《编译技术》

课程设计文档

学号： 15061144

姓名： 陈鑫

二〇一七年十一月二十日

## 一．需求说明

### 1．文法说明

＜加法运算符＞ ::= +｜-  
＜乘法运算符＞  ::= \*｜/  
＜关系运算符＞  ::=  <｜<=｜>｜>=｜!=｜==  
＜字母＞   ::= ＿｜a｜．．．｜z｜A｜．．．｜Z  
＜数字＞   ::= ０｜＜非零数字＞  
＜非零数字＞  ::= １｜．．．｜９  
＜字符＞    ::=  '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'  
＜字符串＞   ::=  "｛十进制编码为32,33,35-126的ASCII字符｝"  
＜程序＞    ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}  
 ＜主函数＞  
＜常量说明＞ ::=  const＜常量定义＞;{ const＜常量定义＞;}  
＜常量定义＞   ::=   int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}  
                               | char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}  
＜无符号整数＞  ::= ＜非零数字＞｛＜数字＞｝  
＜整数＞        ::= ［＋｜－］＜无符号整数＞｜０  
＜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＞‘]’ )}   
＜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＞‘{’＜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＞‘)’‘{’＜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＞    ::= ＜参数表＞  
＜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＞}  
＜项＞     ::= ＜因子＞{＜乘法运算符＞＜因子＞}  
＜因子＞    ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’|‘(’＜表达式＞‘)’｜＜整数＞|＜字符＞｜  
 ＜有返回值函数调用语句＞           
＜语句＞    ::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;   
                           |＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;   
 |＜情况语句＞｜＜返回语句＞;  
＜赋值语句＞   ::=  ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞  
＜条件语句＞::= if ‘(’＜条件＞‘)’＜语句＞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＞   ::=  while ‘(’＜条件＞‘)’＜语句＞  
＜情况语句＞  ::=  switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞[＜缺省＞] ‘}’  
＜情况表＞   ::=  ＜情况子语句＞{＜情况子语句＞}  
＜情况子语句＞  ::=  case＜常量＞：＜语句＞  
＜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＞‘)’|<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类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符  
（2）标识符不区分大小写字母  
（3）写语句中的字符串原样输出   
（4）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句   
（5）数组的下标从0开始

### 2．目标代码说明

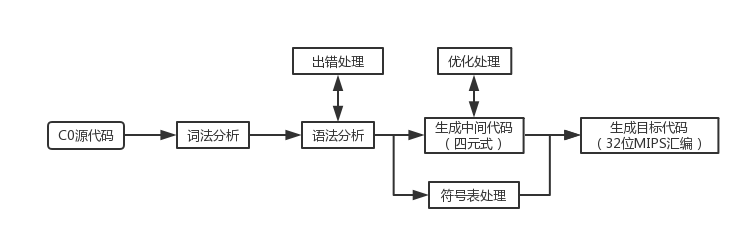
32位MIPS汇编指令，详见MARS - help

### 3. 优化方案\*

* 基本块内部的公共子表达式删除（DAG图）；
* 全局寄存器分配（引用计数）；
* 代码生成时合理利用临时寄存器（临时寄存器池），并能生成较高质量的目标代码；

## 二．详细设计

### 1．程序结构



### 2．类/方法/函数功能

void error(int no); // 输出错误信息，退出程序

void printTabs(); // 输出符号表信息

void getch(); // 读取源代码的一个字符

void getsymbol(); // 读取一个单词

string genLabel(int i); // 生成标签

void genQuard(string a, string b, string c, string d, string e, string f); // 生成四元式

void enter(kinds kind, types type, int block); // 登录符号表

void enterArray(types eltype, int block); // 登录数组到符号表

int findTab(string name); // 返回标识符在符号表中的位置

void constDefi(types ic, int block); // 处理一个常量定义

void constsDefi(int block); // 处理一行常量定义

void varDefi(types ic, int block); // 处理一个变量定义

void varsDefi(types ic, int block); // 处理一行常量定义

void parameter(int block); // 处理一个参数

void parameters(int block); // 处理<参数表>

quad\_item call(int position); // 处理函数调用语句

quad\_item factor(); // 处理<因子>

quad\_item term(); // 处理<项>

quad\_item expression(); // 处理<表达式>

void condition(); // 处理<条件>

void statement(); // 处理<语句>

void comState(int block); // 处理<复合语句>

void funDefi(types tp); // 处理一个函数定义

void fun\_main(); // 处理主函数定义

void program(); // 处理<程序>

void optimize(); // 优化四元式

void optimize\_count(); // 引用计数法优化

bool cmp\_by\_value(const pair<string, int>& lhs, const pair<string, int>& rhs); // 比较函数，用于计数排序

void var\_count(map<string, int> &local\_count\_map, map<string, int> &global\_count\_map, string name); // 统计

void transform(); // 四元式转MIPS

void transform\_opt(); // 优化后的生产目标代码

### 3．调用依赖关系

program调用constsDefi、varsDefi、funDefi、fun\_main；

optimize\_count()调用var\_count、cmp\_by\_value

constsDefi调用constDefi；

constDefi调用enter；

varsDefi调用varDefi；

varDefi调用enter、enterArray；

funDefi调用enter、parameters、comState、genQuard；

parameters调用parameter；

parameter调用enter、genQuard；

comState调用constsDefi、varsDefi、statement；

statement调用condition、expression、find、genLabel、genQuard以及自身递归调用；

condition调用expression、genQuard；

expression调用term、genQuard；

term调用factor、genQuard；

factor调用find、expression、call、genQuard；

call调用expression、genQuard ；

fun\_main调用comState;

getsymbol调用getch；

error和getsymbol作为基础功能函数几乎被每个函数都调用。

### 4．符号表管理方案

/\* 符号表 \*/

enum kinds { CONSTANT, VARIABLE, FUNCTION };

extern string kinds\_string[3];

enum types { INTTP, CHARTP, ARRAYTP, VOIDTP };

extern string types\_string[4];

struct tab\_item { // tab的项

string name; // 标识符名

kinds kind; // 标识符种类：常量、变量、函数

types type; // 标识符类型：整型、字符型、数组、无返回值函数

int ref; // 当标识符类型为数组时，记录其在atab中的位置；为函数时，记录其在ftab中的位置

int block; // 所属区块，0为全局，其余为各局部编号

int adr; // 常量的值 或 变量在S中的相对地址

};

extern vector<tab\_item> tab;

extern size\_t pt\_block; // 区块标记指针

/\* 数组信息表 \*/

struct atab\_item {

types eltype; // 元素的类型

int length; // 数组的长度

};

extern vector<atab\_item> atab;

/\* 函数信息表 \*/

struct ftab\_item {

int lastpara; // 函数最后一个参数在符号表中的位置

int size; // 参数、局部变量在运行栈中所占的存储单元总数

int tsize; // 临时变量(T)所占大小

};

extern vector<ftab\_item> ftab;

/\* 字符串表 \*/

extern vector<string> stab;

程序先用struct定义表项结构，再以vector的方式建立动态数组来保存符号表。通过push\_back()实现表项的插入，通过数组下标进行数组的访问和修改。

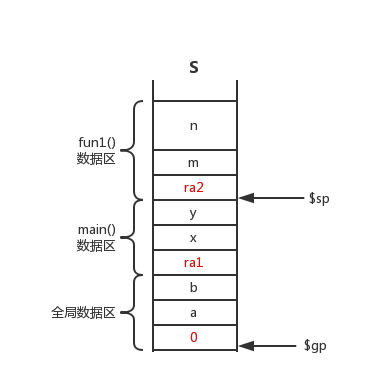
### 5．存储分配方案

S栈保存各层的数据区和返回地址。

从main函数开始执行，令$sp = $gp

进入别的函数体前，先保存当前函数返回地址至$sp，

再将$sp加上当前数据区大小，即为新函数的基址。

从别的函数返回后，先将$sp减去当前数据区大小，

再将之前保存的返回地址赋给$ra。

通过符号表中记录的变量相对地址adr，再加上当前层基址$sp，即可确定变量在运行栈中的绝对地址，从而进行读写操作。

例如：

int a;

char b;

void fun1(int m){

int n[10];

}

void main(){

int x,y;

fun1(1);

}

此外，还用了一个栈来保存函数的中间变量和局部寄存器，原理同上。各函数的中间变量内存大小保存在ftab.tsize中。

### 6. 解释执行程序\*

C0文法最终目标是生成MIPS汇编，无需解释执行。

### 7. 四元式设计\*

原则上按照中缀表达式格式输出中间代码，即，形如x = y op z，其中x为结果，y为左操作数，z为右操作数，op为操作符。以下根据基本语法现象举例说明。

1. 函数声明

源码形如：

int foo( int a, int b, int c, int d)

中间代码：

int foo ( ) （参数信息保存在符号表中）

2. 函数调用

源码形如：

i = tar(x,y)

中间代码：

PUSH x

PUSH y

CALL tar

i = RET

3. 函数返回

源码形如：

return (x)

中间代码：

RET x

4. 变量声明

源码形如：

int i, j;

符号表信息输出，程序中不生成真正的中间代码

5. 常数声明

源码形如：

const int c = 10

符号表信息输出，程序中不生成真正的中间代码

const int c = 10

6. 表达式

源码形如：

x = a \* (b + c)

中间代码（可优化）：

t1 = b + c

t2 = a \* t1

x = t2

7. 条件判断

源码形如：

x == y

中间代码：

x == y

8. 条件或无条件跳转

中间代码：

GOTO LABEL1 //无条件跳转到LABEL1

BNZ LABEL1 //满足条件跳转到LABEL1

BZ LABEL1 //不满足条件跳转到LABEL1

9. 带标号语句

源码形如：

Label\_1: x = a + b

中间代码：

Label\_1 :

x = a + b

10. 数组赋值或取值

源码形如：

a[i] = b \* c[j]

中间代码：

T1 = c[j]

T2 = b \* t1

a[i] = T2

11. 读写函数

源码形如：

scanf(a,b,c);

中间代码：

SCANF a

SCANF b

SCANF c

源码形如：

printf(“hello”,a);

中间代码：

PRINTF “hello”

PRINTF a

### 8. 目标代码生成方案\*

【说明代码生成有关的数据结构、关键算法】

采用$gp保存全局数据区基地址，$sp保存当前函数的数据区基地址。

初始$sp=$gp+ftab[0].size，之后每调用一次函数，$sp加上本函数数据区的size，返回时再减去。这样通过$sp和符号表中的adr即可确定变量的绝对地址。

四元式结构：

struct quaditem {

string name;

types type;

};

struct quadruple {

quaditem a, b, c, d, e, f;

};

extern vector<quadruple> quad\_code;

extern int pt\_r; // 中间变量指针

根据四元式的结构，确定语句的类型，结合MIPS32指令集进行生成。

### 9. 优化方案\*

1. 表达式常量合并

诸如a=1-2-3+4;的语句在语义分析时会进行常量合并，直接生成a=0这样的中间代码，从而减少汇编代码量。

1. 公共子表达式的优化

诸如x=a+b;y=a+b;的语句，在转化成四元式时会优化为T1=a+b;

x=T1;y=T1;这样的中间代码。原理为利用DAG图重复利用同一基本块的公共子表达式，实现结构如下：

struct dag\_node {

int id;

string op;

int x, y; // 左右子结点号

}node;

struct dag\_item {

string name;

int id;

}item;

1. 采用引用计数进行全局寄存器的分配

对源代码中出现的变量进行统计，将$s0~$s6分配给出现次数最多的全局变量，将$t0~$t7分配给各函数体内出现次数最多的局部变量（包括四元式产生的中间临时变量）。

map<string, int> local\_count\_map;

map<string, int> global\_count\_map;

### 10. 出错处理

【说明出错处理方案、错误信息及含义】

void error(int no) {

string msg[100];

msg[1] = "超过int上限"; msg[2] = "缺少'"; msg[3] = "字符串内容出错";

msg[4] = "!只能与=连用"; msg[5] = "无法识别符号";

msg[6] = "标识符重复定义"; msg[7] = "缺少(或;";

msg[8] = "void函数不能有返回值或跟()";msg[9] = "函数缺少返回值返回值应在()内";

msg[10] = "缺少标识符"; msg[11] = "缺少="; msg[12] = "缺少int或char";

msg[13] = "缺少const"; msg[14] = "常量类型出错"; msg[15] = "缺少;";

msg[16] = "缺少整数"; msg[17] = "缺少]"; msg[18] = "无参数的函数调用不能跟()"; msg[19] = "传入参数过多"; msg[20] = "传入参数过少"; msg[21] = "缺少)";

msg[22] = "标识符未定义"; msg[23] = "[使用出错"; msg[24] = "调用函数无返回值";

msg[25] = "因子类型出错"; msg[26] = "缺少("; msg[27] = "缺少else";

msg[28] = "标识符非变量"; msg[29] = "缺少case"; msg[30] = "缺少常量标号"; msg[31] = "case标号重复定义"; msg[32] = "缺少:"; msg[33] = "缺少[";

msg[34] = "不能对常量进行赋值"; msg[35] = "缺少}"; msg[36] = "无法识别语句";

msg[37] = "变量定义出错"; msg[38] = "缺少{"; msg[39] = "函数定义出错";

msg[40] = "main函数类型必须为void"; msg[41] = "缺少main函数";

msg[42] = "main函数后有多余字符"; msg[43] = "case常量类型错误";

msg[44] = "标识符定义出错"; msg[45] = "非0整数不能以0开头";

msg[46] = "无参数函数定义时不允许加()"; msg[47] = "<整数>0前不允许有正负号";

msg[48] = "数组下标越界";

fout << endl << msg[no];

fin.close();

fout.close();

exit(no);

}

语法分析时出错则调用该函数，并输出相关错误信息。

## 三．操作说明

### 1．运行环境

VS2010

### 2．操作步骤

打开项目工程，启动调试，输入源文件和输出文件的绝对地址。若源码正确，可将生产的MIPS指令拷贝进Mars4\_5执行；若源码有问题，则打开输出文件查看错误位置和信息。

## 四．测试报告

### 1．测试程序及测试结果

正确程序：

1. void fun{

const char c='c';

int a[2],b;

b=0\*(1+'C'+c);

a[B]=('c'-1)/2;

a[1]=a[0]+1;

if(a[B]>=49)

if(a[b]!=49)

printf("fun() error");

else

printf("Success");

else

printf("fun() error");

{

{}

}

;

}

void main(){

fun;

}

结果：

输出“Success”。

1. void choose{

int x,n;

scanf(x);

switch(x){

case 1: {

printf("case 1");

}

case 2:

printf("case 2");

default:

printf("default");

}

}

void main(){

choose;

}

输入：0

输出：default

输入：1

输出：case 1

输入：2

输出：2

1. int check1(int n){

if(n<=0)

return(0);

else

return(n);

}

void move(int n,char a,char b,char c){

if(n==1){

printf(a);

printf("->",c);

printf(";");

}

else {

move(n-1,a,c,b);

printf(a);

printf("->",c);

printf(";");

move(n-1,b,a,c);

}

}

void main(){

int n;

scanf(n);

n=check1(n);

if(n)

move(n,'A','B','C');

else

printf("The number should be greater than 0.");

}

输入：-1

输出：The number should be greater than 0.

输入：2

输出：A->B;A->C;B->C;

输入：3

输出：A->C;A->B;C->B;A->C;B->A;B->C;A->C;

1. int n;

int check2{

int n,f;

f=0;

scanf(n);

while(f<2){

f=0;

if(n<1){

printf("The number should be between 1 and 26.Please enter again:");

scanf(n);

}

else

f=f+1;

if(n>26){

printf("The number should be between 1 and 26.Please enter again:");

scanf(n);

}

else

f=f+1;

}

return(n);

}

void main(){

char alphabet[26];

int i;

i=0;

while(i<26){

alphabet[i]='a'+i;

i=i+1;

}

n=check2;

printf(alphabet[n-1]);

return;

}

输入：0

2

输出：The number should be between 1 and 26.Please enter again:

b

输入：27

26

输出：The number should be between 1 and 26.Please enter again:

z

1. int x,y,z;

void main(){

int a,b,c[5];

x=0+0\*123;

y=1-0/123;

z=2\*(120-119);

c[0]=x+y;

a=x+y+z;

b=x+y+z;

c[1]=c[0];

c[a-3]=0;

c[2]=c[0];

printf(a);

printf(" ",b);

printf(" ",c[0]);

printf(" ",c[1]);

printf(" ",c[2]);

}

输出：3 3 0 1 0

错误程序：

1. void main(){

int a[+10];

}

结果：

报错“第2行：缺少无符号整数”。

2. void main(){

int a;

a=+-0;

}

结果：

报错“第3行：<整数>0前不允许有正负号”。

3. int a;

const int b;

void main(){}

结果：

报错“第2行：变量定义出错”。（变量定义应在常量定义之后）

4. int fun(){ }

void main(){ }

结果：

报错“第1行：无参数函数定义时不允许加()”。

5. void main(){ }

int fun(){ }

结果：

报错“第5行：main函数后有多余字符”。

### 2．测试结果分析

基本覆盖完所有文法，递归、循坏和公共子表达式的优化也都进行了测试，容易出现的语法错误、语义错误也能正常报错。

## 五．总结感想

通过历时两个月左右的编译课设，自己亲手写出了一个能编译简单C0文法的编译器，总代码量达到了近三千行，总的来说还是很有成就感的。在理论课上学到各种基础知识，再在实战中加以运用，这对我们这样的工科学生是很有必要的，学院开设的核心专业课也大多采取这样的学习方式，个人认为是十分合理的。因为亲自实现了编译的各个阶段，所以对编译流程了解清晰，能够高屋建瓴地看待整个编译工作。做完课设的受益颇多：一是对编译课的理论知识理解得更加透彻，毕竟独立构造出一个完整的编译器需要将许多的诸多相关知识结合起来，在这个过程中，知识得到了巩固和提高；二是复习了以前所学课程的知识，比如符号表、运行栈的管理就用到了数据结构课学到的知识，最终生成汇编代码更是需要我们用到计算机组成课程中学到的MIPS指令集、MARS的使用等知识，这算是对过往所学的一个很好复习和应用了；三是提高了自己代码能力，例如之前写代码不清楚局部变量和全局变量同名的情况应当如何处理，现在学了编译知道是通过符号表由后向前查找标识符。总而言之，编译课设对我们更加深刻地理解程序内部的结构有着莫大帮助。