南開大學
Nankai University

计算机学院

编译系统原理实验报告

OT1 实现词法分析器构造算法

姓名：孙蕗

学号：2112060

专业：信息安全

指导教师：王刚老师

2023 年 11 月 5 日

目录

1 摘要

本次实验中实现了词法分析器核心构造算法: 正则表达式->NFA 的 Thompson 构造法、NFA->DFA 的子集构造法、以及 DFA 的最小化算法

关键字：词法分析 NFA DFA Thompson 构造法 子集构造法 DFA 最小化算法

2 正则表达式->NFA 的 Thompson 构造法

2.1 正则表达式

正则表达式是一组由字母和符号组成的特殊文本，它可以用来从文本中找出满足想要的格式的句子。一个正则表达式是一种从左到右匹配主体字符串的模式，常使用缩写的术语“regex”或“regexp”。

2.1.1 正则表达式语法

元字符	描述
.	句号匹配任意单个字符除了换行符。
[ ]	字符种类。匹配方括号内的任意字符。
[^]	否定的字符种类。匹配除了方括号里的任意字符
*	匹配>=0个重复的在*号之前的字符。
+	匹配>=1个重复的+号前的字符。
?	标记?之前的字符为可选.
{n,m}	匹配num个大括号之前的字符或字符集 (n <= num <= m).
(xyz)	字符集，匹配与 xyz 完全相等的字符串.
|	或运算符，匹配符号前或后的字符.
\	转义字符,用于匹配一些保留的字符 `[ ] ( ) { } . * + ? ^ $ \ |`
^	从开始行开始匹配.
$	从末端开始匹配.

- *

  * 号匹配在 * 之前的字符出现大于等于 0 次。

  例如：表达式 [a-z]* 匹配一个行中所有以小写字母开头的字符串。

- +

  + 号匹配 + 号之前的字符出现 >=1 次。

  例如, 表达式 c+t 匹配以首字母 c 开头以 t 结尾，中间跟着至少一个字符的字符串。

- |

| 运算符就表示或，用作判断条件。

例如 (T|t)he|car 匹配 (T|t)he 或 car。

2.1.2 正则表达式定义规则

字母表 $\sum$ 上的正规式 r 的定义规则，以及 r 所表示的语言 L(r) 定义如下：
基本情况：
1. ε 是正规式，表示语言ε
2. 若 a $\in \sum$，则 a 是正规式，标识语言 {a}
3. 递归规则
r,s 为正规式，表示语言 L(r) 和 L(s)，则

- (r)|(s) 是正规式，表示语言 L(r)$\bigcup$L(s)
- (r)(s) 是正规式，表示语言 L(r)L(s)
- (r)* 是正规式，表示语言 (L(r))*
- (r) 是正规式，表示语言 L(r)

2.1.3 正规式运算的特性

公理	描述
r | s = s | r	| 满足**交换率**
r | (s | t) = (r | s) | t	| 满足**结合率**
(r s) t = r (s t)	连接满足**结合率**
r (s | t) = r s | r t (s | t) r = s r | t r	连接和 | 满足**分配率**
ε r = r r ε = r	ε是连接运算的**单位元**
r* = (r | ε)*	* 和ε间的关系
r** = r*	* 是**幂等**的

2.2 NFA

NFA(Non-Deeterministic Finite State Automata) 不确定的有穷自动机：对一个输入符号，有两种或两种以上可能对状态，所以是不确定的。

2.2.1 NFA 定义

数学模型，表示为五元组 M = (S, Σ, δ, s0, F)

- S: 有限状态集
- Σ: 有穷字母表，其中元素为输入符号
- δ: S × Σ 到 S 的子集的映射，即 S × (Σ $\bigcup$ { ε}) -> 2^s，状态转换函数

- s0 ∈ S 是唯一的初态
- F ⊆ S 是一个终态集（可空）

2.3 正则表达式 (描述单词)-> NFA(定义语言)

2.3.1 算法描述

语法制导方式——根据正规式的语法结构设计转换为 NFA 的方法

- 基本情况

  基本符号、ε、直接构造简单 NFA

- 递归规则

  子正规式通过各种运算：|、连接、闭包构造出复杂正规式。子 NFA 需要组合为复杂 NFA。

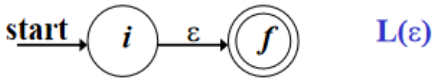
2.3.2 Thompson 构造法

输入字母表 Σ 上的一个正规式 r, 输出一个 NFA N，L(N)=L(r)。

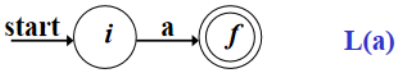
- 将 r 分解为子正规式
- 对其中每个基本符号（字母表中符号和 ε），按下面的给出的规则 (1)、(2) 构造 NFA。同一符号在不同位置需构造不同 NFA。
- 按照 r 的语法结构，对每个正规式运算，按规则 (3) 的方法，将运算对象———子正规式对应的子 NFA 组合成更大的 NFA，直至形成完整正规式 r 对应的最终的 NFA。

2.3.3 构造规则

1. 对于 ε，构造 NFA

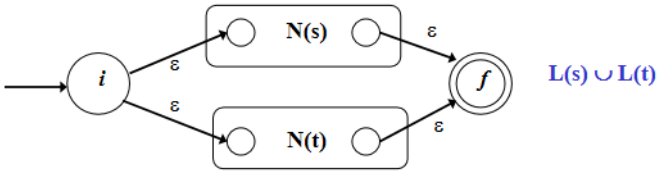


2. 对于 a ∈ Σ，构造 NFA



3. 假定 N(s), N(t) 是正规式 s, t 对应的 NFA

- 对正规式 s|t，构造如下组合 NFA N(s|t)

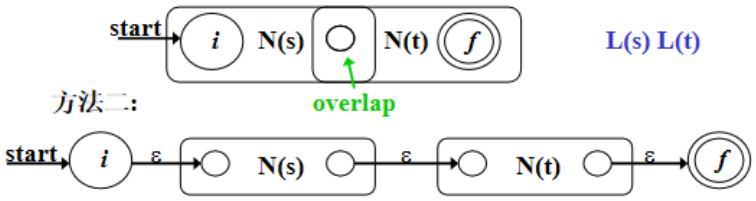


两个新状态：i——新初态 f——新终态

原初态和终态不再是组合 NFA 的初态和终态

四条新 ε 边：i->N(s) 初态 i->N(t) 初态 N(s) 终态->f N(s) 终态->f

- 对正规式 st，构造如下组合 NFA N(st)



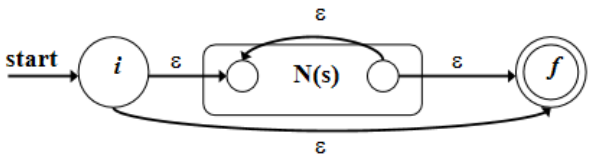
方法一：新初态——N(s) 初态，新终态——N(t) 终态

N(s) 终态与 N(t) 初态合并

方法二：新初态、终态——i、f

三条新 ε 边：i->N(s) 初态 N(t) 终态->f N(s) 终态->N(t) 初态

- 对正规式 s*，构造如下组合 NFA N(s*)



新初态、终态——i、f

四条新 ε 边：i->f i->N(s) 初态 N(s) 终态->f N(s) 终态->N(s) 初态

- 对正规式 (s)，N((s)) = N(s)

2.3.4 算法特性

- 每个步骤最多增加两个新状态->N(r) 状态数 <=2*(r 的符号数 + 操作符数)
- N(r) 有且只有一个初态和一个终态
- N(r) 的每个状态，或者有一条标记为某个 a∈ 的输出边，或者至多有两条 ε 输出边
- 为状态取名要小心

2.3.5 C++ 代码实现及相关注释

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <stack>
#include <unordered_set>
```

```cpp
#include <queue>
#include <deque>
#include <algorithm>
using namespace std;

// 定义 State 类,表示 NFA/DFA 中的状态
class State
{
public:
    bool isEnd;// 表示该状态是否是终止状态
    map<char, State*> transition;// 存储状态的字符转移关系
    vector<State*> epsilonTransitions;// 存储状态的 epsilon 转移关系

    State(bool isEnd) : isEnd(isEnd) {}// 构造函数, 用于初始化状态
};

// 定义 NFA 类, 表示非确定有限自动机
class NFA
{
public:
    State* start;// NFA 的起始状态
    State* end;// NFA 的终止状态
    // 构造函数, 用于初始化 NFA
    NFA(State* start, State* end) : start(start), end(end) {}
};

// 定义 DFA 类, 表示确定有限自动机
class DFA
{
public:
    set<State*> startStates;// DFA 的起始状态集合
    set<State*> endStates;// DFA 的终止状态集合
    map<set<State*>, map<char, set<State*>>> transitions;// 存储 DFA 的状态转移关系
    // 构造函数, 用于初始化 DFA
    DFA(set<State*> startStates, set<State*> endStates)
        : startStates(startStates), endStates(endStates) {}
};

// 添加 epsilon 转移关系
void addEpsilonTransition(State* from, State* to)
{
    from->epsilonTransitions.push_back(to);
}

// 添加字符转移关系
void addTransition(State* from, State* to, char symbol)
{
    from->transition[symbol] = to;
}

```

```
58 // 从字符创建 NFA
59 NFA* fromSymbol(char symbol)
60 {
61     State* start = new State(false); //创建了一个新的起始状态
           start, 并将其标记为非终止状态
62     State* end = new State(true); //创建了一个新的终止状态 end, 并将其标记为终止状态
63     //调用 addTransition 函数, 将起始状态 start 和终止状态 end 之间添加一个字符转移
64     //其中 symbol 是传递给 fromSymbol 函数的字符参数
65     addTransition(start, end, symbol);
66     //函数返回一个指向新创建的NFA对象的指针
67     //该NFA仅包含一个字符转移, 该字符转移从起始状态到终止状态。
68     return new NFA(start, end);
69 }
70
71 // 创建连接操作的 NFA
72 NFA* concat(NFA* first, NFA* second)
73 {
74     //调用 addEpsilonTransition 函数, 在第一个NFA的终止状态 first->end
           和第二个NFA的起始状态 second->start 之间添加一个 epsilon 转移。
75     addEpsilonTransition(first->end, second->start);
76     //将第一个NFA的原终止状态 first->end
           标记为非终止状态, 因为连接操作后, 原来的终止状态不再是终止状态。
77     first->end->isEnd = false;
78     //创建一个新的NFA对象, 该对象的起始状态是第一个NFA的起始状态
           first->start, 终止状态是第二个NFA的终止状态
           second->end。这个新的NFA表示了连接操作的结果。
79     return new NFA(first->start, second->end);
80 }
81
82 // 创建选择操作的 NFA
83 NFA* unionNFA(NFA* first, NFA* second)
84 {
85     State* start = new State(false);//创建一个新的起始状态
           start, 并将其标记为非终止状态
86     //调用 addEpsilonTransition 函数, 分别将新的起始状态 start
           与两个输入NFA的起始状态 first->start 和 second->start 之间添加两个 epsilon
           转移。
87     //从新的起始状态可以通过 epsilon 转移到两个输入NFA的起始状态。
88     addEpsilonTransition(start, first->start);//
89     addEpsilonTransition(start, second->start);
90     //创建一个新的终止状态 end, 并将其标记为终止状态
91     State* end = new State(true);
92     //将第一个输入NFA的终止状态 first->end 和第二个输入NFA的终止状态 second->end
           与新的终止状态 end 之间添加两个 epsilon 转移。
93     addEpsilonTransition(first->end, end);
94     first->end->isEnd = false;
95     addEpsilonTransition(second->end, end);
96     second->end->isEnd = false;//将两个输入NFA的原终止状态标记为非终止状态
97     return new NFA(start, end);
98 }
```

```
99
100 // 创建闭包操作的 NFA
101 NFA* closure(NFA* nfa)
102 {
103     State* start = new State(false);//创建一个新的起始状态
        start, 并将其标记为非终止状态
104     State* end = new State(true);//创建了一个新的终止状态 end, 并将其标记为终止状态
105
106     addEpsilonTransition(start, end);//调用 addEpsilonTransition 函数, 将新的起始状态
        start 与新的终止状态 end 之间添加一个 epsilon
        转移。这是闭包操作的起始和终止状态之间的 epsilon 转移
107     addEpsilonTransition(start, nfa->start);//将新的起始状态 start
        与输入NFA的起始状态 nfa->start 之间添加一个 epsilon
        转移。这允许从新的起始状态通过 epsilon 转移到输入NFA的起始状态。
108     addEpsilonTransition(nfa->end, end);//将输入NFA的终止状态 nfa->end 与新的终止状态
        end 之间添加一个 epsilon 转移。这允许从输入NFA的终止状态通过 epsilon
        转移到新的终止状态。
109     addEpsilonTransition(nfa->end, nfa->start);//将输入NFA的终止状态 nfa->end
        与输入NFA的起始状态 nfa->start 之间添加一个 epsilon 转移。
110     nfa->end->isEnd =
        false;//将输入NFA的原终止状态标记为非终止状态, 因为在闭包操作后, 它不再是终止状态
111
112     return new NFA(start, end);
113 }
114
115 // 将中缀表达式转换为后缀表达式
116 string shunt(string infix)
117 {
118     map<char, int> specials = { {'*', 50}, {'.', 40}, {'|', 30} };
119     string postfix = "";//转换后的后缀表达式
120     stack<char> stack;//处理操作符和括号
121
122     for (char c : infix) //遍历中缀表达式中的每个字符
123     {
124         if (c == '(')//如果字符是左括号, 将其压入栈
125         {
126             stack.push(c);
127         }
128         //如果字符是右括号, 弹出栈中的操作符并添加到后缀表达式中, 直到遇到左括号
129         //然后移除左括号
130         else if (c == ')')
131         {
132             while (!stack.empty() && stack.top() != '(')
133             {
134                 postfix += stack.top();
135                 stack.pop();
136             }
137             // 移除 '('
138             stack.pop();
139         }
```

```
140         //如果字符是操作符，比较其优先级与栈顶操作符的优先级
141         //如果栈顶操作符的优先级高于或等于当前操作符，弹出栈顶操作符并添加
142         //到后缀表达式中，直到栈为空或栈顶操作符的优先级低于当前操作符。
143         //将当前操作符压入栈。
144         else if (specials.find(c) != specials.end())
145         {
146             while (!stack.empty() && specials[c] <= specials[stack.top()])
147             {
148                 postfix += stack.top();
149                 stack.pop();
150             }
151             stack.push(c);
152         }
153         //如果字符是字母或其他字符，直接将其添加到后缀表达式中
154         else
155         {
156             postfix += c;
157         }
158     }
159     //处理完所有字符后，将栈中剩余的操作符依次弹出并添加到后缀表达式中
160     while (!stack.empty())
161     {
162         postfix += stack.top();
163         stack.pop();
164     }
165 
166     return postfix;
167 }
168 
169 // 将后缀表达式转换为 NFA
170 NFA* toNFA(string postfix)
171 {
172     stack<NFA*> stack;
173     //遍历后缀表达式中的每个字符
174     for (char c : postfix)
175     {
176         //如果字符是点号（连接操作符）
177         //从栈中弹出两个NFA对象，然后将它们连接起来，并将新的NFA对象推回栈中。
178         if (c == '.')
179         {
180             NFA* nfa2 = stack.top();
181             stack.pop();
182             NFA* nfa1 = stack.top();
183             stack.pop();
184             NFA* newNFA = concat(nfa1, nfa2);
185             stack.push(newNFA);
186         }
187         //如果字符是竖线（选择操作符）
188         //从栈中弹出两个NFA对象，然后将它们进行选择操作，并将新的NFA对象推回栈中。
189         else if (c == '|')
```

```
190         {
191             NFA* nfa2 = stack.top();
192             stack.pop();
193             NFA* nfa1 = stack.top();
194             stack.pop();
195             NFA* newNFA = unionNFA(nfa1, nfa2);
196             stack.push(newNFA);
197         }
198         //如果字符是星号（闭包操作符）
199         //从栈中弹出一个NFA对象，然后将它应用闭包操作，并将新的NFA对象推回栈中。
200         else if (c == '*')
201         {
202             NFA* nfa = stack.top();
203             stack.pop();
204             NFA* newNFA = closure(nfa);
205             stack.push(newNFA);
206         }
207         //如果字符是字母或其他字符，将它转换为一个NFA对象，并将其推入栈中。
208         else
209         {
210             NFA* nfa = fromSymbol(c);
211             stack.push(nfa);
212         }
213     }
214 
215     return stack.top();
216 }
217 
218 // 遍历状态和转移关系，并输出
219 void printNFA(NFA* nfa)
220 {
221     cout << "Start State: " << nfa->start << endl;
222     cout << "End State: " << nfa->end << endl;
223 
224     // 遍历状态
225     stack<State*> stateStack;
226     stateStack.push(nfa->start);
227 
228     while (!stateStack.empty())
229     {
230         State* currentState = stateStack.top();
231         stateStack.pop();
232 
233         cout << "State: " << currentState << (currentState->isEnd ? " (End)" : "") <<
                endl;
234 
235         // 遍历字符转移
236         for (auto& transition : currentState->transition)
237         {
238             cout << "Transition on '" << transition.first << "' to State: " <<
```

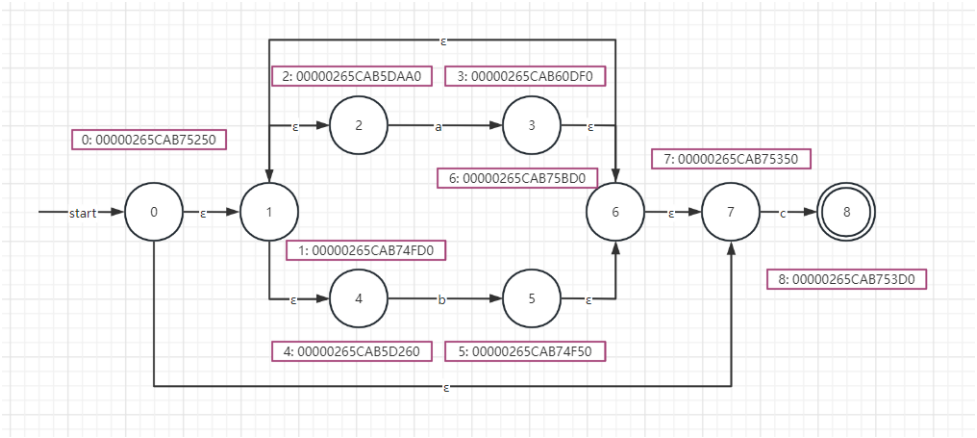
```
                transition.second << endl;
239         }
240
241         // 遍历 epsilon 转移
242         for (State* epsilonTransition : currentState->epsilonTransitions)
243         {
244             cout << "Epsilon Transition to State: " << epsilonTransition << endl;
245             stateStack.push(epsilonTransition);
246         }
247     }
248 }
```

2.3.6 运行截图及分析

输入的测试正则表达式字符串 regex = ”(a|b)*c”
运行截图:

```
Regular Expression -> NFA
Start State: 00000265CAB75250
End State: 00000265CAB753D0
State: 00000265CAB75250
Epsilon Transition to State: 00000265CAB74FD0
Epsilon Transition to State: 00000265CAB75350
State: 00000265CAB74FD0
Epsilon Transtion to State: 00000265CAB5DAA0
Epsilon Transtion to State: 00000265CAB5D260
State: 00000265CAB5DAA0
Transition on 'a' to State: 00000265CAB60DF0
State: 00000265CAB5D260
Transition on 'b' to State: 00000265CAB74F50
State: 00000265CAB60DF0
Epsilon Transtion to State: 00000265CAB75BD0
State: 00000265CAB74F50
Epsilon Transtion to State: 00000265CAB75BD0
State: 00000265CAB75BD0
Epsilon Transtion to State: 00000265CAB74FD0
Epsilon Transtion to State: 00000265CAB75350
State: 00000265CAB75350
Transition on 'c' to State: 00000265CAB753D0
```

画图对运行结果分析如下，已经将地址对应的状态标在图中:



3 NFA->DFA 的子集构造法

3.1 DFA

DFA (Deterministic Finite State) 确定的有穷自动机: 一个状态对一个输入符号，至多一个动作

3.1.1 DFA 定义

数学模型，表示为五元组 M = (S, Σ, δ, s0, F)

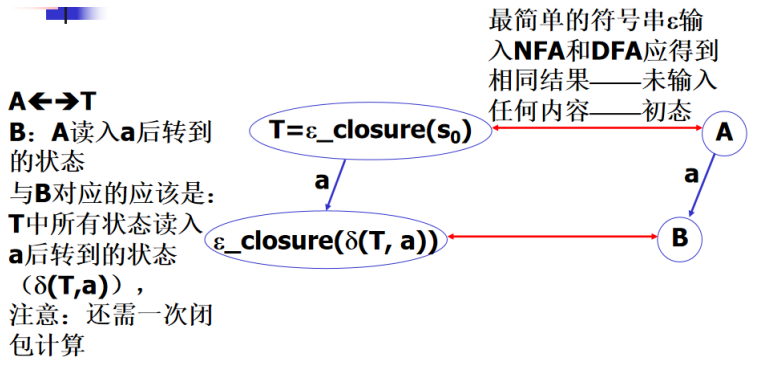
- S: 有限状态集
- Σ: 有穷字母表
- δ: S × Σ 到 S 的单值映射，即 δ: S × Σ ->S
- s0 ∈ S 是唯一的初态
- F ⊆ S 是一个终态集（可空）

3.2 NFA->DFA

NFA 存在缺点，同一符号/ ε 和其他符号存在多义性，不精确。

3.2.1 算法的基本思想

使用递推方法
最简单的符号串 ε：NFA 状态集合 <–>DFA 状态
长度为 1 的串 a=εa，在自动机中可达的状态为：从 ε 对应的状态经过标记为 a 的边可达的状态
长度为 2 的串…



3.2.2 NFA->DFA 的子集构造法

对于输入字符集合 ∑={a1,a2…,ak}，构造一张 k+1 列的表格（行数未做限制）。

（1）表格的第一行第一列的位置写的是从 NFA 的起始节点经过任意个 ε 所能到达的结点集合 S0 的 ε-closure(S0)。

（2）接着填写该行剩余位置的信息，做法是在对应的位置上填写 la= ε-closure(move(l,a))。la 表示从该集合开始经过一个 a 所能到达的集合，经过一个 a 的意思是可以略过前后的 ε。

（3）检查该行上的所有状态子集，如果未在第一列出现，则将该状态子集写到第一列。

输入：一个NFA N

输出：一个DFA D，L(D)=L(N)

算法中用到下列操作，其中s表示NFA的一个状态，T表示NFA的一个状态集

操作	描述
ε-closure(s)（ε闭包）	s以及从s出发仅通过ε边可到达的所有状态的集合
ε-closure(T)	∪ε-closure(s), s∈T
δ(T, a)	∪δ(s, a), s∈T

（4）重复（2）（3）的步骤，直到所有状态子集均在第一列上出现即可。

（5）给状态子集重新编号，包含原来终态的状态子集为新的终态，按照对应的转换函数 f，构造对应的 DFA。

3.2.3 C++ 代码实现及相关注释

```
// 计算状态集合的 epsilon 闭包
set<State*> epsilonClosure(set<State*> states)
{
    //将输入的状态集合 states 复制到一个新的状态集合 closure 中，作为起始闭包
    set<State*> closure = states;
    //创建一个栈 stateStack，用于存储需要处理的状态
    stack<State*> stateStack;
    //将输入状态集合中的每个状态压入栈中，以便开始处理
    for (State* state : states)
    {
        stateStack.push(state);
    }

    while (!stateStack.empty())
    {
        //从栈中弹出一个状态，作为当前处理的状态
        State* currentState = stateStack.top();
        //从栈中移除已处理的状态
        stateStack.pop();
        //遍历当前状态的 epsilon 转移关系
        for (State* epsilonTransition : currentState->epsilonTransitions)
        {
            //如果找到一个 epsilon 转移的状态 epsilonTransition，并且它不在闭包
                closure 中，
            //就将它添加到闭包中，并将它压入栈中，以便后续处理它的 epsilon 转移。
            if (closure.find(epsilonTransition) == closure.end())
            {
                closure.insert(epsilonTransition);
                stateStack.push(epsilonTransition);
            }
```

```
30          }
31      }
32      //返回包含原状态集合及其 epsilon 闭包的新状态集合 closure
33      return closure;
34  }
35
36  // 构建DFA
37  DFA* subsetConstruction(NFA* nfa)
38  {
39      //创建一个集合 dfaStates, 用于存储DFA的状态集合
40      set<set<State*>> dfaStates;
41      //创建一个映射 dfaTransitions, 用于存储DFA的状态转移关系
42      map<set<State*>, map<char, set<State*>>> dfaTransitions;
43      //创建一个初始状态集合 initialState, 并将NFA的起始状态 nfa->start 添加到其中
44      set<State*> initialState;
45      initialState.insert(nfa->start);
46      //计算 initialState 的 epsilon 闭包, 确保包括所有通过 epsilon 转移可达的状态
47      initialState = epsilonClosure(initialState);
48      //将初始状态集合添加到DFA的状态集合中
49      dfaStates.insert(initialState);
50      //创建一个队列 stateQueue, 用于广度优先搜索DFA状态
51      queue<set<State*>> stateQueue;
52      //将初始状态集合添加到队列中, 以便开始构建DFA
53      stateQueue.push(initialState);
54
55      //通过广度优先搜索, 逐个处理每个DFA状态
56      while (!stateQueue.empty())
57      {
58          set<State*> currentState = stateQueue.front();
59          stateQueue.pop();
60          //对于每个字符（在 'a' 到 'z' 之间）,
61          //计算从当前状态经过字符转移后达到的状态集合, 并计算它们的 epsilon 闭包。
62          for (char symbol = 'a'; symbol <= 'z'; symbol++)
63          {
64              set<State*> newState;
65              // 遍历当前状态集合, 查找通过字符 symbol 转移可达的状态
66              for (State* state : currentState)
67              {
68                  if (state->transition.find(symbol) != state->transition.end())
69                  {
70                      State* nextState = state->transition[symbol];
71                      newState.insert(nextState);
72                  }
73              }
74              // 计算新状态集合的 epsilon 闭包
75              set<State*> newStateClosure = epsilonClosure(newState);
76
77              //如果得到的状态集合不为空, 将其添加到DFA的状态集合中, 并记录状态转移关系。
78              if (!newStateClosure.empty())
79              {
```

```
80                  // 检查新状态是否已存在于DFA状态集合中，如果不存在，将其添加到状态队列
81                  if (dfaStates.find(newStateClosure) == dfaStates.end())
82                  {
83                      dfaStates.insert(newStateClosure);
84                      stateQueue.push(newStateClosure);
85                  }
86                  // 记录状态 currentState 经过字符 symbol 转移到 newStateClosure
87                  dfaTransitions[currentState][symbol] = newStateClosure;
88              }
89          }
90      }
91      // 创建 DFA 的起始状态集合
92      set<State*> dfaStartStates;
93      dfaStartStates.insert(nfa->start);
94      dfaStartStates = epsilonClosure(dfaStartStates);
95
96      // 创建 DFA 的终止状态集合
97      set<State*> dfaEndStates;
98
99      // 遍历 DFA 的状态集合，寻找终止状态
100     for (set<State*> stateSet : dfaStates)
101     {
102         for (State* state : stateSet)
103         {
104             if (state->isEnd)
105             {
106                 // 如果状态是终止状态，将其添加到 DFA 的终止状态集合中
107                 dfaEndStates.insert(stateSet.begin(), stateSet.end());
108                 break;
109             }
110         }
111     }
112
113     // 创建 DFA 对象，并设置其起始状态、终止状态和状态转移关系
114     DFA* dfa = new DFA(dfaStartStates, dfaEndStates);
115     dfa->transitions = dfaTransitions;
116
117     return dfa;
118 }
119
120 void printDFA(DFA* dfa)
121 {
122     cout << "DFA States:" << endl;
123     // 遍历 DFA 的状态转移映射，其中 stateTransitionPair 是一个 DFA
        状态集合到状态转移映射的映射
124     for (const auto& stateTransitionPair : dfa->transitions)
125     {
126         // 当前状态集合
127         const set<State*>& stateSet = stateTransitionPair.first;
128         cout << "State { ";
```

```
129
130         // 打印当前状态集合中的每个状态
131         for (State* state : stateSet)
132         {
133             cout << state << " ";
134         }
135         cout << "}" << endl;
136     }
137
138     cout << "DFA Transitions:" << endl;
139
140     // 再次遍历 DFA 的状态转移映射
141     for (const auto& transition : dfa->transitions)
142     {
143         // 起始状态集合
144         const set<State*>& fromStates = transition.first;
145         // 符号到目标状态集合的映射
146         const map<char, set<State*>>& toStates = transition.second;
147
148         // 遍历字母表中的字符 'a' 到 'z', 检查每个字符的状态转移
149         for (char symbol = 'a'; symbol <= 'z'; symbol++)
150         {
151             // 检查当前字符是否有转移
152             if (toStates.find(symbol) != toStates.end())
153             {
154                 // 目标状态集合
155                 const set<State*>& nextStateSet = toStates.at(symbol);
156
157                 cout << "Transition from { ";
158
159                 // 打印起始状态集合中的每个状态
160                 for (State* state : fromStates)
161                 {
162                     cout << state << " ";
163                 }
164                 cout << "} on symbol '" << symbol << "' to { ";
165
166                 // 打印目标状态集合中的每个状态
167                 for (State* state : nextStateSet)
168                 {
169                     cout << state << " ";
170                 }
171                 cout << "}" << endl;
172             }
173         }
174     }
175
176     cout << "Start States: { ";
177
178     // 打印 DFA 的起始状态集合中的每个状态
```

```
179     for (State* state : dfa->startStates)
180     {
181         cout << state << " ";
182     }
183     cout << "}" << endl;
184 
185     cout << "End States: ";
186 
187     // 打印 DFA 的终止状态集合中的每个状态
188     for (State* state : dfa->endStates)
189     {
190         cout << "State { " << state << " }" << endl;
191     }
192 }
```

3.2.4 运行截图及分析

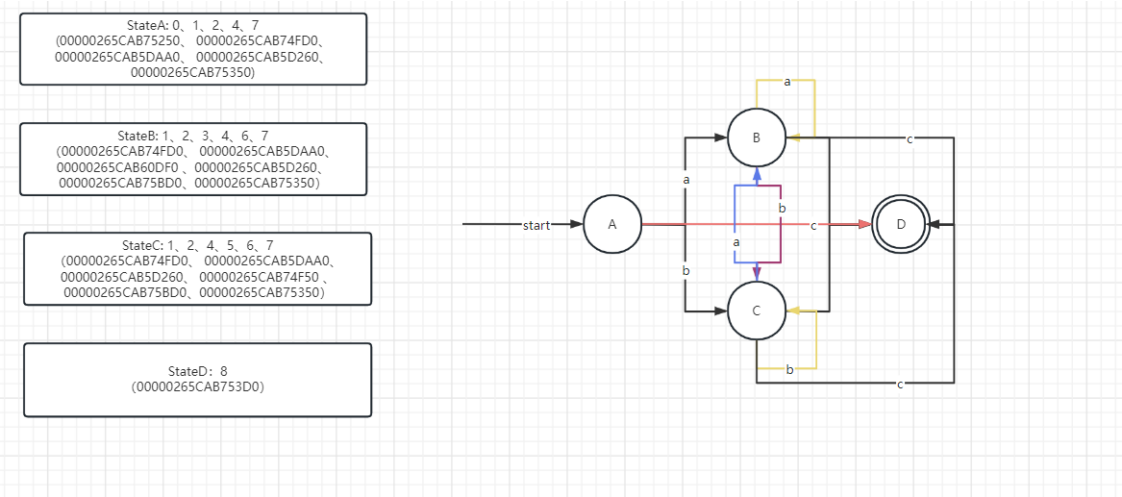
输入的测试正则表达式字符串 regex = ”(a|b)*c”
结果分析:

闭包计算	状态
ε-closure(0) =0,1,2,4,7	A
ε-closure(δ(A,a)) =3 =1,2,3,4,6,7	B
ε-closure(δ(A,b)) =5 =1,2,4,5,6,7	C
ε-closure(δ(A,c)) =8	D
ε-closure(δ(B,a)) =3 =1,2,3,4,6,7	B
ε-closure(δ(B,b)) =5 =1,2,4,5,6,7	C
ε-closure(δ(B,c)) =8	D
ε-closure(δ(C,a)) =3 =1,2,3,4,6,7	B
ε-closure(δ(C,b)) =5 =1,2,4,5,6,7	C
ε-closure(δ(C,c)) =8	D

State	Input Symbol		
	a	b	c
A	B	C	D
B	B	C	D
C	B	C	D

运行截图:
画图对运行结果截图分析如下，已经将地址对应的状态标在图中:

```
NFA->DFA
DFA States :
State{00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB75350}
State{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350}
State{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350}
DFA Transitions :
Transition from{00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB75350} on symbol 'a' to{00000265CAB74FD0 00000265CAB5DAA0 000
00265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350}
Transition from{00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB75350} on symbol 'b' to{00000265CAB74FD0 00000265CAB5DAA0 000
00265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350}
Transition from{00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB75350} on symbol 'c' to{00000265CAB753D0}
Transition from{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350} on symbol 'a' to{00000265CAB74FD0 000
00265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350}
Transition from{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350} on symbol 'b' to{00000265CAB74FD0 000
00265CAB5DAA0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350}
Transition from{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350} on symbol 'c' to{00000265CAB753D0}
Transition from{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350} on symbol 'a' to{00000265CAB74FD0 000
00265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB75BD0 00000265CAB75350}
Transition from{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350} on symbol 'b' to{00000265CAB74FD0 000
00265CAB5DAA0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350}
Transition from{00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350} on symbol 'c' to{00000265CAB753D0}
Start States : { 00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB5D260 00000265CAB75350 }
End States :State { 00000265CAB753D0 }
```



4 最小化 DFA

对于一个正规式，识别它的 DFA 中，存在唯一一个状态数最少的 DFA

DFA M，状态集 S，字母表 Σ，需要化简使其状态数最少。

符号串 w 区分状态 s、t, 分别从 s、t 开始，读入 w、转换状态，读取完毕后，一个到达终态，另一个到达非终态

4.1 概念

4.1.1 区分

- 可区分：对于任何两个状态 t 和 s，若从一状态出发接受输入字符串 w，而从另一状态出发不接受 w，或者从 t 出发和从 s 出发到达不同的接受状态，则称 w 对状态 t 和 s 是可区分的。即两个（或多个）状态根本不是一类，根据下一个输入的符号，不同状态得到的结果会出现不同。
- 不可区分：设想任何输入序列 w 对 s 和 t 均是不可区分的，则说明从 s 出发和从 t 出发，分析任何输入序列 w 均得到相同结果, 即从单词识别的角度，两者不等价，无论下一个接收的符号是什么，得到的结果都是一样的。因此，s 和 t 可以合并成一个状态

DFA 的最小化就是找出 DFA 中所有的不可区分状态，把它们合并为一个状态。

4.2 算法的基本思想

- 初始将所有状态分为两个组：终态与非终态。

对于状态组 A=s1,s2,⋯,sk，对符号 a，得到其转换状态 t1,t2,⋯,tk。

若 t1,t2,⋯,tk 属于不同状态组，则需将 A 对应划分为若干组

- 寻找所有可被区分的状态组，如果不产生新状态不可区分，则状态合并。
- 状态 <——> 已读入符号串集合
- 符号串集合并
- 符号串 s、t 并入一个集合（一个状态）<-对任何符号串 u，su、tu 都同时（不）被 DFA 接受——两个状态是不可“区分”的
- 从终态“逆向”计算

4.3 最小化 DFA 算法

输入：一个 DFA M，状态集 S，字母表 Σ，初态 s0，终态集 F，对所有 (状态，符号) 对，状态转换均有定义

输出：一个 DFA M’，L(M’)=L(M)，且 M’具有最少的状态数目

方法：

1. 构造状态集的初始划分 Π：分为两个组，终态组 F，和非终态组 S-F
2. 利用下面给出的算法将 Π 继续划分为 Πnew
3. 若 Π=Πnew，Πfinal <- Πnew，转 4，否则，Π <- Πnew，转 2
4. 每个组选出一个代表，作为 M’的状态

(1) 令 s 为一代表状态，δ(s,a)=t，t 所在组的代表状态为 r，则 δ’(s,a)=r

(2) s0 所在组的代表作为 M’的初态

(3) 终态所在组的代表作为 M’的终态

(4) 每个组或者只有终态，或者没有终态

5. 删除死状态——非终态，且所有状态转换都指向自身

删除所有从初态不可达的状态, 其他状态指向这些状态的转换都变为未定义

4.4 C++ 代码实现及相关注释

```cpp
// 这个函数接受两个参数: 一个用于检查的状态集合 'group' 和一个目标状态集合 'target'。
bool containsAll(const set<State*>& group, const set<State*>& target)
{
    // 使用 for 循环遍历目标状态集合 'target' 中的每个状态 'state'。
    // 在循环中, 我们检查状态 'state' 是否存在于状态集合 'group' 中。
    // 如果状态存在, find() 返回的迭代器不等于 'group.end()'。
    for (State* state : target)
    {
        // 如果状态 'state' 不存在于状态集合 'group', 则返回
            false, 表示未包含全部目标状态。
        if (group.find(state) == group.end())
        {
            return false;
        }
    }
```

```
15      // 如果 for 循环完成后仍未返回 false, 表示状态集合 'group' 包含了 'target'
            中的所有状态,
16      return true;
17  }
18
19  // 最小化DFA的算法
20  DFA* minimizeDFA(DFA* dfa)
21  {
22      // 初始化分组, 包含终止状态和非终止状态
23      set<set<State*>> P; // 当前分组集合
24      set<State*> nonEndStates; // 非终止状态集合
25      // 遍历 DFA 的终止状态, 并为每个状态创建一个单元素的分组, 将其添加到 P 中
26      for (State* state : dfa->endStates)
27      {
28          set<State*> endStateGroup;
29          endStateGroup.insert(state);
30          P.insert(endStateGroup);
31      }
32      // 找出 DFA 的非终止状态, 将它们添加到单独的分组中, 然后将这些分组添加到 P 中
33      for (State* state : dfa->startStates)
34      {
35          if (dfa->endStates.find(state) == dfa->endStates.end())
36          {
37              nonEndStates.insert(state);
38          }
39      }
40
41      P.insert(nonEndStates);
42
43      // 迭代合并, 直到不能再细分组
44      bool changed = true;
45      while (changed)
46      {
47          changed = false;
48          set<set<State*>> newP; // 用于存储下一轮的分组
49
50          for (set<State*> group : P)
51          {
52              // 针对每个字符检查状态是否在同一组中
53              map<char, set<State*>> transitionsForGroup;
54              for (State* state : group)
55              {
56                  // 对于当前状态 group 中的每个状态
                        state, 检查它们在不同字符下的状态转移目标,
57                  // 并将这些目标状态分组存储在 transitionsForGroup 中。
58                  for (char symbol = 'a'; symbol <= 'z'; symbol++)
59                  {
60                      if (state->transition.find(symbol) != state->transition.end())
61                      {
62                          State* nextState = state->transition[symbol];
```

```
                    for (set<State*> nextGroup : P)
                    {
                        if (nextGroup.find(nextState) != nextGroup.end())
                        {
                            transitionsForGroup[symbol] = nextGroup;
                            break;
                        }
                    }
                }
            }
        }

        // 如果 transitionsForGroup 中包含多个不同的分组，则需要拆分当前分组
        if (transitionsForGroup.size() > 1)
        {
            // 分组内有状态可以跳转到不同的组，需要拆分分组
            changed = true;
            // 将 transitionsForGroup 中的不同分组添加到 newP 中
            for (const auto& transition : transitionsForGroup)
            {
                newP.insert(transition.second);
            }
        }
        else
        {
            // 如果 transitionsForGroup 中只包含一个分组，将当前分组保持不变
            newP.insert(group);
        }
    }
    // 更新分组集合 P 为新的分组集合 newP
    P = newP;
}

// 构建最小化DFA
set<State*> startStates;
set<State*> endStates;
map<set<State*>, map<char, set<State*>>> transitions;
// 遍历最小化后的分组集合 P
for (set<State*> group : P)
{
    // 找出包含初始状态的分组并存储为 startStates
    if (containsAll(group, dfa->startStates))
    {
        startStates = group;
    }
    // 找出包含终止状态的分组并存储为 endStates
    if (containsAll(group, dfa->endStates))
    {
        endStates.insert(group.begin(), group.end());
    }
```

```
113         // 为每个字符计算状态转移关系，并存储在 transitions 中
114         map<set<State*>, set<State*>> transitionsForSymbol;
115 
116         for (char symbol = 'a'; symbol <= 'z'; symbol++)
117         {
118             for (State* state : group)
119             {
120                 if (state->transition.find(symbol) != state->transition.end())
121                 {
122                     State* nextState = state->transition[symbol];
123 
124                     for (set<State*> nextGroup : P)
125                     {
126                         if (containsAll(nextGroup, set<State*>{nextState}))
127                         {
128                             transitionsForSymbol[group].insert(nextGroup.begin(),
                                nextGroup.end());
129                             break;
130                         }
131                     }
132                 }
133             }
134             // 如果 transitionsForSymbol 非空，将其存储在 transitions 中
135             if (!transitionsForSymbol.empty())
136             {
137                 transitions[group][symbol] = transitionsForSymbol[group];
138             }
139         }
140     }
141 
142     // 构建最小化DFA
143     DFA* minimizedDFA = new DFA(startStates, endStates);
144     minimizedDFA->transitions = transitions;
145 
146     return minimizedDFA;
147 }
148 
149 void printMinDFA(DFA* dfa)
150 {
151     cout << "Minimized DFA States:" << endl;
152     // 遍历最小化的 DFA 中的状态和状态转移
153     for (const auto& stateTransitionPair : dfa->transitions)
154     {
155         const set<State*>& stateSet = stateTransitionPair.first;
156         cout << "State { ";
157         // 输出当前状态集合的所有状态
158         for (State* state : stateSet)
159         {
160             cout << state << " ";
161         }
```

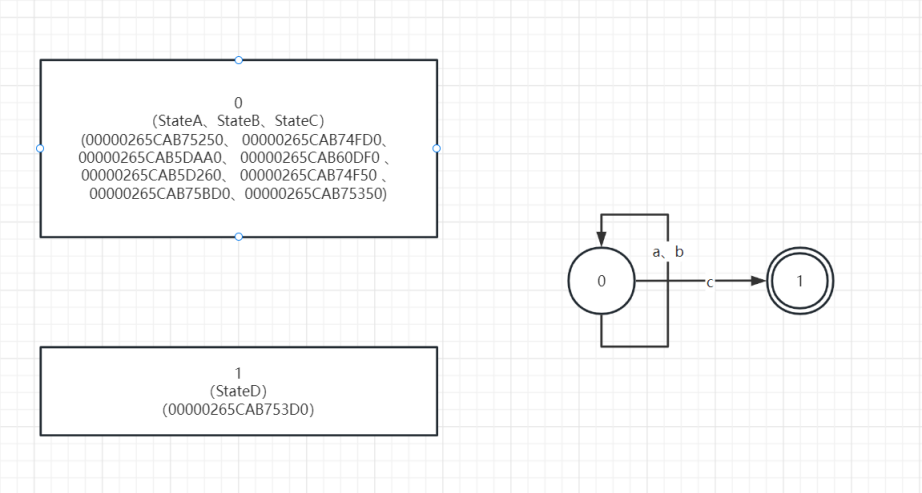
```
162         cout << "}" << endl;
163     }
164 
165     cout << "Minimized DFA Transitions:" << endl;
166     for (const auto& transition : dfa->transitions)
167     {
168         const set<State*>& fromStates = transition.first;
169         const map<char, set<State*>>& toStates = transition.second;
170         // 遍历 DFA 的状态转移关系
171         for (char symbol = 'a'; symbol <= 'z'; symbol++)
172         {
173             if (toStates.find(symbol) != toStates.end())
174             {
175                 const set<State*>& nextStateSet = toStates.at(symbol);
176 
177                 cout << "Transition from { ";
178                 // 输出状态转移的起始状态
179                 for (State* state : fromStates)
180                 {
181                     cout << state << " ";
182                 }
183                 cout << "} on symbol '" << symbol << "' to { ";
184                 // 输出状态转移的目标状态
185                 for (State* state : nextStateSet)
186                 {
187                     cout << state << " ";
188                 }
189                 cout << "}" << endl;
190             }
191         }
192     }
193 
194     cout << "Minimized DFA Start States: { ";
195     // 输出最小化的 DFA 的起始状态
196     for (State* state : dfa->startStates)
197     {
198         cout << state << " ";
199     }
200     cout << "}" << endl;
201 
202     cout << "Minimized DFA End States: " ;
203     // 输出最小化的 DFA 的终止状态
204     for (State* state : dfa->endStates)
205     {
206         cout << "State { " << state << " }" << endl;
207     }
208 }
```

4.5 运行截图及分析

输入的测试正则表达式字符串 regex = ”(a|b)*c”
运行截图:

```
Minimized DFA
Minimized DFA States:
State { 00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350 }
Minimized DFA Transitions :
Transition from {00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350} on
 symbol 'a' to {00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350}
Transition from {00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350} on
 symbol 'b' to {00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350}
Transition from {00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 00000265CAB75350} on
 symbol 'c' to {00000265CAB753D0}
Minimized DFA Start States : { 00000265CAB75250 00000265CAB74FD0 00000265CAB5DAA0 00000265CAB60DF0 00000265CAB5D260 00000265CAB74F50 00000265CAB75BD0 000002
65CAB75350 }
Minimized DFA End States : State{ 00000265CAB753D0 }
```

画图对运行结果截图分析如下，已经将地址对应的状态标在图中:



5 总结

本次实验中完成了词法分析器核心构造算法: 正则表达式->NFA 的 Thompson 构造法、NFA->DFA 的子集构造法、以及 DFA 的最小化算法。经过本次实验，对词法分析有了更深入的了解，更深入的掌握了相关的重要知识，对未来在编译原理课程方面进一步的学习奠定下坚实的基础。

5.1 源码链接

https://pan.baidu.com/s/1aYaI2-PP8R4KL4vUQ2HG1A?pwd=n68p