第三章　词法分析

对高级语言的源程序进行结构分析包括两个层次，一个是判断构成源程序的单词是否合法，即词法分析，另一个是判断整个程序是否合法，即语法分析。一般的编译器都将词法分析和语法分析分成两个相对独立的模块来实现。语法分析往往是基于（上下文无关）文法的，而一个语言的词法规则通常很简单，不需要使用文法来描述这些规则。一般情况下使用正规表达式就可以描述常见的单词符号的结构，如标识符、常量、关键字、运算符等。一个语言的所有单词符号构成一个正规语言，正规语言是乔姆斯基文法分类体系中的 3 型语言，可以由有限自动机识别。本章重点介绍正规表达式、有限自动机的概念，还将介绍正规表达式、有限自动机和正规文法之间的等价性。最后简要介绍词法分析程序的自动构造工具——LEX。

3.1 词法分析程序的设计

词法分析的主要任务是从左到右逐个字符地扫描源程序，将构成源程序的单词符号一个一个地切分出来，把字符流形式的源程序转化为单词符号的序列。单词是源语言中具有独立含义的最小语法单位，包括保留字（或关键字）、标识符、运算符、分界符（标点符号）和常量（整型常量、实型常量、字符串常量）等。

词法分析器负责完成词法分析任务，其主要功能包括：

（1）读入源程序的字符序列；

（2）对源程序进行预处理，如删除程序的注释、空格、回车换行符等，必要的话对宏进行展开；

（3）将单词符号与行号关联起来，以便编译器能将错误信息与源程序位置联系起来；

（4）创建各类符号表，包括标识符表、常数表、函数表等；

（5）识别源程序中的单词符号，并把单词符号及其相关信息登记到符号表中；

（6）输出单词符号的序列。

词法分析器所输出的单词符号常常表示成二元式：<词类表示，单词的属性值>，其中第一个元素表示单词的类别，第二个元素表示单词的属性值。

一个源语言的单词符号按什么标准分类？具体分成几类？类别如何编码？没有统一的规则，编译器开发者可以根据需要自行决定。通常的做法是：

（1）将全体保留字（每个源语言都有一个保留字集合）作为一类，也可以是每个保留字作为一类；

（2）将标识符作为一类，标识符由用户定义，可以作为变量名、常量名、函数名等；

（3）常量有不同的种别，如数值型常量、字符串常量等，可以将每个种别作为一类；

（4）所有运算符可以视为一类，也可以一符一类；

（5）所有分界符可以视为一类，也可以一符一类。

一符一类的话，类别本身就唯一确定了一个单词，单词的属性值就不需要了。多符一类的话，就需要用属性值来刻画单词符号的唯一性。如源程序中的一个片段“25”，词法分析时除了要指明它是整数（词类），还要指明它的值是“25”（属性）。有的单词符号属性值不止一个，如变量标识符的属性就包括类型、存储分配信息、在何处被定义等。这时候用一个属性域是无法存储这么多属性值的。一个处理方法是在符号表中为单词创建一个表项，用一个指针作为单词的属性，指针指向符号表中该单词的表项，表项中存储这个单词的属性值。需要单词的属性值时通过指针到符号表中去访问。

【例 3-1】：考察如下一个简单的 C 语言程序。

```
#include <stdio.h>
int main(void)
{
    printf("HelloWorld!");
    return 0;
}
```

假设编号 01 代表保留字，编号 02 代表标识符，编号 03 代表字符串常量，编号 04 代表整型常量，运算符和分界符一符一类（以符号本身表示，不分配代码）。经词法分析器处理以后，它将被转换为如下的单词符号序列：

```
<#, _>
<01, include>
<<, _>
<02, 指向符号表中 stdio.h 表项的指针>
<>, _>
<01, int>
<01, main>
<(, _>
```

<01, void>

<), _>

<{, _>

<02, 指向符号表中 printf 表项的指针>

<(, _>

<", _>

<03, “HelloWorld!”>

<", _>

<), _>

<;, _>

<01, return>

<04, 0>

<;, _>

<}, _>

词法分析和语法分析都是考察源程序在书写上是否正确，即对源程序进行结构分析。那为什么要将词法分析和语法分析分割成两个独立的逻辑阶段呢？主要原因有以下几点：

（1）简化编译器的设计：词法分析阶段重点考察单词本身是否合法，语法分析阶段重点考察单词序列是否合法。实际的编译器中词法分析模块往往是作为语法分析器的一个子程序被调用，见图 3-1；

（2）提高编译器的开发效率：可以分别使用词法分析器自动生成工具和语法分析器自动生成工具来辅助编译器的开发；

（3）增加编译器的可移植性：可以将输入设备相关的特殊性限制在词法分析器中。

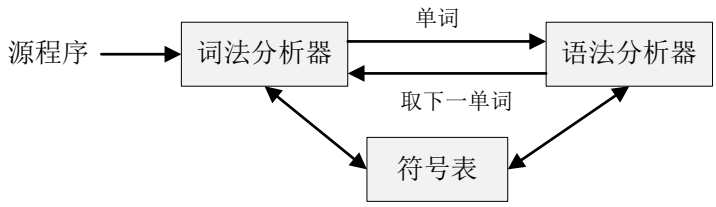


图 3-1 词法分析器和语法分析器的关系

3.2 单词的描述——正规表达式

正规表达式（Regular Expression）是一个表示字符串格式的模式，可以用来描述单词的结构，通常记为 r。每一个正规表达式 r 都匹配一个符号串的集合，称为正规集，记为 L(r) 。

正规表达式简称为正规式，可以由较小的正规式按照一定的规则递归地构建。

定义 3-1：（字母表Σ上的）**正规式**与**正规集**的递归定义：

（1）Φ是Σ上的正规式，它所表示的正规集 L(Φ)是Φ，即空集{ }；

（2）ε是Σ上的正规式，它所表示的正规集 L(ε)仅含一个空符号串ε，即{ε}；

（3）对于任意 $a_i \in \Sigma$， $\mathbf{a_i}$ 是的Σ上的一个正规式，它所表示的正规集 L($\mathbf{a_i}$)是由字符串 a_i 所组成，即{a_i}；

（4）如果 r 和 s 是正规式，令 L(r)=R，L(s)=S，则：

（a）r 与 s 的“或”，记为“r|s”是正规式，且 L(r|s)=R∪S；

（b）r 与 s 的“连接”，记为“rs”是正规式，且 L(rs)=R · S；

（c）r 的 Kleene 闭包“r^*”和正闭包“r^+”是正规式，且 L(r^*)=R^*， L(r^+)=R^+；

（d）(r)和(s)是正规式，且 L((r))=R、L((s))=S。

（5）只有满足（1）、（2）、（3）、（4）的才是正规式。

定义 3-1 不仅定义了正规式和正规集，还定义了正规式的运算。这几个运算都是左结合的，其中“*”和“+”的运算优先级高于“连接”运算和“|”运算，“连接”的运算优先级又高于“|”运算。按照定义，正规式可以包含“(”、“)”，括号用于指定运算的先后顺序，意义清楚时，括号可以省略，如正规式“(a)|((b)*(c))”中的括号就可以去掉，去掉括号之后的式子“a|b*c”与原式意义相同。

【例 3-2】：设Σ={a, b, c}，以下式子是合法的正规式吗？如果是，它的正规集是哪一个？

（1）a|b

（2）(a|b)(a|b)

（3）a*

（4）(a|b)*

（5）a|a*b

解：

（1）根据定义 3-1 的规则（3），a 和 b 都是正规式，再根据定义 5-1 规则（4）的子规则（a），知 a|b 是正规式。其对应的正规集为{a, b}。

（2）已知 a|b 是正规式，根据定义 3-1 规则（4）的子规则（d）知(a|b)是正规式，再根据定义 5-1 规则（4）的子规则（b）知(a|b)(a|b)是正规式。其对应的正规集为{aa, ab, bb, ba}。

（3）a 是正规式，根据定义 3-1 规则（4）的子规则（c），知 a^*是正规式。其对应的正规集是{ ε , a, aa, aaa, ...}，即包含 ε 的由任意多个 a 组成的串的集合。

（4）(a|b)是正规式，根据定义 3-1 规则（4）的子规则（c），知$(a|b)^*$是正规式。其对应的正规集是{ ε , a, b, ab, ba, aab, baa, ...}，即包含 ε 的由任意多个 a、b 组成的串的集合。

（5）根据定义 3-1 相关规则，知 a|a*b 是正规式。其对应的正规集是{a, b, ab, aab, aaab, aaaab, ...}。

如果两个正规式 r 和 s 对应的正规集相同，则称 r 和 s **等价**，记为 r=s，如(a|b)=(b|a)。

令 r、s、t 均为正规式，表 3-1 给出了正规式的一些代数性质。

表 3-1 正规式的代数性质

恒等式	说明
r|s = s|r	“|”运算是可交换的
r|(s|t) = (r|s)|t	“|”运算是可结合的
r(st) = (rs)t	“连接”运算的可结合的
r(s|t) = rs|rt (s|t)r = sr|tr	“连接”运算对“|”运算满足分配律 “|”运算对“连接”运算满足分配律
ε r = r　　r ε = r	对“连接”运算，ε 是单位元
$r^* = (r\|\varepsilon)^*$	“*”运算和 ε 之间的关系
$r^* = r^{**}$	“*”运算是幂等的

为方便表示，有时候希望给某些正规式命名，并在之后的正规式中像使用符号一样使用这些名字。令 Σ 是字母表，那么一个**正规定义**（Regular Definition）是具有如下形式的定义序列：

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

.....

$d_n \rightarrow r_n$

其中：

（1）每个 d_i 都是一个新符号，它们都不在 Σ 中，并且各不相同；

（2）每个 r_i 都是字母表 $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$ 上的正规式。

限制每个 r_i 中只含有 Σ 中的符号和在它之前定义的 d_i ，避免了递归定义的问题。每一个这样定义的 d_i 都可以展开为只包含 Σ 中符号的正规式。

正规式可以描述高级语言中的各类单词符号，如标识符、整数、浮点数等。

【例 3-3】：C 语言的变量标识符是由字母、数字、下划线组成的符号串，C 语言变量标识符的一个正规定义如下：

letter_ → A|B|...|Z|a|b|...|z|_

digit → 0|1|2|...|9

id → letter_(letter_ | digit)*

【例 3-4】：无符号数是形如 5280、0.01234、6.336E4 或 1.89E-4 这样的串。无符号数的正规定义如下：

digit → 0|1|2|...|9

digits → digit digit*

optionalFraction → . digits| ε （可选的小数部分）

optionalExponent → (E (+|-| ε)digits)| ε （可选的指数部分）

number → digits optionalFraction optionalExponent

3.3 单词的识别——有限自动机

3.3.1 有限自动机的定义

在乔姆斯基文法分类体系中，有限自动机是 3 型语言（即正规语言）的识别器。有限自动机是具有离散输入与离散输出的一种数学模型，其输入是符号串，输出是逻辑值“是”或“否”。有限自动机能对输入的字符串是否属于某个模式，或者是否属于某个正规集，或者是否属于某个正规语言作出判断（是的话输出“是”、不是的话输出“否”）。通常高级语言的单词集合是一个正规语言，可以用有限自动机来识别。有限自动机分为非确定有限自动机（NFA，Nondeterministic Finite Automata）和确定有限自动机（DFA，Deterministic Finite Automata）。

定义 3-2：一个非确定有限自动机（NFA）M 是一个五元组，M=（Q， Σ， δ，q_0，F），其中：

Q——状态的非空有穷集合；

Σ——字母表，即输入符号的集合；

δ——是一个映射，称为状态转换函数，δ=（Q×Σ∪{ε} → 2^Q）；

q_0——q_0∈Q，称为开始状态；

F——F⊆Q，是接受状态的非空有穷集合。

每个 NFA 都有一个以上的状态，其中只有一个开始状态，有至少一个接受状态。通常 NFA 都能识别一类字符串，能识别的字符串都是由 Σ 上的字母构成的。NFA 的核心是状态转换函数 δ，δ=（Q×Σ∪{ε} → 2^Q）。该函数的意义是：Q 中的一个状态，面临 Σ 中的一个字母或者空符号 ε 时，映射到 Q 的一个子集。

【例 3-5】：考察一个 NFA，M=（Q， Σ， δ，q_0，F），其中：

Σ={ a, b };

Q = { 0, 1, 2, 3 };

q_0 = 0；

F = { 3 };

δ = { δ (0,a)={0,1}，δ (0,b)={0}， δ (1,a)= Φ， δ (1,b)={2}， δ (2,a)= Φ， δ (2,b)={3}， δ (3,a)= Φ， δ (3,b)= Φ }。

该 NFA 的状态转换函数 δ 可以用一个状态转换表来表示，见表 3-2。

表 3-2 例 3-5 中 NFA 的状态转换表

状态说明	状态	输入字母	
		a	b
开始状态	0	{0, 1}	{0}
	1	Φ	{2}
	2	Φ	{3}
接受状态	3	Φ	Φ

可以采用一个更直观的方式来表示有限自动机。图 3-2 是【例 3-5】定义的有限自动机的有向图表示。图中标记为数字的结点代表 NFA 的状态，标记了“start”的单箭头“→”指向

的状态是开始状态，双圈状态是接受状态，有向边代表状态之间的映射关系，有向边上标记的符号是字母表中的字母。后面通常直接给出状态转换图来表示 NFA。

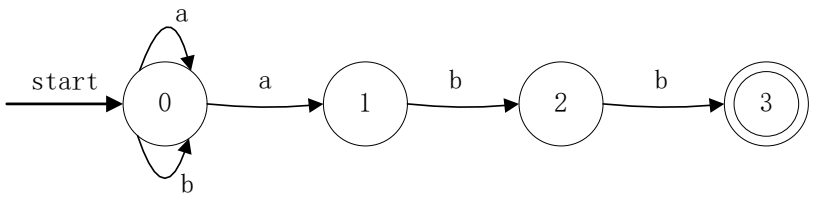


图 3-2 例 3-5 中 NFA 的状态转换图

考察 NFA 状态转换图中每一条从开始状态到接受状态的通路，将构成一条通路的所有有向边上的字母顺次连接构成一个符号串，该符号串称为是 **NFA 可识别的**。有向边上标记的空符号 ε 可以忽略，因为它不会对符号串产生影响。

给定一个 NFA M 和一个符号串 s，判断 M 是否能识别 s 的方法如下：首先将开始状态设为当前状态，读入 s 中的第一个符号 a，并考察当前状态有没有标记为 a 的出边，有的话转移到该出边指向的状态，并将这个状态设为当前状态。然后读入 s 中的下一个符号并执行以上操作。以此类推依次读入 s 中的每个符号。当 s 中最后一个符号读入后，当前正好处在 NFA 的某个接受状态时，识别完成。

要注意的是，在以上识别过程中，如果当前状态有多条出边标记为读入的符号，可以随机选择一条出边继续执行识别操作。如果后续识别过程无法完成，可以回头再选择另外一条出边执行识别操作，这种操作称为**回溯**。在用 NFA 识别符号串的过程中有可能会有很多回溯。如果在识别过程中，当前状态存在标记为 ε 的出边，那么可以在不读入 s 中下一个符号的情况下，进展到 ε 指向的状态。

NFA M 能识别的所有符号串构成的集合称为 **NFA 定义的语言**，记为 L（M）。一个 NFA，有可能可以识别很多符号串。【例 3-5】中定义的 NFA，能识别由 a、b 构成的，以 abb 结尾的任意长度的符号串。这个符号串集合用正规表达式来表示就是$(a|b)^*abb$。

【例 3-6】：考察如下 NFA，

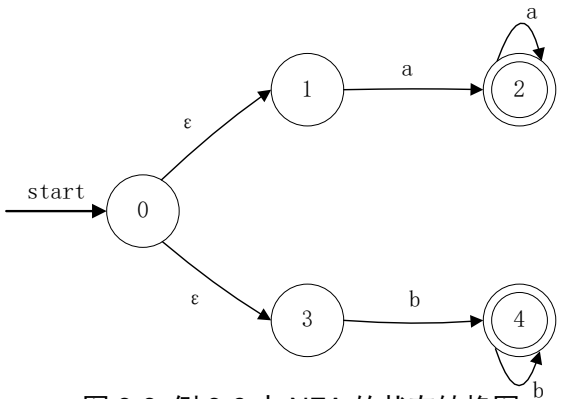


图 3-3 例 3-6 中 NFA 的状态转换图

该 NFA 能接受正规表达式 $aa^*|bb^*$的正规集。

定义 3-3：一个确定有限自动机（DFA）M 是一个五元组，M=（Q，Σ，δ，q_0，F），其中：

Q——状态的非空有穷集合；

Σ——字母表，即输入符号的集合；

δ——是一个映射，称为状态转换函数，δ=（Q×Σ → Q）；

q_0——$q_0 \in Q$，称为开始状态；

F——$F \subseteq Q$，是接受状态的非空有穷集合。

DFA 与 NFA 的区别在于状态转换函数 δ 的不同。DFA 没有标记为 ε 的出边，一个状态面临一个输入符号时最多只转移到一个状态，而不是一个状态集合。

【例 3-7】：考察如下有限自动机，

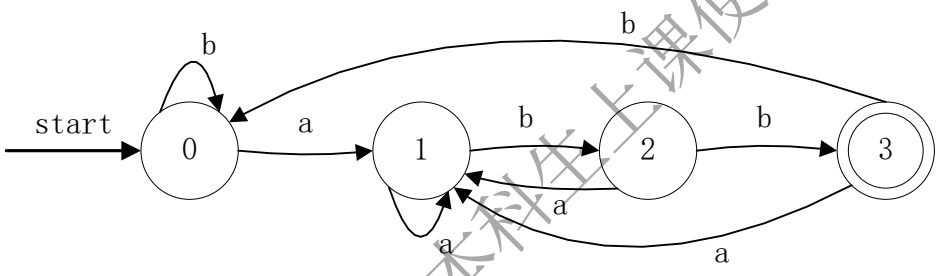


图 3-4 例 3-7 中 DFA 的状态转换图

其状态转换表见表 3-3。

表 3-3 例 3-7 中 DFA 的状态转换表

状态说明	状态	输入字母	
		a	b
开始状态	0	1	0
	1	1	2
	2	1	3
接受状态	3	1	0

该有限自动机没有标记为 ε 的出边，每个状态在面临一个输入符号的时候，只转移到了一个状态。因此，该有限自动机是 DFA。

DFA 识别符号串的方法和 NFA 是相同的。由于 DFA 的确定性（一个状态面临一个输入符号最多只转移到一个状态，并且没有标记为 ε 的有向边），在识别符号串的过程中不会有回溯操作，因此效率更高。

同样，一个 DFA M’能识别的所有符号串构成的集合称为 **DFA 定义的语言**，记为 L（M’）。【例 3-7】中定义的 DFA，能识别的也是正规表达式$(a|b)^*abb$ 对应的正规集。

3.3.2 NFA 到 DFA 的转换

一般来说，NFA 比较容易理解和获得，但是由于回溯操作的存在使得其识别符号串的效率不高。在构造词法分析器的时候，真正实现或者模拟的是 DFA。

给定一个 NFA，总可以构造一个 DFA，使得它们定义的语言是相同的（等价的），即它们识别的是同一个符号串集合。对于给定的 NFA，构造与其等价的 DFA 可以采用子集构造法。子集构造法的基本思想是将 NFA 中一个状态面临一个输入符号转移到的状态集合（子集）作为 DFA 中的一个状态。DFA 在读入符号 a_1、a_2、a_3、...、a_n 之后到达的状态对应于相应 NFA 从开始状态出发，沿着 a_1、a_2、a_3、...、a_n 为标记的路径能够到达的状态的集合。

首先定义几个子集构造法需要用到的函数。

（1） ε -closure(t)（状态 t 的 ε -闭包）：定义为一个状态集合，是状态 t 经过任意条连续 ε 边（标记为 ε 的有向边）到达的状态所组成的集合；

（2） ε -closure(I)（状态集合 I 的 ε -闭包）：定义为一个状态集合。假设 I 是 NFA 的状态集的一个子集，则 ε -closure(I)为：

（a）若 s∈I，则 s∈ ε -closure(I)；

（b）若 s∈I，那么从 s 出发经过任意条连续 ε 边而能到达的任何状态 s’都属于 ε -closure(I)。

（3）edge(t, a)（状态 t 的 a 边转换）：定义为一个状态集合，是状态 t 经过 a 边（标记为 a 的有向边）到达的状态集合；

（4）edge(I, a)（状态集 I 的 a 边转换）：定义为一个状态集合，是指从 I 中的任意一个状态 t 经过 a 边到达的状态集合，记为 J=∪edge (t, a)，t∈I；

（5）DFA_edge(I, a)：定义为一个状态集合，DFA_edge(I, a)= ε -closure(J)。

算法 3-1：NFA 到 DFA 的转换，子集构造（Subset Construction）算法：

输入：一个 NFA N；

输出：一个识别相同语言的 DFA D；

方法：算法为 D 构造一个由 N 的状态子集构成的集合 Dstates 和这些子集之间的转换关系 Dtrans。Dstates 就是 D 的状态集合，Dtrans 是 D 的状态转换函数。

```
Dstates[1] :=  ε -closure(t1);    //求第一个状态子集，t1 是 NFA 的开始状态
p := 1;   j := 1;
WHILE   j <= p DO
 { for   each   a∈ Σ
        □{ e := DFA_edge ( Dstates[j] , a ) ;
              IF e = Dstates[i]   for   some   i <= p      //判断是否是已有的状态子集
                THEN   Dtrans[j, a] = i       //是，建立状态转换关系
                      ELSE   {   p := p+1 ;          //否
                                 Dstates[p] :=e ;      //添加一个新的状态子集
                                 Dtrans[j, a] := p ;   } ;    //建立状态转换关系
           };
       j := j+1;
  }
```

算法从 NFA 的开始状态 t_1 出发，调用函数 ε -closure 求第一个状态子集 Dstates[1]。然后依次考察 Dstates 中每个状态子集面临符号表 Σ 中的符号时，调用函数 DFA_edge 转换到的状态子集，如果出现新的状态子集则加入 Dstates 中。Dtrans 由调用函数 DFA_edge 时反映的状态子集之间的映射关系来构建。算法中的 p 是当前状态集合中子集的最大编号，j 是条件控制变量，是当前正在考察的状态子集的编号。

以上算法求出的 DFA D 的符号表和 NFA N 的一致，D 的开始状态是 Dstates[1]，其接受状态是包含 N 的接受状态的所有子集。

【例 3-8】：试求与以下 NFA 等价的 DFA。

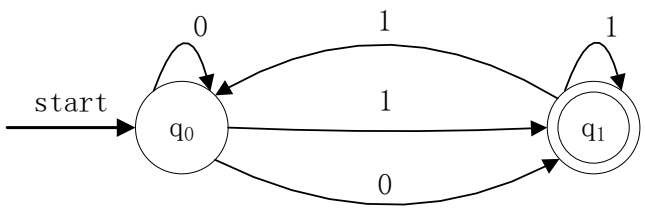


图 3-5 例 3-8 中 NFA 的状态转换图

解：

首先 Dstates[1]= ε -closure(q_0) = {q_0}，这个状态子集是 DFA 的开始状态；

由 DFA_edge (Dstates[1] , 0) = {q_0、q_1}，求得 Dstates[2]= {q_0、q_1}；

由 DFA_edge (Dstates[1] , 1) = {q_1}，求得 Dstates[3]= {q_1}；

进一步求解状态子集{q_0、q_1}和{q_1}面临输入符号 0、1 时的 DFA_edge 转换，构造 Dtrans，即求 DFA 的状态转换关系。

最后求得的 DFA 的状态转换关系如表 3-4 所示。其中{q_0、q_1}和{q_1}均包含状态 q_1，而 q_1 是 NFA 的接受状态，因此状态子集{q_0、q_1}和{q_1}均是 DFA 的接受状态。

表 3-4 例 3-8 中 DFA 的状态转换表

状态说明	Dstates	输入符号	
		0	1
开始状态	{q_0}	{q_0、q_1}	{q_1}
接受状态	{q_0、q_1}	{q_0、q_1}	{q_0、q_1}
接受状态	{q_1}	Φ	{q_0、q_1}

将{q_0}替换为 p_0，{q_0、q_1}替换为 p_1，{q_1}替换为 p_2。画出该 DFA 的状态转换图，见图 3-6。

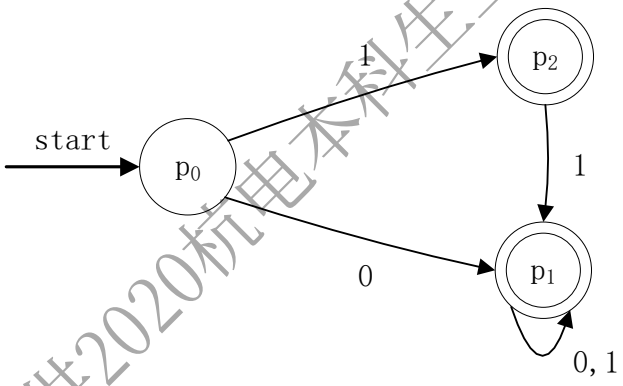


图 3-6 例 3-8 中 DFA 的状态转换图

【例 3-9】：试将以下 NFA 转换为 DFA。

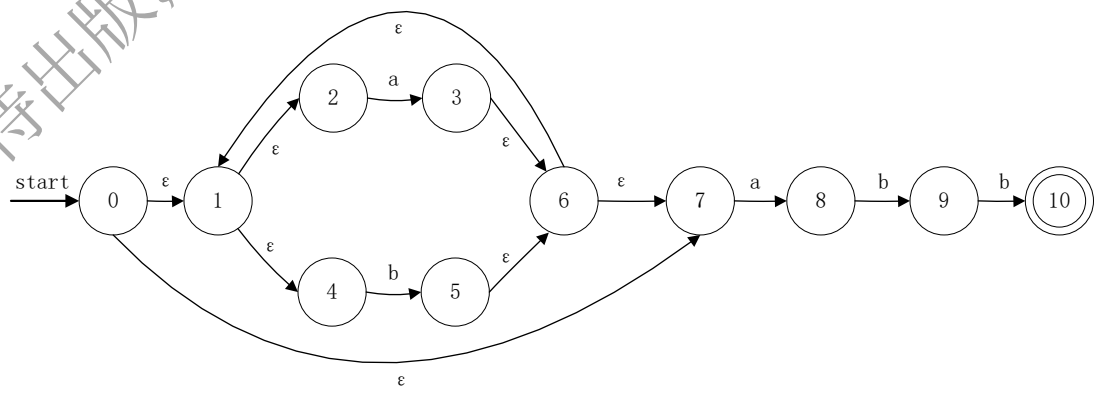


图 3-7 例 3-9 中 NFA 的状态转换图

解：

Dstates[1]= ε -closure(0) = {0、1、2、4、7}

Dstates[2]=DFA_edge (Dstates[1] , a) = {1、2、3、4、6、7、8}

Dstates[3]=DFA_edge (Dstates[1] , b) = {1、2、4、5、6、7}

Dstates[4]=DFA_edge (Dstates[2] , b) = {1、2、4、5、6、7、9}

Dstates[5]=DFA_edge (Dstates[4] , b) = {1、2、4、5、6、7、10}

最终求得的 Dtrans 如表 3-5 所示。

表 3-5 例 3-9 中 DFA 的状态转换表

状态说明	Dstates	输入符号	
		a	b
开始状态	{0、1、2、4、7}	{1、2、3、4、6、7、8}	{1、2、4、5、6、7}
	{1、2、3、4、6、7、8}	{1、2、3、4、6、7、8}	{1、2、4、5、6、7、9}
	{1、2、4、5、6、7}	{1、2、3、4、6、7、8}	{1、2、4、5、6、7}
	{1、2、4、5、6、7、9}	{1、2、3、4、6、7、8}	{1、2、4、5、6、7、10}
接受状态	{1、2、4、5、6、7、10}	{1、2、3、4、6、7、8}	{1、2、4、5、6、7}

将 {0、1、2、4、7}替换为 A，{1、2、3、4、6、7、8}替换为 B，{1、2、4、5、6、7}替换为 C，{1、2、4、5、6、7、9}替换为 D，{1、2、4、5、6、7、10}替换为 E。画出该 DFA 的状态转换图，见图 3-8。

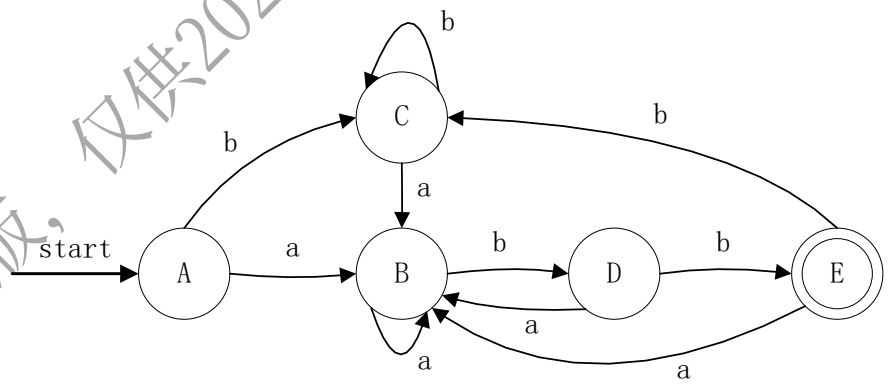


图 3-8 例 3-9 中 DFA 的状态转换图

3.3.3 DFA 的最小化

能识别同一个符号串集合（语言）的 DFA 是等价的，对于同一个语言可能会有多个等价的 DFA 可以识别它。例如上一节中两个 DFA（见图 3-4 和图 3-8）识别的都是 L((a|b)*abb)。这两个 DFA 各个状态的名字不同，状态个数也不一样，但是识别能力是一样的。我们通常采

用模拟 DFA 运行的方式来实现词法分析器，这时候希望 DFA 的状态个数要尽可能地少。因为状态数越少，状态转换表的规模就越小，所需的存储空间也就越小，查询状态转换表也就越快。

可以证明对于任意一个正规语言都有唯一的一个状态数最少的 DFA 识别它。而且从任意一个识别相同语言的 DFA 出发，总可以通过算法将它转换为这个状态数最少的 DFA。将一个 DFA 转化为一个等价的状态数最少的 DFA，称为 DFA 的**化简**或者**最小化**。

DFA 的最小化主要有两种方法，分别是**求同法**和**求异法**。

求同法的基本思想是：寻找 DFA 中的等价状态并合并这些等价状态。所谓**等价状态**就是在识别符号串过程中功能相同的状态，等价状态需满足 2 个条件（考察两个待比较状态 p 和 q）：

（1）一致性条件：状态 p 和 q 必须同时为接受状态或非接受状态；

（2）蔓延性条件：对于所有的 $a\in\Sigma$，令 $\delta(p, a)=s$，$\delta(q, a)=r$，需满足 s 和 r 是等价的。

用求同法对 DFA 进行化简，首先构造一张表，对每一个状态对（q_i, q_j）（i<j）有一表项，每当发现一对状态不等价时，就放一个×到相应表项中，如果一对状态等价，就放一个○到相应表项中。具体操作步骤如下：

（1）根据一致性条件，在每一对接受状态和非接受状态对应的表项中放上一个×。

（2）根据蔓延性性条件，考察状态对(q_i, q_j)（i<j），对于 $a\in\Sigma$，若 $\delta(q_i, a)=s$，$\delta(q_j, a)=r$，如果 r 和 s 不等价，则(q_i, q_j)也不等价，否则(q_i, q_j)等价。重复（2），直到所有表项中均填入了×或者○（所有状态对都考察过了）。

（3）将等价的状态合并成新的状态，将所有等价状态的入边和出边作为新状态的入边和出边。

【例 3-10】：试将图 3-8 中的 DFA 化简。

解：首先构造表 3-6，在这个表中每对状态对应一个表项。

由于 A、B、C、D 是非接受状态，E 是接受状态，首先在(A, E)、(B, E)、(C, E)、(D, E)对应的表项中添加×。

考察(C, D)，因为 $\delta(C, a)=B$，$\delta(D, a)=B$，$\delta(C, b)=C$，$\delta(D, b)=E$，故(C, D)不等价，对应表项添加×；

考察(A, B)，因为 $\delta(A, a)=B$，$\delta(B, a)=B$，$\delta(A, b)=C$，$\delta(B, b)=D$，故(A, B)不等价，对应表项添加×；

考察(A, C)，因为δ(A, a)=B，δ(C, a)=B，δ(A, b)=C，δ(C, b)=C，故(A, C)等价，对应表项添加○；

考察(B, C)，因为δ(B, a)=B，δ(C, a)=B，δ(B, b)=D，δ(C, b)=C，故(B, C)不等价，对应表项添加×；

考察(A, D)，因为δ(A, a)=B，δ(D, a)=B，δ(A, b)=C，δ(D, b)=E，故(A, D)不等价，对应表项添加×；

考察(B, D)，因为δ(B, a)=B，δ(D, a)=B，δ(B, b)=D，δ(D, b)=E，故(B, D)不等价，对应表项添加×。

最终计算结果见表 3-6。

表 3-6 例 3-10 中的等价状态计算表

B	×			
C	○	×		
D	×	×	×	
E	×	×	×	×
	A	B	C	D

将等价状态 A、C 合并，同时保留 A、C 的入边和出边作为合并后状态的入边和出边，化简之后的 DFA 见图 3-9。

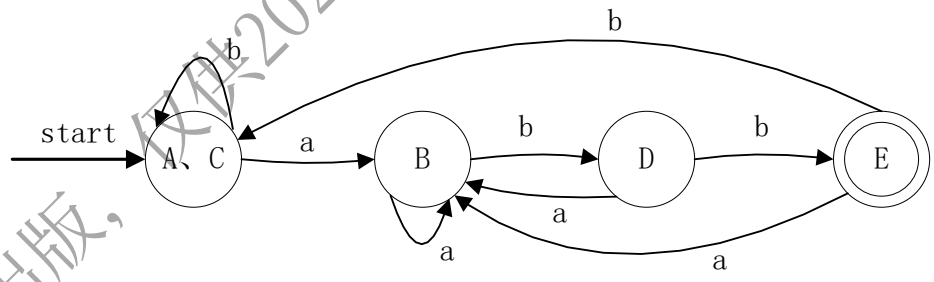


图 3-9 例 3-10 中化简后 DFA 的状态转换图

图 3-9 中的 DFA 和图 3-4 中的 DFA 除了状态的名字不同，其它部分都是相同的，这样的 DFA 我们称为是**同构**的。同构的 DFA 识别符号串的能力相同。

【例 3-11】：试将图 3-10 中的 DFA 最小化。

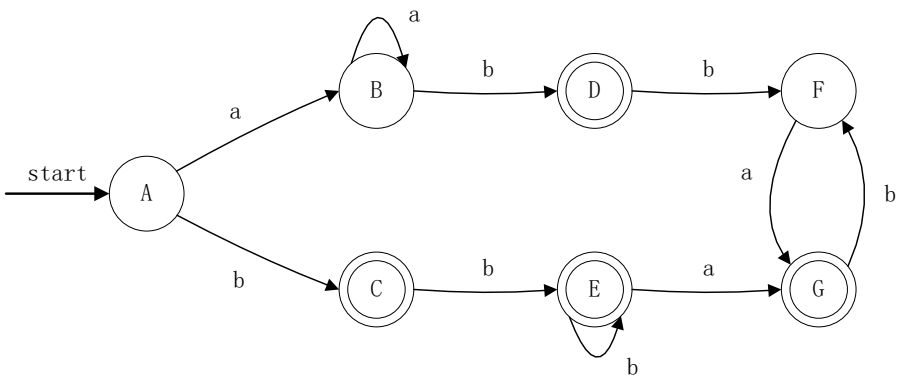


图 3-10 例 3-11 中 DFA 的状态转换图

解：首先构造等价状态计算表，见表 3-7。在接受状态和非接受状态对应的表项中添加×。然后逐一考察其它状态对。其中，D 和 G 面临 a 都没有状态转换，面临 b 都转换到 F，因而是等价的。经过计算其它状态对均不等价。最终计算结果见表 3-7。

表 3-7 例 3-11 中的等价状态计算表

	A	B	C	D	E	F
B	×					
C	×	×				
D	×	×	×			
E	×	×	×	×		
F	×	×	×	×	×	
G	×	×	×	○	×	×

将状态 D、G 合并之后得到最小化 DFA，见图 3-11。

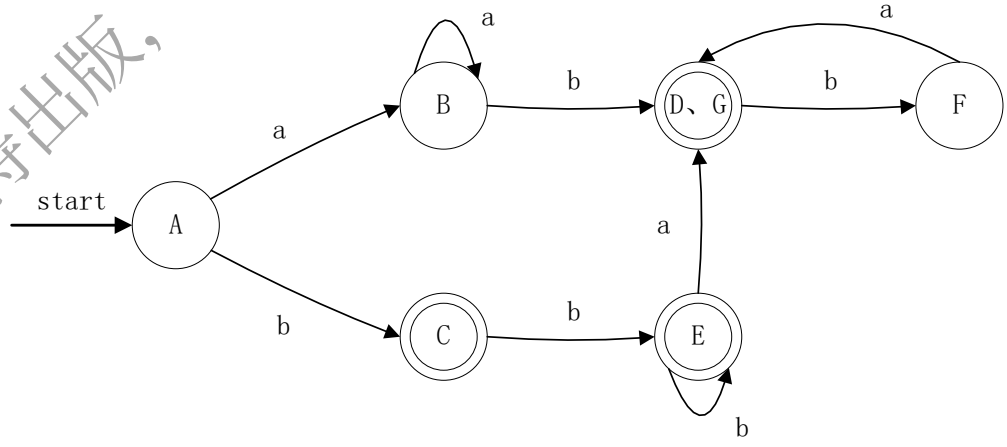


图 3-11 例 3-11 中化简后 DFA 的状态转换图

【例 3-12】：试将图 3-12 中的 DFA 最小化，

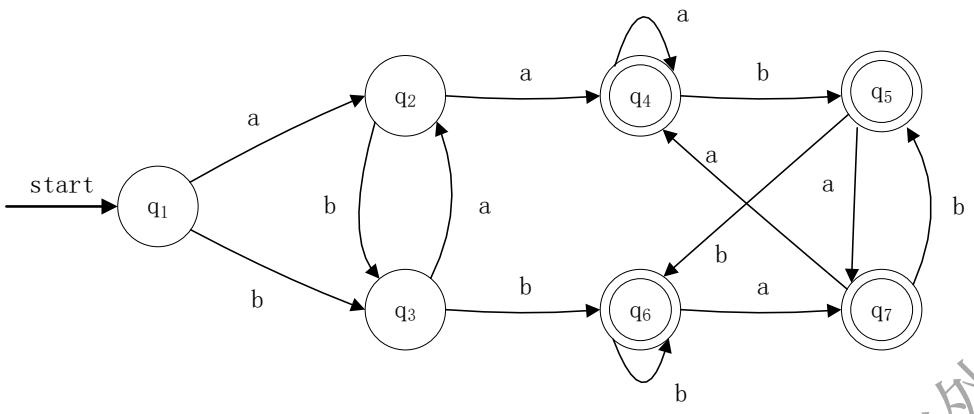


图 3-12 例 3-12 中 DFA 的状态转换图

解：等价状态计算表如表 3-8 所示。

表 3-8 例 3-12 中的等价状态计算表

q_2	×					
q_3	×	×				
q_4	×	×	×			
q_5	×	×	×	○		
q_6	×	×	×	○	○	
q_7	×	×	×	○	○	○
	q_1	q_2	q_3	q_4	q_5	q_6

图 3-12 中状态分为两块，非接受状态 q_1、q_2、q_3，接受状态 q_4、q_5、q_6、q_7。非接受状态和接受状态不等价，对应表项先添加×。

先计算非接受状态 q_1、q_2、q_3 之间是否等价（略），计算后知 3 个非接受状态均不等价。

再计算接受状态 q_4、q_5、q_6、q_7 之间是否等价。

考察(q_4, q_7)，因为 $\delta(q_4, a)=q_4$， $\delta(q_7, a)=q_4$， $\delta(q_4, b)=q_5$， $\delta(q_7, b)=q_5$，故(q_4, q_7)等价，对应表项添加 O；

考察(q_5, q_6)，因为 $\delta(q_5, a)=q_7$， $\delta(q_6, a)=q_7$， $\delta(q_5, b)=q_6$， $\delta(q_6, b)=q_6$，故(q_5, q_6)等价，对应表项添加 O；

考察(q_4, q_5)，因为 δ (q_4, a)=q_4， δ (q_5, a)=q_7， δ (q_4, b)=q_5， δ (q_5, b)=q_6，故(q_4, q_5)等价，对应表项添加 O；

考察(q_4, q_6)，因为 δ (q_4, a)=q_4， δ (q_6, a)=q_7， δ (q_4, b)=q_5， δ (q_6, b)=q_6，故(q_4, q_6)等价，对应表项添加 O；

考察(q_5, q_7)，因为 δ (q_5, a)=q_7， δ (q_7, a)=q_4， δ (q_5, b)=q_6， δ (q_7, b)=q_5，故(q_5, q_7)等价，对应表项添加 O；

考察(q_6, q_7)，因为 δ (q_6, a)=q_7， δ (q_7, a)=q_4， δ (q_6, b)=q_6， δ (q_7, b)=q_5，故(q_6, q_7)等价，对应表项添加 O。

将等价状态 q_4、q_5、q_6、q_7 合并为一个新状态 q*，得到最小化 DFA，见图 3-13。

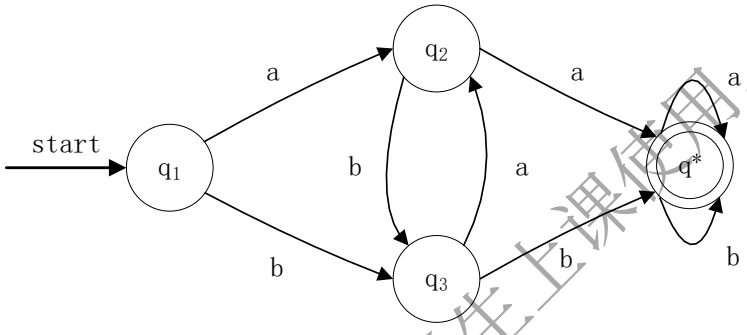


图 3-13 例 3-12 中化简后 DFA 的状态转换图

DFA 化简的第二种方法是求异法，其基本思想是：首先将状态划分为接受状态与非接受状态两组，然后逐步将这个划分细化，最后得到一个不可再细化的状态集的划分，每个状态子集作为一个状态。

具体步骤如下（设待化简的 DFA 为 M，其状态集合为 S）：

（1）首先将 DFA M 的状态集 S 中的接受状态与非接受状态分开，形成两个子集，即得到基本划分 Π。

（2）对 Π 建立新的划分 Π_{New} ，对 Π 的每个状态子集 G，进行如下变换：

（a）把 G 划分成新的子集，使得 G 中的两个状态 s 和 t 属于同一子集，当且仅当对任何输入符号 a，状态 s 和 t 转换到的状态都属于 Π 的同一子集。

（b）用 G 划分出的所有新子集替换 G，形成新的划分 Π_{New}。

（3）如果 Π_{New} = Π，则执行第（4）步；否则令 Π = Π_{New} ，重复第（2）步。

（4）划分结束后，将划分中的每个状态子集作为一个单独的新状态，子集中状态的入边和出边作为新状态的入边和出边。这样得到的 DFA M’是与 DFA M 等价的所有 DFA 中状态数最少的。

【例 3-13】：采用求异法化简图 3-12 中的 DFA。

解：

首先把该 DFA 的状态分为 2 组：接受状态组{q_4, q_5, q_6, q_7}，非接受状态组{q_1, q_2, q_3}；

接着考察子集{q_4, q_5, q_6, q_7}，当输入 a 或 b 的时候，该子集中每个状态可到达的状态集包含于{q_4, q_5, q_6, q_7}，因此该子集不可再划分；再考察子集{q_1, q_2, q_3}，由于 q_2 经过 a 边到状态 q_4，而 q_1、q_3 均到达 q_2，因此把 q_2 单独划分出来；再考察子集 {q_1, q_3}，由于 q_3 经过 b 边到状态 q_6，而 q_1 到达 q_3，因此 q_1 和 q_3 也必须分开。

这样将所有状态划分为 4 个状态子集{q_1}、{q_2}、{q_3}、{q_4, q_5, q_6, q_7}，每个状态子集均不可再分。将{q_4, q_5, q_6, q_7}作为一个单独的新状态，取名 q*，每个状态的入边和出边作为 q* 的入边和出边。最小化后的 DFA 见图 3-13，与求同法的结果一致。

3.4 正规表达式与有限自动机的等价性

以上两节分别介绍了正规表达式和有限自动机。每个正规表达式都对应一个正规集，正规集是正规语言，而正规语言是乔姆斯基文法体系中的 3 型语言。3 型语言的识别器是有限自动机。从这个意义上来说，正规式和有限自动机的描述能力是一样的。

下面给出一个算法，该算法可以将任何正规表达式转化为识别相同语言的 NFA。这个算法是基于正规表达式的语法结构的，遵循正规表达式的递归定义。可以根据对一个正规表达式结构的分析，构造正规表达式的分析树。

【例 3-14】：考察正规表达式$(a|b)^*abb$。该表达式首先由子表达式 a 和 b 用“或”运算符“|”连接起来构成较大的表达式“a|b”，然后用括号括起来得到表达式“(a|b)”，接着做“*闭包”运算得到“$(a|b)^*$”，再依次连接上 a、b 和 b，形成最终的正规表达式(a|b)*abb。根据以上分析，可以构造该表达式的分析树，见图 3-14。

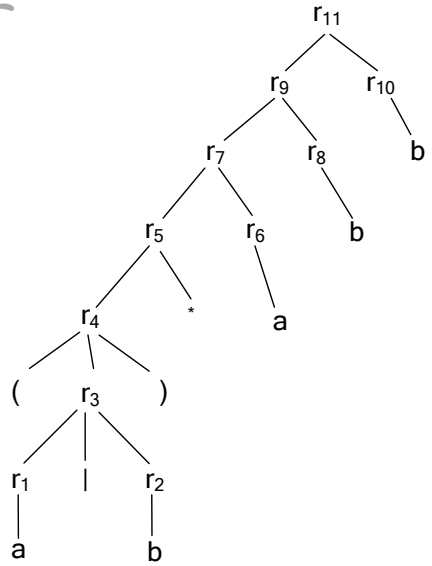


图 3-14 正规表达式$(a|b)^*abb$ 的分析树

要构造任何正规表达式的 NFA，首先构造能够识别 ε 和字母表中单个符号对应的自动机，然后一步步构造包含“或”运算、“连接”运算及“*闭包”运算的正规表达式的自动机。

算法 3-2：将正规表达式转化为一个 NFA 的算法。

输入：字母表 Σ 上的一个正规表达式 r；

输出：一个接受 L(r)的 NFA N；

方法：首先对 r 进行语法分析，构造反映其层次结构的语法树，然后基于 r 的语法结构逐步构造与其对应的 NFA。构造一个 NFA 的规则分为基本规则和归纳规则两组。基本规则处理不包含运算符的子表达式，而归纳规则根据给定表达式的直接子表达式的 NFA 构造出这个表达式的 NFA。

基本规则：

（1）构造正规表达式 ε 的 NFA。ε 对应的正规集是{ ε }，只识别空串 ε 的一个 NFA 如图 3-15 所示，其中 *i* 是 NFA 的开始状态，*f* 是 NFA 的接受状态。

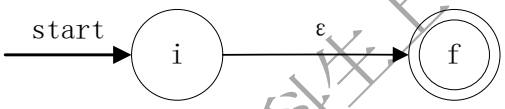


图 3-15 正规式 ε 的 NFA

（2）构造字母表 Σ 中单个符号 a 构成的表达式的 NFA。a 对应的正规集是{a}，只识别一个符号串 a 的 NFA 如图 3-16 所示，其中 *i* 是 NFA 的开始状态，*f* 是 NFA 的接受状态。

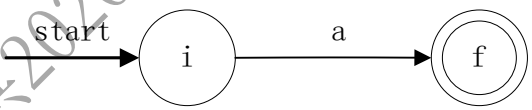


图 3-16 只识别符号串 a 的 NFA

注意，对于 ε 或某个 a 的作为 r 的子表达式的每次出现，都应使用这两个基本构造规则分别构造出一个独立的 NFA。

归纳规则：

假设正规表达式 s 和 t 的 NFA 分别为 N(s)和 N(t)，它们分别识别 L(s)和 L(t)。

（1）假设 r=s|t，令 r 对应的 NFA 为 N(r)，N(r)应能识别 L(s)∪L(t)。可以按照图 3-17 中的方式构造得到 N(r)。其中 *i* 和 *f* 是新状态，分别是 N(r)的开始状态和接受状态。从 *i* 到 N(s)和 N(t)的开始状态各构造一条标记为 ε 的边，从 N(s)和 N(t)的每一个接受状态到 *f* 也各构造一条 ε 边。注意，N(s)和 N(t)的接受状态在 N(r)中不再是接受状态。因为从 *i* 到 *f* 的任何路径

要么只通过 N(s)，要么只通过 N(t)，且离开 *i* 或进入 *f* 的 ε 边都不会改变路径上的标记，因此 N(r)可识别 L(s)∪L(t)，也就是 L(r)。

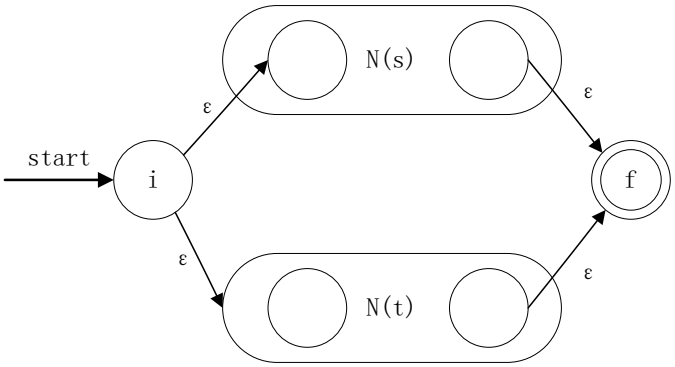
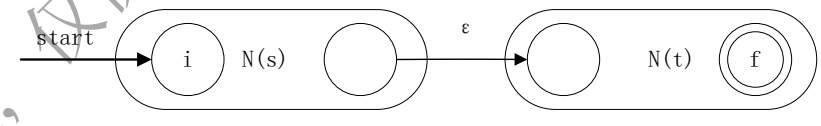


图 3-17 两个正规表达式的“|”的 NFA

（2）假设 r=st，N(r)应能识别 L(s)L(t)。可按图 3-18 所示构造 N(r)。N(s)的开始状态作为 N(r)的开始状态，N(t)的接受状态作为 N(r)的接受状态。如果 N(s)只有一个接受状态，则将 N(s)的接受状态和 N(t)的开始状态合并为一个状态，合并后的状态保留合并前两个状态的全部入边与出边，如图 3-18(a)所示。如果 N(s)有多个接受状态，则须从 N(s)的每个接受状态构造 ε 边指向 N(t)的开始状态，如图 3-18(b)所示。图中一条从 *i* 到 *f* 的路径必须首先经过 N(s)，故这条路径上的标记以 L(s)中的某个串开始。然后这条路径继续通过 N(t)，故这条路径上的标记以 L(t)中的某个串结束。因此这个 N(r)恰好识别 L(s)L(t)。

(a)



(b)

图 3-18 两个正规表达式的“连接”的 NFA

（3）假设 $r=s^*$，N(r)应能识别 $L(s)^*$。可按图 3-19 所示构造 N(r)。其中 *i* 和 *f* 是两个新状态，分别是 N(r)的开始状态和唯一的接受状态。要从 *i* 到达 *f*，可以沿着新引入的标记为 ε 的路径前进，这个路径识别 ε，是 $L(s)^0$ 中的一个串。也可以到达 N(s)的开始状态，然后经过该 NFA，再零次或多次从它的接受状态回到他的开始状态并重复上述过程。这些路径使得 N(r)可以识别 $L(s)^1$ 中、$L(s)^2$ 中、.....、$L(s)^n$ 中、.....等等集合中的所有串，因此 N(r)识别的所有串的集合就是 $L(s)^*$ 。

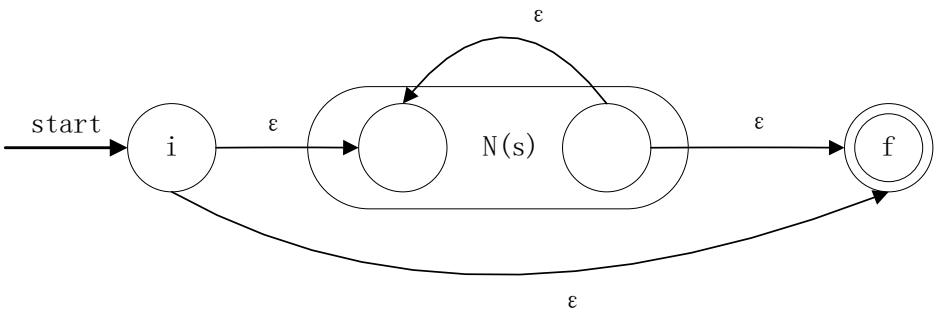


图 3-19 一个正规表达式的“*闭包”的 NFA

（4）假设 r=(s)，那么 L(r)=L(s)，可以直接把 N(s)作为 N(r)。

【例 3-15】：求正规表达式 $01^*|1$ 对应的 NFA。

解：首先分析该表达式的结构，并构造其语法分析树，见图 3-20。

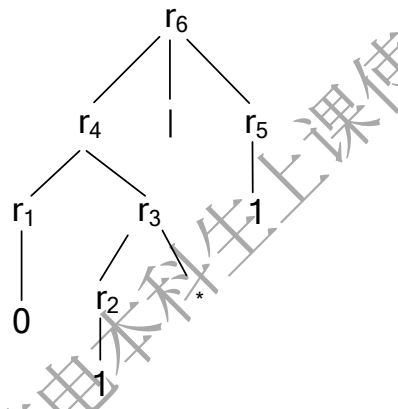


图 3-20 正规表达式 $01^*|1$ 的语法分析树

然后画出 NFA 的状态转换图，见图 3-21。

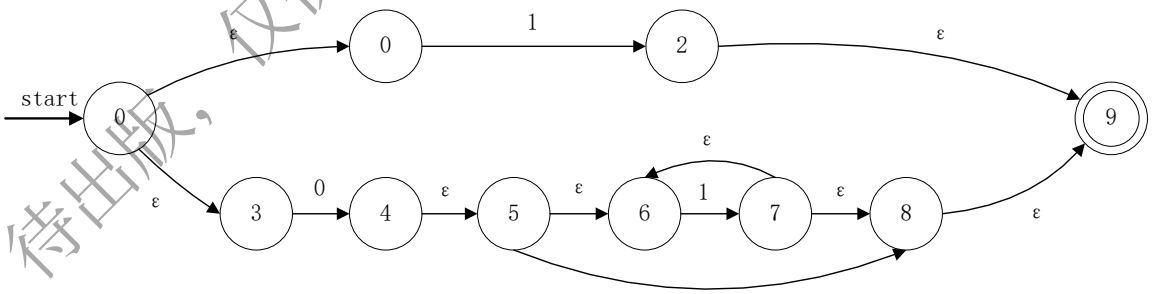


图 3-21 例 3-15 求得的 NFA

【例 3-16】：求正规表达式(a|b)*abb 对应的 NFA。

解：首先构造该表达式的语法分析树，见图 3-14。然后画出 NFA 的状态转换图，见图 3-22。

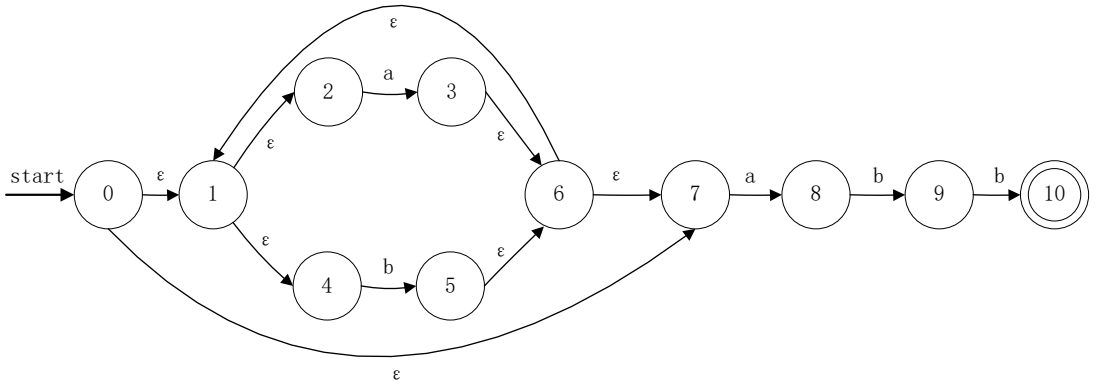


图 3-22 例 3-16 求得的 NFA

3.5 词法分析程序的自动构造工具

为提高开发效率，可借助自动化构造工具来辅助词法分析器的开发。用得最广泛的词法分析程序自动构造工具是 LEX（Lexical Analyzer Generator）。LEX 最早由 AT&T 公司的 Mike Lesk 和暑期实习生 Eric Schmidt 在 1975 年编写并逐渐流行起来。1987 年左右，Lawrence Berkeley 实验室的 Vern Paxson 将 Lex 的一个版本改写为 C 语言实现，称为 Flex（Fast Lexical Analyzer Generator）。由于它比 AT&T 的 LEX 更快速和可靠，逐渐成为主流的 LEX 版本。Flex 现在是 SourceForge 的一个基于伯克利许可证的开源项目。

LEX 的使用过程如图 3-23 所示。首先，使用 LEX 语言规范编写词法分析器的源程序 lex.l，然后通过 LEX 编译器将 lex.l 转换为 C 语言程序 lex.yy.c。LEX 编译器根据 lex.l 中给定的正规表达式生成有限自动机的状态转换表，并产生以该表为基础的单词识别驱动程序。Lex.yy.c 经过 C 编译器生成目标程序 a.out。a.out 即为一个词法分析器。

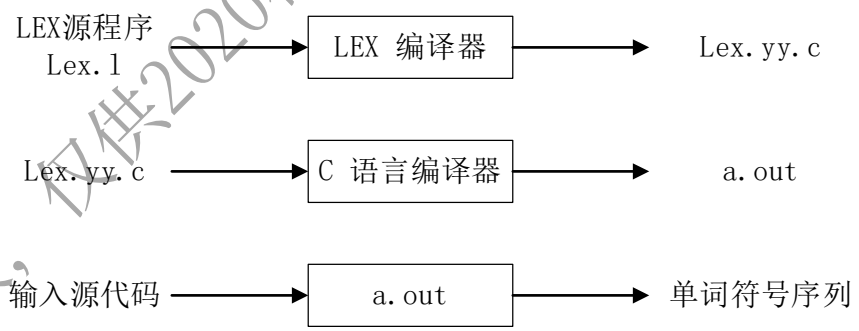


图 3-23 LEX 构建词法分析器的过程

LEX 程序包含三个部分，各部分之间通过仅有%%的行隔开。第一个部分包含声明和选项设置，第二个部分是一系列的模式和动作定义，第三部分为辅助过程，是会被拷贝到生成的词法分析器程序里面的 C 语言代码。LEX 程序的基本构成如下：

声明部分

%%

识别规则部分

%%

辅助过程部分

在声明部分，%{和%}之间的代码会原封不动地拷贝到生成的 C 代码文件的开头部分。声明部分还可以给定一些正规定义，相当于定义一系列正规表达式的名称。第二个部分中每一行的开头部分是模式（正规表达式），后面用{ 、}括起来的内容为匹配到该模式后所需要执行的 C 语言代码。第三部分辅助过程中的 C 语言代码是主程序，负责调用 LEX 提供的词法分析函数 yylex()，并输出结果。

下面看一个简单的 LEX 的例子，它的功能是统计程序中的字符数、单词数和行数，程序文本见图 3-24。

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+        {words++; chars +=strlen(yytext); }
\n               {chars++; lines++;}
.                {chars++;}
%%
int main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
        return 0;
}
```

图 3-24 一个简单的 LEX 例子

这个例子的声明部分定义了字符数、单词数和行数变量，并作了初始化。例子中共设置了三个模式，第一个模式“[a-zA-Z]+”用来匹配一个单词，对应的动作是更新单词数和字符数。第二个模式“\n”用来匹配换行符，对应的动作是更新行数和字符数。第三个模式是一个点号，代表任意一个字符，对应的动作是更新字符数。

LEX 编译器产生的词法分析器的基本结构如图 3-25 所示，其核心是一个有限自动机。

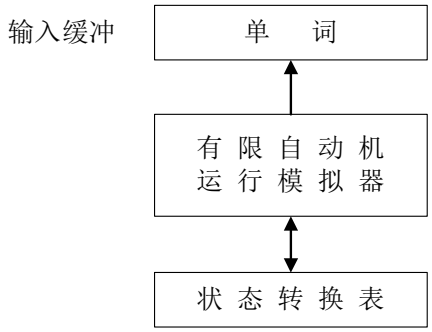


图 3-25 LEX 产生的词法分析器结构图

输入缓冲采用双指针形式：一个指针指向单词的开始位置，另一个为向前搜索其余符号的向前指针。LEX 编译器根据正规表达式构造有限自动机的状态转换表，然后结合有限自动机模拟器，构造一个完整的词法分析器。有限自动机模拟器通过查询状态转换表即可从输入缓冲中分析出正规表达式所描述的单词符号。整个过程可以分为以下三个步骤：

（1）对于 LEX 源程序中识别规则部分的每一个模式 p_i ($1 \leqslant i \leqslant n$)，构造一个相应的 NFA N_i；

（2）通过引入一个全局的初始状态 q_0，并用 ε 边将 q_0 和每个 N_i 的开始状态连接起来，从而得到一个 NFA N，N 可以识别 LEX 源程序中所定义的各类单词；

（3）利用子集构造法将 N 确定化，必要的时候进行最小化，并将所得到的确定性有限自动机以状态转换表形式，与模拟器驱动程序 yylex 一起输出到 lex.yy.c 中。

从 LEX 的基本原理可以看出，有限自动机在词法分析程序的生成器中起着至关重要的作用。这也进一步说明要实现程序的自动化构造，首要条件是对程序功能进行合适的形式化描述。

3.6 小结

源程序的初始形态是字符流，编译的第一个步骤是词法分析，即对源程序进行预处理，并分析构成源程序的字符序列，从源程序中识别出所有的单词符号。词法分析负责将源程序从字符流的形式转化为单词符号的序列。单词是具有独立含义的最小语法单位，每个单词用一个形如<词类表示，单词的属性值>的二元式来表示。高级语言的单词符号一般都包括保留字、标识符、运算符、分界符和常量等这样几类。单词是符号的有穷序列，要识别单词符号首先需要采用一个数学工具对单词的结构进行描述。正规表达式是一个描述字符串格式的模式，可以作为描述单词结构的数学工具。每一个正规表达式都匹配了一个符号串集合，这个符号串集合称为正规集。高级语言的单词符号都可以用正规表达式来描述，一个高级语言的所有单词符号构成一个正规集。正规集是乔姆斯基文法分类体系中的 3 型语言（正规语言），可以由有限自动机来识别。有限自动机是一种具有离散输入与离散输出的数学模型，它能对输入的字符串是否属于某个正规集作出判断。有限自动机分为非确定的有限自动机（NFA）和确定的有限自动机（DFA）。NFA 的不确定性体现在两个方面，一个是它的一个

状态面临一个输入符号的时候有可能转移到多个状态上，另一个是 NFA 中允许存在 ε 边。NFA 的不确定性使得它在识别符号串的时候可能产生回溯，严重影响符号串的识别效率。DFA 克服了 NFA 的这个缺点，它的一个状态在面临一个输入符号的时候最多只转移到一个状态，也不允许有 ε 边。但是 DFA 不容易理解，直接构造也比较困难。直接构造的有限自动机往往是 NFA，为提高识别字符串的效率，可以通过子集构造法将 NFA 转化为等价的 DFA，还可以进一步对 DFA 进行化简。可以证明，对于任何一个 NFA（或 DFA）总存在一个状态数最少的和它们等价的 DFA。正规表达式和有限自动机都是描述正规集（正规语言）的数学工具，它们的描述能力是相同的。给定一个正规表达式，可以分析它的语法结构，构造反映它层次结构的语法分析树，并在此基础上构造和它等价的有限自动机。

本章要求理解掌握正规式和正规集的递归定义，理解掌握 NFA 和 DFA 的定义及其特点，重点是正规式到有限自动机的转换方法，NFA 确定化和 DFA 最小化的算法。构造词法分析器的一般过程是首先用正规式描述单词的结构，再将正规式转化为 NFA，在将 NFA 确定化及化简之后，编写能模拟最小化 DFA 运行的词法分析程序。实现词法分析器还可以借助自动化构造工具（如 LEX）来提高效率。

习题

3.1 词法分析的输入是源代码文件，在你熟悉的操作系统上，用熟悉的程序语言编写一个程序，将一个文件读入内存，并统计字符序列 abcabc 出现的次数。要求程序运行速度越快越好。（注意：序列可能会存在重叠的情况，如 abcabcabc 就存在 2 个 abcabc 的序列）

3.2 给定正规表达式 $a^*(a|b)aa$，

(1) 构造一个对应的 NFA；

(2) 通过子集构造算法将该 NFA 转换为 DFA。

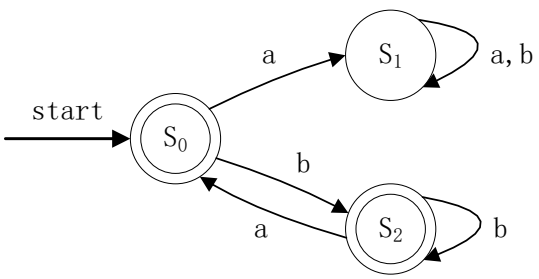
3.3 给定正规表达式 $((a|b)(a|bb))^*$，

(1) 构造一个对应的 NFA；

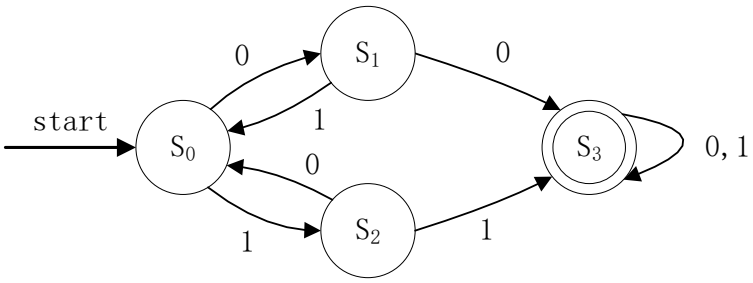
(2) 通过子集构造算法将该 NFA 转换为 DFA。

3.4 用自己的语言描述如下有限自动机所接受的语言。

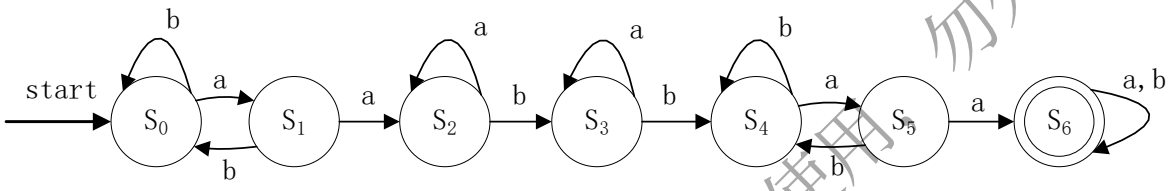
(1)



(2)

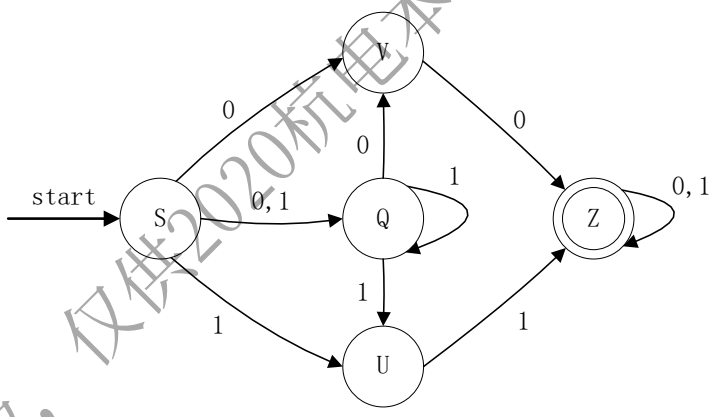


(3)



3.5 构造某个语言中“标识符”的正规表达式，其中标识符的定义为：以字母开头的字母数字串，或者通过“.”或“-”连接起来的字母开头的字母数字串。

3.6 将下面的 NFA 转化为最小状态的 DFA。



3.7 某语言的合法句子形式如下：{x | x∈{0, 1}⁺，且 x 以 1 开头、以 101 结尾}，请写出能描述该语言的正规表达式，构造相应的 NFA，并将其转换为 DFA，如能化简，则进行最小化处理。

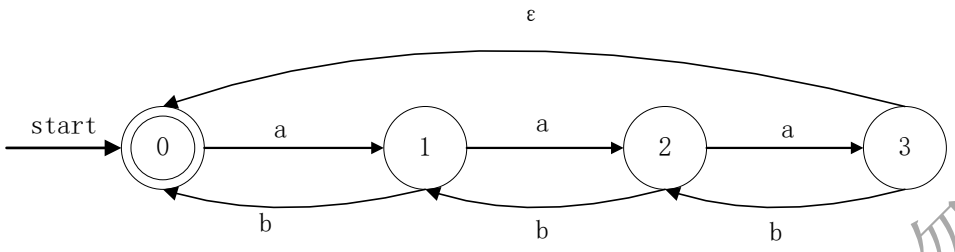
3.8 请写出在Σ={a, b}上，不以 a 开头，但以 aa 结尾的字符串集合的正规表达式，并构造与之等价的最小 DFA。

3.9 构造正规式 r=(a|ba)*对应的最小化 DFA。

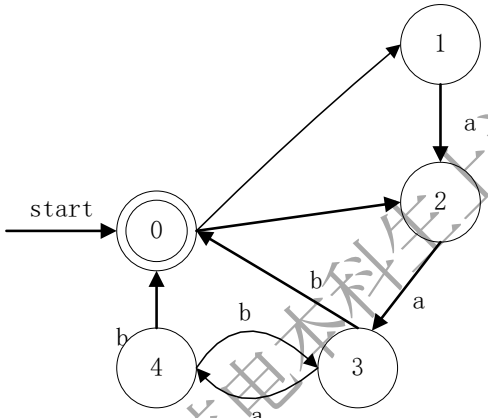
3.10 设计一个 DFA，它能接受以 0 开始，以 1 结尾的所有序列（提示：输入的字母表为Σ={0,1}，首先构造该语言的正规式，然后转换为对应的 DFA）

3.11 给定Σ={a, b}上的正规式 $r=b^*ab(b|ab)^*$，构造一个能识别该正规式所描述的语言的 DFA。

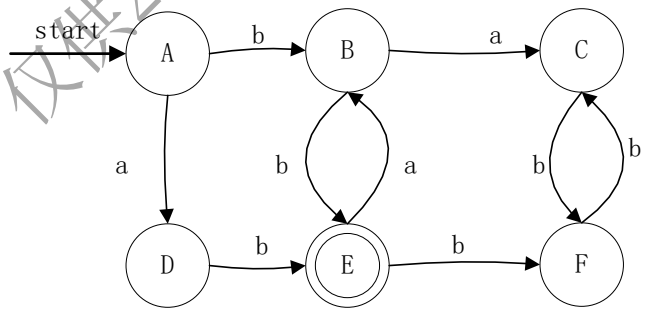
3.12 通过子集构造算法将下列 NFA 转换为 DFA。



3.13 最小化下列 DFA。



3.14 最小化下列 DFA。



3.15 构造下列正规语言对应的 DFA，其中字母表均为{a, b}

(1) 所有包含恰好 3 个 b 的符号串集合（任意个 a）；

(2) 含有 3 的倍数个 b 的符号串集合（任意个 a）。

3.16 假设一个二进制数串，高位在左，并可能以 0 开头，试构造下列语言对应的 DFA：

(1) 所有 4 的倍数的二进制数字串，如：0，100 和 10100 等；

(2) 所有 5 的倍数的二进制数字串，如：0，101，10100 和 11001 等；

(3) 给定任意非零自然数 n，构造识别 n 的倍数的二进制数字串的 DFA，至少需要多少个状态？提示：可以把 n 写作为 $a*2^b$，其中 a 是奇数。