基于 C 语言 do-while 循环语句的中间代码生成

1. 系统描述

1.1 设计目的

通过设计、编制、调试一个 DO-WHILE 循环语句的语法及语义分析程序，加深对法及语义分析原理的理解，并实现词法分析程序对单词序列的词法检查和分析。

1.2 设计内容及步骤

对循环语句: do〈赋值语句〉while〈表达式〉

(1) 按给定的题目写出符合自身语法分析方法要求的文法和属性文法描述。

(2) 按给定的题目给出语法分析方法的思想及分析表设计。

(3) 按给定的题目给出中间代码序列的结构设计。

(4) 完成相应的词法分析、语法分析和语义分析程序设计。

(5) 编写好分析程序后，依据给出的测试用例，上机测试并通过所设计的分析程序。

1.3 实验内容

(1) 根据给出的样例程序，补充填写部分并运行;

(2) 根据所设计的分析程序和所给定的测试用例，自行构造识别活前缀的 DFA、ACTION 表和 GOTO 表，上机测试并通过所设计的分析程序。

2. 文法及属性文法的描述

2.1 文法描述

（0） S'→S　　（1）S→doAwhileB;

（2）A→{F}　　（3）F→M

（4）F→FM　　（5）M→i=E;

（6）E→E+i　　（7）E→E-i

（8）E→E*i　　（9）E→E/i

（10） E→i　　（11）E→(E)

（12）B→(iOi)　　（13）O→<

（14）O→<=　　（15）O→>

（16）O→>=　　（17）O→!=

2.2 属性文法的描述

产生式	属性文法
S -> doAwhileB	S.begin:=newlabel; S.next:=newlabel; B.true:=newlabel; B.false:=S.next; A.next:=S.begin; S.code:=gen(S.begin, ‘:’)|| B.code||gen(S.true, ‘:’) ||A.code||gen(‘goto’ ,S.begin)|| gen(B.false, ‘:’)||gen(‘goto A.next’);
A→{F}	E.place:=F.place; E.code:=F.code
F→M	F.place:=M.place; F.code:=M.code
F→FM	F.place:=newlabel; F.code:=F.code || F.code gen(F.place:=F.place, M.place);
M→i=E;	S.code:=E.code || gen(i.place ‘:=’ E.place)
E→E1+i	E.place:=newtemp; E.code:=E1.code || i.code ||gen(E.place ‘:=’ E1.place ‘+’ i.place)

E→E1-i	E.place:=newtemp; E.code:=E1.code || E2.code ||gen(E.place ‘:=’ E1.place ‘-’ E2.place)
E→E1*i	E.place:=newtemp; E.code:=E1.code || i.code ||gen(E.place ‘:=’ E1.place ‘*’ i.place)
E→E1/i	E.place:=newtemp; E.code:=E1.code || E2.code ||gen(E.place ‘:=’ E1.place ‘/’ E2.place)
E→i	E.place:=i.place; E.code= ‘ ’
E→(E1)	E.place:=E1.place; E.code:=E1.code
B→(i1Oi2)	B.TC=NXQ; B.FC=NXQ+1; gencode(jrop,i1.PLACE,i2.PLACE,0) gencode(j, _, _, 0)
O→<|<=|>|>=|!=	O.val=Lexval

3 语法分析表的设计

3.1 各非终结符的 FIRST 集合:

FIRST (S) ={do}

FIRST (A) ={ { }

FIRST (F) ={ i }

FIRST (M) ={ i }

FIRST (E) ={ I , (}

FIRST (B) ={ (}

FIRST (O) ={ <, <= , > , >= , != }

3.2 各非终结符的 FOLLOW 集合:

FOLLOW（S’）= { # }

FOLLOW（S）= { # }

FOLLOW（A）= { while }

FOLLOW（F）= { } , i}

FOLLOW（M）= { } , i }

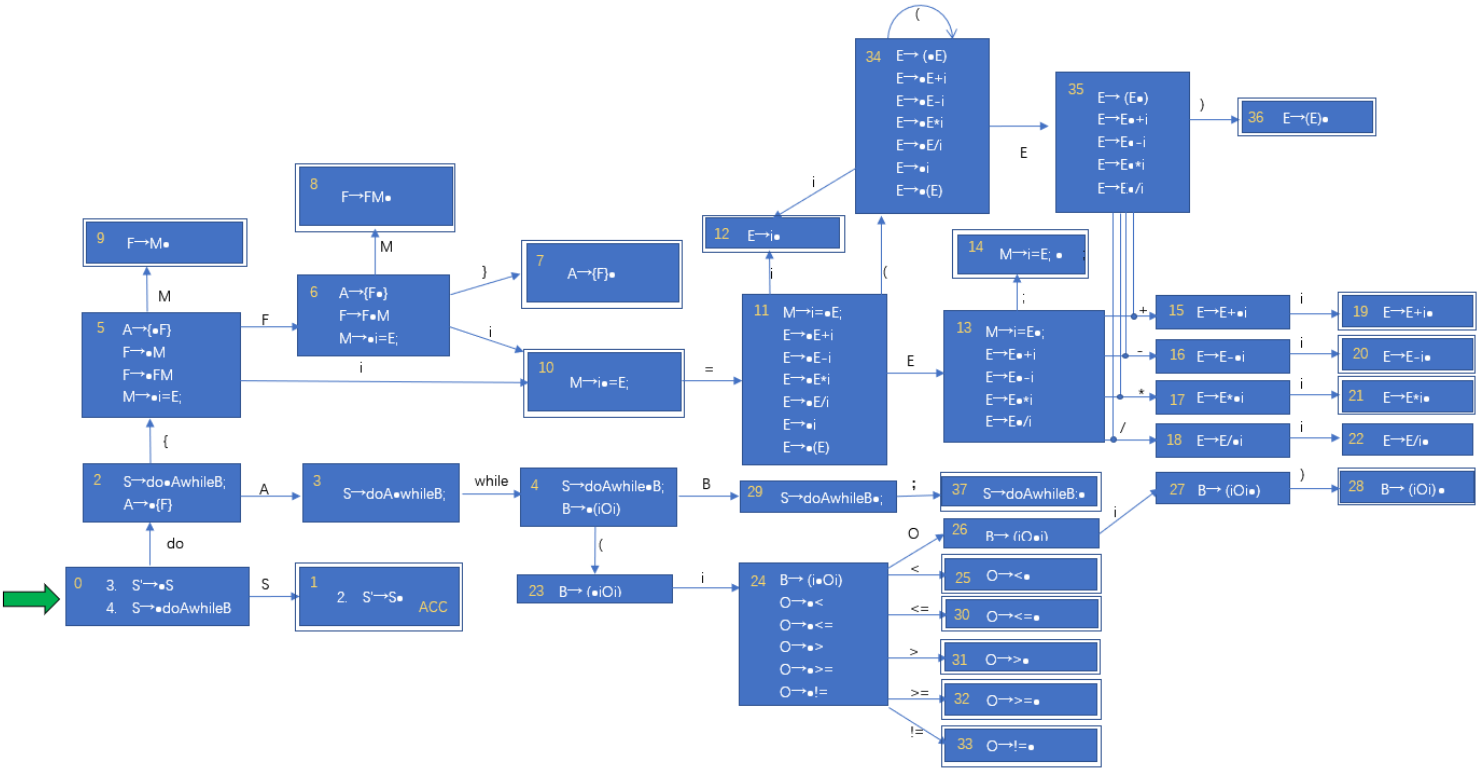
FOLLOW（E）= { +,-,*,/,; ,)}

FOLLOW（B）= { ; }

FOLLOW（O）= { i }

3.3 DFA

构造识别文法活前缀的 DFA 如图所示：



3.4 构造预测分析表

根据 DFA 和各非终结符的 FOLLOW 集构造文法的 SLR(1)分析表如表 1 和表 2 所示：

表 1：ACTION 表

	do	while	i	=	+	-	*	/	;	{	}	(	)	#	<	<=	>	>=	!=
0	S2																		
1														ACC					
2										S5									
3		S4																	
4																			
5			S10																
6											S7								
7		R2																	
8			R4								R4								
9			R3								R3								
10				S11															
11			S12									S34							
12					R10	R10	R10	R10	R10				R10						
13					S15	S16	S17	S18	S14										
14			R5								R5								
15			S19																
16			S20																
17			S21																
18			S22																
19					R6	R6	R6	R6	R6				R6						
20					R7	R7	R7	R7	R7				R7						
21					R8	R8	R8	R8	R8				R8						
22					R9	R9	R9	R9	R9				R9						
23			S24																
24															S25	S30	S31	S32	S33
25			R13																
26			S27																
27													S28						
28									R12										
29									S37										
30			R14																
31			R15																
32			R16																
33			R17																
34												S34							
35					S15	S16	S17	S18					S36						
36					R11	R11	R11	R11	R11				R11						
37																			R1

表 2：GOTO 表

	S	A	F	M	E	B	O
0	1						
2		3					
4						29	
5			6	9			
6				8			
11					13		
24							26
34					35		

4. 三地址表示的描述

三地址代码（Three Adress Code，简称 TAC）是指每条代码包含一个运算和三个地址，两个地址用于存放运算对象，一个地址用于存放运算结果，一般形式为：

x = y op z

y 和 z 为名字、常量或编译时产生的临时变量，x 为名字或临时变量；op 为运算符。

四元式是目前最常用的中间代码形式：

（运算符，第一运算数，第二运算数，结果）

例：a=b*c+b*d

(1) (*, b, c, t1)　　(2) (*, b, d, t2)

(3) (+, t1, t2, t3)　　(4) (=, t3, , a)

也可以写成赋值语句形式：

(1) t1=b*c　　(2) t2=b*d

(3) t3=t1+t2　　(4) a=t3

实验结果的中间代码使用四元式表示。

5. 设计并完善源程序

源程序

```
#include<string>
#include<iostream>
#include <fstream>
#include<cctype>
using namespace std;

int const  ACC =  1<<31-1;
string filename;    //用来输入文件名用
string symbol[200];
string blankCharacter = " ";
int num = 0 ;
//int start;      //用在输出输入串时确定开始的位置

//-----------------------------------------------------
struct stateStack //定义状态栈
{
    int top;
    int m[100];
};
void InitStateStack(stateStack& s)//建立状态栈
{
    s.top = 0;
}
int PushState(stateStack& s, int i)//把元素压入栈
{
    if (s.top == 100) return -1;
    else
    {
        s.m[s.top] = i;
        s.top++;
    }
    return 0;
}
int GetStateTop(stateStack& s)   //取栈顶符号
{
    int i;
    if (s.top == 0) return -1;
    else
    {
```

```
        i = s.m[ s.top - 1];
    }
    return i;
}
void PopStateTop(stateStack& s)//删除栈顶元素
{
    if (s.top == 0) cout << "wrong!";
    else
        --s.top;
}
//-----------------------------------------------------------------
//定义一个用于四元式输出的栈
struct quaternionStack
{
    int top;
    string s[5];
};

//-----------------------------------------------------------------
//
struct symbolStack//定义符号栈
{
    int top;
    string st[100];
};
void InitSymbolStack(symbolStack& s)//建立符号栈
{
    s.top = 0;
}
int PushSymbol(symbolStack& s, string str)//把元素压入栈
{

    if (s.top == 100) return -1;
    else
    {
        s.st[s.top] = str;
        s.top++;
    }
    return 0;
}
string GetSymbolTop(symbolStack& s)   //取栈顶符号
{
    string str;
    if (s.top == 0) cout << "wrong!";
```

```
    else
    {
        str = s.st[ s.top - 1];
    }
    return str;
}
void PopSymbolTop(symbolStack& s)//删除栈顶元素
{
    if (s.top == 0) cout << "wrong!";
    else
        --s.top;
}

//------------------------------------------------------------

//词法分析部分
const string keyWord[] =//定义关键字表
    {
        "int","double","float","void","long","for","if","else","while","include",
        "return","break","continue","do","true","false","case","switch"
    };
void showLex(string& s)
{
    symbol[num] = s;
    ++num;
    ofstream fout("词法分析.txt", ios::out | ios::app);
    if (isdigit(s[0]))
    {
        fout << "< " << s << ",常数 >" << endl;
        s = "";
    }
    else
        for (int i = 0; i < 18; ++i)
        {
            if (s == keyWord[i])
            {
                fout << "< " << s << ", 关键字 | 指向 " << s << " 的关键字表项的指
                        针 >" << endl;
                break;
            }
            if (i == 17)
            {
                if (ispunct(s[0]) && s[0] != '_')
                {
```

```
                fout << "< " << s << ",运算符 | 指向 " << s << " 的运算符表项的
                    指针 >" << endl;
            }
            else
            {
                fout << "< id,标识符 | 指向 " << s << " 的符号表项的指针 >" <<
                    endl;
            }
        }
    }
    s = "";
}
bool isDelimiters(char ch)     //判断是否为界限符
{
    bool tag = 0;
    string s = "{[()]},;\'\"";
    for (int i = 0; i < s.size(); i++)
    {
        if (ch == s[i])
        {
            tag = 1;
            break;
        }
    }
    return tag;
}

bool isBlank(char ch)   //判断是否为空
{
    bool tag = 0;
    if (ch == ' ')
        tag = 1;
    return tag;
}

bool isDecimalPoint(char ch)    //判断是否为小数点
{
    bool tag = 0;
    if (ch== '.' )
        tag = 1;
    return tag;
}
void lexAnalysis(string s)
{
```

```
    string yunsf, biaosf;
    for (int i = 0; i < s.size(); i++)
    {
        if (isBlank(s[i]))       //判断是否为空
        {
            if (biaosf != "")
                showLex(biaosf);
        }
        else if (ispunct(s[i]))
        {//C 库函数 int isgraph(int c) 检查所传的字符是否有图形表示法。带有图形表
            示法的字符是除了空白字符（比如 ' '）以外的所有可打印的字符。
            if (isDelimiters(s[i]))
            {
                if (yunsf != "")
                    showLex(yunsf);
                if (biaosf != "")
                    showLex(biaosf);
                ofstream fout("词法分析.txt", ios::out | ios::app);
                fout << "< " << s[i] << ",界限符 | 指向 " << s[i] << " 的界限符的指
针 >" << endl;
                symbol[num] = s[i];
                num++;
            }
            else if (isDecimalPoint(s[i]) || s[i] == '_')
            {
                biaosf += s[i];
            }
            else
            {
                if (biaosf != "")
                    showLex(biaosf);
                yunsf += s[i];
            }
        }//end ispunct( )
        else
        {
            if (yunsf != "")
                showLex(yunsf);
            biaosf += s[i];
        }
    }
    symbol[num] = "#";
}//词法分析结束
```

```
//-------------------------------------------------------------------
//语法分析部分
//推导式部分
string production[18] = {
                  "S'-->S",
                  "S-->doAwhileB",
                  "A-->{F}",
                  "F-->M" ,
                  "F-->FM",
                  "M-->i=E;",
                  "E-->E+i",
                  "E-->E-i",
                  "E-->E*i",
                  "E-->E/i",
                  "E-->i",
                  "E-->(E)",
                  "B-->(iOi)",
                  "O--><",
                  "O--><=",
                  "O-->>",
                  "O-->>=",
                  "O-->!=",
              };

//-------------------------------------------------------------------

//GOTO控制部分
void GOTO(int i, char b, stateStack& state_stack, symbolStack& symbol_stack)
{
    // cout<<"GOTOzhong fterminals wei:  "<<b<<endl;
    if (i == 0 && b == 'S')
    {
        PushState(state_stack, 1);
    }
    else if (i == 2 && b == 'A')
    {
        PushState(state_stack, 3);
    }
    else if (i == 4 && b == 'B')
    {
        PushState(state_stack, 29);
    }
    else if (i == 5 && b == 'F')
    {
```

```
        PushState(state_stack, 6);
    }
    else if (i == 5 && b == 'M')
    {
        PushState(state_stack, 9);
    }
    else if (i == 6 && b == 'M')
    {
        PushState(state_stack, 8);
    }
    else if (i == 11 && b == 'E')
    {
        PushState(state_stack, 13);
    }
    else if (i == 24 && b == 'O')
    {
        PushState(state_stack, 26);
    }
    else if (i == 34 && b == 'E')
    {
        PushState(state_stack, 35);
    }

    string B = "";
    B = B + b;
    PushSymbol(symbol_stack, B);
}

//-----------------------------------------------------------------------
//Action控制表
//n大于0表示移进
//n小于0则达标要进行规约，-n就代表用哪个推导式进行规约
//ACC代表规约成功 0表示出错
const int Action[38][19] = {
                {2},                                          //状态0       入栈do
                {0,0,0,0,0,0,0,0,0,0,0,0,0,0,ACC},            //状态1    # ACC
                {0,0,0,0,0,0,0,0,0,0,5},                      //状态2    入栈{
                {0,4},                                  //状态3    入栈while
                {0,0,0,0,0,0,0,0,0,0,0,0,23 },                //状态4      入栈 (
                {0,0,10},                                 //状态5        入栈i
                {0,0,10,0,0,0,0,0,0,0,7,0},               //状态6        入栈i   }
                {0,-2},    //状态7        规约      2.    A→{F}   //while
                {0,0,-4,0,0,0,0,0,0,0,-4},  //状态8    规约4. F→FM  //i    }
                {0,0,-3,0,0,0,0,0,0,0,-3},  //状态9    规约3. F→M    //i  }
```

```
                {0,0,0,11},                         //状态10    入栈=
                {0,0,12,0,0,0,0,0,0,0,0,34},//状态11        入栈 i  (
                {0,0,0,-10,-10,-10,-10,-10,-10,0,0,0,-10},
                    //状态12规约10.E→i= + - * / ;
                {0,0,0,0,15,16,17,18,14},     //状态13  入栈 + - * / ;
                {0,0,-5,0,0,0,0,0,0,0,-5},    //状态14 //规约 5.  M→i=E;//i  }
                {0,0,19},                                 //状态15     入栈i
                {0,0,20},                                   //状态16         入栈i
                {0,0,21},  //状态17          入栈i
                {0,0,22},  //状态18          入栈i
                {0,0,0,0,-6,-6,-6,-6,-6,0,0,0,-6},
                  //状态19 规约 6.E→E+i // + - * / ;
                {0,0,0,0,-7,-7,-7,-7,-7,0,0,0,-7},
                      //状态20规约 7.    E→E-i // + - * / ;
                {0,0,0,0,-8,-8,-8,-8,-8,0,0,0,-8},
                      //状态21规约 8.    E→E*i // + - * / ;
                {0,0,0,0,-9,-9,-9,-9,-9,0,0,0,-9},
                    //状态22  规约 9.E→E/i // + - * / ;
                {0,0,24},  //状态23       入栈i
                {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,25,30,31,32,33},入栈 < <= > >= !=
                {0,0,-13},                          //状态25        规约 13.   0→<
                {0,0,27},                           //状态26        入栈 i
                {0,0,0,0,0,0,0,0,0,0,0,0,28},                 //状态27       入栈 )
                {0,0,0,0,0,0,0,0,-12,0,0,0,0,0},//状态28 规约12. B→(i0i)   //;
                {0,0,0,0,0,0,0,0,37},                          //状态29     入栈 ;
                {0,0,-14},        //状态30     规约14.    0→<=       //i
                {0,0,-15},        //状态31     规约15.    0→>        //i
                {0,0,-16},        //状态32     规约16. 0→>=       //i
                {0,0,-17},        //状态33      规约17.    0→!=       //i
                {0,0,12,0,0,0,0,0,0,0,0,34}, //状态34      入栈 i (
                {0,0,0,0,15,16,17,18,0,0,0,0,36},//状态35      入栈 + - * / )
                {0,0,0,0,-11,-11,-11,-11,-11,0,0,0,-11},//状态36 规约
                                                        //11.E→(E)//+ - * / ;
                {0,0,0,0,0,0,0,0,0,0,0,0,0,-1}//状态37 规约 1.S→doAwhileB; //#
            };
//-------------------------------------------------------------
//终结符表
string terminals[19] =
        { "do","while","i","=","+","-","*","/",";","{","}","(",")","#","<","<=",">",">=","!=" };

//-------------------------------------------------------------------

//输出每步的分析过程 存储在文件“语法分析.txt”中
```

```
void showResult(stateStack& state_stack, symbolStack& symbol_stack, int start)
{
    //每次归约完毕后查看当前符号栈栈顶状态
    cout<<GetSymbolTop(symbol_stack)<<"   "<<GetStateTop(state_stack)<<endl;

    ofstream yffout("语法分析.txt", ios::out | ios::app);

    for (int i = 0; i < state_stack.top; i++)
    {
        if (state_stack.m[i] >= 10)
        {
            yffout << "(" << state_stack.m[i] << ")";
        }
        else
            yffout << state_stack.m[i];
    }
    yffout << "                                ";
    for (int j = 0; j < symbol_stack.top; j++)
        yffout << symbol_stack.st[j];
    yffout << "                                ";
    for (int k = start; k <= num + 1; k++)
        yffout << symbol[k];
    yffout << endl;
}

//----------------------------------
//表达式规约部分
void Reduce()
{
    ofstream syfout("四元式.txt", ios::out | ios::app);
    int start = 0, now = 0;
    quaternionStack quaternion_stack;
    quaternion_stack.top = 0;
    int order = 0;
    int m = 0, n, j;
    stateStack state_stack;    //建状态栈
    InitStateStack(state_stack);
    PushState(state_stack, m);
    symbolStack symbol_stack;//建符号栈
    InitSymbolStack(symbol_stack);
    PushSymbol(symbol_stack, "#");

    //对栈进行初始化
    for (int i = 0; i <= num; i++)
```

```
{
    showResult(state_stack, symbol_stack,i); //i+1代表已归约现在可查看新的状态
    string temp = symbol[i];
    j = 0;
    while (terminals[ j ] != temp)  //查找终结符对应的下标
    {
        ++j;
        if  ( j >= 19)
        {
            j = 2; //如果是变量或者常量，令：j = 2，即对应终结符 i
            break;
        }
    }
    m = GetStateTop(state_stack);       //查看当前状态栈的栈顶
    n = Action[m][ j];
    int rightLength;
    while ( n <0 )
    {
        n=-n;
        if (n  == 14||n==16||n==17)
            请填写程序语句;
         else if(n==1)
            请填写程序语句;
        else
            rightLength = production[n ].size() - 4;
        //cout<<"表达式"<<(n-40)<<"的右部长度为"<<rightLength<<endl;

        //-------------------------------------------------------
        //四元式输出控制部分

        string ss0, ss1, ss2;
        string t[] = {"t0","t1","t2","t3","t4","t5","t6","t7","t8","t9","t10"};

        if (n == 10)     //当用 10.E→i 式归约时,开始建立栈
        {
            quaternion_stack.s[quaternion_stack.top] = symbol[i - 1];
            ++quaternion_stack.top;
        }
        else   if (n  >= 6 && n  <= 9)//6. E→E+i 7.E→E-i   8.E→E*I 9.E→E/i
        {
            --quaternion_stack.top;
            ss0 = quaternion_stack.s[quaternion_stack.top];
```

```
        --quaternion_stack.top;
        ss1 = quaternion_stack.s[quaternion_stack.top];
        请填写程序语句；

        请填写程序语句；


        syfout << now << "  (" << ss1 << "," << ss2 << "," << ss0 << "," << t[order]
        << ")" << endl;//运算的四元式
        ++now;
        quaternion_stack.s[quaternion_stack.top] = t[order];
        ++quaternion_stack.top;
        ++order;
    }
    else if (n == 5)          //赋值语句的四元式；5.    M→i=E;
    {
        int l = symbol_stack.top - 4;
        quaternion_stack.top = 0;
        syfout << now << "  (=:," << quaternion_stack.s[quaternion_stack.top]
        << ",-," << symbol_stack.st[l] << ")" << endl;
        ++now;
    }
    else if (n  == 13 || n == 14 || n  == 15 || n == 16)//判断语句的四元式

        syfout << now << "  if  " << symbol[i - 2] << symbol[i - 1] << symbol[i]
        << "  goto " << start << endl;



    //-----------------------------------------------------
    for (int k = 1; k <= rightLength; k++)
    {
        PopSymbolTop(symbol_stack);
        PopStateTop(state_stack);
    }
    //  showLex(state_stack,symbol_stack,i);
    GOTO(GetStateTop(state_stack), production[n][0], state_stack,
    symbol_stack);//此处用到了状态栈和符号栈的引用
    showResult(state_stack, symbol_stack, i);
    m = GetStateTop(state_stack);   //while xunhuanblankCharacterzhibufen
    n = Action[m][j];
}
if (n == 0)
```

```
        {
            cout << "归约出错";
            break;
        }
        if (n == ACC)
            cout << "规约成功" << endl;
        else
        {
            PushState(state_stack, n);
            // cout<<n<<endl;
            PushSymbol(symbol_stack, temp);
            //每次移进后都重新输出状态栈，符号栈和符号串
            if (quaternion_stack.top > 0)
            {
                quaternion_stack.s[quaternion_stack.top] = temp;
                //      cout<<quaternion_stack.s[quaternion_stack.top];
                ++quaternion_stack.top;
            }
        }
    }
}
//-------------------------------------------------------------------
//主函数
int main()
{
    //cout << "输入需要进行分析的程序的名称:\n";
    //cin >> filename;
    filename = "test.cpp";
    ifstream fin(filename.c_str());     //待添加当文件不存在 的处理
    string line, s;
    fin >> line;   //直接读取到空格，并且省略回车,最后至少要有空格或省略号
    while (!fin.eof())
    {
        s = s + line + blankCharacter;
        fin >> line;
    }
    fin.close();
    ofstream fout("词法分析.txt");
    fout << "";
    cout << s <<"--------" << endl;   //在字符界面查看需要分析的字符串，可以删除
    lexAnalysis( s );              //词法分析
        for(int i=0;i<=num;i++)
        cout<<i<<":   "<<symbol[i]<<endl;
    ofstream yffout("语法分析.txt");
```

```
    yffout << "状态栈" << "\t\t符号栈" << "\t\t\t\t输入串" << endl;
    ofstream syfout("四元式.txt");
    syfout << "";
    Reduce();                    //语法分析，进行规约
    return 0;
}
```

6. 程序 test.cpp 测试用例如下：

```
do
{
    a = a * b + c - d;
    b = (a + b);
    n = m + f;
    k = k / 5;
} while (i < i) ;
```

7. 实验报告模板

编译原理 DO-WHILE 循环语句的中间代码生成
实验报告

学院：

班级：

姓名：

学号：

实验三：中间代码生成

一． 实验题目

DO-WHILE 循环语句的中间代码生成

二． 实验目的

三． 实验内容

1. 根据给出的样例程序, 补充填写部分并运行;

程序测试用例

```
do
{
   a = a * b + c - d;
   b = (a + b);
   n = m + f;
   k = k / 5;
} while (i < i) ;
```

2.根据所设计的分析程序和所给定的测试用例,自行构造识别活前缀的 DFA、ACTION 表和 GOTO 表，上机测试并通过所设计的分析程序。

（1）自行构造的识别活前缀的 DFA 图；
(2)自行设计的 ACTION 表和 ACTION 表所对应的 C 语言源代码；
(3)自行设计的 GOTO 表和 GOTO 表所对应的 C 语言源代码；
（4）修改并测试程序，测试案例：

```
do
{
      a = a * b + c - d;
      b = (a + b);
      n = m + f;
      k = k / 5;
} while (i < i) ;
```

四． 实验结果

1. 完善后的程序词法分析、语法分析和四元式的运行结果图。

(1)程序清单，填写部分用红色标注

（2）程序测试用例

```
do
{
     a = a * b + c - d;
     b = (a + b);
     n = m + f;
     k = k / 5;
 } while (i < i) ;
```

(3)词法分析生成文件内容截图

（4）语法分析生成文件内容截图

（5）四元式生成文件内容截图

2. 自行构造识别活前缀的 DFA、ACTION 表和 GOTO 表后，上机测试后的程序词法分析、语法分析和四元式的运行结果图：

（1） DFA 图

（2） ACTION 表

（3） GOTO 表

（4） 代码清单

（5） 词法分析生成文件内容截图

（6） 语法分析生成文件内容截图

（7） 四元式生成文件内容截图

五． 实验心得