编译技术

课程实验指导书

杨金民，陈果，黎文伟

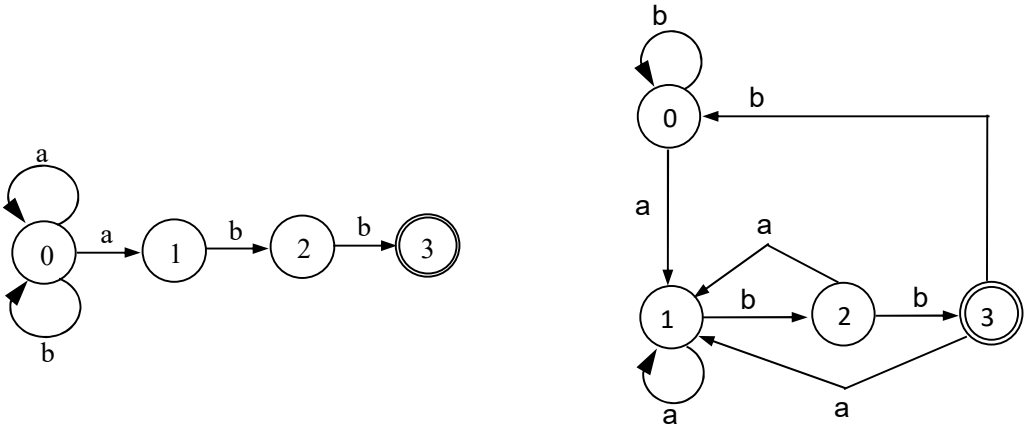
2022 年

目录

试验一、正则运算表达式的 DFA 构建

1. 正则运算表达式的最简 NFA 构建

对于正则表达式，构建其 NFA 不是目标而是策略和手段，构建 DFA 才是目标。对于正则表达式 r →(a|b)*abb，按照 NFA 原生构造法得到 NFA 如图 2.11(c)所示。再使用子集构造法得到的 DFA 如图 2.12 所示。对于该正则表达式，其最简 NFA 如图 2.13(a)所示。使用子集构造法从其得到的 DFA 如图 2.13(b)所示。对照图 2.12 和图 2.13(b)所示的两个 DFA，发现后者少了一个状态和两条边，比前者简单。对于一个正则表达式，当然希望得到的 DFA 越简单越好，这样有利于降低词法分析的计算和存储开销。



（a）正则表达式(a|b)*abb 的最简 NFA　　（b） 由最简 NFA 得出的 DFA

图 2.13 正则表达式(a|b)*abb 的最简 NFA 和由其得到的 DFA

思考题 2-12：图 2.13(a)所示 NFA 中尽管没有空转换，但它是一个 NFA，为什么？既然它是一个 NFA，那么就可使用 DFA 子集构造法，来得出其状态转换表 DTran。该状态转换表中有几个状态集合？每一状态集合含哪些状态？写出其状态转换表 DTran。

上述同一个正则表达式的两个 DFA，后者比前者简单。再对照它们对应的 NFA，发现后者的 NFA 要比前者的 NFA 简单很多。由此可知，由简单的 NFA 能得出简单的 DFA。NFA 的简单性体现在空转换要少很多。对于 NFA 原生构造法中的空转换，是不是有条件可省掉？省掉的条件是什么？这就是**最简 NFA 构建法**要回答的问题。

对于正则表达式的最简 NFA 构建，先通过举例来展示空变换的作用。有五个正则表达式 r_1 = a 和 r_2 = b，$r_3 = r_1^+$和 $r_4 = r_2^+$，以及 $r_5 = r_3 \cdot r_4$。按照正则表达式的 NFA 原生构造法，得出的这五个正则表达式的 NFA 如图 2.14(a)至(e)所示。现在来观察正则表达式 r_3 和 r_4，以及 r_5 的 NFA 特性。r_3 和 r_4 的 NFA 的开始状态都有入边，结束状态都有出边。如图 2.14(e)所示 r_5 的 NFA 其实不正确。其理由是：字符串“abab”匹配 r_5 的 NFA，但是根据 r_5 的语义，不应该匹配。因此按原生构造法得到的 r_5 的 NFA 错误。其原因是：r_3 的结束状态有出边，而 r_4 的开始状态有入边，当把 r_3 的 NFA 的结束状态与 r_4 的 NFA 的开始状态接合在一起时，会出现倒灌情形。倒灌是指：由 2 状态通过空转换到达 1 状态，再通过空转换到达 0 状态。而语义并不是这样。语义是：由 2 状态通过空转换到达 1 状态，不允许再到 0 状态。

为了得出正确的 r_5 的 NFA，需要引入一个空转换，如图 2.14(f)所示。通过引入一个空转换，便消除了倒灌情形，保证了 NFA 构造的正确性。从这个案例分析可知，正则表达式的 NFA 构建，要对输入的 NFA 检查其开始状态是否有入边，其结束状态是否有出边，确保不出现倒灌情形。

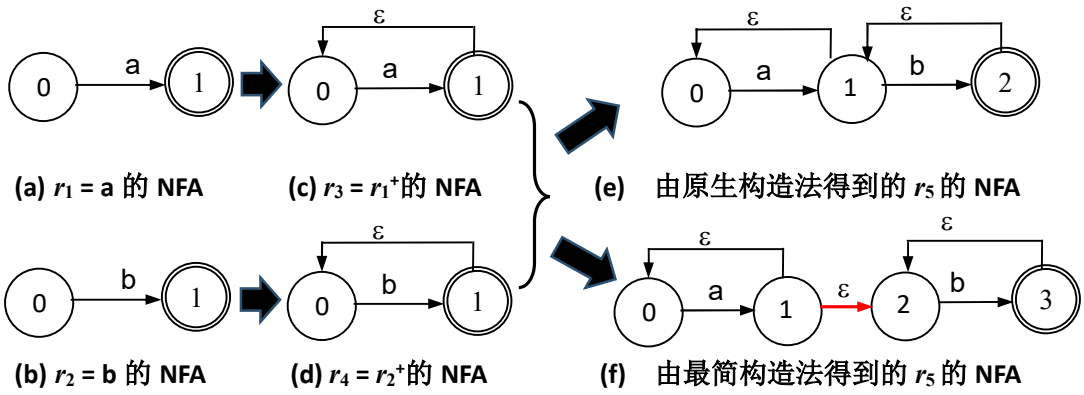
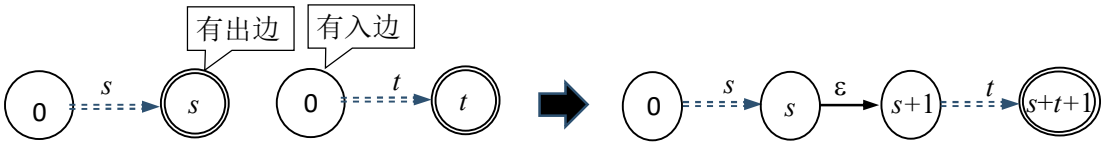


图 2.14 正则运算的 NFA 构建示例

对于连接运算 $s \cdot t$，其最简 NFA 由正则表达式 s 的 NFA 和正则表达式 t 的 NFA 结合而成。要将其区分成两种情形来考虑，如图 2.15 所示。只有在 s 的 NFA 的结束状态有出边，且 t 的 NFA 的开始状态有入边时，需要引入一个空转换边来消除倒灌，如图 2.15(a)所示。在其他情形下，不会出现倒灌，构造方法和原生构造法一样，如图 2.15(b)所示。



（a）s 的 NFA 的结束状态 s 有出边且 t 的 NFA 的开始状态 0 有入边

（b）其他情形

图 2.15 联接运算的最简 NFA 构造法

在连接运算 $s \cdot t$ 的最简 NFA 构建中，要对输入的两个 NFA 重新编排状态序号，保证结果 NFA 中每个状态的序号唯一，而且是连续编排。也要保证结果 NFA 的开始状态的序号为 0，结束状态的序号最大。

对于闭包运算 s^*，其最简 NFA 由正则表达式 s 的 NFA 得出。要分门别类将其区分成 4 种情形来考虑，如图 2.16 所示。对于 s 的 NFA，当其开始状态有入边，且其结束状态有出边时，要采用原生构造法构建。在该情形下，引入的空转换最多，为 4 个。在其他情形下，空转换数可以减少。每种情形的减少数不一样。对于 s 的 NFA，如果其开始状态无入边，且其结束状态无出边时，引入的空转换数可减少 2 个，如图 2.16 的最后一种情形所示。从 4 种情形的对比可知，原生构造法是一种保守的构造法，以一概全，带来了构造的简单性，但是没有取得空转换数最少。

对于 0 个或 1 个运算 s？，其最简 NFA 由正则表达式 s 的 NFA 得出。和闭包运算一样，要对 s 的 NFA 区分成四种情形来分别考虑，如图 2.17 所示。

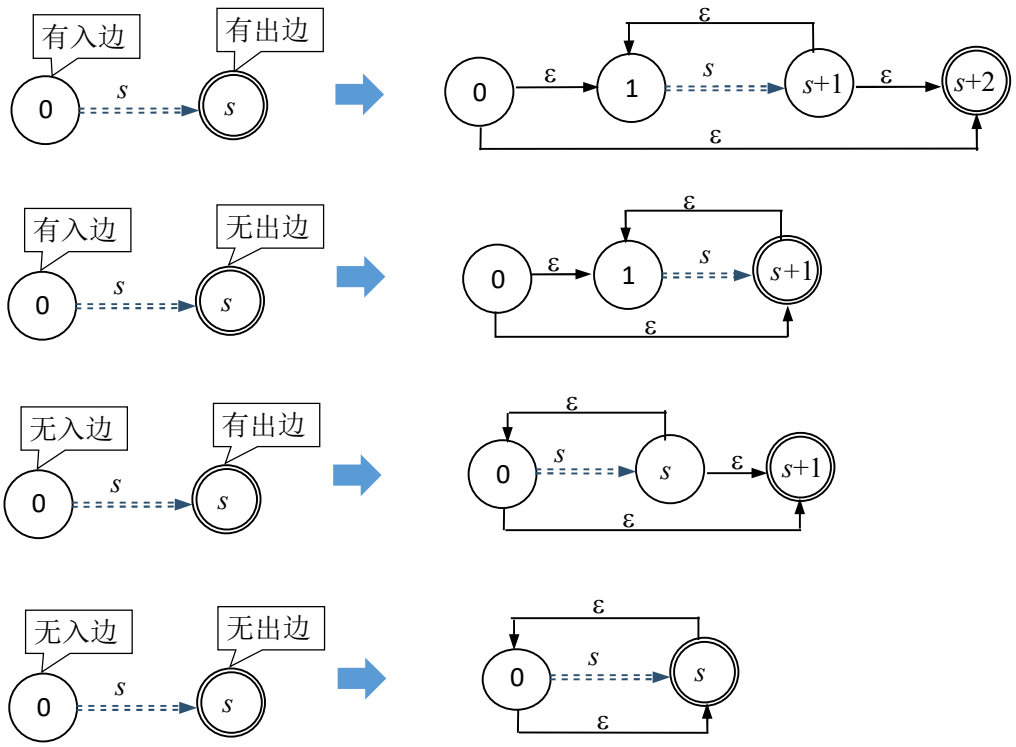


图 2.16 闭包运算的最简 NFA 构造中的四种情形

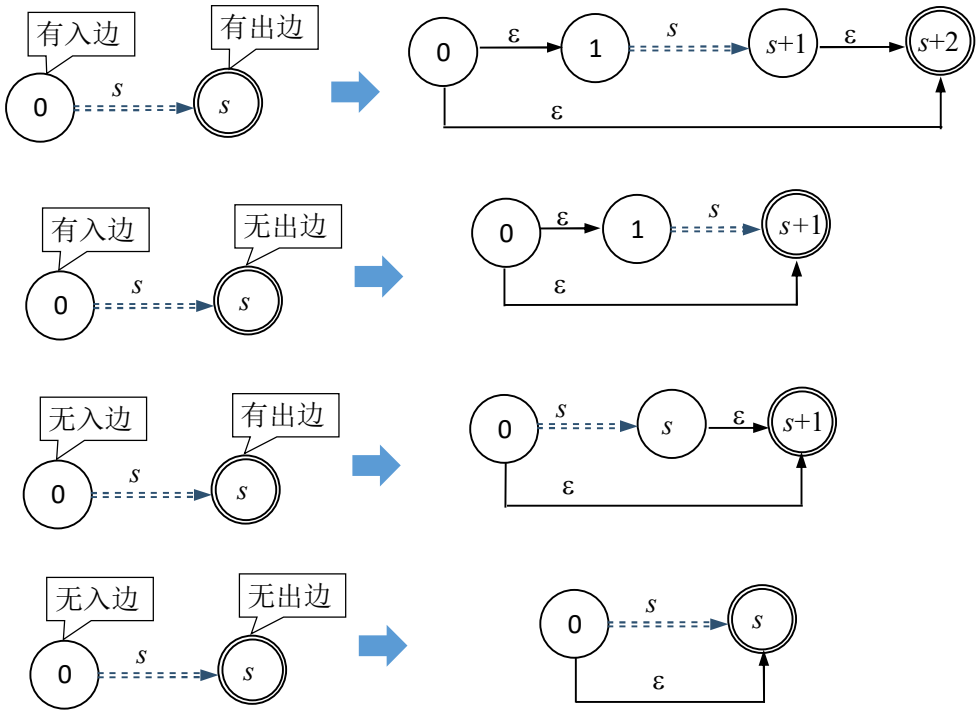


图 2.17 0 个或者 1 个运算的最简 NFA 构造中的四种情形

对于并运算 $s \mid t$，其最简 NFA 由正则表达式 s 的 NFA 和正则表达式 t 的 NFA 组合得出。对于 s 的 NFA 和 s 的 NFA，当它们的开始状态都无入边，结束状态都无出边且 category 属性值都为空时，并运算的结果 NFA 如图 2.18 所示。从其可知，组合中无须引入空转换边。结果 NFA 中共有 $s+t$ 个状态，开始状态序号为 0，结束状态序号为 $s+t-1$。组合中，要重新编排状态序号，保证结果 NFA 中每个状态的序号唯一，而且是连续编排。其中对于 s 的 NFA，仅只须将 s 状态的序号改为 $s+t-1$，其他的保持不变。对于 t 的 NFA，除了 0 状态之外的其他状态，序号都要修改。具体来说，从 1 状态至 t 状态，每个状态的序号都加上 $s-1$。

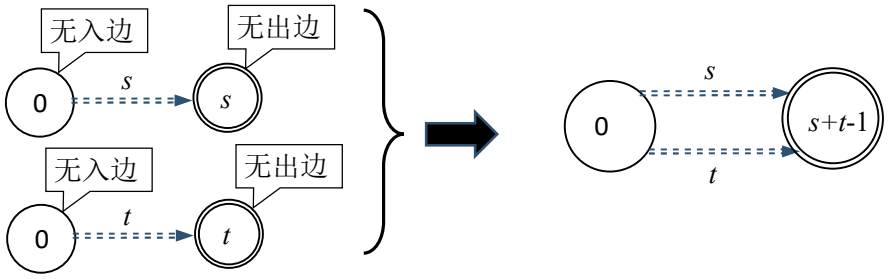


图 2.18 并运算的最简 NFA 构造法

当 s 的 NFA 的开始状态有入边，或者结束状态有出边，或者结束状态的 category 属性值不为空时，就要先对其改造，然后再去参与并运算。改造分两步。第 1 步改造是针对开始状态有入边，或者结束状态有出边。这种改造包含有 3 种子情形，如图 2.19 情形中，第 1 种和第 3 种子情形会导致状态序号要重新编排。原有状态的序号因此会发生改变，但其 category 属性值保持不变。对于 t 的 NFA 也是如此。

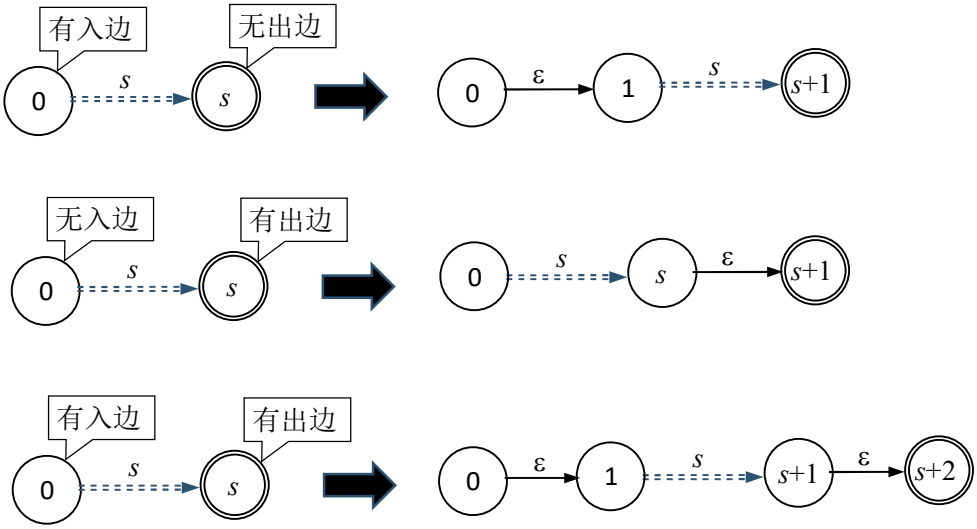


图 2.19 并运算之前对其分量 NFA 的等价改造

在第 1 步改造之后，如果 s 的 NFA 的结束状态，其 category 属性值不为空，那么其结束状态要接一个空转换，以便保留其标志性。改造情形如图 2.20 所示。category 属性值不为空的状态都是标志状态，在并运算的组合中不能和其他状态合并。这就是要做该改造的缘由。对于此情形，意味着在第 1 步改造前，s 的 NFA 的结束状态肯定无出边。否则第 1 步改造之后，s 的 NFA 的结束状态后已经增加了一个空转换，新的结束状态的 category 属

性值肯定为空。对于 t 的 NFA 也要如此。

对并运算的分量执行上述两步改造之后，再按照图 2.18 所示方法得出并运算的结果 NFA。在 s 和 t 的 NFA 的开始状态都有入边，结束状态都有出边或者 category 属性值都不为空时，并运算的结果 NFA 如图 2.21 所示。这就是原生构造法。由此可知，原生构造法是一种保守的构造法，以一概全，带来了构造的简单性，但是没有取得空转换边的数量最少。最简 NFA 构造法的区别在于，依据具体情形分门别类考虑，消除不必要的空转换边，使得结果 NFA 最简。

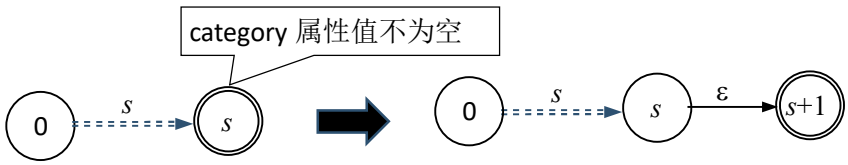


图 2.20 并运算之前对其分量 NFA 的第 2 步等价改造

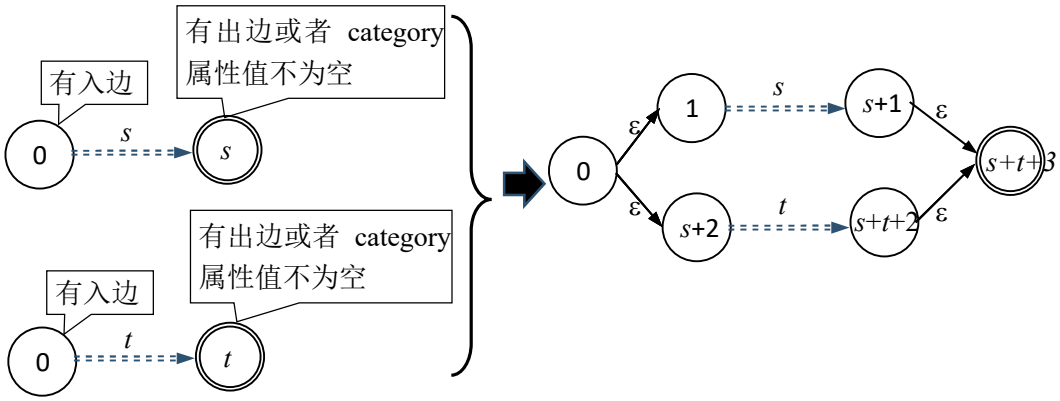


图 2.21 并运算的 NFA 构造中空转换增加最多的情形

对于闭包运算 s^*，当 s 的 NFA 只含两个状态，且开始状态无入边，结束状态无出边时，可进一步化将结果 NFA 由两个状态化简成一个状态，将两个空转换边消除。最终结果 NFA 如图 2.22 所示。

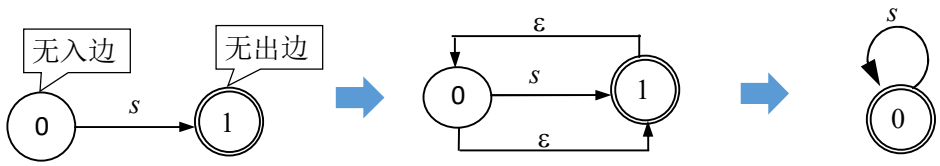
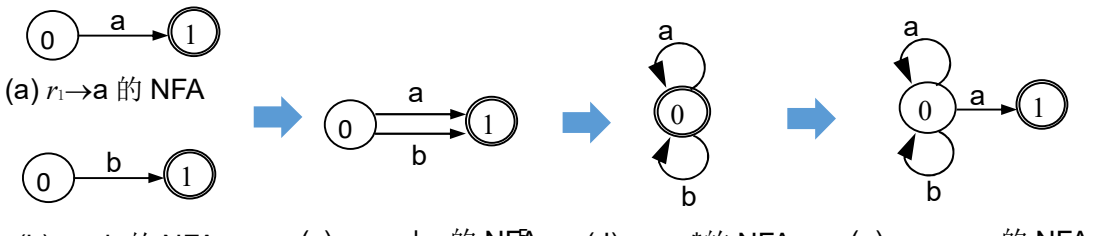


图 2.22 特殊情形下闭包运算的最简 NFA 构建

最简 NFA 构建法则具有动态性和量体裁衣性，能够使得构造出来的 NFA 具有最少的空转换边，从而使得所获的 DFA 具有简单性。下面举例说明。对于正则表达式(a|b)*a，其含义可理解为如下 5 个正则表达式：$r_1 \rightarrow$a；$r_2 \rightarrow$b；$r_3 \rightarrow r_1 \mid r_2$；$r_4 \rightarrow r_3^*$；$r_5 \rightarrow r_4 \cdot r_1$。根据上述运算的 NFA 原生构造法，这 5 个正则表达式的 NFA 分别如图 2.23 所示。



(a) $r_1 \rightarrow$a 的 NFA　(b) $r_2 \rightarrow$b 的 NFA　(c) $r_3 \rightarrow r_1 \mid r_2$ 的 NFA　(d) $r_4 \rightarrow r_3^*$ 的 NFA　(e) $r_5 \rightarrow r_4 \cdot r_1$ 的 NFA

图 2.23 正则表达式(a|b)*a 的最简 NFA 构建过程

2. 特殊正则表达式的最简 NFA 构造

多行注释的前缀为“/*”, 后缀为“*/”，中间内容的正则表达式为：

$r \rightarrow$ (character - ‘*’)* · (‘*’)$^{+}$ · ((character - ‘*’ - ‘/’) · (character - ‘*’)* · (‘*’)$^{+}$)*

这个表达式的来历已在 2.2 节讲解。使用最简 NFA 构造法，得到其 NFA 如图 2.24 所示。图中将 character 简化成了 c。

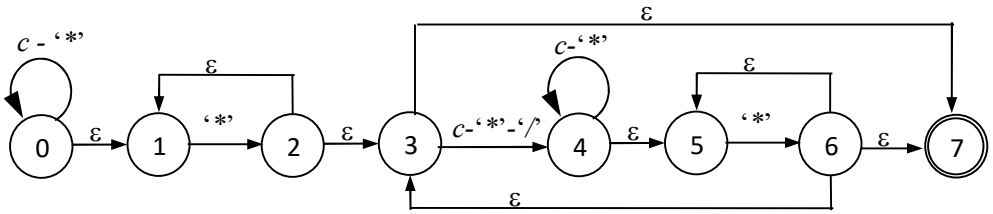


图 2.24 多行注释中间内容的最简 NFA

对该 NFA，使用子集构造法，得到的到 DFA 状态转换表 DTran 如表 2.2 所示。

表 2.2 多行注释中间内容的 DFA 状态转换表 DTran

NFA 中对应的状态集合	状态集合序号（即 DFA 状态序号）	驱动字符对应的下一状态集合序号		
		c - ‘*’	‘*’	c - ‘*’- ‘/’
{0, 1}	0	0	1	
{1, 2, 3, 7}	1		1	2
{4, 5}	2	2	3	
{3, 5, 6, 7}	3		3	2

将上述状态转换表，以状态转换图形式画出，得到的 DFA 如图 2.25 所示。

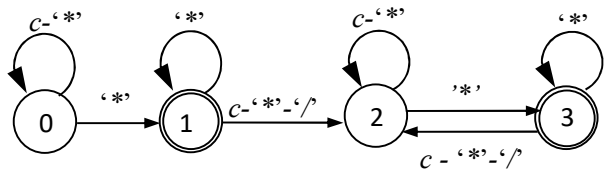


图 2.25 由子集构造法得到的 DFA

对于图 2.24 所示的 NFA，人为观察，可简化成如图 2.26 所示的 NFA。对该 NFA，使用子集构造法，得到的 DFA 如图 2.27 所示。对比图 2.25 和图 2.27 所示的两个 DFA，可知后一个要简单很多。

对上述多行注释中间内容的正则表达式进行归纳，可得出其一般化形式为 $r \rightarrow s \cdot (t \cdot$

$s)^*$。其中 s 和 t 为正则表达式。它的 NFA 也可由 s 的 NFA 和 t 的 NFA 组合出来。组合情形如图 2.28 所示。由于 t 的 NFA 中含有 t+1 个状态，s 的 NFA 的双虚线中含有 s-1 个状态，因此结果 NFA 的状态数为 s+t 个。从而得知结果 NFA 的结束状态序号为 s+t-1。

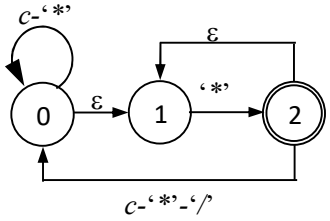


图 2.26 多行注释中间内容的等价 NFA

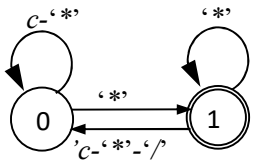


图 2.27 由子集构造法得到的最简 DFA

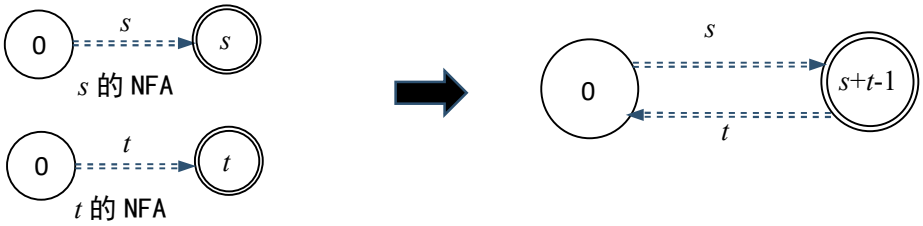


图 2.28 正则表达式 $s \cdot (t \cdot s)^*$ 的最简 NFA 构造法

为了使得结果 NFA 中的每个状态的序号唯一，且连续编排，要对 t 和 s 的 NFA 中状态的序号重新布局。从结果 NFA 可知，对 t 的 NFA，其结束状态的序号，要由 t 改为 0，而开始状态的序号则要由 0 改为 s+t-1。为此先将 t 状态与 0 状态的序号对换。这种对换并不改变 t 的逻辑。对换方法是：先将 0 状态的序号改为 t+1，将 0 序号腾空出来，然后把 t 状态的序号改为 0，再把 t+1 状态的序号为 t，于是就实现了它们序号的互换。此时，t 的 NFA 中，t 状态是开始状态，0 状态是结束状态。再将 t 的 NFA 中开始状态的序号由 t 改为 s+t-1，即结果 NFA 的结束状态序号。现在 t 的 NFA，状态序号已全部重新布局完毕，占用的序号段为 0 至 t-1，以及 s+t-1。因此 s 的 NFA 能用的状态序号段只能为 t 至 s+t-1。对于 s 的 NFA，开始状态的序号保持不变，还是为 0。其他从 1 至 s 的状态序号都要加上 t-1，使其变为从 t 至 s+t-1。

思考题 2-13：对于 s 的 NFA，其 1 状态至 s 的状态，序号都要加上 t-1，使其变为从 t 至 s+t-1。修改时不能顺着从 1 状态开始改，直至 s 状态，而要从 s 状态开始，倒着改，直至 1 状态。为什么？请结合上述状态序号重新布局方案，对多行注释前后缀之间夹着的中间内容，根据其正则表达式，看能否得出图 2.26 所示的最简 NFA？

就正如闭包运算一样， $s \cdot (t \cdot s)^*$也是一个复合运算表达式，给它取一个运算名，将其提升为基本运算。暂且将这个运算称作残正闭包运算。记作 s ^ t。残的意思是表达式中第一个 s 前面少了一个 t，如果有，那么就是正闭包了。

3. NFA 和 DFA 中状态属性值的确定方法

由 NFA 而得的 DFA 中，每个状态都有 type 属性。对于 type 属性值为 MATCH 的状态，它还有 category 属性。这两个属性的含义已在 2.3 节讲解。当使用 NFA 原生构建法，或者最简 NFA 构建法来构建一个正则表达式的 NFA 时，每一个 NFA 有且仅有一个状态(即结束

状态)，其 type 属性值为 MATCH。其他状态的 type 属性值都为 UNMATCH。因此，对于 NFA 来说，不用考虑状态的 type 属性值，只需要在 NFA 中记录它的结束状态序号即可。

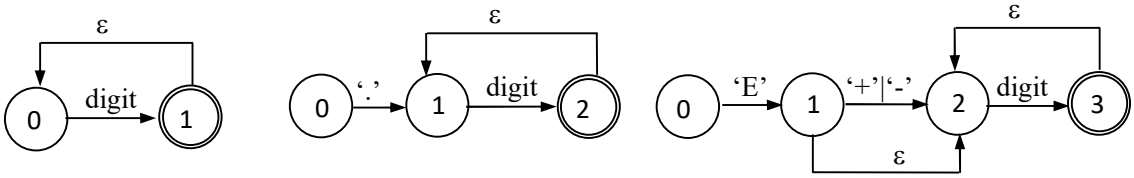
开始状态序号和结束状态序号都是 NFA 的重要属性。给 NFA 中状态分配序号时，将其开始状态序号设为 0，所有状态的序号连续编排，使结束状态的序号最大。如此处理之后，对于每种正则运算，按照最简 NFA 构造法，其结果 NFA 中状态数都能算出来。于是也就知道了其结束状态的序号。因此，在 NFA 构建中并不用关心状态的 type 属性，只须明确结束状态序号即可。

正则表达式有 category 属性。在定义正则表达式时，要给其确定 category 属性值。例如，假定在一门高级程序语言中，数值常量要区分整数常量，实数常量，和科学计数法常量。另外还有变量，预留字等概念。为了满足这一要求，就要定义如下 7 个正则表达式：

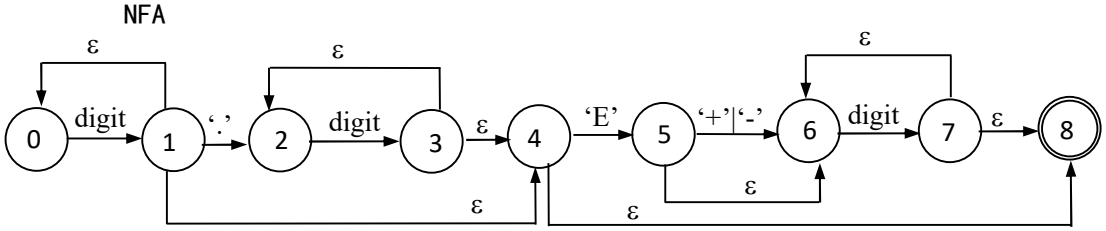
① int→ ‘i’ · ‘n’ · ‘t’
② if → ‘i’ · ‘f’
③ id → letter^{+}
④ integerConst → digit^{+}
⑤ optionalFraction → ‘.’ · digits
⑥ optionalExponent → ‘E’ · (‘+ ’ | ‘-’)? · digits
⑦ numberConst → IntegerConst · optionalFraction? · optionalExponent?

并且对前 6 个正则表达式的 category 属性值，分别设置为 RESERVED，RESERVED，ID，INTERGER_CONST，FLOAT_CONST，和 SCIENTIFIC_CONST。第 7 个正则表达式的 category 属性值不用定义，让其为空。

当构造出一个正则表达式的 NFA 时，将其 NFA 中结束状态的 category 属性值设为该正则表达式的 category 属性值。当一个正则表达式参与某个正则运算时，其 NFA 中每个状态的 category 属性值要带入到正则运算的结果 NFA 中。例如，对于上述第④至第⑦的 4 个正则表达式，使用最简 NFA 构造法分别构建其 NFA，其结果分别如图 2.29 (a)，(b)，(c)，(d)所示。图 2.29(a)所示 NFA，其结束状态 1 的 category 属性值为 INTERGER_CONST，来自正则表达式 IntegerConst 的定义。图 2.29(b)所示 NFA，其结束状态 2 的 category 属性值为 FLOAT_CONST，来自正则表达式 optionalFraction 的定义。图 2.29(c)所示 NFA，其结束状态 3 的 category 属性值为 SCIENTIFIC_CONST，来自正则表达式 optionalExponent 的定义。



(a) **IntegerConst** 的 NFA　(b) **optionalFraction** 的 NFA　(c) **optionalExponent** 的 NFA



(d) **numberConst** 的 NFA

图 2.29 四个正则表达式的简化 NFA

图 2.29(d)所示正则表达式 numberConst 的 NFA 由前 3 个 NFA 组合所得。它有 3 个状态的 category 属性的值不为空。其中 1 状态的 category 属性值为为 INTERGER_CONST，由第 1 个 NFA 带入；3 状态的 category 属性值为 FLOAT_CONST，由第 2 个 NFA 带入；7 状态的 category 属性值为 SCIENTIFIC_CONST，由第 3 个 NFA 带入。

用子集构造法将 NFA 转化为 DFA 后，DFA 中状态的 type 属性值和 category 属性值按照如下方法确定。对于 DFA 的状态 i，设它在 NFA 中对应的状态集合为 α_i。如果集合 α_i 中包含了 NFA 的结束状态，那么 DFA 的状态 i 的 type 属性值就为 MATCH。如果集合 α_i 包含有一个状态，其 category 属性值不为空，那么它就是状态 i 的 category 属性值。

现以图 2.29(d)所示 NFA 为例来展示 DFA 状态属性的确定方法。根据子集构造法得出的 DFA 状态转换表 DTran 如表 2.3 所示。

表 2.3 正则表达式 **numberConst** 的状态转换表

NFA 中对应的状态集合	状态集合序号 (即 DFA 状态序号)	驱动字符下的目标状态序号			
		digit	‘.’	‘E’	‘+’|‘-’
{**0**}	0	1			
{0,**1**,4,8}	1	1	2	3	
{**2**}	2	4			
{**5**,6}	3	6			5
{2,**3**,4,8}	4	4		3	
{**6**}	5	6			
{6,**7**,8}	6	6			

将该状态转换表以状态转换图形式画出，得到的 DFA 如图 2.30 所示。这就是图 2.3 所示状态转换图上面部分的由来。

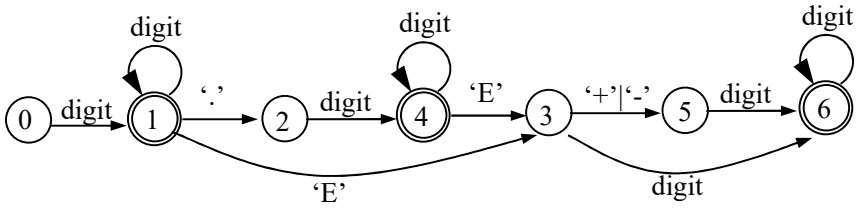


图 2.30 由正则表达式 **numberConst** 的 NFA 得出的 DFA

从状态转换表的第 1 列（即 NFA 中对应的状态集合）来看，NFA 的结束状态 8 出现在状态集合序号为 1, 4, 6 的行中。因此，DFA 的 1，4，6 状态的 type 属性值为 MATCH。

再来看 category 属性，NFA 中的 1，3，7 这三个状态的 category 属性值不为空。这 3 个状态出现在集合序号为 1, 4, 6 的状态集合中。因此，DFA 中 1，4，6 状态的 category 属性值不会为空。接下来求这 3 个状态的 category 属性值。

表中集合序号为 1 的状态集合为{0, 1, 4, 8}，其中 1 状态在 NFA 中的 category 属性值

INTERGER_CONST，其他的都为空。于是 DFA 中 1 状态的 category 属性值就为 INTERGER_CONST。表中集合序号为 4 的状态集合为{2, 3, 4, 8}，其中 3 状态在 NFA 中的 category 属性值为 FLOAT_CONST，其他的为空。于是 DFA 中 4 状态的 category 属性值为 FLOAT_CONST。表中集合序号为 6 的状态集合为{6, 7, 8}，其中 7 状态在 NFA 中的 category 属性值为 SCIENTIFIC_CONST，其他的为空。于是 DFA 中 6 状态的 category 属性值为 SCIENTIFIC_CONST。

4. 正则表达式之间的包含关系

两个正则表达式所表达的集合可能具有包含关系，即一个集合是另一个集合的真子集。例如，如下的四个正则表达式：

(1) int→ ‘i’ · ‘n’ · ‘t’
(2) if → ‘i’ · ‘f’
(3) id → letter^{+}
(4) keyword&id → id | if | int

其中正则表达式 id 就包含了正则表达式 int 和 if，因为输入串 int 和 if 都匹配 id 正则表达式。int 和 if 是描述预留词 int 和 if 的正则表达式，而 id 是描述变量的正则表达式。因此，正则表达式 int 和 if 的 category 属性值都为 KEYWORD，而 id 的 category 属性值为 ID。正则表达式 keyword&id 则描述了变量和两个预留字。

按照最简 NFA 构造法，得出正则表达式 id，if 和 int 的 NFA 如图 2.31 的左边部分所示。id 的 NFA 中，1 状态的 category 属性值为 ID。if 的 NFA 中，2 状态的 category 属性值为 RESERVED。int 的 NFA 中，3 状态的 category 属性值为 KEYWORD。正则表达式 keyword&id 为 int，if 和 id 三者做并运算。按照最简 NFA 构造法，组合前要对三个分量 NFA 分别作改造。改造后的三个 NFA 分量如图 2.31 的右边部分所示。其中 id 的 NFA 添加了两个空转换，因为开始状态有入边，结束状态有出边，另结束状态的 category 属性值不为空。而 if 和 int 的 NFA 的右边分别添加了一个空转换，其原因是它们的结束状态的 category 属性值不为空。改造之后，因为状态序号的重新编排，id，if 和 int 三者的 NFA 中，分别是 2 状态，2 状态，3 状态的 category 属性值不为空。

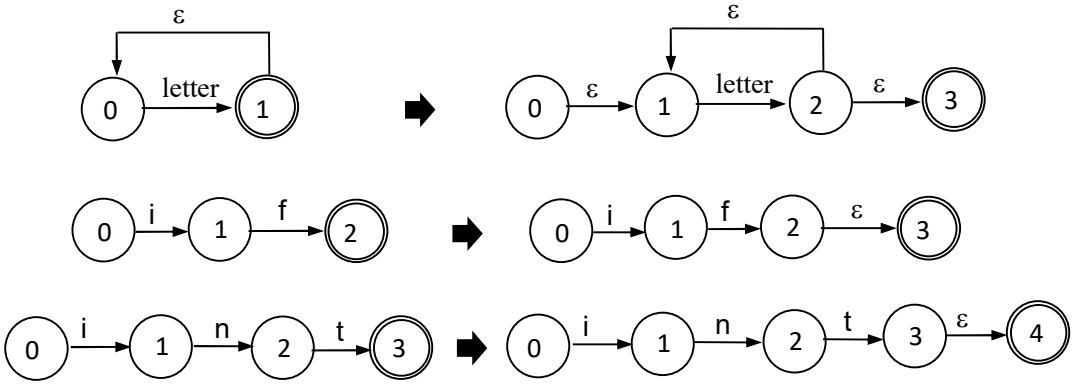


图 2.31　参与并运算的三个正则表达式的 NFA 及其改造结果

按照最简 NFA 构造法，得出正则表达式 keyword&id 的 NFA 如图 2.32 所示。状态序

号重新编排后，keyword&id 的 NFA 中，2 状态，4 状态，7 状态的 category 属性值不为空，分别为 ID，KEYWORD，KEYWORD。

用子集构造法从 keyword&id 的 NFA 得出其 DFA。NFA 的开始状态集合为{0,1}。该例有一个特殊性的地方是：0 状态的实出边的驱动字符为‘i’；1 状态的实出边的驱动字符为 letter，其中包含了字符‘i’。因此它们存在交集。DFA 的一个最基本要求是：对于一个状态的多条出边，它们的驱动字符不能存在交集。于是要将驱动字符分为‘i’和 letter -‘i’，使其不相交。

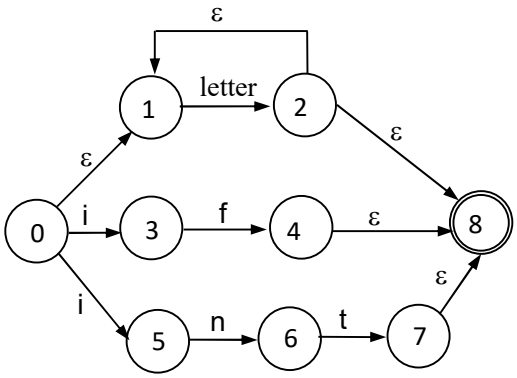


图 2.32 正则表达式 keyword 和 id 的 NFA

首先穷举 NFA 开始状态集合的出边，驱动字符，以及下一状态集合。ε_closure({0}) = {0, 1}。 驱动字符设为‘i’和 letter - ‘i’。用 move 函数和 ε_closure 函数分别求 DTran 函数值，即可得出每一转换的下一状态集合：

move({0,1}, ‘i’) = {2,3,5}; ε_closure({2,3, 5}) = {1, 2, 3 ,5, 8};
move({0,1}, letter - ‘i’) = {2}; ε_closure({2}) = {1, 2, 8};

注意：因 letter 中含有字符‘i’，因此 2 状态在 move({0,1}, ‘ i’)的结果中。从 move 结果可知，这两个下一状态集合与前面已知的状态集合（即开始状态集合）不相同。因此它们都是新状态集合，给它们分别分配集合序号 1 和 2。

接下来对 1 号状态集合进行穷举。从 {1, 2, 3 ,5, 8}中包含的状态可知，在 1 状态上有一条实出边，其驱动字符为 letter。在 3 状态上有一条实出边，其驱动字符为‘f’。在 5 状态上有一条实出边，其驱动字符为‘n’。这三者之间存在交集。为了彼此不相交，三个驱动字符设为‘f’，‘n’，letter -‘f’ - ‘n’。这三个驱动的下一状态集合分别为：

move({1, 2, 3 ,5, 8}, ‘f’) = {2, 4}; ε_closure({2, 4}) = {1, 2, 4, 8};
move({1, 2, 3 ,5, 8}, ‘n’) = {2, 6}; ε_closure({2, 6}) = {1, 2, 6, 8};
move({1, 2, 3 ,5, 8}, letter -‘f’ - ‘n’) = {2};

从 move 结果可知，只有前二个下一状态集合为新状态集合，分别给其分配集合序号 3 和 4。第 3 个为已有的序号为 2 的状态集合。

接下来对序号为 2 的状态集合进行穷举。从 {1, 2, 8}中包含的状态可知，只有在 1 状态上有一条实出边，其驱动字符为 letter。其下一状态集合为：move({1, 2, 8}, letter) = {2}。它为已有的序号为 2 的状态集合。

接下来对 3 号状态集合进行穷举。从 {1, 2, 4, 8}中包含的状态可知，只有在 1 状态上

有一条实出边，其驱动字符为 letter。其下一状态集合：move({1, 2, 4, 8}， letter) = {2}。它是已有的 2 号状态集合。

接下来对 4 号状态集合进行穷举。从 {1, 2, 6, 8}中包含的状态可知，在 1 状态上有一条实出边，其驱动字符为 letter。在 6 状态上有一条实出边，其驱动字符为‘t’。两者之间存在交集。为了彼此不相交，两个驱动字符设为‘t’和 letter -‘t’。它们的下一状态集合分别为：

move({1, 2, 6, 8}， ‘t’) = {2, 7}； ε_closure({2, 7}) = {1, 2, 7, 8}；
move({1, 2, 6, 8}， letter - ‘t’) = {2}；

从 move 结果可知，只有第一个下一状态集合为新状态集合，给其分配集合序号 5。

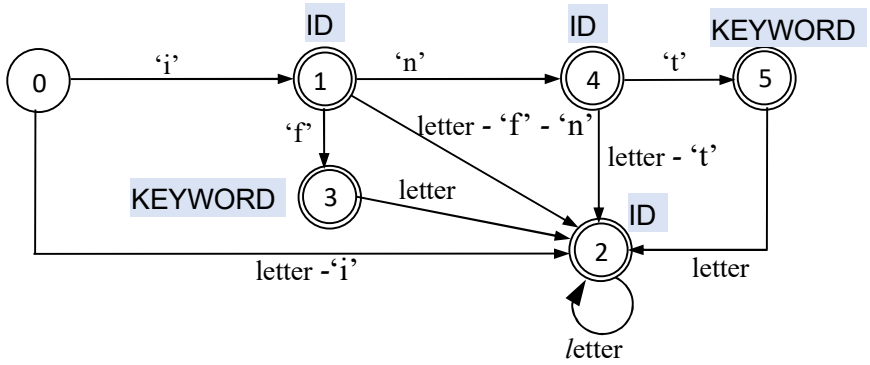
接下来对 5 号状态集合进行穷举。从 {1, 2, 7, 8}中包含的状态可知，只有在 1 状态上有一条实出边，其驱动字符为 letter。其下一状态集合为：move({1, 2, 7, 8}， letter) = {2}。它为已有的 2 号状态集合。

至此已全部穷举出来。将上述穷举结果汇总到 DFA D 状态转换表 DTran 中，如表 2.4 所示。

表 2.4 keyword&id 的 DFA 状态转换表 DTran

NFA 中对应的状态集合	状态集合序号(即 DFA 状态序号)	驱动字符对应的下一状态集合序号							
		‘i’	‘n’	‘f’	‘t’	letter - ‘i’	letter - ‘f’- ‘n’	letter - ‘t’	letter
{0,1}	0	1				2			
{1, **2**, 3 ,5, 8}	1		4	3			2		
{ 1, **2**, 8}	2								2
{ 1, **2**, **4**, 8}	3								2
{1, **2**, 6, 8}	4				5			2	
1, **2**, **7**, 8	5								2

将上述状态转换表以状态转换图形式画出，得到的 DFA 如图 2.33 所示。



2.33 正则表达式 keyword&id 的 DFA

接下来识别 DFA 中的哪些状态的 type 属性值为 MATCH，然后求这些状态的 category 属性值。NFA 的结束状态序号为 8。从转换表可知，集合序号为 1, 2, 3, 4, 5 的这五个集合中都含有 NFA 中的 8 状态。因此，DFA 中序号为 1, 2, 3, 4, 5 的这五个状态的 type 属性值

为 MATCH。

再来看 category 属性。NFA 中的 2, 4, 7 这三个状态的 category 属性值不为空。集合序号分别为 1, 2, 4 的状态集合分别为{1, 2, 3 ,5, 8}，{1, 2, 8}，{1, 2, 6, 8}。这三个集合中都含 NFA 的 2 状态，不含 4 状态和 7 状态。而 NFA 中 2 状态的 category 属性值为 ID。因此 DFA 中 1, 2, 4 这三个状态的 category 属性值都为 ID。集合序号为 3 的状态集合为{1, 2, 4, 8}，其中含有 2 个 category 属性值不为空的 NFA 状态（2 和 4）。2 和 4 的 category 属性值分别为 ID 和 KEYWORD。这就是正则表达式之间的包含关系被暴露出来的具体体现。DFA 中 3 状态的 category 属性值取为 RESERVED。DFA 中 5 状态也是如此。标明了 Type 和 category 属性值的 DFA 如图 2.33 所示。

正则表达式之间的包含关系可通过它们排列的前后关系来暗示。把正则表达式 int 和 if 排在 id 的前面，以此表明一旦出现某个集合中含有多个 category 属性值时，就优先选择排在前面的正则表达式的 category 属性值。

5. NFA 和 DFA 构造中所涉及的数据结构

2.4 节讲解了正则运算式的 NFA 构造方法，以及 DFA 的生成方法。要编程实现上述方法，首先要设计出正则运算式，NFA 的数据结构。其中最基本的概念是字符和字符集，以及正则运算。

正则运算式的数据结构定义如下：

```
class regularExpression {
    int regularId;
    String name;
    char operatorSymbol;   //正则运算符，共有 7 种：‘=’, ‘~’，‘-’, ‘|’，‘.’, ‘*’，  ‘+’，  ‘?’
    int operandId1;     //左操作数
    int operandId2;     //右操作数
    OperandType type1;   //左操作数的类型
    OperandType type2;   //右操作数的类型
    OperandType resultType;   //运算结果的类型
    LexemeCategory category;          //  词的 category 属性值
    Graph *pNFA;            //对应的 NFA
}
```

正则运算表的定义为：List <regularExpression *> *pRegularTable;

其中 OperandType 为枚举类型，其取值有三种：CHAR，CHARSET，REGULAR，分别表示字符，字符集，正则表达式。对于第 1 个操作数（也叫左操作数），如果其类别为字符（即字段 type1 取值为 CHAR）时，operandId1 字段的值为字符在 ASC 码表中的序号；如果其类别为字符集（即字段 type1 取值为 CHARSET）时，operandId1 的值为字符集的 id。字符集的定义在后面给出。如果其类别为正则表达式（即字段 type1 取值为，REGULAR）时，operandId1 的值为正则运算式的 id，即正则运算表中另一行的行 id。右操作数也是如此。

LexemeCategory 指词的类别，也为枚举类型，其取值已在 2.3.2 小节给出。

对于一元运算，就没有右操作数概念，即 operandId2 和 type2 这两个字段的值都为 null。

以正则表达式(a|b)*abb 为例，它在正则运算表 pRegularTable 中便有如下 5 行数据：

regularId	name	operatorSymbol	operandId1	operandId2	type1	type2	resultType
1	r1	|	'a'	'b'	CHAR	CHAR	CHARSET
2	r2	*	1		CHARSET		REGULAR
3	r3	.	2	'a'	REGULAR	CHAR	REGULAR
4	r4	.	3	'b'	REGULAR	CHAR	REGULAR
5	r5	.	4	'b'	REGULAR	CHAR	REGULAR

注：表中最后两个字段没有给出。

字符集的数据结构定义如下：

```
class CharSet {
    int indexId;        //字符集 id
    int segmentId;     //字符集中的段 id。一个字符集可以包含多个段
    char fromChar;   //段的起始字符
    char toChar;   //段的结尾字符
}
```

字符集表的定义如下：List <charSet *> *pCharSetTable;

注意：一个字符集可以只包含一段字符，也可包含多个段。例如，letter | digit | '_' 就包含 4 段字符。第 1 段为'a'至'z'，第 2 段为'A'至'Z'，第 3 段为'0'至'9'，第 4 段为'_'至'_'。

字符与字符之间的运算有两种：①范围运算（运算符为'~'），例如'a'~'z'；②并运算（运算符为'|'），例如'a'|'b'。这两种运算的结果都是一个新的字符集对象。字符集和字符之间也有差运算和并运算，其运算结果也都是一个新的字符集对象，例如 letter -'i'和 letter | '_'。字符集和字符集之间有并运算，其运算结果是一个新的字符集对象，例如 letter | digit。

NFA 和 DFA 的数据结构相同，统称为图，其定义如下：

```
class Graph {
    int graphId;
    int numOfStates;
    List <Edge *>*pEdgeTable;
    List <State *>*pStateTable;
}
```

```
class Edge {
    int fromState;
    int nextState;
    int driverId;
    DriverType type;
}
```

```
class State {
    int stateId;
    StateType type;
    LexemeCategory
category;
}
```

其中 DriverType 为枚举类型，其取值有三种：NULL，CHAR，CHARSET，分别表示空字符 ε，字符，字符集。StateType 和 LexemeCategory 也都为枚举类型。StateType 的取值有两种：MATCH 和 UNMATCH。LexemeCategory 的取值已在前面给出。

对于 NFA，其状态表中只需存储 category 属性值不为空的那些状态，其他状态不需要存储。另外，在 NFA 中，只有结束状态的 type 属性值为 MATCH，其他状态的 type 属性

值都为 UNMATCH。在 DFA 中，则是只有 type 属性值为 MATCH 的状态，其 category 属性值才不为空。

6. 实验任务

1. 基于上述数据结构的定义，针对字符集的创建，实现如下函数：

```
int range (char fromChar, char toChar);   // 字符的范围运算
int union(char c1, char c2);   // 字符的并运算
int union(int charSetId, char c);// 字符集与字符之间的并运算
int union(int charSetId1,int charSetId2);//字符集与字符集的并运算
int difference(int charSetId, char c);   // 字符集与字符之间的差运算
```

这 5 个函数都会创建一个新的字符集对象，返回值为字符集 id。创建字符集，表现为往字符集表中添加新的行。当一个字符集包含多个段时，便会在字符集表中有多行，一行记录一段。

2. 基于上述 NFA 的数据结构定义，请按照最简 NFA 构造法，实现如下函数：

```
Graph * generateBasicNFA(DriverType driverType，int driverId );
Graph * union(Graph *pNFA1, Graph *pNFA2);        // 并运算
Graph * product(Graph *pNFA1, Graph *pNFA2);     // 连接运算
Graph * plusClosure(Graph *pNFA)                  //正闭包运算
Graph * closure(Graph *pNFA)                      // 闭包运算
Graph * zeroOrOne(Graph *pNFA);                  // 0 或者 1 个运算。
```

其中第 1 个函数 generateBasicNFA 是针对一个字符或者一个字符集，创建其 NFA。其 NFA 的基本特征是：只包含两个状态（0 状态和 1 状态），且结束状态（即 1 状态）无出边。后面 5 个函数则都是有关 NFA 的组合，分别对应 5 种正则运算，创建一个新的 NFA 作为返回值。

3. 针对上述 NFA 的数据结构定义，实现如下函数：

(1) 子集构造法中的 3 个函数：move，ε_closure，DTran；

(2) 将 NFA 转化为 DFA 的函数：

```
Graph * NFA_to_DFA(Graph *pNFA);
```

在这个函数的实现代码中，会创建一个 DFA，作为返回值。

4. 实现了上述函数之后，请以正则表达式(a|b)*abb 来测试，检查实现代码的正确性。然后再以 TINY 语言的词法来验证程序代码的正确性，得出 TINY 语言的词法的 DFA；

试验二、上下文无关文法的 DFA 构建

1. 语法分析表构造中所涉及的数据结构

要编程实现语法分析表的构造，先要对其中有关概念定义其数据结构。语法分析中最基本的概念是文法符。文法符有终结符和非终结符两种。对于非终结符，必有描述其构成的产生式。文法符的数据结构定义如下：

```
class GrammarSymbol   {            //文法符
    String name;                   //名字
    SymbolType type;               //  文法符的类别
}
```

SymbolType 为枚举类型，取值有三种：TERMINAL（终结符），NONTERMINAL（非终结符），NULL（ε）

```
class TerminalSymbol :public GrammarSymbol {            //终结符
    LexemeCategory category;        //  终结符的词类
}
```

LexemeCategory 为枚举类型，其取值见第 2 章中 NFA 和 DFA 构造中所涉及的数据结构。

```
class NonTerminalSymbol : public GrammarSymbol    {      //非终结符
    List <Production *>    *pProductionTable;          //有关非终结符构成的产生式
    int numOfProduction;                               //产生式的个数
    Set <TerminalSymbol *>    *pFirstSet;              //非终结符的 FIRST 函数值
    Set <TerminalSymbol *>    *pFollowSet;             //非终结符的 FOLLOW 函数值
    Set <NonTerminalSymbol *> *pDependentSetInFollow;
}
```

注：求非终结符的 FOLLOW 函数值时，集合 pDependentSetInFollow 存放所依赖的非终结符。

产生式的数据结构定义如下：

```
class Production   {              //产生式
    int productionId;                 //产生式序号，起标识作用
    int bodySize;                     //产生式体中包含的文法符个数
    List <GrammarSymbol *>   *pBodySymbolTable;    //产生式体中包含的文法符
    Set <TerminalSymbol *>   *pFirstSet;           //产生式的 FIRST 函数值
}
```

注：产生式体中，文法符之间都是连接运算，因此也就可省去连接运算符。把产生式中的某个文法符放入 pBodySymbolTable 前，要强制类型转换，变成 GrammarSymbol *类型。这种类型转换没有问题。因为 TerminalSymbol 和 NonTerminalSymbol 都是 GrammarSymbol 的子类。在使用 pBodySymbolTable 中元素时，检查其成员变量 type 的值，如果为 NONTERMINAL，则将其强制类型转换，变回 NonTerminalSymbol *类型。如果为 TERMINAL，则将其强制类型转换，变回 TerminalSymbol *类型。

构造语法分析表的已知条件是文法符表和开始符。其定义为：

```
List <GrammarSymbol *> *pGrammarSymbolTable;
NonTerminalSymbol *RootSymbol；
```

在 LL(1)语法分析表中，格的数据结构定义如下：

```
class Cell {
    NonTerminalSymbol *nonTerminalSymbol;
    TerminalSymbol *terminalSymbol;
    Production *production;
}
```

LL(1)语法分析表的数据结构定义为：

```
List <Cell *> *pParseTableOfLL;
```

对于 LR 文法的 DFA 构造，有 LR(0)项目，项集，状态，变迁边，DFA 这 5 个概念。它们的数据结构定义如下：

```
class LR0Item   {                    //LR(0)项目
    NonTerminalSymbol *nonTerminalSymbol;      //非终结符
    Production *production；                   //产生式
    int dotPosition；                          //圆点的位置
    ItemCategoy type;        //类型。两种：CORE(核心项)；NONCORE(非核心项)
}

class ItemSet   {                        //LR(0)项集
    int stateId;                         //状态序号
    List <LR0Item *> *pItemTable;        //LR0 项目表
}

class TransitionEdge {                   //变迁边
    GrammarSymbol *driverSymbol;         //驱动文法符
    ItemSet *fromItemSet;                //出发项集
    ItemSet *toItemSet;                  //到达项集
}
```

驱动文法符有终结符和非终结符两种。在给 driverSymbol 赋值时，要先对驱动符进行强制类型转换，变成 GrammarSymbol *类型。

```
class DFA {                              //DFA
    ItemSet *startupItemSet;             //开始项集
    List <TransitionEdge *>   *pEdgeTable;        //变迁边表
}
```

在构造 LR 文法的 DFA 时，将所有项集放在一个表中：

```
List <ItemSet *> *pItemSetTable;
```

LR(1)语法分析表包含 ACTION 和 GOTO 两个部分。它们的数据结构定义如下：

```
class ActionCell {
    int stateId;                         //纵坐标：状态序号
```

```
    String *terminalSymbolName;     //横坐标：终结符
    ActionCategory type ;           //Action 类别
    int id;                         //Action 的 id
}
```

ActionCategory 为枚举类型，取值有三种：‘r’和‘s’，以及‘a’。‘r’是规约，‘s’是移入, ‘a’是接受。当 Action 类别为规约时，id 的取值为产生式 id。当 Action 类别为移入时，id 的取值为下一状态 id。

```
class GotoCell {
    int stateId;                    //纵坐标：状态序号
    String *nonTerminalSymbolName;  //横坐标：非终结符
    int nextStateId;                //下一状态
}

List <ActionCell *> *pActionCellTable;      //LR 语法分析表的 ACTION 部分

List <GotoCell *>   *pGotoCellTable;        //LR 语法分析表的 GOTO 部分
```

如果编程语言支持字典类型，最好将 pActionCellTable 和 pGotoCellTable 定义为字典类型，而不是 List 类型。这样有利于提升查找性能。

LR 语法分析器构造工具的输出有两个内容：① LR 语法分析表；②产生式概述表。产生式概述表的数据结构定义如下：

```
class ProductionInfo {
    int indexId;                //产生式序号
    String *headName;           //头部非终结符
    int bodySize;               //产生式体中文法符的个数
}

List < ProductionInfo *>    *pProductionInfoTable;      //产生式概述表
```

2. 实验任务

1. 基于前面给出的数据结构，对文法写出下列功能函数的实现代码：
   1) 产生式的 FIRST 函数求解；
   2) 非终结符的 FIRST 函数求解；
   3) 非终结符的 FOLLOW 函数求解；
2. 基于前面给出的数据结构，就 LR 语法分析写出下列功能函数的实现代码：
   1) 一个项集中 LR(0)核心项的闭包求解，即实现函数：
      `void getClosure( ItemSet *itemSet)；`
   2) 穷举一个 LR(0)项集的变迁，其中包括驱动符的穷举，后继项集的创建，后继项集中核心项的确定，后继项集是否为新项集的判断。即实现函数：
      `void exhaustTransition(ItemSet *itemSet)；`

3) 文法的 LR(0)型 DFA 求解；

4) 文法是否为 SLR(1)文法的判断；

5) LR 语法分析表的填写；

3. 首先以算术运算表达式的文法来验证程序代码的正确性，然后再以 TINY 语言的文法来验证程序代码的正确性，并得出 TINY 语言的语法分析表。

试验三、词法分析器构造工具的实现

1. 词法分析器构造工具的实现方法

下面以词法分析器构造工具的实现为例来展示上述语法制导的翻译通用框架的应用。词法分析器构造工具（以后简称工具）的输入为一个文本文件。该文件是用正则语言写出的程序。该程序描述了某门高级程序语言（以后简称目标语言）的词法。该程序可以只含一个语句，也可以含有多个语句。每个语句都是一个命名的正则表达式。其中最后那个命名的正则表达式（即最后那个语句）描述了目标语言的词法。词法分析器构造工具的输出是输入中最后一个命名的正则表达式的 DFA，即目标语言词法的 DFA。DFA 自动构建方法已在第 2 章讲解，即首先得出正则表达式的 NFA，然后再从 NFA 得出 DFA。

因此该例中的翻译目标是由正则表达式得出其 DFA。既然工具的输入是用正则语言写出的程序，那么工具首先要基于正则语言的词法将其切分成词，然后基于正则语言的语法得出该程序的语法分析树。其中每一步规约的物理含义体现在产生式的 SDD 上。因此工具中就包含了正则语言的词法分析器和语法分析器。于是也就首先要描述正则语言的词法和语法，然后得出其词法的 DFA，以及语法的 DFA。

正则语言是一门非常简单的语言。高级程序语言的词法和语法均可用正则语言描述出来。于是描述结果自然就是用正则语言写出的一个程序。由第 2 章知识可知，一门语言的词法最终由一个正则表达式描述。在描述词法时，可以写成多个命名的正则表达式。由于词的构成呈线性结构，在一个命名的正则表达式中，可以引用在其前面定义的命名正则表达式。反过来则不允许。命名的正则表达式相当于 C 语言中的赋值语句。于是，目标语言的词法用正则语言描述时，写出的程序由一个或者多个语句构成。每个语句都是一个命名的正则表达式。

正则语言的文法中，描述语句的产生式仅只有一个，即“$S \rightarrow \text{id} \rightarrow E$ crlf”。该产生式的头部为非终结符 S，表示正则表达式定义语句。产生式体由如下 4 个文法符连接而成：id， $\rightarrow$， E 和 crlf。其中终结符 id 为给正则表达式取的名字，终结符$\rightarrow$的含义是给正则运算表达式取名，相当于 C 语言中的等于号。非终结符 E 表示正则运算表达式，终结符 crlf 是回车换行符，表示语句的结束。

对于正则语言自身的词法，同样可以用正则语言描述出来，描述结果如表 4.6 所示。整个程序由 8 个语句构成，即定义了 8 个命名的正则运算表达式。其中第 3 个至第 7 个命名的正则运算表达式，在名字前面增加了一个@，表示该名字是正则语言中的词类名。也就是说该正则表达式的 NFA，其结束状态的 category 属性值不为空，为该正则表达式的名字。为了简洁起见，此处没有定义注释词，也没有定义哪些类别的词应该过滤掉，不输出。

由表 4.6 所示词法定义的示例可知，命名的正则运算表达式定义语句分带@的和不带@的两种。对于带@的，其文法定义的产生式为：$S \rightarrow @S$ ，其产生式体中的非终结符 S，就是不带@的定义语句。正则语言的词有五类，类名分别为 reserved，id，cc，space，和 crlf，表示预留字，变量，字符常量，空格，和回车换行，如表 4.6 中的第 3 个产生式至第 7 个产生式所示。

由此可知，正则语言的语法非常简单，只有一种语句（即赋值语句），其文法如表 4.7 所示。其中非终结符只有 P，S，E 三个。P 表示用正则语言写出的程序。S 表示命名正则

表达式定义语句，或者说赋值语句。其含义是将一个正则表达式的结果赋值给一个变量。*S* 还有另一含义，即语句序列，通过产生式 *S*→*S S* 表达。*E* 表示正则运算表达式。正则运算共有 10 种，即表 4. 7 中的第 5 个产生式至第 15 个产生式。连接运算的运算符可以省略。于是第 8 和第 9 个产生式都表示连接运算。字符之间有范围运算（运算符为~），例如‘a’~‘z’。字符集合与字符之间有差运算（运算符为-），例如 letter -‘i’。字符集合与字符或者字符集合之间有并运算（运算符为 | ），例如 letter | digit。

表 4. 6　正则语言的词法描述

①	character→‘\0’~’\127’
②	letter→‘a’~’z’	‘A’~’Z’
③	@reserved→ ‘(’ | ‘)’	‘|’	‘·’	‘*’	‘+’	‘?’	‘→’	‘@’	‘$’	‘-’	‘~’
④	@id →letter +
⑤	@cc→“’ · character · ‘”
⑥	@space→空格字符+
⑦	@crlf→(回车字符 · 换行字符)+
⑧	lexeme → reserved	id	cc	space	crlf

表 4. 7　正则语言的文法描述

(1) *P*→*S* $	*(6)* *E*→*E* ~ *E*	① *E*→*E* +
(2) *S*→*S S*	*(7)* *E*→*E* - *E*	② *E*→*E* ?
(3) *S*→ id → *E* crlf	*(8)* *E*→*E* · *E*	③ *E*→(*E*)
(4) *S*→ @*S*	*(9)* *E*→*E E*	④ *E*→id
(5) *E*→*E* | *E*	(10) *E*→*E* *	⑤ *E*→cc

正则运算的优先级是单元运算高于双元运算。双元运算中，连接运算的优先级高于并运算。对于字符集合的运算，范围运算和差运算的优先级高于并运算。括号运算的优先级最高。

对于表 4.6 所示正则语言的词法描述，可按照第 2 章知识手工画出其 DFA，如图 4.4 所示。其中 10 状态为接受状态，其 category 属性值为 id，即变量，也就是正则表达式的名字。12 状态刻画回车换行标志，15 状态刻画任一字符。在编写命名正则表达式时，允许出现空格，1 状态刻画一个或者连续多个空格。空格在正则语言的词法分析中会被过滤掉，不输往语法分析器。

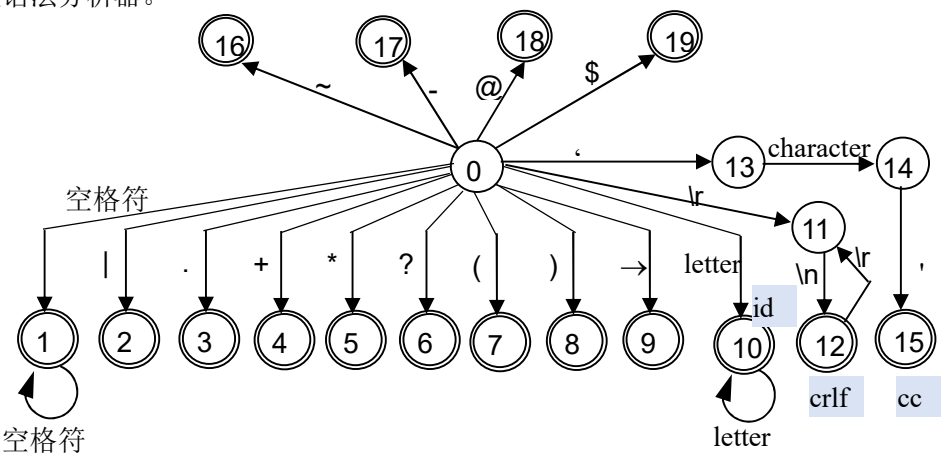


图 4.4 正则语言词法的 DFA

对于表 4.7 所示正则语言的语法描述，按照第 3 章知识可手工得出其 DFA，如图 4.5 所示。其中运算的优先级已在该 DFA 中得以体现。优先级是基于 LR(0)项目的物理含义对非核心项，以及对出边进行取舍而实现。由图 4.5 所示 DFA 得出的 LR(1)语法分析表如表 4.8 所示。

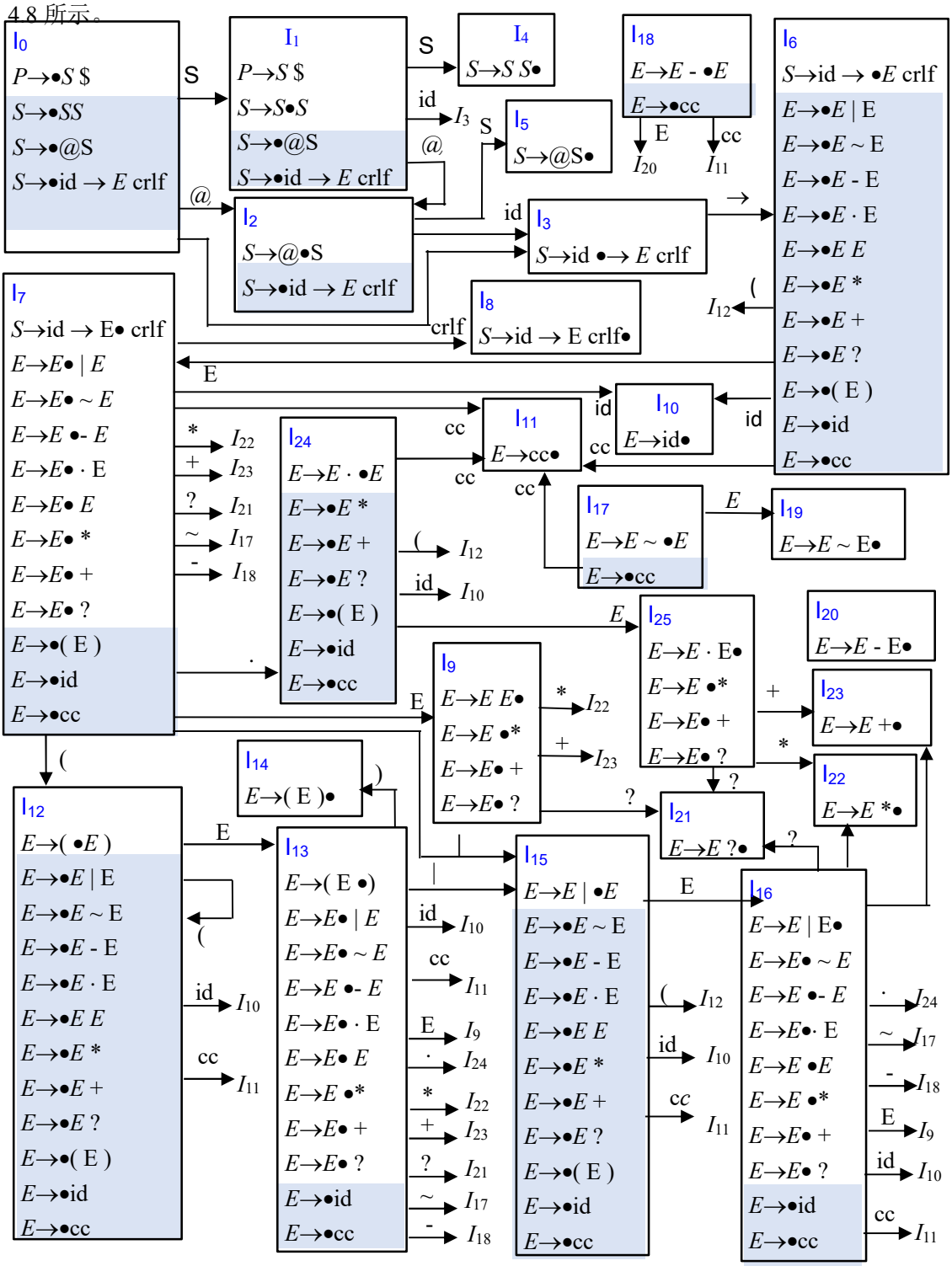


图 4.5 正则语言文法的 DFA

有了正则语言的词法 DFA，以及 LR(1)语法分析表，也就得出了正则语言的词法分析

器以及语法分析器。对于词法分析器构造工具来说，先对输入的程序进行词法分析，然后执行语法分析，在语法分析的规约过程中完成翻译目标。

表 4.8 正则语言的 LR(1)语法分析表

	ACTION															GOTO		
	(	)	|	~	-	·	*	+	?	crlf	@	$	→	id	cc	P	S	E
0											s2			s3			1	
1											s2	acc		s3			4	
2														s3			5	
3													s6					
4											r2	r2		r2				
5											r4	r4		r4				
6	s12													s10	s11			7
7	s12		s15	s17	s18	s24	s22	s23	s21	s8				s10	s11			9
8											r3	r3		r3				
9	r9	r9	r9			r9	s22	s23	s21	r9		r9		r9	r9			
10	r14	r14	r14	r14	r14	r14	r14	r14	r14	r14		r14		r14	r14			
11	r15	r15	r15	r15	r15	r15	r15	r15	r15	r15		r15		r15	r15			
12	s12													s10	s11			13
13		s14	s15	s17	s18	s24	s22	s23	s21					s10	s11			9
14	r13	r13	r13	r13	r13	r13	r13	r13	r13	r13		r13		r13	r13			
15	s12													s10	s11			16
16		r5	r5	s17	s18	s24	s22	s23	s21	r5	r5	r5		s10	s11			9
17															s11			19
18															s11			20
19		r6	r6		r6					r6	r6	r6						
20		r7	r7		r7					r7	r7	r7						
21	r12	r12	r12			r12				r12	r12	r12		r12	r12			
22	r10	r10	r10			r10				r10	r10	r10		r10	r10			
23	r11	r11	r11			r11				r11	r11	r11		r11	r11			
24	s12													s10	s11			25
25	r8	r8	r8			r8	s22	s23	s21	r8		r8		r8	r8			

翻译目标是生成输入的正则表达式的 DFA。因此语法制导的翻译方案设计中，要给表示正则语言文法中的非终结符 P，S，E 设置一个 pNFA 属性，其类型为 Graph＊，记录正则运算表达式的 NFA。类型 Graph 是状态转换图的数据结构，其定义见第 2 章 NFA 和 DFA 构造所涉及的数据结构部分给出。非终结符 E 的每个产生式都表达一种正则运算。于是产生式头部非终结符 E 的 NFA 是按照第 2 章最简 NFA 构造法，由产生式体中的文法符的 NFA 组合得出。当以产生式 $P \rightarrow S$ \$规约出非终结符 P 时，表示语法分析已完成。其中 S 的 pNFA 属性值就为目标语言的词法的 NFA，将其转化成 DFA，就得到了词法分析器构造

工具的输出，即目标语言的词法 DFA。

词法分析器构造工具的 SDD 设计方案，如表 4.9 所示。其中第 3 个产生式相当于 C 语言的赋值语句。其含义是把非终结符 *E* 的 pNFA 属性值，保存到变量 id 中，以便下次使用该变量时，能得到其 NFA 值。使用变量的产生式为第 14 个产生式 *E*→id。该产生式是使用该变量保存的 NFA 值，将其赋给 *E* 的 pNFA 属性。对于第 3 个产生式，规约时要把 id->val 赋给 *S*->val，其目的是为第 4 个产生式规约时，能知道 id->val。在第 4 个产生式规约时，已没有 id 概念了，只有产生式体中的 *S*。

表 4.9 词法分析器构造工具的 SDD

序号	产生式	SDD	规约时调用的翻译函数(即 SDT)
1	$P \rightarrow S\ \$$	output (NFA_to_DFA(*P*.pNFA))	NFA_to_DFA
2	$S \rightarrow S_1\ S_2$	*S*.pNFA = S_2.pNFA	transfer
3	$S \rightarrow \text{id} \rightarrow E\ \text{crlf}$	*S*.pNFA = *E*.pNFA; *S*.val = id.val; map(id.val, *E*.pNFA)	save
4	$S \rightarrow @S_1$	setCategory(S_1.pNFA, S_1.val) *S*.pNFA = S_1.pNFA;	setCategory
5	$E \rightarrow E_1 \mid E_2$	*E*.pNFA =union(E_1.pNFA, E_2.pNFA)	union
6	$E \rightarrow E_1 \cdot E_2$	*E*.pNFA =product(E_1.pNFA, E_2.pNFA)	product
7	$E \rightarrow E_1 E_2$	*E*.pNFA =product(E_1.pNFA, E_2.pNFA)	product
8	$E \rightarrow E_1^+$	*E*.pNFA =plusClosure(E_1.pNFA)	plusClosure
9	$E \rightarrow E_1^*$	*E*.pNFA =closure(E_1.pNFA)	closure
10	$E \rightarrow E_1?$	*E*.pNFA =zeroOrOne(E_1.pNFA)	zeroOrOne
11	$E \rightarrow (E_1)$	*E*.pNFA =E_1.pNFA	setValue
12	$E \rightarrow \text{id}$	*E*.pNFA =unmap (id.val)	get
13	$E \rightarrow \text{cc}$	*E*.pNFA =generateBasicNFA(cc.val)	generateBasicNFA

注意：表 4.9 所示 SDD 没有把有关字符集的三种运算（~，-，|）给出。有兴趣的读者可自己加上。

SDD 中的这些函数已在第 2 章习题中给出，其返回值为新构建的一个 NFA。字符或者字符集合的 NFA 有一个基本特征，那就是其 NFA 只包含两个状态（0 状态和 1 状态），且结束状态（即 1 状态）无出边。因此有关字符集的三种运算（~，-，|），其结果 NFA 还是只有两个状态，其变迁边上的驱动不是字符，而是字符集合。

有了词法分析器构造工具的 SDD 之后，其 SDT 设计非常简单。因采用 LR 语法分析，翻译动作都是在规约时执行，只需给每个产生式的翻译动作确定一个函数名, 如表 4.9 中最后一列所示。对于 SDD 中出现的函数 NFA_to_DFA，union，product，plusClosure，closure，zeroOrOne，generateBasicNFA，它们要做的事情，以及相关的数据结构和数据的存储已在第 2 章中的习题部分给出。另外的 set 函数，是把命名正则运算表达式的名字和 pNFA 存储起来。当后面的正则表达式引用它时，再用 get 函数将其读取出来，于是就得到了名字的 pNFA。

现举例说明，设工具的输入文本文件中包含有两行：“ra→(‘a’|‘b’)* crlf”和

“rb→ra‘a’crlf”。其中第 1 行的含义是将字符 a 和 b 做并运算，再做闭包运算，然后命名为 ra。第 2 行是 ra 和字符 a 做连接运算，然后命名为 rb。crlf 是回车换换符。词法分析后，‘a’和‘b’为字符常量，ra 和 rb 为变量，其他为预留词。语法分析时，先后执行 11 次规约，具体情形如下：

① 按照产生式 E→cc 将‘a’规约成 E_1，将‘b’规约成 E_2；
② 按照产生式 E→$E|E$ 将 E_1 和 E_2 规约成 E_3；
③ 按照产生式 E→(E)将 E_3 规约成 E_4；
④ 按照产生式 E→E*将 E_4 规约成 E_5；
⑤ 按照产生式 S→id → E crlf 规约出 S_1，此时 id->val 为“ra”，S_1->pNFA= E_5->pNFA, S_1->val=id->val，调用函数 set 将 E_5->pNFA 赋给变量 ra；
⑥ 按照产生式 E→id 规约出 E_6，此时 id->val 为“ra”，调用函数 get 读取变量 ra 的值，将其赋给 E_6->pNFA；
⑦ 按照产生式 E→cc 将‘a’规约成 E_7；
⑧ 按照产生式 E→E E 将 E_6 和 E_7 规约成 E_8；
⑨ 按照产生式 S→id → E crlf 规约出 S_2，此时 id->val 为“rb”，S_2->pNFA = E_8->pNFA, S_2->val = id->val，调用 set 函数将 E_8->pNFA 赋给变量 rb；
⑩ 按照产生式 S→S S 将 S_1 和 S_2 规约成 S_3，此时 S_3->pNFA = S_2->pNFA；
11 按照产生式 P→S $规约，调用函数 NFA_to_DFA，将 S_3->pNFA 转化为 DFA。至此语法分析完毕，得到输出结果。

思考题 4-5：词法分析器构造工具和语法分析器构造工具其实是正则语言的两种编译器。第一种编译器的翻译目标(即输出)是词法的 DFA，第二种编译器的翻译目标(即输出)是语法分析表。这种说法对吗？当用正则语言来描述某门高级程序语言的词法时，写出的代码中，每一行都为一个命名的正则表达式。后面的正则运算表达式中，可以通过名字引用前面定义的正则表达式。反过来则不允许，而且不允许自引，为什么？当用正则语言来描述某门高级程序语言的文法时，写出的代码中，每行都为一个产生式。产生式也是命名的正则表达式。对于语法描述，对于命名的正则表达式，允许出现自引和递归引用。这种说法正确吗？

思考题 4-6：语法分析器构造工具的输入也是一个文本文件。该文件描述了某门高级程序语言的文法。该文件中的每行都为一个产生式。产生式是命名的正则表达式。因此该文件是一个用正则语言写出的程序。此说法正确吗？表 4.7 所示正则语言文法, 去掉每行前面的编号之后，是用正则语言写出的程序吗？注意：正则语言中，对于连接运算，可将运算符省略掉。对于表 4.7 的第二个产生式 S→SS，其产生式体中的两个 S 之间一定要有一个空格，为什么？有空格时，经词法分析后得到几个词？每个词的值和类别名分别是什么？如果没有空格，词法分析的结果又是什么？

思考题 4-7：对于语法分析器构造工具，输入的解析不是问题。因为文法描述中，产生式的定义仅只使用了连接运算，非常简单。对输入不使用词法分析器和语法分析器，也能轻易地解析出产生式的定义。对解析出的所有产生式，如何区分其中所含的变量，哪些是非终结符？如果识别出了所有非终结符，那么剩下的变量自然就是终结符。对于终结符，还需对其进一步区分成是自定义词和预定义词，如何区分？另外，还要识别出产生式中，哪一个为根产生式。观察表 4.7 所示文法示例，其中第一个产生式为根产生式，它有什么特征？如何识别？解决了上述问题，便可构建文法 LR(0)项集的转换图（即 DFA）。这么一看，似乎语法分析器构造工具的实现，要比词法分析器构造工具的实现简单，这种观点正

确吗？

2. 实验任务

1. 基于第四章知识，用正则语言描述出 TINY 语言的词法，然后得出 TINY 语言的词法分析器的完整代码：

2. 以用 TINY 语言写出的源程序 sample.tny 作为输入，输出出其词序列。以此验证 TINY 语言词法分析器的正确性。

Sample.tny 源程序包含下面 10 行代码：

```
1.  { Sample program in TINY language - computes factorial}
2.  read x; { input an integer }
3.  if 0 < x then { don't compute if x <= 0 }
4.     fact := 1;
5.     repeat
6.        fact := fact * x;
7.        x := x - 1
8.     until x = 0;
9.     write fact    { output factorial of x }
10. end
```

试验四、TINY 语言编译器的实现

1. 实验描述

基于第 5 章的课程知识，以及前面 3 个实验的结果，写出一个完整的 TINY 语言编译器，该编译器的输入为一文本文件，即 TINY 源程序，其输出也是一个文本文件，即以 tm 中间语言写出的源程序，其后缀名为 tm。TM.exe 为一个虚拟机，通过它来运行 tm 程序，得出计算结果。运行 TM.exe 的时候，输入是一个文本文件，即 tm 程序，输出为输入程序的运行结果。TINY 源程序的样例文件为 sample.tny，对其编译后得到 sample.tm 文件。使用 TM.exe 运行 sample.tm，便能得到 sample.tm 的运行结果。

说明：TINY 语言编译器有源代码，见附录。在以往的教学中，发现学生根本就读不懂这个源代码，也就无法把握其构成框架。因此须要从最基本的内容入手，一步一步地推进，才可能登上新的台阶。

tm 中间语言的词法和语法通过阅读 sample.tm 文件来感知，也可参考 TINY 语言编译器的源代码来得知。

鼓励和欢迎学生从 TINY 语言编译器的源代码摘取程序段，加入到自己的程序中来，为我所用。自己写的编译器一定要符合课程所述的框架。如果是照抄整个程序，那么性质就完全变了，整个实验记为 0 分。

2. 实验任务

1. 写出 TINY 语言的语义分析和中间代码生成的 SDT。

2. 基于 TINY 语言的语义分析和中间代码生成的 SDT，得出 TINY 语言编译器的完整源代码。

3. 以 sample.tny 作为输入，得出输出 sample.tm，验证所写 TINY 语言编译器的正确性。

4. 用 TM.exe 程序运行 sample.tm 程序，得出 sample.tm 程序的运算结果。

5. 阅读 TM.exe 程序的源程序 TM.c，阅读时要知道 TM.c 是一个虚拟机，也可说是一个解释执行器，也可以说是一个编译器，为什么？。通过这个例子，就知道虚拟机，解释执行器，编译器，这三者其实同义。

提示：Tm.c 程序其实是一个 tm 中间语言的编译器。它要进行 tm 程序的词法分析，语法分析，要作翻译。只是翻译目标不是得出程序，而是要得出程序的运行结果。