《编译技术》课程设计文档

学号：\_\_\_\_\_\_\_\_\_13051259\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_\_\_江涛\_\_\_\_\_\_\_\_\_\_\_

年 月 日

## 一．需求说明

### 1．文法说明

* ＜加法运算符＞ ::= +｜-

定义加减法运算符

例子： 1+3 a+b c+9

* ＜乘法运算符＞ ::= \*｜/

定义乘除法运算符

例子： 1/2 a\*b c/9

* ＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

定义关系运算符

例子： 4<5 2>=0 a!=b

* ＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

定义字母可以为英文大小写字母以及下划线

例子： a B \_

* ＜数字＞ ::= ０｜＜非零数字＞

定义数字为0-9的单个数字字符

例子： 1 0 2

* ＜非零数字＞ ::= １｜．．．｜９

定义非零数字为1-9的单个数字字符

例子： 2 3 9

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

定义字符为以单引号为开头和末尾，中间有且仅有一个字符

例子： ‘+’ ‘/’ ‘\_’ ‘2’

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

定义字符串为以双引号为开头和末尾，中间没有或有任意多个ascii码在32,33,35-126范围内的字符

需要注意双引号本身不包含在范围内，但包含了空格、单引号，换行符、制表符等也不在范围内，因而斜杠就是单纯的斜杠，不视为转义符，这一点在后期实现汇编输出时需要格外注意，要将斜杠转义再使用。

例子： “Hello World!”

* ＜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＞

定义程序结构由全局常量定义、全局变量定义、函数定义、主函数定义四部分构成，且顺序固定，每个部分定义的内容可以有若干条，也可以没有。

需要注意的是函数定义中又包括有返回值的函数定义和无返回值的函数定义，该部分中两种函数定义无顺序约束，可以任意交换。

例子：const char a=’a’;

int b;

void func1(){}

int func2(){return (1);}

void main(){return (0);}

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

定义常量说明可以有不止一条常量定义，且每条格式统一为const＜常量定义＞;

例子： const int a = 1;

const char c = ‘c’;

* ＜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＞}

定义常量定义可以是单独的一条，也可以用逗号分隔进行连续的多条。常量的类型为int或者char，且常量在定义时进行赋值。

例子：const int a = 1;

const int b=2,c=3,d=-4;

* ＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

定义无符号整数以非零数字开头，后面接若干个数字的形式。说明无符号整数不会有前导零存在。0也不是无符号整数。

例子：123 300

* ＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

定义整数为前面可能存在一个正负号的无符号整数，或者为0。

需要注意的是虽然整数0前面不带正负号，但放在表达式中时0前面是可能有符号的。

例子： 0 +123 -22

* ＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

定义标识符是必须以字母开头，后面跟若干个字母或数字的字符串

例子： \_123 a1 b\_2

* ＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

定义声明头部为int或char后跟标识符的形式

* ＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

定义变量说明部分可以有不止一条变量定义，且每条格式统一为{<变量定义>;}

例子： int a;

char b;

* ＜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＞‘]’ }

定义变量定义必须以类型标识符开头，后跟若干个单独的标识符或带中括号的标识符。中括号中的数字只能是无符号整数，因而大于等于1。

例子： int a,b,c,d[2];

* ＜常量＞ ::= ＜整数＞|＜字符＞

定义常量只能为整数或者字符

* ＜类型标识符＞ ::= int | char

定义类型标识符只能是int或char

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

定义有返回值函数定义格式为以声明头部开头，后面跟小括号包含的参数部分，之后是大括号包含的复合语句部分

例子： int func(){}

char func2(int a){}

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

定义无返回值函数定义格式为以void开头跟标识符，后面跟小括号包含的参数部分，之后是大括号包含的复合语句部分

例子： void func1(){}

* ＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

定义复合语句由常量说明、变量说明、语句列三个部分构成，其中常量说明和变量说明不是必须存在的，且这三个部分的顺序是固定的。

* ＜参数＞ ::= ＜参数表＞

定义参数由参数表构成

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

定义参数表要么为空，要么由连续的、以逗号分隔的类型标识符和标识符构成。

例子： int a, char b

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

定义主函数格式

* ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

定义表达式开头可能存在正负号，之后是若干个用加法运算符连接的项。

例子： +3-a+b-c

* ＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

定义项为若干个用乘法运算符连接的因子

例子： a\*b/c

* ＜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＞‘)’

定义因子是由标识符或带中括号里包含表达式的标识符或整数或字符或有返回值函数调用语句或以小括号包含的表达式构成的。

例子： (s+1) str[2] func()

* ＜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＞;

需要注意空语句后面也要有一个分号

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

例子： a= 3

b[2] = 1

* ＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

定义条件语句中，else部分可有可无

* ＜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

定义条件格式为单独的一个表达式或用一个关系运算符连接两个表达式的形式，对于单独一个表达式构成的条件，以非零为布尔真，零为布尔假

需要注意的是条件中只能有一个关系运算符，而不能连接多个

例子： a > b

* ＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

例子： while(a< 3) while(1)

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

需要注意的是缺省部分可有可无，因此情况语句中可以没有default

* ＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

定义情况表中至少有一条情况子语句

* ＜情况子语句＞ ::= case＜常量＞：＜语句＞

需要注意的是情况子语句中的常量只能是整数或字符，不能是标识符

* ＜缺省＞ ::= default : ＜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＞‘)’

两种函数调用语句格式一样，由符号表区分

* ＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

定义值参数表要么为空，要么由若干个用逗号分隔的表达式构成

* ＜语句列＞ ::= ｛＜语句＞｝

定义语句列由零个或若干个语句构成

* ＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

需要注意的是读语句中的参数只能是标识符，因此不能给数组元素进行读入

例子： scanf(a) scanf(b,c)

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

定义写语句为输出字符串加表达式结果和单独输出字符串和单独输出表达式结果三种形式，需要注意的是格式是固定的，且由于没有占位符，表达式的结果直接跟在字符串后面。

例子： printf(“a=”,a)

* ＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

定义返回语句为以return起始，后面接可有可无的小括号包含的表达式。需要注意的是无返回值的情况，那么不会有小括号。

例子： return (2); return ;

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）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．目标代码说明

目标代码选择为32位MIPS指令集，输出的MIPS汇编程序的后缀名为.txt，可以直接在MARS上运行。目标代码生成生出的输入为存放中间代码的类数组。

|  |  |  |
| --- | --- | --- |
| **指令** | **示例** | **语义** |
| 系统调用 | syscall |  |
| 取立即数 | li $v0,10 | $v0 = 10 |
| 加 | add $t0,$t1,$t2 | $t0 = $t1 + $t2 |
| 立即加 | addi $t0,$t1,1 | $t0 = $t1 + 1 |
| 减 | sub $t0,$t1,$t2 | $t0 = $t1 - $t2 |
| 乘 | mult $t1,$t2 | {Hi,Lo} = $t1 \* $t2 |
| 除 | div $t1,$t2 | Lo = $t1 / $t2  Hi = $t1 % $t2 |
| 取字 | lw $t0,0($t1) | $t1 = memory[$t1 + 0] |
| 写字 | sw $t0,0($t1) | memory[$t1 + 0] = $t0 |
| 取地址 | la $a0,str | $a0 = &str |
| 相等跳转 | beq $t1,$t2,lab1 | if($t1 == $t2 ) goto lab1 |
| 不等跳转 | bne $t1,$t2,lab2 | if($t1 != $t2) goto lab2 |
| 大于等于0跳转 | bgez $t1,lab3 | if($t1 >= 0 )goto lab3 |
| 大于0跳转 | bgtz $t1,lab4 | if($t1 > 0 )goto lab4 |
| 跳转至标签 | j lab | goto lab |
| 跳转至$ra | jr $ra | goto $ra |
| 函数跳转 | jal func | $ra = PC + 4  goto func |

### 3. 优化方案\*

窥孔优化：

1. 二的次方的乘除法都用左移右移元算代替
2. 数组声明时，生成循环汇编代码进行指定个数的数组元素压栈操作
3. 涉及判断跳转语句的条件和jne这两条连续的四元式合成为一条bgez或bgtz
4. 将返回四元式ret和函数结束四元式end合成为一条
5. 优化data区字符串的数量，将冲出现的字符串整合掉，只在data区中保存一次

## 二．详细设计

### 1．程序结构



### 2．类/方法/函数功能

|  |  |
| --- | --- |
| **所有类/方法/函数名** | **说明** |
| **词法分析：** |  |
| Void getChar(); | 获取一个字符，若为大写字母则转为小写 |
| Void getSym() | 获取一行的词语 |
| Void getOneSym(); | 获取一个词语 |
| **语法分析：** |  |
| void programProc(); | 程序递归子程序 |
| void headProc(); | 主函数以前的头部定义递归子程序 |
| void constProc(); | 常量说明语句块递归子程序 |
| void constDec(); | 常量声明递归子程序 |
| void varProc(); | 变量说明语句块递归子程序 |
| void varDec(); | 变量声明递归子程序 |
| void funcProc(); | 函数定义递归子程序 |
| void returnFuncDec(); | 有返回函数定义递归子程序 |
| void voidFuncDec(); | 无返回函数定义递归子程序 |
| void mainProc(); | 主程序递归子程序 |
| void compoundStatement(bool ret); | 复合语句递归子程序 |
| void statementList(bool ret); | 语句列递归子程序 |
| void statement(bool ret); | 语句递归子程序 |
| void condition(bool ret); | 条件语句递归子程序 |
| void loop(bool ret); | 循环语句递归子程序 |
| void valueParamList(); | 值参数表递归子程序 |
| void assignment(); | 赋值语句递归子程序 |
| void scanfStatement(); | 读语句递归子程序 |
| void printfStatement(); | 写语句递归子程序 |
| void returnStatement(bool retv); | 返回语句递归子程序 |
| void switchStatement(bool ret); | 选择语句递归子程序 |
| void caseStatement(bool ret); | 情况语句递归子程序 |
| string caseTable(string,string,bool ret); | 处理case语句中的标号，将各标号对应的目标代码入口地址填入casetab表中，并检查标号有无重复定义 |
| void oneCase(); | 处理一条case语句 |
| void callvoid(); | 无返回函数调用递归子程序 |
| void paramTable(); | 参数表递归子程序 |
| void expression(); | 表达式递归子程序 |
| void simpleExpression(); | 处理简单表达式 |
| void term(); | 项递归子程序 |
| void factor(); | 因子递归子程序 |
| void callReturn(); | 有返回函数调用递归子程序 |
| void testSemicolon(); | 检查应有分号 |
| void uInt(); | 无符号整数递归子程序 |
| **中间代码：** |  |
| Void insMidCode(); | 插入中间代码 |
| string nextLabel(); | 返回下一个label名 |
| string nextVar(); | 返回下一个临时变量名 |
| **汇编相关：** |  |
| Void midcode2MIPS(); | 中间代码转mips汇编 |
| Int get\_from\_stack(string name) | 在变量表中查找变量的位置 |
| void clean\_t() | 清空临时t系列寄存器池 |
| void clean\_s() | 清空临时s系列寄存器池 |
| int get\_t() | 从临时t系列寄存器池中取一个空闲寄存器 |
| int get\_s() | 从临时s系列寄存器池中取一个空闲寄存器 |
| void addtempvar(string name) | 将一个临时寄存值登录到临时寄存值表 |
| void sinktempvar() | 更新临时寄存值表中各寄存值的位置信息 |
| void pushFuncIntoSStack() | 进行函数调用的一开始必要进行的隐式参数处理 |
| void push\_Const\_Var(QUAT \* q) | 将全局常变量放入运行栈 |
| void get\_from\_stack(int t, string name, bool get\_addr = 0, int t\_offset = -1) | 从运行栈中读取变量name的值放入临时寄存器t中 |
| void asm\_func(QUAT\* q) | 函数头部 |
| void asm\_param(QUAT \* q) | 形式参数 |
| void asm\_end(QUAT\* q) | 函数结尾 |
| void asm\_call(QUAT\* q) | 函数调用 |
| void asm\_plus(QUAT\* q) | 加法运算 |
| void asm\_minus(QUAT\* q) | 减法运算 |
| void asm\_mult(QUAT\* q) | 乘法运算 |
| void asm\_div(QUAT\* q) | 除法运算 |
| void asm\_become(QUAT\* q) | 赋值语句 |
| void asm\_arrybecome(QUAT\* q) | 数组元素赋值语句 |
| void asm\_fupa(QUAT\* q) | 实际参数 |
| void asm\_morethan(QUAT\* q) | 大于 |
| void asm\_moreequ(QUAT\* q) | 大于等于 |
| void asm\_lessthan(QUAT\* q) | 小于 |
| void asm\_lessequ(QUAT\* q) | 小于等于 |
| void asm\_equ(QUAT\* q) | 等于 |
| void asm\_neq(QUAT\* q) | 不等于 |
| void asm\_jne(QUAT\* q) | 不相等跳转 |
| void asm\_jmp(QUAT\* q) | 跳转 |
| void asm\_ret(QUAT\* q) | 返回语句 |
| void asm\_scan(QUAT\* q) | 读语句 |
| void asm\_print(QUAT\* q) | 写语句 |
| void asm\_label(QUAT\* q) | 插入标签 |
| void asm\_switch(QUAT\* q) | 情况语句 |
| **出错处理：** |  |
| void error(int errNum) | 出错处理报错 |
| void errhandler(int handleNum) | 出错处理方案执行 |

### 3．调用依赖关系



### 4．符号表管理方案

#define MAX\_SYMTAB\_LEN 1024

#define MAX\_SUBSYMTAB\_LEN 1024

#define MAX\_IDXTAB\_LEN 1024

enum symTabobj {

objCONST,

objVARIABLE,

objFUNCTION,

objPARAM

};

enum symTabtyp {

typINT,

typCHAR,

typVOID,

typINTARRY,

typCHARARRY

};

// 符号表项定义

class symTab{

public:

string name;//标识符名

symTabobj obj;//标识符种类：0常量\1变量\2函数\3参数

symTabtyp typ;//标识符类型：

// 0-int\1-char\2-void\3-int[]\4-char[]

int value;//存储常量值

int addr;//地址

int paramNum;//参数个数 对数组而言为元素个数

};

// 分程序表定义

class subSymbolTable{

public:

int last;//该分程序的最后一个标识符在符号表中的位置

int lastparm;//该分程序的最后一个参数在符号表中的位置

int psize;//参数及该分程序在运行栈中的内务信息区所占的存储单元数

int ssize;//整个记录所占的总存储单元数

};

class SymbolTable{

public:

symTab element[MAX\_SYMTAB\_LEN];

int level;//当前符号表层数

int subTotal;//分程序总数

subSymbolTable subSymTab[MAX\_SUBSYMTAB\_LEN];

int indexTab[MAX\_IDXTAB\_LEN];// 分程序索引表

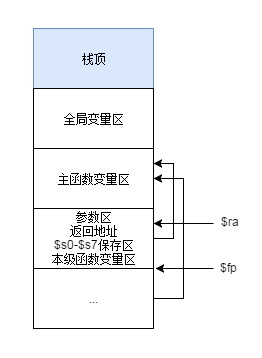
};

extern SymbolTable symtable;

### 5．存储分配方案

存储方式：采用引用计数算法进行寄存器分配

运行栈结构：



$t9:指向全局常变量区的开始，即栈底

$fp:指向自己的头部，即栈帧

preabp:指向上一级模块的头部，即上一级$fp

### 6. 解释执行程序\*

不采用解释执行方式

### 7. 四元式设计\*

四元式的每条指令具有如下形式的四个域：

**<操作符>,<操作数1>,<操作数2>,<结果>**

其中，<操作数1>和<操作数2>分别表示第一和第二个操作数；<结果>表示计算的结果，该结果通常是一个临时变量，编译程序可以为该变量分配一个寄存器或一个主存地址。

数据结构：

Class QUAT{

string opt; //操作符

string figure1; //操作数1

string figure2; //操作数2

string result; //结果

};

QUAT quat[1024];//四元式数组

|  |  |  |
| --- | --- | --- |
| 种类编号 | 四元式种类 | 含义 |
| 1 | =,2, ,t | t = 2; |
| 2 | =,a,n,b | b = a[n] |
| 3 | int, , ,a | int a; |
| 4 | const,int,4,a | const int a = 4; |
| 5 | char, ,31,a | char a[31]; |
| 6 | fupa, , ,p | p is a function parameter |
| 7 | call,f, ,a | a = f() |
| 8 | call,f, , | f() |
| 9 | +,a,b,temp | temp = (a + b); |
| 10 | jne,c, ,label | if c!=true then jump to label |
| 11 | jmp, , ,label | jump to label |
| 12 | label, , ,labx | set label |
| 13 | []=,t,i,a | a[i] = t; |
| 14 | ret, , ,(a) | return a/return |
| 15 | print,a,b,(int|char) | printf(“a”,(int|char)b) |
| 16 | scan, , ,a | scanf(a) |
| 17 | func,int, ,f | start of function int f() |
| 18 | param,int, ,a | f(int a,…) |
| 19 | end, , ,f | end of function f |
| 20 | switch,,,start | start of switch statement |
| 21 | switch,,,end | end of switch statement |

### 8. 目标代码生成方案\*

基础代码：为每种类型的中间代码设计一个机械的代码生成模版，以此翻译中间代码。

优化：采用临时寄存器池，提升目标代码的生成质量。

方法：

1. 进入基本块时，清空临时寄存器池。
2. 为当前中间代码生成目标代码时，无论临时变了还是局部变量（抑或全局变量和静态变量），如需使用临时寄存器，都可以向临时寄存器池申请。
3. 临时寄存器池接到申请后，如寄存器池中有空闲寄存器，则可将该寄存器标识为被该申请变量占用，并返回该空闲寄存器。
4. 临时寄存器池接到申请后，如寄存器池中没有空闲寄存器，则将在即将生成代码中不会被使用的寄存器写回相应的内存空间，标识该寄存器被新的变量占用，返回该空闲寄存器。
5. 在基本块结尾或者函数调用发生前，将寄存器池中所有被占用的临时寄存器写回相应的内存空间，清空临时寄存器池。

数据结构：

class MipsTemplate{

int midCodeNum;//四元式种类编号

QUAT\* quat;//四元式对象

void translate();//根据种类编号生成对应的目标代码

};

### 9. 优化方案\*

1. 划分基本块

将以+,-,\*,/,[]=,=为operation语句开头的连续四元式规约为同一基本块

1. 窥孔优化

通过窥孔优化合并常数：

将常数运算的四元式变为直接赋值操作；

消除冗余代码：

将出现在同一基本块中连续的两条两个变量相互赋值的指令删除一条；

如果跳转语句的label就紧接在跳转指令之后，则删除该跳转指令；

将乘以或除以2的倍数的操作变为左移或右移操作；

1. 消除公共子表达式：

利用课本上的算法将基本块内的中间代码化为DAG图，再转化回来得到消除公共子表达式后的中间代码。

### 10. 出错处理

**错误信息及含义**

|  |  |
| --- | --- |
| 错误编号 | 错误原因 |
| 0 | 缺少源文件 |
| 1 | 无法识别的字符 |
| 2 | 单独的一个感叹号 |
| 3 | 单引号中出现多个字符 |
| 4 | 单引号中放了非法字符 |
| 5 | 字符串未一行输完 |
| 6 | 字符串中有非法字符 |
| 7 | 无法识别的标识符 |
| 8 | 标识符重复定义 |
| 9 | 找不到main函数 |
| 10 | main函数必须是void类型 |
| 11 | 缺少类型声明 |
| 12 | 应是'(' |
| 13 | 应是'[' |
| 14 | 应是')' |
| 15 | 应是']' |
| 16 | 应是'}' |
| 17 | 应是':' |
| 18 | 应是';' |
| 19 | 只能对变量进行赋值 |
| 20 | 数字太大，最多10位数字，且绝对值不超过2147483647 |
| 21 | int常量等号后面必须是整数 |
| 22 | char常量的等号后面必须是字符 |
| 23 | 变量定义的'['后面必须是无符号整数 |
| 24 | 表达式格式不合法 |
| 25 | 有返回的函数返回语句必须有返回值 |
| 26 | 实参和对应形参类型应该相同，除非形参类型为int，实参为char |
| 27 | 应是变量 |
| 28 | 数组标识符必须跟随下标 |
| 29 | 实参个数与形参个数不等 |
| 30 | scanf或printf的参数类型不正确 |
| 31 | 表达式中的函数类型只能是int或char |
| 32 | 赋值语句中被赋值的变量必须与表达式类型相同，除非表达式是char |
| 33 | 内存溢出 |
| 34 | 应是常量 |
| 35 | 声明顺序有误 |
| 36 | 复合语句声明顺序有误 |
| 37 | 数组长度有前导零，或长度为0 |
| 38 | 应是标识符 |
| 39 | 常量只能定义为int或char |
| 40 | 应是'=' |
| 41 | 应是';'或',' |
| 42 | 应是'{' |
| 43 | 形参类型只能是int或char |
| 44 | 数组定义的中括号内应为无符号正整数 |
| 45 | 语句中变量不能单独出现而没有其他符号 |
| 46 | 因子中有不合法的内容 |
| 47 | 至少应该有一条case语句 |
| 48 | 返回语句后面只能是';'或者'(' |
| 49 | 符号表溢出 |
| 50 | 函数定义部分不能有语句 |
| 51 | 主函数的函数体之后不允许有内容 |
| 52 | 无返回的函数返回语句不能有返回值 |

**错误处理方案**

|  |  |
| --- | --- |
| 处理方案编号 | 处理方式 |
| 0 | 中止程序 |
| 1 | 不做任何处理 |
| 2 | 读下一个单词 |
| 3 | 跳到下一个分号或逗号 |
| 4 | 跳到下一个类型声明符号int\char\void |
| 5 | 跳到下一个分号 |
| 6 | 跳到下一个可识别字符并读下一个单词 |

## 三．操作说明

### 1．运行环境

代码可在windows，Linux和mac系统下编译正常执行，所生成的目标代码需要在mars虚拟机上运行。

### 2．操作步骤

1、 使用codeblocks编译本工程文件后会出现C0compiler\_jt.exe程序，此即为编译器主程序。

2、 运行编译器程序。

3、 将需要编译的文件拖入程序库中，如果路径存在空格，则需要加上引号。之后按回车键开始编译。

4、 如果编译成功，则不会显示任何提示，正常结束。

如果存在编译错误导致编译失败，则会显示错误的个数，并且会在生成的汇编程序开头放入一条compile failed阻止运行。详细的编译过程以及错误信息会在result2.txt中显示，错误信息包括错误代号、发生错误的行号、该行代码内容、错误原因。

5、 编译成功后，将会生成asm.txt，midcode.txt，result2.txt，symtable.txt四个文件，其中asm.txt为生成的目标代码，midcode.txt为生成的中间代码即四元式，result2.txt为编译过程信息如语法分析的过程记录、符号表的登录信息等，symtable.txt为最终符号表信息。

6、 打开mars创建一个新的.asm文件，将asm.txt中的目标代码复制到asm文件中，编译执行即可。

7、 需要注意的是，mars中除了单个字符以外的任何输入都必须用换行符隔开，单个字符的输入只需要直接按下要输入的字符即可。

## 四．测试报告

### 1．测试程序及测试结果

【给出提供的测试程序以及每个程序的测试结果，至少5个正确程序，5个错误程序，无需截屏】

正确程序1：

const int num1=1,num\_2=-2,num\_0=0;

const int num\_a=0;

const char ch\_a='a',ch\_0 = '0',ch\_\_='\_',ch\_star='\*',ch\_plus='-';

const char ch\_B = 'B';

int num3,num4,num\_arry1[3];

int num5;

char ch\_c,ch\_d,ch2,ch\_arry1[2];

char ch\_minus;

void v\_func\_add(int x){

x = x +1;

printf("x+1=",x);

}

void v\_func\_ret(){

printf("void return check OK");

return;

printf("failed");

}

int i\_func\_return\_1(){

return (1);

}

void main(){

const int m = 0;

int n;

v\_func\_add(m);

v\_func\_ret();

printf("return 1 check");

n =i\_func\_return\_1();

if(n==1)

printf("OK");

}

结果：

x+1=

1void return check OK

return 1 check

OK

正确程序2：

const int num1=1,num\_2=-2,num\_0=0;

const int num\_a=0;

const char ch\_a='a',ch\_0 = '0',ch\_\_='\_',ch\_star='\*',ch\_plus='-';

const char ch\_B = 'B';

int num3,num4,num\_arry1[3];

int num5;

char ch\_c,ch\_d,ch2,ch\_arry1[2];

char ch\_minus;

int i\_func\_add(int y){

return (y+1);

}

void i\_func\_calc(int a,int b){

const int c = 100;

int ans;

num3 = 100;

ans = a -b + c \* num\_2 + num3;

printf("simple expression");

printf("ans = a - b + 100 \* -2 + num3 = ",ans);

printf("complex expression:");

printf("3a-50=");

printf( (num1-num\_2)\*a + ( num\_0/b + (num\_0\* 5 - (c/i\_func\_add(num1)) ) ) );

}

char c\_func\_return\_d(){

printf("return 'd'");

return ('c'+1);

}

char c\_func\_ch\_calc(char ch,int n){

char t;

n = 1;

t = ch + n ;

printf("ch+1=",t);

return (t);

}

void main(){

const int m = 0;

int n;

n = 3;

i\_func\_calc(num\_2,n);

printf(c\_func\_return\_d());

scanf(ch\_c);

printf(c\_func\_ch\_calc(ch\_c,num3));

}

结果：

simple expression

ans = a - b + 100 \* -2 + num3 =

-105complex expression:

3a-50=

-56return 'd'

100ach+1=

b

98

正确程序3：

const int num1=1,num\_2=-2,num\_0=0;

const int num\_a=0;

const char ch\_a='a',ch\_0 = '0',ch\_\_='\_',ch\_star='\*',ch\_plus='-';

const char ch\_B = 'B';

int num3,num4,num\_arry1[3];

int num5;

char ch\_c,ch\_d,ch2,ch\_arry1[2];

char ch\_minus;

int i\_func\_add(int y){

return (y+1);

}

int i\_func\_return\_1(){

return (1);

}

void func\_if\_cmp(){

num5 = 1;

if(num5 == 2){

printf("== check failed");

}

num5 = i\_func\_add(num5);

if(num5 ==2 )

printf("== check OK");

if(num5 != 2){

printf("!= check failed");

}

num5 = i\_func\_return\_1();

if (num5 != 2){

printf("return 1 check OK");

printf("!= check OK");

}

if(num5 > 5){

;

printf("> check failed");

}else{

printf("else check OK");

printf("> check OK");

}

num5 = 5;

if(num5 <5){

printf("< check failed");

}

else printf("< check OK");

num5 =4;

if(num5 >= 5){

printf(">= check failed");

}

num5 =5;

if(num5 >= 5){

printf("5>=5 check OK");

}

num5 = 6;

if(num5 >= 5){

printf("6>=5 check OK");

}

num5=6;

if( num5 <= 5){

printf("<= check failed");

}

num5 =5;

if(num5 <=5){

printf("5<=5 check OK");

}

num5 =4;

if(num5 <=5){

printf("4<=5 check OK");

}

if(num5 <= 10){

if(num5 != 9){

if(num5 == 4){

printf("multiple if check OK");

}

else printf("multiple if check failed1");

}else printf("multiple if check failed2");

}else printf("multiple if check failed3");

}

void main(){

const int m = 0;

int n;

func\_if\_cmp();

}

结果：

== check OK

return 1 check OK

!= check OK

else check OK

> check OK

< check OK

5>=5 check OK

6>=5 check OK

5<=5 check OK

4<=5 check OK

multiple if check OK

正确程序4：

const int num1=1,num\_2=-2,num\_0=0;

const int num\_a=0;

const char ch\_a='a',ch\_0 = '0',ch\_\_='\_',ch\_star='\*',ch\_plus='-';

const char ch\_B = 'B';

int num3,num4,num\_arry1[3];

int num5;

char ch\_c,ch\_d,ch2,ch\_arry1[2];

char ch\_minus;

void func\_while(){

const int len = 3;

num5 = 0;

while(num5 < len){

printf("while for 3 times");

num5 = num5+1;

}

while(num5 > 0){

if(num5 > 0)

printf("while if num5 > 0 for 3 times");

num5 = num5-1;

}

}

void func\_switch(){

int len;

char str[10];

int cnt;

num5=1;

printf("number 1 switch check");

switch(num5){

case 1:

printf("match check OK");

case 2:

printf("check failed at ",2);

case 3:

printf("check failed at ",3);

}

num5=2;

printf("number 1 switch check");

switch(num5){

case 1:

printf("check failed at ",1);

case 2:{

printf("match check OK");

printf("multiple switch statement check OK");

}

case 3:

printf("check failed at ",3);

}

num5 = -1;

printf("default check");

switch(num5){

case 1:

printf("check failed at ",1);

case 2:

printf("match check OK");

case 3:

printf("check failed at ",3);

default:

printf("default check OK");

}

ch\_c = 'c';

printf("char switch check ");

switch(ch\_c){

case 'a':

printf("check failed at ",1);

case 'b':

printf("check failed at ",2);

case 'c':

printf("match check OK");

}

printf("check while switch:");

printf("enter length of string(1-10):");

scanf(len);

num5 = 0;

while(num5<len){

printf("enter a letter:");

scanf(ch2);

str[num5] = ch2;

num5=num5+1;

}

num5=0;

cnt=0;

while(num5<len){

switch(str[num5]){

case 'a':

cnt=cnt+1;

case 'e':

cnt=cnt+1;

case 'i':

cnt=cnt+1;

case 'o':

cnt=cnt+1;

case 'u':

cnt=cnt+1;

}

num5=num5+1;

}

printf("cnt=",cnt);

}

void main(){

const int m = 0;

int n;

func\_while();

func\_switch();

}

结果：

while for 3 times

while for 3 times

while for 3 times

while if num5 > 0 for 3 times

while if num5 > 0 for 3 times

while if num5 > 0 for 3 times

number 1 switch check

match check OK

number 1 switch check

match check OK

multiple switch statement check OK

default check

default check OK

char switch check

match check OK

check while switch:

enter length of string(1-10):

3

enter a letter:

ienter a letter:

senter a letter:

scnt=

1

正确程序5：

const int num1=1,num\_2=-2,num\_0=0;

const int num\_a=0;

const char ch\_a='a',ch\_0 = '0',ch\_\_='\_',ch\_star='\*',ch\_plus='-';

const char ch\_B = 'B';

int num3,num4,num\_arry1[3];

int num5;

char ch\_c,ch\_d,ch2,ch\_arry1[2];

char ch\_minus;

int power(int a,int b){

int t;

t = a;

if( b == 1)

return (a);

t = t\*power(a,b-1);

return (t);

}

void main(){

const int m = 0;

int n;

printf("check power:");

printf("-5^3=",power(-5,3));

}

结果：

check power:

-5^3=

-125

错误程序1：

const int b=0;

int a;

void main(){

int t;

char c;

printf("1");

t = 1;

!;

a = 2;

}

结果：

line: 9

code: !;

reason: 单独的一个感叹号

错误程序2：

const int a;

int a;

void main(){

int t;

char c;

printf("1");

t = 1;

a = 2;

}

结果：

line: 1

code: const int a;

reason: 应是'='

错误程序3：

const int a=2;

int b;

void main(){

int t;

char c;

b = 100000000000;

printf("1");

b = 1;

}

结果：

line: 6

code: b = 100000000000;

reason: 数字太大，最多10位数字，且绝对值不超过2147483647

错误程序4：

const int a=2;

int r(){

return;

}

void main(){

int t;

char c;

}

结果：

line: 3

code: return;

reason: 有返回的函数返回语句必须有返回值

错误程序5：

const int a=2;

int r(){

}

void main(){

int t;

char c;

}

结果：

line: 3

code: }

reason: 有返回的函数必须至少有一条返回语句

### 2．测试结果分析

正确程序1：覆盖全局变量、全局常量、数组、加法、无返回函数、有返回函数、空返回语句、无返回语句、函数调用、写语句输出字符串、条件if语句。

正确程序2：覆盖返回表达式、加法、减法、乘法、除法、写语句输出字符串+表达式、函数返回字符表达式、多个参数函数调用、读语句、写语句输出函数返回值。

正确程序3：覆盖条件if-else语句、关系表达式（大于、大于等于、等于、小于、小于等于、不等于）、条件语句嵌套。

正确程序4：覆盖循环while语句、情况switch语句、while-switch嵌套、数组元素赋值语句。

正确程序5：覆盖函数递归。

## 五．总结感想

通过本学期课程设计的学习实践，我感到受益匪浅。对我而言，本次课程设计，是非常难得的一次独立经历从设计到编写超过四千行的程序全过程的机会，从这个过程中，我不仅加深和巩固了编译原理相关知识的认知与体会，更训练了我严谨的思维，以及模块化构建一个较大程序的经验。通过高级文法C0的仔细解读，使我对于C语言编程也有了更加细致、具体的认知，收获颇多。

从11月初开始，选择完文法后拿到了我的C0文法，第一次作业是对文法进行解读，一开始接触还挺迷茫的，参考着老师的例子写了一篇，后来在具体编码过程中才发现自己当时做的实在太片面了，有好多的细节藏在文法里，一不留神就会导致最后的小型编译系统存在bug。

整个11月都在课程设计编码中度过，平均3天一个的ddl让我每天的时间过得相当充实与规律。词法分析、语法语义分析、符号表管理、运行栈实现、中间代码生成、目标代码生成、出错处理，每个模块各自都有着自己的特点，在编写时要将注意力集中在不同的方面，有时要关注每一个字符位置读取的细节，有时要把眼光提到全局的高度仔细规划，前期的一个细节设计不足，就可能导致后期调试时花费大量的时间精力。

考核的前三次测试程序都顺利地通过了，其中包含了我长久的心血，因而感到很有成就感。眼下学期末尾，在考试复习中完成了最终的程序调试与完善，希望期末的最终考核也可以顺利通过~