**实验一：词法分析器编程 Lab1: Lexical Analyzer Programming**

09019204

曹邹颖

1. **实验目的Motivation/Aim**

本次词法分析器的编程实验目的在于：

1. 了解词法分析程序的任务是从左至右扫描源程序的字符串，按照此法规则（正规文法规则）识别出一个个正确的单词，并转换该单词成相应的二元式（类号、内码）交语法分析使用；
2. 掌握词法分析程序设计的原理和构造方法;
3. 对编译的基本概念、原理和方法有完整的和清楚的理解,并能正确地、熟练地运用。
4. **实验内容Content description**

实验一要求如下：

1. 输入：字符流，REs（REs的数量自行确定）
2. 输出：token序列
3. 词类号自行定义
4. 可能包括错误处理

确定实验内容如下：

用C++语言实现对C++语言子集的源程序进行词法分析。通过输入源程序从左到右对字符串进行扫描和分解，依次输出各个单词的内部编码及单词符号自身值；若遇到错误则显示“Error”，然后跳过错误部分继续显示；同时进行标识符登记符号表的管理。

1. **实验方法Ideas/Methods**

本次实验采用龙书的3.4.4小节——基于有限自动机的编程方法：

1. 首先自行定义一些正则表达式；
2. 将正则表达式（REs）转换为不确定有限自动机（NFAs），即对待分析的简单的词法（关键词/标识符/整常数/运算符/界限符等）先分别建立自己的FA；
3. 再将这些NFA合并为单个NFA，即将各个FA用ε产生式连接起来并设置一个唯一的开始符，终结符不合并；
4. 然后将该NFA转换为具有最小状态的DFAo，即NFA的确定化加上DFA的简化；
5. 最后基于DFAo程序设计。
6. **实验假设Assumptions**

待分析的简单词法：

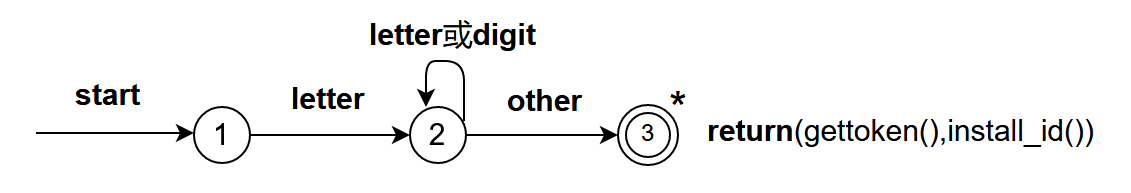
1. 关键词：

"auto"，"int"，"float"，"double"，"char"，"if"，"else"，"switch"，"case"，"break"，"continue"，"while"，"do"，"bool"...

1. 标识符：ID=letter(letter|digit)\*

26位小写字母，"fun"，"number"，"value"，"set"，"ptr"，"str"，"count"，"next"，"pre"，"words"，"keyword"，"num"，"fp"，"name"......

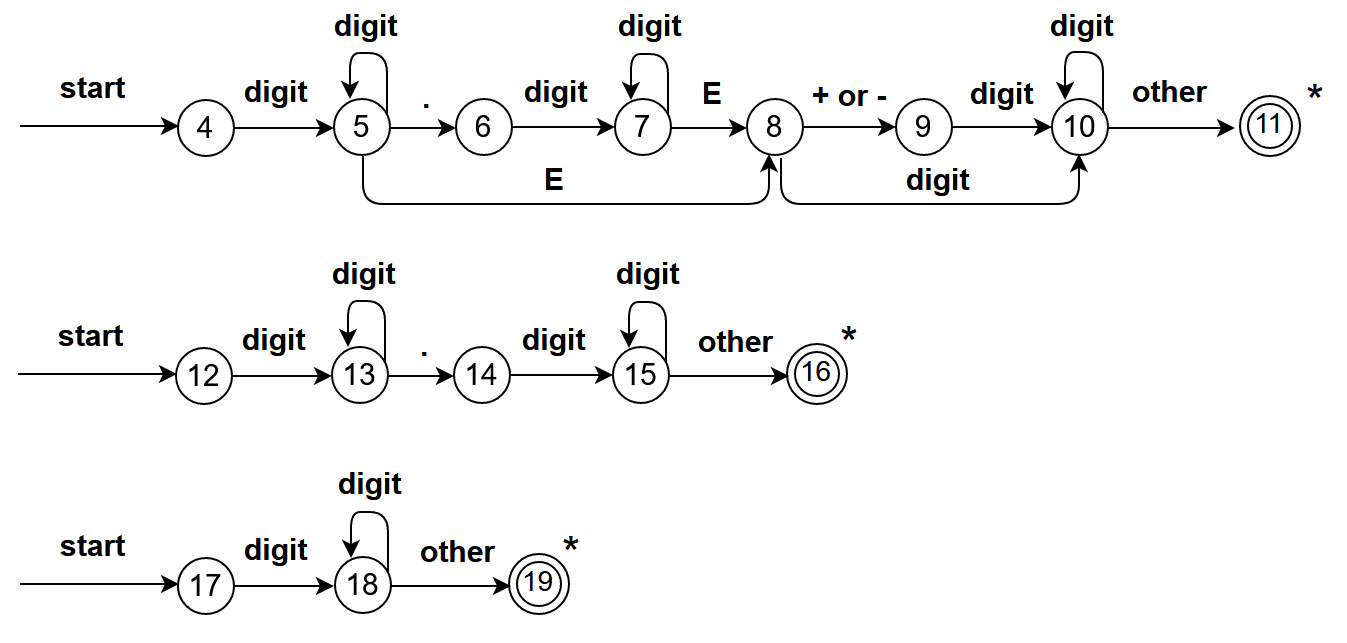
Transition Diagram for Identifier and Keywords：



1. 整常数：整数、有符号整数、无符号整数

0-9,......

Transition Diagram for Unsigned Numbers



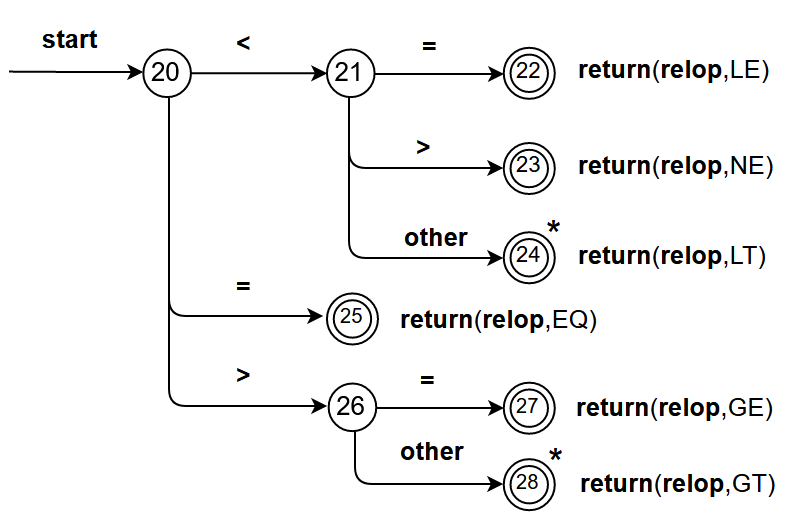
而C++语言中不支持科学计数法的常数输入；

1. 运算符：

"+"，"-"，"\*"，"/"，"%"，"++"，"--"，"+="，"-="，"\*="，"/="，"%="，"&&"，"||"，

"<="，"=="，">="，"<<"，"="，">>"，"!"，"&"，"|"，"^"，"!="，"&="，"|="，"^="......

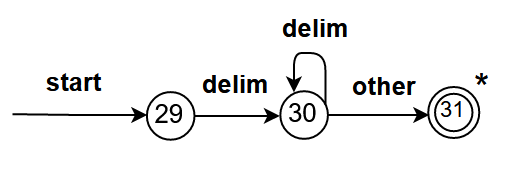
Transition Diagram for Relational Operators



1. 分隔符："("，"["，")"，"]"，","，"{"，";"，"}".....

其中空格、制表符和换行符这三者在词法分析阶段通常被忽略。

Transition Diagram for Separators



由此，相关单词类号与内码建表如下：

|  |  |  |
| --- | --- | --- |
| 类型type | 类号classNumber | 内码internalCode |
| 关键词keywords | 1 | keywords在关键词表中的位置 |
| 标识符id | 2 | id在符号表中的位置 |
| 整常数constantNumber | 3 | num在常数表中的位置 |
| 运算符op | 4 | op运算符表中的位置 |
| 分割符delim | 5 | 分隔符在分隔符表中的位置 |

关键词keywords：

|  |  |  |  |
| --- | --- | --- | --- |
| 单词符号word | 内码 | 单词符号word | 内码 |
| asm | 1 | namespace | 32 |
| auto | 2 | new | 33 |
| bool | 3 | operator | 34 |
| break | 4 | private | 35 |
| case | 5 | protected | 36 |
| catch | 6 | public | 37 |
| char | 7 | register | 38 |
| class | 8 | reinterpret\_cast | 39 |
| const | 9 | return | 40 |
| const\_cast | 10 | short | 41 |
| continue | 11 | signed | 42 |
| default | 12 | sizeof | 43 |
| delete | 13 | static | 44 |
| do | 14 | static\_cast | 45 |
| double | 15 | struct | 46 |
| dynamic | 16 | switch | 47 |
| else | 17 | template | 48 |
| enum | 18 | this | 49 |
| explicit | 19 | throw | 50 |
| export | 20 | true | 51 |
| extern | 21 | try | 52 |
| false | 22 | typedef | 53 |
| float | 23 | typeid | 54 |
| for | 24 | typename | 55 |
| friend | 25 | union | 56 |
| goto | 26 | unsigned | 57 |
| if | 27 | using | 58 |
| inline | 28 | virtual | 59 |
| int | 29 | void | 60 |
| long | 30 | volatile | 61 |
| mutable | 31 | while | 62 |

标识符id：

|  |  |  |  |
| --- | --- | --- | --- |
| 单词符号word | 内码 | 单词符号word | 内码 |
| a | 1 | main | 27 |
| b | 2 | fun | 28 |
| c | 3 | number | 29 |
| d | 4 | value | 30 |
| e | 5 | state | 31 |
| f | 6 | width | 32 |
| g | 7 | height | 33 |
| h | 8 | max | 34 |
| i | 9 | min | 35 |
| g | 10 | ptr | 36 |
| k | 11 | str | 37 |
| l | 12 | token | 38 |
| m | 13 | words | 39 |
| n | 14 | constant | 40 |
| o | 15 | pre | 41 |
| p | 16 | next | 42 |
| q | 17 | count | 43 |
| r | 18 | example | 44 |
| s | 19 | comment | 45 |
| t | 20 | keyword | 46 |
| u | 21 | set | 47 |
| v | 22 | fp | 48 |
| w | 23 | name | 49 |
| x | 24 | type | 50 |
| y | 25 | num | 51 |
| z | 26 | letter | 52 |

整常数constantNumber：

|  |  |  |  |
| --- | --- | --- | --- |
| 单词符号word | 内码 | 单词符号word | 内码 |
| 0 | 0 | 5 | 5 |
| 1 | 1 | 6 | 6 |
| 2 | 2 | 7 | 7 |
| 3 | 3 | 8 | 8 |
| 4 | 4 | 9 | 9 |

运算符op：

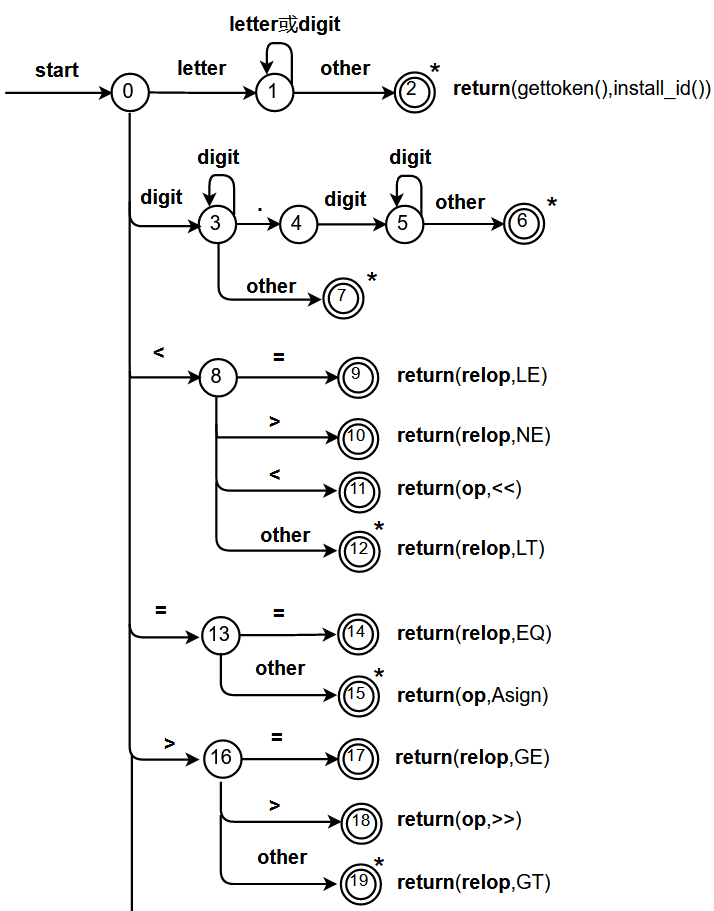
|  |  |  |  |
| --- | --- | --- | --- |
| 单词符号word(Op) | 内码 | 单词符号word(relop) | 内码 |
| + | 1 | ! | 22 |
| ++ | 2 | != | 23 |
| += | 3 | && | 24 |
| - | 4 | || | 25 |
| -- | 5 | ~ | 26 |
| -= | 6 | < | 27 |
| \* | 7 | <= | 28 |
| \*= | 8 | <> | 29 |
| / | 9 | > | 30 |
| /= | 10 | >= | 31 |
| % | 11 | == | 32 |
| %= | 12 |  |  |
| | | 13 |  |  |
| |= | 14 |  |  |
| & | 15 |  |  |
| &= | 16 |  |  |
| ^ | 17 |  |  |
| ^= | 18 |  |  |
| << | 19 |  |  |
| >> | 20 |  |  |
| = | 21 |  |  |

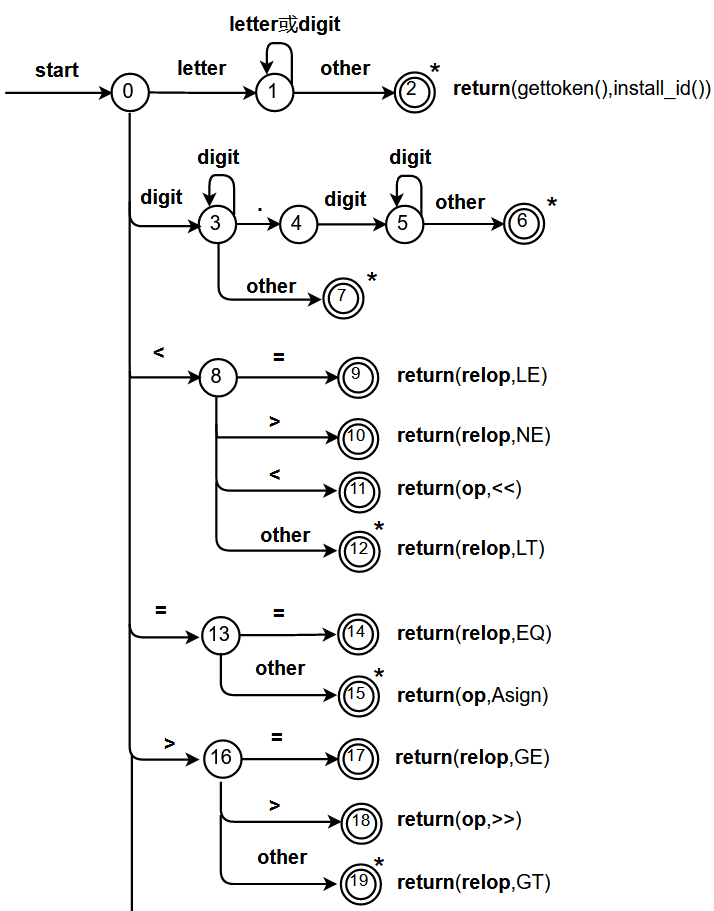
分割符delim：

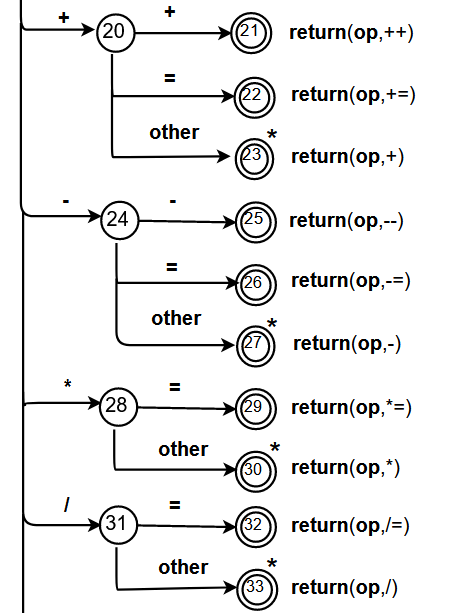
|  |  |  |  |
| --- | --- | --- | --- |
| 单词符号word | 内码 | 单词符号word | 内码 |
| ( | 1 | [ | 5 |
| ) | 2 | ] | 6 |
| , | 3 | { | 7 |
| ; | 4 | } | 8 |

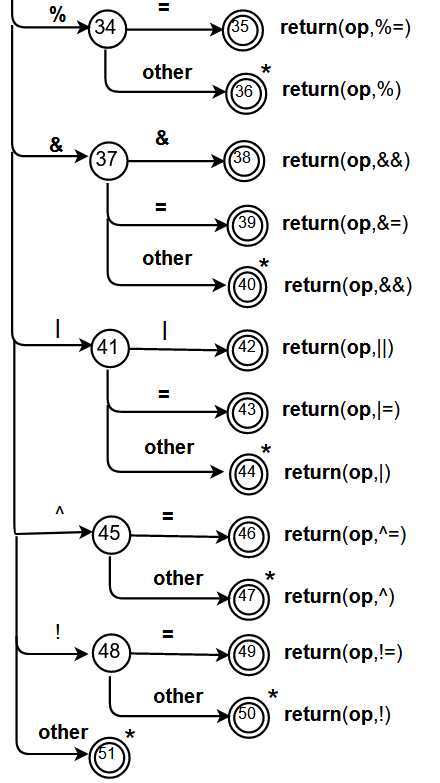
1. **相关自动机描述Related FA descriptions**

将各个NFA合并为单个NFA：

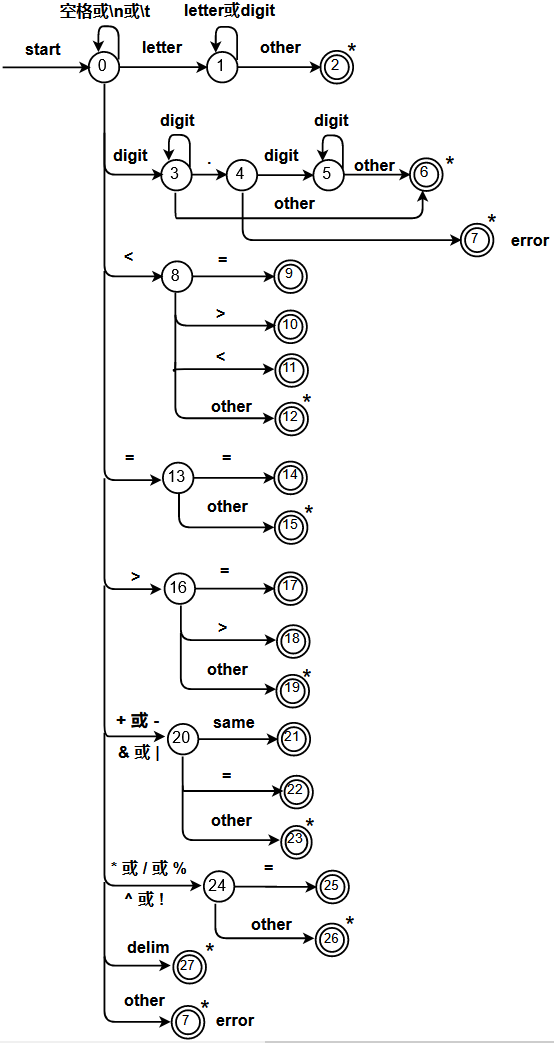








将该NFA转换为具有最小状态的DFAo：



1. **核心数据结构描述Description of important Data Structures**
2. 生成的token序列由word、type、classNumber、internalCode组成，利用struct构建，由于大小未知，采用vector保存。

struct token {

string word; //单词

string type; //类型

int classNumber; //类号

int internalCode; //内码

};

vector<token>Token;

1. 单词的类号、内码表的建立，全部用map保存，这样便于查找，提高效率。

map<string, int>Keywords;

map<string, int>Identifier;

map<string, int>constantNumber;

map<string, int>Op;

map<string, int>Delim;

1. **核心算法描述Description of core Algorithms**
2. void lexical\_analyzer()

词法分析器，从测试文件中按字符读入，利用switch...case...结构将DFAo状态转移刻画出来，从状态0开始处理，如果是回车、换行、制表符、空格则维持状态0，如果第一个字符为字母，则转为状态1；若第一个字符为数字，则转为状态2，详见状态转换图以及打包代码（太长就不粘贴进报告了）。而每次分析出来一个token都要利用addToken()函数将其添加到Token表中并回到状态0，继续往下处理。

1. void addToken(string word, int classNumber)

将分析出来的token（string word）添加到Token序列表中，如果类型（int classNumber）是1，查看关键词表，若未找到再查标识符表，类型是2查标识符表，找不到但满足标识符特征的也可添加进表中附以内码，类型3同理类型2，类型4查运算符表，类型5查分隔符表，其他类型就记作词法错误，将错误信息记录下来。

void addToken(string word, int classNumber)

{

map<string, int>::iterator iter;

token t;

switch (classNumber) {

case 1: //查看关键词表

iter = Keywords.find(word);

if (iter != Keywords.end()) {

t = { word,"keywords",1,iter->second };

Token.push\_back(t);

}

else { //查找是否为id

iter = Identifier.find(word);

if (iter == Identifier.end()) { //未在id表中找到

Identifier[word] = idNum; //添加到id表中

t = { word,"identifier",2,idNum++ };

Token.push\_back(t);

}

else {

t = { word,"identifier",2,iter->second };

Token.push\_back(t);

}

}

break;

case 2:

iter = Identifier.find(word);

if (iter != Identifier.end()) {

t = { word,"identifier",2,iter->second };

Token.push\_back(t);

}

else {

Identifier[word] = idNum; //添加到id表中

t = { word,"identifier",2,idNum++ };

Token.push\_back(t);

}

break;

case 3:

iter = constantNumber.find(word);

if (iter != constantNumber.end()) {

t = { word,"constant\_number",3,iter->second };

Token.push\_back(t);

}

else {

constantNumber[word] = nNum;

t = { word,"constant\_number",3,nNum++ };

Token.push\_back(t);

}

break;

case 4:

iter = Op.find(word);

if (iter != Op.end()) {

t = { word,"operator",4,iter->second };

Token.push\_back(t);

}

break;

case 5:

iter = Delim.find(word);

if (iter != Delim.end()) {

t = { word,"delimeter",5,iter->second };

Token.push\_back(t);

}

break;

default: //error

t = { word,"lexical error",-1,-1 };

Token.push\_back(t);

break;

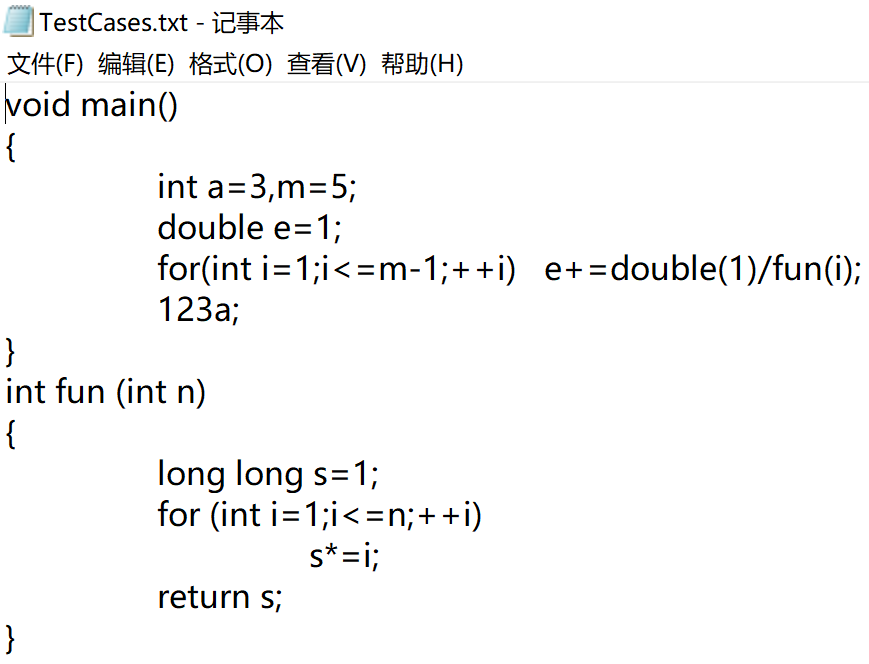
}

}

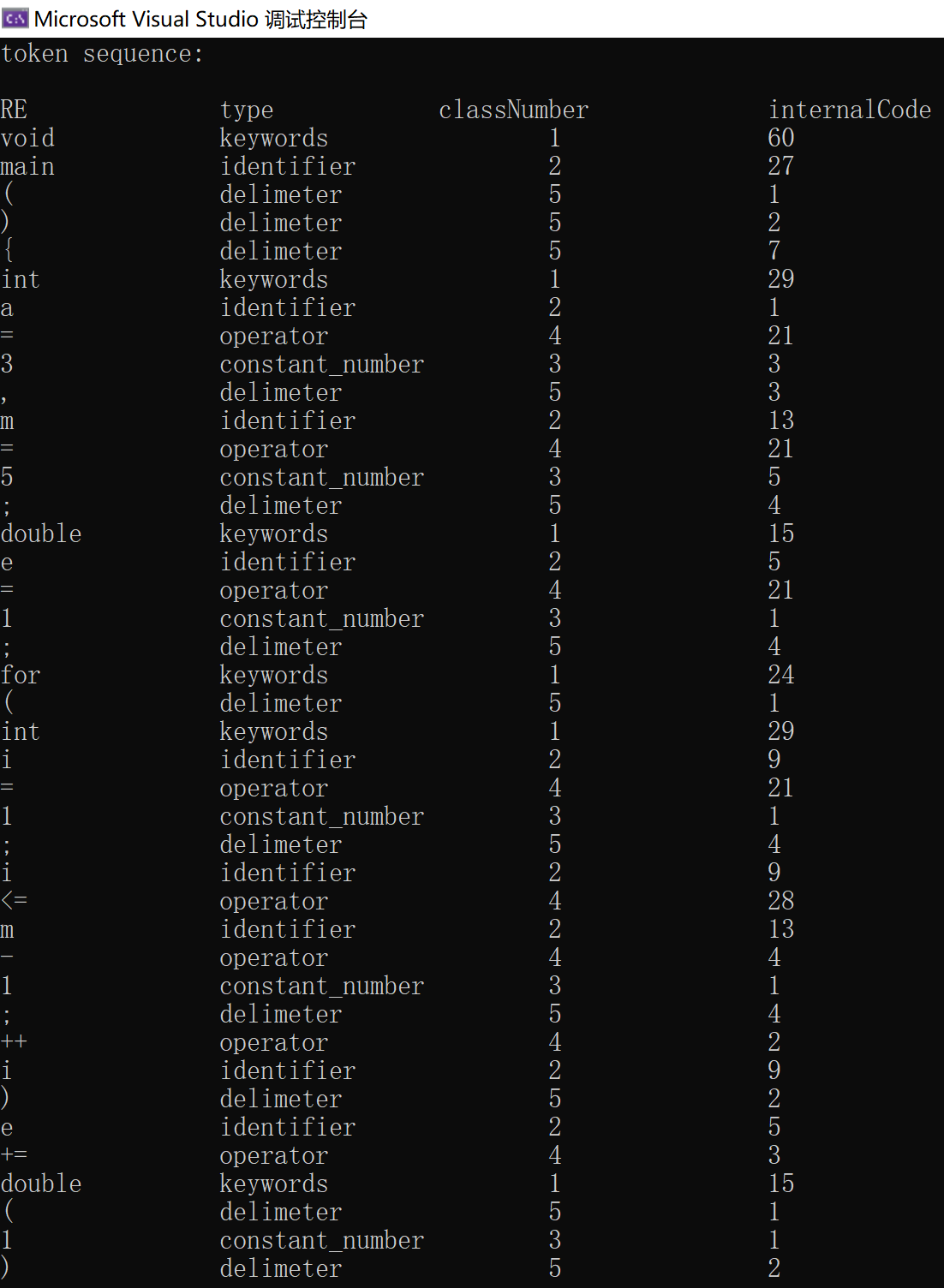
1. **测试用例Use cases on running**

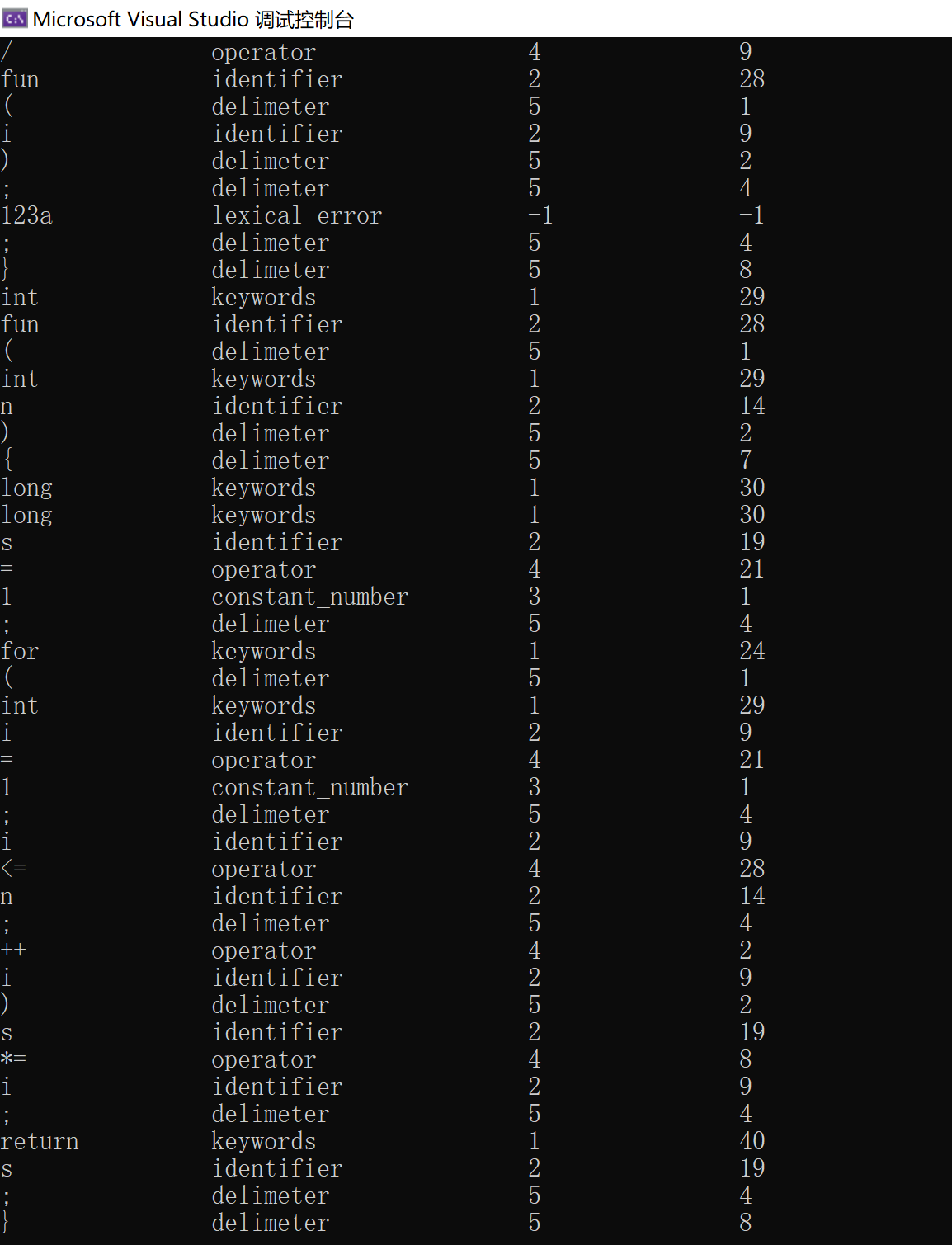
待测字符串放在TestCases.txt文件中，

写了一个简单的C++语言用例如下：



输出打印结果如下：





可见，例如123a;这样的词法错误也检测出来了。

1. **出现的问题与解决方案Problems occurred and related solutions**

**出现的问题有：**

1. 对待分析的简单的词法（关键词/标识符/整常数/运算符/界限符等）先分别建立自己的FA时，判断终态上是否要带星号\*表示将输入回退一个位置的设计；
2. 将多个FA合并并化简为一个具有最小状态的DFA，不明确哪些状态可以合并；
3. 代码中出现死循环没有输出；

**解决方案为：**

1. 通过实际生活中的C++语言去设想如果某类词法输入other后是否需要回退，回退后判断的单词二元式是否合理等；
2. 有些状态在化简时不确定是否可以合并，可以保留进行代码编写，在代码层面可能会发现某两个状态(case state)后的代码是（相同或）等价的，从而会发现这两个状态是可以合并，并进一步化简DFA的；
3. 原因是测试文件放错位置，导致读取错误，一直输出换行符，更换到正确目录下即可。
4. **心得体会Your feelings and comments**

通过动手实践，使我对构造词法分析器的基本原理有更为深入的理解和掌握，将词法分析过程的理论运用到实际，更加熟练地构造与化简FA。其中，实验中有一处设计算是比较满意的，对于C+语言的关键词可以建表存储，但是能识别的常量与识别符是无限的，光靠建表存储是不可能囊括的，我们会通过其特征识别并将其分配特有的内码，存储在相应表中，而不是一味的在表中查询内码。不足的是，算法采用的是基于FA的词法分析方法，存在大两switch...case...语句，可能并不高效，对于Lex并没有尝试。不过总的来说，这次实验还是达到了实验初衷，实现了一定的词法分析功能，得到了锻炼。