**中国矿业大学计算机学院**

**2018 级本科生课程设计报告**

课程名称 系统软件开发实践

实验名称 C编译器后端设计

报告时间 2021年4月7日

学生姓名 丁旭行

学 号 14184501

专 业 计算机科学与技术

任课教师 徐东红

**目 录**

[**1 实验目的 1**](#_Toc68616483)

[**2 实验内容 1**](#_Toc68616484)

[**3 编译器原理 1**](#_Toc68616485)

[**4 安装DOSBox环境 3**](#_Toc68616486)

[**5 后端代码设计 4**](#_Toc68616487)

[**5.1 关键代码 4**](#_Toc68616488)

[**5.2 生成汇编代码 18**](#_Toc68616489)

[**5.3 生成汇编程序 19**](#_Toc68616490)

[**5.4 对可执行程序进行反汇编 21**](#_Toc68616491)

[**6 基于之前实验的编译器后端设计 24**](#_Toc68616492)

[**6.1 代码修改 24**](#_Toc68616493)

[**6.2 生成汇编代码 34**](#_Toc68616494)

[**6.3 在DOSBox中进行验证 37**](#_Toc68616495)

[**7 实验体会 39**](#_Toc68616496)

# 1 实验目的

1. 在学习和熟悉了编译器前端具体过程并进行相关的实验练习后，继续深入学习编译器后端的设计和实现；
2. 通过对后端的了解并且结合前面所学知识能够对编译器有着具体的轮廓和系统的认识；
3. 在将来的具体项目中能够依据自己的见解和认识做出来具体成果。

# 2 实验内容

将前四周得到的编译器前端结果(抽象语法树)转换为四元式，最后生成X86汇编(例如:386)。要求：

1. 将得到的汇编代码在DOSBox环境下生成汇编程序；
2. 将生成的汇编程序在masm汇编生成可执行程序；
3. 运用masm相关工具对可执行程序进行反汇编，查看寄存器内容，单步调式，检查汇编程序的正确性。

# 3 编译器原理

1. 编译器工作流程

编程语言对人或机器来说是用来描述计算任务的符号和规则。能够在机器上运行的所有软件都是用某种编程语言编写的程序。但是在程序运行前，必须要将其转化为机器能够执行的格式，编译器的功能就是“转化”。

编译器的第一阶段就是词法分析，将输入的字符串流进行分析，并按一定的规则将输入的字符串流进行分割，从而形成所使用的源程序语言所允许的记号(token)，在这些记号序列将送到随后的语法分析过程中，与此同时将不符合规范的记号识别出来，并产生错误提示信息。通常采用确定的有限状态自动机(Deterministic Finite Automaton，DFA)来构造词法分析工具。已有一些专门的、开源的词法分析程序自动生成器可供免费使用，例如Lex，其GNU版本为Flex。

第二阶段是语法分析，分析词法分析产生的记号序列，并按一定的语法规则识别并生成中间表示形式，以及符号表。符号表记录程序中所使用的标识符及其标识符的属性；同样，将不符合语法规则的记号识别出其位置并产生错误提示语句。目前大多数开源编译器使用的语法分析方法有两类，一种是自顶向下(Top-Down Parsing)的分析方法，另一种是自底向上(Bottom-Up Parsing)的分析方法。目前已有一些专门的、开源的语法分析程序自动生成器可供免费使用，例如yacc，其GNU版本为bison。在编译器开发中通常联合使用词法分析工具LEX和语法分析工具YACC，这样可以大大减少前端设计工作量。

第三阶段是语义分析与中间代码生成，语义分析也即静态语法检查，静态语义检查的作用是分析语法分析过程中产生的中间表示形式和符号表，以检查源程序的语义是否与源语言的静态语义属性相符合。由于现有的大部分高级编程语言的语言是语法制导翻译的语言，因此目前通常使用的静态语义检查的方法是语法制导翻译的方法。语法制导翻译的方法是通过语法制导的属性文法来进行程序的语义分析。静态语义检查的根本目的是确认程序是否满足源编程语言要求的静态语义属性，可以理解检查为标识符的声明和使用是否相一致。中间代码生成是产生中间代码的过程。所谓“中间代码”是一种结构简单、含义明确的记号系统，这种记号系统复杂性介于源程序语言和机器语言之间，容易将它翻译成目标代码。另外，还可以在中间代码一级进行与机器无关的优化。中间代码可以有若干种形式，它们的共同特征是与具体机器无关。最常用的一种中间代码是三地址码，它的一种实现方式是四元式。三地址码的优点是便于阅读、便于优化。

第四阶段是代码优化，指对程序代码进行等价(指不改变程序的运行结果)变换。程序代码可以是中间代码(如四元式代码)，也可以是目标代码。优化的含义是最终生成的目标代码短(运行时间更短、占用空间更小)，时空效率优化。原则上，优化可以在编译的各个阶段进行，但最主要的一类是对中间代码进行优化，这类优化不依赖于具体的计算机。

第五阶段是目标代码生成，目标代码生成器把语法分析后或优化后的中间代码变换成目标代码。目标代码(object code)指计算机科学中编译器或汇编器处理源代码后所生成的代码，它一般由机器代码或接近于机器语言的代码组成。在生成目标代码时要考虑以下几个问题：计算机的系统结构、指令系统、寄存器的分配以及内存的组织等。编译器生成的目标程序代码可以有多种形式：汇编语言、可重定位二进制代码、内存形式。

1. 编译器分组介绍

编译器一般分为前端和后端，主要目的是在多种源语言和多种目标语言的开发过程中，可以灵活搭配组合，消除重复开发的工作量，提高编译系统的开发效率。前端依赖于语言并很大程度上独立于目标机器。一般包括语法分析、词法分析、符号表的建立、语义分析、中间代码生成以及相关错误处理。后端依赖于目标机器的阶段或某些阶段的某些部分。一般来说，后端完成的任务不依赖于源语言而只依赖于中间语言。主要包括代码优化、代码生成以及相关的错误处理和符号表操作。

符号表管理：记录源程序中符号的必要信息，并加以合理组织，从而在编译器的各个阶段能对它们进行快速、准确的查找和操作。符号表中的某些内容甚至要保留到程序的运行阶段。

错误处理：用户编写的源程序中往往会有一些错误，可分为静态错误和动态错误两类。所谓静态错误又可分为语法错误和静态语义错误。语法错误是指有关语言结构上的错误，如单词拼写错、表达式中缺少操作数、begin和end不匹配等。静态语义错误是指分析源程序时可以发现的语言意义上的错误，如加法的两个操作数中一个是整型变量名，而另一个是数组名等。动态错误，是指源程序中的逻辑错误，它们发生在程序运行的时候，也被称作动态语义错误，如变量取值为零时作为除数，数组元素引用时下标出界等。

# 4 安装DOSBox环境

访问网址<https://www.dosbox.com/download.php?main=1>，点击按钮进行 DOSBox的下载，然后按照安装指示完成DOSBox的安装。



图1 下载网址

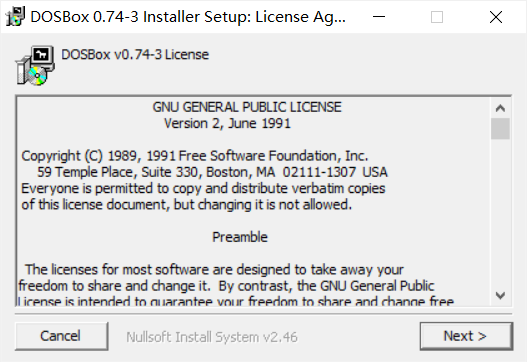


图2 安装DOSBox

安装DOSBox完成后，我需要创建一个用于保存汇编工具(MASM、Link、DEBUG等工具)和汇编文件(\*.asm)的目录：C:\asm，然后把MASM文件夹中的文件都拷贝到该目录中。



图3 保存汇编工具和文件的目录

# 5 后端代码设计

## 5.1 关键代码

// (1) 定义各种单词的key值

#define K\_DIGIT 3 //整数

#define K\_CHAR 4 //字符

#define K\_STRING 5 //字符串

#define K\_TYPE 6 //数据类型

#define K\_KEYWORDS 7 //关键字

#define K\_OPERATOR 8 //运算符

#define K\_IDENTIFIER 9 //标识符

#define K\_BRACKET 10 //括号

using namespace std;

// (2) 目标代码元素

typedef struct Target

{

string dsf; //结果

string op; //操作

string dst; //目的操作数

string dsc; //源操作数

string mark; //标志

string step; //跳转位置

}Target;

//保存声明变量

vector<Variable> var\_table;

//保存目标代码

vector<Target> target\_code;

char lab='A'; //记录跳转标志

char vab='A'; //记录中间变量

// (3) 生成的汇编文件名称

string asmfile(string source)

{

if(source.size()==0)

{

cout<<"源文件名不能为空"<<endl;

exit(-1);

}

string temp="";

int i,j;

j = source.size();

for(i = j-1;i>=0;i--)

{

// if(source[i] == '\\' || source[i]== '/')

// break;

if(source[i] == '.')

{

j = i;

break;

}

}

temp = source.substr(0,j) + ".asm";

return temp;

}

// (4) 保存到目标代码

void add\_target\_code(string dsf,string op,string dst,string dsc,string mark,string step)

{

Target tmp;

tmp.dsf = dsf;

tmp.op = op;

tmp.dst = dst;

tmp.dsc = dsc;

tmp.mark = mark;

tmp.step = step;

target\_code.push\_back(tmp);

}

// (5) 判断操作符、界符、空白以及单词类型

//是否为运算操作符

int is\_operator(char c)

{

if(c == '+' || c=='-'||c=='\*'||c=='/'||c==','||c=='=' ||c=='>' || c=='<')

return 1;

else

return 0;

}

//是否为大括号、小括号、分号

int is\_bracket(char c)

{

if(c=='{' || c=='}' || c=='(' || c==')' ||c==';')

return 1;

else

return 0;

}

//是否为空白

int is\_blank(char c)

{

if(c=='\n' || c=='\t' || c==' ' || c=='\r')

return 1;

else

return 0;

}

//判断单词类型

int word\_token(string s)

{

int size = s.size();

//字符数据

if(s[0]=='\'')

{

if(s[size-1] == '\'')

return K\_CHAR;

else

{

cout<<"错误的字符串数据："<<s<<endl;

exit(-1);

}

}

//字符串数据

else if(s[0]=='\"')

{

if(s[size-1]=='\"')

return K\_STRING;

else

{

cout<<"错误的字符串数据："<<s<<endl;

exit(-1);

}

}

//整数

else if(isdigit(s[0]))

{

for(int i=1;i<size;i++)

{

if(!isdigit(s[i]))

{

cout<<"不合法的标识符："<<s<<endl;

exit(-1);

}

}

return K\_DIGIT;

}

else

{

for(int i=0;i<size;i++)

{

if(!isalnum(s[i]) && s[i]!='\_')

{

cout<<"不合法的标识符："<<s<<endl;

exit(-1);

}

}

//数据类型

if(s=="int" || s=="char")

return K\_TYPE;

//关键字

else if(s=="if" || s=="else" || s=="printf" || s=="main")

return K\_KEYWORDS;

//自定义标识符

else

return K\_IDENTIFIER;

}

}

//添加分词结果

void add\_keywords(vector<IDwords> &v,int id,string word)

{

IDwords temp;

temp.id = id;

temp.word = word;

v.push\_back(temp);

}

// (6) 词法分析

void lexical\_analysis(string source,vector<IDwords> &AnalysisResults)

{

char ch;

ifstream rfile(source.c\_str());

if(!rfile.is\_open())

{

cout<<"无法打开源文件"<<endl;

exit(-1);

}

rfile>>noskipws; //不过滤空格

while(rfile>>ch)

{

int state=0; //判断状态

string temp(""); //字符串缓存

char try\_ch; //探测前面的字符

switch(state)

{

case 0:

if(ch=='/') //可能是注释

{

rfile>>try\_ch;

if(try\_ch=='/')

{

while(rfile>>try\_ch)

{

if(try\_ch=='\n')

break; //这是一行注释

}

break;

}

else if(try\_ch=='\*')

{

while(rfile>>try\_ch)

{

if(try\_ch=='\*')

{

rfile>>try\_ch;

if(try\_ch=='/')

break; //这是多行注释

}

}

break;

}

else

{

add\_keywords(AnalysisResults,K\_OPERATOR,char\_to\_str(ch));

ch = try\_ch; //继续状态1

}

}

case 1:

if(is\_operator(ch)) //判断操作符

{

add\_keywords(AnalysisResults,K\_OPERATOR,char\_to\_str(ch));

break;

}

case 2:

if(is\_bracket(ch)) //大括号、小括号

{

add\_keywords(AnalysisResults,K\_BRACKET,char\_to\_str(ch));

break;

}

case 3:

if(is\_blank(ch)) //空白符

break;

case 4:

if(ch=='#') //跳过预处理

{

while(rfile>>ch)

{

if(is\_blank(ch))

{

break;

}

}

break;

}

default://判断单词类型

temp = temp + char\_to\_str(ch);

while(rfile>>try\_ch)

{

if(try\_ch == '\"')

{

temp = temp + char\_to\_str(try\_ch);

if(ch == '\"')

{

add\_keywords(AnalysisResults,word\_token(temp),temp);

break;

}

else

{

cout<<"不合法的标识符："+temp<<endl;

exit(-1);

}

}

else if(is\_blank(try\_ch) )

{

if(ch != '\'' && ch != '\"')

{

add\_keywords(AnalysisResults,word\_token(temp),temp);

break;

}

else

temp = temp + char\_to\_str(try\_ch);

}

else if(is\_operator(try\_ch) )

{

if(ch !='\'' && ch != '\"' )

{

add\_keywords(AnalysisResults,word\_token(temp),temp);

add\_keywords(AnalysisResults,K\_OPERATOR,char\_to\_str(try\_ch));

break;

}

else

temp = temp + char\_to\_str(try\_ch);

}

else if(is\_bracket(try\_ch))

{

add\_keywords(AnalysisResults,word\_token(temp),temp);

add\_keywords(AnalysisResults,K\_BRACKET,char\_to\_str(try\_ch));

break;

}

else

temp = temp + char\_to\_str(try\_ch);

}

}

}

rfile.close();

}

// (7) 表达式分析

void expression(vector<IDwords>::iterator &it)

{

string dsf,op,dst,dsc;

//保存非操作符栈

stack<string> word\_stack;

//操作符栈

stack<string> oper\_stack;

oper\_stack.push("#");

while(it->word != ";") //遇到';'一条语句结束

{

if(it->word == "(")

oper\_stack.push(it->word);

else if(it->word == ")")

{

while(oper\_stack.top() != "(")

{

op = oper\_stack.top();

oper\_stack.pop();

// oper\_stack.push(it->word);

dsc = word\_stack.top();

word\_stack.pop();

dst = word\_stack.top();

word\_stack.pop();

vab = vab+1;

if(vab == 91)

vab = '0';

dsf = "tmp" + char\_to\_str(vab);

Variable tmp;

tmp.var = dsf;

var\_table.push\_back(tmp);

word\_stack.push(dsf);

add\_target\_code(dsf,op,dst,dsc," "," ");

}

oper\_stack.pop();

}

else if(it->id != K\_OPERATOR)

word\_stack.push(it->word);

else if(oper\_stack.top() == "(")

{

oper\_stack.push(it->word);

}

else if(level(it->word) < level(oper\_stack.top())) //优先级低

{

op = oper\_stack.top();

oper\_stack.pop();

oper\_stack.push(it->word);

dsc = word\_stack.top();

word\_stack.pop();

dst = word\_stack.top();

word\_stack.pop();

vab = vab+1;

if(vab == 91)

vab = '0';

dsf = "tmp" + char\_to\_str(vab);

Variable tmp;

tmp.var = dsf;

var\_table.push\_back(tmp);

word\_stack.push(dsf);

add\_target\_code(dsf,op,dst,dsc," "," ");

}

else //优先级高

oper\_stack.push(it->word);

it++;

}

//弹出剩下的

while(oper\_stack.top() != "#")

{

op = oper\_stack.top();

oper\_stack.pop();

dsc = word\_stack.top();

word\_stack.pop();

dst = word\_stack.top();

word\_stack.pop();

if(op=="=")//赋值运算

{

add\_target\_code(dst,op,dsc," "," "," ");

}

else

{

vab = vab+1;

if(vab == 91)

vab = '0';

dsf = "tmp" + char\_to\_str(vab);

Variable tmp;

tmp.var = dsf;

var\_table.push\_back(tmp);

word\_stack.push(dsf);

add\_target\_code(dsf,op,dst,dsc," "," ");

}

}

}

// (8) 语法分析

void syntax\_analysis(vector<IDwords> &AnalysisResults)

{

vector<IDwords>::iterator it=AnalysisResults.begin();

if(it->word != "main")

{

cout<<"缺少main"<<endl;

exit(-1);

}

it = it+3; //跳过“（）”

if(it->word != "{")

{

cout<<"main函数缺少'{'"<<endl;

exit(-1);

}

it++;

//获取变量声明

add\_var\_table(it);

//获取代码段的操作

while(it != AnalysisResults.end())

{

//遇到printf

if(it->word == "printf")

{

printf\_analysis(it);

}

// if 语句

else if(it->word == "if")

{

if\_analysis(it);

}

else if(it->word == "}")

break;

//表达式分析

else

{

expression(it); //表达式分析

}

it++;

}

}

// (9) 生成汇编文件

void create\_asm(string file)

{

//变量声明

ofstream wfile(file.c\_str());

if(!wfile.is\_open())

cout<<"无法创建汇编文件"<<endl;

vector<Variable>::iterator it\_var;

wfile<<"ASSUME CS:codesg,DS:datasg"<<endl;

//数据段

wfile<<"datasg segment"<<endl;

for(it\_var=var\_table.begin();it\_var!=var\_table.end();it\_var++)

{

wfile<<" "<<it\_var->var<<" DB ";

if(it\_var->value != "")

wfile<<it\_var->value<<endl;

else

wfile<<"\'?\'"<<endl;

}

wfile<<"datasg ends"<<endl;

//代码段

wfile<<"codesg segment"<<endl;

wfile<<" start:"<<endl;

wfile<<" mov AX,datasg"<<endl;

wfile<<" mov DS,AX"<<endl;

vector<Target>::iterator it;

Target tmp;

for(it = target\_code.begin();it != target\_code.end();it++)

{

//加减法转化

if(it->op == "+" || it->op=="-")

addsub\_asm(wfile,it->dsf,it->op,it->dst,it->dsc);

//乘法转换

else if(it->op == "\*")

mul\_asm(wfile,it->dsf,it->dst,it->dsc);

//除法转换

else if(it->op == "/")

div\_asm(wfile,it->dsf,it->dst,it->dsc);

//赋值运算

else if(it->op == "=")

sign\_asm(wfile,it->dsf,it->dst);

//输出操作

else if(it->op == "p")

print\_asm(wfile,it->dsf,it->mark);

//if语法分析

else if(it->op == "if")

{

if\_asm(wfile,it->dst,it->dsc,it->mark,it->step);

}

else if(it->op == "else")

{

cout<<"else 没有找到匹配的 if"<<endl;

exit(-1);

}

//跳转语句

else if(it->op == "jmp")

{

wfile<<" JMP "<<it->step<<endl;

}

//跳转语句段标识

else if(it->op == "pstep")

{

wfile<<" "<<it->step<<":"<<endl;

}

//其他

else

{

cout<<"编译器暂不支持该语法操作"<<endl;

exit(-1);

}

}

## 5.2 生成汇编代码

(1) 设计一个测试案例如下图所示：

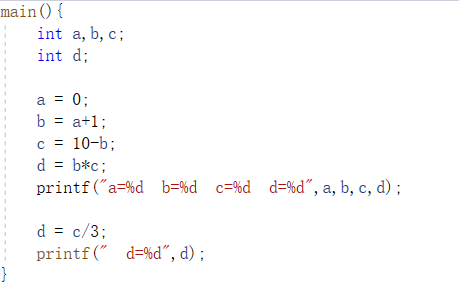


图4 测试代码

(2) 执行编译器程序，打开cmd命令窗口输入路径并运行compiler.exe test.c命令，得到相关变量声明与初始化信息以及四元式。

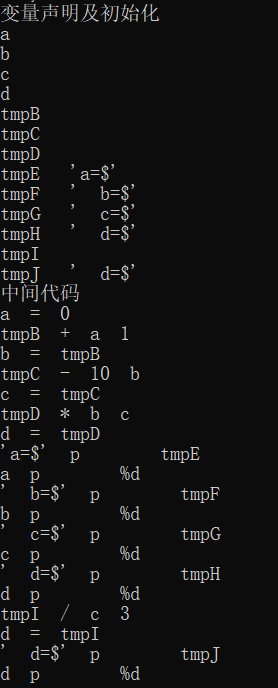
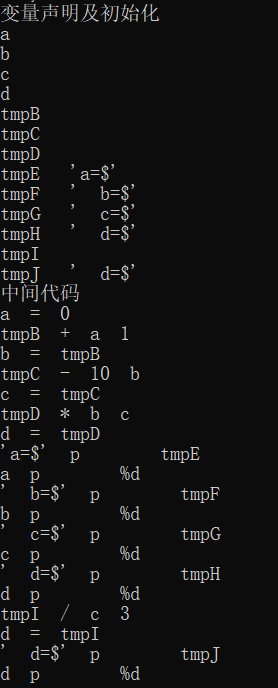


图5 显示信息

(3) 在相应文件夹下发现新生成的test.asm汇编代码文件。打开test.asm文件，具体汇编代码如下所示：

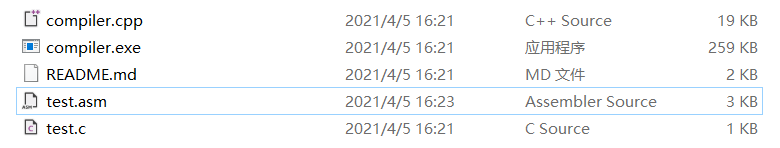


图6 生成文件

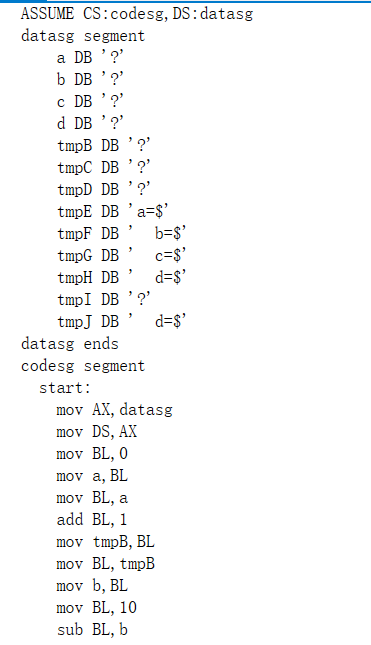


图7 asm文件部分代码

## 5.3 生成汇编程序

(1) 打开DOSBox程序，在小窗口内输入mount c C:\asm，将本机工作目录挂载到DOSBox虚拟C盘下，再输入c：进行工作空间转移。C:\asm为4中建立的目录文件夹用来保存汇编工具和汇编文件。

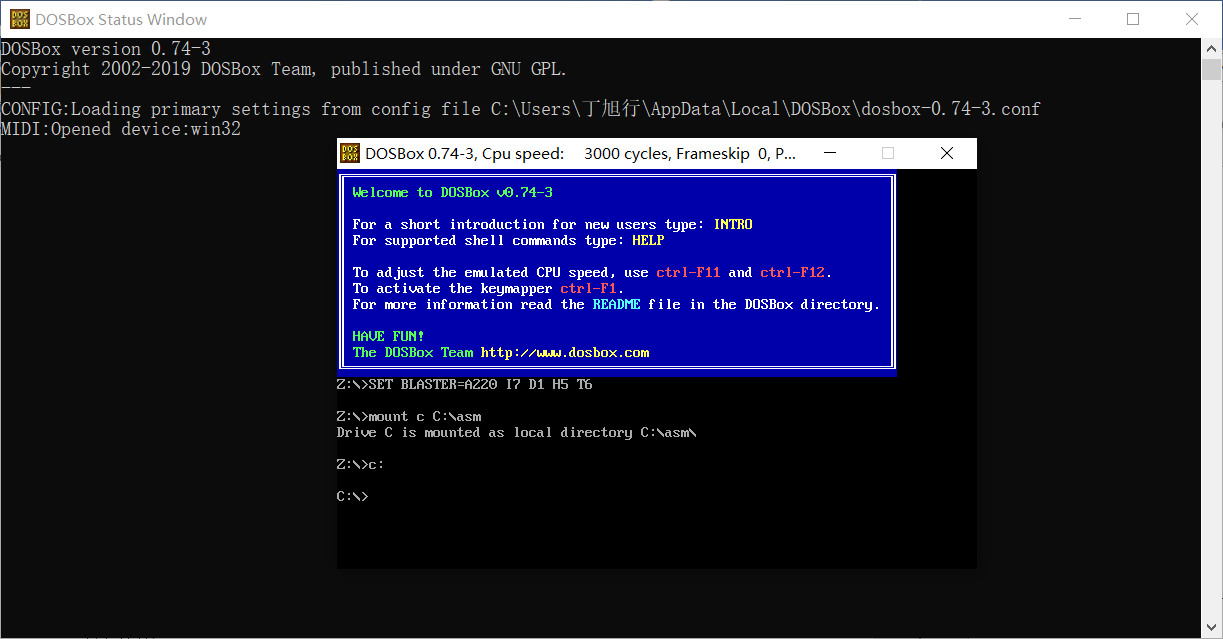


图8 挂载命令

(2) 将得到的汇编文件test.asm放入C:\asm文件夹下。



图9 转移文件

1. 输入指令masm test.asm，对源程序test.asm执行汇编，得到目标TEXT.OBJ文件。

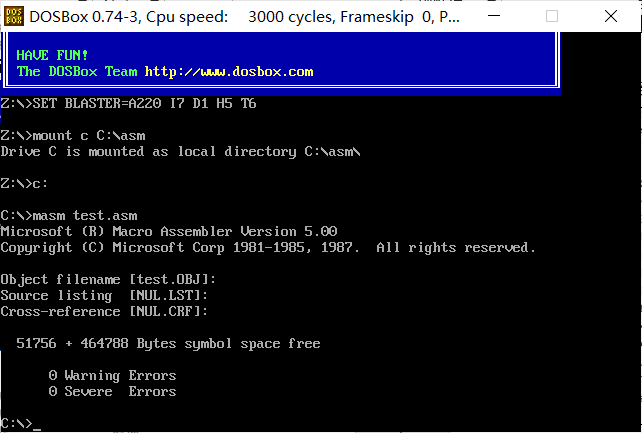


图10 执行汇编



图11 生成文件

1. 输入指令link test，链接目标程序，得到可执行程序TEST.EXE文件。

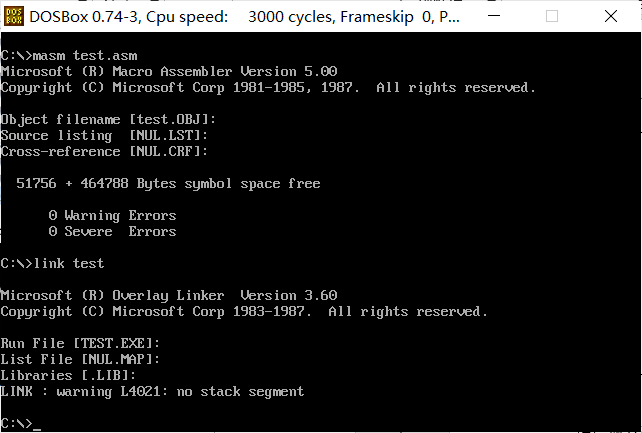


图12 生成执行程序

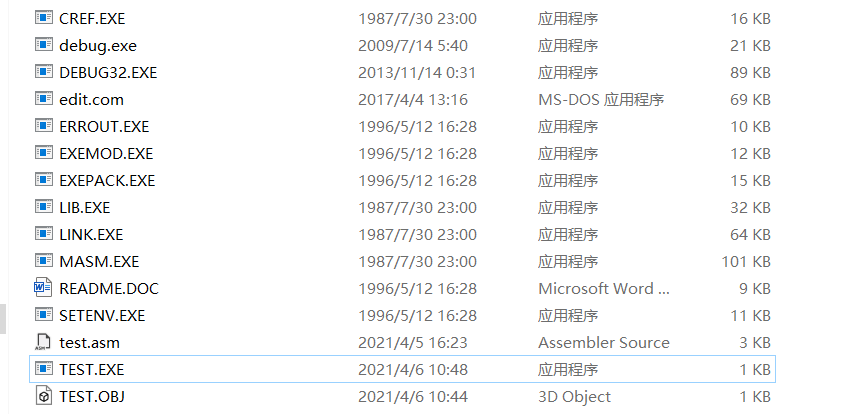


图13 生成文件

## 5.4 对可执行程序进行反汇编

输入指令debug test.exe，出现命令符’-‘后开始操作。

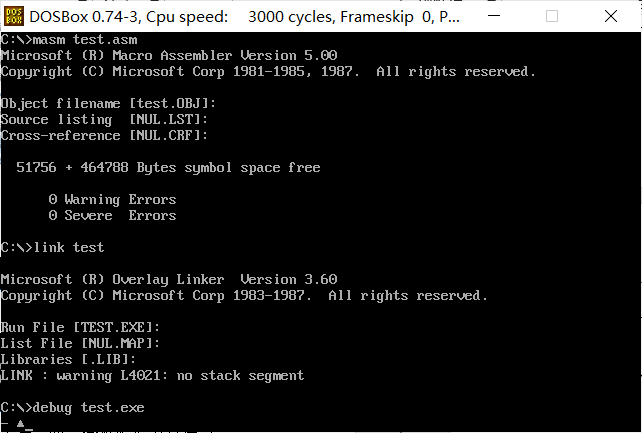


图14 输入命令

输入指令–r，查看程序运行前的寄存器组初始值：

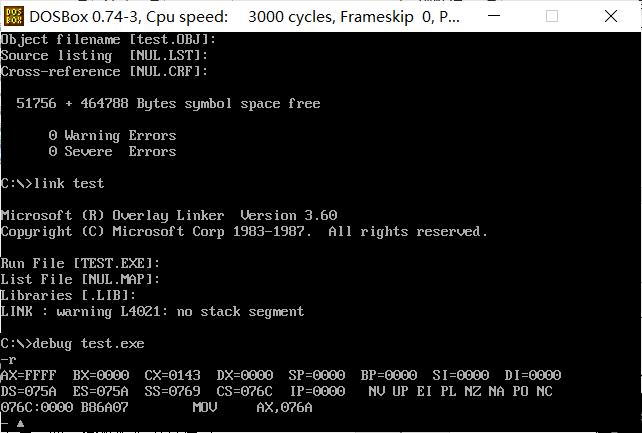


图15 寄存器组初始值

输入指令–u，查看程序反汇编代码。从反汇编代码中可看出，变量会被汇编为直接寻址方式，使用变量在数据段内的有效地址表示。

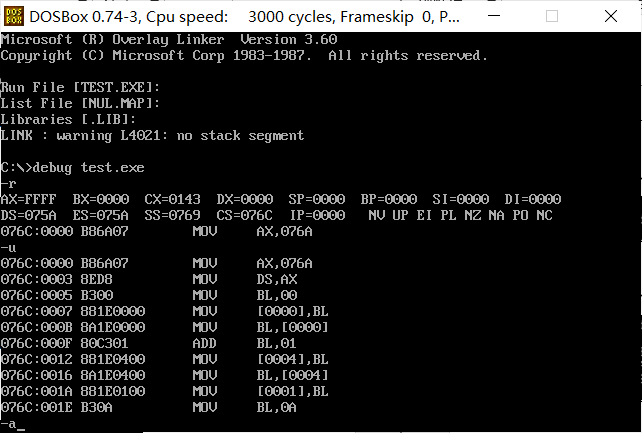


图16 变量直接寻址

输入指令–t,进行单步调试，数据段赋初值：

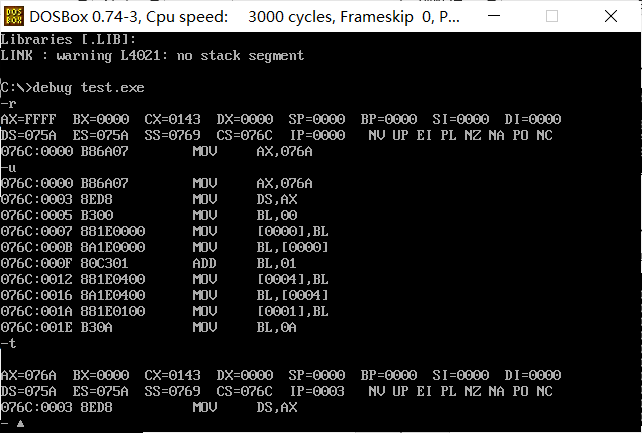
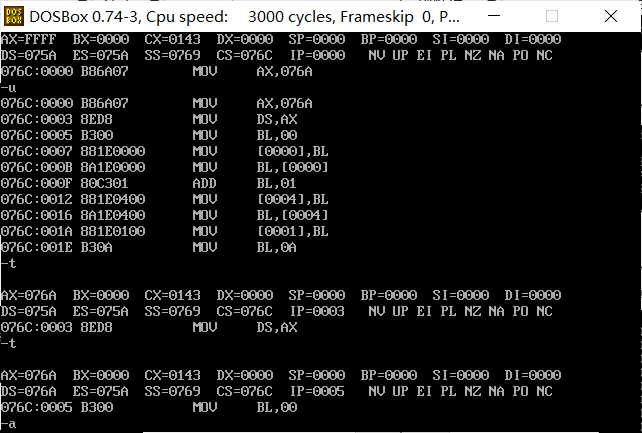


图17 数据段赋初值

BL=0H存入数据栈DS：0000：—a值



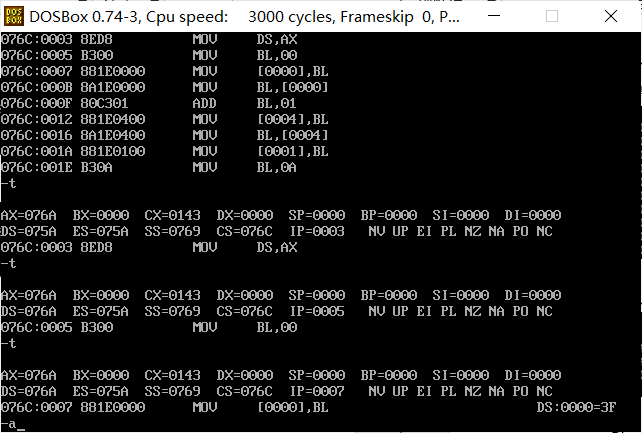


图18 a=0语句

取出a值，BL=0H：

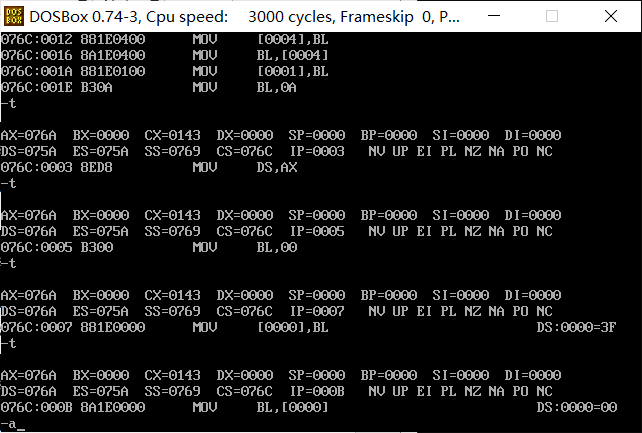


图19 取出a值

执行ADD运算，BL=BL+01H=01H，运算结果存入数据段DS：0004：

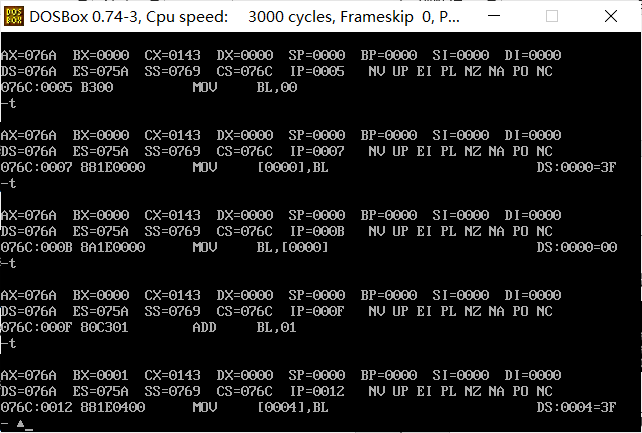


图20 加法运算

取出tmpB值，BL=01H，存入数据段DS:0001：—b值

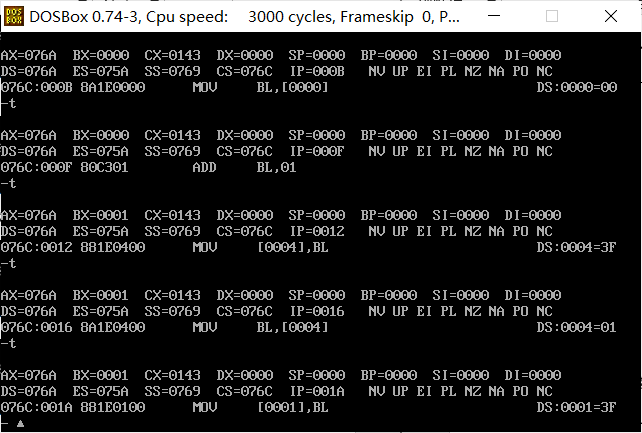


图21 b=a+1语句

执行SUB运算，BL=BL-01H=0AH-01H=09H，存入数据段DS:0005：

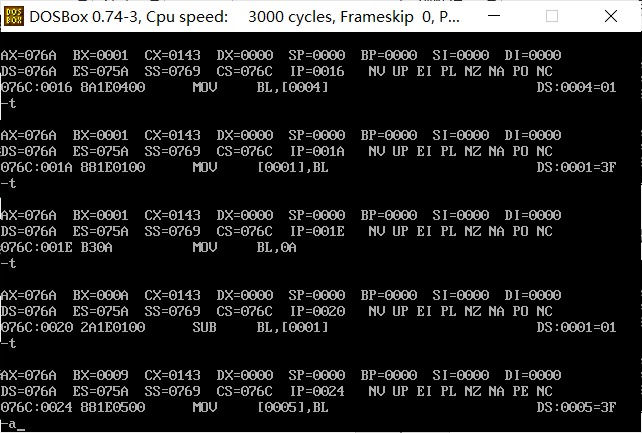


图22 减法运算

取出tmpC值，BL=09H，存入数据段DS:0002：—c值：

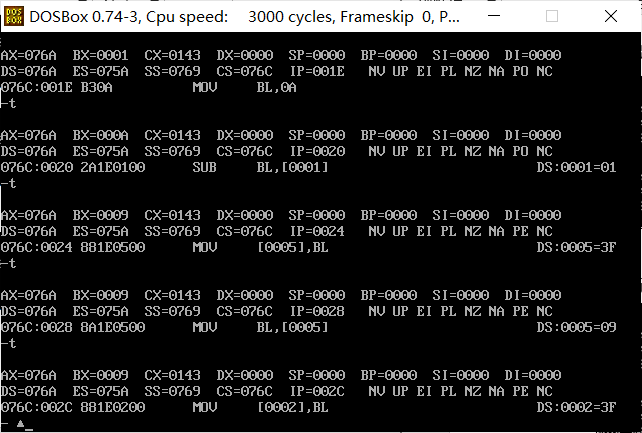


图23 c=10-b语句

语句d=b\*c和d=c/3执行过程与上述类似，继续调试，最后计算结果 a=0 b=1 c=9 d=9 d=3与预期结果一致。

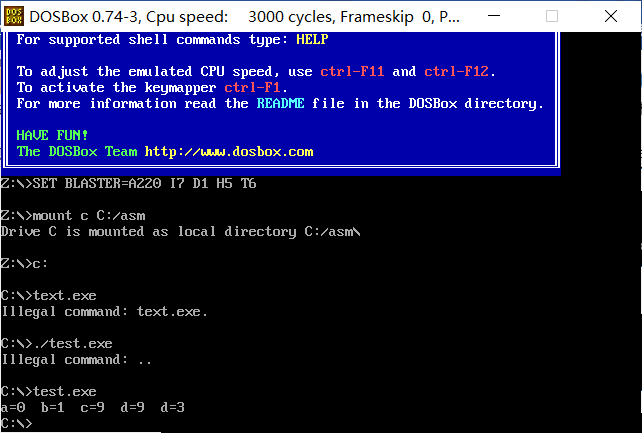


图24 输出结果

# 6 基于之前实验的编译器后端设计

## 6.1 代码修改

### 6.1.1 修改.l文件

(1) 在用户代码段重新编写主函数main()，待分析文件为in.txt，分析结果输出至文件out.txt，中间代码存于文件temp.txt，生成的汇编源程序为ans.asm。

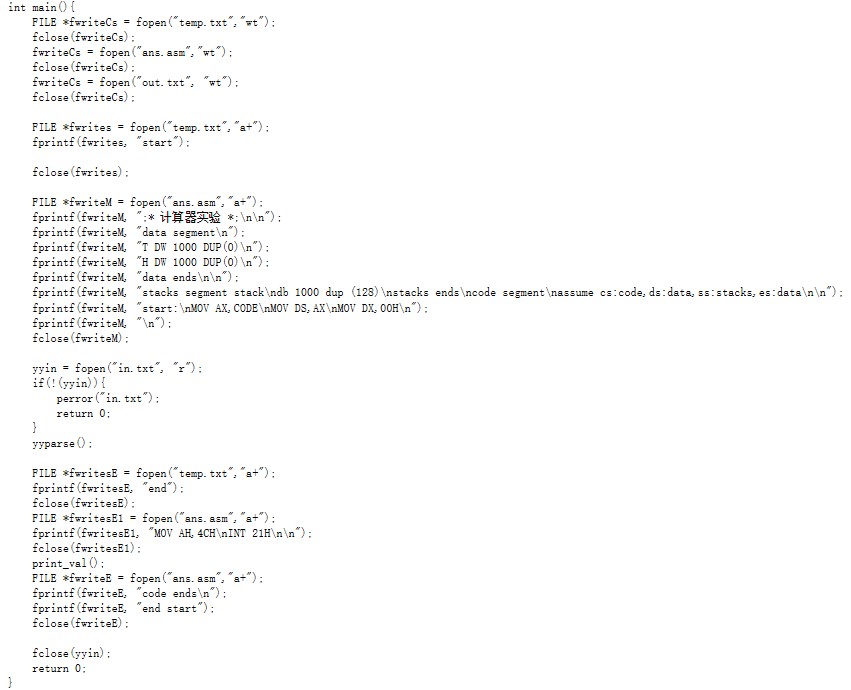


图25 改写main()函数

(2) 加入关键字int和main，使两者能被词法分析程序所识别并删除不需要的内置函数，只保留最基本的print()函数。

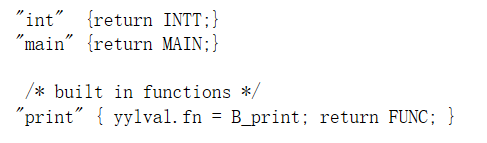


图26 加入关键字

### 6.1.2 修改.y文件

(1) 只保留简单的四则运算：

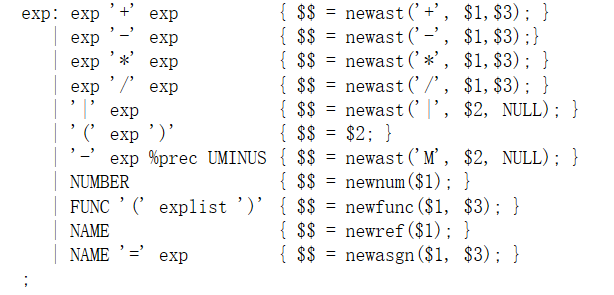


图27 四则运算

(2) 顶部规约规则改为calclist->calclist INTT MAIN ‘(‘ ‘)’ ‘{‘ stmt 。

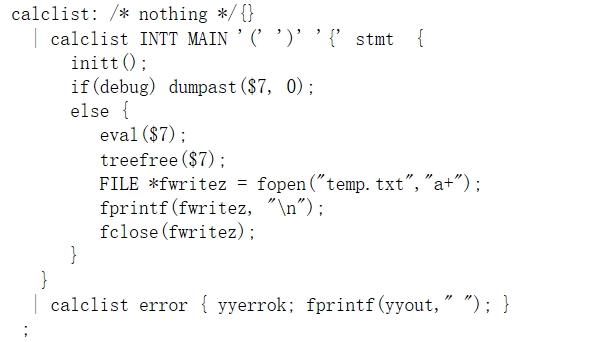


图28 修改顶部规约规则

### 6.1.3 修改.h文件

(1) 添加结构体adress，用于定义计算结果。

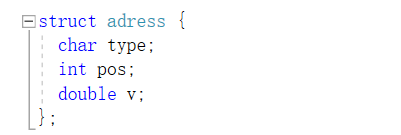


图29 结构体adress

(2) 添加结构体four，用于定义四元式。(操作符、操作数、类型、位置)

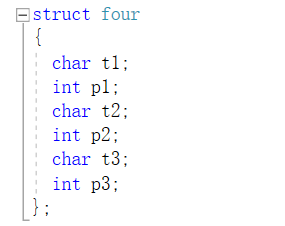


图30 结构体four

(3) 在符号表结构体symbol中新增pos属性。

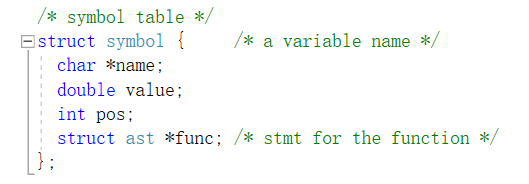


图31 新增pos属性

### 6.1.4 修改funcs.c文件

funcs.c文件的关键在于求值函数eval()，对于该函数，我们既要执行计算功能，返回struct address\*类型值，又要在计算的同时生成对应的四元式和汇编代码。

struct adress \*eval(struct ast \*a){

double v;

struct adress \*re = malloc(sizeof(struct adress));

struct adress \*l = malloc(sizeof(struct adress));

struct adress \*r = malloc(sizeof(struct adress));

struct four \*f = malloc(sizeof(struct four));

f->t1 = 'N';

f->t2 = 'N';

f->t3 = 'N';

if (!a){

yyerror("internal error, null eval");

re->v = 0.0;

re->type = 'N'; //-1为空

return re;

}

switch (a->nodetype){

/\* constant常量 \*/

case 'K':

v = ((struct numval \*)a)->number;

re->v = v;

re->type = 'T';

re->pos = ti++;

int pv = v;

FILE \*fwrite = fopen("temp.txt", "a+");

fprintf(fwrite, "MOV,%d,,%c%d\n", pv, re->type, re->pos);

fclose(fwrite);

f->t3 = 'T';

f->p1 = pv;

f->p3 = re->pos;

MOV(f);

break;

/\* name reference名字引用 \*/

case 'N':

v = ((struct symref \*)a)->s->value;

re->v = v;

re->type = 'H';

if (((struct symref \*)a)->s->pos == -1){

FILE \*fwrite1 = fopen("temp.txt", "a+");

pv = v;

((struct symref \*)a)->s->pos = re->pos = hi++;

fprintf(fwrite1, "MOV,%d,,%c%d\n", pv, re->type, re->pos);

fclose(fwrite1);

f->t3 = 'H';

f->p1 = pv;

f->p3 = re->pos;

MOV(f);

}

else

re->pos = ((struct symref \*)a)->s->pos;

break;

/\* assignment赋值 \*/

case '=':

re = eval(((struct symasgn \*)a)->v);

ti = 0;

v = re->v;

int Hpos, Tpos, Ttyp;

if (((struct symasgn \*)a)->s->pos == -1)

Hpos = ((struct symasgn \*)a)->s->pos = hi++;

else

Hpos = ((struct symasgn \*)a)->s->pos;

Tpos = re->pos;

Ttyp = re->type;

FILE \*fwrite2 = fopen("temp.txt", "a+");

fprintf(fwrite2, "MOV,%c%d,,H%d\n", Ttyp, Tpos, Hpos);

fclose(fwrite2);

f->t1 = Ttyp;

f->p1 = Tpos;

f->t3 = 'H';

f->p3 = Hpos;

MOV(f);

((struct symasgn \*)a)->s->value = re->v;

break;

/\* expressions 表达式\*/

case '+':

l = eval(a->l);

r = eval(a->r);

v = l->v + r->v;

int lt, rt, ls, rs;

lt = l->type;

rt = r->type;

ls = l->pos;

rs = r->pos;

re->pos = ti++;

re->type = 'T';

re->v = v;

FILE \*fwrite3 = fopen("temp.txt", "a+");

fprintf(fwrite3, "ADD,%c%d,%c%d,T%d\n", lt, ls, rt, rs, re->pos);

fclose(fwrite3);

f->t1 = lt;

f->p1 = ls;

f->t2 = rt;

f->p2 = rs;

f->t3 = 'T';

f->p3 = re->pos;

ADD(f);

break;

case '-':

l = eval(a->l);

r = eval(a->r);

v = l->v - r->v;

/\*int lt,rt,ls,rs;\*/

lt = l->type;

rt = r->type;

ls = l->pos;

rs = r->pos;

re->pos = ti++;

re->type = 'T';

re->v = v;

FILE \*fwrite4 = fopen("temp.txt", "a+");

fprintf(fwrite4, "SUB,%c%d,%c%d,T%d\n", lt, ls, rt, rs, re->pos);

fclose(fwrite4);

f->t1 = lt;

f->p1 = ls;

f->t2 = rt;

f->p2 = rs;

f->t3 = 'T';

f->p3 = re->pos;

SUB(f);

break;

case '\*':

l = eval(a->l);

r = eval(a->r);

v = l->v \* r->v;

/\*int lt,rt,ls,rs;\*/

lt = l->type;

rt = r->type;

ls = l->pos;

rs = r->pos;

re->pos = ti++;

re->type = 'T';

re->v = v;

FILE \*fwrite5 = fopen("temp.txt", "a+");

fprintf(fwrite5, "MUL,%c%d,%c%d,T%d\n", lt, ls, rt, rs, re->pos);

fclose(fwrite5);

f->t1 = lt;

f->p1 = ls;

f->t2 = rt;

f->p2 = rs;

f->t3 = 'T';

f->p3 = re->pos;

MUL(f);

break;

case '/':

l = eval(a->l);

r = eval(a->r);

v = l->v / r->v;

/\*int lt,rt,ls,rs;\*/

lt = l->type;

rt = r->type;

ls = l->pos;

rs = r->pos;

re->pos = ti++;

re->type = 'T';

re->v = v;

FILE \*fwrite6 = fopen("temp.txt", "a+");

fprintf(fwrite6, "DIV,%c%d,%c%d,T%d\n", lt, ls, rt, rs, re->pos);

fclose(fwrite6);

f->t1 = lt;

f->p1 = ls;

f->t2 = rt;

f->p2 = rs;

f->t3 = 'T';

f->p3 = re->pos;

DIV(f);

break;

case 'M':

l = eval(a->l);

v = -l->v;

/\*int lt,ls;\*/

lt = l->type;

ls = l->pos;

re->pos = ls;

re->type = lt;

re->v = v;

FILE \*fwrite7 = fopen("temp.txt", "a+");

fprintf(fwrite7, "NEG,,,%c%d\n", lt, ls);

fclose(fwrite7);

f->t3 = lt;

f->p3 = ls;

NEG(f);

break;

case 'L':

eval(a->l);

re = eval(a->r);

v = re->v;

break;

case 'F':

re = callbuiltin((struct fncall \*)a);

v = re->v;

break;

default:

printf("internal error: bad node %c\n", a->nodetype);

}

return re;

}

在eval()函数中调用的用于生成汇编代码的函数如下：

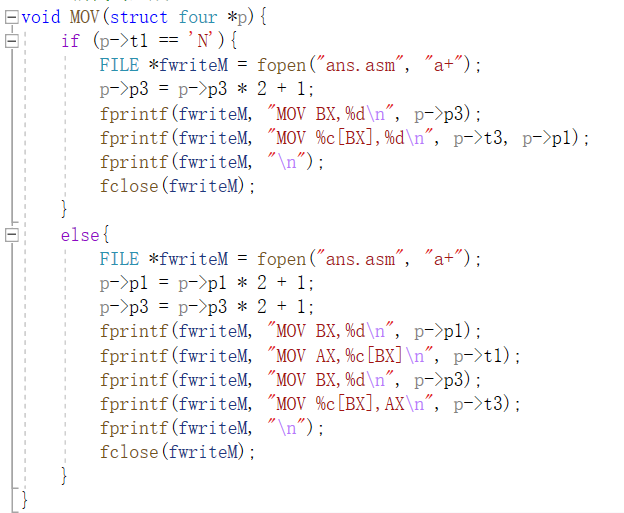


图32 MOV语句

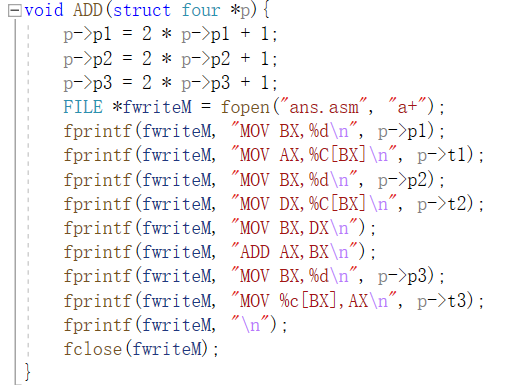


图33 ADD语句

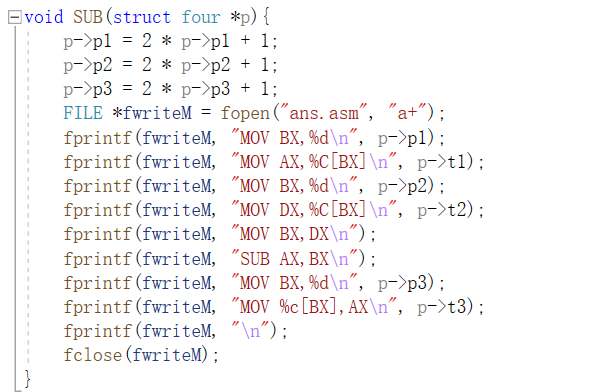


图34 SUB语句

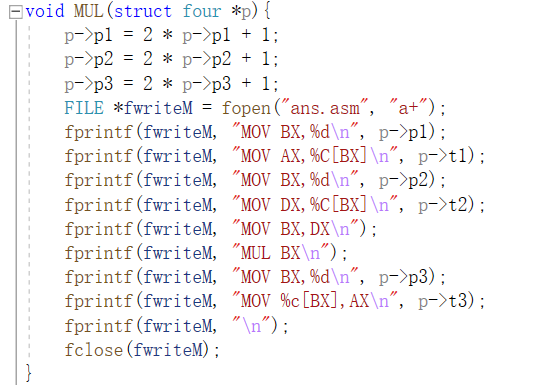


图35 MUL语句

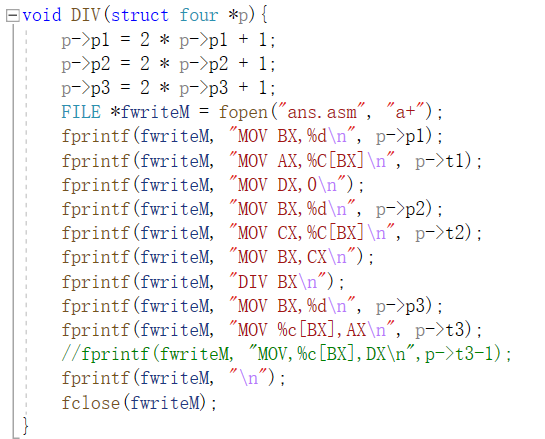


图36 DIV语句

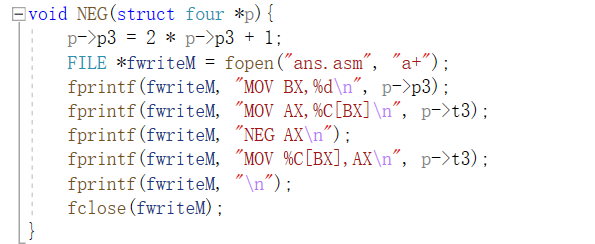


图37 NEG语句

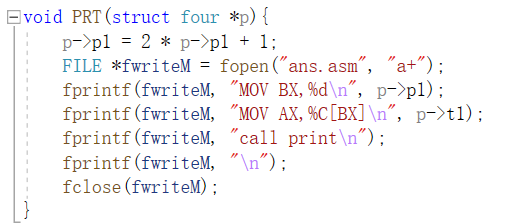


图38 PRT语句

## 6.2 生成汇编代码

(1) 打开Developer Command Prompt for VS 2019命令窗口，依次输入e:、cd E:\Flex\GnuWin32\bin进入安装目录下，再依次输入bison -d fb4-1.y、flex -ofb4-1.lex.c fb4-1.l，生成fb4-1.tab.h、fb4-1.tab.c和fb4-1.lex.c文件。

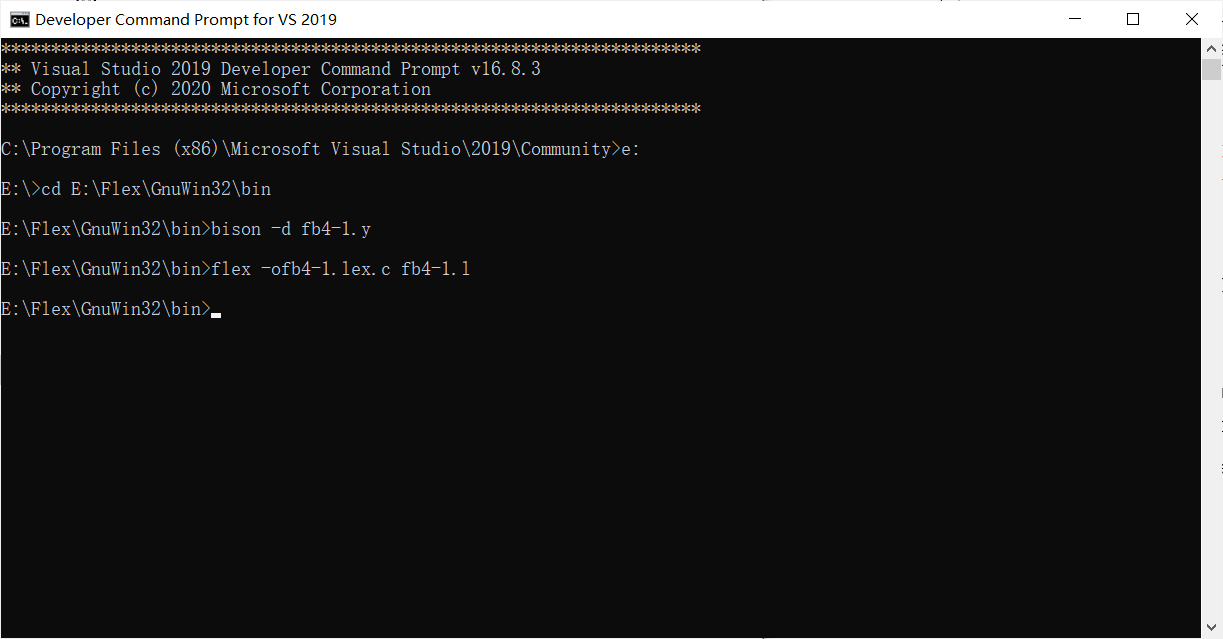


图39 输入命令

(2) 打开Cygwin命令窗口，输入cd E:/Flex/GnuWin32/bin进入所在目录，再输入命令gcc -o”fb4-1.tab.exe” fb4-1.tab.c fb4-1.lex.c fb4-1funcs.c -lm生成其对应的fb4-1.tab.exe文件。

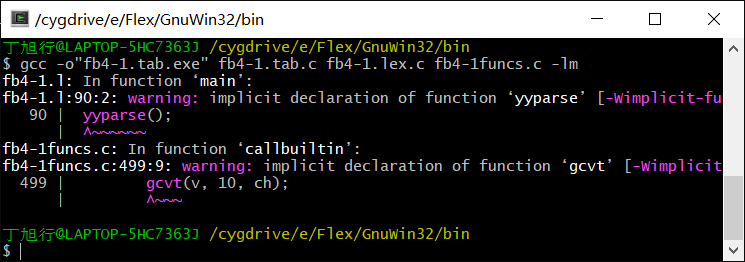


图40 输入命令

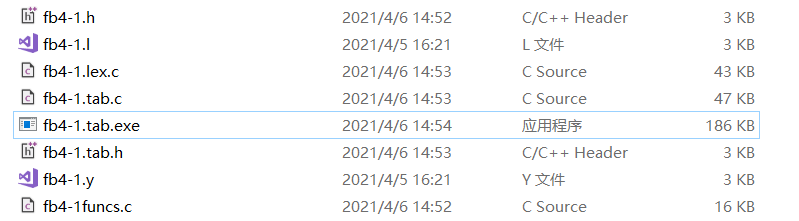


图41 生成文件

(3) 然后新建in.txt，内容如下，实验结果应该输出6和3。



图42 测试用例

(4) 在命令窗口运行生成的exe文件，显示输出结果并在当前目录下生成out.txt、temp.txt和ans.asm文件。

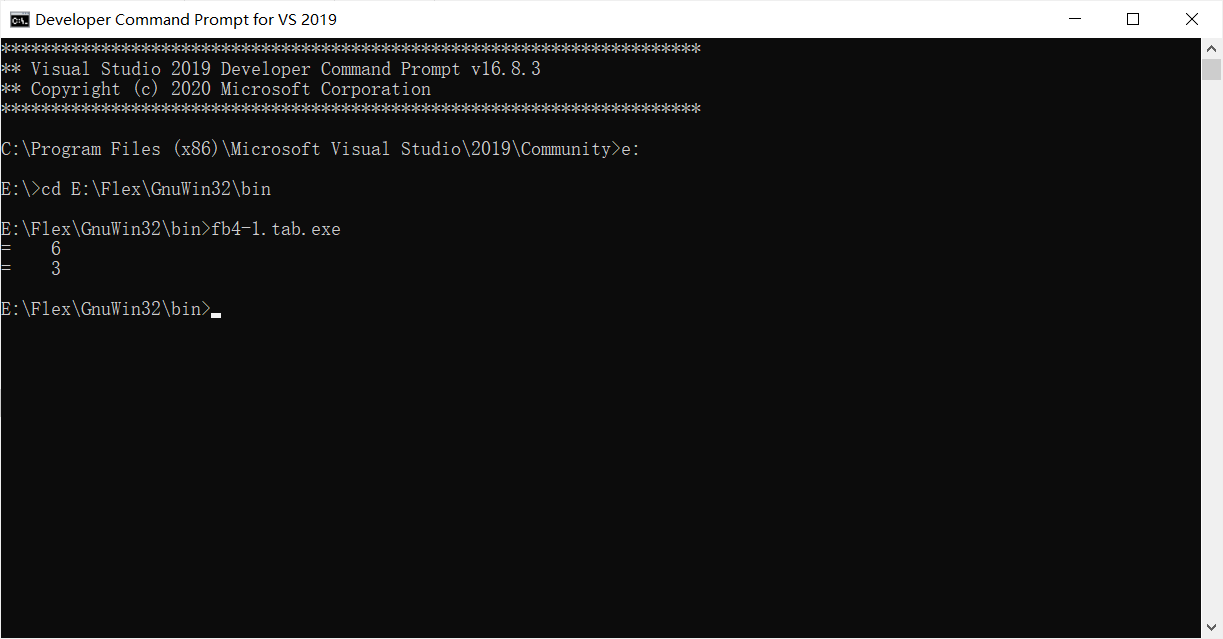


图43 显示结果

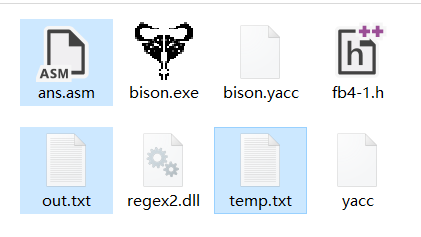


图44 生成文件



图45 out文件内容



图46 中间代码内容

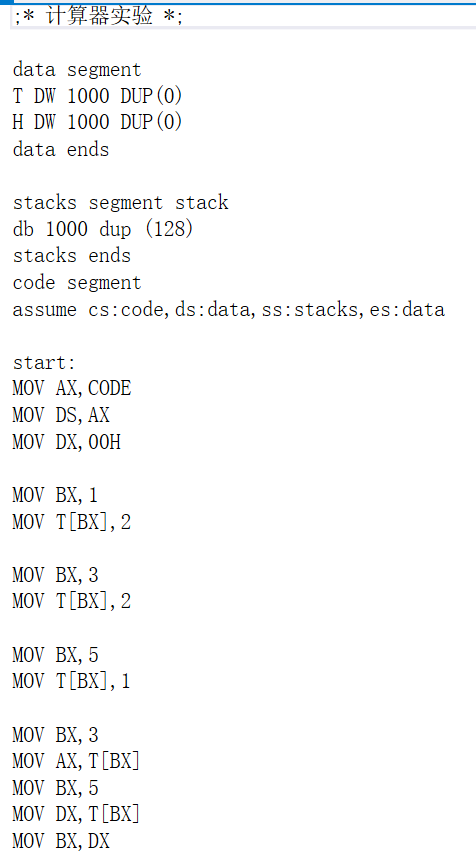


图47 汇编代码部分内容

## 6.3 在DOSBox中进行验证

(1) 将得到的汇编文件ans.asm放入C:\asm文件夹下。



图48 移动文件

(2) 挂载到DOSBox虚拟C盘下后，输入指令masm ans.asm，对源程序ans.asm执行汇编，得到目标程序ANS.OBJ。

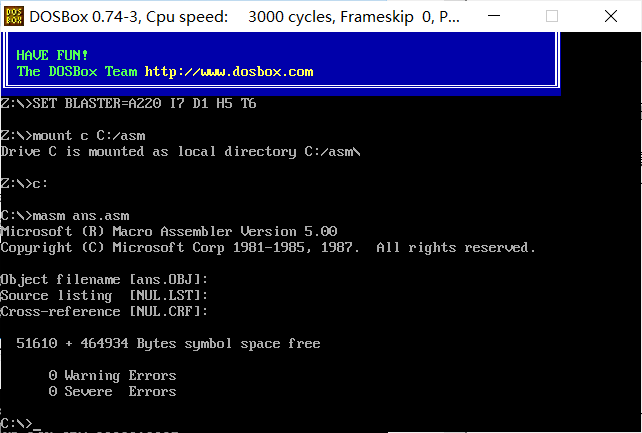


图49 输入命令



图50 生成文件

(3) 输入指令link ans，链接目标程序，得到可执行程序ANS.ASM。

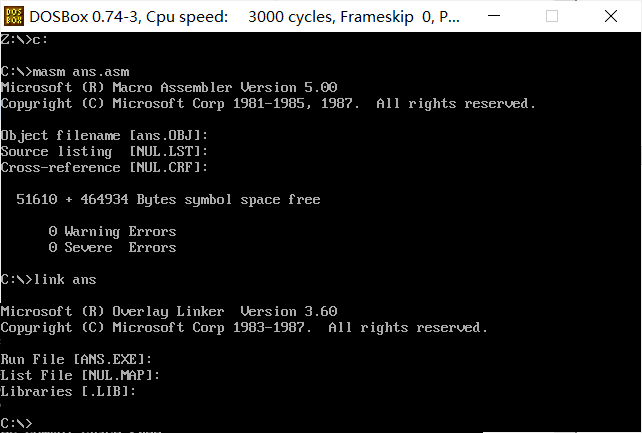


图51 输入命令



图52 生成文件

(4) 执行反汇编，输入指令debug ans.exe，出现命令符’-‘后开始操作。

输入指令–r，查看程序运行前的寄存器组初始值：

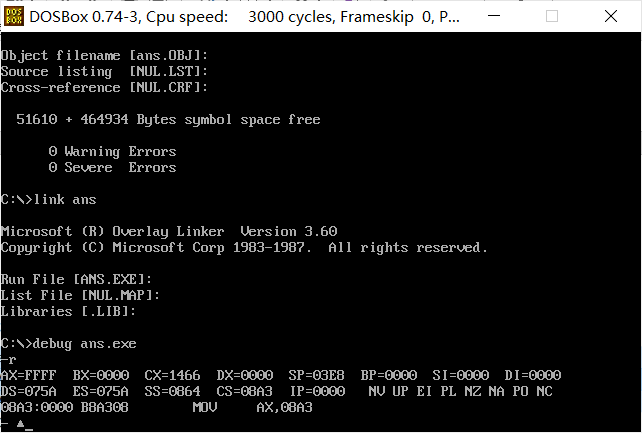


图53 寄存器组初始值

输入指令–u，查看程序反汇编代码：

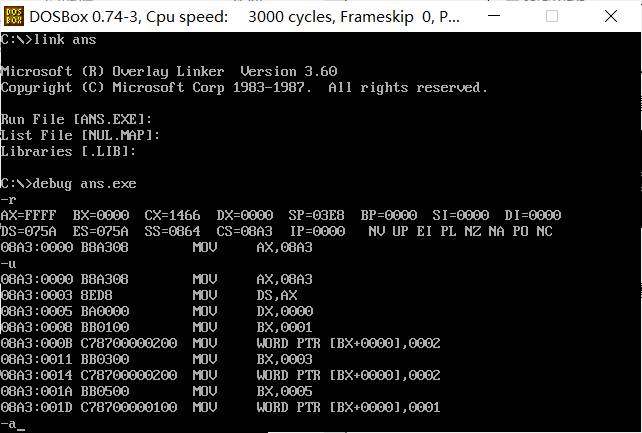


图54 查看程序反汇编代码

1. 最后进行运行调试，显示结果6和3，与预期结果一致。

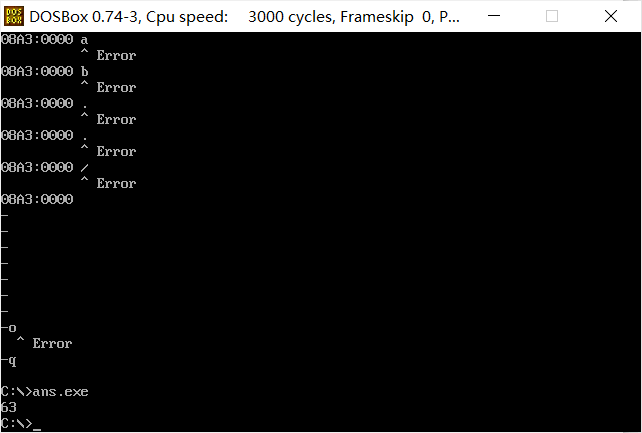


图55 最终结果

# 7 实验体会

本次实验是系统软件开发实践课程的最后一次实验，难度较大。刚开始时，我对利用抽象语法树生成四元式和将四元式翻译成汇编文件的原理不是十分理解，但通过不断查阅资料和阅读探究模板代码，逐渐有了较为清晰的认识。通过实验我了解了c编译器后端的基本原理以及相应的操作流程。通过使用编译器后端程序将前期所得抽象语法树转换成四元式，进而生成汇编代码。再将得到的汇编代码文件在DOSBox环境下编译、链接进而生成可执行程序，最后便可以直接在DOSBox环境下运行。虽然过程艰难，但是带给我的收获也不小，我会在今后的学习和工作中继续钻研探究，继续不断提高自己的能力。