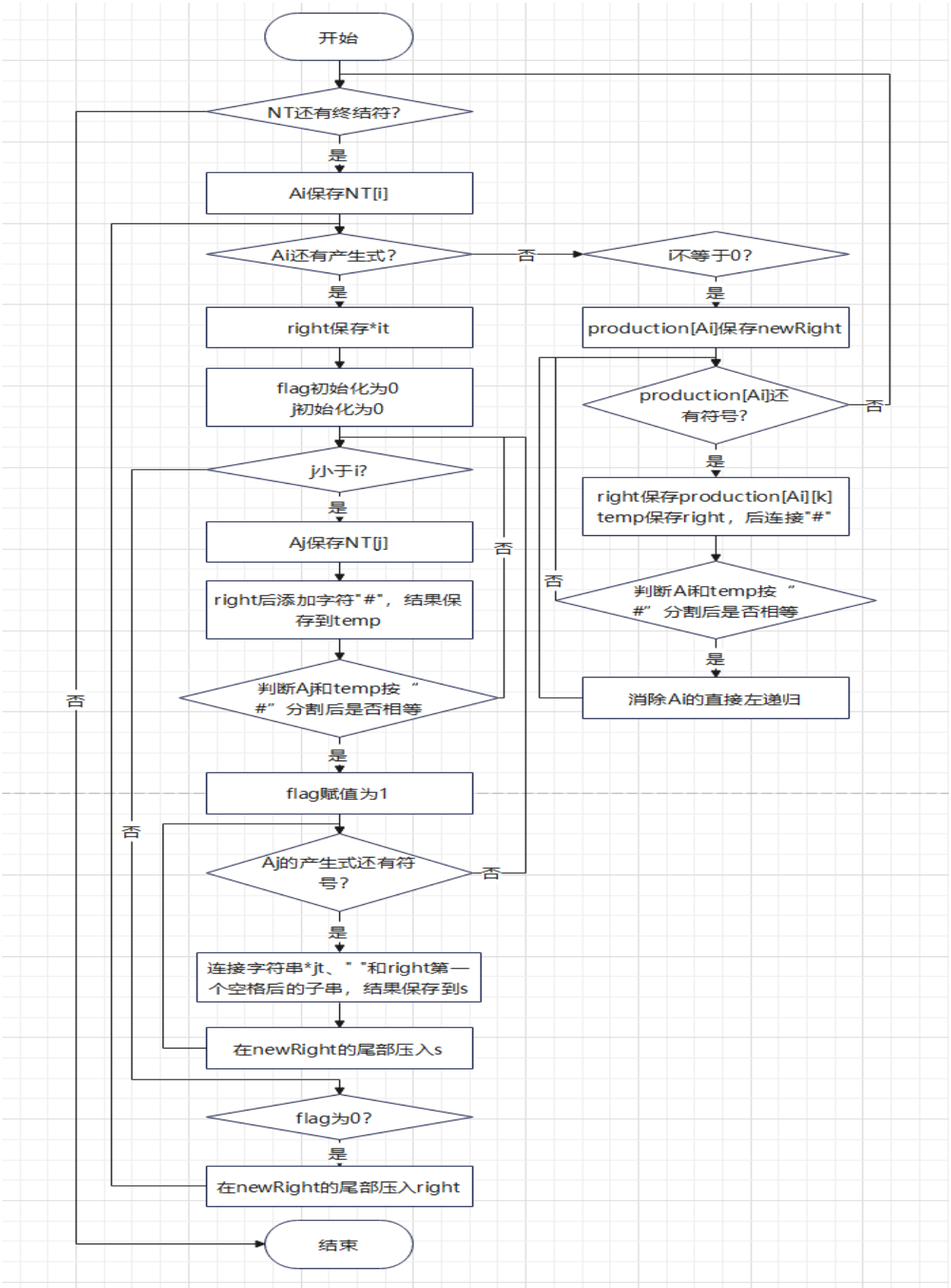
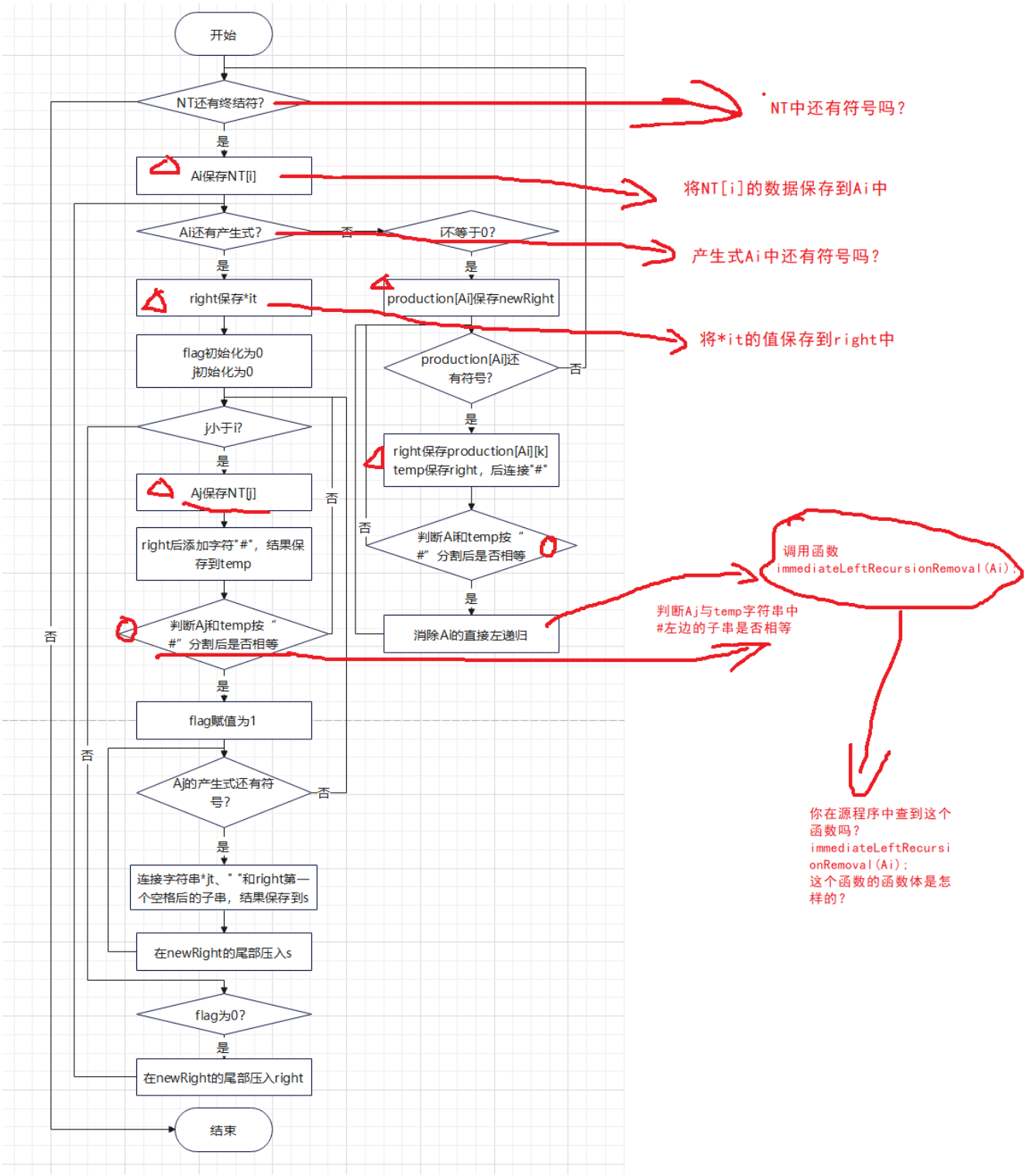
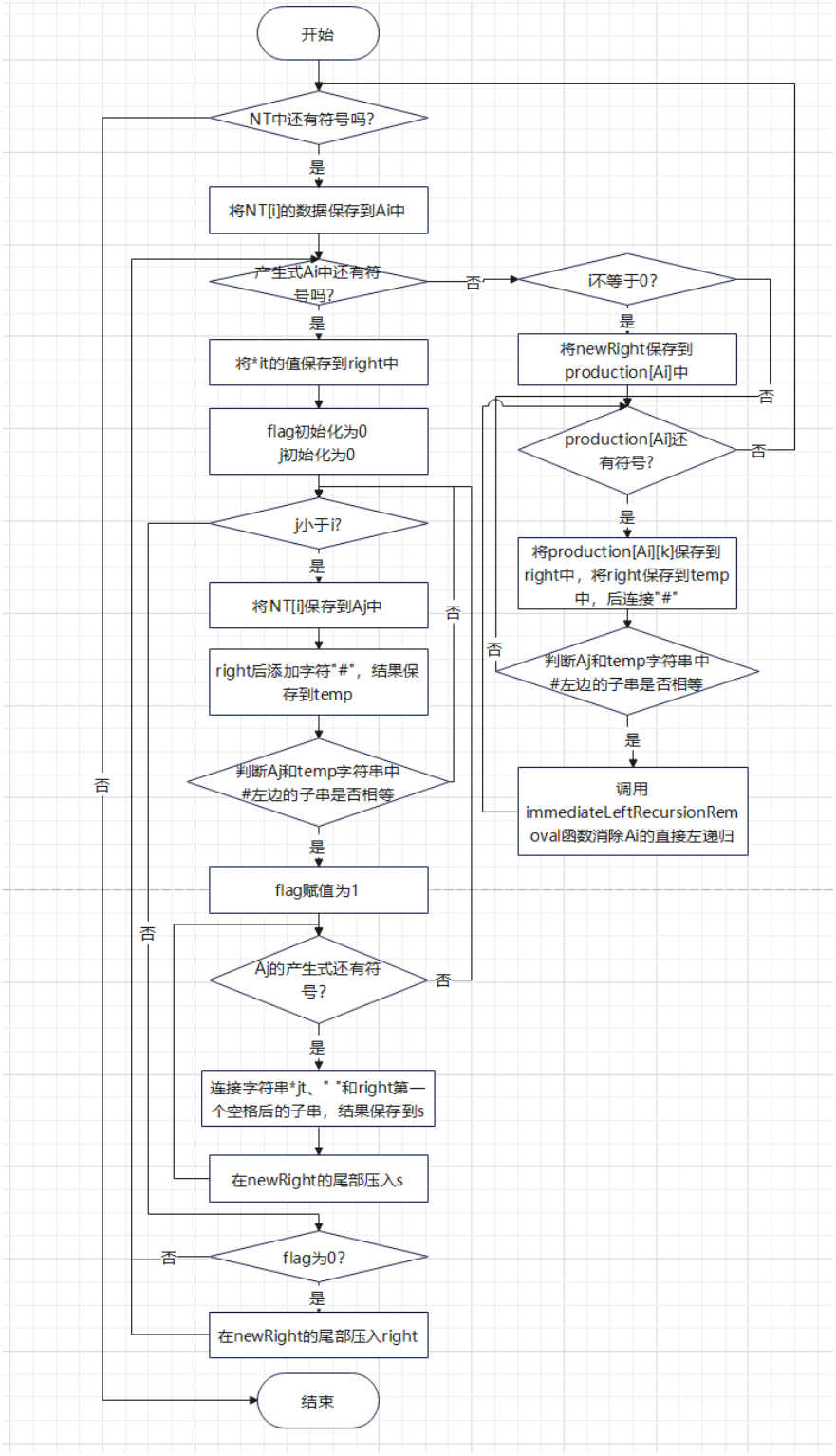
```
void Grammar::leftRecursionRemoval() {
    //遍历每一个 NT
    for (int i = 0; i < NT.size(); i++) {
        string Ai = NT[i];
        //cout<<"Ai:"<<Ai<<endl;
        vector<string>newRight;//新的产生式右部
        //遍历 NT 的每一个产生式
        for (auto it = production[Ai].begin(); it < production[Ai].end(); it++) {
            string right = *it;
            //cout<<"right:"<<right<<endl;
            int flag = 0; //判断是不是左递归
            //遍历改变过的产生式
            for (int j = 0; j < i; j++) {
                string Aj = NT[j];
                //cout<<"Aj:"<<Aj<<endl;
                string temp = right + "#";
                //如果有 Ai->AjB，替换 Aj 为 Aj 的产生式
                if (strcmp(const_cast<char *>(Aj.c_str()), strtok(const_cast<char *>(temp.c_str()), " #")) == 0) {
                    flag = 1;
                    cout << Aj << "lllllllllllllll ";
                    cout << temp << endl;
                    for (auto jt = production[Aj].begin(); jt < production[Aj].end(); jt++) {
                        string s = *jt + " " + right.substr(right.find(" ") + 1); //substr(1)是从空格位置往后的子串
                        //cout<<"s:"<<s<<endl;
                        newRight.push_back(s);
                    }
                }
            }
            //没有可替换的产生式
            if (flag == 0)
                newRight.push_back(right);
        }
        if (i != 0)
            production[Ai] = newRight;
        //去除包含 Ai 的直接左递归
        for (int k = 0; k < production[Ai].size(); k++) {
            string right = production[Ai][k];
            string temp = right;
            temp += "#";
            if (strcmp(const_cast<char *>(Ai.c_str()), strtok(const_cast<char *>(temp.c_str()), " #")) == 0)
                immediateLeftRecursionRemoval(Ai);
        }
    }
}
```


开始
NT还有终结符?
是
否
Ai保存NT[i]
Ai还有产生式?
是
否
i不等于0?
是
right保存*it
flag初始化为0
j初始化为0
production[Ai]保存newRight
production[Ai]还有符号?
否
是
j小于i?
是
否
right保存production[Ai][k]
temp保存right，后连接"#"
Aj保存NT[j]
否
right后添加字符"#"，结果保存到temp
判断Ai和temp按“#”分割后是否相等
是
判断Aj和temp按“#”分割后是否相等
消除Ai的直接左递归
否
是
否
flag赋值为1
否
Aj的产生式还有符号?
是
连接字符串*jt、" "和right第一个空格后的子串，结果保存到s
在newRight的尾部压入s
flag为0?
是
在newRight的尾部压入right
结束

上图需要在文字描述上更加清晰，见下面红色的改写。



最后正确的图例（注意事项：流程图放入实验报告书时需要隐藏网格线）：



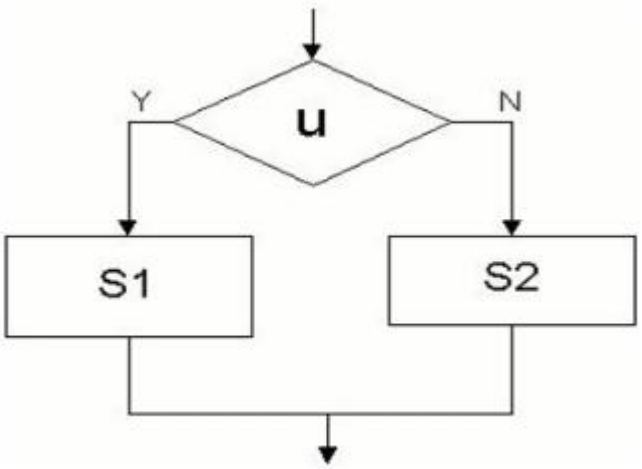
算法处理流程图的画法

只有一个入口和一个出口。

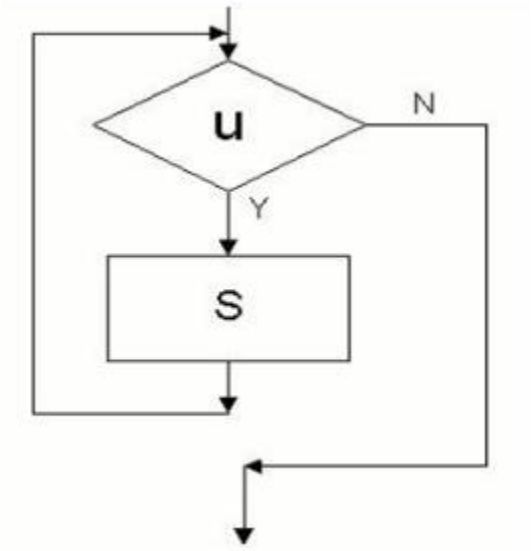
一个源程序就只有三种结构：

（1）一般就是顺序语句，执行完这个语句就是下一个语句了，对应的就一个步骤和下一个步骤，这个语句是就赋值就写赋值，是出栈就写出栈，是进栈就写进栈。

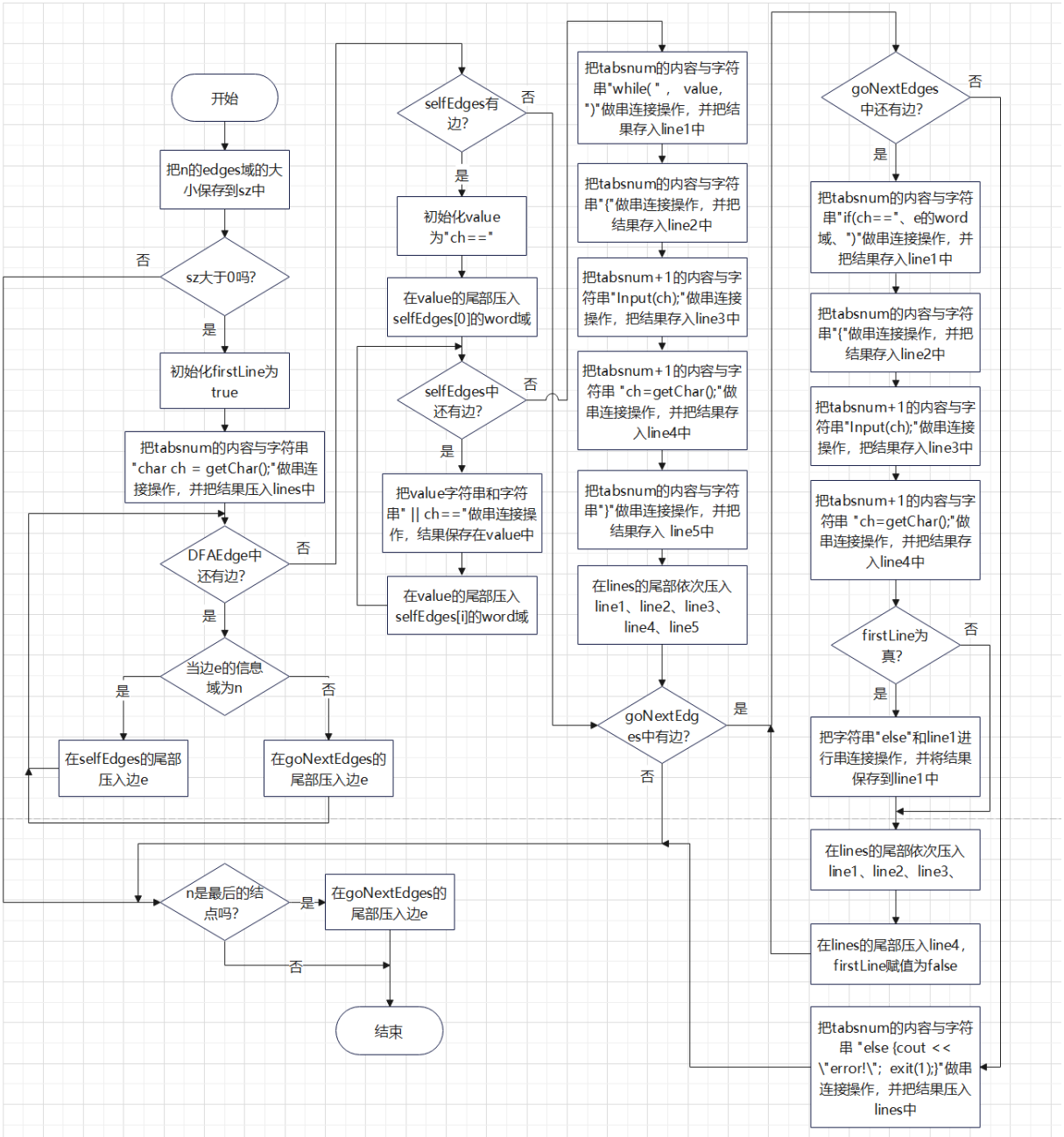
（2）条件判断语句，就是有 if 语句的，有 switch 语句的这种，不就采用下面的结构画出来，文字上就是用如果什么条件成立或不成立就执行什么，



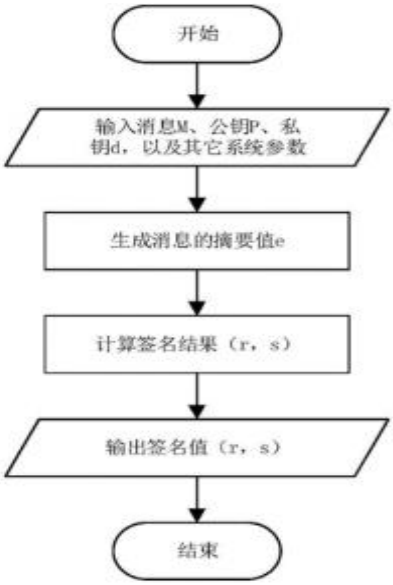
（3）循环语句，就是有 for ,while 或 do~while 语句的这种，就采用下面的结构画出来，文字说明就是用反复执行下面的语句，在什么条件成立或不成立的时候就结束

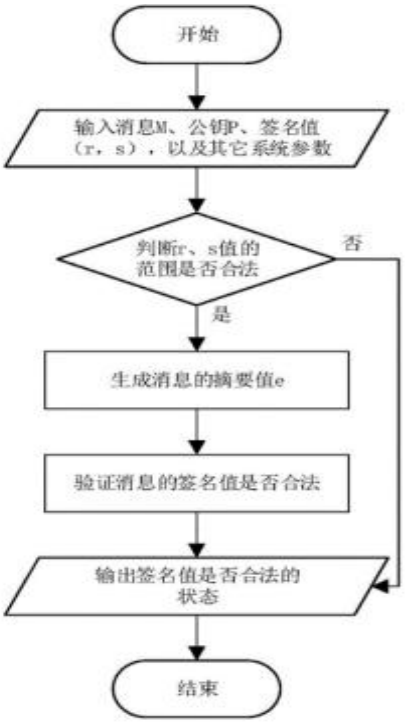


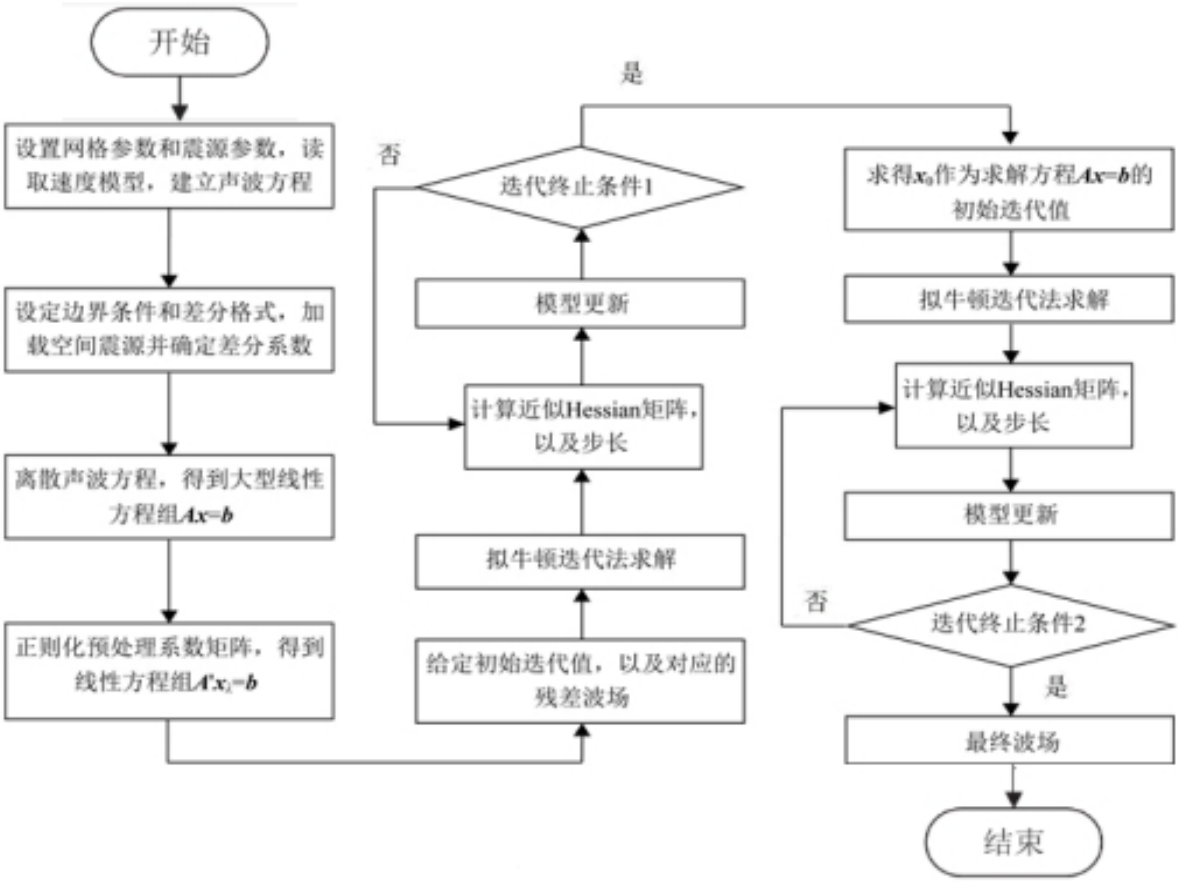
下面是流程图的绘制图例展示：



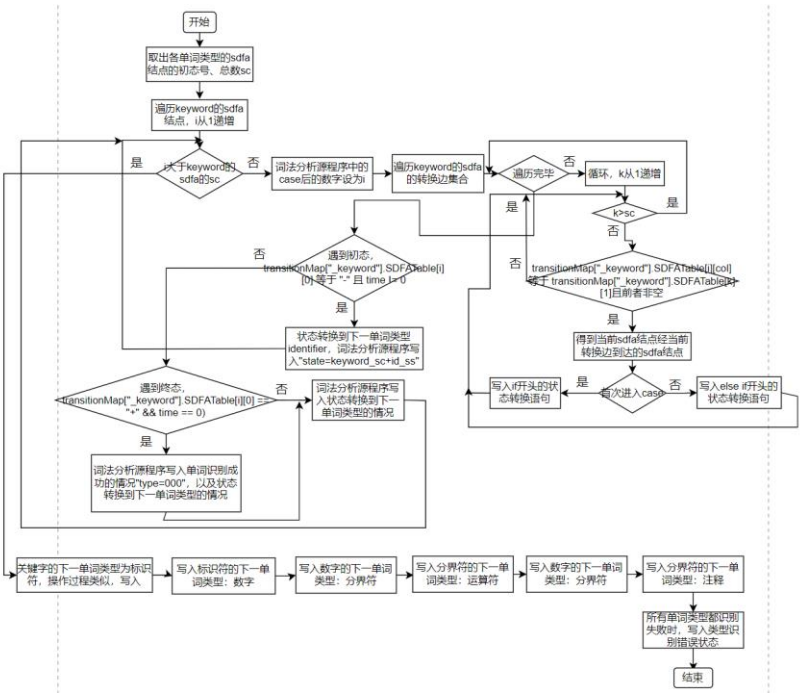
注意：下面的图例展示都只是流程图图例的样子的示范，而不是可以画成这样模糊的流程图。

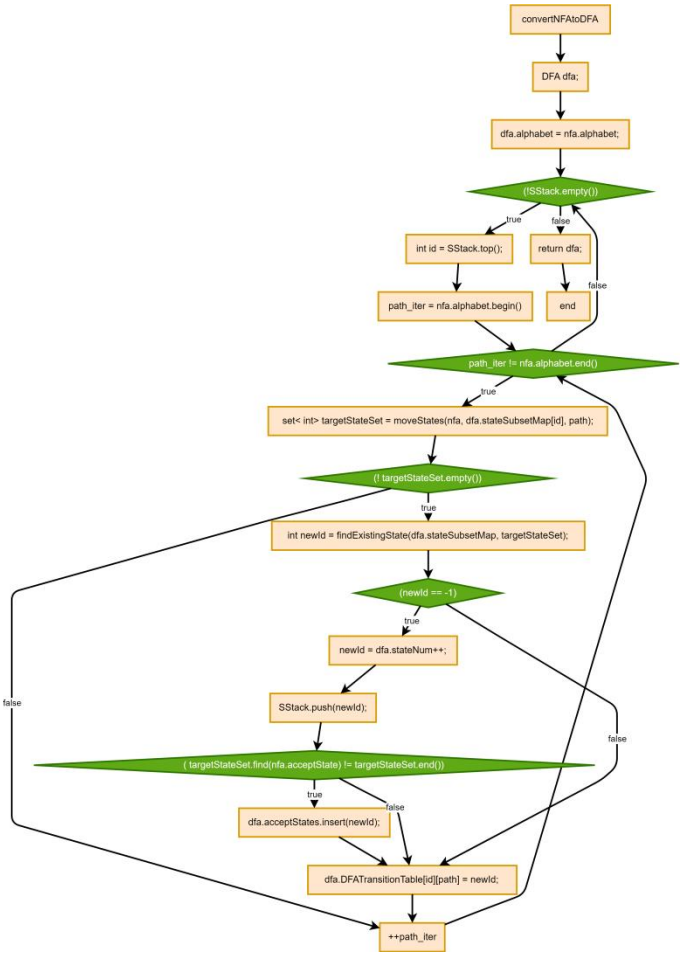


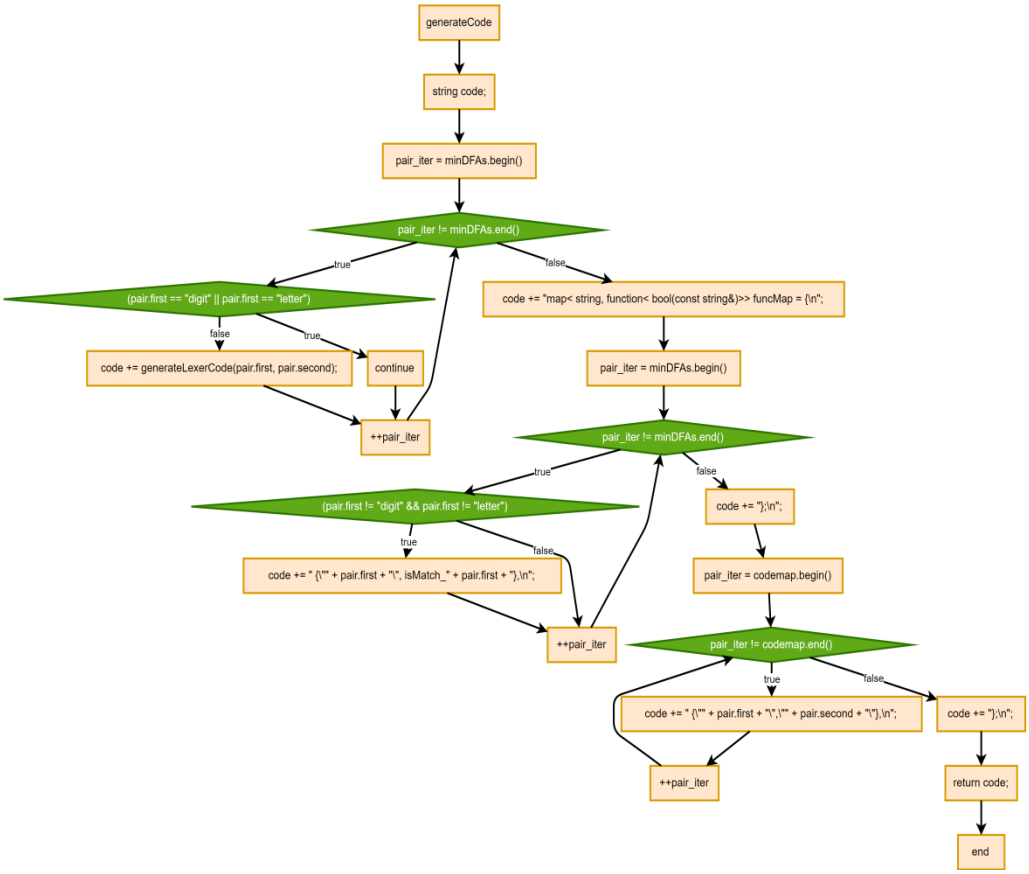


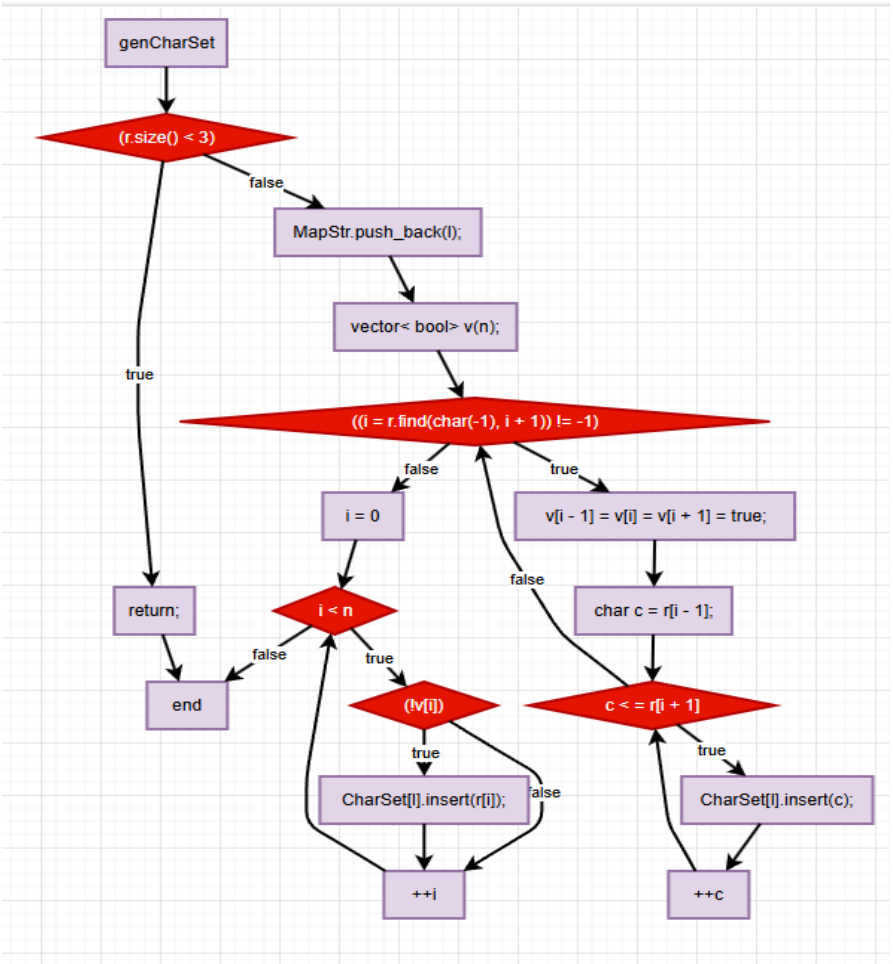


错误的流程图示例：


开始
取出各单词类型的sdfa结点的初态号、总数sc
遍历keyword的sdfa结点，i从1递增
i大于keyword的sdfa的sc
是
否
词法分析源程序中的case后的数字设为i
遍历keyword的sdfa的转换边集合
遍历完毕
否
循环，k从1递增
是
k>sc
是
否
遇到初态，transitionMap["_keyword"].SDFATable[i][0]等于"-"且time != 0
否
是
transitionMap["_keyword"].SDFATable[i][col]等于transitionMap["_keyword"].SDFATable[k][1]目前者非空
否
是
得到当前sdfa结点经当前转换边到达的sdfa结点
状态转换到下一单词类型identifier，词法分析源程序写入"state=keyword_sc+id_ss"
遇到终态，transitionMap["_keyword"].SDFATable[i][0] == "+" && time == 0)
否
是
词法分析源程序写入状态转换到下一单词类型的情况
词法分析源程序写入单词识别成功的情况"type=000"，以及状态转换到下一单词类型的情况
写入if开头的状态转换语句
首次进入case
是
否
写入else if开头的状态转换语句
关键字的下一单词类型为标识符，操作过程类似，写入
写入标识符的下一单词类型：数字
写入数字的下一单词类型：分界符
写入分界符的下一单词类型：运算符
写入数字的下一单词类型：分界符
写入分界符的下一单词类型：注释
所有单词类型都识别失败时，写入类型识别错误状态
结束


convertNFAtoDFA
DFA dfa;
dfa.alphabet = nfa.alphabet;
(!SStack.empty())
true
false
int id = SStack.top();
return dfa;
path_iter = nfa.alphabet.begin()
end
false
path_iter != nfa.alphabet.end()
true
set< int> targetStateSet = moveStates(nfa, dfa.stateSubsetMap[id], path);
(! targetStateSet.empty())
true
int newId = findExistingState(dfa.stateSubsetMap, targetStateSet);
(newId == -1)
true
newId = dfa.stateNum++;
SStack.push(newId);
(targetStateSet.find(nfa.acceptState) != targetStateSet.end())
true
false
dfa.acceptStates.insert(newId);
dfa.DFATransitionTable[id][path] = newId;
++path_iter
false
false


generateCode
string code;
pair_iter = minDFAs.begin()
pair_iter != minDFAs.end()
true
false
(pair.first == "digit" || pair.first == "letter")
true
false
code += generateLexerCode(pair.first, pair.second);
continue
++pair_iter
code += "map< string, function< bool(const string&)>> funcMap = {\n";
pair_iter = minDFAs.begin()
pair_iter != minDFAs.end()
true
false
(pair.first != "digit" && pair.first != "letter")
true
false
code += " {\"" + pair.first + "\", isMatch_" + pair.first + "},\n";
++pair_iter
code += "};\n";
pair_iter = codemap.begin()
pair_iter != codemap.end()
true
false
code += " {\"" + pair.first + "\",\"" + pair.second + "\"},\n";
++pair_iter
code += "};\n";
return code;
end


genCharSet
(r.size() < 3)
true
false
MapStr.push_back(l);
vector< bool> v(n);
((i = r.find(char(-1), i + 1)) != -1)
false
true
i = 0
v[i - 1] = v[i] = v[i + 1] = true;
return;
i < n
false
true
end
(!v[i])
true
false
CharSet[l].insert(r[i]);
++i
false
char c = r[i - 1];
c < = r[i + 1]
true
CharSet[l].insert(c);
++c