词法分析器生成器子模块

软件学院

陈睿 141250013

141250013@smail.nju.edu.cn

2016.10.27

目录

[1. 目标 3](#_Toc465532711)

[2. 内容描述 3](#_Toc465532712)

[3. 构思与方法 3](#_Toc465532713)

[4. 假设 4](#_Toc465532714)

[4.1. 实验环境 4](#_Toc465532715)

[4.2. 输入及符号假设 4](#_Toc465532716)

[5. 有关自动机描述 4](#_Toc465532717)

[6. 数据结构描述 4](#_Toc465532718)

[7. 核心算法描述 6](#_Toc465532719)

[7.1. RE->NFA 6](#_Toc465532720)

[7.2. NFA->DFA 8](#_Toc465532721)

[8. 测试用例 9](#_Toc465532722)

[8.1. Test 1 9](#_Toc465532723)

[8.2. Test 2 11](#_Toc465532724)

[9. 问题与解决 12](#_Toc465532725)

[10. 总结与感受 13](#_Toc465532726)

1. 目标

主要目标是尝试进行词法分析器生成器的构造，使我可以更深入理解龙书上相关算法，并自己通过代码进行实现，同时也锻炼一下自己的能力。

1. 内容描述

本报告主要描述了词法分析器生成器中一部分子模块的构造，主要包括:从正则表达式推出NFA，再从NFA推出DFA这2个过程，以及以上2个过程的构思方法、具体实现过程。

1. 构思与方法

从整体思路来看，考虑到之前构造简易词法分析器的过程，最终核心还是要产生确定的有限状态自动机，即DFA，之后应该就可以生成相应的模板代码去用于分析程序。因此，主要分为由RE构造出DFA和由DFA生成模板代码2部分，在这里主要实现了由RE构造出DFA的过程，借鉴龙书上相关算法，结合一部分自己所学知识即可。

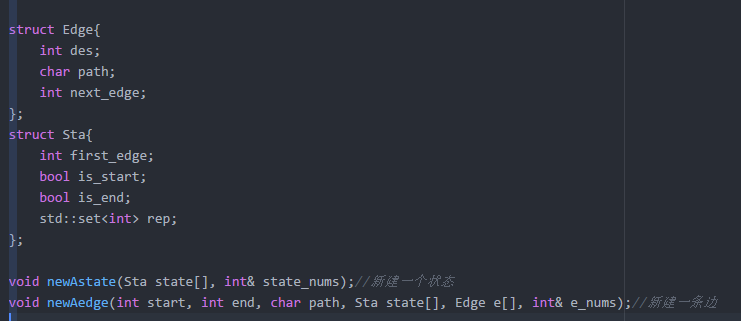
具体来看，代码部分有如下步骤：

1. 输入正则表达式
2. 对输入的正则表达式进行符号化补全，加入相关连接符号
3. 将中缀的正则表达式转换为后缀形式
4. 对确定的后缀表达式进行RE->NFA的过程
5. NFA->DFA
6. 输出所有DFA状态
7. 假设
   1. 实验环境
8. Mac OS
9. Clion + MinGW (gcc)
   1. 输入及符号假设
10. 输入正则表达式仅包含连接、并、闭包运算
11. 仅包含a-z的符号
12. 假设e边用`&`记号表示
13. 有关自动机描述

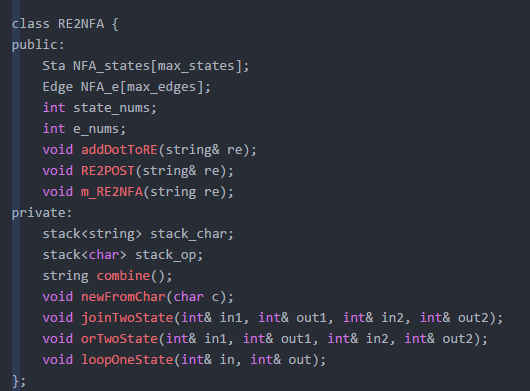
有限状态机由代码构造。根据输入的正则表达式逐步先构造NFA，再利用NFA构造DFA。

分别采用龙书上bottom-up方法和子集构造法。

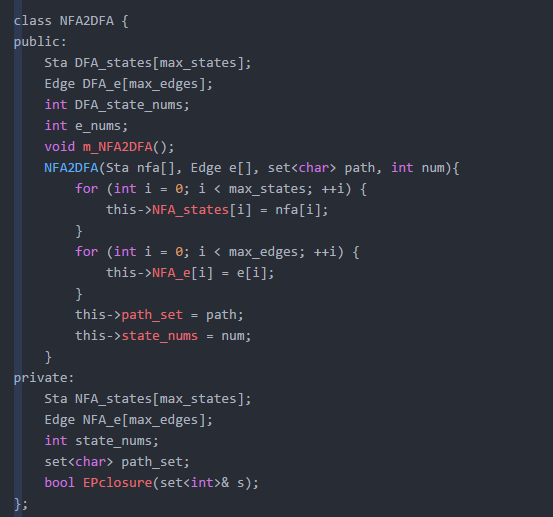
1. 数据结构描述
2. 状态点集以及边集，采用数组模拟链表实现



1. NFA类



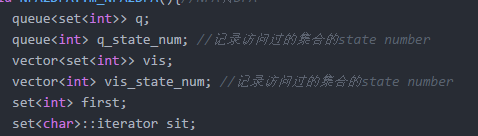
1. DFA类



1. <stack>，用于中缀转后缀的时候记录操作符和操作数



1. <queue>、<vector>、<set>，用于集合元素，以及BFS所需要的队列



1. 核心算法描述
   1. RE->NFA

1. 对输入的中缀正则表达式补全`.`（连接符号），用于进行后缀表达式的转换。只需要判断当前如果是字符，那么如果前一个符号为`)`,`\*`,`字符`中的一种就加入`.`

2. 进行中缀表达式转后缀表达式

2.1 通过符号栈和操作数栈来实现，由于3个操作符无优先级，且\*为单目操作符。

2.2 先判断是否为`\*`，如果是，则直接对操作数栈中栈顶元素+`\*`在压入栈中

2.3 如果是`)`，则一直进行出栈操作直到匹配到`)`

2.4 如果是`.`或`|`则在栈不空的情况下进行出栈操作

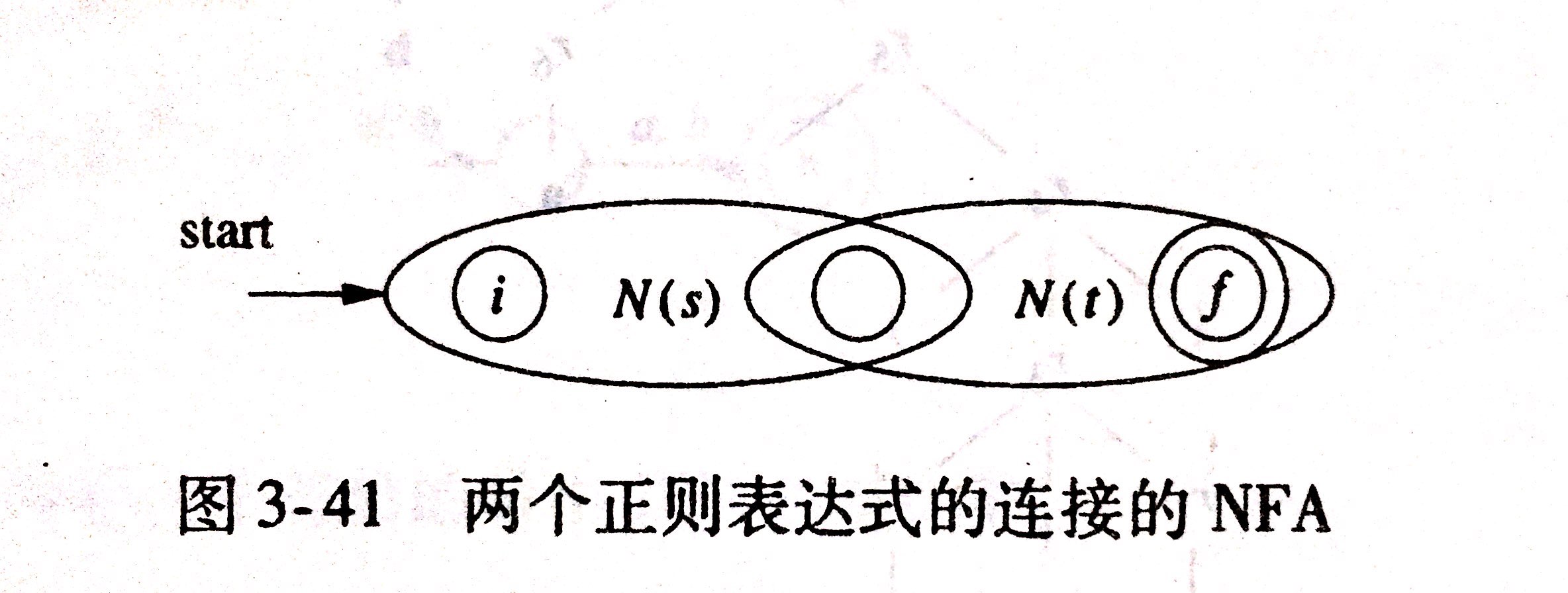
2.5 其他情况进栈

2.6 注意符号栈进行出栈操作实际上需要对操作数栈进行一个`combine`操作，再push回去

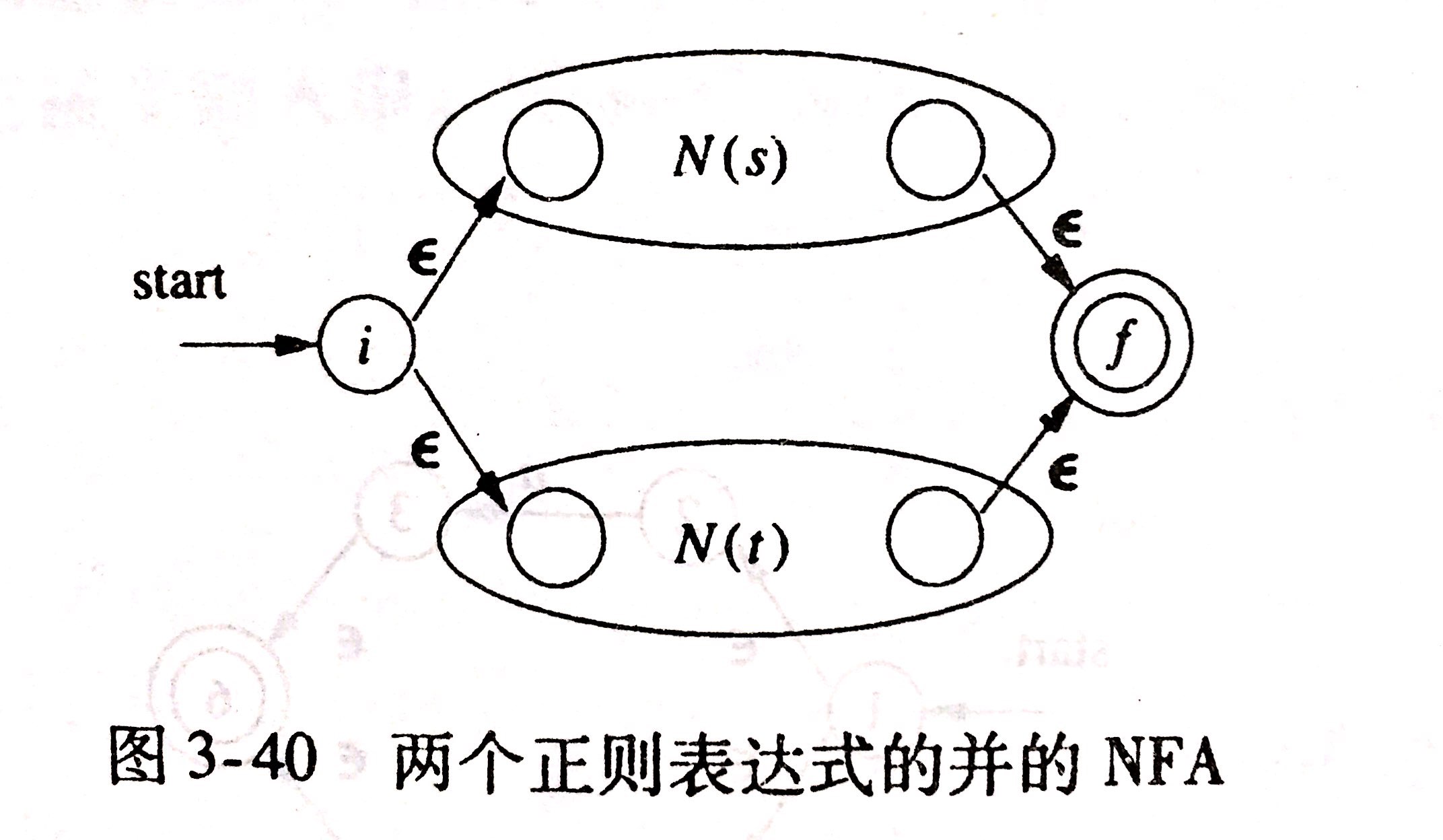
3. 后缀表达式转NFA

3.1 利用`bottom-up`的思想，构建分析树，再根据不同操作符创建状态，具体如下。

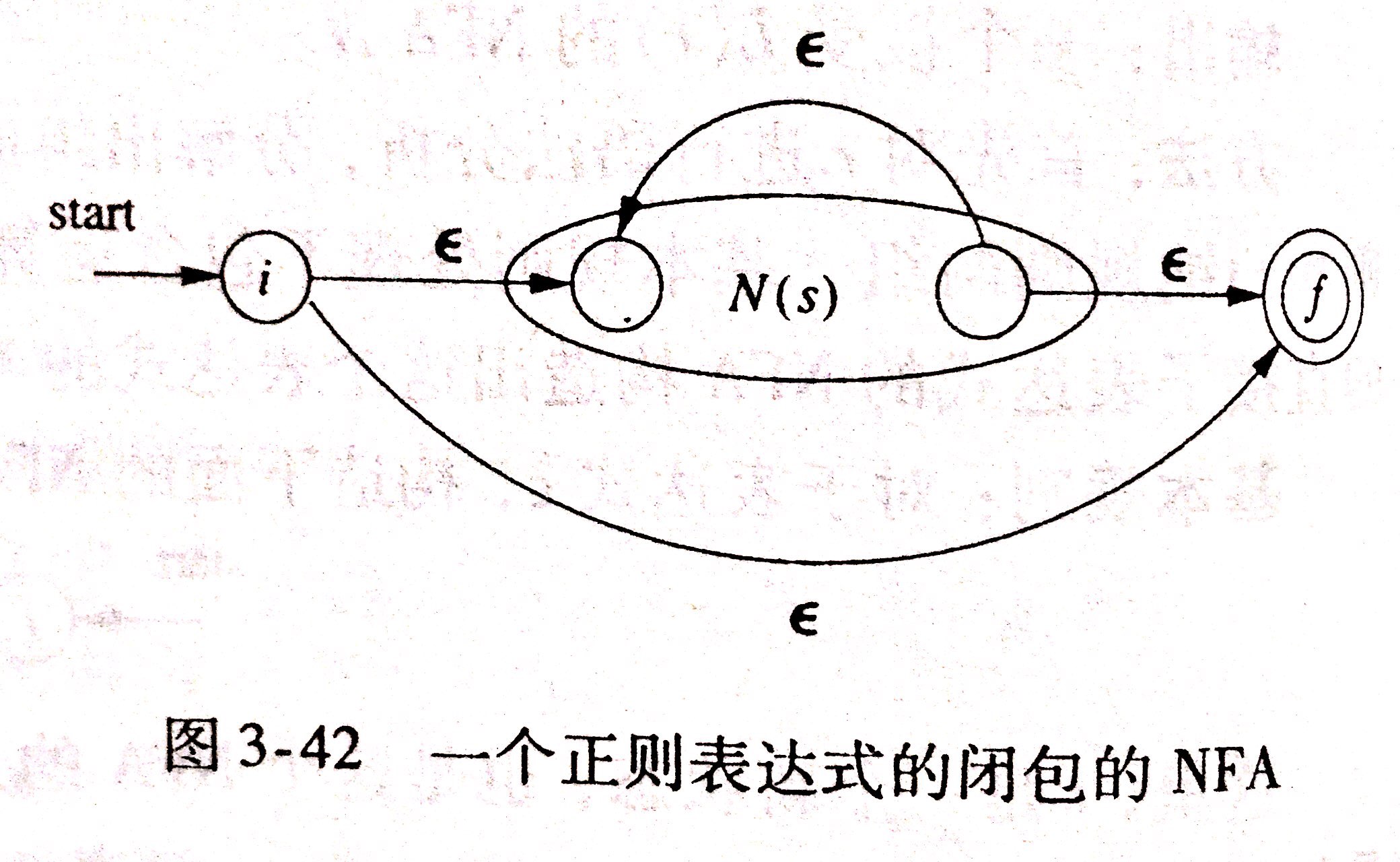
3.2 连接操作



* 1. 并操作



* 1. 闭包操作

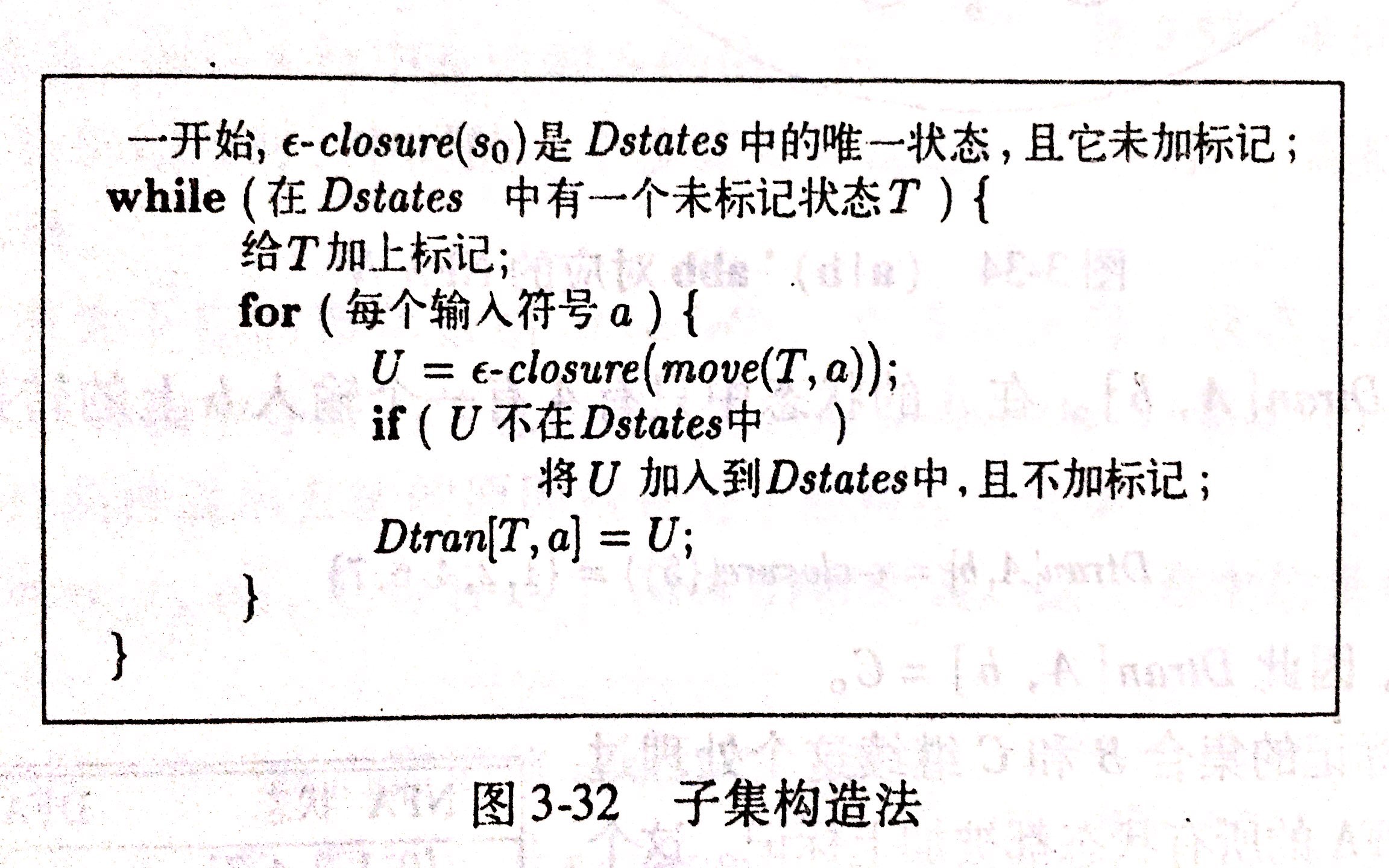


3.5 注意，这里直接把状态作为点，转换作为边进行有向图的构建，加入相应的e边进行构造。

* 1. NFA->DFA

利用子集构造法

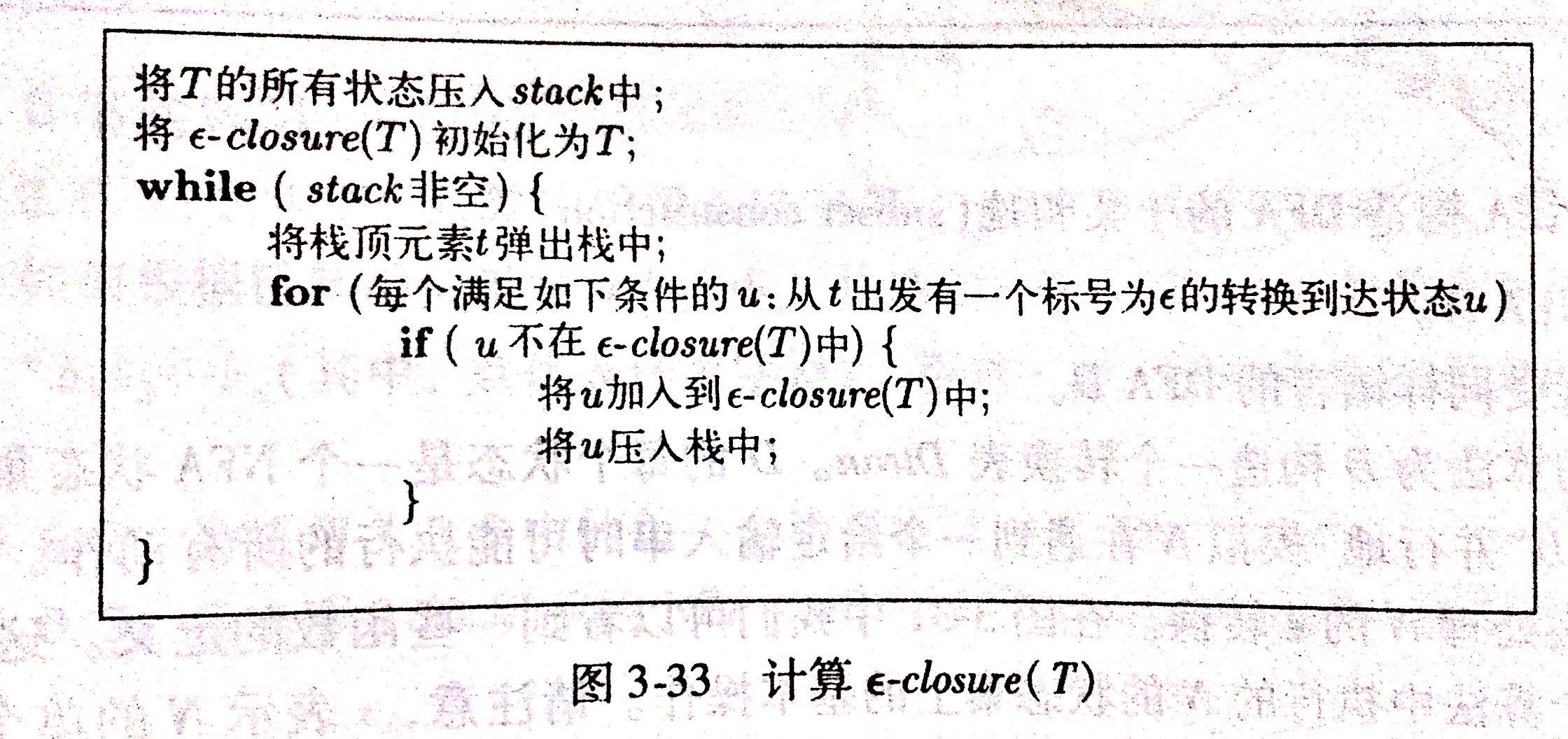
1. 首先状态为NFA中初始状态的e-closure，然后逐步求出各个状态，具体如书上伪代码，见下图：



2. 注意，这边采用STL<queue>作为数据结构，并采用广度优先遍历进行状态生成。

3. 并采用STL<set>作为记录状态集的数据结构，防止重复状态。

4. 其中计算e-closure的方法也如书上所示伪代码，见下图：

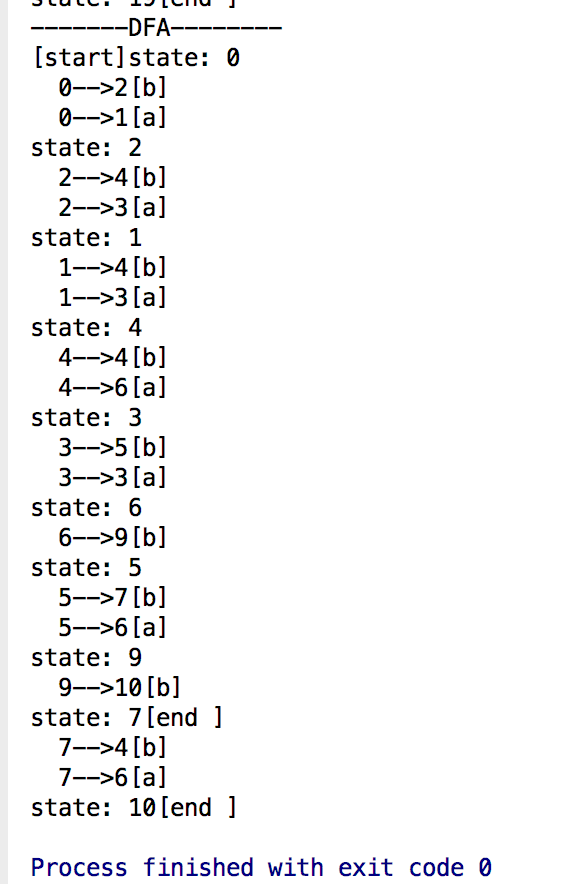
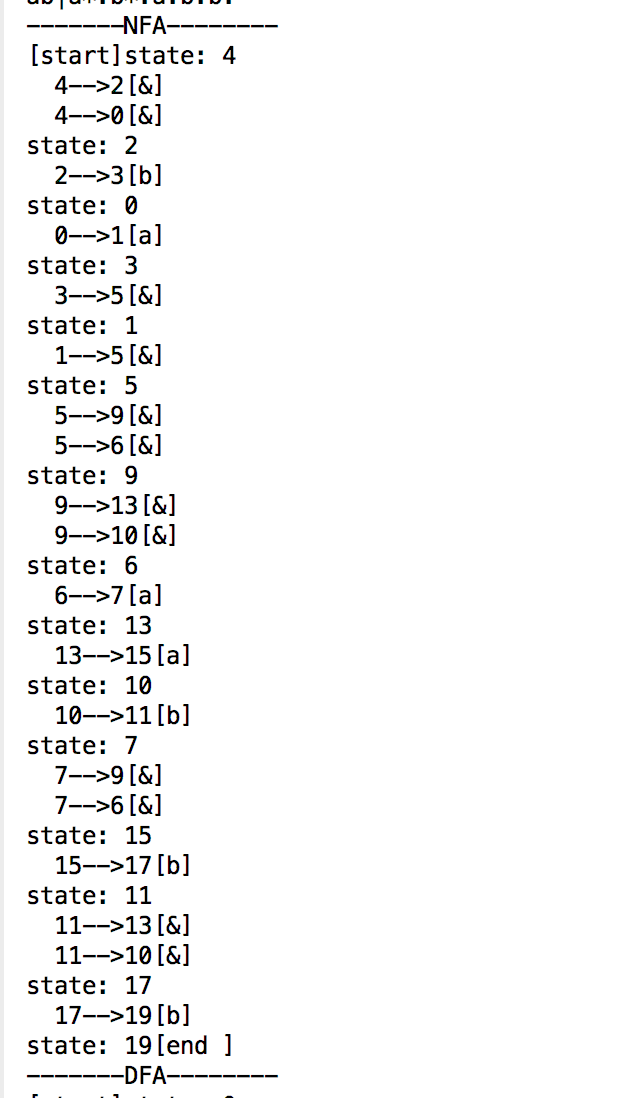
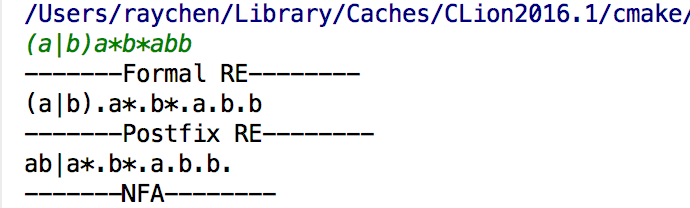


5. 这里同样采用STL<queue>，采用广度优先遍历进行一层一层e边的迭代。

1. 测试用例
   1. Test 1
2. 测试输入

(a|b)a\*b\*abb

1. 测试输出

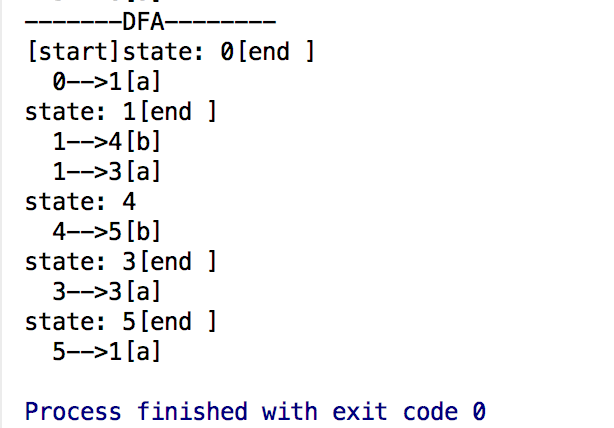
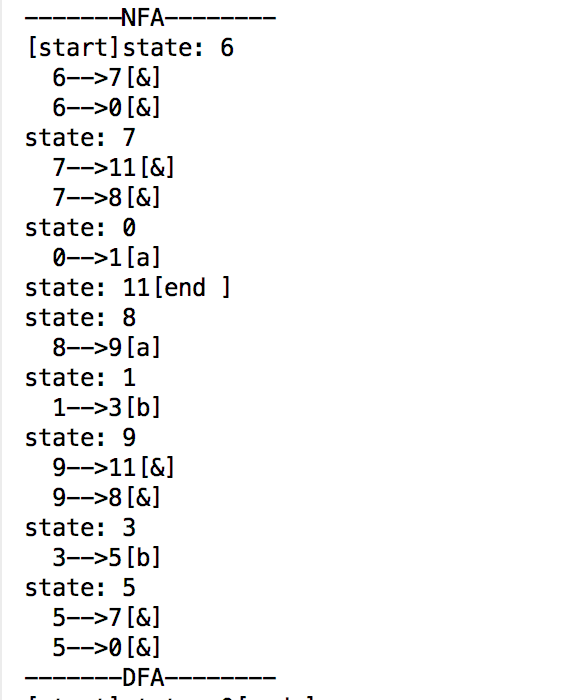
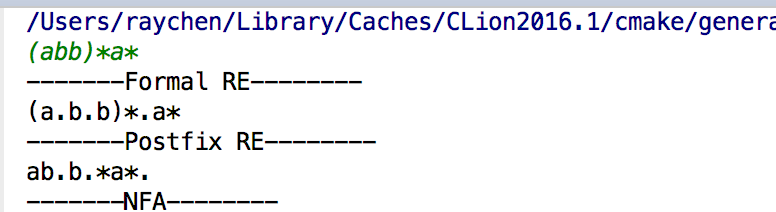


* 1. Test 2

1. 测试输入

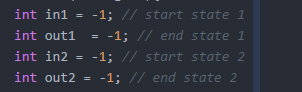
(abb)\*a\*

1. 测试输出



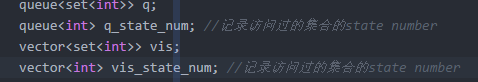
1. 问题与解决
2. 后缀表达式转NFA时，如何记录中间产生的各个状态。

解决：实际上，从语法分析树可以看到，最多只会有2个当前状态集，并且当前状态集实际上只有start状态和end状态是会被后续转换所用到的，于是只需要用4个变量表示即可。如我代码中：



1. NFA转DFA时，如何记录当前状态集所对应的状态号？因为没有建立对应状态集的数据结构，状态集实际上也是一个状态？

解决：采用多一个`STL<vector>`的方式，跟踪记录相应状态号。



1. 总结与感受

通过这次实验，我感觉复习了一些以前学过的算法，如BFS，中缀转后缀等。同时感受到了词法分析器生成器的有趣之处，希望有空可以实现一个自己的简单分析生成器。

没有了一开始看到lex生成出一个分析代码的模板出来那种震惊了，感觉自己也可以实现，开心~