《编译技术》课程设计文档

学号：\_\_\_\_\_\_15061103\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_李海南\_\_\_\_\_\_\_

2018年 1月 13 日

目录

[**一．需求说明** 2](#_Toc503638151)

[**1．文法说明** 2](#_Toc503638152)

[**2．目标代码说明** 16](#_Toc503638153)

[**3. 优化方案\*** 17](#_Toc503638154)

[**二．详细设计** 17](#_Toc503638155)

[**1．程序结构** 17](#_Toc503638156)

[**2．类/方法/函数功能** 18](#_Toc503638157)

[**3．调用依赖关系** 27](#_Toc503638158)

[**4．符号表管理方案** 36](#_Toc503638159)

[**5．存储分配方案** 38](#_Toc503638160)

[**6. 四元式设计\*** 39](#_Toc503638161)

[**7. 目标代码生成方案\*** 41](#_Toc503638162)

[**8. 优化方案\*** 42](#_Toc503638163)

[**9. 出错处理** 43](#_Toc503638164)

[**三．操作说明** 44](#_Toc503638165)

[**1．运行环境** 45](#_Toc503638166)

[**2．操作步骤** 45](#_Toc503638167)

[**四．测试报告** 45](#_Toc503638168)

[**1．测试程序及测试结果** 45](#_Toc503638169)

[**2．测试结果分析** 64](#_Toc503638170)

[**五．总结感想** 64](#_Toc503638171)

**一．需求说明**

**1．文法说明**

扩充c0文法。

文法如下：

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜＜非零数字＞

＜非零数字＞ ::= １｜．．．｜９

＜字符＞ ::= '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'

＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"

＜程序＞ ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}

| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜声明头部＞ ::= int＜标识符＞|char＜标识符＞

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’) }

＜类型标识符＞ ::= int | char

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’

＜无返回值函数定义＞ ::= void＜标识符＞‘(’＜参数表＞‘)’‘{’＜复合语句＞‘}’

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

＜主函数＞ ::= void main‘(’‘)’ ‘{’＜复合语句＞‘}’

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜因子＞ ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜<情况语句>|‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;

|＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;｜＜返回语句＞;

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真

＜循环语句＞ ::= do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞ ::= ＜整数＞|＜字符＞

＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞ ‘}’

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

＜情况子语句＞ ::= case＜常量＞：＜语句＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

＜语句列＞ ::=｛＜语句＞｝

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

＜写语句＞ ::= printf‘(’＜字符串＞,＜表达式＞‘)’|printf ‘(’＜字符串＞‘)’|printf ‘(’＜表达式＞‘)’

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

附加说明：

（1）char类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符

（2）标识符不区分大小写字母

（3）写语句中的字符串原样输出

（4）数组的下标从0开始

（5）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句

文法解读：

1.＜加法运算符＞ ::= +｜-  
＜乘法运算符＞  ::= \*｜/  
＜字母＞   ::= ＿｜a｜．．．｜z｜A｜．．．｜Z  
＜数字＞   ::= ０｜＜非零数字＞  
＜非零数字＞  ::= １｜．．．｜９  
＜字符＞    ::=  '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'

范例：

’+’

’-’ // '＜加法运算符＞'

’\*’

’/’ //'＜乘法运算符＞'

’a’

’A’

’\_’ //'＜字母＞'

’0’ //’<数字>’ ， 0

’1’ //’<数字>’ ，<非零数字>

分析：字符可以是 '＜加法运算符＞'或'＜乘法运算符＞'或'＜字母＞'或'＜数字＞' 中的一个，不能有多个,要注意的是必须有单引号，而且字母并不是只有大小写的英文字母，还包括下划线\_ , 加法运算符是+或-中的一个，乘法运算符是\*或/中的一个，数字是0或者非零数字，非零数字是1到9这9个数字中的一个。

2. ＜字符串＞   ::=  "｛十进制编码为32,33,35-126的ASCII字符｝"

范例：

“” //”空字符串”

“! #￥%&abcdABCD” //” (多个)十进制编码为32,33,35-126的ASCII字符”

分析：字符串由双引号””中放入0个或多个十进制编码为32,33,35-126的ASCII字符构成，值得注意的是中间缺的十进制编码为34的ASCII字符，即 " 只能在字符串两端，可以为空字符串。

3. ＜程序＞    ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞  
＜常量说明＞ ::=  const＜常量定义＞;{ const＜常量定义＞;}  
＜常量定义＞   ::=   int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}  
                            | char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}  
＜标识符＞    ::=  ＜字母＞｛＜字母＞｜＜数字＞｝  
＜变量说明＞  ::= ＜变量定义＞;{＜变量定义＞;}  
＜变量定义＞  ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’) }  
＜类型标识符＞      ::=  int | char  
＜有返回值函数定义＞  ::=  ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’

＜声明头部＞   ::=  int＜标识符＞|char＜标识符＞  
＜无返回值函数定义＞  ::= void＜标识符＞‘(’＜参数表＞‘)’‘{’＜复合语句＞‘}’  
＜参数表＞    ::=  ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞  
＜主函数＞    ::= void main‘(’‘)’ ‘{’＜复合语句＞‘}’

范例：

const int i=0,j=0;const char c1=’a’, c2=’b’;//常量说明

int t[2],x; char s[4],d; //变量说明

int f1(int i, char m){复合语句略，下面详解} //有返回值函数定义

void f2(int j,char n){复合语句略，下面详解} //无返回值函数定义

void main(){复合语句略，下面详解} //主函数

分析1：根据程序的文法可知，程序各个组成成分的声明顺序已经被限定好了，不能随便更改声明顺序，如“int t[2],x; const int i=0,j=0;”此种顺序是不允许的；另外“const int a=10,char b=’A’;”也是不正确的。

分析2：程序按顺序由常量说明部分，变量说明部分，有返回值函数定义或无返回值函数定义部分，主函数部分组成。

其中常量说明，变量说明，有返回值函数定义或无返回值函数定义，主函数之间有严格的先后顺序，不能打乱，而相连的有返回值函数定义和无返回值函数定义可以改变顺序

其中常量，变量，有返回值函数定义和无返回值函数定义对于程序来说是可有可无的，有返回值函数定义和无返回值函数定义可以有0个或多个。只有主函数是必须存在的。

参数表可以为空。

4. ＜表达式＞    ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}  
＜项＞     ::= ＜因子＞{＜乘法运算符＞＜因子＞}  
＜因子＞    ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’

范例：

const int i=0;

const char s=’A’;

int t[3];int z;

int f1(){复合语句略，下面详解}

z=i\*s+t[2]\*f1();

z=(i+s)\*f1();

z=+(i-5)\*f1();

z=-(i+5)/f1(); //多种表达式

分析：表述了表达式组成，包含多种情况，值得注意的是：第一个项前面的+、-是表示第一项的正或负；后面的加法运算符为运算符的+、-；

项可以只由一个因子组成，也可以由因子乘或除以因子组成，因子可以为标识符，＜标识符＞‘[’＜表达式＞‘]’或＜整数＞或＜字符＞或＜有返回值函数调用语句＞或‘(’＜表达式＞‘)’，也就是说字符，数组元素也是可以参与运算的,有返回值的函数调用语句和整数等也可以参与运算。

整数可能是负数，所以需要特别注意：项可能是带负号的，处理时要注意符号和运算符的处理。

5.＜语句＞    ::= ＜条件语句＞｜＜循环语句＞｜<情况语句>|‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞; |＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;｜＜返回语句＞;  
＜赋值语句＞   ::=  ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞  
＜条件语句＞  ::=  if ‘(’＜条件＞‘)’＜语句＞  
＜条件＞    ::=  ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真  
＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜循环语句＞   ::=  do＜语句＞while ‘(’＜条件＞‘)’

＜情况语句＞  ::=  switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞ ‘}’  
＜情况表＞   ::=  ＜情况子语句＞{＜情况子语句＞}  
＜情况子语句＞  ::=  case＜常量＞：＜语句＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’  
＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’  
＜值参数表＞   ::= ＜表达式＞{,＜表达式＞}｜＜空＞  
＜语句列＞   ::=｛＜语句＞｝  
＜读语句＞    ::=  scanf ‘(’＜标识符＞{,＜标识符＞}‘)’  
＜写语句＞    ::=  printf‘(’＜字符串＞,＜表达式＞‘)’|printf ‘(’＜字符串＞‘)’|printf ‘(’＜表达式＞‘)’  
＜返回语句＞   ::=  return[‘(’＜表达式＞‘)’]

范例：

int x,y；

int t[10];

int f1(){

return(1); //返回语句

}

void f2(int i,int j){}

x=1; //两种赋值语句

t[0] = 5\*6;

if(x<3)x=x+1; //条件语句

if(x<3){x=x+1} //条件语句第二种，{语句列}

do printf(x) //写语句

while(x<1) //循环语句

switch(x){

case 1:x=3;

case 2:x=3;

case 3:x=3;

} //情况语句

y = f1(); //有返回值函数调用并赋值

f2(x,y); //无返回值函数调用

scanf(x); //读语句

; //空语句

分析：这部分主要说明了各种语句的组成方式，即说明了程序中能够接受的语语法句格式，值得注意的是循环语句只有do..while一种；

if语句不可以有else。

switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句。

switch中不可以有default语句，没有break关键词。

对于字符的运算，在写语句中输出字符。

return可以有(表达式)，也可以只是return;

6. ＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

范例：

int x;

x=+234;

x=-234;

x=234;

x=0; //多种整数

分析：说明了整数的组成方式，整数可以有正号或负号，也可以没有，值得注意的是无符号整数不可以以0开头，即无符号整数第一位必须是非零数字，之后可以接数字，整数可以为0，但除0以外的数字不可以有前0（第一位不可以为0）。

7. ＜复合语句＞   ::=  ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜语句列＞   ::=｛＜语句＞｝

范例：

void main(){

//主函数内的复合语句

const int x=0; //常量说明

int y; //变量说明

printf(x); //语句列

}

分析：说明了复合语句的组成方式，常量说明，变量说明，语句列之间有严格的先后顺序，不能打乱；

其中常量说明，变量说明对于程序来说是可有可无的，语句列是必须存在的，但值得注意的是语句列可以由0个或多个语句组成。

**2．目标代码说明**

mips指令。

add rd,rs,rt //寄存器加法

addi rt,rs,imm //立即数加

and rd, rs, rt //与

bne rs, rt, offset //不等于时转移

div rd,rs, rt //除

mul rd,rs, rt //乘

jmp //跳转

sub rd,rs,rt //减

subi rd,rs,imm //减立即数

lw rt, offset(base) // 加载字

sw rt, offset(base) //存储字

等等

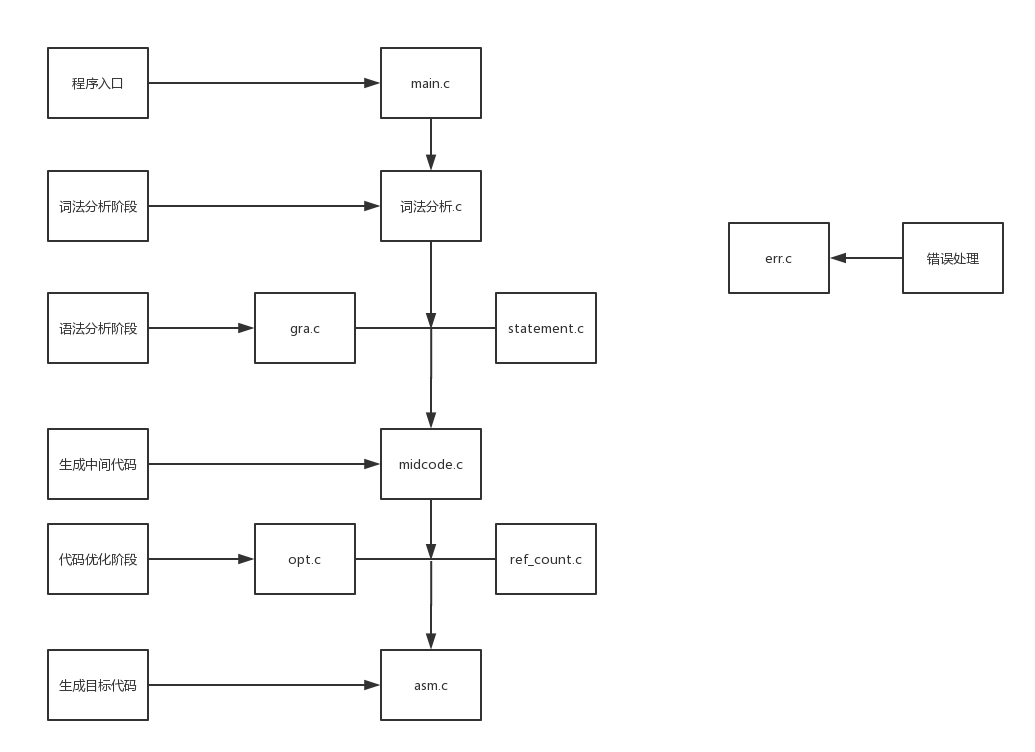
**3. 优化方案\***

（1）.利用DAG图消除局部公共子表达式

（2）.引用计数法分配寄存器。

**二．详细设计**

**1．程序结构**



**2．类/方法/函数功能**

词法分析.c:

setup()函数实现关键字和一些特定符号sy的初始化。

h2l(char c)实现将字母小写化以达到标识符不区分大小写的目的

sytosym(Symbol sy, char sym[])实现将对应sy转换为字符串用于输出。

nextch()实现读取下一个字符并在读完文件后退出。

insymbol()实现读取下一个单词或符号或数字。

gra.c和statement.c:

int loc(char idl[]);

查找标识符

void testsemicolon();

判断分号

enum types testi\_ch();

判断int或char

void enter(char id[], enum objecttyp objtyp, enum types type, int isfun);

填符号表

void entervariable(char temp[], enum types type);

向符号表中填变量

void enter\_arr\_variable(char temp[],enum types type);

向符号表中填数组或变量

int enterconst(enum types type);

向符号表中填常量

void enterarray(enum types type, long size);

向符号表中填数组

void enterid(char name[], enum objecttyp objtyp, enum types type, int adr);

void enterids();

保留字填表

void parameterlist(enum types type);

处理参数

void constdec(enum types type);

常量定义

void v\_a\_fdec(char temp[], enum types type);

变量定义

void statement();

语句处理

void returnstatement();

return语句处理

void printstatement();

写语句处理

void scanfstatement();

读语句处理

void linestatement();

语句列处理

void switchstatement();

switch语句处理

void fact\_sheet(char var[], char label[]);

case语句处理

void dowhilestatement();

循环语句处理

void factor();

因子

void term();

项

void simpleexpression();

表达式

void assignment(struct Tab tabi);

赋值语句处理

void selector(struct Tab tabi);

下标

void codition();

条件

void ifstatement();

条件语句处理

void block();

主程序处理

midcode.c:

void inmidcode(char op[], char v1[], char v2[], char v3[]);

添加新的中间代码到中间代码表

char \* newlabel();

生成新的标签

char \* newvar();

生成新的临时变量

void printmidcode();

打印中间代码

opt.c:

int nodelistloc(int nl, char op[], int left, int right);

查询节点存放的地址

int nodetabnameloc(int nt, char name[]);

节点表按名称查询

void DAG();

生成DAG图并重新导出为中间代码

void nodetabindexloc(int nt, int index);

节点表按索引查询

int can\_in(int x);

DAG导出时判断节点是否可以进入队列

int all\_in(int nl);

判断是否所有中间结点均已进入队列

ref\_count.c:

int refloc(int j, char name[]);

查找变量是否分配了寄存器

void enter\_ref(int j, char name[]);

将变量的引用次数加1，若是未入表的临时变量，将其入表，引用为一。

void ref\_count();

遍历中间代码，以函数为单位统计变量的引用次数

void Sort(int fun);

按引用次数将变量排序。

asm.c:

int isnum(char var[]);

判断是变量还是数字

int addrloc(char name[]);

查找变量地址

int funcloc(char name[]);

查找函数在函数表中的地址

void instack(char var[], int size);

变量入栈

void inaddr(int type, int tmi, int t, int glob);

填入变量地址表

void midcodetoasm();

遍历中间代码，将中间代码转化为目标代码

void callop();

函数调用处理

void GOTOop();

无条件跳转

void BZop();

条件不为1，跳转

void labop();

标签

void plusop();

加

void minuop();

减

void mulop();

乘

void divop();

除

void grtop();

大于

void geqop();

大于等于

void lssop();

小于

void leqop();

小于等于

void eqlop();

等于

void neqop();

不等于

void assop();

赋值

void sassop();

数组元素赋值

void getaop();

将数组元素的值赋给变量

void scfop();

输入

void prtop();

输出

void fpaop();

函数调用时参数的值

void retop();

返回语句

void paraop();

参数

void funcop();

函数

err.c:

void endskip();

在跳过的部分画下划线（错误文件中）

void skipstatement();

跳到下一条语句

void skiprparent();

跳到右括号

void skiprbrack();

跳到右方括号

void skipcase();

跳到右大括号或case

void skiprbrace();

跳到右大括号

void skipfacnext();

跳到下一个因子

void skipfundec();

跳到函数声明

void skipdec();

跳到变量声明

void skipcondec();

跳到常量声明

void skipsem();

跳到分号

void skiplbrace();

跳到左大括号

void err(int n);

输出错误信息。

**3．调用依赖关系**

main setup()和insymbol()；

|  |  |
| --- | --- |
| 函数 | 被调用函数 |
| main() | block() DAG() enterids() err(int n) insymbol() midcodetoasm() printmidcode() setup() ref\_count() |
| block() | constdec(types type) err(int n) i\_cfundec(types type) insymbol()skipfundec() skiplbrace()v\_a\_fdec(char temp[], types type) |
| DAG() | all\_in(int nl) can\_in(int x) isnum(char var[])nodelistloc(int nl, char op[], int left, int right) nodetabindexloc(int nt, int index) nodetabnameloc(int nt, char name[]) |
| enterids() | enterid(char name[], objecttyp objtyp, types type, int adr) |
| insymbol() | err(int n) h2l(char c) nextch() sytosym(Symbol sy, char sym[]) |
| midcodetoasm() | funcloc(char name[])funcop()inaddr(int type, int tmi, int t, int glob) instack(char var[], int size) |
| ref\_count() | enter\_ref(int j, char name[])err(int n) funcloc(char name[])Sort(int fun) |
| constdec(types type) | enterconst(types type) insymbol() skipsem()testi\_ch()testsemicolon() |
| i\_cfundec(types type) | constdec(types type) enter(char id[], objecttyp objtyp, types type, int isfun) enterfun(char name[], int ref, types type) err(int n) inmidcode(char op[], char v1[], char v2[], char v3[]) insymbol()linestatement()parameterlist(types type) skipcondec()v\_a\_fdec(char temp[], types type) |
| skipfundec() | endskip()insymbol() |
| skipsem() | endskip()insymbol() |
| skipstatement() | endskip()insymbol() |
| skipcondec() | endskip()insymbol() |
| skiplbrace() | endskip()insymbol() |
| v\_a\_fdec(char temp[], types type) | enter\_arr\_variable(char temp[], types type) entervariable(char temp[], types type) insymbol()testsemicolon() |
| nextch() | endskip() |
| funcop() | addrloc(char name[]) assop() BZop() callop() divop() eqlop() fpaop() funcloc(char name[])geqop()getaop()GOTOop()grtop()inaddr(int type, int tmi, int t, int glob) instack(char var[], int size) labop()leqop()lssop()minuop()mulop()neqop()paraop()plusop()prtop()refloc(int j, char name[])retop()sassop()scfop() |
| assop()callop()divop()eqlop()geqop()getaop()grtop()leqop()lssop()minuop()mulop()neqop()plusop()prtop()retop()sassop() | addrloc(char name[])isnum(char var[])refloc(int j, char name[]) |
| fpaop() | addrloc(char name[])funcloc(char name[])isnum(char var[])refloc(int j, char name[]) |
| paraop() | inaddr(int type, int tmi, int t, int glob) |
| scfop() | addrloc(char name[])refloc(int j, char name[]) |
| enterconst(types type) | enter(char id[], objecttyp objtyp, types type, int isfun) err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()skipsem() |
| enter(char id[], objecttyp objtyp, types type, int isfun) | err(int n) |
| testi\_ch() | err(int n) insymbol() |
| testsemicolon() | err(int n) insymbol() |
| linestatement() | err(int n) skipstatement()statement() |
| parameterlist(types type) | enter(char id[], objecttyp objtyp, types type, int isfun) err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()parameterlist(types type) skiplbrace() |
| enter\_arr\_variable(char temp[], types type) | enter(char id[], objecttyp objtyp, types type, int isfun) enterarray(types type, long size) err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()skipsem() |
| entervariable(char temp[], types type) | enter(char id[], objecttyp objtyp, types type, int isfun) inmidcode(char op[], char v1[], char v2[], char v3[]) |
| enterarray(types type, long size) | err(int n) |
| statement() | assignment(Tab tabi) dowhilestatement()err(int n) ifstatement()inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()linestatement()loc(char idl[])printstatement()returnstatement()scanfstatement()simpleexpression()skiprparent()skipsem()switchstatement()testsemicolon() |
| assignment(Tab tabi) | err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()selector(Tab tabi) simpleexpression()skipsem()testsemicolon() |
| dowhilestatement() | codition()err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newlabel()statement() |
| ifstatement() | codition()err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newlabel()statement()skipsem() |
| printstatement()returnstatement() | err(int n) inmidcode(char op[], char v1[], char v2[], char v3[]) insymbol() simpleexpression()skiprparent()skipsem()testsemicolon() |
| scanfstatement() | err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()loc(char idl[])skipsem()testsemicolon() |
| simpleexpression() | inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newvar()term() |
| skipcase() | endskip()insymbol() |
| skiprbrace() | endskip()insymbol() |
| skiprparent() | endskip()insymbol() |
| switchstatement() | err(int n) fact\_sheet(char var[], char label[])inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newlabel()simpleexpression()skipcase()skiprbrace()skiprparent() |
| selector(Tab tabi) | err(int n) insymbol()simpleexpression()skiprbrack() |
| skiprbrack() | endskip()insymbol() |
| term() | factor()inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newvar() |
| factor() | err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()loc(char idl[])newvar()selector(Tab tabi) simpleexpression()skiprparent() |
| codition() | inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newvar()simpleexpression()skiprparent() |
| fact\_sheet(char var[], char label[]) | err(int n) inmidcode(char op[], char v1[], char v2[], char v3[])insymbol()newlabel()skipcase()skiprbrace()statement() |

**4．符号表管理方案**

tab:

struct Tab

{

char name[100];

int paranum;

enum objecttyp obj;

enum types type;

int ref;

int normal;

int adr;

int link;

};

struct funtab

{

char name[100];

int paranum;

int offset;

int rt;

struct Ref\_Tab ref\_tab[100];

int ref;

int size;

enum types type;

}ftab[bmax];

struct atab

{

types eltype;

int low, high, elsize, size;

};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 常量 | 变量 | 数组 | 函数 |
| name | 名称 | 名称 | 名称 | 名称 |
| paranum | 0 | 0 | 0 | 参数个数 |
| obj | constant | variable |  | funktion |
| type | int/char | int/char | arrays | 返回值类型int/char |
| ref | 0 | 0 | 数组在atab中的位置 | 函数在btab中的位置 |
| normal |  |  |  | 变量形参为0 |
| adr | 值或ascii码 |  |  |  |

link指向同一分程序的上一个标识符在tab表中的位置

ftab中：name函数名，paranum参数个数，offset 此函数中下一个参数的偏移量，rt和ref\_tab分别是此函数中变量个数和每个变量对应的引用次数（已排序），ref指函数在符号表中的地址，size函数中变量个数（包含临时变量），type函数返回类型。

atab：eltyp为元素类型，low，high为下、上界，elsize为元素大小，size为数组大小。

每个表均有入表，查表等操作。

**5．存储分配方案**

将sp上移（值变小），自顶向下存放全局变量，并将全局变量的顶端保存在t8寄存器中，每次调用函数，先进行保存现场，sp上移足够的空间（40），将s0-s7，被调用函数的上一层运行栈的栈底指针（t9）以及ra寄存器保存在内存中，用t9保存当前层栈底指针，将sp指针根据此函数中的变量个数向上移动到最顶端，根据偏移量确定变量（局部变量），函数结束时恢复现场，将sp赋值为栈底指针，将保存的上一层信息lw回对应寄存器，sp下移保存现场时上移的空间大小。

计划使用引用计数的算法进行寄存器的分配。

图片包含 屏幕截图

已生成极高可信度的说明

**6. 四元式设计\***

const,char,num,name //常量

int ,,,name //变量

ints,,num,name //数组

para,int,,name //参数

func,int,,name

func,void,,name //函数

end,,,name //函数结束

minu,var1,var2,var3 //var3=var1-var2

plus,var1,var2,var3 //var3=var1+var2

mul,var1,var2,var3 //var3=var1\*var2

div,var1,var2,var3 //var3=var1/var2

[]=,var1,var2,var3 //var3[var2] = var1

=,var1,,var3 //var3 = var1

geta,name,t,nowvar //nowvar = name[t]

<,var1,var2,var3 //var3=var1<var2

<=,var1,var2,var3 //var3=var1<=var2

>,var1,var2,var3 //var3=var1>var2

>=,var1,var2,var3 //var3=var1>=var2

==,var1,var2,var3 //var3=var1==var2

!=,var1,var2,var3 //var3=var1!=var2

BZ,,,lab //不为1则跳转

GOTO,,,lab //跳转

lab:,,,lab //设置标签

call,name,, //函数调用（语句）

fpa,,,nowvar //值参数

acall,name,,var3 //var3调用后的返回值

ret,,,var3 //返回

prt,var1,var2,char/int //写语句

scf,,,name //读语句

**7. 目标代码生成方案\***

struct Addr

{

char name[100];

int addr;

int type;

int isglo;

}addrlist[2000];

int ap=0;

通过此表保存变量及其地址，类型，是否全局变量，若是全局变量，则基地址为t8，若不是全局变量，则基地址为sp。未优化时从变量（非数字）对应地址取数据到t0或t1，t2等进行计算，计算结果要保存到变量对应地址中，优化后判断是否有当前变量对应的寄存器，若有，则直接使用该寄存器，否则从内存中存取。

对于PUSH语句，即值参的传递，通过将每个函数的参数顺序固定并都从函数顶端开始存放，将值参直接放到使用它的函数中的确定位置，对于参数与函数的对应，在中间代码部分保证PUSH与call之间没有其他操作，且PUSH的值即为第一次遇到的call中的函数。

**8. 优化方案\***

优化一：DAG图

struct Nodetab

{

char name[100];

int index;

}nodetab[100];

struct Nodelist

{

char op[100];

int is\_leaf;

int in\_order;

int pi;

int left\_index;

int right\_index;

int parent\_index[100];

}\*nodelist[100] = { NULL };

notetab为结点表，nodelist为结点地址表，notetab中的index存放的就是该结点在nodelist数组中的下标，结点结构中的左右孩子和双亲都是nodelist数组中的下标。叶子结点中如果是变量，则将该变量变为$变量，同时生成一条$变量=变量的语句。算法即为书中的生成DAG图，以及导出的启发式算法，不过我改进了一下，若结点中有程序中的变量，导出时将程序中的变量全部导出，临时变量删掉，若只有临时变量，则选择第一个保留，其余删掉。另外导出时还要遍历叶子结点，若叶子结点中出现多个变量，则输出这多个变量互相相等。

优化二：引用计数法分配寄存器

struct Ref\_Tab

{

char name[100];

int num;

};

在函数表中添加上述引用表，将函数中的每个变量都加入表中，并初始化num为1，遍历函数，统计每个变量出现的次数，最后进行降序排序，在生成目标代码时，使用

int refloc(int j,char name[]) {

int i;

for (i = 0; i < ftab[j].rt && i<8; i++) {

if (strcmp(ftab[j].ref\_tab[i].name, name)==0) {

return i;

}

}

return -1;

}

此函数进行查询，函数返回的数字是-1，则说明无寄存器，否则返回的数字即是寄存器的编号。

**9. 出错处理**

出错处理包括打印错误信息（错误类型和错误行号和错误字符号）和跳过错误代码组成，跳过错误代码详见第2点err.c中的函数介绍，错误信息如下：

char msg[100][100]= {

"缺少源文件", //0

"文件不完整", //1

"缺少\"", //2

"不可接受的字符", //3

"缺少“'”", //4

"超出符号表范围", //5

"变量名冲突", //6

"常量声明失败", //7

"等号右侧字符非法", //8

"丢失“;”", //9

"关键字错误", //10

"丢失标识符", //11

"丢失“]”", //12

"调用函数未定义", //13

"不允许前0", //14

"形参个数不匹配", //15

"未定义变量", //16

"丢失“(”", //17

"丢失“)”", //18

"丢失“}”", //19

"函数返回错误", //20

"表达式缺失或错误", //21

"句子不合法", //22

"给常数赋值", //23

"缺少“{”", //24

"缺少main函数", //25

"常量没有初始化", //26

"数字有前零", //27

"数字过大", //28

"字符串内容错误", //29

"字符内容错误", //30

"变量声明失败", //31

"参数错误", //32

"函数声明错误", //33

"文法中不允许有default", //34

"数组下标越界" //35

};

**三．操作说明**

**1．运行环境**

windows系统，直接用vs2010打开项目，点击运行即可。

**2．操作步骤**

右键点击编译器.sln，使用VS2010打开，在解决方案资源管理器中打开main.c文件，点击调试窗口下的直接运行即可开始运行。

**四．测试报告**

**1．测试程序及测试结果**

测试1：

Const int MAX=100;

const CHAR t = 'a';

int token[100];

int num;

int is\_err;

int x,y;

char c;

int f1(int i){

return(i);

}

char f2(){

char n;

n='a';

return(n);

}

void Do\_F() {

int count;

int i;

int num1;

int flag;

char c1;

num1 = 0;

c1 = c;

num1 = 0;

if (c>= '0') {

if(c<='9'){

do

{

token[count] = c-'0';

num1 = num1 \* 10 + token[count];

count = count + 1;

scanf(c);

flag =0;

if(c>='0'){

if(c<='9'){

flag = 1;

}

}

}while (flag)

num = num1;

}

}

}

void Do\_T() {

int x,y;

Do\_F();

switch(c){

case'\*':{

x = num;

scanf(c);

Do\_F();

y = num;

num = x \* y;

}

case'/': {

x = num;

scanf(c);

Do\_F();

y = num;

num = x / y;

}

}

}

void Do\_E() {

int x,y;

Do\_T();

switch(c){

case'-':{

x = num;

scanf(c);

Do\_T();

y = num;

num = x - y;

}

case'+':

{

x = num;

scanf(c);

Do\_T();

y = num;

num = x + y;

}

case'q':

return;

}

}

void main() {

const int m = -1;

char n;

x=0;

is\_err = 0;

y=0;

x=-x;

y=-y;

if(x>m){

x=0;

}

if(m<y){

y=0;

}

x=(x-1)\*1+1;

n=f2();

x=f1(x);

y = f1(y);

scanf(c);

Do\_E();

if(is\_err == 0){

printf("result:",num);

}

;

}

输入：2+3\*5q

输出：17

测试程序二：

Const int max = 10;

void main(){

int i;

i=0;

do{

i=i+1;

printf(i);

}while(i<mAX)

}

输入：

输出：12345678910

测试三：

Const int max = 10;

void main(){

int i;

int a;

i=0;

do{

scanf(a);

switch(a){

case 1:i=i-1;

case 2:i=i+5;

case 3:i=i\*2;

case 4:i=i/3;

}

printf(i);

if(I>Max){

return;

}

}while(1)

}

输入：1 3 2 4 2 2

输出：-1 -2 3 1 6 11

测试四：

int max;

void test1(int i,int Max){

max = MaX-20;

i=i+1;

if(MAx>0){

test1(i,MAX);

}

printf("the",i);

printf("asddd\n",max);

}

void main(){

int i;

int max;

i=0;

max=100;

test1(i,max);

}

输入：

输出：the5asddd\n0the4asddd\n20the3asddd\n40the2asddd\n60the1asddd\n80

测试五：

int max;

void test1(int i,int Max){

printf("the",i);

printf("asddd\n",max);

}

int test2(int i, int j){

return(10);

}

int test3(int i, int j){

return(20);

}

void main(){

int i;

int max;

i=0;

max=100;

test1(test2(1,2+3),test3(i,max));

}

输入：

输出：the10asddd\n20

以上五个测试程序均正确，且结果与预期相符。

错误一：

int max;

void test1(int i,int Max){

printf("the",i);

printf("asddd\n",max);

}

int test2(int i, int j){

return(10);

}

int test3(int i, int j){

return(20);

}

void main(){

int i;

int max

i=0;

max100;

test1(test2(1,2+3),test3(i,max));

}

结果：第16行，第三个字符：丢失“；”

第17行，第8个字符：未定义变量

错误二：

int max;

void test1(int i,int Max){

max = MaX-20;

i=i+1;

if(MAx>0){

test1(i,MAX);

}

printf("the",i);

printf("asddd\n",max);

}

void main(){

int i;

int max;

int max;

i=0;

max=100;

test1(i,max);

}

结果：第15行，第9个字符：变量名冲突

错误三：

void main(){

int i;

int a;

i=0;

do{

scanf(a);

switch(a){

case 1:i=i-1;

case 2:i=i+5;

case 3:i=i\*2;

case 4:i=i/3;

}

printf(i);

if(I>Max){

return;

}

}while(1)

}

结果：第15行，第11个字符：未定义变量

错误四：

Const int max = 10;

void main()

int i;

i=0;

do{

i=i+1;

printf(i);

}while(i<mAX)

}

结果：第3行，第5个字符：缺少“{”

错误五：

Const int MAX=100;

const CHAR t = 'a';

int token[100];

int num;

int is\_err;

int x,y;

char c;

int f1(i){

return(i);

}

char f2(){

char n;

n='a';

return(n);

}

void Do\_F() {

int count;

int i;

int num1;

int flag;

char c1;

num1 = 0;

c1 = c;

num1 = 0;

if (c>= '0') {

if(c<='9'){

do

{

token[count] = c-'0';

num1 = num1 \* 10 + token[count];

count = count + 1;

scanf(c);

flag =0;

if(c>='0'){

if(c<='9'){

flag = 1;

}

}

}while (flag)

num = num1;

}

}

}

void Do\_T() {

int x,y

Do\_F();

switch(c){

case'\*':{

x = num;

scanf(c);

Do\_F();

y = num;

num = x \* y;

}

case'/': {

x = num;

scanf(c);

Do\_F();

y = num;

num = x / y;

}

}

}

void Do\_E() {

int x,y;

Do\_T();

switch(c){

case'-':{

x = num;

scanf(c);

Do\_T();

y = num;

num = x - y;

}

case'+':

{

x = num;

scanf(c);

Do\_T();

y = num;

num = x + y;

}

case'q':

return;

default:

}

}

void main() {

const int m = -1;

char n;

x=0;

is\_err = 0;

y=0;

x=-x;

y=-y;

if(x>m){

x=0;

}

if(m<y){

y=0;

}

x=(x-1)\*1+1;

n=f2();

x=f1(x);

y = f1(y);

scanf(c);

Do\_E();

if(is\_err == 0){

printf("result:",num);

}

;

}

结果：第10行，第10个字符：参数错误

第10行，第11个字符：参数错误

第11行，第13个字符：未定义变量

第52行，第9个字符：丢失“;”

第94行，第9个字符：文法中不允许

第115行，第11个字符：丢失“)”

第116行，第13个字符：丢失“)”

**2．测试结果分析**

覆盖所有文法中的语法成分，考虑了多种特殊情况，考虑了忽略标识符大小写等，错误包含了语法错误，语义错误，能够同时处理多个错误。

**五．总结感想**

学习这门课之后，我了解了代码究竟是如何运行的，通过自己实现编译器，我对c语言的理解也变得更深，写代码的过程也更加得心应手，这门课对比面向对象和计算机组成，我认为这门课的开放度更大，老师讲了知识，给了文法，其他内容都需要自己去想，书上给的算法都是比较概括的说明，具体的数据结构，一些特殊情况等都需要自己去研究，所以纵使我们已经经过了计算机组成和面向对象的“洗礼”，也需要耗费大量的时间来完成这门课。接下来说说我的经历吧，首先是对Pascal编译器的代码阅读，这部分真的很重要，因为在一切都还没开始之前，我相信很多人像我一样一头雾水，读懂Pascal代码可以使我们对编译器的整体架构有一个更好的了解。接下来的词法分析比较简单，基本上仿照Pascal就可以完成，再之后，由于当时的理解不够深刻，语法分析和语义分析分不清楚，所以写出来的代码完成了语法分析之后把语义分析也完成了大部分，现在想想那一周有点痛苦，这一部分需要对整体架构有很深的理解，不要想着代码太多就慌了神，认真想想老师说过的话：每一个非终结符都是一个函数，先把所有函数命名好，写出空函数，然后从顶端入手，分析到什么就调用什么函数，尽管那个函数你可能还没有完成，写出所有框架后再去补充细节，这样思路就会很清晰。中间代码和目标代码在同一周内完成，这部分值得注意的就是千万不要以为把中间代码完成了就完成了一半，可以说这时候连四分之一都没有完成。再写目标代码的时候一定要想清楚栈的结构，记得当时我周二完成了中间代码，然后就开始写目标代码，可是一直没有想清楚架构，改了又改，直到周六猜想清楚并确定自己的栈结构，当天便完成了目标代码生成。至于代码优化部分，这部分自己根据自己的中间代码多想一想，书中给的优化算法有着很大缺陷，需要你根据自己的代码进行改进。

总之，编译这门课给我的感觉还不错，除了检查的时候瑟瑟发抖外，这门课让我确确实实的感受到了自己在进步，感谢这门课，感谢老师学长学姐们的指导。