《编译技术》课程设计

文 档

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

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

年 月 日

## 一．需求说明

### 1．文法说明

**获取的原始文法：**

1. ＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

范例：

a + b; a - b 加法运算符部分

a \* b; a / b 乘法运算符部分

a < b; a <= b; a > b; a >= b; a != b; a == b 关系运算符部分

分析：定义了文法的基本运算符，包括加减乘除四则运算以及各种比较关系运算符

2. ＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜＜非零数字＞

＜非零数字＞ ::= １｜．．．｜９

范例：

\_, a ,B 字母部分

0, 3, 5 数字部分

1, 7, 9 非零数字部分

分析：定义了文法的字母以及数字的组成规则，其中字母包括了下划线\_,以及26个英文字母，数字为10个阿拉伯数字

3. ＜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｝"

范例：

'+', '-', '\*', '/', 'D', '2' 字符部分

"", "STRING", "string" 字符串部分

分析：定义了字符以及字符串的组成规则，其中字符以单引号开头，同样以单引号结尾，长度为1。字符串以双引

号开头，同样以双引号结尾，可以为空字符串，即对应长度为0；也可以包含其他字符，ASCII的十进制编码范围为32,

33,35-126(不能包括双引号字符)，其长度大于等于1;字符串只能在写语句中使用。

4. ＜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＞}

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜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＞‘]’)}

＜类型标识符＞ ::= int | char

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

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

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

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

范例：

const int num = -25; 常量说明部分多种形式

const int num1 = 10, num2 = 0; const char ch\_1 = 'c', ch\_2 = 'd';

int number, array[5]; char ch,c\_arra[10]; 变量说明部分

int foo(int x){

return (x+1);

} 有返回值函数定义部分

void func(){

;

} 无返回值函数定义部分

void main(){

int x;

x = x + 1;

} 主函数部分

void main(){

int x;

x = x + 1;

}

const int num = 5;

void func(){

;

}

int foo(int x){

return (x+1);

}

void main(){

num = foo(num);

func();

} 程序的若干形式

分析1：根据常量说明以及变量说明文法，常量说明必须以const开头，包含int和char两种数据类型;变量说明要以int或者

char开头，可以进行数组类型的变量声明。常量以及变量声明都可以进行连续的声明。整数可以以'+'或者'-'开头，也可以不

包含符号而直接由数字组成。无符号整数必须以非零数字开头。标识符可以包含字母和数字，但是必须要以字母开头。在进行变量

声明时，不能够对其进行赋值操作，这与C语言的文法不相同。

分析2：函数定义包含有返回值函数定义以及无返回值函数定义。有返回值函数定义以int或char两种类型开头，而无返回值函数

定义以void开头，随后两种定义都需进行标识符的声明，后面接'('和参数以及')'，后面再接'{'以及要执行的复合语句以及'}'。

主函数的定义格式为void main(){}，不能为int main(){}，与C语言文法有一些差异，花括号中填写要执行的复合语句 。

分析3：根据程序的文法可知，程序的常量，变量以及函数三个组成成分的声明顺序已经被限定，不能更改各部分的声明顺序，

如“int a; const int b=5;”这种顺序是不允许的。而对于函数的说明顺序没有限制，可以随意改变二者声明的顺序，例如先

进行无返回值函数定义再进行有返回值函数定义，或者有返回值函数与无返回值函数交替定义也是可以的。

分析4：常量说明、变量说明以及有返回值函数和无返回值函数定义对于一个程序来说不是必要的部分，可有可无，但是必须要

包括最后的主函数部分。

5. ＜参数＞ ::= ＜参数表＞

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

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜语句列＞ ::= ｛＜语句＞｝

范例：

空（即没有参数） 参数表为空的举例

int x,char y 参数表部分

复合语句部分：

const int num\_1 = -5; 常量声明

int num\_2; 变量声明

num\_2 = num\_1; 语句列

分析1：参数表有多种组成形式，参数表可以为空，代表此函数不接受任何参数；若包含参数，则可包含单个或者多个参数。常量

声明以及变量声明不是复合语句的必要组成部分，但是复合语句必须包含语句列，而语句列则可由零至多个语句来组成。

分析2：如同上述程序的文法一样，复合语句的文法规定了常量说明以及变量说明的先后顺序，若声明顺序颠倒，

则违反了文法的规定。

6. ＜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＞［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＞ ::= for‘(’＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞‘)’＜语句＞

＜步长＞ ::= ＜非零数字＞｛＜数字＞｝

＜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＞

＜常量＞ ::= ＜整数＞|＜字符＞

＜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＞‘)’

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

＜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＞‘)’]

范例：

a = 10

array[2] = 'c' 赋值语句的若干形式

if(a == 1) b = 2;

if(1){

b = 2;

c = 3;

}

if(a == 1) b = 2;

else b = 3; 条件语句的若干形式

for( i = 0; i < -1; i = i + 1)

sum = sum + i;

for( i = 10; i >= 0; i = i - 2){

sum = sum + i;

res = res \* i;

} 循环语句的若干形式

switch(flag){

case 0:

flag = 1;

case 1:

flag = 0;

default:

flag = 10;

} 情况语句部分

foo(x) 有返回值函数调用语句部分

func() 无返回值函数调用语句部分

scanf(num\_1,num\_2) 读语句部分

printf(a + b)

printf("STRING")

printf("STRING",a \* b) 写语句的若干形式

return

return(a) 返回语句的若干形式

语句部分(仅举若干例子):

a = 10; 赋值语句

for( i = 0; i < 10; i = i + 1){

sum = sum + i;

} 循环语句

foo(x); 有返回值函数调用语句

printf(10); 写语句

return(0); 返回语句

分析1：赋值语句可以实现为int或者char类型的数据赋值，也可以为包含若干int或者char类型的数据的数组的特定

位置的数据赋值;

条件语句以if关键字开头，可以只包含if这一分支，也可以在if之后加上else的分支，其中if的判断条件

的组成形式有两种，要么是＜表达式＞＜关系运算符＞＜表达式＞要么为＜表达式＞，单个表达式只有为0的时候，

if的判断条件才为假，否则均为真;

循环语句以for关键字开头，小括号中的内容由三个部分组成，之后再接操作语句，其中步长必须要以非零数字开头;

情况语句以switch关键字开头，之后的小括号中内容为表达式，之后的花括号中的部分由情况表和缺省组成。情况表为一

到多个情况子语句，每个情况子语句以case关键字开头，之后接常量，再接冒号，之后再接操作语句，其中常量可以

是整型也可以为字符型，缺省语句以default关键字开头，之后接冒号，再接操作语句;

有返回值函数调用语句由有返回值函数名加上左小括号再加上值参数表再加上右小括号组成，同理无返回值函数调用语句

由无返回值函数名加上左小括号再加上值参数表再加上右小括号组成，其中值参数表可以为空，也可以为一至多个表达式

组成，表达式之间应以逗号分隔，具体的值参数表的组成形式应该与该函数定义时的参数组成形式相同;

读语句以scanf关键字开头，之后接一对小括号，小括号中的部分由一至多个标识符组成，标识符之间以逗号分隔;

写语句以printf关键字开头，之后接一对小括号，小括号中的部分要么由字符串和表达式组成，二者之间以逗号分隔，

要么为一个字符串，要么为一个表达式;

返回语句以return关键字开头，之后可以接小括号，小括号中的部分为一个表达式;也可以不接小括号，仅有return组成。

分析2：语句的组成方式有多种，有一些语句需要以分号结束，其他的语句则不需要以分号结束;此外，循环语句在小括号中

的第三部分允许使用的运算符只有加号和减号，步长必须要以非零数字开头;情况语句中的缺省部分必须要有，这与C语言

文法略有不同;读语句传入标识符即可，省略了某些需要取地址的操作;写语句也没有对于输出语句的格式化字符串要求，且输出的字符串以及表达式的个数最多为1，

不同于C语言的文法;返回语句return若要有表达式，则其表达式必须在一对小括号中，也与C语言文法略有不同。

7. ＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜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＞‘)’

范例：

a

- a + 2

a \* 2 + b / 3

+ a \* (2 + 3)

array[4] + a

+123

'c'

a \* foo(3) 表达式的若干形似

b

1 \* 2 \* b

(array[1] + 2) 项的若干形式

c

array[10]

23

'd'

foo(2)

(3 \* 4) 因子的若干形式

分析：表达式的组成形式多种多样，第一个项前面的'+'或者'-'是表示的是第一个项是负数或正数，只能在数字前加上符号；

后面的'+'或者'-'则为加法运算符的组成部分；表达式的多样性使得其不仅能够进行多种数值运算以及规定运算顺序，也使得

其能进行标识符的引用以及实现对有返回值函数的调用等；因子也可以继续转化为‘(’＜表达式＞‘)’的形式，实现为程序时，则

体现为因子分析程序可以递归调用表达式的分析程序。

**对文法的改写和扩充：**

通过对文法的分析，可以发现在抽取到的文法中不存在左递归的问题，但是存在需要用到回溯才能解决的问题，比如说在处理变量说明和有返回值函数定义的时候无法根据当前读入的单个单词符号判读其到底属于哪一类别。对于这个问题，程序中采取预读字符的方式来解决。

### 2．目标代码说明

生成的目标代码为MIPS指令，最终由mips汇编模拟器MARS对由程序生成的目标代码进行执行。

### 3. 优化方案\*

优化部分：

基本块内部利用DAG图进行公共子表达式的删除；

数据流以及活跃变量分析实现全局优化；

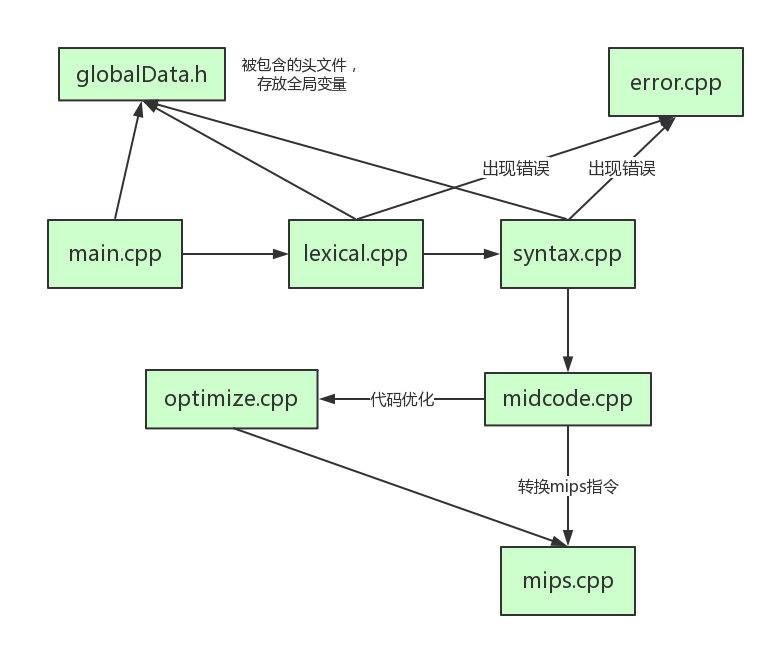
常数合并：当加减乘除运算的操作符的两个操作数均是常数时，进行常数的合并以及对于赋值对象的直接赋值；  
利用启发式图着色算法进行全局寄存器的分配。

## 二．详细设计

### 1．程序结构

程序分为词法分析、语法分析和语义分析、代码优化以及目标代码生成等过程，包含的文件有lexical.cpp, syntax.cpp, error.cpp, midcode.cpp, optimize.cpp, mips.cpp, main.cpp以及globalData.h。

各个文件之间的调用关系如下图所示：



### 2．类/方法/函数功能

**词法分析：lexical.cpp**

涉及到的函数如下：

int charIsLegal(char ch);

int stringCharIsLegal(char ch);

int compareNum(string s1, string s2);

void nextch();

void insymbol();

分别的功能为：

charIsLegal(char ch)：判断ch字符是否符合字符组成文法的要求，符合，返回非0整数，否则返回0；

stringCharIsLegal(char ch)：判断ch字符是否符合字符串组成文法的要求，符合，返回非0整数，否则返回0；

compareNum(string s1, string s2)：对于传入的两个数字字符串进行大小比较，若s1的数值大于s2的数值，则返回1，否则返回0。

nextch()：将输入的文件按照单个字符的形式进行存取，将每次读入的字符填入全局变量ch；

insymbol()：由语法分析程序调用，先读取一个字符，若为字母，则继续读入直到ch不是为止数字或者字母为止，然后查表判断是否为保留字，若为保留字，则相应的保留字类型存入token，否则为标识符，token设置为ident；若读入的是以单引号包围的字符，则token设置为字符类型，inum存储其ASCII码；若读入的是数字，则token设置为整数类型，inum存储数值；读入字符串或者操作符号以及运算符号等，也都将token设置为对应的类型。

**语法分析：syntax.cpp**

涉及到的函数如下：

void writeToFile(string writeString);

void writeToInfixExpressionFile(string str);

string convertNumToString(int num);

void printSymtableTable();

int insertIntoSymTable(string name, int kind, int value, int arraySize, int address, int paraNum);

void flushTable();

int findInSymbolTable(string name, int searchMode);

void programme();

void constExplain();

void constDefine(int type);

void declareHead();

void returnFuncDefine();

void parameterTable();

void noReturnFuncDefine();

void MainDefine();

void variableExplain();

void variableDefine();

void complexStatement();

void statementList();

void statement();

void valueOfParameterTable();

void expression();

void item();

void factor();

void assignStatement();

void ifStatement();

void condition(int conditionType);

void forStatement();

void switchStatement();

void situationTable(string default\_label\_final,string switchResult,int SwitchVarType);

void situationStatement(string default\_label\_final, string switchResult, int SwitchVarType);

void defaultStatement();

void scanfStatement();

void printfStatement();

void returnStatement();

分别的功能为：

依据文法中的非终结符，采用递归下降分析的方法，依次编写处理程序，在应用到该非终结符的分析时，调用该非终结符的分析程序实现语法的分析。

void writeToFile(string writeString)：将语法分析程序分析到的成分writeString输出到文件中，便于观察语法分析运行结果。

void writeToInfixExpressionFile(string str)：将中缀表达式成分str输出到文件中，便于观察结果。

string convertNumToString(int num)：转换输入的num整型数字为该数字对应的字符串形式，返回该字符串

void printSymtableTable()：打印符号表到文件，便于观察符号表中的各个符号表项。

int insertIntoSymTable(string name, int kind, int value, int arraySize, int address, int paraNum)：插入符号表中，包括名字、类型、值、数组大小表示、地址，需要检查插入的表项是否重复，若插入成功返回该表项对应的地址；否则返回-1

void flushTable()：该函数在某个函数符号表处理结束后进行调用，除去函数体在符号表中除了函数名字和函数参数类型以外的其余数据

int findInSymbolTable(string name, int searchMode)：在符号表中查找传入的name代表的标识符或者函数是否存在，参数searchMode为1时，代表搜索函数；为0代表搜索标识符

void programme()：用于分析程序，对应＜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＞

void constExplain()：用于分析常量，对应＜常量说明＞ :: = const＜常量定义＞; { const＜常量定义＞; }

void constDefine(int type) ：用于分析常量定义，传入的type代表常量的类型，对应＜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＞ }

void declareHead()：用于分析声明头部，对应＜声明头部＞ :: = int＜标识符＞ | char＜标识符＞，传入的type为该常量的数据类型

void returnFuncDefine()：用于分析有返回值函数定义，对应＜有返回值函数定义＞ :: = ＜声明头部＞‘(’＜参数＞‘)’ ‘ { ’＜复合语句＞‘ }’

void parameterTable()：用于分析参数表，对应＜参数表＞ :: = ＜类型标识符＞＜标识符＞{ ,＜类型标识符＞＜标识符＞ } | ＜空＞

void noReturnFuncDefine()：用于分析无返回值函数定义，对应＜无返回值函数定义＞ :: = void＜标识符＞‘(’＜参数＞‘)’‘ { ’＜复合语句＞‘ }’

void MainDefine()：用于分析主函数，对应＜主函数＞ :: = void main‘(’‘)’ ‘ { ’＜复合语句＞‘ }’

void variableExplain()：用于分析变量说明，对应＜变量说明＞ :: = ＜变量定义＞; {＜变量定义＞; }

void variableDefine()：用于分析变量定义，对应＜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＞‘]’ }

void complexStatement()：用于分析复合语句，对应＜复合语句＞ :: = ［＜常量说明＞］［＜变量说明＞］＜语句列＞

void statementList()：用于分析语句列，对应＜语句列＞ :: = ｛＜语句＞｝

void statement()：用于分析语句，对应＜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＞;

void valueOfParameterTable()：用于分析值参数表，对应＜值参数表＞ :: = ＜表达式＞{ ,＜表达式＞ }｜＜空＞

void expression()：用于分析表达式，对应＜表达式＞ :: = ［＋｜－］＜项＞{ ＜加法运算符＞＜项＞ }

void item()：用于分析项，对应＜项＞ :: = ＜因子＞{ ＜乘法运算符＞＜因子＞ }

void factor()：用于分析因子，对应＜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＞‘)’

void assignStatement()：用于分析赋值语句，对应＜赋值语句＞ :: = ＜标识符＞＝＜表达式＞ | ＜标识符＞‘[’＜表达式＞‘]’ = ＜表达式＞

void ifStatement()：用于分析条件语句，对应＜条件语句＞ :: = if ‘(’＜条件＞‘)’＜语句＞［else＜语句＞］

void condition(int conditionType) ：用于分析条件，传入的conditionType代表是进行for循环的条件分析还是进行if语句的条件分析，对应＜条件＞ :: = ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞，

void forStatement()：用于分析for循环语句，对应＜循环语句＞ :: = for‘(’＜标识符＞＝＜表达式＞; ＜条件＞; ＜标识符＞＝＜标识符＞(+| -)＜步长＞‘)’＜语句＞

void switchStatement()：用于分析情况语句，对应＜情况语句＞ :: = switch ‘(’＜表达式＞‘)’ ‘ { ’＜情况表＞＜缺省＞‘ }’

void situationTable(string default\_label\_final,string switchResult,int SwitchVarType)：用于分析情况表，对应＜情况表＞ :: = ＜情况子语句＞{ ＜情况子语句＞ }，其中default\_label\_final传入生成对应的default语句结束位置的标签，switchResult为switch中表达式的运算结果，SwitchVarType为运算结果的类型

void situationStatement(string default\_label\_final, string switchResult, int SwitchVarType)：用于分析情况子语句，对应＜情况子语句＞ :: = case＜常量＞：＜语句＞，default\_label\_final传入生成对应的default语句结束位置的标签，switchResult为switch中表达式的运算结果，SwitchVarType为运算结果的类型

void defaultStatement()：用于分析缺省，对应＜缺省＞ :: = default: ＜语句＞

void scanfStatement()：用于分析读语句，对应＜读语句＞ :: = scanf ‘(’＜标识符＞{ ,＜标识符＞ }‘)’

void printfStatement()：用于分析写语句，对应＜写语句＞ :: = printf ‘(’ ＜字符串＞, ＜表达式＞ ‘)’ | printf ‘(’＜字符串＞

void returnStatement()：用于分析返回语句，对应＜返回语句＞ :: = return[‘(’＜表达式＞‘)’]

**错误处理：error.cpp**

涉及到的函数如下：

void error(int num, int dealType = -1);

void dealWithError(int operationType);

bool isNotInTable(ErrorItem erroritem, ErrorItem errortable[]);

分别的功能为：

void error(int num, int dealType = -1):依据传入的num以及dealType设置不同的错误处理方法。

void dealWithError(int operationType):依据传入的operationType的错误处理方式来进行不同的错误处理，进行跳读等操作。

bool isNotInTable(ErrorItem erroritem, ErrorItem errortable[]):判断传入的erroritem是否在errortable数组中，若已存在，返回false；否则返回true。

**中间代码生成：midcode.cpp**

涉及到的函数如下：

void insertMiddleCode(string op, string a, string b, string c);

string genNewRegister();

string genNewLabel();

分别的功能为：

void insertMiddleCode(string op, string a, string b, string c) ：插入四元式到四元式数组，其中传入的四个参数为四元式的组成规则

string genNewRegister()：生成新的临时变量，临时变量的开头为$，便于之后转换成mips的目标代码

string genNewLabel()：生成新的标签

**代码优化：optimize.cpp**

涉及到的函数如下：

void dag();

void deleteLabel();

void deleteVarInVector(string varName);

void divideBasicBlock();

void exportDagToMidcode();

int findInDag(string op, int i, int j);

bool isNumber(string str);

void mergeConstant();

void optAllExpression();

bool parentIsInStack(DagItem\* tempDagItem);

string realName(string str, DagItem \*tempItem);

分别的功能为：

void dag():调用该方法，实现DAG图的构建以及目标四元式的导出功能。

void deleteLabel()：删除冗余标签，将相邻的genLabel标签除了最后一个生成标签外，其余均删除，并且将跳转到被删除标签的指令的标签换为最后一个标签

void deleteVarInVector(string varName)：删除容器中的指定元素，varName传入要删除的元素的名字

void divideBasicBlock()：划分基本块,将皆有跳转功能的语句作为基本块的结束

void exportDagToMidcode()：依据DAG图，将DAG图中的信息导出为四元式代码

int findInDag(string op, int i, int j)：查找op对应的节点是否在DAG图中，若在，返回对应的节点标号；若不在，返回-1

bool isNumber(string str)：//判断输入的字符串str是否为数字

void mergeConstant()：合并常数的运算，将运算中的常数运算改为运算后的数值结果的赋值运算

void optAllExpression()：扫描所有的四元表达式，对中间代码四元式进行优化

bool parentIsInStack(DagItem\* tempDagItem)：若传入的子节点的父节点均在队列中，则返回true；否则返回false；若为叶结点，直接返回false

string realName(string str, DagItem \*tempItem)：得到叶节点的真正的数据名字，包含常数、变量等。

**目标代码生成：mips.cpp**

涉及到的函数如下：

void Assign2Mips();

void AssignArr2Mips();

void calculation2Mips(int type);

void call2Mips();

void compare2Mips(int type);

bool compareVector(const CountStruct &a, const CountStruct &b);

void const2Mips();

void countNumOfVar(string varName);

void CountReg();

int findVarInVariableTable(string name);

void funcContent2Mips();

void funcExplain2Mips();

void funcPa2Mips();

string genCompareLabel();

void genLabel2Mips();

void getArr2Mips();

void globalConst2Mips();

void globalIntOrChar2Mips();

void globalIntOrCharArray2Mips();

void insertIntoFuncTable(int kind, string name);

void insertIntoVariableTable(int kind, int DataType, int address = -1, bool isGlobalData = false);

void intOrChar2Mips();

void intOrCharArray2Mips();

bool isVarName(string name);

void jne2Mips();

void jump2Mips();

void loadSRegisters(string functionName);

void midCode2Mips();

void para2Mips();

void ptf2Mips();

void pushIntoStack(string value, int size = 1, bool isGlobalData = false);

void ret2Mips();

void saveSRegisters(string functionName);

void scf2Mips();

bool VarUseRegister(string FuncName, string varName, string tempRegister, bool needSw = false);

分别的功能为：

void Assign2Mips()：变量赋值转换为mips指令

void AssignArr2Mips()：数组赋值转换为mips指令

void calculation2Mips(int type)：加减乘除运算符转换为mips指令

void call2Mips()：调用函数四元式转换为mips指令

void compare2Mips(int type)：比较运算符转换为mips指令

bool compareVector(const CountStruct &a, const CountStruct &b):自定义比较函数(从大到小排序)，用于结构体的比较

void const2Mips()：局部常量四元式转换为mips指令

void countNumOfVar(string varName)：为传入的变量计数，传入的varName为要进行统计的变量名字

void CountReg()：引用计数方法，调用该方法，则开始执行引用计数方法来分配寄存器

int findVarInVariableTable(string name)：查找输入的变量名字是否在变量表中（没找到返回-1，找到返回地址（4的倍数（有正有负），相对地址））

void funcContent2Mips()：处理函数中的部分，转换为mips指令

void funcExplain2Mips()：函数声明四元式转换为mips指令

void funcPa2Mips()：实参四元式转换为mips指令

string genCompareLabel()：为执行的跳转语句生成跳转标签并返回

void genLabel2Mips()：生成标签四元式转换为mips指令

void getArr2Mips()：获取数组元素转换为mips指令

void globalConst2Mips()：全局常量声明转换为mips指令

void globalIntOrChar2Mips()：全局变量（非数组）转换为mips指令

void globalIntOrCharArray2Mips()：全局变量（数组）转换为mips指令

void insertIntoFuncTable(int kind, string name)：将传入的函数加入函数表中，kind为函数的返回值类型，name为函数名字

void insertIntoVariableTable(int kind, int DataType, int address = -1, bool isGlobalData = false)：将传入的变量插入到变量表中，便于之后的寻址操作，kind为插入的数据类型，DataType 为传入的变量的数据类别，address默认为-1，若-1，则存储相对地址，否则存储传入的地址，isGlobalData代表插入的是全局数据还是局部数据，默认为false

void intOrChar2Mips()：局部变量（非数组）四元式转换为mips指令

void intOrCharArray2Mips()：局部变量（数组）四元式转换为mips指令

bool isVarName(string name):判断是否为变量的名字，如果是空的或者是数字，则返回false，否则是变量名字,返回true

void jne2Mips()：不满足条件跳转四元式转换为mips指令

void jump2Mips()：无条件跳转的四元式转换为mips指令

void loadSRegisters(string functionName):恢复$s0-$s7寄存器中的数值

void midCode2Mips()：将中间指令转换为mips指令

void para2Mips()：参数声明四元式转换为mips指令

void ptf2Mips()：写语句转换为mips指令

void pushIntoStack(string value, int size = 1, bool isGlobalData = false)：将传入的数据压入栈中,压入sp或者gp指针位置，传入的value为要压入栈的数值，size标识传入的元素大小，默认为1，isGlobalData代表压入栈的数据为全局数据还是局部数据，默认为false

void ret2Mips()：返回语句转换为mips指令

void saveSRegisters(string functionName):保存$s0-$s7寄存器中的数值

void scf2Mips()：读语句转换为mips指令

bool VarUseRegister(string FuncName, string varName, string tempRegister, bool needSw = false):在函数的寄存器分配map中查找，若存在该变量，则使用分配的寄存器即可，无需再lw或者sw

**主函数：main.cpp**

涉及到的函数如下：

void openFile();

void setup();

分别的功能为：

void openFile():打开要进行编译的文件

void setup():建立表信息，进行初始化工作

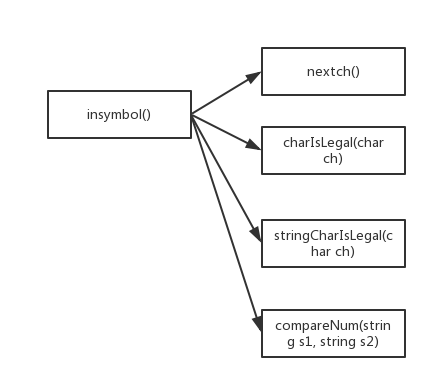
**全局变量头文件：globalData.h**

功能为定义一些全局变量、函数声明以及结构体，方便数据的管理以及使用。

### 3．调用依赖关系

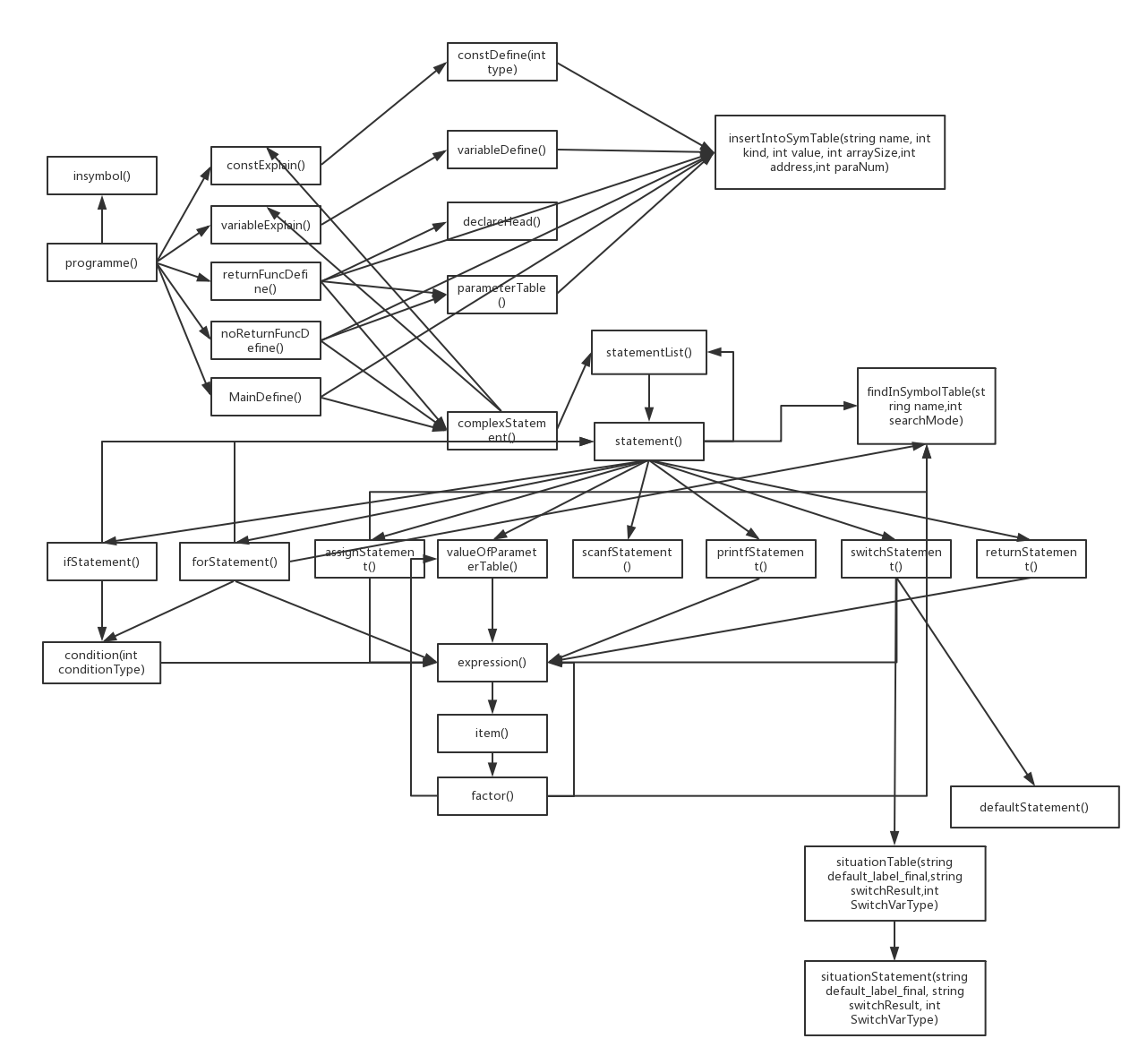
**词法分析：lexical.cpp**

涉及到的函数调用依赖关系如下：



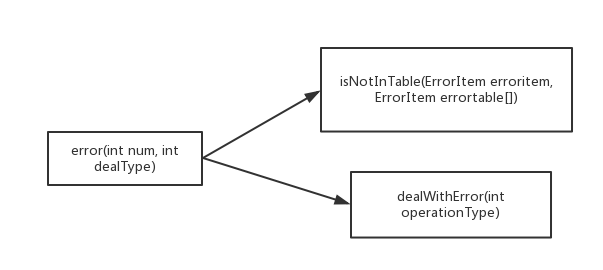
**语法分析：syntax.cpp**

涉及到的函数调用依赖关系如下：



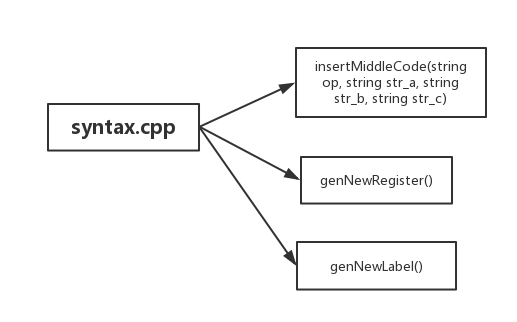
**错误处理：error.cpp**

涉及到的函数调用依赖关系如下：



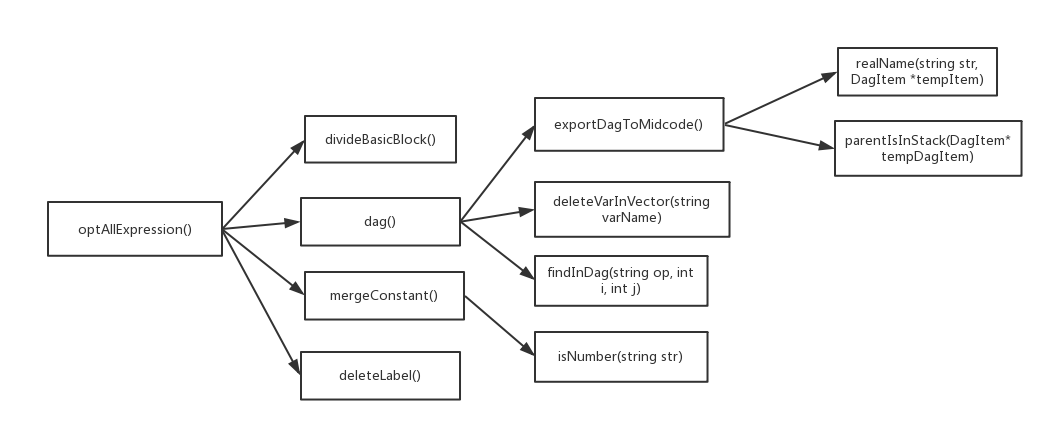
**中间代码生成：midcode.cpp**

涉及到的函数调用依赖关系如下：



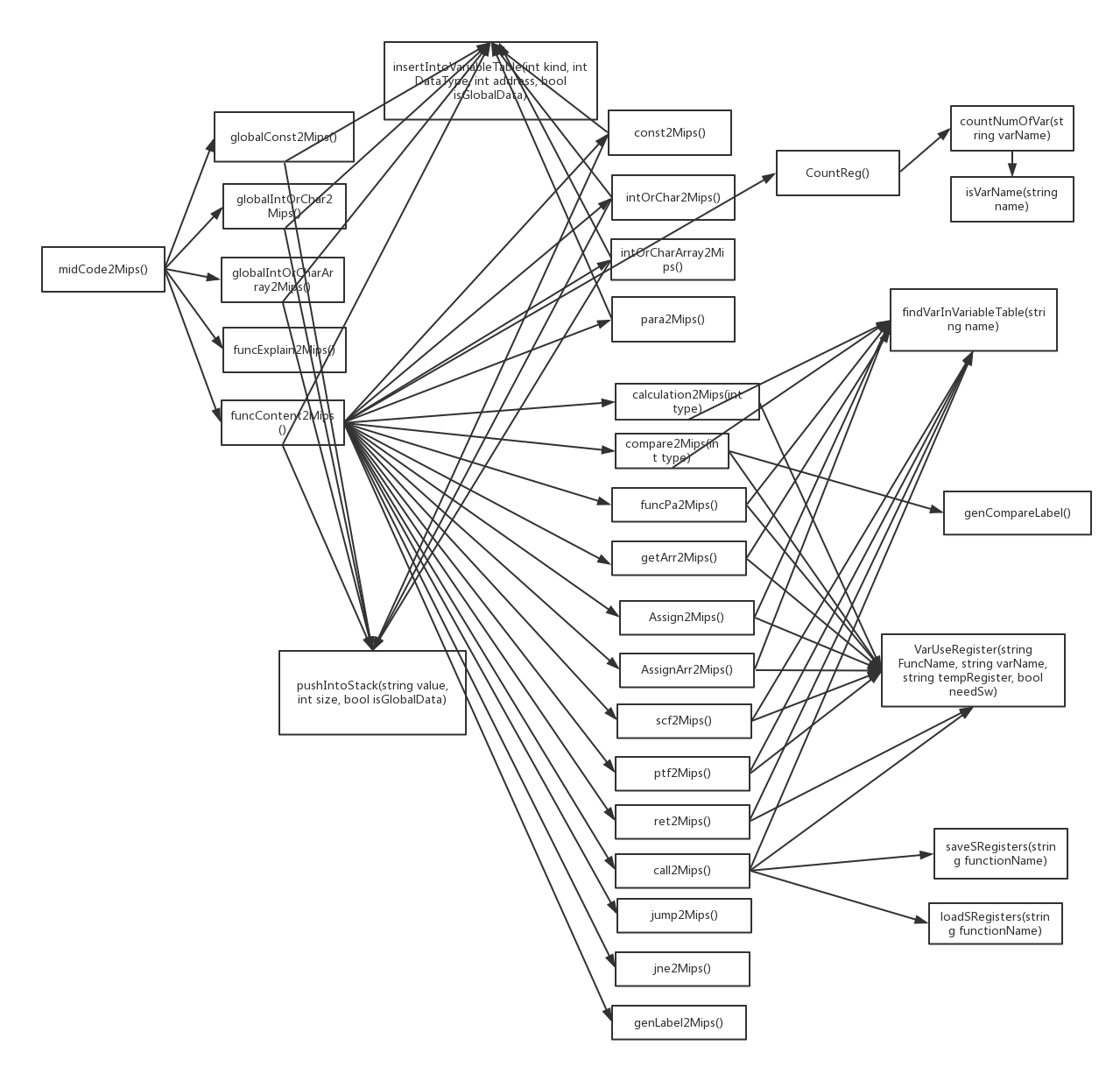
**代码优化：optimize.cpp**

涉及到的函数调用依赖关系如下：



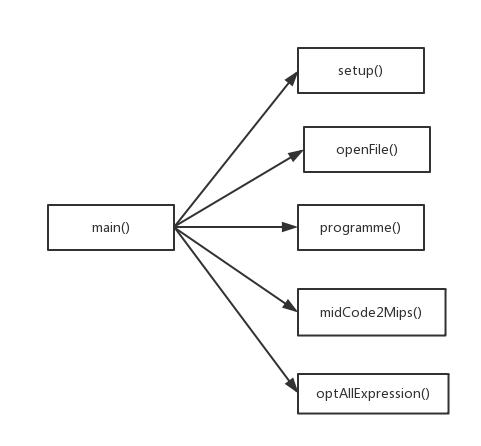
**目标代码生成：mips.cpp：**

涉及到的函数调用依赖关系如下：



**主函数：main.cpp**

涉及到的函数调用依赖关系如下：



### 4．符号表管理方案

符号表项结构：

typedef struct {

string name; //标识符名字

int kind; //标识符的类别

int value; //数值存储，若为函数，则有返回值存储1，无返回值存储0

int arraySize; //如果是数组的话，存储数组的size大小，其他情况size为0；

int address; //变量的地址；函数返回值的地址

int paraNum; //若为函数，代表参数个数，否则为0；

} signTable; //符号表项结构

符号表：

signTable symbolTable[maxTableSize]

分程序索引表结构：

int displayTable[maxTableSize];

对于符号表，各个种类的数据字段存储形式如下：

对于非数组变量：

name为变量名字，kind = charsy | intsy， address = symbolTableTop， arraySize = 0, paraNum = 0, value = 0

对于数组变量：

name为数组名字，kind = charsy | intsy， address = symbolTableTop， arraySize为数组大小, paraNum = 0, value = 0

对于常量：

name为常量名字，kind = charconst | intconst， address = symbolTableTop， arraySize = 0, paraNum = 0, value为常量数值

对于参数：

name为参数名字，kind = PARAMETER， address = symbolTableTop， arraySize = 0, paraNum = 0, value = intsy | charsy

对于函数：

name为函数名字，kind = FUNCTION， address = symbolTableTop， arraySize = 0, paraNum为参数个数, value = voidsy | intsy | charsy

displayTable[maxTableSize]分程序索引数组存储每个函数在符号表中的地址，通过该分程序索引表即可快速查找到与该函数有关数据信息的地址。

### 5．存储分配方案

存储方式初定采用引用计数算法或者启发式图着色算法对于寄存器进行分配，用分配到的寄存器来存储相应数据。

运行栈结构如下图所示：

|  |
| --- |
| 栈顶 |
| 上一级函数fp  函数返回地址ra  函数临时变量区  函数变量区  实参数区域（如果有实参） |
| 上一级函数fp  上一级函数返回地址ra  函数临时变量区  函数变量区  实参数区域（如果有实参） |
| `````` |
| 全局数据区 |
| `````` |
|  |

### 6. 解释执行程序\*

此编译器不采用解释执行方式，故无需考虑解释执行。

### 7. 四元式设计\*

四元式组成的规则为：

操作符号，标识符，标识符，标识符

对于只需要四个以下的元素即可表示的四元式，可传入空字符代表不需要该位置的符号

四元式种类：

常量声明：

const, int/char, 数值, 常量标识符名字

数组声明：

intArr/charArr, 0, 数组大小, 数组名字

变量声明：

int/char, , , 变量名字

函数声明：

func, int/char/void, , 函数名字

函数结束：

end, , , 函数名字

参数声明：

para, int/char, , 参数名字

函数调用（不需要将结果存储到寄存器中）:

call, 函数名字， ，

函数调用（需要用寄存器存储）

call, 函数名字， ，寄存器变量

值参数（实参）：

funcPa， ， ，实参存储的寄存器

运算(结果存储到r3寄存器):

+、-、\*、/, r1, r2, r3

条件运算符(<、<=、>、>=、!=、==)：

以 “<” 举例：

lss, r1, r2/数字，

得到数组元素：

getArr, 数组名字， 数组下标， 结果存储的寄存器

数组赋值：

[]=, 数组名字， 数组下标， 要存储的数值

变量赋值：

=， 要赋的数值， ， 变量标识符名字

不满足条件跳转：

jne， ， ，label

无条件跳转：

jump， ， ，label

生成标签：

genLabel， ， ，label

读语句：

scf， int/char， ， 变量标识符

写语句：

ptf，字符串（可以为空）， 结果寄存器，打印类型 int/char

返回语句（无返回值）：

ret， ， ，

返回语句（有返回值）：

ret， ， ，保存结果的寄存器

### 8. 目标代码生成

### 方案\*

目标代码为MIPS指令，在MARS中即可编译运行。

### 9. 优化方案\*

优化方法分为以下几种：

DAG图优化来消除公共子表达式的局部优化、常数合并优化、冗余代码删除以及寄存器分配等等。

一、DAG图优化方法，采用书中的算法构建DAG图，而后使用启发式算法将DAG图导出优化后的四元式代码。

用到的结构体：

//DAG图节点结构体：

struct Dagitem {

string name;

int num;

vector<string> varItems;

vector<Dagitem\*> parent;

Dagitem\* lchild;

Dagitem\* rchild;

//是否已经添加到队列中

bool isInStack;

};

首先要进行基本块的划分，划分方法参照书上的方法（三条规则确定入口语句），以函数为单位，进行基本块的划分，将基本块的划分结果存入basicBlockVector容器中。而后使用DAG图，对于+、-、\*、/、数组取值进行DAG图节点的构建，map<string, int> DagItemTable为节点表，map<int, DagItem\*> dagIndexMap为DAG图树状结构的索引map，通过节点编号即可找到树中对应的节点。DAG图构建完成后通过书上的启发式算法，将DAG图的信息导出为中间式中间代码。

二、常数合并优化：合并常数运算将+ - \* / 指令的var1 var2两个操作数均为常数的指令进行运算得到运算结果，而后合并常数转换为赋值语句。例如加法：+, r1, r2, r3， 若 r1，r2为常数，则合并为 = , (r1+r2), , r3

三、冗余代码的删除：删除冗余标签，将相邻的genLabel生成标签语句，除了最后一个语句生成标签外，其余生成语句均删除，并且将跳转到被删除标签的指令的part\_3标签替换为最后一个标签。

四、寄存器分配：通过引用计数法进行寄存器分配，将

用到的结构体：

//对应变量在变量表中的编号以及出现数量

typedef struct {

string name;

int number;

int address;//记录相对偏移量

int count; //变量出现的个数

}CountStruct;

//对应变量在变量表中的编号以及要分配的寄存器

typedef struct {

string name;

int number;

int address;//记录相对偏移量

int regNum; //分配的寄存器编号

}RegStruct;

引用计数法以函数为单位进行局部变量出现次数的统计，将统计结果存入一个map容器FunctionRegMap中，该容器键为函数的名字，值为函数中变量的分配情况vector容器，即为map<string, vector<RegStruct>> FunctionRegMap，通过该map即可通过对应关系将对应的局部变量存入寄存器中，减少程序对于内存的访问次数，实现优化。

### 10. 出错处理

对于不同的错误，error()方法在出现错误的地方进行跳读，直到读到下一个可以继续分析的单词截止，同时在控制台打印出错信息。

enum errorType {

NoSuchFile, LackDoubleQuotes, LackSingleQuote, LackSemicolon, LackComma, LackColon, LackLSmallBrack,LackRSmallBrack, LackRMidBrack, LackLBigBrack, LackRBigBrack, LackIdent,LackReturnValue, LackPlusOrMinus, LackNum, LackBecome, NotIntOrChar, MainNumNotOne,CharIllegal, StringIllegal, FunctionNotDefine, VariableNotDefine,KeywordError, TableFull, SymbolNameConflict, CodeAfterMain, FormalParaNumNotMatch, ParaTypeNotMatch,AssignWithNoReturnFunc, AssignConst, NumOutOfRange, AssignTypeNotMatch, BeginWithZeroButNotZero, CharLenBigThanOne,LexicalIlegalCh, ConstDefAfterVarDef, SignBeforeZero, SignBeforeChar, DataTypeNotMatch, NoReturnFuncWithReturn, ArrayIndexNotUnsignedNum,ArraySizeIsZero, StatementIsIllegal, StatementIllegalSign, ArrayIndexOfRange, FactorIllegalSign, TypeNotArray, ArrayIndexNotIntType,forConditionNotContainVar, forUpdateNotContainVar, StepIsZero, notConst, SwitchTypeNotMatch , AssignArray, DivideZero

};

现阶段主要的错误分为没有该文件、缺少双引号、缺少单引号、缺少分号、缺少逗号、缺少冒号、缺少左小括号、缺少右小括号、缺少右中括号、缺少左大括号、缺少右大括号、缺少标识符、缺少返回值、缺少加减号、缺少数字的声明、缺少 =号，不是int或者char的类型标识符、main函数个数不是1、字符由非法字符组成、字符串由非法字符组成、使用的函数未定义、使用的变量未定义、关键字错误、表满了（多种表，符号表，分程序索引表，四元式表等）、符号表插入数据重名、main函数后还有代码、形参个数不匹配、参数类型不匹配、无返回值函数作为当做右值、给常量赋值、整数范围过大、赋值语句两边类型不对、数字以0开头且不是0这个整数、读取到的字符长度大于1、词法分析程序读取到非法的字符（不符合任何单词的构成规则）、常量声明在变量后面、0前面有符号、字符前面有符号、变量或者常量声明时候的类型不匹配、无返回值函数却有返回值、（定义时）数组下标不为无符号整数、数组大小为0、句子不合法（例如某句子应该为赋值或者函数调用语句，却都不是等）、语句处理时均不符合条件、数组下标越界、因子处理时均不符合条件、类型不为数组、数组调用时表达式结果不为整型、for循环判断条件与循环变量无关、for循环更新条件与循环变量无关、步长为0、不为常量、switch表达式类型与case判断的类型不匹配、数组名字赋值、除数为0等等。

不同错误的处理方案如下表所示：

|  |  |  |
| --- | --- | --- |
| 处理方案 | 对应的标识号 | 处理方法 |
| Quit | 0 | 程序退出，结束 |
| DoNothing | 1 | 不做处理，继续分析 |
| IntCharVoid | 2 | 读取int/char/void |
| ConstIntChar | 3 | 读取到const/int/char |
| CommaSemicolon | 4 | 读取到逗号或者分号 |
| LSmallBrack | 5 | 读取到左小括号 |
| RBigBrack | 6 | 读取到右大括号 |
| IfForLBigBrackICVISPSRS | 7 | 读取到if/for/int/char/void/return/scanf/printf/switch/左大括号/分号/标识符 |
| PlusMinusTimesDivSemicolon | 8 | 读取到加减乘除或者分号 |
| SemiColon | 9 | 读取到分号 |
| RSmallBrack | 10 | 读取到右小括号 |
| CaseDefaultRBigBrack | 11 | 读取到case或者default或者右大括号 |

对于程序中的错误，会依据错误类型转到对应的错误处理方案，直到程序处理结束。

## 三．操作说明

### 1．运行环境

运行环境为VS2010，Mars4.5

### 2．操作步骤

打开main.cpp文件，编译后运行，在控制台输入要编译的文件路径即可，在程序成功运行后，控制台会输出目标代码已经生成的提醒；若出现错误，控制台也会提示错误出现的位置。

## 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：

测试1（正确）

程序的功能是输入一个整型数字，输出其斐波那契数，对于负数或者输入大于最大全局变量MAX的数据，函数返回-1；对于输入0，函数返回0；对于输入1，函数返回1；对于大于等于2的输入，返回其斐波那契数，并在主程序打印结果，结果正确。

输入 输出

-2 -1

0 0

60 -1

1 1

4 3

测试2（正确）

程序的功能是输出给定的由多种类型字符组成的字符串（包括空字符串）。

输入：无需输入

输出： !#$&()+-\*/,.?abcdefgABCDEFG|[-^]{}1234567890

测试3（正确）

程序的功能是按照给定的数组进行从小到大排序，最后逆序输出排序后数组的元素（按照从大到小的顺序）。

输入：无需输入

输出： 5 4 3 2 1

测试4（正确）

程序的功能是输入一个整型数代表操作类型以及两个整型操作数，依据输入的整型操作数进行分析，若为0，则进行加法运算，输出'+'并返回结果；若为1，进行减法运算，输出'-'并返回结果；若为2，进行乘法运算，输出'\*'并返回结果；若为3，进行除法运算，若除数不为0，则输出'/'并返回结果，否则输出can't divide zero!提示并返回-1；若输入上述条件都不满足，则输出can't recognise the input!，返回-1。

输入 输出

0 1 2 + 3

1 10 2 - 8

2 3 5 \* 15

3 6 3 / 2

3 4 0 can't divide zero! -1

4 2 3 can't recognise the input! -1

测试5（正确）

本程序的功能是对于给定的a, b, c以及数组元素，对其进行各种表达式的运算，最后输出所有的运算结果，运行一条空语句，结果正确。

输入：无需输入

输出： 6 -3 7 5 5 -5 8 31 99 c \_

错误程序：

测试6（错误）

根据文法，常量定义时必须赋初值，本程序中常量num\_1声明后未赋初值；变量声明时不能赋值，故num\_2赋初值报错；报错；读语句调用错误，报错；有返回值函数缺少返回语句，报错。

测试7（错误）

根据文法，ch被赋值为非法字符，报错；1num标识符声明以未以字母开头，故报错。

测试8（错误）

程序中foo无返回值函数却返回整型数字结果，报错； foo规定参数为2个，且参数类型为整型，但是其中一个调用语句只传入一个整型参数，报错；一个调用语句传入两个字符型变量，报错。

测试9（错误）

程序定义了一个能够存储5个整型数字的array数组，利用循环语句对数组内容进行赋值；数组大小为5，对于array[5]，下标越界，报错；数组下标不为整型而传入浮点型，报错。

测试10（错误）

程序中，使用的num\_1变量未定义，报错；for循环语句在更新循环变量时，使用乘号，报错。

### 2．测试结果分析

测试1：

本程序重点测试了有返回值的函数调用（包括函数递归），同时还包括对if条件语句，关系运算符的测试等等。

测试2：

本程序重点测试了对于字符串组成规则以及字符串对应的输出格式。

测试3：

本程序重点测试了无返回值的函数调用，同时包含对for循环语句、写语句、返回语句的测试。

测试4：

本程序重点测试了情况语句、加法运算符、乘法运算符、赋值语句以及读语句。

测试5：

本程序重点测试了对表达式、项、因子的各种表示形式以及空语句的使用。

测试6：

本程序主要测试声明变量，常量声明以及使用时的一些错误，包括常量定义时未赋值，变量声明时赋初值,读语句调用格式错误，有返回值函数缺少返回语句等错误。

测试7：

本程序主要测试词法分析时候的错误，包括字符为非法字符，标识符声明以数字开头等错误。

测试8：

本程序主要测试与函数定义以及使用有关的错误，包括无返回值函数返回值，传入函数的参数个数不匹配，传入函数的参数类型不匹配等错误。

测试9：

本程序主要测试与数组有关的错误，包括数组越界，数组下标不是整数等错误。

测试10：

本程序主要测试与for语句以及变量有关的错误，包括for循环语句更新循环变量的符号错误，使用未声明的标识符等错误。

上述程序分支情况覆盖完全。

## 五．总结感想

在刚开始学习编译原理课程的时候，我学习到的编译有关知识仅仅局限于理论方面，对于许多知识不是很理解。经过了一个学期的时间，在完成了编译课程设计中的C0编译器设计后，我对于课本上的许多知识进行了代码上的实现，对于书中的知识也有了深入的理解。我收获了许多方面的知识，包括编译的若干阶段的具体流程、程序运行时候的运行栈的管理以及空间分配、对于MIPS体系的再一次深入理解等等。

在刚开始进行编译课程设计的时候，我感觉无从下手，所谓万事开头难，我总是觉得要从头开始搭建一个编译器的难度真的是非常的大。不过课程设计的体系考虑到了我们会在这方面遇到的问题，便将完成一个编译器的任务分成了按照周的计划来一步步实现，这为我的编译器课程设计指引着前进的方向。

编译器的设计阶段分为以下几个部分：词法分析、语法分析、语义分析以及中间代码生成、代码优化以及目标代码生成，中间贯穿着符号表的管理以及出错处理。其中词法分析相对简单，不断地读取单词供语法分析程序使用即可。而重点在于语法分析，一定要读懂自己的文法然后在正确理解文法的基础上，依据文法编写递归下降分析程序，这个过程非常重要，一旦有理解有误的地方，之后的调试工作会变得很复杂。当然语法分析过程中伴随着语义上的检查，此时就要涉及到符号表的设计，一个好的符号表意味着用较少的空间来存储更多有意义的信息以及能够用较少的时间来进行相关信息的查询获取，我在符号表设计部分以及相关索引表的设计花费了不少的时间。中间代码生成我已开始以为是有一定难度的，但是在完成语法分析后，我发现中间代码只是一个从语法分析出的结果转化到目标代码的桥梁，只要设计好四元式的格式以及保证覆盖完全语法成分即可，在语法分析得到相应的语法成分后向四元式表插入相关的四元式便可完成中间代码的生成，难度并不是很大。再之后我完成的是目标代码的生成，相比于提高目标代码的效率性，目标代码运行的正确性更为重要，先保证目标程序的正确性才是首要任务。目标代码生成的难度还是有一些的，运行栈的设计以及空间分配、对于MIPS体系的理解以及对于程序运行的控制都需要我们花费一定的时间思考，特别是运行栈的设计，运行栈与程序运行的正确性有着极大的关系。最后一部分则是代码优化了，代码优化依据书中的算法来实现，涉及到DAG的优化，寄存器的分配，常数合并以及冗余代码的删除等。整个设计流程中还涉及到出错处理，为不同的错误定义相应的错误类别以及解决方案，遇到相应错误则传入该错误，而后调用相应的解决方案即可。

一个学期的时间，理论课加上实验课的学习使我对编译技术有了深刻的理解，我也懂得了一个道理：不要过于畏惧之前没有接触过的内容，一切都着眼于当下，一步一步地按照计划去实现，当走过这一阶段后，你会发现自己能够完成的工作任务超乎你的想象，在这个过程中，你也会收获颇丰！