串；

（3）<小数点> 为‘.’；

（4）<整数部分>和<小数部分>可以为空串，但不可以同时为空；

（5）<整数部分>的首字符可以是‘0’。

解 本题目的难度并不大，只是各种情况较为复杂，需要进行周密考虑。图8是一个可能的 *NFA*。我们来分析一下各个状态的意义，从中不难理解这个 *NFA* 的设计思想。初态为q_0，下一个输入符号可以是正符号，小数点以及任何一个数字。到达状态 q_1，表明首字符一定是数字；到达状态 q_2，表明首字符一定是正符号；到达状态 q_3，表明首字符一定是正符号，并且整数部分一定不为空；到达状态 q_4，表明已读入小数点，并且整数部分可能为空，下面一个输入符号只能是数字；到达状态 q_5 时，后面的输入符号只能是数字，若可遇到输入符号结束，则输入符号串被接受。

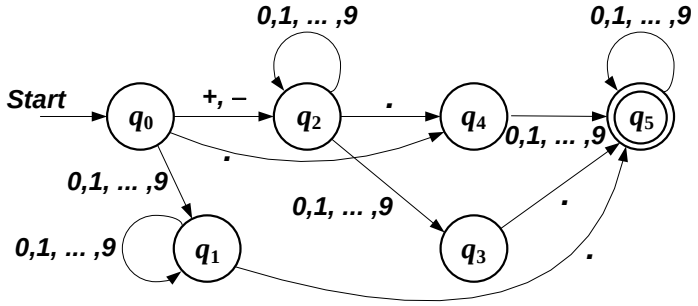


图 8　接受某种浮点数的 *NFA*

2.2.3 带ε-转移的非确定有限自动机

如果我们允许 *NFA* 的转移边被标记为 ε，称为 **ε-转移边**，表示不需要读入任何输入符号就可以转移到下一个状态，那么通常情况下可以使设计思路更加简便，设计结果更加易读。

例如，对于例4的问题，我们重新设计为图9的有限自动机。显然，这个结果更加简单明了，容易理解。我们将这种类型的有限自动机称为带ε-转移的非确定有限自动机，简称 ε-*NFA*。

下面，我们给出 ε-*NFA* 的形式定义。

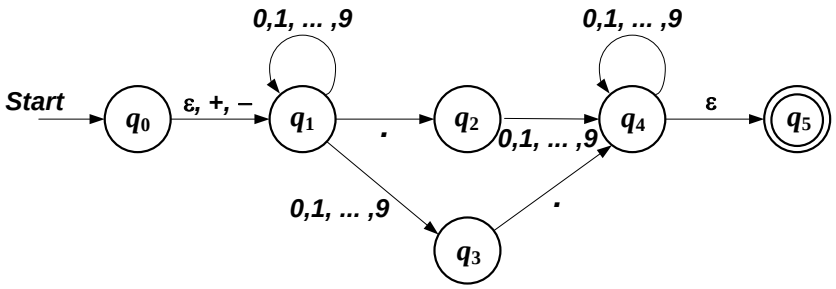


图 9　接受某种浮点数的 ε-*NFA*

带ε-转移的非确定有限自动机 ε-*NFA* 是一个五元组

$$A = (Q, \Sigma, \delta, q_0, F)$$

其中，*A* 是所讨论的 *NFA* 的名称，*Q*，Σ，q_0，*F* 和 *NFA* 的定义中一样，需要满足的条件也一样，唯一的差别就是转移函数 δ 的定义：

- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$

这样，对于任何状态 $q \in Q$，我们定义了 $\delta(q, \varepsilon)$ 也是 *Q* 的一个子集。例如，对于图9中的 ε-*NFA*，我们有 $\delta(q_0, \varepsilon) = \{q_0\}$，$\delta(q_4, \varepsilon) = \{q_5\}$，而对于其它状态则有 $\delta(q_1, \varepsilon) = \delta(q_2, \varepsilon) = \delta(q_3, \varepsilon) = \delta(q_5, \varepsilon) = \Phi$。

同样，也可以用转移表的形式表示一个ε-*NFA*。如，对于图9中转移图形式的ε-*NFA*，图10 是相应的转移表形式。相比较 *NFA* 的转移表，在 ε-*NFA* 的转移表中，有一列对应的是 ε 转移。

另外，在图10中，我们把 0，1，…，和 9 对应的列合并为一个列，因为对于任何状态，将它们作为输入符号时的后续状态集合是一致的。同样，+ 和 – 对应的列也合并为一列。

	ε	+ . –	.	0,1,⋯,9
→ q_0	$\{q_1\}$	$\{q_1\}$	Φ	Φ
q_1	Φ	Φ	$\{q_2\}$	$\{q_1, q_4\}$
q_2	Φ	Φ	Φ	$\{q_3\}$
q_3	$\{q_5\}$	Φ	Φ	$\{q_3\}$
q_4	Φ	Φ	$\{q_3\}$	Φ
∗q_5	Φ	Φ	Φ	Φ

图 10　ε-*NFA* 的转移表

与 *NFA* 的情形类似，一个 ε-*NFA* 可以接受某个输入符号串，当且仅当存在从初态开始到某一个终态的路径，将该路径上每条边的所标记的符号依次连接起来恰好是这个输入符号串（要注意到 ε 是连接运算的幺元）。例如，对于图9 中的 ε-*NFA*，下列输入符号串都是可被接受的：

- 3.14
- +.314
- –314.

为了形式化定义ε-*NFA* 的语言，我们也需要扩展转移函数 δ。为此，我们先要引入一个重要概念：ε-闭包。

直观来看，ε-*NFA* 中某个状态 *q* 的ε- 闭包，记为 *ECLOSE*(*q*)，是从 *q* 经所有的ε 路径可以到达的状态（包括*q*自身）集合，如：

- *ECLOSE*(q_0) = { q_0, q_1 }

- $ECLOSE(q_2) = \{ q_2 \}$
- $ECLOSE(q_4) = \{ q_4, q_5 \}$

设 ε-*NFA* $A = (Q, \Sigma, \delta, q_0, F)$，$q \in Q$，则 ***ECLOSE(q)*** 为满足如下条件的最小集：

- $q \in ECLOSE(q)$ ；
- 若 $p \in ECLOSE(q)$，以及 $r \in \delta(p, \varepsilon)$，则有 $r \in ECLOSE(q)$

例 5 图 11 是一个 ε-*NFA* 的转移图，试计算 *ECLOSE*(1)，*ECLOSE*(2)，*ECLOSE*(7)。

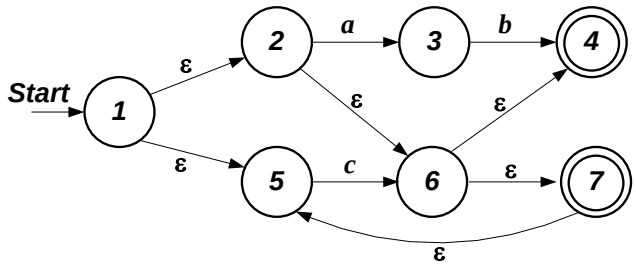


图 11　一个 ε-*NFA*

解　直接使用上述的归纳定义，可得出：

- $ECLOSE(1) = \{ 1, 2, 4, 5, 6, 7 \}$
- $ECLOSE(2) = \{ 2, 4, 5, 6, 7 \}$
- $ECLOSE(7) = \{ 5, 7 \}$

设 ε-*NFA* $A = (Q, \Sigma, \delta, q_0, F)$，其中 $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ 为转移函数。我们定义扩展的转移函数 $\delta': Q \times \Sigma^* \to Q$ 为：对任何 $q \in Q$，

- $\delta'(q, \varepsilon) = ECLOSE(q)$ ；
- 若 $w = xa$，其中 $x \in \Sigma^*, a \in \Sigma$，假设 $\delta'(q, x) = \{ p_1, p_2, \ldots, p_k \}$，并且令

  $$\delta(p_1, a) \cup \delta(p_2, a) \cup \ldots \cup \delta(p_k, a) = \{ r_1, r_2, \ldots, r_m \}$$

  则有，$\delta'(q, w) = ECLOSE(r_1) \cup ECLOSE(r_2) \cup \ldots \cup ECLOSE(r_m)$。

例 6 对于图 9 的 ε-*NFA*，计算 $\delta'(q_0, 5.6)$。

解　直接使用上述的归纳定义，可得出：

例如，对于图9的ε-*NFA*，我们有

- $\delta'(q_0, \varepsilon) = ECLOSE(q_0) = \{ q_0, q_1 \}$
- 由于，$\delta(q_0, 5) \cup \delta(q_1, 5) = \{ q_1, q_4 \}$，所以

  $$\delta'(q_0, 5) = ECLOSE(q_1) \cup ECLOSE(q_4) = \{ q_1, q_4 \}$$

- 由于，$\delta(q_1, .) \cup \delta(q_4, .) = \{ q_2, q_3 \}$，所以

$$\delta'(q_0, 5.) = ECLOSE(q_2) \cup ECLOSE(q_3) = \{ q_2, q_3, q_5 \}$$

- 由于，$\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{ q_3 \}$，所以

$$\delta'(q_0, 5.6) = ECLOSE(q_3) = \{ q_3, q_5 \}$$

在 δ' 定义的基础上，我们给出 **ε-*NFA* 的语言**的定义。

设 ε-*NFA* $A = (Q, \Sigma, \delta, q_0, F)$，定义 A 的语言

$$L(A) = \{ w \mid \delta'(q_0, w) \cap F \neq \Phi \}$$

对于图9的 *NFA* A，因为 $\delta'(q_0, 5.6) = \{ q_3, q_5 \}$，$q_5 \in F$，所以有 $5.6 \in L(A)$。

2.2.4 有限自动机的确定化

在这一小节里，我们主要介绍从 *NFA* 或 ε-*NFA* 构造等价的 *DFA* 的算法。

所谓一个 *NFA* N 和某个 *DFA* D 等价，当且仅当 $L(N) = L(D)$ 。同样，一个 ε-*NFA* E 和某个 *DFA* D' 等价，当且仅当 $L(E) = L(D')$。

我们要介绍的算法称为**子集构造法**。先来考虑 *NFA* 的情形。

对任意的 *NFA* $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$，我们定义一个 *DFA* $D = (Q_D, \Sigma, \delta_D, \{ q_0 \}, F_D)$，其中

- $Q_D = \{ S \mid S \subseteq Q_N \}$
- 对 $S \in Q_D$ 和 $a \in \Sigma$，$\delta_D(S, a) = \cup_{q \in S} \delta_N(q, a)$
- $F_D = \{S \mid S \subseteq Q_N \wedge S \cap F_N \neq \Phi \}$

可以证明（参见[1]）：$L(N) = L(D)$。

例如，对于图7（*a*）中的 *NFA*，我们可以通过子集构造法获得如图12的 *DFA*。原来的 *NFA* 中有 3 个状态 p，q，r，因而所构造的 *DFA* 有 8 个状态，每个状态可以表示成 $\{p, q, r\}$ 的一个子集。原来 *NFA* 的初态为 p，所以这一 *DFA* 的初态为 $\{p\}$。原来 *NFA* 只有一个终态 r，所以该 *DFA* 的终态共有 4 个：$\{r\}$，$\{p, r\}$，$\{q, r\}$，$\{p, q, r\}$。

		0	1
	Φ	Φ	Φ
→	$\{p\}$	$\{q\}$	Φ
	$\{q\}$	$\{q\}$	$\{q, r\}$
*	$\{r\}$	Φ	Φ
	$\{p, q\}$	$\{q\}$	$\{q, r\}$
*	$\{p, r\}$	$\{q\}$	Φ
*	$\{q, r\}$	$\{q\}$	$\{q, r\}$
*	$\{p, q, r\}$	$\{q\}$	$\{q, r\}$

图 12　子集构造：从 *NFA* 到 *DFA*

读者可能已经发现，图12所表示的 *DFA* 中，有些状态从初态 {*p*} 是不可到达。在实际使用子集构造法时，我们可以不把所有的子集都包括进来，而是在构造过程中逐步加入由初态可达的状态。这样，我们可以得到等价于图7（*a*）中 *NFA* 的一个更加简化的 *DFA*，如图13所示。

		0	1
→	$\{p\}$	$\{q\}$	Φ
	$\{q\}$	$\{q\}$	$\{q, r\}$
*	$\{q, r\}$	$\{q\}$	$\{q, r\}$
	Φ	Φ	Φ

图 13　简化的子集构造：从 *NFA* 到 *DFA*

对于 ε-*NFA* 的确定化，方法是类似的，只是要注意，针对 ε-*NFA* 的子集构造法需要考虑到每个状态都应该是求极大 ε-闭包之后的结果。一个状态集合 *S* 的**极大 ε-闭包** *ECLOSE*(*S*) 为满足如下条件的最小集：

- 若 $q \in S$，则 $q \in ECLOSE(S)$ ；
- 若 $p \in ECLOSE(S)$，以及 $r \in \delta(p,\varepsilon)$，则有 $r \in ECLOSE(S)$

对任意的 ε-*NFA* $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$，我们定义一个 *DFA* $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$，其中

- $Q_D = \{ S \mid S \subseteq Q_E \wedge S = ECLOSE(S)\}$
- $q_D = ECLOSE(q_0)$
- $F_D = \{S \mid S \in Q_D \wedge S \cap F_E \neq \Phi\}$
- 对 $S \in Q_D$ 和 $a \in \Sigma$，令 $S = \{ p_1, p_2, \ldots, p_k \}$，并设 ，

  $$\delta_E(p_1, a) \cup \delta_E(p_2, a) \cup \ldots \cup \delta_E(p_k, a) = \{ r_1, r_2, \ldots, r_m \}$$

  则有，$\delta_D(S, a) = ECLOSE(r_1) \cup ECLOSE(r_2) \cup \ldots \cup ECLOSE(r_m)$。

可以证明（参见[1]）：$L(E) = L(D)$。

同样，在实际使用这一子集构造法时，我们也可以不把所有的子集都包括进来。

对于图11 中的 ε-*NFA*，我们可以通过子集构造法获得如图14的 *DFA*。

	a	*b*	*c*
→ ∗{1,2,4,5,6,7}	{3}	*Φ*	{4,5,6,7}
{3}	*Φ*	{4}	*Φ*
Φ	*Φ*	*Φ*	*Φ*
∗ {4,5,6,7}	*Φ*	*Φ*	{4,5,6,7}
{4}	*Φ*	*Φ*	*Φ*

图 14　简化的子集构造：从 ε-*NFA* 到 *DFA*

2.2.5 DFA 的最小化

前面我们提到过，图 2 和图 5 的 *DFA* 是等价的，即它们的语言是相同的。后者的状态数目比前者少，那么它是否是所有与它们等价的 *DFA* 中状态数目最少的呢？这一小节里，我们将介绍一种 *DFA* 的最小化算法，可以为任何 *DFA* 给出一个与之等价的拥有最小状态数目的 *DFA*。

首先，我们来讨论一下 *DFA* 中哪些状态是可以合并的。为此，我们引入以下概念：

设 *DFA* $A = (Q, \Sigma, \delta, q_0, F)$，定义 上的一个二元关系 R 为：对任何 $p, q \in Q$，

$$pRq \text{ 当且仅当 } \forall w \in \Sigma^*. (\delta'(p,w) \in F \leftrightarrow \delta'(q,w) \in F).$$

若 pRq，则称 p 和 q 是**不可区分的**；否则就是**可区分的**。直观地说，p 和 q 是不可区分的，就是说没有任何输入符号串 w，使得 $\delta'(p,w)$ 和 $\delta'(q,w)$ 一个是终态而另一个是非终态；相反，p 和 q 是可区分的，则一定存在输入符号串 w，使得 $\delta'(p,w)$ 和 $\delta'(q,w)$ 一个是终态而另一个是非终态。

容易证明，R 是 Q 上的一个等价关系，即满足自反性，对称性和传递性。*DFA* 最小化算法的一个核心步骤就是先找出 R 的所有等价类，得到 Q 的一个划分，然后将每个等价类的状态合并成一个。下面我们介绍一种找出 R 的所有等价类的算法，称为**填表算法**。我们以图15中的 *DFA* 为例配合说明。

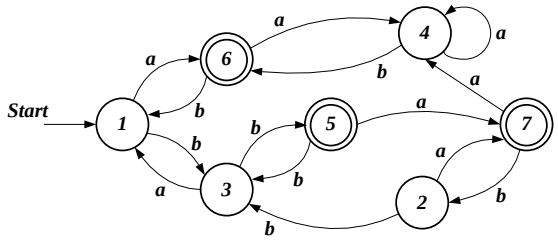


图 15　*DFA* 最小化的例子

填表算法的总体思路是将 Q 中可以区别的全部状态偶对先后标出来，然后找出所有等价类。我们可以基于如下归纳过程标记出全部可区别的状态偶对：

（1）基础。如果 p 为终态，而 q 为非终态，则 p 和 q 标记为可区分的。这是因为，

终态和非终态之间总是可被空串 ε 区分。

（2）归纳。设 *p* 和 *q* 已标记为可区分的，如果状态 *r* 和 *s* 通过某个输入符号 *a* 可分别转移到 *p* 和 *q*，即有 $\delta(r,a)=p$ 和 $\delta(s,a)=q$，则 *r* 和 *s* 也标记为可区分的。这是因为，*p* 和 *q* 可被符号串 *w* 区分，即 $\delta'(p,w)$ 和 $\delta'(q,w)$ 一个是终态而另一个是非终态，那么 *r* 和 *s* 就可被符号串 *aw* 区分，即 $\delta'(r, aw)$ 和 $\delta'(s, aw)$ 一个是终态而另一个是非终态。

具体实现时，可以采用一种填表过程。这里提到的表，是指 *R* 的关系表。因为 *R* 是等价关系，所以可以只考虑关系表的一部分。例如，图15中的 *DFA* 有 7 个状态 1，2，3，4，5，6，7，那么关系表可以表示为图15中的形式。完整的关系表应该是一个7×7的方阵，但又如关系 *R* 满足自反性和传递性，所以只用到方阵的下三角部分，并且去掉了对角线部分。

在图15的 *DFA* 中，状态 1，2，3，4 是非终态，状态 5，6，7 是终态。根据上述归纳基础，我们先将所有非终态和终态两两之间标记为可区分的。如图15（*a*）所示。

接下来，我们在图15（*a*）的基础上，根据上述归纳步骤分析所有尚未标记的状态偶对，如果发现可区分的偶对，就进行标记。如，考虑偶对（1, 3）。由于在图15的 *DFA* 中有 $\delta(1,a)=6$ 和 $\delta(3,a)=1$，而（6, 1）在图15（*a*）中已被标记，所以我们因该将（1, 3）进行标记。考虑图15（*a*）所有未标记的状态偶对之后，我们可得到图15（*b*）。

在图15（*b*）的基础上重复这一过程，我们可得到图15（*c*）。

在图15（*c*）的基础上再重复这一过程，我们会发现已经没有新的可被标记的偶对。此时，填表过程就终止了。

根据图15（*c*），我们得到划分的结果为：{1,2}, {3}, {4}, {5}, {6,7}。其中的每一个状态子集代表一个等价类，每一个等价类中的状态两两之间是不可区分的，取自两个不同等价类的状态之间都是可区分的。

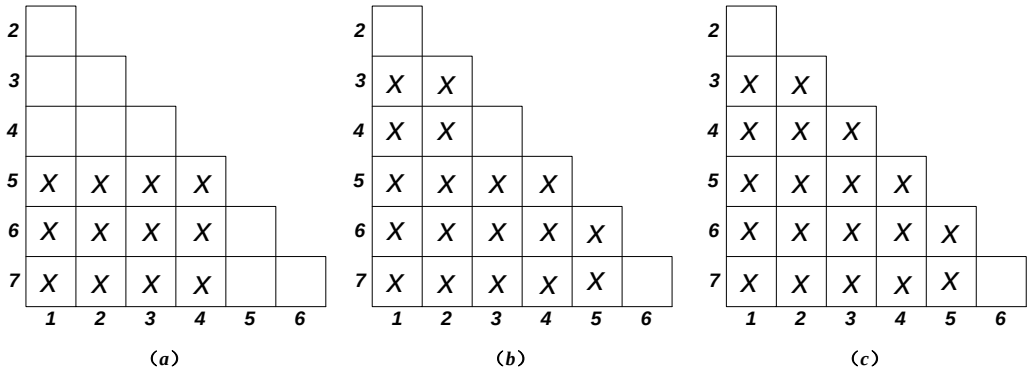


图 15　填表算法

在填表算法基础上，我们可得出一种 *DFA* 的最小化算法，步骤如下：

（1）删除所有从开始状态不可到达的状态及与其相关的边，设所得到的 *DFA* 为

$$A = (Q, \Sigma, \delta, q_0, F)$$

（2）使用填表算法找出状态集合 Q 的一个划分，每一等价类中的状态相互之间不可区分，而不同等价类中的状态之间都是可区分的。我们把包含状态 q 的等价类表示为 $[q]$。

（3）构造与 A 等价的 DFA $B = (Q_B, \Sigma, \delta_B, [q_0], F_B)$，其中

- $Q_B = \{[q] \mid q \in Q\}$
- $F_B = \{[q] \mid q \in F\}$
- $\delta_B([q], a) = [\delta(q, a)]$

例如，对于图 15 的 DFA，我们应用上述最小化算法。首先，所有状态都是可达的，所以可以直接应用填表算法。根据前面的结果，所得到的等价类包括 {1, 2}, {3}, {4}, {5}, {6, 7}，分别表示为 [1], [3], [4], [5], [6]。注意，{1, 2} 也可以表示为 [2]，{6, 7} 也可以表示为 [7]。

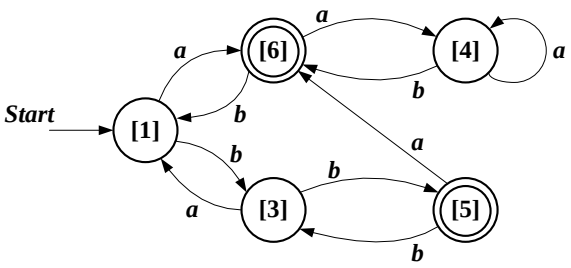


图 16　最小化的 *DFA*

由上面的步骤（3），我们得出**最小化的 *DFA*** 如图 16 所示。

读者可以作一个练习，应用以上步骤将图 2 的 DFA 最小化。将结果与图 5 的 DFA 进行比较，会发现二者只有状态名称的差异。

可以证明（参考[1]），通过上述步骤所得到的 DFA 一定是最小的，即等价于原来的 DFA 的同时所包含的状态最少。

值得注意的是，在本书中，我们允许将 DFA 中的死状态也删除掉，这样可以在实际应用中使 DFA 得到进一步的简化。这里提到的**死状态**，是指那些不能到达任何终态的状态。

2.2.6 从正规表达式到有限自动机

在这一小节，我们介绍一种从正规表达式构造等价的 $\varepsilon - NFA$ 的算法。

这个算法的存在说明了有限自动机可以模拟正规表达式。另外，也存在从有限自动机构造正规表达式的算法（参见[1]），说明正规表达式也可以模拟有限自动机。也就是说，有限自动机与正规表达式这两类模型可相互模拟，因而有限自动机模型可以处理的语言同样是正规语言。

在下面的算法中，对于给定的正规表达式 R，我们可以通过归纳于 R 的结构来构造等价于 R 的满足如下条件的 ε- NFA：

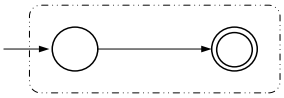
- 恰好一个终态；
- 没有转移边进入初态；

- 没有转移边离开终态。

归纳构造过程如下：

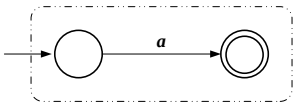
（1）基础。

若 $R=\varepsilon$ ，则构造为



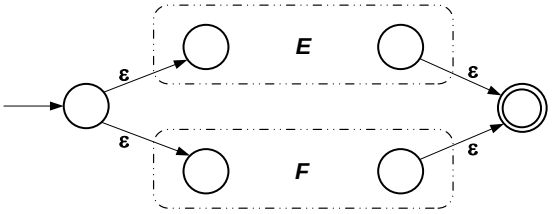
若 $R=\phi$ ，则构造为

若 $R=a$ ，则构造为

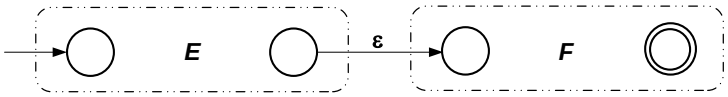


（2）归纳。设已经得到正规表达式 E 和 F 的转移图，在下面的图中分别用虚线框表示，只画出唯一的初态（左边）和唯一的终态（右边）。

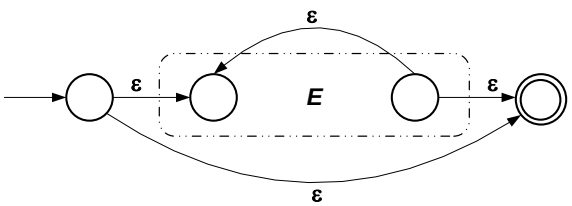
若 $R=E\mid F$ ，则构造为



若 $R=EF$ ，则构造为



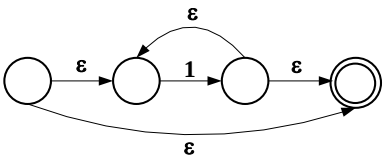
若 $R=E^*$ ，则构造为



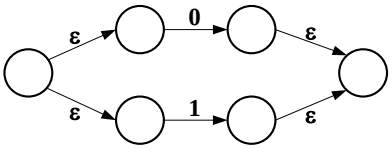
例 7 设有正规表达式 1*0(0 | 1)*，构造与之等价的 *ε-NFA*。

解 使用上述归纳构造过程来构造。为简洁，我们省略了很直接的几步。

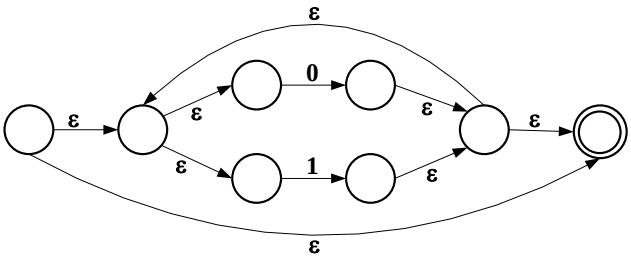
对于 1*，构造 *ε-NFA* 为



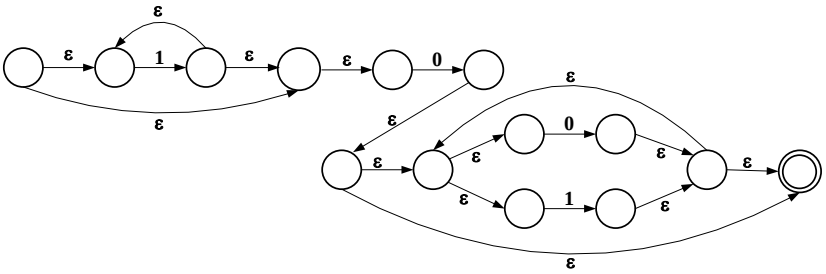
对于 (0 | 1)，构造 *ε-NFA* 为



对于 (0 | 1)*，构造 *ε-NFA* 为



最后，对于 1*0(0 | 1)*，构造 *ε-NFA* 为



3. 上下文无关语言及其描述

3.1 上下文无关文法

3.1.1 上下文无关文法的基本概念

上下文无关文法有四个基本要素:

- **终结符**的有限集合，相当于输入字母表。
- **非终结符**的有限集合，每个非终结符可以看作是一个变量符号。

- **开始符号**，是一个特殊的非终结符。
- **产生式**的有限集合，每个产生式形如：<*head*> → <*body*>。其中<*head*>称为产生式的左部，只含有一个非终结符；<*body*> 称为产生式的右部，是由终结符和非终结做构成的有限串。

如，一个简单的表达式文法 G_{exp} 包含如下产生式：

$E \rightarrow EOE$
$E \rightarrow (E)$
$E \rightarrow v$
$E \rightarrow d$
$O \rightarrow +$
$O \rightarrow *$

G_{exp} 中包含两个非终结符 E 和 O，其中 E 是开始符号。G_{exp} 的终结符有 6 个，(，)，v，d，＋ 和 ∗，分别代表左括号，右括号，变量标识符号，常量标识符号，加号和乘号等，构成该文法的输入字母表。

形式上，上下文无关文法 G 可表示为一个四元组 $G = (V_N, V_T, P, S)$。其中，V_N 表示非终结符集合，V_T 表示终结符集合，P 表示产生式集合，而 S 表示开始符号。V_N ，V_T ，P，S 满足以下条件：

- $V_N \cap V_T = \Phi$
- $S \in V_N$
- P 中的每一产生式形如 $A \rightarrow \alpha$，其中 $A \in V_N$ ，$\alpha \in (V_N \cup V_T)^*$

如，上述文法 G_{exp} 可表示为四元组 $G_{exp} = (\{E, O\}, \{ (,), +, *, v, d \}, P, E)$，其中 P 表示以上 6 个产生式的集合。

为简化表示，常将形如 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_n$ 的产生式集合简写为

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

如，可将 G_{exp} 的产生式集合简写为

$E \rightarrow (E) \mid v \mid d$
$O \rightarrow + \mid *$

为方便，在很多时候，我们只需给出产生式就可以表示一个文法。此时，需要指明开始符号。本书中，我们用 $G[S]$ 表示开始符号为 S 的文法 G。从 G 的产生式集合中，容易知道哪些符号是非终结符，而哪些符号是终结符。如，我们可以通过上述文法文法 G_{exp} 的产生式来表示这个简单的表达式文法，可记为 $G_{exp}[E]$。

与上下文无关文法类似，一般文法也可以表示成四元组的形式。根据对产生式形式的不同要求，*Chomsky* 将文法分为四种类型。与程序语言的设计和实现关系最密切的是上下文无关文法，本书将不涉及任何其它类型的文法。如果不特别指明，本文后续部分所提到的文法均指上下文无关文法。

3.1.2 归约与推导

上下文无关文法可以用来对程序语言的语法进行描述，并在此基础上进行语法分析。语法分析的核心问题是句型分析，即：对任意上下文无关文法 $G=(V_N, V_T, P, S)$ 和任意 $w \in V_T^*$，是否有 $w \in L(G)$？ 这里，$L(G)$ 是文法 G 所表示的语言。为解释上下文无关文法的语言以及语法分析的基本思想，我们先介绍两个重要概念：归约与推导。

归约与推导可用于推理某个字（串）是否属于文法所定义的语言。进行这种推理时，归约是自下而上的过程，而推导则是自上而下的过程。

归约过程的每一步是将产生式的右部（<*body*>）替换为产生式的左部（<*head*>），而**推导**过程的每一步是将产生式的左部（<*head*>）替换为产生式的右部（<*body*>）。

例如，3.1.1节中的文法 $G_{exp}[E]$ 有如下产生式：

（1）$E \rightarrow EOE$
（2）$E \rightarrow (E)$
（3）$E \rightarrow v$
（4）$E \rightarrow d$
（5）$O \rightarrow +$
（6）$O \rightarrow *$

对于字（串） $v*(v+d)$ 的一个归约过程为

$v*(v+d)$ // 使用产生式（4）
$\Leftarrow v*(v+E)$ // 使用产生式（6）
$\Leftarrow vO(v+E)$ // 使用产生式（3）
$\Leftarrow vO(E+E)$ // 使用产生式（5）
$\Leftarrow vO(EOE)$ // 使用产生式（1）
$\Leftarrow vO(E)$ // 使用产生式（2）
$\Leftarrow vOE$ // 使用产生式（3）
$\Leftarrow EOE$ // 使用产生式（1）
$\Leftarrow E$

这里，我们用“$\Leftarrow$”表示一步归约关系。在每一步归约的右面，给出了所用到的产生式。

相应地，对于同一个字（串） $v*(v+d)$ 的一个推导过程为

E // 使用产生式（1）
$\Rightarrow EOE$ // 使用产生式（6）
$\Rightarrow E*E$ // 使用产生式（2）
$\Rightarrow E*(E)$ // 使用产生式（3）
$\Rightarrow v*(E)$ // 使用产生式（1）
$\Rightarrow v*(EOE)$ // 使用产生式（5）
$\Rightarrow v*(E+E)$ // 使用产生式（3）
$\Rightarrow v*(v+E)$ // 使用产生式（4）
$\Rightarrow v*(v+d)$

这里，我们用“$\Rightarrow$”表示一步推导关系。在每一步推导的右面，给出了所用到的产生式。

若推导过程的每一步总是替换出现在最左边的非终结符，则这样的推导过程称为**最左推导**。如，对于上下文无关文法 G_{exp} ，下面是关于 $v*(v+d)$ 的一个最左推导：

$E \Rightarrow EOE \Rightarrow vOE \Rightarrow v*E \Rightarrow v*(E) \Rightarrow v*(EOE) \Rightarrow v*(vOE) \Rightarrow v*(v+E) \Rightarrow v*(v+d)$

类似地，若推导过程的每一步总是替换出现在最右边的非终结符，则这样的推导称为**最右推导**（或称为**规范推导**）。如，对于上下文无关文法 G_{exp} ，下面是关于 $v*(v+d)$ 的一个最右推导：

$E \Rightarrow EOE \Rightarrow EO(E) \Rightarrow EO(EOE) \Rightarrow EO(EOd) \Rightarrow EO(E+d) \Rightarrow EO(v+d) \Rightarrow E*(v+d) \Rightarrow v*(v+d)$

虽然也可以引入最左归约和最右归约的概念，但二者的讨论比较复杂。实际上，有了最左和最右推导的概念，对于语法分析过程的理解来说已经足够了。

3.1.3 上下文无关语言

本节定义上下文无关文法的语言，然后给出上下文无关语言的概念。

首先给出上述推导过程的形式定义。

设上下文无关文法 $G = (V_N, V_T, P, S)$。若 $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$，以及 $A \to \gamma \in P$，则定义 $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$，称 $\alpha A \beta$ 可**直接推导**或**一步推导**出 $\alpha\gamma\beta$ 。若 G 在上下文中是明确的，则可简记为 $\alpha A \beta \Rightarrow \alpha\gamma\beta$。

上述**直接推导关系** $\Rightarrow$ 的自反传递闭包 $\Rightarrow^*$ 可归纳定义如下：

（1） 对任何 $\alpha \in (V_N \cup V_T)^*$，满足 $\alpha \Rightarrow^* \alpha$ ；

（2） 设 $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$，若 $\alpha \Rightarrow^* \beta$ ，$\beta \Rightarrow \gamma$ 成立，则有 $\alpha \Rightarrow^* \gamma$ ；

（3） 关系 $\Rightarrow^*$ 只能通过（1）、（2）建立。

推导关系 $\Rightarrow^*$ 对应**多步推导**（包括 0 步）。若是不包含 0 步的多步推导，则将其记为 $\Rightarrow^+$。

现在，我们可以定义上下文无关文法 $G = (V_N, V_T, P, S)$ 的语言为

$$L(G) = \{ w \mid w \in V_T^* \wedge S \Rightarrow^* w \}$$

如果语言 L 是某个上下文无关文法 G 的语言，即 $L(G) = L$，则称 L 为**上下文无关语言**。

虽然判定任何一个语言是否上下文无关语言的算法是不存在的，但在实践中，我们常会遇到为给定语言设计一个文法的需求。如果可以找到一个上下文无关文法可以产生该语言，那么它就是上下文无关语言。

例 8 考虑如下语言：

$$L = \{ a^n b^{2n} c^m \mid n, m \geq 0 \}$$

试通过给出 L 的文法来说明 L 是上下文无关语言。

解 根据上下文无关文法的特点，要产生形如 $a^n b^{2n} c^m$ 的串，可以分别产生形如 $a^n b^{2n}$ 和形如 c^m 的串。对于 L，存在一个由以下产生式定义的上下文无关文法 $G[S]$：

$$S \to AB$$
$$A \to \varepsilon \mid aAbb$$
$$B \to \varepsilon \mid cB$$

所以，L 为上下文无关语言。

所设计的文法是否就是该语言的文法呢？严格讲，应该给出形式证明。证明的方法一般要采用归纳法。由于在实践中所遇到的文法设计问题往往比较直观，因此，若不是特别指明，我们通常会忽略这一点。

例 9 构造产生语言 $L = \{ a^m b^n c^p d^q \mid m+n = p+q \}$ 的上下文无关文法。

解 可以通过“剥洋葱”的办法考虑：

（1）对 L 中任何一个形如 $a^i w d^i$ 的形式，我们可以在两边剥掉相同个数的 a 和 d 直到不能再剥为止，剩下的 w 也一定是 L 中的串。

（2）现在剩下的 w 要么是 $a^i u c^i$ 的形式，要么是 $b^i v d^i$ 的形式。如果是前一种形式，那么我们在两边剥掉相同个数的 a 和 c 直到不能再剥为止；如果是后一种形式，那么我们在两边剥掉相同个数的 b 和 d 直到不能再剥为止。这两种情况最终剩下的 u 或者 v 都仍然是 L 中的串，而且只可能是 $b^k c^k$ 的形式。（想想为什么？）

（3）显然，形如 $b^k c^k$ 的串可以通过规则 $C \to \varepsilon \mid bCc$ 产生。

（4）现在我们把这个“剥”的过程倒过来，把字母“包”回去，就可以获得 L 的一个上下文无关文法 $G[S]$：

$$
\begin{aligned}
&S \to A \mid B \mid aSd \\
&A \to C \mid aAc \\
&B \to C \mid bBd \\
&C \to \varepsilon \mid bCc
\end{aligned}
$$

例 10 已知 L 是 $\{a, b\}$ 上的上下文无关语言，$\alpha \in L$ 当且仅当 α 不具有 ww 的形式（其中，$w \in \{a, b\}^*$）。试给出 L 的一个上下文无关文法。

解 首先，奇数长度的串都不是 ww 的形式，这个容易处理。其次，偶数长度的串分成两个长度相等的段，不妨设每段的长度为 n；因为不具有 ww 的形式，所以存在 $1 \le i \le n$，该串第 i 位和第 $n+i$ 位不同；分别以第 i 位和第 $n+i$ 位为中心将该串重新划分为两段，长度分别为 $2(i-1)+1$ 和 $2(n-i)+1$；这两段的中心不同，而围绕中心的其它位可以任意。

根据以上分析过程，如下产生式构成了 L 的一个上下文无关文法 $G[S]$：

$$
\begin{aligned}
&S \to E \mid O \\
&O \to a \mid b \mid COC \\
&E \to AB \mid BA \\
&A \to CAC \mid a \\
&B \to CBC \mid b \\
&C \to a \mid b
\end{aligned}
$$

其中，非终结符 O 负责产生奇数长度的串；非终结符 E 负责产生偶数长度的串；非终结符 A 负责产生以 a 为中心的串；非终结符 B 负责产生以 b 为中心的串。

3.1.4 句型，句子与分析树

继上面的归约和推导等概念之后，本节我们继续介绍在理解基于上下文无关文法的语法分析时所用到的一些基本术语。

设上下文无关文法 $G=(V_N, V_T, P, S)$。称 $\alpha \in (V_N \cup V_T)^*$ 为 G 的一个**句型**，当且仅当 $S \Rightarrow^* \alpha$.。

若存在从 S 到 α 的一个一个最左推导，则 α 是一个**左句型**。若存在从 S 到 α 的一个最右推导，则 α 是一个**右句型**（或称为**规范句型**）。

若句型 $\alpha \in V_T^*$，则称 α 为一个**句子**。

这样，也可以将文法 G 的语言 $L(G)$ 看作是 G 中所有句子的集合。

下面我们介绍描述语法分析过程和结果的另一重要概念 —— 分析树。

从 3.1.2 节不难看出，归约过程相当于自下而上构造了一棵树，而推导过程相当于自上而下构造了一棵树。对于上下文无关文法 $G=(V_N, V_T, P, S)$，一棵**分析树**（或**语法分析树**）是满足下列条件的有向树：

- 树中的每个内部结点由一个非终结符标记。
- 树中的每个叶结点或由一个非终结符，或由一个终结符，或由 ε 来标记；但标记为 ε 时，它必是其父结点唯一的孩子。
- 如果一个内部结点标记为 A，而其孩子从左至右分别标记为 X_1，X_2，…，X_k，则一定有：$A \to X_1X_2...X_k$ 是 P 中的一个产生式。注意：只有 $k=1$ 时上述 X_i 才有可能为 ε，此时结点 A 只有唯一的孩子，且 $A \to \varepsilon$ 是 P 中的一个产生式。

将分析树的每个叶结点按照从左至右的次序连接起来，得到一个 $(V_N \cup V_T)^*$ 中的字符串，称为该语法树的**果实**。

文法 $G[S]$ 的每个句型都是某个根结点为 S 的分析树的果实；这些分析树中，有些树的果实为句子，由这些分析树的果实构成的集合即为 G 的语言。

例如，对于 3.1.1 节中的文法 $G_{exp}[E]$，图 17 表示以 E 为根结点的一棵分析树，它的果实为 $v*(v+d)$，这一果实也是一个句子。

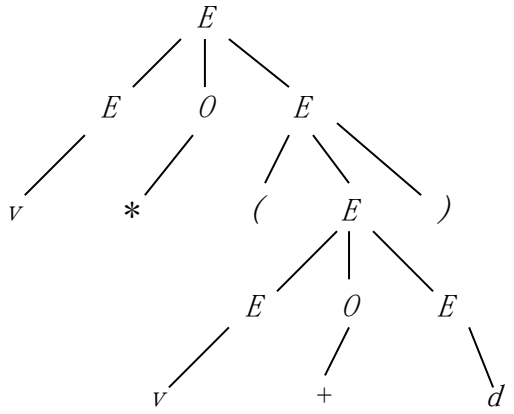


图 17　分析树

3.1.5 归约、推导与分析树之间关系

设上下文无关文法 $G = (V_N, V_T, P, S)$，可以证明（可参考[1]），以下命题是相互等价的：

- 字符串 $w \in V_T^*$ 可以归约到非终结符 A；
- $A \Rightarrow^* w$；
- $A \Rightarrow_{lm}^* w$；
- $A \Rightarrow_{rm}^* w$；
- 存在一棵根结点为 A 的分析树，其果实为 w。

其中，$\Rightarrow_{lm}^*$ 和 $\Rightarrow_{rm}^*$ 分别表示最左推导关系和最右推导关系。可以参考 3.1.3 的方法给出 $\Rightarrow_{lm}^*$ 和 $\Rightarrow_{rm}^*$ 的定义，这里不再赘述。

3.1.6 文法的二义性

文法的二义性在程序语言的语法设计和分析中是需要重点关注的问题之一。

上下文无关文法 $G = (V_N, V_T, P, S)$ 为**二义的**，如果对某个 $w \in V_T^*$，存在两棵不同的分析树，它们的根结点都为开始符号 S，果实都为 w。反之，如果对每一 $w \in V_T^*$，至多存在一棵这样的分析树，则 G 为**无二义的**。

例如，考虑3.1.1节中的文法 $G_{exp}[E]$，对于终结符串 $v + v * d$，存在两棵不同的分析树。如图18所示，这两棵分析树的根结点都为开始符号 E，果实都为 $v + v * d$。

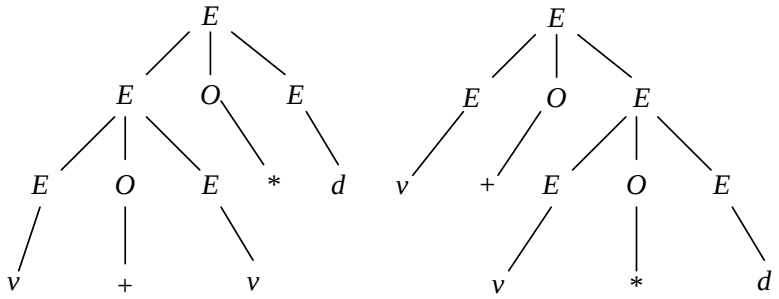


图 18　同一终结符串的不同分析树

文法的二义性可以有另一种定义方法：上下文无关文法 $G = (V_N, V_T, P, S)$ 为二义的，如果对于某个 $w \in V_T^*$，存在两个不同的从开始符号 S 到 w 的最左推导。

例 11 给出下列语言 L 的二义文法和非二义文法各一个：

$$L = \{ a^n b^m \mid m \geq n \geq 0 \}$$

解 如果不考虑二义性，这个语言的上下文无关文法是比较容易设计的。若考虑二义性，需

要确保每一步推导或归约的确定性。一般的设计思路是在产生一个 *a* 时，同时要产生一个与之配对的 *b*；多出来的 *b* 什么时候产生、如何产生，将是推导步或归约步是否确定的关键；如果能够保证 *a* 和 *b* 的配对方式唯一，则能够保证推导步或归约步的确定性（如图 19 中前两种配对方式，但不限于此），对应的文法即为非二义的；若是配对方式不唯一，（如图 19 中后一种配对方式，一个 *a* 可能与不同 *b* 的配对），对应的文法即为二义的。

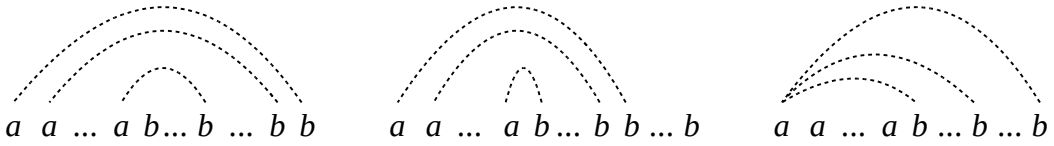


图 19　归约时不同的配对方式

基于以上分析，可以给出 *L* 的一个二义文法为：

$S \to aSb \mid Sb \mid \varepsilon$

L 的一个非二义文法为（按照第一种配对方式）：

$S \to aSb \mid B$
$B \to Bb \mid \varepsilon$

L 的另一个非二义文法为（按照第二种配对方式）：

$S \to AB$
$A \to aAb \mid \varepsilon$
$B \to Bb \mid \varepsilon$

对于任意给定的上下文无关文法，能否确定该文法是二义的，还是无二义的呢？答案是否定的。上下文无关文法是否为二义的问题是不可判定的，即不存在解决该问题的通用算法。关于这个结论的证明，有兴趣的读者可参考[1]。

在语法设计和分析中，许多情况下需要对原文法进行改造，以消除二义性。遗憾的是，下面就可看到，没有通用的办法可以消除上下文无关文法的二义性。

先来看一个关于语言二义性的概念：如果上下文无关语言 *L* 的所有文法都是二义的，则称 *L* 是**固有（天生）二义的**。

例如，上下文无关语言

$$L = \{ a^n b^n c^m d^m \mid n \geq 1，\ m \geq 1 \} \cup \{ a^n b^m c^m d^n \mid n \geq 1，\ m \geq 1 \}$$

是固有二义的。以下是 *L* 的一个上下文无关文法 *G*[*S*]：

$S \to AB \mid C$
$A \to aAb \mid ab$
$B \to cBd \mid cd$
$C \to aCd \mid aDd$

$$D \rightarrow bDc \mid bc$$

该文法是二义的。显然，对于任何形如 $a^k b^k c^k d^k$（$k \geq 1$）的符号串，都存在两棵不同的分析树。证明 L 不存在无二义的上下文无关文法是比较复杂的，有兴趣者请参阅相关文献。

在此例的基础上，容易得出结论：没有通用的办法可以消除上下文无关文法的二义性。否则，对于上述语言 L 就会存在一个无二义的上下文无关文法。

如果所设计的程序设计语言文法是二义的，那么在实现其编译程序时必须要解决好这一问题，要保证程序单元不出现有二义的分析结果。

解决二义性问题的基本方法有两类。一类是允许文法有二义性（某些情况下为了简洁、明了，更倾向于使用二义文法），而对分析过程规定某些规则，在出现二义时根据这些规则来确定哪棵分析树是正确的。另一类方法是对二义文法进行某种变换，构造一个等价的无二义文法。两类方法各有利弊，这里不作讨论。本小节只关心后者，即将二义文法设法变换为无二义文法。

如上所述，没有通用的办法可以消除上下文无关文法的二义性。但在编译程序构造的实践中形成了一些比较实用的技术，可以用于将某些特定的二义文法变换为无二义文法。这里主要列举三种变换技术：（1）优先性级联；（2）规定左结合或右结合；（3）最近嵌套匹配。

还是以文法 $G_{exp} = (\{E, O\}, \{ (,), +, *, v, d \}, P, E)$ 为例。为方便，将其产生式集合重新列出：

$$E \rightarrow E\ O\ E \mid (\ E\ ) \mid v \mid d$$
$$O \rightarrow + \mid *$$

从图 18 可以看到，对于终结符串 $v+v*d$，分析树不止一棵。为消除此类二义性，可以通过规定产生式 $O \rightarrow + \mid *$ 中终结符 “+”和“*”的优先级来改造原文法，以消去二义性。比如规定“*”优先于“+”，引入不同的非终结符 M 和 A 分别体现这两个优先级别。增加另一个非终结符 T 来辅助实现这一优先关系。上述文法可变换为：

$$E \rightarrow E\ A\ E \mid (\ E\ ) \mid v \mid d \mid T$$
$$T \rightarrow T\ M\ T \mid (\ E\ ) \mid v \mid d$$
$$A \rightarrow +$$
$$M \rightarrow *$$

当然，还可以简写为：

$$E \rightarrow E\ A\ E \mid T$$
$$T \rightarrow T\ M\ T \mid (\ E\ ) \mid v \mid d$$
$$A \rightarrow +$$
$$M \rightarrow *$$

显然，还可以将非终结符 A 和 M 省去，直接替换为 “+”，与“*”，这样可将后两个产生式去掉。但如果在同一优先级别里不只有一个符号时，比如与“+”同一优先级的符号还有“–”，与“*”同一优先级的符号还有“/”，A 和 M 的存在就显得有必要了。

对于该文法，串 $v+v*d$ 存在唯一的分析树，见图 20。

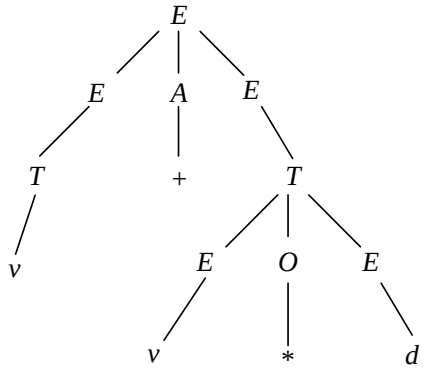


图 20　体现终结符优先级的分析树

读者不难总结出这种变换的技术特点，即优先性级联方法。

然而，经过这样的变换，上述文法还是一个二义文法。比如，终结符串 *v*+*v*+*d* 可以拥有如图 21 的两棵不同的分析树。

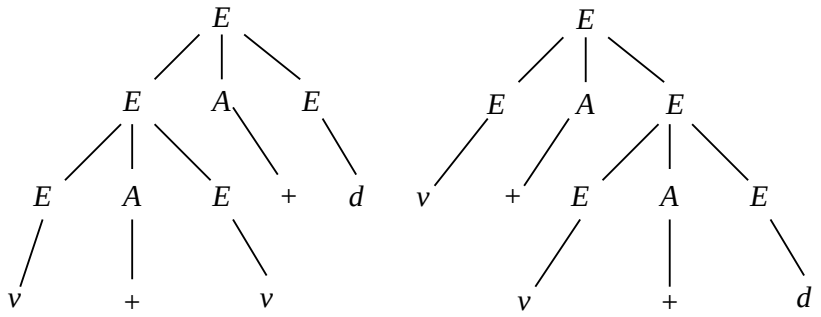


图 21　需要通过规定结合性来解决的二义性

如果我们对产生式 $E \to EAE$ 的符号规定一种结合性，或为左结合，或为右结合，则可以消除此类二义性。我们可以对文法进行适当变换，使新的文法具有所规定的结合性。比如，我们规定符号 *A* 和 *M* 都是左结合的，则将上述文法变换为（读者可从中找出变换的一般规律）：

$E \to E\,A\,T \mid T$
$T \to T\,M\,(E) \mid T\,M\,v \mid T\,M\,d \mid (\,E\,) \mid v \mid d$
$A \to \ +$
$M \to \ *$

当然，有时引入必要的非终结符，可以使文法更加简洁。如，上述文法可改写为：

$E \to E\,A\,T \mid T$
$T \to T\,M\,F$
$F \to (E) \mid v \mid d$
$A \to \ +$
$M \to \ *$

对于该文法，串 *v*＋*v*＋*d* 存在唯一的分析树，见图 2.6。

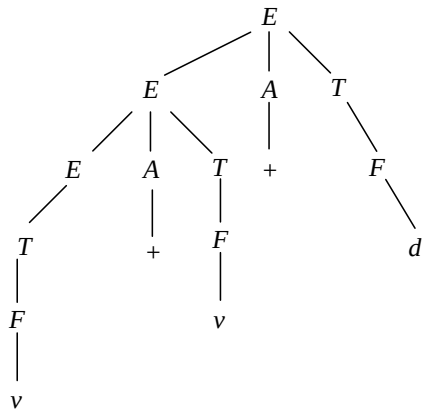


图 2.6　体现左结合的分析树

本小节叙述的另一类二义性消除技术为最近嵌套匹配，可以解决悬挂 else 二义性。考虑以下文法：

$$S \to \varepsilon \mid SS \mid iS \mid iSeS$$

这是从褚如 C 语言中的条件语句抽象出来的文法。例如，*ieie*、*iie*、*iei* 及 *iiee* 是可由上述文法产生的，代表合法的 if 和 else 序列；而 *ei* 和 *ieeii* 不能由上述文法产生，代表非法的 if 和 else 序列。但该文法是二义的，如序列 *iie* 可以有如下不同的分析树（悬挂 else 二义性），参见图 2.7。

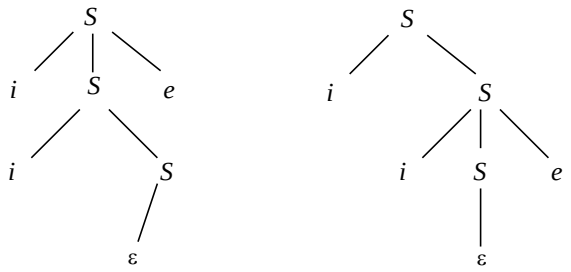


图 2.7　悬挂 else 二义性

如果规定 else 应当匹配前面离它最近的一个未匹配的 if，可将原来文法变换为如下形式的文法 *G*[*S*]，可以保证这一匹配原则（读者可自行总结其一般的变换规律）：

$$S \to \varepsilon \mid SS \mid iS \mid iMeS$$
$$M \to \varepsilon \mid iMeM$$

这里，引入了非终结符 *M*，从 *M* 可以推导出所有可以匹配 if 和 else 的句型。

对于该文法，终结符序列 *iie* 存在唯一的分析树，见图 2.8。

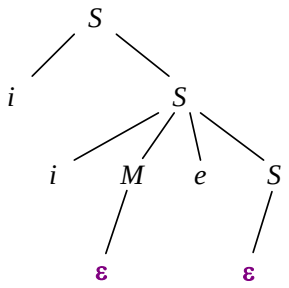


图 2.8　最近嵌套匹配

注意：虽然该文法解决了悬挂 else 问题，但仍然是一个二义文法。如，*iii* 存在不同的分析树，读者可以用前面介绍的方法变换文法，消去此类二义性。

例 12 适当变换文法，找到下列文法所定义语言的一个无二义的文法：

$$S \to SaS \mid SbS \mid ScS \mid d$$

解 该文法的形式很典型，可以先采用优先性级联方法变换文法，然后再规定结合性对文法做进一步变换，即可消除二义性。

设 *a*、*b* 和 *c* 的优先级别依次增高，将文法变换为：

$$S \to SaS \mid A$$
$$A \to AbA \mid C$$
$$C \to CcC \mid d$$

规定 *a*、*b* 和 *c* 的结合性为左结合，进一步将文法变换为（为简洁，引入了不同的非终结符）：

$$S \to SaD \mid D$$
$$D \to DbE \mid E$$
$$E \to EcF \mid F$$
$$F \to d$$

该文法为无二义的。

例 13 找到下列文法所定义语言的一个无二义的文法：

$$S \to aS \mid aSbS \mid \varepsilon$$

解 该文法会产生悬挂 else 二义性，可以先采用最近嵌套匹配方法来变换文法，即强迫 *b* 就近匹配前面没有匹配过的 *a*，可消除二义性。引入新的非终结符 *T*，只产生 *a* 和 *b* 已匹配好的句型。由此，可以将原文法变换为：

$$S \to aS \mid aTbS \mid \varepsilon$$
$$T \to aTbT \mid \varepsilon$$

该文法为无二义的。

练习

1　试构造接受下列语言的一个 DFA：

$L = \{ w \mid w \in \{a, b\}^*$，$w$ 中 a 的数目可

被 2 整除、b 的数目可被 3 整除 $\}$

2　设字母表为 $\{a,b\}$，试给出下列语言的正规表达式各一个：

（1）$\{ w \mid w$ 至少包含 2 个‘a’$\}$

（2）$\{ w \mid w$ 包含偶数个‘a’$\}$

（3）$\{ w \mid w \in \{a, b\}^*$，w 的长度可被 3 整除 $\}$

3　构造产生如下语言的上下文无关文法：

（1）$\{a^n b^{2n} c^m \mid n, m \geqslant 0\}$

（2）$\{a^n b^m c^{2m} \mid n, m \geqslant 0\}$

（3）$\{ a^m b^n \mid m \geq n \}$

（4）$\{ a^m b^n c^p d^q \mid m+n = p+q \}$

（5）$\{ uawb \mid u,w \in \{a, b\}^* \wedge |u| = |w| \}$

4　给出语言 $\{ a^m b^n \mid m \geq 2n \geq 0 \}$ 的二义文法和非二义文法各一个

5　适当变换文法，找到下列文法所定义语言的一个无二义的文法：

$S \rightarrow SaS \mid SbS \mid ScS \mid d$