南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

OT1 实现词法分析器构造算法

谢雯菲

年级：2021 级

专业：信息安全

指导教师：王刚

2023 年 11 月 5 日

摘要

关键字：编译原理 正则表达式 NFA DFA DFA 最小化

目录

一、 算法实现

本次实验使用 C++ 语言完成正则表达式到 NFA 最小化的实现。

(一) 正则表达式 →NFA

1. 获取字符集

由于在后续的很多操作中需要知道输入的正则表达式中的字符集，因此在开始分析正则表达式之前先对正则表达式中的字符进行扫描。

从输入的正则表达式中获取字符集的函数如下所示：

获取字符集

```
int charNum = 0;
vector<char> charSET;
void getChar(string s) {
    for (const auto c : s) {
        if ((isText(c)=='a') && findChar(c) == -1) {
            charSET.emplace_back(c);
            charNum++;
        }
    }
}
```

为了便于后续的判断，还有一个判断某字符是否存在于字符集中的函数，成功则返回在字符集数组中的下标，否则返回-1：

获取字符下标

```
int findChar(char c) {
    for (int i = 0; i < charNum; i++) {
        if (charSET[i] == c) {
            return i;
        }
    }
    return -1;
}
```

2. 正则表达式预处理

为了确认正则表达式的计算优先级，选用将正则表达式的中缀表现形式转换为后缀表现形式。因此，在读取正则表达式后，先对正则表达式进行一个预处理。

判断正则表达式是否合法及添加连接符 首先，对正则表达式进行一个初步的检测。由于本次实验只实现了对正则表达式中 |、*、(、) 四个符号，因此需要保证输入的表达式中的运算符只包含这四个。

由于正则表达式除了实现上述符号的运算以外，还有一个连接的运算没有符号的参与，因此，在需要表示连接的地方手动加上“-”符号，便于后续程序的编写。

使用的函数及注释如下所示：

判断正则表达式是否合法

```
//预处理
//判断某个字符是否是字符
char isText(char c) { //如果是运算符返回自身，否则返回'a'
    if (c == '(' || c == ')' || c == '*' || c == '|') {
        return c;
    }
    return 'a';
}

//判断表达式是否是合法的正则表达式
bool isRegex(string &s) {
    if (!s.size()) {
        //长度是否为0
        return false;
    }
    int l = 0;
    for (const auto &t : s) {
        //判断括号是否匹配
        if (t == '(') {
            l++;
        }
        else if (t == ')') {
            l--;
        }
        if (l < 0) {
            return false;
        }
    }
    if (l != 0) {
        return false;
    }
    //用-连接a-a,a*-a,a-(,)-a,)-(,a*-(
    for (int i = 0; i < s.size()-1; i++) {
        char a = isText(s[i]);
        char b = isText(s[i + 1]);
        if ((a == 'a' && b == 'a') || (a == '*' && b == 'a') || (a == 'a' &&
            b == '(') || (a == ')' && b == 'a') || (a == ')' && b == '(') ||
            (a == '*' && b == '(')) {
            s.insert(i + 1, "-");
            i++;
        }
    }
    cout << s << "是合法正则表达式! " << endl;
    return true;
}
```

获取运算符优先级 在将中缀表达式转为后缀表达式时，需要确认运算符的优先级，设置的运算符的优先级如下所示（由于右括号不需要入栈，因此此处不包括右括号）：

运算符	优先级
(	1
|	2
-	3
*	4

表 1: 优先级表

中缀表达式转后缀表达式 在实现该算法时使用的想法和普通运算符的想法基本类似。

1. 使用一个栈压入和弹出运算符，使用一个新的字符串存储最终的结果。从第一个字符开始读取预处理后的正则表达式。
2. 如果是字符，直接加入字符串末尾；如果是运算符，进行如下操作：
   - 如果运算符栈为空或当前运算符为“(”，直接入栈；
   - 如果当前运算符的优先级大于栈顶运算符，直接入栈；
   - 如果当前运算符的优先级小于等于栈顶运算符，弹出栈顶运算符加到字符串末尾，直到当前运算符的优先级大于栈顶运算符后，当前运算符入栈；
   - 如果当前运算符为“)”，依次弹出栈顶运算符加入字符串末尾，直到遇到“(”，舍弃括号符号。
3. 重复 2 操作，直到读取字符串为空。最后将运算符栈顶符号依次弹出加入结果字符串，直到栈为空，完成转换。

代码算法实现如下：

中缀转后缀

```
string toSuffix(const string& expr) {
        stack<char> op; //存储运算符
        string suffix; //存储表达式
        //遍历正则表达式
        for (const auto& c : expr) {
                if (isOperator(c)) {
                        //如果是运算符的话
                        if (op.empty()) { //如果栈空, 直接入栈
                                op.push(c);
                        }
                        else {
                                //如果栈顶运算符优先级大于等于当前的符号
                                //则全部弹出栈, 加入suffix末尾
                                while (!op.empty()) {
                                        char t = op.top();
                                        if (getPriority(t) >= getPriority(c))
                                                {
```

```
                                        op.pop();
                                        suffix.push_back(t);
                                    }
                                    else {
                                        break;
                                    }
                                }
                                //直到弹出所有优先级大于当前运算符的符号时
                                //将当前运算符压入栈
                                op.push(c);
                            }
                        }
                        else { //如果不是运算符
                            if (c == '(') { //左括号直接入栈
                                op.push(c);
                            }
                            else if (c == ')') {
                                //如果是右括号
                                //依次弹出栈顶运算符加入suffix, 直到遇到左括
                                    号
                                while (op.top() != '(') {
                                    suffix.push_back(op.top());
                                    op.pop();
                                }
                                op.pop(); //括号舍弃
                            }
                            else {
                                //操作符直接加入suffix
                                suffix.push_back(c);
                            }
                        }
                    }
                    //取出剩余运算符
                    while (!op.empty()) {
                        suffix.push_back(op.top());
                        op.pop();
                    }
                    return suffix;
                }
```

3. 后缀表达式转 NFA

教材中有说明将正则表达式转换为 NFA 的规则。实验中用到的规则整理如图1所示:

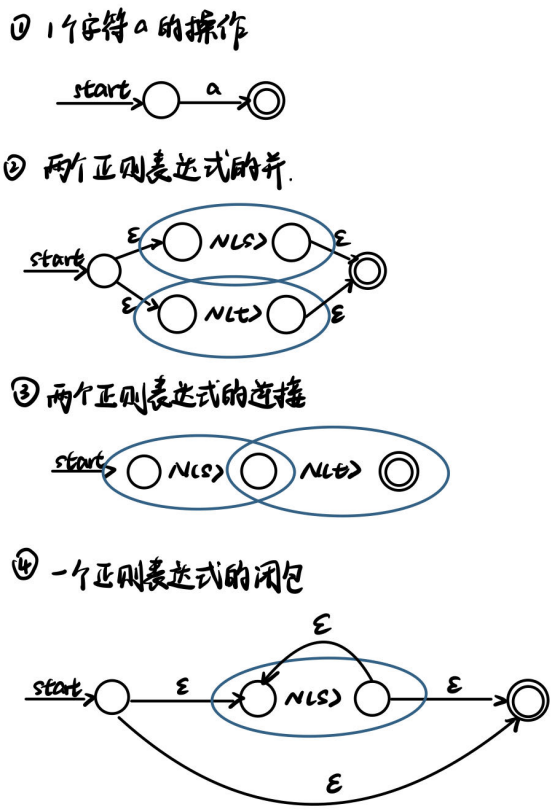

图 1: 子表达式转 NFA 规则

设计存储 NFA 的结构如下图2所示:

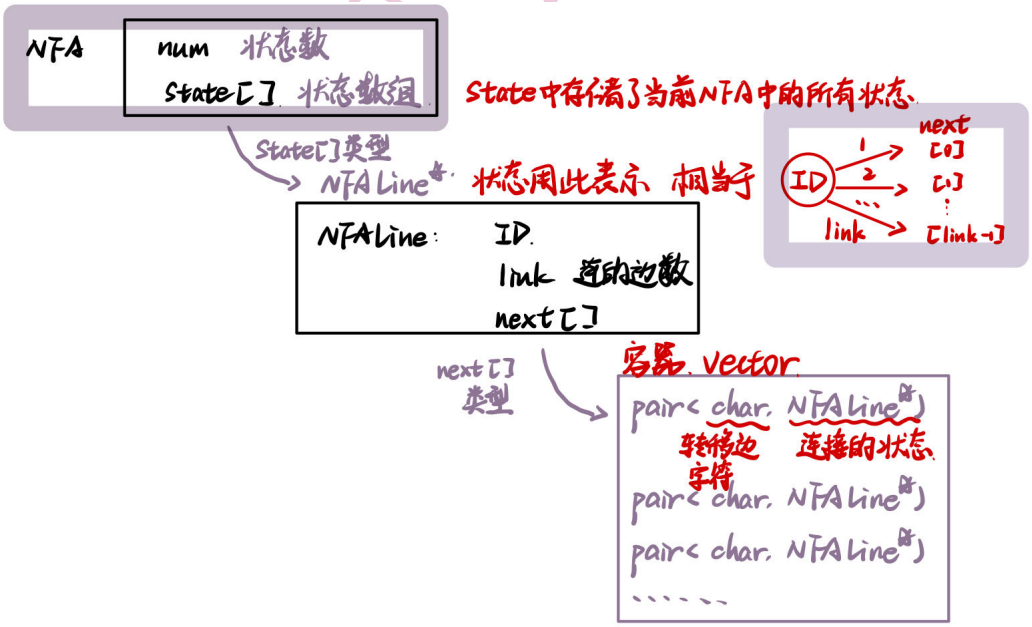

图 2: 存储 NFA 的结构

- NFALine：用来存储 NFA 状态节点及其连接的节点的结构。包括 ID 号、连接的节点数量

和连接的节点地址的数组。

- NFALine.next[]：用于存储连接的节点地址的数组。使用一个 Vector<pair<char,NFALine*»的结构表示，char 相当于导出的一条边上的字符。
- NFA：表示 NFA 子表达式的结构。包括该子表达式所有的状态数以及存储所有状态地址的数组。其中，每个 NFA 中还用两个指针分别标明了该子表达式的开始状态地址和结束状态地址。

创建 NFA 的思路如下：

1. 创建一个 NFA 类型的栈存储子表达式。从后缀表达式的第一个字符开始读取。
2. 取出剩下字符串的第一个字符并判断：
   - 如果是一个字符，使用规则 创建一个子表达式并入栈；
   - 如果是“|”符号，从栈中先后弹出 N(t) 和 N(s)，使用规则 创建一个子表达式并入栈；
   - 如果是“-”符号，从栈中先后弹出 N(t) 和 N(s)，使用规则 创建一个子表达式并入栈；
   - 如果是”*” 符号，从栈中弹出 N(s)，使用规则 创建一个子表达式并入栈。
3. 重复 2 步骤，直到字符串读取完毕。最后留在栈中的 NFA 就是最终构造的 NFA。

代码实现如下所示：

构造 NFA

```
//定义存储NFA的结构
//NFA的边(状态->连接的状态)
struct NFALine {
        int ID = -1;
        int link; //后续转移的状态数,-1表示该状态是结束状态
        //下一个转移的节点
        //转移边的字符, '#'表示空
        vector<pair<char, NFALine*>> next;
        NFALine() {
                link = 0;
        }
        NFALine(int l) {
                link = l;
        }
};
//NFA有关操作
struct NFA {
        int num; //当前NFA中的状态数
        vector<NFALine*> State; //为当前所有的状态编号
        NFALine* start; //当前子NFA的开始状态
        NFALine* end;  //当前子NFA的结束状态
        NFA() {
                start = new NFALine();
```

```
24              State.emplace_back(start);
25              end = start;
26              num = 1;
27          }
28          //加一个字符a的情况
29          void addLineA(char c) {
30              NFALine* e = new NFALine(-1);
31              start->link++;
32              start->next.emplace_back(c, e);
33              end = e;
34              num = 2;
35              State.emplace_back(end);
36          }
37          //并的情况
38          void orLineA(NFA* linka) {
39              State.insert(State.end(), linka->State.begin(), linka->State.
                    end());
40              NFALine* e = new NFALine(-1);
41              end->link = 1;
42              end->next.emplace_back('#', e);
43              linka->end->link = 1;
44              linka->end->next.emplace_back('#', e);
45              end = e;
46              NFALine* s = new NFALine();
47              s->link += 2;
48              s->next.emplace_back('#', start);
49              s->next.emplace_back('#', linka->start);
50              start = s;
51              num = num + linka->num + 2;
52              State.insert(State.begin(), start);
53              State.emplace_back(end);
54          }
55          //连接的情况
56          void linkLineA(NFA* linka) {
57              State.insert(State.end(), linka->State.begin()+1, linka->
                    State.end());
58              end->link = linka->start->link;
59              end->next = linka->start->next;
60              end = linka->end;
61              num = num + linka->num - 1;
62          }
63          //闭包
64          void eLinkA() {
65              NFALine* e = new NFALine(-1);
66              end->link = 2;
67              end->next.emplace_back('#', e);
68              end->next.emplace_back('#', start);
69              end = e;
```

```
                NFALine* s = new NFALine();
                s->link += 2;
                s->next.emplace_back('#', start);
                s->next.emplace_back('#', end);
                start = s;
                State.insert(State.begin(), start);
                num += 2;
                State.emplace_back(end);
        }
        //设置状态号
        void SetID() {
                for (int i = 0; i < num; i++) {
                        NFALine* temp = State[i];
                        int ID = i;
                        temp->ID = ID;
                }
        }
};
//创建NFA的函数
NFA buildNFA(string s) {
        //存储当前已构造的子NFA
        stack<NFA> stk;
        for (auto c : s) {
                //如果是用一个字符连接
                if (c != '|' && c != '*' && c != '-') {
                        NFA a;
                        a.addLineA(c);
                        stk.push(a);
                }
                //如果是*
                else if (c == '*') {
                        NFA a = stk.top();
                        stk.pop();
                        a.eLinkA();
                        stk.push(a);
                }
                //如果是|
                else if (c == '|') {
                        NFA b = stk.top();
                        stk.pop();
                        NFA a = stk.top();
                        stk.pop();
                        a.orLineA(&b);
                        stk.push(a);
                }
                //如果是-
                else if (c == '-') {
                        NFA b = stk.top();
```

```
                        stk.pop();
                        NFA a = stk.top();
                        stk.pop();
                        a.linkLineA(&b);
                        stk.push(a);
                }
        }
        return stk.top();
}
//检验
void testprint(NFA r) {
        r.SetID();
        for (int i = 0; i < r.num; i++) {
                NFALine* temp = r.State[i];
                cout << "状态" << i << ":";
                int k = temp->link;
                if (k != -1) {
                        cout << "连接了" << k << "条边" << endl << '\t';
                        for (int j = 0; j < k; j++) {
                                cout << temp->next[j].first << "->" << temp->
                                    next[j].second->ID << ",␣";
                        }
                }
                else {
                        cout << "终止状态";
                }
                cout << endl;
        }
}
```

(二) NFA→DFA

教材中有提及 NFA 转 DFA 的算法。伪代码如下图3所示:

一开始, ϵ-$closure(s_0)$是 $Dstates$ 中的唯一状态, 且它未加标记;
while (在 $Dstates$ 中有一个未标记状态 T) {
    给 T 加上标记;
    for (每个输入符号 a) {
        $U = \epsilon\text{-}closure(move(T, a))$;
        if (U 不在 $Dstates$ 中)
            将 U 加入到 $Dstates$ 中, 且不加标记;
        $Dtran[T, a] = U$;
    }
}

图 3: NFA 转 DFA 伪代码

1. e_ closure() 算法

求某一个状态的空转移闭包的过程如下：

1. 创建一个栈，将所选状态集合中的所有的状态都压入栈。
2. 如果栈不为空，取出栈顶状态，求出该状态的所有空转移状态，添加到状态集合中并压入栈。
3. 重复步骤 2，直到栈为空，最后得到的集合就是状态的空转移闭包。

代码实现如下所示：

e _ closure 算法

```
//求空串转移状态集合
void e_closure(set<int>& state) {
        stack<int> wait;
        set<int>::iterator it;
        for (it = state.begin(); it != state.end(); it++) {
                wait.push(*it);
        }
        while (!wait.empty()) {
                int temp = wait.top();
                wait.pop();
                NFALine* tempLine = NFAstate[temp];
                int num = tempLine->link;
                for (int i = 0; i < num; i++) {
                        if (tempLine->next[i].first == '#') {
                                state.insert(tempLine->next[i].second->ID);
                                wait.push((tempLine->next[i].second)->ID);
                        }
                }
        }
}
```

2. move() 字符转移算法

实现 move() 函数的思路如下：

1. 创建一个新状态集合。
2. 遍历所选集合的状态，将遍历状态能通过选定字符到达的状态加入创建的集合。
3. 最后得到的集合就是字符转移后的状态集合。

代码实现如下所示：

字符转移算法

```
void move(set<int>& s, char a,set<int>& change) {
        set<int>::iterator it;
        //存储状态转移后的包
```

```
        for (it = s.begin(); it != s.end(); it++) {
                traState(*it, a, change);
        }
}
```

3. 状态转移算法

根据上述伪代码实现的状态转移算法如下:

状态转移算法

```
//存储状态数
int StateNum = 0;
//存储状态和标志
vector<pair<bool, set<int>>> Dstates;
//存储状态转移矩阵
vector<int*> Dtran;
//NFAnum
int NFAend = 0;

//求转移后的状态集合
void traState(int s,char c,set<int>& state) {
        NFALine* temp = NFAstate[s];
        int num = temp->link;
        for (int i = 0; i < num; i++) {
                if (temp->next[i].first == c) {
                        state.insert((temp->next[i].second)->ID);
                }
        }
}

//判断集合是否在Dstates中
int findState(set<int> a) {
        for (int i = 0; i < StateNum; i++) {
                if (a == Dstates[i].second) {
                        return i;
                }
        }
        return -1;
}

void getTransition(NFA nfar) {
        NFAstate = nfar.State;
        NFAend = nfar.num - 1;
        //已标记状态位移
        int offset = 0;
        //载入初始状态
        set<int> start;
        start.insert(0);
```

```
        e_closure(start);
        Dstates.emplace_back(false, start);
        int* tran = new int[charNum];
        Dtran.emplace_back(tran);
        StateNum++;
        while (offset < StateNum) {
                //加上标记
                set<int> T = Dstates[offset].second;
                Dstates[offset].first = true;
                for (int i = 0; i < charNum; i++) {
                        set<int> change;
                        move(T, charSET[i], change);
                        if (change.size() == 0) {
                                Dtran[offset][i] = -1;
                                continue;
                        }
                        e_closure(change);
                        set<int> U = change;
                        //U的状态号
                        int index = findState(U);
                        if (index==-1) {
                                Dstates.emplace_back(false, U);
                                int* tran = new int[charNum];
                                Dtran.emplace_back(tran);
                                index = StateNum;
                                StateNum++;
                        }
                        Dtran[offset][i] = index;
                }
                offset++;
        }
}
```

(三) DFA 的最小化算法

算法定义了一个 vector<set<int>,int> 结构存储 pai 分组。

1. 初始划分

构造状态集的初始划分。终态为一组，非终态为一组，分别存入 pai。
算法实现如下:

初始划分

```
//初始划分
void initPai() {
        set<int> endyes;
        set<int> endno;
        for (int i = 0; i < StateNum; i++) {
```

```
                set<int> temp = Dstates[i].second;
                if (temp.find(NFAend) == temp.end()) {
                        endno.insert(i);
                }
                else {
                        endyes.insert(i);
                        DFAend.insert(i);
                }
        }
        pai.emplace_back(endyes, 0);
        paiNum++;
        if (endno.size() != 0) {
                pai.emplace_back(endno, 0);
                paiNum++;
                removeDied();
        }
}
```

其中，removeDied() 函数的作用是去除死状态，将非终态集合中所有状态转移后还是自身的状态去除。算法实现如下：

去除死状态

```
void removeDied() {
        set<int>::iterator it;
        for (it = pai[1].first.begin(); it != pai[1].first.end(); it++) {
                bool flag = true;
                for (int j = 0; j < charNum; j++) {
                        if (Dtran[*it][j] != *it) {
                                flag = false;
                                break;
                        }
                }
                if (flag) {
                        pai[1].first.erase(*it);
                }
        }
}
```

2. 划分算法

实验设计实现的划分算法思路如下：

1. 使用 offset 表示 pai 中开头已经确认过的分组。如果 offset 不等于状态数，每次读取 pai 中末尾的状态集合。
2. 创建一个新的 vector<set<int>,int> temp 存储集合中状态通过字符的转移情况。遍历字符集中的字符，查询状态通过该字符所得的状态号：
   - 如果所得的状态号在 temp 中已经出现过，则将当前状态归入；

- 否则在 temp 中添加新的一行，将状态归入。

3. 根据 temp 的状态进行划分：
   - 如果 temp 超过 1 行，说明当前所在状态集合需要重新划分。将 pai 末尾的状态弹出，将按照 temp 进行的新划分加入 pai 末尾。由于有新状态的加入，开头已经确认过的分组也需要重新确认，因此将 offset 清零；
   - 如果 temp 只有 1 行，说明当前状态所在的集合不用重新划分。将集合加到 pai 的开头，并且将 offset+1.
4. 重复 1、2 步骤，直到 offset 等于状态数，说明全部状态都已经确认。pai 中所有的状态集合就是划分的结果。

算法示意图如下图4所示：

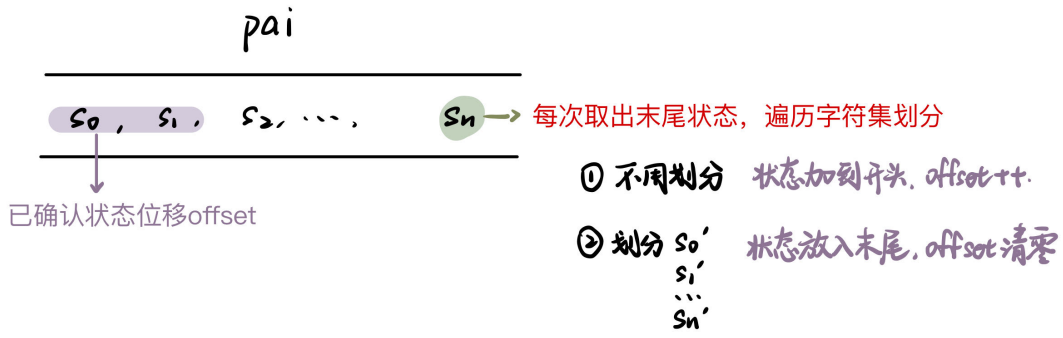

图 4: 状态划分算法示意图

代码实现如下所示：

状态划分

```
//状态划分
//offset表示已经完成分类的状态
//paiNum表示数组最后一个状态
//当全部完成检验时最小化完成
void dividePai() {
        if (paiNum == 1) {
                cout << "只有终态" << endl;
                return;
        }
        while (offset < paiNum) {
                set<int> now = pai[paiNum - 1].first;
                //index表示该集合之前确认过几个字符
                int index = pai[paiNum - 1].second;
                //用来存储有几个新的状态<状态集合，转移的状态号所在集合>
                vector<pair<set<int>, int>> temp;
                set<int>::iterator it;
                for (it = now.begin(); it != now.end(); it++) {
```

```
                int toS = Dtran[*it][index];
                int toSindex = getPaiIndex(toS);
                int tempnum = temp.size();
                bool exit = false;
                for (int j = 0; j < tempnum; j++) {
                        if (toSindex == temp[j].second) {
                                exit = true;
                                temp[j].first.insert(*it);
                                break;
                        }
                }
                if (!exit) {
                        set<int> newset;
                        newset.insert(*it);
                        temp.emplace_back(newset, toSindex);
                }
        }
        pai.pop_back();
        paiNum--;
        int tempnum = temp.size();
        //如果转移后有空状态，并且只有两个状态的话，不用区别
        //否则报错，让我再研究下怎么处理
        bool havEmpty = false;
        for (int j = 0; j < tempnum; j++) {
                if (temp[j].second == -1) {
                        havEmpty = true;
                        break;
                }
        }
        if (havEmpty) {
                if (tempnum == 2) {
                        temp.clear();
                        temp.emplace_back(now, -1);
                        tempnum = 1;
                }
        }
        //如果只有一个集合并且index=charNum-1，加到开头
        //如果index!=charNum-1，index+1重新加回去
        if (tempnum == 1) {
                if (index == charNum - 1) {
                        pai.insert(pai.begin(), make_pair(temp[0].
                            first, 0));
                        offset++;
                }
                else {
                        pai.emplace_back(temp[0].first, index + 1);
                }
                paiNum++;
```

```
65                          continue;
66                  }
67                  offset = 0;
68                  //如果集合中只有一个元素，直接放到开头
69                  //否则index清零，重新加到末尾
70                  for (int j = 0; j < tempnum; j++) {
71                          if (temp[j].first.size() == 1) {
72                                  pai.insert(pai.begin(), make_pair(temp[j].
                                        first, 0));
73                                  offset++;
74                          }
75                          else {
76                                  pai.emplace_back(temp[j].first, 0);
77                          }
78                          paiNum++;
79                  }
80          }
81  }
```

二、 实验验证

(一) 实验验证 1

使用正则表达式：a(ab)*|(b|c)
手算各个过程如下所示：

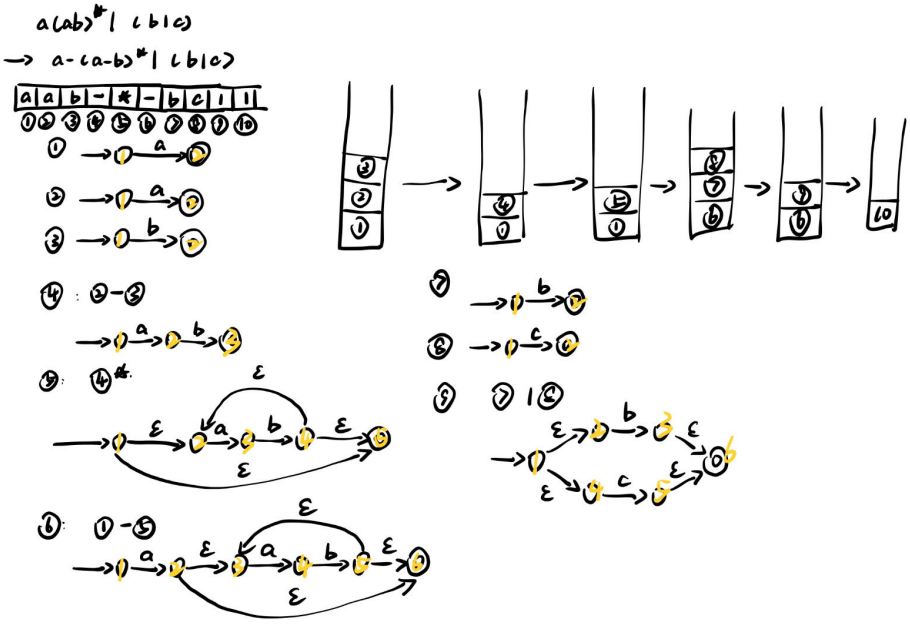

图 5: 手算过程 1

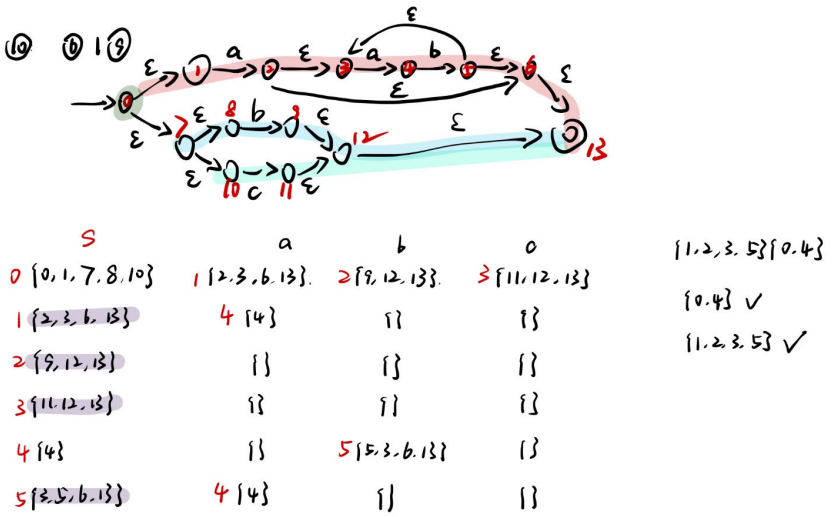

图 6: 手算过程 2

代码运行结果如下:

例 1 结果

```
a(ab)*|(b|c)
输入字符数为: 3
a-(a-b)*|(b|c)是合法正则表达式!
后缀表达式: aab-*-bc||
状态0:连接了2条边
        #->1, #->7,
状态1:连接了1条边
        a->2,
状态2:连接了2条边
        #->3, #->6,
状态3:连接了1条边
        a->4,
状态4:连接了1条边
        b->5,
状态5:连接了2条边
        #->6, #->3,
状态6:连接了1条边
        #->13,
状态7:连接了2条边
        #->8, #->10,
状态8:连接了1条边
        b->9,
状态9:连接了1条边
        #->12,
状态10:连接了1条边
        c->11,
```

状态11:连接了1条边
#->12,
状态12:连接了1条边
#->13,
状态13:终止状态

状态转换表:

State	a->	b->	c->
{0,1,7,8,10,} {11,12,13,}	{2,3,6,13,}	{9,12,13,}	
{2,3,6,13,}	{4,}	空	空
{9,12,13,}	空	空	空
{11,12,13,}	空	空	空
{4,}	空	{3,5,6,13,}	空
{3,5,6,13,}	{4,}	空	空

状态转换矩阵:

S	a	b	c
0	1	2	3
1	4	空	空
2	空	空	空
3	空	空	空
4	空	5	空
5	4	空	空

获取的新状态集合:
最终化简为2个状态:
状态0: {0,4,}
状态1: {1,2,3,5,}

化简后的状态转移矩阵: (带*表示为终态)

S	a	b	c
0	1	1	1
1*	0	-1	-1

(二) 实验验证 2

使用正则表达式: (ab)*(a*|b*)(ba)*
手算过程如下所示:

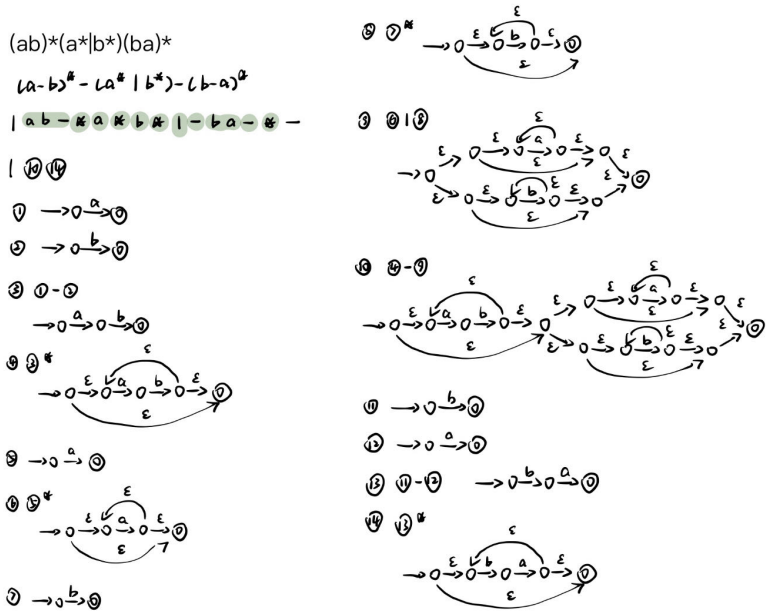

图 7: 手算过程 1

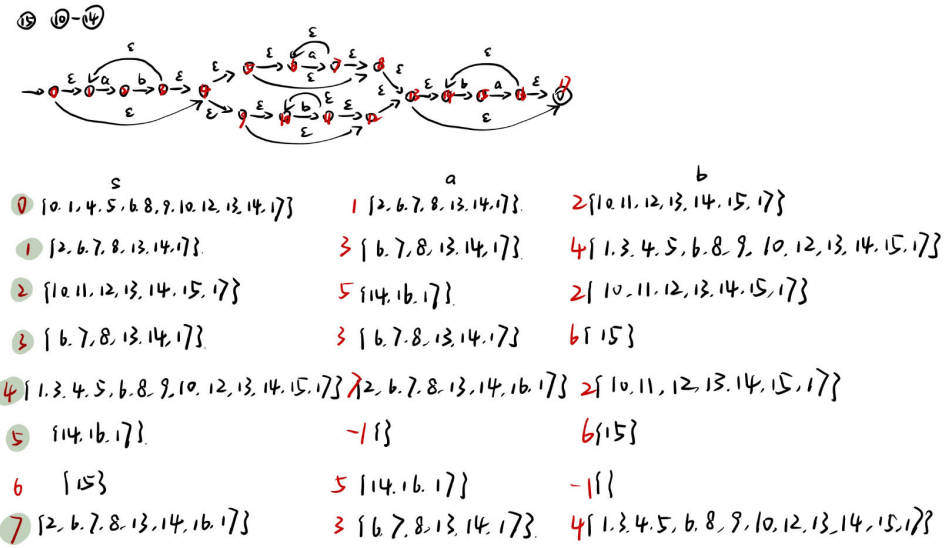

图 8: 手算过程 2

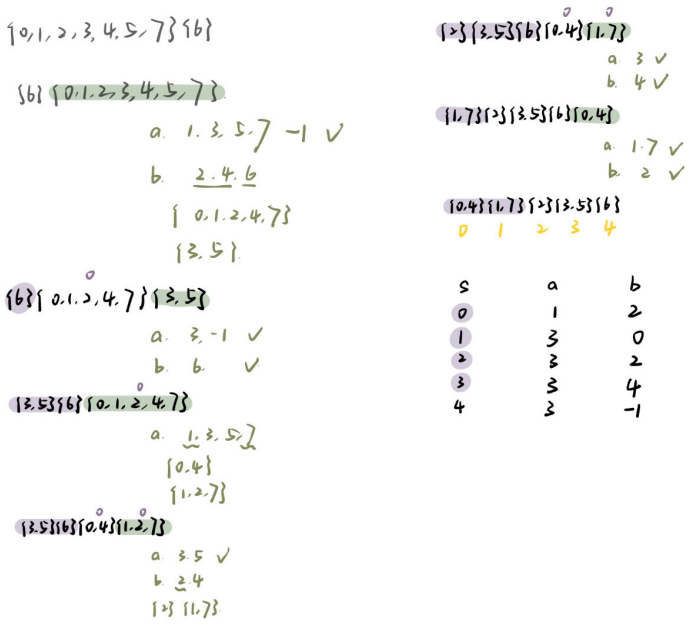

图 9: 手算过程 3

程序运行结果如下:

例 2 结果

```
(ab)*(a*|b*)(ba)*
输入字符数为: 2
(a-b)*-(a*|b*)-(b-a)*是合法正则表达式!
后缀表达式: ab-*a*b*|-ba-*-
状态0:连接了2条边
        #->1, #->4,
状态1:连接了1条边
        a->2,
状态2:连接了1条边
        b->3,
状态3:连接了2条边
        #->4, #->1,
状态4:连接了2条边
        #->5, #->9,
状态5:连接了2条边
        #->6, #->8,
状态6:连接了1条边
        a->7,
状态7:连接了2条边
        #->8, #->6,
状态8:连接了1条边
        #->13,
状态9:连接了2条边
        #->10, #->12,
状态10:连接了1条边
        b->11,
```

状态11:连接了2条边
#->12, #->10,
状态12:连接了1条边
#->13,
状态13:连接了2条边
#->14, #->17,
状态14:连接了1条边
b->15,
状态15:连接了1条边
a->16,
状态16:连接了2条边
#->17, #->14,
状态17:终止状态

状态转换表:

State	a->	b->
{0,1,4,5,6,8,9,10,12,13,14,17,}	{2,6,7,8,13,14,17,}	{10,11,12,13,14,15,17,}
{2,6,7,8,13,14,17,}	{6,7,8,13,14,17,}	{1,3,4,5,6,8,9,10,12,13,14,15,17,}
{10,11,12,13,14,15,17,}	{14,16,17,}	{10,11,12,13,14,15,17,}
{6,7,8,13,14,17,}	{6,7,8,13,14,17,}	{15,}
{1,3,4,5,6,8,9,10,12,13,14,15,17,}	{2,6,7,8,13,14,16,17,}	{10,11,12,13,14,15,17,}
{14,16,17,}	空	{15,}
{15,}	{14,16,17,}	空
{2,6,7,8,13,14,16,17,}	{6,7,8,13,14,17,}	{1,3,4,5,6,8,9,10,12,13,14,15,17,}

状态转换矩阵:

S	a	b
0	1	2
1	3	4
2	5	2
3	3	6
4	7	2
5	空	6
6	5	空
7	3	4

获取的新状态集合:
最终化简为5个状态:
状态0: {0,4,}
状态1: {1,7,}
状态2: {2,}
状态3: {3,5,}
状态4: {6,}

化简后的状态转移矩阵：（带∗表示为终态）

S	a	b
0∗	1	2
1∗	3	0
2∗	3	2
3∗	3	4
4	3	−1

实验验证完毕。