《编译技术》课程设计文 档

学号：\_\_\_15061129\_\_\_\_\_\_\_\_\_

姓名：\_\_\_解小锐\_\_\_\_\_\_\_\_\_\_\_

2018年 1 月 12 日

目录

[一.需求说明 1](#_Toc445322747)

[1.文法说明 1](#_Toc445322748)

[2.目标代码说明 1](#_Toc445322749)0

[3优化方案说明 1](#_Toc445322750)1

[二.详细设计 12](#_Toc445322751)

[1.程序整体结构分析 13](#_Toc445322752)

[2.词法分析部分 1](#_Toc445322752)4

[3.语法分析和语义分析 1](#_Toc445322752)6

[4.符号表部分 1](#_Toc445322752)7

[5.中间代码部分 22](#_Toc445322752)

[6.存储分配方案 2](#_Toc445322752)6

[7.优化部分 2](#_Toc445322752)7

[8.出错处理 4](#_Toc445322752)0

[9.目标代码生成 4](#_Toc445322752)2

[三.操作说明 4](#_Toc445322753)4

[1.运行环境 4](#_Toc445322754)4

[2.操作步骤 4](#_Toc445322755)4

[四.测试报告 4](#_Toc445322758)6

[1.测试程序以及测试结果 4](#_Toc445322755)6

[2.测试结果分析 4](#_Toc445322755)6

[五.总结感想 5](#_Toc445322759)0

## 一．需求说明

### 1．文法说明

**1.1.获取文法说明**

这次获取到的文法是C0难度的文法，具体文法详细信息如下：

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜＜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＞‘]’ )}

＜常量＞ ::= ＜整数＞|＜字符＞

＜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＞

＜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

1. 数组的下标从0开始

**1.2对文法的改写说明**

因为获取到的文法某些非终结符右部的FIRST集合有冲突，故我对上述文法进行了等价的修改，修改后的文法如下：

**1. <整数> int\_()**

**::= ［＋｜－］＜无符号整数＞｜０**

**开始单词： + ,- ,ZERO UNSIGNEDINT**

1. **<常量定义> constDef()**

**::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}**

**| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}**

**开始单词：int,char**

**3. <常量说明> constState()**

**::= const＜常量定义＞;{ const＜常量定义＞;}**

**开始单词:const**

**4. <常量> constant()**

**::= ＜整数＞|＜字符＞**

**开始单词：<字符> + - ZERO UNSIGNEDINT**

**5. <值参数表> varParaList()**

**::= ＜表达式＞{,＜表达式＞}**

**开始符号：<标识符> ( + - <字符> ZERO UNSIGNEDINT**

**6. ＜因子＞ factor()**

**::=<标识符>(<空>这里还有待仔细考虑，有两种情况|‘[’＜表达式＞‘]’|‘(’＜值参数表＞‘)’ )|**

**‘(’＜表达式＞‘)’｜**

**<常量>**

**开始单词： 标识符 ( + - <字符> ZERO UNSIGNEDINT**

**7. <项> term()**

**::= ＜因子＞{＜乘法运算符＞＜因子＞}**

**开始单词： 标识符 ( + - 字符 ZERO UNSIGNEDINT**

**8. ＜表达式＞ expression()**

**::= ［＋｜－］＜项＞ {＜加法运算符＞＜项＞}**

**开始单词：<标识符> ( + - <字符> ZERO UNSIGNEDINT**

**9. <变量定义处理> varDef()**

**::= (ε | ‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’ )};**

**开始单词：[ , ;**

**10. <声明头部> decHead()**

**::= int＜标识符＞ |char＜标识符＞**

**开始单词：int char**

**11. <参数> para()**

**::= <声明头部>{,<声明头部>}**

**开始单词：int char**

**12. <无参函数定义处理> noParaFuncDef()**

**::= ‘{’＜复合语句＞‘}’**

**开始单词：{**

**13. <有参函数定义处理> haveParaFuncDef()**

**::= ‘(’＜参数＞‘)’ <无参函数定义处理>**

**开始单词：(**

**14. <普通无返回值函数定义> notMainVoidFuncDef()**

**::= ＜标识符＞(’＜参数＞‘)’‘{’＜复合语句＞‘}’|＜标识符＞{’＜复合语句＞‘}’**

**::= <标识符><有参函数定义处理>|<无参函数定义处理>**

**开始单词：<标识符>**

**15. <主函数定义> mainDef()**

**::= main‘(’‘)’<无参函数定义处理>**

**开始单词 ：main**

**16. <可能以函数定义开始的程序> beginWithFunc()**

**::=<有返回值函数定义><可能以函数定义开始的程序>**

**|<无返回值函数定义><可能以函数定义开始的程序>**

**|<主函数>**

**::=<声明头部>(<有参函数定义处理>|<无参函数定义处理>)<可能以函数定义开始的程序>**

**|void<普通无返回值函数定义><可能以函数定义开始的程序>**

**|void<主函数定义>**

**开始单词： void int char**

**17. <可能以变量开始的程序> beginWithVar()**

**= (<声明头部>)((<有参函数定义处理>| <无参函数定义处理>)<可能以函数定义开始的程序> | (<变量定义处理><可能以变量开始的程序>))**

**|void<普通无返回值函数定义><可能以函数定义开始的程序>**

**|void <主函数定义>**

**开始单词：void char int**

**18. <程序> program()**

**::= <常量说明><可能以变量开始的程序>|<可能以变量开始的程序>**

**开始单词：const int void char**

**19. ＜条件＞ condition()**

**::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞**

**::= <表达式>(<关系运算符><表达式>|ε)**

**开始符号：<标识符> ( + - <字符> ZERO UNSIGNEDINT**

**20. ＜条件语句＞ ifStatement()**

**::= if ‘(’＜条件＞‘)’＜语句＞else＜语句＞**

**开始符号：if**

**21. ＜情况子语句＞ caseStatement()**

**::= case＜常量＞：＜语句＞**

**开始符号：case**

**22. ＜情况表＞ caseTab()**

**::= ＜情况子语句＞{＜情况子语句＞}**

**开始符号：case**

**23. ＜情况语句＞ switchStatement()**

**::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞[default : ＜语句＞] ‘}’**

**开始符号：switch**

**24. <循环语句> whileStatement()**

**::= while ‘(’＜条件＞‘)’＜语句＞**

**开始符号：while**

**25. ＜读语句＞ scanfStatement()**

**::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’**

**开始符号：scanf**

**26. ＜写语句＞ printfStatement()**

**::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’| printf ‘(’＜字符串＞ ‘)’| printf ‘(’＜表达式＞‘)’**

**开始符号：printf**

**27. ＜返回语句＞ returnStatement()**

**::= return[‘(’＜表达式＞‘)’]**

**开始符号： return**

**28. <语句> statement()**

**::= ＜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＞;**

**::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜＜读语句＞;｜＜写语句＞;｜＜空＞;|＜情况语句＞｜＜返回语句＞;｜＜有返回值函数调用语句＞; |＜无返回值函数调用语句＞;｜＜赋值语句＞;**

**::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜＜读语句＞;｜＜写语句＞;｜＜空＞;|＜情况语句＞｜＜返回语句＞;｜**

**(<标识符>(‘(’＜值参数表＞‘)’;|;|(＝＜表达式＞;|‘[’＜表达式＞‘]’=＜表达式＞);))**

**开始单词： if while { scanf printf ; switch return <标识符>**

**29. <语句列> statementList()**

**::= ｛＜语句＞｝**

**开始单词：if while { scanf printf ; switch return <标识符>**

**Follow集：}**

**30. <变量说明> varState()**

**::= <声明头部><变量定义处理>{<声明头部><变量定义处理>}**

**开始单词：int char**

**31. ＜复合语句＞ comStatement()**

**::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞**

**= <常量说明><变量说明>＜语句列＞**

**| <常量说明>＜语句列＞**

**| <变量说明>＜语句列＞**

**| ＜语句列＞**

**开始单词：const int char if while { scanf printf ; switch return <标识符> <空>**

修改后的文法已经避免了FIRST集冲突的情况，因此可以用递归下降方法对其进行语法分析，具体的程序框图见下面的程序结构说明。

1. **目标代码说明**

C0难度的文法选择mips汇编语言作为要生成的目标语言，关于mips汇编语言因为计算机组成原理的学习已经很熟悉了，所以这里不再赘述。这里仅仅给出可能需要用到的指令，具体含义不再解释：

|  |  |
| --- | --- |
| add $t0, $t1, $t2 | 寄存器加运算 |
| addi $t0, $t1, 100 | 寄存器和立即数加运算 |
| sub $t0, $t1, $t2 | 寄存器减运算 |
| subi $t0, $t1, 100 | 寄存器与立即数减运算 |
| li $t0, 100 | 加载立即数到寄存器 |
| sw $t0, 0($t1) | 寄存器值保存到内存中 |
| lw $t0, 0($t1) | 内存上的值加载到寄存器中 |
| beq, bne, bgez, bgtz, blez, bltz | 条件跳转 |
| j jal jr | 跳转 |
| mul $t0, $t1, $t2 | 乘法 |
| div $t0, $t1 mflo $t2 | 除法 |
| syscall | 系统调用 |
| la $a0, 0x20000000 | 加载地址 |

1. **优化方案**

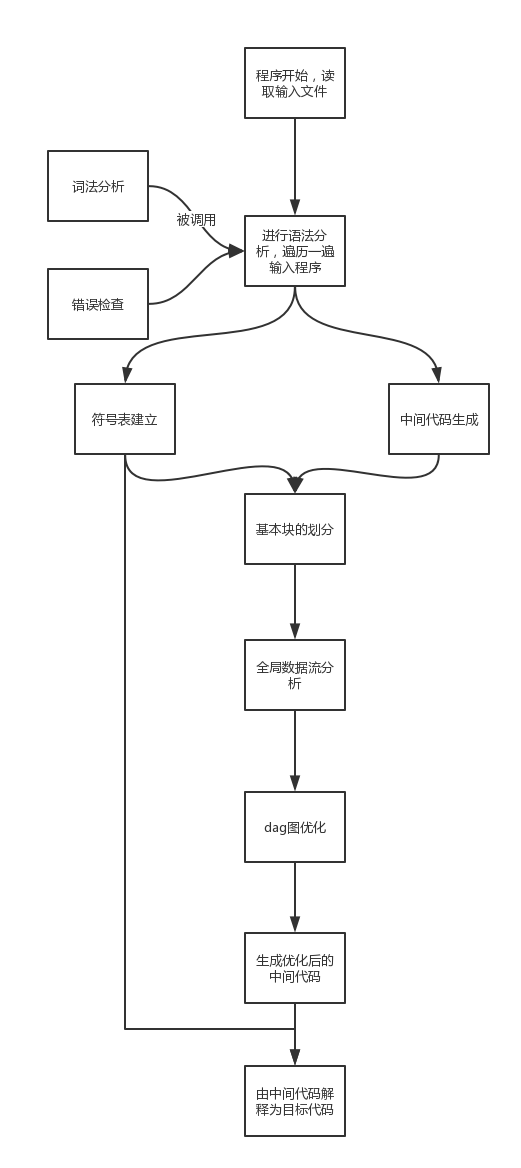
本次课程设计我实现了如下优化：

1. 简单的常数合并和常量替换
2. 参考课本上的数据流算法做数据流分析，并构建冲突图利用图着色算法以完成寄存器的分配
3. 配合数据流分析的结果，完成dag图对基本块内公共表达式的删除和无用临时变量的删除
4. 对连续的无条件跳转和连续的label进行删除

## 详细设计

1. **程序整体结构分析**

这次课程设计，我觉得从整体上看，是3个层次，2次映射。第一个层次就是高级语言，第二个层次是中间代码和符号表，第三个层次是目标语言，2次映射分别是高级语言到中间代码的映射和中间代码到目标语言的映射。所以本着这个整体思路，我的程序的整体逻辑框图如图1.1：

****

图**1.1** 程序逻辑框图

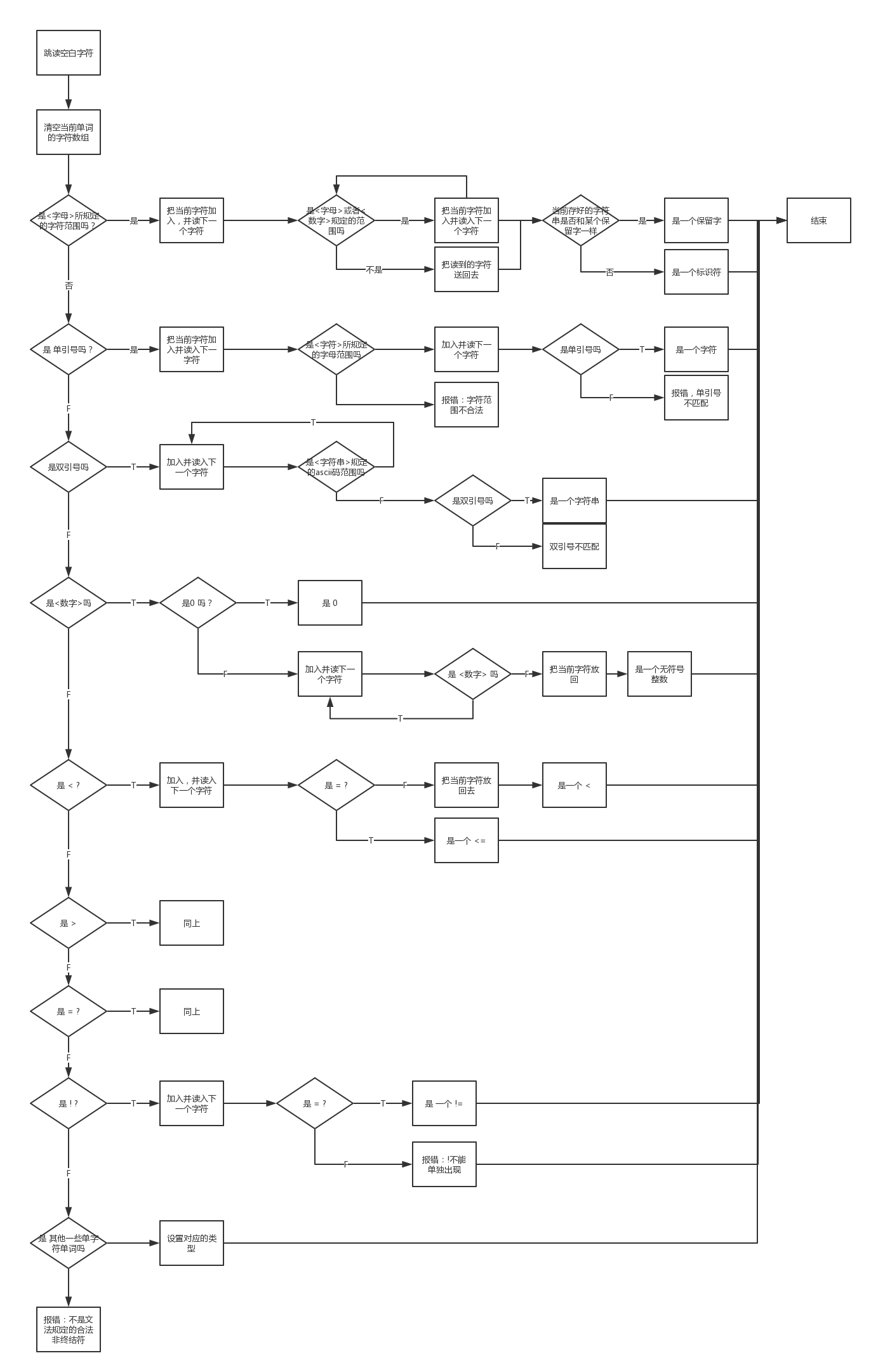
**2.词法分析部分详细说明**

**2.1 单词表**

在《编译原理》这本书上，单词被称为具有“独立语法意义”，是一个“基本语法单元”，而且单词是被后面的语法分析所使用的，所以单词就是那些需要在后面的语法分析中使用的基本单元。

|  |  |  |
| --- | --- | --- |
| 枚举名称 | 说明 | 单词的值 |
| NOTYPE | 是每一次单词分析时，单词类型的初值，如果在分析后单词类型还是这个值，那么说明分析出错 |  |
| ID | 标识符 | 就是该标识符的字符串 |
| UNSIGNEDINT | 表示一个非0的无符号整数常量 | 该常量的值 |
| ZERO | 这里我把0和非0无符号整数给分开了，因为在<整数>的分析中，0之前没有符号，而且在老师给定的文法定义中，<无符号整数>就不包括0 | 0 |
| SINGLECHAR | 表示是单个字符，是<字符>所规定的字符范围内的字符，不包括外面的单引号 | 相应的字符 |
| STRING | <字符串>的两个双引号包围起来的部分 | 相应的字符串 |
| CONST/INT/CHAR/VOID/MAIN/IF/ELSE/WHILE/SWITCH/CASE/DEFAULT/SCANF/PRINTF/RETURN | 表示保留字 | 相应的保留字字符串 |
| PLUS, MINUS, STAR, DIV, LESS, MORE, SQ, DQ,COMMA, SEMI, COLON, ASSIGN, LBRACKET, RBRACKET, LBRACE, RBRACE, LPARENT, RPARENT, | 表示单字符的分界符，比如 + - \* /等 | 相应的字符 |
| LESSEQU, MOREEQU, EQU, NEQU | 这里双字符的分界符就是<关系运算符>了 | 相应的字符串 |

**2.2词法分析程序逻辑框图**



（在图片文件夹“词法分析”中可以查看清晰的图片）

**2.3 词法分析部分关键数据结构和函数功能说明**

**2.3.1 数据结构**

typedef union {

char c;

int number;

std::string \*str;

} tokenVal;

typedef struct token{

tokenVal val; //单词对应的值

int id; //单词对应枚举变量的标识

std::string \*output; // 用于输出调试信息

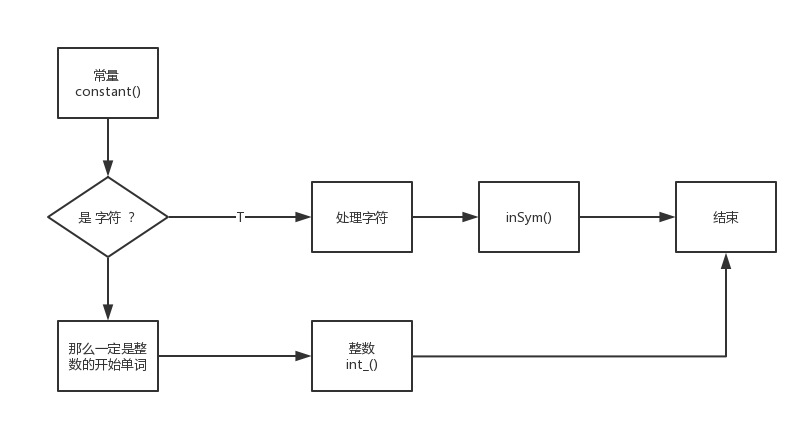
}token;

tokenVal是一个联合体，因为不同的单词有不同的单词值，所以这里使用了联合体作为**单词的值**的存储方式

**2.3.2词法分析部分的关键函数**

1. 判断字符的相关函数，包括判断是否为空白字符，是否为某个特定字符等等
2. nextChar() 方法，这个方法可以从文件流中读入一个字符并保存下来
3. retract()方法，这个方法可以把当前字符回退到文件流中，用于回溯
4. inSym()方法，调用这个方法，就可以获取下一个单词并保存下来，便于语法分析时使用。
5. **语法分析和语义分析部分详细说明**

**3.1 语法分析逻辑框图**

****

（见附录“语法流程图”文件夹，这里选择<常量>的分析框图作为示例贴出）

**3.2 语义分析关键方法说明**

语义分析需要完成的任务有：

（1）中间代码的翻译

（2）符号表的建立

（3）错误的检查

这三块在下面各个部分会分别详述，这里简单介绍一下其他的方法：

（4）临时变量的生成

临时变量的生成原理就是通过一个int类型的变量记录生成的临时变量个数，当需要生成临时变量时，使用”$t”与这个变量做拼接作为最后临时变量的标识符，然后这个变量自增1，这样做生成的临时变量不会重复而且不会与原来程序中出现的标识符冲突

1. label标签的产生

label标签的产生与临时变量的生成原理类似，也是通过一个变量自增来生成不同的label标签。

1. **符号表部分详细说明**

**4.1 符号表整体说明**

我的符号表组织分为三个部分：

1. 局部函数符号表
2. 全局符号表
3. 字符串表

其中，局部函数符号表记录了这个函数中的局部变量信息，全局符号表记录了全局的符号信息，字符串表记录了需要输出的字符串信息。

符号表的管理算法，我使用了书上介绍的hash分离链表法，关于这个算法的详细说明见下面。

**4.2 符号表表项详细说明**

首先介绍需要用到的枚举变量定义

（1）eRetType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 |

1. eSymType表示标识符的种类

|  |  |
| --- | --- |
| CONSTSYM | 表示是一个常量 |
| SIMPLESYM | 表示是一个简单变量（临时变量也归为这一类） |
| 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 |

然后介绍符号表的表项的各个属性域：

|  |  |  |
| --- | --- | --- |
| 属性名 | 类型 | 说明 |
| name | std::string \* | 记录名称 |
| returnType | eRetType | 如果是一个函数，表示该函数的返回类型，如果是其他种类，那么表示该变量的类型 |
| symbolType | eSymType | 表示该标识符的种类 |
| 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 |
| decLine | int | 记录了它的声明位置，便于调试 |
| paraList | eRetType类型的数组 | 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 |
| address | int | 记录了这个变量的相对地址 |
| flag | bool类型的数组 | 记录了这个变量在基本块里的活跃情况 |
| regIndex | 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 |

**4.3 符号表的组织管理方式说明**

**4.3.1 数据结构**

（1）对于全局符号

|  |  |
| --- | --- |
| symbol\* symTab[] | 记录了当前正在编译的局部函数符号和全局函数符号内容 |
| int top | 记录了symTab的第一个空闲位置 |
| int index | 记录了局部函数符号开始的位置 |
| int symHash[] | hash分离链表法所需要的hash表 |

（2）对于局部函数内的符号

|  |  |
| --- | --- |
| symbol\*\* funcSymTab[] | 记录了已经编译完成的局部函数的符号信息 |
| int funcSymNum[kMaxFuncNum] | 记录了每个分函数的符号的个数 |
| int funcNum | 记录有多少个分函数 |

（3）对于字符串表

|  |  |
| --- | --- |
| std::string \*stringTab[kMaxStringNum]; | 记录了需要输出的字符串 |
| int stringNum ; | 记录了有多少个这样的字符串 |

**4.3.2 符号表管理算法详细说明**

1. hash分离链表法

这个方法在书上有介绍，但是对于C0文法的实际情况我又做出了一些改变，不过整体思路还是和书上的一样。

初始化的步骤：

①初始化index，top等指针为0

②初始化symHash每一项都为-1，关于初始化为-1的原因是为了便于之后的管理，-1表示该项可用

编译到一个函数开始时：

注意维护index指针指向函数的第一个符号的位置，书上原来分程序索引是一个数组，但是因为C0文法的特点是函数不能嵌套定义，所以现在只需要一个变量值即可表示分程序的开始位置。

插入算法的步骤：

①根据hash函数计算这个标识符对应的hash值，这里的hash函数是我从网上找到的一个hash算法

②根据计算后的hash值查hash表symHash，获得对应hash表项的原来的值（用于link域的赋值），然后把这个hash表项的值更新为top指针

③初始化表项的每一个属性，尤其注意初始化相应的regIndex为-1表示尚未分配寄存器，初始化link域为之前获得的hash表项的值。

④把这个新的符号项插入symTab中，并维护top指针。

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，维护相应的hash表，top指针，还要注意的是每删除一个局部符号，就把它加入到相应分函数的符号表中

②维护top指针，index指针等等

查询时的步骤：

查询需要分为局部查询和全局查询

对于局部查询，当查询到的下标小于index时就需要停止查询

对于全局查询，则需要把整个冲突链都查询一遍

hash分离链表法在查询上速度肯定比栈式符号表的查询速度快，但是维护的成本也增加了很多。

**4.4 符号表在整个编译过程中的使用说明**

**4.4.1 符号表项的产生**

符号表项的产生主要有以下几个来源：

1. 全局，局部变量，常量的声明，参数的声明。
2. 临时变量的产生。在我的实现中每产生一个临时变量，那么就把这个临时变量作为简单变量类型，插入到相应函数的符号表中便于统一管理。
3. 函数的定义。

**4.4.2 符号表项的查找**

需要查找符号表的情况有以下几种

1. 全局变量，常量定义，局部变量，常量定义，还有参数声明，函数声明时需要保证没有重复定义
2. 在<因子>中，当用到简单变量或者常量或者函数调用时，需要查表以获取相关信息，以保证标识符已被定义
3. 在优化时，需要为每一个可以分配寄存器的变量指定其所属的寄存器，此时也需要函数表的查询
4. 字符串表在输出字符串时，需要被用到
5. 在最后目标代码生成时，还需要通过查表，获取该标识符所对应的寄存器

所以可以看出，在我的实现中符号表是从语义分析阶段一直到目标代码生成一直在被使用 。符号表是几乎在整个编译阶段都是至关重要的。

1. **中间代码详细说明**

**5.1中间代码介绍**

**5.1.1 中间代码各个操作符类型介绍**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| op | op含义 | 操作数1 | 操作数2 | 结果操作数 | 在dag图中的处理 | 解释为目标代码 |
| REALPARAOP | push | 和结果操作数一样 | 0 | 可能是标识符或者是数字 | 生成相应的中间节点 | 生成参数压栈代码 |
| RARRAYOP | =[]  a = b[1]; | 数组标识符 | 下标操作数 | 要赋值的操作数 | 参与dag图的建立 | 生成相应的数组取值并赋值的代码 |
| CALLOP | call  表示调用这个函数 |  |  | 函数标识符 | 不处理，作为基本块的分隔 | 转化为jal |
| GOTOOP | goto  表示无条件跳转 |  |  | label | 不处理，作为基本块的分隔 | 转化为j |
| FUNCBEGINOP | init  表示这个函数的开始 |  |  | 函数标识符 | 不处理，单独作为基本块 | 转化为一个函数开始时相应的代码 |
| RETOP | ret  表示这个函数的结束 |  |  | 函数标识符 | 不处理，单独作为基本块 | 转化为一个函数结束时的代码 |
| MULOP | \* | 标识符或数字 | 标识符或数字 | 标识符 | 生成中间节点 | 转化为相应的乘法代码 |
| DIVOP | / | 标识符或数字 | 标识符或数字 | 标识符 | 生成中间节点 | 转化为相应的除法代码 |
| ADDOP | + | 标识符或数字 | 标识符或数字 | 标识符 | 生成中间节点 | 转化为相应的加法代码 |
| SUBOP | - | 标识符或数字 | 标识符或数字 | 标识符 | 生成中间节点 | 转化为相应的减法代码 |
| EQUALOP, NEQOP, MOREOP, MOREOREQUOP, LESSOP, LESSOREQUOP, | ==  !=  >  >=  <  <=  分别表示相应的条件跳转语句 | 标识符或数字 | 标识符或数字 | label | 不处理 | 转化为条件跳转语句 |
| SCANFOP | scanf  输入 | 结果操作数 | int/char | 标识符 | 生成中间节点 | 转化为相应的输入代码 |
| PRINTFOP | printf  （输出表达式） | 结果操作数 | int/char | 标识符 | 生成中间节点 | 转化为相应的输出代码 |
| PRINTFOP | printf  （输出字符串） | 结果操作数 | 0 | 字符串在字符串表中的位置 | 生成中间节点 | 转化为相应的输出代码 |
| LARRAYOP | []=  a[1] = b; | 给数组值的操作数 | 下标操作数 | 数组标识符 | 生成中间节点 | 转化为相应的给数组赋值代码 |
| LABOP | label:  表示标签 | 0 | 0 | label | 不参与dag图 | 生成相应标签 |
| EXITOP | 表示程序结束 |  |  |  | 不参与dag图 | 生成程序结束代码 |

说明：

1. 为了更好的在dag图中做优化，我统一把一些类型的无意义的操作数用0来表示

**5.1.2 中间代码操作数类型介绍**

对于运算类型的中间代码，操作数的类型只有可能是以下三种：

1. **#RET** 专门表示mips架构中的$v0寄存器，即表示函数的返回值
2. 标识符，这里标识符包括①全局变量②局部变量③临时变量，它们都有自己对应的存储空间
3. 数字

了解了可能有的类型，对于代码优化，目标代码生成，尤其是目标指令的最优选取很重要。

**5.2 高级语言到中间代码转换过程说明**

因为这次语法分析采用递归下降的分析方法，所以从高级语言到中间代码的转换也是采用语法制导的翻译来完成的，

这个过程太过繁琐，所以这里我只简要介绍下面几个转换过程

1. <表达式>的翻译。

因为在<表达式>的翻译过程中，我还做了一些简单的常数合并。

在<因子>的翻译中，当<因子>对应一个常量或者对应一个常量标识符时，我会直接把这个常量的值作为这个<因子>返回的操作数。

在<项>的翻译中，当<因子>返回连续的常量时，我会直接把常量计算的结果作为<项>的结果操作数，直到某个<因子>返回了一个标识符。那么我会把这个标识符直接作为结果操作数

在<表达式>的翻译中，与<项>同理，当<项>返回的操作数是连续的常量时，我会直接计算结果并作为<表达式>的操作数，直到中间<项>返回一个标识符，那么我就把这个标识符作为结果操作数。

所以如果表达式从左边开始有连续的常数运算，那么我会直接把这些常数运算的结果作为最终结果生成中间代码。但是这样做不足就是对于在中间的连续的常数运算和最右边的连续的常数运算，没有实现相应的常数合并。

1. printf语句的中间代码生成

printf语句的易错点在于：printf也是一个函数调用，所以只有在所有的参数都准备好以后才能调用printf输出，也就是说如果printf的表达式中有函数调用，那么必须要先把这个函数调用执行完毕后，才能printf最后的结果。

所以printf的转换需要先生成内部表达式的四元式，然后才能生成外层的printf的四元式.

**5.3 中间代码部分数据结构说明**

typedef struct {

midop op;

std::string \*op1name;

std::string \*op2name;

std::string \*rstname;

}midcode;

//四元式数组

midcode\* codes[kMaxMidCode];

//四元式数组下标

int midindex;

1. **存储分配方案**

**6.1 静态存储分配方案**

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。

**6.1.1 全局变量和全局常量地址分配**

全局变量和全局常量我统一按照mars的$gp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其实不需要分配地址的，因为常量替换的优化做了之后，常量在表达式中的运算直接被替换为相应的常量值。

不过为了防止字符串地址空间与全局变量地址空间冲突，我在一开始给$gp加上了0x10000大小，这样除非有很多的字符串的极端情况，也不会有地址空间上的冲突了。

**6.1.2 字符串地址分配**

字符串的存储和输出，我直接使用了mars自带的伪指令.asciiz来实现存储分配，而且经过测试后发现.asciiz占用的地址是从0x10010000开始。

上面两种情况都是在编译时即可确定地址的分配。

**6.2 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） |
| 上一次的帧指针，即$fp寄存器 |
| 上一次的$ra寄存器 |

所以其实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。

1. **优化方案详细说明**

这一次我主要做了以下优化

①删去无用label和无条件跳转语句，删除对同一变量的无用赋值语句，比如a=a这种情况

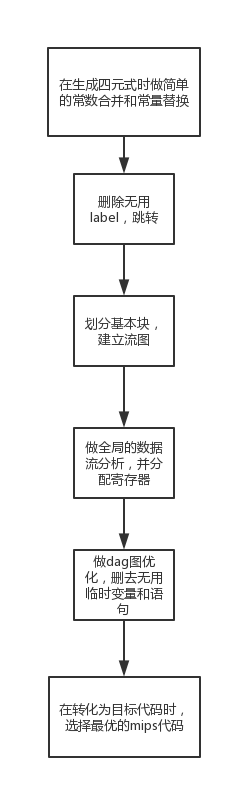
②划分基本块，并建立流图，进行数据流分析，并根据分析结果建立冲突图，并根据图着色算法进行寄存器分配

③运用**改编后的dag图算法**删除基本块内的公共子表达式，并配合数据流分析结果**删除无用的临时变量或者局部变量**

④简单的常数合并和常量替换

⑤目标代码的最优选择

上述优化执行的步骤是：



下面按照顺序对这些优化做详细说明。

**7.1 常量合并和常量替换**

**7.1.1 常量合并**

关于常量合并之前在讲<表达式>的四元式代码生成过程时已经说明了，这里不再赘述。而且这里我只实现了表达式左边有连续的常量才会合并的机制，比如 a = 1 + 2 + 3，对于表达式中间，最右边有连续常量（例如 a = b + 1 + 2 + 3）并没有实现合并优化，这是一个缺陷

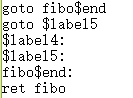
**7.1.2 常量替换**

在翻译<因子>时，如果发现<因子>是一个常量标识符，那么直接把这个常量的值作为操作数返回而不是这个操作数。常量标识符唯一可以被用到的地方就是在<因子>中，所以如果这里常量标识符等价为它的常量值，那么其实完全不必存储常量，这也是优化之一。

**7.2 删除无用label，跳转和赋值**

**7.2.1 连续label和连续无条件跳转的代码删除**

在中间代码到目标代码的映射过程中，经常会遇到几个label连续出现，或者几个无条件跳转同时出现，比如：



对于连续的goto，第一句之后的goto语句是不会起作用的，可以删除

对于连续的label，都是等价的，可以选择第一个label作为最后的label，用到其他label的情况都转化为第一个label

下面是优化后的结果：

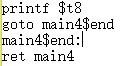


多余的goto和label都被删去了

**7.2.2 无条件跳转要跳转的label和紧挨其后的label是一样的**

在经过上面的多余goto的删除和多余label的转换后，我们还可以发现在一些地方完全不需要goto跳转，就是当goto要跳转的label地址正好是顺序执行不跳转所要到达的label地址，即直接顺序执行即可，不需要无条件跳转。

如：



优化后：



**7.2.3 无用运算语句的删除**

在我们写程序或者生成四元式时，可能会遇到这样的情况：



这样的代码无用是因为结果操作数完全没有变化，那么这些代码是可以删去的。不仅包括加上0，还有减0，乘1，除以1等，这些代码完全是可以删除而且没有一点影响的。

**7.3 划分基本块并建立流图**

**7.3.1基本块开始语句的确定**

在书上，基本块的开始语句是这样确定的：

1. 整个语句序列的第一条语句属于入口语句
2. 任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句
3. 紧跟在跳转语句之后的第一条语句属于入口语句

但是我们不能照搬书上的模式！我们需要**结合我们文法语言的实际情况，以及我们做基本块划分的理由来考虑，**首先我们的语言是C0文法，然后我们做基本块划分主要是为了数据流分析。而且我们的dag图的构建也是在一个个基本块里的。所以基本块的划分很重要！

在确定基本块的开始语句的过程中，最大的难题应该是和dag图的配合分析了。关于这一部分的详细内容，会在dag图优化中详细说明，这里仅仅简略说一下，一开始做dag图优化，发现一些四元式无法参与dag图优化，例如：scanf 语句，printf语句等等，因为怕dag图优化前后的顺序不一致。所以不得已一开始会在这些语句处划分基本块，但是这样划分后的基本块支离破碎，而且因为基本块过小，优化也不好做。

不过在与同学的讨论与自己的思考过程中，逐渐的，dag图优化可以支持的四元式类型越来越多（具体会在dag图优化详述），所以现在基本块的划分语句仅仅只有下面几种：

①label标签

②call 函数调用，函数开始 和 函数返回

③条件跳转和无条件跳转

④程序结束exit

可以看到，除了label标签语句，其他的都是跳转语句，符合课本上基本块的特性。

而且值得注意的是，我们这里一个数据流分析的程序单元应该是一个**函数块**，这是根据C0文法的特点决定的。而且为了和课本上的流图的**“入口块”和“出口块”**一致，这里我把标识一个函数开始的四元式语句作为“入口块”，把一个函数的返回语句作为“出口块”，这样做便于之后的函数内的数据流分析。这也是把函数开始语句和函数返回语句作为一个基本块的开始语句的原因。

**7.3.2 流图的建立**

**（1）确定前驱和后继基本块的逻辑：**

在划分好基本块后，就可以根据基本块最后的跳转语句确定基本块的前驱后继基本块，并在此基础上建立流图。

基本块最后的跳转语句可能是

1. 条件跳转
2. 无条件跳转
3. 函数调用
4. 函数返回
5. exit程序退出语句

对于（1）（2）只需要确定跳转后的label对应的基本块号，就可以设置相应的前驱和后继了。对于（5）因为执行后程序退出了，所以不设置其相应的后继。

根据之前的分析，数据流分析是对于一个函数来讲的，所以函数内基本块的前驱和后继基本块也必须是在这个函数内部的基本块，所以这里对于：

1. 函数调用语句
2. 函数返回语句

这两个跳转语句不设置其相应后继基本块，因为这两个跳转会跳出这个函数。

综上，我们就可以确定基本块的前驱和后继了。

1. **具体的数据结构和实现方法**

一个基本块连接信息的数据结构：

typedef struct {

int prearray[400]; //前驱基本块的块号数组

int prenum; //相应的数量

//后继块最多有两个！

//next1固定为正常执行的后继基本块，next2固定为跳转执行的后继基本块

int next1; //默认为-1，表示没有这种后继

int next2;

}Block;

思考后发现在给定C0文法的限制下，一个基本块的后继只有可能有两种①正常执行的下一个基本块②跳转后的下一个基本块。

//记录每个基本块的信息

Block\* blockArray[kMaxBasicBlock];

//记录每个基本块的开始的四元式的index

int blockBegin[kMaxBasicBlock];

//记录基本块的数量

int blockIndex;

一个程序的基本块的划分信息就用上面这三个数据结构存储。

**7.4 全局数据流分析并构建冲突图，分配寄存器**

**7.4.1 全局数据流分析**

1. **每个基本块use和def集合的确定**

根据书上的定义：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，它的操作数对于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

通过上面def和use集合的定义我们就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

1. **计算每个基本块的in/out集合**

计算的公式就是书上的公式了，这里严格按照书上的算法计算即可。

1. **实现的具体数据结构**

use[j] = new bool[symnum];

def[j] = new bool[symnum];

in[j] = new bool[symnum];

out[j] = new bool[symnum];

抓住数据流分析仅仅是对于函数块内部局部变量和临时变量进行分析这个特点，可以在当前函数块的符号表的基础上进行分析，利用书上的**位运算**表示集合的交并差的思想，这里我为函数符号表中每一个符号在每一个基本块指定一个use/def/in/out的标志位，如果use[blockindex][index] = true ，则表示符号表位置为index的变量在下表为blockindex的基本块内是在use集合内的。

这样集合交并差运算就变成了标志位的逻辑运算。

**7.4.2 冲突图的构建和图着色算法对寄存器的分配**

**（1）活跃的定义和冲突的定义**

通过数据流分析我们就可以知道一个函数的各个基本块内，某个变量的活跃情况，这里我自己定义**一个变量在一个基本块内活跃**当且仅当**这个变量在这个基本块的def集合或者是in集合中。**

在def集合中表示这个变量在这个基本块内被重新定义，在in集合说明这个变量要么在这个基本块内被使用，要么说明这个基本块在之后要被使用。

定义好了活跃的含义，接下来定义**两个变量冲突**当且仅当**这两个变量在一个基本块内活跃**。在书上我们可以看到这个定义其实不太准确，但是准确性是可以保证的，而且我只做了数据流分析，所以这里采取这个保守的定义。

1. **寄存器的分配**

冲突图完全可以用之前的def/in集合来构建，在冲突图构建完成后，就可以进行寄存器的分配了。

在书上讲到“只有跨越基本块仍活跃的变量才能分配到全局寄存器”，即它假设仅在一个基本块内被定义和使用的变量不会被分配到全局寄存器。这里我并没有采用这个原则，我自己的理由是**mips架构提供通用寄存器的数量很多**，t和s加起来一共有18个通用寄存器可用。所以我们完全没必要在寄存器的分配上过于吝啬。所以这里我把函数内部**所有的简单变量和临时变量以及参数都分配了相应的寄存器**，并不管它是否跨越基本块活跃！

由冲突图分配寄存器的算法很简单，就是按照书上所讲的图着色算法进行分配。如果这个变量不能被分配到寄存器，那么就直接从相应的内存读写它。

寄存器分配的效果非常好，因为我自己设计时最大可用寄存器是**14**个，即$t0~$t6和$s0~$s6（其他通用寄存器留作机动使用），因为可用寄存器太多，所以很难碰到分配不到寄存器的情况。而且在寄存器分配后，load/store就仅仅被限制在以下用途中：

①数组元素的存取

②全局变量的存取

③函数调用和返回寄存器上下文的切换

④未被分配寄存器的变量的读写

访存指令大大减少，性能提高了很多！

**7.5 dag图优化算法详细说明**

**7.5.1 书上的dag图算法的扩展**

越来越觉得实践的重要性了，很多东西，书上讲的和实际实现的还是有很大的差距的，在dag图算法这里体现的很明显。主要原因可能就是书上面对的目标体系结构可能和我们实际的体系结构特点不一样，这个之后详述，下面先讲一讲dag图算法的逐步的扩展过程

扩展一. 每个dag图中间节点增加一个**mainname属性**

如果完全按照书上的算法，那么只能够有加减乘除等简单运算参与dag图构建，push，printf，scanf等不好参与dag图的构建，而且书上的dag图算法在实际情况下有很多问题，一个最大的问题就是：如果某个变量从一个dag图中间节点转移到另一个dag图中间节点，那么可能导致原来那个中间节点没有对应的变量了，这样在恢复时完全不知道这个节点的结果操作数该写哪个。

同样可能出现一个中间节点对应了很多变量，那么在恢复时该使用哪个临时变量？

为了解决这样的问题，我们让每个中间节点增加一个属性**mainname**，存储这个中间节点对应的变量名，在dag图恢复时就是用这个名字作为结果操作数，关于**mainname**的维护，下面具体讲。

这个扩展就可以保证dag图在构建完成恢复后，不会出现一个中间节点在节点表中一个对应的变量都没有。

同时对于节点表中index相同的变量，其含义就是这些变量在这个基本块的代码执行完后值是一样的！我觉得这个理解很重要。

扩展二. 从dag图重新导出中间代码算法的改变

书上的从dag图导出中间代码的启发式算法，一直强调先把“最左子节点”加入队列，书上也说了这样的原因：“当某个中间节点的左操作数计算完成后，能够立即继续使用该计算结果进行下一步运算，这样可以减少对临时寄存器的分配压力”。所以dag图采用这个启发式顺序主要是因为书上所面对的目标体系结构寄存器太少了，但是对于mips来说，寄存器数量比较多，所以不需要遵循这样的启发式顺序。

这样的启发式顺序也是有**坏处**的，在于按照这个算法恢复后的中间代码的执行顺序可能与原来的不一样，而顺序的不可保证直接导致了printf，scanf等语句不能轻易参与dag图创建，因为顺序不敢保证。

所以我在dag图实现时，放弃了这样的启发式顺序，**直接按照创建的顺序作为中间代码的恢复顺序。**这样就可以把printf，scanf，push以及数组取值和赋值都加入进来参与dag图的优化了。

扩展三. dag图对printf，scanf，push，给数组元素赋值等操作的兼容

之前提到书上的dag图算法只能让简单的运算参与dag图的创建和中间代码的恢复。而如果这样的话，基本块只能在这些四元式处断开，这样基本块四元式数量少，优化的机会也很少。

经过对dag图构建算法的改写，可以让这些操作参与dag图的创建。下面对每一种操作分别讨论如何兼容。

1. 对scanf的兼容。对scanf的兼容其实很简单，只需要把scanf这个四元式的操作数1设置为要被输入的变量（设置为0也可以，也没有太大的实际意义），操作数2设置为0（无实际意义），结果操作数也设置为要被输入的变量。因为scanf操作是对要被输入变量的**重新定义**，或者说是重新赋值，和赋值操作其实很像。
2. 对printf和push操作的兼容。push一个表达式和printf 一个表达式其实意义是一样的，是对这个变量的**使用**，一开始我是想把push/printf作为这个中间节点的一个特殊属性的，但是我发现这样**保证不了恢复顺序。**因为根据之前所说的恢复规则“按照中间节点的创建顺序恢复”，所以要想保证顺序的正确性，首先必须要保证push/printf是**作为一个中间节点出现的**。其次，push和printf与运算操作的不同之处在于，对于push和printf，即使两句完全相同的printf和push，也**不能合并为一句**。而对于两句完全相同的运算，比如a = b + c ，执行两次的结果是一样，不如就叫运算语句的这种连续执行效果相同的性质叫做**幂等性，**printf和push操作是没有幂等性的，**有一句就必须要有一句的效果**。所以为了解决这个问题，经过与同学的讨论，在遇到printf和push 时，不进行中间节点的查找，直接生成新的中间节点。
3. 对给数组元素赋值操作的兼容。我的“给数组元素赋值”四元式的操作数1是赋值变量，操作数2是数组下标，结果操作数是数组名称。给数组赋值操作需要注意的是，只有当操作数1，操作数2，以及结果操作数都相同时，才不需要创建新的中间节点。

综上所述，下面给出改编后的dag图算法。

**7.5.2 dag图算法详细描述**

**（1）dag图构建算法**

输入：基本块内的中间代码序列

输出：dag图

从第一条中间代码开始，按照以下规则构建dag图：

①对于z = x op y，如果op是减/乘/除/scanf/数组元素给其他元素赋值，那么具体创建逻辑和书上的一样

②如果op是加，那么当以下条件同时满足时①第二个操作数是0（即这是一个赋值语句）②x操作数已经有对应的中间节点③z操作数对应的结点尚未被创建，当这三个条件同时满足时，我们直接在节点表中新增(z, x对应的index)这一项，表示z操作数对应的中间节点就是x对应的中间节点。并continue分析下一句

③如果op是给数组元素赋值，那么改动的地方是：在寻找已有的中间节点时，只有在两个子节点相同以及中间节点名字也相同时，才不需要创建新的中间节点，否则需要创建新的中间节点，并维护相应的节点表信息

④如果op是push或者是printf，那么放弃寻找已有的中间节点，直接创建新的中间节点

1. **中间节点的mainname属性的维护方法**

①当mainname的标志位表示空缺或者当新中间代码的结果操作数在这个基本块之后仍然活跃时（通过之前的数据流分析判断），把中间节点的mainname换成这个新的中间代码的结果操作数名字

②在需要修改节点表中一个操作数的index时，如果发现这个操作数的index对应的中间节点的mainname就是这个操作数，那么就需要设置标志位表示该中间节点的mainname空缺

**（3）dag图恢复为中间代码算法**

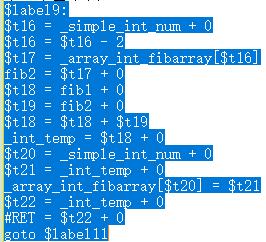
①按照dag图节点的创建顺序遍历每一个中间节点

②如果这个中间节点是printf或者是push，那么在生成新的中间代码时，结果操作数使用左子节点的mainname属性，否则生成新中间代码时，操作数1就用左子节点mainname，操作数2就用右子节点的mainname，结果操作数就用当前节点的mainname

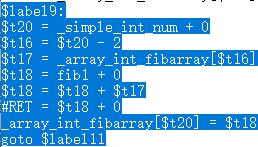
③还需要为和这个节点相同值的变量视情况生成相应的补偿语句，此时找到所有和当前中间节点的index相同的结点表中的变量名，如果该节点是“#RET”，那么需要生成相应的补偿语句；如果该节点是一个临时变量或者标识符，那么首先判断是不是一个全局变量，如果是，生成相应的补偿语句，如果不是全局变量，那么根据之前的活跃变量分析的结果，如果还活跃，那么生成相应的补偿语句，如果不活跃，那么还需要检查该基本块的最后一句跳转是否用到了这个要被删去的变量，如果用到了，那么需要替换为当前节点的mainname，如果没有用到，那么这个不生产这个临时变量或者局部变量的补偿语句，这个临时变量或者局部变量被删除。

**7.5.3 dag图优化算法心得总结**

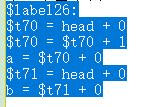
可以看到之前dag图的改编算法是很复杂的，但是也是值得做的！有大量的临时变量在dag图优化后被删去，大量的重复子表达式被删去。如果有时间，我一定要从逻辑上证明这个算法的完备性，但是现在没有时间，只能通过大量的测试程序验证正确性，目前这个算法的表现很不错：



（优化前的代码）



（优化后的代码）



（优化前的代码）

dag优化后2

（优化后的代码）

在这个过程中也学到了很多，更明白一个道理，就是不要怕困难，办法总比困难多！

**7.6 最优的目标代码选择**

书上也提到了，目标代码的选择也是可以优化的部分，因为mips是精简指令集架构，所以可选的指令很少，最优的选择也很容易得到。例如加法，如果两个操作数都是数字，那么先计算最后的结果，然后使用li指令。

这里因为简单就不再赘述。

1. **出错处理**

**8.1 错误类型介绍**

**8.1.1词法分析错误**

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| SQMISMATCH | 没有找到匹配的右单引号 |
| DQMISMATCH | 没有找到匹配的右双引号 |
| INVALIDCHAR | 不是给定文法范围内的字符 |
| ONLYEM | !单独出现了 |
| INVALIDVT | 出现了不在给定文法范围内的非终结符 |

**8.1.2 语法分析错误**

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| NOTSEMI | 不是 ; |
| NOTID | 不是一个标识符 |
| NOTASSIGN | 不是 = |
| NOTSINGLECHAR | 不是 一个<字符> |
| NOTINTORCHAR | 不是 int或char保留字 |
| NOTRBRACKET | 不是 ] |
| NOTRPARENT | 不是 ) |
| NOTFACTOR | 不是 一个<因子>的开始单词 |
| NOTLBRACE | 不是 { |
| NOTRBRACE | 不是 } |
| NOTMAIN | 不是 main保留字 |
| NOTLPARENT | 不是 ( |
| NOTUNSIGNEDINT | 不是一个非0无符号类型 |
| NOTSTATEMENT | 不是 一个<语句>的开始单词 |
| NOTELSE | 不是 else |
| NOTCONSTANT | 不是 常量 |
| NOTCOLON | 不是 : |
| NOTCASE | 不是 case |
| NOTVARDEF | 不是 变量定义的开始符号 |

**8.1.3 语义分析错误**

|  |  |
| --- | --- |
| 错误类型 | 报错信息 |
| DUPLICATEDEF | 标识符重复定义 |
| PARATYPEDISMATCH | 参数类型不匹配 |
| PARANUMDISMATCH | 参数数量不匹配 |
| IDNOTDEF | 标识符未定义 |
| NOTAINT | 不是 int类型的表达式 |
| NOTANARRAY | 不是 数组类型的标识符 |
| NOTAFUNC | 不是 函数名 |
| NOTANOPARAFUNC | 不是 一个无参函数 |
| INVALIDID | 不是 一个被正确使用的标识符 |
| NOTASIMPLE | 不是 一个简单变量 |
| CASESWITCHDISMATCH | case常量类型和switch的表达式类型不一样 |
| NONERETURN | 有返回值的函数没有一句return |
| CANNOTRETURN | void类型的函数return了一个实际值 |
| RETURNTYPEDISMATCH | return的类型不匹配 |
| INTTOCHARNOTALLOW | 不允许int到char的转换 |
| NORETURNVALUE | 不是 一个有返回值的函数 |
| CASEVALUEDUPLICATE | case 常量重复 |
| DIVCASE | 可以检测到的除以0 |

1. **目标代码生成**

**9.1 中间代码到目标代码的映射详细说明**

|  |  |
| --- | --- |
| op | 映射说明 |
| REALPARAOP | 保存相应的参数到$sp上，并让$sp自减4 |
| RARRAYOP | 生成相应的代码获得数组元素的地址，然后load到相应的变量对应的寄存器上 |
| CALLOP | jal到相应的函数开始label处 |
| GOTOOP | j label |
| FUNCBEGINOP | 需要做①生成label表示开始②保存寄存器上下文③维护栈指针和帧指针 |
| RETOP | 需要做①load出调用前的寄存器情况②维护栈帧指针③生成jr指令跳回 |
| MULOP | 使用mul指令计算最后结果 |
| DIVOP | 使用div和mflo指令计算最后结果 |
| ADDOP | ①如果是两个数字，那么就计算出最后结果，并用li指令保存到相应的寄存器上②一个数字，一个是标识符，那么使用addi指令③两个标识符，那么使用add指令 |
| SUBOP | 类似加法的处理 |
| EQUALOP, NEQOP, MOREOP, MOREOREQUOP, LESSOP, LESSOREQUOP, | 根据两个操作数的类型，生成代码获取两个操作数的差的结果，然后根据这个结果生成相应的条件跳转语句 |
| SCANFOP | 根据类型，获取用户的输入 |
| PRINTFOP | 根据表达式类型，调用相应的系统调用 |
| PRINTFOP | 根据字符串的下标，调用相应的系统调用输出字符串 |
| LARRAYOP | 获取相应的地址，使用sw把相应的值存到相应的地址上 |
| LABOP | label:  表示标签 |
| EXITOP | 转化为结束程序的系统调用语句 |

**9.2 中间代码到目标代码的关键方法说明**

在中间代码到目标代码的映射中，最关键的其实就是每个变量被正确的“**读写**”，每个变量都有其对应的存储空间，要么是地址空间，要么是寄存器（之前全局寄存器分配算法分配的），但是mips架构的特点是：**运算只能在寄存器之间**，所以即使这个变量对应的存储空间是内存，那么这个变量的使用还是需要先load到寄存器里，写还是需要先把结果放在寄存器里，然后再写到内存里。

**（1）操作数被“读”时，如何转化为它所对应的寄存器**

所以我专门写了一个方法，用来获取操作数相应的寄存器，分为下面几种情况：

①这个操作数是数字，那么就先把这个数字load到临时寄存器里

②这个操作数是#RET，那么就直接返回$v0

③这个操作数是变量，那么此时需要判断：

1. 如果是一个全局变量，对于全局变量，需要保证它的值是最新的，所以直接把这个全局变量对应地址上的值load到临时寄存器中；
2. 如果是一个局部变量，那么需要查看它是否是一个参数，如果是参数，还需要看它被分配的寄存器是不是就是表示这个参数，如果是，那么直接返回这个寄存器，如果不是，那么还需要使用load语句先把这个参数load到相应的寄存器中。如果不是参数，那么就看它有没有被分配寄存器，如果有，那么返回这个寄存器，如果没有，那么需要把这个变量对应的地址上的值先load到临时寄存器中
3. **操作数被“写”时，如何转化为相应的寄存器进行运算**

①这个操作数是#RET,那么直接转化为$v0;

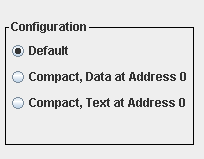
②这个操作数是变量，那么还需要判断

1. 如果是一个全局变量，那么就直接使用临时寄存器保存结果
2. 如果不是全局变量，那么就看这个变量是否有指定的寄存器，如果有，那么使用这个寄存器保存最后的结果，如果没有，那么就用临时寄存器保存最后结果

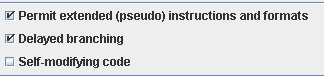
**（3）运算或者赋值后的结果需要“写回”**

运算或者赋值后，结果会保存在寄存器里，对于指定存储空间是寄存器的变量，赋值结果不需要写回，但是对于对应存储空间是内存空间的变量（比如全局变量或者未能分配寄存器的变量），就需要把寄存器中的值写到它对应的内存上去，这就是写回。

1. **操作说明**
2. **运行环境**
3. C++工程文件最好使用VS2010打开，也可以用其他的编辑器打开代码文件
4. mips代码运行的虚拟环境是Mars 4.5 ，①支持延迟槽②内存使用default模式



（内存选择default模式）



（可以支持延迟槽）

1. **操作步骤**

运行工程文件后，会让你输入测试文件的路径，在输入完毕后，会生成8个文件，分别是：

①dag图以及小优化调试信息文件，这个文件记录了哪些label和跳转语句被删除，以及dag图在哪些基本块做了优化，做了什么优化等

②符号表信息文件，记录了全局和每个分函数的符号信息

③基本块划分情况文件，记录了每个基本块的标号，有哪些语句，以及前驱和后继基本块的标号

④数据流分析信息文件，记录了每个基本块的use/def/in/out集合，同时记录了每个函数内局部变量和临时变量寄存器的分配情况（这个也可以在符号表信息文件查看）

⑤优化后目标代码文件，第一条注释记录了代码条数

⑥优化后中间代码文件，第一条注释记录了代码条数

⑦优化前目标代码文件

⑧优化前中间代码文件

调试信息文件可以自行查看调试，如果要运行目标代码文件，那么就把相应的目标代码文件在Mars中打开运行即可。

1. **测试报告**
2. **测试程序及测试结果**

测试1（正确）

本程序重点测试了函数的递归调用，程序的功能是输入一个数字，输出其对应的斐波那契数

测试2（正确）

本程序重点测试了全局变量和局部变量的作用域，重名等问题，同时测试了表达式的嵌套，功能是输入一个数字，输出从0到这个数字位置的所有斐波那契数

测试3（正确）

本程序重点考察了循环和数组取值赋值，还有函数的递归调用，功能是输入一个数字n，表示有几个数字，然后输入n个数字，然后输出这n个数字按照从小到大排序的结果，以及交换的次数

测试4（正确）

本程序重点考察了表达式以及表达式的输出，表达式的嵌套，函数的调用等，功能是输出指定表达式的值，与正确的值做比较，看是否正确

测试5（正确）

本程序是一个综合性的测试程序，除了上面那些功能性测试外，还测试了switch case，函数参数是函数调用等，还测试了优化的效果等。

功能是输入a~h，分别执行相应的功能。

测试6到测试11，每个分别测试一部分的错误类型，下面是分别的测试点。

1. **测试结果分析**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 错误类型 | 测试6 | 测试7 | 测试8 | 测试9 | 测试10 | 测试11 |
| SQMISMATCH | √ |  |  |  |  |  |
| DQMISMATCH | √ |  |  |  |  |  |
| INVALIDCHAR | √ |  |  |  |  |  |
| ONLYEM | √ |  |  |  |  |  |
| NOTSEMI | √ |  |  |  |  |  |
| NOTID |  | √ |  |  |  |  |
| NOTASSIGN |  | √ |  |  |  |  |
| NOTSINGLECHAR |  | √ |  |  |  |  |
| NOTINTORCHAR |  | √ |  |  |  |  |
| NOTRBRACKET |  | √ |  |  |  |  |
| NOTRPARENT | √ |  |  |  |  |  |
| NOTFACTOR | 不是 一个<因子>的开始单词 |  | √ |  |  |  |
| NOTLBRACE | 不是 { |  | √ |  |  |  |
| NOTRBRACE | 不是 } |  | √ |  |  |  |
| NOTMAIN | 不是 main保留字 |  | √ |  |  |  |
| NOTLPARENT | 不是 ( |  | √ |  |  |  |
| NOTUNSIGNEDINT | 不是一个非0无符号类型 |  | √ |  |  |  |
| NOTSTATEMENT | 不是 一个<语句>的开始单词 |  |  | √ |  |  |
| NOTELSE | 不是 else |  |  | √ |  |  |
| NOTCONSTANT | 不是 常量 |  |  | √ |  |  |
| NOTCOLON | 不是 : |  |  | √ |  |  |
| NOTCASE | 不是 case |  |  | √ |  |  |
| NOTVARDEF | 不是 变量定义的开始符号 |  |  |  |  |  |
| DUPLICATEDEF |  |  | √ | √ |  |  |
| PARATYPEDISMATCH |  | √ |  | √ |  |  |
| PARANUMDISMATCH |  | √ |  | √ |  |  |
| IDNOTDEF | √ |  | √ | √ |  |  |
| NOTAINT | 不是 int类型的表达式 |  |  |  | √ |  |
| NOTANARRAY | 不是 数组类型的标识符 |  |  |  | √ |  |
| NOTAFUNC | 不是 函数名 |  |  |  | √ |  |
| NOTANOPARAFUNC | 不是 一个无参函数 |  |  |  | √ |  |
| INVALIDID | 不是 一个被正确使用的标识符 |  |  |  | √ |  |
| NOTASIMPLE | 不是 一个简单变量 |  |  |  | √ |  |
| CASESWITCHDISMATCH | case常量类型和switch的表达式类型不一样 |  |  |  | √ |  |
| NONERETURN | 有返回值的函数没有一句return |  |  |  |  | √ |
| CANNOTRETURN | void类型的函数return了一个实际值 |  |  |  |  | √ |
| RETURNTYPEDISMATCH | return的类型不匹配 |  |  |  |  | √ |
| INTTOCHARNOTALLOW | 不允许int到char的转换 |  |  |  |  | √ |
| NORETURNVALUE | 不是 一个有返回值的函数 |  |  |  |  | √ |
| CASEVALUEDUPLICATE | case 常量重复 |  |  |  |  | √ |
| DIVCASE | 可以检测到的除以0 |  |  |  |  | √ |

1. **总结感想**

这个学期的编译终于完成了，现在在写总结感想的我感慨万千。

感想一：

**世上无难事，只怕有心人**。在经过计组和OO的洗礼后，编译对我来说不是那么的难以逾越的鸿沟了。一切困难都是纸老虎！

从一开始的不知道怎么语义分析，到后来全局数据流分析，寄存器分配，再到后来dag图算法的扩展和修改，曾经觉得很难理解的东西，现在觉得也就是那么回事了，关键在于你敢不敢迈出第一步！

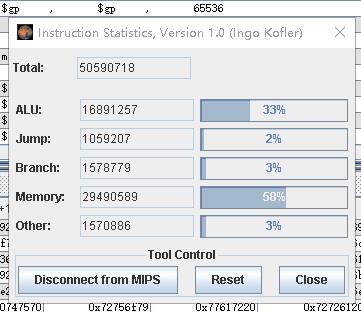
感想二：

**纸上得来终觉浅，绝知此事要躬行**。书上有很多优化算法，但是都没有很明确严谨的算法阐述，或者说书上的算法都是在某些特定条件下的算法，比如dag图的找最左子节点的启发式算法，在我们的课程设计C0文法的条件下完全不需要遵守，所以，我们不仅要理解书上的算法，**更要把这个算法的思想运用到我们的实际问题中**！

感想三：

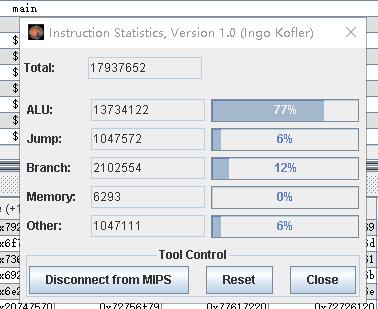
**锲而不舍，金石可镂**。做任何事情都贵在坚持，坚持对自己所做事情的高要求，坚持对编译的最大热情，坚持克服一切可以克服的难题，就像长跑一样，保持最佳状态跑完全程才是最难能可贵的。

优化虽然很难，但是进步和成果是让人兴奋的，老师给的优化测试程序，没有优化时，我执行完所有指令的统计如下：



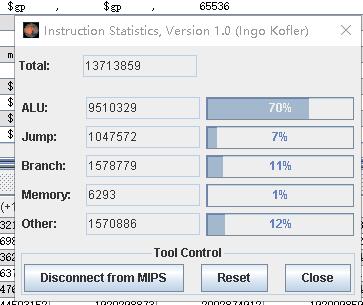
可以看到，memory指令占据了大部分，效率极低。

在做了优化，分配了全局寄存器之后，统计如下：



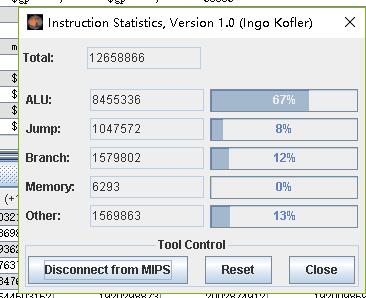
可以看到执行的指令数量从5000万降到了1800万左右，最重要的是memory指令仅仅只有6000多条！

在对dag图算法进行进一步优化后，统计如下：



指令条数进一步降到了1300万

之后我又做了一些赋值的小优化后，统计如下：



可以看到指令条数又少了100万！

从一开始的5000万到现在的1200万左右，优化的效果还是非常明显的。不过没有太多时间继续优化了，只能作罢，所以这个过程虽然痛苦，但是学到的东西还是很多的。

最后感谢老师和助教的帮助！