苏 州 大 学 实 验 报 告

院、系	计算机科学与技术学院	年级专业	21 计科	姓名	赵鹏	学号	2127405037
课程名称	编译原理实践					成绩	
指导教师	段湘煜	同组实验者	无		实验日期	2023.9.18	

实 验 名 称 词法分析深入

一. 实验题目

实现词法分析器中正则表达式部分的相关功能

输入正则表达式对应的表达式树吗，基于 MYT 算法利用表达式树构建正则表达式对应的 NFA，利用子集构造法将构造出的 NFA 转化为 DFA，再将 DFA 进行最简化。

二. 实验原理及流程框图

1.利用正则表达式树构建 NFA：

正则表达式树：正则表达式树是一种用于表示正则表达式结构的树形数据结构。每个节点表示操作符或字符，而边表示操作符之间的关系。

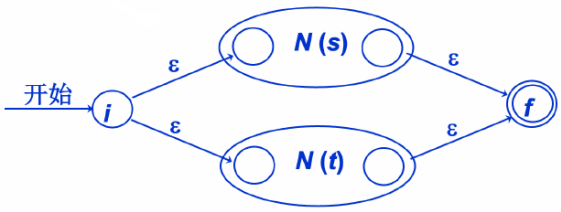
MYT 算法： MYT 算法（此处替换为您的算法名称）是一种特定的算法，用于将正则表达式树转换为非确定性有限自动机（NFA）。它基于一系列规则和策略，通过遍历正则表达式树的节点，逐步构建 NFA 的状态和状态转移。

NFA 的构建过程： 使用 MYT 算法，我们首先创建一个空的 NFA，并从正则表达式树的根节点开始按深度优先遍历。在遍历的过程中，我们根据不同的节点类型执行不同的操作，例如，对于字符节点，我们创建状态和状态转移；对于操作符节点，我们执行合适的操作，如闭包操作和连接操作。

MYT 算法的具体构建方法如下：

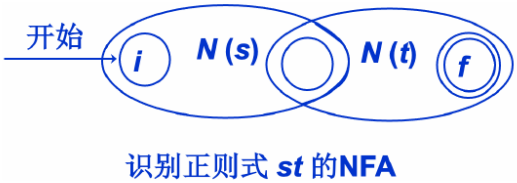
识别正则式ε的NFA

识别正则式a的NFA

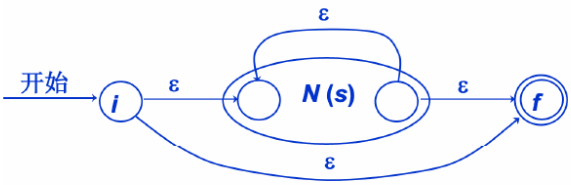


识别正则式s | t 的NFA

教务处制



识别正则式 *st* 的NFA



识别正则式 s^* 的NFA

2.利用子集构造法将 NFA 转为 DFA

NFA 和 DFA：非确定性有限自动机（NFA）和确定性有限自动机（DFA）是两种不同类型的有限自动机，它们用于匹配字符串模式。NFA 允许在某个状态下有多个可能的转移，而 DFA 每个状态只有一个确定的转移。

子集构造法：子集构造法是一种用于将 NFA 转换为 DFA 的经典方法。该方法的核心思想是将 NFA 的状态集合映射到 DFA 的状态集合，并确定 DFA 的状态转移函数。

NFA 到 DFA 的过程：使用子集构造法，我们从 NFA 的开始状态开始，计算其 ε -closure（epsilon 闭包），表示该状态及其可通过 ε 转移到达的状态。然后，我们根据输入符号计算新状态的转移，并重复此过程，直到不再有新的状态产生。通过这个过程，我们建立了 DFA 的状态集合和状态转移函数。

子集构造法的流程如下：

最小子集构造法：首先构造某个状态集合 *T* 的闭包函数

```
把 T 的所有状态压入栈;
ε-closure(T)的初值置为 T;
while 栈非空
    把栈顶元素 t 弹出栈;
    for 每个状态 u（条件是从 t 到 u 的边上的标记为ε）
        if u 不在ε-closure(T)中
            把 u 加入ε-closure(T);
            把 u 压入栈;
        end
    end
end
```

然后构造 DFA:

```
初始，ε-closure(s0)是 Dstates 仅有的状态，并且尚未标记;
while Dstates 有尚未标记的状态 T
    标记 T;
    for 每个输入符号 a
        U := ε-closure(move(T,a)) ;
        if U 不在 Dstates 中
            把 U 作为尚未标记的状态加入 Dstates;
        Dtran[T, a] := U;
    end
end
```

3.DFA 最简化

DFA 化简概述：在编译器前端的正则表达式处理中，最简 DFA 是一种关键的数据结构，它具有与原始 DFA 相同的识别能力，但具有更少的状态，从而提高了匹配性能。DFA 的化简是通过合并不可区分的状态来实现的，这些状态在输入符号下具有相同的状态转移行为。

DFA 最简化的流程如下：

构造状态集合的初始划分 π：两个子集——接受状态子集 F 和非接受状态子集 S－F
应用下面的过程构造 π_{new}
　　最初，令 $\pi_{new} = \pi$
　　For π 中的每个子集 G
　　　　把 G 划分为若干子集，使得两个状态 s 和 t 在同一子集中，当且仅当对任意输入符号 a，s 和 t 的 a 转换都到 π 的同一子集中
　　　　在 π_{new} 中，用 G 的划分代替 G
　　End
　　如果 $\pi_{new} = \pi$，则 $\pi_{final} = \pi$；否则令 $\pi = \pi_{new}$，转上步
　　在 π_{final} 的每个状态子集中选一个状态代表它，即为最简 DFA 的状态

三. 实验步骤

1.基于表达式树构建 NFA

首先定义 NFA 状态，每个状态存储一个状态编号，以及状态的边集，出边存储边上的字符和到达状态点的指针。

再定义 NFA 类，NFA 类主要存储开始节点和结束节点的指针。构造函数传入转换表和树根，转换表用$map < string, vector < string >>$存储，随后从树根开始按照深度优先遍历的顺序递归构建 NFA。

递归的基本思路如下：

1.如果是中间节点并且只有一个孩子，则调用$single_char()$方法利用孩子创建单字符的自动机。

2.如果是中间节点并且有三个孩子，则判断中间的孩子是那种操作。如果是|则调用 OR 方法，传入递归调用的左右儿子返回的 NFA 指针作为参数。如果左右儿子是括号则直接忽略，递归调用中间儿子。

3.如果只有两个儿子，如果两个儿子都是中间节点则进行 AND 操作，否则调用 CLOSURE 操作，构建左儿子的闭包并返回。

对于 MYT 算法的 single_char、AND、OR、CLOSURE 操作，只需按照实验原理中的相关图片创建节点，为节点添加相关的边即可。

最后实现 Preperation 方法，将 NFA 转换表存储为$map < int, map < char, int >>$，便于实现 NFA 转 DFA。

2.利用子集构造法将 NFA 转为 DFA

首先定义 DFAState 类和 DFA 类，DFAState 类构造函数传入编号和对应的 NFA 状态编号集合，并存储该状态的转换信息以及是否为接受状态，转换表可以使用 map<char,int>存储，同时定义相关成员函数用于设置这些变量和返回一些信息。DFA 类的构造函数传入用于转换为 DFA 的 NFA 的指

针。在构造函数中进行子集构造法的转换，利用 C++的 STL 中的相关模板，定义临时变量存储状态集、状态转换并逐步实现子集构造法。如 利用$set<int>$存储状态集，用$map<set<int>,int>$给状态集编号等。定义并实现子集构造法所需的相关辅助函数: getEpsilonClosure 用于求解空闭包，getUnsigned 用于判断状态集是否被标记，move 用于求解状态集按照任一字符移动后的状态集等。随后根据实验原理中的算法流程模拟实现子集构造法即可。在完成了这一部分后，将这些信息转化为 DFAState 并存储在 DFA 的 vector<DFAState>中。

3.将 DFA 进行最简化

DFA 的最简化是算法中对状态集进行划分的部分。定义 Partition 函数实现划分，用于辅助 DFA 最简化的实现。Partition 函数传入当前状态的划分结果作为参数，实现时可以用$set<set<int>>$。首先对每个状态集进行编号，并记录状态集的每个状态所归属的状态集的编号。随后遍历状态集的每个状态，遍历所有可能的转换字符，将每个状态途径所有字母所到达的节点编号存储为一个$vector$，依据$vector$对状态集进行划分。

若所有集合都无法划分，则最简化结束。

最后将最简化后的状态集重新编号，重新构造 DFAState 并存储在 DFA 的 vector<DFAState>中即可。

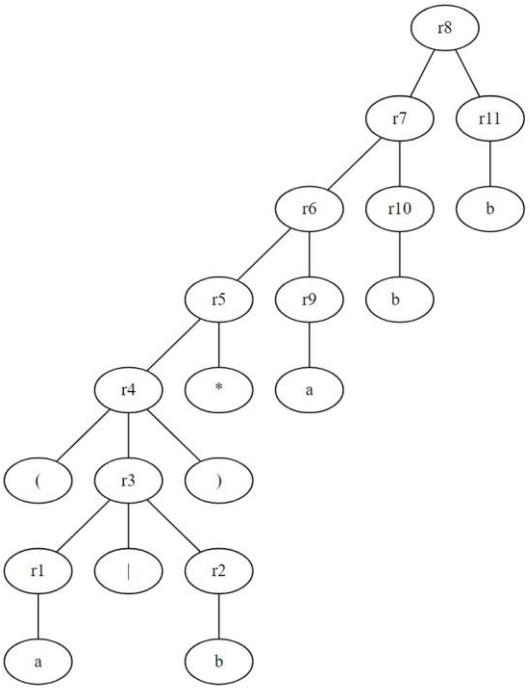
对于用自动机实现的正则表达式的匹配功能在 DFA 中用 match 函数实现，实现原理是从 DFA 最简化后的开始状态开始，一次读入输入字符串的字符，并进行当前状态下该字符对应的转换。若出现转换到-1 或在字符集中没有出现的情况则直接返回不接受，否则返回最终状态是否为接受状态即可。

四. 实验结果及分析

测试一：

正则表达式：(a|b)*abb

对应的表达式树为：



输入数据为：

19

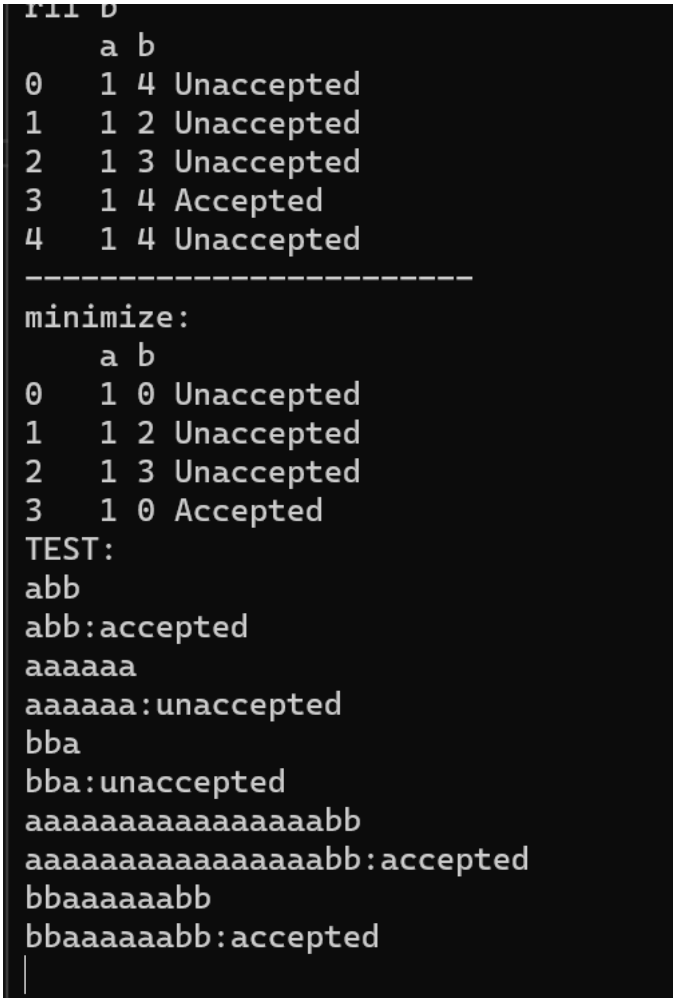
r8 r7

r8 r11

r7 r6

r7 r10
r6 r5
r6 r9
r5 r4
r5 *
r4 (
r4 r3
r4)
r3 r1
r3 |
r3 r2
r1 a
r2 b
r9 a
r10 b
r11 b

程序运行结果为:

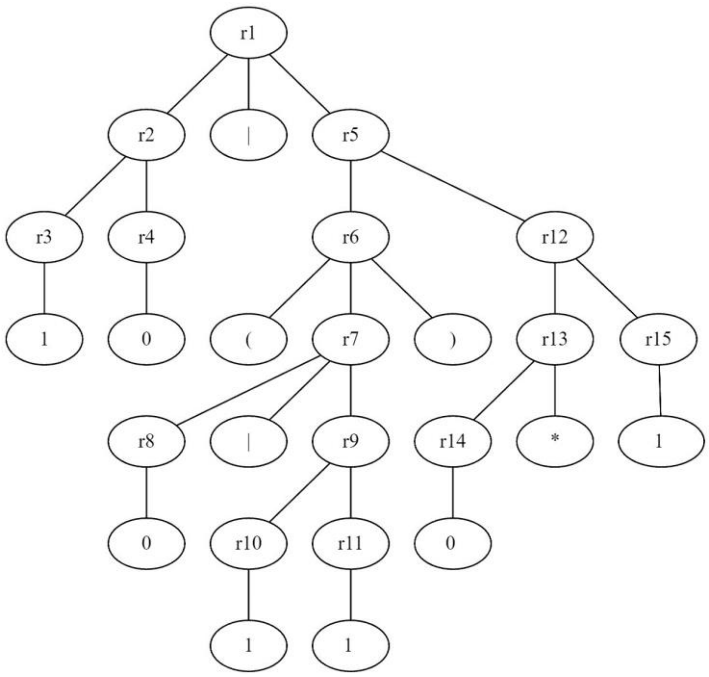


输入一定字符串进行测试后发现，对于正则表达式所表示的以 abb 结尾的串，dfa 都能够接受，对于没有以 abb 结尾或者出现了其他字母的字符串，都没有接受。

测试二：

正则表达式为：10|(0|11)0*1

对应的表达式树为：



输入数据为：

26
r1 r2
r1 |
r1 r5
r2 r3
r2 r4
r3 1
r4 0
r5 r6
r5 r12
r6 (
r6 r7
r6)
r7 r8
r7 |
r7 r9
r8 0
r9 r10
r9 r11
r10 1
r11 1
r12 r13

教务处制

r12 r15

r13 r14

r13 *

r14 0

r15 1

程序运行结果为:

```
    0 1
0   1 4 Unaccepted
1   3 2 Unaccepted
2   -1 -1 Accepted
3   3 2 Unaccepted
4   6 5 Unaccepted
5   3 2 Unaccepted
6   -1 -1 Accepted
------------------------
minimize:
    0 1
0   1 3 Unaccepted
1   1 2 Unaccepted
2   -1 -1 Accepted
3   2 1 Unaccepted
TEST:
10
10:accepted
001
001:accepted
1101
1101:accepted
110000001
110000001:accepted
01
01:accepted
000
000:unaccepted
100
100:unaccepted
0000001
0000001:accepted
11000001
11000001:accepted
```

其中-1 代表无对应转换

输入一定字符串测试后发现，对于这个正则所表示的 10 或者以 0 或 11 开头并以 1 结尾，中间填写任意数量 0 的字符串都能够接受，对于不符合这一要求的都不接受。

五. 实验总结

通过本次实验，我对编译原理中的使用 MYT 算法将正则转为 NFA、利用子集构造法将 NFA 转为 DFA 和 DFA 最简化相关算法有了更加深刻的理解，能够编写代码实现上述算法，对词法分析器的工作原理有了进一步的理解。

六. 代码

实验涉及到 DFA.hpp,NFA.hpp,utility.h,main.cpp 等文件，代码较长，已添加到附件。