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：

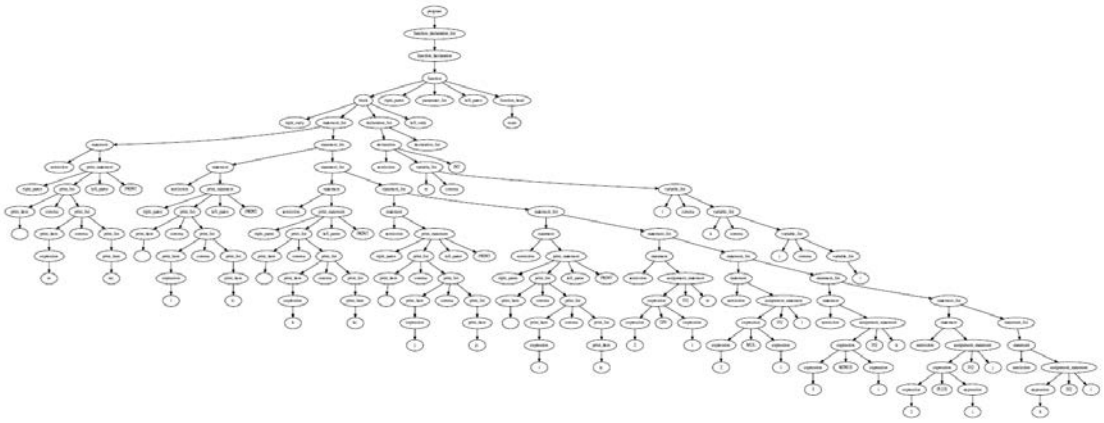
```
dot -Tpdf draw.dot -o draw.pdf
```

***注意：dot 命令需要先安装 graphviz，安装方法为：

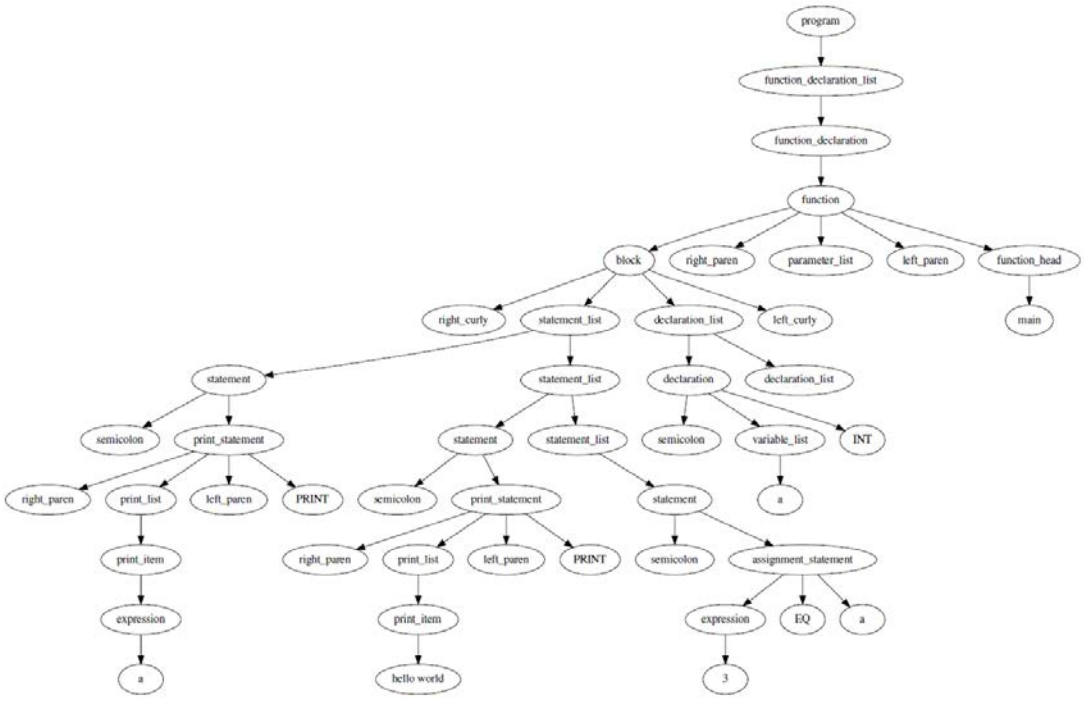
```
sudo apt-get install graphviz graphviz-doc
```

【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：

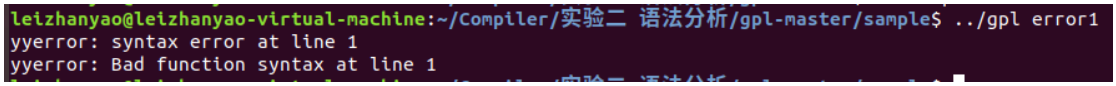
```
main(   // line 1
```

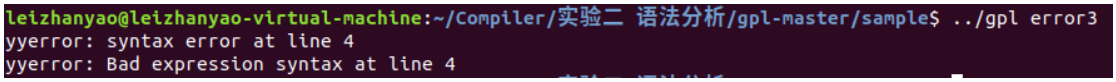
```
main()
{
    int a;
    a = 1;
    a =  ;  // line 5
}
```

```
main()
```

```
{
    int a;
    a = 1**2;   //line 4
}
```

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。

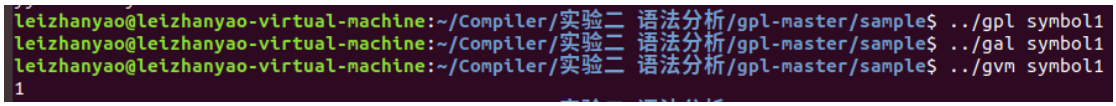
【测试用例 1】

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】

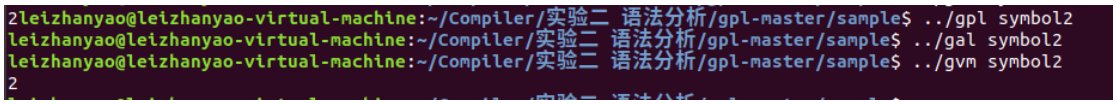
symbol2.gpl

```
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

```
main()
{
    int a;
    a = 4;
    funct(a);

}
funct(a)
{
    print(a,"\n");
}
```

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：

```
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验二 语法分析/gpl-master/sample$ ../gpl symbol4
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验二 语法分析/gpl-master/sample$ ../gal symbol4
leizhanyao@leizhanyao-virtual-machine:~/Compiler/实验二 语法分析/gpl-master/sample$ ../gvm symbol4
4
```

正确

【测试用例 5】

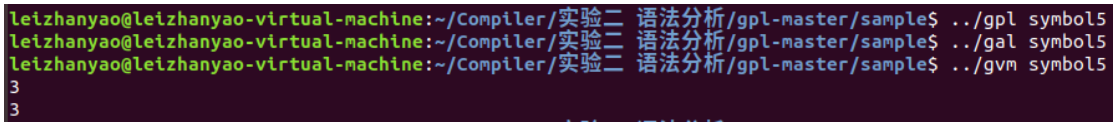
symbol5.gpl

```
main()
{
    int a;
    a = 3;
    print(a,"\n");
    func(a);
    print(a,"\n");
}
func(a)
{
    a = 5;
    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

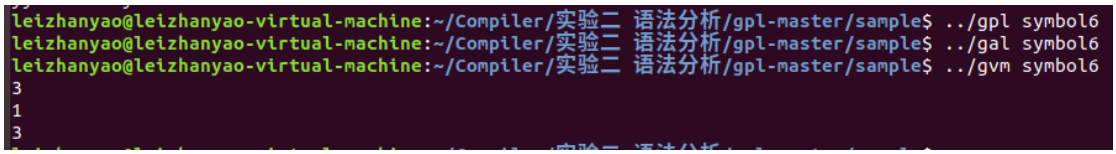
【测试用例 6】

symbol6.gpl

```
1.  main()
2.  {
3.      int a;
4.      a = 3;
5.      func(a);
6.      print(a,"\n");
7.  }
8.
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。