**编译原理第二次实验报告**

**The YACC Parser Generator: Calculator**

19计算机科学与技术（1）班 梅雨欣 2019329621004

目录

[一、实验目的 2](#_Toc89169261)

[二、实验要求 2](#_Toc89169262)

[三、实验准备与设计 3](#_Toc89169263)

[3.1 第三方工具Linux、VSC、gcc、Flex、Bison 3](#_Toc89169264)

[3.2 YACC、Bison、BNF 3](#_Toc89169265)

[3.3 YACC语法与结构 4](#_Toc89169266)

[3.4 歧义处理（二义性） 6](#_Toc89169267)

[四、完整代码 8](#_Toc89169268)

[五、实验结果 14](#_Toc89169269)

[六、实验总结 17](#_Toc89169270)

# 一、实验目的

1、构造一个**《科学计算器》**的语法分析程序，程序要求能对输入的字符串流进行词法分析和语法分析，其中语法分析要调用词法分析的结果。

2、学习使用工具软件 LEX 和 YACC 完成词法分析和语法分析。

# 二、实验要求

**1、科学计算器的功能**

作为一个科学计算器，最重要的自然是能够计算，本实验需实现以下功能：

1） 基本四则运算

2） 三角函数计算

3） 指幂运算

4） 表达式求值

5） log 运算

6） 提供一些基本常数，如 PI、e 等

7） 进制转换

// 此项暂不要求 8） 变量存储(计算器上的 MC,MR,M+,M-)

**2、实现该科学计算器，并能正确计算 4 个输入的表达式：**

1.0+2\*3=

1.0+(2\*3+cos3)/3.6-6=

tg(1.0+(sin2\*3+cos3)/3.6-6)=

4.0log(1.0+(sin2\*3+cos3)/3.6-6)=

**3、给出测试用例，证明 1 项的 7 个功能均已实现。**

**4、给出实验结论和反思。**

# 三、实验准备与设计

## 3.1 第三方工具Linux、VSC、gcc、Flex、Bison

本次实验是由Visual Studio Code编辑器通过SSH连接Linux操作系统（Ubuntu 64位）环境下，使用Flex + Bison工具完成。因为Linux编译执行文件只需要命令少，很**方便使用且能查看执行过程**。而Vistual Studio过于庞大，**占用内存**和**性能速度**远不如Linux直接使用terminal命令进行处理。

其执行过程如下：

1. 创建语言描述文件 (.y 文件)
2. 编写词法分析器函数 yylex()
3. 编写错误报告函数 yyerror()
4. 在 main() 中调用分析器函数 yyparse()
5. 执行 bison -d，由 .y 文件 产生 .tab.c 和 .tab.h 文件
6. 执行 gcc，把 .tab.c 文件编译和链接成可执行程序

Linux环境命令行如下所示：

|  |
| --- |
| flex -o CompileExp02.yy.c CompileExp02.l  bison -d CompileExp02.y  gcc CompileExp02.yy.c CompileExp02.tab.c -lm -o CompileExp02  ./CompileExp02 |

## 3.2 YACC、Bison、BNF

由于Lex相关预习知识已经在第一次实验报告中体现，此次实验着重于LEX与YACC（Flex和Bison）的知识与应用。

**YACC**(Yet Another Compiler Compiler)，是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器）。 它使用巴克斯范式(BNF) 定义语法，能处理上下文无关文法(context-free)。出现在每个产生式左边(left-hand side：lhs)的符号是nonterminal，出现在产生式右边(right-hand side：rhs)的符号有nonterminal和terminal，但terminal只出现在右端。 YACC是开发编译器的一个有用的工具，采用LALR(1) 语法分析方法。

**Bison** 是一种通用的语法分析器生成器，Yacc 的 GNU 版，将 **LALR(1)** 上下文无关文法的描述转化成分析该文法的C程序，它经常和 Flex 一起使用。

关于**BNF**可以查阅：[语法规范：BNF与ABNF](https://kb.cnblogs.com/page/189566/" \o "https://kb.cnblogs.com/page/189566/" \t "_blank)

## 3.3 YACC语法与结构

* **符号**

YACC语法由符号组成，即语法的“词”。符号是一串不以数字开头的字母、数字、句点和下划线。符号error专用于错误恢复。 由词法分析程序产生的符号叫做终结符号或者标记。定义在规则左侧的叫做非终结符号或者非终结。标记也可能是字面上引用的字符。

* **书写风格**

1. 终端符名全部用大写字母，非终端符全部用小写字母；
2. 把语法规则和语义动作放在不同的行；
3. 把左部相同的规则写在一起，左部只写一次，而后面所有规则都写在竖线“|”之后；
4. 把分号“;”放在规则最后，独占一行；
5. 用制表符来对齐规则和动作。

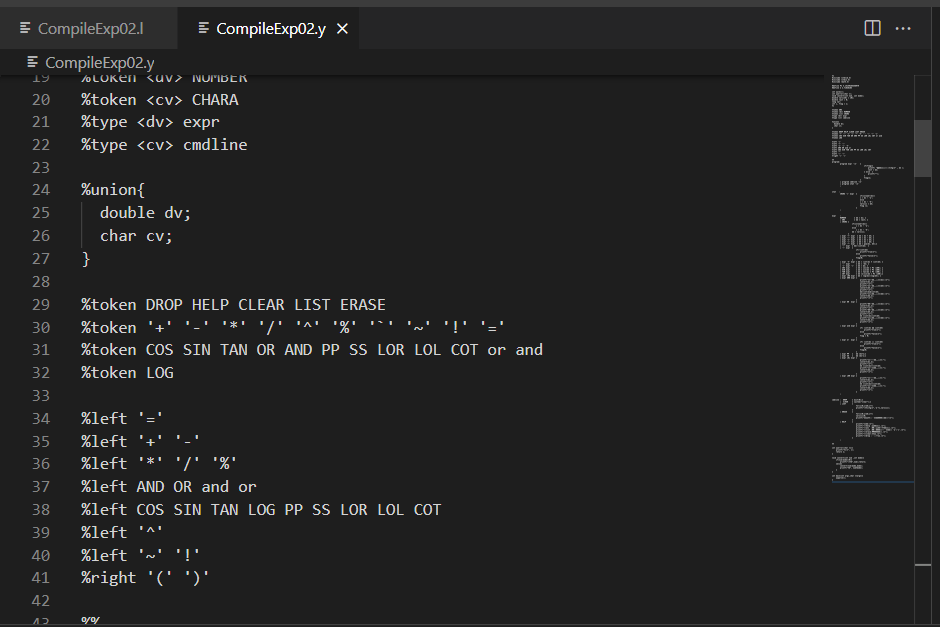
* **结构**

包括与 Flex 程序相同的三个部分结构：声明部分、规则部分、C代码部分，三个部分之间用%%隔开。

* **声明部分**

包括文字块，逐字拷贝到生成的C文件开头部分的C代码，通常包括声明和#include行。可能有%union %start %token %type %left %right %nonassoc声明。 也可以包含普通的C语言风格的注释，所有这些都是可选的，在简单的语法分析程序中，定义段可能完全是空的。 %{到%}部分会被拷贝到目标分析程序的开头。 %token记号我的理解是词法分析器得到的结果，用于传入语法分析器进行分析。通常记号总是使用大写。

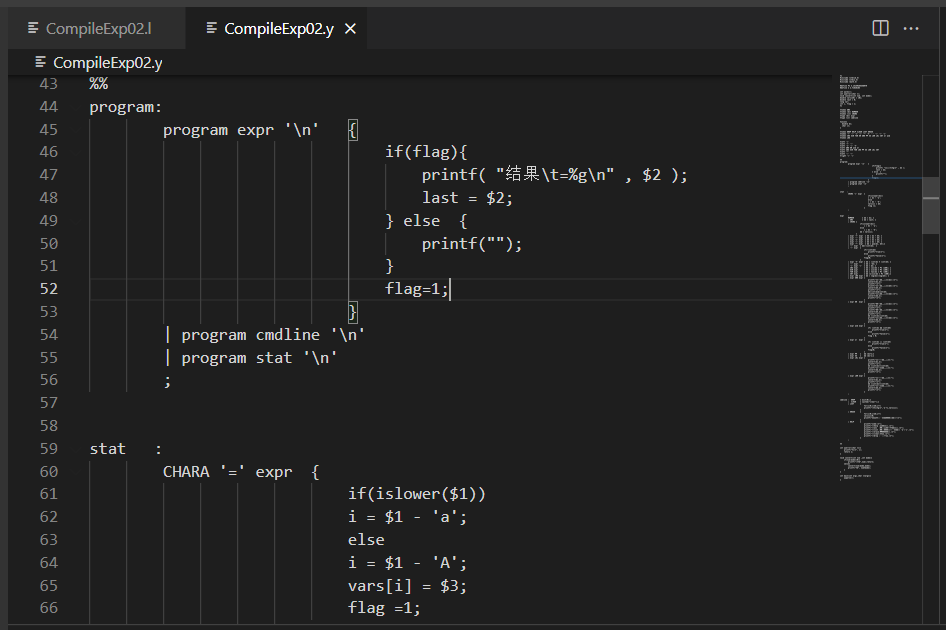
返回加号或减号，注意要把减号放在前面，避免被认作是范围符号。 对于操作符，可以定义%left和%right： %left表示左相关（left-associative） %right表示右相关（right-associative） 可以定义多组%left或%right，在后面定义的组有更高的优先级。



图表 1 实验中%token声明

* **规则部分**

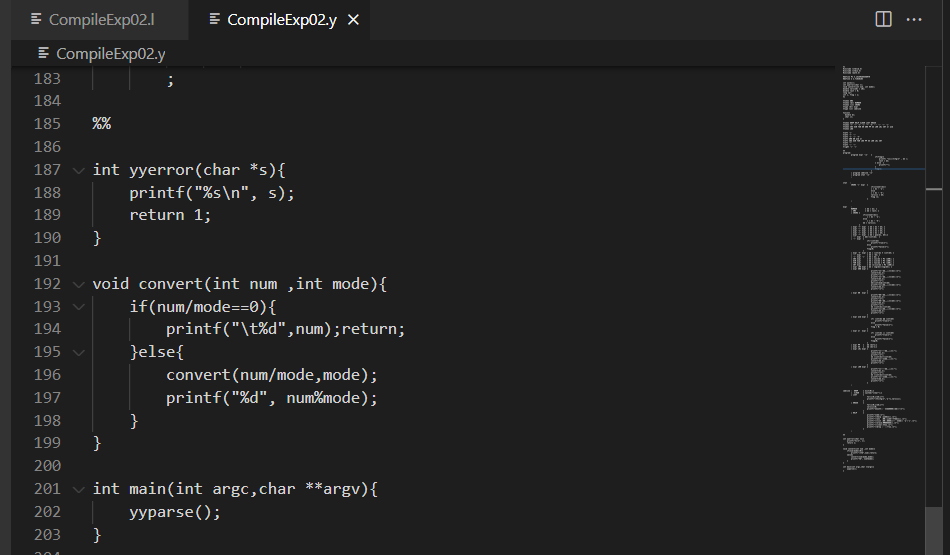
1. 用于定义规则。;表示一个规则的结束。
2. 在每个规则之后，使用{}括起，表明如何处理。
3. 规则按照语法树从上到下定义规则，第一条规则为语法起始符号，必须与输入匹配。
4. 目标符号（:左边的语法符号）的值在动作中代码用$$代替，右边语法符号的语义值依次为$1,$2，直到这条规则的结束。
5. 如果一个规则缺少现实的动作，语法分析器将把$1赋予$$，这是一个内部设定。
6. 返回记号的时候，记号对应的值是存储在yylval变量中。



图表 2实验中规则部分

* **C代码部分**

主程序，负责调用yyparse()，并输出结果。



图表 3实验中C代码部分

## 3.4 歧义处理（二义性）

如1+2\*3的处理结果可能是(1+2)\*3或1+(2\*3)。此时Bison将报错：存在移进规约冲突。表明它无法决定是先从堆栈上shift还是reduce。

有 2 种方式可以指定语法中的优先级和结合规则：隐式、显式。

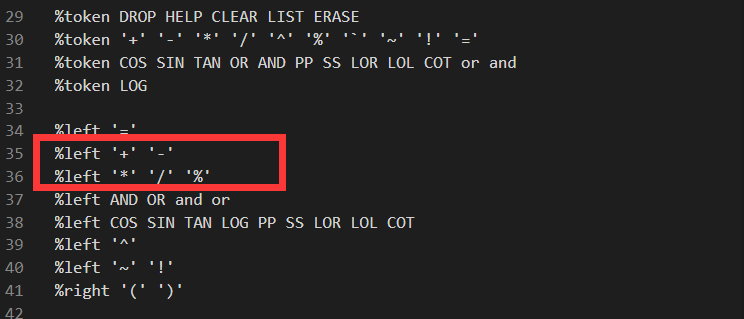
**隐式**： 使用单独的nonterminal为每个优先级重新编写语法。

|  |
| --- |
| expression: expression '+' mulexp         | expression '-' mulexp         | mulexp         ;  mulexp: mulexp '\*' primary         | mulexp '/' primary         | primary         ;  primary: '(' expression ')'         | '-' primary         | NUMBER         ; |

**显式**： 定义部分添加这些行。自上而下，优先级递增。

|  |
| --- |
| %left ‘+’ ‘-’  %left ‘\*’ ‘/’  %nonassoc UMINUS |

实验中处理方法如下：（显示）



图表 4实验中处理二义性问题

# 四、完整代码

**CompileExp02.l**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | %{  #include "CompileExp02.tab.h"  extern int yyerror(const char \*);  %}  %%  [" "; \t] { }  (0(\.[0-9]+)?)|([1-9][0-9]\*(\.[0-9]+)?) { yylval.dv = strtod(yytext,0);return NUMBER;}  [a-zA-Z] { yylval.cv = \*yytext; return CHARA;}  [-+\*/()^%~!=\n] {return \*yytext;}  "log" {return LOG;}  "cos" {return COS;}  "sin" {return SIN;}  "tan" {return TAN;}  "cot" {return COT;}  "&" {return AND;}  "|" {return OR;}  "&&" {return and;}  "||" {return or;}  "++" {return PP;}  "--" {return SS;}  "<<" {return LOL;}  ">>" {return LOR;}  "ans" {return ANS;}  "drop" {return DROP;}  "list" {return LIST;}  "erase" {return ERASE;}  "clear" {return CLEAR;}  "help" {return HELP;}  %%  int yywrap(){  return 1;  } |

**CompileExp02.y**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202 | %{  #include <stdlib.h>  #include <stdio.h>  #include <math.h>  #define PI 3.14159265358979  #define e 2.71828183  int yylex();  int yyerror(char \*);  void convert(int num, int mode);  double vars[26] = {0};  double last = 0;  long var;  int i, flag = 1;  %}  %token ANS  %token <dv> NUMBER  %token <cv> CHARA  %type <dv> expr  %type <cv> cmdline  %union{  double dv;  char cv;  }  %token DROP HELP CLEAR LIST ERASE  %token '+' '-' '\*' '/' '^' '%' '`' '~' '!' '='  %token COS SIN TAN OR AND PP SS LOR LOL COT or and  %token LOG  %left '='  %left '+' '-'  %left '\*' '/' '%'  %left AND OR and or  %left COS SIN TAN LOG PP SS LOR LOL COT  %left '^'  %left '~' '!'  %right '(' ')'  %%  program:  program expr '\n' {  if(flag){  printf( "你的结果是：\t=%g\n" , $2 );  last = $2;  } else {  printf("");  }  flag=1;  }  | program cmdline '\n'  | program stat '\n'  ;  stat :  CHARA '=' expr {  if(islower($1))  i = $1 - 'a';  else  i = $1 - 'A';  vars[i] = $3;  flag =1;  }  ;  expr :  NUMBER { $$ = $1; }  | ANS { $$ = last; }  | CHARA {  if(islower($1))  i = $1 - 'a';  else  i = $1 - 'A';  $$ = vars[i];  }  | expr '+' expr { $$ = $1 + $3; }  | expr '-' expr { $$ = $1 - $3; }  | expr '\*' expr { $$ = $1 \* $3; }  | expr '/' expr { $$ = $1 / $3; }  | expr '^' expr { $$ = pow($1, $3);}  | '~' expr { $$=~(int)$2; }  | '!' expr {  if(!(int)$2)  printf("true\n");  else  printf("false\n");  flag=0;  }  | expr '%' expr { $$ = (int)$1 % (int)$3; }  | '-' expr { $$ = -$2; }  | '(' expr ')' { $$ = $2; }  | COS expr { $$ = cos($2 \* PI /180); }  | SIN expr { $$ = sin($2 \* PI /180); }  | TAN expr { $$ = tan($2 \* PI /180); }  | COT expr { $$ =1/sin($2 \* PI /180);}  | expr LOG expr { $$ = log($1)/log($3); }  | expr AND expr {  printf("与前的二进制($1):\n");  convert($1,2);  printf("\n");  printf("与前的二进制($3):\n");  convert($3,2);  printf("\n");  $$=(int)$1&(int)$3;  printf("结果的二进制($$):\n");  convert($$,2);  printf("\n");  }  | expr OR expr {  printf("或前的二进制($1):\n");  convert($1,2);  printf("\n");  printf("或前的二进制($3):\n");  convert($3,2);  printf("\n");  $$ =(int)$1|(int)$3;  printf("结果的二进制($$):\n");  convert($$,2);  printf("\n");  }  | expr and expr {  if( (int)$1 && (int)$3)  printf("true\n");  else  printf("false\n");  flag = 0;  }  | expr or expr {  if( (int)$1 || (int)$3)  printf("true\n");  else  printf("false\n");  flag=0;  }  | expr PP { $$ =$1+1;}  | expr SS { $$ =$1-1;}  | expr LOL expr {  printf("移位前的二进制:");  convert($1,2);  printf("\n");  $$ =(int)$1<<(int)$3;  printf("移位后的二进制:");  convert($$,2);  printf("\n");  }  | expr LOR expr {  printf("移位前的二进制:");  convert($1,2);  printf("\n");  $$ =(int)$1>>(int)$3;  printf("移位后的二进制:");  convert($$,2);  printf("\n");  }  ;  cmdline : DROP { exit(0);}  | CLEAR { system("clear");}  | LIST {  for(i=0;i<26;i++)  printf("\t%c=%g\n",'a'+i,vars[i]);  }  | ERASE {  for(i=0;i<26;i++)  vars[i]=0;  printf("已经清空所有的寄存器的值!\n");  }  | HELP {  printf("命令：\n");  printf(">>help :帮助\n");  printf(">>ans :列出上次计算的结果\n");  printf(">>list :列出寄存器中所有的值 'a'/'z'\n");  printf(">>erase:重置寄存器\n");  printf(">>clear:清屏\n");  printf(">>drop :退出程序\n");  }  ;  %%  int yyerror(char \*s){  printf("%s\n", s);  return 1;  }  void convert(int num ,int mode){  if(num/mode==0){  printf("\t%d",num);return;  }else{  convert(num/mode,mode);  printf("%d", num%mode);  }  }  int main(int argc,char \*\*argv){  yyparse();  } |

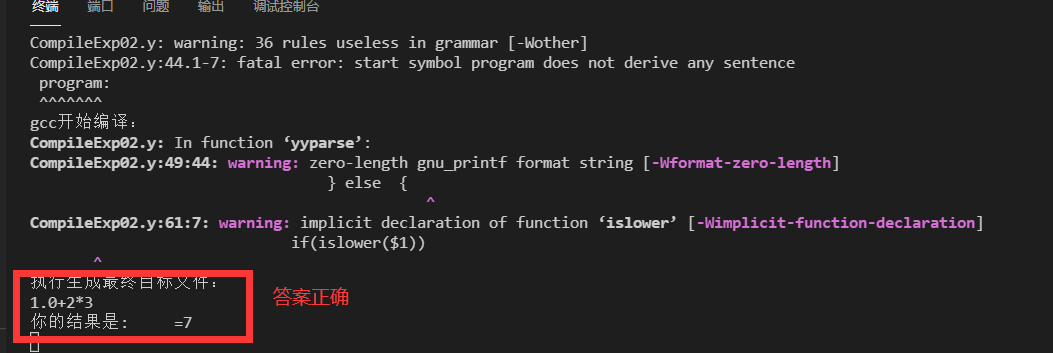
**CompileExp02.sh**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #!/bin/bash  echo "flex开始执行："  flex -o CompileExp02.yy.c CompileExp02.l  echo "bison开始执行："  #bison -o CompileExp02.tab.h CompileExp02.y  bison -d CompileExp02.y  echo "gcc开始编译："  #gcc -g -o CompileExp02 CompileExp02.yy.c CompileExp02.tab.h -lfl -lm  gcc CompileExp02.yy.c CompileExp02.tab.c -lm -o CompileExp02  #echo "删除中间文件..."  #rm CompileExp02.yy.c CompileExp02.tab.\*  echo "执行生成最终目标文件："  ./CompileExp02 |

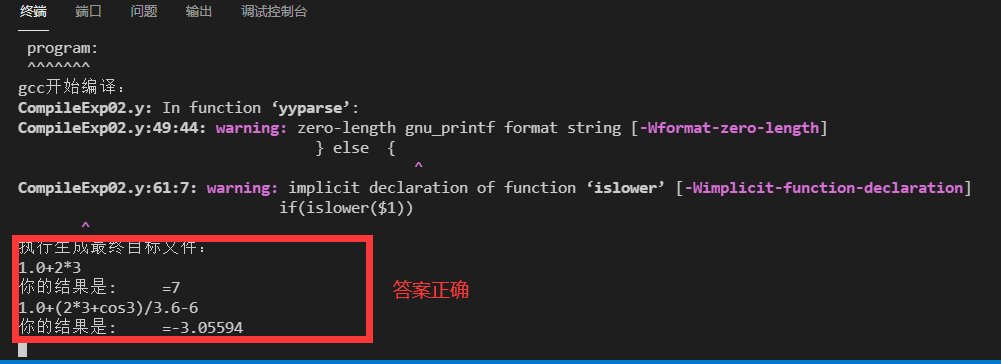
# 五、实验结果

能实现较多功能，如基本四则运算、三角函数计算、只幂运算、表达是求值、log运算、提供基本常数（PI、e）、进制转换等**基本内容**。除此之外实现**存储上次计算答案、位运算、逻辑运算、左移右移**等内容。

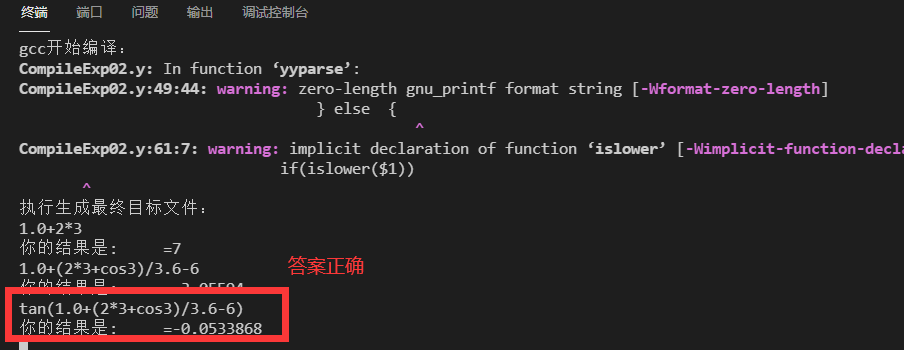
* **输入：1.0+2\*3**



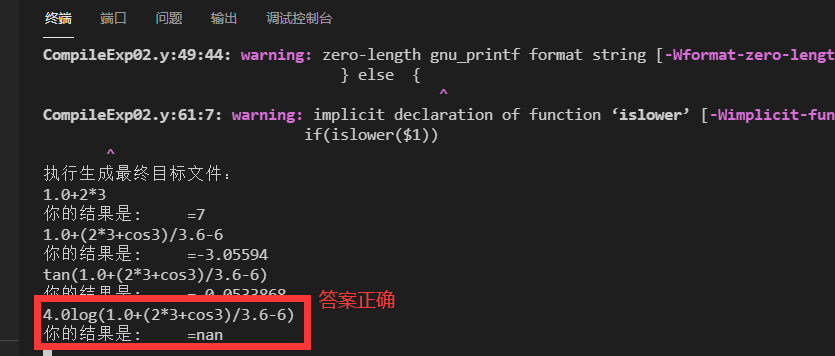
* **输入：1.0+(2\*3+cos3)/3.6-6**



* **输入：tan(1.0+(2\*3+cos3)/3.6-6)**

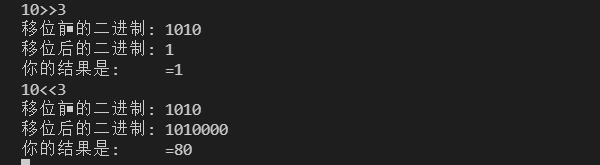


* **输入：4.0log(1.0+(2\*3+cos3)/3.6-6)**



以上是对实验内容的基本测试，以下是本程序的**其他功能展示**：

* **进制转换：左移右移，输入：10<<3和10>>3**



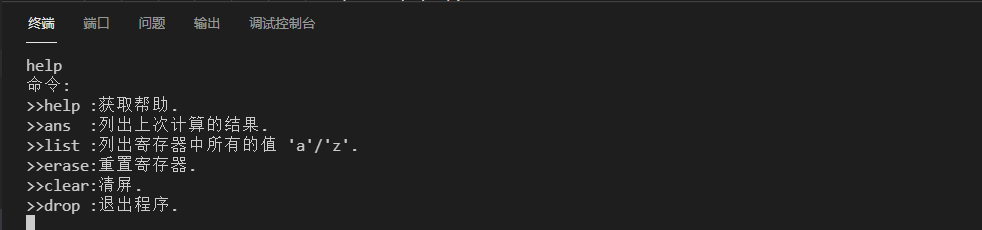
* **逻辑运算：输入0||1、1||1、0||0、1&&1、1&&0、0&&0**



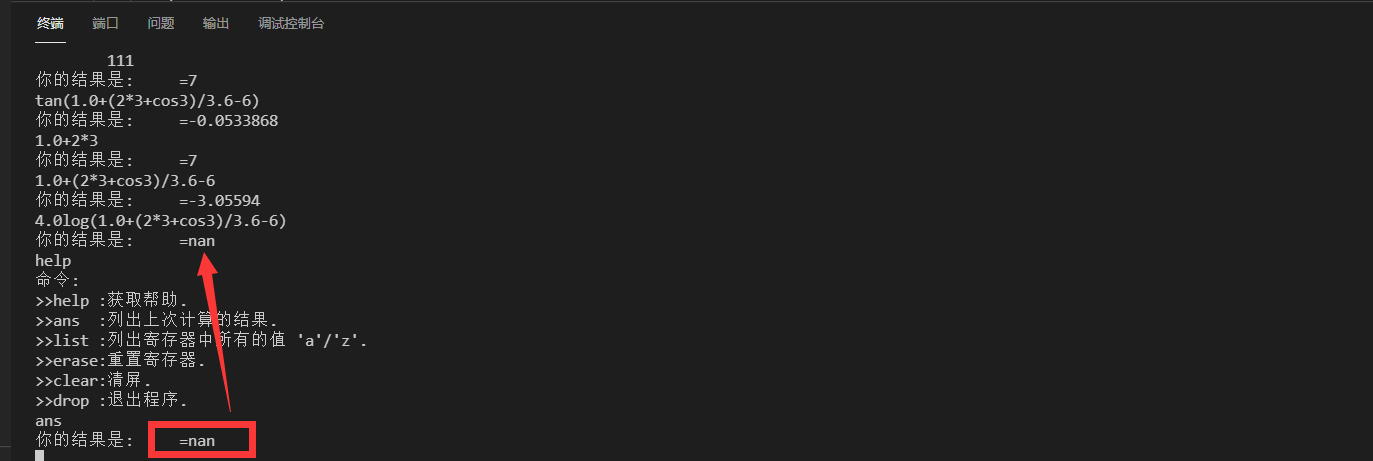
* **位运算：输入2&5、3|6**



* **help命令内容**



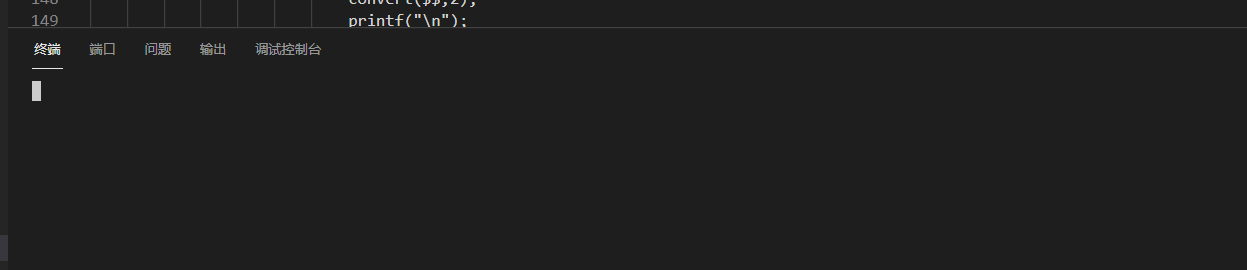
**上次计算答案（ans）**



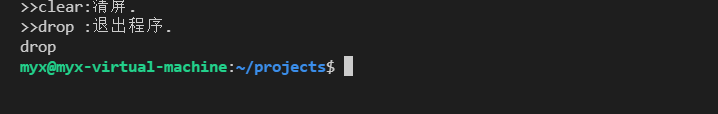
**查看寄存器的值和清空（list和erease）**



**清屏（clear）**



**退出（drop）**



# 六、实验总结

本次实验是让我们根据教材给出的内容（LEX+YACC），自己编写一个识别算术表达式的程序。

通过本次实验练习并使用了YACC（Yet Another Compiler-Compiler） 这一语法分析程序生成器。我们所需要做的工作实际上分成两部分：首先表示语法分析中的各个表达式内容（即翻译规则部分），还有词法分析将每个input转换为token的过程。我个人认为语法分析本身的难点在于对于各个符号优先级的分析和确定，以及如何处理优先级问题。其实这个也就是解决二义性问题，我采用的是显式方法定义%left按照编写的先后顺序，改变+—与\*/的优先级。

通过这次实验，我对于LEX和YACC的使用更加熟悉。其中因为我使用的环境是Linux Ubuntu，执行gcc命令时会出现问题，而且我为了方便直接编写是shell脚本直接执行命令，这也导致我没有直接发现分C语言分文件（.h+.c）不能直接引用math.h库这个问题。后面通过询问老师和查找资料才发现需要多添加“-lm”命令才能真正把这个库链接起来。非常感谢老师帮我debug，我一度以为是linux的gcc版本过低、与vsc的插件存在兼容问题，幸好最后还是解决了这个问题，也确实给我上了一课，关于各个os的常用命令的基本使用需要了解。