《编译技术》课程设计文 档

学号： 16231181

姓名： 郭容辰

2018年 11 月 15 日

目录

[一．需求说明 3](#_Toc533967745)

[1．文法说明 3](#_Toc533967746)

[2．目标代码说明 9](#_Toc533967747)

[3. 优化方案\* 9](#_Toc533967748)

[二．详细设计 9](#_Toc533967749)

[1．程序结构 9](#_Toc533967750)

[2．类/方法/函数功能 10](#_Toc533967751)

[3．调用依赖关系 13](#_Toc533967752)

[4．符号表管理方案 16](#_Toc533967753)

[5．存储分配方案 17](#_Toc533967754)

[6. 四元式设计\* 18](#_Toc533967755)

[7. 目标代码生成方案\* 19](#_Toc533967756)

[8. 优化方案\* 20](#_Toc533967757)

[9. 出错处理 21](#_Toc533967758)

[三．操作说明 22](#_Toc533967759)

[1．运行环境 22](#_Toc533967760)

[2．操作步骤 22](#_Toc533967761)

[四．测试报告 22](#_Toc533967762)

[1．测试程序及测试结果 22](#_Toc533967763)

[2．测试结果分析 35](#_Toc533967764)

[五．总结感想 36](#_Toc533967765)

## 一．需求说明

### 1．文法说明

#### （1）扩充C0文法：

* + - 1. ＜加法运算符＞ ::= +｜-
      2. ＜乘法运算符＞ ::= \*｜/
      3. ＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==
      4. ＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z
      5. ＜数字＞ ::= ０｜１｜．．．｜９
      6. ＜字符＞ ::= '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'
      7. ＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"
      8. ＜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＞
      9. ＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}
      10. ＜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＞}
      11. ＜无符号整数＞ ::= ＜数字＞｛＜数字＞｝
      12. ＜整数＞ ::= ［＋｜－］＜无符号整数＞
      13. ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝
      14. ＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞
      15. ＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}
      16. ＜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＞‘]’ )}
      17. ＜类型标识符＞::= int | char
      18. ＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’|＜声明头部＞‘{’＜复合语句＞‘}’
      19. ＜无返回值函数定义＞ ::= void＜标识符＞(’＜参数表＞‘)’‘{’＜复合语句＞‘}’| void＜标识符＞{’＜复合语句＞‘}’
      20. ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞
      21. ＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}
      22. ＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’
      23. ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}
      24. ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}
      25. ＜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＞
      26. ＜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＞;|＜返回语句＞;
      27. ＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞
      28. ＜条件语句＞::= if ‘(’＜条件＞‘)’＜语句＞[else＜语句＞]
      29. ＜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
      30. ＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞ |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞
      31. ＜步长＞::= ＜无符号整数＞
      32. ＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符>
      33. ＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符>
      34. ＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}
      35. ＜语句列＞ ::= ｛＜语句＞｝
      36. ＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’
      37. ＜写语句＞ ::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’| printf ‘(’＜字符串＞ ‘)’| printf ‘(’＜表达式＞‘)’
      38. ＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

#### （2）文法语句分析：

＜加法运算符＞ ::= +｜-

作用：定义了加法和减法的运算符号

限定条件：用于表达式计算

例：3+4, 6-7, a+b, c-d-e

＜乘法运算符＞ ::= \*｜/

作用：定义了乘法和除法的运算符号

限定条件：用于表达式的计算

例：3\*4，6/8，a\*b，c/d\*e

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

作用：定义关系比较运算符小于、小于等于、大于、大于等于、不等于、等于

限定条件：用于关系比较运算,产生条件判断的依据bool值

例：a>8, 5-4==1, 3\*4!=10

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

作用：定义了字母的集合，包括大写字母和小写字母和下划线

限定条件：用于标识符命名，字符串表示，字符常量表示等

例： a, A, Z, \_

＜数字＞ ::= ０｜１｜．．．｜９

作用：定义了阿拉伯数字的集合

限定条件：用于数字串的组合表示，也可表示逻辑判断值

例：100，99，0，622

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

作用：定义所有单字符

限定条件：所有单字符只能由此文法产生

例：‘+’, ‘\*’, ‘a’, ‘Z’, ‘9’

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

作用：定义字符串所能包含的所有字符

限定条件：字符串中的只能含有以上字符，否则非法

例：”ABC@D”, “a4766%BF”

＜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＞

作用：定义了整个扩充文法支持的程序代码的框架

限定条件：程序从前往后的组成为：常量说明（此处为全局常量）、变量说明（此处为全局变量）、函数的定义（可有可无）、主函数main

例：const int a1 = 33, a2 = 22;

char char1, char2;

int sum( int a, int b){

return a + b;

}

void main(){

c = a1 + a2;

}

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

作用：定义了常量的组成成分

限定条件：常量说明需以const字符开头，以分号结尾。常量说明中可以包含很多个const字符。

例：const int a1 = 33, a2 = 22;

const char char1 = ‘a’, char2 = ‘b’;

＜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＞}

作用：定义了常量说明中常量定义部分的具体组成成分

限定条件：常量定义包括int型常量和char型常量

例：int a1 = 33, a2 = 22;

char char1 = ‘a’, char2 = ‘b’;

int a3 = 66;

＜无符号整数＞ ::= ＜数字＞｛＜数字＞｝

作用：定义了无符号整数，由1至多位0~9的数字组成

限定条件：无符号整数包括前导零

例：007， 100， 9999， 0， 6

＜整数＞ ::= ［＋｜－］＜无符号整数＞

作用：定义了所有整数

限定条件：即在无符号整数的基础上加上正负号或不加符号，包含前导零

例：0, 007, +100, -080, 099, -140

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

作用：定义了标识符的组成成分

限定条件：标识符必须以字母或下划线开头，由字母和数字组成，标识符不区分大小写字母

例：Beijing, August, \_grant, \_love20

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

作用：定义了声明语句中声明头部的组成成分，用于有返回值的函数定义

限定条件：有返回值的函数的返回值只能是int整型或char字符型。char类型的变量或常量，用字符的ASCII码对应的整数参加运算

例：int sum, char strcat

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

作用：定义了变量说明的形式，可以包含若干个变量定义，并以分号分开

限定条件：用于全局变量声明和复合语句中的变量声明

例：int a1, a2; char a, b, c;

＜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＞‘]’ )}

作用：定义了变量说明中变量定义的组成部分

限定条件：定义必须以类型标识符开头，可以定义多个整型或字符型标识符，以逗号分隔开。数组的下标从0开始

例：int sum, counter, array[5]

＜类型标识符＞ ::= int | char

作用：定义了程序所支持的标识符类型

限定条件：程序只支持整型和字符型

例：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＞‘{’＜复合语句＞‘}’

作用：定义了有返回值的函数的组成成分

限定条件：有参数时参数用小括号括起，注意无参数时不可写小括号（因为参数表非空）！函数体内部用大括号括起

例：int sum( int a, int b){

return a + b;

}

int play{

return 1;

}

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

作用：定义了无返回值的函数的组成部分

限定条件：必须以void开头，有参数时参数用小括号括起，注意无参数时不可写小括号（因为参数表非空）！函数体内部用大括号括起

例：void play { return;}

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

作用：定义了复合语句的组成结构，是函数程序体内部的所有语句集合

限定条件：适用于函数体，函数体外不可用

例：const int total = 32;

int a1, a2;

int a3 = 10;

a2 = a + a1 + 10;

return a2;

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

作用：定义了参数表的组成成分

限定条件：参数表不可为空，参数表包含若干类型标识符加标识符，中间用逗号分开

例：int a, int b, char c

＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

作用：定义了主函数的组成成分

限定条件：必须以void main ()开头，后接复合语句并用大括号括起

例：void main () {

int a=2, b = 3;

int c;

c = a + b + 5;

return;

}

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

作用：定义了表达式的组成成分

限定条件：只有第一项之前可以有加号或减号，后面的项之前必须是加号

例：3 + a + b, -7 + 8 + 9

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

作用：定义了项的组成成分

限定条件：是表达式的组成成分，由因子和若干\*因子组成

例：a, a\*5

＜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＞

作用：定义了因子的组成部分

限定条件：因子可以是变量、数组中的某个元素、某个括号括起的表达式、某个整数、某个字符、亦或是有返回值函数的调用语句

例：array[6], (3+4\*5), ‘c’, A

＜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＞;|＜返回语句＞;

作用：定义了程序中可能含有的语句种类

限定条件：用于函数体内部

例：if( a + b == c) return 1; else return 0;

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

作用：定义了赋值语句的组成成分

限定条件：必须包含等号的左部和右部

例：array[3+4\*5] = 6, a = b[2]

＜条件语句＞::= if ‘(’＜条件＞‘)’＜语句＞[else＜语句＞]

作用：定义了条件转移语句的组成成分

限定条件：允许没有else语句

例：if ( a > 5) return;

if ( a == 4 ) a = a + 1; else if( a == 5) a = a+2; else a = a+3;

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞

作用：定义了条件语句中的判断条件的具体组成成分

限定条件：条件最终得到一个整数结果，结果为0条件为假，否则为真

例：1， a==5, b>6

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞ |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞

作用：定义了循环语句的组成结构

限定条件：包括for循环的while循环，for语句先执行一次循环体中的语句再进行循环变量是否越界的测试

例：while ( a < 5) a = a+1;

for (i=1; i<5; i=i+1) a[i] = b[i];

＜步长＞::= ＜无符号整数＞

作用：定义了for循环体中的步长

限定条件：在实际实现中可以用无符号整数替代步长

例：008， 567， 1， 0

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符>

作用：定义有返回值的函数调用的具体组成成分

限定条件：对于无参数的函数调用需要注意不能加小括号

例：sum (a, b)

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符>

作用：定义无返回值的函数调用的具体组成成分

限定条件：对于无参数的函数调用需要注意不能加小括号

例：play (a, b, c)

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

作用：定义了调用语句中值参数表的组成成分

限定条件：值参数表不能为空，包含至少一个表达式，多个表达式之间用逗号隔开

例：a, b, c, 666, y+1

＜语句列＞ ::= ｛＜语句＞｝

作用：定义了语句列的组成成分

限定条件：语句列中由若干语句组成，可以为空

例：{

c = a + b;

}

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

作用：定义读语句的组成成分

限定条件：以scanf关键字开头，读入若干标识符，以逗号隔开，并用小括号括起

例：scanf( a, b, c)

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

作用：定义写语句的组成成分

限定条件：以printf关键字开头，字符串原样输出，单个字符类型的变量或常量输出字符，其他表达式按整型输出

例：printf( “adjlfd”, 2)

printf( “Hello”)

printf( 2 )

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

作用：定义返回语句的组成成分

限定条件：必须以return 关键字开头，后面括号加表达式可有可无

例：return (c)

return (a + b +2)

return

#### （3）对文法的扩充：

扩充后的文法可以读入注释，即允许测试程序中包含注释。编译程序中，注释会自动过滤掉。

### 2．目标代码说明

目标代码为Mars可运行的mips指令集：

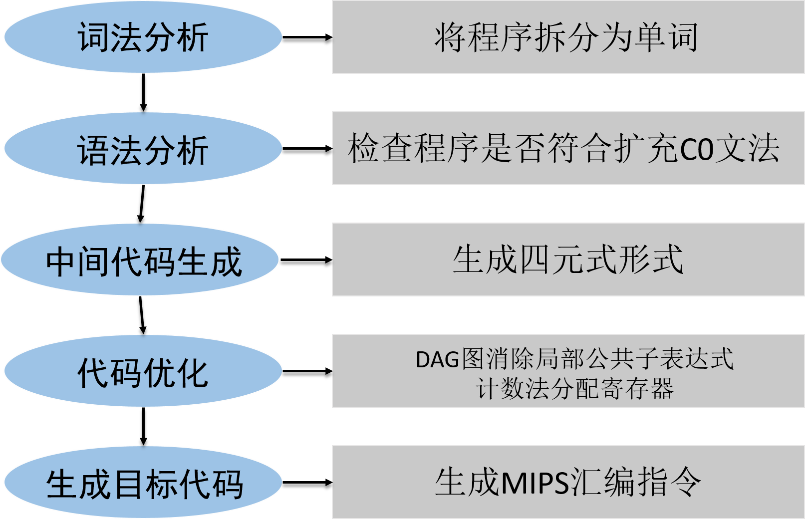
* add rd,rs,rt //加运算
* addi rt,rs,imm //立即数加法
* sub rd,rs,rt //减运算
* subi rd,rs,imm //立即数减法
* mul rd,rs, rt //乘运算
* div rd,rs, rt //除运算
* and rd, rs, rt //与运算
* bne rs, rt, offset //不等于时转移
* j //跳转指令
* lw rt, offset(base) //加载字
* sw rt, offset(base) //存储字

### 3. 优化方案\*

* 利用DAG图消除局部公共子表达式
* 引用计数法分配寄存器
* 活跃变量分析

## 二．详细设计

### 1．程序结构



### 2．类/方法/函数功能

#### （1）函数定义及功能：

// 错误处理

void err**(**int n**);** // 报第n号错误

void endjump**();** // 在跳过的部分画下划线

void j2statement**();** // 跳到下一条语句

void j2rparent**();** // 跳到右括号

void j2rbracket**();** // 跳到右方括号

void j2rbrace**();** // 跳到右大括号

void j2facnext**();** // 跳到下一个因子

void j2fundec**();** // 跳到函数声明

void j2dec**();** // 跳到变量声明

void j2condec**();** // 跳到常量声明

void j2sem**();** // 跳到分号

void j2lbrace**();** // 跳到左大括号

// 词法分析

void nextch**();** // 读取下一个字符

void nextsym**();** // 读取下一个除注释外的单词

void get\_a\_sym**();** // 读取下一个单词（含注释）

// 语法分析

void parameterlist**(**enum types type**);** // 参数表

void constdec**(**enum types type**);** // 常量说明（定义）

void variabledec**(**char temp**[],**enum types type**);** // 变量定义

void functiondec**(**enum types type**);** // 函数定义

void proceduredec**();** // 程序

void statement**();** // 语句

void returnstatement**();** // 返回语句

void printstatement**();** // 打印语句

void scanfstatement**();** // 读入语句

void linestatement**();** // 语句列

void ifstatement**();** // if语句

void forstatement**();** // for循环

void whilestatement**();** // while循环

void factor**();** // 因子

void term**();** // 项

void simpleexpression**();** // 表达式

void assignment**(**struct Tab tabi**);** // 赋值

void selector**(**struct Tab tabi**);** // 数组下标

void codition**();** // 条件

// 符号表管理

int lookup**(**char idl**[]);** // 查询变量是否在符号表中

void entering**();** // 填符号表

void variable\_entering**();** // 向符号表中填变量

void array\_entering**();** // 向符号表中填数组

void arr\_variable\_entering**();**// 向符号表中填数组或变量

int const\_entering**();** // 向符号表中填常量

void id\_entering**();** // 保留字填表

void ids\_entering**();** // 保留字填表

// 中间代码生成

void inmidcode**();** // 将一条中间代码加入中间midcodelist

char **\*** newlabel**();** // 生成一个新标签

char **\*** newvar**();** // 生成一个新变量

void printmidcode**();** // 中间代码输出到文件

// 目标代码生成

int lookup\_addr**();** // 查找变量在addrlist中的地址

int lookup\_func**();** // 查找函数在函数表ftab中的地址

void instack**();** // 变量入栈

void inaddr**();** // 变量填入addrlist

void midcodetoasm**();** // 中间代码生成目标代码

void callop**();** // 函数调用处理

void GOTOop**();** // 生成j 跳转语句

void BZop**();** // 不等于1时跳转

void labop**();** // 生成新标签

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**();** // 读入

void prtop**();** // 输出

void fpaop**();** // 函数调用时参数的值PUSH,,,c

void retop**();** // 返回语句

void paraop**();** // 参数

void funcop**();** // 函数

// 引用计数法分配寄存器

int lookup\_ref**();** // 查找变量是否分配了寄存器，返回寄存器编号

void ref\_entering**();** // 将变量的引用次数+1，若是未入表的临时变量，将其入表，引用次数置1

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 sorttable**(**int fun**);** // 对变量按其引用次数从高到低排序

// DAG图

void DAG**();** // 生成DAG图

int lookup\_nodelist**();** // 查询节点存放地址

int lookup\_nodetabname**();** // 按名称查询节点表

void lookup\_nodetabindex**();** // 按下标索引查询节点表

int can\_in**(**int x**);** // DAG图导出时判断节点是否可以进入队列

int all\_in**(**int nl**);** // DAG图导出时判断是否所有中间结点均已进入队列

#### （2）关键函数：

* nextsym()函数为词法分析的核心函数。此函数将源程序拆分为一个个单词，为语法分析做好准备。
* procedure()函数为语法分析的核心函数。此函数为规则<程序>的递归下降分析程序，方法同教科书所讲，一切其他语法成分均可通过procedure()到达。
* midcodetoasm()为中间代码生成目标代码的函数，此函数中除了对每一条四元式中间代码进行解析外，还对生成的目标代码保证了执行的顺序。将中间代码以函数为单位进行划分，先存储全局的常量和变量。之后每次进入函数调用时生成保存现场的代码，将$sp上移，然后继续向栈内堆叠此函数中的局部变量。函数调用返回时，遗弃局部变量并对保存的现场进行恢复。中间代码生成目标代码，每次遇到一个变量，就用lw加载指令，从对应地址加载数据到$t0或$t1寄存器进行计算，计算后，生成一条sw保存指令，结果向对应地址空间存储计算后的结果。
* ref\_count()是引用计数法分配寄存器的核心函数。以函数为单位对函数中每个变量进行引用计数，每个变量的引用次数存储在ftab中的ref\_tab[]域。ref\_tab又是一个结构体，ref\_tab[i]包含了函数中第i个变量的名称name[]以及其在函数中被引用的次数num。sorttable()函数会对函数中变量的引用次数从高到低排序。由于只有7个临时寄存器，取出引用次数最多的7个变量，分给它们临时寄存器。int lookup\_ref(int j,char name[])函数可以查找第j个函数中名为name的变量是否被分配了临时寄存器，若分配了寄存器，则返回临时寄存器编号，否则返回-1表示要从内存中存取数据。在进入函数的一开始，由于临时寄存器中还没有变量的值，需要添加7条lw指令，将可以分配临时寄存器的7个变量的值从内存中取出并赋给寄存器；在函数执行结束后，同样需要添加7条sw指令，将函数最终运算得的7个变量的值存回内存中的地址空间。
* DAG()是DAG图优化的核心函数。根据中间代码构建DAG图，并适时更新结点编号。导出DAG图采用启发式算法，只有某个结点的父结点全部已经导出后，此结点才可以被导出。之后按导出顺序的逆序计算结点的值。结点中如果是有用的变量则将其计算出来，如果是临时变量则不做计算。但如果某一结点全是临时变量，则取其中一个（第一个）进行计算并保留。若叶子结点中出现多个变量，则输出这多个变量互相相等。

### 3．调用依赖关系

|  |  |
| --- | --- |
| 函数名称 | 所依赖的其它函数 |
| 错误处理部分 | |
| err(n) | None |
| endjump() | None |
| j2statement() | nextsym(); endjump(); |
| j2rparent() | nextsym(); endjump(); |
| j2rbracket() | nextsym(); endjump(); |
| j2rbrace() | nextsym(); endjump(); |
| j2facnext() | nextsym(); endjump(); |
| j2fundec() | nextsym(); endjump(); |
| j2dec() | nextsym(); endjump(); |
| j2condec | nextsym(); endjump(); |
| j2sem() | nextsym(); endjump(); |
| j2lbrace() | nextsym(); endjump(); |
| 词法分析部分 | |
| nextch() | None |
| sytosym() | None |
| get\_a\_sym() | nextch(); sytosym(); err(); |
| nextsym() | get\_a\_sym(); |
| 语法分析 | |
| parameterlist() | nextsym(); entering(); inmidcode(); parameterlist(); err();  j2lbrace(); |
| constdec(type) | nextsym(); j2sem(); const\_entering(); testsemicolon(); |
| variabledec() | variable\_entering(); nextsym(); arr\_variable\_entering();  testsemicolon(); |
| functiondec() | entering(); inmidcode(); fun\_entering(); nextsym();  parameterlist(); err(); j2condec(); variabledec();  linestatement() |
| proceduredec() | constdec(); err(); j2fundec(); nextsym(); functiondec();  variabledec(); j2lbrace(); |
| statement() | lookup(); nextsym(); err(); j2sem(); j2rparent();  inmidcode(); assignment(); simpleexpression(); ifstatement();  forstatement(); whilestatement(); linestatement();  scanfstatement(); printfstatement(); returnstatement(); |
| returnstatement() | nextsym(); simpleexpression(); j2rparent(); inmidcode();  err(); j2sem(); |
| printfstatement() | nextsym(); inmidcode(); simpleexpression(); j2sem(); |
| scanfstatement() | nextsym(); inmidcode(); lookup(); err(); j2sem(); |
| linestatement() | statement(); err(); j2statement(); |
| ifstatement() | nextsym(); condition(); inmidcode(); statement(); err();  j2sem(); newlabel(); |
| forstatement() | nextsym(); inmidcode(); err(); condition(); lookup();  newlabel(); simpleexpression(); j2sem(); j2rparent();  statement(); |
| whilestatement() | nextsym(); inmidcode(); err(); condition(); statement();  newlabel(); |
| factor() | nextsym(); inmidcode(); selector(); err(); j2rparent();  simpleexpression(); j2sem(); |
| term() | factor(); nextsym(); inmidcode(); |
| simpleexpression() | nextsym(); inmidcode(); term(); |
| assignment() | selector(); simpleexpression(); nextsym(); err(); j2sem(); |
| selector() | nextsym(); simpleexpression(); j2rbracket(); err(); |
| codition() | simpleexpression(); j2rparent(); nextsym(); inmidcode(); |
| 符号表管理 | |
| lookup(idl[]) | None |
| entering() | None |
| variable\_entering() | entering(); inmidcode(); err(); j2sem(); |
| array\_entering() | err(); j2sem(); |
| arr\_variable\_entering() | entering(); array\_entering(); inmidcode(); err(); j2sem(); |
| const\_entering() | entering(); inmidcode(); err(); j2sem(); |
| id\_entering() | None |
| ids\_entering() | id\_entering(); |
| fun\_entering() | None |
| 中间代码生成 | |
| inmidcode() | None |
| \* newlabel() | None |
| \* newvar() | None |
| printmidcode() | None |
| 目标代码生成 | |
| lookup\_addr(name[]) | None |
| lookup\_func(name[]) | None |
| instack(var[]) | None |
| inaddr() | None |
| midcodetoasm() | lookup\_func(name[]); instack(var[]); inaddr(); |
| callop() | lookup\_ref(); lookup\_addr(); lookup\_func(); |
| GOTOop() | None |
| BZop() | None |
| labop() | None |
| plusop() | lookup\_ref(); lookup\_addr(); |
| minuop() | lookup\_ref(); lookup\_addr(); |
| mulop() | lookup\_ref(); lookup\_addr(); |
| divop() | lookup\_ref(); lookup\_addr(); |
| grtop() | lookup\_ref(); lookup\_addr(); |
| geqop() | lookup\_ref(); lookup\_addr(); |
| lssop() | lookup\_ref(); lookup\_addr(); |
| leqop() | lookup\_ref(); lookup\_addr(); |
| eqlop() | lookup\_ref(); lookup\_addr(); |
| neqop() | lookup\_ref(); lookup\_addr(); |
| assop() | lookup\_ref(); lookup\_addr(); |
| sassop() | lookup\_ref(); lookup\_addr(); |
| getaop() | lookup\_ref(); lookup\_addr(); |
| scfop() | lookup\_ref(); lookup\_addr(); |
| prtop() | lookup\_ref(); lookup\_addr(); |
| fpaop() | lookup\_ref(); lookup\_addr(); lookup\_func(); |
| retop() | lookup\_ref(); lookup\_addr(); |
| paraop() | inaddr(); |
| funcop() | lookup\_func(); inaddr(); lookup\_ref(); instack(); callop();  GOTOop(); BZop(); labop(); plusop(); minuop();  mulop(); divop(); grtop(); geqop(); lssop(); leqop();  eqlop(); neqop(); assop(); sassop(); getaop(); scfop();  prtop(); fpaop(); retop(); paraop(); |
| 引用计数法分配寄存器 | |
| lookup\_ref() | None |
| ref\_entering() | None |
| ref\_count() | lookup\_func(); ref\_entering(); sorttable(); |
| sorttable(fun) | None |
| DAG图 | |
| lookup\_nodelist() | None |
| lookup\_nodetabname() | None |
| DAG() | lookup\_nodetabname(); lookup\_nodelist();  lookup\_nodetabindex(); can\_in(x); all\_in(n1); |
| lookup\_nodetabindex() | None |
| can\_in(x) | None |
| all\_in(n1) | None |

### 4．符号表管理方案

struct tab**{**

char name**[**100**];**

int paranum**;**

enum objecttyp obj**;**

enum types type**;**

int refs**;**

int normal**;**

int addr**;**

int link**;**

**}** tab**[**tmax**];**

struct funtab**{**

char name**[**100**];** int paranum**;** int offset**;** int rt**;** struct Ref\_Tab ref\_tab**[**100**];** int refs**;** int sizes**;** enum types type**;**

**}**ftab**[**fmax**];**

struct Ref\_Tab**{**

char name**[**100**];**

int num**;**

**};**

struct Atab**{**

enum types eltype**;**

int elref**,** low**,** high**,** elsize**,** sizes**;**

**}**atab**[**amax**];**

上图所示中，左表格为符号表tab，中间表格为函数索引表ftab，右侧表格为函数变量引用次数表格ref\_tab和数组表atab。

#### （1）符号表tab：

所有常量、变量、和函数都要填入符号表中。

* name表示常量变量或函数的名字。
* paranum域当且仅当填入的标识符为函数名时有用，其记录了函数所包含的参数个数。
* obj域记录标识符的类型，常量为constant，变量为variable，函数为funktion。
* type域记录标识符的具体类型，为整型int或字符型char。若存入的变量为数组，其记录了数组的类型。若存入变量为函数，其记录了函数的返回值类型，为void/ int/ char。
* ref为索引域。若存入的是数组，其记录了数组在atab中的位置，可通过此域进入atab中查找数组的类型、大小等信息。若存入的是函数，其记录了函数在函数表ftab中的位置，可通过此域进入ftab查询函数的更多信息。
* normal域对于标识符为函数时有用，其记录了函数的形参个数是否为0，这对于语法分析时，函数调用是否需要加括号有用。
* link域为指向同一分程序的上一个标识符在符号表tab中的位置。
* isglob域记录了常量或变量是否为全局变量或常量，此域在讨论区要求的表达式要从左至右处理时有用，可以有效防止右边的运算改变左边的全局变量的值。

#### （2）函数表ftab

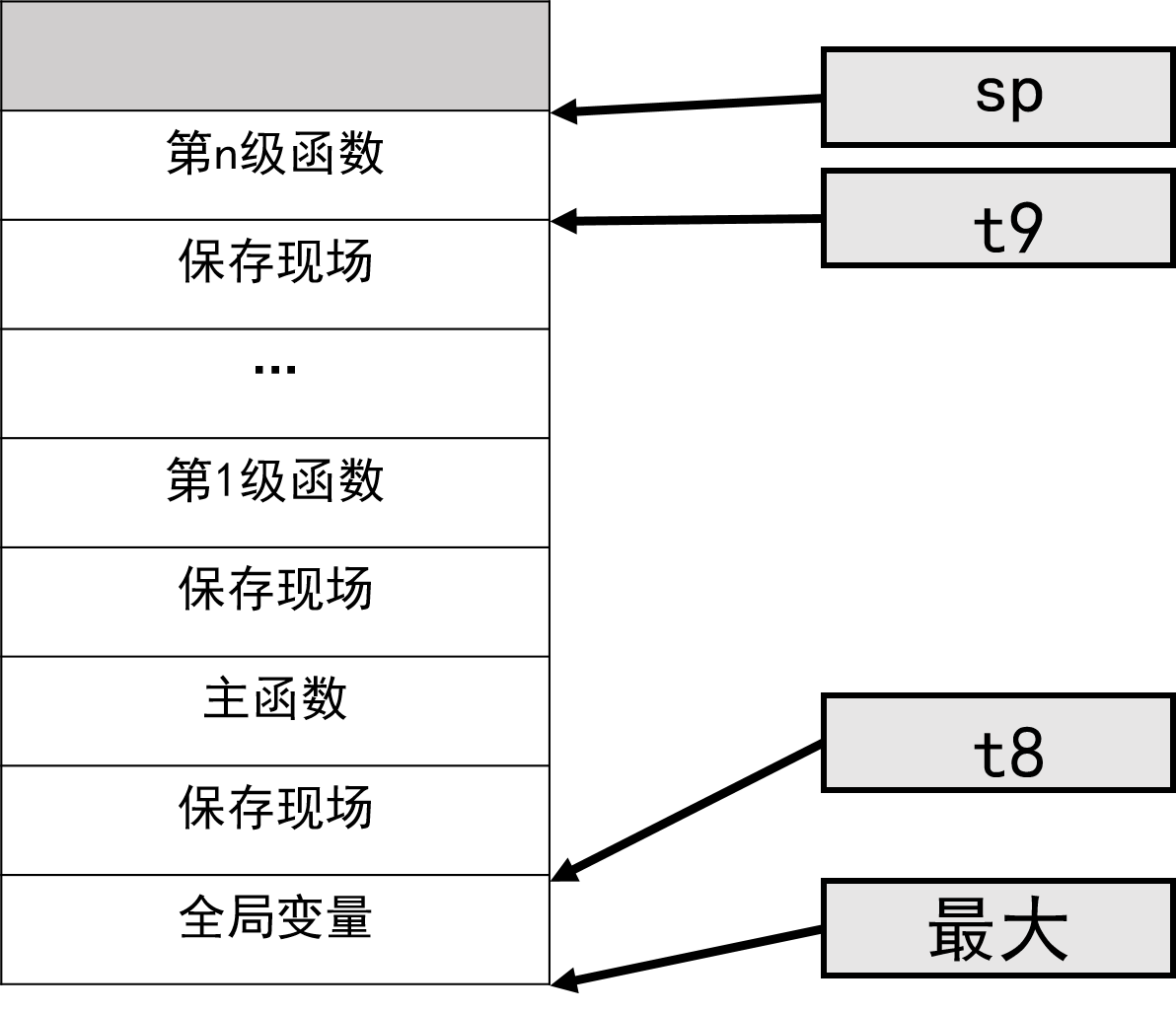
* name[100]：函数名
* paranum：函数参数个数
* offset：此函数中下一个参数的偏移量
* rt：函数中变量个数
* ref\_tab[100]：每个变量对应的引用次数（已排序）
* refs：函数在符号表中的位置
* sizes：函数中变量个数，注意包含中间代码生成的临时变量
* type：函数返回值类型, void/ int/ char

#### （3）数组表atab

* eltyp：数组元素类型，int/ char
* low, high： 数组上下界
* elsize：元素大小
* sizes：数组大小

### 5．存储分配方案

Mars中初始的栈空间$sp指针位于0x2ffc处。程序运行过程中不断将$sp指针上移。自0x2ffc处始，先存储全局的常量和变量。之后是主函数和其它函数的调用：在进入每个函数时（包括主函数），先进行现场保存，将$sp上移，临时寄存器的值、上层运行栈的栈底指针$t9和$ra的返回值都存在栈内，然后继续向栈内堆叠此函数中的局部变量。函数调用返回时，遗弃局部变量（将$sp下移即可），并对保存的现场（临时寄存器、上层运行栈栈底指针和$ra寄存器的值进行恢复），生成一条jr $ra指令返回上一层函数。具体的栈空间分配示意图见下：



然而要想实现上述的栈空间分配与进栈出栈的过程，仅仅使用一个$sp寄存器是不够的，于是引入$t8,$t9两个寄存器。将sp上移（值变小），自顶向下存放全局变量，并将全局变量的顶端保存在$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。

### 6. 四元式设计\*

四元式设计及含义如下：

|  |  |
| --- | --- |
| 四元式 | 说明 |
| const,char,num,name | 常量定义 |
| int ,,,name | 整型变量定义 |
| char,,,name | 字符型变量定义 |
| ints,,num,name | 整型数组定义 |
| chars,,num,name | 字符型数组定义 |
| para,int,,name | 参数定义 |
| func,int,,name | 返回值为整型的函数定义 |
| func,char,,name | 返回值为字符型的函数定义 |
| func,void,,name | 无返回值函数定义 |
| end,,,name | 函数结束 |
| plus,var1,var2,var3 | 加法运算，最后一个操作数为结果 |
| minu,var1,var2,var3 | 减法运算，最后一个操作数为结果 |
| mul,var1,var2,var3 | 乘法运算，最后一个操作数为结果 |
| div,var1,var2,var3 | 除法运算，最后一个操作数为结果 |
| []=,var1,var2,var3 | 对数组元素进行赋值var3[var2] = var1 |
| =,var1,,var3 | 赋值运算，最后一个操作数为结果 |
| geta,name,t,nowvar | 将变量赋为数组的值nowvar = name[t] |
| <,var1,var2,var3 | 小于时置1 |
| <=,var1,var2,var3 | 小于等于时置1 |
| >,var1,var2,var3 | 大于时置1 |
| >=,var1,var2,var3 | 大于等于时置1 |
| ==,var1,var2,var3 | 等于时置1 |
| !=,var1,var2,var3 | 不等时置1 |
| BZ,,,lab | 不为1时则跳转 |
| GOTO,,,lab | 无条件转移 |
| lab:,,,lab | 设置标签 |
| call,var,, | 函数调用 |
| PUSH,,,nowvar | 值参数压栈 |
| ret,,,var3 | 函数返回，若有返回值，存在最后一个操作数中 |
| prt,var1,var2,char/int | 写语句 |
| scf,,,var3 | 读语句 |

### 7. 目标代码生成方案\*

struct Midcode

**{**

char op**[**100**]** **;**

char var1**[**100**];**

char var2**[**100**];**

char var3**[**100**];**

**}**midcodelist**[**Maxcode**]**

struct Addr

**{**

char name**[**100**];**

int addr**;** int type**;**

int isglo**;**

**}**addrlist**[**2000**];**

midcodelist[]存储了编译器生成的中间代码的四元式序列。op为操作符，var1、var2、var3分别为第一个操作数、第二个操作数、第三个操作数。

使用midcodetoasm()函数将中间代码转换为目标代码。addrlist[]保存了各种量（包括全局、局部常量、变量）在程序运行时在$sp栈空间相对于栈底0x2ffc的位移。name[100]为变量或常量的名称。addr域记录了常量或变量所存的地址的偏移量，若是全局变量，则基地址为$t8，若不是全局变量，则基地址为$sp，具体原因详见存储分配方案部分。type域有三种选择const/ int/ char，根据是否为变量、变量的类型而不同。isglo域记录了是否为全局的常量变量。

中间代码生成目标代码，未优化时，每次遇到一个变量，就用lw加载指令根据addrlist[]中存储的变量的偏移量，从对应地址加载数据到$t0或$t1寄存器进行计算，计算后，生成一条sw保存指令，结果同理要根据addrlist[]中存储的变量的偏移量，向对应地址空间存储计算后的结果。

使用引用计数法进行寄存器分配，优化后要先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，否则按上一段中所述方法从内存中存取数据。

对于函数调用时用PUSH四元式进行参数值的传递，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\*

1. 引用计数法分配寄存器

以函数为单位对函数中每个变量进行引用计数，每个变量的引用次数存储在ftab中的ref\_tab[]域。ref\_tab又是一个结构体，ref\_tab[i]包含了函数中第i个变量的名称name[]以及其在函数中被引用的次数num。sorttable()函数会对函数中变量的引用次数从高到低排序。由于只有7个临时寄存器，取出引用次数最多的7个变量，分给它们临时寄存器。int lookup\_ref(int j,char name[])函数可以查找第j个函数中名为name的变量是否被分配了临时寄存器，若分配了寄存器，则返回临时寄存器编号，否则返回-1表示要从内存中存取数据。

在进入函数的一开始，由于临时寄存器中还没有变量的值，需要添加7条lw指令，将可以分配临时寄存器的7个变量的值从内存中取出并赋给寄存器；在函数执行结束后，同样需要添加7条sw指令，将函数最终运算得的7个变量的值存回内存中的地址空间。

1. 利用DAG图消除局部公共子表达式

struct DAG

**{**

char op**[**100**];**

int left\_child**;**

int right\_child**;**

int parent**[**100**];**

int isleaf**;**

int order**;**

**} \***daglist**[**100**];**

struct Nodetab

**{**

char name**[**100**];**

int index**;**

**}**

struct Nodetab nodetab**[**100**];**

DAG图存在daglist[]中，结点索引表为nodetab[]。

结点索引表nodetab中，name域记录了结点存放的变量的名称，index域存储了结点在DAG图中的结点标号，具体对应于daglist[]中的某个下标号。

DAG图以DAG结构体形式存储，daglist[]中的一个元素即表示一个DAG图结点。op域记录了一个结点如果是非叶子节点，那么它的运算符是什么（加减乘除）。right\_child域和left\_child域记录了一个结点如果是非叶子结点，那么它的左孩子和右孩子结点的标号是多少，此域值也对应于daglist[]中某个下标的编号。parent [100]记录了结点的父结点标号，此域值也对应于daglist[]中某个下标的编号，因为一个结点可能有多个父结点，因而parent[]是数组的形式。isleaf域记录了结点是否为叶子结点，只有非叶子结点的op域、right\_child域、left\_child域才有效。order域记录了DAG图导出时此结点的导出顺序编号，只有其父结点全部已经导出后，此结点才可以被导出。

根据中间代码构建DAG图，并适时更新结点编号。导出DAG图采用启发式算法，只有某个结点的父结点全部已经导出后，此结点才可以被导出。之后按导出顺序的逆序计算结点的值。结点中如果是有用的变量则将其计算出来，如果是临时变量则不做计算。但如果某一结点全是临时变量，则取其中一个（第一个）进行计算并保留。若叶子结点中出现多个变量，则输出这多个变量互相相等。

1. 活跃变量分析

对于已生成的中间代码，以函数为单位，按跳转指令所跳到的指令和跳转指令的下一条指令为入口语句的方式划分基本块。对每一基本块做活跃变量分析，最终得到哪些变量冲突、哪些变量不冲突。对于不冲突的变量考虑分配同一个临时寄存器。

### 9. 出错处理

出错处理中打印出了错误信息，包括错误的类型和错误出现的行号和字符号。编译器可以跳读错误代码，对后续程序继续进行编译。跳过错误代码分为跳至分号、跳至大括号、跳至Block末尾等等，详见第2点错误处理部分的函数介绍。错误信息和错误类型定义和说明如下：

|  |  |
| --- | --- |
| 错误编号 | 错误类型 |
| 0 | 缺少源文件 |
| 1 | 文件不完整 |
| 2 | 缺少\" |
| 3 | 不可接受的字符 |
| 4 | 缺少“'” |
| 5 | 超出符号表范围 |
| 6 | 变量名冲突 |
| 7 | 常量声明失败 |
| 8 | 等号右侧字符非法 |
| 9 | 丢失“;” |
| 10 | 关键字错误 |
| 11 | 丢失标识符 |
| 12 | 丢失“]” |
| 13 | 调用函数未定义 |
| 14 | 等式两边类型不一致 |
| 15 | 形参个数不匹配 |
| 16 | 未定义变量 |
| 17 | 丢失“(” |
| 18 | 丢失“)” |
| 19 | 丢失“}” |
| 20 | 函数返回错误 |
| 21 | 表达式缺失或错误 |
| 22 | 句子不合法 |
| 23 | 给常数赋值 |
| 24 | 缺少“{” |
| 25 | 缺少main函数 |
| 26 | 常量没有初始化 |
| 27 | 数字有前零 |
| 28 | 数字过大 |
| 29 | 字符串内容错误 |
| 30 | 字符内容错误 |
| 31 | 变量声明失败 |
| 32 | 参数错误 |
| 33 | 函数声明错误 |
| 34 | 函数定义或调用无参数时有括号 |
| 35 | 数组下标越界 |
| 36 | 主函数main后缺少左括号 |
| 37 | 主函数main后缺少右括号 |
| 38 | 有返回值函数无返回值 |
| 39 | for循环缺少初值或步长 |
| 40 | 推测赋值符错写成了等于号 |

## 三．操作说明

### 1．运行环境

windows系统，直接打开Codeblocks，打开工程项目文件“C\_Compiler.cbp”，点击运行即可。

### 2．操作步骤

windows系统，直接打开Codeblocks，打开工程项目文件“C\_Compiler.cbp”，点击运行。屏幕上提示输入文件路径，此时将源程序的路径输入，回车表示结束。屏幕上紧接着会提示询问是否要进行优化，输入整数0表示不做优化编译，输入整数1表示编译时进行优化，回车结束。关闭Codeblocks工程文件，若选择不做优化，未做优化的MIPS代码在工程文件夹下”16231181\_ aimcode.asm”中；若选择进行优化，优化的MIPS代码在工程文件夹下”16231181 \_aimcode\_opt.asm”中。打开Mars，运行上述asm文件即可得到最终结果。

## 四．测试报告

### 1．测试程序及测试结果

#### （1）错误程序：

注：在测试错误时，也测试了较长的程序段出现错误。具体方法是将覆盖完整的正确程序改出几处错误进行测试。这里由于空间限制，仅展示了五个较短的错误测试程序。

**错误程序一：**

void main**(){**

int i**;**

**for(**i **=** 1**;** i **<** 10**;** i **=** i**+**1**){**

sum **=** sum **+** i**;**

**}**

printf**(**i**);**

**}**

**运行结果：**

in line 4, in sym 6: 未定义变量

**错误程序二：**

int array**[**5**];**

void main**(){**

array**[**0**]** **=** 0**;**

array**[**1**]** **=** 1**;**

array**[**2**]** **=** 2**;**

array**[**3**]** **=** 3**;**

array**[**4**]** **=** 4**;**

array**[**5**]** **=** 5**;**

**}**

**运行结果：**

in line 8, in sym 12: 数组下标越界

**错误程序三：**

int a**;**

int b**;**

char c

char e**;**

char f**,**g

int d**;**

void main**(){**

printf**(**"testing"**);**

**}**

**运行结果：**

in line 4, in sym 5: 丢失“;”

in line 6, in sym 4: 丢失“;”

**错误程序四：**

void main**(){**

int i**;**

int ans**;**

ans **=** 0**;**

**for(** i **=** 0**;** i **<** 7**;** i **==** i **+** 2**){**

ans **=** ans **\*** i**;**

**}**

printf**(**ans**)**

**}**

**运行结果：**

in line 5, in sym 25: 推测赋值符错写成了等于号

in line 9, in sym 2: 丢失“;”

**错误程序五：**

const int a1 **=** 1**,** a2 **=** 2**;**

const int a3 **=** 3**;**

const int a4 **=** 4**;**

int a**,**b

int c**;**

int d**;**

void main**({**

a **=** a1**;**

b **=** a2**;**

c **=** a3**;**

d **=** a4**;**

printf**(**c**);**

printf**(**d**);**

printf**(**e**);**

a4 **=** 9;

**}**

运行结果：

in line 5, in sym 4: 丢失“;”

in line 7, in sym 12: 主函数main后缺少右括号

in line 14, in sym 10: 未定义变量

in line 15, in sym 2: 给常数赋值

#### （2）正确程序：

**正确程序一：**

// 16231181\_test.txt

// global constant declaration

const int glob\_const\_int1 **=** 0622**,** glob\_const\_int2 **=** 1020**;**

const char glob\_const\_char1 **=** '\_'**,** glob\_const\_char2 **=** 'f'**,** glob\_const\_char3 **=** 'G'**;**

const int glob\_const\_int3 **=** **-**0213**;**

const int glob\_const\_int4 **=** 9**;**

const char glob\_const\_char4 **=** '\*'**,** glob\_const\_char5 **=** '+'**;**

// global variable declaration

char glob\_var\_char**,** glob\_var\_array1**[**10**];**

int glob\_var\_int1**,** glob\_var\_int2**,** glob\_var\_array2**[**9**];**

// a function to calculate the factorial of n

int factorial **(**int n**){**

**if** **(**n **==** 1**)** **return** **(**1**);**

**else** **return** **(**n**\***factorial**(**n**-**1**));**

**}**

// 返回值为char，无传入参数的函数

char returnPlus**{**

**return** **(**'+'**);**

**}**

// 返回值为int,无传入参数的函数

int returnBirth**{**

int temp**;**

int vari**;**

vari **=** glob\_const\_int2**;**

temp **=** 002**;**

vari **=** temp **+** vari **-**temp**\***temp**;**

**return** **(**vari**);**

**}**

// 无返回值，有传入参数的函数

void printchar **(**char ch**){**

printf**(**ch**);**

**return;**

**}**

// 无返回值，无传入参数的函数

void playing**{**

printf**(**"successfully reach the playing func"**);**

**return;**

**}**

// 检测条件语句

void testif **(**int n**){**

int ans**;**

printf**(**"testing if"**);**

**if** **(**n **==** 3**){**

ans **=** factorial**(**n**);**

printf**(**"calculating the factorial of 3"**);**

printf**((**ans**));** //检验双括号是否当作一个

**}**

**else** **if** **(**n **==** 4**){**

ans **=** factorial**(**4**);**

printf**(**"calculating the factorial of 4"**);**

printf**(**ans**);**

**}**

**else** printf**(**"the number is not 3 or 4"**);**

**}**

// 检测for循环语句

int testfor**(**int n**){**

int sum**;**

int i**;**

sum **=** 0**;**

printf**(**"testing for"**);**

**for** **(**i **=** 0**;** i **<=** n**;** i **=** i**+**1**){**

sum **=** sum **+** i**;**

**}**

**return** **(**sum**);**

**}**

// 检测while循环语句

int testWhile**(** int n **){**

int sum2**;**

int i**,**temp**;**

sum2 **=** 00100**;**

i **=** n**;**

temp **=** glob\_const\_int1**;**

printf**(**"testing while"**);**

**while(**i**>**0**){**

sum2 **=** sum2 **-** i**;**

i **=** i **-**1**;**

**}**

**return** **(**sum2 **+** temp**\***2**);**

**}**

void main**(){**

int array**[**9**];**

int i**,**temp**;**

char ch**;**

int x1**;**

int x2**;**

int x3**;**

int x4**;**

int a**,** b**,** c**;**

x1 **=** 63**;**

x2 **=** 9**;**

x3 **=** 2**;**

**for** **(**i**=**0**;** i**<**GloB\_cOnST\_int4**;** i**=**i**+**1**){**

// 检测赋值语句和循环语句

array**[**i**]** **=** i**\***i**;**

printf**(**array**[**i**]);**

//检测全局数组变量

glob\_var\_array2**[**i**]** **=** array**[**i**]** **+** 1**;**

printf**(**glob\_var\_array2**[**i**]);**

**}**

/\*检测一下是否过滤掉了\*\*\*\*注释\*\*

继续检测中。。。\*\*\*/

x4 **=** x1**\*(**x3**+**1**)/**x2**;** // 检测加减乘除法

printf**(**x4**);**

scanf**(**a**,** b**,** c**);** // 检测读语句

x4 **=** **(**x1 **+** 72**)/**x2 **-** testWhile**(**c**);**

printf**(**x4**);**

**if** **(**x1 **!=** x2**)** printf **(**"True"**,** 1**);**

**if** **(**x1 **>=** x3**)** printf**(**"True"**,** 2**);**

printf**(**"! ~}|{abcABC<=+-$%&\*()/"**);**

testif**(**a**);** // 检测条件语句

temp **=** testfor**(**b**);** // 检测for循环语句

printf**(**temp**);** // 检测写语句

temp **=** testWhile**(**c**);** // 检测while循环语句

printf**(**temp**);**

temp **=** factorial**(**6**);** // 检测函数递归调用

printf**(**temp**);**

playing**;** // 检测无返回值无传入参数的函数调用

printchar**(**glob\_const\_char1**);** // 检测全局常量的调用

printchar**(**glob\_const\_char5**);** // 检测无返回值有传入参数的函数调用

ch **=** returnPlus**;** // 检测返回值为char无传入参数的函数调用

printf**(**ch**);**

temp **=** returnBirth**;** // 检测返回值为int无传入参数的函数调用

printf**(**temp**);**

**}**

**运行结果：**

Case1:

输入：4 10 4

输出：

0 1 1 2 4 5 9 10 16 17 25 26 36 37 49 50 64 65 21 testing while -1319 True1 True2

! ~}|{abcABC<=+-$%&\*()/ testing if calculating the factorial of 4 24

testing for 55 testing while 1334 720 successfully reach the playing func \_++ 1018

Case2:

输入：3 99 99

输出：

0 1 1 2 4 5 9 10 16 17 25 26 36 37 49 50 64 65 21 testing while 3621 True1 True2

! ~}|{abcABC<=+-$%&\*()/ testing if calculating the factorial of 3 6 testing for

4950 testing while -3606 720 successfully reach the playing func \_++ 1018

Case3:

输入：

100 75 57

输出：

0 1 1 2 4 5 9 10 16 17 25 26 36 37 49 50 64 65 21 testing while 324 True1 True2

! ~}|{abcABC<=+-$%&\*()/ testing if the number is not 3 or 4 testing for 2850

testing while -309 720 successfully reach the playing func \_++ 1018

**正确程序二：**

cONSt int num1**=**10**,** num2**=**001**,** num3**=+**0010**,** num4**=-**0019**;**

const char char1**=**'a'**,** char2**=**'b'**;**

int num5**,** num6**,** array1**[**009**];**

char char3**,** char4**,** array2**[**0010**],** testresult**[**10**];**

int return1

**{**

**return** **(**1**);**

**}**

int factorial**(**int n**)**

**{**

**if(**n**==**1**)return** **(**n**);**

**else** **return(**n**\***factorial**(**n**-**1**));**

**}**

void testvar

**{**

const int chAr2**=-**000109**;**

char NUM1**;**

int array2**,** char1**[**10**];**

array2**=**10**;**

num1**=**'a'**;**

**}**

int testfor**(**int a**,** char b**,** int c**,** char d**,** int f**,** char e**){**

int i**;**

a**=**9**;**

c**=**0**;**

**for(**i**=**100**;**i**<**10**;**i**=**i**-**1**)**a**=**a**+**1**;**

**for(**i**=**0**;**i**<**10**;**i**=**i**+**1**)**c**=**c**+**1**;**

**if(**a**==**10**){**

**if(**c**==**10**){**

**return** **(**1**);**

**}**

**}**

**return** **(**0**);**

**}**

int testwhile

**{**

int num1**,** num2**,** num3**,** num4**;**

num1**=**0**;**

num2**=**10**;**

num3**=-**1**;**

num4**=**0**;**

**while(**num1**!=**10**){**

num2**=**num2**-**1**;**

num1**=**num1**+**1**;**

**}**

**while(**num3**>**0**){**

num4**=**num4**+**10**;**

num3**=**num3**-**10**;**

**}**

**if(**num2**==**0**){**

**if(**num4**==**0**)**

**return** **(**1**);**

**}**

**else** **return** **(**0**);**

**}**

int testexpression

**{**

int a**,**c**;**

char b**;**

a**=**10**;**

b**=**'a'**;**

array1**[**0**]=**100**;**

array1**[**1**]=**99**;**

c**=(**a**-**b**+**'a'**)\***return1**-**array1**[**0**]/**10**+**1**;**

**;;;**

**if(**array1**[**c**]==**99**)return** **(**1**);**

**return** **(**0**);**

**}**

int testarray

**{**

int a**,**b**,**c**;**

a**=**1**;**

b**=**2**;**

c**=**3**;**

array1**[**0**]=**1**;**

array1**[**4**]=**array1**[**0**];**

array1**[**c**+**a**\*(**b**+**return1**)-**array1**[**0**]\***2**]=**10**;**

**if(**array1**[**4**]==**10**)return** **(**1**);**

**return** **(**0**);**

**}**

void testprintf**(**int flag**)**

**{**

**if(**flag**<=**0**)**printf**(**"this is a string, !#$%&'()\*+,-./0123456789:;<=>?@[\]^\_`{|}~|"**);**

**else** **if(**flag**>=**2**){**

printf**(**'a'**);**

printf**(**'2'**);**

printf**(**'\*'**);**

**}**

**else** printf**(**"this is a string "**,** num1**-**5**);**

**}**

char testfactorial

**{**

**if(**factorial**(**1**)==**1**)**

**if(**factorial**(**3**)==**6**)**

**return** **(**'T'**);**

**return** **(**'F'**);**

**}**

void main**()**

**{**

int i**,** flag**;**

num5**=**9**;**

num6**=**8**;**

char3**=**'a'**;**

char4**=**'b'**;**

scanf**(**flag**);**

**for(**i**=**0**;**i**<**10**;**i**=**i**+**1**)**testresult**[**i**]=**'T'**;**

**for(**num5**=**9**;**num5**>=**0**;**num5**=**num5**-**1**)**array2**[**num5**]=**testresult**[**num5**];**

**for(**num6**=**0**;**num6**!=**8**;**num6**=**num6**+**1**)**array1**[**num6**]=**0**;**

testvar**;**

**if(**testfor**(**num5**,** char1**,** num6**,** chAr2**,** 10**,** char4**)==**1**)**testresult**[**0**]=**'T'**;**

**else** testresult**[**0**]=**'F'**;**

**if(**testwhile**)**testresult**[**1**]=**'T'**;**

**else** testresult**[**1**]=**'F'**;**

**if(**testfactorial**==**'T'**)**testresult**[**2**]=**'T'**;**

**else** testresult**[**2**]=**'F'**;**

**if(**testexpression**)**testresult**[**3**]=**'T'**;**

**else** testresult**[**3**]=**'F'**;**

**if(**testarray**)**testresult**[**4**]=**'T'**;**

**else** testresult**[**4**]=**'F'**;**

testprintf**(**flag**);**

**for(**i**=**0**;**i**<**5**;**i**=**i**+**1**)**printf**(**testresult**[**i**]);**

**}**

**运行结果：**

Case1:

输入：0

输出：this is a string, !#$%&'()\*+,-./0123456789:;<=>?@[\]^\_`{|}~|TTTTT

Case2:

输入：1

输出：5TTTTT

Case3：

输入：2

输出：a2\*TTTTT

**正确程序三：**

int x**;**

int add**{**

x **=** 2**;**

**return** **(**x**);**

**}**

void main**(){**

int ans**;**

x **=** 1**;**

ans **=** x **+** add**;**

printf**(**ans**);**

**}**

**运行结果：**

3

**正确程序四：**

const int a **=** 10**;**

int factorial**(**int n**){**

**if(**n **==** 1**){**

**return** **(**1**);**

**}**

**else{**

**return** **(**n **\*** factorial**(**n**-**1**));**

**}**

**}**

void main**(){**

int ans**;**

ans **=** factorial**(**a**);**

printf**(**ans**);**

**}**

**运行结果：**

3628800

**正确程序五：**

const int zero **=** 0**;**

const int one **=** 1**,** minus\_two **=** **-**0002**;**

const char renn **=** '0'**;**

const char first\_alpha **=** 'a'**,** first\_captical **=** 'A'**;**

int x**,** yy**,** zz**,** yyy**[**7**];**

int fact**(**int t**)**

**{**

**if(**t **<** 3**)**

**return(**t**);**

**else**

**return(**fact**(**t**-**1**)\***t**);**

**}**

int rt15

**{**

const int zero**=**1**;**

**return(**3**\***5**/**ONE**+**one**-**ZERO**);**

**}**

char fun2**(**int a**,** int b**,** int c**,** int d**,** int e**,** int f**)**

**{**

printf**(**a**+**b**\***c**\***d**/**e**-**f**);**

**return(**'A'**);**

**}**

void main**()**

**{**

const int b **=+**01**,** c**=**2**,** d**=**3**,** e**=**4**;**

char ar**[**2**];**

int r**,** m1**[**8**];**

int aa**,** a**;**

x **=** **+-**1**;**

r **=** **-**4**;**

aa **=** 6**;**

printf**(**"!@#$%%abcd114514ABCD963287~;: "**,** fact**(**AA**));**

AR**[**d**\***e**+**6**\*(**minus\_two**+**b**-**1**)]** **=** fun2**(**x**-**r**,** aa**,** b**,** e**,** c**,** minus\_two**)** **;**

printf**((**ar**[**ZeRo**/**fact**(**4**)]));**

ar**[**1**]** **=** '0'**;**

aa **=** **-**b **\*** **(**ar**[**1**]** **+** rt15**)** **+** **(**c **-** e **\*** 9**)** **+** 2 **\*** 'm'**;**

printf**(**aa**);**

a **=** aa**;**

scanf**(**a**,** x**);**

**if** **(**a **>** 0**)**

**{**

printf**(**"1 hit "**);**

**if** **(**x **<=** 0**)** **{**

r **=** a **/** x **\*** a**;**

printf**(**"2 hit "**);**

**}**

r **=** r **+** aa**;**

**}**

**if** **(**x **!=** 0**)**

**{**

printf**(**"3 hit "**);**

**if** **(**a **==** **-**4**)**

**{**

r **=** r **+** 1**;**

printf**(**"4 hit "**);**

**}**

**}**

**if** **(**a **<** 0**)**

**{**

printf**(**"5 hit "**);**

**if** **(**x **>=** 0**)**

**{**

r **=** r **+** a**;**

printf**(**"6 hit "**);**

**}**

**}**

**else**

**{**

r **=** r **\*** **-**1**;**

printf**(**r**);**

**}**

**for(**a **=** **-**2**;** a**<**d**;** a **=** a**+**1**)**

**{**

x **=** x **+** e**;**

**}**

**while** **(**x **>** 0**)**

**{**

x **=** x **-** c**;**

**}**

printf**(**x**);**

**}**

**运行结果：**

Case1：

输入：1 3

输出：!@#$%abcd114514ABCD963287~;: 720 17 65 121 1 hit 3 hit -117 -1

Case2：

输入：-3 1

输出：!@#$%abcd114514ABCD963287~;: 720 17 65 121 3 hit 5 hit 6 hit -1

Case3：

输入：6 -1

输出：!@#$%abcd114514ABCD963287~;: 720 17 65 121 1 hit 2 hit 3 hit -85 -1

Case4：

输入：-4 3

输出：!@#$%abcd114514ABCD963287~;: 720 17 65 121 3 hit 4 hit 5 hit 6 hit -1

### 2．测试结果分析

#### （1）错误测试：

错误包含语法错误和语义错误。从五个测试用例来看，各种典型的错误类型基本覆盖完善。这五个测试程序中包含的语法错误有丢失分号、主函数main后缺少右括号、赋值符错写成了等于号等等。语义错误包含了未定义变量、数组下标越界、给常数赋值等等。

第4、5个错误测试程序主要测试编译器是否能处理多个错误以及出错后跳读是否正确。从测试结果可见，程序具备处理多个错误的能力，且跳读错误后仍能继续编译后续程序。

在测试错误时，也测试了较长的程序段出现错误。具体方法是将覆盖完整的正确程序改出几处错误进行测试。这里由于空间限制，仅展示了五个较短的错误测试程序。老师可见我的五个正确测试程序比较长、比较完善、且覆盖情况良好。

#### （2）正确测试：

测试程序完整且覆盖情况良好：

* 包含每条规则的每个可能的产生式右部；
* 测试了for循环先进入循环执行一次再进行判断；
* 测试了标识符不区分大小写；
* 包含有返回值的函数调用和无返回值的函数调用；
* 包含无参数传入的函数调用和有一至多个参数传入的函数调用；
* 关系运算符包含> >= < <= == !=全部；
* 包含了整型与字符型的常量、变量、数组；
* 包含读语句和写语句；
* 包含for循环,while循环,if条件语句；
* 包含上述三种语句以及各种语句的嵌套使用。

这里单独提出三点：

* 测试程序一包含了注释，且主函数中有一段含有很多’/’或’\*’符，专门用来测试编译器是否能过滤掉所有注释，以证明我扩充的带注释的文法的正确性；
* 测试程序三专门用来测试讨论区的第七条约定，即表达式的计算要从左至右。如果右侧改变了左侧某变量的值，已经读入的左侧变量在运算时的值不能改变。具体到此程序，输出结果应为1+2=3，而非2+2=4；
* 测试程序四专门用来测试递归调用以及函数的多层调用，以测试编译器是否能处理多层函数嵌套或递归调用的情况。

这里特别感谢秦子柠大佬在讨论区分享自己的测试程序，并同意我将他的测试代码用在我上文的正确测试程序5中。经过检验，此测试程序与我设计的其它测试程序都很完善，覆盖全面，帮助我降低了我编译器出现Bug的可能性。

## 五．总结感想

编译技术是大三上这一学期的重头戏，学完收货颇多，是值得向学弟力荐的一门课。

首先，这门课从帮我提升了我的动手编程能力。之前大一学习C语言时编写的程序都十分短小，我之前编写过的最长的程序也就是数据结构课程的大作业，顶多四五百行C语言代码。而这次的C0编译器加起来要达到四五千行。如此庞大的工程在短短一个多月内完成，路遇不少Bug与艰辛，但一路走来其实现在再去回首，这一个月熬的夜都是值得的。我在Debug方面长了不少经验，了解某一个错误最可能是由于什么原因导致的，而不用再每次都像从前一样从第一行开始运行代码，一行一行去观察结果，这样既费时又费力。

当然说起这是一个庞大的工程，做完看下来，我觉得最值得我珍藏的就是高内聚、低耦合的思想。同一模块或函数内尽量去实现一个单一且统一的功能，而对于不同的功能，尽量将其封装成不同的模块或函数。大到编译工程，小到每个编译的阶段，都需要去提前设计，划分函数与模块，否则后续会耽误大量的时间。整个编译工程分为词法分析、语法分析、语义分析、中间代码生成、目标代码生成、优化处理、符号表管理等步骤，于是我们不妨将这些步骤分开，为每个步骤编写一个.c文件，最后链接起来编译。而对于每一个编译步骤，亦即每一个.c文件，我们又可以将其拆分为若干个功能单一的小函数。记得老师在理论课上说过，编译器是一个大的工程，要想做好，必须提前设计，将大化为小，逐一击破。我想，逐一击破，大概就是说要我们不要拿到一个大任务就畏首畏尾，而是想办法将其分解成小任务，一个一个去做。拿到任务不要马上开始码代码，花费在设计上的时间永远是值得的，是事半功倍的。我也秉承了这么思想，例如在做语法分析时，将每个规则左部封装成一个函数。千万不能将表达式的处理函数中也一同包含了项和因子的处理，而要将表达式、项、因子封装成三个函数，否则编写起来既没有“欲望”，又增加了出现Bug的风险。

最后对于一个大的工程，我体会到了测试的重要性。我们必须将设计完善、覆盖情况全面。例如，我最初在设计测试程序时，认为大于、小于、等于、不等于等关系运算符都是等价的，反正都是关系运算符，测试某一个就不用测试其它几个了。我最初版的测试程序没有覆盖到大于等于号，然而我的代码疏漏正是出现在大于等于这个符号的地方，在读完大于等于后，我多写了一句读取下一个字符nextsym()，导致大于等于号后面的标识符被忽略，程序编译出错。巧合的是，目标代码测试一的当堂测试恰巧考到了大于等于符号，导致我直接挂掉了那次测试。回来后苦苦Debug才终于找到问题所在。所以，不能因为一个东西小，或看似与其他等价，我们就去忽略测试，将测试集覆盖完全，也可以起到事半功倍的效果，快速发现代码的漏洞。这也让我理解了更大的工程为什么会有那么多Bug。例如我平时玩儿的游戏，开发者在游戏测试时并不能很好地测试到游戏中的某些极端情况，导致玩家玩儿游戏时才会出现许多Bug。在我今后步入公司职场后，亦是同理，公司开发的项目一定更大更广。然只有从小处做起，从每条规则做起，从每个函数做起，才能为庞大的工程项目打好坚实的基础。