实验一 自定义小语言的编译实验

一、实验内容

**基本要求：**

* 1. 词法分析器 产生下述小语言的单词序列

这个**小语言**的所有的单词符号，以及它们的种别编码和内部值如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 单词符号 | 种别编码 | 助记符 | 内码值 |
| DIM  IF  DO  STOP  END 标识符  常数（整）  =  +  \*  \*\*  ，  （  ） | 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | $DIM $IF $DO $STOP $END $ID $INT $ASSIGN $PLUS $STAR $POWER $COMMA $LPAR $RPAR | -  -  -  -  -  -  内部字符串  标准二进形式  -  -  -  -  -  - |

对于这个**小语言**，有几点重要的限制：

**首先**，所有的关键字（如IF﹑WHILE等）都是“保留字”。所谓的保留字的意思是，用户不得使用它们作为自己定义的标示符。例如，下面的写法是绝对禁止的：

IF（5）=x

**其次**，由于把关键字作为保留字，故可以把关键字作为一类特殊标示符来处理。也就是说，对于关键字不专设对应的转换图。但把它们（及其种别编码）预先安排在一张表格中（此表叫作保留字表）。当转换图识别出一个标识符时，就去查对这张表，确定它是否为一个关键字。

**再次**，如果关键字、标识符和常数之间没有确定的运算符或界符作间隔，则必须至少用一个空白符作间隔（此时，空白符不再是完全没有意义的了）。例如，一个条件语句应写为

　 IF i>0 i= 1;

而绝对不要写成

　　　　　　　　　　　　 　IFi>0 i=1;

因为对于后者，我们的分析器将无条件地将IFI看成一个标识符。

我做的是将后缀表达式转化为NFA的部分。

二、设计思路

1、创建一个NFA类，用于表示非确定有限自动机。该类包含正则表达式的后缀形式、状态数、起点和终点状态编号以及NFA状态关系图等成员变量。

2、在NFA类中定义构造函数、创建NFA函数以及打印NFA函数等等。

3、实现类中定义的函数。

三、实现方法

1、NFA类的定义

class NFA {

string suffix; // 正则表达式的后缀形式

int stateNum; // 状态数

pair<int, int> se; // 起点和终点状态编号

vector<vector<pair<int, char> > > graph; // NFA状态关系图

public:

NFA(string ss);

int newState();

void createNFA();

void printNFA();

};

2、初始化NFA

构造函数接收一个后缀表达式ss作为参数，并将其赋值给成员变量suffix。

同时初始化状态数量为0，并在NFA状态关系图中添加一个空的边集。

NFA::NFA(string ss) {

suffix = ss;

stateNum = 0;

graph.push\_back(vector<pair<int, char> >());

}

3、生成一个新状态

该函数会向NFA状态关系图中添加一个空的边集，并返回生成的新状态的编号。同时，状态数量加1。

int NFA::newState() {

graph.push\_back(vector<pair<int, char> >());

return ++stateNum;

}

4、后缀表达式转为NFA

（1）使用了一个栈作为状态栈states来辅助生成NFA。

（2）遍历后缀表达式的每个字符：

①如果字符不是\*、&或|，表示是一个字符，生成两个新的状态作为起点和终点，并将起点和终点压入栈states。在NFA状态关系图中添加一条从起点到终点的边，标记为该字符。

②如果字符是\*，表示闭包运算。从栈states中弹出栈顶的状态对origin，生成两个新的状态作为起点和终点，并将起点和终点压入栈states。在NFA状态关系图中添加以下边：

从起点到origin.first的空边

从起点到终点的空边

从origin.second到终点的空边

从origin.second到origin.first的空边

③如果字符是&，表示连接运算。从栈states中弹出两个状态对right和left，将left.first作为新的起点，right.second作为新的终点，并在NFA状态关系图中添加一条从left.second到right.first的空边。

④如果字符是|，表示选择运算。从栈states中弹出两个状态对down和up，生成两个新的状态作为起点和终点，并将起点和终点压入栈states。在NFA状态关系图中添加以下边：

从起点到up.first的空边

从起点到down.first的空边

从up.second到终点的空边

从down.second到终点的空边

（3）将栈states中剩余的状态对的第一个状态的编号作为起点，第二个状态的编号作为终点，赋值给成员变量se。

void NFA::createNFA() {

stack<pair<int, int> > states; // 状态栈

int start, end; // 状态边起点和终点状态编号

for (auto c : suffix) {

if (c != '\*' && c != '&' && c != '|') {

start = newState();

end = newState();

states.push(make\_pair(start, end));

graph[start].push\_back(make\_pair(end, c));

continue;

}

switch (c)

{

case '\*': {

pair<int, int> origin = states.top(); states.pop();

start = newState();

end = newState();

states.push(make\_pair(start, end));

graph[start].push\_back(make\_pair(origin.first, ' '));

graph[start].push\_back(make\_pair(end, ' '));

graph[origin.second].push\_back(make\_pair(end, ' '));

graph[origin.second].push\_back(make\_pair(origin.first, ' '));

break;

}

case '&': {

pair<int, int> right = states.top(); states.pop();

pair<int, int> left = states.top(); states.pop();

states.push(make\_pair(left.first, right.second));

graph[left.second].push\_back(make\_pair(right.first, ' '));

break;

}

case '|': {

pair<int, int> down = states.top(); states.pop();

pair<int, int> up = states.top(); states.pop();

start = newState();

end = newState();

states.push(make\_pair(start, end));

graph[start].push\_back(make\_pair(up.first, ' '));

graph[start].push\_back(make\_pair(down.first, ' '));

graph[up.second].push\_back(make\_pair(end, ' '));

graph[down.second].push\_back(make\_pair(end, ' '));

break;

}

default:

break;

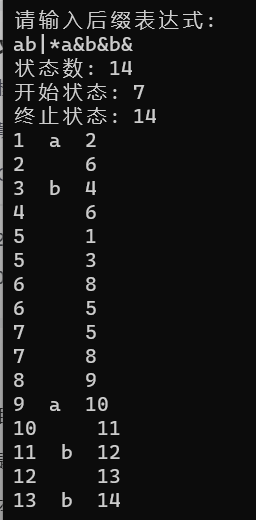
}

}

se = make\_pair(states.top().first, states.top().second);

}

四、测试结果



五、心得体会

这次实验给我带来了丰富的收获和体会，通过这次实验，我深入理解了后缀表达式和NFA之间的联系，学会了将后缀表达式转换为NFA，并对NFA进行测试和验证，通过实际操作提升了自己的算法设计和调试能力。