电子科技大学计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称：编译原理

电子科技大学教务处制表

**实验一、词法分析评审材料**

本程序实现了所有要求的功能

【操作步骤】

首先进入gpl-master目录，然后输入：

make

cd sample

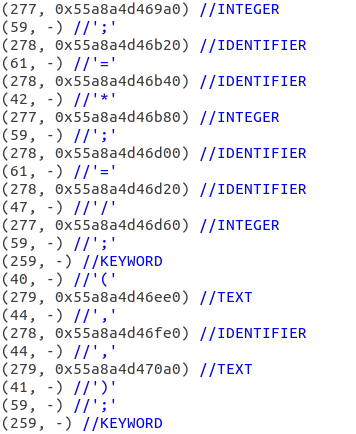
../gpl calc -lex

程序会在sample目录下输出calc.lex文件

其中就有需要输出的内容

1.输出单词（以二元式形式）

【运行结果】



按照老师的要求给出了输出的二元式形式，并且为了方便辨别识别到单词的具体内容，我在后面加了例如//KEYWORD //TEXT //’;’这样的注释，可以方便人阅读。

【代码简述】

主要是在yylex里面 按照老师给出的一些例子 然后用类似的方法写出要支持的功能

2.出错处理

【运行结果】

【用例1】非法组合出错

error1.gpl

!? //这是程序的内容 就一行!?

结果：



【说明】遇到!? !! !. !\*等非法组合时会报错

【用例2】非法字符出错

error2.gpl

我爱你 //这是程序的内容 就一行中文字符“我爱你”

结果：



【说明】遇到非法字符 如中文字符时会报错

2）部分代码与细节

【思路】在yylex识别单词的过程中

我主要在yylex中增加了以下代码处理异常：

if(character=='!')

{

concat();

getch();

if(character=='=')

{

concat();

printf("(%d, -) //NE\n", NE);

return NE;

}

else if(!digit()||!letter())

{

yyerror("invalid symbol combination");

exit(1);

}

else

{

retract();

printf("(%d, -) //'!'\n", '!');

return '!';

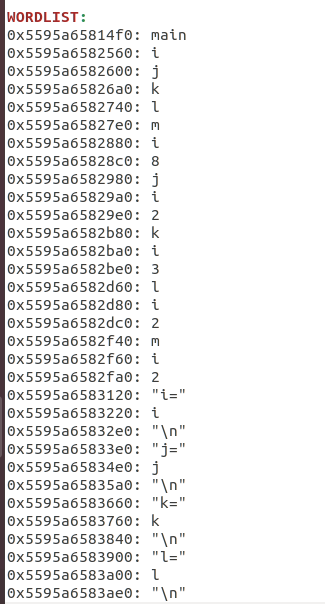
}

}

在识别’!’的同时，如果识别到!? !@ !#等非法组合，可以报错

3.符号表管理

【运行结果】



【思路】通过创建符号表结构体，创建全局变量，在yylex进行识别单词时，同时把相关地址和字面量存入表中，最后再输出表即可。

【详细细节与部分代码】

结构体定义如下：

typedef struct wordlist

{

int \*p;

char m[100];

}WORDLIST;

相关代码如下：

if(letter())

{

while(letter() || digit())

{

concat();

getch();

}

retract();

num=keyword();

if(num!=0)

{

printf("(%d, -) //KEYWORD\n", num);

return num; // return keyword

}

else

{

lexeme=malloc(strlen(token+1));

strcpy(lexeme,token);

yylval.string=lexeme;

wl[lenwl].p = (int\*)lexeme;

strcpy(wl[lenwl].m, token);

lenwl++;

printf("(%d, %p) //IDENTIFIER\n", IDENTIFIER, lexeme);

return IDENTIFIER;

}

}

if(digit())

{

while(digit())

{

concat();

getch();

}

retract();

lexeme=malloc(strlen(token+1));

strcpy(lexeme,token);

yylval.string=lexeme;

int \*p = malloc(4);

\*p = atoi(lexeme); // \*p is an int

wl[lenwl].p = (int\*)p;

strcpy(wl[lenwl].m, token);

lenwl++;

printf("(%d, %p) //INTEGER\n", INTEGER, p);

return INTEGER;

}

if(character=='"')

{

concat();

getch();

while(character!='"' && character!=EOF)

{

concat();

getch();

}

if(character==EOF)

{

printf("lex error: \" expected\n");

exit(1);

}

concat();

lexeme=malloc(strlen(token+1));

strcpy(lexeme,token);

yylval.string=lexeme;

wl[lenwl].p = (int\*) lexeme;

strcat(wl[lenwl].m, token);

lenwl++;

printf("(%d, %p) //TEXT\n", TEXT, lexeme);

return TEXT;

}

最后再main函数中输出词汇表即可：

if(mode==1)

{

yylex();

printf("\n\n\nWORDLIST:\n");

//printf("%p, %s", wl[0].p, wl[0].m);

int i;

for(i=0;i<lenwl;i++)

{

printf("%p: %s\n", wl[i].p, wl[i].m);

}

exit(0);

}

4.问题一：你的代码用于存放3种符号的值的代码位置在哪里

主要是通过（3）中提到的词汇表结构体来存储的

相关代码如下，在gpl.y中：

if(letter())

{

while(letter() || digit())

{

concat();

getch();

}

retract();

num=keyword();

if(num!=0)

{

printf("(%d, -) //KEYWORD\n", num);

return num; // return keyword

}

else

{

lexeme=malloc(strlen(token+1));

strcpy(lexeme,token);

yylval.string=lexeme;

wl[lenwl].p = (int\*)lexeme;

strcpy(wl[lenwl].m, token);

lenwl++;

printf("(%d, %p) //IDENTIFIER\n", IDENTIFIER, lexeme);

return IDENTIFIER;

}

}

if(digit())

{

while(digit())

{

concat();

getch();

}

retract();

lexeme=malloc(strlen(token+1));

strcpy(lexeme,token);

yylval.string=lexeme;

int \*p = malloc(4);

\*p = atoi(lexeme); // \*p is an int

wl[lenwl].p = (int\*)p;

strcpy(wl[lenwl].m, token);

lenwl++;

printf("(%d, %p) //INTEGER\n", INTEGER, p);

return INTEGER;

}

if(character=='"')

{

concat();

getch();

while(character!='"' && character!=EOF)

{

concat();

getch();

}

if(character==EOF)

{

printf("lex error: \" expected\n");

exit(1);

}

concat();

lexeme=malloc(strlen(token+1));

strcpy(lexeme,token);

yylval.string=lexeme;

wl[lenwl].p = (int\*) lexeme;

strcat(wl[lenwl].m, token);

lenwl++;

printf("(%d, %p) //TEXT\n", TEXT, lexeme);

return TEXT;

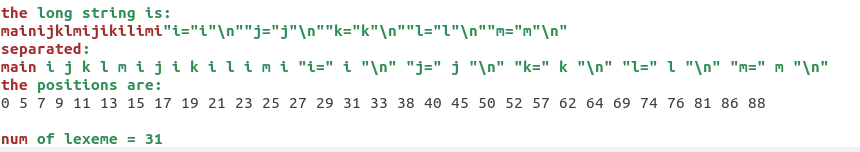
}

（5）实现一个存放字面量的字符串

已经实现

主要是用一个全局变量ls实现的

打印效果如图



（6）实现了哈希表

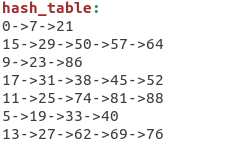
主要就是使用数据结构里面哈希表相关的指示

定义哈希表的大小为7

hash函数为：data % 7

这样就可以尽可能多的看到哈希表的效果

打印效果如图：



实验一所有功能都已实现

**实验二、语法分析实验评审说明**

实现了所有功能

1.输出语法树

【描述】在进行分析后，可以生成名为draw.dot的dot脚本语言文件，该文件可在linux下通过终端的dot命令生成相应png、pdf等文件，里面的内容就是语法树。

【代码简述】

主要是在前面的产生式定义的地方 每个产生式对应的动作里面生成相应dot代码，并且写入draw.dot文件。并且自己模拟实现了一个符号栈。

【操作步骤】

（1）进入gpl-master目录

（2）输入命令:

make

（3）进入sample目录

（4）输入命令:

../gpl calc -syn

（5）输入命令：

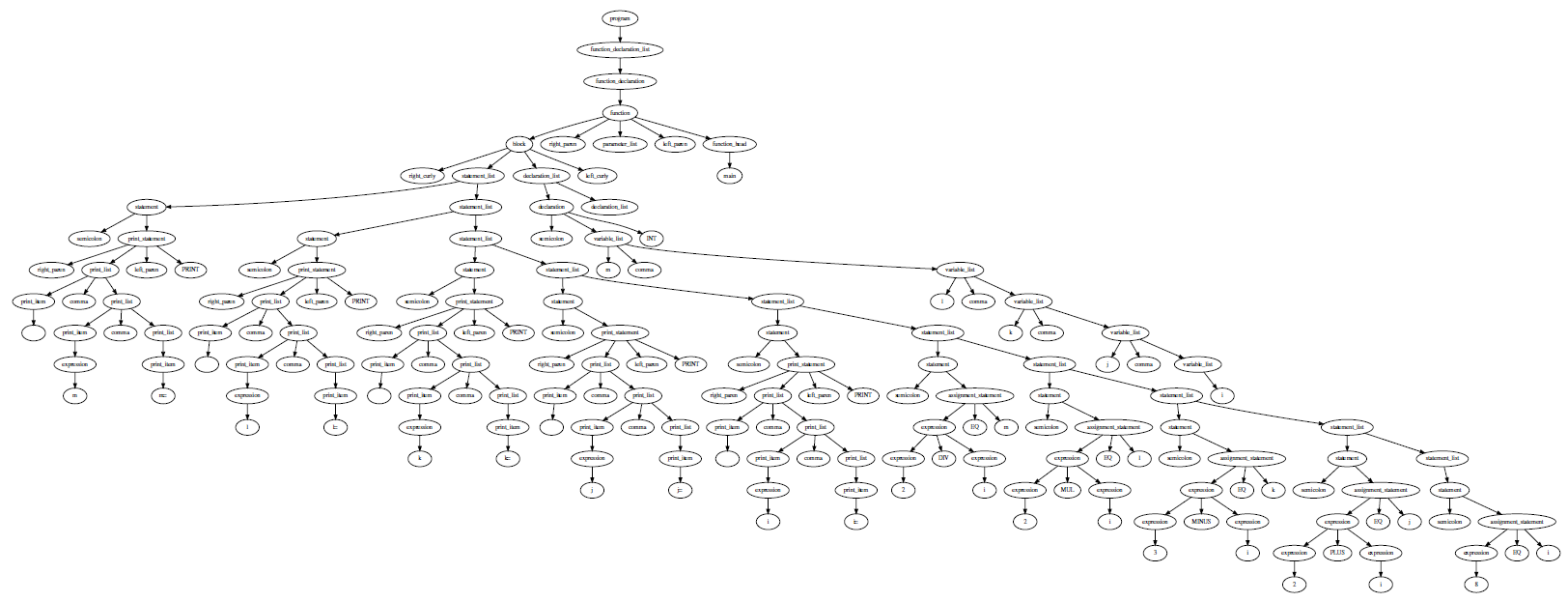
dot -Tpdf draw.dot -o draw.pdf

\*\*\*注意：dot命令需要先安装graphviz，安装方法为：

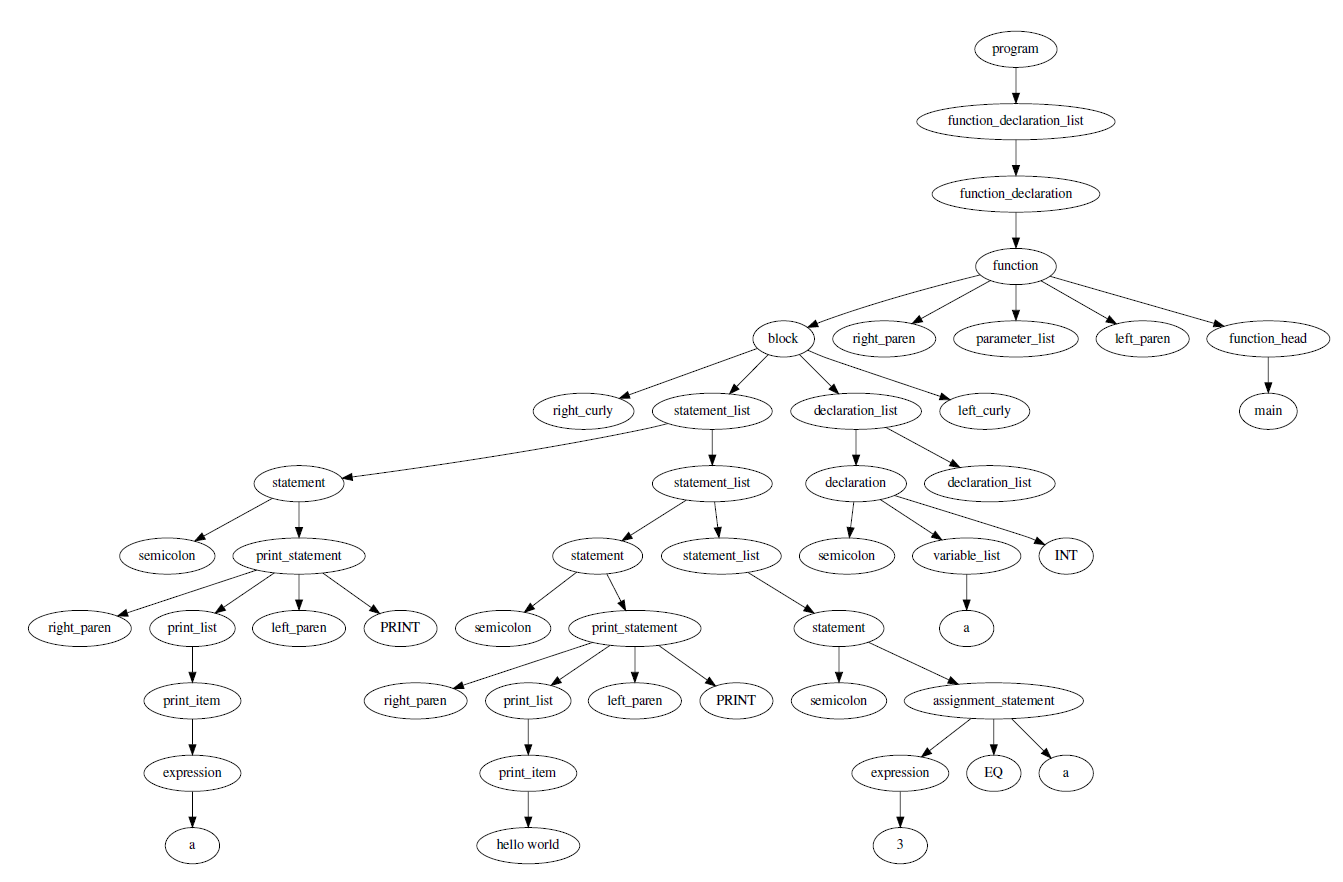
sudo apt-get install graphviz graphviz-doc

【结果说明】

完成上述操作后，在sample文件夹中，会生成“calc.syn”、“draw.dot”和“draw.pdf”三个文件。打开draw.pdf文件，就会呈现出相应的语法树。如下图：



由于这个语法树太大了可能看不清，我编写了一个test.gpl在sample目录中。用同样的步骤生成的语法树如图：



可见，该语法树正确，美观，展示效果好，完成了输出语法树的任务。

【特别说明】

（1）在linux下使用dot命令需要先安装graphviz，安装方法为，输入命令：

sudo apt-get install graphviz graphviz-doc

即可

（2）我所生成的语法树跟书上的不一样，是从右边开始画节点的。就是说，这棵语法树的边缘是从右往左看的。这个不影响结果的，生成的语法树是完全正确的。

（3）看图中的细节。

比如parameter\_list之后就没有生成式子了。这就说明parameter\_list是空的。因为在.l文件中相应的产生式是空的，所以我就没有在下面添加节点。

再比如：语言中的‘（’，‘）’，‘，’等字符我在语法树中分别用的left\_paren,right\_paren,comma等英文表示，这个不影响的。

再比如有些节点内容是空的，这是因为他们可能是’\n’，所以没有显示出来，这个也不影响的。

（4）我生成了一些语法树，可以直接看我生成好的pdf文件。他们放在评审材料的“语法树”文件夹中（有if、calc、while、test1、test2五个文件）。另外，还有他们对应的dot文件，在“语法树”文件夹中的“dot文件”文件夹

2.出错处理

【说明】实现了出错处理，对不同类型的错误，有相应类型的错误提示，并且指出了出错位置，完美完成了要求。

【代码说明】在产生式带有error规约的地方，添加了打印错误的代码。

【操作步骤】

【操作步骤】

（1）进入gpl-master目录

（2）输入命令:

make

（3）进入sample目录

（4）输入命令:

../gpl error1

【结果说明】

为了测试不同类型的错误，我编写了三个测试样例，分别为error1.gpl、error2.gpl、error3.gpl。他们覆盖了所有的语法分析中的错误。他们的内容分别为：

1. main(   // line 1
2. main()
3. {
4. int a;
5. a = 1;
6. a =  ;  // line 5
7. }
8. main()
9. {
10. int a;
11. a = 1\*\*2;   //line 4
12. }

运行gpl的截图分别为：







通过结果可以看出，结果完全正确。

3.符号表管理

【说明】老师课上说，区分同名全局变量和不同函数中同名的局部变量，这个功能已经实现，只需要自己写出测试用例即可。

【操作步骤】

（1）进入gpl-master目录

（2）输入命令:

make

（3）进入sample目录

（4）输入命令:

../gpl symbol1

../gal symbol1

../gvm symbol1

\*注意：可能需要在输入../gal symbol1之前，需要手动symbol1.gal中前面输出的符号表删去（因为这部分代码本来是词法分析里面的，但是我一起带进来了），只留下gal代码。

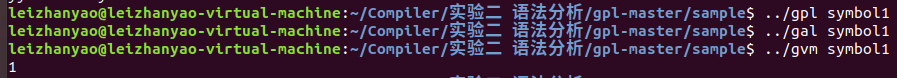
【测试用例1】

symbol1.gpl

1. int i;
2. main()
3. {
4. i = 1;
5. print(i,"\n");
6. }

这个例子主要是测试main函数内部能否识别外部的全局变量。

执行结果如图：



正确

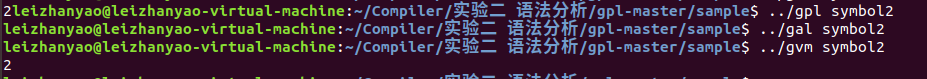
【测试用例2】

symbol2.gpl

1. int a;
2. main()
3. {
4. func();
5. }
6. func()
7. {
8. a = 2;
9. print(a);
10. print("\n");
11. return 0;
12. }

这个例子主要是测试func函数内部能否识别外部的全局变量。

执行结果如图：



正确

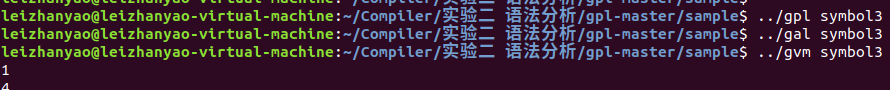
【测试用例3】

symbol3.gpl

1. main()
2. {
3. int a;
4. a = 1;
5. print(a,"\n");
6. func();
7. }
8. func()
9. {
10. int a;
11. a = 4;
12. print(a,"\n");
13. return 0;
14. }

这个例子主要是测试程序能否区分不同函数间的同名变量。

执行结果如图：



正确

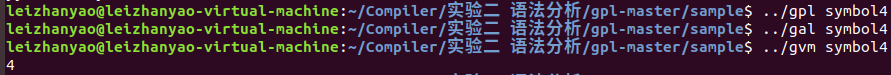
【测试用例4】

symbol4.gpl

1. main()
2. {
3. int a;
4. a = 4;
5. funct(a);
7. }
8. funct(a)
9. {
10. print(a,"\n");
11. }

这个例子主要是测试程序能否区分同名的实参和形参的传递。

执行结果如图：



正确

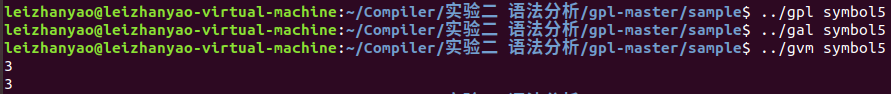
【测试用例5】

symbol5.gpl

1. main()
2. {
3. int a;
4. a = 3;
5. print(a,"\n");
6. func(a);
7. print(a,"\n");
8. }
9. func(a)
10. {
11. a = 5;
12. return 0;
13. }

这个例子主要是测试程序能否区分同名的实参和形参的传递,以及函数不会对传入的参数值进行改变的功能。

执行结果如图：



正确

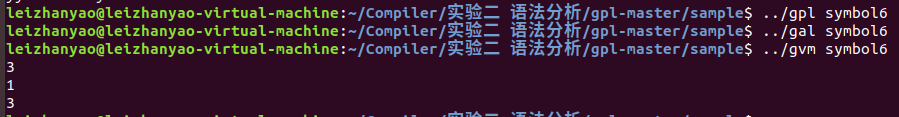
【测试用例6】

symbol6.gpl

1. main()
2. {
3. int a;
4. a = 3;
5. func(a);
6. print(a,"\n");
7. }
9. func(c)
10. {
11. int a;
12. print(c,"\n");
13. a = 1;
14. print(a,"\n");
15. }

这个例子主要是综合测试不同作用域的同名变量处理和传递参数的处理。

执行结果如图：



正确

综上所述，程序很好的实现了符号表管理。

4.问题一：你的代码中是如何区分同名的全局变量和不同函数中同名的局部变量的？

主要是通过作用域切换来实现的。

程序中有一个全局变量LocalScope，还有局部符号表LocalSymbolTable和全局符号表GlobalSymbolTable。当它的值为1时，表示当前作用域在局部作用域，这时只对局部符号表LocalSymbolTable操作。当它的值为0时，表示作用域在全局，此时对全局符号表进行操作GlobalSymbolTable。

通过LocalScope实现作用域切换，然后用LocalSymbolTable和GlobalSymbolTable分别实现不同作用域的符号管理，就不会产生冲突了。

5.问题二：是否提供了足够的测试用例（至少5个）证明你的程序实现了相关功能？

提供了足够的测试用例。

首先，在输出语法树部分，有if、while、calc、test1、test2五个测试用例（见测试用例文件夹和语法树文件夹）

其次，在出错处理部分，有error1，error2，error3三个测试用例（见测试用例文件夹和上面的描述部分）

最后，在符号表管理部分，有symbol1、symbol2、symbol3、symbol4、symbol5、symbol6共六个测试用例（见测试用例文件夹和上面的描述部分）

6.问题三：是否能够详细说明你的代码中最具有特色或个性化的功能的实现方法？

我的代码最有特色的我认为是语法树的生成那里。因为生成的语法树正确、清晰，而且是图形的形式，布局非常美观。

这里主要是用到一个很方便的绘制图形的工具——graphviz。

主要思想是，利用gpl，在语法分析的过程中产生graphviz的语言，生成一个draw.dot文件。当gpl分析完后，再执行一个dot命令即可生成对应的图形文件。好处是，比较方便，而且图形的布局、排版非常简洁美观。

由于语法分析中的归约过程，也就是语法树形成的过程。所以我在每个产生式的后面的动作里都会有生成相关graphviz语言的命令。而且这里我还定义了一个新的结构体MYVSP来辅助进行操作。这个MYVSP其实就是一个类似于符号栈的东西，所以我的代码里面还有栈相关的操作。

**实验三、中间代码生成评审材料**

所有功能已实现

1.输出三地址码

（1）输出中间代码

【操作步骤】

进入gpl-master目录，打开终端

1. make

2. cd sample

3. ../gpl calc -tac

【测试用例】

calc.gpl

func.gpl

if.gpl

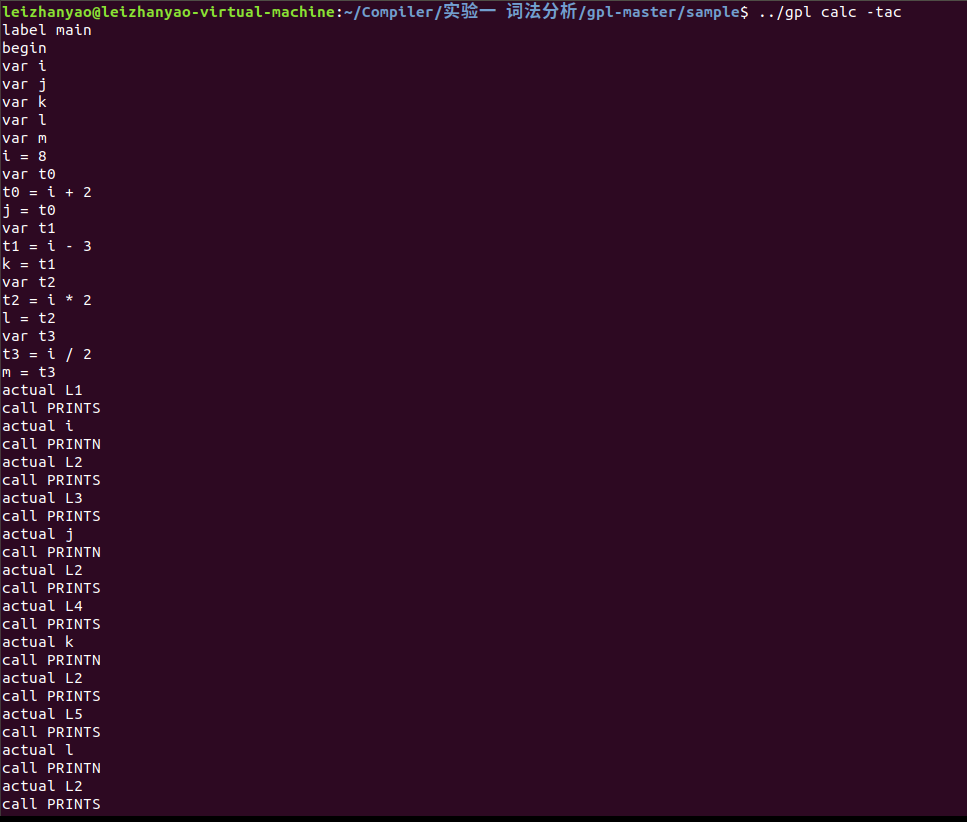
while.gpl

for.gpl

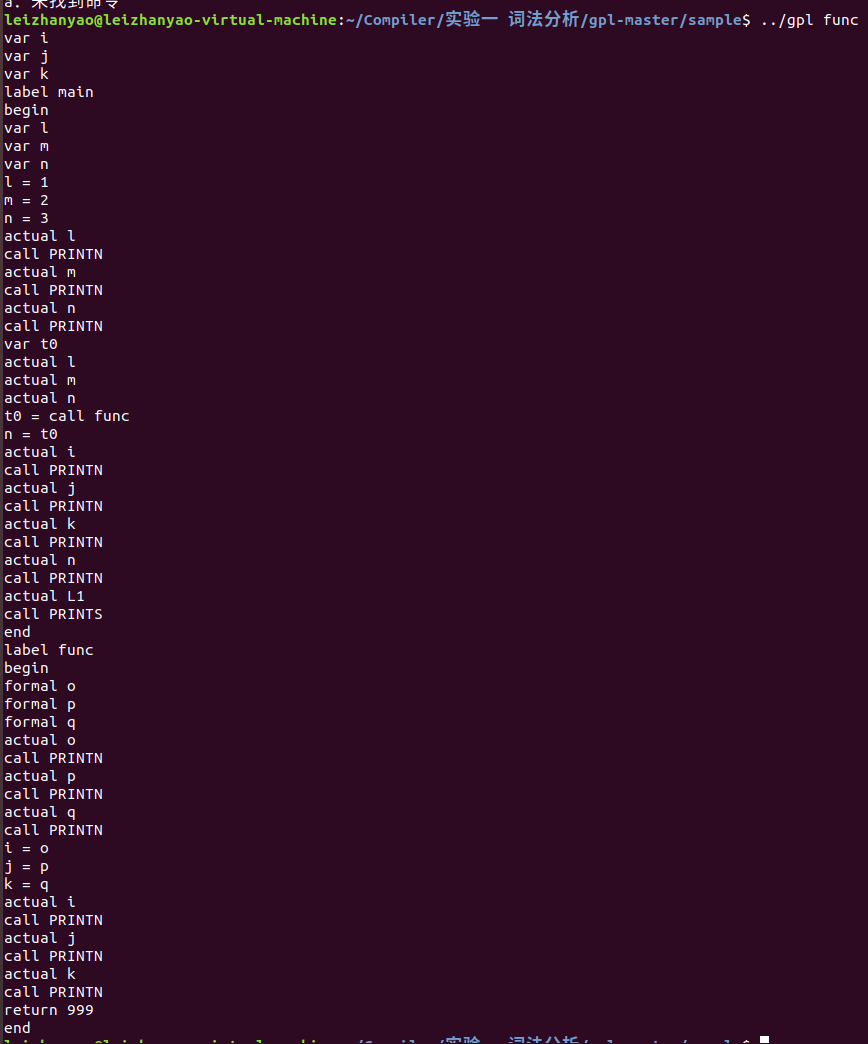
【结果】

输入上述命令后，会输出tac。以上测试用例的结果分别如下面的图所示：

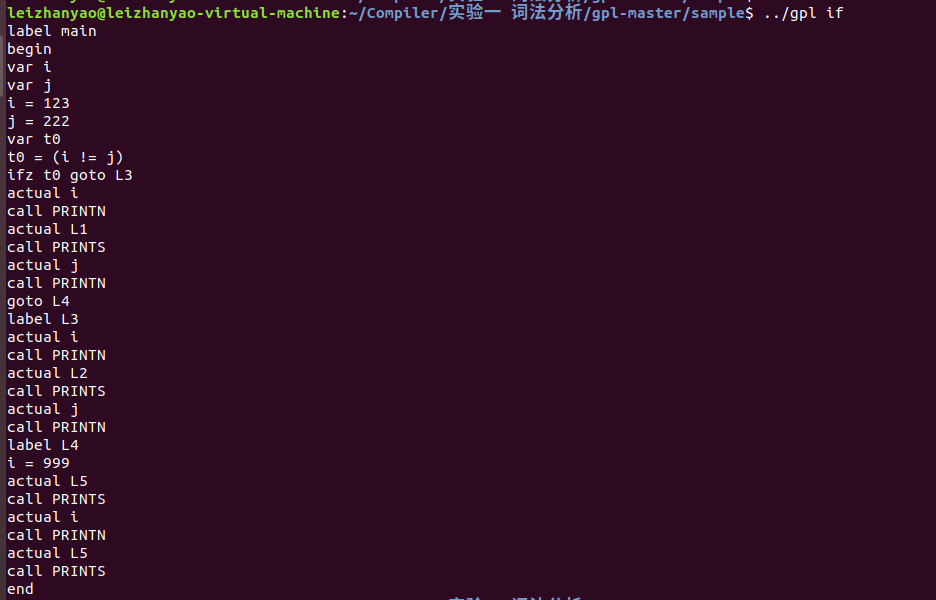
calc的：



func的：



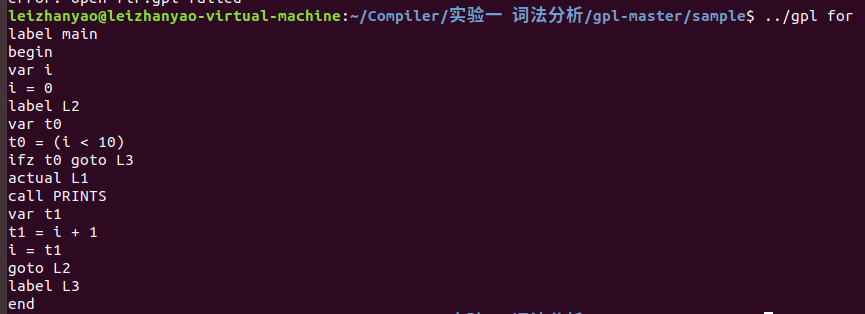
if的：



while的：



for的：



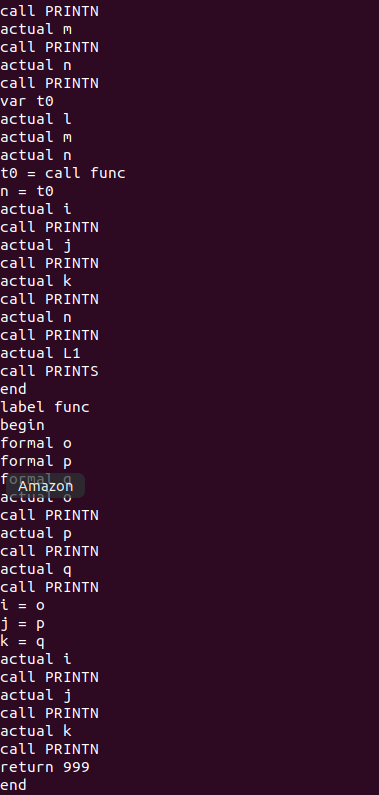
经过比对，发现中间代码正确，功能完成。

【分析】这个功能其实已经在老师给的代码中实现。我发现主要是在语义子程序里面实现的tac的创建和连接等。主要是在.y文件中的产生式后面的函数操作里面完成的。

（2）按照源代码中顺序说明三地址码中实参

【操作说明】对func.gpl使用 结果如下

【运行结果】



可见 对于源代码中的print func等函数的调用 其实参的传递都是按照源代码顺序的（如里面的actual o、actual p、actual q等）

说明结果正确，功能实现。

【代码简述】

主要是872~900行的do\_call函数。这个函数主要负责函数调用时的tac生成。原来的代码是反过来压栈的，我们只需要把它改成按源代码顺序的压栈即可。

（3）支持for语句

【测试用例】

for.gpl

main()

{

for(int i=0;i<10;i=i+1)

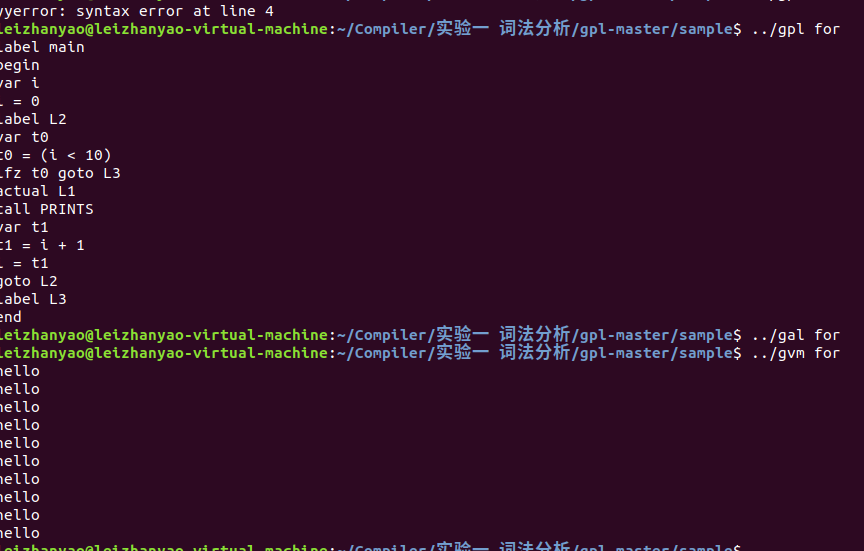
{

print("hello\n");

}

}

【结果】



如图所示 可以生成正确tac 并且可以成功执行 说明for语句支持成功

【代码简述】

其实很类似。直接对比老师给出其他的while if等语句的支持的代码，照葫芦画瓢就可以。主要是三点，一是在前面定义token for，二是修改中间的产生式，使其支持for语句，最后是添加相关处理的函数。如do\_for（见代码的983~991行）

（4）支持变量（包括局部变量、全局变量）的初始化操作。

【测试用例】

init.gpl

int a=1;

main()

{

int b=2;

print(a,"\n",b,"\n");

}

【结果】



从结果可见，已经支持了初始化操作。（老师原本的代码没有支持初始化）

【代码简述】

利用已有的非终结符和产生式，在原有的产生式里添加支持初始化的代码即可。

主要是这里：

declaration : INT variable\_list ';'

{

$$=$2;

}

|

INT IDENTIFIER '=' expression ';'

{

TAC \*var=declare\_var($2);

$$=join\_tac(var,do\_assign(getvar($2),$4));

}

把declaration加一个产生式就好了，没有加其他的非终结符。

2.出错处理

【测试用例】

error1.gpl

main()

{

int a;

a == 3;

}

【运行结果】



【代码简述】

这里明显是a=3写成了a==3。这句话本身没有什么语法问题，也是一个表达式，但是在语义上有问题。代码主要是在yylex里面加的，里面有些本来识别到错误，但是没有出错提示，而是直接退出系统的，在前面加一句报错即可。

3.问题一：你的三地址码中是如何表示变量的初始化操作的？

如图：



可以看到，这里的初始化的变量和一般的先定义，再赋值的变量的三地址码一样，都是类似于“var a a = 1”这种类型的两句。其实对于初始化变量的处理，只需要把他看作是一句定义加一句赋值语句就好，然后对于tac的处理就是先处理定义语句，再处理赋值语句。

3.问题二：是否提供了足够的测试用例（至少5个）证明你的程序实现了相关功能。

有足够的测试用例

在TAC生成中有calc.gpl、func.gpl、if.gpl、while.gpl、for.gpl共5个测试用例

在说明实参中有func.gpl一个测试用例

在for语句支持中有for.gpl一个测试用例

在出错处理中有error1.gpl一个测试用例

所有测试用例和相应生成的其他文件都放在评审材料的“测试用例文件夹中”

4.问题三：是否能够详细说明你的代码中最具特色或个性化的功能的实现方法。

我认为最具特色的功能就是在初始化变量的支持那里。起初我以为这个功能挺难实现的，但是没想到只加了几行就实现了。

declaration : INT variable\_list ';'

{

$$=$2;

}

|

INT IDENTIFIER '=' expression ';'

{

TAC \*var=declare\_var($2);

$$=join\_tac(var,do\_assign(getvar($2),$4));

}

在declaration这个终结符对应的产生式中，他前面的第一个产生式就是传统的不带初始化的定义的产生式。而后面那个产生式是我加的。首先按照C语言中初始化的模式写出产生式。然后在语义子程序中加入定义变量和赋值的TAC操作即可。

**实验四、目标代码生成评审材料**

所有功能都已实现

1.输出目标代码

（1）对于语义正确的输入串，输出正确的目标代码（汇编代码）。

【测试用例】

calc.gpl

func.gpl

if.gpl

while.gpl

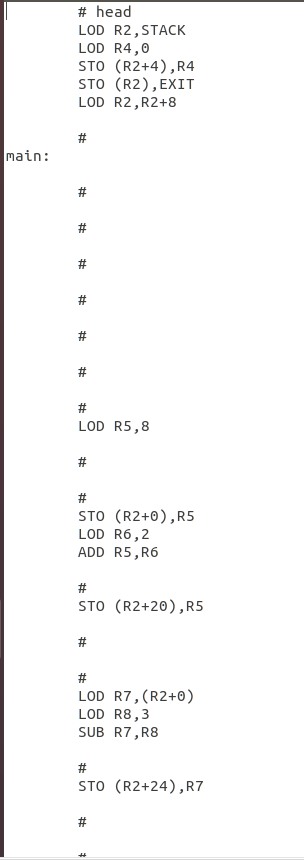
for.gpl

init.gpl

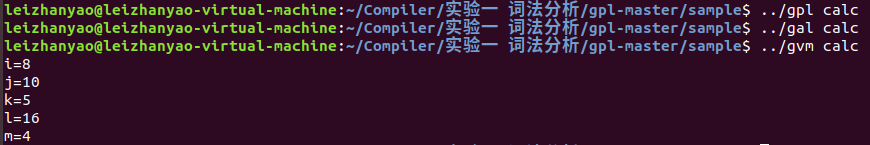
共六个

执行结果分别为

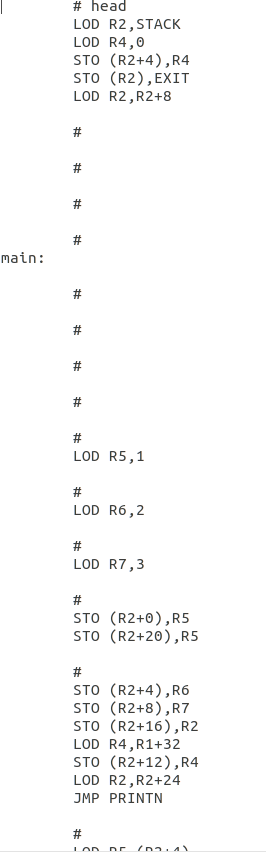
1）calc.gal为：



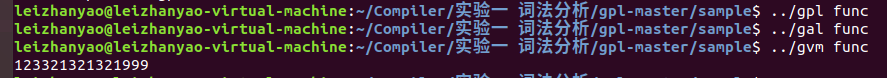
程序执行结果为



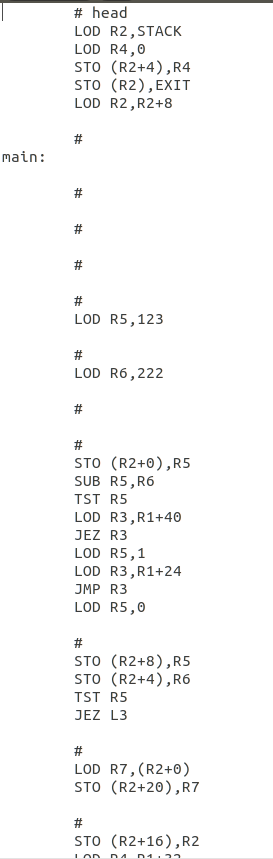
2）func.gal为：



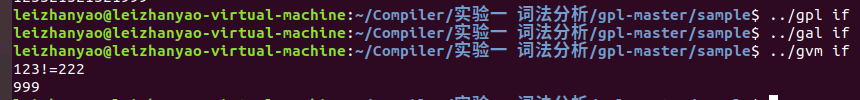
程序执行结果为



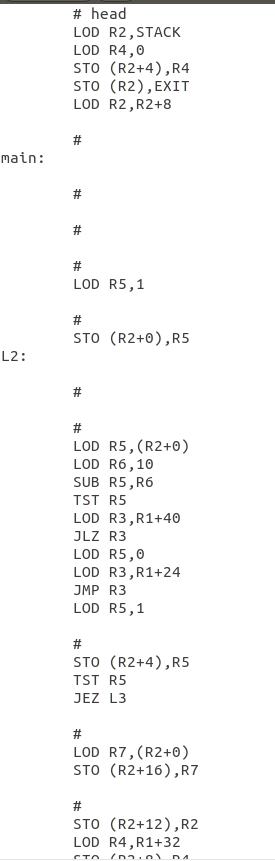
3）if.gal为



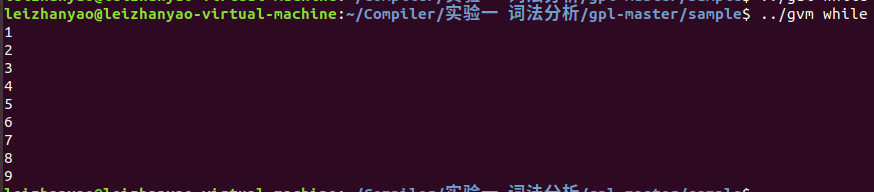
程序执行结果为



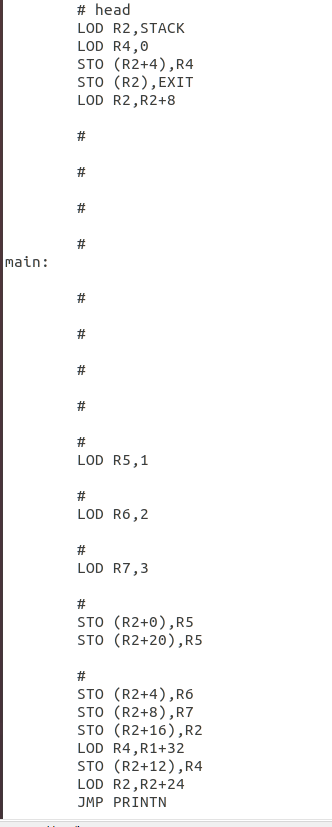
4）while.gal为



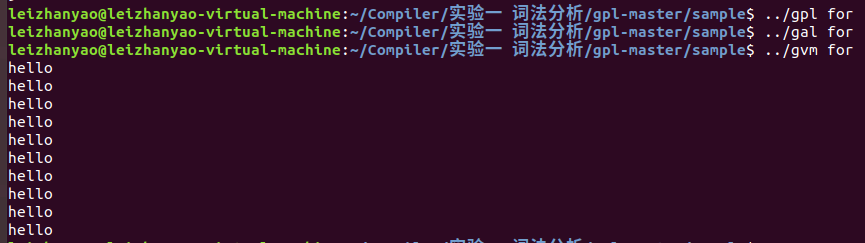
程序执行结果为



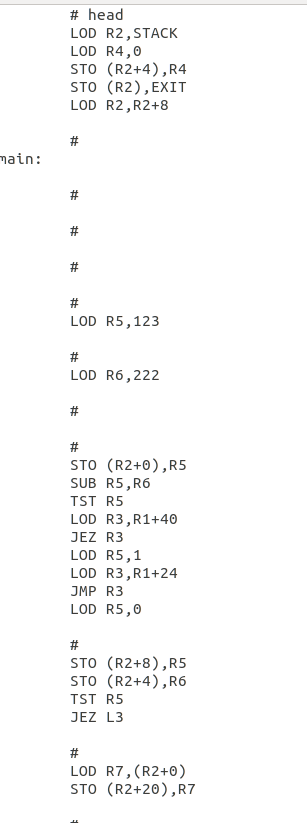
5）for.gal为



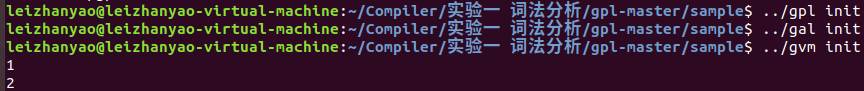
程序执行结果为



6）init.gal为



程序执行结果为



\*注：所有测试用例的完整版在评审材料的“测试用例”中有。

根据以上结果，证明目标代码生成正确，程序可以正常运行，实验成功。

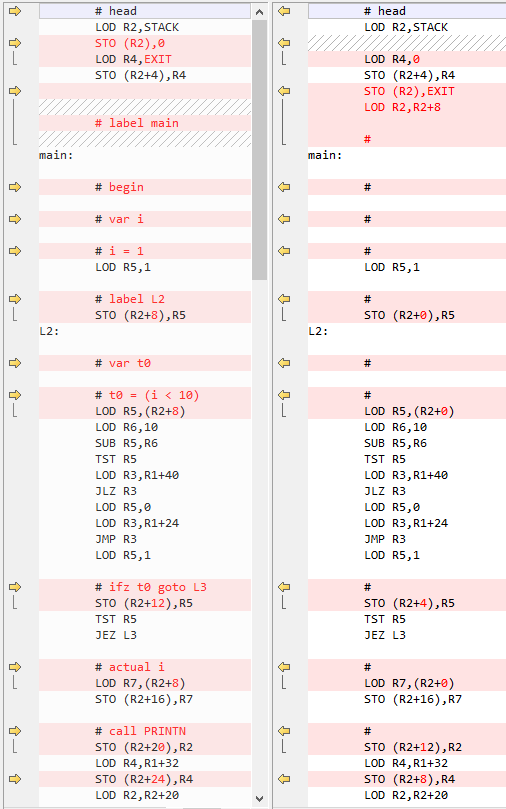
【代码简述】老师说这个已经实现了，只要前面实验三的中间代码生成正确，这里就一定正确。

（2）按照活动记录中各个字段的先后顺序按照课堂上的顺序排列。

【测试用例】

以while.gal为例

【执行结果】



对比前后生成结果，左边是原本的gal，右边是新的gal。可以看出，程序的执行结果不变，而只是LOD和STO指令中存储器操作数的一些偏移量发生了变化。这里实际上就是内存中他们的存放位置不同了。这就是活动记录顺序的改变。实验成功。

【代码简述】

主要是在1348行开始的cg\_code函数以及下面和它有关的一些列函数中改的。老师给的代码的顺序是“参数区、动态连接、返回地址、变量区”，我们需要改成“返回地址、动态连接、参数区、变量区”。只需要在上述函数里面找到相应内容的位置，调换他们的顺序即可。

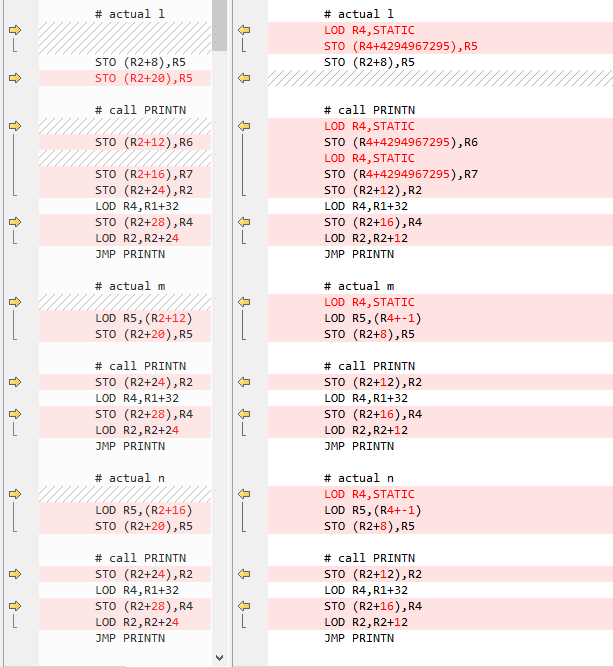
具体可见代码的1348~1628行。

（3）按照源代码中顺序传递实参的值

【测试用例】

以func.gal为例

【执行结果】



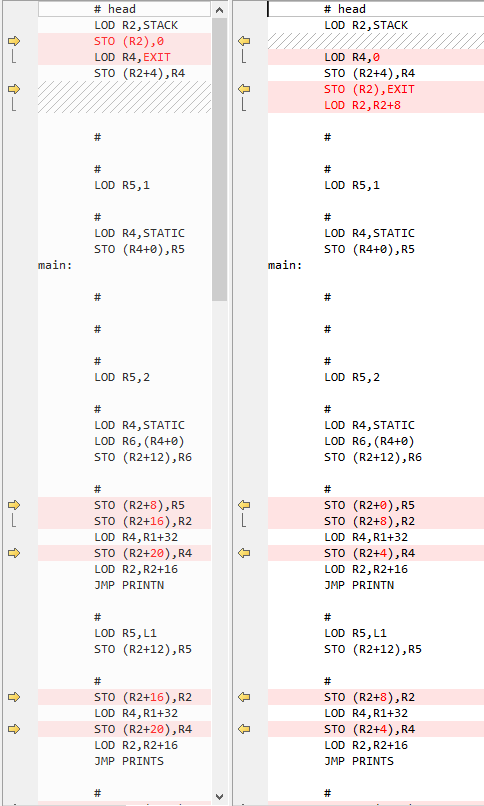
通过对比前后代码发现，参数传递顺序确实发生了改变，而程序执行结果一样，成功。

【代码简述】实际上，只要在实验三里面实现了改变传参顺序，这里的目标代码的传参顺序就一起改变了，因为目标代码就是直接按照中间代码翻译的。因为我实验三中，中间代码的顺序改好了，所以这里就直接是对的，没有什么需要修改代码的地方。

（4）支持变量（包括局部变量、全局变量）的初始化操作

【测试用例】以init.gal为例

【结果描述】



通过前后对比，发现最上面那里，是变量的初始化操作，gal代码发生了一些变化。结合程序执行和gal代码的逻辑，说明结果正确，实验成功。

【代码简述】这个也是需要一点代码修改的。具体也是1348~1628行中改的。

3.问题一：你的目标代码是如何访问参数、局部变量和全局变量的。

我们知道函数的活动记录中有“参数区”和“变量区”这两个区域，而且“变量区”实际上还分为局部变量区和全局变量区的。所以只需要在这里面找就可以了。具体来说，参数是通过bp+偏移量找到的，局部变量=局部变量区首地址+偏移量找到的，全局变量=全局变量区首地址+偏移量找到的。

4.问题二：是否提供了足够的测试用例（至少5个）证明你的程序实现了相关功能？

提供了足够的用例。

输出目标代码中提供了calc.gpl、func.gpl、if.gpl、while.gpl、for.gpl、init.gpl共六个用例

字段顺序、传参顺序、变量初始化又分别以while、func、init三个用例进行详解，同时还使用了以前代码生成的gal文件进行对比。

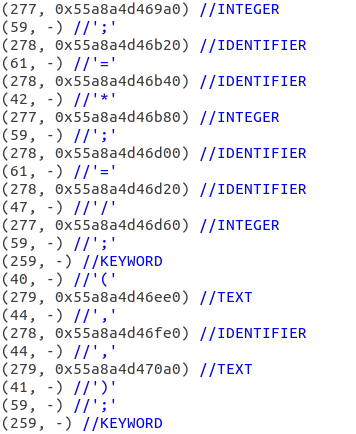
5.问题三：是否能够详细说明你的代码中最具特色或个性化的功能的实现方法？

有特色我觉得还是支持了for语句吧。因为我在实验三中加入了支持for语句的功能，就相当于重新按照lex的产生式的编写方法实现了for的tac，从而进一步实现for语句的支持。这个功能实现的重点就在于一定要了解lex特有部分的编写方法才能写得出来。这也加深了我对编译原理的理论知识的理解。

五、加分项申请

1.新增功能点

**功能点（1）**语法分析中，在输出二元式中，额外输出了相应匹配到的符号是什么，如图：



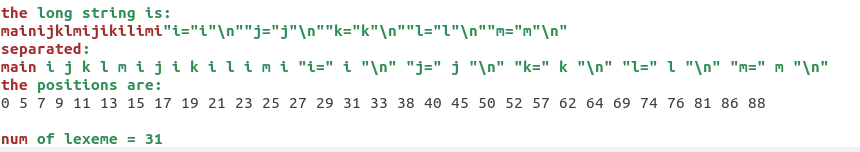
原本只要求输出(277, 0x55a8a4d469a0)

我还增加了在后面输出了一个“//INTEGER”，表明我匹配到的这个东西是一个INTEGER。

同理 其他的单词 诸如“；”、“=”、“\*”、“IDENTIFIER”、“TEXT”、“KEYWORD”，我都在相应的二元式后进行了标注。

如果不进行标注，只看前面的二元式，很难知道这究竟是什么，也不好检查代码是否正确。如果输出了后面的内容，不但可以方便检查代码正确性，还能清晰地看到词法分析器匹配单词的过程。

**功能点（2）**语法分析中，在附加问题2中，可以输出单词的个数，以及分割后的长单词串，如图：

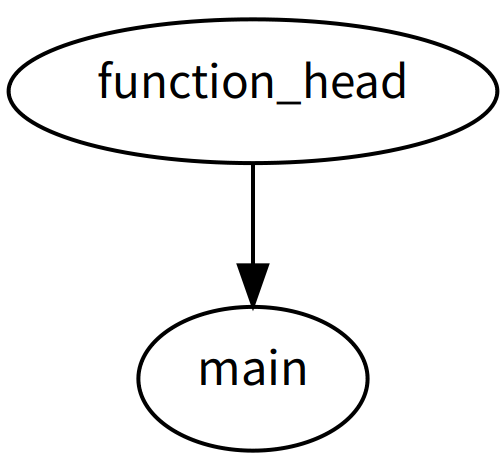


原本只要求输出单词串，不要求输出单词个数和分割后的单词串，我这里输出了，更方便看单词串的内容和检查代码是否正确。

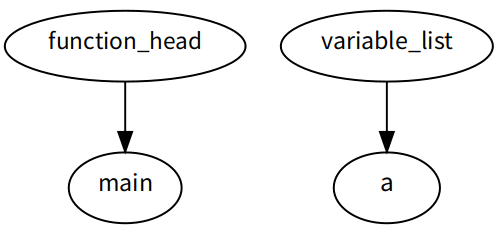
**功能点（3）**语法分析中，可以按照步骤规约的步骤生成每次的规约图，这样就可以看到整棵树规约的过程，有利于理解规约过程。

例如：

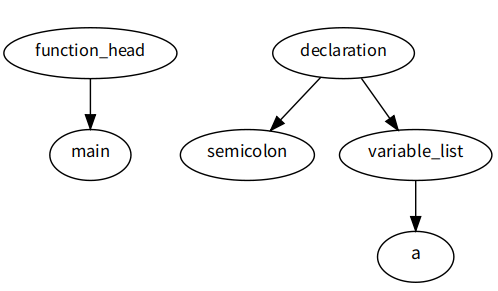
1）



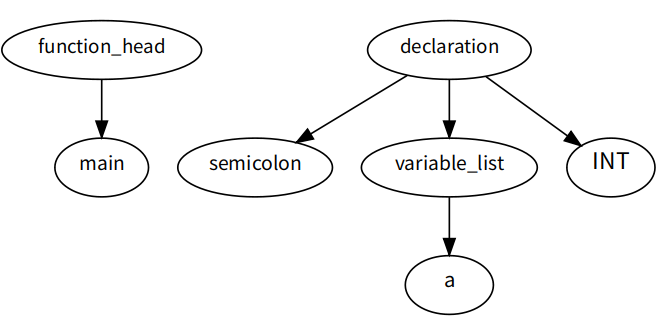
2）



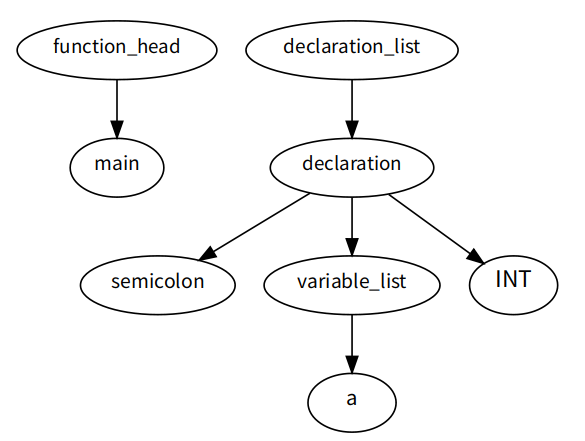
3）



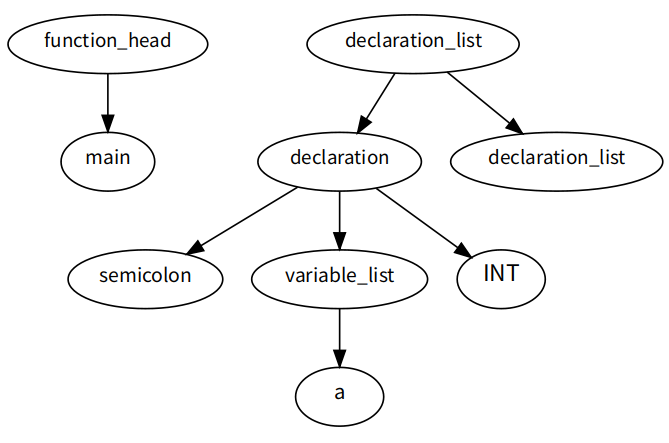
4）



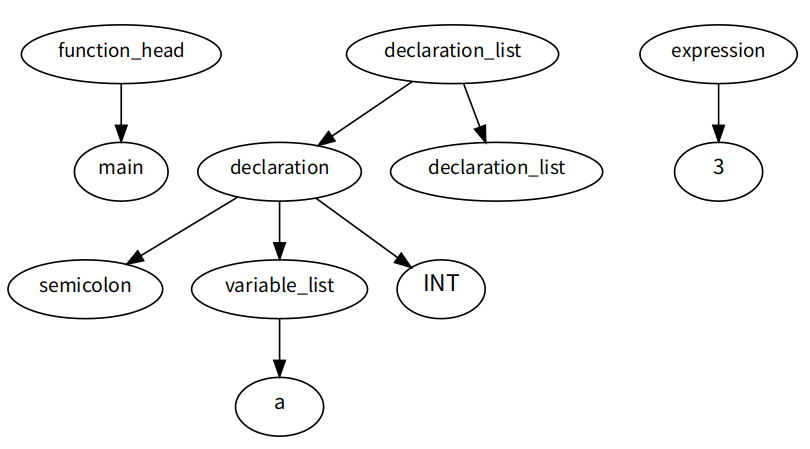
5）



6）



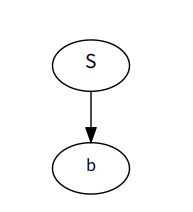
7）



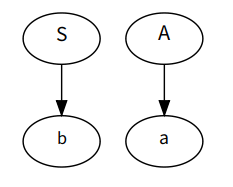
。。。以此类推，最后可以生成一棵语法树。

由于这个树太大了，我做了一个另外的例子，可以输出规约的每一个步骤，来看到语法树的生成过程：

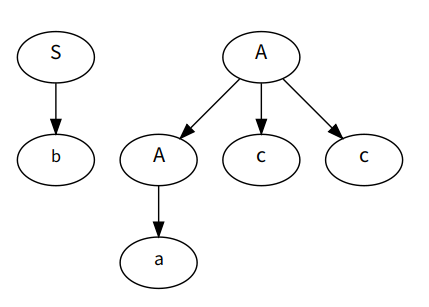
1）



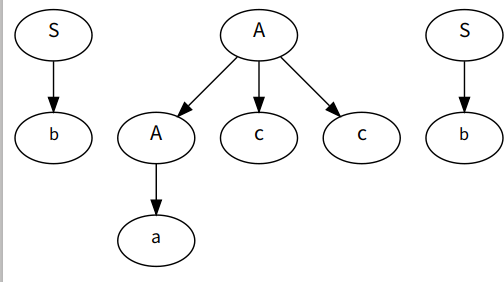
2）



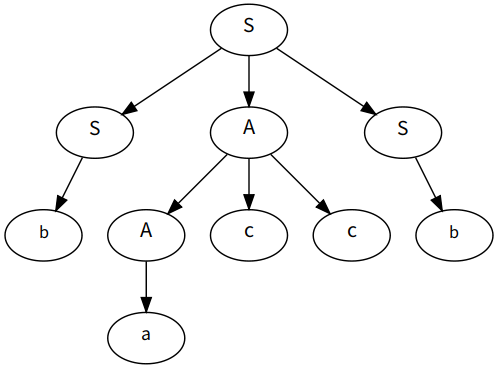
3）



4）



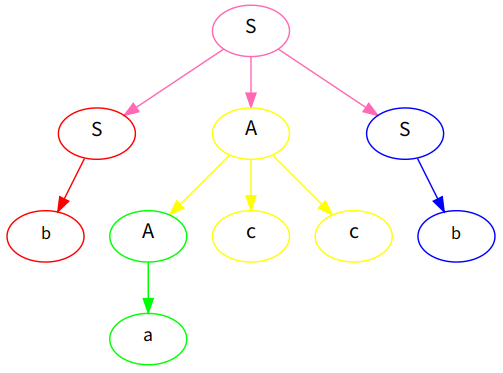
5）



所以我觉得，这个还是很形象直观的。

**功能点（4）**可以对每次规约的点和边，用不同的颜色，这样就可以很好地看出语法树的规约步骤以及结构了。

如图：



可以很好地看出语法树的规约过程和结构