

**杭州电子科技大学**

**《编译原理课程实践》**

**实验报告**

题 目：词法分析核心算法实现

学 院： 计算机学院

专 业： 计算机科学与技术

班 级： 22052312

学 号： 22050201

姓 名： 黄江晔

完成日期： 2024.11.2

**一、实验目的**

**（1）掌握基本概念与转换方法：**

深入理解正规表达式、非确定有限自动机（NFA）和确定有限自动机（DFA）的基本概念，以及它们之间的转换方法。

**（2）了解构建与转换过程：**

熟悉NFA的构建过程、NFA到DFA的子集构造算法，以及DFA最小化原理和Hopcroft算法。

**（3）实现转换与验证：**

编程实现正规表达式到NFA、NFA到DFA的转换算法，并验证其正确性，特别是DFA的最小化过程。

**（4）设计合理数据结构：**

设计合理的数据结构，以支持自动机系列实验的连贯性。

**（5）提升编程能力与算法优化：**

通过实践提升编程能力、算法设计与优化技能，加深对编译原理的理解

**二、实验内容与实验要求**

**实验内容：**

理解正规表达式与非确定有限自动机（NFA）之间的关系，NFA用于词法分析，通过算法可以将正规表达式转换为NFA，从而实现字符串的模式匹配。

理解NFA与确定有限自动机（DFA）之间的关系，DFA是NFA的确定版本，利用子集构造算法可以将NFA转换为等价的DFA。

理解DFA最小化的原理，特别是Hopcroft算法，通过合并等价状态来减少DFA的状态数。

**实验要求：**

**实验一：NFA构造**

（1）输入输出要求：输入是正规表达式和多个测试字符串；输出是NFA的状态集合及其转换关系，并指明每个测试字符串是否被NFA接受。

（2）算法要求：实现Thompson构造法，支持基本的正规表达式运算符（连接、或、闭包）。

（3）数据结构要求：设计表示NFA的合理数据结构，包括状态集、状态转移表、初始状态和接受状态，支持状态的高效增删查操作。

（4）程序要求：使用C/C++、Java、Python等语言编写，代码结构清晰并具备良好的注释。

（5）实验报告要求：包含实验目的、算法设计、实现过程、测试结果和问题分析。

**实验二：DFA构造**

（1）输入输出要求：输入是NFA（状态集、转换表、初始状态和接受状态集合）和多个测试字符串；输出是DFA的状态集合及其转换关系，指明每个测试字符串是否被DFA接受。

（2）算法要求：实现子集构造算法，处理ε-闭包及其状态转换，生成对应的DFA。

（3）数据结构要求：设计DFA的数据结构，包括状态集合、转换关系、初始状态和接受状态集合。

（4）实验报告要求：包含实验目的、子集构造算法的原理、步骤及数据结构设计思路。

**实验三：DFA最小化**

（1）输入输出要求：输入是DFA（状态集合、状态转换表、初始状态和接受状态集合）；输出是最小化后的DFA状态集合及其转换关系，指明最小化前后的状态数和状态转换关系。

（2）算法要求：实现Hopcroft算法，支持状态等价性判定及状态的合并操作。

（3）数据结构要求：设计适合Hopcroft算法的高效数据结构，保持与前两次实验数据结构一致。

（4）实验报告要求：包含实验目的、Hopcroft算法的原理和实现步骤，分析最小化前后的差异。

**三、设计方案与算法描述**

**步骤一：正规表达式从中缀形式变换为后缀形式**

后缀表达式是一种不需要括号的表达式，它将运算符放在操作数的后面，遵循栈的先进后出原则，可以方便地用计算机进行求值，而不需要考虑运算符的优先级和结合性。

该步骤的目标是将给定的正则表达式（中缀形式）转换为后缀形式（逆波兰表示法）。为了做到这一点，Shunting Yard 算法被应用在正则表达式上。算法步骤如下：

**1. 处理字符类和连字符**

字符类处理：首先，代码将正则表达式中的字符类（[...]）转换为合适的形式，以便后续处理。这里的规则是将字符类 [abc] 转换为 (a|b|c) 形式。换句话说，将字符类的每个字符通过 | 操作符连接。

连字符处理：连字符（如 a-z）被转换为一系列字符的“或”连接，即 a|b|c|...|z。

**2. 插入连接符（.）**

连接符插入规则：在任何两个相邻的字符之间插入连接符 .（表示连接操作）。这种情况发生在以下几种情况：

如果前一个字符是操作符（如 \*, + 等），而下一个字符不是操作符。

如果前一个字符是字面量字符（即非操作符字符），而下一个字符是字面量字符或左括号 (。

**3. 使用 Shunting Yard 算法转换**

初始化：定义操作符优先级字典 operators，其中包括：

* \* 的优先级是 5
* + 的优先级是 4
* ? 的优先级是 3
* . 的优先级是 2
* | 的优先级是 1

初始化空字符串postfix存储后缀表达式，空字符串stack用于存放操作符。

遍历表达式：对每一个字符 c 进行如下处理：

* 左括号 (：将左括号 ( 压入栈中。
* 右括号 )：从栈中弹出所有操作符，直到遇到左括号 (，将这些操作符添加到后缀表达式中。
* 操作符（如 \*, +, |, 等）：根据当前操作符的优先级，将栈中优先级大于或等于当前操作符的操作符弹出并添加到后缀表达式中，然后将当前操作符压入栈中。
* 操作数（即字面量字符或操作数）：直接将操作数添加到后缀表达式中。

**4. 清空栈**

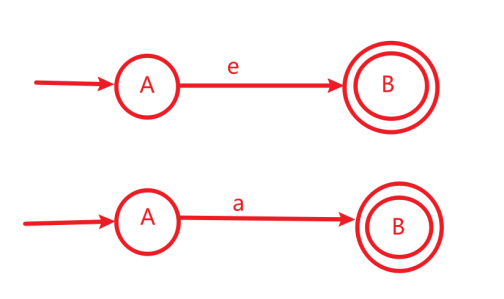
当遍历完正则表达式后，将栈中剩余的操作符全部弹出并添加到后缀表达式中。

**5. 返回结果**

最终，返回的 postfix 字符串即为转换后的后缀正则表达式。

**步骤二：按照Thompson构造法的规则，实现单个符号、连接、并联、闭包这几种基本转换**

**（1）对于正则表达式为ε或者只由一个符号构成的情况，则无需继续递归**。





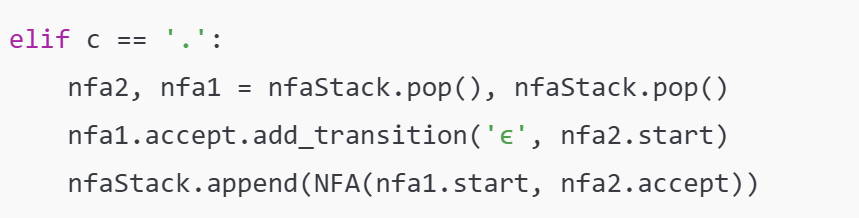
**（2）连接（ab）**

NFA（a）的初态成为新的NFA的初态。

原来NFA(a)的终态成为 NFA(b)的初态。

原来NFA(b)的终态成为新的NFA的终态。





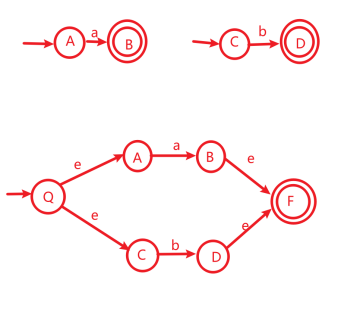
连接操作符将两个 NFA 合并在一起。在这里，nfa1 和 nfa2 分别是栈顶的两个 NFA。

连接操作的实现是将 nfa1 的接受状态通过空转移（ϵ）指向 nfa2 的起始状态，然后将 nfa1 的起始状态和 nfa2 的接受状态构成新的 NFA。

**（3）并联（a|b）**

通过ε转移，新NFA的初状态可以直接到达NFA(a)或NFA(b)的初态。

NFA(a)或NFA(b)原来的终态可以通过ε转移直接到达整个NFA的新终态。





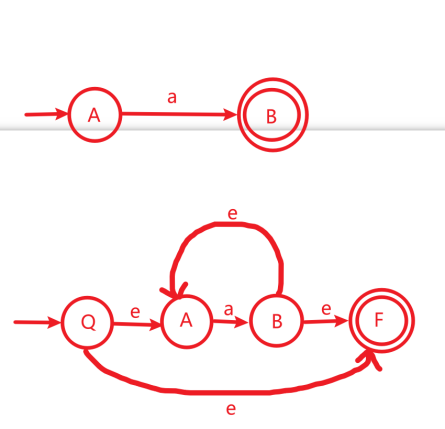
并联操作符需要创建一个新的 start 和 accept 状态，作为两个 NFA 的公共入口和出口。

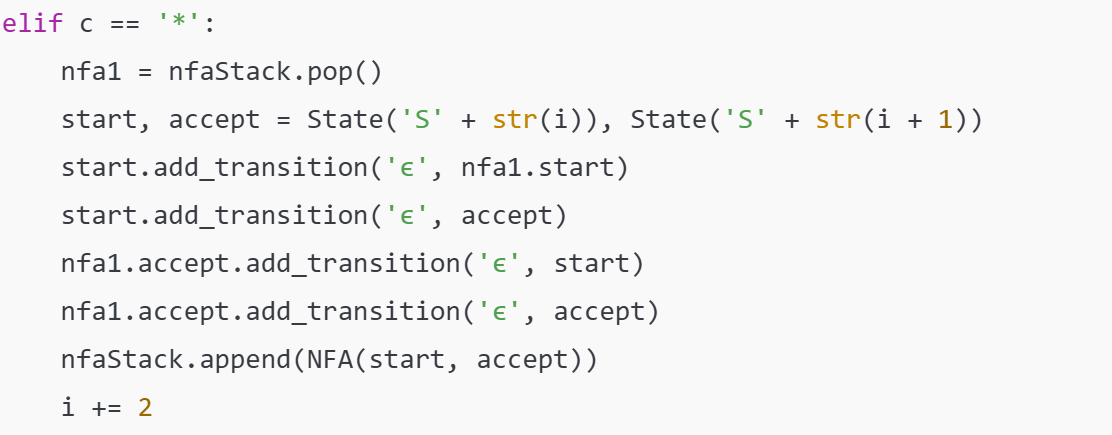
具体实现是：从新创建的 start 状态通过空转移 ϵ 跳转到 nfa1 和 nfa2 的起始状态，从 nfa1 和 nfa2 的接受状态通过空转移 ϵ 跳转到新的 accept 状态。

**（4）闭包（a\*）**

将新表达式的初态和终态以及夹在中间的子表达式的NFA(a)连接起来的ε转移，使得可以选择经过或者不经过子表达式。

从NFA(a)的终态到初态的ε转移，使得a可以重复任意多次。



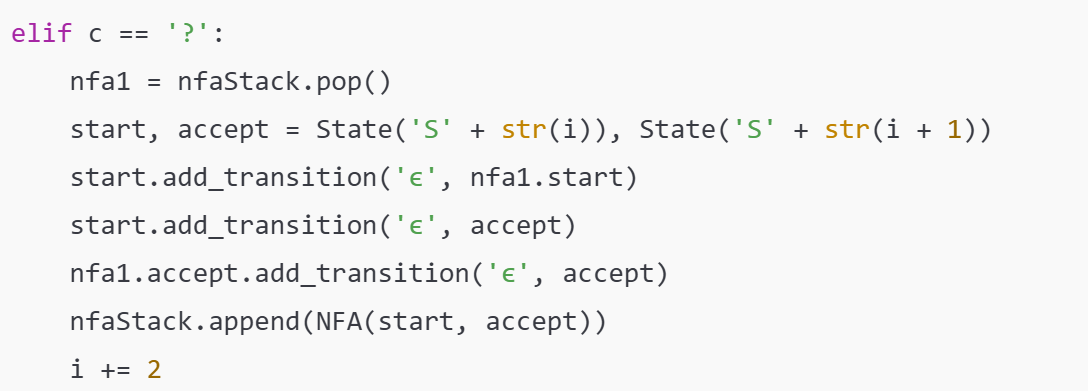


1. **闭包（a+）**



+ 表示至少一次匹配。与 \* 类似，但没有直接的空转移到 accept 状态，保证至少执行一次匹配。

1. **闭包（a？）**



? 表示零次或一次匹配。它在 nfa1 的起始状态和 accept 状态之间插入空转移，允许零次或一次匹配。

**步骤三：将正则表达式转换成NFA**

首先，创建一个空的堆栈 nfaStack，它将用于存储构建过程中的中间NFA。遍历正则表达式,依次读取后缀表达式中的每个符号（字符或运算符），根据不同的符号执行不同的操作。当遇到一个普通字符（如字母、数字等），创建一个简单的NFA，其中有一个起始状态和一个接受状态，并且在它们之间添加一个对应字符的转换（即从起始状态到接受状态的转换），将生成的NFA推入堆栈。

当遇到\*时，表示对栈顶的NFA应用克林闭包操作，从当前NFA中弹出一个NFA。创建一个新的起始状态和接受状态，新的起始状态通过空字符（ϵ）转换到原NFA的起始状态，同时也能直接跳转到新的接受状态。原NFA的接受状态通过空字符（ϵ）转换回新的起始状态，也通过空字符转换到新的接受状态。将修改后的NFA重新推入堆栈。

当遇到.时，表示连接操作，要求将两个NFA连接在一起。从堆栈中弹出两个NFA。将第一个NFA的接受状态通过空字符转换到第二个NFA的起始状态，形成连接。将第一个NFA的起始状态和第二个NFA的接受状态用作新的起始状态和接受状态，生成新的NFA。将生成的NFA重新推入堆栈。

当遇到|时，表示选择操作（即或操作），需要构造两个NFA之间的选择。从堆栈中弹出两个NFA。创建一个新的起始状态，并通过空字符（ϵ）转换到两个NFA的起始状态。将两个NFA的接受状态通过空字符（ϵ）转换到一个新的接受状态。将生成的NFA重新推入堆栈。

遇到+，与\*相似，但要求至少执行一次转换。从堆栈中弹出一个NFA。创建新的起始状态和接受状态，类似于\*，但这个NFA的接受状态必须回到原起始状态，并且还可以通过空字符转换到新的接受状态。将生成的NFA重新推入堆栈。

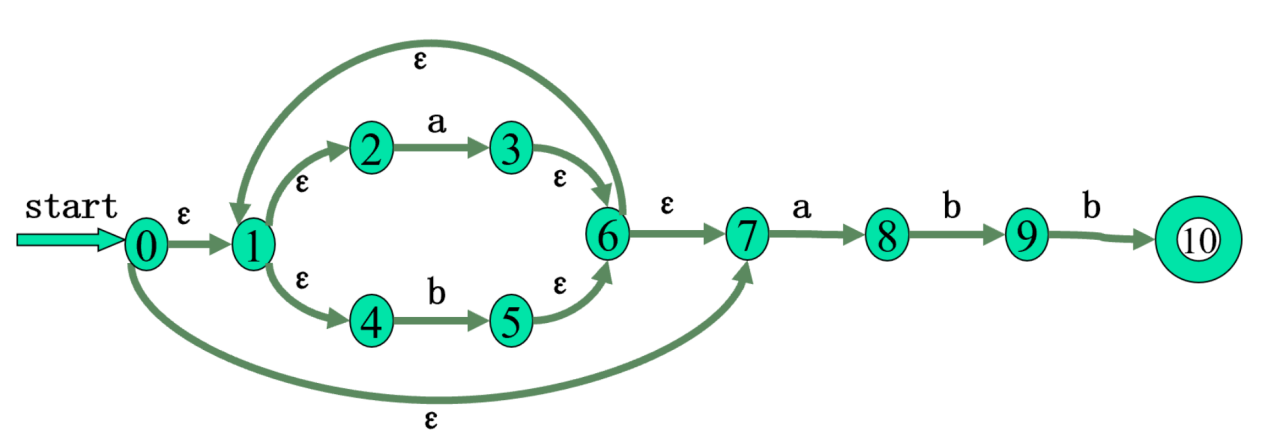
当遇到?时，表示该部分为可选，即匹配0次或1次。从堆栈中弹出一个NFA。

创建新的起始状态和接受状态，新的起始状态通过空字符转换到原始NFA的起始状态，原NFA的接受状态通过空字符转换到新的接受状态。将生成的NFA重新推入堆栈。

当遍历完整个后缀正则表达式后，堆栈中会剩下一个NFA，这个NFA即为对应正则表达式的NFA表示。返回最终生成的NFA，它包含了从正则表达式到NFA转化的所有状态、转换和接受状态。

**步骤四：计算NFA各状态的ε-闭包，并根据子集构造算法生成DFA的状态**

ε闭包是包含这个集合和所有通过空转移可以到达的状态的集合。简单来说，就是这个状态，通过空串的转移，最终能到达哪些状态（包括它自己）。

 例如上图中，0状态通过ε可以转移到1，2，4，7状态，因此ε-closure(0) = {0,1,2,4,7} 。

**1. ε-闭包的计算：**

目标：

通过计算给定状态集合的ε-闭包，获取所有通过ε转换（即空字符转换）能够到达的状态。这个操作是NFA到DFA转换的第一步。

操作：

给定NFA的某个状态集合（如NFA的起始状态），从这些状态出发，沿着所有的ε转换递归遍历所有相关的状态。最后得到一个包含所有这些状态的集合，称为该状态集合的ε-闭包。

实现：

从给定的状态集合出发，将这些状态加入到一个栈（stack）中，逐一遍历栈中的状态。对于每个状态，检查它的转换是否是ε。如果是，将目标状态加入栈中，并继续遍历。最终，栈中的所有状态都可以通过ε转换到达，因此它们构成了该状态集合的ε-闭包。

**2. 子集构造算法：**

子集构造算法用于从NFA生成DFA，它的核心思想是将NFA的状态集合作为DFA的状态，并通过状态集合间的转换构建DFA的转移表。

DFA的初始状态由NFA的起始状态的ε-闭包组成。即，NFA的起始状态通过ε转换所能到达的所有状态构成DFA的初始状态。在DFA的构造过程中，每一个状态都是NFA状态的一个子集。因此，DFA的状态集是NFA状态集的所有子集。

对于每个DFA的状态（一个NFA状态的子集），对于每个输入符号（即NFA的每个非空符号），计算所有从该状态集合中的任一状态出发，根据该符号可以到达的NFA状态集合。将该状态集合的ε-闭包作为该符号的转移结果。DFA的接受状态是那些包含NFA的接受状态的子集。

**3. DFA的构建：**

队列和广度优先搜索（BFS）：子集构造算法通过广度优先搜索（BFS）遍历所有可能的状态集合（子集）。从DFA的初始状态开始，依次计算每个状态集合的转移，生成DFA的所有状态和转换。

已见状态集合：为了避免重复计算，使用一个seen集合来记录已经生成的状态集合。

记录转移：对于每个状态集合及其对应的输入符号，记录DFA的转移。

**4. DFA的最终状态表示：**

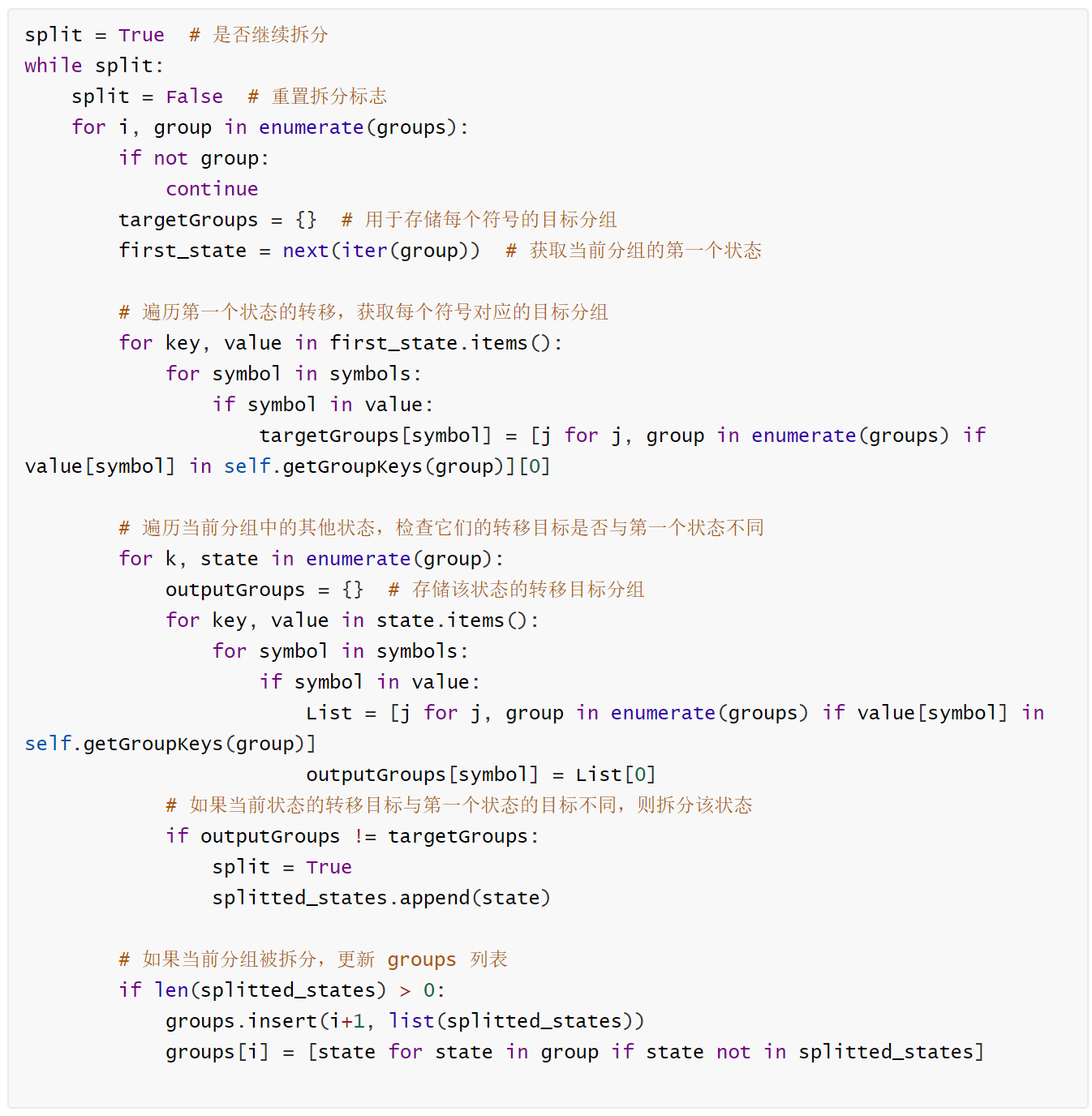
通过BFS过程，所有可能的状态集合被生成并标记为DFA的状态。对于每个状态，标记其是否为接受状态（即该状态集合中是否包含NFA的接受状态）。最终，DFA的所有状态、转移和接受状态都被记录下来，形成DFA的转移图。

**步骤五：最小化DFA【初始状态分割+使⽤Hopcroft算法细化状态分割】**

在最小化 DFA 时，首先需要对 DFA 的状态进行初始化分割，分为接受状态和非接受状态两组。final\_states 存储所有接受状态，non\_final\_states 存储所有非接受状态，然后将这两个组添加到 groups 列表中，作为初始分组。

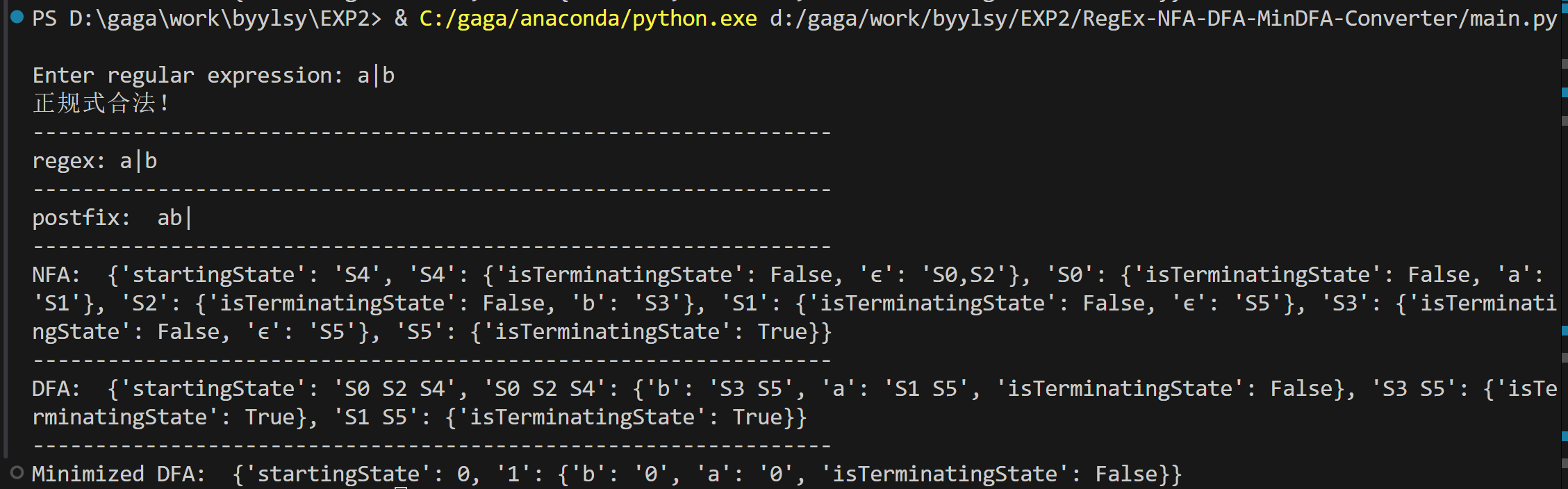


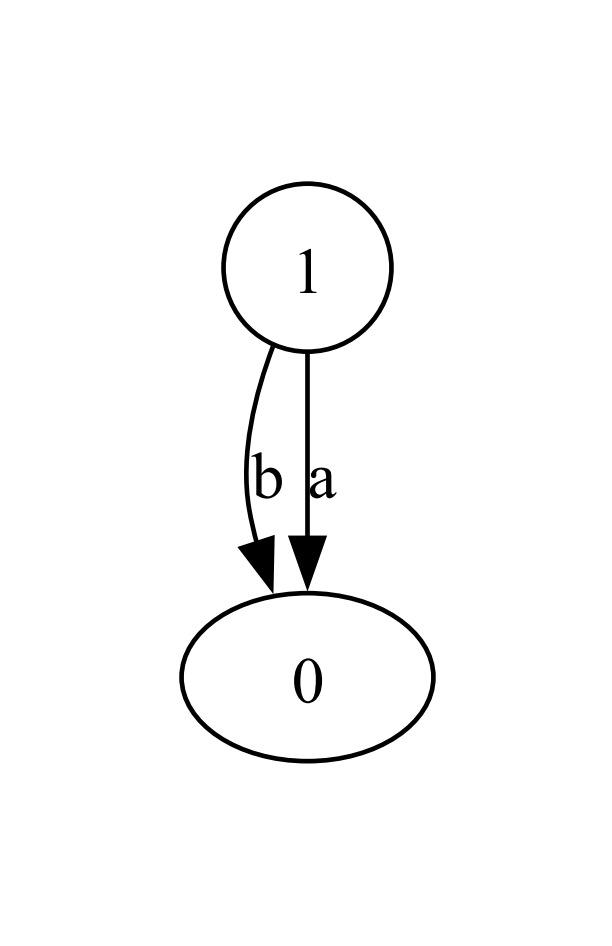
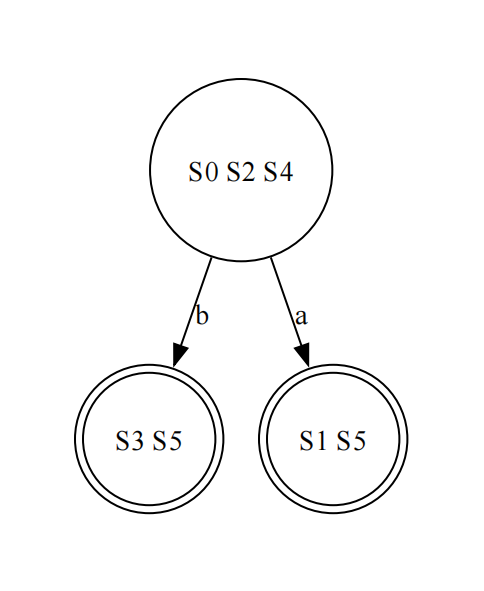
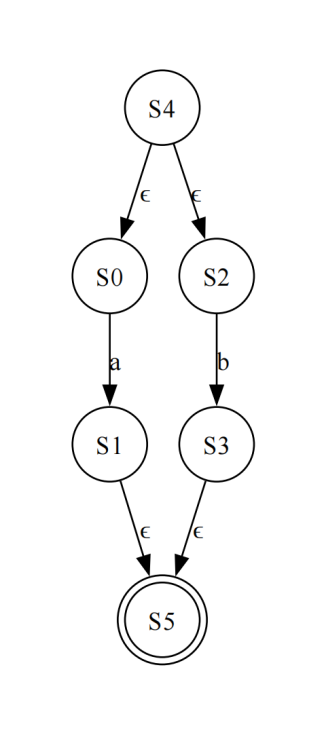
Hopcroft 算法会根据状态在每个符号上的转移关系，细化状态分割，直到没有更多的细化。在 while split 循环中，针对每个状态分组，检查该组内所有状态的转移目标，如果某个状态的转移行为与其他状态不同（即它的转移目标不同），则将其拆分到新的组，直到没有更多的拆分发生，split 标志变为 False，循环结束。



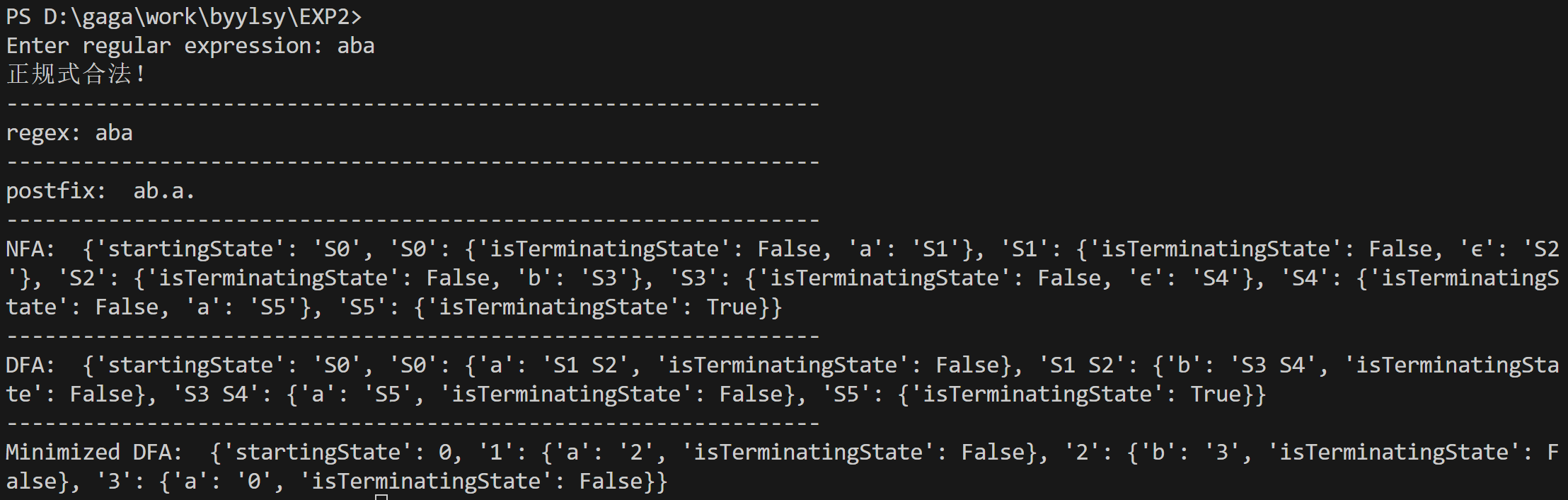
**四、测试结果**

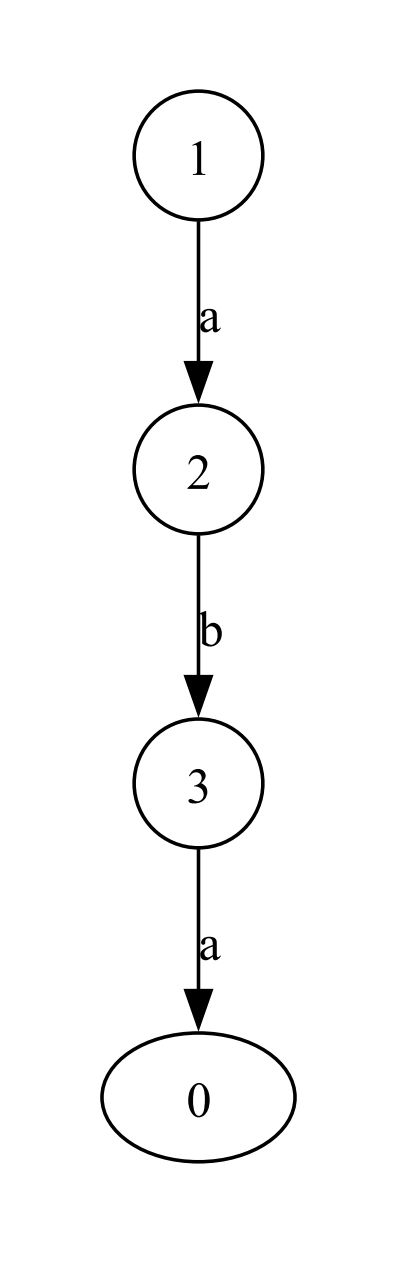
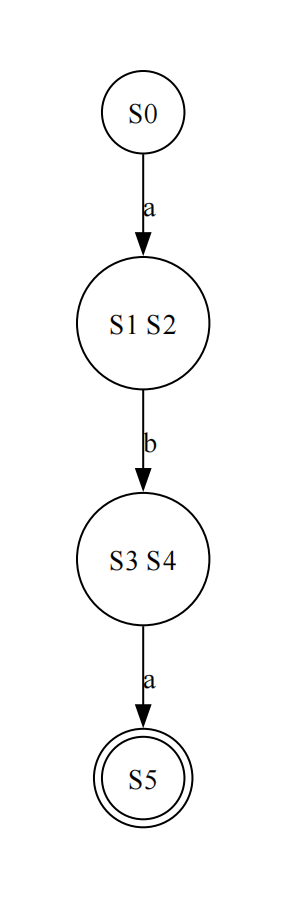
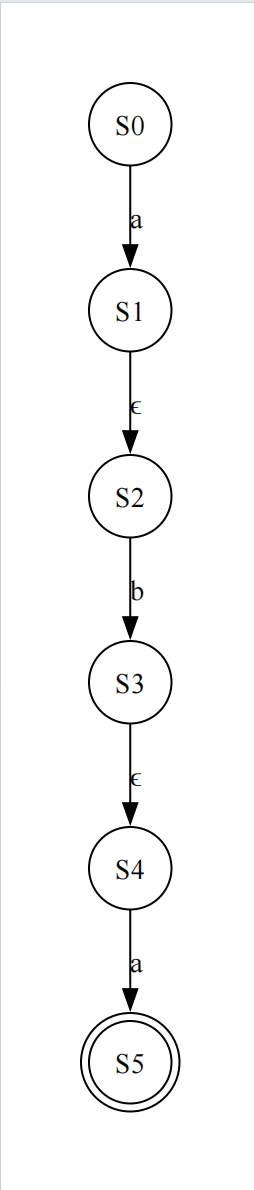
（1）a|b



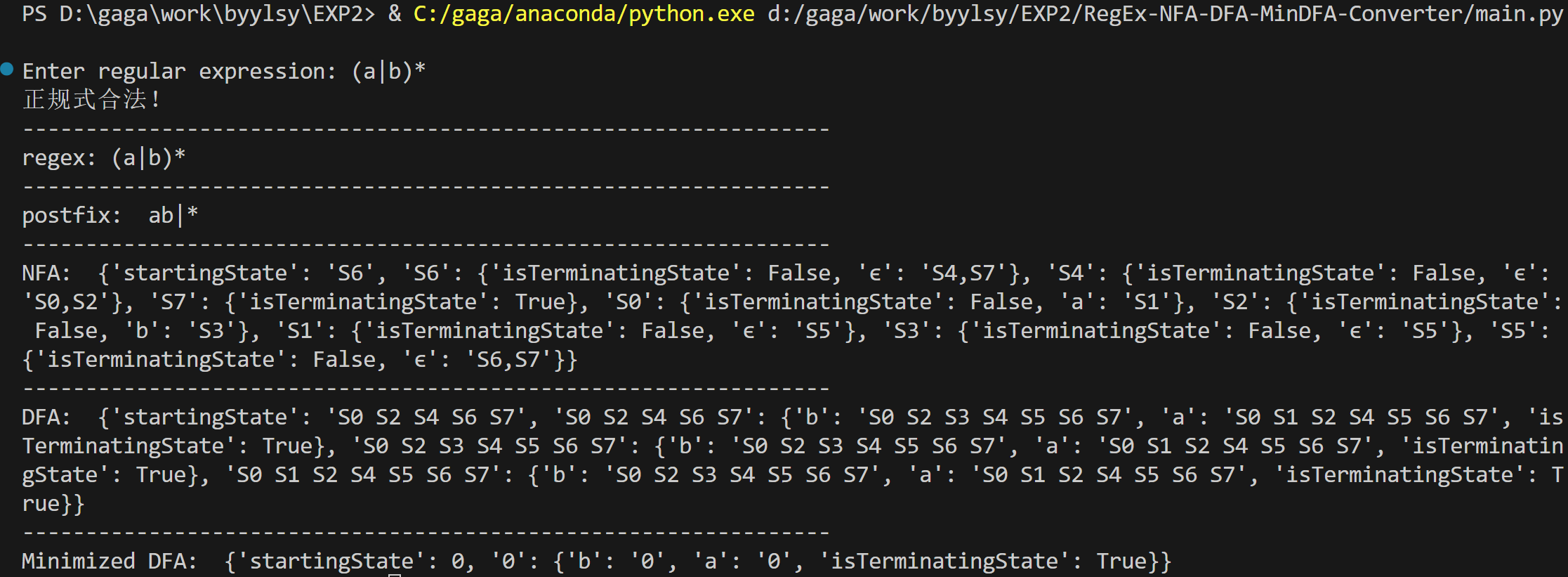


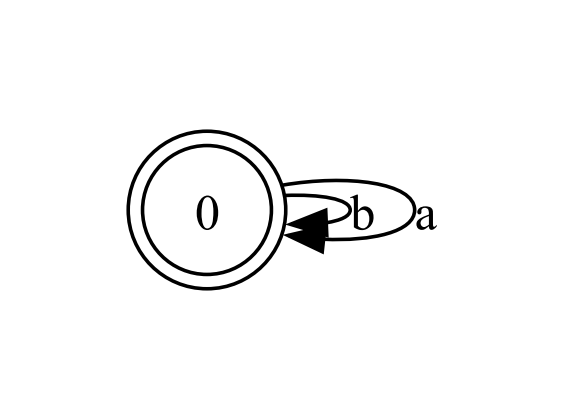
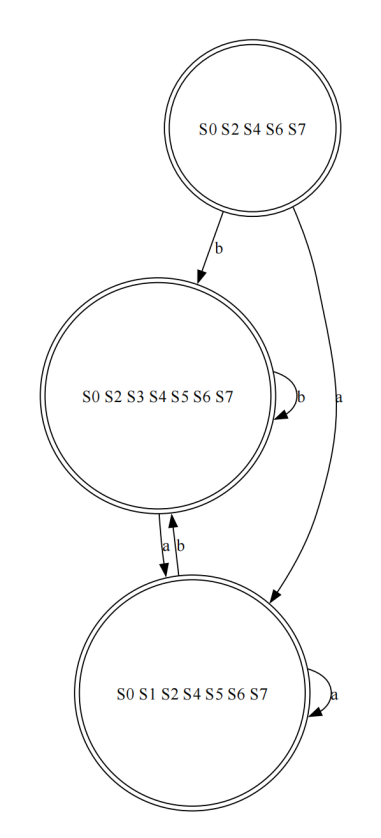
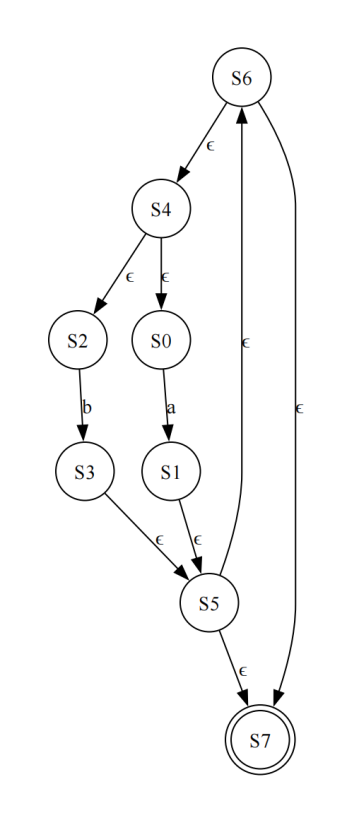
（2）aba



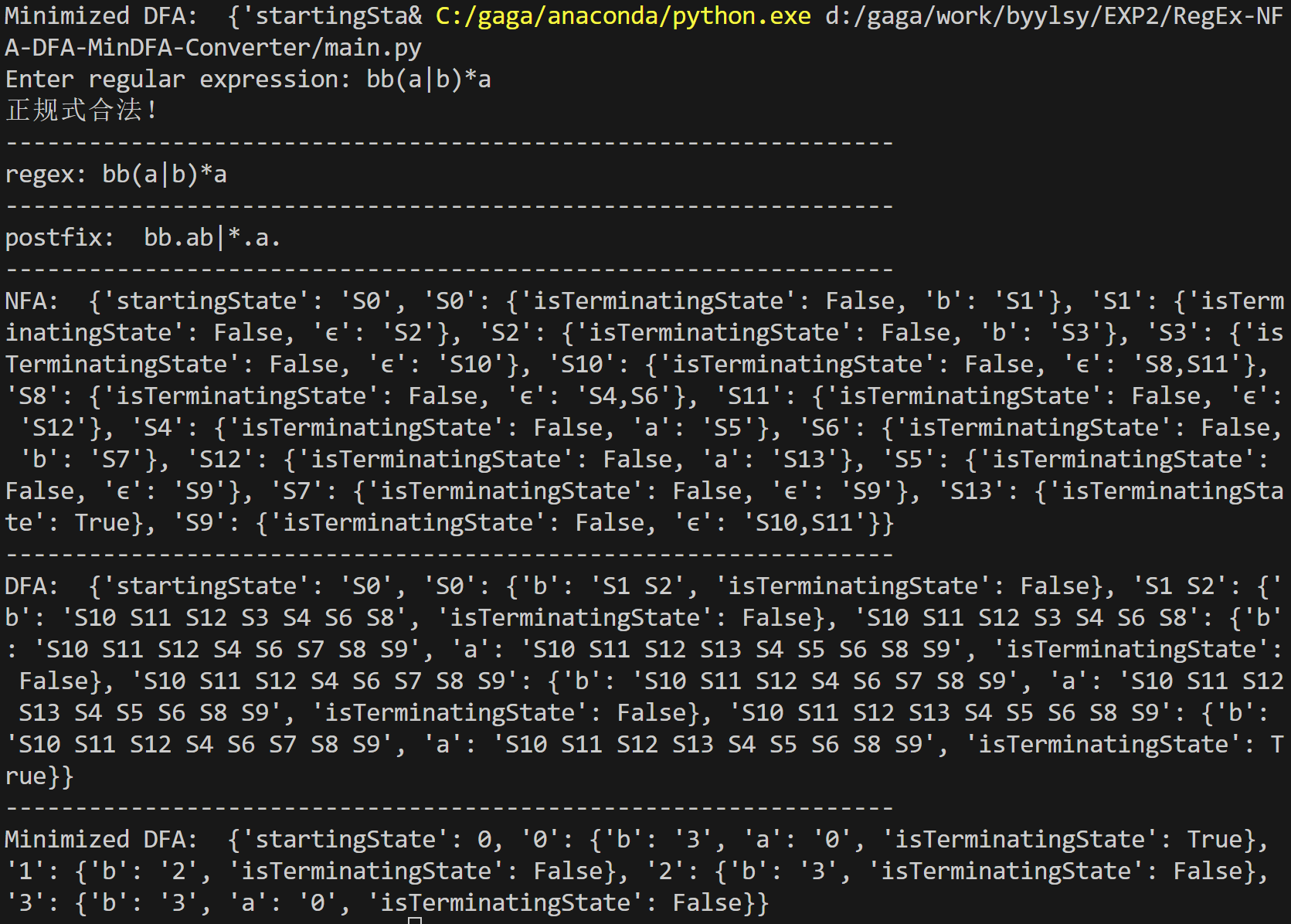


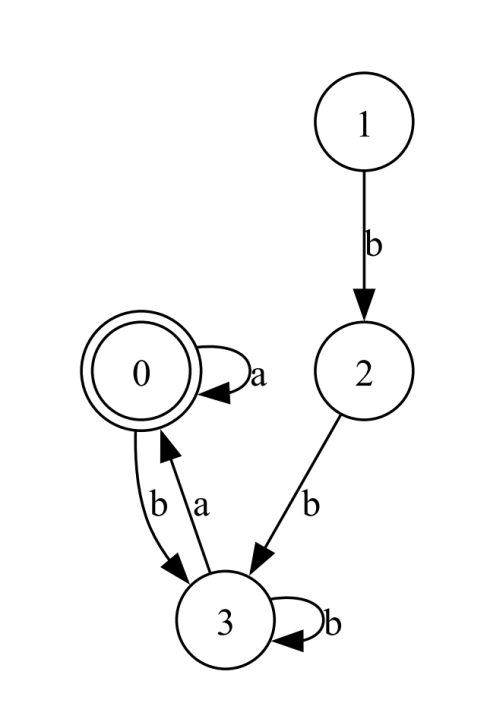
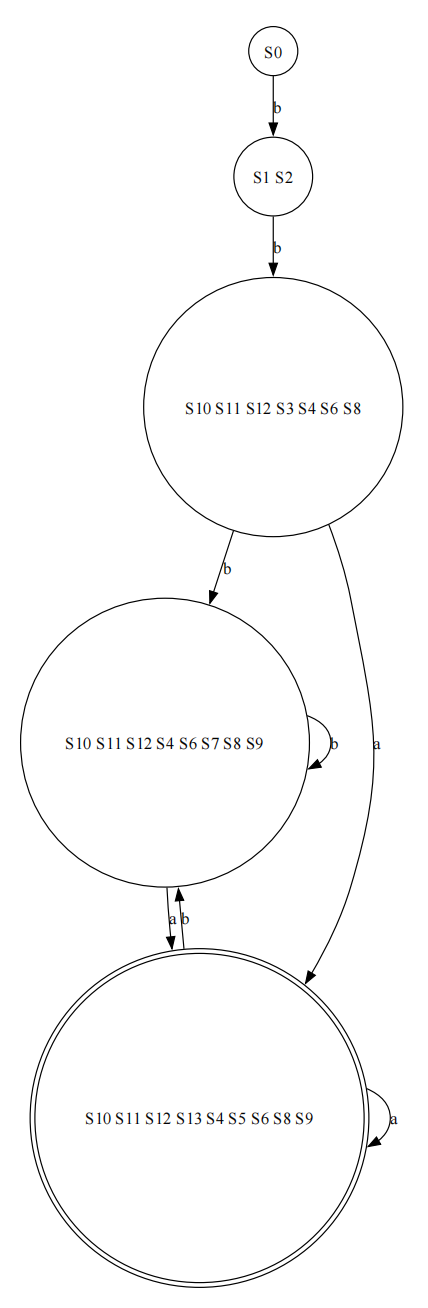
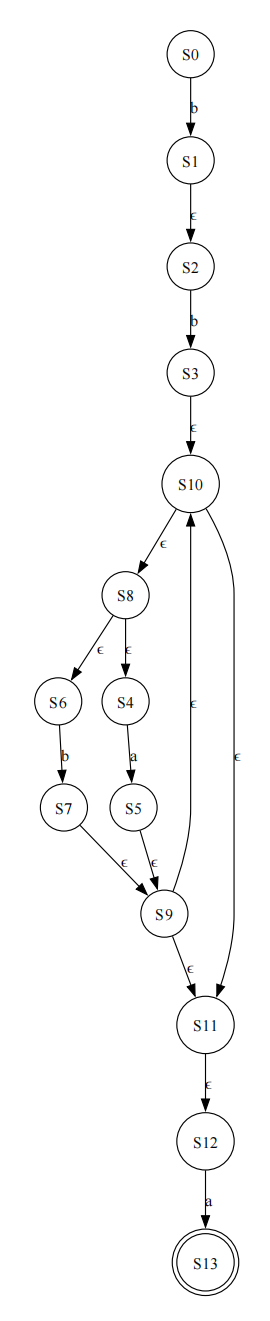
（3）(a|b)\*



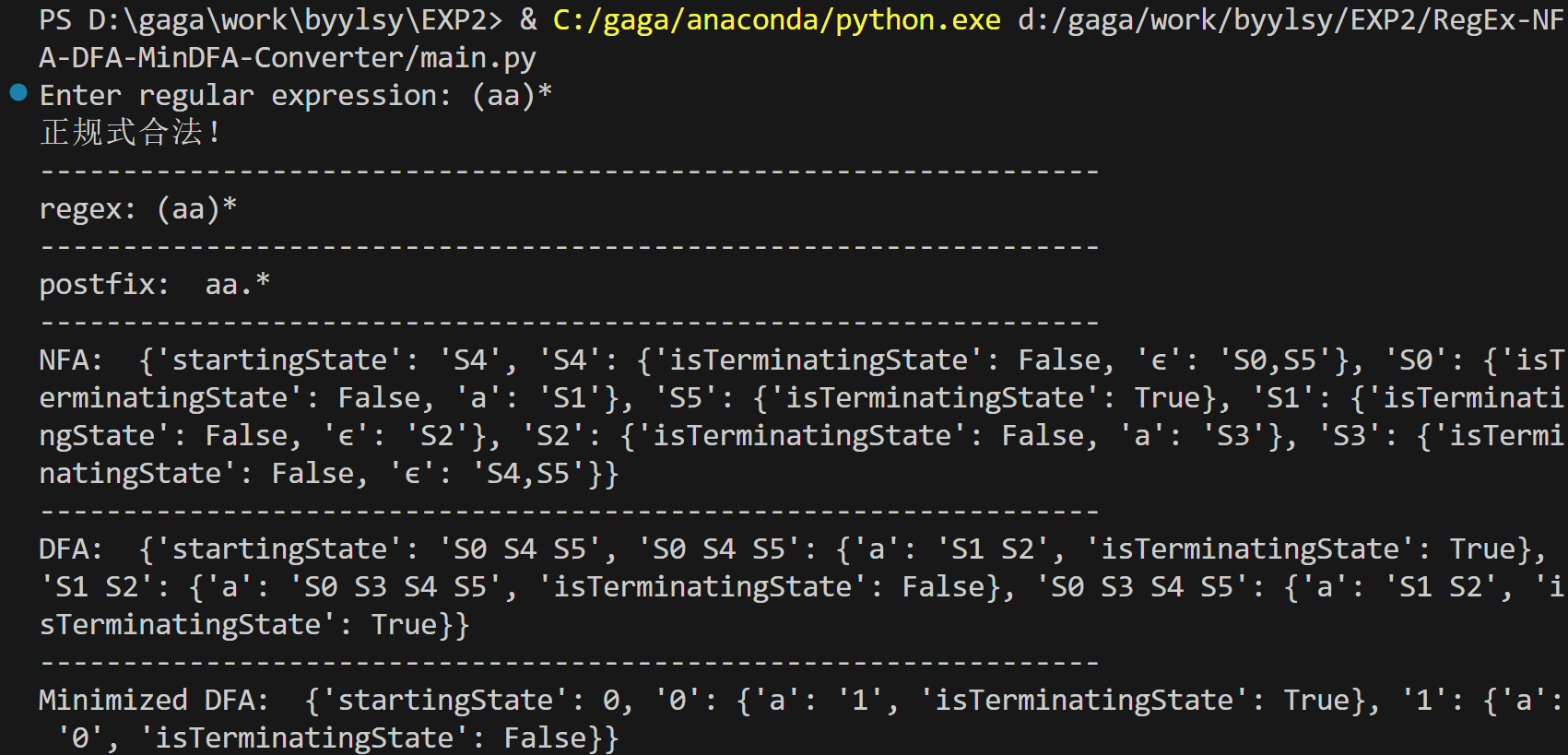


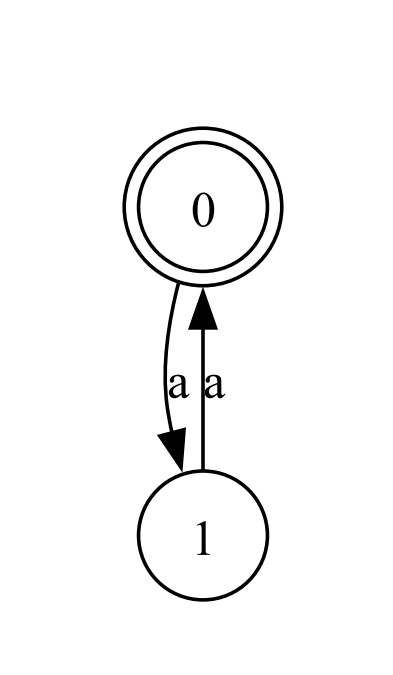
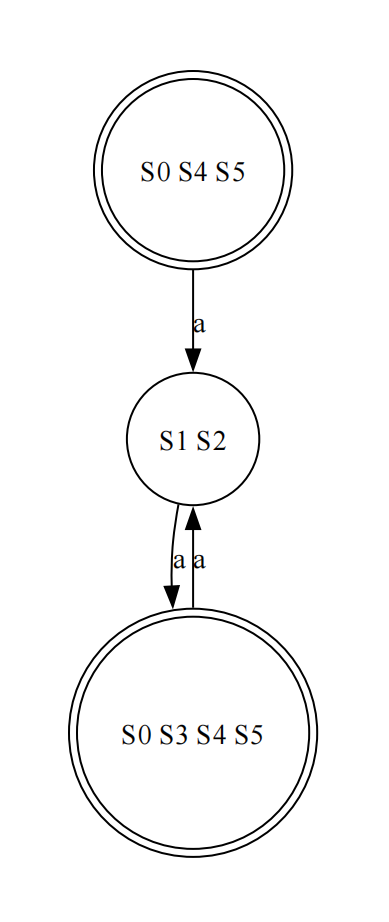
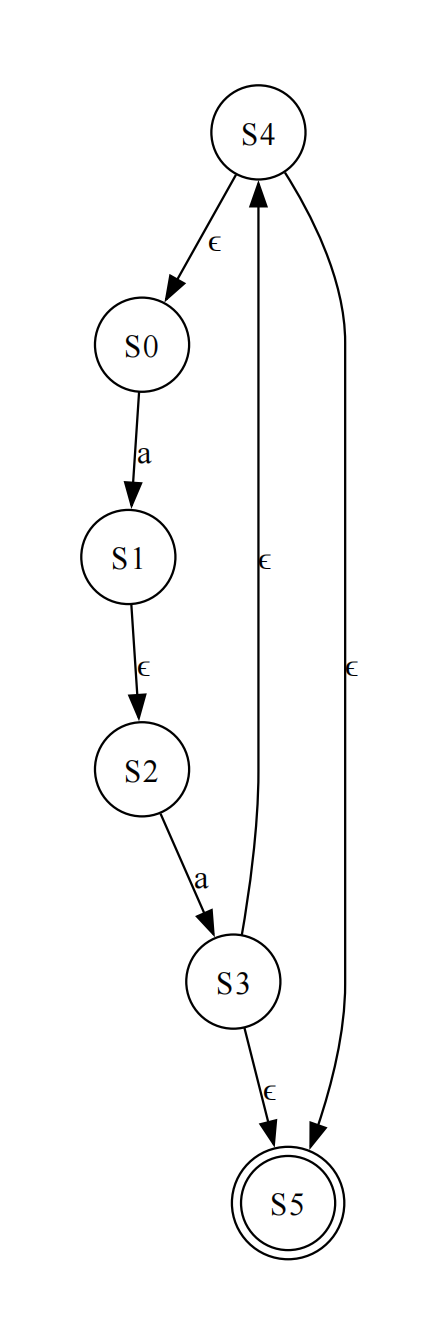
1. bb(a|b)\*a





（5）(aa)\*





**五、源代码**

**main.py**



**regex2postfix.py**



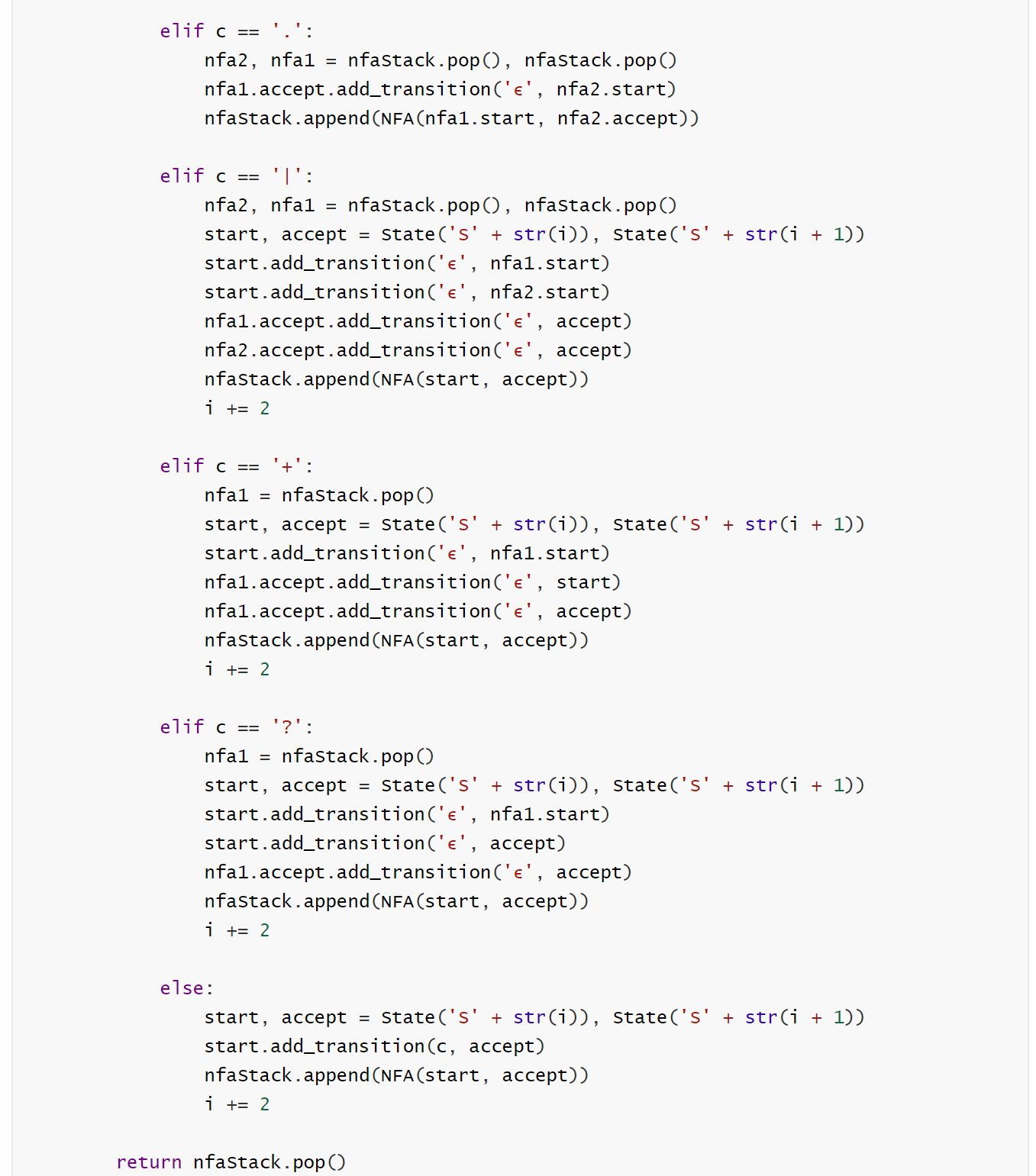




**postfix2nfa.py**





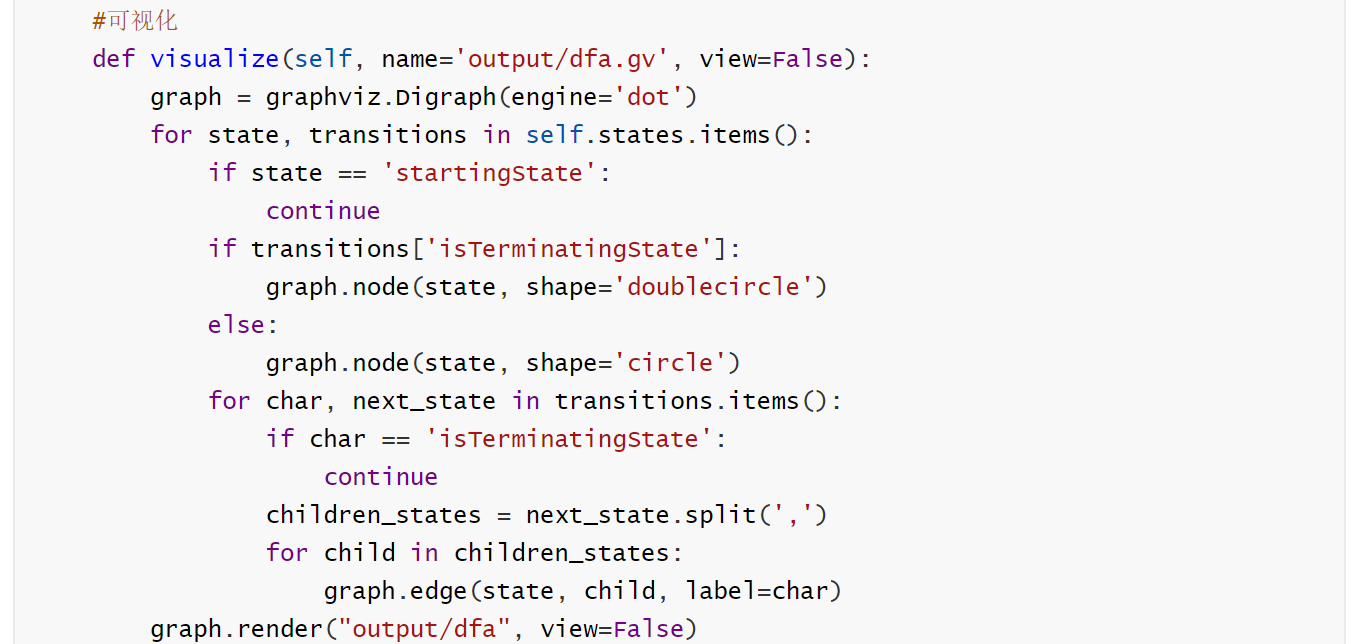




**nfa2dfa.py**





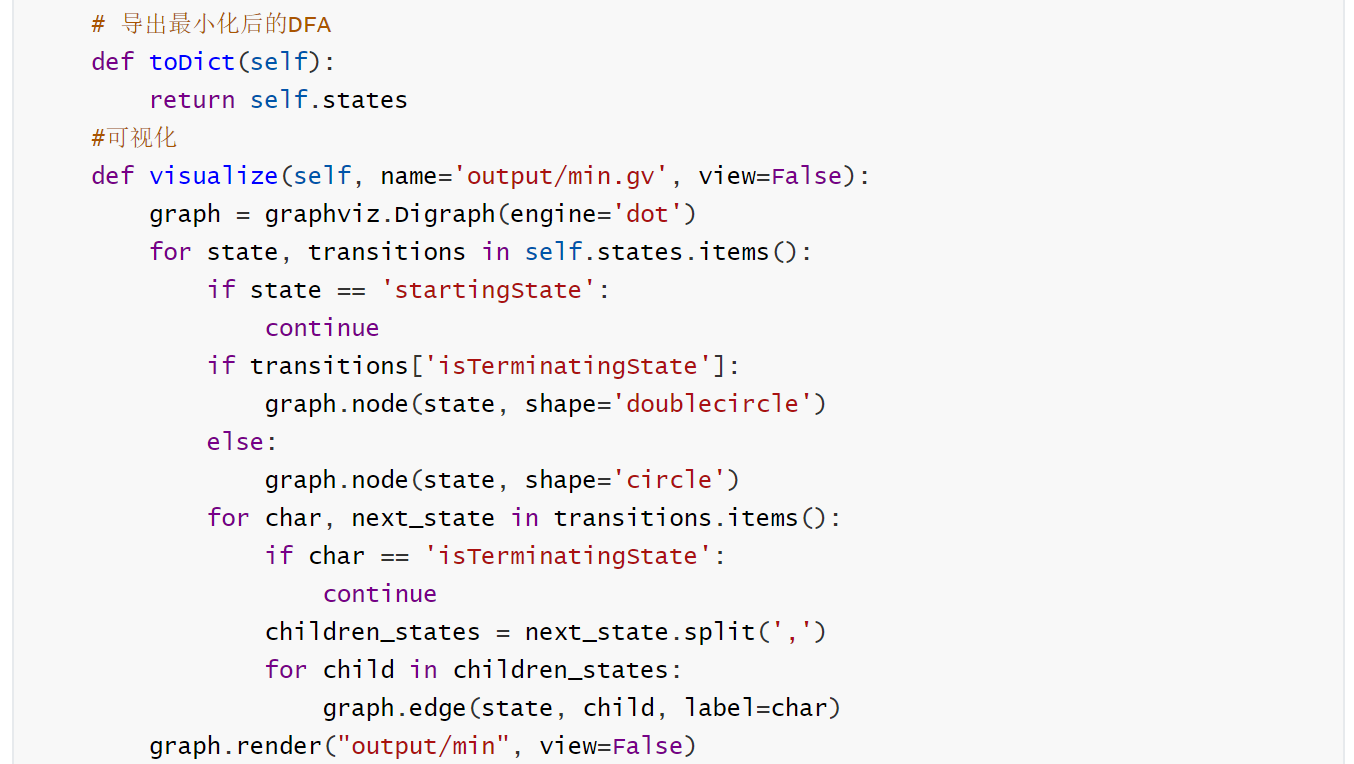


**dfa2min.py**









**State.py**

