《编译技术》课程设计文 档

学号：\_\_\_\_\_\_\_\_\_16231137\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_\_\_李航宇\_\_\_\_\_\_\_\_\_\_\_

2019年 1 月 4 日

## 一．需求说明

### 1．文法说明

#### 1.1 具体文法展示

抽到的文法是do-while + switch型的文法

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜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＞ ::= ＜数字＞｛＜数字＞｝

＜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

＜类型标识符＞ ::= int | char

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数表＞‘)’ ‘{’＜复合语句＞‘}’|＜声明头部＞‘{’＜复合语句＞‘}’ //第一种选择为有参数的情况，第二种选择为无参数的情况

＜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＞‘}’//第一种选择为有参数的情况，第二种选择为无参数的情况

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

＜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＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’

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

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

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

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞else <语句>

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

＜循环语句＞ ::= do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞ ::= ＜整数＞|＜字符＞

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

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

＜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）标识符不区分大小写字母

（3）写语句中，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

（4）数组的下标从0开始

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

#### 1.2 文法解读

**1.2.1程序调用部分：**

＜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＞

分析及限定条件：程序进入入口，可以选择性对全局的常量与变量进行说明。之后可选择性定义0个或多个函数，包括有无返回值类型用于主函数调用，最后进入主函数部分。

＜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＞}

分析及限定条件：用于刚进入程序的全局常量量或者函数内部的常量说明与定义。“const＜常量定义＞”可以定义一次或多次，中间以分号隔开；每一个常量定义只能定义一种类型（int或char）的常量，定义两种类型需要两个常量定义。常量定义中针对一种类型（例如int）可以定义一个或多个具有标识符的int型常量，形如：const int a1 = 2, a2 = 4。const int a1, a2 = 3非法，常量定义赋值号的右侧只能为整数或字符，不可以出现表达式。

样例：const int a1 = 3, a2 = +0; const char c1 = ‘a’, c2 = ‘+’;

＜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

＜类型标识符＞ ::= int | char

分析及限定条件：用于刚进入程序的全局变量或者函数内部的变量说明与定义。<变量定义>可以定义一次或多次，中间以逗号隔开；每一个常量定义只能定义一种类型(int或char)，定义两种类型需要两个变量定义。在一次变量定义中针对一种类型的变量可以定义一个或多个。对于int或char类型的单个变量（整数或字符），给出对应的标识符即可，但对于int或char类型的数组变量需要给出确定数组元素个数（上界-1），以数字表示。变量定义中不可以进行赋值！标识符只可以以\_,a-z,A-Z开头，由这些和0-9数字构成，无符号整数支持’+/-007’写法

样例：int a1 = 3, a2 = +0, a3 = array[3\*5]; char c1 = ‘a’, c2 = ‘+’, char c3 = carray[4];

＜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＞‘}’

//第一种选择为有参数的情况，第二种选择为无参数的情况

＜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＞‘}’

//第一种选择为有参数的情况，第二种选择为无参数的情况

＜声明头部＞ ::= int＜标识符＞|char＜标识符＞

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

分析及限定条件：用于定义所有的主程序会调用的函数类型。有返回值的函数的声明头部代表的是返回值类型，若是没有返回值的函数需要用保留字void占位，标识符代表函数定义的名字。有参数表的函数参数表以int a或char c方式定义参数变量，一次只能定义一种类型的一个变量，以逗号隔开。之后的<复合语句>进行函数内部的常量与变量的说明定义，之后进入各个语句进行语句的分析与执行。

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

分析及限定条件：主函数的定义，以”void main()”开头，之后进入<复合语句>部分进行。

程序调用部分样例：

const int const\_a=+1, const\_b=-210;  
 const char const\_c='+',const\_d='\*',const\_e='a',const\_f='1';  
  
 int a[12],b,c[10];

int fun1(int a1, char c1){

return (c1+a1);

}

void fun2(int a1, char c1){

return;

}

void main(){

const con1 = 1;

int in1;

char in2;

scanf(in1,in2);

if(fun1(in1,in2)>8){

{

return;

}

}

else

printf(con1);

}

**1.2.2表达式部分：**

＜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＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’

分析与限定条件：用于构建表达式的基础项的说明。

首先<表达式>是由<项>和<加法运算符>构成，项前面可带有正负号，正号也可以！。表达式至少有一项也可以有多项。表达式最后一定有一个整型的结果值。即使没有<常量>项，表达式也可以用标识符调用const常量，表达式中无字符串，有返回值的函数调用语句只能在表达式中调用。

<项>中至少一个因子也可以有多个。因子和因子之间用’\* /’连接。

<因子>可看作执行数字运算的最小单元。可以是单个的标识符代表单个变量，也可以是标识符与[<表达式>]的结合代表数组的特定一项，也可以是常量的整数或字符，函数返回值，同时还可以直接是表达式，但是这个表达式一定要用一个或多个括号包上，最外层的表达式（即调用处）的表达式也可以通过<项>::=’(‘<表达式>’)’获取）。

样例：

<因子>：c = d c = a[2\*a1] c = 3 c = ‘+’ c = func(a) c = (3 / b)

<表达式>：a = -(func(\_00b1)\*b[20-c] / (3+’t’)) + fibs(5\*b[4])

＜无符号整数＞ ::= ＜数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜１｜．．．｜９

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

分析与限定条件：对于整数部分，可以以正负号开头也可以以整数（包括0）开头；对于字母部分，可用字母为下划线以及大小写英文字母。标识符必须以字母开头，之后才是数字与字母均可，字母开头可以以’\_’开头。字符可以被解释为引号包裹的加法乘法运算符以及字母和数字，一个字符中只能有一个ascii码意义上的字符，不能含多个，<比较运算符>没有ascii码意义的字符值！想出现”-+4”或者”+-3”等形式只有在<表达式>::=-<因子> -> -+<无符号整数> ->-+4。

样例：

<整数>：-007 +000 -+4

<标识符>：\_ \_00Tv4w WC0 dc9

**1.2.3语句部分：**

＜语句列＞ ::=｛＜语句＞｝

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜<情况语句>|＜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＞;

分析与限定条件：语句列中语句可以为空或多个，其中空可以为{}或{;}，同时语句中也可以调用语句列。语句中会调用各个不同的语句执行不同的功能。其中<条件语句>，<循环语句>和<情况语句>不需要在<语句>的维度填’；’

样例：

<语句列>：{} {;}

{{return (0);}}

＜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＞else <语句>

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

＜循环语句＞ ::= do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞ ::= ＜整数＞|＜字符＞

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

＜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＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

＜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＞‘)’]

分析与限定条件：

赋值语句：标识符代表接受值的变量或数组变量，等号右侧给出一个表达式的返回值。

条件语句：else不可缺省，可以出现if(表达式){}else…与if(表达式){;}else…或if(表达式){{}} else…。语句的大括号”{}”可加可不加，但只有单条语句时候才可以不加。条件中只允许表达式单目或二目条件运算，<条件>最终返回一个bool逻辑值。

循环语句：注意格式即可。

情况语句：对特定变量进行值的分支判断，进入不同的处理函数中进行处理。必须有default，可嵌套语句；情况表中至少有一个case语句，变量为单个的整数或字符。

返回值函数调用语句：分为有形参和无形参的情况。有形参的需要在<标识符>后面的()中以值参数表的形式给出参数；无形参的情况不用写调用符号()!

值参数表：传入函数的参数说明；必须非空，至少有表达式中的一个因子

读语句：按定义好的标识符进行变量读入，不允许直接读入数组！

写语句：可以只写字符串或者表达式的值，也可以同时写二者，但最多只能写两个且一个为字符串，一个为表达式的值，顺序为字符串在前。

返回语句：return（用于void函数）或者return (<表达式>)（用于有返回值的函数），一定要加外小括号！

样例：

/\*程序头部说明及定义略\*/

/\*进入主函数\*/

{

int a1 , a2 , a3,i,a4[5];

char c1 , c2 , c3;

scanf(a1,a2,c1,c2,c3);

a3 = a2+1;

i= 0; //赋值

do{

a4[i] = i;

if(i == 3){

c1 = ‘a’;

c1 = c1 + i;

}

else

c2 = ‘a’+10;

} while(i<a3)

switch(c1){

case ‘d’: printf(a1);

default: printf(a2);

}

printf(a1,a2,a3);

return;

}

### 2．目标代码说明

生成32位mips汇编代码，包含伪指令，以下给出所有使用的mips代码，具体的作用不再赘述。

|  |  |
| --- | --- |
| add/addu $t0, $t1, $t2 | 寄存器加运算 |
| addi/addiu $t0, $t1, 100 | 寄存器和立即数加运算 |
| sub $t0, $t1, $t2 | 寄存器减运算 |
| li $t0, 100 | 加载立即数到寄存器 |
| sw $t0, 0($t1) | 寄存器值保存到内存中 |
| lw $t0, 0($t1) | 内存上的值加载到寄存器中 |
| beq, bne, bge, bgt, ble, blt $t1, $t2, TARGET | 条件跳转 |
| j jal jr | 跳转 |
| mul $t0, $t1, $t2 | 乘法 |
| div $t0, $t1 mflo $t2 | 除法 |
| move $t1, $t2 | 赋值运算 |
| sll $t1, $t0, $t2 | 数组偏移计算 |
| syscall | 系统调用 |
| la $a0, 0x20000000 | 加载地址 |

### 3. 优化方案\*

#### 3.1 消除局部公共子表达式(dag图)

按照书上的划分基本块的方法进行基本块的划分，在进行dag公共子表达式的删除的时候由于书上没有将push，scanf，printf等等函数考虑进去，这里我做了一些修改使得这些函数可以加入dag里面进行公共子表达式的删除，但是导出的时候就是按照中间代码的先后顺序进行的导出。具体细节见优化部分。

#### 3.2 常数合并

在语法分析的时候将所有的字面常量与用户定义常量进行常量折叠，即在表达式部分将所有可以检测到的常量进行运算，将可以通过四则运算算出来的常量值自factor向上经过item最终返回到expr上进行常数合并与折叠。

#### 3.3 活跃变量分析（活跃变量分析构建冲突图，染色法分配寄存器）

按照书上进行活跃变量的全局数据流分析，染色法分配全局寄存器。

#### 3.4 引用计数法

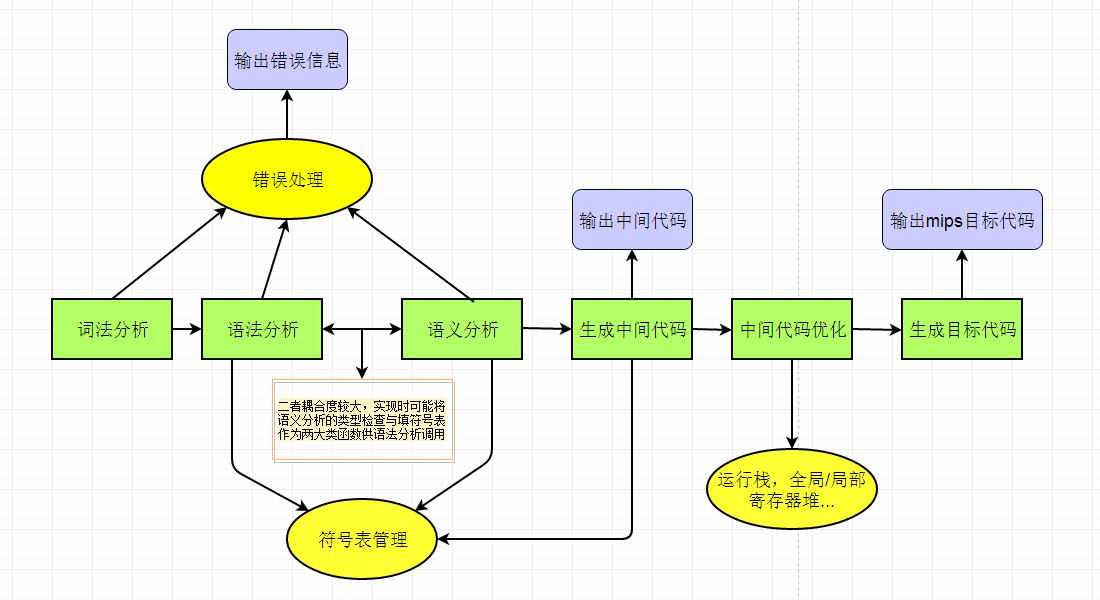
按照书上的引用计数法对全局和局部变量进行分析，最终并没有使用这种方法。

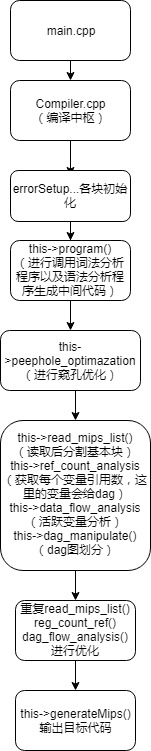
#### 3.5 窥孔优化

对连续的label语句，跳转结果的后面还是跳转，做了跳转的删除，同时对跳转时可以判断条件真假的情况将branch改为了j或顺序执行。

## 二．详细设计

### 1．程序结构





### 2．类/方法/函数功能

#### 2.1 错误处理 error

主要函数及功能说明：

void errorSetup(); // 错误处理的setup

void getCurrentline(); //找到出错处在编译文件的行数，将出错行代码放入wrongString中

void errorManipulate(string errormsg); //将出错信息连带出错行打印

void skip(const tokenType\* kind, int n); //跳读集

void skip(tokenType kind); //跳读单个符号

其中暴露给其它函数的接口就是skip和errorManipulate，skip负责跳读而errorManipulate负责所有的错误处理。用户可以向里面输入想要输出的错误名称而后进行输出。

#### 2.2 词法分析 Lexical

词法分析器，完成从文件读取符号并分类的功能，主要使用getSym的功能。其中用到的strToken返回当前读取的单词的值，curToken返回读取到单词的类别，若是数字则放到num中。

函数：

void getNextchar(); //取下一个字符

void skipSpace(); //跳过’ ’, ‘\n’, ‘\t’

string toLower(string s); //大写转小写，标识符不区分大小写，因此都转为小写存储

bool isLetter(); //字母判断函数

bool isDigit(); //数字判断函数

bool isChar(); //字符判断函数

int searchKey(); //查找保留字，返回保留字枚举值

int searchOp(); //查找操作符与分界符

int searchOpforChar(); //查找字符停止集，用于跳错误

int transtoNum(); //将字符串转变为整型数

void catToken(); //将当前读入字符加入标识符的值串中

void reTract(); //回退一个字符

int getSym(); //获得下一个单词种类

int lexical\_main() //词法分析主函数入口

#### 2.3 语法分析 Grammar

主要是调用词法分析的getSym与词法分析交互，同时在语法分析中穿插语义分析、填符号表与中间代码生成的函数进行中间四元式代码的生成。词法和语法分析都是偏过程的的语法，这里均采用类C语言函数写法。

**函数:**

各个非终结符的语法分析函数，我抽到的文法共有42个非终结符，其中有五个终结符是可以直接推出非终结符，还有几个可以直接推出字符，字符串或者数字，剩下的终结符都可以各自形成一个函数处理对应的文法子句，这里不一一赘述，可以将这些不同的处理函数归类以便编程时结构清晰，以下给出归类后的函数分析：

void program(); // 程序开始

void const\_declare(); // 常量声明

void const\_def(); // 常量定义

void var\_declare(); // 变量声明

void var\_def(string name, tokenType retType); //变量定义

void statelist(); // 语句列

void statement(); // 语句

void compoundstat(); // 复合语句

void fun\_call(FuncItem\* it); // 函数调用

void fun\_def(); // 函数定义

void void\_iden\_fun(string name, tokenType retType); // 无返回值函数入口

void ret\_iden\_fun(string name, tokenType retType); // 有返回值函数入口

void main\_fun(string name, tokenType retType); // main函数入口

// 各种语句

void assign\_st(string\* name);

void if\_st();

void cond(bool\* certain, int\* value, string\* name);

void do\_while\_st();

void switch\_st();

void one\_case(tokenType cal, string\* end\_label, string\* next\_label,string\* tmp\_name);

void read\_st();

void write\_st();

void return\_st();

void paralist(string name, tokenType retType); // 参数列表

void declare\_head(string\* name, tokenType\* retType); // 声明头部

tokenType expr(bool\* is\_const, int\* value, string\* tmp\_name); // 表达式分析

tokenType item(bool\* is\_const, int\* value, string\* tmp\_name); // 项分析

tokenType factor(bool\* is\_const, int\* value, string\* tmp\_name); // 因子分析

#### 2.3 语义分析 Syntax

**函数：**

int check\_type(int type); //确定表达式或者赋值号两侧的变量类型是否一致，不一致则转入报错并进行跳读

bool check\_defined(string name);//确定某标识符是否被多次声明

Item\* read\_identity(); //对程序中的某个变量，获取其类别：int变量（常量）/字符变量（常量）/函数名

void gen\_tmp\_name(string\* name); //产生临时变量名

void gen\_string\_name(string \* name); // 产生字符串名

void gen\_label\_name(string \* name); // 产生label

Midcode\* push\_mid\_code(mid\_op M, string \* src1, string \* src2, string \* dst); // 存入中间代码

void print\_mid\_code(Midcode\* mid); // 中间代码的打印

#### 2.4 符号表 SymbolList

通过类来实现符号表的管理，由于我们的类C文法没有函数的嵌套定义，所以不会涉及某个程序的变量在多个分程序之间进行符号表搜索，所以建立全局符号表以及函数符号表两种结构即可。

**数据结构：**

1. **class** Item {
2. **private**:
3. std::string name;           // 常量或变量名称
4. Genre genre;                // 变量种类有常量，变量，参数， 对函数为有参函数以及无参函数
5. tokenType kind;             // 变量类型有int/char,void代表无法计算，对函数为函数返回值void/int/char
6. **int** value;                  // 对常量代表值
7. **int** length;                 // 对变量代表变量长度
9. **public**:
10. **int** offset;                 // 对全局变量与常量代表相对gp偏移；对函数内部则代表相对当前函数fp的偏移
11. **int** regNum;                 // 全局/局部/临时变量存放的寄存器
12. **bool** global;                // 指示是否为全局变量，便于生成代码时使用
14. Item(std::string name, Genre genre, tokenType kind, **int** value = 0, **int** length = 0);
15. std::string getName();
16. Genre getGenre();
17. tokenType getKind();
18. **int** getValue();
19. **int** getLength();
20. **void** setName(std::string s);
21. **void** setGenre(Genre genre);
22. **void** setKind(tokenType kind);
23. **void** setValue(**int** value);
24. **void** setLength(**int** length);
25. };
27. **class** FuncItem : **public** Item {
28. **private**:
29. std::map<std::string, Item\*>constMap; // 函数常量表
30. std::vector<Item>paraList;                // 函数参数表
31. std::map<std::string, Item\*>varMap;       // 函数变量表
32. **int** paranum;                            // 函数参数个数

35. **public**:
36. FuncItem(std::string name, Genre genre, tokenType kind, **int** paranum = 0);
37. std::set<**int**> used\_regs;              // 当前函数分配寄存器情况
38. Item\* get\_const(std::string name);
39. Item\* get\_var(std::string name);
40. Item\* get\_para(std::string name);
41. **friend** **class** Compiler;                  // 设置compiler为友元以便读取
43. };

**函数：**

map<string, Item\*>global\_const\_tab; // 全局常量表

map<string, Item\*>global\_var\_tab; // 全局变量表

map<string, FuncItem\*>fun\_tab; // 函数符号表

// 当前活跃函数符号表，上次活跃函数符号表(用于生成mips代码时跳回)

FuncItem\* cur\_active\_funtab; // 当前活跃函数表

vector<string>string\_tab; // 字符串vector表

bool check\_defined(string name); // 查看符号表中是否定义

Item \* get\_const\_var\_item(string name); // 获取符号表中的常量或变量信息，先找当前活跃函数然后寻找全局

FuncItem \* get\_func\_item(string name); // 获取函数符号表中的函数信息

void put\_const(string name, tokenType kind, int value); // 登入常量表

void put\_var(string name, tokenType kind, int value, int length); // 登入变量表

void put\_func(string name, Genre genre, tokenType kind, int paranum); //登入函数符号表

**方法：**对于变量与常量的符号表，主要就是对应域的get函数，变量表需要判断一下是否是数组。对于函数list，需要外加函数判断var，const与para是否存在，向三个map中插入新项push函数以及从这三个map中获取对应项的函数。具体参考上文代码以及代码说明。

#### 2.5 生成代码

**函数：**

vector<Mipscode> mips\_list; // 中间代码序列

RegRecord\* regs; // 寄存器分配情况

void generateMips(); // 生成代码主入口

void gen\_mips\_code(string\* op, string\* dst, string\* src1, string\* src2); // 产生mips代码指令

void asciiz\_setup(); // mips中.data的初始化

void init\_reg(RegRecord\* reg\_list); // 寄存器分配初始化

void get\_reg(string\* reg\_name, RegType\* reg\_kind, string\* name, bool\* hit);//为变量获取可使用reg

void get\_res\_reg(string\* reg\_name, RegType\* reg\_kind, string\* name, bool\* hit);//为目标变量获取reg

void get\_arr\_reg(string\* reg\_name, RegType\* reg\_kind, string\* name);//获取数组变量偏移的reg

void put\_in\_reg(string\* reg\_name, string\* var\_name);//mem->reg

void put\_in\_memory(string\* reg\_name, string\* var\_name);//reg->mem

string get\_ts\_reg(int n, RegType\* rg); //获取s/t寄存器

/\*下面是各大中间代码对应的mips处理函数\*/

void handle\_init(Midcode\* mid, int i);

void handle\_push(Midcode\* mid, int i);

void handle\_call(Midcode\* mid, int i);

void handle\_branch(Midcode\* mid, int i);

void handle\_jmp(Midcode\* mid, int i);

void handle\_scanf(Midcode\* mid, int i);

void handle\_printf(Midcode\* mid, int i);

void handle\_return(Midcode\* mid, int i);

void handle\_operation(Midcode\* mid, int i);

void handle\_label(string\* label);

void handle\_assign(Midcode\* mid, int i);

void handle\_arrto(Midcode\* mid, int i);

void handle\_arrget(Midcode\* mid, int i);

void handle\_exit();

void clear\_tmp\_reg(int i); // 跨基本块时写回临时寄存器

#### 2.6 代码优化

**数据结构：**

Blocks code\_blocks[MaxCodeBlocks]; //基本块

RegrefTab reg\_ref\_tab; //所有函数的引用计数

funRegref\* cur\_fun\_reg\_ref; // 当前函数的引用计数

bool\*\*\* outflow;(use,in.def) // 函数out集

int\*\* cur\_alloc\_reg; //当前全局寄存器分配情况

funBlocks fun\_blocks[MaxFunNum]; // 函数块

conflictGraph conflict\_nodes[MaxFunNum][MaxCflNodes]; //冲突图

int global\_reg\_num; //设置的全局寄存器个数

void read\_mips\_list();

void dag\_init();

void dag\_manipulate();

void peephole\_optimization();

void data\_flow\_analysis();

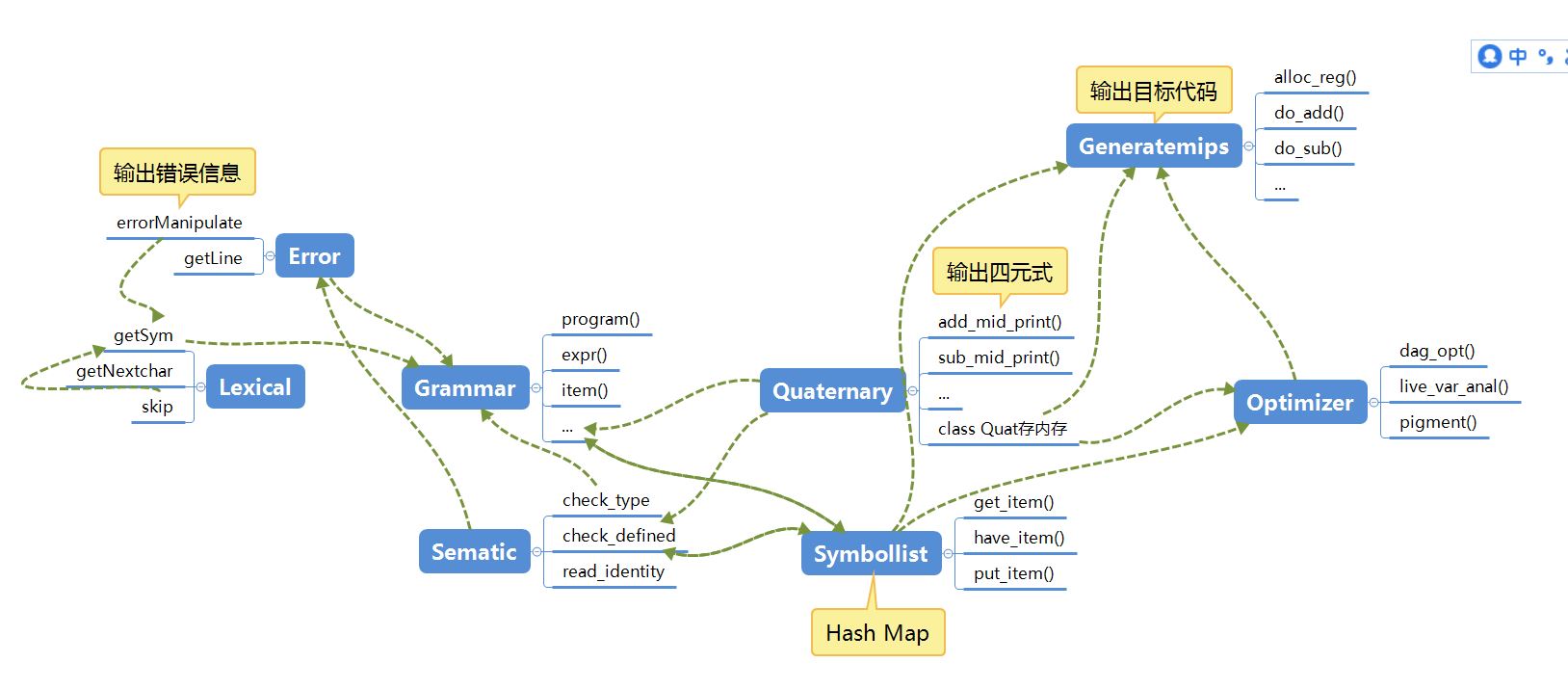
void ref\_count\_begin(int begin, int end);

void var\_count\_alloc(string name);

void ref\_count\_regalloc();

### 3．调用依赖关系

由于时间原因这里只给出一个全局角度的一个不同子类的一个相互调用关系，大体的功能是这样，里面的小细节没有在文档里面叙述得特别详细。



### 4．符号表管理方案

#### 4.1 数据结构

数据结构在2.4已经进行了具体的说明，主要就是全局变量与全局常量都用

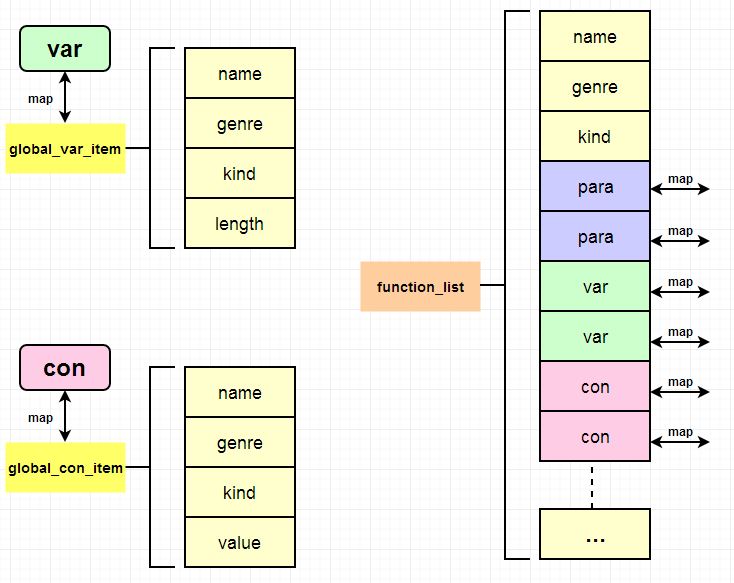
map<string, Varitem\*>或map<string,Conitem\*>

来实现，非别存储全局变量信息地址和全局常量信息地址而函数体内则将函数参数，函数局部变量以及函数局部常量放在一个class FuncList中实现封装。具体的函数已经在2.4中给出，这里给出一个图片来说明这些map的关系。

#### 4.2 相关算法

声明时需要向表中存储数据，需要提前扫描所在函数的信息表中的varmap和conmap两个信息表查看是否有变量已经声明的情况，没有同名的情况出现则名称合法，可以存入对应信息表中。

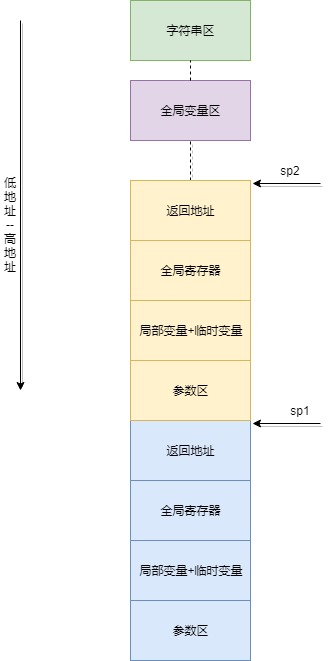
使用有关变量的时候需要先遍历当前函数信息内的局部常量与变量信息表，找到则取用，若是没有找到则查找全局信息变量与常量表，没有则报错。



### 5．存储分配方案

由于类C文法没有函数嵌套定义说明，所以运行栈不必使用display区来跟踪其外部定义，只需要开辟一个全局变量区即可，此时常量应该已经全部被转换为整型存在四元式中，不需再在运行栈中记录

在代码生成与优化部分要用到全局寄存器的分配以及临时寄存器的暂时占用，所以要在程序的数据区分配出一定量的全局寄存器进行模拟，便于最后mips上面的寄存器进行对应，这部分的设计如下：



在.data区将写入字符串，0x10010000开始存入全局变量，之后然后运行栈为从高地址到底地址的设计，从初始sp 0x7fffeffc开始向下，一次存入当前函数的参数，局部与临时变量，要保存的全局寄存器以及函数的返回地址，这里没有保存sp或者fp是因为每一次函数调用的时候被调用函数上述各个部分在进行语法分析后均可以确定，所以每次调用某个函数在被调函数init阶段减掉该偏移，在ret阶段加上该偏移即可

### 6. 解释执行程序\*

【难度3的代码没有此项】

### 7. 四元式设计\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| op | src1 | src2 | dst | 说明 |
| ADD | √ | √ | √ | 相加运算 |
| SUB | √ | √ | √ | 相减运算 |
| MUL | √ | √ | √ | 相乘运算 |
| DIV | √ | √ | √ | 相除运算 |
| ASS | √ | × | √ | 单个字符/整形赋值 |
| ARGET | t1 | t2 | t3 | t1[t2]=t3 |
| ARTO | t1 | t2 | t3 | t3=t1[t2] |
| J | √ | × | × | 无条件跳转 |
| LABEL | √ | × | × | 在此处打标签 |
| BGJ | √ | √ | √ | src1>src2跳转至src3 |
| BGEJ | √ | √ | √ | src1>=src2跳转至src3 |
| LSJ | √ | √ | √ | src1<src2跳转至src3 |
| LSEJ | √ | √ | √ | src1<=src2跳转至src3 |
| EQJ | √ | √ | √ | src1=src2跳转至src3 |
| NEQJ | √ | √ | √ | src1!=src2跳转至src3 |
| @fun | foo |  |  | 定义函数 |
| @para | type | a |  | 定义参数 |
| @push | x |  | 1 | 传入参数,dst指示参数位置 |
| @call | fun |  |  | 调用函数 |
| @scanf | type | a |  | 读函数(2类型) |
| @printS | strname |  |  | 写字符串函数，  strname扫描对应函数区域字符串表 |
| @printV | int | a |  | 写表达式函数，有类型 |
| @retN |  |  |  | 无返回值函数 |
| @retV | a |  |  | 有返回值函数 |
| @exit |  |  |  | 退出程序语句 |

**数据结构：**

1. // 四元式保存结构体
2. **typedef** **struct** {
3. mid\_op mid\_op;
4. std::string\* src1;
5. std::string\* src2;
6. std::string\* dst;
7. } Midcode;

所有四元式放在一个vector中构成四元式数组。一份放在内存中代码优化时备用，一份作为四元式输出函数的输出参考。

四元式的四个部分以空格分隔，op为操作符，src1与src2为两个操作数，dst为四元式的操作结果。

### 8. 目标代码生成方案\*

1. **typedef** **struct** {
2. std::string name;   //占用寄存器的名字
3. **bool** occupied;      //是否被全局寄存器占用
4. **bool** available;     //当前寄存器是否可用
5. **bool** global;        //当前寄存器是否被分给全局变量
6. **int** offset;     //当前分配变量在函数的位置
7. **int** idx;        //数组变量指示下标
8. }RegRecord;
9. **int**\*\* cur\_alloc\_reg;            //函数各变量分得的寄存器是什么
11. // mips代码不保存结构体，直接由函数生成目标代码
12. **void** gen\_mips\_code(string\* op, string\* dst, string\* src1, string\* src2);

在没有优化的时候只要选取3个寄存器进行每次的承接以及写回mem即可，用的时候直接在mem中load，这个时候只需要进行符号表的查找将变量的偏移找出来即可。在进行优化之后就需要RegRecord在染色法的时候进行每个函数的变量分配，其中occupied代表是否有变量占有这个全局寄存器，若是occupied为false则该寄存器可以进入临时寄存器池进行临时寄存器的分配，最后进行完着色法分配寄存器之后就会获得该函数的cur\_alloc\_reg，里面记录了各个偏移下的变量对应的寄存器分配情况。

### 9. 优化方案\*

#### 9.1 常数合并

在语法分析时，程序会解析出表达式中各项的单词类型，此时的三个语法分析函数expr，item以及fator可以通过递归调用将常量进行合并。

目前的想法是在三个函数进行分析时分别传入\*is\_const指针，\*value指针与\*tmp\_name，两个指针分别能从递归的语法分析项获取底层常数分析结果，其中is\_count为真时，value中会传回底层计算获得的常数值，当is\_count为假时，底层会产生一个临时变量名，同时将底层处理的运算写入四元式vector中，向上返回临时变量记号。如果向上传递后某运算符左右的变量均可确定为常量，就用这个常量代替这个表达式，这样经过一层一层地向上传递就可以将常量折叠，但是缺点是这种阅读代码的方式是从左向右，从上至下的，所以当一个表达式第一个不是常量而后面的都是常量，比如a = b + 1 / 3等等，这种方法对后面可以合并的常量就无能为力了。

1. tokenType expr(**bool**\* is\_const, **int**\* value, string\* tmp\_name);// 表达式分析
2. tokenType item(**bool**\* is\_const, **int**\* value, string\* tmp\_name);   // 项分析
3. tokenType factor(**bool**\* is\_const, **int**\* value, string\* tmp\_name); // 因子分析
4. **void** gen\_tmp\_name(string\* name);  // 产生临时变量

#### 9.2 基本块内部公共子表达式的删除

#### 9.2.1 基本块的划分

基本块的划分按照书上的算法的修改版来做，因为书上并不支持函数调用等中间代码的形式，这里将基本块的划分修改为(以中间代码为例)

label或init\_fun开始

call函数调用结束，函数返回开始

条件跳转beq与无条件跳转j

程序结束exit

**数据结构：**

**基本块数据结构的设计：**

1. **typedef** **struct** {
2. std::string name;               // 当前块名称
3. std::string j\_name;             // 目标块名称
4. **bool** is\_print;                  // 需要打印吗？
5. **int** idx;                        // 当前基本块的序号
6. **int** first\_line\_num;             // 基本块第一行代码
7. **int** last\_line\_num;              // 基本块最后一行代码
8. **int** pre\_blocks[MaxCodeBlocks];  // 前面指向本块的block的序号
9. **int** pre\_blocks\_cnt;             // 前面指向本块的count
10. **int** follow\_block;               // 直接后继块的序号
11. **int** j\_block;                    // 跳转后继块的序号
12. }Blocks;
14. **typedef** **struct** {
15. **int** begin\_block\_line;
16. **int** end\_block\_line;
17. **int** begin\_block;
18. **int** end\_block;
19. std::string name;
20. }funBlocks;

**说明：**

经过考虑，同一个基本块会有很多个入口，但对于出口只有顺序执行和跳转两个，否则这个基本块的划分就不完全。

编译执行时，每个函数的第一个和最后一个基本块分别设置为函数入口与出口，入口没有前驱基本块而出口没有后继基本块，这样便于基本块结构的划分以及数据流划分。

**算法：**

建立基本块的算法主要就是根据不同的操作符进行基本块的划分，首先读取存在内存中的序列，然后根据mid->op进行基本块的划分，遇到条件跳转指令，j指令，call就进行基本块的分割，当前基本块可以结束，同时开辟一个新的基本块，同时条件跳转与call的follow\_block设置为对应新block值，遇到label，fun\_init也是如此开启新的基本块，ret也是如此。此时所有新块就被划分出来了，然后进行第二次扫描进行连接操作，如果当前基本块的j\_name与另一基本块的name相同则进行连接，最终就可以分配好流图。

**公共子表达式的删除：**

**数据结构：**

1. // DAG节点表
2. **typedef** **struct** {
3. std::string name;           // 结点名称
4. **int** idx;                    // dag图中序数
5. **bool** modified;              // 表示该结点对应的符号是否被重新定义过
6. }dagNodelist;
7. // DAG图
8. **typedef** **struct** {
9. std::string name;                   // 结点名称
10. mid\_op op;                          // 结点运算符
11. **int** idx;                            // 结点在dag图中位置
12. **int** value;                          // 判断该语句的值可否确定,若是参数则代表参数的idx
13. **int** child\_1;
14. **int** child\_2;
15. **int** prt\_num;
16. std::set<std::string> name\_candidates;// 历史定义过的名字集
18. **bool** isLeaf;                        // 是否是叶节点
19. }dagNode;

**算法说明：**

由于书上的公共子表达式删除支持的指令种类太少，所以我们将它扩充至包含scanf，printf，push上面，同时书上也没有写明白dag图上的变量被kill掉的情况。

1. scanf printf push

scanf相当于对变量的重新定义，与赋值语句别无二致，所以这里就把scanf看作没有src的一个赋值语句，node的操作符设为SCANF，这样相当于会建立一个由空到目标变量的一段dag图，之后再将dag图的名称以及dag\_list上的名称设为对应值即可。

printf与push是对变量的使用，但是这里printf与push有顺序问题，所以必须单拿出一个结点进行存储，所以最后实现时在戴用变量的上方建立新的node，同时将node的op标注为PRINT或PUSH，相当于原样输出这个中间代码。

2.中间结点的名称保留

为每一个中间结点建立一个node\_candidate的set集合，每一次一旦有一个新的结点指向了这个node结点，就把新节点的名称传给该node的node\_candidate，然后node的名称如果是临时变量就尽量变为局部或者全局变量。然后在取用一个结点表的一个结点src时，如果命中，那么要检查一下结点表中的idx(对应dag图中的序数)指示的变量a在结点表中是否有比当前更大的序数，即该变量a是否有新定义值，如果有新定义值的话就要在当前的dagnode的node\_candidate中删去该name，将src的name填入原来a的位置中以防恢复中间代码的时候产生冗余的恢复代码。

3.中间代码的导出

书上采用的是启发式算法，书上采用如此的顾虑是因为编译器的寄存器较少，其实在我们这么多寄存器的前提下，不需要采用启发式算法完全得以应付绝大多数情况，同时还可以将printf与push等按顺序输出而不至于产生错误。

#### 9.3 活跃变量分析

**数据结构：**

in[fun\_idx][MaxB][MaxItem], out[fun\_idx] [MaxB][MaxItem],

use[fun\_idx][MaxB][MaxItem],def[fun\_idx][MaxB][MaxItem]

**说明：**

活跃变量分析首先就是对各个函数块中各个基本块的in，out，use，def的分析，in和out是通过进出基本块的流图来确定的，在之前建立的基本块的基础上先确定use/def图

1. 对于四则运算与正负赋值，检查src1，src2可否加入use集合，dst可否加入def集合。

2. 数组给变量赋值，被赋值变量检查是否加入def，数组下标操作数及数组变量检查加入use集合。

3. 变量给数组赋值，检查赋值变量加入use，数组与下表操作数加入def。

4. 条件跳转语句判断操作数是否加入use

5. scanf语句输入变量检查是否加入def集合

6. printf，push语句使用的操作数是否加入use集合。

之后先扫描一遍各函数块的DAG块即可给出对应的use与def集合，再由in与out的书上公式：

in[B] = use[B]∪(out[B]-def[B])

out[B] = ∪B的后继基本块in[F]

来计算各个in与out

**算法：**

采用书上的位运算，为符号表中每一个符号给出一个bool值，之后运用位运算0 1的求差与按位或即可得出每一个符号在基本块的活跃情况

#### 9.4 着色法对全局寄存器的分配

**算法说明：**

在活跃变量分析的基础上，定义基本块内变量之间冲突图，这个冲突的定义按书上“两个变量中的一个变量在另一个变量定义(赋值)处是活跃的，它们之间有一条边连接”的工程量有一些大，先保守地取in集与def集同时为1的时候判断该变量为活跃，在一个基本块同时出现活跃的变量为冲突变量，之后执行书上算法15.1的图着色算法即可。在着色时，首先进行结点删除，这时对不可分配的结点的remove置为-1，否则分配的话按照先后顺序从零开始增加，相当于进入队列的顺序，恢复的时候逆序恢复分配即可。

**数据结构：**

1. // 冲突图
2. **typedef** **struct** {
3. **int** join[1000];
4. **int** join\_num;
5. **bool** connect[1000];
6. }conflictGraph;
7. int remove[max]; // 确定某个变量是否被删除，同时未删除的话大小代表进入队列先后顺序

#### 9.5 窥孔优化

1.连续的label，若是连续的label中有函数开始名则将这些保存为函数开始名，若是没有函数开始名则将这个保存为最后一个的label。

2.跳转的目标语句还是跳转语句，可以将第后面的跳转语句删除。

3.跳转与对应label直接相连，这是可以直接将j删去

### 10. 出错处理

向errorManipulate()发送行号和错误信息进行打印看，同时程序会根据停止集进行跳读代码。由于后续实现时并没有将具体的错误进行命名，只是将错误信息暴露给errorManipulate，所以错误类型按语义进行一下简单翻译。

**10.1词法分析错误**

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| SQMISMATCH | 没有找到匹配的单引号 |
| DQMISMATCH | 没有找到匹配的双引号 |
| INVALIDCHAR | 不是给定文法范围内的字符 |
| INVALIDDQCHARMSG | 双引号内部包含的字符不合法 |
| INVALIDSQCHARMSG | 单引号内部包含的字符不合法 |
| LACKEQLMSG | !单独出现了 |
| ILLEGALVARTYPEMSG | 不合法的类型！ |

**10.2 语法分析错误**

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| ARRIDXNOTPOS | 数组定义时下标不是正整数 ！ |
| LACKSEMMI | 缺少”;” |
| NOTMACTHSTH | 没有匹配’x’ ！(多种类型(保留字/标识符均可)) |
| NOFUNDEF | 没有函数类型定义 ！ |
| NOMAIN | 没有main函数 ！ |
| NOCONSTAFTERCASE | case后面接的不是常量 |
| UNKNOWNSTATE | 无法识别的语句开始符号 ！ |
| WRONGSEQINPRT | 在输出语句中字符串与表达式的顺序反了！ |
| CFLTIDRESERVERED | 标识符与保留字冲突 ! |
| NODEFAULT | 没有匹配default |

**10.3 语义分析错误**

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| DIVCASE | 可以检测到的除以0 |
| NOMATCH | 没有匹配所需字符 |
| NORETURN | 有返回值函数存在没有分支返回的情况 ！ |
| PARANUMWRONG | 调用函数时参数个数不匹配！ |
| WRONGBRACEINFUN | 无参函数在调用时加入小括号！ |
| NOTDEF | 变量或函数未定义 |
| ASSTOCONST | 对常量赋值 |
| ARRIDXOUT | 目标数组发生数组越界 ！ |
| ASSTOFUN | 向函数进行赋值 ！ |
| WRONGTYPEINASS | 赋值号两边单词类型不一致 ！ |
| WRONGTYPEINCOND | 条件表达式的两侧存在不是int类型情况 ！ |
| SWITCHCASENOTMATCH | switch后面的表达式和case后面的常量类型不匹配 ！ |
| SCANNOTDEFINED | scanf读取未定义符号！ |
| READTOCONFUN | 向常量或函数名进行读入！ |
| NOTIDAFTERSCANF | 在scanf函数后面没有标识符 ！ |
| RETURNINNORET | 在无返回值的函数中的return语句存在返回值！ |
| RETURNTYPENOTMATCH | 函数的实际返回值类型与定义不符！ |
| UNKNOWNINEXPR | 表达式中出现不可求值项 ！ |
| ARRNOTDEF | 数组变量未定义！ |
| PARANUMMISMATCH | 函数调用时参数个数不匹配 |
| PARAGENREMISMATCH | 函数调用时参数类型不匹配 |

## 三．操作说明

### 1．运行环境

1. C++工程文件使用VS2017打开，也可以用其他的编辑器打开代码文件
2. mips代码运行的虚拟环境是Mars 4.5，满足 ①不支持延迟槽②内存使用default模式

### 2．操作步骤

打开工程文件之后，在main.cpp中输入待测文件路径，输入完毕后直接运行，在当前工程文件下的output文件夹下可以看到所有的输出文件信息，其中mips.asm为最终生成的汇编文件，dag\_mid\_out.txt表示经过dag图优化后的中间代码，dag\_out表示dag图信息，lexi\_out.txt, error\_out.txt, grammer.tx,mid\_out分别表示词法，错误处理，语法以及原始中间代码输出信息，live\_out.txt以及reg\_count\_out.txt分别对应活跃变量以及引用计数输出，这些文件都有相应的开关函数。

## 四．测试报告

### 1．测试程序及测试结果

**1．1测试1（正确）**

本程序重点测试了函数的递归调用调用，同时结合函数的输入输出。

程序的功能是输入一个数，输出其斐波那契数，结果正确。

**输入**5： **输出**：8

**1．2测试2（正确）**

本程序重点测试所有条件跳转语句与表达式的计算，包括 减法，除法，乘法，函数调用以及数组调用的嵌套，对switch-case循环的两种常量形式进行了测试，对递归调用fib进行了一个测试同时还测试空语句等等语法成分。

**输入2：**2 3 a **输出：** FUNC WITH PARA BUT NO RETURN

103-1022966

2 !#$%&'()\*+,-./0123456789:;<=>?@ ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^\_`abcdefghijklmnopqrstuvwxyz{|}~

**输入3.1：**3 3 \* **输出：**none1 ok

**输入3.2：**3 3 / **输出：**none2 ok

**1．3测试3（正确）**

本程序重点测试了函数的递归调用调用，同时还有do\_while的循环语句，数组变量的存取以及大小写与下划线命名等问题

**输入**3： **输出**：55

**1．4测试4（正确）**

本程序重点表达式运算以及函数调用时的传参顺序，同时测试了输入输出语句，函数调用等。

程序的功能输入一个代表全局变量的数，然后由一系列表达式以及函数运算对全局变量判断先后调用顺序进行求值。

**输入**：10 **输出**：-215

**1．5测试5 （正确）**

本程序主要测试大量传参及输出语句的情况，保证寄存器和栈式传参方式互不干扰。

**程序输出**：abcde15a

**1．6测试6（错误）**

本程序主要测试词法分析部分和部分语法分析与语义分析部分错误，包括缺少单引号，双引号，’!=’缺少等号，还有数组定义时下标为非正整数以及赋值号两边单词类型不匹配等（等5处错误。）

首先，第五行flag数组的定义出现了-100这个负整数，之后在main函数中的第二行对字符c进行赋值时缺失单引号，连带着语法分析求不出该项的值类型。main中第4行右侧双引号缺失，第五行缺少’=’导致报错。

**1．7测试7（错误）**

本程序主要测试语法分析时候的错误。

根据文法，第8行只有main函数在无参数时可以进行()调用操作，main1函数不可；第9行定义了一个不符合文法的字符，抛出没有匹配标识符错误；第10行保留字与标识符冲突报错，第11行语句缺少分号，第12行表达式与变量输出时顺序反了；第13行缺少右大括号；最后一行程序没有检测到main函数，报错。

**1．8测试8（错误）**

本程序主要测试表达式运算有关错误。具体包括第11行的除以零异常，同时第12行加了一个无返回值函数不能返回值的错误，第十七行使用为定义变量，第18行赋值号两边变量类型不一致，第19行对flag数组取用写入时超过数组边界，第20行对常量值进行赋值等等共计6个错误。

**1．9测试9（错误）**

本程序主要测试函数调用有关错误。

第11行中，fun1函数结束却所有分支均没有返回值，第13行fun2函数的返回值与定义的返回值类型不符，第17行在main函数里面调用fun1的时候只传了一个参数，第18行虽然传了两个参数但是第二个参数的类型不匹配，第19行多传了一个参数，以上5处均进行报错。

**1．10测试10（错误）**

本程序为default类的测试。

第10行中，switch接的表达式与case后面呢的常量类型不一致；第11行case没有匹配冒号；第12行case后面没有接常量，同时还没有匹配defauult；第14行return的时候没有匹配左右小括号，第19行在条件表达式的一端存在不是int的类型，总共6种错误。

### 2．测试结果分析

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

## 五．总结感想

这个学期的编译终于完成了，现在在写总结感想的我真是一言难尽。

作为一名高等理工学院的学生，相比计算机学院的我们还没有经历oo这门重课的洗礼，所以在最开始面对编译课设这门课的时候的感觉就是无从下手。这门课程和之前的计组感觉完全不一样，这门课需要我们实现的东西没有可以直接参考的指导书进行学习，大多数的东西都是自己看书找资料一点一点体会实现出来的，我在这门课上耗费的精力应该有的时候比计组还多，经常就是坐在电脑前面拿张纸画来画去但始终没有头绪，不过最后这门课也算结束了，也算是“功德圆满”吧。最开始的pascal代码的研读还是需要花些时间的，因为这个是我们第一次接触到完善的编译程序，对我们来说也是理论向实践过渡的第一次接触，把pascal那里看完之后就会对整个语法语义分析以及符号表部分有一个大致的了解。记得第一次写设计文档的时候只进行完了词法分析，对后面的编译完全不了解，老师说了提前架构的重要性所以当时自己花了一个周末想后面的实现方法，虽然说后面也基本上改了不少(没错不少都是错的)，但是也算提前把编译的大体数据流向走了一遍，这一遍对后面的代码生成与优化部分还是很有用的。后面代码生成的部分感觉还可以，大概都是机械化的进行内存的写进写出，不过当时符号表的架构用的c++标准库里的map然后我就没有记变量的顺序，最后后面优化要用的时候就需要改动符号表，又费了点周折。还有一个感想就是代码优化真的很难很难，本来自己就属于写程序脑子没那么灵光的人，遇到这种这么大架构而耦合性又这么强的程序实在是走了好多弯路，在写的时候没有全面考虑一个统一的优化架构而是做一个优化定义一个数据结构，这样就造成后来自己也搞不清这些数据结构的意思了，需要来回翻看很累人，造成了很多现在看来不必要的debug时间。还有编译的理论课一定要好好学，我的编译理论课上得感觉并不好，当时听课的时候脑子厉害没有形成代码生成与优化的概念，听完之后只是能够根据书上的算法解题，但是并不知道这么做在什么情况下，针对什么样的程序有较好的优化效果，它的一些实现上面应该注意的地方是什么，这些都需要细致的思考。

总之，编译这门课给我的感觉还不错，也让我知道了自己在实现代码上确实和大佬们有些差距，但是做完这些之后这门课让我确确实实的感受到了自己在进步，非常感谢感谢这门课设，感谢老师学长学姐们的热心指导。