# ****词法分析器实现****

**一、实验目的**

(1) 巩固有限自动机理论与正规文法，正规式三者之间的关系和相关的知识点。

(2)了解词法分析程序具体的实现方法及所用算法的原理。

(3) 编写相关的词法分析程序实现对 C++语言程序文件的词法分析。

**二、实验内容**

(1) 选择一个你熟悉的程序设计语言，找到它的规范（referrence or standard）。在规范中找到其词法的BNF或正规式描述。

(2) 选择该语言的一个子集（能够构成一个mini的语言，该语言至少能够进行函数调用、控制流语句（分支或循环）、简单的运算和赋值操作。）给出该mini语言的词法的正规文法或正规式。

(3) 手工将该正规文法或正规式转换为NFA。给出转换过程。

(4) 将上述NFA转换为DFA，并最小化。给出具体过程。

(5) 根据最小化后的DFA写出词法分析程序。

(6) 用mini语言写出若干个小程序作为测试用例。用上述词法分析程序对这些小程序进行词法分析，并将测试用例和分析结果保存在输出文件中。

**三、实验过程**

对于程序语言的选择，这里我采用最熟悉的C语言。

**1、C的BNF或正规式描述：【参考自CSDN】**

以C为例， BNF就是巴科特·瑙尔式的缩写， 在计算机的史前时代（1950s)，曾有一位大师，他奠定了现代计算机的基础，在他老人家的诸多成就之中，包括了对形式语言的研究，和发明了高级语言：FORTRAN。 为了纪念他老人家，我们把他提出的一套描述语言的方法叫做BNF。它以递归方式描述语言中的各种成分，凡遵守其规则的程序就可保证语法上的正确性。BNF由于其简洁、明了、科学而被广泛接受，成为描述各种程序设计语言的最常用的工具。

下面是C++的BNF文法，并给出了相关语义的简短解释：

**1.program → declaration-list**

**2.declaration-list → declaration-list declaration | declaration**

**3.declaration → var-declaration | fun-declaration**

程序由声明的列表(或序列)组成，声明可以是函数或变量声明，顺序是任意的。至少必须有 一个声明。接下来是语义限制(这些在C中不会出现)。所有的变量和函数在使用前必须声明 (这避免了向后backpatching引用)。程序中最后的声明必须是一个函数声明，名字为main。 注意，C1缺乏原型，因此声明和定义之间没有区别(像C一样)。

**4.var-declaration → type-specifier ID | type-specifier ID[NUM];**

**5.type-specifier → int | void | float**

变量声明或者声明了简单的整数或浮点类型变量，或者是基类型为整数或浮点的数组变量， 索引范围从0到NUM-1。注意，在C1中仅有的基本类型是整型和空类型。在一个变量声明中， 只能使用类型指示符int。void用于函数声明(参见下面)。也要注意，每个声明只能声明一 个变量。

**6.fun-declaration → type-specifier ID(params) compound-stmt**

**7.params → param-list | void**

**8.param-list → param-list , param | param**

**9.param → type-specifier ID |type-specifier ID[]**

函数声明由返回类型指示符、标识符以及在圆括号内的用逗号分开的参数列表组成，后面跟 着一个复合语句，是函数的代码。如果函数的返回类型是void，那么函数不返回任何值(即 是一个过程)。函数的参数可以是void(即没有参数)，或者一列描述函数的参数。参数后面 跟着方括号是数组参数，其大小是可变的。简单的整型参数由值传递。数组参数由引用来传 递(也就是指针)，在调用时必须通过数组变量来匹配。注意，类型“函数”没有参数。一个 函数参数的作用域等于函数声明的复合语句，函数的每次请求都有一个独立的参数集。函数 可以是递归的(对于使用声明允许的范围)。

1. **compound-stmt → { local-declarations statement-list }**

复合语句由用花括号围起来的一组声明和语句组成。复合语句通过用给定的顺序执行语句序 列来执行。局部声明的作用域等于复合语句的语句列表，并代替任何全局声明。

**11.local-declarations → local-declarations var-declaration | empty**

**12.statement-list → statement-list statement | empty**

注意声明和语句列表都可以是空的(非终结符empty表示空字符串，有时写作ε。)

**13.statement → expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt**

**14.expression-stmt → expression; |**

表达式语句有一个可选的且后面跟着分号的表达式。这样的表达式通常求出它们一方的结 果。因此，这个语句用于赋值和函数调用。

**15.selection-stmt → if (expression) statement | if (expression) statement else statement**

if语句有通常的语义：表达式进行计算；非0值引起第一条语句的执行；0值引起第二条语句的执行，如果它存在的话。这个规则导致了典型的悬挂else二义性，可以用一种标准的方法解决：else部分通常作为当前if的一个子结构立即分析(“最近嵌套”非二义性规则)。

**16.iteration-stmt → while (expression) statement**

while语句是C－中唯一的重复语句。它重复执行表达式，并且如果表达式的求值为非0，则 执行语句，当表达式的值为0时结束。

**17.return-stmt → return; | return expression** 返回语句可以返回一个值也可无值返回。函数没有说明为void就必须返回一个值。函数声明

为void就没有返回值。return引起控制返回调用者(如果它在main中，则程序结束)。

**18.expression → var=expression | simple-expression**

**19.var→ ID | ID[expression]**

表达式是一个变量引用，后面跟着赋值符号(等号)和一个表达式，或者就是一个简单的表达 式。赋值有通常的存储语义：找到由var表示的变量的地址，然后由赋值符右边的子表达式 进行求值，子表达式的值存储到给定的地址。这个值也作为整个表达式的值返回。var是简 单的(整型)变量或下标数组变量。负的下标将引起程序停止(与C不同)。然而，不进行下标 越界检查。

var表示C1比C的进一步限制。在C中赋值的目标必须是左值(l-value)，左值是可以由许多操

作获得的地址。在C1中唯一的左值是由var语法给定的，因此这个种类按照句法进行检查， 代替像C中那样的类型检查。故在C1中指针运算是禁止的。

**20.simple-expression → additive-expression relop additive-expression |additive-expression**

**21.relop → <= | < | > | >= | == | != | && | ||**

简单表达式由无结合的关系操作符组成(即无括号的表达式仅有一个关系操作符)。简单表达 式在它不包含关系操作符时，其值是加法表达式的值，或者如果关系算式求值为ture，其值 为1，求值为false时值为0。

**22.additive-expression → additive-expression addop term | term**

**23.addop → + | -**

**24.term → term mulop factor | factor**

**25.mulop → \* | /**

加法表达式和项表示了算术操作符的结合性和优先级。/符号表示整数除；即任何余数都被 截去。

**26.factor → (expression) | var | call | NUM**

因子是围在括号内的表达式；或一个变量，求出其变量的值；或者一个函数调用，求出函数 的返回值；或者一个NUM，其值由扫描器计算。数组变量必须是下标变量，除非表达式由单个ID组成，并且以数组为参数在函数调用中使用(如下所示)。

**27.call → ID(args)**

**28.args → arg-list | empty**

**29.arg-list → arg-list,expression | expressio**n

函数调用的组成是一个ID(函数名)，后面是用括号围起来的参数。参数或者为空，或者由逗 号分割的表达式列表组成，表示在一次调用期间分配的参数的值。函数在调用之前必须声明， 声明中参数的数目必须等于调用中参数的数目。函数声明中的数组参数必须和一个表达式匹 配，这个表达式由一个标识符组成表示一个数组变量。

**2、mini语言的词法的正规文法或正规式**

词法分析是编译程序进行编译时第一个要进行的任务，主要是对源程序进行编译预处理（去除注释、无用的回车换行找到包含的文件等）之后，对整个源程序进行分解，分解成一个个单词，这些单词有且只有五类，分别是标识符、关键字、常数、运算符、界符。可以说词法分析面向的对象是单个的字符，目的是把它们组成有效的单词（字符串）；而语法的分析则是利用词法分析的结果作为输入来分析是否符合语法规则并且进行语法制导下的语义分析，最后产生四元组(中间代码)，进行优化（可有可无）之后最终生成目标代码。可见词法分析是所有后续工作的基础，如果这一步出错，比如明明是‘<=’却被拆分成‘<’和‘=’就会对下文造成不可挽回的影响。因此，在进行词法分析的时候一定要定义好这五种符号的集合。下面是我构造的一个C语言子集：

**·第一类：**标识符：   letter(letter | digit)\*  无穷集  
·**第二类：**常数：    (digit)+  无穷集  
·**第三类：**关键字：  (32)：auto、break、case、char 、const、continue、 default、do、double 、else、enum、extern 、float 、for、goto、if、int、long、register、return、short、signed、sizeof 、static  、struct、switch、typedef、union、unsigned、void、volatile、while  
·**第四类：**界符：  ‘/\*’、‘//’、 () { } [ ] " "  ' 等  
·**第五类：**运算符： <、<=、>、>=、=、+、-、\*、/、^、等

**有如下定义：**

digit [0~9]

letter [A~Z a~z]

delim [ \n \t]

id {letter}({letter}|{digit}|\_)\*

number {digit}+(.{digit}+)?(E[+-]?{digit}+)?

op [+ - \* / = < <= > >= ^]=?

ws {delim}+

**对所有可数符号进行编码：**  
<$,0>、<auto,1>...<while,32>、<+，33>、<-,34>、<\*,35>、</,36>、<<,37>、<<=,38>、<>,39>、<>=,40>、<=,41>、<==,42>、<!=,43>、<;,44>、<(,45>、<),46>、<^,47>、  
<,,48>、<",49>、<',50>、<#,51>、<&,52>、<&&,53>、<|,54>、<||,55>、<%,56>、  
<~,57>、<<<,58>左移、<>>,59>右移、<[,60>、<],61>、<{,62>、<},63>、<\,64>、  
<.,65>、<?,66>、<:,67>、<!,68>、"[","]","{","}"、<常数99  ,数值>、<标识符100 ，标识符指针>。

上述二元组中左边是单词的符号，右边为其种别码，其中常数和标识符有点特别，因为是无穷集合，因此常数用自身来表示，种别码为99，标识符用标识符符号表的指针表示（当然也可用自身显示，比较容易观察），种别码100。根据上述约定，一旦见到了种别码syn=63,就唯一确定了‘}’这个单词。

**正规式如下：**

1、id -> {letter}({letter}|{digit})\*

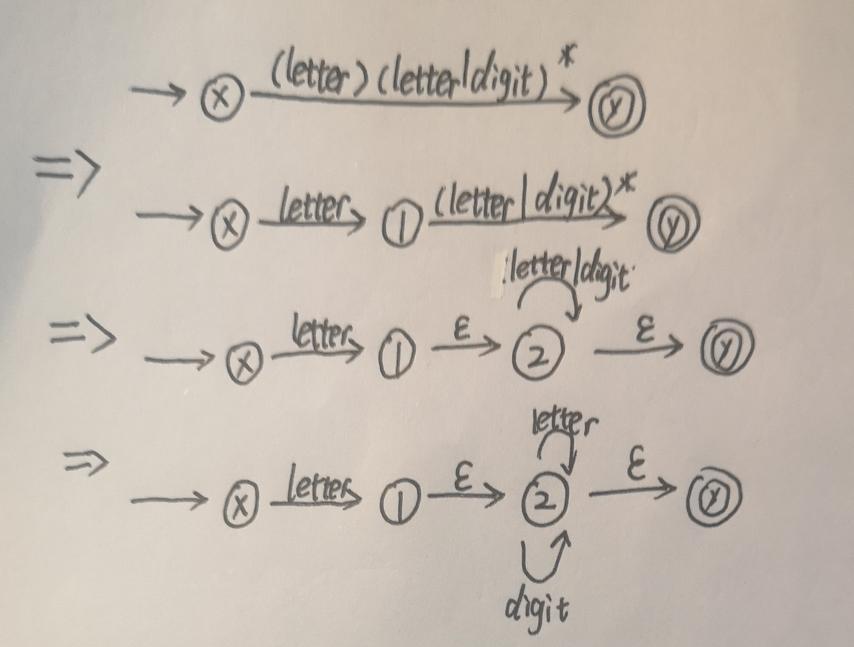
2、number -> {digit}+(.{digit}+)?(E[+-]?{digit}+)?

3、ws -> {delim}+

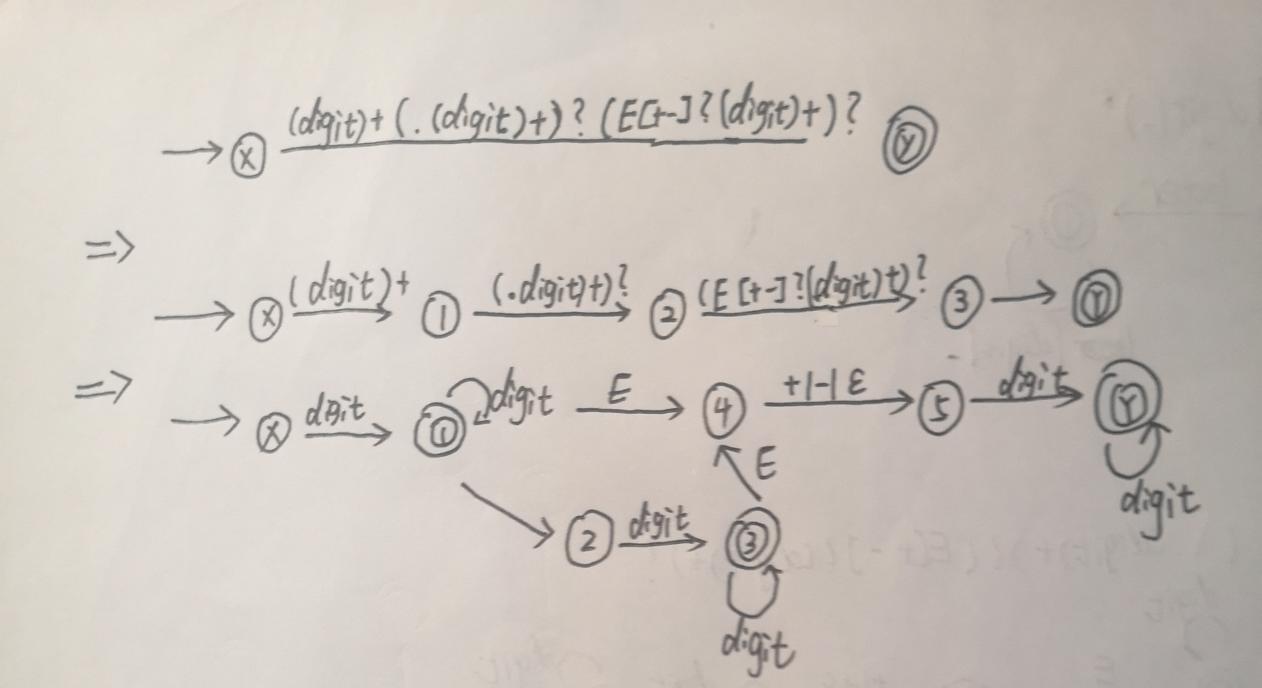
4、op -> [+ - \* / = < <= > >= ^]=?

**3、手工将该正规文法或正规式转换为NFA。给出转换过程。**

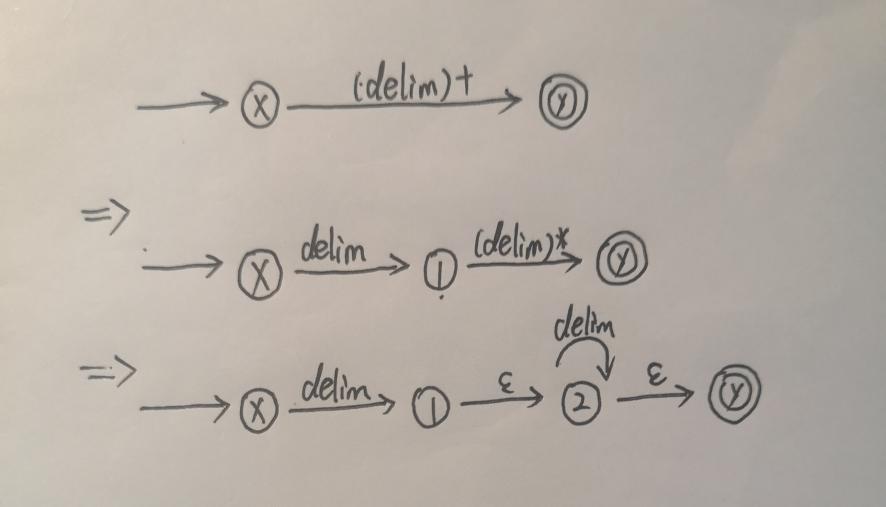
1、NFA of id -> {letter}({letter}|{digit})\*;



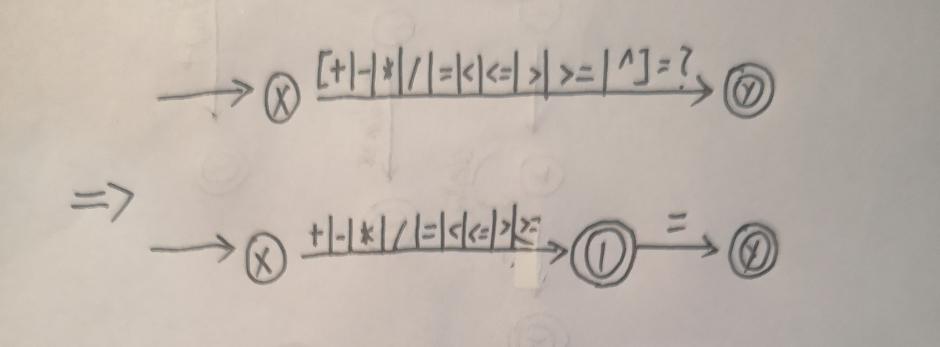
1. NFA of number -> {digit}+(.{digit}+)?(E[+-]?{digit}+)?;



1. NFA of ws -> {delim}+

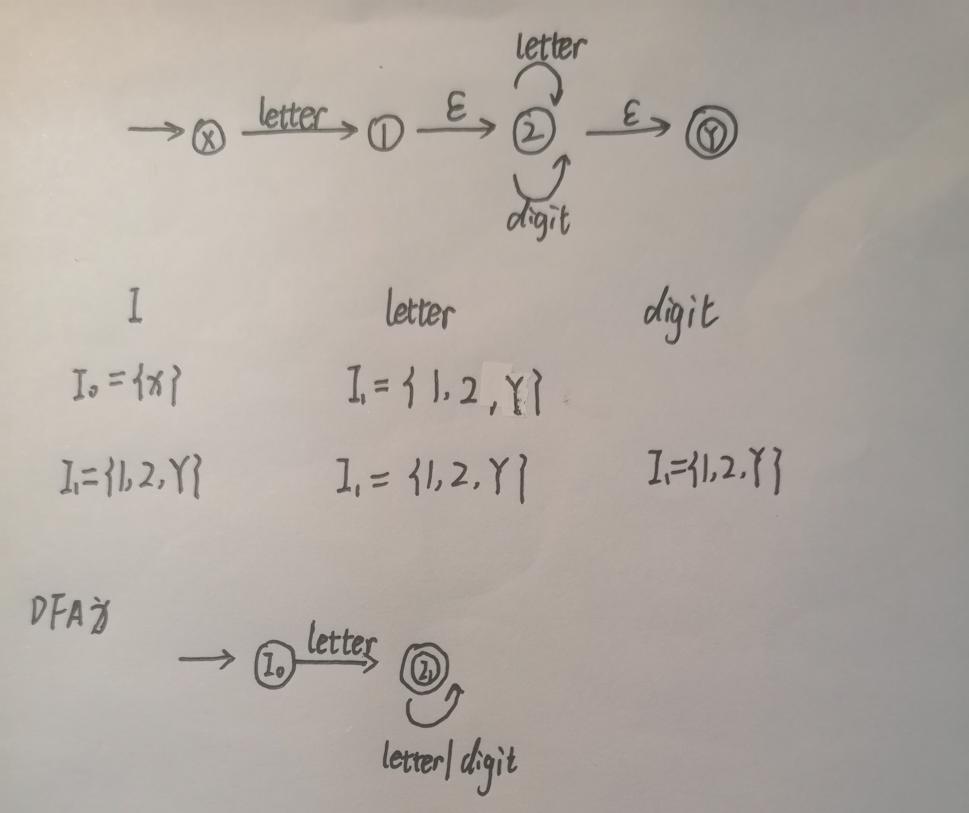


4、NFA of op -> [+ - \* / = < <= > >= ^]=?



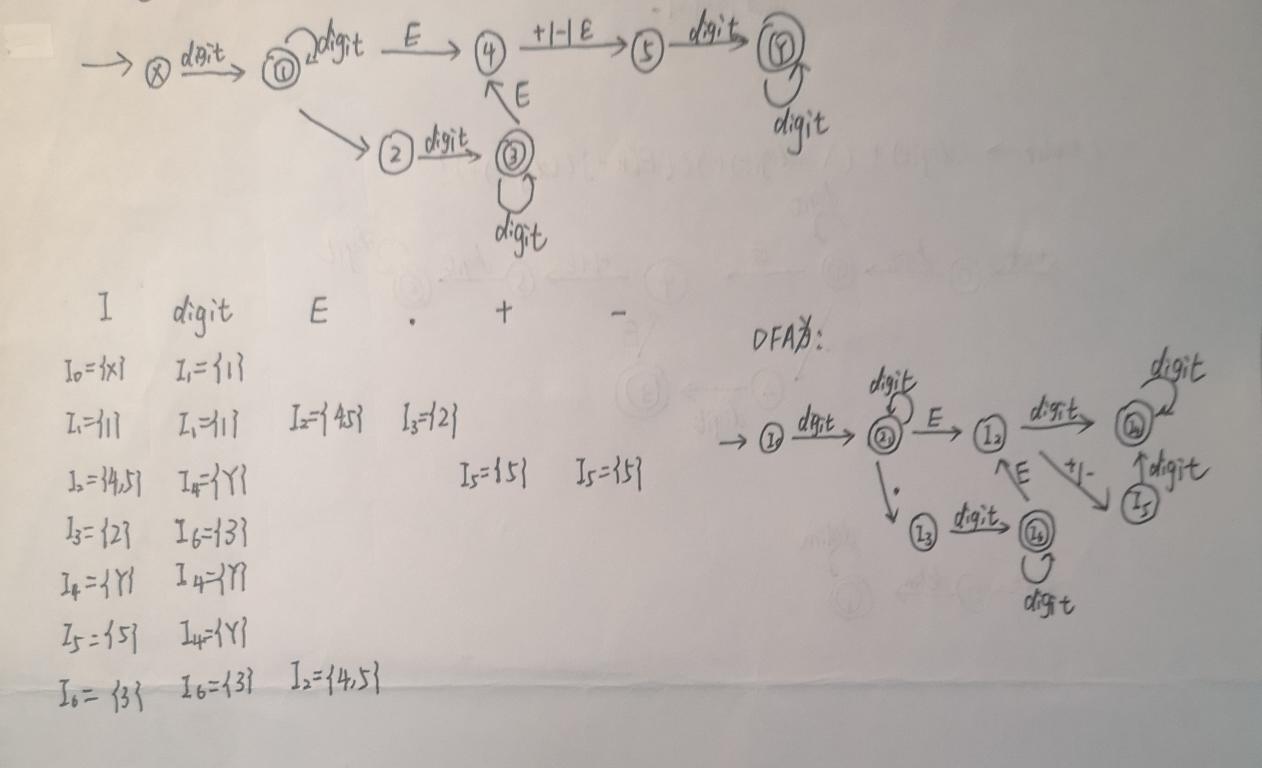
**4、将上述NFA转换为DFA，并最小化。给出具体过程。**

1、DFA of id -> {letter}({letter}|{digit})\*;



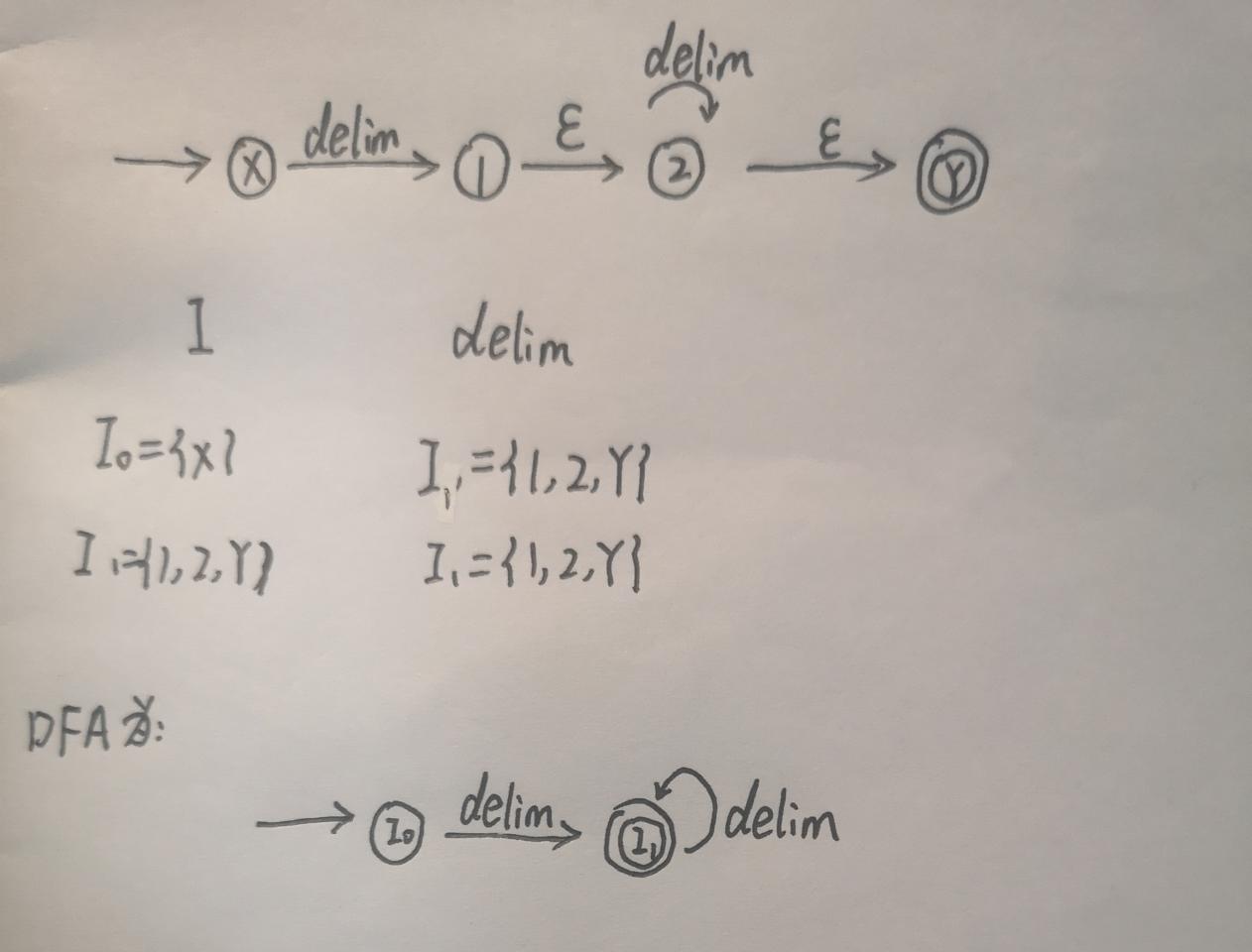
经检验，已经是最简状态。

1. DFA of number -> {digit}+(.{digit}+)?(E[+-]?{digit}+)?;



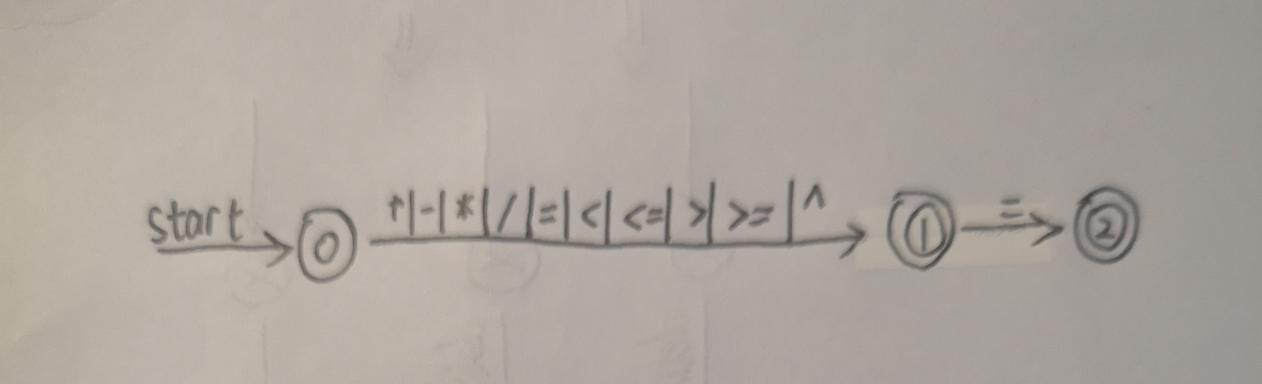
经检验，已经是最简状态。

1. DFA of ws -> {delim}+



经检验，已经是最简状态。

4、DFA of op -> [+ - \* / = < <= > >= ^]=?

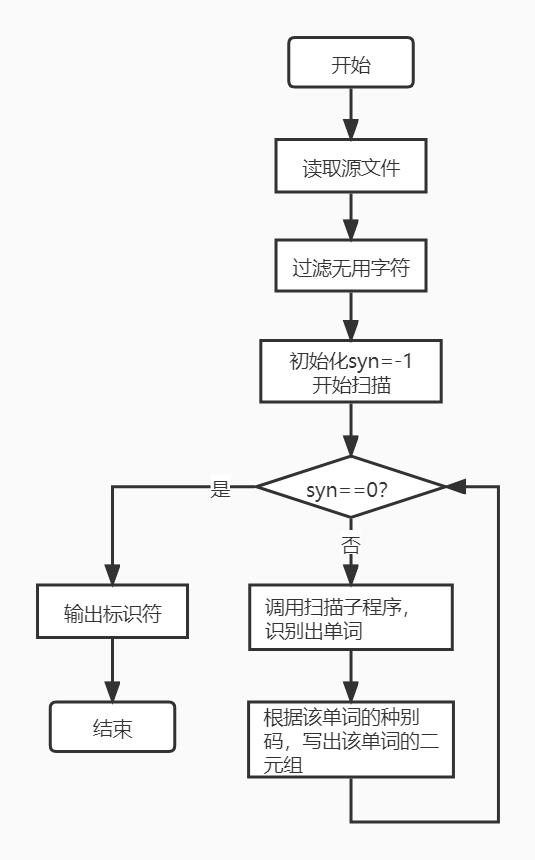
****

经检验，已经是最简状态。

**5、根据最小化后的DFA写出词法分析程序。**

下面是程序的思想：

1. 词法分析程序打开源文件，读取文件内容，直至遇上’$’文件结束符，然后读取结束。  
      2.对读取的文件进行预处理，从头到尾进行扫描，去除//和/\*  \*/的内容，以及一些无用的、影响程序执行的符号如换行符、回车符、制表符等。但是千万注意不要在这个时候去除空格，因为空格在词法分析中有用，比如说int i=3;这个语句，如果去除空格就变成了“inti=3”,这样就失去了程序的本意，因此不能在这个时候去除空格。  
     3.选下面就要对源文件从头到尾进行扫描了，从头开始扫描，这个时候扫描程序首先要询问当前的字符是不是空格，若是空格，则继续扫描下一个字符，直至不是空格，然后询问这个字符是不是字母，若是则进行标识符和保留字的识别；若这个字符为数字，则进行数字的判断。否则，依次对这个字符可能的情况进行判断，若是将所有可能都走了一遍还是没有知道它是谁，则认定为错误符号，输出该错误符号，然后结束。每次成功识别了一个单词后，单词都会存在token[ ]中。然后确定这个单词的种别码，最后进行下一个单词的识别。这就是扫描程序进行的工作，可以说这个程序彻底实现了确定有限自动机的某些功能，比如说识别标识符，识别数字等。为了简单起见，这里的数字只是整数。  
     4.主控程序主要负责对每次识别的种别码syn进行判断，对于不同的单词种别做出不同的反应，如对于标识符则将其插入标识符表中。对于保留字则输出保留字的种别码和助记符，等等吧。直至遇到syn=0;程序结束。



**6、用mini语言写出若干个小程序作为测试用例。用上述词法分析程序对这些小程序进行词法分析，并将测试用例和分析结果保存在输出文件中。**

这里我们用example.txt文件来存储测试用例，对应的结果存在result.txt文件中。

**example1：**

char resourceProject[10000];

char token[20] = { 0 };

int syn = -1, i;//初始化

int pProject = 0;//源程序指针

FILE\* fp, \* fp1;

if ((fp = fopen("example.txt", "r")) == NULL)

{//打开源程序

cout << "can't open this file";

exit(0);

}

resourceProject[pProject] = fgetc(fp);

while (resourceProject[pProject] != '$')

{//将源程序读入resourceProject[]数组

pProject++;

resourceProject[pProject] = fgetc(fp);

}

resourceProject[++pProject] = '\0';

fclose(fp);

$

**example2：**

int a[12];

int i = -15;

double j = 11.04;

int b\_x = 3.14E+10;

char array[8];

int x = i;

if(x == i){

while(i) i = i + 13;

}

return 0;

$

**example3：**

void transform(char \*a)

{

while(\*a)

{

if(\*a>='A'&&\*a<='Z') \*a=\*a-'A'+'a';

else if(\*a>='a'&&\*a<='z') \*a=\*a-'a'+'A';

a++;

}

}

$

**example4：**

if(m==1)

{

int a=1;

double y=0;

while(fabs(t)>=pow(0.1,k))

{

t=pow(x,2\*n-1)\*a/jc(2\*n-1);

a=-a;

y+=t;

n++;

}

cout<<"sin(x)="<<y<<endl;

}

if(m==2)

{

int b=-1;

double y=1;

while(fabs(t)>=pow(0.1,k))

{

t=pow(x,2\*n)\*b/jc(2\*n);

b=-b;

y+=t;

n++;

}

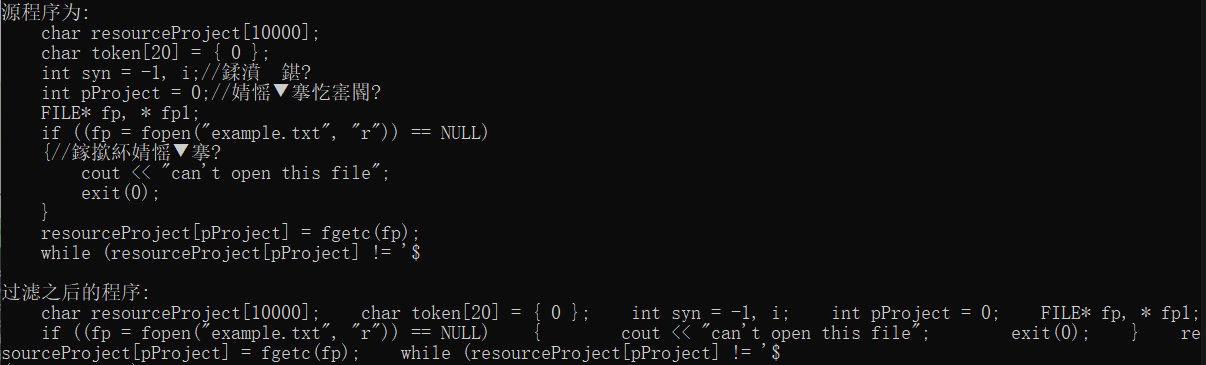
cout<<"cos(x)="<<y<<endl;

}

$

1. **实验结果**

此处我们已example1为例，运行后的结果为：



**输出文件中的token list为：**

(char , 关键字)

(resourceProject , 标识符)

([ , 界符)

( 10000 , 常数)

(] , 界符)

(; , 界符)

(char , 关键字)

(token , 标识符)

([ , 界符)

( 20 , 常数)

(] , 界符)

(= , 运算符)

({ , 界符)

( 0 , 常数)

(} , 界符)

(; , 界符)

(int , 关键字)

(syn , 标识符)

(= , 运算符)

(- , 运算符)

( 1 , 常数)

(, , 界符)

(i , 标识符)

(; , 界符)

(int , 关键字)

(pProject , 标识符)

(= , 运算符)

( 0 , 常数)

(; , 界符)

(FILE , 标识符)

(\* , 运算符)

(fp , 标识符)

(, , 界符)

(\* , 运算符)

(fp1 , 标识符)

(; , 界符)

(if , 关键字)

(( , 界符)

(( , 界符)

(fp , 标识符)

(= , 运算符)

(fopen , 标识符)

(( , 界符)

(" , 界符)

(example , 标识符)

(. , 运算符)

(txt , 标识符)

(" , 界符)

(, , 界符)

(" , 界符)

(r , 标识符)

(" , 界符)

() , 界符)

() , 界符)

(== , 运算符)

(NULL , 标识符)

() , 界符)

({ , 界符)

(cout , 标识符)

(<< , 运算符)

(< , 运算符)

(" , 界符)

(can , 标识符)

(' , 界符)

(t , 标识符)

(open , 标识符)

(this , 标识符)

(file , 标识符)

(" , 界符)

(; , 界符)

(exit , 标识符)

(( , 界符)

( 0 , 常数)

() , 界符)

(; , 界符)

(} , 界符)

(resourceProject , 标识符)

([ , 界符)

(pProject , 标识符)

(] , 界符)

(= , 运算符)

(fgetc , 标识符)

(( , 界符)

(fp , 标识符)

() , 界符)

(; , 界符)

(while , 关键字)

(( , 界符)

(resourceProject , 标识符)

([ , 界符)

(pProject , 标识符)

(] , 界符)

(!= , 运算符)

(' , 界符)

**五、实验总结**

其实这次的实验，算法部分并不难，只要知道了DFA，这个模块很好写，比较麻烦的就是五种类型的字符个数越多程序就越长。但为了能识别大部分程序，我还是用了比较大的子集，结果花了很长时间才写完，同时也加深了对字符的认识。通过这次实验，我对编译原理专业必修课有了进一步深入的了解，将理论知识运用到了实验中，也使我熟悉了C语言增益的相关内容，加深了我对C语言知识的深化和运用的理解。我相信我能在我以后的项目上取得更大的进步。其中感受最深的是学习编译原理必须要做实验，写程序，这样才会提高自己的动手能力，加深自己对难点的理解。