《编译技术》

课程设计文档

学号： 15071054

姓名： 杨伯文

2018年 11 月 18 日

## 一．需求说明

### 1．文法说明

【说明获取的文法、对文法的改写和扩充】

1.1个人获取的文法如下

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜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＞ ::= ＜数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜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＞‘]’) } //＜无符号整数＞表示数组元素的个数，其值需大于0

＜类型标识符＞ ::= int | char

＜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

＜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

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜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＞} //[+|-]只作用于第一个<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＞‘)’

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜<情况语句>|＜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＞;

＜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＞ ::= do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞ ::= ＜整数＞|＜字符＞

＜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＞‘)’|<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类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符不区分大小写字母

（3）写语句中，字符串原样输出，单个字符类型的变量或常量输出字符，其他表达式按整型输出

（4）数组的下标从0开始

（5）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

1.3文法改写

为满足递归下降子程序要求，修改获得的文法的某些非终结符的FIRST集以防止冲突，故需要对部分文法进行改写，修改后的文法如下，其中标红是需要修改的文法：

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜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＞

！！！

读三个解决问题

INT IDENT SEMI -> 变量说明

INT IDENT RPARENT 或者 INT IDENT RBRACE -> 有返回值函数定义

CHAR同理

VOID IDENT 如果 IDENT不是main -> 无返回值函数定义

如果是main ->主函数

大概率是需要回溯了

＜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＞ ::= ＜数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞

＜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＞‘]’) } //＜无符号整数＞表示数组元素的个数，其值需大于0

＜类型标识符＞ ::= int | char

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

预读即可

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

预读即可

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜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＞} //[+|-]只作用于第一个<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＞‘)’

需要判断读到的标识符是函数还是普通标识符，+-号的归属问题不影响结果，所以自定义即可

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞｜ ‘{’＜语句列＞‘}’｜<情况语句>|＜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＞;

＜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＞｜E ) //表达式为0条件为假，否则为真

因为条件的FOLLOW集为’)’，所以无回溯

＜循环语句＞ ::= do＜语句＞while ‘(’＜条件＞‘)’

＜常量＞ ::= ＜整数＞|＜字符＞

＜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＞}

需要注意，可能有问题

＜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＞‘)’| E ) //第一种选择为有参数的情况，第二种选择为无参数的情况

因为＜有返回值函数调用语句＞的FOLLOW集为；或+-\*/不会为‘(’，所以没事

＜无返回值函数调用语句＞ ::= ＜标识符＞(‘(’＜值参数表＞‘)’| E ) //第一种选择为有参数的情况，第二种选择为无参数的情况

同理，＜无返回值函数调用语句＞的FOLLOW集仅为；，所以也没事

＜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＞‘)’

可以通过预读处理

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

附加说明：

（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符不区分大小写字母

（3）写语句中，字符串原样输出，单个字符类型的变量或常量输出字符，其他表达式按整型输出

（4）数组的下标从0开始

（5）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

### 2．目标代码说明

【说明要生成的目标代码指令及含义】

### 3. 优化方案\*

【说明需要完成的优化方案及其要求】

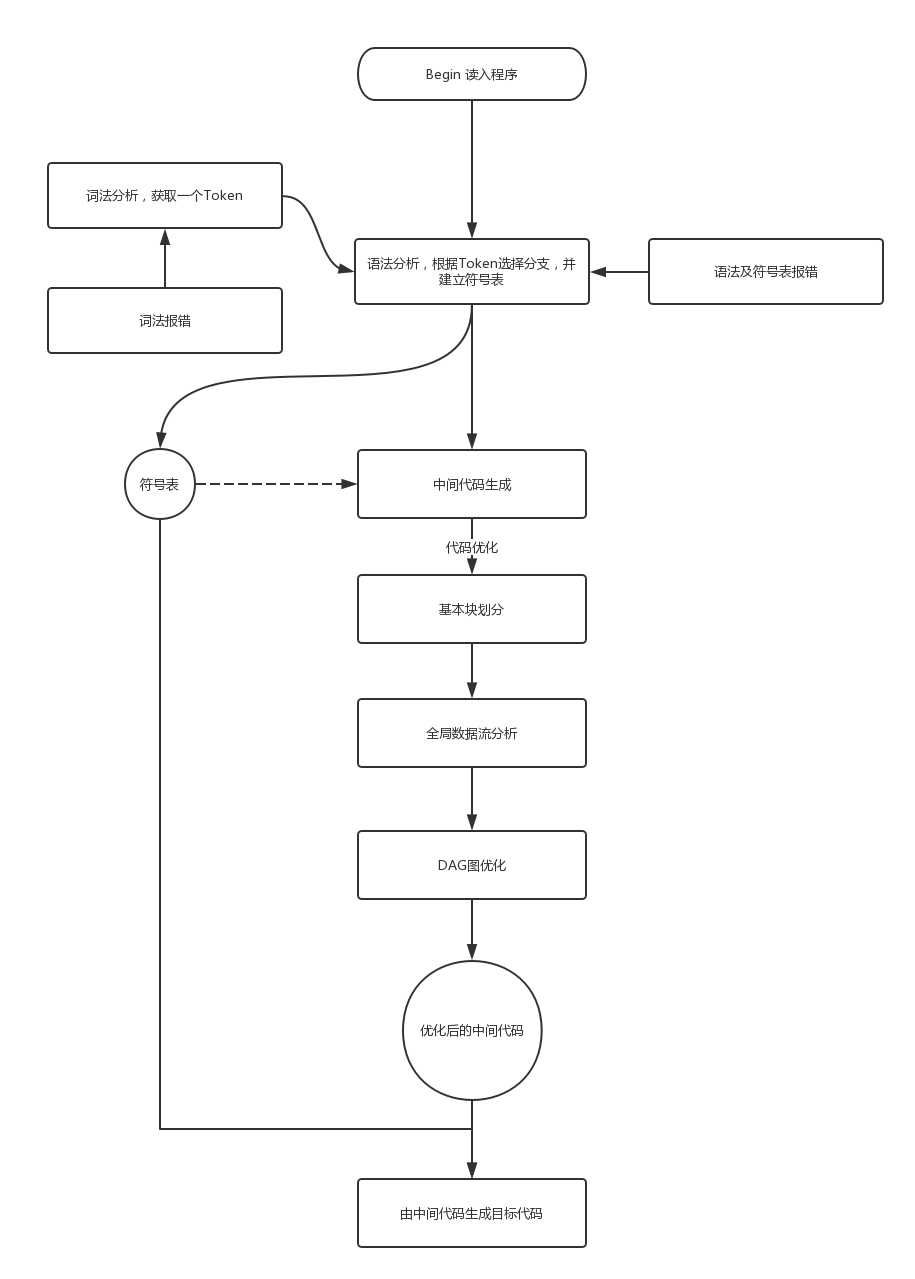
## 二．详细设计

【应包括但不限于以下内容】

### 1．程序结构

【从总体上描述程序的结构，文字或图示均可】

程序设计整体分为五部分：

1. 语法分析并进行该层次的错误处理
2. 语义分析并进行语义上的错误处理
3. 中间代码生成，生成四元式
4. 根据四元式进行代码优化
5. 目标代码生成，生成MIPS汇编

### 2．类/方法/函数功能

A.词法分析相关

①Token next\_symbol()

该函数为词法分析的核心，同样也是语法分析的基石。意为读取下一个有意义的符号，这意味着自动过滤空格等符号、自动分隔单词、并进行简单的符号级别的错误处理。

返回的是一个自定义类型，其中存储着该符号的类型和值。

union token\_value{

int number;

char letter;

std::string \*pstring;

};

enum token\_value\_type{

NUMBER,

LETTER,

PSTRING

};

typedef struct{

int line;

int pos;

token\_type t\_type;

token\_value t\_value;

token\_value\_type t\_value\_type;

}token;

以Union和Enum结合存储token 的值和类型，同时存储token的位置便于报错。

②void next\_letter()

读取下一个字符，存入current\_char中，并更新current\_pos

③void next\_line()

读取下一行，存入new\_line中，并更新current\_line

④void clear\_token()

清空current\_token中的内容

⑤对于current\_letter的一系列bool类型判断

用于判断当前的字符是什么

⑥int is\_symbol()

用于判断当前的标识符是不是保留字

B.语法分析相关

①多个递归下降子程序

Void int\_()

Void char\_()

Void string\_()

Void program() //程序

Void const\_decl()

Void const\_def()

Void unsigned\_int()

Void decl\_head()

Void var\_decl()

Void var\_def()

Void type\_ident()

Void return\_func\_def()

Void not\_return\_func\_def()

Void complex\_state()

Void para\_list()

Void main\_state()

Void expression()

Void item()

Void factor()

Void statement()

Void become\_state()

Void if\_state()

Void condition()

Void dowhile\_state()

Void const\_val()

Void switch\_state()

Void case\_table()

Void case\_state()

Void default\_state()

Void return\_func\_call()

Void non\_return\_func\_call()

Void val\_para\_list()

Void statement\_list()

Void scanf\_state()

Void printf\_state()

Void return\_state()

②符号表添加和检查

Bool add\_in\_symbol\_table(string name, int ref)

Symbol search\_in\_symbol\_table(string symbol\_name, int ref)

Bool update\_symbol\_table(string symbol\_name, int ref, int which\_value, int value)

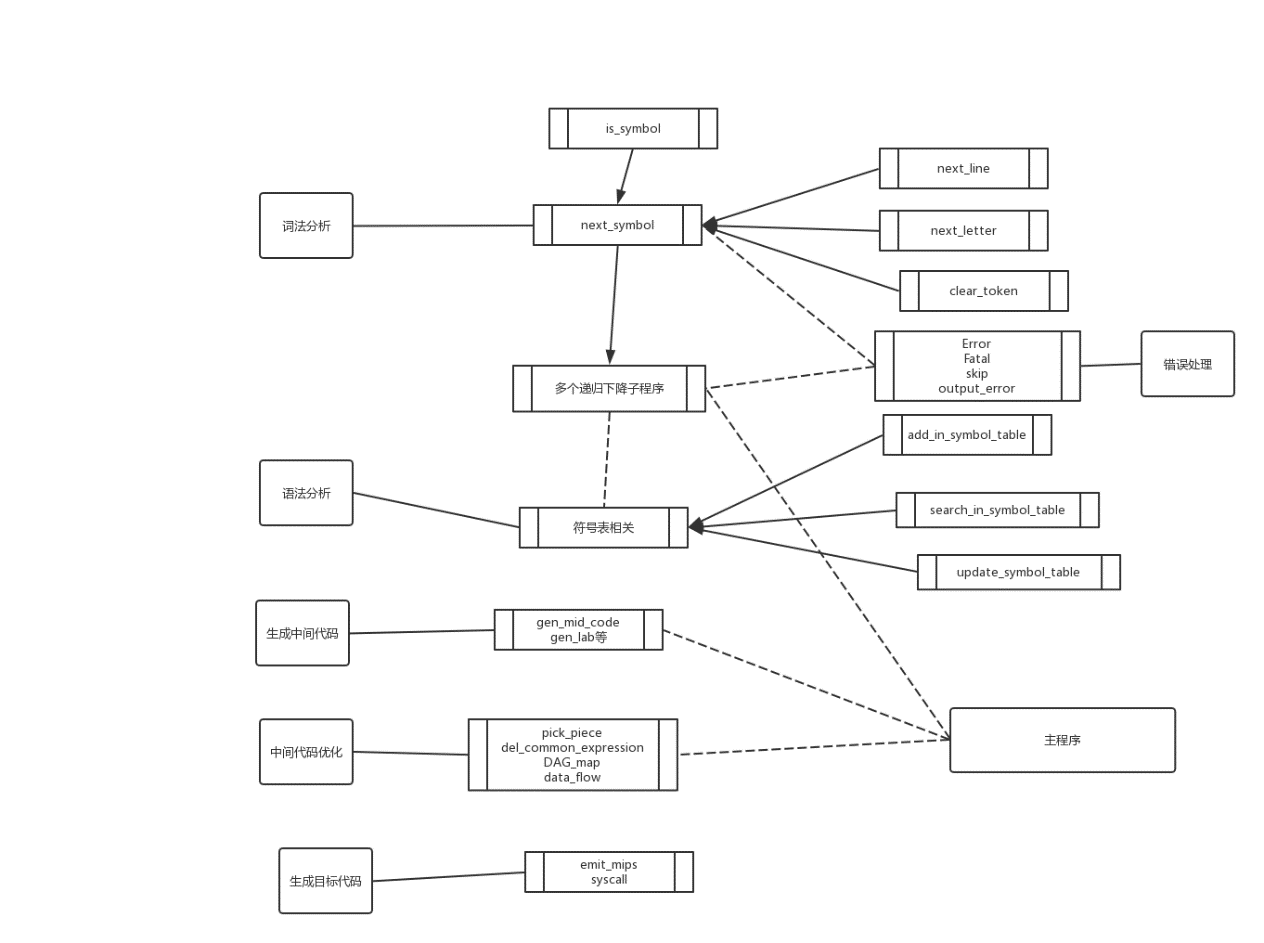
C.中间代码生成相关

D.四元式的优化

E.目标代码生成

此三部分的思路在后续的章节进行了描述，对于类和函数的说明暂时不涉及。

### 3．调用依赖关系



### 4．符号表管理方案

首先定义符号表中需要使用到的枚举类型：

（1）return\_type表示函数返回值的类型，或者是标识符的类型

|  |  |
| --- | --- |
| INTRET | 类型是int |
| CHARRET | 类型是char |
| VOIDRET | 类型是void |

（2）symbol\_type表示标识符的种类

|  |  |
| --- | --- |
| CONSTSYM | 表示是一个常量 |
| VARSYM | 表示是一个变量 |
| ARRAYSYM | 表示是一个数组变量 |
| FUNCSYM | 表示是一个函数标识符 |
| PARASYM | 表示是一个参数标识符 |

（3）单个符号的数据结构如下：

|  |  |  |
| --- | --- | --- |
| 属性名 | 类型 | 说明 |
| name | std::string \* | 记录名称 |
| R\_type | Return\_type | 表示标识符的类型，是int还是char还是数组还是字符数组或者void |
| S\_type | Symbol\_type | 示标识符的种类，表示的是变量、常量还是函数 |
| feature | int | 对于函数，记录了参数的个数；对于常数，记录了常数的值；对于数组，记录了数组的最大长度；对于变量或者参数，值为-1没有意义 |
| ref | int | 对于参数，ref记录了它在分函数表的位置 |
| Decl\_line | int | 记录了它的声明位置，便于调试 |
| para\_list | Return\_type类型的数组指针 | 仅仅对函数有效，按顺序记录了该函数的参数个数以及顺序 |
| number | int | 记录了这个标识符的编号 |
| flag | bool类型的数组 | 记录了这个变量在基本块里的活跃情况 |
| Reg\_index | int | 记录了给这个变量分配的寄存器的编号，默认为-1，表示不分配 |
| link | int | 是为了使用hash分离链表所需的属性，记录了它的下一个冲突项的位置，默认为-1，表示没有下一个冲突项 |

符号表的管理使用散列符号表的栈式存储：

使用C++自带的HashMap进行散列的计算，每次插入更新HashTable中的指向数据即可。

A.符号表中插入新符号的步骤：

①根据hash函数计算这个标识符对应的hash值

②根据计算后的hash值查hash表symbol\_hash，获得对应hash表项的原来的值，记录下来；若没找到，则为空

③初始化符号表中新加入的项，更新top指针，并将link域的值赋为②中所记录

④更新symbol\_hash对应表项的值为新加入项所在的位置。

B.退出函数时的步骤：

在退出函数时，需要把之前函数的局部符号全部删除。

①遍历每一个局部符号，每删除一个局部符号，就要更新HashTable中的值

②维护top指针

C.查询时的步骤：

查询需要分为局部查询和全局查询

①对于局部查询，当查询到的下标小于ref时就需要停止查询

②对于全局查询，则需要把整个冲突链都查询一遍

### 5．存储分配方案

①静态存储

在本次课程设计的实现中，可以在编译阶段静态的确定其地址的应该就是全局变量，全局常量和字符串。

A.全局变量和全局常量地址分配

全局变量和全局常量配合符号表中记录的相对地址进行相对寻址。其中全局常量在之后的优化中不需要分配地址的，因为常量替换优化后，常量在表达式中的运算直接被替换为相应的常量值。

B.字符串地址分配

字符串的存储和输出，使用mars自带的伪指令.asciiz来实现存储分配，在目标代码生成层次实现。

②动态存储分配方案

每个函数调用的运行栈从高到低依次为：

|  |
| --- |
| 参数区 |
| 局部变量区 |
| 寄存器栈（保存调用者的寄存器的情况） |
| 上一次的$fp寄存器 |
| 上一次的$ra寄存器 |

在做了寄存器分配后，load/store指令主要集中在数组元素的存取和函数调用中。

### 6. 解释执行程序\*

选择的C0文法为生成目标代码执行，无需解释翻译程序。

### 7. 四元式设计\*

①简单的表达式

如X = X + 1；

四元式为X = X + 1

②函数声明

如int foo( int a, int b, int c, int d)

四元式为 int foo()

para int a

para int b

para int c

para int d

③函数调用

如I = add(X, Y)

四元式为

push x

push y

call tar

i = RET

④函数返回

如return (X)

四元式为ret x

⑤变量声明

如int i,j;

四元式为var int i

Var int j

⑥常量声明

如const int c = 10

四元式为const int c = 10

⑦表达式

如x = a \* (b + c)

四元式为t1 = b + c

t2 = a \* t1

x = t2

⑧条件判断

如x == y

四元式为x == y

⑨条件跳转或者无条件跳转

BNZ LABEL1 //满足条件跳转到LABEL1

BZ LABEL1 //不满足条件跳转到LABEL1

⑩数组赋值或取值

如a[i] = b \* c[j]

四元式为t1 = c[j]

t2 = b \* t1

a[i] = t2

其余语句，如循环语句、情况语句均可由条件跳转语句实现完成。

只需额外定义SCANF和PRINTF即可。

如scanf(x,y,z)

四元式为

SCANF x

SCANF y

SCANF z

Printf(x,”123”)

四元式为

PRINTF x

PRINTF “123”

### 8. 目标代码生成方案\*

|  |  |
| --- | --- |
| OP | 映射说明 |
| Int foo() | 声明函数，生成Label至参数列表下的代码处 |
| Para int a | 将a放入$a1寄存器中，函数调用时根据参数列表数量和顺序取值 |
| Push x | 将x压入栈中，以供函数调用 |
| Call add | Jal到相应的函数开始label处 |
| Ret x | 把x放入$v0中，并生成jr $r0指令跳回 |
| Var int i | 声明变量i |
| Const int c = 10 | 声明常量c=10 |
| B + C | ADD指令 |
| B – C | SUB指令 |
| B \* C | MUL指令 |
| B / C | DIV和mflo指令 |
| B = C | MOV指令 |
| X == Y等条件判断 | SLT和NOT和OR指令 |
| BNZ Label | BENZ指令 |
| BZ Label | BEQZ指令 |
| c[j] | 通过$sp寄存器取值 |
| PRINTF | 根据表达式的类型，调用相应的系统调用 |
| SCANF | 根据类型，调用系统调用，获取用户输入 |

重点在于寄存器的使用和标记，具体的实现方式可能需要后续进一步学习才能深入体会，现在只能给出大概的四元式转汇编指令的思路。

### 9. 优化方案\*

暂定使用三种优化方法，基本块的划分、全局数据流分析、DAG图优化

①基本块的划分

基本块的定义如下：

A.基本块中的代码是连续的语句序列

B.程序的执行（控制流）只能从基本块的第一条语句进入

C.程序的执行只能从基本块的最后一条语句离开

基本块的划分算法：

（1）首先确定入口语句（每个基本块的第一条语句）的集合。

a.整个语句序列的第一条语句属于入口语句

b.任何能由条件/无条件语句跳转到的第一条语句属于入口语句

c.紧跟在跳转语句之后的第一条语句属于入口语句

（2）每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块。

所谓流图，定义如下：

A.流图是一种有向图

B.流图的结点是基本块

C.如果在程序的执行路径中，基本块B2紧随着基本块B1，那么从B1到B2存在一条有向边。所谓“紧随”，可以是B1的最后一条语句有条件或无条件的跳转到B2；也可以是B1最后的条件转移指令的判断条件失败后进入B2执行；还可以是B2的第一条语句紧跟在B1的最后一条语句之后，且B1的最后一条语句不为无条件跳转语句指令。则B1称为B2的前驱，B2称为B1的后继。

typedef struct {

int pre\_blocks[1000]; //前驱基本块的块号数组

int pre\_number; //相应的数量

int next\_later; //下一个正当延续的基本块号，默认为-1，表示没有这种后继

int next\_jmp; //下一个跳转的基本块号

}Block;

②全局数据流分析

每个基本块use和def集合的确定

一个基本块的def集合是在这个基本块中被定义（赋值）先于任何对它们的使用，use集合是被使用先于任何对它们的定义。

对于不同的四元式，不同的操作数对于def和use集合影响不同：

①运算四元式（加减乘除），操作数1和操作数2需要检查是否可以加入use集合，结果操作数需要检查是否可以加入def集合

②push和printf，需要检查被使用的操作数是否需要加入use集合

③数组给变量赋值，需要检查下标操作数是否加入use集合，被赋值变量是否加入def集合

④变量给数组赋值，需要检查下标操作数是否加入use集合，赋值变量是否加入use集合

⑤条件跳转语句，需要检查被使用的操作数是否需要加入use集合

⑥scanf输入语句，需要检查是否需要加入def集合

因此我们可以通过这个基本块的每一句四元式确定这个基本块的def集合和use集合

计算每个基本块的in/out集合

Out[s] = gen[s] U (in[s] – kill[s])

In[b] = U\_b的前驱基本块 Out[s]

In[b] = use[b] U (out[b] – def[b])

Out[b] = U\_B的后继基本块 In[s]

实现的具体数据结构

use[j] = new bool[symbol\_number];

def[j] = new bool[symbol\_number];

in[j] = new bool[symbol\_number];

out[j] = new bool[symbol\_number];

如果use[block\_index][index] = true ，则表示符号表位置为index的变量在下表为block\_index的基本块内是在use集合内的，这样用集合交并差运算就可以表示为标志位的逻辑运算。

③基本块内DAG图优化

DAG图的定义如下：

（1）图的叶结点由变量名或常量所标记。对于那些在基本块内先引用再复制的变量，可以采用变量名加下标0的方式命名其初值。

（2）图的中间节点由中间代码的操作符所标记，代表着基本块中一条或者多条中间代码。

（3）基本块中变量的最终计算结果，都对应着图书的一个结点；具有初值的变量，其初值和最终值可以分别对应不同的结点。

得到基本块的DAG图后，可以完成诸如消除局部公共子表达式等优化。

### 10. 出错处理

①词法分析错误

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| SINGLE\_QUOT\_ERROR | 没有找到匹配的右单引号 |
| DOUBLE\_QUOT\_ERROR | 没有找到匹配的右双引号 |
| UNIDENTIFIED\_CHAR | 非法字符 |
| NOT\_EQUAL\_ERROR | !单独出现 |
| NOT\_LEGAL\_STRING | 字符串中出现非法字符 |
| NULL\_CHAR\_ERROR | 出现空字符错误 |

②语法分析错误

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| NOT\_SEMI | 不是 ; |
| NOT\_IDENT | 不是一个标识符 |
| NOT\_ASSIGN | 不是 = |
| NOT\_A\_CHAR | 不是 一个<字符> |
| NOT\_IN\_OR\_CHAR | 不是 int或char保留字 |
| NOT\_RBRACKET | 不是 ] |
| NOT\_RPARENT | 不是 ) |
| NOT\_FACTOR | 不是 一个<因子>的开始单词 |
| NOT\_LBRACE | 不是 { |
| NOT\_RBRACE | 不是 } |
| NOT\_MAIN | 不是 main保留字 |
| NOT\_LPARENT | 不是 ( |
| NOT\_STATEMENT | 不是 一个<语句>的开始单词 |
| NOT\_ELSE | 不是 else |
| NOT\_CONST | 不是 常量 |
| NOT\_COLON | 不是 : |
| NOT\_CASE | 不是 case |
| NOT\_VAR\_DEF | 不是 变量定义的开始符号 |

③语义分析错误

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| DUPLICATE\_DEF | 标识符重复定义 |
| PARA\_TYPE\_DISMATCH | 参数类型不匹配 |
| PARA\_NUM\_DISMATCH | 参数数量不匹配 |
| IDENT\_NOT\_DEF | 标识符未定义 |
| NOT\_A\_INT | 不是 int类型的表达式 |
| NOT\_AN\_ARRAY | 不是 数组类型的标识符 |
| NOT\_A\_FUNC | 不是 函数名 |
| NOT\_NONE\_PARA\_FUNC | 不是 一个无参函数 |
| INVALID\_IDENT | 不是 一个被正确使用的标识符 |
| NOT\_A\_VAR | 不是 一个简单变量 |
| CASE\_SWITCH\_DISMATCH | case常量类型和switch的表达式类型不一样 |
| NONE\_RETURN | 有返回值的函数没有一句return |
| CAN\_NOT\_RETURN | void类型的函数return了一个实际值 |
| RETURN\_TYPE\_DISMATCH | return的类型不匹配 |
| INT\_TO\_CHAR\_NOT\_ALLOW | 不允许int到char的转换 |
| NO\_RETURN\_VALUE | 不是一个有返回值的函数 |
| DUPLICATE \_CASE\_VALUE | case 常量重复 |
| DIV\_ZERO | 可以检测到的除以0 |

## 三．操作说明

### 1．运行环境

【说明搭建运行环境的步骤】

### 2．操作步骤

【详细说明操作步骤】

## 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】

### 2．测试结果分析

【说明上述测试程序对语法成分的覆盖情况】

## 五．总结感想

【说明在完成课程设计中的收获、认识和感想】

注：【】内的文字为文档模板说明，完成的作业中需去掉。

标\*的章节需根据题目的难度进行取舍。