南開大學
Nankai University

计算机学院

编译系统原理报告

OT1 词法分析器核心构造算法

姓名：王茂增

学号：2113972

专业：计算机科学与技术

2023 年 11 月 5 日

摘要

在本次实验中，我实现了词法分析器的核心构造算法，首先输入正则表达式字符串，然后构造正则表达式语法树、构造 NFA、构造 DFA、进行化简、输出状态转移矩阵和进行测试。

对于正则表达式语法树，每个树结点有左右子节点指针、类别。字符等成员变量，构成语法树，然后本程序使用根节点代表整棵树，没有实现语法树类。

对于 NFA，NFA 节点类有四个成员变量，分别为节点编号当前节点的 eps 闭包、当前节点的转换函数、是否为接受状态。通过转换函数将各个节点连接形成 DFN，记录 eps 闭包方便生成 NFA 节点。然后，实现 NFA 类，存储开始节点和结束节点的指针，以及使用静态成员变量 vector 记录下所有节点。

对于 DFA，DFA 节点类有四个成员变量，分别为节点编号、该 DFA 节点包含了哪些 NFA 节点的编号、当前节点的转换函数、是否为接受状态。通过 trans_func 将各个节点连接形成 DFN。然后，实现 DFA 类，存储开始节点的编号、字符集，以及使用静态成员变量 vector 记录下所有节点。

对于 DFA 化简，本程序在 DFA 类中实现了成员函数进行化简。

代码链接：https://github.com/mzwangg/Compilers

关键字：词法分析器　正则表达式　NFA　DFA

目录

1 代码讲解

1.1 生成正则表达式语法树

1. 正则表达式语法树节点类型

通过枚举为正则表达式语法树节点设定了 4 种类型，分别是 CHARR: 字符类型、CONCAT: 拼接类型、UNION: 并集类型、CLOSURE: 闭包类型。

```
enum TYPE { CHARR,
  CONCAT,
  UNION,
  CLOSURE };
```

2. 正则表达式语法树节点

节点有四个成员变量，分别为 type: 类别、c: 字符、re0: 左子节点、re1: 右子节点，这些属性不一定都有，没有的置为 NULL。通过该类构成一棵树来存储正则表达式的语法树，不过本程序并没有实现正则表达式树类，而是使用根节点代表这棵树。

```
class Regex
{
  public:
  // 下列属性不一定都有，没有的置为NULL
  TYPE type;  // 类别
  char c;     // 字符
  Regex *re0; // 左子节点
  Regex *re1; // 右子节点

  Regex(){};
  Regex(TYPE, char, Regex *, Regex *);
  ~Regex();
};
```

3. 构建正则表达式语法树

该函数处理 str 中下标为 [l,r] 的部分，生成语法树，并返回当前子树的根节点，对于括号中的部分，该函数会递归调用。

在实现的过程中，该函数会首先寻找最外层的"|" 然后处理最外层的'|'，生成语法树并返回。如果最外层没有'|'，则对最外层按照拼接操作将正则表达式划分为各个部分。对于各个部分，如果是闭包运算，则 new 一个闭包节点；如果是左右括号，则对括号内的部分进行递归；如果是其他字符，则 new 一个字符节点。当各个部分完成之后，再按照顺序生成拼接节点，最后返回根节点代表这棵正则表达式语法树。

```
// str为正则表达式字符串，l和r为左右边界下标
// 该函数处理str中下标为[l,r]的部分，生成语法树，并返回当前子树的根节点
Regex *strToRegex(const char *str, int l, int r)
{
  // 寻找最外层的"|"
```

```
6   vector<int> unions;
7   int level = 0;
8   for (int i = l; i < r; i++) {
9     if (str[i] == '(') {
10      level++;    } else if (str[i] == ')') {
11      level--;
12    } else if (str[i] == '|' && level == 0) {
13      unions.push_back(i);
14    }
15  }
16
17  // 处理最外层的'|', 生成语法树
18  if (!unions.empty()) {
19    int size = unions.size();
20    unions.push_back(r);
21    Regex *regex = strToRegex(str, l, unions[0]);
22    for (int i = 0; i != size; ++i)
23    regex = new Regex(UNION, NULL, regex, strToRegex(str, unions[i] + 1, unions[i +
          1]));
24    return regex;
25  }
26
27  // 处理最外层的其他字符
28  vector<Regex *> concats;
29  // 对最外层按照拼接操作划分正则表达式
30  for (int i = l; i < r; i++) {
31    if (str[i] == '*') { // 处理"*"号
32      Regex *back = concats.back();
33      concats.pop_back();
34      concats.push_back(new Regex(CLOSURE, NULL, back, NULL));
35    } else if (str[i] == '(') { // 处理"("和")"
36      int level = 0;
37      for (int j = i; j < r; ++j)
38      if (str[j] == '(')
39      ++level;
40      else if (str[j] == ')' && --level == 0) {
41        concats.push_back(strToRegex(str, i + 1, j));
42        i = j;
43        break;
44      }
45    } else { // 处理其他字符
46      concats.push_back(new Regex(CHARR, str[i], NULL, NULL));
47    }
48  }
49
50  // 对最外层的拼接生成语法树
51  int size = concats.size();
52  Regex *regex = concats.front();
53  for (int i = 1; i != size; ++i)
```

```
    regex = new Regex(CONCAT, NULL, regex, concats[i]);
    return regex;
}
```

1.2 生成 NFA

1. **NFA 节点**

NFA 节点有四个成员变量，分别为 id: 节点编号、eps_closure: 当前节点的 eps 闭包、trans_func: 当前节点的转换函数、accept: 是否为接受状态。通过 trans_func 将各个节点连接形成 DFN，trans_func 记录则当前节点在输入字符 c 时会转移到哪个节点（通过编号指定）;通过 eps_closure 记录当前节点的 eps 闭包（存储对应结点的编号），方便生成 NFA 节点。

```
\item \textbf{Dfn节点}
class NfaNode
{
  public:
  int id;                      // 节点编号
  vector<int> eps_closure;     // 当前节点的eps闭包
  map<char, int> trans_func; // 当前节点的转换函数
  bool accept = false;         // 是否为接受状态

  NfaNode(); // 构造函数
};
```

2. **NFA**

NFA 有三个成员变量，分别是 begin：开始节点指针、end：结束节点指针、nfaNodes：存储所有的 NFA 节点。其中，nfaNodes 使用 vector 存储 NFA 结点的指针，并且是静态变量，存储该类的全部 NFa 结点。

```
class Nfa
{
  public:
  NfaNode *begin = nullptr;           // 开始节点
  NfaNode *end = nullptr;             // 结束节点
  static vector<NfaNode *> nfaNodes; // 静态变量，存储所有Nfa节点

  Nfa(){};
  Nfa(const Regex *);
};
```

3. **NFA 构造函数**

该构造函数以正则表达式语法树的根节点作为输入，递归构造 NFA。首先，该函数递归构造左右子树的 NFA，当不存在左右子树时相当于调用默认构造函数。然后根据当前语法树节点的类型分别处理，生成对应的 NFA，通过 trans_func 记录节点之间的转移函数，通过 eps_closure 计算节点的 eps 闭包，并不断传递 accept 参数，以识别接受状态。

执行完该函数之后，会成功构建 NFA。它的成员变量 begin 指示了开始状态，end 指示了接受状态，而在 nfaNodes 中存储了 NFA 的所有节点。我们可以从开始节点开始，通过结点的 trans_func 连接每个结点，使用 NFA 进行正则表达式的判断。

```
// 构造函数，以正则表达式语法树的根节点作为输入，构造NFA
// 该函数会递归进行构造
Nfa::Nfa(const Regex *regex)
{
  // 先构造左右子树的NFA
  // 当regex为空时相当于调用默认构造函数
  if (regex == NULL) {
    return;
  }
  Nfa nfa0 = Nfa(regex->re0);
  Nfa nfa1 = Nfa(regex->re1);

  // 根据当前语法树节点的类型分别处理，生成对应的NFA
  // 通过trans_func记录节点之间的转移函数
  // 通过eps_closure计算节点的eps闭包
  // 并不断传递accept参数，以识别接受状态
  if (regex->type == CHARR) {
    begin = new NfaNode;
    end = new NfaNode;
    end->accept = true;
    begin->trans_func[regex->c] = end->id;
  } else if (regex->type == CONCAT) {
    begin = nfa0.begin;
    end = nfa1.end;
    nfa0.end->eps_closure.push_back(nfa1.begin->id);
    nfa0.end->accept = false;
  } else if (regex->type == UNION) {
    begin = new NfaNode;
    end = new NfaNode;
    begin->eps_closure.push_back(nfa0.begin->id);
    begin->eps_closure.push_back(nfa1.begin->id);
    nfa0.end->eps_closure.push_back(end->id);
    nfa1.end->eps_closure.push_back(end->id);
    nfa1.end->accept = false;
    nfa0.end->accept = false;
    end->accept = true;
  } else if (regex->type == CLOSURE) {
    begin = new NfaNode;
    end = new NfaNode;
    begin->eps_closure.push_back(end->id);
    begin->eps_closure.push_back(nfa0.begin->id);
    nfa0.end->eps_closure.push_back(end->id);
    nfa0.end->eps_closure.push_back(nfa0.begin->id);
    nfa0.end->accept = false;
    end->accept = true;
```

```
46    }
47 }
```

1.3 生成 DFA

1. **DFA 节点**

   DFA 节点有四个成员变量，分别为 id: 节点编号、id_set: 该 DFA 节点包含了哪些 NFA 节点的编号、trans_func: 当前节点的转换函数、accept: 是否为接受状态。通过 trans_func 将各个节点连接形成 DFN，trans_func 记录着当前节点在输入字符 c 时会转移到哪个节点（通过编号指定）。

```
class DfaNode
{
  public:
  int id;                       // 编号
  set<int> id_set;              // 该DFA节点包含了哪些NFA节点的编号
  map<char, int> trans_func;    // 转移函数
  bool accept = false;          // 是否为接受状态

  DfaNode(){};
  DfaNode(set<int> id_set, bool accept);
};
```

2. **DFA**

   DFA 节点有三个成员变量，分别为 begin: 开始节点的编号、charSet: 字符集、dfaNodes: 存储所有的 DFA 节点。其中，dfaNodes 使用 vector 存储 DFA 结点的指针，并且是静态变量，存储该类的全部 DFA 结点。charSet 用于记录正则表达式出现的字符，以用于之后生成状态转移矩阵。

```
class Dfa
{
  public:
  int begin;                            // 开始节点的编号
  set<char> charSet;                    // 字符集
  static vector<DfaNode *> dfaNodes; //
      静态变量，存储所有Dfa节点,开始节点为dfaNodes[0]
  Dfa(const Nfa &nfa);
  void simplify();
  void showTransMatrix();
  void test();

  private:
  static pair<DfaNode *, bool> makeNode(const set<int> &, bool);
};
```

3. **makeNode**

该函数输入 dfn 的 id 集合和是否是接受状态, 返回构建的 dfn 结点指针和是否新建结点。为实现该函数，首先查找 dfaNodes，当存在相同的 id_set 时直接返回这个结点；否则新建结点并返回。另外还要返回是否是新建的节点。

```
// 用于构建DFA节点
// 输入dfn的id集合和是否是接受状态, 返回构建的dfn结点指针和是否新建结点
pair<DfaNode *, bool> Dfa::v(const set<int> &id_set, bool accept)
{
  // 当该状态已存在时, 返回该状态
  int size = dfaNodes.size();
  for (int i = 0; i != size; i++) {
    if (dfaNodes[i]->id_set == id_set) {
      return {dfaNodes[i], false};
    }
  }

  // 当是新状态时, 创建DfaNode
  DfaNode *new_node = new DfaNode;
  new_node->id = size;
  new_node->id_set = id_set;
  new_node->accept = accept;
  dfaNodes.push_back(new_node);
  return {new_node, true};
}
```

4. **epsClosure**

该函数用于计算编号为 id 的 DFN 节点的 eps 闭包，输入编号 id，输出 eps 闭包集合和是否为终结状态。为实现目标功能，该函数遍历 id 对应结点以及其 eps 转移结点直到找到所有 id 结点能通过 eps 转移得到的结点集合，即 eps 闭包。

```
// 计算编号为id的DFN节点的eps闭包
// 输入编号id, 输出eps闭包集合和是否为终结状态
static pair<set<int>, bool> epsClosure(int id)
{
  // 声明变量
  bool accept = false;
  set<int> new_set;
  queue<int> id_q;

  // 插入求eps闭包的id
  id_q.push(id);
  new_set.insert(id);

  // 求eps闭包
  while (!id_q.empty()) {
    NfaNode *eps_node = Nfa::nfaNodes[id_q.front()];
    id_q.pop();
    accept = accept || eps_node->accept;
```

```
    for (int temp_id : eps_node->eps_closure) {
      if (new_set.find(temp_id) == new_set.end()) {
        new_set.insert(temp_id);
        id_q.push(temp_id);
      }
    }
  }

  return {new_set, accept};
}
```

5. **构造函数**

该函数输入 NFA，构造 DFA。为实现目标功能，该函数首先根据开始节点的 eps 闭包生成 DFA 节点，然后进行循环。在循环中，不断获取队列首元素、获取可导致状态转移的字符集合 sigma，然后求出当前 DFN 节点相对字符 c 的转移状态集合，并根据转移状态集合生成 DFN 节点，如果是新生成的则加入队列，继续求该节点的转移状态。另外，如果集合中有一个结点是接受状态，那么新节点也会是接受状态。最后，当不再有新状态时，循环结束。

```
// 输入NFA，构造DFA
Dfa::Dfa(const Nfa &nfa)
{
  // 根据开始节点的eps闭包生成DFA节点
  pair<set<int>, bool> eps_ret = epsClosure(nfa.begin->id);
  DfaNode *dfnNode = Dfa::makeNode(eps_ret.first, eps_ret.second).first;
  queue<DfaNode *> dfn_q;
  dfn_q.push(dfnNode);

  // 不断循环，直到不再有新状态
  while (!dfn_q.empty()) {
    // 获取队列首元素
    DfaNode *dfnNode = dfn_q.front();
    dfn_q.pop();

    // 获取可导致状态转移的字符集合sigma
    // 集合可去重
    set<char> sigma;
    for (int nfa_id : dfnNode->id_set)
    for (pair<char, int> trans_func : Nfa::nfaNodes[nfa_id]->trans_func)
    sigma.insert(trans_func.first);
    // 在成员变量中记录所有的转移字符
    charSet.insert(sigma.begin(), sigma.end());

    // 构成新的dfnNode
    for (char c : sigma) {
      set<int> id_set;
      bool accept = false;

      // 求出当前DFN节点相对字符c的转移状态集合
```

```
        for (int nfa_id : dfnNode->id_set) {
          NfaNode *nfaNode = Nfa::nfaNodes[nfa_id];
          auto found = nfaNode->trans_func.find(c); // 查找对c的转移函数
          if (found != nfaNode->trans_func.end()) { // 如果有对c的转移函数
            // 求出转移之后的闭包并插入id_set
            pair<set<int>, bool> eps_ret = epsClosure(found->second);
            id_set.insert(eps_ret.first.begin(), eps_ret.first.end());
            // 只要有一个是终结状态那么就是终结状态
            accept = accept || eps_ret.second;
          }
        }

        // 根据转移状态集合生成DFN节点，如果是新生成的则加入队列，继续求该节点的转移状态
        pair<DfaNode *, bool> ret = Dfa::makeNode(id_set, accept);
        DfaNode *new_dfnNode = ret.first;
        dfnNode->trans_func[c] = new_dfnNode->id;
        if (ret.second) {
          dfn_q.push(new_dfnNode);
        }
      }
    }

    // 将开始节点编号设置为0
    begin = 0;
}
```

1.4 化简 DFA

在 DFA 类中实现成员函数 simplify，用于化简 DFA。该函数首先构造 tempRefs 存储原 DFA 节点到化简状态的映射关系，然后根据 accept 参数，将原节点分为接受的和非接受的，并存储映射关系。然后不断尝试切分状态，直到不能切分。

在切分的过程中，使用 map<map<char, int>, set<int» Map; 进行是否状态能切分的判断，Map 中 map<char, int> 表示状态转移的映射，set<int> 表示符合该映射的 DFN 节点 id 集合，Map 的 size 为几，就说明该状态有几个不同的映射，也就能分出几个新状态。如果 size 为 1，说明该状态已不可再分，直接开始下一个状态；否则根据 Map 中的元素新建结点，并再遍历新状态一次，以分解新状态。

最后，更新 id 属性，每个化简之后在状态的 id 等于其下标；更新 trans_func 属性，遍历原始状态，更新其对应新状态的转移函数；更新 dfaNodes 并更新开始节点下标。

```
// 进行化简
void Dfa::simplify()
{
  // tempRefs存储原DFA节点到化简状态的映射关系
  auto size = dfaNodes.size();
  vector<DfaNode *> tempRefs(size);

  // 首先将节点分为接受的和非接受的
```

```
9   // tempSimDfaNodes用于临时存储化简状态
10  vector<DfaNode *> tempSimDfaNodes;
11  tempSimDfaNodes.reserve(dfaNodes.size());
12  // 使用默认构造函数初始化接受和非接受状态
13  tempSimDfaNodes.push_back(new DfaNode);
14  tempSimDfaNodes.push_back(new DfaNode);
15  auto yes = tempSimDfaNodes.begin();
16  auto no = ++tempSimDfaNodes.begin();
17  (*yes)->accept = true;
18  (*no)->accept = false;
19  // 根据accept参数将每个节点加入到yes或no，并存储映射关系
20  for (int i = 0; i != size; ++i)
21  if (dfaNodes[i]->accept) {
22    (*yes)->id_set.insert(i);
23    tempRefs[i] = *yes;
24  } else {
25    (*no)->id_set.insert(i);
26    tempRefs[i] = *no;
27  }
28  // 如果没有非接受状态，则删除
29  if ((*no)->id_set.empty())
30  tempSimDfaNodes.erase(no);
31
32  // 不断尝试切分状态，直到不能切分
33  while (true) {
34    bool ok = true;
35    for (auto iter = tempSimDfaNodes.begin(); iter != tempSimDfaNodes.end();) {
36      // Map中map<char, int>表示状态转移的映射，set<int>表示符合该映射的DFN节点id集合
37      // Map的size为几，就说明该状态有几个不同的映射，也就能分出几个新状态
38      map<map<char, int>, set<int>> Map;
39      for (int id : (*iter)->id_set) {
40        map<char, int> trans_map;
41        for (pair<char, int> trans_func : Dfa::dfaNodes[id]->trans_func)
42        trans_map.insert({trans_func.first, tempRefs[trans_func.second]->id});
43        Map[trans_map].insert(id);
44      }
45
46      // size为1，说明该状态已不可再分，直接开始下一个状态
47      if (Map.size() == 1) {
48        ++iter;
49        continue;
50      }
51
52      //根据Map中的元素新建结点
53      for (pair<map<char, int>, set<int>> pair : Map) {
54        // 在tempSimDfaNodes中新插入一个状态，并得到该状态的迭代器
55        tempSimDfaNodes.push_back(new DfaNode);
56        DfaNode *newNode = *(--tempSimDfaNodes.end());
57
```

```
58            // 设置节点集合和是否为接受状态属性
59            newNode->id_set = pair.second;
60            newNode->accept = (*iter)->accept;
61
62            // 更新原始DFN节点到新DFN节点的映射
63            for (int id : newNode->id_set)
64            tempRefs[id] = newNode;
65          }
66
67          // 如果有某个状态可再分，就会再遍历新状态一次，以分解新状态
68          ok = false;
69          iter = tempSimDfaNodes.erase(iter);
70        }
71
72        // 如果没有任何新状态，则循环终止
73        if (ok)
74        break;
75      }
76
77      // 更新id属性，每个化简之后在状态的id等于其下标
78      int newSize = tempSimDfaNodes.size();
79      for (int i = 0; i != newSize; ++i) {
80        tempSimDfaNodes[i]->id = i;
81      }
82
83      // 更新trans_func属性
84      // 遍历原始状态，更新其对应新状态的转移函数
85      for (int i = 0; i != size; ++i) {
86        map<char, int> &trans_func = tempRefs[i]->trans_func;
87        for (pair<char, int> kv : dfaNodes[i]->trans_func)
88        trans_func.insert({kv.first, tempRefs[kv.second]->id});
89      }
90
91      // 更新dfaNodes
92      for (DfaNode *item : dfaNodes) {
93        delete item;
94      }
95      dfaNodes.clear();
96      swap(dfaNodes, tempSimDfaNodes);
97
98      // 更新开始节点下标
99      begin = tempRefs[begin]->id;
100   }
```

2 结果验证

2.1 输出及测试

1. 输出转移函数矩阵

该函数输出状态转移矩阵，如果是开始状态则在之后输出一个^, 如果是接受状态则在之后输出一个 *。通过 dfaNodes 中结点的 trans_func 构造该矩阵。

```
// 输出转移矩阵
void Dfa::showTransMatrix()
{
  // 输出第一行
  printf("The State Transition Matrix is as Follow:\n\t");
  for (char charr : charSet) {
    printf("%c\t", charr);
  }
  printf("\n");

  for (int i = 0; i < dfaNodes.size(); ++i) {
    DfaNode *dfaNode = dfaNodes[i];
    // 开始状态在之后输出一个^, 接受状态在之后输出一个*
    const char *format = ((i == begin && dfaNode->accept) ? "%d^*\t" : (i == begin ?
        "%d^\t" : (dfaNode->accept ? "%d*\t" : "%d\t")));
    printf(format, dfaNode->id);
    for (char charr : charSet) {
      auto found = dfaNode->trans_func.find(charr);
      if (found != dfaNode->trans_func.end()) {
        printf(format, dfaNode->trans_func[charr]);
      } else {
        printf("-1\t");
      }
    }
    printf("\n");
  }
}
```

2. 进行测试

测试函数不断获取输入，并根据 trans_func 进行转移，直到没有对应的转移函数或者字符串长度为 0. 输入\q 会退出该程序。

```
// 测试函数
void Dfa::test()
{
  printf("\nPlease Enter String to be Determined:\n");
  char str[100];
  while (true) {
    scanf("%s", str);

```

```
    // 输入\q时退出
    if (str[0] == '\\' && str[1] == 'q') {
      break;
    }

    // 根据当前字符和当前状态的trans_func进行有限状态机的判断
    DfaNode *nowNode = dfaNodes[begin];
    int i = 0;
    for (; str[i] != '\0'; ++i) {
      auto found = nowNode->trans_func.find(str[i]);
      if (found != nowNode->trans_func.end()) {
        nowNode = dfaNodes[found->second];
      } else {
        printf("False\n");
        break;
      }
    }
    if (str[i] == '\0') {
      printf("True\n");
    }
  }
}
```

3. 主函数

在主函数中，首先输入正则表达式字符串，然后构造正则表达式语法树、构造 NFA、构造 DFA、进行化简、输出状态转移矩阵和进行测试。

```
int main()
{
  //输入正则表达式字符串
  char input[100];
  cin >> input;

  //执行词法分析核心构造算法
  Regex *regex = strToRegex(input, 0, strlen(input));//构造正则表达式语法树
  Nfa nfa = Nfa(regex);//构造NFA
  Dfa dfa = Dfa(nfa);//构造DFA
  dfa.simplify();//进行化简

  //进行测试与验证
  dfa.showTransMatrix();//输出状态转移矩阵
  dfa.test();//进行测试
}
```

2.2 进行实验

```
(a*(b|ba))
The State Transition Matrix is as Follow:
        a       b
0^      0^      2^
1*      -1      -1
2*      1*      -1

Please Enter String to be Determined:
aaaaaaaaa
True
aaaaab
True
aaaaaba
True
aaabaa
False
aaalda
False
\q
```

图 2.1: 进行实验

在上图中，我们以 (a*(b|ba)) 作为正则表达式，包含了括号、闭包、拼接、或运算等运算。执行程序，输出了转移矩阵，与手工构造的一致。并且在之后的测试中也全部正确进行了正则表达式的判断，最后输入\q 结束程序。以上表明本程序实现正确。

3 总结

在本次实验中，我实现了词法分析器的核心构造算法，首先输入正则表达式字符串，然后构造正则表达式语法树、构造 NFA、构造 DFA、进行化简、输出状态转移矩阵和进行测试。

对于正则表达式语法树，每个树结点有左右子节点指针、类别。字符等成员变量，构成语法树，然后本程序使用根节点代表整棵树，没有实现语法树类；对于 NFA，NFA 节点类有四个成员变量，分别为节点编号当前节点的 eps 闭包、当前节点的转换函数、是否为接受状态。通过转换函数将各个节点连接形成 DFN，记录 eps 闭包方便生成 NFA 节点。然后，实现 NFA 类，存储开始节点和结束节点的指针，以及使用静态成员变量 vector 记录下所有节点；对于 DFA，DFA 节点类有四个成员变量，分别为节点编号、该 DFA 节点包含了哪些 NFA 节点的编号、当前节点的转换函数、是否为接受状态。通过 trans_func 将各个节点连接形成 DFN。然后，实现 DFA 类，存储开始节点的编号、字符集，以及使用静态成员变量 vector 记录下所有节点；对于 DFA 化简，本程序在 DFA 类中实现了成员函数进行化简。

经过本次实验，我对词法分析的各个阶段以及各阶段实现的算法都有了更加深入的理解。