《编译技术》课程设计文档

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

姓名：\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

2016年 12 月 28 日

## 一．需求说明

### 1．文法说明

**1）＜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｝"**

范例： 字符串：abc&d\*,e

字符：’+’ ‘\*’ ‘\_’

数字：0，1

分析：

这里规定了一些C0中的一些基本的元素，没什么好说的，但要注意一些问题，字符串中不包括双引号（因为引号要作为字符串的界定符号），也没有换行符。估计这些符号的使用是通过转义，但由于没有语义分析，现在也不得而知。字母中有大写字母和小写字母。另外，字符串可以是空串。

作用：是构成C0语言的基本元素。

限制：字符串中没有双引号，下划线也算在字母之中。 !=表示不等号。没有!、+=、-=、++、--等常用运算符，所以在词法分析的时候要稍加注意。

**2）＜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＞｝**

**＜类型标识符＞ ::= int | char**

范例： 类型标识符：int、char

整数： 0、-123

标识符： \_ab1， c2d

分析：

这里规定了C0的一些标识符等元素。类型标识符被限定为只有char和int两种。无符号整数是由非零数字加数字的重复构成的，整数是0 或者在无符号整数前加+-构成。标识符由字母加字母或数字的重复构成，且标识符区分大小写。

作用：构成C0语言的元素，规定了整数的定义，标识符的构成等。

限制： 非0整数前面可有正负号（可有可无）， 但0前面一定没有正负号； 标识符一定要以字母开头，后面可以跟字母或数字；类型标识符只用所给出的两种；无符号整数必须要以非零数字开头，这也意味着非零数字前面不能有零。

1. **＜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＞‘]’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’**

范例：因子：a, b[c+1], 2, ‘t’, funcA(), a + i

项：a \* b[c+1] / 2, funcA() \* t

表达式：+ a \* b[c+1] – a / 2, k – 1 + ‘t’

分析1：项由因子或因子后跟带乘法运算符的因子所组成。

作用：由因子能够构成项。

限制：项不为空，且因子与因子之间必须为乘法运算符，第一个因子前不能有符号。

分析2：因子由标识符、标识符[<表达式>]、整数、字符、函数调用、带括号的表达式等组成。

作用：定义了因子的结构。

限制：函数调用是否有返回值无法通过语法来判断，只能借助语义分析。若因子由表达式构成，则表达式的左右必须有小括号。

分析3：表达式由一个或多个项构成，项之间用加法运算符号连接，第一个项前面的正负号可有可无。

作用：定义了表达式的构成方法。

限制：表达式不能为空，项与项之间必须有加法运算符。

**4）＜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＞;**

分析：这里主要要注意的就是各类语句后面是否有分号，比如 有分号的有函数调用、赋值、读写、返回语句和空语句，这与c语言的语法相一致，但需要特别注意空语句，只有它后面加了分号才不会在其它语句里引起歧义。 不加分号的有：条件、循环、语句块，这个也与C语言的平时用法相一致。

作用：定义了C0语言的语句的种类。

限制：空语句后面必须有分号。语句列前后必须用大括号包围。

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

范例：a = k – 1 + ‘t’ b[2 + k] = a \* b[c+1] – a / 2

分析： 赋值语句有两种，分为左部是普通变量和左部是数组的情况，等号右部则都是表达式。需注意分号在<语句>中定义了，所以这里不需要分号。

作用：定义了赋值语句的构成。

限制：注意句尾无分号。等号左侧只能是标识符或标识符[表达式]，不能有数字或字符串等字面量。

**6)＜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**

范例： if( a + b != k \* t -2)

{

a = 3; k = k + 1;

}else {

a = 4; k = k + 2;

}

If(a == 3){

funcA();

}

If(a < b);

else;

分析：条件语句中，if及其所带的语句是必须有的，else和它所带的语句可有可无。但一旦有else，则必须遵循先if后else的原则。由此条件也可以看出else是与离它最近的if相匹配的。另外if和else的后面必须有语句（当然语句可以有<空>;）。

If的条件语句有两种构成方法，其一是两个表达式，中间用关系运算符连接，其二是单个表达式。

作用：定义了条件语句的构成方法。

限制：if 和它的条件和它的语句必须有， else及其语句可有可无。 条件中只能出现0或1个关系运算符。另外要注意没有’！’这个单目逻辑运算符，所以条件中无法出现否定操作。

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

**＜步长＞::= ＜无符号整数＞**

范例：do{

a = a \* c;

i = i + 1;

}while(i < 10)

for(i = 10\*3; i < 1000; i = i + 3){

a = a\*c;

funcA();

}

步长： 12， 16

分析：循环语句定义了两种循环，do-while和for循环。其中do-while语句中的do和while保留字必须配套使用，不能单独出现。while中的条件只允许有一个。for语句后面的括号里，条件部分也只允许有一个条件。＜标识符＞＝＜标识符＞(+|-)＜步长＞也只允许有一条。

作用：定义了do-while和for两种循环语句。

限制：do-while必须同时出现。for循环中循环变量的初始化、条件判断、循环变量的递增都不能为空，且每部分只有一条。步长必须是无符号整数。

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

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

**＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞**

范例：funcA(a, b+4)

funcB()

funC(2, funcA(a, b+4))

分析：有无返回值，二者在调用时的语法是相同的。函数名后面的小括号必不可少。值参数列表由若干以逗号分隔的表达式组成。另外注意值参数表可以为空。还有最重要的一点，有返回值的函数调用语句可以作为项中的一个因子，而无返回值的函数调用语句则不能。

作用：定义了如何调用函数。

限制：标识符后面必须有括号。值参数表可以为空，也可以有多个表达式，以逗号分隔。

**9)＜语句列＞ ::=｛＜语句＞｝**

范例：

{

A = b;

funcA(a, b+4);

t = t + 1;

}

分析：语句列由大于等于0条语句组成。特别要注意这里的大括号是BNF里的大括号，不同于’{’ ‘}’，虽然早就知道带引号的才是文法里的符号，但当二者都是大括号时，我还是忘掉了，想了半天...

作用：大括号加上语句列就可以组成语句块。

限制：没有限制，大于等于0条语句就可以构成语句列。

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

范例：scanf(a, b, c)

scanf(k)

分析：这条语法限制比较死，没有对输入的数据的类型做以限制，所以括号内跟1到n个标识符即可，至于类型，应该是语义分析的时候检查吧。

作用：规定了读语句的格式。

限制：读语句括号内至少要有一个标识符，且不能读取并存入数组的元素，类似这样a[1]。

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

范例：printf(“abcde”, c\*4+5)

printf(“ajsdlkjas”)

printf(a+b+c-1)

分析：三种输出方式，输出一个字符串和一个表达式的值、输出一个字符串、 输出一个表达式的值。注意只有这三种输出方式，无法输出n个字符串或者n个表达式或者先表达式后字符串的类型。

作用：规定了写语句的格式。

限制：只有三种输出方式，输出一个字符串和一个表达式的值、输出一个字符串、 输出一个表达式的值；其它格式都为错误。

**12)＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]**

范例： return (a + b)

return

return (10\*10)

分析：两种返回，一种什么也没有，仅仅return，一种返回表达式的值，但该表达式必须用括号括起来。

作用：规定了返回语句的格式。

限制：若return后面有返回值，则必须用括号括起来；若没有返回值，则仅仅写return。

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

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

**＜声明头部＞ ::= int＜标识符＞|char＜标识符＞**

**＜参数＞ ::= ＜参数表＞**

**＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞**

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

**＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞**

**＜语句列＞ ::=｛＜语句＞｝**

范例：

int funcA(int c, char d)

{

const int x = 1;

char t;

语句略

}

funcB( )

{

const int x = 2;

char t;

语句略

}

void main()

{

const int x = 3;

char t;

语句略

}

分析1：有返回值函数与无返回值函数只有一点不同，前者函数名前有类型标识符，而后者函数名前有void。

分析2：所有函数的函数体可以为空，也没有规定必须有return语句。

分析3：常量声明变量声明和语句都是可有可无的，但一旦出现，常量声明—>变量声明—>语句的顺序是必须的。

分析4：<类型标识符><标识符>可以有0到n个，之间用逗号分隔。

作用：这些文法定义了一个函数在语法上的结构。

限制：无返回值函数名前要有void；有返回值和无返回值的函数的参数表都可以为空，若不为空，参数只能为int或char型；main函数前必须有void，参数列表必须为空。

**14)＜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＞}**

**＜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＞‘]’) }**

范例：

const int a = +5, b = -3;

const char c = ‘1’;

int d, e[2];

void funcA()

{

函数内容略

}

int funcB()

{

函数内容略

}

void main()

{

函数内容略

}

分析1：有所给的文法可知，程序的各个部分的顺序是确定的，常量说明—>变量说明—>函数定义—>主函数；其中常量说明和变量说明和函数定义都是可有可无的，但一旦有，则必须按照上面的先后顺序出现。主函数必须存在

分析2：对于常量说明部分，允许有多次常量定义； 对于常量定义，一个const后面只有一个类型标识符，所以一个const后面只跟一种类型；

分析3：常量定义必须初始化，且赋值方式只有一种，int使用整数，char使用字符。（这与c语言的常量定义不太一样，C语言允许 char a = 10， 而该文法char的右边只允许字符）另外，c语言可以用表达式初始化常量，而该文法不允许。 另外，常量定义没有数组。

分析4：变量说明和常量说明类似，但没有const，它允许存在多条变量定义； 对于变量定义，除了普通变量之外，还有数组。此外，变量定义仅仅定义，不对其初始化。

分析5：还有一些发现，比如该文法没有注释系统，也没有C语言常有的include等。

作用：这些语法规定了一个完整的C0程序应该有的语法结构。

限制：程序的各个部分的顺序是确定的，常量说明—>变量说明—>函数定义—>主函数；常量定义右侧仅能使用数字或字符，不能用表达式。变量定义只能定义，不能初始化。一个程序必须有且仅有一个主函数（其它成分可有可无）。

### 2．目标代码说明

本编译器生成的目标代码为mips码。以下为简要说明

|  |  |
| --- | --- |
| 指令 | 含义 |
| Add | 加法指令 |
| Sub | 减法指令 |
| Mul | 乘法指令 |
| Div | 除法指令 |
| Blez | 小于等于0跳转 |
| Addi | 加立即数 |
| Bgez | 大于等于0跳转 |
| Bne | 不等于跳转 |
| Bltz | 小于0跳转 |
| Bgtz | 大于0跳转 |
| Beq | 等于跳转 |
| J | 立即跳转 |
| Sw | 存数指令 |
| Subi | 减立即数 |
| Lw | 取数指令 |
| Jal | 链接并跳转 |
| Jr | 按指定寄存器跳转 |
| Syscall | 系统调用 |

### 3. 优化方案\*

1. 针对全局寄存器的优化

采用活跃变量分析法找出跨基本块活跃的变量，在其中找出活跃范围重合的变量，构建冲突图，并采用染色法进行寄存器分配。

1. 针对局部寄存器的优化

采用寄存器池对局部寄存器进行管理，负责局部寄存器的分配和载入和写回。

常量合并与传播。

## 二．详细设计

### 程序结构

**整个程序共分为四大部分：**

1. 词法分析部分：该部分进行词法分析，并向语法分析提供识别出的单词。
2. 语法语义分析及中间代码生成：该部分进行语法分析，识别不合语法的成分并报告错误。在语法分析的同时进行语义分析，并报告语义错误。在语义分析的同时可以做中间代码的生成，即四元式生成。
3. 代码优化：该部分分为两块，做跨基本块的活跃变量分析，并根据分析结果进行全局寄存器的分配。面向基本块内优化，进行局部寄存器的管理与分配，常量合并和传播。
4. 目标代码生成： 在中间代码与mips汇编代码之间建立映射关系，将四元式翻译成目标代码。

### 类/方法/函数功能

按功能划分C函数：

* 1. **词法分析部分：**

getch()函数：从源程序读取一个字符，将其存入全局变量供词法分析程序使用。

getsym()函数：通过getch()函数不断获取字符，并将其组装成一个符号，存入全局变量以便使用。在分析过程中如果遇到不和语法的情况会相应报错。

erro()函数：错误处理函数，被调用后会使错误数加一，并且根据错误编号和当前行号报告错误详细信息。

isLetter()函数：判断是否是字母。

IsNumber()函数： 判断是否是数字。

Output()函数： 调试使用，输出单词的类型。

* 1. **语法分析部分：**

1. program()函数：

对应文法＜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＞

1. constdec()函数：

对应文法＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

1. constdef()函数：

对应文法＜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＞}

1. variabledec()函数：

对应文法＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

1. variabledef()函数：

＜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＞‘]’) }

1. funcdef()函数：

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

7) rfuncdef()函数：

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

1. parameterlist()函数：

＜参数＞ ::= ＜参数表＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

1. mainfunc()函数：

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

1. comStatement()函数：

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

1. statements()函数：

＜语句列＞ ::=｛＜语句＞｝

11）statement()函数：

＜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＞;

12）ifstatement()函数：

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

13）condition()函数：

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞

14）loopstatmenet()函数：

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

15）step()函数：

＜步长＞::= ＜无符号整数＞

16）callfunc()函数：

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’

17）callrfunc()函数：

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)

18）vparameterlist()函数：

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

19）readstatement()函数：

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

20）writestatement()函数：

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

21）returnstatement()函数：

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

22) becomestatement()函数：

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

22）expression()函数：

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

23）term()函数：

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

24）factor()函数：

＜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＞‘)’

25）integ()函数：处理整数

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

* 1. **语义分析及中间代码生成部分**

1. gen()函数：

按照给定的参数和操作符生成一个四元式的结构体，并存在一个列表中。

1. stringNameGen()函数：

tag的生成器，每次调用都生成一个独一无二的tag

1. symTableNew()函数：

创建一个新的符号表,并且在符号表中插入该函数名字。

1. findsymCur()函数：

在当前符号表中查找符号名字，如果查找到，返回1。

1. findsymGol()函数：

在全局符号表中查找name，查找到返回1.

1. findFuncName()函数：

在所有符号表中查找符号表名（函数名）。

1. findFunc()函数：

查找函数名，并返回函数名的符号表项。

1. findFuncCur()函数：

获得当前所在函数的函数项。

1. findcvpa()函数：

查找常量、变量或参数或数组，先找本层，找不到就找全局，还找不到就返回NULL。

10) constdefInsert()函数：

插入常量定义至当前符号表。

11）vardefInsert()函数：

插入变量定义至当前符号表。

12）paraInsert()函数：

生成并插入一个参数至符号表。

13) genTempVarName()函数：

为临时变量产生一个独一无二的名字。

14）genTempVar()函数：

产生临时变量，并存入符号表。

15）genCints()函数：

生成字面量item，不插入符号表。

16）genChars()函数：

生成字面量item，不插入符号表。

17）genCstring（）函数：

生成字符串item，不存入符号表。

18）itemInit()函数：

初始化一个符号表项。

19）funcInsert()函数：

补充当前函数的详细信息至符号表。

20）parameterInfoInsert()函数：

补充参数总数到当前符号表的所有形参。

21）addrdispatch()函数：

内存分配函数。

* 1. **目标代码生成和优化**

1. 根据基本块的定义对中间代码进行基本块的划分，在进行活跃变量分析并使用图着色算法分配全局寄存器。

Find()函数：

查找节点在活跃信息表中的位置，如果没有则新建一个，并传回其位置

getBlocks()函数：

划分基本块。

use\_loaction()函数：

分析use，返回第一次被使用的位置。

def\_loaction()函数：

分析一个变量的def，第一次被定义的位置。

use\_def()函数：

综合根据use信息和def信息来确定一个基本块的use与def集合。

Union()函数：

求两个集合的并。

next\_Gunion()函数：

对该基本块的所有后继基本块求广义并，存入该基本块的out集合。

Minus()函数：

求a与b的差集，放入c。

in\_value()函数：

求该基本块的in集合。

confi\_graph()函数：

构建冲突图。

remove\_node()函数：

从冲突图中移除一个节点，并做纪录。

color\_reg ()函数：

染色算法进行图着色，并利用启发式算法分配寄存器。

active\_value()函数：

活跃变量分析。

1. 使用寄存器池进行临时寄存器的分配管理，并生成目标代码。

Loadreg()函数：

将一个变量载入指定的寄存器。

Savereg()函数：

将一个变量从指定寄存器写回运行栈。

Addressing()函数：

数组寻址函数，根据传入的数组名和数组的偏移进行寻址，将得到的数组元素的运行栈地址写入$t9寄存器。

clear\_pool()函数：

清空寄存器池。

writeback\_pool()函数：

写回寄存器池中的所有寄存器。

req\_reg()函数：

申请临时寄存器，如果该变量已经被分配了寄存器，则直接返回该寄存器。

genMips()函数：

建立起中间代码到目标代码之间的映射，将四元式翻译成mips汇编指令。

fetchcode()函数：

对所有代码进行翻译，并建立数据段和代码段，将字符串存储在data段。

printcode()函数：

输出目标代码至文件。

### 3．调用依赖关系

**3.1 各功能之间的关系**

词法分析

单词符

目标代码生成

语法语义分析

四元式

四元式

经过优化的四元式

优化

**3.2 函数之间的调用关系**

getsym()函数调用：getch、erro、isLetter、isNumber、Output等函数

语法分析部分的函数调用与文法成分间的关系保持一致，不再叙述。

program()函数调用：gen、stringhandle、enter、test、skip、position等函数

### 4．符号表管理方案

由于C语言不允许函数的嵌套定义，所以函数之间是平行的关系。编译时一个变量在本函数中没找到，只能再查找全局变量，所以符号表的设计相对于pascal就没有那么复杂。

对于符号表，以函数为单位进行存储，一个函数拥有一个自己的符号表。对于全局变量，把它们也存储在一个固定的函数中（该函数不实际存在），存在固定的位置。

符号表中存储的基本单元为item元素，其数据结构如下：

typedef class node{ //符号表项, 函数名，参数，常量，变量，数组，临时变量

public:

string name;

\_kind kind;

\_type type;

int level; //所在层次，只有两种取值，0全局，1局部

int isTemp; //标记是否是临时变量

int valid; //优化时使用，查看是否被优化删除

int addr; //相对于基地址的偏移

int paranum; //函数名使用,参数个数

\_type returnV; //函数名使用,返回值类型，取值为ints、chars、nop，nop表示无返回值

int length; //数组长度（由于只有一维数组）

long valueInt; //字面量int的值

char valueChar; //字面量char的值

string svalue; //字符串字面量的值,也可以作为tag

int funlen; //包括了参数+数组元素+变量+常量的个数(仅函数使用)

int parath; //说明自己是第几个参数(形参或实参)，仅参数使用

int isOptimized; //是否被全局优化

int reg; //全局优化所分配的寄存器, 0-7

int isDelete; //是否在构建的dag图中被删掉，0表示未删掉

} \*item, ITEM;

### 5．存储分配方案

1）栈空间向上生长，栈空间基地址0x10040000。

2）传参未采用mips的传参建议方法，即不通过$a0 ~ a3寄存器传递，直接采用复制内存的方式。

3）在进入函数时对相应的寄存器（全局寄存器、fp、ra等）进行备份，在当前函数结束并退栈前要用取数指令将所有的备份寄存器全部恢复。

以函数A调用B，B调用C为例：

函数A

|  |
| --- |
| 实际参数 |
| $Ra 返回地址 |
| fp、sp(调用者栈帧) |
| $S0 ~ S7 保留寄存器的备份 |
| 本地变量、临时变量、形参 |
| 实际参数 |
| $ra 返回地址 |
| fp、sp(调用者栈帧) |

函数B

地址增大

函数C

### 6. 解释执行程序\*（无）

### 7. 四元式设计\*

四元式采用结构体存储。结构体成员有：

opc: 枚举类型，操作符

op1： 第一个操作数

op2： 第二个操作数

obj： 目标操作数（同a）或跳转位置标记等。

isBlockB: 标识该四元式是否为基本块的入口语句。

isBlockE: 标识该四元式是否为基本块的结束语句。

其中op1、op2、obj只存储item项的指针，item的定义在符号表中已经说明。

四元式设计：

|  |  |
| --- | --- |
| 操作 | 详细语义 |
| ADD | op1与op2相加，结果存入obj |
| SUB | op1与op2相减，结果存入obj |
| MUL | op1与op2相乘，结果存入obj |
| DIV | op1与op2相除，结果存入obj |
| NEG | op1取负，结果存入obj |
| ASSIGN | op1的值赋给obj |
| GTR | 大于置1，op1大于op2，则给obj赋值1 |
| LES | 小于置1，op1小于op2，则给obj赋值1 |
| EQL | 等于置1，op1等于op2，则给obj赋值1 |
| GEQ | 大于等于置1，op1大于等于op2，则给obj赋值1 |
| LEQ | 小于等于置1，op1小于等于op2， 则给obj赋值1 |
| NEQ | 不等于置1，op1不等于op2，则给obj赋值1 |
| NEQZ | 不等于0置1， op1不等于0，则给obj赋值1 |
| TJMP | 为真跳转，op1为真（1），则跳转到obj |
| FJMP | 为假跳转，op1为假（0），则跳转到obj |
| JMP | 无条件跳转，直接跳转到obj |
| TAG | 代表标签 |
| READ | 读语句，读一个值到obj |
| WRITE | 写语句，输出obj到标准输出 |
| HEADER | 函数开始，obj存放函数名 |
| FOOT | 函数结束， obj存放函数名 |
| RARRY | 读数组操作，op1为数组，op2为偏移，obj为目标 |
| WARRY | 写数组操作，op1为数组，op2为偏移，obj为要写的值 |
| FORPARA | 函数形参声明， obj为形参 |
| ACTPARA | 函数实参声明，obj为实参 |
| RCALL | 有返回值的函数调用，op1为函数名，obj为返回值 |
| CALL | 无返回值函数调用，op1为函数名 |
| NRETN | 无返回值返回 |
| RETN | 有返回值返回，obj为返回的值 |

### 8. 目标代码生成方案\*

代码生成在四元式的基础上进行。

8.1 使用一个string数组存储不断生成的mips代码。

8.2 在进入一个函数之后，需要执行函数序言。保存一些寄存器信息，在函数返回之前，需要恢复保存的寄存器。需要执行的操作如下：

mipscd = "sw $ra, 0($sp)"; //保存ra

codes.push\_back(mipscd);

mipscd = "sw $fp, 4($sp)"; //保存fp

codes.push\_back(mipscd);

Npara = obj->paranum;

Npara = Npara \* 4;

mipscd = "subi $t0, $sp," + num\_to\_str(Npara); //t0作为sp的回退地址

codes.push\_back(mipscd);

mipscd = "sw $t0, 8($sp)"; //保存sp

codes.push\_back(mipscd);

mipscd = "add $fp, $zero, $sp"; //建立自己的栈

codes.push\_back(mipscd);

mipscd = "sw $s0, 12($sp)"; //保存s0

codes.push\_back(mipscd);

mipscd = "sw $s1, 16($sp)"; //保存s1

codes.push\_back(mipscd);

mipscd = "sw $s2, 20($sp)"; //保存s2

codes.push\_back(mipscd);

mipscd = "sw $s3, 24($sp)"; //保存s3

codes.push\_back(mipscd);

mipscd = "sw $s4, 28($sp)"; //保存s4

codes.push\_back(mipscd);

mipscd = "sw $s5, 32($sp)"; //保存s5

codes.push\_back(mipscd);

mipscd = "sw $s6, 36($sp)"; //保存s6

codes.push\_back(mipscd);

mipscd = "sw $s7, 40($sp)"; //保存s7

codes.push\_back(mipscd);

funlen = obj->funlen;

funlen = funlen \* 4 + 44; // 分配变量、数组、参数的空间 （从44开始）

mipscd = "addi $sp, $sp, " + num\_to\_str(funlen); //sp移动到栈顶

codes.push\_back(mipscd);

8.3 由于中间代码与目标代码的形式已经较为相似，所以采取对不同的语句建立映射的函数的方法进行翻译，算法：

1）对一条读写型的四元式指令，先向寄存器池申请寄存器，获得寄存器之后再执行翻译。

2）对于非读写型四元式，则直接翻译即可。

### 9. 优化方案\*

**9.1** **划分基本块：**

以函数为基本单位存储基本块,数据结构：

typedef class class\_fblock{

public:

vector<block> myblocks; //以函数为单位存储基本块

vector<actinfo> myactinfo; //活跃节点信息

vector<confi\_node> myconfi\_info; //冲突图

vector<int> myorder; //移走节点的顺序

}FBLOCK;

基本块的数据结构：

typedef class class\_block{ //基本块

public:

int id;

vector<class\_block\*> pre;

vector<class\_block\*> next;

vector <quater> block\_quter;

vector <item> use;

vector <item> def;

vector <item> in;

vector <item> out;

}BLOCK;

算法：

输入：中间代码序列

输出：基本块序列

方法：

1. 确定入口语句
2. 整个语句序列的第一句为入口语句
3. 跳转语句跳转到的第一句为入口语句
4. 紧跟在跳转语句的下一句为入口语句

2）每个入口语句到下一个入口语句之间为一个基本块

**9.2 目标代码生成优化：**

采用寄存器池对临时寄存器进行管理与分配：

寄存器池的数据结构如下：

typedef class class\_regs\_pool{

public:

int regPointer; //指针,始终指向

item regs[7]; //临时寄存器池，t0-t6

}REGS\_POOL;

算法：

1）进入基本块时，清空临时寄存器池

2）为当前中间代码生成目标代码时，无论临时变量还是局部变量，只要需要使用临时寄存器，都可以向临时寄存器池申请。

3）如有空闲寄存器，则分配，并将该寄存器纪录为已占用。

4）如果没有空闲寄存器，则将当前生成代码中不会被使用到的寄存器写回运行栈。并将该寄存器分配为给申请者。

1. 在基本块结束或函数调用发生前，写回所有临时寄存器至运行栈，清空临时寄存器池。

**9.3 活跃变量分析及全局寄存器分配：**

A．基本块的活跃变量数据流分析：

输入：程序流图，且基本块的use集合和def集合已经计算完成。

输出：每个基本块入口和出口处的in和out集合，即in[b]和out[b]。

方法：

1. 将包括代表流图出口的基本块的所有基本块的in集合，初始化为空集。
2. 根据方程out[B] = U后继块P in[P]， in[B] = use[B] U (out[B] – def[B])。为每个基本块B依次计算out[B]和in[B]。如果计算得到的某个基本块的in[B]与此前计算得出的该基本快的in[B]不同，则循环执行步骤2），直到所有的基本块的in[B]集合不再变化为止。

B．图染色法分配全局寄存器：

输入：变量冲突图G，全局寄存器数量K，对应K种不同的颜色。

输出：全局寄存器的分配结果，变量到寄存器的对应关系。

方法：

1. 在冲突图中找到第一个连接边数目小于K的节点，把它从图G种移走，并记录。
2. 重复步骤1，记录节点的移走顺序，直到无法再从冲突图中移走任何节点。
3. 在图中选取一个恰当的节点，将他标识为不分配全局寄存器的节点，并从图中移走。
4. 重复1-3步骤，直到图中只剩下一个节点。
5. 给剩余的最后一个节点任意选取一种颜色，然后按照节点被移走的逆序，将点和边添加回去，并依次给新加入的节点选取颜色，在着色的过程中要保证有连接边的节点需着不同颜色。

### 10. 出错处理

**10.1 错误信息列举:**

|  |  |
| --- | --- |
| 错误代码 | 错误内容 |
| 0 | 出现未知符号 |
| 1 | 数字长度超过限制 |
| 2 | 字符串缺少” |
| 3 | 字符缺少’ |
| 4 | 错误的字符（不能作为字符常量的字符） |
| 5 | 数字前面出现前导0 |
| 6 | 函数返回值部分有错误 |
| 7 | 缺少const |
| 8 | 缺少; |
| 9 | 缺少= |
| 10 | 缺少数组长度 |
| 11 | 缺少] |
| 12 | 缺少（ |
| 13 | 缺少） |
| 14 | 缺少{ |
| 15 | 缺少} |
| 16 | 缺少返回值符号 |
| 17 | 缺少标识符 |
| 18 | 缺少类型说明 |
| 19 | Main函数返回值不为void |
| 20 | 缺少main |
| 21 | 赋值语句错误 |
| 22 | 缺少do |
| 23 | 缺少while |
| 24 | 缺少for |
| 25 | 缺少scanf |
| 26 | 缺少printf |
| 27 | 错误的符号 |
| 28 | 缺少return |
| 29 | 缺少数字 |
| 30 | 重复定义 |
| 31 | 等号右边非法类型 |
| 32 | 0前面出现正负号 |
| 33 | 未初始化的常量 |
| 34 | 函数重复定义 |
| 35 | 未定义的函数或常量或变量 |
| 36 | 赋值对象不合法 |
| 37 | 非法的表达式的值 |
| 38 | 非法的返回值类型 |
| 39 | 错误地使用[] |
| 40 | 非法的数组索引 |
| 41 | 无效的条件数据类型 |
| 42 | 非法的循环变量 |
| 43 | 非法的循环步长或缺少步长 |
| 44 | 无返回值的函数 |
| 45 | 参数不匹配 |
| 46 | 不应有的返回值 |
| 47 | 没有返回值 |
| 48 | 非法的因子 |
| 49 | 非法的符号 |
| 50 | 缺少if |
| 51 | 缺少运算符 |
| 52 | 程序提前结束 |
| 53 | 括号不匹配或出现未知符号 |

**10.2 出错处理方案：**

1. 指出出错位置，以及错误信息。
2. 对错误个数进行计数，在编译的最后，只有错误数为0才可以认为编译正确。
3. 对已经识别出的错误，尽可能进行校正，以免影响后续的编译过程。
4. 采取错误局部化处理，当发现错误之后，若不确定能否校正，可以采取跳读，找到新的语法成分继续分析。

**10.3 涉及的相关函数：**

1）skipUntill()函数：跳读符号，知道遇到给定的符号集合中的符号，说明已经成功跳过错误代码部分，可以正常进行分析。

2）erro()函数：对错误计数器加一，并且根据错误代号和行缓冲区的指针报告错误内容和位置。

## 三．操作说明

### 1．运行环境

本编译器在windows操作系统下采用vs2010开发，使用语言为c++。本编译器的运行环境为windows。无需搭建环境，可直接运行。若要运行生成的mips汇编文件，可以先安装java环境，再使用mars的mips虚拟机进行执行。

源语言：扩充的C0文法语言

目标语言：mips汇编指令

### 2．操作步骤

双击执行编译器exe文件，键入待编译的文件路径即可，编译完成后即可在当前目录下生成一个mips汇编文件。

## 四．测试报告

### 1．测试程序及测试结果

Test1:

const int a2 = +2, a3 = 0;

const int \_a = 4;

const char b1 = 'a', b2c\_ = 'B';

const char b3 = '\*';

int a1 ,c, c1[8], c2;

char d, d1[10], d2;

char g3[100];

int add(int a, int b){

int c;

c = a + b;

return (c);

}

int mult(int a, int b){

return (a \* b);

}

void Fnull1(){

}

void Fnull2(){

return;

}

void Fnull3(){

;

}

char move1(char c){

const int c3 = 1;

int c4;

c4 =add(c3,c);

return (c4);

}

int fact(int h){

if(h <= 1){

return (1);

}

else {

return (h \* fact(h - 1));

}

}

void main()

{

const char endchar = 'Z';

int e1, e2, e3, e4[20], i;

char g1, g2[10];

char g3[100];

e1 = 1; e2 = 0; e3 = 0; i = 0;

g1 = 'a';

printf(b3);

for(i = 1; i < 6; i = i + 1){

g3[i] = 'a' + i;

}

for(i = 0; i < 5; i = i + 1){

printf(g3[i+1]);

}

g3[0] = '\_';

g3[11] = 'k';

g3[12] = 'l';

g3[13] = 'm';

g3[14] = 'n';

g3[15] = 'o';

g3[16] = 'p';

g3[17] = 'q';

g3[18] = 'r';

g3[19] = 's';

g3[20] = 't';

g3[21] = 'u';

printf(b3);

for(i = 1 + 2+-3; i < 10; i = i + 1){

e4[i] = ((i + i \* 3) - 4 \* i / 4 ) /3;

printf("e4[i]", e4[i]);

}

printf(b3);

for(a1 = 0; a1 < 10; a1 = a1 + 1){

e4[a1] = a1 \* a1;

printf(e4[a1]);

}

printf(b3);

for(c = 0; c < 10; c = c + 1){

d1[c] = 'a' + c;

printf(d1[c]);

}

printf(b3);

g2[0] = endchar;

i = 1;

do {

g2[i] = -1 + g2[i - 1];

printf(g2[i]);

i = i - -1;

}while(i < 10)

;

scanf(e1);

if(e1){

printf("fact ", fact(e1));

}else{

scanf(e2,e3);

printf("add");

printf("");

printf( add(e2, e3));

}

for(i = 20; i > 10; i = i - 1){

printf(" ");

printf( g3[i] - 'a');

}

scanf(e1);

if(e1 < 0);

else;

if(e1 == 0);

if(e1 >= 0){

printf("e1>=0");

}

for(i = 100; i > 0; i = i - 1);

i = -1;

do;while(i > 0);

if(i != -1);

Fnull1();

Fnull2();

Fnull3();

mult(2, 3);

g1 = move1('b');

printf("move1 ", g1);

}

输入：5 9

输出：\*bcdef\*e4[i]0 e4[i]1 e4[i]2 e4[i]3 e4[i]4 e4[i]5 e4[i]6 e4[i]7 e4[i]8 e4[i]9\*0149162536496481\*abcdefghij\*YXWVUTSRQ fact120 19 18 17 16 15 14 13 12 11 10 e1>=0move1 c

输入：0 3 4 -1

输出：\*bcdef\*e4[i]0 e4[i]1 e4[i]2 e4[i]3 e4[i]4 e4[i]5 e4[i]6 e4[i]7 e4[i]8 e4[i]9\*0149162536496481\*abcdefghij\*YXWVUTSRQ add7 19 18 17 16 15 14 13 12 11 10 mov1 c

Test2: Test1的错误版：

错误内容：第一行行末删掉了分号，Fnull2函数返回1

输出：

line:2 error:1 缺少;

line:27 error:2 不应有的返回值

错误数：2

Test3: 常量合并

const int a = 2, b = 3;

const char c = 'c';

void f1(int a, int b){

printf("a: ", a);

printf("b: ", b);

}

void main(){

const int a1 = 4, a2 = 5;

int a3;

char a4;

a3 = ((+1++2--3 + +4) \* -6 \* -1 + 3) / 7;

f1(a3, ((+1++2--3 + +4) \* -6 \* -1 + 3) / 7);

a4 = (a1 + a + -1 -b) \*a2 / 2;

f1(a4 + a4, c);

printf(a \* 3 - 1);

}

输出：a: 9 b: 9 a: 10 b: 99 5

Test4： Test3的错误版

错误内容：const int a = 2, b = 3;

const char c = 'c';

void f1(int a, int b){

printf("a: ", a);

printf("b: ", b);

}

void main(){

const int a1 = 4, a2 = 5;

int a3;

char a4;

a3 = ((+1++2--3 + +4) \* -6 \* -1 + 3) / 7;

f1(a3, ((+1++2--3 + +4) \* -6 \* -1 + 3) / 7);

a4 = (a5 + a + -1 -b) \*a2 / 2; //error

f1(a4 + a4, c);

printf(a6 \* 3 - 1); ///////error

}

输出：

line:17 error:1 未定义的函数或常量或变量

line:17 error:2 缺少)

line:17 error:3 非法的表达式的值

line:20 error:4 未定义的函数或常量或变量

line:20 error:5 缺少)

line:20 error:6 缺少;

line:20 error:7 缺少}

line:20 error:8 括号不匹配或出现未知符号

错误数：8

Test5: 数组索引嵌套

const int f = 5;

int a[10], b[10], c[10];

void main(){

int d[10],e[10];

a[5] = 3;

b[3] = 7;

c[7] = 9;

d[9] = 4;

e[4] = 88888888;

printf(e[d[c[b[a[f]]]]]);

}

输出：88888888

Test6：Test5的错误版：

const int f = 5;

int a[10], b[10], c[10];

void main(){

int d[10],e[10];

a[5] = 3;

b[3] = 7;

c[7] = 9;

d[9] = 4;

e[4] = 88888888;

printf(e[d[c[b[a[f]]]]); //缺少一个]

}

输出：

line:15 error:1 缺少]

错误数：1

Test7：全局与局部数组索引

const int a = 3, b = 4, c=5;

int d[10];

char e[10];

void main(){

int f[10], i, j, v;

char g[10];

i = 9;

j = 2;

scanf(v);

d[a] = v;

printf(d[a]);

scanf(v);

e[i] = v;

printf(e[i]);

scanf(v);

f[b] = v;

printf(f[b]);

scanf(v);

g[j] = v;

printf(g[j]);

}

输入：3 97 4 98

输出： 3 a 4 b

Test8：Test7的错误版

const int a = 3, b = 4, c=5;

int d[10];

char e[10];

void main(){

int f[10], i, j, v;

char g[10];

i = 9;

j = 2;

scanf(v);

d[] = v; //error

printf(d[a]);

scanf(v);

e[,] = v; //error

printf(e[i]);

scanf(v);

f[ffff] = v; //error

printf(f[b]);

scanf(v);

g[j] = v;

printf(g[j]);

}

输出:

line:11 error:1 非法的符号

line:11 error:2 非法的数组索引

line:16 error:3 非法的符号

line:16 error:4 非法的数组索引

line:20 error:5 未定义的函数或常量或变量

line:20 error:6 非法的数组索引

错误数：6

Test9:递归函数

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

if(n == 1){

printf(" move", n);

printf(" from", a);

printf(" to", c);

printf(" ");

}

else{

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

printf(" move", n);

printf(" from", a);

printf(" to", c);

printf(" ");

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

}

}

void main(){

int n;

scanf(n);

hanota(n, 'a', 'b', 'c');

}

输入：3

输出：move1 froma toc move2 froma tob move1 fromc tob move3 froma toc move1 fromb toa move2 fromb toc move1 froma toc

Test10：Test9的错误版

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

if(n == 1){

printf(" move", n);

printf(" from", a);

printf(" to", c);

printf(" ");

}

else{

hanota(); //error

printf(" move", n);

printf(" from", a);

printf(" to", c);

printf(" ");

hanota(b, a, c); //error

}

}

void main(){

int n;

scanf(n);

hanota(n, 'a', 'b', 'c');

}

输出：

line:10 error:1 参数不匹配

line:15 error:2 参数不匹配

错误数：2

### 2．测试结果分析

Test1主要针对语义和语法，其它测试程序均只针对语义，所以下面说明test1：

1. 字母下划线
2. 字符串
3. 有返回值的函数与无返回值的函数
4. 带（）的返回与直接return；
5. Int char 及其数组的常量（无数组）、变量、全局定义、局部定义等。
6. 空语句
7. 各种表达式，包括了++a-+c之类
8. 各种语句
9. 条件语句为单个表达式

10）sancf单个参数与两个参数

11）printf的参数的各种形式

12）return表达式

13）数组索引为表达式

14）带else的if和不带else的if

## 五．总结感想

这学期的编译器大作业中我收获很大，从理论知识到写代码的功力再到做事的态度。首先编译的理论知识在这个编译器上得到了很大的实现。其中理论课对课设帮助最大的应该是语法分析、语义分析部分。递归下降分析法、改写文法、向前预读等方法都得以运用到我的编译器中。其次是寄存器的分配和运行栈的设计，结合大二学习过的计算机组成，这让我对计算机的整体又有了更深的认识。另一方面，我的写代码的能力又得到了进一步的锻炼，在这次课设之前，从来没有写过5000+的代码，而这次的编译课设，从程序的框架设计到具体实现，都是一步一个脚印做出来的，比面向对象的课设还要多的代码量让我感到自己的能力得到了提升，较遗憾的是一开始没有构思好，使用了C语言，后来写到一半发现需要用到集合和字符串的地方太多，于是又换成了C++，使得我的代码风格很差，用了C++却没有使用面向对象。最后，编译课设也影响了我做事的态度，以前什么事情都喜欢往后拖，导致很多事情都无法保证质量完成。这次我的进度完全按照老师安排的日程来，最后完成整个项目时，发现只要按时保质保量完成每一个小目标，最终的目标自然就完成了。

这次的课设还是有遗憾的，在生成中间代码之后，我个人更想生成x86的汇编码，但自信心不够，怕自己做不出来，就选择了大二计组已经学习过的更熟悉的mips码。在整个课程完成后，我现在回过头来想想，如果当时选了x86，努力做的、多投入时间也应该能做好。

其实做事情也像编译中的递归下降分析法一样，分解成一个一个的基础部分，只要每一部分都完成好，整个问题也就迎刃而解了。