中国地质大学

题　　目:	编译原理实验报告
姓　　名:	常文瀚
院　　系:	计算机学院
班　　级:	191181
学　　号:	20181001095

2020 年 12 月 1 日

目录

一、实验题目

算术表达式文法的词法分析部分。

二、问题描述

针对下面文法 G(S)：

S→ v = E

E→E+E | E-E | E*E | E/E | （E） | v | i

其中，v 为标识符，i 为整型或实型数。要求使用正规式等技术实现一个词法分析程序。

三、基本要求

小组分工实现如下功能：

1. 文法→NFA
2. 有限自动机的确定化
3. 确定有限自动机的最小化
4. 模拟 DFA

四、小组分工

- 刘乐菲：正规表达式→NFA。编写一个模块，输入一个正规表达式，输出与该文法等价的有限自动机。
- 高天翔，常文瀚：有限自动机的确定化及最小化。编写一个模块，将一个非确定有限自动机转换为确定有限自动机并将其最小化。
- 刘宇洁：模拟 DFA。编写一个模块，模拟最少状态的确定有限自动机，判断输入串能否识别。
- 王泽清，苏浩文：程序框架和界面设计，编写整个程序的框架、界面设计、数据结构设计

五、整体设计

1. 数据结构

- FANode 类　//自动机状态结点类

```
class FANode
{
public:
    string name; //状态结点名称
    FANode()
    {
    };
    FANode(string name) { this->name = name; }
 };
```

- FA 类 //自动机类

```
class FA
{
public:
    set<FANode> startSet; //初态集
    set<FANode> endSet; //终态集
    set<FANode> states; //状态集
    map<FANode, map<char, set<FANode>>> f; //nfa 转换函数
    map<FANode, map<char, FANode>> Dtran; //dfa 转换函数
    string a; //输入符号集
    map<set<FANode>, FANode> dfamap; //nfa 状态集与 dfa 结点的映射
    FA()
    {
    };
    FA(set<FANode> startSet, set<FANode> endSet, set<FANode> states, map<FANode,
map<char, set<FANode>>> f)
    {
        this->startSet = startSet;
        this->endSet = endSet;
        this->states = states;
        this->f = f;
```

```
    }
    void createNFA(string in); //根据读入的一条产生式 in，更新 NFA
    set<FANode> eClosure(set<FANode> T); //能够从 T 中的某个状态开始只通过 epsilon 转换到达的状态集合
    set<FANode> move(set<FANode> T, char a); //能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的状态的集合
    void subset(FA& dfa); //用子集法将 FA dfa 构造为 DFA
};
```

2. 文件结构

- LexAnalysis.h //自动机状态结点类，自动机类的建立
- LexAnalysis.cpp
- main.cpp //词法分析主程序，包括读入文法，构造 NFA，NFA 确定化等

3. 基本思想

本实验采用的基本思想是根据 C++语言的基本状态集，根据 txt 文件给出的文法同时利用 c++的封装类 map、set 以及 pair 等实现轻量化构造 NFA 与 DFA，将自动机存储在定义的类成员中。

1 文法产生式来源采用文件导入的方式，对输入文法的文件进行逐行的读入构造非确定有限自动机。
2 然后利用子集法将一个非确定有限自动机转换为确定有限自动机。
3 最后利用分割法解决自动机最小化问题，从而得到用来识别输入串的最少状态的确定有限自动机。
4 对于输入字符串支持对标识符、关键字、整数、浮点数、分界符、运算符等识别。对于读取的每一行代码保存在字符数组中。识别每一个单词依赖于其相应的词法分析的状态转换图。

编译实际上就是一种对模式的识别，利用编译的原理我们能够解决任何的关于识别的问题。任何一种模式都肯定由一种明确的文法进行规定，那么我们就能够根据这种文法进行模式的识别，而作为模式识别，词法分析也遵循同样的规则。正规文法与有限自动机具有等价性，所以，词法分析我们就以有限自动机作为识别的主要驱动。

4.文法表示

数据结构：长度固定为 5 和 4 的字符串

1 文法：（VN,VT,P,S）

Ø 初状态设置为 S，非终结符 VN 包括 S、E、I、C、O、B、W、R、P，终结符 VT 包括 a-z,0-9,+,-,*,/,(,),.。Ø

ØP 中含有的规则为

S->EI	I->o	I->gW	I->xW	W->p	W->7	W->oW	W->6W	C->4C	P->0P	
S->EC	I->p	I->hW	I->yW	W->q	W->8	W->pW	W->7W	C->5C	P->1P	
S->EO	I->q	I->iW	I->zW	W->r	W->9	W->qW	W->8W	C->6C	P->2P	
S->EB	I->r	I->jW	W->a	W->s	W->aW	W->rW	W->9W	C->7C	P->3P	P->8
I->a	I->s	I->kW	W->b	W->t	W->bW	W->sW	C->EL	C->8C	P->4P	
I->b	I->t	I->lW	W->c	W->u	W->cW	W->tW	C->0	C->9C	P->5P	P->9
I->c	I->u	I->mW	W->d	W->v	W->dW	W->uW	C->1	C->0C	P->6P	O->+
I->d	I->v	I->nW	W->e	W->w	W->eW	W->vW	C->2	C->0R	P->7P	
I->e	I->w	I->oW	W->f	W->x	W->fW	W->wW	C->3	C->1R	P->8P	O->-
I->f	I->x	I->pW	W->g	W->y	W->gW	W->xW	C->4	C->2R	P->9P	O->*
I->g	I->y	I->qW	W->h	W->z	W->hW	W->yW	C->5	C->3R	P->0	
I->h	I->z	I->rW	W->i	W->0	W->iW	W->zW	C->6	C->4R	P->1	O->/
I->i	I->aW	I->sW	W->j	W->1	W->jW	W->0W	C->7	C->5R	P->2	O->=
I->j	I->bW	I->tW	W->k	W->2	W->kW	W->1W	C->8	C->6R	P->3	
I->k	I->cW	I->uW	W->l	W->3	W->lW	W->2W	C->9	C->7R	P->4	E->=
I->l	I->dW	I->vW	W->m	W->4	W->mW	W->3W	C->1C	C->8R	P->5	B->(
I->m	I->eW	I->wW	W->n	W->5	W->nW	W->4W	C->2C	C->9R	P->6	B->)
I->n	I->fW	I->xW	W->o	W->6	W->oW	W->5W	C->3C	R->.P	P->7	

Ø 其中 I 和 C 可表示单个标识符序列、多个标识符组成的序列、标识符和数字混合组成的序列等等，O 表示运算符，E 表示等于号，则从 S 推导可从而形成表达式的文法。

5. 文法转化 NFA

1、 核心：

读入一行规则，将规则的左部存到状态 1，若是 5 个字符，则右部组成为输入符号加状态 2；若是 4 个字符，则右部为输入字符加末状态，从而形成 NFC 的一条路径。

2、 数据结构：

FANode 类储存节点名称，set<FANode>stetes 储存的所有状态，map<FANode, map<char, set<FANode>>> f 为 NFA，它储存的是：state1-path-state2。

3、具体流程：

a）读入一条规则，初状态设置为 S

b）输入字符串长度为 5，将所在字符串 0 和 4 位置的状态存入 states 中，将所在字符串 3 位置的输入字符和 state1 一起作为 f 的键值，并将其对应的 state2 存储在键值所在的地方，形成 NFC 的一条路径

c） 输入字符为长度为 4，输出 state1-path-末状态

d）读完所有规则，得到 NFA f。

6. 有限自动机的确定化

根据课本中介绍的**子集构造法**，重点是要在代码中完成以下两个函数：

1、E_closure(T)：

T 为 NFA 状态集合，找到 T 集合的每个 E 转换的集合。这里需要递归，我的做法是新建一个递归栈 S，R，S 初值是 T 集合的值，R 栈为空。每个状态通过 E 转换获得的状态加入到递归栈 S、R 中，并把该状态从 S 中出栈。这样直到递归栈为空结束循环。最后返回结果栈 R。

2、 Move(T,a)：

T 为 NFA 状态集合，a 为转换条件。初始化一个结果栈 R，初始化一个栈 S，S 初始值和 T 集合值相同。每次进行一次 a 转换，就将该状态出栈，并把转换结果压入 R 栈中。直到 S 为空，结束循环。最后返回结果栈 R。

3、在整体的 getDFA()函数中，（DFA 状态有两个信息，一个是节点序号，一个是对于的 NFA 集合）。

1） 我们设置一个 DFA 状态栈 S，当这个栈为空的时候，结束循环。

2） 首先求出 NFA 初始状态的 e_closure 的 E 转换集合，记为 DFA 的初始状态 A。并把该状态加入到 DFA 状态栈 S 中。并把该状态加入到 DFA 的节点集合中。

3）从初始状态开始，循环字母表，字母表的每个元素都是转换条件。

i. 对每个转换条件，执行 e_closure(move(<当前 DFA 状态>,<当前字母表>))，会得到一个新生成的 NFA 状态集合

ii. 检查该 NFA 状态集合在已有的 DFA 状态集合中是否已经存在。

iii. 如果不存在，就把该新生成的 DFA 状态加入到 S 栈中，把该状态加入到 DFA 的节点集合中，并把原来的 DFA 状态出栈。

iv. 并根据转换条件，给 DFA 的图中把对应的两个 DFA 节点连接起来。

7. 最小化 DFA

根据课本上面的算法可以直接写出代码：

1. 定义一个划分集合 dividedArrays，集合中初始有两个元素，第一个元素是非终止状态集合，第二个元素是终止状态集合。向划分集合中加入元素的时候，每个元素都会被标记为 false，表示可划分。
2. 定义一个循环变量 flag。默认为 true，当值为 false 时候结束循环。当且仅当划分集合的每个元素都不可划分，即每个集合都被标记为 true，此时退出大循环。
3. 对该划分集合的每个元素遍历：
   a) 每个元素也是一个状态集合 S，对该集合 S 的每个元素进行字母表的转换循环。通过获取 S 集合中的每个元素经过某个字母转换的结果状态属于划分集合的序号，最后根据属

于划分集合的序号将该状态集合进行划分。

b) 如果某个状态集合 S 的所有元素经过所有字母转换的结果都属于划分集合的同一个元素，说明该集合不可划分，将该集合标记为 true。

4. 最后根据划分集合的每个元素是否可继续划分，如果都已经被标记为 true，退出循环
5. 最后，循环扫描划分集合的每个元素，如果某个元素的大小大于 1，则将该元素内的状态都合并成同一个状态。具体做法就是将需要被合并的节点的所有边的状态都移到目标节点中。

8. 模拟 DFA

1、核心：模拟最少状态的确定有限自动机，判断输入串能否识别。

2、具体流程：

a）读入文件，遍历文件每行。

b）遍历该行的每个字符。

c）if 字符为空，判断当前状态是否为终态，不是则报错，是则输出 type 和 token。初始化 state 和 path 并开始读取下一个字符。

d）if 当前状态是终态，判断读入字符能否使状态转换为另一个状态，能则更新；否则将终态输出并初始化 state 和 path。

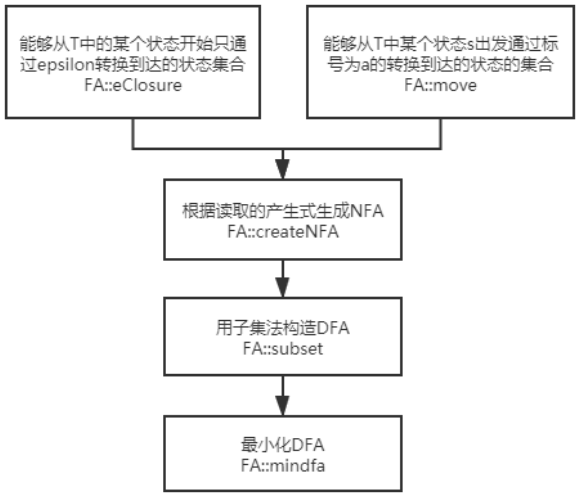
e）if 读入状态不是终态，判断当前读入字符能否使状态转换为另一个状态，能则更新状态，否则抛出错误。

f）处理该行最后的 token 状态，不是终态则抛出错误，是终态则进行输出。

g）初始化 state 和 path 并读取下一行。

六、自己负责的模块

1、 函数调用关系图



2、 模块结构说明

（1）构造函数

FA(set<FANode> startSet, set<FANode> endSet, set<FANode> states, map<FANode, map<char, set<FANode> > > f)

（2）更新并生成 NFA

void createNFA(string in)

（3）子集法构造 DFA

void subset(FA& dfa)

（4）分劈法最小化 DFA

void mindfa(FA& dfa)

3、 函数的功能实现

```
//根据读入的一条产生式 in，更新 NFA
void createNFA(string in);
//能够从 T 中的某个状态开始只通过 epsilon 转换到达的状态集合
set<FANode> eClosure(set<FANode> T);
//能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的状态的集合
set<FANode> move(set<FANode> T, char a);
//用子集法将 FA dfa 构造为 DFA
void subset(FA& dfa);
//判断 s 是否为关键字
bool isKey(string s);
//判断 s 是否为类型标识符
bool isType(string s);
//判断 s 是否为运算符
bool isOp(string s);
//判断 s 是否为界符
bool isBorder(string s);
//将输入字符 c 转换为适合词法分析的符号
char ex(char c);
//将两 FANode 集合 a，b 合并
set<FANode> setunion(set<FANode> a, set<FANode> b);
//判断两 FANode 集合 a，b 是否相等
```

```
bool equal(set<FANode> a, set<FANode> b);
//判断某 FANode 集合 u 是否在队列 q 中
bool find(queue<set<FANode> > q, set<FANode> u);
//判断字符串 s 的 token 类型
string judge(FANode node, string s);
```

4、有限自动机的确定化

DFA 的最小化：

1. 定义一个划分集合 dividedArrays，集合中初始有两个元素，第一个元素是非终止状态集合，第二个元素是终止状态集合。向划分集合中加入元素的时候，每个元素都会被标记为 false，表示可划分。
2. 定义一个循环变量 flag。默认为 true，当值为 false 时候结束循环。当且仅当划分集合的每个元素都不可划分，即每个集合都被标记为 true，此时退出大循环。
3. 对该划分集合的每个元素遍历：
   a) 每个元素也是一个状态集合 S，对该集合 S 的每个元素进行字母表的转换循环。通过获取 S 集合中的每个元素经过某个字母转换的结果状态属于划分集合的序号，最后根据属于划分集合的序号将该状态集合进行划分。
   b) 如果某个状态集合 S 的所有元素经过所有字母转换的结果都属于划分集合的同一个元素，说明该集合不可划分，将该集合标记为 true。
4. 最后根据划分集合的每个元素是否可继续划分，如果都已经被标记为 true，退出循环
5. 最后，循环扫描划分集合的每个元素，如果某个元素的大小大于 1，则将该元素内的状态都合并成同一个状态。具体做法就是将需要被合并的节点的所有边的状态都移到目标节点中。

七、算法设计

自动机关键类函数实现：

①词法分析：void LexAnalyse()

```
{
    //读入正规文法
    string in;
    ifstream ifile_reg;
    ifile_reg.open("reg.txt", ios::in);

    //根据读入产生式构造 NFA
```

```
FA fa = FA();
FANode nfaend = FANode("@"); //终态用"@"表示
fa.endSet.insert(nfaend);
while (ifile_reg)
{
    ifile_reg >> in;
    fa.createNFA(in);
}

//NFA 转换为 DFA
FA dfa = FA();
fa.subset(dfa);

//读入源代码，进行分析
ifstream ifile_code;
ifile_code.open("code.txt", ios::in);
string path;
FANode start = *dfa.startSet.begin();

//输出 token 序列
ofstream ofile_token;
ofile_token.open("token.txt", ios::out);

cout << "type" << "\t\ttoken" << endl;

while (getline(ifile_code, in))
{
    in = in.substr(in.find_first_not_of(" "));
    FANode state = start;
    for (int i = 0; i < in.length(); i++)
```

```
{
    char c = ex(in[i]);
    if (c == ' ')
    {
        if (dfa.endSet.count(state) == 0)
        {
            cout << "error   " << "\t" << path << endl;
            return;
        }
        else
        {
            cout << judge(state, path) << "\t" << path << endl;
            ofile_token << judge(state, path) << endl;
        }
        state = start;
        path = "";
        continue;
    }
    if (dfa.endSet.count(state) == 1)
    {
        if (dfa.Dtran[state].find(c) != dfa.Dtran[state].end())
        {
            state = dfa.Dtran[state][c];
            path += in[i];
        }
        else
        {
            cout << judge(state, path) << "\t" << path << endl;
            ofile_token << judge(state, path) << endl;
            state = start;
```

```
                path = "";
                i--;
            }
            continue;
        }

        path += in[i];
        if (dfa.Dtran[state].find(c) == dfa.Dtran[state].end())
        {
            cout << "error   " << "\t" << path << endl;
            state = start;
            path = "";
            return;
        }
        else state = dfa.Dtran[state][c];
    }

    if (dfa.endSet.count(state) == 0)
    {
        cout << "error   " << "\t" << path << endl;
        return;
    }
    else
    {
        cout << judge(state, path) << "\t" << path << endl;
        ofile_token << judge(state, path) << endl;
    }
    state = start;
    path = "";
}
```

```
    ofile_token << "#" << endl;
}
```

②DFA 构造：void FA::createNFA(string in)

```
{
    string name;
    char path;
    string target;
    int len = in.length();
    if (len == 5)
    {
        name = in[0];
        path = in[3];
        target = in[4];
        FANode node1 = FANode(name);
        FANode node2 = FANode(target);
        this->states.insert(node1);
        this->states.insert(node2);
        if (name == "S") this->startSet.insert(node1);
        this->f[node1][path].insert(node2);
    }
    else
    {
        name = in[0];
        path = in[3];
        FANode node1 = FANode(name);
        this->states.insert(node1);
        this->f[node1][path] = setunion(this->f[node1][path], this->endSet);
    }
    if (path != 'E' && a.find(path) == string::npos) a += path;
}
```

③用子集法将 FA DFA 构造为 DFA：void FA::subset(FA& dfa)

```
{
    //一开始 eclosure(s0)是 Dstates 中的唯一状态，且未被标记
    queue<set<FANode>> Dstates0; //表示未标记
    queue<set<FANode>> Dstates1; //表示已标记
    set<FANode> ecs0 = this->eClosure(this->startSet);
    Dstates0.push(ecs0);
    //构造 dfa 的初始状态结点
    FANode dfastart = FANode("0");
    dfa.startSet.insert(dfastart);
    dfa.states.insert(dfastart);
    int cnt = 1; //用以表示结点名称
    queue<FANode> nodeque; //用来保存已构造的 dfa 结点
    nodeque.push(dfastart);
    //当 Dstates 中有一个未标记状态 T
    while (!Dstates0.empty())
    {
        //给 T 加上标记
        set<FANode> T = Dstates0.front();
        Dstates1.push(T);
        Dstates0.pop();
        FANode pre = nodeque.front(); //暂存由状态 T 构造的 dfa 结点
        nodeque.pop();

        //对每个输入符号
        int len = a.length();
        for (int i = 0; i <= len - 1; i++)
        {
            set<FANode> U = eClosure(move(T, a[i]));
            //若 U 不在 Dstates 中
```

```
            if (!find(Dstates0, U) && !find(Dstates1, U))
            {
                //将 U 加入到 Dstates 中且不标记
                Dstates0.push(U);
                //同时由状态 U 构造 dfa 结点
                string name = to_string(cnt);
                FANode node = FANode(name);
                this->dfamap[U] = node;
                cnt++;
                nodeque.push(node);
                dfa.states.insert(node);
                //如果状态 U 包含终态结点，则将新构造的结点加入 dfa 的终态集
                set<FANode>::iterator it;
                set<FANode> e = this->endSet;
                for (it = e.begin(); it != e.end(); it++)
                {
                    if (U.find(*it) != U.end())
                    {
                        dfa.endSet.insert(node);
                        break;
                    }
                }
            }

            //更新 dfa 的转换函数
            if (!U.empty()) dfa.Dtran[pre][a[i]] = this->dfamap[U];
        }
    }
}
```

④分割法实现 DFA 最小化 void FA::mindfa(FA& dfa)

```
 {
    set<FANode> NoEndState; //非终态集
    set_difference(dfa.states.begin(), dfa.states.end(), dfa.endSet.begin(),
dfa.endSet.end(), inserter(NoEndState, NoEndState.begin()));
    FANode minname = *dfa.startSet.begin();
    for (set<FANode>::iterator it = NoEndState.begin(); it != NoEndState.end(); it++)
    {
        cout << it->name << endl;
        int key = stoi(it->name);
        if (dfa.Dtran.count(it->name) > 0)
        {
            cout << "通过 key:  " << it->name << endl;
        }
        else {
            minname.name = it->name;
        }
    }
    NoEndState.erase(minname);
    for (set<FANode>::iterator it = NoEndState.begin(); it != NoEndState.end(); it++)
    {
        cout << it->name << endl;
    }
    //遍历终态集，找到指向了非终态集的节点；
    map<FANode, map<char, FANode> >::iterator iter;
    map<FANode, map<char, FANode> > FINAL_end;  //最后的终态
    map<char, FANode> wait2;
    map<char, FANode> wait3;    //wait1 和 wait2 汇合成新的 map<FANode, map<char,
FANode> >
    FANode wait1 = *dfa.endSet.end();   //10
    FANode wait4;
```

```
    auto we = dfa.Dtran;
    for (iter = we.begin(); iter != we.end(); iter++) {
        map<char, FANode>::iterator it;
        auto wr = iter->second;
        for (it = wr.begin(); it != wr.end(); it++)
        {
            //set< FANode >::iterator i = NoEndState.find();
            if (it->second.name == "0" || it->second.name == "9") {
                // do something with *it
                cout << iter->first.name << endl;
                wait4 = iter->first;
                wait2.insert(wr.begin(), wr.end());
            }
            else {
                wait3.insert(wr.begin(), wr.end());
            }
        }
    }
}
```

八、调试分析

8.1 运行结果

8.2 优点分析

- 模块化设计，便于功能得扩展以及函数得复用；
- 通过读取文件获取文法，便于文法的修改和扩展；
- 调用 c++的封装类作为数据结构，减少了空间复杂度，充分体现了 c++语言的优越性；
- 除了可以识别简单的算法运算符号，还可以识别指数、实数以及科学记数法符号；

8.3 缺点分析

- 输出效果比较简单，没有中间过程的细节

8.4 改进方法

- 可以对算法进行进一步重构，改善算法的输出格式
- 可以将中间过程的算法运算结果输出在最终结果上方

九、使用方式

该程序的使用方式如下：

1）将所要分析的文法放置在“reg.txt”文件中。

2）将所要分析的句子放置在“code.txt”文件中。

3）运行程序即可。

十、总结

通过编译原理实验课，我掌握了什么是编译程序，编译程序工作的基本过程及其各阶段的基本任务，熟悉了编译程序总流程框图，了解了编译程序的生成过程、构造工具及其相关的技术对课本上的知识有了更深的理解，课本上的知识是机械的，表面的。通过把该算法的内容，算法的执行顺序在计算机上实现，把原来以为很深奥的书本知识变的更为简单，对实验原理有更深的理解。

然而，我认为自己在实验课上做的并不够好。首先，在总体实验内容来说，我没能够完成所有的内容，只完成了词法分析中 NFA 确定化、DFA 最小化的程序;其次，就完成的两个实验来说，功能上还是不够完善，有一些 bug。最后，运行界面上过于简陋，不够美观。由于时间的有限性，这些不足是无法继续改善了。这也让我意识到对于做每一件事，你的付出时间和你的成果是成正比的。所以，对于做一件事，要舍得花时间，肯花时间，这样最后的效果才会更好。对于编译原理，整体看下来自己做得不算好。意识到不足，我更加地明白自己能力不够强，需要在今后的学习中更加深入的理解学习过的知识。

十一、致谢

感谢刘远兴老师对我们的悉心教导以及提供我们这样一个学以致用的机会。通过这一阶段对编译原理课程的学习，特别是通过这次对于词法分析器的编程实践，我对于编译原理中的词法分析这一过程理解的更加清楚明了，收获很大。同时更要感谢小组内成员与我进行积极的沟通，互相帮助解决实验中的难题，没有他们一定不可能顺利完成这次试验。

十二、参考文献

[1]王生原. 编译原理.第 3 版[M]. 清华大学出版社, 2015.

[2]韩光辉. 有限自动机的最小化理论[J]. 江汉大学学报: 自然科学版, 2005, 33(4): 14-16.

[3]熊德兰, 田胜利. DFA 最小化算法的探讨与改进[J]. 计算机教育, 2008 (9): 71-72.

[4]范书义, 孟晨, 王成. 一种新的 DFA 状态最小化算法[J]. 计算机工程与应用, 2012 (2012 年 01): 47-48+ 67.

[5]毛红梅, 聂承启. 一种将 NFA 到最小化 DFA 的方法[J]. 计算机与现代化, 2004 (10): 6-7.