**编译技术**

**课程实验指导书**

杨金民，陈果，黎文伟

2024年

目录

[试验一、正则运算表达式的DFA构建 1](#_Toc11254)

[1. 正则运算表达式的最简NFA构建 1](#_Toc3463)

[2. 特殊正则表达式的最简NFA构造 6](#_Toc29751)

[3. NFA和DFA中状态属性值的确定方法 7](#_Toc16431)

[4. 正则表达式之间的包含关系 10](#_Toc13727)

[5. NFA和DFA构造中所涉及的数据结构 13](#_Toc31987)

[6. 实验任务 15](#_Toc21755)

[试验二、上下文无关文法的DFA构建 16](#_Toc5132)

[1. 语法分析表构造中所涉及的数据结构 16](#_Toc19918)

[2. 实验任务 18](#_Toc5715)

[试验三、词法分析器构造工具的实现 20](#_Toc2899)

[1. 词法分析器构造工具的实现方法 20](#_Toc28889)

[2. 实验任务 26](#_Toc8744)

[试验四、TINY语言编译器的实现 27](#_Toc6728)

[1. 实验描述 27](#_Toc14510)

[2. 实验任务 27](#_Toc2138)

## 试验一、正则运算表达式的DFA构建

### 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。**

1

3

2

b

b

0

a

b

a

a

a

b

0

a

b

3

a

1

b

2

b

（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。

答：

NFA:因为他的0状态在一个驱动字符下有多个下一状态。

五个状态集合；状态转换表如下：



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。DFA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。

答：在构造NFA时，所谓的“空转换”是指从一个状态可以通过不消耗输入符号（即ε）直接到达另一个状态。在将NFA转换为DFA的过程中，确实有一个步骤是处理这些ε转换的，因为DFA不允许有ε转移。在子集构造法中，我们通常需要考虑所有通过任意数量（包括零个）ε转换所能达到的状态集合，并将这些状态集合视为DFA中的一个新状态。这个过程可以消除原NFA中的单个ε转移，确保转换后的DFA不再含有ε转移。

当构建最简形式的NFA或DFA时，会进一步考虑到状态的等价性，即两个或多个状态集合在接收相同输入后能够到达相同的状态集合，则这些状态集合是可以合并的，从而简化自动机结构。在这个上下文中，“省掉的条件”不是指省略处理ε转换的过程，而是指在化简过程中找出并消除冗余状态和转移，使得最终得到的自动机是最小化的，即每个状态都是不可约简的，且对于给定语言而言，是最简洁的形式。如果NFA已经是最简形式并且不包含ε转移，那么在构建等价DFA时就无需处理ε转移问题。

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⋅*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。

a

0

1

a

0

1

ε

b

0

1

ε

b

0

1

a

0

ε

b

1

2

ε

a

0

ε

ε

b

2

3

ε

1

1. ***r*1 = a的NFA (c) *r*3 = *r*1+的NFA (e) 由原生构造法得到的*r*5的NFA**

1. ***r*2 = b的NFA (d) *r*4 = *r*2+的NFA (f) 由最简构造法得到的*r*5的NFA**

图2.14 正则运算的NFA构建示例

对于连接运算 *s* ⋅ *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。

*s*

0

*t*

*s*+*t*

*s*

*s*

0

*s*+*t+*1

*t*

*s+*1

ε

*s*

*s*

0

*s*

*t*

0

*t*

有入边

有出边

*s*

0

*s*

*t*

0

*t*

1. 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* ⋅ *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。

*s*

1

*s*+1

ε

*s*+2

ε

0

ε

ε

*s*

0

*s*

ε

ε

*s*

0

*s*

ε

*s*+1

ε

ε

*s*

1

*s+*1

ε

0

ε

ε

*s*

0

*s*

有入边

有出边

*s*

0

*s*

有入边

无出边

*s*

0

*s*

无入边

有出边

*s*

0

*s*

无入边

无出边

图2.16 闭包运算的最简NFA构造中的四种情形

*s*

1

*s*+1

ε

*s*+2

ε

0

ε

*s*

0

*s*

ε

*s*+1

ε

*s*

0

*s*

ε

*s*

1

*s+*1

ε

0

ε

*s*

0

*s*

有入边

有出边

*s*

0

*s*

有入边

无出边

*s*

0

*s*

无入边

有出边

*s*

0

*s*

无入边

无出边

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* | *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。

*s*

0

*s*

*t*

0

*t*

无入边

无出边

无入边

无出边

*s*

0

*s+t*-1

*t*

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。

*s*

0

*s*

有入边

*s*

1

*s*+1

ε

0

无出边

*s*

0

*s*

无入边

*s*

0

*s*

有入边

ε

0

有出边

有出边

*s*

*s*

*s*+1

ε

0

*s*

*s*+1

*s*+2

ε

1

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。

*s*

0

*s*

category属性值不为空

*s*

*s*

*s*+1

ε

0

图2.20 并运算之前对其分量NFA的第2步等价改造

*s*

0

*s*

有入边

有出边或者category属性值不为空

*s*

1

*s*+1

*t*

*s*+2

*s*+*t*+2

*s+t+3*

0

ε

ε

ε

ε

*t*

0

*t*

有入边

有出边或者category属性值不为空

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。

*s*

0

1

ε

ε

*s*

0

1

无入边

无出边

*s*

0

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→a；*r*2→b；*r*3→*r*1 | *r*2；*r*4→*r*3\*；*r*5→*r*4 ⋅ *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

0

1

b

0

1

a

0

1

b

a

0

b

a

0

b

a

1

1. *r*1→a的NFA
2. *r*2→b的NFA
3. *r*3→*r*1 | *r*2的NFA
4. *r*4→*r*3\*的NFA
5. *r*5→*r*4 ⋅ *r*1的NFA

图2.23 正则表达式(a|b)\*a的最简NFA构建过程

### 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* → (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*。

*c-*‘*\**’*-*‘*/*’

ε

ε

3

ε

2

1

*c -* ‘*\**’

0

ε

‘*\**’

ε

7

ε

6

5

*c-*‘*\**’

4

ε

‘*\**’

ε

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） |  | | |
| *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。

*c-*‘*\**’

‘*\**’

1

2

‘*\**’

0

*c-*‘*\**’*-*‘*/*’

3

‘*\**’

*c -* ‘*\**’*-*‘*/*’

*c-*‘*\**’

*’\*’*

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* → *s* ⋅ (*t* ⋅ *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。

*c-*‘*\**’*-*‘*/*’

2

1

*c-*‘*\**’

0

ε

‘*\**’

ε

*c-*‘*\**’

‘*\**’

1

‘*\**’

0

*’c-*‘*\**’*-*‘*/*’

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

*s*的NFA

*s*

0

*s*

*t*

0

*t*

*t*的NFA

*s*

0

*s*+*t*-1

*t*

图2.28 正则表达式*s* ⋅ (*t* ⋅ *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？

答：在构造正则表达式`s(ts)\*`对应的NFA时，通常我们会按照从右到左的自底向上的递归方式来构建子表达式的NFA，并将它们组合成一个整体。

对于`s`和`t`各自对应的状态集合，为了保持状态序号的连续性和唯一性，在合并NFA时确实可能需要对已有状态进行重新编号。原因在于，当我们组合两个NFA时，尤其是当其中一个NFA（这里是`s`）的状态要作为另一个NFA（这里是`t`之后的重复部分）的起点时，为了避免状态编号冲突以及清晰地表示新NFA中状态之间的转换关系，应当确保`s`的所有状态编号都在`t`之后，这样才能反映`s`紧接在`t`之后至少一次或多次出现这一结构。

如果从`s`状态开始倒着修改序号至1状态，这是因为这样做可以确保在调整过程中不会影响到已经正确分配好新序号的后续状态。若从1状态开始顺次修改，可能会导致在修改过程中尚未处理的状态与已处理状态之间产生重叠或不连续的编号，进而破坏状态之间的转换逻辑。

就正如闭包运算一样， *s* ⋅ (*t* ⋅ *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。

对于正则表达式`s(ts)\*`，其最简NFA会包含以下特征：

- 一个起始状态（假设为状态1），它通过ε转换到达`t`的起始状态。

- `t`对应的NFA部分。

- `s`对应的NFA部分，其所有状态的编号都按要求进行了偏移，使其位于`t`对应状态之后。

- `s`的结束状态（原状态`s`）通过ε转换回到`t`的起始状态，同时也通过ε转换到达接受状态（假设是新的状态集中的某个状态）。

- 因为有 Kleene 星号(\*)，所以存在从`t`的任何接受状态（包括自身可能是接受状态的情况）到`t`起始状态的循环，以允许`t`序列的零次或任意多次重复。

### 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。

1

0

ε

‘.’

0

2

1

digit

ε

1

0

‘E’

‘+’|‘-’

ε

3

2

digit

ε

digit

ε

1

0

digit

‘.’

ε

3

ε

2

digit

ε

8

ε

5

4

‘E’

7

‘+’|‘-’

ε

ε

6

digit

ε

1. **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。

‘E’

2

digit

digit

1

‘.’

digit

4

‘E’

3

digit

6

‘+’|‘-’

5

digit

digit

digit

0

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。

### 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。

i

0

2

1

f

f

1

3

2

ε

i

0

1

0

letter

ε

2

ε

1

letter

0

ε

3

ε

1

0

i

n

3

2

t

1

0

i

n

2

4

ε

t

3

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。

f

3

4

8

ε

i

0

2

ε

1

letter

ε

5

i

n

6

ε

t

7

ε

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。

1

letter *-*‘i’

‘i’

0

‘n’

letter *-* ‘f’ *-* ‘n’

3

‘f’

letter

letter *-* ‘t’

2

*l*etter

4

‘t’

5

letter

KEYWORD

KEYWORD

ID

ID

ID

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。

### 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‘\_’。

举例说明如下：

对于字符集：letter - ‘i’-‘w’，它包括3段：1）‘a’~‘h’; 2)‘j’~‘x’; 3)‘y’~‘z’

在charSet表中就有3行数据：

|  |  |  |  |
| --- | --- | --- | --- |
| IndexId | segmentId | fromChar | toChar |
| 1 | 1 | ‘a’ | ‘h’ |
| 1 | 2 | ‘j’ | ‘x’ |
| 1 | 3 | ‘y’ | ‘z’ |

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。

### 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。

1. 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。

1. 针对上述NFA的数据结构定义，实现如下函数：
2. 子集构造法中的3个函数：move，ε\_closure，DTran；
3. 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。

1. 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

### 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enum，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

### 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：
2. 产生式的FIRST函数求解；
3. 非终结符的FIRST函数求解；
4. 非终结符的FOLLOW函数求解；
5. 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：
6. 一个项集中LR(0)核心项的闭包求解，即实现函数：

void getClosure( ItemSet \*itemSet)；

1. 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)；

1. 文法的LR(0)型DFA求解；
2. 文法是否为SLR(1)文法的判断；
3. LR语法分析表的填写；
4. 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

### 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* → id→ *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， →， *E* 和 crlf。其中终结符id为给正则表达式取的名字，终结符→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* → *@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. *E*→*E* +
2. *E*→*E* ?
3. *E*→( *E* )
4. *E*→id
5. *E*→cc
6. *P*→*S* $
7. *S*→*S* *S*
8. *S*→ id→ *E* crlf
9. *S*→ *@S*
10. *E*→*E* | *E*
11. *E*→*E* ~ *E*
12. *E*→*E* - *E*
13. *E*→*E* ⋅ *E*
14. *E*→*E* *E*
15. *E*→*E* \*

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。

0

9

→

4

+

3

.

‘

15

'

7

(

8

)

6

?

5

\*

2

|

letter

letter

10

14

character

13

*\*r

11

12

\n

1

空格符

空格符

18

19

@

17

$

16

~

-

crlf

*\*r

cc

id

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。

1. I0

*P*→•*S* $

*S*→•*SS*

*S*→•@S

*S*→•id → *E* crlf

S

I4

*S*→*S* *S*•

S

@

I1

*P*→*S* $

*S*→*S*•*S*

*S*→•@S

*S*→•id → *E* crlf

1. I2

*S*→@•S

*S*→•id → *E* crlf

S

1. I5

*S*→@S•

*S*→id• ( E *crlf*

1. I18

*E*→*E* - •*E*

*E*→•cc

id

1. I7

*S*→id → E• crlf

*E*→*E*• | *E*

*E*→*E*• ~ *E*

*E*→*E* •- *E*

*E*→*E*• ⋅ E

*E*→*E*• *E*

*E*→*E*• \*

*E*→*E*• +

*E*→*E*• ?

*E*→•( E)

*E*→•id

*E*→•cc

cc

(

E

crlf

~

*I*17

\*

*I*22

+

*I*23

?

*I*21

-

*I*18

⋅

I10

*E*→id•

1. I24

*E*→*E* ⋅ •*E*

*E*→•*E* \*

*E*→•*E* +

*E*→•*E* ?

*E*→•( E)

*E*→•id

*E*→•cc

1. I12

*E*→( •*E* )

*E*→•*E* | E

*E*→•*E* ~ E

*E*→•*E* - E

*E*→•*E* ⋅ E

*E*→•*E* *E*

*E*→•*E* \*

*E*→•*E* +

*E*→•*E* ?

*E*→•( E)

*E*→•id

*E*→•cc

1. I13

*E*→( E •)

*E*→*E*• | *E*

*E*→*E*• ~ *E*

*E*→*E* •- *E*

*E*→*E*• ⋅ E

*E*→*E*• *E*

*E*→*E* •\*

*E*→*E*• +

*E*→*E*• ?

*E*→•id

*E*→•cc

)

1. I14

*E*→( E)•

id

*I*10

*I*11

cc

1. I17

*E*→*E* ~ •*E*

*E*→•cc

E

1. I3

*S*→id •→ *E* crlf

1. I6

*S*→id → •*E* crlf

*E*→•*E* | E

*E*→•*E* ~ E

*E*→•*E* - E

*E*→•*E* ⋅ E

*E*→•*E* *E*

*E*→•*E* \*

*E*→•*E* +

*E*→•*E* ?

*E*→•( E)

*E*→•id

*E*→•cc

1. I8

*S*→id → Ecrlf•

(

*I*12

→

id

(

E

1. I19

*E*→*E* ~ E•

id

*I*10

id

E

*I*20

(

*I*12

I11

*E*→cc•

cc

cc

@

cc

*I*11

id

*I*3

cc

id

*I*10

~

*I*17

*I*11

cc

+

*I*23

?

*I*21

-

*I*18

E

*I*9

⋅

*I*24

\*

*I*22

\*

*I*22

+

*I*23

1. I9

*E*→*E* *E*•

*E*→*E* •\*

*E*→*E*• +

*E*→*E*• ?

*E*

1. I15

*E*→*E* | •*E*

*E*→•*E* ~ E

*E*→•*E* - E

*E*→•*E* ⋅ E

*E*→•*E* *E*

*E*→•*E* \*

*E*→•*E* +

*E*→•*E* ?

*E*→•( E)

*E*→•id

*E*→•cc

E

|

?

(

*I*12

**|**

c*c*

*I*11

id

*I*10

+

1. I21

*E*→*E* ?•

\*

?

1. I22

*E*→*E* \*•

?

1. I23

*E*→*E* +•

1. I20

*E*→*E* - E•

1. I25

*E*→*E* ⋅ E•

*E*→*E* •\*

*E*→*E*• +

*E*→*E*• ?

1. I16

*E*→*E* | E•

*E*→*E*• ~ *E*

*E*→*E* •- *E*

*E*→*E*•⋅ E

*E*→*E* •*E*

*E*→*E* •\*

*E*→*E*• +

*E*→*E*• ?

*E*→•id

*E*→•cc

~

*I*17

*I*24

-

*I*18

E

*I*9

⋅

cc

*I*11

id

*I*10

*E*

0

9

→

4

+

3

.

‘

15

'

7

(

8

)

6

?

5

\*

2

|

letter

letter

10

14

character

13

*\*r

11

12

\n

1

空格符

空格符

18

19

@

17

$

16

~

-

crlf

*\*r

cc

*id*

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*→*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*→*S* $ | output (NFA\_to\_DFA(*P*.pNFA) ) | NFA\_to\_DFA |
| 2 | *S*→*S*1 S2 | *S*.pNFA *= S*2.pNFA | transfer |
| 3 | *S*→id→ *E* crlf | *S*.pNFA *= E*.pNFA;  *S*.val *=* id.val;  map( id.val*, E*.pNFA) | save |
| 4 | *S*→@*S*1 | setCategory( *S*1.pNFA*, S*1.val)  *S*.pNFA *= S*1.pNFA; | setCategory |
| 5 | *E*→*E*1|*E*2 | *E*.pNFA *=*union(*E*1.pNFA*, E*2.pNFA) | union |
| 6 | *E*→*E*1 ⋅ *E*2 | *E.*pNFA *=*product(*E*1*.*pNFA*, E*2*.*pNFA) | product |
| 7 | *E*→*E*1 *E*2 | *E.*pNFA *=*product(*E*1*.*pNFA*, E*2*.*pNFA) | product |
| 8 | *E*→*E*1+ | *E.*pNFA *=*plusClosure(*E*1*.*pNFA) | plusClosure |
| 9 | *E*→*E*1\* | *E.*pNFA *=*closure(*E*1*.*pNFA) | closure |
| 10 | *E*→*E*1? | *E.*pNFA *=*zeroOrOne(*E*1*.*pNFA) | zeroOrOne |
| 11 | *E*→(*E*1) | *E.*pNFA *=E*1*.*pNFA | setValue |
| 12 | *E*→id | *E.*pNFA *=*unmap (id.val) | get |
| 13 | *E*→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S3，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？

### 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

### 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。

### 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。