《编译技术》课程设计文 档

学号：\_\_\_\_\_16061002\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_张凯宁\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2019年 1 月 4 日

## 一．需求说明

### 1．文法说明

＜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＞｝| 0

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜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＞']' )} //＜无符号整数＞表示数组元素的个数，其值需大于0

＜常量＞ ::= ＜整数＞|＜字符＞

＜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＞ ::= ＜参数＞{,＜参数＞}| ＜空>

＜参数＞ ::= ＜类型标识符＞＜标识符＞

＜主函数＞ ::= void main'('')''{'＜复合语句＞'}'

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞| '{'＜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＞;|＜情况语句＞｜＜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＞ ::= while '('＜条件＞')'＜语句＞

＜情况语句＞ ::= switch '('＜表达式＞')' '{'＜情况表＞＜缺省＞ '}'

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

＜情况子语句＞ ::= case＜常量＞：＜语句＞

＜缺省＞ ::= default : ＜语句＞|＜空＞

＜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类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符区分大小写字母

（3）写语句中，字符串原样输出，单个char类型的变量或常量输出字符，其他表达式按整型输出

（4）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

1. 数组的下标从0开始

· 文法解读说明

### 【1】

### ＜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＞

### 例：

const int a = 6, b = 7; //常量说明,以分号结尾

const int k = 0; //此种说明也是允许的

const char c = ‘d’;

int a\_var, a\_array[10]; //变量说明

char b\_var;

char b\_array[20];

int func1(){//没有参数也要有小括号

...

return ... //没有这句也符合文法,但需要报错。.return后面没有值依旧需要报错

}

void func2(int a,int b){

...

}

void main(){

[<常量说明>]

[<变量说明>]

{语句} //允许语句列的零次出现

...

}

### 分析：

定义了<程序>的组成部分和严格出现顺序。首先，<常量说明>和<变量说明>可有可无，而＜有返回值函数定义＞和＜无返回值函数定义＞中的任何一个都可以交替出现多次，也可以不出现。接下来，必须有且仅有一个<主函数>。也即如果出现，顺序也应遵循：常量说明-字符串说明-有返回值函数定义/无返回值函数定义-主函数 的字符串组合。

附加：

（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符区分大小写字母

（3）写语句中，字符串原样输出，单个char类型的变量或常量输出字符，其他表达式按整型输出

（4）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

（5）数组的下标从0开始

### 【2】

### ＜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＞｝

### ＜整数＞ ::= ［＋｜－］＜无符号整数＞

### ＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝| 0

### ＜非零数字＞ ::= １｜．．．｜９

### ＜数字＞ ::= ０｜＜非零数字＞

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

### ＜加法运算符＞ ::= +｜-

### ＜乘法运算符＞ ::= \*｜/

### ＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

### 分析：

1. 常量的声明，前面必须加const关键字才能生效。后接<常量定义>，最后以分号结尾。可以重复一次或多次。同一类型的常量可以分开说明。定义到的常量必须要赋值。

例：const int a = 1 ; const int b = 2;（正确） const char a;（错误）

1. <常量定义>可以是int型或char类型的定义。一个const后面只跟一次char或int，但可定义多个同类型变量，变量格式为赋值语句<标识符>=<整数>或<字符>，变量之间用逗号隔开。常量定义的时候等于号后面只能是整数或者字符,不能是表达式

例：const int a = 1,b = 2,c = 3  char a = ‘1’,b = ‘2’,c = ‘3’（正确） const int a = 2\*3 （错误）

1. <标识符>必须由<字母>开头，之后可以跟0个或多个<字母>或<数字>。注意字母可以是a~z和A~Z和\_，但不能是除此之外的其他特殊字符。

例：a1,aa,\_1,\_a

1. <整数>可以有‘+’和‘-’号也可以没有。后跟<无符号整数>。若是存在符号的无符号整数，则只能有一个符号，要么为+，要么为- ，可以有+0 -0

例：+9,-1,+0,-0（正确） +-0（错误）

1. <无符号整数>除可以是0外，其他以0为开头的数字都视为非法。

例：0,123（正确） 001（错误）

1. <数字>（0~9）和<非零数字>(1~9)都是一位数，和常识的理解一致。

例：9,0,1（正确）99,00（错误）

1. <加法运算符>不只是包括‘+’还包括‘-’
2. <乘法运算符>不只是包括‘\*’还包括‘/’
3. <字母>可以是大写和小写的a-z，注意还可以是\_
4. <字符>包括<加法运算符>、<乘法运算符>、<字母>、<数字>

（11）运算时'A'！= 'a'

（12）字母中的"\_"指的是ASCII码为95的下划线

＜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＞']' )} //＜无符号整数＞表示数组元素的个数，其值需大于0

### 【3】

### ＜类型标识符＞ ::= int | char

### ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

### ＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝| 0

### 例：

int i;（正确） int i=0（错误）

char s[20];（正确）

### 分析：

1. <变量说明>中的<变量定义>后加;结束。

（2）<变量定义>的时候不允许被赋值,例如: int a = 10;这种语法不支持。每次定义类型标识符只写一次（只能是int和char），后面可以跟一个或多个一般标识符或一般标识符加中括号即数组，标识符之间用逗号隔开，注意逗号为英文逗号，且中括号中的无符号整数即数组的元素个数必须大于0，也即此时＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝.但使用时下标仍然从0开始，不要混淆。

（3）<类型标识符>只支持int和char。

（4）字符数组的声明是char类型。

（5）特殊规定：char类型的变量或常量，用字符的ASCII码对应的整数参加运算，但如果是单个字符打印输出就输出该字符而不是ascll码。

### 【4】

### ＜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＞

### ＜参数表＞ ::= ＜参数＞{,＜参数＞}| ＜空>

### ＜参数＞ ::= ＜类型标识符＞＜标识符＞

### ＜类型标识符＞ ::= int | char

### ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

### ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

### 例：

int add(int a,char b){

const int c = 5;

int d;

d = a+b+c;

return d;

}

### 分析：

（1）<参数表>里面可以声明0个、1个或多个参数，多个参数用,隔开，必须一个类型标识符跟一个标识符,不允许出现"int add(int a,b)"这种形式，也不允许没有类型标识符或没有标识符。

（2）定义时，小括号后面都跟‘{’，函数均以'}'结尾,后面都没有分号.

（3）参考语句的格式知道，允许在函数内部再出现"{}"包住的代码块

（4）由于有"＜语句＞ ::= ...|＜空＞;|...",所以可以存在 int add{;} 这种情况

（5）一定要检查有无返回语句及其格式，返回值的类型和声明的类型是否相符。

（6）声明头部一定要先写返回值类型标识符，int和char二者只能选择其一进行声明。

（7）<复合语句>必须有语句列，但语句可为空，但语句后一定要有分号。可选项常量说明必须在变量说明前，变量说明必须在语句列前。

（8）如果传入的参数是字符，在函数运算时当做ascll码参与处理。

（9）虽然语句列可以是语句的零次及以上出现，但int abc(){}是不允许的，因为没有return 语句

（10）应该注意检查有返回值的函数是否有<返回语句>，返回语句的格式是否正确。而无返回值函数则可以有return也可以没有。有返回值的函数仅支持int 和char两种返回类型,不支持返回数组和指针。不存在return()的情况。返回语句后面要有分号

（11）函数调用语句若没有参数也需要加小括号,后面以分号结尾

### 【5】

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

### ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

### ＜参数表＞ ::= ＜参数＞{,＜参数＞}| ＜空>

### ＜参数＞ ::= ＜类型标识符＞＜标识符＞

### ＜类型标识符＞ ::= int | char

### ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

### ＜语句列＞ ::= ｛＜语句＞｝

### 例：

void abc(){

printf(“zkn”);

}

### 分析：

（1）由于有"＜语句＞ ::= ...|＜空＞;|...",所以可以存在void main(){;} 这种情况

（2）没有参数的函数定义及其调用需要有小括号

（3））因为"＜语句列＞ ::=｛＜语句＞｝",所以允许语句列的零次出现,即可以有:

void abc(){}

void abc(){;}

void abc(){

const int a = 0;

char b = 'c';

}

这两种情况出现

（4）其他部分内部性质参考有返回值函数。可以有返回语句也可以没有。有的话必须是return;而不是return();

### 【6】

### ＜主函数＞ ::= void main'('')''{'＜复合语句＞'}'

### ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

### 例：

void main(){

const char c = ‘z’;

Int a;

printf(“zkn”);

}

### 分析：

1. main函数虽然没有参数但是有小括号

（2）函数不能重名

（3）因为"＜语句列＞ ::=｛＜语句＞｝",所以允许语句列的零次出现,即可以有:

void main(){}

void main(){

const int a = 0;

char b = c;

}

这两种情况出现

（4）因为语句可以为<空>; 可以有:void main(){;;;}

### 【7】

### ＜返回语句＞ ::= return['('＜表达式＞')']

### ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

### 例：

return(+a)

### 分析：

1. 应该注意检查有返回值的函数是否有<返回语句>，返回语句的格式是否正确。而无返回值函数则可以有return也可以没有。有返回值的函数仅支持int 和char两种返回类型,不支持返回数组和指针。不存在return()的情况。返回语句后面要有分号

（2）[+|-]只作用于第一个<项>，后面可以跟‘+’/‘-’和其他项

（3）除可以返回变量标识符以外还可以返回表达式，项也可以是标识符

（4）若有返回值则必须在表达式外加上（）。

（5）"return"后面如果有表达式则需要加小括号,且只能出现一个小括号一个表达式

（6）"return;"这种形式,在main函数和无返回值函数中允许; 在有返回值函数中出现时则需要报错。

### 【8】

### ＜语句＞    ::= ＜条件语句＞｜＜循环语句＞| '{'＜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＞;|＜情况语句＞｜＜返回语句＞;

### 分析：

有些类型的语句最后需要用;，有些不用。有分号结尾的情况，除了条件、循环、情况语句和'{'＜语句列＞'}'之外，都有分号结尾。空语句也有一个分号。

### 【9】

### ＜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

### ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

### ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

### ＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

### 例：

If(+(a[b\*c+d)]-8)==-m){

printf(“zkn”);

;

}

### 分析：

1. 条件可以是一个表达式，表达式为0条件为假，否则为真，但不能为空。
2. 条件也可以是两个表达式做关系运算的结果。
3. 组成表达式的项的因子，可以是标识符、数组（数组的方括号里面可以是表达式但要符合要求）、用括号括起来的表达式（不能不用括号）、整数、字符（使用ascll码参与运算）、或者有返回值的调用语句（注意写参数和()）。

（4）因为项也可以推出+/-号，所以表达式可以是+-、++、-- 、-+作为前缀

### 【10】

### ＜循环语句＞ ::= while '('＜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

### 例：

while(+(a[b\*c+d)]-8)==-m）{

m=m+1;

}

### 分析：

语法和if类似，循环执行

### 【11】

### ＜有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＞

### 例：

M=abc(); t=abc(a,b,c);

### 分析：

1. 注意有返回值函数调用语句在语句定义中，其应有的分号在语句定义中体现。
2. 参数不得和定义不符，包括类型、顺序，个数（语义）。多个参数使用,隔开，参数是表达式，也可以是整体是空（无参数），但不可以传入多个参数其中一个为空。
3. 调用语句必须后接;，这一点定义在语句中

### 【12】

### ＜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＞

### 例：

A();

### 分析：

（1）没有返回值。不能用一个变量在左边接收它的赋值。

（2）调用语句必须后接;，这一点定义在语句中

### 【13】

### ＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞'['＜表达式＞']'=＜表达式＞

### 例：

q = 1

W[8]= ‘c’

W[a\*9+10]=1

### 分析：

1. 赋值语句左边可以是普通的标识符和数组，数组的下标可以用表达式表示，右边也都是表达式。在使用数组时下标可以从0开始，但定义时需要大于0
2. 赋值语句的;在<语句>中定义

### 【14】

### ＜读语句＞ ::= scanf '('＜标识符＞{,＜标识符＞}')'

### 例：

scanf(a,b,c)

### 分析：

（1）可以一次性读入一个或多个标识符，用,隔开，但不能scanf()

（2）以标识符的类型为准，如scanf内的标识符a为一个int值，输入一个字符，则将该字符的ascll码带入赋值运算

（3）输入时不能用整数代替ascll码，键盘输入的是什么字符就是什么字符。

（4）scanf(a) 如果a是int型，输入c，a=’c’的ascll码值 如果a是char型，输入1，则a=’1’

### 【15】

### ＜写语句＞ ::= printf '(' ＜字符串＞,＜表达式＞ ')'| printf '('＜字符串＞ ')'| printf '('＜表达式＞')'

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

### ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

### ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

### ＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

### 例：

Printf(“a=”,a)

### 分析：

1. 写语句中，字符串原样输出，单个char类型的变量或常量输出字符，其他表达式按整型输出。也即写语句可以输出一个字符串或者可以输出一个表达式的值再或者同时输出一个字符串和一个表达式，中间用逗号隔开，顺序不可调换。
2. 认为printf自带换行，如果字符串内容有\n则原样输出。
3. printf(表达式)： 如果是整数，则直接输出整数的值；如果是单个字符，例如b(b=’x’)或a(a=’1’)则输出该字符；如果是数组成员，即标识符[表达式]，则输出其值即可；如果是标识符（表达式,表达式,表达式...），即函数调用返回语句，这里语义上应该是有返回参数的函数调用返回语句，输出返回值即可；如果是带运算的表达式，每个字符转换为ascll码参与运算即可，最后输出表达式结果。
4. printf(字符串):字符串原样输出不支持变量替换。
5. Print(字符串,表达式)：字符串远洋输出，表达式按上面的（3）类似处理。Printf(“a=”,a)如果a为整型则输出结果为a=1.但如果a是char型，则输出代表的字符。

（6）char a[10]; printf(a); 是错误的。作业中对数组元素的引用都是以变量下标的形式给出的，没有用一个数组名来表示对数组所有元素进行访问的情况。

（7）写语句printf一次最多打印一个字符串加一个表达式,不允许多个字符串的打印也不允许多个表达式的打印,"printf()"也不允许,但允许"printf("")"

（8）字符串中不支持双引号出现(因为不支持34号字符) 例如: char a[3] = "iu""; 是错误的。

### 【16】

### ＜情况语句＞ ::= switch '('＜表达式＞')' '{'＜情况表＞＜缺省＞ '}'

### ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

### ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

### ＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

### ＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

### ＜情况子语句＞ ::= case＜常量＞：＜语句＞

### ＜常量＞ ::= ＜整数＞|＜字符＞

### ＜缺省＞ ::= default : ＜语句＞|＜空＞

### 例：

swtich(a){

case 0:

B=1;

case 1:

B=2;

default:

B=3;

}

### 分析：

1. 情况语句由关键字switch(){}引导，（）中的表达式为待判断的量，由{}包含情况表，可以选择性的包含缺省情况default。
2. 情况表可以由一个或多个case语句组成。
3. 情况子语句开头是case 然后接常量和语句，二者用冒号隔开。
4. 常量必须是整数或者字符。
5. 缺省语句中，如果有default，后面语句可以为空，语句后跟;。或者直接没有default语句、

（6）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。

### 2．目标代码说明

### MipsOpKind[0] = "addu"; //add z, x, y: z = x + y

### MipsOpKind[1] = "sub"; //sub z, x, y: z = x - y

### MipsOpKind[2] = "mul"; //mul z, x, y: z = x \* y

### MipsOpKind[3] = "div"; //div z, x, y: z = x / y

### MipsOpKind[4] = "addi"; //addi z, x, offset: z = x + offset

### MipsOpKind[5] = "subi"; //subi z, x, offset: z = x - offset

### MipsOpKind[6] = "mul"; //mul z, x, offset: z = x \* offset

### MipsOpKind[7] = "div"; //div z, x, offset: z = x / offset

### MipsOpKind[8] = "lw"; //lw z, x, offset: lw z, offset(x)

### MipsOpKind[9] = "sw"; //sw z, x, offset: sw z, offset(x)

### //有条件跳转指令打印时 打印 op, x, y offset, z

### MipsOpKind[10] = "beq"; //beq z, x, y/offset: beq x, y/offset, z z为label

### MipsOpKind[11] = "bne"; //bne z, x, y/offset: bne x, y/offset, z z为label

### MipsOpKind[12] = "bge"; //bge z, x, y/offset bge x, y/offset, z x >=y 时转移到z

### MipsOpKind[13] = "bgt"; //bgt z, x, y/offset bgt x, y/offset, z x > y 时转移到z

### MipsOpKind[14] = "ble"; //ble z, x, y/offset ble x, y/offset, z x <=y 时转移到z

### MipsOpKind[15] = "blt"; //blt z, x, y/offset blt x, y/offset, z x < y 时转移到z

### MipsOpKind[16] = "j"; //j z 无条件跳转

### MipsOpKind[17] = "jr"; //jr z 跳转到寄存器z

### MipsOpKind[18] = "jal"; //jal z 跳转并链接, z为标签名

### MipsOpKind[19] = "li"; //li z, offset

### MipsOpKind[20] = "move"; //move z, x z = x

### MipsOpKind[21] = "syscall"; //syscall 系统调用

### MipsOpKind[22] = "label"; //一个标签 z

### MipsOpKind[23] = "la"; // la z, x 将标签x的地址赋给z

### 优化方案

·寄存器的分配：

1. 全局寄存器的分配：引用计数法
2. 临时寄存器的分配：临时寄存器池的管理

· 局部优化：

DAG图——局部公共子表达式的删除

### 二．详细设计

### 1．程序结构

### 

### 类/方法/函数功能

*斜体*为可供外部文件调用函数, 正常体为内置函数

（1）Var\_Info.cpp

定义枚举类型、结构体、宏定义常量和全局变量。

· 枚举变量symbol和Symbol\_info数组——列举词法分析中得到的各种词的枚举类型，便于使用。

· objecttyp 和 objecttyp\_info数组——constant, variable, function, array等标识符的种类。

· 枚举变量types和types\_info数组——ints, chars, notyp等数据类型。

· 各种表格tabe，tabt，atabe，atabt，btabe，btabt，midcode，midcodetable，mipscode，mipscodetable，stabt，MidCodeT、MidCodeOptT、MipsTable、MipsTableOpt

· 数据栈stacke

· 寄存器map中用于记录变量信息的结构体var\_info。

· 各种输入输出文件。

· 栈

· display display[0]值为0,btab.btabArray[display[0]].last为全局变量最后一个标识符位置。display[1]存储当前局部作用域函数在btab中位置。

· 全局变量：

extern char ch; //last character read from source program

extern int inum; //intvalue from insymbol

extern int sleng; //string length

extern int cc; //character counter

extern int lc; //program line location counter

extern int ll; //length of current line

extern char line[LLNG + 1];//input line, last char is ' '

extern int errpos; //error positon

extern symbol sy; //last symbol read by insymbol

extern char id[NAME\_SIZE]; //indentifier from insymbol

extern int level; //当前分程序层,只存在0和1两层

extern bool IsError;

· 中间代码四元式指令对照表MidOpKind[MID\_KIND]、mips指令对照表MipsOpKind[MIPS\_KIND]

· 存储临时变量的信息的std::map<std::string, var\_info> T\_Map等

**（2） I**nit\_Deal.cpp ——用于进行一系列初始化操作

*· init() : 整体调用函数*

· Init\_Var() :对全局变量赋初值，并归零一些指针变量

· Set\_Up() : 通过初始化建立一系列对应关系，便于输出中间代码和代码。初始化关键字数组和关键字符数组。初始化中间代码指令符号数组、目标代码指令数组。

· Open\_File() : 打开txt源代码文件以便读取、打开中间代码输出文件、打开目标代码输出文件、报错信息输出文件。

**（3）W**ord\_Deal.cpp —— 词法分析文件

*· return\_symbol\_info(symbol sy)：由输入的参数sy对照开始声明的枚举变量，得到sy对应的类型名*

· nextch() : 由insymbol调用，读入下一个字符存到ch。应注意，第一次读到文件末尾时不会结束程序，而如果第二次读到收到文件末尾读的请求则证明有错误正在跳读，为了能正常跳出循环则报错并终止程序。

*· insymbol() : 供block调用。每次*读入下一个单词，将单词类型赋予全局变量sy，如果读到为标识符将名称赋予全局变量id，如果读到数字则将其计算出来将结果赋予全局变量inum，如果读到字符串则将其压入字符串表，如果读到符号设置sy，读到非法字符报错。

（4）blockdeal.cpp —— 语法语义分析文件

*· block() : 总函数，一次读取源文件内容并调用相关函数进行分析处理。*

· Skip\_Read(symbol st[], int n): 跳读源程序直到取到的字符属于给定的符号集st，n为数组大小

· Creat\_Labell(std::string s, int l\_kind, int pc): 生成标签字符串并返回。

0：函数类起始标签 1：函数类结束标签 2：if、while等跳转 3：switch\_end标签

4：call函数之后返回地址标签,用于标记入口语句

· Creat\_Tempvar(): 生成临时变量并将其登记在T\_Map中

· Sy\_Trans\_Types(symbol sym):由传入的sym参数对照得到type

· Deal\_Const(): 处理常量的声明和定义部分

· Deal\_Varible(types typ, int &n): 处理变量的声明和定义部分

· Deal\_Factor(bool& is\_char): 处理因子

· Deal\_Item(int minus\_flag, bool& is\_char): 处理项，嵌套Deal\_Factor

· Deal\_Expression(bool& is\_char):处理表达式，嵌套Deal\_Item

· Deal\_Para(int& para\_num, int& pv\_addr ):处理参数，返回值为登记后最后一个参数tab中的位置

· Deal\_Condition(int kind): 跳转指令跳转条件的处理。返回该跳转指令所在的中间代码中的位置,以供后续回填label域。

· Deal\_ifStatement():分析处理 if条件语句

· Deal\_whileStatement: 分析处理while语句

· Deal\_reUseFunc(): 处理有返回值函数调用语句

· Deal\_voUseFunc(): 处理无返回值函数调用语句

· Deal\_assignStatement(int array\_flag): 处理赋值语句

· Deal\_scanfStatement():处理读语句

· Deal\_printfStatement(): 处理写语句

· Deal\_caseTable(std::string x, std::string switch\_end\_label, bool is\_char): 处理情况表

· Deal\_switchStatement(): 处理switch语句

· Deal\_returnStatement(): 处理返回语句

· statement(): 处理各种语句，通过分析开头调用不同的分析函数

· statementCompond(int& var\_size, types& return\_typ, int& pv\_addr): 处理复合语句，返回值为函数最后一个局部变量在tab中的位置

· Deal\_reFunc(types typ): 处理有返回值函数定义

· Deal\_voFunc(types typ): 处理无返回值函数定义

· Deal\_Main(): 处理主运行函数

（5）Enter\_Info.cpp ——登记信息函数文件

*· enter(std::string name, objecttyp obj, types typ):* 遇到标识符进行登记

*· enterVar(int pos, int adr):*登记变量/常量信息

*· enterArray(int t\_pos, types typ, int high) :*登记数组信息

*· enterFunc(int t\_pos, types typ) :* 登记函数信息

· Enter\_Mid\_List(int op, std::string z, std::string x, std::string y, int isstart)：isstart > 0 都要为入口地址， 其中isstart = 2 代表 mian函数入口标签, isstart = 3代表函数调用返回地址, isstart = 4表示非main函数开始位置

· Enter\_Mid\_Opt\_List(int op, std::string z, std::string x, std::string y, int isstart)

· Enter\_Mips\_List(int op, std::string z, std::string x, std::string y, int offset)

（6）Func\_Info.cpp —— 一些功能函数

*· int\_to\_string(int n):* int转换为string

*· string\_to\_int(std::string s):* string转换为int

*· find\_sy(symbol st[], int n):* 查找当前sy是否在st[]

*· string\_is\_num(std::string s):* 判断当前字符串是否为整数

· *find\_in\_tab(std::string name):* 查找标识符在符号表中的位置并返回

（7）Error\_Deal.cpp ——错误处理文件，定义error\_set和fatal\_set，采用数组结构建立对应关系。

*· error(int n, int pos) : 一旦被调用将IsError改为true，输出*错误信息和错误行数定位，同时输出到文件和控制台

*· fatal(int n, int pos) : 各种*溢出报错，一旦被触发强制退出。将错误信息打印到控制台和文件。

*（8）Optimized\_Mid\_Code——dag图优化文件*

*· Make\_Opt(): 调用dag图优化的接口函数，负责切分函数块并安排整个优化过程。*

· In\_Node\_Map\_Or\_Not(tree\_node\* t): 检查当前节点是否在节点信息表中

· Delete\_From\_NodeMap(tree\_node\* t): 将节点t从节点表中删除

· get\_correct\_node():从Dag图中选取一个没有父节点或父节点都已经被取出的合法中间节点并返回

· findInNodeMap(std::string name): tree\_node\* 在node\_map表中寻找当前变量名是 否已经存在, 若存在返回其节点指针

· In\_Replace\_Map\_ornot(std::string name):检查当前变量名是否在新旧变量名更新表中

· expEqual(int op, tree\_node\* lchild, tree\_node\* rchild): 判断当前处理的表达式是否已经有等价的节点, 若有则返回其节点指针

· dealChild(std::string name): 处理操作数的变量名, 若已经存在则返回其节点指针, 若不存在则生成一个叶子结点

· dealFather(std::string name, int op, tree\_node\* lchild, tree\_node\* rchild, bool is\_assign): 处理表达式结果的变量名, 增加新的叶子节点或者对应已有的叶子节点

· getdag\_queue(tree\_node\* root): void 递归调用得到以root为根节点的子树，将节点插入中间节点队列中

· dealNodeVarnames(): 处理每个中间节点所对应的变量名，找出被重新定义的 局部变量，若被重新定义的局部变量个数大于被优化删除的临时变量名个数，则返回false，不进行优化处理(新增的a = a0条数大于优化的指令条数)，否则返回true。

· get\_Opt\_Code(): 将中间节点从队列中逆序导出生成中间代码，进行了优化返回true，没进行优化返回false

· Deal\_Basicblock(int mid\_exp\_start, int mid\_exp\_end):对中间代码基本块进行优化, 优化范围[mid\_exp\_start, mid\_exp\_end)

· Deal\_Func(int mid\_func\_start, int mid\_func\_end): void 以函数体为单位进行表达式优化, 优化范围[mid\_func\_start, mid\_func\_end)

（9）Mid\_to\_Mips ——中间代码转换成目标代码

· *midtomips():*外部调用函数，中间代码转目标代码的主要函数

*· Get\_Label\_For\_Str*(std::string str): 生成string标签并返回，将对应信息放到str\_info\_map中

· id\_Trans\_Regnum(int num): 根据分派到的寄存器号得到具体Mips指令中寄存器的表达方式

· pushReg(int reg\_pos): 将当前寄存器的值保存进运行栈

· clearRegs(int kind):几种清空寄存器的策略，kind = 0全部清空不存入内存， kind = 1全部清空并存入内存，kind = 2全部清空，全局变量存入内存，其余不存入

· findInMap(std::string name, int kind, var\_info& vinfo):对于出现的变量在Map表中寻找 kind = 0在T\_Map中找，Kind = 1在S\_Map中找，找到返回true并修改vinfo的值，未找到返回false

· allocR\_REG(std::string name, int offaddr, int regkind):申请一个r\_reg寄存器，必要时把该寄存器存储的值存入内存中，返回R\_Regs下标

· deleteBusy(): 删除寄存器的繁忙标志位

· varCount(std::string name): 对变量的使用个数进行计数

· cmp\_by\_value(const PAIR& lhs, const PAIR& rhs): sort函数使用的比较函数

· loadScountMap(int mid\_pos):根据引用计数分配全局寄存器

· funcAllocRegs(int func\_start): 寄存器分配

· dealOperand(std::string s, std::string& reg\_name, int&num): 处理z，y，x 返回值为1说明处理的为变量，相应的寄存器名为reg\_name， 返回值为2说明处理的为临时数字，相应的值为num，返回值为0说明出错

· returnReg(int regkind)：归还寄存器

· calcuDeal(int mid\_pos): +、-、\*、/ 的处理, 传入参数为第几条指令

· assignDeal(int mid\_pos):赋值指令的处理

· noConJumpDeal(int mid\_pos): 无条件跳转指令的处理

· conditionJumpDeal(int mid\_pos):条件跳转指令的处理

· setlabDeal(int mid\_pos):对标签的处理

· funcDeal(int mid\_pos):函数声明的处理

· inDeal(int mid\_pos): 输入的处理

· outDeal(int mid\_pos): 输出的处理

· passparaDeal(int mid\_pos):传递参数的处理

· callDeal(int mid\_pos):函数调用的处理

### 3.符号表管理方案

**(1) 符号表tab**

struct tabe{

std::string name;//标识符名字,取前十个字符

objecttyp obj; //标识符种类 {constant, variable, function, array}

types type; //标识符的数据类型 {ints, chars, notyp}

int adr;

uint32\_t ref; //指向数组在atab中的位置或者函数名在btab中的位置

uint32\_t link;//指向同一个分程序中上一个标识符在符号表中的位置

int lev; //变量所在分程序层

};

其中adr: 对于变量(形参)名, 填入该变量(或形参)在运行栈S中分配的存储单元的相对地址；对于函数名,填入相应中间代码的入口地址; 对于整数和字符常量名，填入其具体的值;

struct tabt{

struct tabe tabArray[TMAX]; //TMAX = 1000, 符号表最大容量为1000

uint32\_t t; //索引变量t

};

1. **数组表array**

struct atabe{

types eltyp; //数组元素类型,只能为int或者char

uint32\_t high; //数组元素的上界,数组元素从0开始(int a[10],上界为10, 数

组下标小于10),同时也是数组元素的大小

};

struct atabt{

struct atabe atabArray[AMAX];

uint32\_t a; //索引变量a

};

1. **分程序表btab**

struct btabe{

uint32\_t last; //指向该分程序中说明的当前(最后)一个标识符在tab表中的位置

uint32\_t lastpar; //函数的最后一个变量在tab中的位置

types kind; // notyp代表无返回值, ints返回值为int, chars代表返回值为

//char

uint32\_t paranum; //参数个数

uint32\_t varsize; //局部(全局)变量所占栈空间大小

int t\_varnum; //临时变量的个数

};

struct btabt{

struct btabe btabArray[BMAX];

uint32\_t b; //索引变量b

};

1. **分程序索引表display** (由于不存在分程序嵌套所以层数只有0,1两层)

uint32\_t display[2]; //作为索引表的索引,display[0]值为0

btab.btabArray[display[0]].last为全局变量最后一个标识符位置，display[1]存储当前局部作用域函数在btab中位置, 即btab.btabArray[display[1]]找到当前函数作用域

所以在查符号表时先从btab.btabArray[display[1]].last开始逐步link前找，找不到之 后再从btab.btabArray[display[0]].last开始向前(其实也就是btab.btabArray[0].last)

### 4.存储分配方案

**(1) 运行时的存储组织及方案**

· 对于局部变量(常量)、形参、全局变量(常量)利用tab表查找找到其在数据栈中的相对当前数据区的基地址偏移

· 临时变量建立T\_map表进行存储，根据临时变量的名字查找其所在的数据栈地址偏移

1. **运行栈结构:**

|  |
| --- |
| 临时变量 |
| 局部常量/变量 |
| 参数区 |
| Returnvalue |
| prev abp |
| ret addr |
| 临时变量 |
| 局部变量 |
| Returnvalue |
| Prev\_abp |
| Ret\_addr(main函数) |
| 全局var |

### 5. 四元式设计

**5.1 存储结构**

struct midcode{

int op; //四元式操作符,配合MidOpKind表来使用

std::string z;

std::string x;

std::string y;

int isstart; //标志该四元式中间代码是否为入口语句

}

RET; //函数返回值

+ z, x, y //加法指令 addu/addi 不支持两个数字相加赋予z

- z, x, y //减法指令 sub

\* z, x, y //乘法指令 mul

/ z, x, y //除法指令 div

= z, (x), y //赋值指令 mov z = y or z = y[x] or z[x] = y 具体区分根据编号不同

goto z //无条件跳转到label/PC值 (midcode数组下标索引) PC = z

beq z, x, y //x == y, goto z(z此时为label名称)

bne z, x, y // x!=y, goto z

bge z, x, y // x>=y, goto z

bgt z, x, y //x>y, goto z

ble z, x, y //x<=y, goto z

blt z, x, y //x<y, goto z

ret y //函数返回操作, RET = y

jal z //跳转并链接, 紧跟call指令使用

setlab y //设置标签名字为y

func x, y //函数声明，y为函数名, x为类型 int, char, void

para x , y //函数定义时的形参，x为类型，y为形参名

in y //调用scanf，将值放到y

out x, y //输出字符串x, 表达式的值y

para y //传递参数操作

call y //函数调用操作, y为函数名称

### 6.目标代码生成方案

6. 1 存储结构

struct mipscode {

int op; //操作符

std::string z; //label名或者寄存器

std::string x;

std::string y;

int offset; //在需要地址偏移或者使用立即数时采用

};

struct mipscodetable {

struct mipscode mipscodeArray[MIPSCODEMAX];

int mpc; //索引变量

};

struct reg {

std::string name; //该存储的临时变量/局部变量的名字

int regKind; //0-临时数字, 1-临时变量, 2-局部变量(包括数组, //map里面用array[1]表示其名字), 3-全局变量

int offAddr; //若为临时变量(如果之前在栈中存在的话,不存在则设 //置为-1)、局部变量或者全局变量的话其相对各自数据 //区的地址偏移, sw的时候用

int isEmpty; //标志当前寄存器是否可用

int isBusy; //之后根据冲突设置当前指令是否使用

};

struct r\_reg {

reg Regs[R\_REG\_NUM]; ////临时寄存器 $8-$25

int ptr; //指向当前空闲位置或者当前该被替换的寄存器(FIFO)

bool isfull; //表示是否满了

};

struct var\_info {

bool inReg; //是否在寄存器中, 1表示是, 0表示不是

int r\_addr; //寄存器的数组下标 0 - 17

int stack\_addr; //数据栈中相对基地址偏移

bool isTFirst; //临时变量才会用到, 是否为第一次使用

};

std::map<std::string, var\_info> T\_Map; 临时变量

std::map<std::string, var\_info> S\_Map; 局部变量

std::map<std::string, int> S\_Count\_Map;：引用计数阶段使用的局部变量计数

**6.2 处理方案**

**· 处理操作数**

数字——返回，暂时不给其分配寄存器。

临时变量——从T\_Map表中查找其所在位置，若在寄存器中则返回相应的寄 存器编号，若在内存中则从内存中取出并为其分配一个寄存器并将寄存器号返回。

局部变量——从S\_Map中查找其状态，若查不到说明是第一次引用，则通过查tab表得到其在内存中的地址并将信息加入S\_Map中，然后返回分配的寄 存器编号。

**· 寄存器清空策略**

kind = 0， 清空 $8-$25寄存器, 不存入内存; 函数声明开始时使用

kind = 1, 清空 $8-$25寄存器, 将相应内容存入内存中; 调用其他函数时在jal之前使用

kind = 2, 清空 $8-$25寄存器, 将全局变量存入内存,其余不存入 函数返回时使用

kind = 3, 清空 临时变量寄存器，保留全局寄存器(func一开始分配的保留, 函数内跨块使用)

· **标签处理**

Kind =1 普通的跳转标签：先使用上述3策略清空寄存器，再设置标签。

Kind =2 main函数入口标签：设置$fp(函数基址)和$sp(栈顶)的值，分配全局寄存器。

Kind =3 函数返回地址标签：恢复栈顶空间$sp，恢复上层函数基址$fp，更新$ra。

Kind =4 函数开始标签：打印标签，分配全局寄存器。

**· 函数声明处理**

设置当前display索引层(变量查tab表使用)，使用kind=1策略清空寄存器并清空S\_Map表。

**· 函数调用**

保存当前$ra，设置当前被调用函数的prev\_abp，移动栈顶指针，并将当前函数标签保存在call栈(多次调用时使用)中。

**· 函数返回处理**

处理返回值，使用Kind=3策略清空寄存器，Kind=3方式对环境进行恢复。

### 7. 优化方案

· DAG图——局部公共子表达式的拆除

以函数为单位进入优化函数，并以除+-\*/之外的指令为分割划分基本块。建立表达式树生成DAG图，再使用启发式算法进行导出。

a.变量名的更新：由于公共子表达式的删除会使删除程序中一些冗余的变量，为了防止表达式块后的代码对删除掉的变量进行引用导致出错，以函数块为单位建立更名对照表，然后在一个函数块优化完成之后再统一更新变量名。

b.局部变量重定义：利用F\_S\_Map。如果fcount大于tcount，则不对该代码块进行优化，否则则无法完成@变量的替换。

· 寄存器分配

使用$8-$25共18个寄存器，其中全局和局部变量只能参与分配s型寄存器，临时变量只能参与分配t型寄存器。

**s型——引用计数**

扫描函数体内出现的局部变量，统计使用次数记录在S\_Count\_Map。小于9个直接分配，否则为使用频率最高的前十个变量分配全局寄存器。

**t型——临时寄存器池先进先出**

### 8. 出错处理

error报错。fatal将退出整个程序。

1. **error:**

std::string error\_set[55] = {

"文件读完了但文件尾有错误,编译结束", //0

"整数太大了", //1

"字符不能为空", //2

"字符结尾应是\'", //3

"字符串中出现非法字符,该字符已跳过", //4

"字符串未读到双引号就进行了换行操作,该字符串不被记录", //5

"!后面必须有=", //6

"程序中出现特殊字符", //7

"字符定义出现非法字符(违反字符文法)", //8

"非零整数以0开头,已进行容错,例如0123取123", //9

"main函数声明错误", //10

"应该有标识符", //11

"函数返回值错误(返回类型错误或者无返回值)", //12

"缺少\'(\'", //13

"缺少\')\'", //14

"缺少\'[\'", //15

"缺少\']\'", //16

"缺少\'{\'", //17

"缺少\'}\'", //18

"缺少\';\'", //19

"缺少\'=\'", //20

"if、while后面必须跟条件语句", //21

"case后面必须为常量", //22

"标识符(变量名 or 常量名 or 函数名)未定义", //23

"标识符(变量名 or 常量名 or 函数名)重复定义", //24

"数组越界", //25

"应该为关键字int 或 char", //26

"常量的赋值与声明的类型不符", //27

"变量声明格式错误", //28

"const声明应该在最前", //29

"数组声明格式错误", //30

"函数声明格式出错", //31

"常量声明、变量声明或者语句 首单词出错",//32

"表达式格式错误", //33

"应该为有返回值函数调用", //34

"缺少while", //35

"形参个数与实参个数不符", //36

"不能对常量赋值", //37

"赋值语句格式错误", //38

"scanf语句格式错误", //39

"printf语句格式错误", //40

"return语句格式错误", //41

"switch语句错误", //42

"case语句格式错误", //43

"除数不能为0", //44

"形参与实参类型不符", //45

"赋值语句左右类型不一致", //46

"case 常量与switch中表达式类型不符", //47

"函数定义内部存在错误", //48

"数组下标错误(越界或非法表达)", //49

"条件格式错误", //50

"比较运算符左右类型不一致",//51

"case重复",//52

"条件中的表达式不是int型"//53

};

1. **fatal:**

std::string fatal\_set[10] = {

"程序中字符串太多了,导致字符串表溢出", //0

"符号表tab溢出", //1

"分程序表btab溢出", //2

"数组表atab溢出", //3

"中间代码表溢出", //4

"mips代码表溢出", //5

"中间代码优化表溢出" //6

};

### 9.感想与建议

作为一个非“大神”，在设计的过程中有一个可以一起探讨的同学非常重要，尤其是在遇到一些感觉非常棘手不知道该怎么修改的细节问题上。同时，应该适当参考一些优秀经典的架构和算法思想，我一开始把所有的内容写在了一个文件中，设计的架构漏洞百出，后来仔细研究经典结构后了解到了全局变量和结构、枚举的妙用，可以说极大的扩展了视野，提高了代码能力。在设计的过程中，切忌没有脑子里架构就开始打代码，因为编译器的工程量很大，很难一蹴而就，一个长期的工程必须有一个可靠的设计架构，防止大规模返工，浪费了时间和精力。注释非常重要，写好注释可以提高效率。而且要注意使用github进行多版本存储。遇到BUG不要着急改动，应该注重使用“回溯法”一步步输出找到错误的根源，改动之前也要想好会不会引起其他方面的BUG。最后，一个完备的测试程序真的非常非常非常重要！！

如果说有什么建议的话，希望课程组能在学生没有完成基本架构之前多费心把一些细节上的要求提前说清楚，避免架构已经写好，测试也测过好多次，突然因为一些细节原因做出了修改，之前的测试工作也白费了的情况，也避免不集中的发布要求，导致信息错乱。感谢课程组的辛勤付出！