

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：

```
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl calc -tac
label main
begin
var i
var j
var k
var l
var m
i = 8
var t0
t0 = i + 2
j = t0
var t1
t1 = i - 3
k = t1
var t2
t2 = i * 2
l = t2
var t3
t3 = i / 2
m = t3
actual L1
call PRINTS
actual i
call PRINTN
actual L2
call PRINTS
actual L3
call PRINTS
actual j
call PRINTN
actual L2
call PRINTS
actual L4
call PRINTS
actual k
call PRINTN
actual L2
call PRINTS
actual L5
call PRINTS
actual l
call PRINTN
actual L2
call PRINTS
```

func 的:

```
a. 未找到命令
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl func
var i
var j
var k
label main
begin
var l
var m
var n
l = 1
m = 2
n = 3
actual l
call PRINTN
actual m
call PRINTN
actual n
call PRINTN
var t0
actual l
actual m
actual n
t0 = call func
n = t0
actual i
call PRINTN
actual j
call PRINTN
actual k
call PRINTN
actual n
call PRINTN
actual L1
call PRINTS
end
label func
begin
formal o
formal p
formal q
actual o
call PRINTN
actual p
call PRINTN
actual q
call PRINTN
i = o
j = p
k = q
actual i
call PRINTN
actual j
call PRINTN
actual k
call PRINTN
return 999
end
```

if 的:

```

leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl if
label main
begin
var i
var j
i = 123
j = 222
var t0
t0 = (i != j)
ifz t0 goto L3
actual i
call PRINTN
actual L1
call PRINTS
actual j
call PRINTN
goto L4
label L3
actual i
call PRINTN
actual L2
call PRINTS
actual j
call PRINTN
label L4
i = 999
actual L5
call PRINTS
actual i
call PRINTN
actual L5
call PRINTS
end

```

while 的:

```

leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl while
label main
begin
var i
i = 1
label L2
var t0
t0 = (i < 10)
ifz t0 goto L3
actual i
call PRINTN
actual L1
call PRINTS
var t1
t1 = i + 1
i = t1
goto L2
label L3
end

```

for 的:

```

leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl for
label main
begin
var i
i = 0
label L2
var t0
t0 = (i < 10)
ifz t0 goto L3
actual L1
call PRINTS
var t1
t1 = i + 1
i = t1
goto L2
label L3
end

```

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】

```
call PRINTN
actual m
call PRINTN
actual n
call PRINTN
var t0
actual l
actual m
actual n
t0 = call func
n = t0
actual i
call PRINTN
actual j
call PRINTN
actual k
call PRINTN
actual n
call PRINTN
actual L1
call PRINTS
end
label func
begin
formal o
formal p
formal q
actual o
call PRINTN
actual p
call PRINTN
actual q
call PRINTN
i = o
j = p
k = q
actual i
call PRINTN
actual j
call PRINTN
actual k
call PRINTN
return 999
end
```

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】

```

yerror: syntax error at line 4
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl for
label main
begin
var i
i = 0
label L2
var t0
t0 = (i < 10)
ifz t0 goto L3
actual L1
call PRINTS
var t1
t1 = i + 1
i = t1
goto L2
label L3
end
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gal for
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gvm for
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$

```

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】



```

leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl init
var a
a = 1
label main
begin
var b
b = 2
actual a
call PRINTN
actual L1
call PRINTS
actual b
call PRINTN
actual L1
call PRINTS
end
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gal init
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gvm init
1
2

```

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。

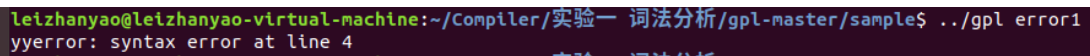
2. 出错处理

【测试用例】

error1.gpl

```
main()
{
    int a;
    a == 3;
}
```

【运行结果】

A terminal window with a dark background and light green text. The prompt is 'leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample\$'. The command entered is './gpl error1'. The output is 'yyerror: syntax error at line 4'.

【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 `yyllex` 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：

```

leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验一 词法分析/gpl-master/sample$ ../gpl init
var a
a = 1
label main
begin
var b
b = 2
actual a
call PRINTN
actual L1
call PRINTS
actual b
call PRINTN
actual L1
call PRINTS
end

```

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。