****

编译原理

实验报告

姓名： 郑云川

学号： 09015131

东南大学计算机科学与工程学院

School of Computer Science & Engineering

Southeast University

二0 一八 年 一 月

目录

[实验一 语法分析程序（自己写Lex生成） 3](#_Toc504206598)

[一、实验目标 3](#_Toc504206599)

[二、内容描述 3](#_Toc504206600)

[1 可选方法 3](#_Toc504206601)

[2 我的选择 3](#_Toc504206602)

[3 代码描述 3](#_Toc504206603)

[三、思路方法 5](#_Toc504206604)

[1 大致流程 5](#_Toc504206605)

[2 解析自定义lex 5](#_Toc504206606)

[3 生成NFA 6](#_Toc504206607)

[4 把NFA转化为DFA 6](#_Toc504206608)

[四、假设 6](#_Toc504206609)

[五、FA描述 7](#_Toc504206610)

[1 Lex中定义的正规表达式 7](#_Toc504206611)

[2 NFA和DFA 7](#_Toc504206612)

[六、部分数据结构描述 7](#_Toc504206613)

[1 NFA和DFA的边 7](#_Toc504206614)

[2 正规表达式 7](#_Toc504206615)

[3 NFA和DFA 7](#_Toc504206616)

[七、部分算法描述 8](#_Toc504206617)

[1 解析Lex文件 8](#_Toc504206618)

[2 正规表达式转NFA 8](#_Toc504206619)

[3 NFA生成DFA 9](#_Toc504206620)

[八、用例测试 9](#_Toc504206621)

[1 测试环境 9](#_Toc504206622)

[2 测试脚本 9](#_Toc504206623)

[3 测试结果 10](#_Toc504206624)

[九、困难与解决方案 12](#_Toc504206625)

[十、感受 12](#_Toc504206626)

[实验二 语法分析程序 13](#_Toc504206627)

[一、实验目标 13](#_Toc504206628)

[二、内容描述 13](#_Toc504206629)

[1 可选方法 13](#_Toc504206630)

[2 我的选择 13](#_Toc504206631)

[3 代码描述 13](#_Toc504206632)

[三、思路方法 14](#_Toc504206633)

[1 大致流程 14](#_Toc504206634)

[2 从文件中解析文法 14](#_Toc504206635)

[3 构建LR(1) DFA以及预测分析表 14](#_Toc504206636)

[4 对Token序列进行分析 14](#_Toc504206637)

[四、假设 14](#_Toc504206638)

[五、FA描述 15](#_Toc504206639)

[1 示例文件中的上下文无关文法 15](#_Toc504206640)

[2 DFA 15](#_Toc504206641)

[六、部分数据结构描述 15](#_Toc504206642)

[1 LR(1)项 15](#_Toc504206643)

[2 struct Lex 16](#_Toc504206644)

[3 Action与Goto子表 16](#_Toc504206645)

[4 DFA状态 16](#_Toc504206646)

[七、部分算法描述 17](#_Toc504206647)

[1 点之后非终结符的扩展 17](#_Toc504206648)

[2 构造DFA以及Action和Goto子表 17](#_Toc504206649)

[3 DFS 17](#_Toc504206650)

[4 主控程序（monitor） 17](#_Toc504206651)

[八、用例测试 18](#_Toc504206652)

[1 测试环境 18](#_Toc504206653)

[2 测试脚本 18](#_Toc504206654)

[3 测试结果 18](#_Toc504206655)

[九、困难与解决方案 22](#_Toc504206656)

[十、感受 22](#_Toc504206657)

# 实验一 语法分析程序（自己写Lex生成）

## 一、实验目标

这学期开设了编译原理的课程，我们系统地学习了用正规表达式定义特定类型的字符串，以及去分析程序里的某个“单词”是什么类型，也就是语法分析；我们通过正规文法来定义一种特定的语言，通过文法把语言抽象出来，然后对给定的程序片段来分析它是否符合文法的规则，也就是语法分析；光有词法和文法还不够，添加了相应的语义动作才更有意义；我们想要将高级语言翻译成汇编代码并不容易，先通过翻译成中间代码做铺垫，我们学的中间代码形式有三地址码（TAC）和四元式（Quadruple）；为了提高程序的效率，我们还要通过对控制流和数据流的分析来优化我们的中间代码；此外我们还以C语言为例子，学习运行时刻存储空间的分配。

我认真学习了这门课程，也做了一些习题来巩固所学，但是通过自己编写代码来实现其中一些功能会让我对编译部分过程的理解更加深刻，因此我们选择了语法分析和语法分析两个步骤来编写程序。第一个实验是词法分析，要自选一种方式构造语法分析程序，输入字符流（以及正规表达式），输出Token序列。

## 二、内容描述

### 1 可选方法

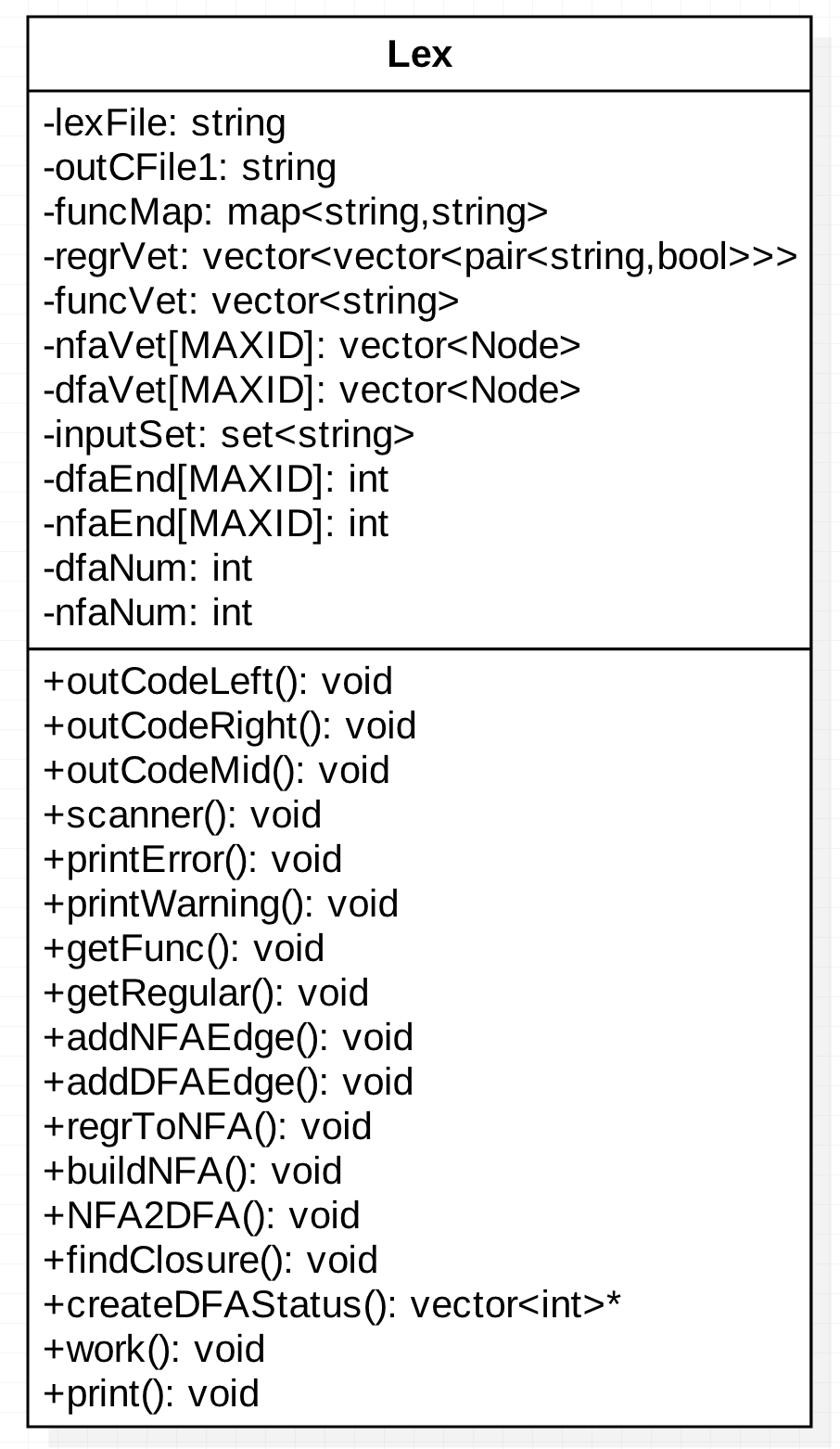
老师的指导书中提供了三种思路：一是直接模仿龙书2.6.5小节的代码；二是自己定义一些正规表达式，手动转化为NFA，然后化为DFA，最后再把DFA化为最少状态的DFAo，就像我们平时做题目那样，以此为基础来编写自己的词法分析程序；三是自己定义lex文件，编写自己的lex，借助lex文件生成词法分析程序。

### 2 我的选择

前两种方法的功夫没有在编程上，主要用纸用笔来演算，编程则比较机械，语句没有难度，我认为着对编程能力的提高没有很大帮助，于是我选择第三种思路。借鉴Lex的思想，想通过正规表达式来生成一个简单的词法分析程序，为了简化实验，定义了自己的.l文件，在程序中读取并构造NFA，转化为DFA，时间有限，没来得及完成DFA状态的最小化，因为词法分析程序主要是读取字符并进行状态的转换，程序语句多为判断语句，为我生成词法分析程序的代码提供了便利。

### 3 代码描述

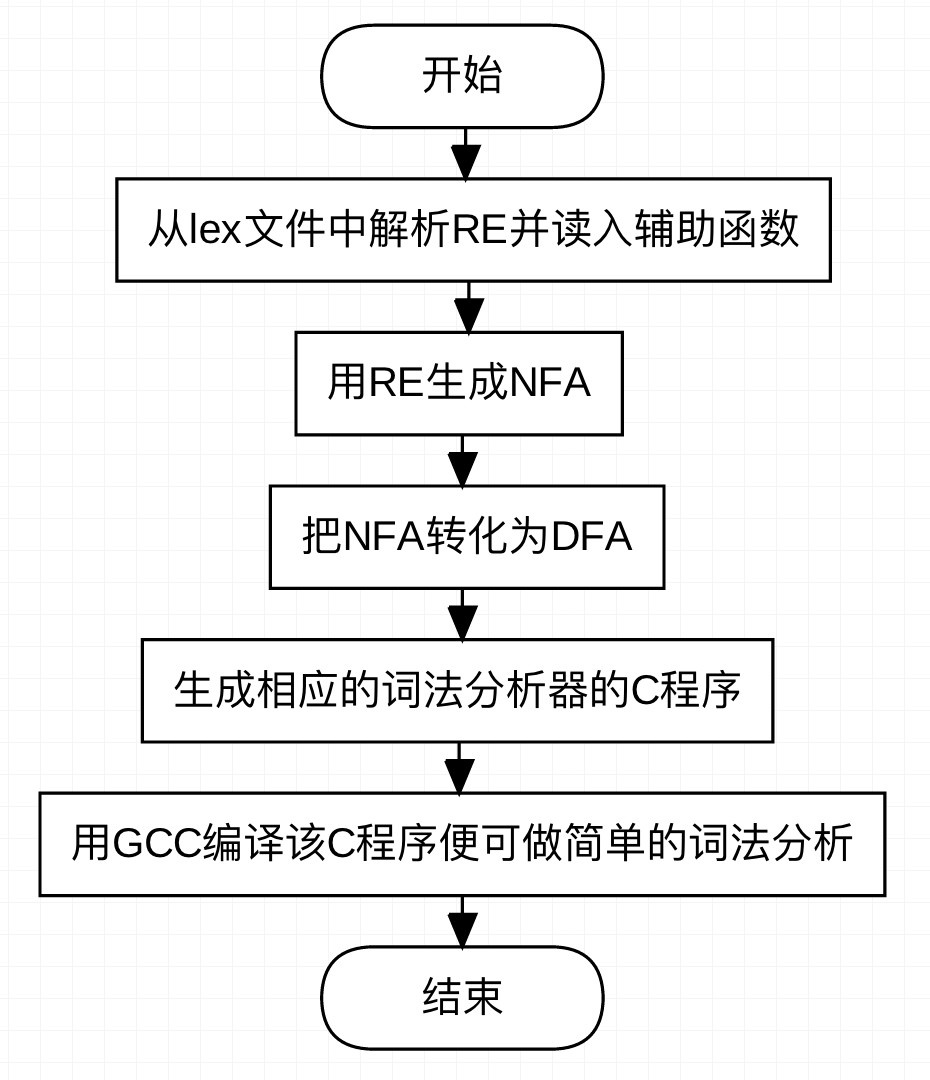
主要功能均集中在Lex类中，下图为类的结构：



## 三、思路方法

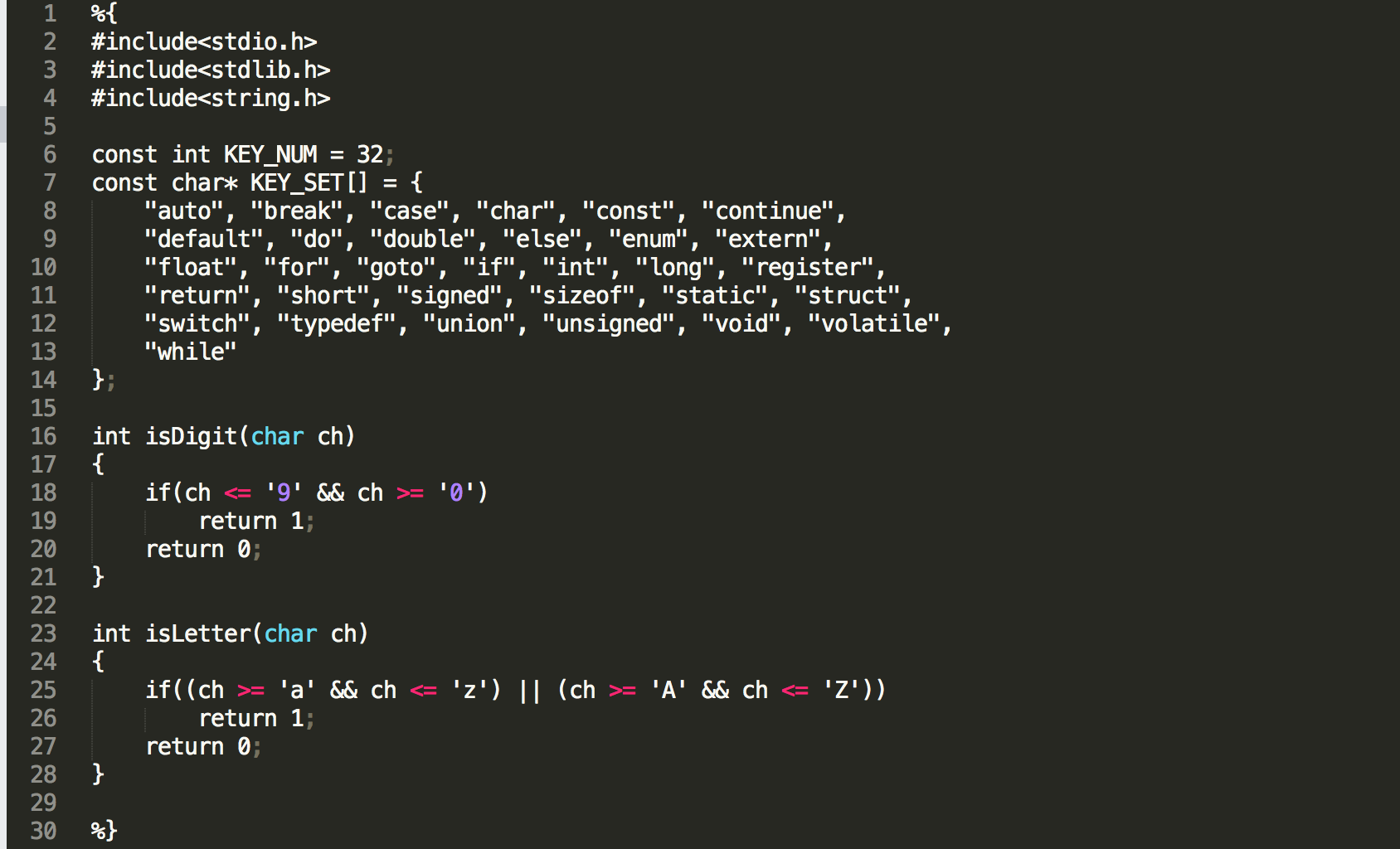
### 1 大致流程

程序大体处理流程如下所示：

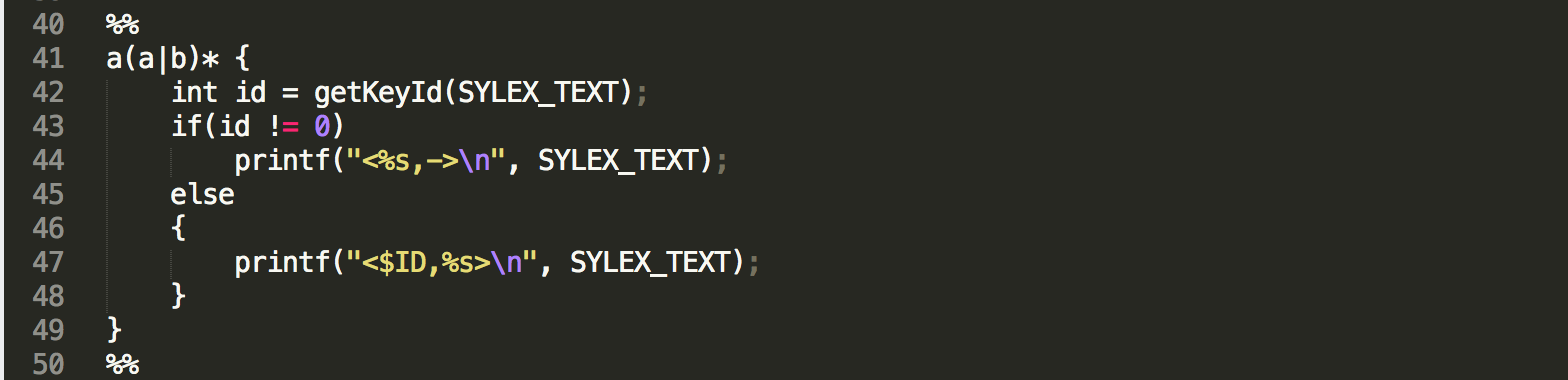


### 2 解析自定义lex

借鉴lex思想，通过部分符号来分隔不同的部分，%{与%}之间是直接拷贝到语法分析程序中的辅助的函数；%!和%!之间将接下来可能用到的一些符号同相应的判断的辅助函数关联起来，%%和%%之间放我们定义的各个正规表达式，每个正规表达式放在%$和%$之间，包括我们课上学到的正轨表达式的定义，其中如果涉及括号、乘号等在正规表达式中用到的符号，在前面加上%相当于转义，来进行区分。以下为示例：







### 3 生成NFA

这里生成NFA模拟课上所讲自上而下的思路，借助栈的结构可以实现，详细描述在下面几个部分展开。

### 4 把NFA转化为DFA

这部分的思想和上课讲的一致，求初始状态的闭包，根据发出边扩展状态，利用子集构造法，迭代至没有新的状态产生。详细的在后面部分呈现。

## 四、假设

为方便起见，本实验假设不区分函数名和变量名，一致当作identifier对待；遇到错误，解析不向下进行，直接报错退出。

## 五、FA描述

### 1 Lex中定义的正规表达式



### 2 NFA和DFA

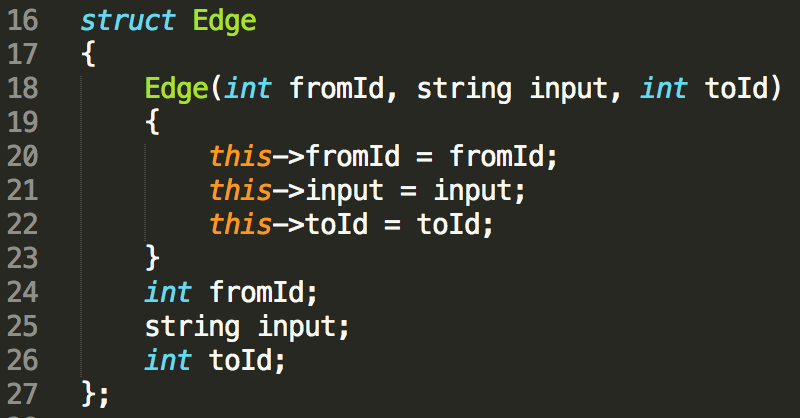
这部分由程序生成，将在结果中展示。

## 六、部分数据结构描述

这里我会简单描述一下用到的重要的数据结构。

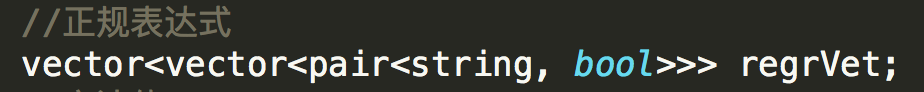
### 1 NFA和DFA的边

包括发出点、到达点，以及边上的符号：



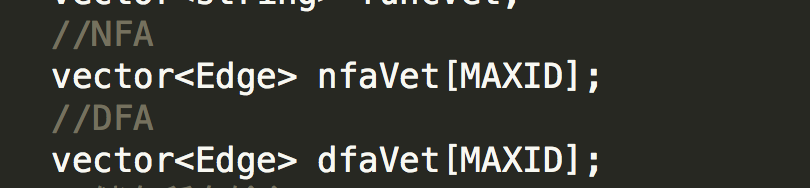
### 2 正规表达式

正规表达式用了一个稍显复杂的结构：一个vector，它的类型是vector，里面这个vector的类型是pair，pair的first是一个string，储存符号，second是一个bool变量，它的值为true表示这个符号是输入的符号，值为false表示它是运算符。



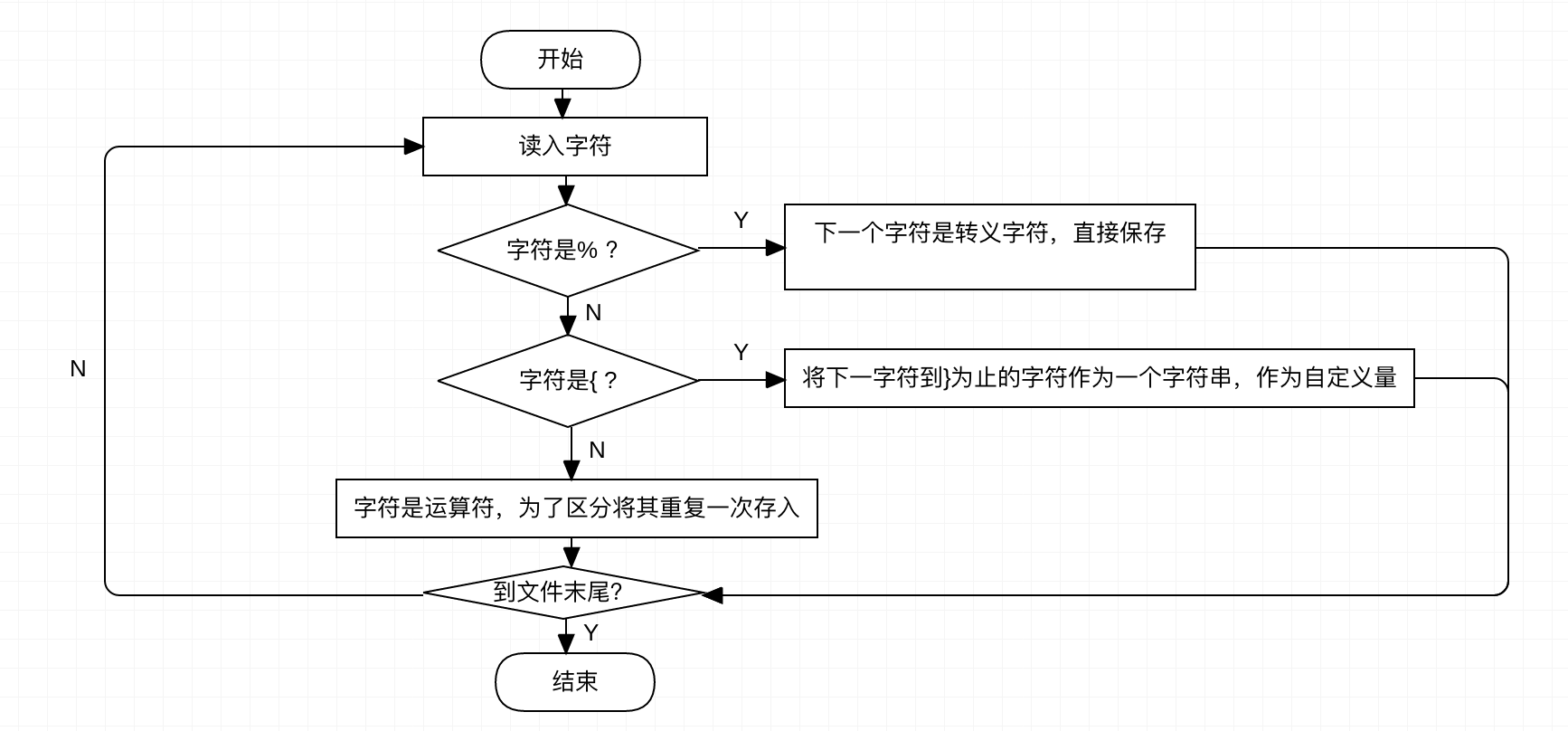
### 3 NFA和DFA

NFA和DFA同样用vector来表示，类型是Edge，Edge是1中表述的FA的边。



## 七、部分算法描述

### 1 解析Lex文件



### 2 正规表达式转NFA

首先我们定义三个栈：s\_str，保存输入信息；s\_st，保存起始状态序号；s\_ed，保存终止状态序号。下面用一个例子来解释栈的进出规则。

ab\*(a\*|(ab)\*)

首先，将整个式子的起始状态id设为0和终止状态id为1，将其分别入栈s\_st和s\_ed

接下来对整个表达式进行扫描

[a] 添加状态id=3，入栈s\_st，a入栈s\_str.

[b] 添加状态id=4，入栈s\_st，b入栈s\_str.

[(] 因为上一个符号b已经添加的状态4可以作为括号内部分的起始状态，那么只添加终止状态=5，入栈s\_ed，(入栈s\_str.

[a] 添加状态id=6，入栈s\_st，a入栈s\_str.

[\*] 获取s\_st栈顶的两个状态，4和6, 添加4–ε–>6 和 6–ε–>4

[|] 依次对s\_str进行出栈直到栈顶为(，同时对s\_st进行出栈，

并将其两两用s\_str进行连接. 如a出栈(s\_str)，6出栈(s\_st)，添加4–a–>6.

最重要的地方是，将最先出栈的状态id与s\_ed进行连接，即添加 6–ε–>5.

[(] 添加终止状态=7，入栈s\_ed，(入栈s\_str.

[a] 添加状态id=8，入栈s\_st，a入栈s\_str.

[b] 添加状态id=9，入栈s\_st，b入栈s\_str.

[)] 依次对s\_str进行出栈直到栈顶为(，同时对s\_st进行出栈，

并将其两两用s\_str进行连接(和|有些类似)，然后将该括号部分的终止id出栈=7，并将其入栈s\_st，

同时入栈’-‘至s\_str，’-‘作为括号里面部分式子的输入符号(其实是起占位符的作用，

这样的话，括号整体部分就与单个输入没有区别，既便于闭包符的处理，又解决了多个括号并列的情况).

[\*] 获取s\_st栈顶的两个状态，6和7, 添加7–ε–>6 和 6–ε–>7

[)] 与上面）的处理是完全一样的。

### 3 NFA生成DFA

这里使用的方法是子集法： 每个状态表示为一个数字，那么我们用一个vector表示一个状态集合。 再使用一个set和一个queue，set用于对vector进行查重，queue用于遍历，从起始状态的集合开始，将其经每个输入到达的状态加入queue，当然，前提是该状态集合没有在set中出现过。 这里有个重点是关于空的处理，关键代码如下。

//i为当前状态，input为输入，state为存放可到达的状态的集合

void Lex:: findClosure (int i, string input, vector<int>\* state)

{

for(auto x : nfaVet[i])

{

int sId = x.toId;

bool flag = true;

for(auto iter=state->begin(); iter!=state->end(); iter++)

if((\*iter) == sId)

{

flag = false;

break;

}

if(flag && input.compare(x.input) == 0)

{

state->push\_back(sId);

findClosure (sId, "", state);

}

}

}

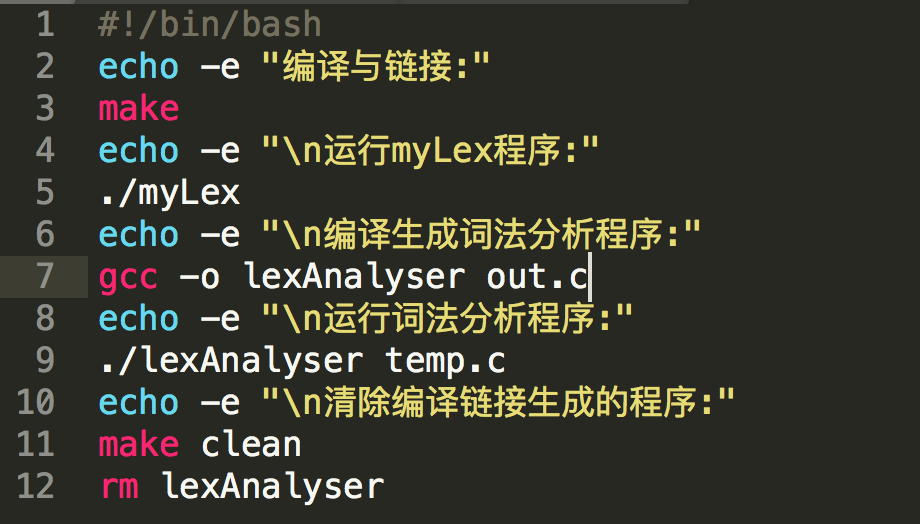
## 八、用例测试

### 1 测试环境

Mac OS 10.12.6, Ubuntu 16.04, clang-900.0.39.2, gcc 5.4.0

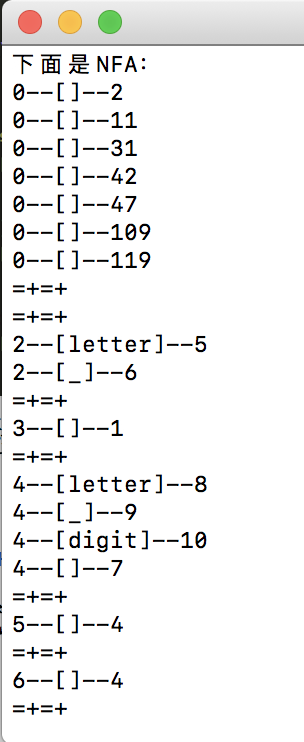
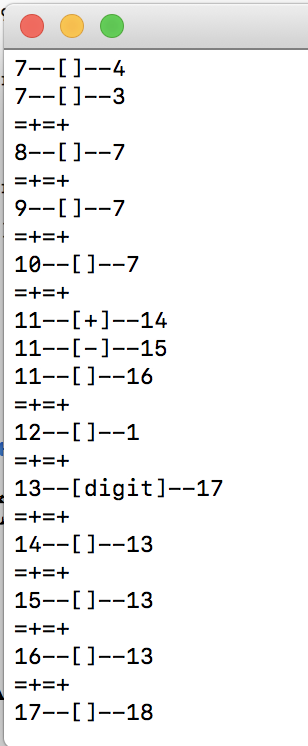
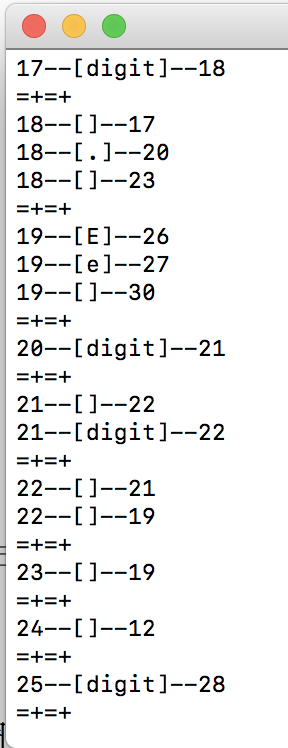
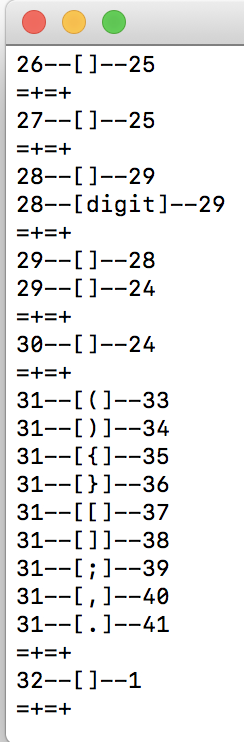
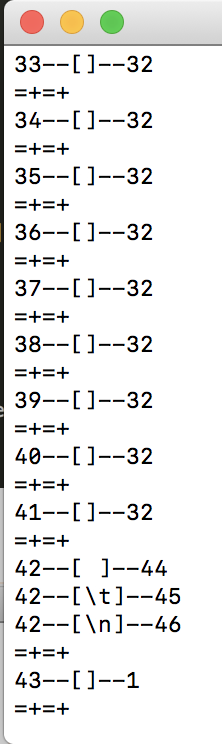
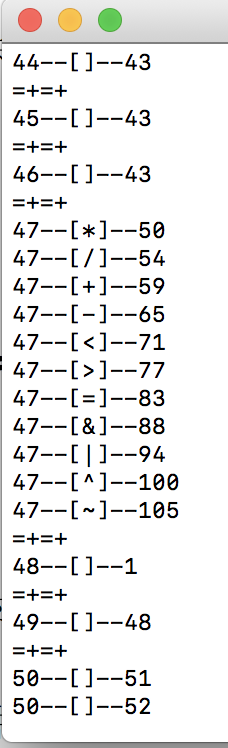
### 2 测试脚本

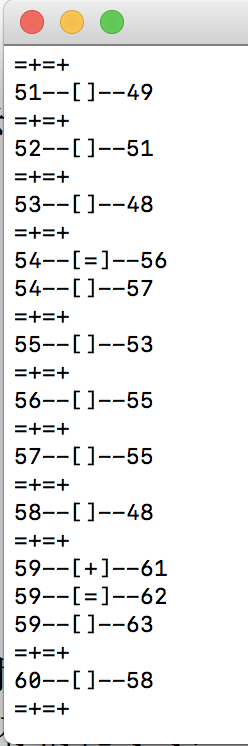
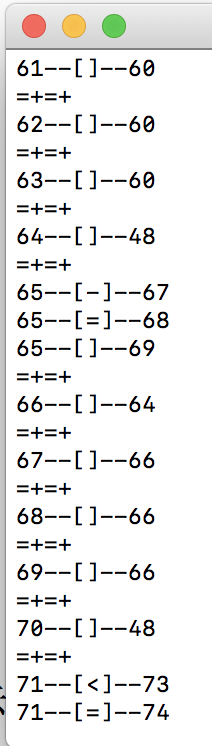
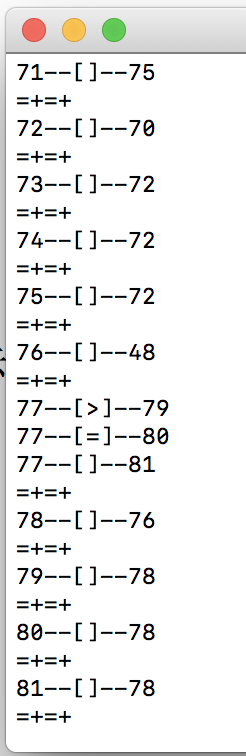
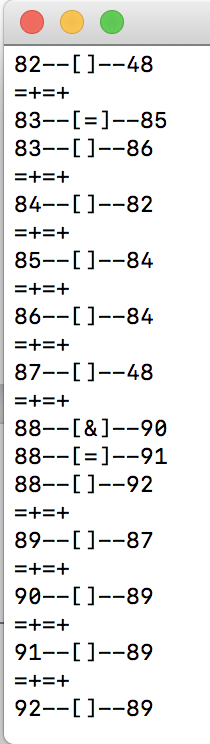
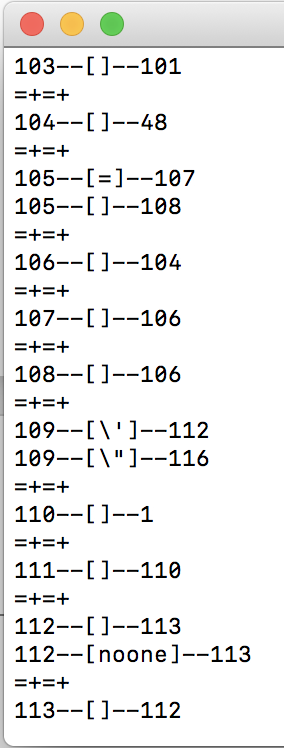
自己写了makefile文件用于编译与链接；同时写了Unix脚本run.sh用于测试，在脚本中首先make进行编译链接，然后运行myLex程序，接着用生成的词法分析器去分析程序temp.c，最后make clean命令删除之前的.o和.gch以及可执行文件。

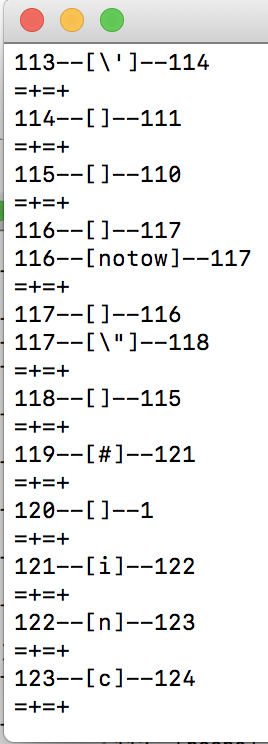
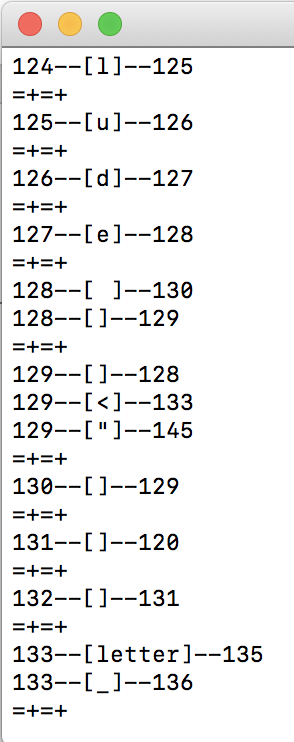
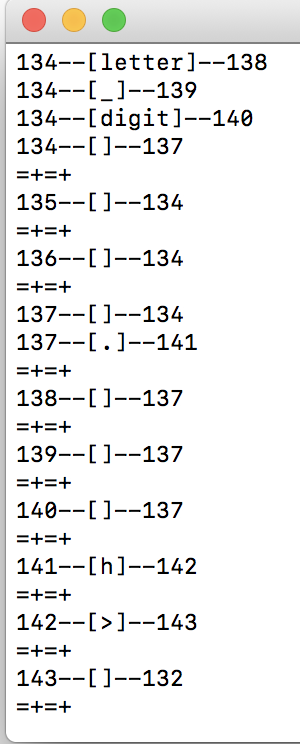
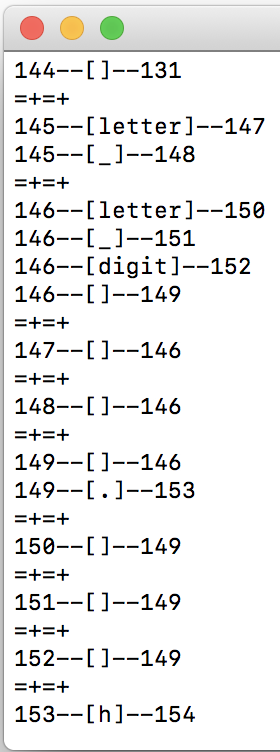
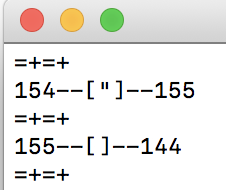


### 3 测试结果

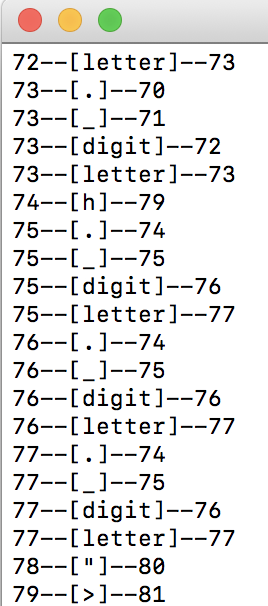
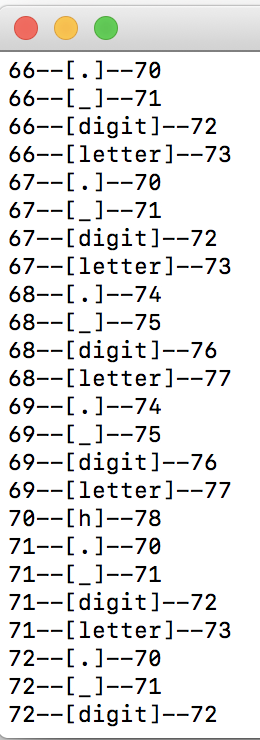
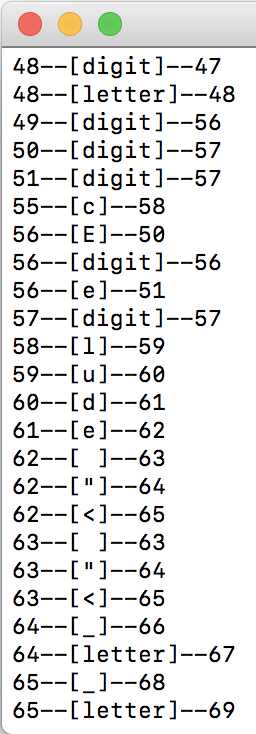
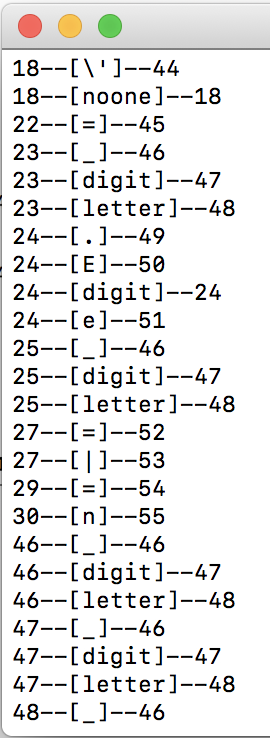
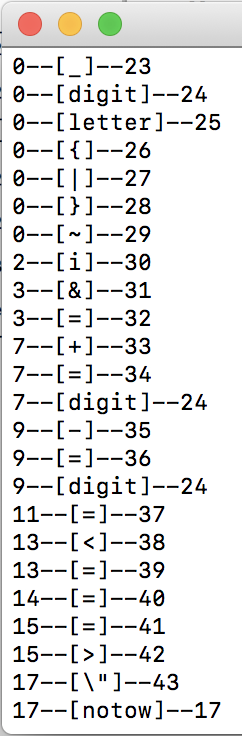
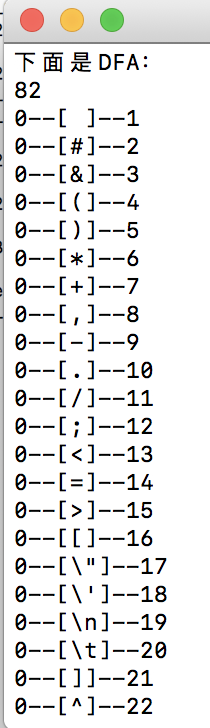
生成的NFA：

生成的DFA：



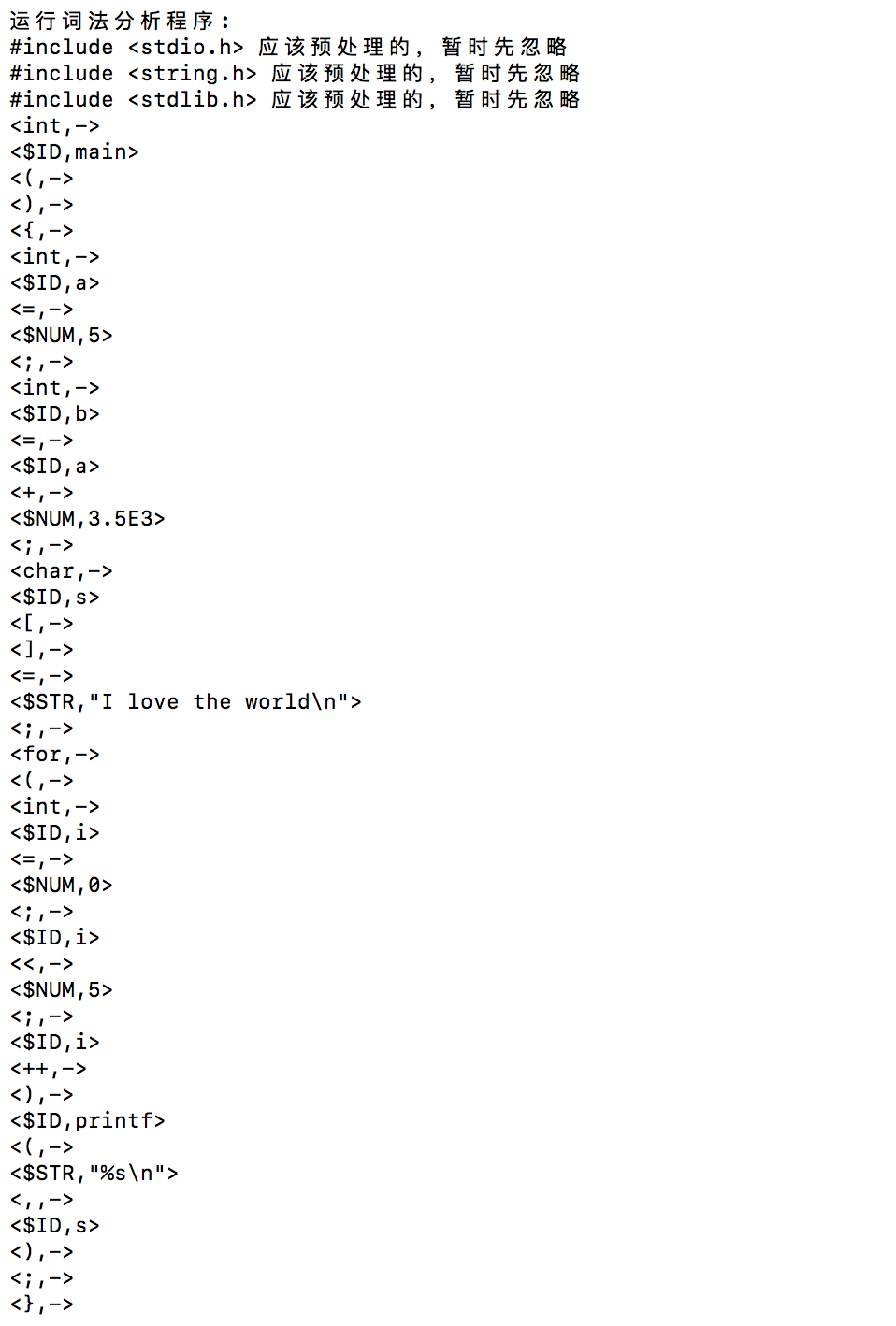
生成的DFA的终态：

1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 52, 53, 54, 56, 57, 80, 81

测试用程序：



结果：



## 九、困难与解决方案

首先是如何定义自己的lex文件并把正规表达式顺利地读入，这里有参考正统的lex的规范，但是因为没有用到所有的特性，因此省略了一些内容，也和同学讨论了一下，参考了网上部分实现，最终采用了上文所述的规则和实现方法。

其次是将正规表达式转化为NFA的过程，我想尝试不用Thompson方法，转化成后序再进行处理，恰好在网上看到一篇博文，借助3个栈从左向右读取RE便可完成解析，这一部分我直接用了作者的代码，用更简单的例子来验证也通过了，看来栈的妙用还要更仔细地去研究。

将NFA转化为DFA的过程便是实现课上学到的思想。

## 十、感受

其实实现这个长得有点像Lex的程序并不那么容易，认真地读龙书，也在网上进行了大量的搜索，参考了他人的诸多代码。一开始本打算用C语言实现，因为之后想要多学习系统编程，想用程序进行练手，但过程中发现，C语言的标准库的内容不多，很多数据结构没有，最后还是转向C++，C++的STL提供了很多数据结构，在参考他人的代码的过程中也学到了很多，课上的C++没有讲STL，但这恰恰是C++最强大的地方之一，还是需要靠写程序去学习。另外，本实验没有进行优化，例如最简化DFA的状态，以及中间诸多遍历过程，可能留待课程设计中解决。

# 实验二 语法分析程序

## 一、实验目标

实验一完成了词法分析的实验，输入的是正规表达式和待分析的C程序，可以输出Token序列。当我们获得Token序列之后，可以输入到语法分析程序中，来判定这些词语的排列是否符合我们设定的文法，我们能够得到推导的序列或者是规约的序列。这个环节是整个编译系统中最重要的环节，起到承上启下的作用，原理复杂，中间环节也比较多，用代码来实现简化的语法分析的过程，可以帮助我们更好理解编译原理。

## 二、内容描述

### 1 可选方法

一是选择使用递归下降分析法，基于推导的思想；它的好处是理解起来简单直观，代码实现也相对容易，代价是它很大几率引起回溯，如果中文法里有产生式有公共左因子，或者存在左递归，可能引起无限递归，使用它对文法限制比较大。然而与老师交流中得知，gcc好像是采用这样的方法，那应该是复杂性和效率的权衡吧。

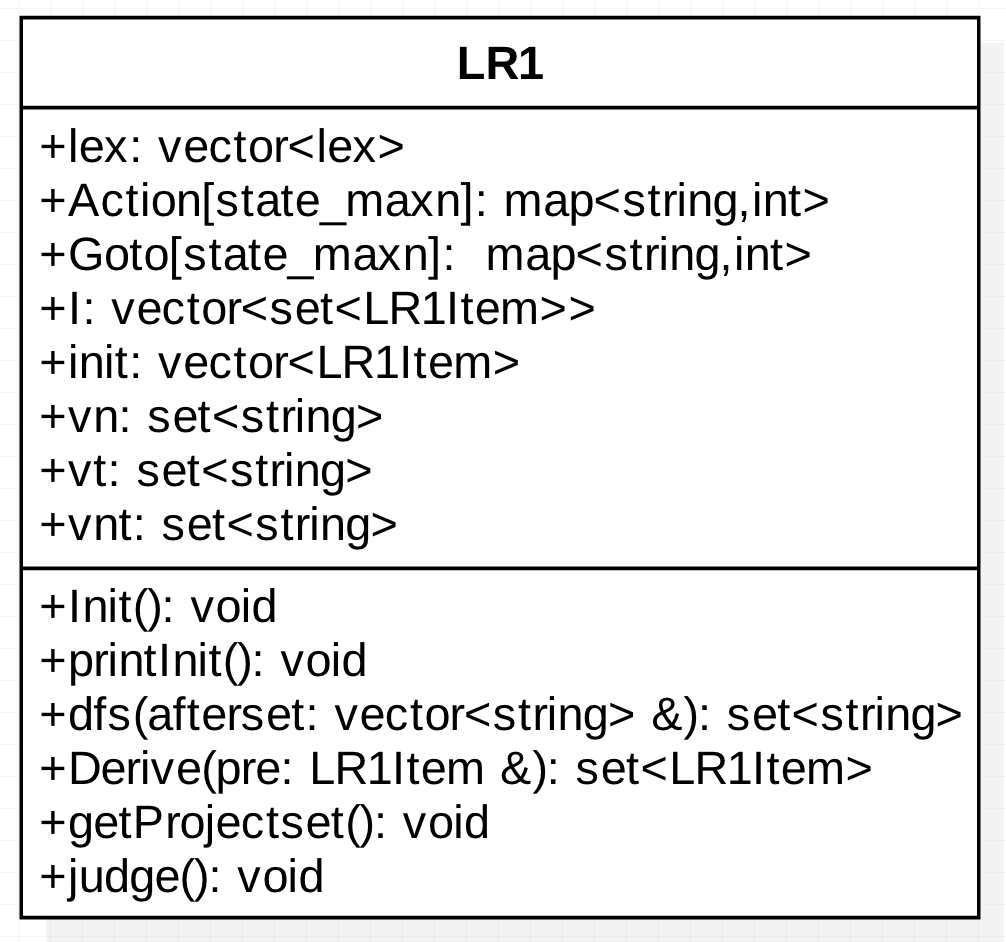
二是基于LL(1)的分析器，通过构造LL(1)分析表来进行语法分析。

三是基于LR(1)的分析器，通过构造LR(1)分析表来进行语法分析。LR(1)从原理和方法上来说在这三种方法里是较难的，但相应的能力也要更强大一些。

### 2 我的选择

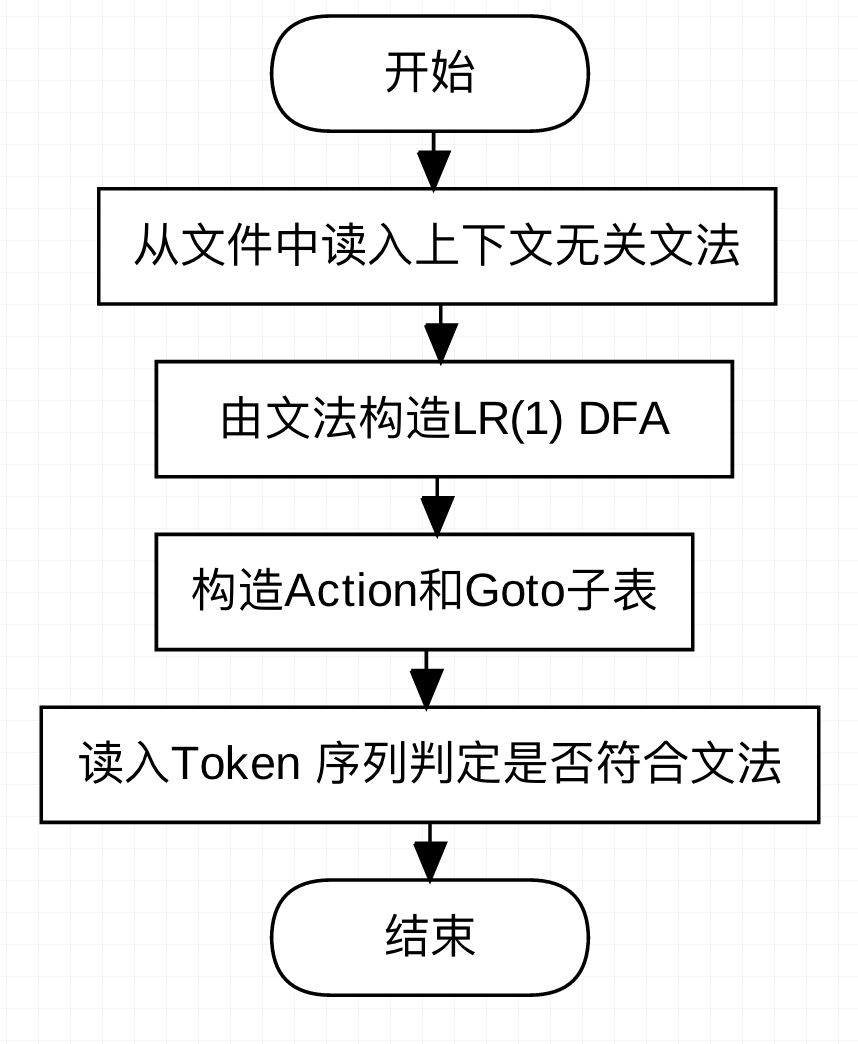
在本实验中我选择了方案三，也就是通过构造LR(1)分析表来完成语法分析。和不同的同学讨论也得知，他们中绝大部分是用纸和笔运用上课学到的知识，给出一个上下文无关文法，然后命名LR(1)项，构造出LR(1) DFA，以此来构造出LR(1)预测分析表。这是一个选择，并且在正确地得到构造分析表后写出代码也就不是那么困难了。结合之前实验一的经验教训，我仍希望尝试通过程序解析上下文无关文法，构造分析表，最终实现语法分析。

### 3 代码描述



## 三、思路方法

### 1 大致流程



### 2 从文件中解析文法

为了方便编程我定义终结符均由小写字母构成，非终结符均由大写字母构成，借助stringstream类来读入字符串，根据大小写便能够判定终结符和非终结符。

### 3 构建LR(1) DFA以及预测分析表

这一部分会在后面算法部分详细介绍。

### 4 对Token序列进行分析

这一部分理念同课上所讲的monitor程序结合预测分析表的思路，细节在后面部分呈现。

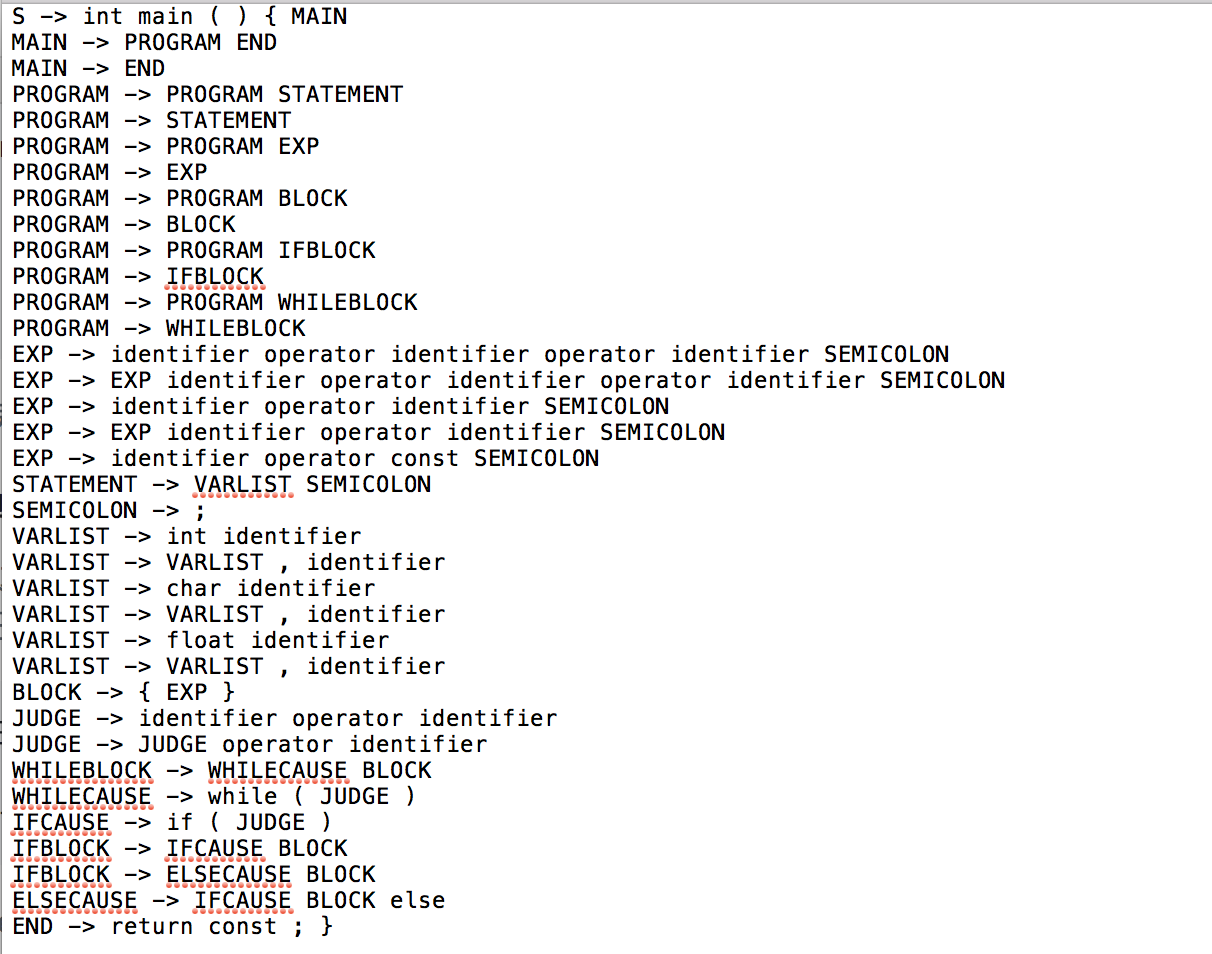
## 四、假设

为了方便实验，本实验假设遇见错误程序停止执行，报错退出。

假设输入文法不是二义文法。

## 五、FA描述

### 1 示例文件中的上下文无关文法



### 2 DFA

这部分会由程序生成，将在用例测试中进行展示。

## 六、部分数据结构描述

### 1 LR(1)项

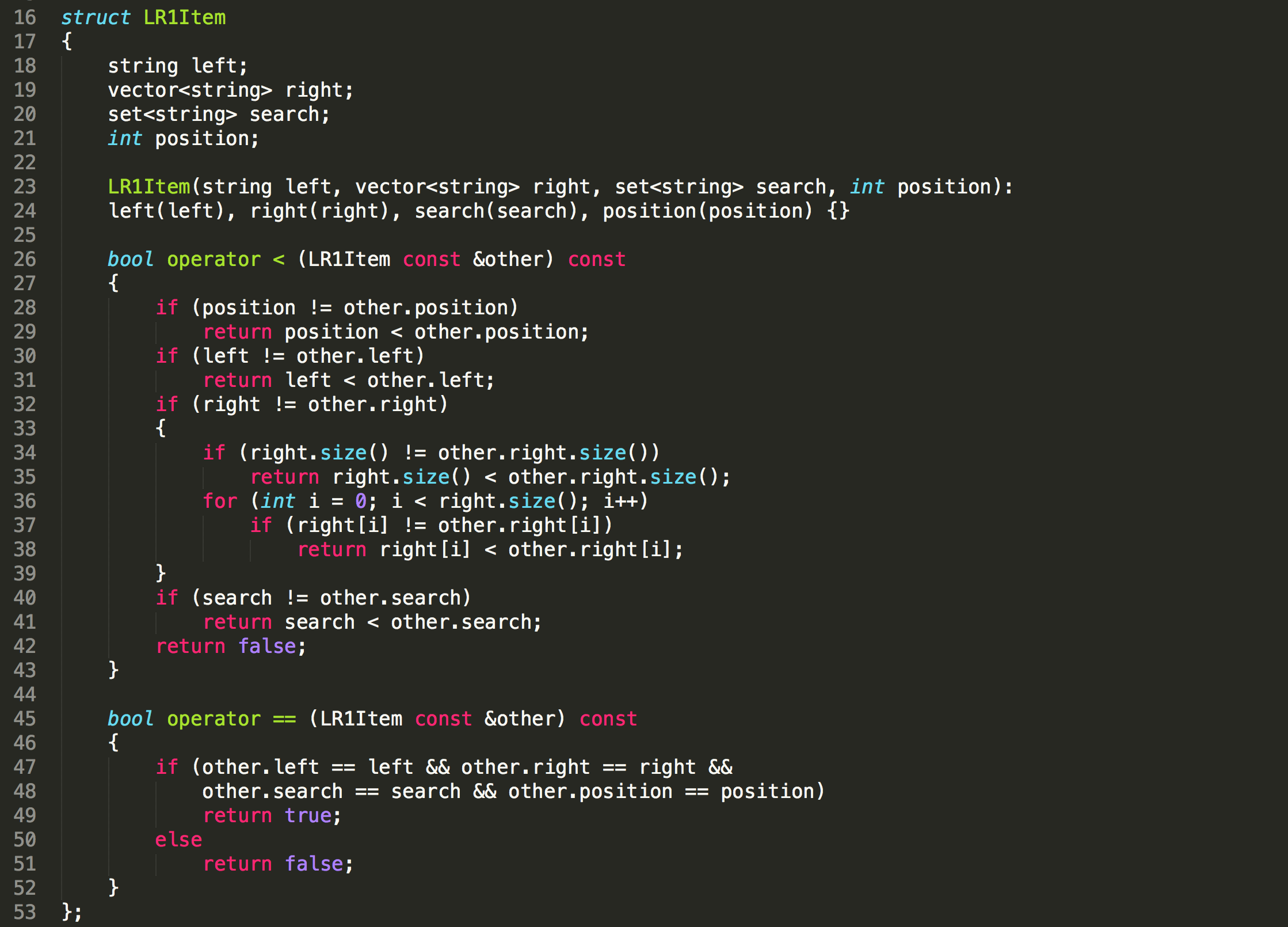
LR(1)项是LR(1)分析法中很重要的结构，我在程序中定义了struct LR1Item，它有几个变量：

string的类型的left，存储产生式左部；

vector<string>类型的right，储存产生式的右部的符号；

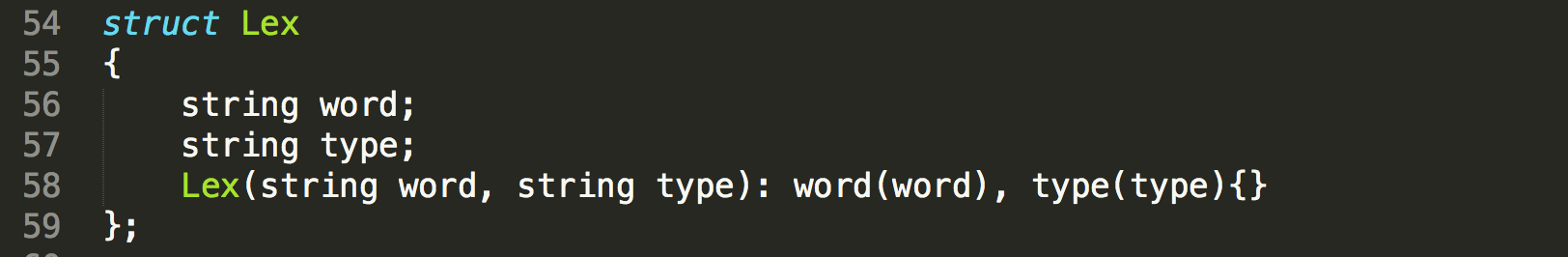
set<string>类型的search，储存预测符，防止重复于是采用了集合的数据结构；

int类型的position，记录当前项中“点”的位置。



### 2 struct Lex

这是用来保存从文件读入的Token序列的结构，有两个string变量，保存值和Token。



### 3 Action与Goto子表

它们均是map<string,int>类型的数组，每一个状态对应一个map，通过这个map的映射可以获取当前状态下读头下遇到的符号对应的跳转的状态，当然也有可能查找不到而报错。



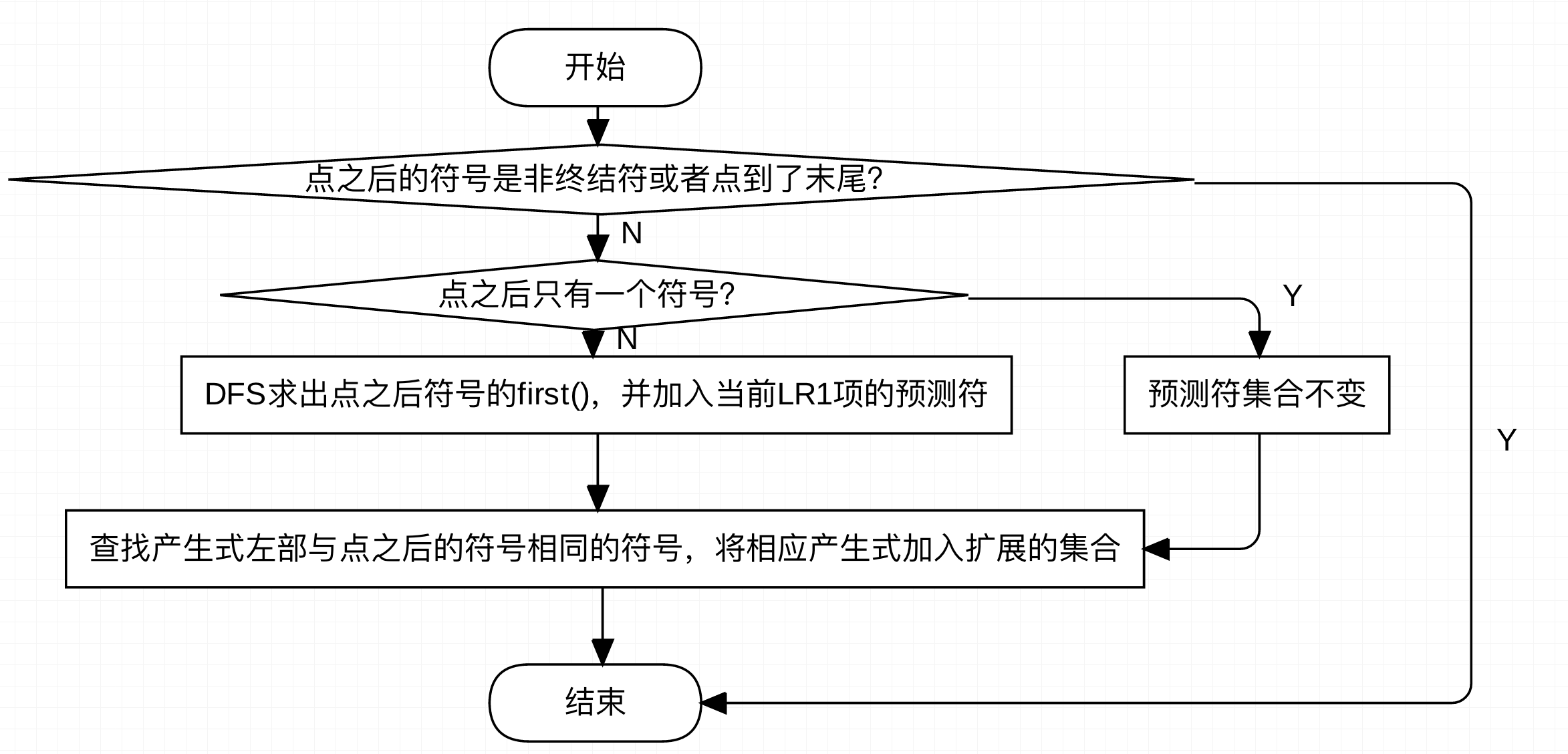
### 4 DFA状态

DFA的状态用vector保存，vector的类型是set<LR1Item>，没个状态都是一个set。



## 七、部分算法描述

### 1 点之后非终结符的扩展



### 2 构造DFA以及Action和Goto子表

首先构造0号产生式以及0号状态，填写Action和Goto子表中的0状态那一行；

对于已有的每一个状态，遍历所有的符号，有可移进的移点并进行扩展，得到一个状态；

如果该状态是原I状态中已有的，找出它的序号，填写Action和Goto子表；

如果该状态是一个新状态，加入到I状态中，填写当前处理状态的Action和Goto表中的相应行；

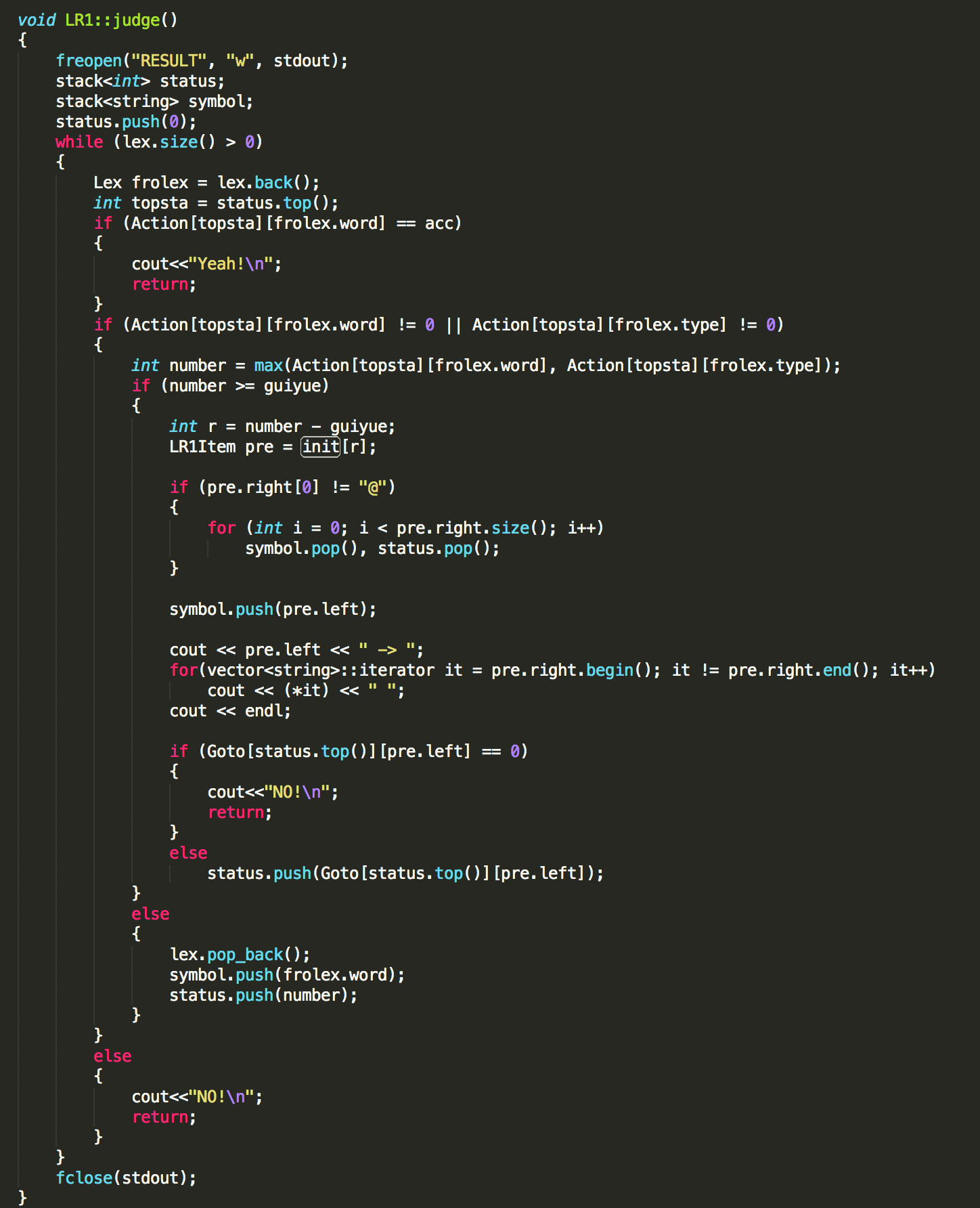
对于新状态处理可以规约的情况。

### 3 DFS

此处dfs实质是上课所讲的求First()，代码中需要对每一个产生式右部的点之后的每一个符号进行处理，如果处理到终结符，将该终结符加入到First集合中即可以返回，如果是非终结符，还要递归求解下去，并且要处理非终结符可能推导出的情况。

### 4 主控程序（monitor）

此处程序的思想就是上课所讲的内容，借助状态栈与符号栈，读入文件中的Token序列，由当前状态与读头下的符号来判定当前动作。



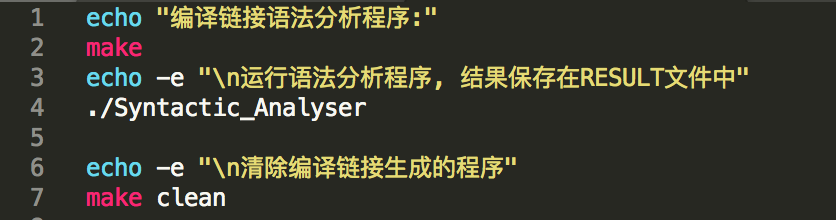
## 八、用例测试

### 1 测试环境

Mac OS 10.12.6, Ubuntu 16.04, clang-900.0.39.2, gcc 5.4.0

### 2 测试脚本

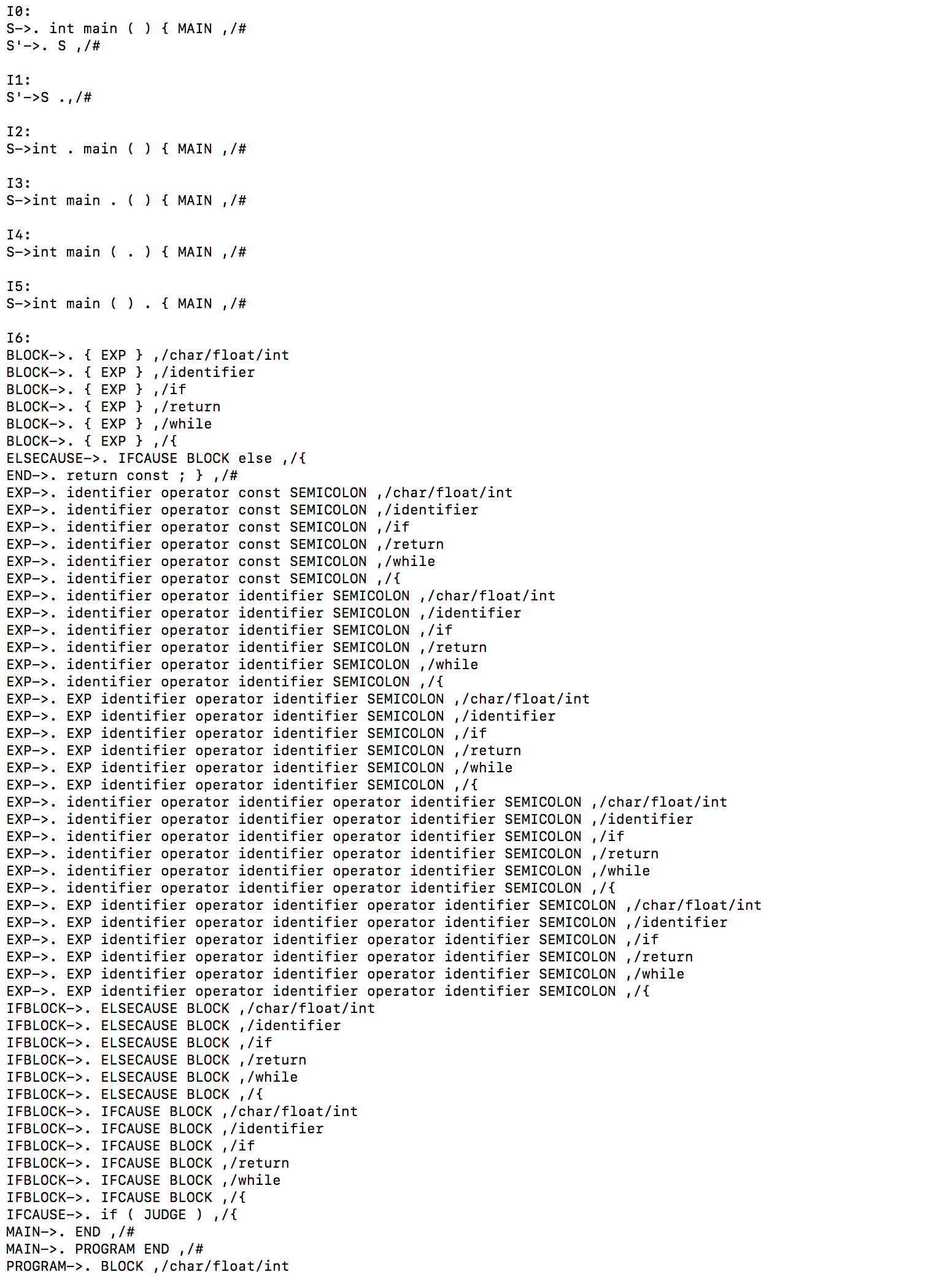
自己写了makefile文件用于编译与链接；同时写了Unix脚本run.sh用于测试，在脚本中首先make进行编译链接，然后运行生成的Syntactic\_Analyser程序，可以看到目录下有RESULT文件，终端里有DFA的输出，最后make clean命令删除之前的.o和.gch以及可执行文件。



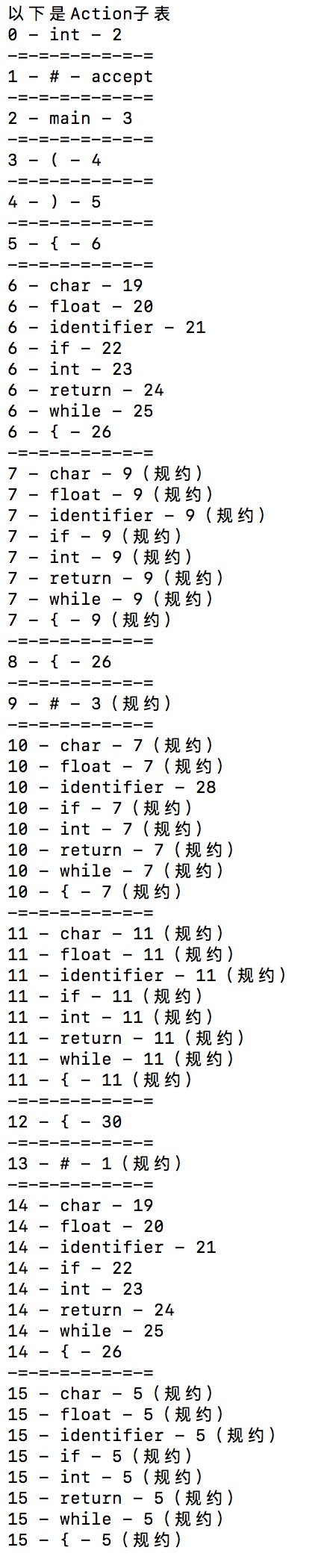
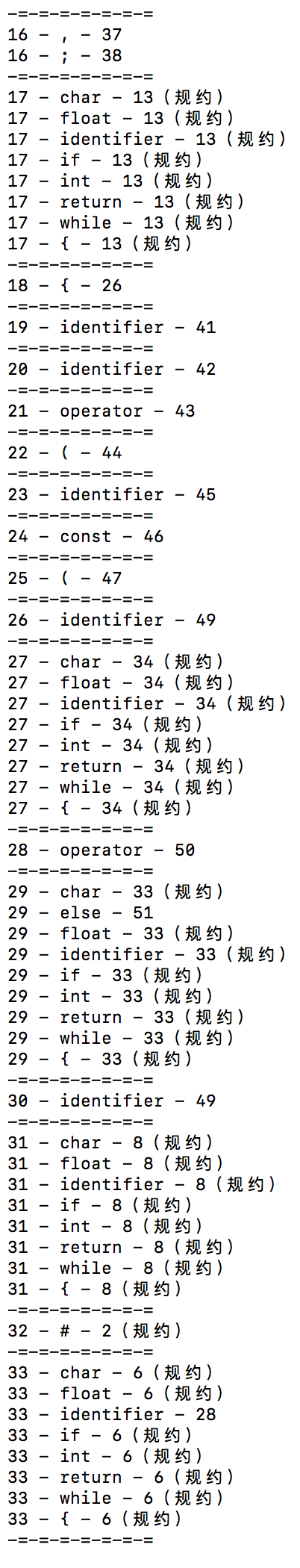
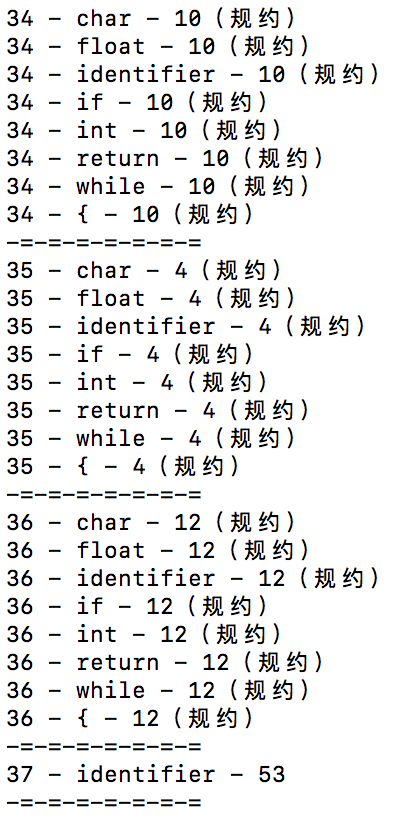
### 3 测试结果

DFA：

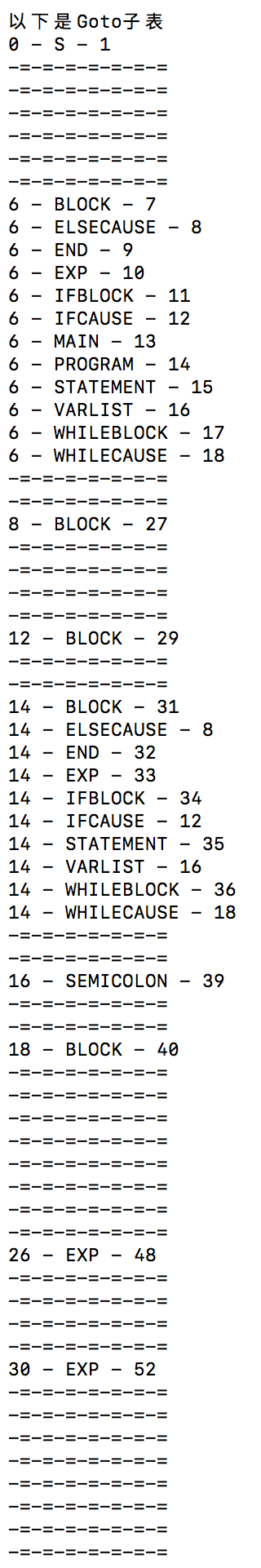
因内容较多，不便于在文档中完全展示，可以运行脚本，在终端中看结果，以下是部分结果：



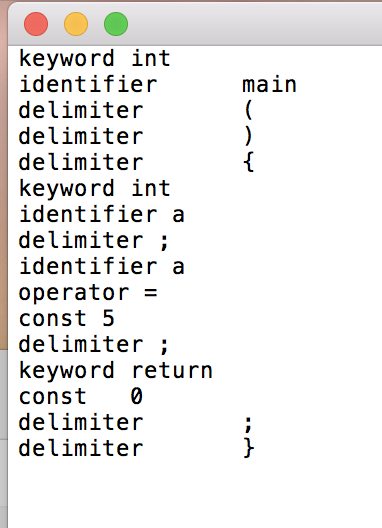
Action子表：

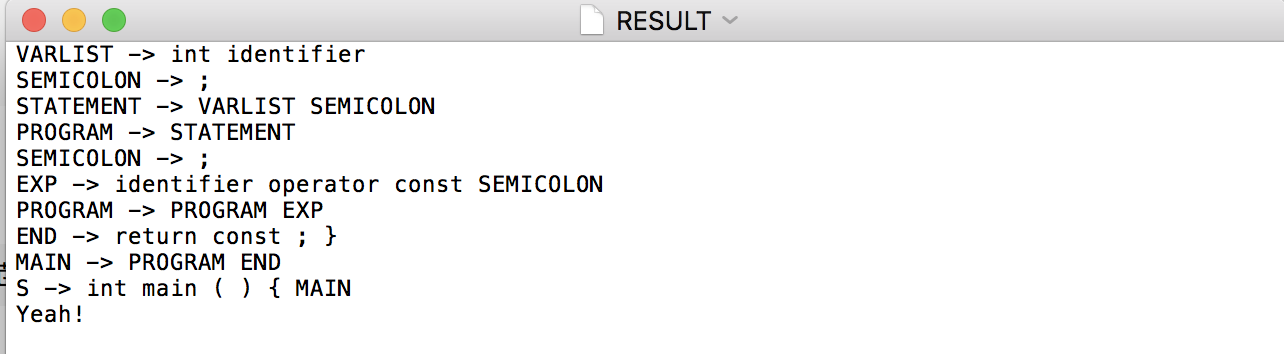
Goto子表：



测试的输入文件：



分析结果：



## 九、困难与解决方案

采用大小写来区分终结符和非终结符为我解析文法省了很多事。

在求预测符的First()时不知道怎样用代码实现，在查阅资料后选择借助深度优先搜索的思路结合上课所讲的思想，只是注意特殊情况的处理。

在实现monitor的时候需要细心，弄清楚什么时候入栈，什么时候出栈就没有太大问题了。

## 十、感受

因时间关系没有去实现Yacc，写了单纯的语法分析程序，但是距离简单的Yacc已经比较近了。有了词法分析器的实现做铺垫，语法分析器的实现更有底气了，二者均为FA-based，涉及很多状态的切换。

在做题目的时候感觉移点，规约，状态的扩展看起来都是那么自然，即使状态多了一点点，耐着性子最后也能做出来，然而计算机来实现就不是那么回事了。人的视角和思维可以有跳跃，但是程序不行，本实验仍未进行优化，直接最暴力的遍历，对每个状态，看文法中的每一个符号对其的影响，循环套循环，但最终还是出结果了，优化的过程还是留待课程设计了。

做完两个实验之后，对于C++ STL中的set，map等等有了一定了解，还认识了pair，果然写程序还是做中学比较有趣，也能有更大进步。