词法分析程序的报告

--111141-13 吴子鸿

一、编写环境

OS X 10.11.6　　Xcode7.3.1　　Swift2.2

二、大致过程

计算正则式：

1. 读入正则表达式
2. 对正则表达式处理、建图、生成 ε-NFA
3. 将 ε-NFA 去除空节点、转化为 NFA
4. 将 NFA 转化为 DFA
5. 对 DFA 图进行遍历每个节点，获取到每个节点通过某个字符到达哪下一个状态，并找到终态、构造 DFA 表、输出显示

验证字符串：

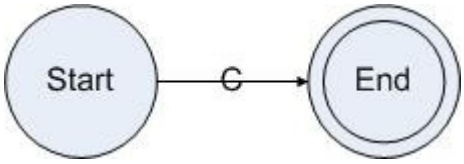
1. 在成功通过正则表达式构建 DFA 图的基础上，读入任意字符串
   从字符串第一个字符、DFA 图的第一个节点开始
2. 判断是否有当前的字符串的字符可以使当前的 DFA 图节点走到下一个节点
3. 若有，则走向下一个节点，重复 2 操作，若无，则返回 false
4. 当字符串从头到尾监测完成后，判断当前所在节点是否是终态节点，若是，则返回 true，反之则返回 false

三、详细过程

一)正则表达式-> ε-NFA

1. 字符集

   字符集是正则表达式最基本的元素，因此反映到状态图上，字符集也会是构成状态图的基本元素。对于字符集 C，如果有一个规则只接受 C 的话，这个规则对应的状态图将会被构造成以下形式：

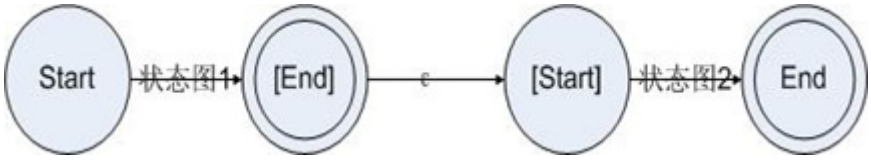


   解释：这个状态图的初始状态是 Start，结束状态是 End。Start 状态读入字符集 C 跳转到 End 状态，不接受其他字符集。

2. 串联 （形如正则表达式 abc）

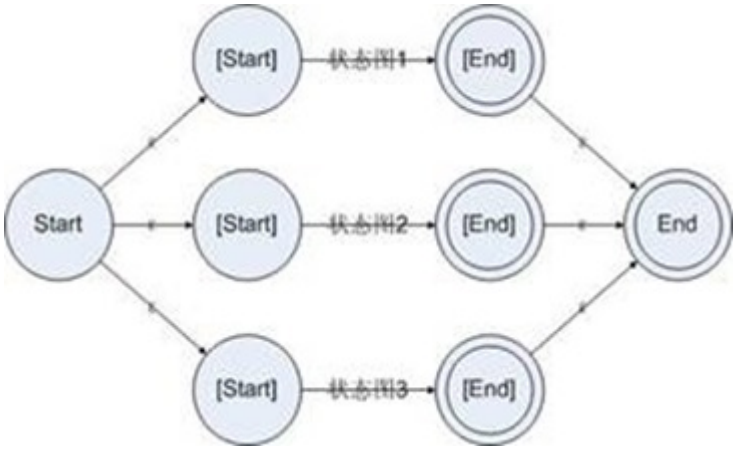
   如果我们使用 AB 表示规则 A 和规则 B 的串联，我们可以很容易的知道串联这个操作具有结合性，也就是说(AB)C=A(BC)。因此对于 n 个规则的串联，我们只需要先将前 n-1 个规则进行串连，然后把得到的规则看成一个整体，跟最后一个规则进行串联，那么就得到了所有规则的串联。如果我们知道如何将两个规则串联起来的话，也就等于知道了如何把 n 个规则进行串联。

为了将两个串联的规则转换成一个状态图，我们只需要先将这两个规则转换成状态图，然后让第一个状态的结束状态跳转到第二个状态图的起始状态。这种跳转必须是不读入字符的跳转，也就是令这两个状态等价。因此，第一个状态图跳转到了结束状态的时候，就可以当成第二个状态图的起始状态，继续第二个规则的检查。因此我们使用了 ε 边连接两个状态图：



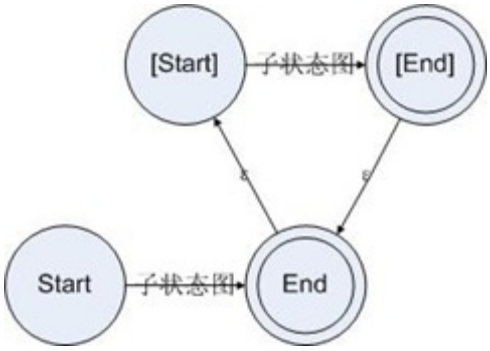
3. 并联（形如正则表达式 0(010|101)1）

并联的方法跟串联类似。为了可以在起始状态读入一个字符的时候就知道这个字符可能走的是并联的哪一些分支并进行跳转，我们需要先把所有分支的状态图构造出来，然后把起始状态连接到所有分支的起始状态上。而且，在某个分支成功接受了一段字符串之后，为了让那个状态图的结束状态反映在整个状态图的结束状态上，我们也把所有分支的结束状态都连接到大规则的结束状态上。如下所示：



4. 重复（形如正则表达式 01(010)*101）

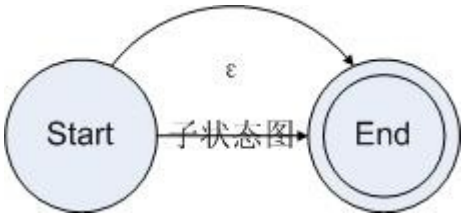
对于一个重复，我们可以设立两个状态。第一个状态是起始状态，第二个状态是结束状态。当状态走到结束状态的时候，如果遇到一个可以让规则接受的字符串，则再次回到结束状态。这样的话就可以用一个状态图来表示重复了。于是对于重复，我们可以构造状态图如下所示：



5. 可选(形如 1(0)*11)

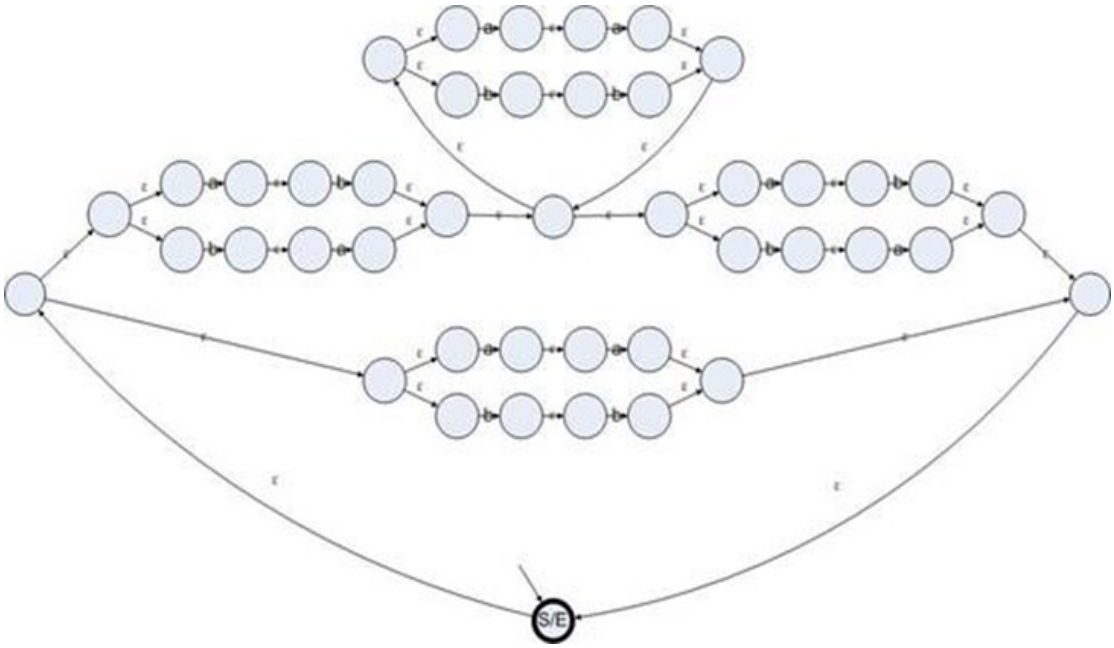
为可选操作建立状态图比较简单。为了完成可选操作，我们需要在接受一个字符的时候，如果字符串的前缀被当前规则接受则走当前规则的状态图，如果可

选规则的后续规则接受了字符串则走后续规则的状态图，如果都接受的话就两个图都要走。为了达到这个目的，我们把规则的状态图的起始状态和结束状态连接起来，得到了如下状态图：



通过上述的转换，可以将输入的正则表达式转化为对应的 ε-NFA，如下所示：

((aa|bb)|((ab|ba)(aa|bb)*(ab|ba)))*

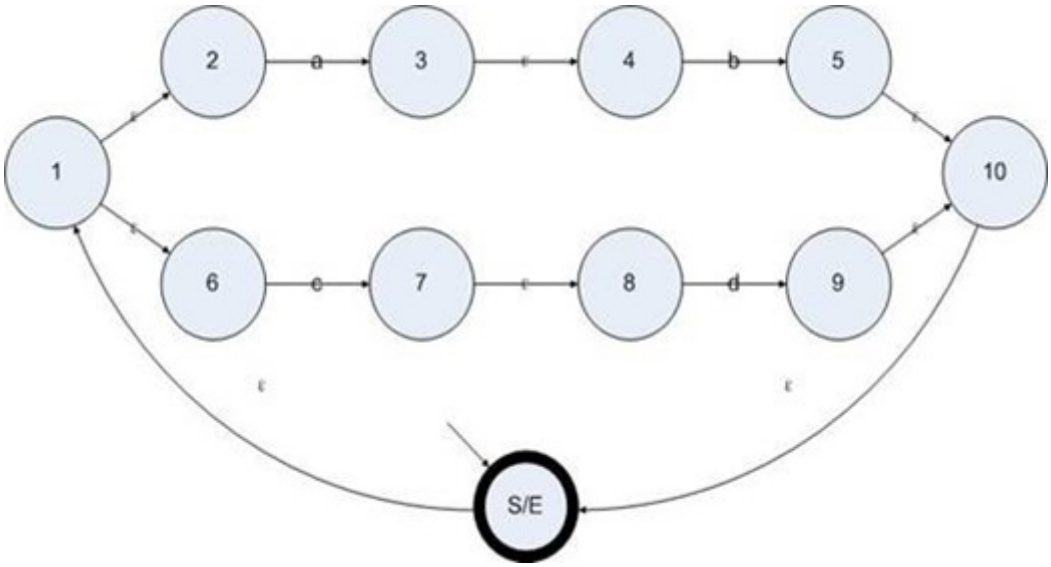


二)去除空节点(将 ε-NFA 转换为 DFA)

一个 DFA 中不可能出现 ε 边，所以我们首先要消除 ε 边。消除 ε 边算法基于一个很简单的想法：如果状态 A 通过 ε 边到达状态 B 的话，那么状态 A 无需读入字符就可以直达状态 B。如果状态 B 需要读入字符 x 才可以到达状态 C 的话，那么状态 A 读入 x 也可以到达状态 C。因为从 A 到 C 的路径是 A B C，其中 A 到 B 不需要读入字符。

方法：消除从状态 A 出发的 ε 边，只需要寻找所有从 A 开始仅通过 ε 边就可以到达的状态，并把从这些状态触出发的非 ε 边复制到 A 上即可。剩下的工作就是删除所有的 ε 边和一些因为消除 ε 边而变得不可到达的状态。为了更加形象地描述消除 ε 边算法，我们从正则表达式(ab|cd)*构造一个 ε-NFA，并在此状态机上应用消除 ε 边算法。

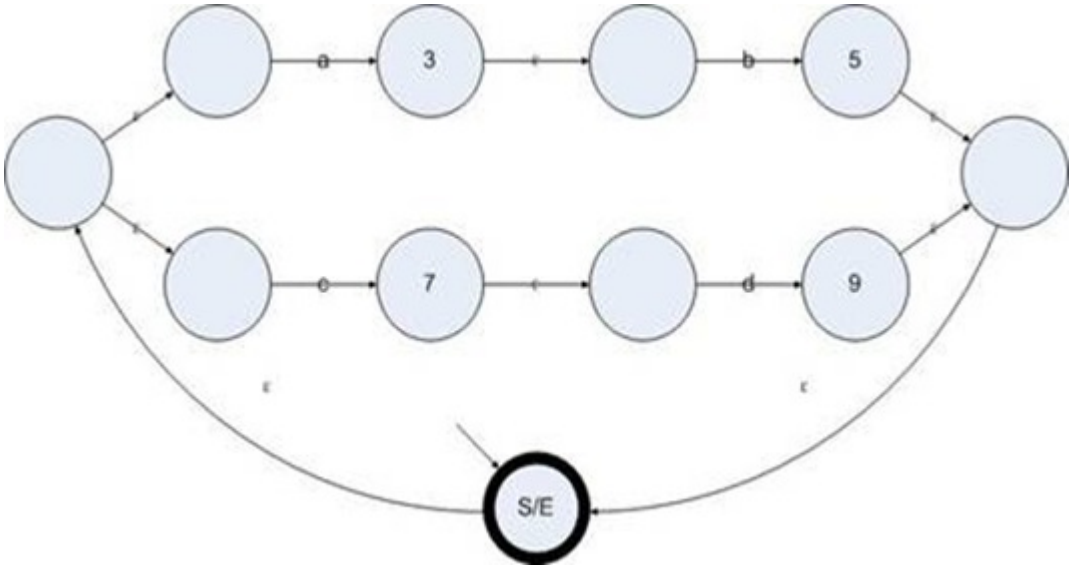
正则表达式(ab|cd)*的状态图如下所



1. 找到所有有效状态。

有效状态就是在完成了消除 ε 边算法之后仍然存在的状态。我们可以在开始整个算法之前就预先计算出所有有效状态。有效状态的特点是存在非 ε 边的输入。同时，起始状态也是一个有效状态。结束状态不一定是有效状态，但是如果存在一个有效状态可以仅通过 ε 边到达结束状态的话，那么这个状态应该被标记为结束状态。因此对一个 ε-NFA 应用消除 ε 边算法产生的 NFA 可能出现多个结束状态。不过起始状态仍然只有一个。

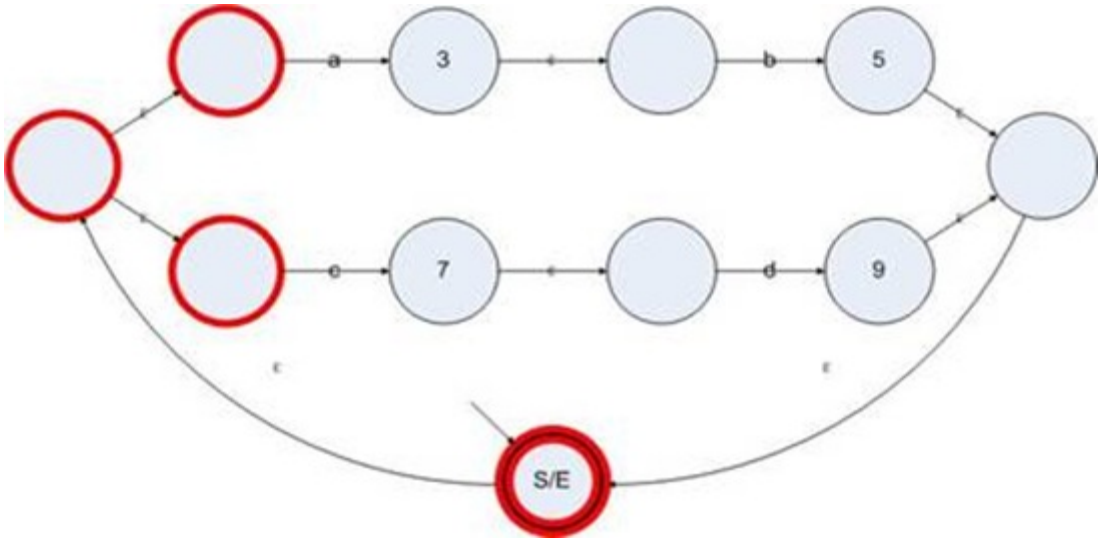
我们可以把“存在非 ε 边的输入或者起始状态”这个判断方法应用在上图的每一个状态上，计算出其中所有的有效状态。结果如下图所示。



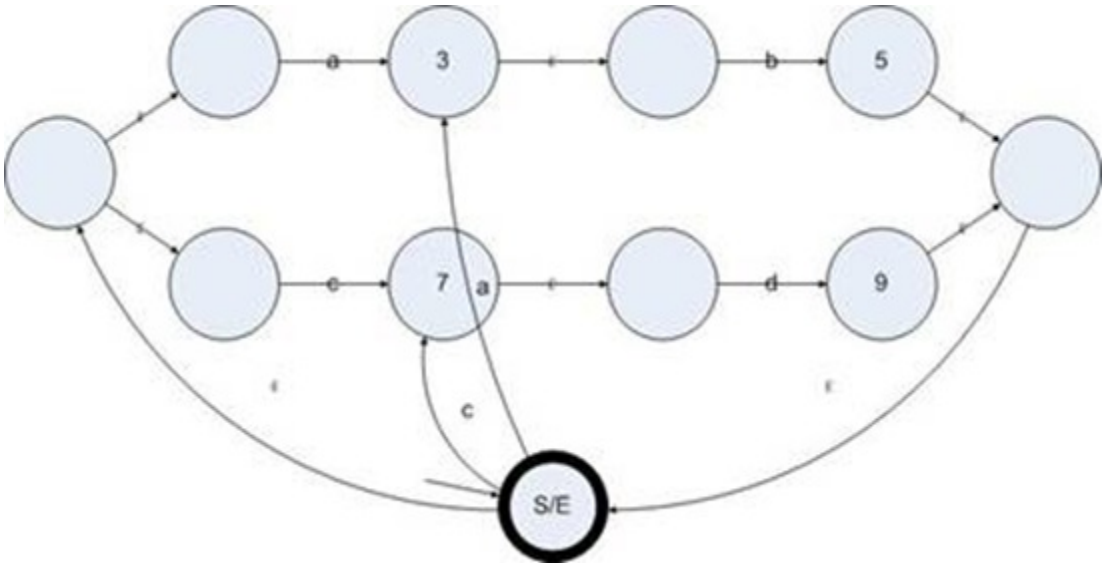
2. 添加所有必要的边

接下来我们要对所有的有效状态都应用一个算法。这个算法分成两步。第一步是寻找一个状态的 ε 闭包，第二步是把这个状态的 ε 闭包看成一个整体，把所有从这个闭包中输出的边全部复制到当前状态上。从标记有效状态的结果我们得到了图 5.1 状态图的有效状态集合是{S/E 3 5 7 9}。我们依次对这些状态应用上述算法。第一步，计算 S/E 状态的 ε 闭包。所谓一个状态的 ε 闭包就是从这个状态出发，仅通过 ε 边就可以到达的所有状态的

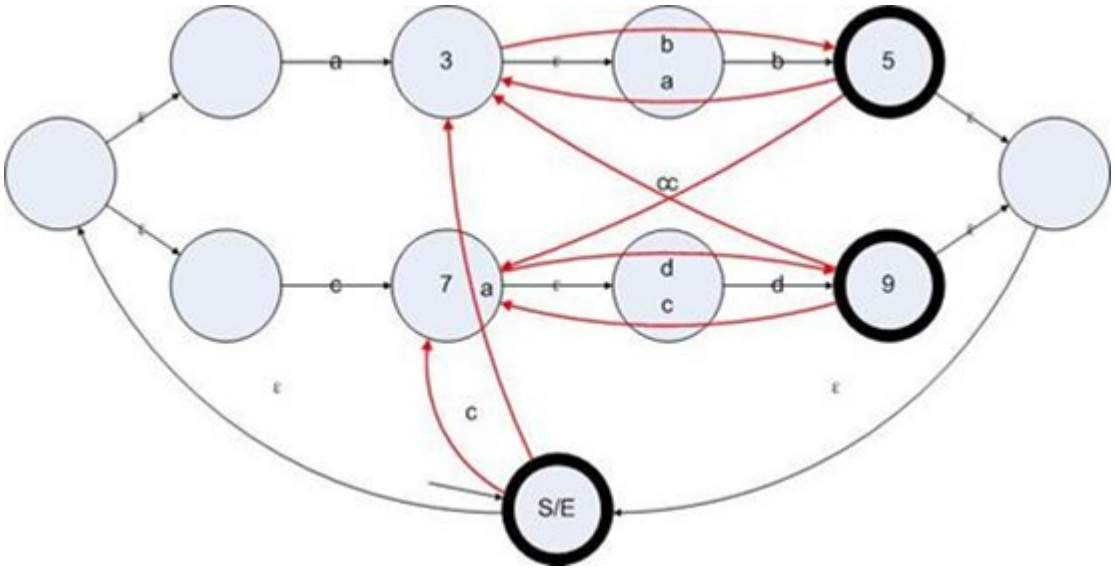
集合。下图中标记出了状态 S/E 的 ε 闭包：



现在，我们把状态 S/E 从状态 S/E 的 ε 闭包中排除出去。因为从状态 A 输出的非 ε 边都属于从状态 A 的 ε 闭包中输出的非 ε 边，复制这些边是没有任何价值的。接下来就是找到从状态 S/E 的 ε 闭包中输出的非 ε 边。在图 5.3 我们可以很容易地发现，从状态 1 和状态 6(见图 5.1 的状态标签)分别输出到状态 3 和状态 7 的标记了 a 或者 b 的边，就是我们所要寻找的边。接下来我们把这些边复制到状态 S/E 上，边的目标状态仍然保持不变，可以得到下图



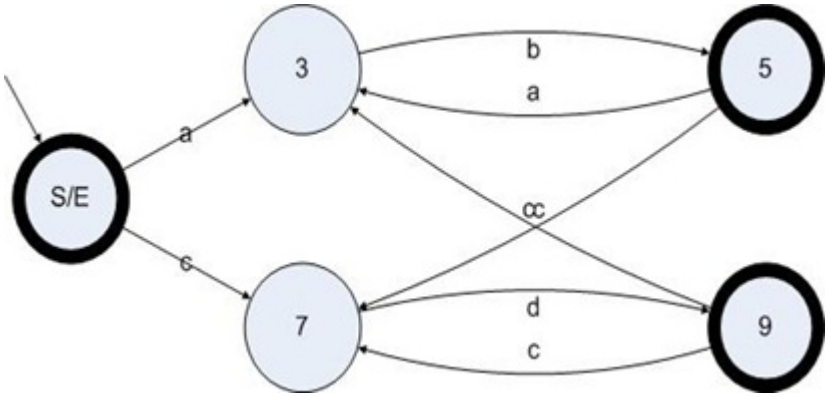
至此，这个算法在 S/E 上的应用就结束了，接下来我们分别对剩下的有效状态{3 5 7 9}分别应用此算法，可以得到下图：



红色的边为新增加的边

3. 删除所有 ε 边和无效状态

这一步操作是消除 ε 边算法的最后步骤。我们只需要删除所有的 ε 边和无效状态就完成了整个消除 ε 边算法。现在我们对图 5.5 的状态机应用第三步，得到如下状态图：



至此，已经将 ε-NFA 转换为 DFA

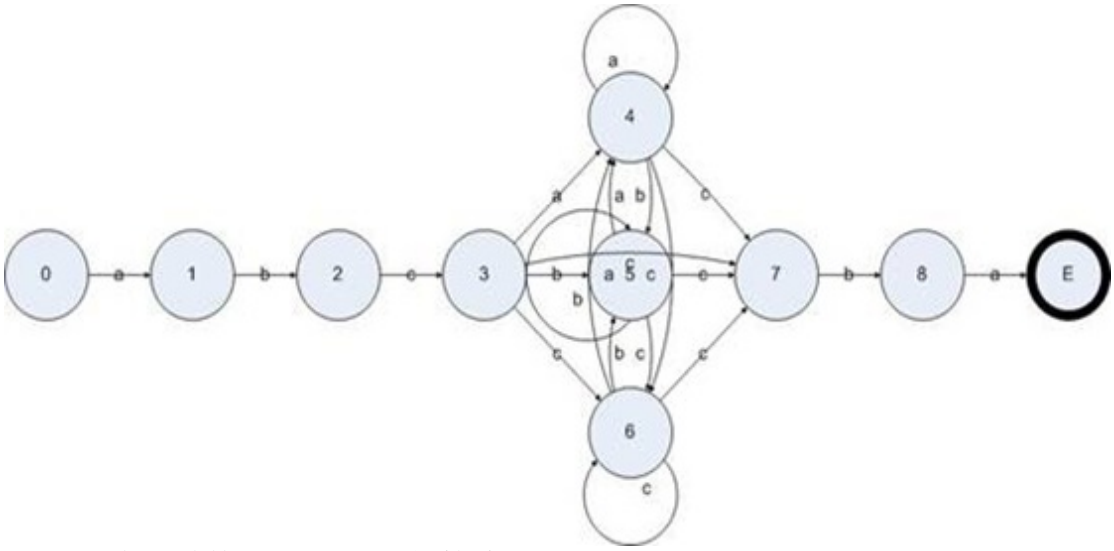
三)从 NFA 到 DFA

NFA 是非确定性的状态机，DFA 是确定性的状态机。确定性和非确定性的最大区别就是：从一个状态读入一个字符，确定性的状态机得到一个状态，而非确定性的状态机得到一个状态的集合。如果我们把 NFA 的起始状态 S 看成一个集合{S}的话，对于一个状态集合 S'，给定一个输入，就可以用 NFA 计算出对应的状态集合 T'。因此我们在构造 DFA 的时候，只需要把起始状态对应到 S'，并且找到所有可能在 NFA 同时出现的状态集合，把这些集合都转换成 DFA 的一个状态，那么任务就完成了。因为 NFA 的状态是有限的，所以 NFA 所有状态的集合的幂集的元素个数也是有限的，因此使用这个方法构造 DFA 是完全可能的。

为了形象地表达出这个算法的过程，我们将构造一个正则表达式，然后给出该正则表达式转换成 NFA 的结果，并把构造 DFA 的算法应用在 NFA 上。假设一个字符串只有 a、b 和 c 三种字符，判断一个字符串是不是以 abc 开始并且以 cba 结束正则表达式如下：

abc(a|b|c)*cba

通过上文的算法，可以得出如下图所示的 NFA：

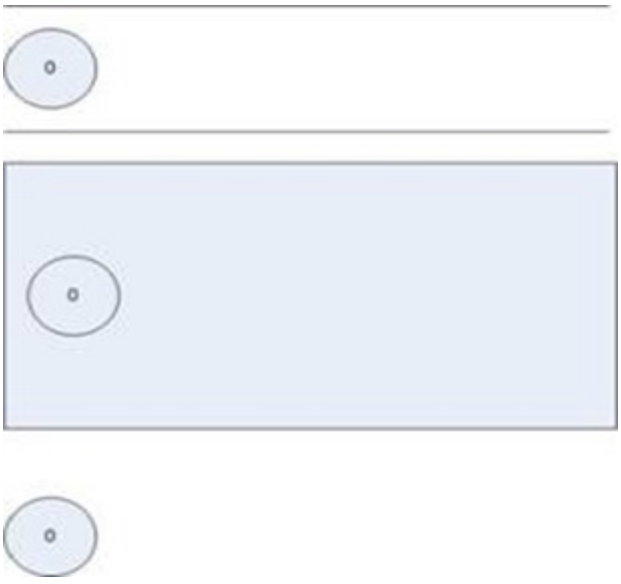


现在开始构造 DFA，具体算法如下：

1. 把{S}放进队列 L 和集合 D 中。其中 S 是 NFA 的起始状态。队列 L 放置的是未被处理的已经创建了的 DFA 状态，集合 D 放置的是已经存在的 DFA 状态。根据上文的讨论，DFA 的每一个状态都对应着 NFA 的一些状态。
2. 从队列 L 中取出一个状态，计算从这个状态输出的所有边所接受的字符集的并集，然后对该集合中的每一个字符寻找接受这个字符的边，把这些边的目标状态的并集 T 计算出来。如果 T∈D 则代表当前字符指向了一个已知的 DFA 状态。否则则代表当前字符指向了一个未创建的 DFA 状态，这个时候就把 T 放进 L 和 D 中。在这个步骤里有两层循环：第一层是遍历所有接受的字符的并集，第二层是对每一个可以接受的字符遍历所有输出的边计算目标 DFA 状态所包含的 NFA 状态的集合。
3. 如果 L 非空则跳到 2。

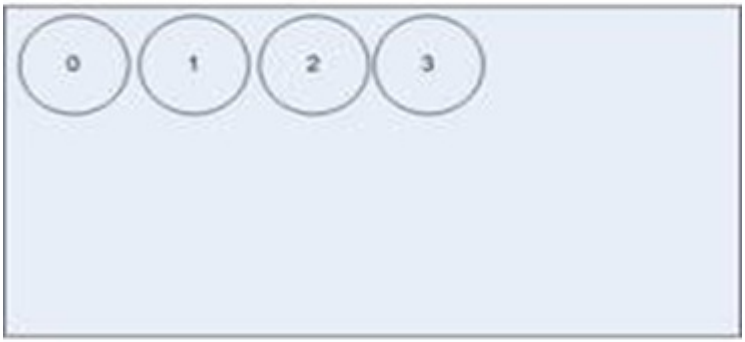
对于上图所示的状态机应用 DFA 构造算法

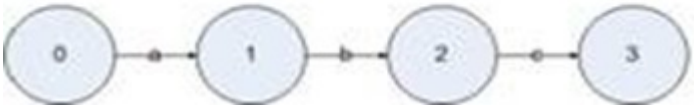
1. 首先执行第一步，我们得到：



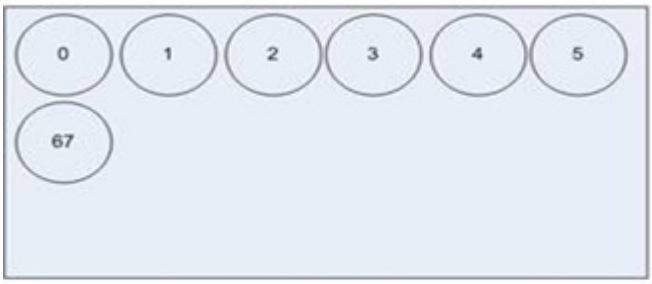
从上到下分别是队列 L、集合 D 和 DFA 的当前状态。就这样一直执行该算法直到状态

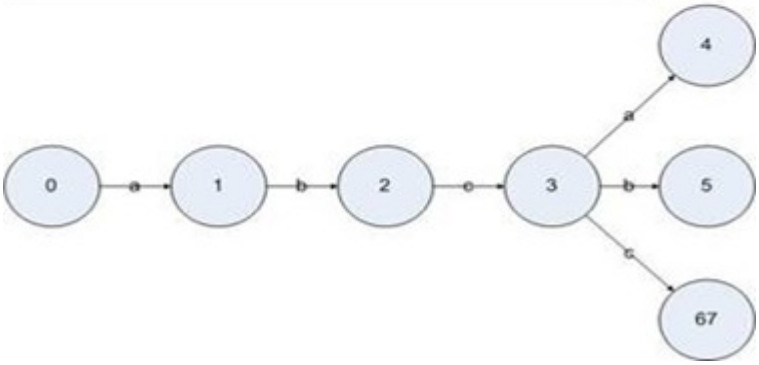
3 进入集合 D，我们得到：



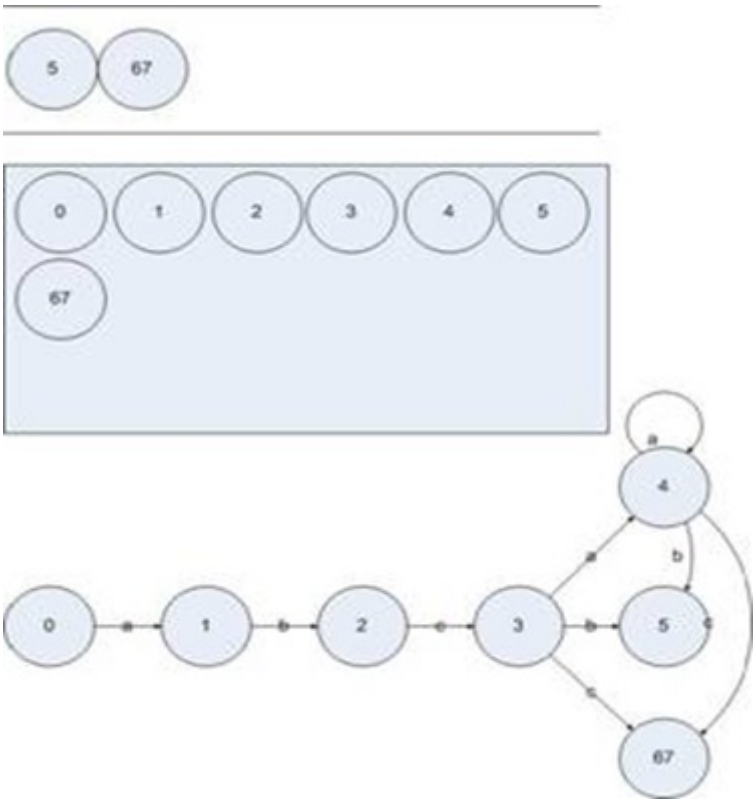


现在从队列 L 中取出{3}，经过分析得到状态集合{3}接受的字符集合为{a b c}。{3}读入 a 到达{4}，读入 b 到达{5}，读入 c 到达{6 7}。因为{4}、{5}和{6 7}都不属于 D，所以把它们都放入队列 L 和集合 D：

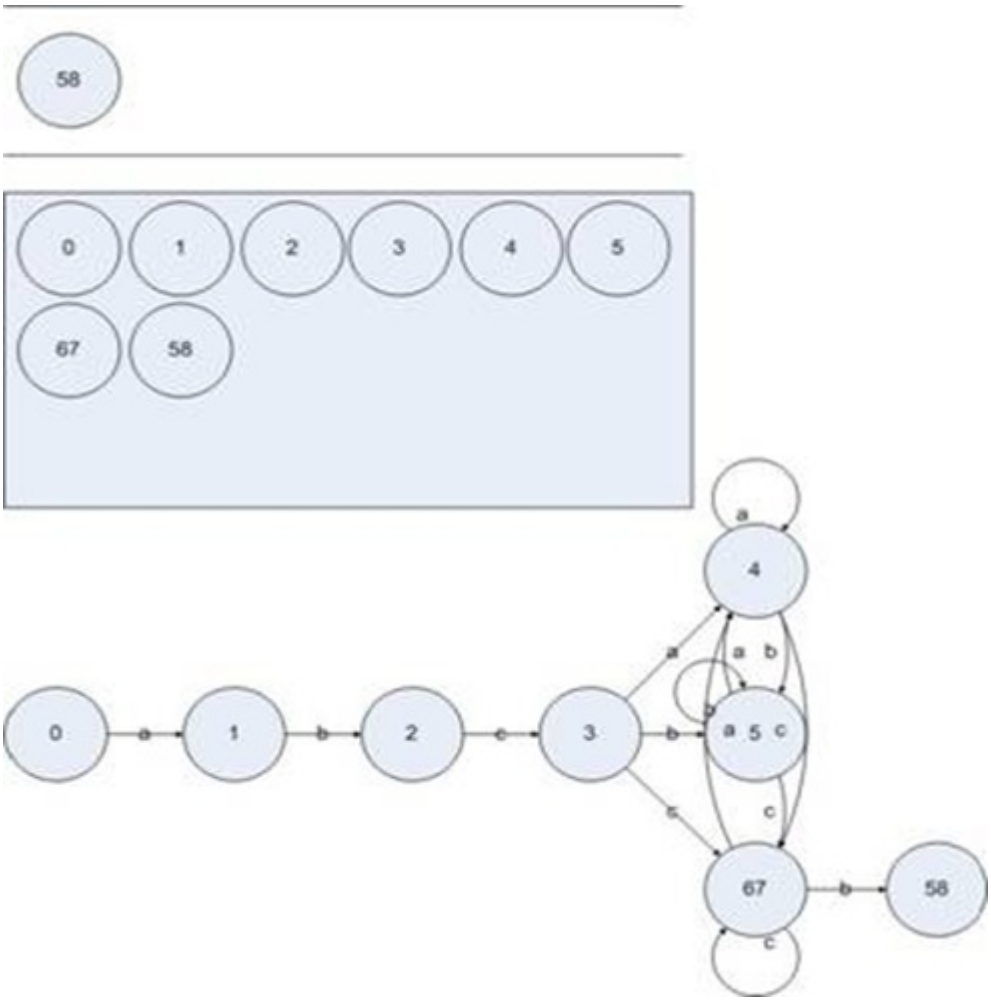




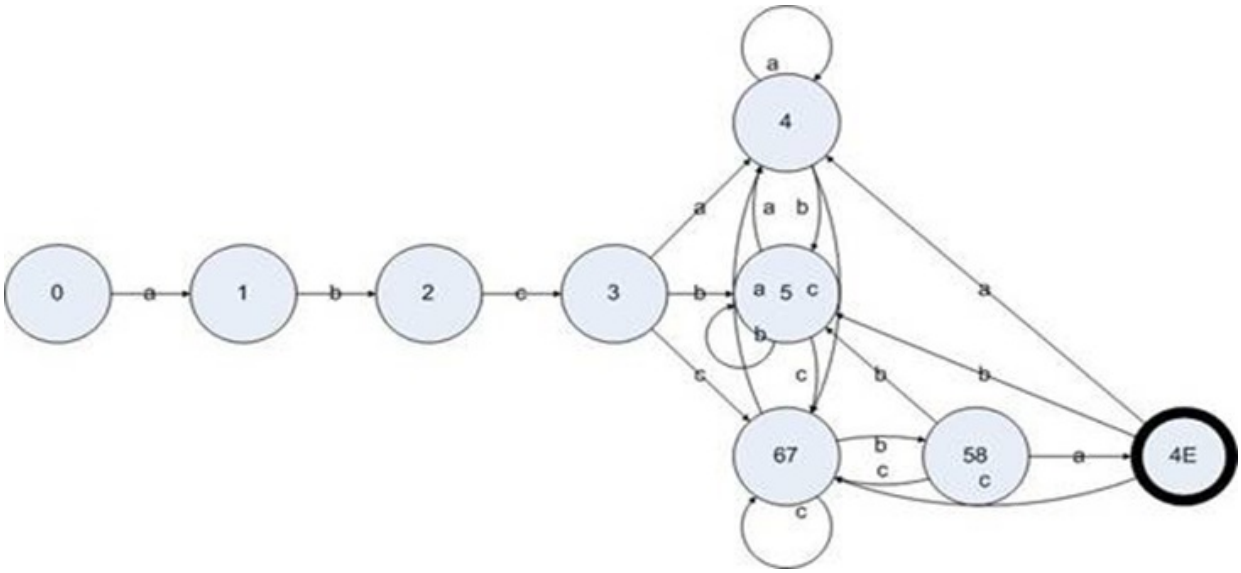
从队列中取出 4 进行计算，我们得到：



显然，对于状态{4}的处理并没有发现新的 DFA 状态。于是处理{5}和{6 7}，我们可以得到：



在处理状态{5}的时候没有发现新的 DFA 状态，处理{6 7}在输入{a c}之后的跳转也没有发现新的 DFA 状态，但是我们发现了{6 7}输入 b 却得到了新的 DFA 状态{58}。把算法执行完，我们得到一个 DFA:

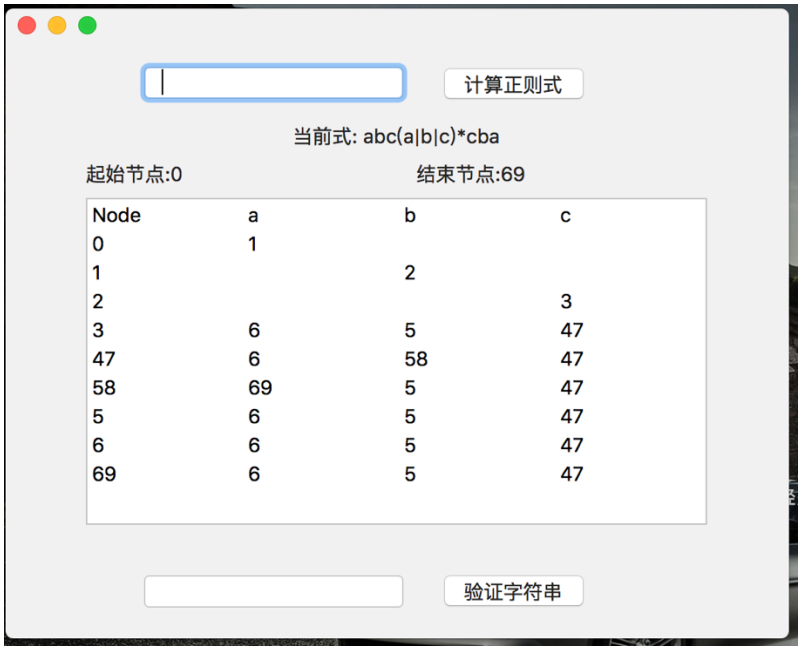


至此，对图 5.7 的状态机应用 DFA 构造算法的流程就结束了，我们得到了图 5.13 的 DFA，其功能与图 5.7 的 NFA 等价。在 DFA 中，起始状态是 0，结束状态是 4E。凡是包含了 NFA 的结束状态的 DFA 状态都必须是结束状态。

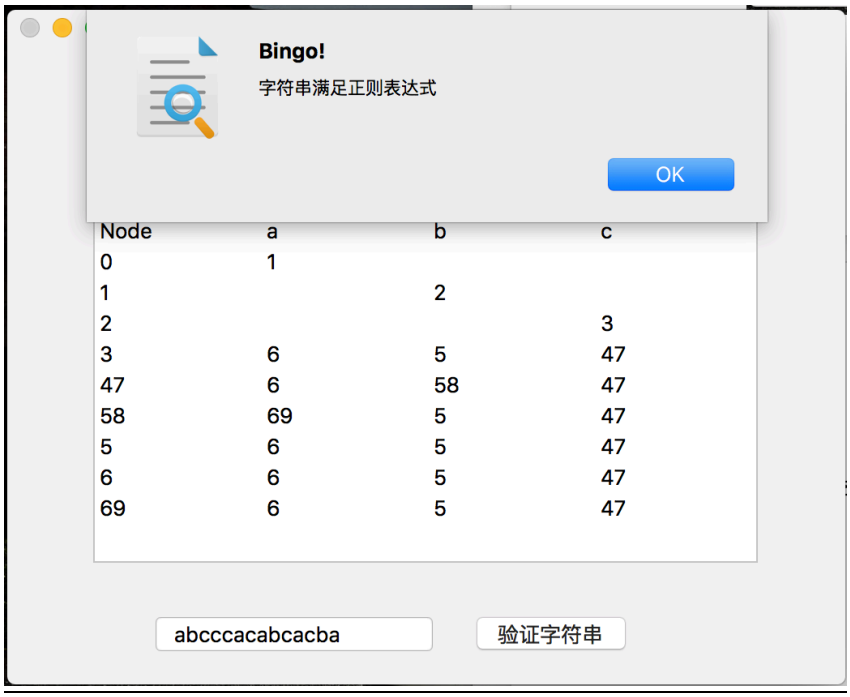
四、程序演示：

（注：表格中第一行为跳转字符、第一列为各节点的表示值）

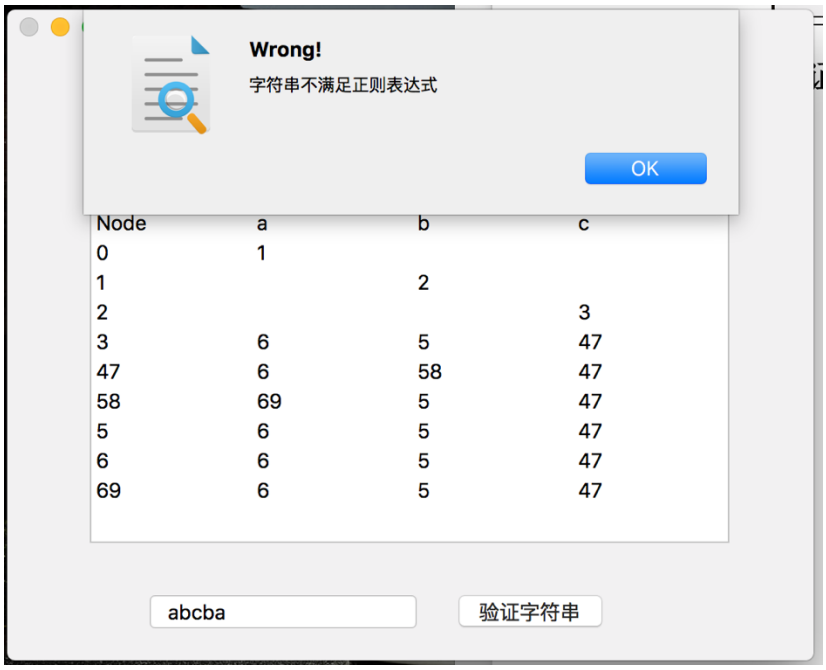
输入字符串 abc(a|b|c)*cba 构建 dfa 表：



验证字符串 abcccacabcacba：

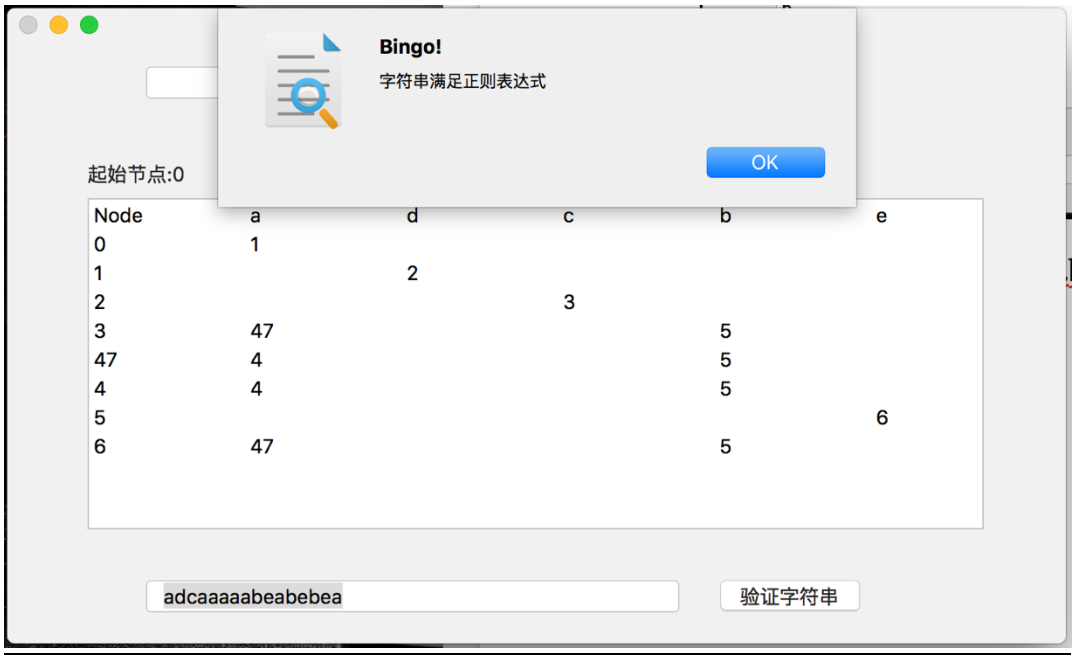


验证字符串 abcba：

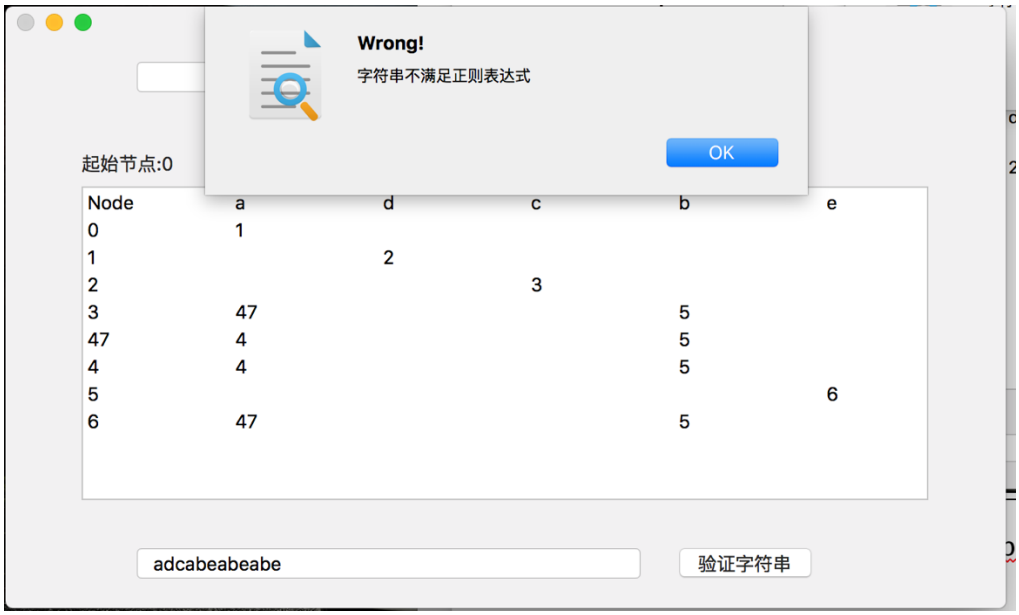


输入字符串 adc(a*be)*a 构建 dfa 表：

验证字符串 adcaaaaabeabebea



验证字符串 adcabeabeabe



五、参考网页：

正则表达式 DFA 构造方法

http://blog.csdn.net/chinamming/article/details/17166577