《编译技术》课程设计文 档

学号：\_\_\_\_\_16061175\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_刘卫\_\_\_\_\_\_\_\_\_

2018年11月16日

**一．需求说明**

**1．文法说明**

### （1）获取的文法

获取的文法为扩充的C0文法，含有while循环语句和switch语句。

具体文法如下：

＜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＞｝| 0

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜类型标识符＞ ::= int | char

＜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＞

＜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

（5）数组的下标从0开始

### （2）对文法的改写和扩充

为了简化编译器的设计，对原文法中标黄的部分进行了部分改写和简化，主要改写部分如下：

＜声明头部＞ ::= int＜标识符＞ | char＜标识符＞

改为：

＜声明头部＞ ::= ＜类型标识符＞＜标识符＞

＜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＞;}

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

改写为：

＜程序＞ ::= {const＜常量定义＞;} {＜变量定义＞;}{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

改写为：

＜复合语句＞ ::= { const＜常量定义＞;} {＜变量定义＞;}＜语句列＞

＜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＞']' )}

改写为：

＜变量定义＞ ::= ＜声明头部＞[ '['＜无符号整数＞']' ]{,(＜标识符＞['['＜无符号整数＞']' ]}

又因为<变量定义>和<有返回值的函数定义>的开头都是＜声明头部＞，初始符号都含有int或char，所以在自顶向下的递归分析法中可能出现回溯，为了避免这种情况，应采取预读的方法，进行<程序>的分析时，如果遇到＜声明头部＞，应该先分析完＜声明头部＞，再根据下一个单词是如果是‘(’则进入<有返回值函数定义>对应的函数进行分析，否则进入<变量定义>对应的函数进行分析，同时要注意二者的顺序问题，出现过<有返回值函数定义>后，不能再出现<变量定义>。

**2．目标代码说明**

生成的目标代码为MIPS32汇编指令集，可在MARS上直接运行，具体指令如下：

1. 伪指令：

（1）.data address(address参数可选) 用来表明接下来的若干条指令是用于数据存储的，无参数的情况下则从汇编器默认的数据段的起始位置开始存储，有参数的话会从参数指定的地址开始存储，数据存储超过数据段的范围会报错。用于预先存储数据的伪指令必须紧跟着该指令，并且数据是按照伪指令的顺序有序存储的。

（2）.text address(address参数可选) 用来表明接下来的若干条指令是程序的代码，无参数的情况下则从汇编器默认的代码段的起始位置开始存储，有参数的话会从参数的地址开始存储，指令超过代码段的范围会报错。该指令后面紧跟着的就是程序代码。

（3）addr: .space n 申请n个字节未初始化的内存空间，类似于高级程序语言中的数组声明。这段数据的初始地址保存在标签addr中。

（4）addr: .word data1,data2 … 在内存数据段中以字为单位连续存储数据data1,data2,…。这段数据的初始地址保存在标签addr中。

（5）addr: .asciiz "sdfsdfefvrwsdw" 以字节为单位存储字符串，末尾以NULL('\0')结尾。这个字符串在内存数据区的初始地址保存在标签addr中。

2. 指令（不区分大小写）：

add：符号加 add $t1,$t2,$t3

addi：符号加立即数 addi $t1,$t2,100

beq：相等时转移 beq $t1,$t2,label

bgez：大于等于0时转移bgez $t1,label

bgtz：大于0时转移 bgtz $t1,label

blez：小于等于0时转移 blez $t1,label

bltz：小于0时转移 bltz $t1,label

bne：不等于时转移 bne $t1,$t2,label

div：符号除 div $t1,$t2,$t3（扩展指令）

j：跳转 j target

jal：跳转并链接 jal taget

jalr：跳转到寄存器并链接 jalr $t1,$t2

jr：跳转至寄存器 jr $1

lb：加载字节 lb $t1,-100($t2)

lui：立即数加载至高位 lui $t1,100

lw：加载字 lw $t1,-100($t2)

mflo：读lo寄存器 mflo $t1

mul：符号乘 $t1,$t2,$t3

sb：存储字节 sb $t1,-100($t2)

slt：小于置1(有符号) slt $t1,$t2,$t3

slti：小于立即数置1(有符号) slt $t1,$t2,-100

sub：符号减 sub $t1,$t2,$t3

sw：存储字 sw $t1,-100($t2)

syscall：系统调用

nop：空指令

la 加载地址 la $t1, 0x20000000（扩展指令）

li 加载立即数 li $t1,100000（扩展指令）

move复制寄存器的值 move $t1,$t2（扩展指令）

**3. 优化方案\***

1. 基本块内优化

（1）利用代数性质（代数变换）：编译时完成常量表达式的计算，char类型向int类型的自动转换。下标变量引用时，其地址计算的一部分工作可在编译时预先做好（运行时只需计算“可变部分”即可）。用一种需要较少执行时间的运算代替另一种运算，以减少运行时的运算强度时、空开销。

（2）常数合并和传播

（3）删除冗余代码

（4）消除公共子表达式，通过DAG图

2. 全局优化

（1）数据流分析——到达定义分析

（2）数据流分析——活跃变量分析

（3）构建冲突图

3. 全局寄存器分配优化

（1）引用计数算法

（2）着色图算法

**二．详细设计**

**1．程序结构**

程序的总体结构类似于Pascals编译器，采用面向过程的编程方法，从main函数开始执行，先调用语法分析函数block，语法分析block根据不同的语法成分（函数，常量声明，变量声明等）调用相应的分析处理函数，对应的函数再调用词法函数insymbal，进行语法分析的同时进行一系列的优化操作，通过emit函数生成中间代码四元式，再由中间代码进行优化，最后由genmips函数生成目标代码mips汇编指令。

**2．类/方法/函数功能**

编译器采用面向过程的编程方法，不使用类和对象，只有函数。各部分的函数及主要功能和算法如下：

（1）词法分析部分

void setup(file \*in) 进行词法分析前的初始化工作，初始化相关变量，设置源文件指针，初始化保留字表和特殊符号表。

void nextch(); 从源文件中获取下一个字符，并记录其行号和列号。

void insymbol() 词法分析主函数，分析并记录当前单词符号的类型和相关的值，分析到不合法的字符时进行报错。

void printsymbol() 当前单词分析结束且未发现错误时在控制台输出词法分析结果，便于检查。

（2）语法分析部分

void block(); 语法分析的主要函数，处理分程序的递归子程序

void enter(string id, objtyp k); 将名为id的符号存入符号表

int loc(string id); 查找标识符id在符号表中的位置，未找到时返回0

void constdef(); 处理常量定义的子程序，将常量名及其相应信息填入符号表

void vardef(); 处理变量定义的子程序，将变量名及其相应信息填入符号表

void paralist(); 处理有返回值或无返回值的函数定义中的形参数表的子程序，将形参及其有关信息登录到符号表中。

void funcdef(); 处理函数定义的子程序，将函数名，返回值类型等填入符号表，调用参数表分析子程序paralist()分析参数，调用复合语句分析程序 compoundstatement()分析处理下一层分程序

void compoundstatement() 处理复合语句的子程序，依次调用相应子程序处理常量定义，变量定义和语句列。

void expression(); 分析处理简单表达式的子程序

void term(); 分析处理项的子程序

void factor(); 分析处理因子的子程序，可能会继续调用简单表达式的子程序，形成间接递归调用。

void funccall(); 分析处理有返回值或无返回值的函数调用的子程序

void statement(); 处理语句的子程序，分析处理各种语句

void assignment(); 分析处理赋值语句的子程序

void ifstatement(); 分析处理if语句的子程序

void whilestatement(); 分析处理while循环语句的子程序

void switchstate(); 分析处理switch的子程序

void onecase(); 分析处理case语句中的一个分支的子程序

void statementlist(); 分析处理语句列的子程序

void printfstatement(); 处理printf调用语句的子程序

void scanfstatement(); 处理scanf调用语句的子程序

void returnstatement(); 处理return语句的子程序

（3）中间代码生成部分

void emit(int op, int a, int b, int c); 根据参数中的操作符和操作数生成四元式。

void printmidcode(); 输出优化前的中间代码

void printoptmidcode(); 输出优化后的中间代码

（4）代码优化部分

void constfolding(); 常量合并和替换

void deldeadcode(); 删除冗余的label，跳转以及赋值等冗余代码

void delsubexp(); 消除公共子表达式，通过DAG图算法

void dataflowanal(); 数据流分析，通过到达定义分析或活跃变量分析建立冲突图

void regalloc(); 全局寄存器分配优化，通过引用计数算法或着色图算法

（5）目标代码生成部分

void genasm(); 将中间代码转化为汇编代码，目标代码生成的主函数

void genconstdefasm(); 生成常量定义的汇编代码

void genvardefasm(); 生成变量定义的汇编代码

void genfuncdefasm(); 生成函数定义的汇编代码

void genassignasm(); 生成赋值语句的汇编代码

void setlab(int index); 在指定位置设置label

void genifasm(); 生成条件语句的汇编代码

void genswitchasm(); 生成分支语句的汇编代码

void genwhileasm(); 生成循环语句的汇编代码

void gencallasm(); 生成函数调用语句的汇编代码

void genscanfasm(); 生成scanf语句的汇编代码

void genprintfasm(); 生成printf语句的汇编代码

void genreturnasm(); 生成return语句的汇编代码

void printasm(); 输出生成的目标代码

（6）错误处理部分

void error(int a); 根据编号输出当前遇到的错误并作相应的处理

**3．调用依赖关系**

main函数先调用setup函数进行初始化，然后调用一次insymbol函数分析出一个正确的单词，然后调用block函数进行递归下降语法分析，分析完成后调用printmidcode函数输出生成的中间代码，然后依次调用constfolding， constreplace，deldeadcode， delsubexp，dataflowanal， regalloc等函数进行代码优化，最后调用genasm()函数生成汇编代码，调用printm函数输出生成的汇编代码。

insymbol函数调用nextch函数，printsymbol函数

block函数调用insymbol函数，constdef函数，vardef函数，funcdef函数

constdef函数调用insymbol函数，enter函数

vardef函数调用insymbol函数，enter函数

funcdef函数调用insymbol函数，enter函数， paralist函数，compoundstatement函数

compoundstatement函数调用insymbol函数，constdef函数，vardef函数，statementlist函数

statementlist函数调用调用insymbol函数，statement函数

statement函数调用insymbol，funccall，assignment，ifstatement，whilestatement，switchstate，statementlist，printfstatement，scanfstatement，returnstatement等函数

funccall函数调用insymbol，loc函数。

assignment函数调用insymbol，loc函数，expression函数，

ifstatement，whilestatement调用insymbol，expression，statement函数

switchstate函数调用insymbol，onecase函数

onecase调用insymbol，statement函数

printfstatement，returnstatement调用insymbol，expression函数

scanf函数调用insymbol，loc函数

expression函数调用insymbol，term函数

tem函数调用insymbol，factor函数

factor函数调用insymbol，expression，funccall，loc函数

以上函数在执行分析的过程中发现语法或语义错误时都会调用error函数进行报错和处理。

**4．符号表管理方案**

（1）符号表数据结构

struct tabitem{//符号表

char name[max\_len];

int link; //指向同一函数中上一个标识符在tab表中的位置

int obj; // 标识符种类：常量、变量、函数等

int typ; // 标识符类型：int型、char型、int数组型、char数组型、void型等

int ref; //函数名：指向其在btab表中的指针值

int arrcnt; //数组元素个数

int adr; // 变量、形参：在运行栈s中的相对地址；过程名、函数名：相应目标代码的入口地址；整型常量：对应数值；

int lev; // 标识符所在的静态层次，全局常量、变量或函数为0，局部常量、变量为1

} tab[max];

int t; //符号表索引

struct atabitem{

int last, lastpar, psize, vsize;

/\*

last指向函数最后一个标识符在tab表中的位置；

lastpar指向函数的最后一个参数在tab表中的位置；

psize为参数及内务信息区所占的存储单元数；

vsize局部常量、变量、参数及内务信息区所占的存储单元总数。

\*/

}btab[max];

int b; //分程序表索引，btab[0]默认保存全局常量，变量，函数名的信息。

vector<string> stab; //字符串常量表

int sx; //字符串常量表索引

（2）符号表的管理算法，

本程序计划采用无序的符号表组织方式，符号表的表项按照插入的先后顺序排列，使用enter函数将对应的标识符登记到符号表中。使用loc函数查找某个标识符是否出现在符号表中，如果存在则返回对应索引，不存在时返回0（规定tab[0]不保存有效的标识符）。

**5．存储分配方案**

（1）全局变量和全局常量的存储组织

编译时将全局变量和全局常量进行统一分配，都保存在mips中以.data开始的内存地址中，用label记录对应变量的地址。对于字符串常量，用伪指令.asciiz来实现字符串常量的存储分配，局部变量，常量则保存在运行栈中。

（2）运行栈的设计

运行时每个函数调用的运行栈的基本结构为

|  |
| --- |
| 局部变量区 |
| 参数区 |
| 寄存器栈（保存调用者的寄存器的情况） |
| 上一函数的帧指针，即$fp寄存器 |
| 上一函数的返回地址，即$ra寄存器 |

**6. 解释执行程序\***

编译器生成的目标代码为mips32汇编代码，不需要解释执行。

**7. 四元式设计\***

生成四元式的中间代码时，应先建立中间代码表midcodetab保存每个四元式，表中每个元素应含有四个成员，分别保存四元式的每一元。具体的四元式见下表。

|  |  |  |
| --- | --- | --- |
| 类型 | 源代码 | 中间代码（四元式） |
| 函数定义 | int foo( int a, int b, int c, int d) | int foo()  para int a  para int b  para int c  para int d |
| 有返回值函数调用 | i = tar(x,y) | push x  push y  call tar  i = RET |
| 无返回值函数调用 | tar(x, y) | push x  push y  call tar |
| scanf语句 | scanf(x,y) | push x  push y  call scanf |
| printf语句 | printf(x) | push x  call printf |
| 函数返回 | return (x) | ret x |
| 变量定义 | int i, j; | var int i  var int j  （符号表信息输出，程序中可不生成真正的中间代码） |
| 常量定义 | const int c = 10 | const int c = 10 |
| 表达式 | x = a \* (b + c) | t1 = b + c  t2 = a \* t1  x = t2 |
| 条件判断 | x == y | x == y |
| 条件或无条件跳转 |  | GOTO LABEL1 //无条件跳转到LABEL1  BNZ LABEL1 //满足条件跳转到LABEL1  BZ LABEL1 //不满足条件跳转到LABEL1 |
| 带标号语句（？） | Label\_1: x = a + b | Label\_1 :  x = a + b |
| 数组赋值或取值 | a[i] = b \* c[j] | t1 = c[j]  t2 = b \* t1  a[i] = t2 |

**8. 目标代码生成方案\***

根据优化前或优化后的中间代码进行目标代码生成时，需结合符号表对标识符进行局部变量、常量或全局变量、常量的判断，进而分配到不同的存储空间：全局变量和常量用伪指令.word以及.space保存在mips中以.data开始的数据区内存地址中，用label记录对应变量的地址（对于字符串常量，用伪指令.asciiz来实现字符串常量的存储分配，用label记录对应变量的地址）；局部变量，常量则保存在运行栈中。

生成一般四元式的目标代码时，需先生成对应的取值语句，对变量进行取值时应该先查符号表，先从局部变量对应的符号表进行查找，若找到则用lw或lb指令从对应地址进行取值，否则查全局变量、常量表，在全局变量、常量区对应的数据区进行取值。然后根据操作符op的不同生成不同的运算类指令。最后还要生成保存运算结果的值的语句，存值过程可类比取值过程。

对于函数定义的生成，需要将函数名以标签的形式插入到函数汇编代码的入口地址。

**9. 优化方案\***

1. 基本块内优化（先要划分基本块）

（1）利用代数性质（代数变换）：编译时完成常量表达式的计算，char类型向int类型的自动转换。下标变量引用时，其地址计算的一部分工作可在编译时预先做好（运行时只需计算“可变部分”即可）。用一种需要较少执行时间的运算代替另一种运算，以减少运行时的运算强度时、空开销。

（2）常数合并和传播，通过窥孔优化进行常数合并操作，减少查表时间。

（3）删除冗余代码。

（4）消除公共子表达式，通过DAG图和采用启发性算法实现公共子表达式的删除并重新导出中间代码。

算法1 划分基本块

输入：四元式序列

输出：基本块列表，每个四元式仅出现在一个基本块中

方法：

1、首先确定入口语句（每个基本块的第一条语句）的集合

规则1：整个语句序列的第一条语句属于入口语句

规则2：任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句

规则3：紧跟在跳转语句之后的第一条语句属于入口语句

2、每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块

算法2 构建DAG图的算法-消除公共子表达式

输入：基本块内的中间代码序列

输出：完成局部公共子表达式删除后的DAG图

方法：

1.首先建立节点表，该表记录了变量名和常量值，以及它们当前所对应的DAG图中节点的序号。该表初始状态为空。

2.从第一条中间代码开始，按照以下规则建立DAG图。

3.对于形如z = x op y的中间代码，其中z为记录计算结果的变量名，x为左操作数，y为右操作数，op为操作符：首先在节点表中寻找x，如果找到，记录下x当前所对应的节点号i；如果未找到，在DAG图中新建一个叶节点，假设其节点号仍为i，标记为x（如x为变量名，该标记更改为x0）；在节点表中增加新的一项(x, i)，表明二者之间的对应关系。右操作数y与x同理，假设其对应节点号为j。

算法3 从DAG导出中间代码的启发式算法

输入：DAG图

输出：中间代码序列

方法：

1.初始化一个放置DAG图中间结点的队列。

2.如果DAG图中还有中间节点未进入队列，则执行步骤3，否则执行步骤5

3.选取一个尚未进入队列，但其所有父节点均已进入队列的中间节点n，将其加入队列；或选取没有父节点的中间节点，将其加入队列

4.如果n的最左子节点符合步骤3的条件，将其加入队列；并沿着当前节点的最左边，循环访问其最左子节点，最左子节点的最左子节点等，将符合步骤3条件的中间节点依次加入队列；如果出现不符合步骤3条件的最左子节点，执行步骤2

5.将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列

2. 全局优化

（1）数据流分析——到达定义分析

算法4 基本块的到达定义数据流分析

输入：程序流图，且基本块的kill集合和gen集合已经计算完毕

输出：每个基本块入口和出口处的in和out集合，即in[B]和out[B]

方法：

1.将包括代表流图出口基本块Bexit的所有基本块的out集合，初始化为空集。

2.根据方程in[B] = ∪B的前驱基本块Pout[P]，out[B] = gen[B] ∪( in[B] –kill[B] )，为每个基本块B依次计算集合in[B]和out[B]。如果某个基本块计算得到的out[B]与该基本块此前计算得出的out[B]不同，则循环执行步骤2，直到所有基本块的out[B]集合不再产生变化为止。

（2）数据流分析——活跃变量分析

算法5 基本块的活跃变量数据流分析

输入：程序流图，且基本块的use集合和def集合已经计算完毕

输出：每个基本块入口和出口处的in和out集合，即in[B]和out[B]

方法：

1.将包括代表流图出口基本块Bexit在内的所有基本块的in集合，初始化为空集。

2.根据方程out[B] = ∪B的后继基本块Pin[P]，in[B] = use[B] ∪(out[B] –def[B])，为每个基本块B依次计算集合out[B]和in[B]。如果计算得到某个基本块的in[B]与此前计算得出的该基本块in[B]不同，则循环执行步骤2，直到所有基本块的in[B]集合不再产生变化为止。

（3）构建冲突图：通过活跃变量分析构建（精度不太高的）冲突图

3. 全局寄存器分配

（1）引用计数算法：通过统计变量在函数内被引用的次数，并根据被引用的特点赋予不同的权重，最终为每个变量计算出一个唯一的权值，根据权值的大小排序，将全局寄存器依次分配给权值最大的变量。如果有N个全局寄存器可供分配，则前N个变量拥有全局寄存器，其余变量在程序运行栈（活动记录）分配存贮单元。

（2）着色图算法：通过构建变量之间的冲突图，在图上应用着色算法，将不同的全局寄存器分配给有冲突的变量。

算法6 一种启发式图着色算法

输入：冲突图G，寄存器数目为K

输出：染好色的冲突图

1. 找到第一个连接边数目小于K的结点，将它从图G中移走，形成图G’。
2. 重复步骤1，直到无法再从G’中移走结点。
3. 在图中选取适当的结点，将它记录为“不分配全局寄存器”的结点，并从图中移走。
4. 重复步骤1~步骤3，直到图中仅剩余1个结点。
5. 给剩余的最后一个结点选取一种颜色，然后按照结点被移走的顺序，反向将结点和边添加进去，并依次给新加入的结点选取颜色。（保证有链接边的结点着不同的颜色）。

**10. 出错处理**

错误处理有可能在词法分析或语法分析中遇到，一旦在词法分析或语法分析的程序中检测到错误，就调用void error(int n)进行处理，其中n用来代表错误的类型。错误编号及对应的错误类型如下：

|  |  |  |
| --- | --- | --- |
| 错误编号 | 错误类型 | 处理方案 |
| 1 | 字符常量中含不合法的字符 | 报错后继续分析 |
| 2 | 字符常量缺少右单引号 | 跳过该字符常量继续分析 |
| 3 | 字符串缺少右双引号 | 一直读到行末换行符或读到制表符等非法字符时停止 |
| 4 | 源程序中冒号后不与等号相接 | 跳过该冒号继续分析 |
| 5 | 源程序中含有非法字符 | 跳过该字符继续分析 |
| 6 | 引用未定义的标识符 |  |
| 7 | 标识符出现重定义 |  |
| 8 | 程序结构错误（如先变量定义，后常量定义等） |  |
| 9 | 缺少分号 |  |
| 10 | 缺少冒号 |  |
| 11 | 缺少逗号 |  |
| 12 | 缺少加法或乘法运算符 |  |
| 13 | 缺少比较运算符 |  |
| 14 | 缺少左小括号 |  |
| 15 | 缺少左中括号 |  |
| 16 | 缺少左大括号 |  |
| 17 | 缺少右小括号 |  |
| 18 | 缺少右中括号 |  |
| 19 | 缺少右大括号 |  |
| 20 | 缺少标识符 |  |
| 21 | 函数调用时参数数量或类型不一致 |  |
| 22 | 赋值符号左右类型不一致 |  |
| 23 | 赋值符号左侧不为变量或数组成员 |  |
| 24 | 缺少main函数 |  |
| 25 | 有返回值的函数缺少return语句 |  |
| 26 | 不合法输入 |  |
| 27 | 返回值类型不一致 |  |
| 28 | 使用无返回值的函数调用语句作为表达式参与运算 |  |
| 29 | 常量使用前未初始化 |  |
| 30 | 变量使用前未初始化 |  |
| 31 | 数组越界 |  |
| 32 | 数组元素个数不大于0 |  |
| 33 | 除0错误 |  |
| 34 | case语句后不为常量 |  |

**三．操作说明**

**1．运行环境**

【说明搭建运行环境的步骤】

**2．操作步骤**

【详细说明操作步骤】

**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】

**2．测试结果分析**

【说明上述测试程序对语法成分的覆盖情况】

**五．总结感想**

【说明在完成课程设计中的收获、认识和感想】

注：【】内的文字为文档模板说明，完成的作业中需去掉。

标\*的章节需根据题目的难度进行取舍。