《编译技术》课程设计文 档

学号：\_\_\_\_\_\_\_16061184\_\_\_\_\_\_\_\_\_\_\_\_\_

姓名：\_\_\_\_\_\_\_\_王冬冬\_\_\_\_\_\_\_\_\_\_\_\_

2018年 11 月 18 日

# 一．需求说明

## 1．文法说明

获取的文法如下：

1、＜加法运算符＞ ::= +｜-

范例：c=a+b，d=a-b

分析：加法运算符前后是两个项，进行加减运算，或者在一个项前表示正负

2、＜乘法运算符＞  ::= \*｜/

范例：c=a\*b,d=a/b

分析：乘法运算符前后是两个项，进行乘除运算

3、＜关系运算符＞  ::=  <｜<=｜>｜>=｜!=｜==

范例：if(a!=b){}

if(c==d){}

分析： <表示小于；<=表示小于等于；>表示大于；>=表示大于等于；!=表示不等于；==表示恒等于；这些比较符号多用于条件语句中，作为执行条件

4、＜字母＞   ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

示例：char flag::=v;

分析：字母包括小写字母a-z和大写字母A-Z和下划线

5、＜数字＞   ::= ０｜１｜．．．｜９

示例：int line::=90;

分析：数字包括0至9的数字

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

示例：’+’，’0’

分析：字符包括+，-，\*，/，字母和数字，不包括特殊符号

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

示例：chat s=”\*bu6 6\_a^5”

分析：字符串和字符没有确切关系，包含广泛，长度不受限制

8、＜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＞

示例：const int a=5,b=7;

const char sll=’v’,sdd=’b’;//常量说明

int c,line[10];//变量说明

int fun(int x,char y){fun分程序(略)}

void sort(int d[5],int e){sort分程序(略)}

int main(){主程序(略)}

分析：常量说明部分与变量说明部分可有可无但顺序不能变，有返回值函数与无返回值函数可以没有可以有多个，且顺序没有要求，主函数必须在最后，且只有一个

9、＜常量说明＞ ::=  const＜常量定义＞;{ const＜常量定义＞;}

示例：const int a=5,b=7;

const char sll=’v’,sdd=’b’;

分析：常量说明部分可以有多个const，但每一个const中只包含一种类型int或char，该类型的声明至少有一个

10、＜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＞}

范例：int line = 20, cc = 50;

char s = ’k’, flag = ‘ a’;

分析：常量定义只包含一种类型，int或char，每种类型都可以有多个声明

11、＜无符号整数＞  ::= ＜数字＞｛＜数字＞｝

范例：int a = 200398;

分析：无符号整数由0到9之间的字符串组成，没有长度顺序限制

12、＜整数＞        ::= ［＋｜－］＜无符号整数＞

范例：整数在无符号整数的前面加上加号或减号，加号或减号不一定要有

13、＜标识符＞    ::=  ＜字母＞｛＜字母＞｜＜数字＞｝

示例：char lost2first = ‘d’;

分析：标识符的第一个字符必须是字母，其后的字符为字母或数字，顺序数量不限

14、＜声明头部＞   ::=  int＜标识符＞ |char＜标识符＞

示例：int fun(char s,int a){}

分析：声明头部用于声明函数的类型和名字，类型有int、char两种，名字为标识符

15、＜变量说明＞  ::= ＜变量定义＞;{＜变量定义＞;}

示例：int add, line[22];

char s, str[45];

分析：变量声明包括至少一个变量定义

16、＜变量定义＞  ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’ )}  //＜无符号整数＞表示数组元素的个数，其值需大于0

示例：int add, line[22];

分析：每一个变量定义包含一个int或char，其后为一般变量或数组，数组方括号中必须为无符号整数，一般变量和数组一共至少有一个，声明顺序不限

17、＜类型标识符＞      ::=  int | char

示例：int a = 8;char prinf(int num,char cs){}

分析：类型标识符包括int类型和char类型，用于声明函数或变量、常量的类型

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

示例：int fun(int a, int b){

int add, line[22];

if(a<b){

add=3;

}

else{

add=2;

}

return add;

}

分析：有返回值函数定义首先为声明头部，用于定义返回值的类型，有参数的函数有参数表，无参数的函数无参数表，函数主体为复合语句。

19、＜无返回值函数定义＞  ::= void＜标识符＞(’＜参数表＞‘)’‘{’＜复合语句＞‘}’| void＜标识符＞{’＜复合语句＞‘}’//第一种选择为有参数的情况，第二种选择为无参数的情况

示例：void sort{

printf(“there is an error!”);

}

分析：无返回值函数定义开始必须为void<标识符>，有参数的函数有参数表，无参数的函数无参数表，函数主体为复合语句。

20、＜复合语句＞   ::=  ［＜常量说明＞］［＜变量说明＞］＜语句列＞

示例：const int a=5,b=7;

int add, line[22];

if(a<b){

add=3;

}

分析：复合语句中常量说明和变量说明可有可无，但顺序不可变，其后为语句列

21、＜参数表＞    ::=  ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

示例：int fun(int a, char b){}

分析：参数表用于有参数的函数，参数由函数外部输入，进入函数参与运算，函数内部对参数的改变会影响外部

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

示例：void main(){

Printf(“Hello Word!”);

}

分析：主函数不返回值，开头为void，由于为最外层的函数，括号内没有参数，函数主体为复合语句。

23、＜表达式＞    ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}  //[+|-]只作用于第一个<项>

示例：-30+ fun(int a,int b)

分析：表达式至少包含一个项，相邻两项间必须有加法运算符+或-，第一个项前面可以有+或-表示正负，也可以没有默认无符号

24、＜项＞     ::= ＜因子＞{＜乘法运算符＞＜因子＞}

示例：line \* sd

分析：项由至少一个因子组成，相邻因子间必须有乘法运算符\*或/

25、＜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＞

示例：line[2]

fun(int a,int b)

分析：因子可以为标识符，标识符必须有值，可以为数组元素，可以为(表达式)，可以为整数，可以为字符，计算时转为assic码，可以为有返回值调用语句，因子必须能够计算出值，能够用于计算

26、＜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＞;|＜返回语句＞;

示例：while(a){

printf(“ad”);

a=a-1;

}

分析：语句包含条件语句、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、空语句、返回语句，只含有一个上述成分

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

示例：first=a+b;

line[2]=c+d;

分析：赋值语句等号前面为标识符或数组元素，等号后的表达式的值赋给该成分

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

示例：if(a==b){

printf(“true”);

}

else{

printf(“false”);

}

分析：条件语句必须有if和紧跟其后的语句成分，else及其后的语句成分可有可无

29、＜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

示例：a+b<=c

分析：条件为一个表达式或两个表达式和其之间的关系运算符，表达式可以计算出值来进行比较，关系运算符为<｜<=｜>｜>=｜!=｜==，用于比较得出true或flase，进行条件判断

30、＜循环语句＞   ::=  while ‘(’＜条件＞‘)’＜语句＞ |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞  
 示例：for(a=3; a<10;a=a+1){

printf(a);

}

分析：循环语句包括while循环和for循环，while循环的括号内为条件，主体为语句，for循环括号内有三部分，第一部分为＜标识符＞＝＜表达式＞，用于给标识符赋初值，第二部分为条件，判断是否结束循环，第三部分为＜标识符＞＝＜标识符＞(+|-)＜步长＞，改变表示符的值，进行下一次循环和判断

31、＜步长＞::= ＜无符号整数＞

示例：6

分析：步长用于每次循环结束后对标识符进行改变

32、＜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

示例：add(a+b, c);

分析：有返回值函数调用语句用于调用有返回值函数，得到返回值，有参数时括号内为传入函数并进行运算的参数

33、＜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

示例：fun(char c)；

分析；无返回值函数调用语句用于调用无返回值函数，进行相关操作，有参数时括号内为传入函数并进行运算的参数

34、＜值参数表＞   ::= ＜表达式＞{,＜表达式＞}

示例：a+b,c

分析：值参数表至少包含一个表达式，相邻表达式间有逗号隔开，每个表达式都是一个参数，传入函数进行相关运算。

35、＜语句列＞   ::= ｛＜语句＞｝

示例：add(a+b, c);

fun(char c)；

for(a=3; a<10;a=a+1){

printf(a);

}

分析：语句列包含任意多个语句，可以为空

36、＜读语句＞    ::=  scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

示例：scanf(a,b,c);

分析：读语句读取外部输入的值并传递给变量

37、＜写语句＞    ::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’| printf ‘(’＜字符串＞ ‘)’| printf ‘(’＜表达式＞‘)’

示例：printf(“hello world”,a+b);

分析：写语句用于输出字符换或表达式的值，每个写语句最多包含一个字符串和一个表达式，且至少包含其中之一，字符串和表达式同时存在时，顺序不能变

38、＜返回语句＞   ::=  return[‘(’＜表达式＞‘)’]

示例：return a;

分析：返回语句用于返回函数值或跳出循环，没有表达式时，不返回，跳出循环或函数

## 2．目标代码说明

将四元式转换为MIPS指令，MIPS指令大致包括如下：

1. sw

示例：sw $1,10($2)

含义：memory[$2+10]=$1

1. lw

示例：lw $1,10($2)

含义：$1=memory[$2+10]

1. mflo

示例：mflo rd

含义：$rd<-LO

1. mul

示例：mult rd,rs,rt

含义：$rd=$rs\*$rt

5、add

示例：add $1,$2,$3

含义：$1=$2+$3

1. sub

示例：sub $1,$2,$3

含义：$1=$2-$3

7、subi

示例：sub $1,$2,,-1

含义：$1=$2-(-1)

8、beq

示例：beq $1,$2,10

含义：if($1==$2)    
  goto PC+4+40

1. bgez

示例：bgez rs,10000

含义：if $rs>=0

then goto 10000

10、bltz

示例：bltz rs,10000

含义：if $rs<0

then goto 10000

1. bne

示例：bne $1,$2,10

含义：if($1!=$2)   
  goto PC+4+40

12、blez

示例：blez rs,10000

含义：if $rs<=0

then goto 10000

13、div

示例：div rs,rt

含义：(HT,LO)<-$rs/$rt

14、j

示例：j 10000

含义：goto 10000

1. jr

示例：jr $31

含义：goto $31

16、jal

示例：jal 10000

含义：$31<-PC+4;  
  goto 10000

## 3. 优化方案

基本块优化：构建基本块的DAG图可以完成消除局部公共子表达式等优化，采用从DAG导出中间代码的启发式算法还可以重新导出进一步优化的中间代码，方便以后的其他优化。

窥孔优化：由每一条中间代码的输入得到目标代码后，会存在大量的冗余指令或低效率指令，窥孔优化通过关注目标指令的一个较短序列，能够发现多余的指令或者化简指令。好的窥孔优化方案需要对目标体系结构、指令集都有充分的了解。

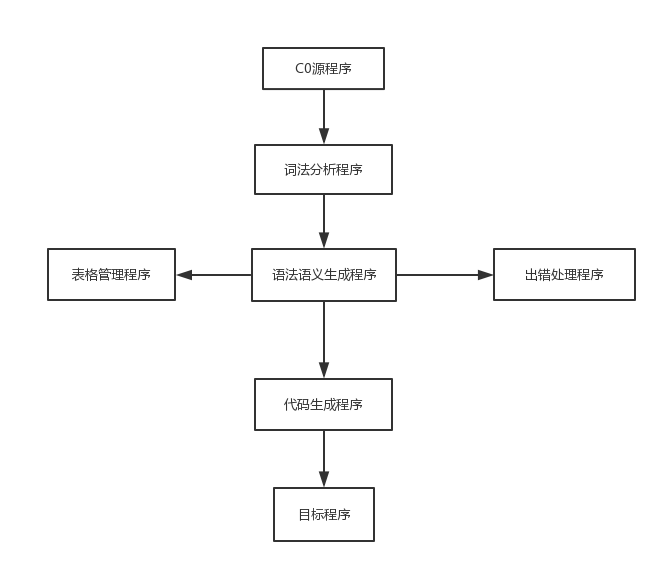
常数合并和传播：这种优化能在将编译时计算出值得表达式用其相应的值代替。

全局优化：通过数据流分析，计算每个程序执行点的可用表达式来判断公共子表达式；对于形如x=y的赋值语句，采用复制传播优化；对于不再被引用的定义变量，删除其定义点。

循环优化：循环体内部优化包含多重循环内外层交换以提高访问速度；循环展开以空间换取时间；代码外提，消除无意义的语句循环；代码强度削弱，代码长度开销换取运算速度 。

# 二．详细设计

## 1．程序结构



## 2．类/方法/函数功能

词法分析程序主要函数为getword和print，geuword作用是从输入文本测试程序中得单词，并进行分析，print由getword的分析结果进行输出。

语法分析程序的主要函数为符号表操作的相关函数以及文法中各个规则所确立的函数在主函数中的体现

中间代码生成程序主要函数为各个负责目标码输出的函数

## 3．调用依赖关系

语法分析程序依赖词法分析程序的输出结果

函数间的大量调用主要存在于语法分析程序中，语法分析程序完全按照文法规则中的关系，由主函数逐步调用各个分程序及语句函数。

中间代码生成程序负责目标码输出的函数需要调用对目标码表进行相关处理的函数

### 4．符号表管理方案

符号表主要包括标识符的名字、种类、性质、值、地址、大小、作用域的嵌套层次、标识符声明的源程序行号、标识符引用的源程序行号、以字母顺序排列的链域等。对于数组还要考虑维数，上下界值，计算下标变量地址所用的信息以及数组元素类型等；对于记录，还要考虑域的个数，每个域名、地址位移、类型等；对于过程或函数，还要考虑形参个数，所在层次，函数返回值类型、局部变量所占空间大小等；对于指针，要考虑所指对象类型等。

符号表的组织方式有建立栈式符号表和散列符号表的栈式实现。

栈式符号表在遇到标识符声明时就将包含有标识符属性的记录推入堆栈，到达分程序结尾时将在这个分程序中声明的所有标识符的记录移出堆栈。由于一个分程序的符号表信息在整个编译过程中的不同环节都可能用到，所以在确定一定不会用到前不能彻底删除分程序的信息。

栈式实现散列符号表时，新纪录被插入到符号表的顶端，分程序索引指向每个分程序的第一个记录项，散列表则存放映射到某一位置的标识符在符号表中的位置，若标识符冲突，则存放最新记录在符号表中的位置，冲突记录间用链域相连。

此外管理符号表时还要用到定位和重定位，从而降低插入符号表的开销。

## 5．存储分配方案

由于存在递归函数的可能，所以C0程序存储采用动态存储分配。

动态存储分配用运行栈来实现。进入一个模块时，就在运行栈的栈顶创建一个活动记录，活动记录包括局部数据区、参数区和display区。局部数据区局部数据区存储该模块的局部变量。参数区包含隐式参数和显示参数，隐式参数包含指向前一个活动记录基的指针和一个返回值；显示参数一般为形式参数。display区保存对当前模块来说相当于全局的其他模块的开始地址，用于传递对当前模块来说相当于全局变量的变量。

## 6. 四元式设计

四元式的每条指令具有4个域：<操作符>,<操作数1>,<操作数2>,<结果>

<操作数1>和<操作数2>分别表示第一和第二个操作数；<结果>表示计算的结果，通常为临时变量。

## 7. 目标代码生成方案

编译器在代码生成阶段需要为目标代码本身及各种变量和其他数据结构指定存储空间，这需要在程序运行时对地址空间进行分配，通常会被分为4个区域。代码区存放目标代码，目标代码所需空间大小在编译过程中可得知。静态数据区保存程序的全局变量和静态变量，该区域大小在编译过程中可得。动态内存区也被称为堆，对象，指针动态数组等数据结构需要动态申请释放他们的内存空间。程序运行栈存放活动记录，并记录函数调用的上下文现场。

代码生成过程中，寄存器起着决定运行速度的重要作用，但程序的变量和数据结构的数量往往远远多于寄存器的数目，因此寄存器的合理分配对提升编译速度也很重要。寄存器分配包括全局寄存器分配和临时寄存器分配。

全局寄存器分配是指在函数内将通用寄存器中的全局寄存器分配给某些变量，通常是那些跨越基本块仍活跃的变量。全局寄存器分配的对象一般为函数的局部变量，为了线程安全，全局变量和静态变量不会被分配。

临时寄存器为一种随机资源，在程序的某些指令必须在寄存器甚至指定寄存器完成时申请使用权。

目标代码生成器的最后一步为中间代码映射。中间代码到目标代码的映射成为指令选择过程，中间代码的表达方式，目标体系结构指令集架构的复杂程度和目标代码水平要求都决定了这一过程的复杂程度。

## 8. 优化方案

相关数据结构：

1. 基本块：确定入口语句集合后每个入口语句直到下一个入口语句，或者程序结束之间的所有语句属于一个基本块
2. DAG图：为基本块内的优化的一种方法，叶节点为变量、常量等，中间节点为操作符。
3. 数据流：可以获取数据如何在程序执行路径中流动等相关信息

关键算法：

基本块划分算法、通过构建DAG图消除局部公共子表达式，从DAG导出中间代码的启发式算法，基本块到达定义数据流分析算法，

## 9. 出错处理

出错处理方案：跳转到下一个分号后面的token

部分出错信息及含义如下：

1. 标识符未定义
2. 重定义
3. 括号不匹配
4. 非法字符
5. 字符数目过多
6. 字符串含有非法字符
7. 缺少分号
8. 数组越界
9. 标识符赋值类型错误

10、单独感叹号

11、空字符

12、双引号不匹配