实验 2 NFA 到 DFA

实验难度：★★★★☆
建议学时：3 学时

一、 实验目的

- 掌握 NFA 和 DFA 的概念。
- 掌握 ε -闭包的求法和子集的构造方法。
- 实现 NFA 到 DFA 的转换。

二、 预备知识

- 完成从正则表达式到 NFA 的转换过程是完成本实验的先决条件。虽然 DFA 和 NFA 都是典型的有向图，但是基于 NFA 自身的特点，在之前使用了类似二叉树的数据结构来存储 NFA，达到了简化的目的。但是，DFA 的结构相对复杂，所以在这个实验中使用了图的邻接链表来表示 DFA。如果读者对有向图的概念和邻接链表表示法有一些遗忘，可以复习一下数据结构中相关的章节。
- 对 DFA 的含义有初步的理解，了解 ε -闭包的求法和子集的构造方法。读者可以参考配套的《编译原理》教材，预习这一部分内容。

三、 实验内容

3.1 阅读实验源代码

NFAToDFA.h 文件

主要定义了与 NFA 和 DFA 相关的数据结构，其中有关 NFA 的数据结构在前一个实验中有详细说明，所以这里主要说明一下有关 DFA 的三个数据结构，这些数据结构定义了 DFA 的邻接链表，其中 DFAState 结构体用于定义有向图中的顶点（即 DFA 状态 ），Transform 结构体用于定义有向图中的弧（即转换）。具体内容可参见下面的表格。

DFA 的域	说明
DFAlist	DFA 状态集合。
length	集合中的 DFA 状态数量。与前一项构成一个线性表。

DFAState 的域	说明
NFAlist	NFA 状态集合。用于保存 DFA 状态中的 NFA 状态集合。
NFAStateCount	集合中的 NFA 状态数量。与前一项构成一个线性表。
firstTran	指向第一个转换。

Transform 的域	说明
TransformChar	状态之间的转换符号。
DFAStateIndex	DFA 状态在线性表中的下标。用于指示转换的目标 DFA 状态。
NFAlist	NFA 状态集合。用于保存构造的子集以及生成新的 DFA 状态。
NFAStateCount	集合中的 NFA 状态数量。与前一项构成一个线性表。
NextTrans	指向下一个转换。

main.c 文件

定义了 main 函数。在 main 函数中首先初始化了栈，然后调用了 re2post 函数，将正则表达式转换到解析树的后序序列,最后调用了 post2dfa 函数将解析树的后序序列转换到 DFA。

在 main 函数的后面，定义了一系列函数，有关函数的具体内容参见下面的表格。关于这些函数的参数和返回值，可以参见其注释。

函数名	功能说明
CreateDFATransform	创建一个新的 DFA 转换。每当求得一个 ε -闭包后可以调用此函数来创建一个 DFA 转换。
CreateDFAState	利用转换作为参数构造一个新的 DFA 状态，同时将 NFA 状态子集复制到新创建的 DFA 状态中。
NFAStateIsSubset	当一个子集构造完成后，需要调用此函数来判断是否需要为该子集创建一个新的 DFA 状态。如果构造的子集是某一个 DFA 状态中 NFA 状态集合的子集，就不需要新建 DFA 状态了。此函数的函数体还不完整，留给读者完成。
IsTransformExist	判断 DFA 状态的转换链表中是否已经存在一个字符的转换。每当求得一个 ε -闭包后可以调用此函数来决定是新建一个转换，还是将 ε -闭包合并到已有的转换中。此函数的函数体还不完整，留给读者完成。
AddNFAStateArrayToTransform	将一个 NFA 集合合并到一个 DFA 转换的 NFA 集合中,并确保重复的 NFA 状态只出现一次。当需要将 ε -闭包合并到已有的转换中的 NFA 集合中时，可以调用此函数。此函数的函数体还不完整，留给读者完成。
Closure	使用二叉树的先序遍历算法求一个 NFA 状态的 ε -闭包。注意，优先使用二叉树的先序遍历算法，否则会造成 ε -闭包中 NFA 状态集合顺序不同，进而导致无法通过自动化验证。此函数的函数体还不完整，留给读者完成。
post2dfa	将解析树的后序序列转换为 DFA。在这个函数中会调用函数post2nfa,所以在此函数中只需要专注于NFA转换到DFA的源代码的编写即可。此函数的函数体还不完整，留给读者完成。

RegexpToPost.c 文件

定义了 re2post 函数,此函数主要功能是将正则表达式转换成为解析树的后序序列形式。

PostToNFA.c 文件

定义了 post2nfa 函数，此函数主要功能是将解析树的后序序列形式转换成为 NFA。关于此函数的功能、参数和返回值，可以参见其注释。注意，此函数的函数体还不完整，读者可以直接使用之前实验中编写的代码。

NFAFragmentStack.c 文件

定义了与栈相关的操作函数。注意，这个栈是用来保存 NFA 片段的。

NFAStateStack.c 文件

定义了与栈相关的操作函数。注意，这个栈是用来保存 NFA 状态的。

RegexpToPost.h 文件

声明了相关的操作函数。为了使程序模块化，所以将 re2post 函数声明包含在一个头文件中再将此头文件包含到“main.c”中。

PostToNFA.h 文件

声明了相关的操作函数。为了使程序模块化，所以将 post2nfa 函数声明包含在一个头文件中再将此头文件包含到“main.c”中

NFAFragmentStack.h 文件

定义了与栈相关的数据结构并声明了相关的操作函数。

NFAStateStack.h 文件

定义了与栈相关的数据结构并声明了相关的操作函数。

3.2 正则表达式 a(a|1)*的状态图

正则表达式 a(a|1)* 对应的 NFA 状态图可以参见图 2-1，DFA 状态图可以参见图 2-2。

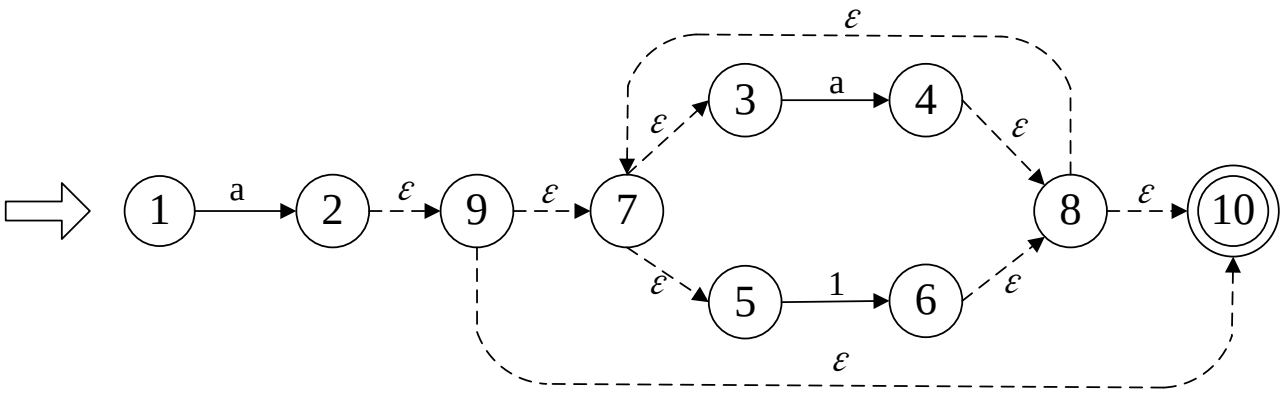

图 2-1：调用 post2nfa 函数后返回的 NFA

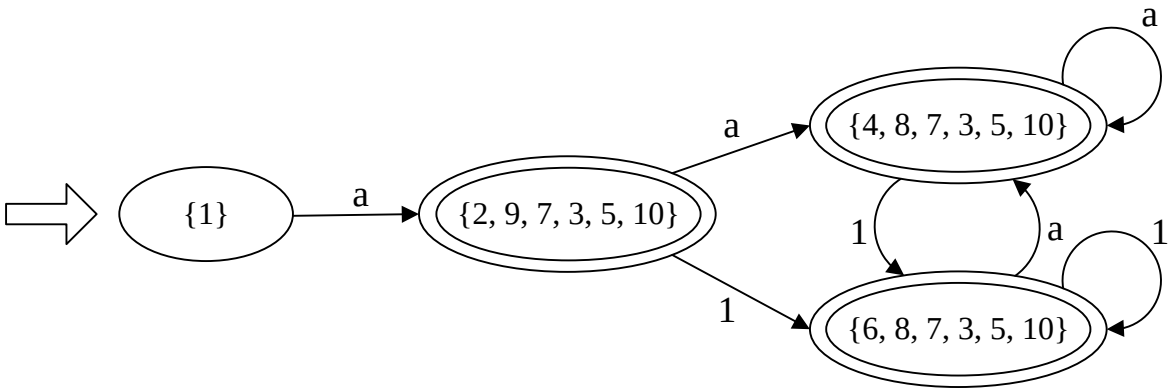

图 2-2：NFA 对应的 DFA

3.3 编写源代码并通过验证

按照下面的步骤继续实验：

1. 为 post2dfa 函数和其他未完成的函数编写源代码。注意尽量使用已定义的局部变量。
2. 按 Ctrl+Shift+B,在弹出的下拉列表中选择“生成项目”。如果生成失败，根据“TERMINAL”窗口中的提示信息修改源代码中的语法错误。
3. 按 Ctrl+Shift+B，然后在下拉列表中选择“测试”，启动验证。在“TERMINAL”窗口中会显示验证的结果。如果验证失败，可以在“EXPLORER”窗口中，右击“result_comparation.html”文件，在弹出的菜单中选择“Open Preview”，可以查看用于答案结果文件与读者编写程序产生的结果文件的不同之处，从而准确定位导致验证失败的原因。

注意：在实现 Closure 函数时，尽量使用二叉树的先序遍历算法，否则会造成 ε-闭包中 NFA 状态集合顺序不同，进而导致无法通过自动化验证。

四、思考与练习

1. 编写一个 FreeNFA 函数和一个 FreeDFA 函数，当在 main 函数的最后调用这两个函数时，可以将整个 NFA 和 DFA 的内存分别释放掉，从而避免内存泄露。
2. 读者可以尝试使用自己编写的代码将 input2.txt 和 input3.txt 中的正则表达式转换成 DFA，并确保能够通过自动化验证。在验证通过后，根据 DFA 的邻接链表数据绘制出 DFA 的状态转换图，并尝试编写一个 Minimize 函数，此函数可以将 DFA 中的状态数最小化。
3. 编写一个 Match 函数，此函数可以将一个字符串与正则表达式转换的 DFA 进行匹配，如果匹配成功返回 1，否则返回 0。