一、C-语言的语法图描述

1、C-语言的关键字

else if int return void while

2、C-语言专用符号：

+ - \* / < <= > >= == != = ; , ( ) [ ] { } /\* \*/

3、其它标记：

ID=letter letter\*

NUM=digit digit\*

letter= a|..|z|A|..|Z

digit=0|..|9

小写与大写有区别

4、空格有空白、换行符和制表符组成。空格通常被忽略，除了它必须分开ID、NUM关键字

5、注释用通常的C语言符号/\*…\*/围起来。注释可放在任何空白出现的位置(即注释不能放在标记内)上，且可以超过一行，注释不能嵌套

6、C-的BNF文法:

S -> program

program -> declaration-list

declaration-list -> declaration-list declaration | declaration

declaration -> var-declaration | fun-declaration

var-declaration -> type-specifier ID ; | type-specifier ID [ NUM ] ;

type-specifier -> int | void

fun-declaration -> type-specifier IDF ( params ) | compound-stmt

params -> params-list | void

params-list -> params-list , param | param

param -> type-specifier ID | type-specifier ID [ ]

compound-stmt -> { local-declarations statement-list }

local-declarations -> local-declarations var-declaration | empty

statement-list -> statement-list statement | empty

statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt

expression-stmt -> expression ; | ;

selection-stmt -> if ( expression ) statement | if ( expression ) statement else statement

iteration-stmt -> while ( expression ) statement

return-stmt -> return ; | return expression ;

expression -> var1 = expression | simple-expression

var -> ID | ID [ expression ]

var1 -> ID1 | ID1 [ expression ]

simple-expression -> additive-expression relop additive-expression | additive-expression

relop -> <= | < | > | >= | == | !=

additive-expression -> additive-expression addop term | term

addop -> + | -

term -> term mulop factor | factor

mulop -> \* | /

factor -> ( expression ) | var | call | NUM

call -> IDF ( args )

args -> arg-list | empty

arg-list -> arg-list , expression | expression

二、系统设计

1、系统的总体设计



一共分成三个主要部分：词法分析，语法和语义分析，代码优化。

其中词法分析使用的是状态机，到达某一终态则截取此字符序列，判定其为此段序列的种类，并输出。

语法和语义分析使用的是LL(1)分析，首先对所需要的语法进行了手动预处理，因为LL(1)不允许出现左递归，所以要消去原本文法中的所有左递归。（关于答辩时所说的存在左递归的问题我们经过重新测试发现原来的语法是可以的，当时出现死循环是因为当时的代码在处理局部变量符号表时出现了问题，我们已经在最新的代码中做了修正）。语义分析时首先载入文法并消去“或”符号（|），将处理后的文法输出到文件，接下来在产生式中插入语义动作（我们只插入标签，具体动作在程序中定义），接下来计算First集和Follow集，然后构造预测分析表M，然后用总控程序不断读入代码。当处理的栈顶为语义动作时执行对应的语句。

最终输出起始符号的code字段即可得到翻译完成的三地址码。

代码优化主要是生成DAG图做局部优化。

2、主要功能模块的设计————词法分析

词法分析系统：

词法分析阶段是编译过程的第一个阶段，是编译的基础。这个阶段的任务是从左到右一个字符一个字符地读入源程序，即对构成源程序的字符流进行扫描然后根据构词规则识别单词(也称单词符号或符号)。而其中的构词规则我们用C-语言规定的操作符、关键字等规定的规则借助状态机进行针对不同程序的词法分析的匹配。

输入：源程序

输出：单词符号Token序列

符号分为：

(1)COP比较操作符 < <= > >= == !=

(2)AOP赋值操作符 =

(3)OOP运算操作符 + - \* /

(4)EOP句末操作符 ;

(5)SOP结构分隔符 ( ) , [ ] { }

(6)RESERVED关键字种类有：

else 、if、int、return、void、while

3、系统运行流程

具体操作：

* 预处理：在pretreat.cpp中和pretreat.h中将源代码中的多余空格换行符以及注释去掉。
* 词法分析：在lexical-analysis.h和lexical-analysis.cpp中利用状态机进行对每个单词按照规则进行匹配。

匹配规则：



状态转换图：



这里主要用lexicalanalysis（fstream&，fstream&）函数进行词法分析，首先读入存储源程序代码文件，从头开始一个字符一个字符的读取，直到文件结尾。其中如果遇到空白字符则跳过，列数加一，如果非空白字符则一直读取一直到遇到下一个空格为止。判断刚才得到的字符串是否是在匹配规则中保留字，如果是则输出所在行列和保留字信息，如果不是则输出是变量的信息以及行列。如果第一个字符不是字母是数字，则读取数字串标为num以及其行列信息。如果不是字母和数字，是其他符号，则依次匹配是否是COP、EOP、SOP、SOP中对应的操作符，如果是则打印出其行列以及其匹配相关信息，其中要注意的是像“<”和“<=”这样的操作符在匹配的时候要先向后多读一个来判断是哪个，如果不是较长的要回退一个读指针。

* 检错

在词法分析过程中如果源程序有明显的逻辑错误，则会在运行时显示[Lexical ERROR]报错信息。例如，如果源程序中非字母和数字的操作符在匹配规则中没有的话，则输出行列报错信息；如果“！”之后没有出现“=”的话输出行列报错信息；如果有字母紧接着出现在数字之后的话输出行列报错信息。

三、系统实现

1、系统主要函数说明

void delzs(fstream &in, fstream &out);删除程序中所有注释

void delblank(fstream &in, fstream &out); 删除程序中所有多余的空格,tab以及回车

void delleftrecursion(fstream &in, fstream &out); 消除左递归以及公共左因子

void splitgrammar(fstream &in, fstream &out); 将语法根据|拆分

int isBC(char ch);判断是否为空白字符

int Reserve(string strToken);判断是否为保留字

void Retract(fstream &source);读取指针回退

void lexicalanalysis(fstream &in, fstream &out); 词法分析

void fixlex(void);ID分类函数

2、系统代码

Pretreat.h:

#pragma once

#include "global.h"

//使用有限状态自动机来实现功能

//start--status:0

//status:1 readin-ch

// '/' goto-2 --

// '\*' goto-1 output-ch

// other goto-1 output-ch

//status:2 readin-ch

// '/' goto-2 output-'/'

// '\*' goto-3 --

// other goto-1 output-'/'and ch

//status:3 readin-ch

// '/' goto-3 --

// '\*' goto-4 --

// other goto-3 --

//status:4 readin-ch

// '/' goto-5 --

// '\*' goto-4 --

// other goto-3 --

//status:5 readin-ch

// '/' goto-2 --

// '\*' goto-1 output-ch

// other goto-1 output-ch

//删除程序中所有注释

void delzs(fstream &in, fstream &out);

//start--status:0

//status:1 readin-ch

// /blank goto-2 --

// other goto-1 output-ch

//status:2 readin-ch

// /blank goto-2 --

// other goto-1 output-pre and ch

//删除程序中所有多余的空格,tab以及回车

void delblank(fstream &in, fstream &out);

//消除左递归以及公共左因子

void delleftrecursion(fstream &in, fstream &out);

//将语法根据|拆分

void splitgrammar(fstream &in, fstream &out);

pretreat.cpp:

#include "pretreat.h"

//in >> noskipws 读取文件中的空白字符

//状态转换表见pretreat.h

void delzs(fstream &in, fstream &out)

{

char temp ,pre;

string temp\_s=""; //存储可能为注释的字符串

int status = 0;

in >> noskipws;

while (!in.eof())

{

if (!status) status = 1;

in >> temp;

if (in.fail())

break;

switch (status)

{

case 1:

if (temp != '/')

{

out << temp; //cout << temp;

status = 1;

}

else

status = 2;

pre = temp;

break;

case 2:

if (temp == '/')

{

out << temp; //cout << temp;

status = 2;

}

else

if (temp != '\*')

{

out << '/' << temp; //cout << '/' << temp;

status = 1;

}

else

status = 3;

pre = temp;

break;

case 3:

if (temp == '\*')

status = 4;

else

status = 3;

temp\_s.push\_back(temp);

if (pre == '/' && temp == '\*')

{

cout << "出现注释嵌套！" << endl;

out << "[Error Occured!]";

SetConsoleTextAttribute(GetStdHandle(STD\_OUTPUT\_HANDLE), 7);

exit(0);

}

pre = temp;

break;

case 4:

if (temp == '/')

status = 5;

else

if (temp == '\*')

status = 4;

else

status = 3;

temp\_s.push\_back(temp);

pre = temp;

break;

case 5:

temp\_s = ""; //一段注释结束 清空暂存的可能为的注释信息

if (temp != '/')

{

out << temp; //cout << temp;

status = 1;

}

else

status = 2;

pre = temp;

break;

default:

cout << "An error occured!" << endl;

break; //进入default分支 说明发生比较神奇的错误

};

}

//如果最后注释格式右侧没有闭合 则需要将 尚为输出 正在判别的部分输出

if (status == 2)

{

out << "/"; //cout << "/";

}

if (status == 3 || status == 4)

{

out << "/\*" << temp\_s; //cout << "/\*" << temp\_s;

}

}

//状态转换表见pretreat.h

void delblank(fstream &in, fstream &out)

{

char temp, pre;

int status=0;

in >> noskipws;

in >> temp;

while (temp == ' ' || temp == '\t' || temp == '\n')

in >> temp;

out << temp; //cout << temp;

pre = temp; //去除程序前所有的空白字符，直到第一个非换行符空格或制表符

while (!in.eof())

{

in >> temp;

if (in.fail())

break;

if (!status) status = 1;

switch (status)

{

case 1:

if (temp == ' ' || temp == '\n' || temp == '\t')

status = 2;

else

{

out << temp; //cout << temp;

}

pre = temp;

break;

case 2:

if (temp != ' ' && temp != '\n' && temp != '\t')

{

status = 1;

out << pre << temp; //cout << pre << temp; //2状态接收到一个非空白字符 则输出最近的一个空白字符

}

if (pre!='\n')

pre = temp; //如换行与制表符或空格同时出现 保留换行 保证程序大体格式不被改变

break;

default:

cout << "An error occured!" << endl;

break; //进入default分支 说明发生比较神奇的错误

};

}

return;

}

//将字符串按照空格分割,并将各部分存入数组中

int splitblank(string str, string strsplit[])

{

int index, start, k = 0;

string t;

start = 0;

while (1) //将箭头后的部分按照空格分开

{

index = str.find(' ', start);

if (index == str.npos)

break;

strsplit[k++].assign(str, start, index - start);

start = index + 1;

}

strsplit[k++].assign(str, start, str.length() - start);

return k; //返回一共分为了几部分

}

//消除左递归以及公共左因子

void delleftrecursion(fstream &in, fstream &out)

{

string temp, part1, part2, p2\_1, p2\_2;

string str1[10], str2[10];

int k1, k2, flag;

int split, index;

while (!in.eof())

{

getline(in, temp);

if (in.fail())

break;

index = temp.find('|');

if (index != part2.npos)

{

split = temp.find("->");

part1.assign(temp, 0, split);

part2.assign(temp, split + 2, temp.length() - split - 1);

fix(part1);

fix(part2); //将每一行语法分为箭头前后的两个部分

index = part2.find('|');

p2\_1.assign(part2, 0, index);

p2\_2.assign(part2, index + 1, part2.length() - index);

fix(p2\_1);

fix(p2\_2); //将part2部分按照|前后分为两部分

if (p2\_1.find(part1) == 0 && p2\_1[part1.length()] == ' ')

{

p2\_1.assign(p2\_1, part1.length() + 1, p2\_1.length() - part1.length());

out << part1 << " -> ";

if (p2\_2 != "empty")

out << p2\_2 << " ";

else

out << p2\_2 << " | ";

out << part1 << "'" << endl;

out << part1 << "' -> " << p2\_1 << " " << part1 << "' | empty" << endl;

} //p2\_1出现part1说明出现了左递归

else

{

k1 = splitblank(p2\_1, str1);

k2 = splitblank(p2\_2, str2);

flag = 0;

while (str1[flag] == str2[flag])

flag++; //判断最左侧有几个公共因子

if (!flag) //flag表示公共左因子个数,为0时无公共左因子

out << temp << endl;

else

{

out << part1 << " -> ";

for (int i = 0;i < flag;i++)

out << str1[i] << " ";

out << part1 << "'" << endl; //输出替换掉的第一行

out << part1 << "' -> ";

for (int i = flag;i < k1;i++)

out << str1[i] << " ";

if (flag == k1) //一条产生式都为公共左因子则在消除时补充empty

out << "empty ";

out << "| ";

for (int i = flag;i < k2;i++)

out << str2[i] << " ";

if (flag == k2)

out << "empty ";

out << endl;

}

}

}

else

out << temp << endl;

}

}

//将语法根据|拆分

void splitgrammar(fstream &in, fstream &out)

{

string temp, output = "", t;

int start, index, count=1;

while (!in.eof())

{

getline(in, temp);

if (in.fail())

break;

start = temp.find("->") + 2;

output.assign(temp, 0, start);

index = temp.find('|', start);

while (index != temp.npos)

{

t.assign(temp, start, index - start);

out << count++ << " " << output << t << endl;

start = index + 1;

index = temp.find('|', start);

}

t.assign(temp, start, temp.length() - start + 1);

out << count++ << " " << output << t;

out << endl;

}

maxline = count - 1;

}

Lexical-analysis.h

#pragma once

#include "global.h"

//种别码syn 单词符号symbol 类型type |种别码syn 单词符号symbol 类型type

//0 # end |14 <= operator

//1 else keyword |15 >= operator

//2 if keyword |16 == operator

//3 int keyword |17 != operator

//4 return keyword |18 ( operator

//5 void keyword |19 ) operator

//6 while keyword |20 [ array

//7 + operator|21 ] array

//8 - operator|22 ; line\_end

//9 \* operator|23 , dot

//10 / operator|24 { struct

//11 < operator|25 } struct

//12 > operator|26 letter|letter\* id

//13 = operator|27 digit|digit\* num

//COP 比较操作符: < <= > >= == !=

//AOP 赋值操作符: =

//OOP 运算操作符: + - \* /

//EOP 句末操作符: ;

//SOP 结构分隔符: ( ) , [ ] { }

//RESERVED 保留字: int if else return void while

//保留字字符表

const string key[6] = { "else","if","int","return","void","while" };

//记录各个token所在的行与列

static int line = 1;

static int column = 1;

int isBC(char ch); //判断是否为空白字符

int Reserve(string strToken);

void Retract(fstream &source);

void lexicalanalysis(fstream &in, fstream &out); //词法分析

void fixlex(void);

Lexical-analysis.cpp

#include "lexical-analysis.h"

//判断是否为空白字符 并且修改行列值

int isBC(char ch) {

switch (ch)

{

case ' ':

column++;

break;

case '\t':

column += 4;

break;

case '\r':

case '\n':

line++;

column = 1;

break;

default:

return 0;

}

return 1;

}

//判断是否为保留字 是返回1 否则返回0

int Reserve(string strToken) {

int i;

for (i = 0; i < 6; i++) {

if (strToken.compare(key[i]) == 0) {

return 1;

}

}

return 0;

}

//所读取文件回退一个字符

void Retract(fstream &source) {

if (!source.eof()) {

source.seekg(-1, ios::cur);

}

}

//词法分析

void lexicalanalysis(fstream &in, fstream &out)

{

string strToken;

char ch;

while (!in.eof())

{

ch = in.get();

if (in.fail())

break;

if (isBC(ch)) { //遇到空白字符

strToken = "";

}

else if (isalpha(ch)) { //遇到字母

while (isalpha(ch) || isdigit(ch)) { //读取整个token 直到下一个空白字符

strToken.push\_back(ch);

column++;

ch = in.get();

}

if (Reserve(strToken)) { //判断token是否为保留字

//cout << strToken << ", RESERVED" << endl;

out << strToken << " RESERVED" << " " << line << " " << column - strToken.length() << endl;

}

else {

// cout << strToken << ", ID" << endl;

out << strToken << " ID" << " " << line << " " << column - strToken.length() << endl;

}

strToken = "";

Retract(in);

else if (isdigit(ch)) { //遇到数字

while (isdigit(ch)) { //读取整个数

strToken.push\_back(ch);

column++;

ch = in.get();

}

//数字之后出现字母 发现错误格式的ID 报错

if (isalpha(ch)) {

cout << "[Lexical ERROR] " << " [" << line << "," << column - strToken.length() << "] " << "Invalid ID: ";

while (isalpha(ch) || isdigit(ch)) {

strToken.push\_back(ch);

column++;

ch = in.get();

}

cout << "\"" << strToken << "\"" << endl;

out << "^ " << strToken << " ID " << line << " " << column - strToken.length() << endl;

out << "[Lexical ERROR] " << " [" << line << "," << column - strToken.length() << "] " << "Invalid ID: ";

out << "\"" << strToken << "\"" << endl;

}

else {

//cout << strToken << ", INT" << endl;

out << strToken << " NUM" << " " << line << " " << column - strToken.length() << endl;

}

Retract(in);

strToken = "";

}

else {

switch (ch) //其他OP类字符

{

case '=':

column++;

ch = in.get();

if (ch == '=')

{

column++;

out << "== COP" << " " << line << " " << column - 2 << endl;

}

else {

out << "= AOP" << " " << line << " " << column - 1 << endl;

Retract(in);

}

break;

case '<':

column++;

ch = in.get();

if (ch == '=') {

column++;

//cout << "<=, COP" << endl;

out << "<= COP" << " " << line << " " << column - 2 << endl;

}

else {

//cout << "<, COP" << endl;

out << "< COP" << " " << line << " " << column - 1 << endl;

Retract(in);

}

break;

case '>':

column++;

ch = in.get();

if (ch == '=') {

column++;

//cout << ">=, COP" << endl;

out << ">= COP" << " " << line << " " << column - 2 << endl;

}

else {

//cout << ">, COP" << endl;

out << "> COP" << " " << line << " " << column - 1 << endl;

Retract(in);

}

break;

case '!':

column++;

ch = in.get();

if (ch == '=') {

column++;

out << "!= COP" << " " << line << " " << column - 2 << endl;

}

else

{

//cout << "[Lexical ERROR] " << " [" << line << "," << column << "] " << "Invalid COP ";

out << "[Lexical ERROR] " << " [" << line << "," << column - 1 << "] " << "Invalid COP ";

Retract(in);

}

break;

case '+':

case '-':

case '/':

case '\*':

column++;

//cout << ch << ", OOP" << endl;

out << ch << " OOP" << " " << line << " " << column - 1 << endl;

break;

case ';':

column++;

//cout << ch << ", EOP" << endl;

out << ch << " EOP" << " " << line << " " << column - 1 << endl;

break;

case '(':

case ')':

case ',':

case '[':

case ']':

case '{':

case '}':

column++;

//cout << ch << ", SOP" << endl;

out << ch << " SOP" << " " << line << " " << column - 1 << endl;

break;

default: //未识别成功的信息 显示出错信息

column++;

//cout << ch << ", UNKNOWN" << endl;

out << "[Lexical ERROR] " << " [" << line << "," << column - 1 << "] " << "UNKNOW EXPRESSION " << ch << endl;

}

}

}

out << "# # " << line + 1 << " 1";

}

void fixlex(void)

{

fstream in, out;

string temp;

vector<struct Token>tokens;

vector<struct Token>::iterator vec;

Token token;

in.open(file\_lex, ios::in);

while (!in.eof())

{

in >> token.re >> token.type >> token.line >> token.colume;

if (in.fail())

break;

tokens.push\_back(token);

}

for (vec = tokens.begin();vec != tokens.end();vec++)

{

if (vec->type == "ID" && (vec + 1)->re == "(")

vec->type = "IDF";

if (vec->type == "ID" && (vec + 1)->re == "=")

vec->type = "ID1";

}

in.close();

out.open(file\_lex, ios::out);

for (vec = tokens.begin();vec != tokens.end();vec++)

out << vec->re << " " << vec->type << " " << vec->line << " " << vec->colume << endl;

out.close();

}

四、运行结果



2、主要功能模块的设计——优化

优化系统：

优化的任务是对代码进行等价变换，使得变换后的代码运行结果和变换代码前运行结果一致，而运行速度加快或是占用空间减少，或者两者都有。这里首先对原代码进行基本块的划分找到每个基本块的入口，之后按每块进行优化，对于常数如果后面重复出现则进行代替，若变量名后面有重复声明则替代，删除原多余四元式。

输入：四元式代码

输出：优化后的四元式代码

3、系统运行流程

具体操作：

首先要进行基本块的划分，讲四元式代码传入之后逐条分析，根据每行四元式按照第一个空格，分开序号和后面的内容。利用循环将后面的内容再利用空格进行的分割都存到由<int,vector(string)>对应的map中，其中int存的是最开始分开的序号。在存放vector时查询如果遇到“Goto”则将后面的序号返回，如果是“if”则将其下一语句的序列返回。并且将每一块的第一句序列返回。

下一步操作定义一个新的map用来存放所有值得到为常数的四元式，从头到尾挨个扫描，首先要确定那些数字和操作运算就需要将字符串转换为有意义的值和操作，如果对于后面又重新赋值的四元式则将在map中的记录删除，而每读入一条同时对map进行操作，同时写入新的map中作为优化后的四元式输出。

三、系统实现

1、系统主要函数说明

void getbetter1(int start,int end) 将序号和四元式拆分

void getbetter2(int start,int end) 进行优化

void optimize(fstream &in, fstream &out) 将结果输入到新文件中

2、系统代码

Optimize.cpp:

void getbetter1(fstream &in) {

string str, part1, part2, t;

int split = 0, index, start, num;

while (!in.eof()) {

getline(in, str);

if (in.fail())

break;

split = str.find(' ');

part1.assign(str, 0, split);

part1 = part1.substr(1, part1.length() - 1);

part2.assign(str, split + 1, str.length() - split - 1);

fix(part1);

fix(part2);

//读入每一行中间代码 将代码地址 与代码内容分开

start = 0;

stringstream ss;

ss << part1;

num = atoi(ss.str().c\_str());

gb.insert({ num,{} });

//将地址提取为数字 插入代码表中

while (1) {

index = part2.find(' ', start);

if (index == part2.npos)

break;

t.assign(part2, start, index - start);

start = index + 1;

gb[num].push\_back(t);

}

t.assign(part2, start, part2.length() - start);

gb[num].push\_back(t);

}

//得到所有基本块的入口

for (gber = gb.begin(); gber != gb.end(); gber++)

{

//1.程序的第一条语句

if (gber == gb.begin())

entry.push\_back(gber->first);

//2.强制跳转语句的目的地址

for (ier = gber->second.begin(); ier != gber->second.end(); ier++) {

if (\*ier == "Goto") {

iertemp = ier + 1;

stringstream ss;

ss << \*iertemp;

num = atoi(ss.str().c\_str());

itertemp = find(entry.begin(), entry.end(), num);

if (itertemp == entry.end())

entry.push\_back(num);

}

//3.条件语句的下一条语句

if (\*ier == "if") {

gbertemp = gber;

gbertemp++;

gbertemp++;

itertemp = find(entry.begin(), entry.end(), gbertemp->first);

if (itertemp == entry.end())

entry.push\_back(gbertemp->first);

}

//4.函数的第一条语句

if (\*ier == "Proc")

entry.push\_back(gber->first);

}

}

gber--;

//记录所有代码地址的最后一条

endline = gber->first;

}

//

int checkdigit(string str)

{

int i;

for (i = 0;i < str.length();i++)

if (!isdigit(str[i]))

return 0;

return 1;

}

//

int checkvar(string str, map<string, double> cons)

{

for (con = cons.begin();con != cons.end();con++)

if (con->first == str)

return 1;

return 0;

}

//

void getbetter2(int start,int end)

{

int i;

int flag = 0;

map<string, double> constant;

for (i=start;i<=end;i++)

{

flag = 0;

if (gb[i].size() == 3 && checkdigit(gb[i][2]))

{

stringstream ss;

ss << gb[i][2];

double ll;

ll = atol(ss.str().c\_str());

constant.insert({ gb[i][0] ,ll });

}

if (gb[i].size() == 3)

{

for (con = constant.begin(); con != constant.end(); con++)

{

if (con->first == gb[i][2])

gb[i][2] = to\_string(con->second);

}

}

if (gb[i].size() == 5)

{

if (checkdigit(gb[i][2]))

flag++;

if (checkdigit(gb[i][4]))

flag++;

for (con = constant.begin(); con != constant.end(); con++)

{

if (con->first == gb[i][2])

{

gb[i][2] = to\_string(con->second);

flag++;

}

if (con->first == gb[i][4])

{

gb[i][4] = to\_string(con->second);

flag++;

}

}

if (flag == 2)

{

double b, d;

b = atol(gb[i][2].c\_str());

d = atol(gb[i][4].c\_str());

switch (op[gb[i][3]])

{

case 1:

gb[i][2] = to\_string(b + d);

break;

case 2:

gb[i][2] = to\_string(b - d);

break;

case 3:

gb[i][2] = to\_string(b \* d);

break;

case 4:

gb[i][2] = to\_string(b / d);

break;

case 5:

gb[i][2] = to\_string(b == d);

break;

case 6:

gb[i][2] = to\_string(b != d);

break;

case 7:

gb[i][2] = to\_string(b <= d);

break;

case 8:

gb[i][2] = to\_string(b < d);

break;

case 9:

gb[i][2] = to\_string(b >= d);

break;

case 10:

gb[i][2] = to\_string(b > d);

break;

};

gb[i].pop\_back();

gb[i].pop\_back();

constant.insert({gb[i][0],atol(gb[i][2].c\_str())});

}

if (checkvar(gb[i][0], constant) && flag == 0)

{

constant.erase(gb[i][0]);

}

}

}

}

//优化中间代码

void optimize(fstream &in, fstream &out)

{

init\_op();

getbetter1(in);

//输出基本块信息

out << "基本块入口:" << endl;

for (it = entry.begin();it != entry.end();it++)

out << \*it << endl;

for (it = entry.begin();it != entry.end();it++)

{

itertemp = it;

itertemp++;

if (itertemp == entry.end())

getbetter2(\*it, endline);

else

getbetter2(\*it, \*itertemp - 1);

}

for (gber = gb.begin();gber != gb.end();gber++)

{

out << gber->first << " ";

for (ier = gber->second.begin();ier != gber->second.end();ier++)

out << \*ier << " ";

out << endl;

}

}

四、运行结果

