**基于C0文法的编译器总结报告**

**10011603 2016302731 冯宇**

目录

[一.文法定义 3](#_Toc13060986)

[二.文法说明 4](#_Toc13060987)

[2.1变量定义的同时赋予初值 4](#_Toc13060988)

[2.2 实现while循环 4](#_Toc13060989)

[2.3 在条件语句中添加逻辑操作“与”和“或” 5](#_Toc13060990)

[2.4 添加addi,subi等指令，相较于原来的加减指令减少对内存的访问 7](#_Toc13060991)

[2.5 对于程序开始的变量声明引起的$sp地址变化，进行累计后一次性变换，减少指令数 8](#_Toc13060992)

[2.6 对于乘除法中的2的幂的数的运算，可以使用移位操作，指令的效率更高 8](#_Toc13060993)

[三.系统总体设计 9](#_Toc13060994)

[3.1 开发环境和工具 9](#_Toc13060995)

[3.2 总体结构 9](#_Toc13060996)

[3.3 文件作用 9](#_Toc13060997)

[四.详细设计 10](#_Toc13060998)

[五.测试用例及测试结果 10](#_Toc13060999)

[六.实验过程中解决的问题清单和说明 22](#_Toc13061000)

[6.1 全局变量声明 22](#_Toc13061001)

[6.2 重编译错误 22](#_Toc13061002)

[6.3 文件调试 23](#_Toc13061003)

[6.4 全局变量初始化 24](#_Toc13061004)

[6.5 函数重定义 24](#_Toc13061005)

[6.6 程序出现报错的处理方式 24](#_Toc13061006)

[6.7 原本的for循环错误 25](#_Toc13061007)

[6.8 strcpy()报错 26](#_Toc13061008)

[6.9 避免plus和sub与默认的冲突 26](#_Toc13061009)

[6.10 变量类型的转化 26](#_Toc13061010)

[6.11 缺少“)”报错 27](#_Toc13061011)

[6.12 while语句设计报错 27](#_Toc13061012)

[6.13 什么是正确的MIPS代码 28](#_Toc13061013)

[6.14 C++中的模板 28](#_Toc13061014)

[6.15 linux中文件的命名不能包含&字符，否则会报错 29](#_Toc13061015)

[6.16 Mars中的乘除法 29](#_Toc13061016)

[6.17 DAG原理 29](#_Toc13061017)

[6.17 优化的时候主要考虑的是添加addi，subi等带常数的指令，来减少对于内存的访问 32](#_Toc13061018)

[6.18 乘除法优化 33](#_Toc13061019)

[6.19 2的幂级数的快速判断 33](#_Toc13061020)

[6.20 文本指针的及时关闭 34](#_Toc13061021)

[6.21 修改不完善 34](#_Toc13061022)

[七.其他附录 34](#_Toc13061023)

[八.心得体会 39](#_Toc13061024)

# 一.文法定义

＜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＞} \*/

＜全局变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;} （考虑变量定义后可能出现的有返回值函数）

＜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＞’=’<表达式>|＜标识符＞‘[’＜无符号整数＞‘]’){,＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’}

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞ （返回值为标识符名）

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

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

＜参数＞ ::= ＜参数表＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

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

＜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＞‘)’

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

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

<条件>::=<条件子项>|'('＜条件子项＞')'[<&& ||>'('＜条件子项＞')']

＜条件子项＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞

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

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

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞|<空>‘)’

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞|<空>‘)’

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

＜返回语句＞ ::= return[‘(’＜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＞‘)’

<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＞;|＜情况语句＞｜

＜返回语句＞;｜＜空＞;

＜语句列＞ ::=｛＜语句＞｝

<while> ::= while ‘(‘ <条件>’)’’{‘<语句>’}’

# 二.文法说明

与C0文法不同，本文法添加了

## 2.1变量定义的同时赋予初值

在变量定义函数void varDef()中添加赋值部分，代码如下：

string infixString = "";//存储数字赋值

if (sy == becomes) { // 读取“=”，进行变量的赋值

if (idIndex != -1) {

if ((idTable[idIndex].cls != vars && idTable[idIndex].cls != params)

|| idTable[idIndex].length != 0) {

error(IdKindNotMatch); 变量名匹配

}

}

insymbol();

expression(rightToEqual);获得“=”右边表达式的值

insertInfix("ASSIGN", " ", rightToEqual, leftToEqual);//进行变量赋值

## 2.2 实现while循环

while循环

该循环主要参照for循环书写完成，由于四元式中无新添加的语句，所以我们主要修改的是main.h、lexical.cpp以及praser.cpp中的代码

首先，在main.h中的枚举类symbol中定义while关键字，将关键字的总个数keywordCount加一

在词法分析中进行读取判断

在语法分析中

首先在<语句>中添加对于while的选择：

case whilesy:

whileState();

break;

添加语法函数：

<while> ::= while ‘(’ <条件>’)’{语句}

void whileState(){

string judgeLabel = "";//判断条件LABEL

string bodyLabel = "";//循环体LABEL

string endLabel = "";//while循环外部语句开头LABEL

首先读取while关键字

定义循环体的LABEL bodyLabel = labelStr + intToString(labelIndex);

定义判断条件的LABEL judgeLabel = labelStr + intToString(labelIndex);并插入infixTable

进行条件的判断judgement(judgeResult)

insertInfix("BNE", judgeResult, "0", bodyLabel); // 如果judgeResult != 0(为真) 转入bodyLabel

endLabel = labelStr + intToString(labelIndex);//定义endLABEL

labelIndex++;

insertInfix("BEQ", judgeResult, "0", endLabel); // 如果judgeResult = 0(为假) 跳转到 endLabel，即while循环外部

读取语句statement();

insertInfix("JMP", " ", " ", judgeLabel);//跳转到判断体的LABEL

insertInfix("LABEL", " ", " ", endLabel);//在此处插入while循环外部的LABEL

}

## 2.3 在条件语句中添加逻辑操作“与”和“或”

首先在main.h中的symbol结构体中添加andsy,orsy类型

在词法分析器中lexical.cpp添加以下代码识别于字符&&和或字符||

else if (ch == '|') {

nextch();

if (ch == '|') {

k++;

sy = orsy;

nextch();

}

}

else if (ch == '&') {

nextch();

if (ch == '&') {

k++;

sy = andsy;

nextch();

}

}

在语法分析phraser.cpp中，将原本的<条件>函数修改为<条件子项>，添加新的条件函数

//<条件>::=<条件子项>|'('＜条件子项＞')'[<&& ||>'('＜条件子项＞')']

void judgement(string &infixString) {

string logicalLeft = "";

string logicalRight = "";

symbol logicalSy = andsy;

if (sy == lparent) {

insymbol();

judgement\_child(logicalLeft);

if (sy == rparent) {

if (skipFlag) {

skipFlag = false;

}

insymbol();

}

else if (!skipFlag) {

error(RightParentLost); // Right parenthesis lost

symbol nexts[] = { semicolon, rbrace };

int length = sizeof(nexts) / sizeof(symbol);

skipUntil(nexts, length);

}

if (sy == andsy || sy == orsy) {

logicalSy = sy;

insymbol();

if (sy == lparent) {

insymbol();

judgement\_child(logicalRight);

if (sy == rparent) {

if (skipFlag) {

skipFlag = false;

}

insymbol();

}

else if (!skipFlag) {

error(RightParentLost); // Right parenthesis lost

symbol nexts[] = { semicolon, rbrace };

int length = sizeof(nexts) / sizeof(symbol);

skipUntil(nexts, length);

}

}

infixString = createTempVar();

insertTable(vars, ints, infixString.c\_str(), 0, level, 0); // 类型为数字，1表示判断成立，0表示判断不成立

switch (logicalSy) {

case andsy:

insertInfix("AND", logicalLeft, logicalRight, infixString);

break;

case orsy:

insertInfix("OR", logicalLeft, logicalRight, infixString);

break;

}

}

}

else {

judgement\_child(infixString);

}

}

四元式部分infixNotation.cpp文件中

void insertInfix(string ioperator, string operand1, string operand2, string operand3)

添加处理语句

else if (ioperator == "AND") {

outputBuff = outputBuff + operand3 + " = " + operand1 + " && " + operand2 + "\n";

}

else if (ioperator == "OR") {

outputBuff = outputBuff + operand3 + " = " + operand1 + " || " + operand2 + "\n";

}

转化为MIPS代码时，添加新的MIPS语句

else if (notation.ioperator == "AND") {

operandsToRegister(notation);

insertCode("and $t0, $t1, $t2");

storeThirdOperand(notation.operand3, "$t0");

}

else if (notation.ioperator == "OR") {

operandsToRegister(notation);

insertCode("or $t0, $t1, $t2");

storeThirdOperand(notation.operand3, "$t0");

}

## 2.4 添加addi,subi等指令，相较于原来的加减指令减少对内存的访问

首先在void funcContent()中判断加减号两边是数字还是变量，然后进行不同处理，如果是数字，那么经过以下处理

insertCode("addi $t0,$zero," + toString(notation.operand1));

insertCode("add $t0,$t0,$t2");

storeThirdOperand(notation.operand3, "$t0");

通过addi指令减少对于内存的访问，subi同理

## 2.5 对于程序开始的变量声明引起的$sp地址变化，进行累计后一次性变换，减少指令数

在变量定义的MIPS转化函数void mipsVarDef(bool isGlobal)中设置一个变量cout ，记录地址变化的量，退出函数的时候统一进行地址的变换

## 2.6 对于乘除法中的2的幂的数的运算，可以使用移位操作，指令的效率更高

添加函数

int log\_2(int value) //非递归判断一个数是2的多少次方

{

int x = 0;

while (value > 1)

{

value >>= 1;

x++;

}

return x;

}

在void muli\_insertcode(infixNotation notation)中对于乘数为2的幂的数使用sll指令，而不是mul指令

if (num & (num - 1)) { //operand1不是2的幂次方

insertCode("mul $t0,$t2," + toString(notation.operand1));

}

else{

i = log\_2(num);

insertCode("sll $t0,$t2," + intToString(i));

}

storeThirdOperand(notation.operand3, "$t0");

除法同理，如果是除数是2的幂次方，使用sra指令，而不是div指令

其他文法 具体说明参考基于C0文法的编译器V1.0——V7.0

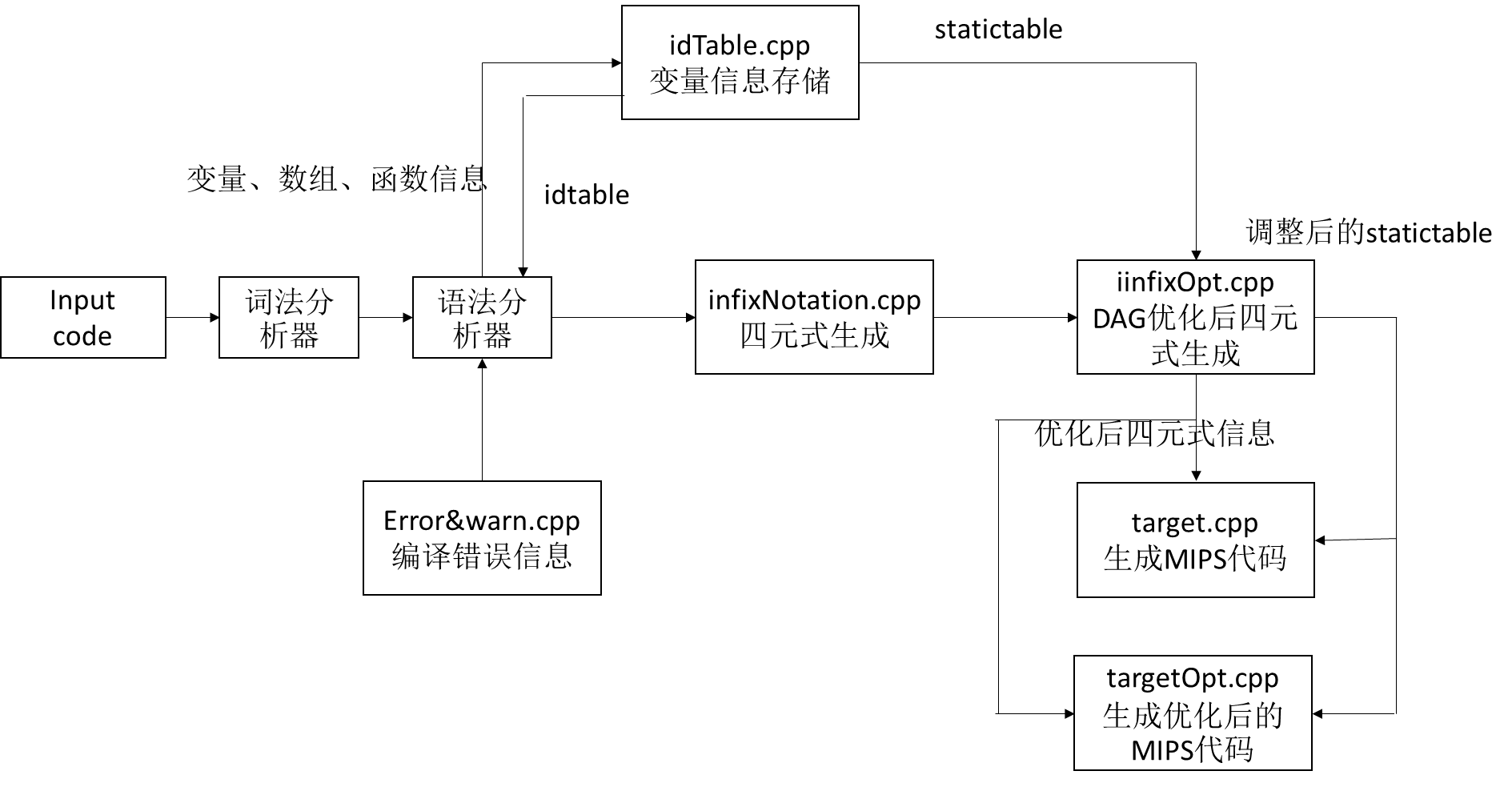
# 三.系统总体设计

## 3.1 开发环境和工具

开发环境为Windows 10

开发工具：VS2017 VScode64位 V1.33.1 Ubuntu 14.04 Mars4.5

## 3.2 总体结构



## 3.3 文件作用

|  |  |
| --- | --- |
| 文件名 | 作用 |
| error&warn.cpp | 对input代码的处理中遇见的错误进行处理 |
| idTable.cpp | 定义idTable和staticTable的vector容器和相关处理函数，idTable中存储变量、数组、函数的名字、大小、地址、level等相关信息，随着语法分析中读取函数对变量进栈出栈，实现编译错误识别。staticTable存储目前读取的所有函数的变量、数组的名字、大小、地址、level等相关信息，用于生成MIPS代码 |
| lexical.cpp | 词法分析器 |
| parser.cpp | 语法分析器 |
| infixNotation.cpp | 生成四元式 |
| infixOpt.cpp | 利用DAG优化四元式 |
| target.cpp | 生成MIPS代码 |
| targetOpt.cpp | 优化生成MIPS |
| tool.cpp | 提供数据类型转化的一系列函数 |
| main.cpp | 提供程序入口 |

# 四.详细设计

详见代码及基于C0文法的编译器V1.0-V7.0的代码实现部分

# 五.测试用例及测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| Input code（测试if语句） | | 四元式 | |
| int main(){  int x,y,z;  x=1+6;  y=2;  z=3;  if(x>0)x=y+1;  else x=x\*2;  return;  } | | int main()  int x  int y  int z  z = 3  y = 2  x = 1 + 6  #t1 = x > 0  goto LABEL0, if #t1 == 0  x = y + 1  jmp LABEL1  LABEL0  x0 = x  x = x0 \* 2  LABEL1  return | |
| Input code（测试for语句） | | 四元式 | |
| int main(){  int x,y,z;  x=1+6;  y=2;  z=3;  for(x=0;y<4;y=y+1)  z=z\*2;  return;  } | | int main()  int x  int y  int z  x = 0  z = 3  y = 2  #t0 = 1 + 6  LABEL1  #t1 = y < 4  goto LABEL0, if #t1 != 0  goto LABEL2, if #t1 == 0  LABEL3  y0 = y  y = y0 + 1  jmp LABEL1  LABEL0  z0 = z  z = z0 \* 2  jmp LABEL3  LABEL2  return | |
| Input code（scanf和printf） | | 四元式 | |
| int main(){  int ax,bx;  int i;  bx=0;  i=0;  scanf(ax);  printf(bx);  return;  } | | int main()  int ax  int bx  int i  i = 0  bx = 0  scan ax  print bx  return | |
| Input code(switch语句) | | 四元式 | |
| const int a=2;  const char c='Q';  int main(){  int x,y,i;  i=3;  switch(i){  case 1: i=i+1;  case 2: i=i\*2;  case 3: i=9;  default:i=1;  }  return ;  } | | const int a = 2  const char c = Q  int main()  int x  int y  int i  i = 3  goto LABEL1, if i != 1  #t0 = i + 1  i = #t0  jmp LABEL0  LABEL1  goto LABEL2, if i != 2  #t1 = i \* 2  i = #t1  jmp LABEL0  LABEL2  goto LABEL3, if i != 3  i = 9  jmp LABEL0  LABEL3  i = 1  LABEL0  return | |
| Input code（函数调用） | | 四元式 | |
| int compute(int x1,int x2)  {  int y1;  y1=x1+x2;  return (y1);  }  int main(){  int x1,x2;  x1=1;  x2=5;  compute(x1,x2);  return;  } | | int compute()  param int x1  param int x2  int y1  #t0 = x1 + x2  y1 = #t0  return y1  int main()  int x1  int x2  x1 = 1  x2 = 5  push x1  push x2  call compute  #t1 = #RET  return | |
| Input code（变量定义时初始化并实现while） | | 四元式 | |
| int main(){  int ax =2;  int i=0;  while(i <=3){  i=i+1;  }  ax=ax+5;  return;  } | | int main()  int ax  ax = 2  int i  i = 0  LABEL1  #t0 = i <= 3  goto LABEL0, if #t0 != 0  goto LABEL2, if #t0 == 0  LABEL0  #t1 = i + 1  i = #t1  jmp LABEL1  LABEL2  #t2 = ax + 5  ax = #t2  return | |
| Input code(MIPS代码生成) | 四元式 | | MIPS |
| int mm(int x)  {  return (x+1);  }  int main()  {  int a=1,b=2;  mm(a);  switch(a){  case 1:b=b+1;  case 2:b=b-1;  default:b=0;  }  a=a\*4;  return;  } | int mm()  param int x  #t0 = x + 1  return #t0  int main()  int a  a = 1  int b  b = 2  push a  call mm  #t1 = #RET  goto LABEL1, if a != 1  #t2 = b + 1  b = #t2  jmp LABEL0  LABEL1  goto LABEL2, if a != 2  #t3 = b - 1  b = #t3  jmp LABEL0  LABEL2  b = 0  LABEL0  #t4 = a \* 4  a = #t4  return | | .data  .text  j main  mmTail:  lw $ra, -4($fp)  move $sp, $fp  lw $fp, ($sp)  jr $ra  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 1  sw $t0, -8($fp)  li $t0, 2  sw $t0, -12($fp)  sub $sp, $sp, 8  lw $t0, -8($fp)  sw $t0, ($sp)  sub $sp, $sp, 4  jal mm  move $t0, $v1  sw $t0, -16($fp)  lw $t1, -8($fp)  li $t2, 1  bne $t1, $t2, LABEL1  lw $t0, -12($fp)  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 1  add $t0, $t1, $t2  sw $t0, -12($fp)  j LABEL0  LABEL1:  lw $t1, -8($fp)  li $t2, 2  bne $t1, $t2, LABEL2  lw $t0, -12($fp)  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 1  sub $t0, $t1, $t2  sw $t0, -12($fp)  j LABEL0  LABEL2:  li $t0, 0  sw $t0, -12($fp)  LABEL0:  lw $t0, -8($fp)  sw $t0, -28($fp)  lw $t1, -28($fp)  li $t2, 4  mul $t0, $t1, $t2  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code(MIPS代码生成) | 四元式 | | MIPS |
| int main(){  int ax,bx;  int i;  bx=0;  i=0;  for(ax=0;i<=2;i=i+1){  ax=bx+i;  }  ax=ax\*2;  return;  } | int main()  int ax  int bx  int i  bx = 0  i = 0  ax = 0  LABEL1  #t0 = i <= 2  goto LABEL0, if #t0 != 0  goto LABEL2, if #t0 == 0  LABEL3  i = i + 1  jmp LABEL1  LABEL0  #t1 = bx + i  ax = #t1  jmp LABEL3  LABEL2  #t2 = ax \* 2  ax = #t2  return | | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 0  sw $t0, -8($fp)  li $t0, 0  sw $t0, -16($fp)  li $t0, 0  sw $t0, -12($fp)  LABEL1:  lw $t1, -16($fp)  li $t2, 2  sle $t0, $t1, $t2  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 0  bne $t1, $t2, LABEL0  lw $t1, -20($fp)  li $t2, 0  beq $t1, $t2, LABEL2  LABEL3:  lw $t0, -16($fp)  sw $t0, -24($fp)  lw $t1, -24($fp)  li $t2, 1  add $t0, $t1, $t2  sw $t0, -16($fp)  j LABEL1  LABEL0:  lw $t1, -12($fp)  lw $t2, -16($fp)  add $t0, $t1, $t2  sw $t0, -8($fp)  j LABEL3  LABEL2:  lw $t0, -8($fp)  sw $t0, -28($fp)  lw $t1, -28($fp)  li $t2, 2  mul $t0, $t1, $t2  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code(MIPS代码生成) | 四元式 | | MIPS |
| int main(){  int ax =2;  int i=0;  while(i <=3){  i=i+1;  }  ax=ax+5;  return;  } | int main()  int ax  ax = 2  int i  i = 0  LABEL1  #t0 = i <= 3  goto LABEL0, if #t0 != 0  goto LABEL2, if #t0 == 0  LABEL0  #t1 = i + 1  i = #t1  jmp LABEL1  LABEL2  #t2 = ax + 5  ax = #t2  return | | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 2  sw $t0, -8($fp)  li $t0, 0  sw $t0, -12($fp)  LABEL1:  lw $t1, -12($fp)  li $t2, 3  sle $t0, $t1, $t2  sw $t0, -16($fp)  lw $t1, -16($fp)  li $t2, 0  bne $t1, $t2, LABEL0  lw $t1, -16($fp)  li $t2, 0  beq $t1, $t2, LABEL2  LABEL0:  lw $t0, -12($fp)  sw $t0, -20($fp)  lw $t1, -20($fp)  li $t2, 1  add $t0, $t1, $t2  sw $t0, -12($fp)  j LABEL1  LABEL2:  lw $t0, -8($fp)  sw $t0, -24($fp)  lw $t1, -24($fp)  li $t2, 5  add $t0, $t1, $t2  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code(MIPS代码生成) | 四元式 | | MIPS代码 |
| int main(){  int T0,T1,T2,T3,T4,T5,T6;  int A,B;  int R,r;  R=2;  r=3;  T0=5;  T1=2\*T0;  T2=R+r;  A=T1\*T2;  B=A;  T3=2\*T0;  T4=R+r;  T5=T3\*T4;  T6=R-r;  B=T5\*T6;  printf(B);  return;  } | int main()  int T0  int T1  int T2  int T3  int T4  int T5  int T6  int A  int B  int R  int r  R = 2  r = 3  T0 = 5  #t0 = 2 \* T0  T1 = #t0  #t1 = R + r  T2 = #t1  #t2 = T1 \* T2  A = #t2  B = A  #t3 = 2 \* T0  T3 = #t3  #t4 = R + r  T4 = #t4  #t5 = T3 \* T4  T5 = #t5  #t6 = R - r  T6 = #t6  #t7 = T5 \* T6  B = #t7  print B  return | | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 3  sw $t0, -20($fp)  li $t0, 2  sw $t0, -16($fp)  lw $t1, -16($fp)  lw $t2, -20($fp)  sub $t0, $t1, $t2  sw $t0, -36($fp)  lw $t1, -16($fp)  lw $t2, -20($fp)  add $t0, $t1, $t2  sw $t0, -28($fp)  li $t0, 5  sw $t0, -8($fp)  li $t1, 2  lw $t2, -8($fp)  mul $t0, $t1, $t2  sw $t0, -24($fp)  lw $t1, -24($fp)  lw $t2, -28($fp)  mul $t0, $t1, $t2  sw $t0, -32($fp)  lw $t1, -32($fp)  lw $t2, -36($fp)  mul $t0, $t1, $t2  sw $t0, -12($fp)  lw $a0, -12($fp)  li $v0, 1  syscall  move $v1, $t0  j mainTail  mainTail: |
| Input code（DAG优化中间代码） | 中间代码（四元式） | | 优化后的中间代码 |
| int main(){  int T0,T1,T2,T3,T4,T5,T6;  int A,B;  int R,r;  R=2;  r=3;  T0=5;  T1=2\*T0;  T2=R+r;  A=T1\*T2;  B=A;  T3=2\*T0;  T4=R+r;  T5=T3\*T4;  T6=R-r;  B=T5\*T6;  printf(B);  return;  } | int main()  int T0  int T1  int T2  int T3  int T4  int T5  int T6  int A  int B  int R  int r  R = 2  r = 3  T0 = 5  #t0 = 2 \* T0  T1 = #t0  #t1 = R + r  T2 = #t1  #t2 = T1 \* T2  A = #t2  B = A  #t3 = 2 \* T0  T3 = #t3  #t4 = R + r  T4 = #t4  #t5 = T3 \* T4  T5 = #t5  #t6 = R - r  T6 = #t6  #t7 = T5 \* T6  B = #t7  print B  return | | int main()  int T0  int T1  int T2  int T3  int T4  int T5  int T6  int A  int B  int R  int r  r = 3  R = 2  #t6 = R - r  #t1 = R + r  T0 = 5  #t0 = 2 \* T0  #t2 = #t0 \* #t1  B = #t2 \* #t6  print B  return |
| Input code（MIPS代码优化） | 原MIPS代码 | | 优化后的MIPS代码 |
| int main(){  int x,y,z;  x=1+6;  y=2;  z=3;  x=y+2;  x=x-20;  z=y\*4;  z=z/2;  return;  } | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 2  sw $t0, -8($fp)  lw $t1, -8($fp)  li $t2, 4  mul $t0, $t1, $t2  sw $t0, -24($fp)  lw $t1, -24($fp)  li $t2, 2  div $t0, $t1, $t2  sw $t0, -32($fp)  lw $t1, -8($fp)  li $t2, 2  add $t0, $t1, $t2  sw $t0, -16($fp)  lw $t1, -16($fp)  li $t2, 20  sub $t0, $t1, $t2  sw $t0, -20($fp)  li $t0, 3  sw $t0, -28($fp)  li $t1, 1  li $t2, 6  add $t0, $t1, $t2  sw $t0, -12($fp)  move $v1, $t0  j mainTail  mainTail: | | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 28  li $t0, 2  sw $t0, -8($fp)  lw $t1, -8($fp)  sll $t0,$t1,2  sw $t0, -24($fp)  lw $t1, -24($fp)  sra $t0,$t1,1  sw $t0, -32($fp)  lw $t1, -8($fp)  addi $t0,$zero,2  add $t0,$t0,$t1  sw $t0, -16($fp)  lw $t1, -16($fp)  subi $t0,$t1,20  sw $t0, -20($fp)  li $t0, 3  sw $t0, -28($fp)  addi $t0,$zero,1  addi $t0,$zero,6  sw $t0, -12($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code（与逻辑测试） | DAG优化四元式 | | 优化MIPS代码 |
| int main(){  int ax,bx;  int i;  bx=3;  i=0;  while((i==0)&&(bx>2)){  ax=bx+i;  }  ax=ax\*2;  return;  } | int main()  int ax  int bx  int i  bx = 3  i = 0  LABEL1  #t0 = i == 0  #t1 = bx > 2  #t2 = #t0 && #t1  goto LABEL0, if #t2 != 0  goto LABEL2, if #t2 == 0  LABEL0  #t3 = bx + i  ax = #t3  jmp LABEL1  LABEL2  #t4 = ax \* 2  ax = #t4  return | | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 28  li $t0, 0  sw $t0, -16($fp)  li $t0, 3  sw $t0, -12($fp)  LABEL1:  lw $t1, -16($fp)  li $t2, 0  seq $t0, $t1, $t2  sw $t0, -20($fp)  lw $t1, -12($fp)  li $t2, 2  sgt $t0, $t1, $t2  sw $t0, -24($fp)  lw $t1, -20($fp)  lw $t2, -24($fp)  and $t0, $t1, $t2  sw $t0, -28($fp)  lw $t1, -28($fp)  li $t2, 0  bne $t1, $t2, LABEL0  lw $t1, -28($fp)  li $t2, 0  beq $t1, $t2, LABEL2  LABEL0:  lw $t1, -12($fp)  lw $t2, -16($fp)  add $t0, $t1, $t2  sw $t0, -8($fp)  j LABEL1  LABEL2:  lw $t0, -8($fp)  sw $t0, -32($fp)  lw $t1, -32($fp)  sll $t0,$t1,1  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |
| Input code （或逻辑操作） | DAG优化四元式 | | 优化后MIPS代码 |
| int main(){  int ax,bx;  int i;  bx=3;  ax=1;  i=0;  while((i==0)&&(bx>2)){  if((bx==3)||(ax>0))ax=bx+i;  else i=i+5;  }  ax=ax\*2;  return;  } | int main()  int ax  int bx  int i  i = 0  ax = 1  bx = 3  LABEL1  #t0 = i == 0  #t1 = bx > 2  #t2 = #t0 && #t1  goto LABEL0, if #t2 != 0  goto LABEL2, if #t2 == 0  LABEL0  #t3 = bx == 3  #t4 = ax > 0  #t5 = #t3 || #t4  goto LABEL3, if #t5 == 0  ax = bx + i  jmp LABEL4  LABEL3  i0 = i  i = i0 + 5  LABEL4  jmp LABEL1  LABEL2  ax0 = ax  ax = ax0 \* 2  return | | .data  .text  j main  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 44  li $t0, 0  sw $t0, -16($fp)  li $t0, 1  sw $t0, -8($fp)  li $t0, 3  sw $t0, -12($fp)  LABEL1:  lw $t1, -16($fp)  li $t2, 0  seq $t0, $t1, $t2  sw $t0, -20($fp)  lw $t1, -12($fp)  li $t2, 2  sgt $t0, $t1, $t2  sw $t0, -24($fp)  lw $t1, -20($fp)  lw $t2, -24($fp)  and $t0, $t1, $t2  sw $t0, -28($fp)  lw $t1, -28($fp)  li $t2, 0  bne $t1, $t2, LABEL0  lw $t1, -28($fp)  li $t2, 0  beq $t1, $t2, LABEL2  LABEL0:  lw $t1, -12($fp)  li $t2, 3  seq $t0, $t1, $t2  sw $t0, -32($fp)  lw $t1, -8($fp)  li $t2, 0  sgt $t0, $t1, $t2  sw $t0, -36($fp)  lw $t1, -32($fp)  lw $t2, -36($fp)  or $t0, $t1, $t2  sw $t0, -40($fp)  lw $t1, -40($fp)  li $t2, 0  beq $t1, $t2, LABEL3  lw $t1, -12($fp)  lw $t2, -16($fp)  add $t0, $t1, $t2  sw $t0, -8($fp)  j LABEL4  LABEL3:  lw $t0, -16($fp)  sw $t0, -44($fp)  lw $t1, -44($fp)  addi $t0,$zero,5  add $t0,$t0,$t1  sw $t0, -16($fp)  LABEL4:  j LABEL1  LABEL2:  lw $t0, -8($fp)  sw $t0, -48($fp)  lw $t1, -48($fp)  sll $t0,$t1,1  sw $t0, -8($fp)  move $v1, $t0  j mainTail  mainTail: |

# 六.实验过程中解决的问题清单和说明

## 6.1 全局变量声明

<https://blog.csdn.net/ouyangxin95/article/details/51273460>

一般的全局变量两种定义方式，一种是在头文件\*.h文件中直接定义，在包含该头文件的.cpp文件中可直接使用，第二种是使用extern关键字，extern用在变量声明中常常有这样一个作用，你在\*.c文件中声明了一个全局的变量，这个全局的变量如果要被引用，就放在\*.h中并用extern来声明。需要注意的是这种方法定义的量在头文件中只是声明，在.cpp文件中才是定义。

在实验中我便犯了这样的额错误，在\*.cpp文件中没有定义变量，只是在头文件中声明，导致程序编译一直没有通过

## 6.2 重编译错误

在包含多个头文件的程序中最重要的便是避免重编译错误

假设你的工程里面有4个文件，分别是a.cpp,b.h,c.h,d.h。

a.cpp的头部是：

#include "b.h "

#include "c.h "

b.h和c.h的头部都是:

#include "d.h "

而d.h里面有class D的定义。

这样一来，编译器编译a.cpp的时候，先根据#include "b.h "去编译b.h这个问题，再根据b.h里面的#include "d.h "，去编译d.h的这个文件，这样就把d.h里面的class D编译了；

然后再根据a.cpp的第二句#include "c.h "，去编译c.h，最终还是会找到的d.h里面的class D，但是class D之前已经编译过了，所以就会报重定义错误。

加上ifndef/define/endif，就可以防止这种重定义错误。

例如：

A.h 里

#ifndef A\_H

#define A\_H

...

#endif

B.h 和 C.h都include "A.h "

D.h里

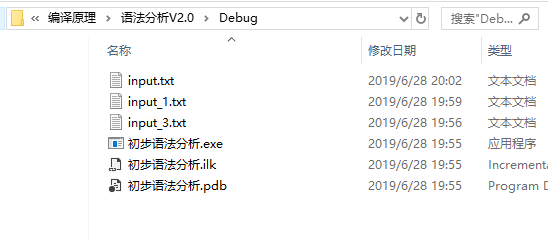
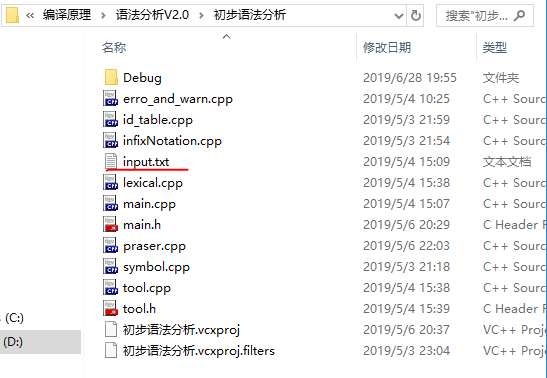
include "B.h "

include "C.h "

在预编译的过程中，执行到include "C.h "时会因为在上一句的时候已经定义了A\_H这个宏，所以此时的ifndef条件不满足，也就不会再一次包含A.h，起到了防止重复引用头文件的效果。

## 6.3 文件调试

由于我使用文本指针打开文本文件的时候，并没有添加路径，所以程序默认打开的是当前文件夹下的文件。而在VS2017中，生成的程序和调试部分的程序并不是在一个位置，所以在调试代码的时候也需要添加一个相同的文本文件

否则，在DEBUG的时候程序会一直提示未找到文本文件，文本指针打开失败。

## 6.4 全局变量初始化

全局初始化变量不能放在局部函数里，多处定义同一全局变量可能会导致引用不同导致出错。

## 6.5 函数重定义

不要在头文件里面定义函数，会出现重定义的情况

## 6.6 程序出现报错的处理方式

在程序对输入代码读取时，如果输入的代码中出现错误，那么这个时候会有以下函数用于报错并跳到“；”“）”“}”关键字以继续编译输入代码。

error(IdLost); // Identifier lost

symbol nexts[] = {semicolon, rbrace};

skipUntil(nexts);

*void* skipUntil(symbol *nexts*[]) {

while (!inNexts(nexts, sy)) {

insymbol();

}

skipFlag = true;

}

*bool* inNexts(symbol *nexts*[], symbol *tempSy*) {

for (*int* i = 0; i < sizeof(nexts) / sizeof(symbol); ++i) {

if (nexts[i] == tempSy) {

return true;

}

}

return false;

}

但原本程序是想要传递一个数组，然后在函数里求出数组长度，并进行处理，这样是会出错的，根本原因在于C/C++ 传递数组，虽然传递的是首地址地址，但是参数到了函数内就成了普通指针，不再是数组首地址了，所以试图在别的函数中得到传递数组的长度是行不通的。只能先计算好长度后再传过去，进行其他的运算。

修改后的代码如下：

error(IfSyLost); // If symbol lost

symbol nexts[] = { lparent, rparent, semicolon, rbrace };

int length = sizeof(nexts) / sizeof(symbol);

skipUntil(nexts, length);

void skipUntil(symbol nexts[],int length) {

while (!inNexts(nexts, sy, length)) {

insymbol();

}

skipFlag = true;

}

//next[]集中数据和传入的tempSy比较，有相同的返回TRUE

bool inNexts(symbol nexts[], symbol tempSy,int length) {

for (int i = 0; i < length; ++i) {

if (nexts[i] == tempSy) {

return true;

}

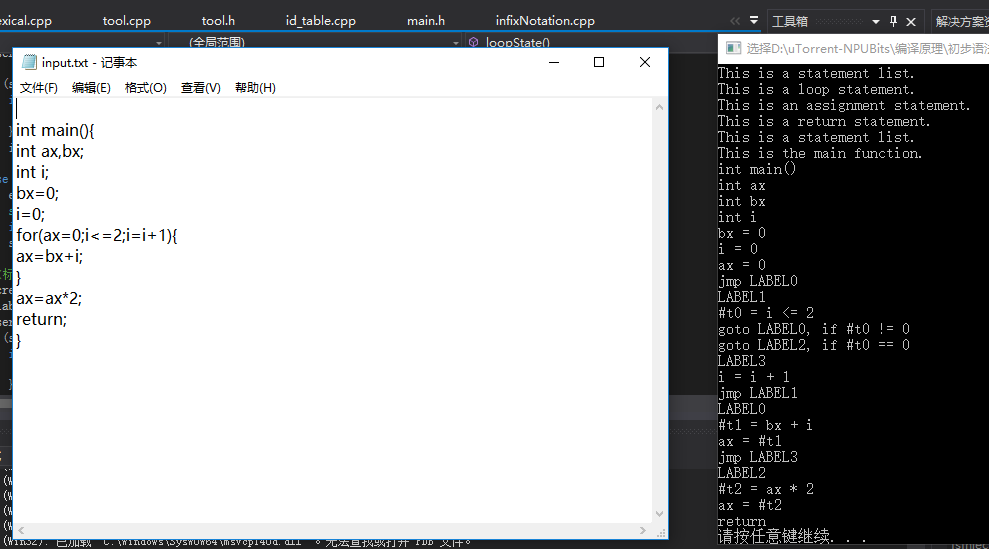
}

return false;

}

## 6.7 原本的for循环错误

在原本的for语句中，有一个地方的LABEL设置错误，按照原本的for循环执行，四元式如下：



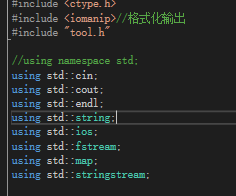
JUMP LABEL0需要修改，出现明显错误。需要删去该四元式

## 6.8 strcpy()报错

在VS2017中strcpy()函数是会报错的，在VS2017中使用了安全的函数strcpy\_s（），必须修改否则会一直报错，但是思路上是没错的

## 6.9 避免plus和sub与默认的冲突

plus和sub在使用std的namesapce情况下无法定义，会与默认的冲突。所以在头文件定义命名空间的时候采用的是：



而不是直接使用using namespace std;

避免和默认定义的冲突，能够将plus和sub定义为枚举量

## 6.10 变量类型的转化

在该程序中整数和String，以及char类型的转化不可避免，我们通过以下函数进行转化：

int strToInt(string str) {

stringstream ss(str);

int integer;

ss >> integer;

return integer;

}

char strToChar(string str) {

stringstream ss(str);

char character;

ss >> character;

return character;

}

string intToString(int num)

{

stringstream ss;

ss << num;

return ss.str();

}

string charToString(char c)

{

string str;

stringstream stream;

stream << c;

str = stream.str();

return str;

}

## 6.11 缺少“)”报错

在修改程序时误删一个“）”导致程序在编译的时候出现大量缺少“;”的报错，由于数量太多导致一开始对于错误原因毫无头绪。后续通过一步一步汇编最终发现是在修改时误删一个“）”导致的

## 6.12 while语句设计报错

在设计while语法的时候，由于思路不清晰，导致词法分析器读取没能与预先设计的内容相匹配，后面通过一步一步调试，不断修改代码，最终得到对应的while语法

## 6.13 什么是正确的MIPS代码

通常而言，linux汇编完成的代码都是X86，如果需要转化为MIPS代码，需要安装额外的工具<https://blog.csdn.net/qq_41595874/article/details/88754760#mips_70>

MIPS的寄存器以及常用指令<https://blog.csdn.net/king523103/article/details/45967991>

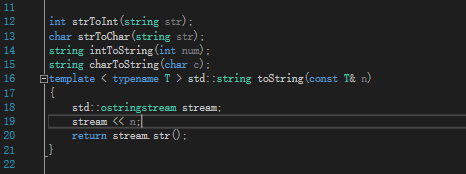
根据以上信息，对四元式进行翻译，生成可执行的MIPS代码。需要注意的是，本程序需要通过的是Mars软件的测试，所以$fp,$sp的操作可以与linux中的不一致，只需要自洽即可。保证程序运行结果的正确性。

## 6.14 C++中的模板

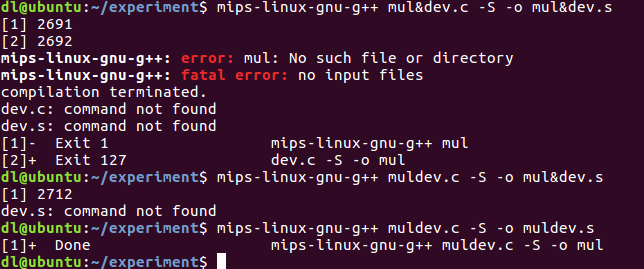
模板是C++支持**参数化**多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型**使用模板的目的就是能够让程序员编写与类型无关的代码。**比如编写了一个交换两个整型int 类型的swap函数，这个函数就只能实现**int** 型，对**double**，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个**swap**函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个**swap**模板函数，即可以实现**int** 型，又可以实现double型的交换。

需要注意的是**注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。**

这种结构能大大减少重复功能函数的使用，使程序更加简洁，本程序主要是在tool.h文件中添加一个模板实现类型的转换：

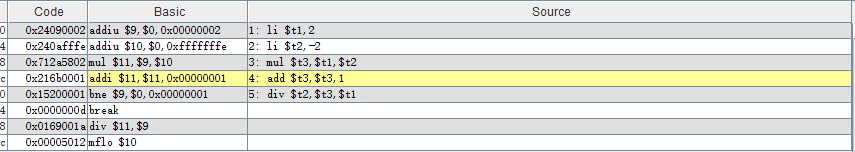


## 6.15 linux中文件的命名不能包含&字符，否则会报错



## 6.16 Mars中的乘除法

在Mars中乘除法指令形式为：



其中乘法将$t1和$t2中数字相乘的结果存储到$hi和$lo寄存器中，高位在$hi中，低位在$lo中，$t0默认从$lo中取数。而除法中Mars自动会对被除数是否为0进行判断，并进行相应处理，所以在翻译为汇编代码的时候忽略这一部分。

## 6.17 DAG原理

<https://max.book118.com/html/2016/0712/48008065.shtm>

1) 基本块的划分

　　入口语句的定义如下：

　　① 程序的第一个语句；或者，

　　② 条件转移语句或无条件转移语句的转移目标语句；

　　③ 紧跟在条件转移语句后面的语句。

有了入口语句的概念之后，就可以给出划分中间代码（四元式程序）为基本块的算法，

　　其步骤如下：

　　① 求出四元式程序中各个基本块的入口语句。

　　② 对每一入口语句，构造其所属的基本块。它是由该入口语句到下一入口语句（不包括下一入口语句），或到一转移语句（包括该转移语句），或到一停语句（包括该停语句）之间的语句序列组成的。

　 ③ 凡未被纳入某一基本块的语句、都是程序中控制流程无法到达的语句，因而也是不会被执行到的语句，可以把它们删除。

2) 基本块的优化手段

由于基本块内的逻辑清晰，故而要做的优化手段都是较为直接浅层次的。目前基本块内的常见的块内优化手段有：

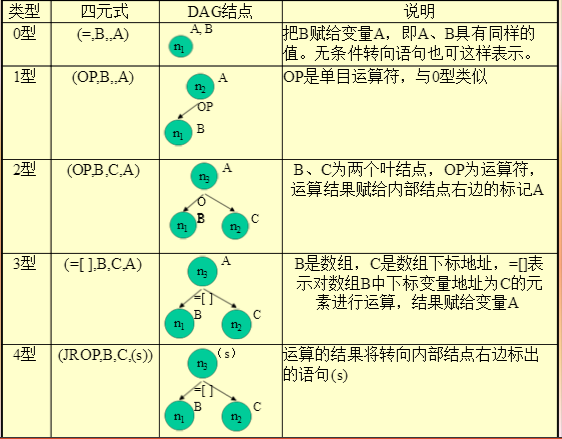
1. 删除公共子表达式

2. 删除无用代码

3. 重新命名临时变量 （一般是用来应对创建过多临时变量的，如t2 := t1 + 3如果后续并没有对t1的引用，则可以t1 := t1 + 3来节省一个临时变量的创建）

4. 交换语句顺序

5. 在结果不变的前提下，更换代数操作（如x∶=y\*\*2是需要根据\*\*运算符重载指数函数的，这是挺耗时的操作，故而可以用强度更低的x∶=y\*y来代替）



对基本块的每一四元式，依次执行：

　　1． 如果NODE（B）无定义，则构造一标记为B的叶结点并定义NODE（B）为这个结点；

　　如果当前四元式是0型，则记NODE（B）的值为n，转4。

　　如果当前四元式是1型，则转2.（1）。

　　如果当前四元式是2型，则：（Ⅰ）如果NODE（C）无定义，则构造一标记为C的叶结点并定义NODE（C）为这个结点，（Ⅱ）转2.（2）。

　　2．

　　（1） 如果NODE（B）是标记为常数的叶结点，则转2.（3），否则转3.（1）。

　　（2） 如果NODE（B）和NODE（C）都是标记为常数的叶结点，则转2.（4），否则转3.（2）。

　　（3） 执行op　B（即合并已知量），令得到的新常数为P。如果NODE（B）是处理当前四元式时 新构造出来的结点，则删除它。如果NODE（P）无定义，则构造一用P做标记的叶结点n。置NODE（P）＝n，转4.。

　　（4） 执行B　op　C(即合并已知量)，令得到的新常数为P。如果NODE（B）或NODE（C）是处理当前四元式时新构造出来的结点，则删除它。如果NODE（P）无定义，则构造一用P做标记的叶结点n。置NODE（P）＝n，转4.。

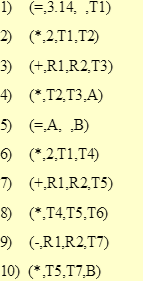
　　3)

　　（1） 检查DAG中是否已有一结点，其唯一后继为NODE（B），且标记为op（即找公共子表达式）。如果没有，则构造该结点n，否则就把已有的结点作为它的结点并设该结点为n，转4.。

　　（2） 检查DAG中是否已有一结点，其左后继为NODE（B），右后继为NODE（C），且标记为op(即找公共子表达式)。如果没有，则构造该结点n，否则就把已有的结点作为它的结点并设该结点为n。转4.。

　　4)

　　如果NODE（A）无定义，则把A附加在结点n上并令NODE（A）＝n；否则先把A从NODE（A）结点上的附加标识符集中删除（注意，如果NODE（A）是叶结点，则其标记A不删除），把A附加到新结点n上并令NODE（A）＝n。转处理下一四元式。

(1) T0∶=3.14

(2) T1∶=2 \* T0

(3) T2∶=R + r

(4) A∶=T1 \* T2

(5) B∶=A

(6) T3∶=2 \* T0

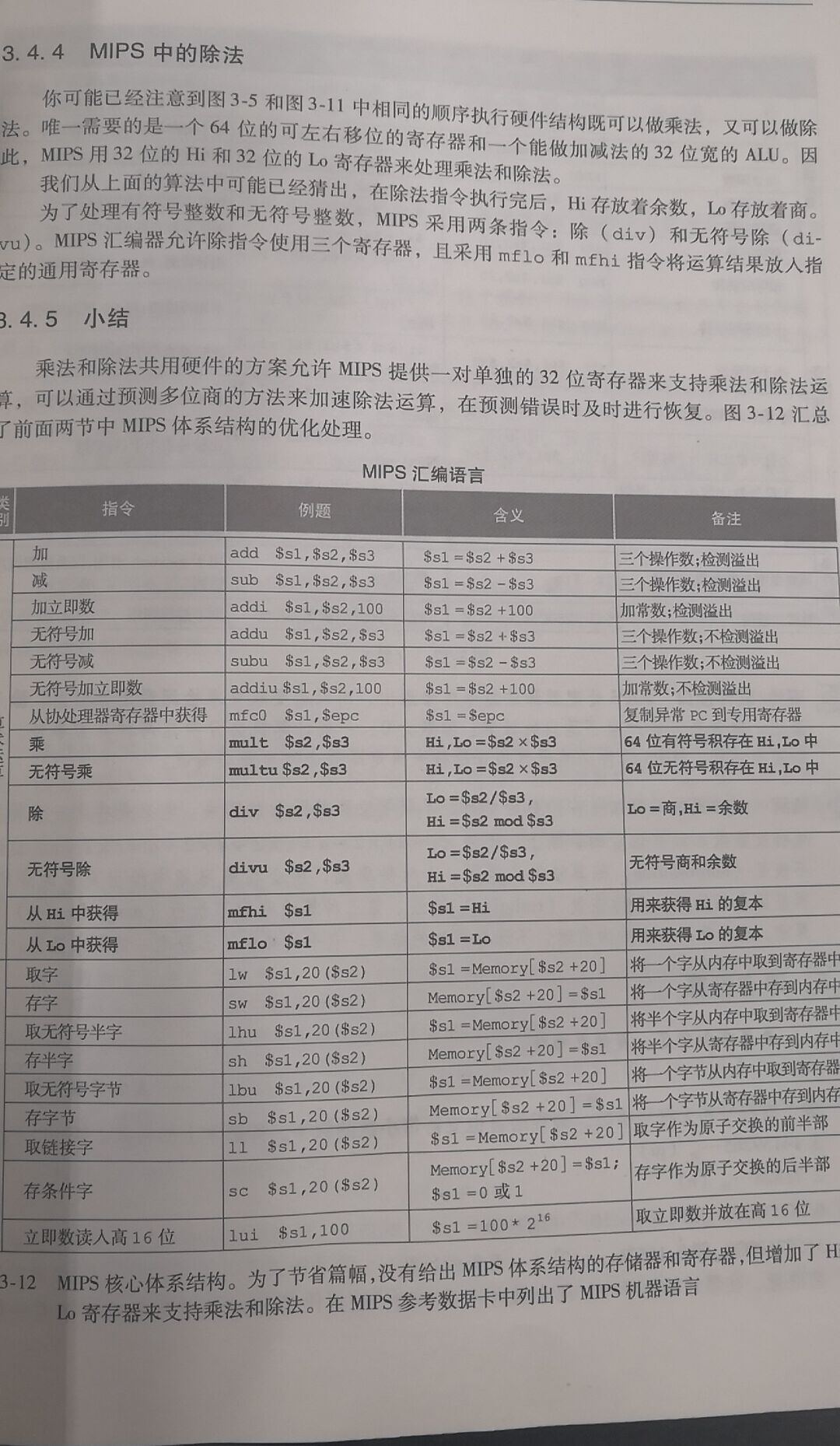
(7) T4∶=R + r

(8) T5∶=T3 \* T4

(9) T6∶=R - r

(10) B∶=T5 \* T6

## 6.17 优化的时候主要考虑的是添加addi，subi等带常数的指令，来减少对于内存的访问



在Mars程序中查找得到结果如下：所以只有加减有针对常数添加新的指令，而乘除没有。

## 6.18 乘除法优化

在乘除法中，想通过使用效率更快的移位指令，来取代对于2的幂级数的操作。

1.算​术左移和逻辑左移一样都是右边补0：

比如 00101011​

算术左移一位:01010110

逻辑左移一位​:01010110

对于二进制的数值来说左移n位等于原来的数值乘以2的n次方

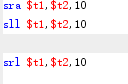
比如00011010十进制是26，左移两位后是011010​00转成十进制是104恰好是26的4倍。

ps：这种倍数关系只适用于左移后被舍弃的高位不含1的情况，否则会溢出。​

​2.算术右移，逻辑右移

逻辑右移很简单，只要将二进制数整体右移，左边补0即可，如10101101逻辑右移一位为01010110算术右移符号位要一起移动，并且在左边补上符号位，也就是如果符号位是1就补1符号位是0就补0。比如：11100算术右移一位为11110（符号位1跟着一起移动并且左边补了1）

对于二进制的数值来说右移n位等于原来的数值除以2的n次方，比如10110100十进制是76（需要先将这个补码转换成原码之后再转换成十进制），右移两位后是11101101转成十进制是19恰好是76的4倍。

而我们需要的是算术左右移指令，即上两条指令。

## 6.19 2的幂级数的快速判断

我们可以通过(number & number - 1) == 0来快速判断number是否是2的幂级数

      原因：因为2的N次方换算是二进制为10……0这样的形式(0除外)。与上自己-1的位数，这们得到结果为0。例如。8的二进制为1000；8-1=7，7的二进制为111。两者相与的结果为0。比如：

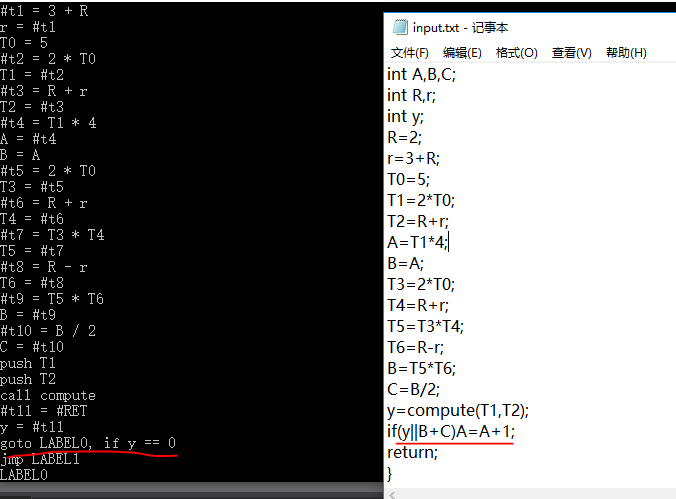
        1000  
     & 0111  
        -------  
        0000

## 6.20 文本指针的及时关闭

最开始修改之后运行程序，代码修改之后，出现mipsCode.asm中无值，而OptMIPSCode.asm中有值的情况，怀疑是fstream文本指针没有及时关闭，添加mipsFile\_old.close();及时关闭 文本指针后两个文件生成结果正常

## 6.21 修改不完善

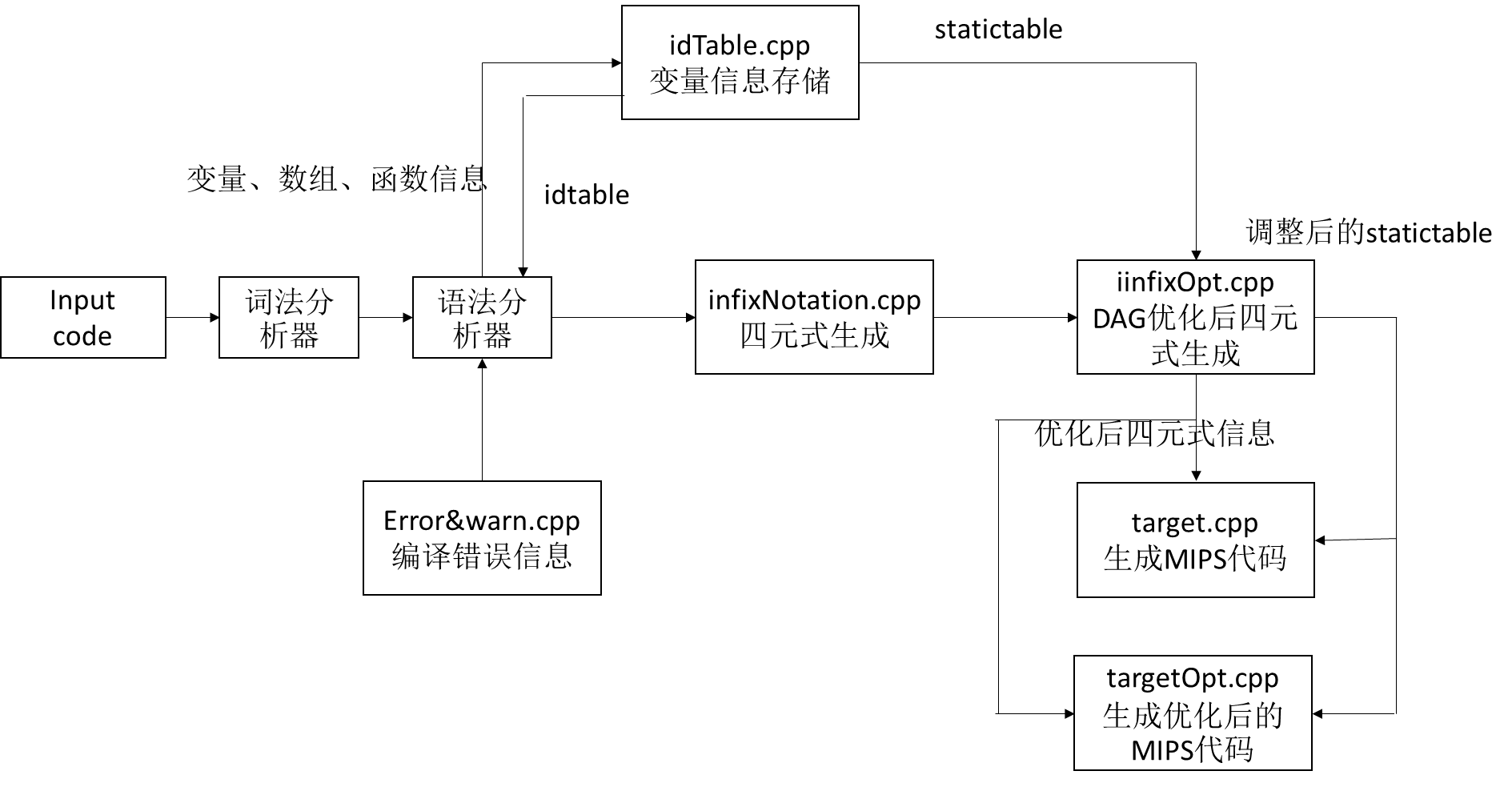
改之后程序运行正确但是优化出错，发现是因为在infixOPt中的insertNewInfix函数没添加AND和OR语句，导致没有输出相关的MIPS代码



# 七.其他附录

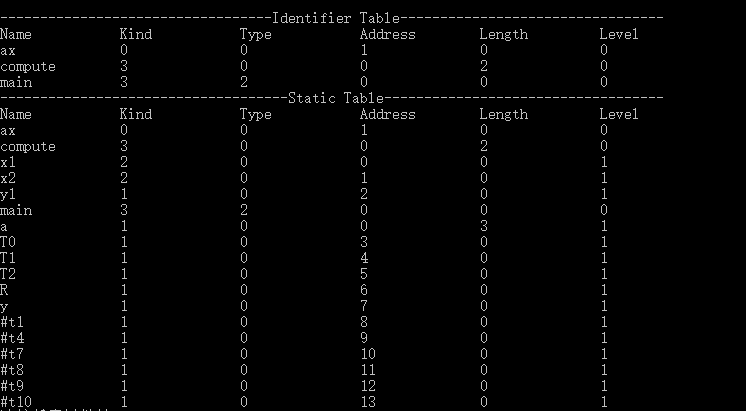
适用一个例子说明各模块间的关系

|  |  |  |
| --- | --- | --- |
| Input code | 四元式 | DAG优化后的四元式 |
| const int ax=1;  int compute(int x1,int x2)  {  int y1;  y1=x1+x2;  return (y1);  }  int main(){  int a[3];  int T0,T1,T2,T3,T4,T5,T6;  int A,B,C;  int R,r;  int y;  R=2;  r=3+R;  T0=5;  T1=2\*T0;  T2=R+r;  A=T1\*4;  B=A;  T3=2\*T0;  T4=R+r;  T5=T3\*T4;  T6=R-r;  B=T5\*T6;  C=B/2;  y=compute(T1,T2);  if(y||B+C)A=A+1;  return;  } | const int ax = 1  int compute()  param int x1  param int x2  int y1  #t0 = x1 + x2  y1 = #t0  return y1  int main()  int a[3]  int T0  int T1  int T2  int T3  int T4  int T5  int T6  int A  int B  int C  int R  int r  int y  R = 2  #t1 = 3 + R  r = #t1  T0 = 5  #t2 = 2 \* T0  T1 = #t2  #t3 = R + r  T2 = #t3  #t4 = T1 \* 4  A = #t4  B = A  #t5 = 2 \* T0  T3 = #t5  #t6 = R + r  T4 = #t6  #t7 = T3 \* T4  T5 = #t7  #t8 = R - r  T6 = #t8  #t9 = T5 \* T6  B = #t9  #t10 = B / 2  C = #t10  push T1  push T2  call compute  #t11 = #RET  y = #t11  goto LABEL0, if y == 0  jmp LABEL1  LABEL0  LABEL1  return | const int ax = 1  int compute()  param int x1  param int x2  int y1  y1 = x1 + x2  return y1  int main()  int a[3]  int T0  int T1  int T2  int T3  int T4  int T5  int T6  int A  int B  int C  int R  int r  int y  R = 2  #t1 = 3 + R  #t8 = R - #t1  T2 = R + #t1  T0 = 5  T1 = 2 \* T0  #t7 = T1 \* T2  #t9 = #t7 \* #t8  #t10 = #t9 / 2  #t4 = T1 \* 4  push T1  push T2  call compute  y = #RET  goto LABEL0, if y == 0  jmp LABEL1  LABEL0  LABEL1  return |
|  | MIPS代码 | 优化后的MIPS代码 |
|  | .data  .text  j main  compute:  sw $fp,16($sp)  add $fp, $sp,16  sw $ra,12($sp)  sub $sp, $sp, 4  lw $t1, -8($fp)  lw $t2, -12($fp)  add $t0, $t1, $t2  sw $t0, -16($fp)  lw $t0, -16($fp)  move $v1, $t0  j computeTail  computeTail:  lw $ra, -4($fp)  move $sp, $fp  lw $fp, ($sp)  jr $ra  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 12  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  sub $sp, $sp, 4  li $t0, 2  sw $t0, -32($fp)  li $t1, 3  lw $t2, -32($fp)  add $t0, $t1, $t2  sw $t0, -40($fp)  lw $t1, -32($fp)  lw $t2, -40($fp)  sub $t0, $t1, $t2  sw $t0, -52($fp)  lw $t1, -32($fp)  lw $t2, -40($fp)  add $t0, $t1, $t2  sw $t0, -28($fp)  li $t0, 5  sw $t0, -20($fp)  li $t1, 2  lw $t2, -20($fp)  mul $t0, $t1, $t2  sw $t0, -24($fp)  lw $t1, -24($fp)  lw $t2, -28($fp)  mul $t0, $t1, $t2  sw $t0, -48($fp)  lw $t1, -48($fp)  lw $t2, -52($fp)  mul $t0, $t1, $t2  sw $t0, -56($fp)  lw $t1, -56($fp)  li $t2, 2  div $t0, $t1, $t2  sw $t0, -60($fp)  lw $t1, -24($fp)  li $t2, 4  mul $t0, $t1, $t2  sw $t0, -44($fp)  sub $sp, $sp, 8  lw $t0, -24($fp)  sw $t0, ($sp)  sub $sp, $sp, 4  lw $t0, -28($fp)  sw $t0, ($sp)  sub $sp, $sp, 4  jal compute  move $t0, $v1  sw $t0, -36($fp)  lw $t1, -36($fp)  li $t2, 0  beq $t1, $t2, LABEL0  j LABEL1  LABEL0:  LABEL1:  move $v1, $t0  j mainTail  mainTail: | .data  .text  j main  compute:  sw $fp,16($sp)  add $fp, $sp,16  sw $ra,12($sp)  sub $sp, $sp, 4  lw $t1, -8($fp)  lw $t2, -12($fp)  add $t0, $t1, $t2  sw $t0, -16($fp)  lw $t0, -16($fp)  move $v1, $t0  j computeTail  computeTail:  lw $ra, -4($fp)  move $sp, $fp  lw $fp, ($sp)  jr $ra  main:  sw $fp, ($sp)  move $fp, $sp  sub $sp, $sp, 8  sub $sp, $sp, 56  li $t0, 2  sw $t0, -32($fp)  lw $t2, -32($fp)  addi $t0,$zero,3  add $t0,$t0,$t2  sw $t0, -40($fp)  lw $t1, -32($fp)  lw $t2, -40($fp)  sub $t0, $t1, $t2  sw $t0, -52($fp)  lw $t1, -32($fp)  lw $t2, -40($fp)  add $t0, $t1, $t2  sw $t0, -28($fp)  li $t0, 5  sw $t0, -20($fp)  lw $t2, -20($fp)  sll $t0,$t2,1  sw $t0, -24($fp)  lw $t1, -24($fp)  lw $t2, -28($fp)  mul $t0, $t1, $t2  sw $t0, -48($fp)  lw $t1, -48($fp)  lw $t2, -52($fp)  mul $t0, $t1, $t2  sw $t0, -56($fp)  lw $t1, -56($fp)  sra $t0,$t1,1  sw $t0, -60($fp)  lw $t1, -24($fp)  sll $t0,$t1,2  sw $t0, -44($fp)  sub $sp, $sp, 8  lw $t0, -24($fp)  sw $t0, ($sp)  sub $sp, $sp, 4  lw $t0, -28($fp)  sw $t0, ($sp)  sub $sp, $sp, 4  jal compute  move $t0, $v1  sw $t0, -36($fp)  lw $t1, -36($fp)  li $t2, 0  beq $t1, $t2, LABEL0  j LABEL1  LABEL0:  LABEL1:  move $v1, $t0  j mainTail  mainTail: |

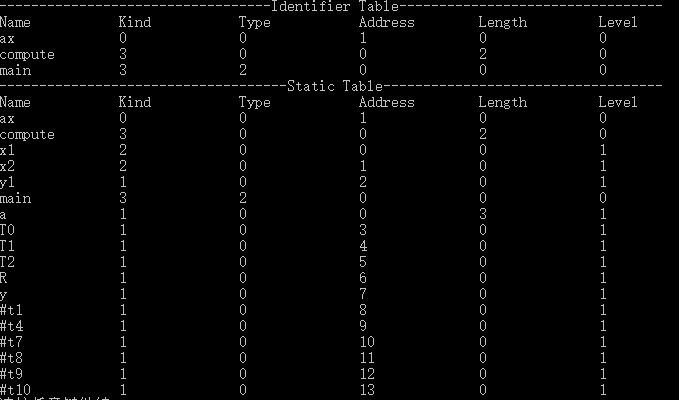


input code输入之后通过词法分析器读取变量名、函数名和关键字，在语法分析器中，将输入的代码转化为四元式，四元式信息保存在outputBuff字符串中（表中的“四元式”），同时将变量以及函数存储进idTable和statictable中，其中idtable中的变量在退出函数时，会清空该函数中的局部变量。而statictable中的变量会一直保存，并在生成MIPS代码时使用。

输入代码运行结束时，idtable为函数和全局变量，statictable内容为：



随后进入infixOpt.cpp文件中，根据outputBuff字符串存储的四元式信息建立DAG树，并以此进行优化，得到newoutputBuff字符串，存储优化后的四元式信息（表中的“DAG优化后四元式”），同时调整statictable。



最后生成MIPS代码时，根据DAG优化后的四元式newoutputBuff以及调整后的statictable信息获得MIPS代码（上表中的“MIPS代码”），随后再根据以上信息，对MIPS代码进行优化输出，获得优化后的MIPS代码（上表中的“优化后的MIPS代码”）。

# 八.心得体会

通过本次实验，我第一次接触到一个大型程序的编写全过程，本次代码的长度大致在4000-5000，在我学习生涯中也算是一个很大的项目。最后完成的成果让人兴奋，但是更加重要的应该是在这个过程中学习到的内容。

1.深入理解编译原理中一个编译器的具体实现。在本次实验中，一开始先是实现了SLR和LL1递归下降算法的四元式生成。随后参照C0文法编译器实现一个完整的编译器，包括前端和后端、优化等部分。在这个过程中，我深刻理解了词法分析器、文法分析器、优化等部分的原理。相较于刻板的课本理解，这种方式更加深刻和灵活。

2.接触如此一个大型的C++程序的编写，我接触到更多的C++语言的知识，如枚举类的使用、模板的使用、以及头文件的编写等知识，在不解、查询、理解、熟悉的过程中，我对C++语言更加熟悉。在之后的编程中，我也能使用这些结构体和知识来进一步优化代码，减少代码的冗余。

3.在大型程序的编写中功能的模块化以及历史版本的保留非常重要。这样对于后续功能的修改，错误的纠正都有很大的帮助。这实际上也是软件工程的相关内容，只可惜这门课当时上的比较晚，在本次实验中未能较好的结合在一起。同时注意随时记录程序中遇到的问题和解决方式，这是日后学习的一笔财富，能够避免后续编程过程中遇到类似错误，或者对于其他项目中的错误也能起到一定的参考价值。