XIAMEN UNIVERSITY MALAYSIA

Course Code	:	CST302
Course Name	:	Compiler Principles
Lecturer	:	Mohammed Alswaitti
Academic Session	:	2019/09
Assessment Title	:	Lab Report
Submission Due Date	:	17 December 2019

Prepared by :

Student ID	Student Name
CST1709800	LEO KEE LIN
CST1709482	ZHANG CHENYU
IBU1609016	HAN YIMING
CST1709390	WEI LAN
CST1709379	WANG XIAOWEI

Date Received :

Feedback from Lecturer:

Mark:

XIAMEN UNIVERSITY MALAYSIA

Own Work Declaration

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

Date:

XIAMEN UNIVERSITY MALAYSIA

MARKING RUBRICS

Component Title			**Percentage (%)**		
Criteria	**Score and Descriptors**			**Weight (%)**	**Marks**
	Excellent (80-100 % of the weight)	**Average (40-60 % of the weight)**	**Poor (0-20 % of the weight)**		
Formating, consistency, readability, and clarity of the parts and their functions.	Complete, clear, and nearly error-free with all results	Contains some errors / missing definitions/ not informative parts	An incomplete or large amount of errors and lack of results	**7**	
The functionality of the code, samples of inputs and outputs.	Complete and nearly error-free with all results	Contains some errors / missing output	An incomplete/ not executable/ large amount of errors and lack of results	**10**	
Some samples of Justifications of what works and what does not work and why.	Clear examples with reasonable justifications	Missing examples / non-convenient justifications	No examples/ poor examples and justifications	**3**	
Additional features, suggestions, and comments	Persuasive and Convenient additions with evidence (not theoretical) will be given 1-3 bonus marks.			**bonus**	
			TOTAL	**20**	

Note to students: Please print out and attach this appendix together with the submission of coursework

XIAMEN UNIVERSITY MALAYSIA

LAB REPORT OF BUILDING CALCULATOR WITH LEX & YACC

Content

1. Introduction

In a modern computer system, there exist a crucial and necessary process between operating system and applications called “compilation”, which denotes the translation of the source code (i.e. the source program written in source language) into the semantically equivalent object code (i.e. the target program written in target language). To realize such function specifically, there is one such thing and it is known by the name of a compiler. An interpreter shares a lot of similarities but the biggest difference compared with a compiler is that, an interpreter can perform the operations implied by the source immediately with a specified input.

There are generally five phases in the compilation process, which are named lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code generation as well. Optimization techniques may also be adopted to save space, improve spend and improve utilization. We can also decompose the compilation or interpretation process into two parts:

- Read the source program and discover its structure.
- Process this structure, e.g. to generate the target program.

Lexical analysis and syntax analysis (or parsing) account for the first part. To be specific, during the lexical analysis, the source file will be split into tokens while during the syntax analysis, the hierarchical structure of the program will be found.

The first phase, that is **lexical analysis**, can be performed with the help of a tool called lexical analyzer. The detailed task of it is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. For example, if we are writing a compiler for the C programming language, the symbols *{ } ( )* all have significance on their own. The letter *a* usually appears as part of a keyword or variable name, and is not interesting on its own. Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes *"like this"*. All of these things are handled by the lexical analyzer.

A lexical analyzer generator or scanner generator, **Lex**, is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions. And these regular expressions can be specified by the user in the source specifications given to Lex. Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level.

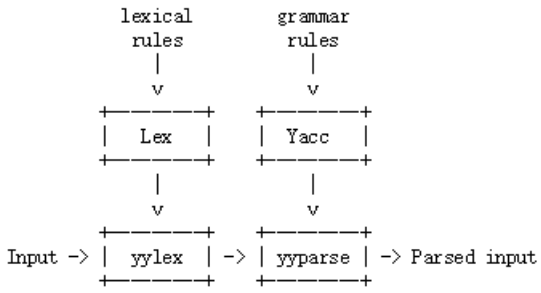
The second phase, that is **syntax analysis**, can be performed with the help of another tool called parser. The parser firstly obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. It can report any syntax errors and recover from commonly occurring errors to continue processing the remainder of the program. It can also construct a parse tree and passes it to the rest of the compiler for further processing.

A parser generator, **Yacc**, provides a general tool for imposing structure on the input to a computer program. With the specification of the input process prepared by the user (including

grammar rules describing the input structure, code to be invoked when these rules are recognized, and also a low-level routine to do the basic input), it can then generate a function (i.e. a parser) to control the input process.

Lex programs recognize only regular expressions, while Yacc writes parsers that accept a large class of context free grammars, though it requires a lower level analyzer to recognize input tokens. Thus, it is appropriate and beneficial to combine Lex and Yacc. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. Note that **yylex** and **yyparse** are functions included in the generated lexer and parser, respectively. While running the combined compiler, yyparse will be called and this function will call yylex automatically since it needs the lexer to recognize the stream of tokens for it.

The figure below illustrates how Lex and Yacc works together to realize the first part of a compiler: (i.e. lexical analysis, syntax analysis)



As a developed tool or computer program for generating lexical analyzers (scanners or lexers) compared with Lex**, Flex** (fast lexical analyzer generator) is used as an alternative to Lex together with Berkeley Yacc parser generator or GNU Bison parser generator. As a part of the GNU Project, **Bison** is a parser generator which reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) which reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. The generated parsers are portable and they do not require any specific compilers.

It can be briefly said that Flex is a successor of Lex while Bison is a GNU extension of Yacc. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.

2. Implementation

According to a detailed tutorial given by Tom Niemann which explains methods and instructions to build a compiler using Lex and Yacc, a basic calculator (can be also considered as a small compiler) was implemented by us as a start.

Firstly, pieces of code were fully understood and integrated into several related files, that is, **calc.l** for all pattern matching rules, **calc.y** for grammar rules, and three versions of the syntax-tree-walking routine. (**interpreter.c** for executing statements during the tree walk, **compiler.c** for generating code for a hypothetical stack-based machine and **graph.c** for generating a syntax tree of the original program)

Then, to create the executable compiler **calc.exe**, the following commands were executed:

```
yacc –d calc.y

lex calc.l

cc lex.yy.c y.tab.c –o calc
```

By executing the first command, the grammar rules in calc.y were read by Yacc and two new files were created: **y.tab.c** contains the function yyparse and **y.tab.h** contains token declarations. The –d option make Yacc generate definitions for tokens and place them in y.tab.h. The second command made Lex read all pattern matching rules in calc.l and created **lex.yy.c** which includes function yylex. To compile and link the generated lexer and parser together, the third command was run.

Note that to declare how we will walk the syntax tree to produce output, one of three versions (interpreter.c, compiler.c and graph.c) can be added to the last command as follows:

```
cc lex.yy.c y.tab.c interpreter.c –o calc

cc lex.yy.c y.tab.c compiler.c –o calc

cc lex.yy.c y.tab.c graph.c –o calc
```

Since Flex and Bison both are considered more flexible than Lex and Yacc and produces faster code, they are more preferable to users these days. To run Lex and Yacc programs (.l extension and .y extension) in Ubuntu or any Linux based OS, Flex and Bison packages need to be installed. Firstly, open the 'Terminal' in Linux OS and then type the following commands:

```
sudo apt-get install flex bison
```

If a 'Candidate Key not Found' error arises, type the following command to update the 'APT' database of Linux:

```
sudo apt-get update
```

To use Flex and Bison for compilation, the following commands should be executed:

```
bison –d calc.y

flex calc.l

cc lex.yy.c calc.tab.c –o calc
```

The first two commands are similar except that this time we use flex and bison for generating lexer and parser, and it is noteworthy that the generated files do not have filenames with y.tab.c or y.tab.h, instead the character ‘y’ is replaced by the filename of *.y (the file that defines grammar rules), which is “calc” in this case. Accordingly, we need to modify the following instruction in file calc.l:

```
#include “y.tab.h”
```

into

```
#include “calc.tab.h”
```

2.0 Analysis of the original calculator

Due to the simpleness and the lack of functions of the basic calculator, multiple modifications and advancements were made. Here are some shortcomings of the original calculator:

1. Firstly, it cannot deal with floating-point numbers since the parameters are defined to be integer.

```
print 5.99999+1.00000;
syntax error
```

2. Secondly, it has no functions for binary, octal or hexadecimal numbers for the same reason that the parameters are defined to be integer.

```
x=12AB
Unknown character
Unknown character
a=00010011
syntax error
```

3. Besides, it can only do simple calculations like addition, multiplication since there genuinely no functions or grammar rules defining such operation. Some complicated mathematical operations like **mod**, **abs** are impossible.

```
y=2^3;
Unknown character
syntax error
```

```
x=2%1;
Unknown character
syntax error
```

```
y=abs(-2);
syntax error
```

4. Additionally, it has not instruction for users to terminate or reset the program.

```
a=1;
b=2;
print a;
1
reset
syntax error
```

```
a=2;
c=3;
print c;
3
exit
syntax error
```

We will discuss our comprehension of the instructions in the given basic calculator as well as the work we did for several weeks to improve the robustness of the calculator in the following four sections:

a. The first section **2.1** will describe and explain the codes block by block, related detailed knowledge, example and definition will be given accordingly.

b. The next section **2.2** will specifically focus on the added functions. We will demonstrate how we added them, which instructions and techniques we adopted and how it worked. Examples with inputs and outputs will be given as well.

c. The third section **2.3** gives outputs of the combined operations.

d. The last section **2.4** is regarded as a manipulation manual which directs users to use our calculator.

2.1 Explanation on code

- **calc.h**

This file includes data structures which we will use in calc.y, compiler.c, interpreter.c and graph.c. **nodeType** is used to store several attributes of nodes in the syntax tree, which defines one node's type and type-related parameters. For instance, if one node is a constant, then the specified parameter is the value of constant. **sym[26]** is an array which is used to store numbers, and its attribute of double gives the calculator ability to have 15 decimal digits of precision. To use these data structures in other files, we just need to type the command as below:

```
#include "calc.h"
```

- **calc.l**

The input to lex is divided into three parts (definitions, rules and subroutines) with dividing notion %%.

In the first section, to-be-used header files are included. The **string.h** header defines one variable type, one macro, and various functions for manipulating arrays of characters and the **stdlib.h** gives us choice to perform general functions. Several functions declarations are made in this part and the definitions to these functions will be done in the subroutine section. Besides, two constants MAX, MIN are defined to set the boundary for numbers the calculator manipulates with. To prevent our calculator from crashing potentially, we limit the signed number of user input between -2^31 and 2^31-1 (32 bit space).

The second section is the translation rules specified. The rules are basically composed by two parts, **Patterns** (regular expressions which are used for pattern matching) on the left side and corresponding **Actions** on the right side.

Except the basic patterns like **[a-z]** for characters, **[-()<>=+*/!;{}.]** for operators, **[\t\n]+** for whitespaces and several reserved words like **"while"**, a lot of new patterns are included by us.

To be specific, **[0-9]+(\.[0-9]+)?** is added for matching float numbers, **PI, E** define Archimedes' constant and Euler's number. Besides, **b(0|1)+**, **o[0-7]+**, **h[0-9A-F]+** allow binary, octal and hexadecimal numbers to be included in our calculation. **"++"**, "--" account for self-increase and self-decrease, **"abs"** means absolute value like [number], **“exit”** gives us option to jump out of the calculation and **“reset”** allows us to start afresh, **“sqrt”|“SQRT”** allows us to calculate the square root value of some specific number, **"^"** calculates the value of the specific power of some number, **"%"** represents quotient, **“exp”|“EXP”**, **“log”|“LOG”** calculates exponential and logarithmic value respectively, **“ln”|“LN”** specifies the base of the logarithm to be **E**, **“ceil”** give the nearest integer greater than or equal to one number while **“floor”** give the nearest integer less than or equal to one number, **“cube”** returns the value of one number to the power of three. Three well-known trigonometric functions **"sin"**, **"cos"**, **"tan"** are also contained. To enrich the printing modes, **“print_all”**, **“print_bin”**, **“print_oct”**, **“print_hex”**, **“print_allf”** are also created.

The last section (subroutines) is mainly responsible for the definitions of used functions. Functions **btof()**, **otof()**, **htof()** are designed to be able to convert binary numbers, octal numbers and hexadecimal numbers to floating-point numbers. **check_overflow()** function checks whether the input number outrange the set limit. Function **yywrap()** is a default function, which will be called by Lex when input is exhausted. It returns 1 if the job is done or 0 if more processing is required.

- **calc.y**

Similar to the input to lex, the input to parser is also divided into three parts (definitions, rules and subroutines) with dividing notion %%.

The same as **calc.l**, in this part, header files are included and several functions declarations are made in this part while the definitions to these functions will be done in the subroutine section. The declarations of token are made subsequently. Besides, **expr** is bound to **nPtr**, and **INTEGER** is bound to iValue so that yacc can generate the correct code. Note that the precedence and associativity are also specified accordingly. **%nonassoc** indicates that there is no associativity. **%right** implies that the operator follows is right-associative while **%left** implies that the operator follows is left-associative. Besides, it can be seen from the code that **‘!’**, **INCR** and **DECR** have the highest priority. From bottom to top, the precedence decreases.

Following the definitions, grammar rules which can express context-free languages are designed and listed in this section. The terms appear on the left hand side is a non-terminal, and the non-terminal is expanded with specific requirements on the right hand side. Note that **‘|’** means “or”, which is not considered as a terminal. To integrate added functions into the calculator, these operators are added to expand the **expr**. **“W”**, **“X”**, **“Y”**, **“Z”** and **“V”** appears as terminals for **“stmt”**, they represents the different printing modes as what we discussed above. Besides, to improve the user experience, user-friendly words are printed out for **exit** instruction.

After every rule, associated actions may be added with braces surrounding by. In every action, **“$$”** represents the top of the stack when there is a reduction happened. **“$1”** represents the first term on the right-hand side of the production, **“$2”** represents the second term on the right-hand side of the production, **“$3”** represents the second term on the right-hand side of the production, and so on. Functions used such as **id()**, **opr()** and **ex()** will be defined in the next

section.

The last section (subroutines) defines functions we used. ***con()**, ***id()** and ***opr()** create nodes of corresponding type constant, identifier and operator, respectively. Within every function, the memory is allocated firstly and the information is copied then. **freeNode()** is used to delete a node and release the memory when necessary. **yyerror()** allows us to print the error information which appears in **calc.l**.

It is noteworthy that **main()** is only in **calc.y**, by calling **main()**, **yyparse()** will be called to run the compiler. And function **yyparse()** will automatically call **yylex()** to seize the needed tokens.

- **compiler.c**

As one of the three versions of tree walk routines, the compiler generates code for a hypothetical machine based on stack. **push**, **pop**, **jmp** and other instructions for program control based on stack are printed according to specified input. Instruction **jmp** is an unconditional jump while **jz** is a conditional jump. (occurs only when two compared values equal) Besides, many other instructions are added such as **pow**, **mod** to represent related manipulation in the stack. **Reset** can also give "All variables cleared" to users. Specific examples and demonstration will be given in the "Additional functions" section.

- **graph.c**

As another one of the three versions of tree walk routines, this version can generate a syntax tree of a given program. **ex()** gives the main entry point of the syntax tree, **exNode()** allows us to draw the tree recursively. **graphInit()**, **graphFinish()**, **graphBox()**, **graphDrawBox()** and **graphDrawArrow()** are regarded as the interface for drawing. Specific examples and demonstration will be given in the "Additional functions" section.

- **interpreter.c**

As the last version of the three versions of tree walk routines, an interpreter will executes statements during the tree walk. Each function is designed for one added mathematical calculation method. For functions which can be easily realized like **ABS**, **INCR**, we generally complement them in the function **ex()** using the actions defined in **calc.y**. While for some functions that are of difficulties and complexities such as **factorial**, we define functions specifically. Additionally, we design error messages for different functions with deep consideration. For instance, to factorial, the operand must be an integer. To **LOG**, the operand must be a positive number while to **SQRT,** the operand need to be a non-negative number. And for aesthetic purpose, **"Output:"** will be shown while running the program. Specific examples and demonstration will be given in the "Additional functions" section.

2.2 Additional functions

<table>
<tr><th colspan="2">1. (FACTORIAL) Function</th></tr>
<tr><td>Function performance</td><td>Find the factorial of a number</td></tr>
<tr><td>Code added to calc.l</td><td>

```
[-()<>=+*/!;{}.]    return *yytext;
```

</td></tr>
<tr><td>Code added to calc.y</td><td>

```
%nonassoc UMINUS
%left '!'
```

```
expr:
    | expr '!'                  { $$ = opr('!', 1, $1); }
```

factorial function is given to be higher precedence than UMINUS to avoid error when calculation

</td></tr>
<tr><td>Code added to interpreter.c</td><td>

```
case '!':       if (ex(p->opr.op[0]) - (int)ex(p->opr.op[0]) != 0)
                {
                    printf("\t\tError: Must be an integer\n");
                    return 0;
                }
                else
                    return factorial(ex(p->opr.op[0]));
```

</td></tr>
<tr><td>Code added to compiler.c</td><td>

```
case '!':
    ex(p->opr.op[0]);
    printf("\tfactorial\n");
    break;
```

</td></tr>
<tr><td>Code added to graph.c</td><td>

```
case '!':       s = "[!]"; break;
```

</td></tr>
<tr><td colspan="2">Output:

Interpreter:

```
x = 3!;
print x;
                Output: 6.000000
```

```
x=-3!;
        push    3.000000
        factorial
        neg
        pop     x
```


Compiler:

```
x = 3!;
        push    3.000000
        factorial
        pop     x
```

```
x=-3!;
        push    3.000000
        factorial
        neg
        pop     x
```


Graph:

```
x = 3!;

Graph 0:

      [=]
       |
   |------|
   |      |
 id(X)   [!]
           |
           |
           |
       c(3.000)
```

```
x=-3!;

Graph 0:

      [=]
       |
   |------|
   |      |
 id(X)   [_]
           |
           |
           |
         [!]
          |
          |
          |
       c(3.000)
```

</td></tr>
</table>

2. (INCR and DECR) Function

Function performance

"++": increment the value on its left/right by 1
"--": decrement the value on its left/right by 1

Code added to calc.l

```
"++"            return INCR;
"--"            return DECR;
```

Code added to calc.y

```
%nonassoc UMINUS
%left '!' INCR DECR
```

```
expr:
    | expr INCR     { $$ = opr(INCR, 1, $1); }
    | expr DECR     { $$ = opr(DECR, 1, $1); }
    | INCR expr     { $$ = opr(INCR, 1, $2); }
    | DECR expr     { $$ = opr(DECR, 1, $2); }
```

both INCR and DECR have the same precedence level with factorial function, and higher precedence than the UMINUS. The calculation is carried out in the order from left to right.

Code added to interpreter.c

```
case INCR:      return ex(p->opr.op[0]) + 1;
case DECR:      return ex(p->opr.op[0]) - 1;
```

Code added to compiler.c

```
case INCR:
    ex(p->opr.op[0]);
    printf("\tpush\t%lf\n", 1.0);
    printf("\tadd\n");
    break;
case DECR:
    ex(p->opr.op[0]);
    printf("\tpush\t%lf\n", 1.0);
    printf("\tsub\n");
    break;
```

Code added to graph.c

```
case INCR:      s = "[++]"; break;
case DECR:      s = "[--]"; break;
```

Output:

Interpreter:

```
x = 3++;
print x;
                Output: 4.000000
y = ++3;
print y;
                Output: 4.000000
x = 3--;
print x;
                Output: 2.000000
y = --3;
print y;
                Output: 2.000000
```

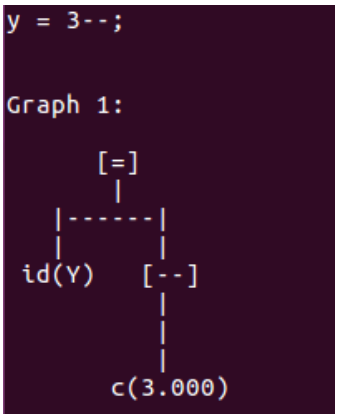
```
x = 5;
print x++!;
                Output: 720.000000
print x!++;
                Output: 121.000000
```

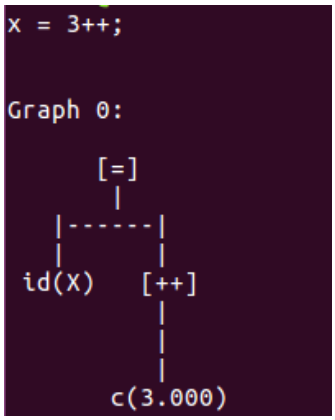
Compiler：

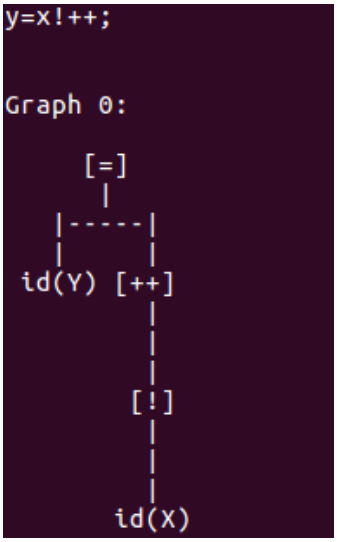
```
x = 3++;
        push    3.000000
        push    1.000000
        add
        pop     x
y = ++3;
        push    3.000000
        push    1.000000
        add
        pop     y
x = 3--;
        push    3.000000
        push    1.000000
        sub
        pop     x
y = --3;
        push    3.000000
        push    1.000000
        sub
        pop     y
```

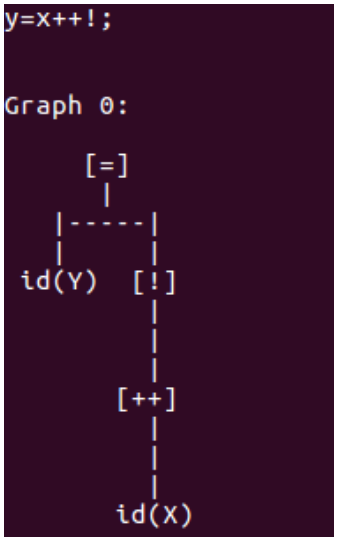
```
x=5;
        push    5.000000
        pop     x
print x++!;
        push    x
        push    1.000000
        add
        factorial
        print
print x!++;
        push    x
        factorial
        push    1.000000
        add
        print
```

Graph：









3. (POWER AND MOD) Function

Function performance

"^": power function, return the left value to the power of the right value
"%": mod function, return the quotient when dividing 2 numbers

Code added to calc.l

```
"^"                 return POW;
"%"                 return MOD;
```

Code added to calc.y

```
%left POW MOD
%nonassoc UMINUS
%left '!' INCR DECR
```

```
expr:
    | expr POW expr             { $$ = opr(POW, 2, $1, $3);}
    | expr MOD expr             { $$ = opr(MOD, 2, $1, $3);}
```

#both POW and MOD have lower precedence than UMINUS, factorial, INCR and DECR. The calculation is carried out in the order from left to right.

Code added to interpreter.c

```
case POW:       return pow(ex(p->opr.op[0]),ex(p->opr.op[1]));
case MOD:       if ((ex(p->opr.op[0]) - (int)ex(p->opr.op[0]) != 0) ||
                    (ex(p->opr.op[1]) - (int)ex(p->opr.op[1]) != 0))
                {
                    printf("\t\tError: Operand must be an integer\n");
                    return 0;
                }
                else
                    return ((int)(ex(p->opr.op[0])) % ((int)ex(p->opr.op[1])));
```

Code added to compiler.c

```
switch (p->opr.oper)
{
    case POW:   printf("\tpow\n"); break;
    case MOD:   printf("\tmod\n"); break;
}
```

Code added to graph.c

```
case POW:       s = "[^]"; break;
case MOD:       s = "[%]"; break;
```

Output:

Interpreter:

```
x = 2^3;
print x;
                Output: 8.000000
y = 5%2;
print y;
                Output: 1.000000
```

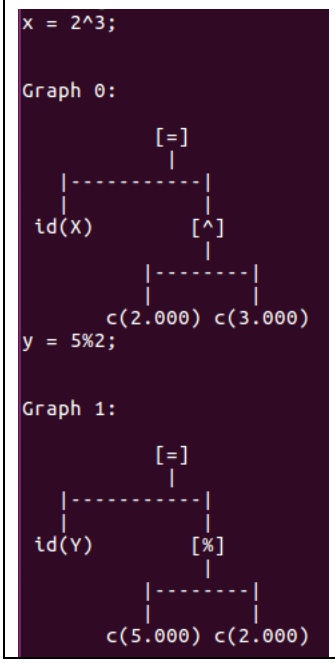
```
print -2^-3;
                Output: -0.125000
print -5%-2;
                Output: -1.000000
```

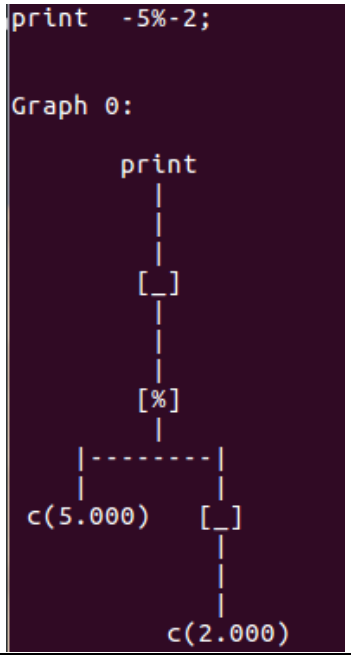
Compiler:

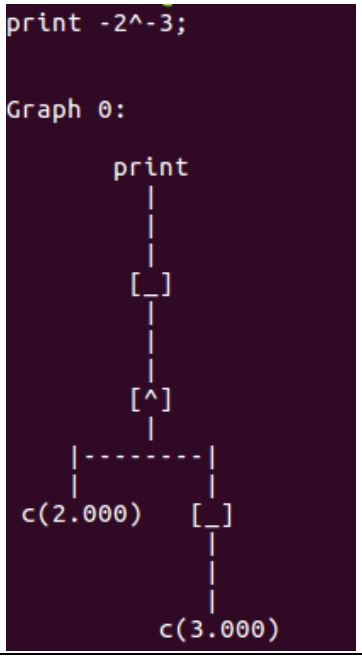
```
x = 2^3;
        push    2.000000
        push    3.000000
        pow
        pop     x
y = 5%2;
        push    5.000000
        push    2.000000
        mod
        pop     y
```

```
-2^-3;
        push    2.000000
        push    3.000000
        neg
        pow
        neg
-5%-2;
        push    5.000000
        push    2.000000
        neg
        mod
        neg
```

Graph：


x = 2^3;
Graph 0:
[=]
id(X)
[^]
c(2.000) c(3.000)
y = 5%2;
Graph 1:
[=]
id(Y)
[%]
c(5.000) c(2.000)


print -5%-2;
Graph 0:
print
[_]
[%]
c(5.000)
[_]
c(2.000)


print -2^-3;
Graph 0:
print
[_]
[^]
c(2.000)
[_]
c(3.000)

4. (RESET and EXIT) Function

Function performance	exit: terminate the program reset: reset all the variables' value to zero
Code added to calc.l	`"reset" return RESET;` `"exit" return EXIT;`

Code added to calc.y	`%token EXIT RESET`
	`program:`
	`function EXIT { printf("THANK YOU!\n");`
	`exit(0); }`
	`;`
	`stmt:`
	`| RESET ';' { $$ = opr(RESET, 0); }x`
	`;`

Code added to interpreter.c	`case RESET: reset();`
	`printf("\t\tAll varaibles cleared\n");`
	`return 0;`

Code added to compiler.c	No
Code added to graph.c	No

Output:

Interpreter:

```
reset;
                All varaibles cleared
exit;
THANK YOU!
```

Compiler:

```
x = 3;
        push    3.000000
        pop     x
print x;
        push    x
        print
reset;
        All varaibles cleared
exit;
THANK YOU!
```

Graph:

```
reset;


Graph 2:

 all variable reset
exit;
THANK YOU!
```

5. Other Function

Function performance

abs() : return the absolute value of the number
exp() : find the n^{th} power of natural constant e
log() : logarithmic function in base 10
ln() : logarithmic function in base e
sqrt() : square root function
cube() : find the third power of a number
sin() : sin function
cos() : cos function
tan() : tan function
ceil() : ceiling function
floor() : floor function

Code added to calc.l

```
"abs"                   return ABS;
"exp"|"EXP"             return EXP;
"log"|"LOG"             return LOG;
"ln"|"LN"               return LN;
"sqrt"|"SQRT"           return SQRT;
"cube"                  return CUBE;
"sin"                   return SIN;
"cos"                   return COS;
"tan"                   return TAN;
"ceil"                  return CEIL;
"floor"                 return FLOOR;
```

Code added to calc.y

```
%token ABS SIN COS TAN SQRT CUBE EXP LOG LN CEIL FLOOR
```

```
| ABS '(' expr ')'          { $$ = opr(ABS, 1, $3); }
| EXP '(' expr ')'          { $$ = opr(EXP, 1, $3); }
| LOG '(' expr ')'          { $$ = opr(LOG,1, $3); }
| LN '(' expr ')'           { $$ = opr(LN, 1, $3); }
| SQRT '(' expr ')'         { $$ = opr(SQRT, 1, $3); }
| CUBE '(' expr ')'         { $$ = opr(CUBE, 1, $3); }
| SIN '(' expr ')'          { $$ = opr(SIN, 1, $3); }
| COS '(' expr ')'          { $$ = opr(COS, 1, $3); }
| TAN '(' expr ')'          { $$ = opr(TAN, 1, $3); }
| CEIL '(' expr ')'         { $$ = opr(CEIL, 1, $3); }
| FLOOR '(' expr ')'        { $$ = opr(FLOOR, 1, $3); }
```

Code added to interpreter.c

```
case ABS:       if (ex(p->opr.op[0]) >= 0)
                    return ex(p->opr.op[0]);
                else
                    return -ex((p->opr.op[0]));
case EXP:       return exp(ex(p->opr.op[0]));
case LOG:       if (ex(p->opr.op[0]) <= 0)
                {
                    printf("\t\tError: Must be a positive number\n");
                    return 0;
                }
                else
                    return log10(ex(p->opr.op[0]));
case LN:        if (ex(p->opr.op[0]) <= 0)
                {
                    printf("\t\tError: Must be a positive number\n");
                    return 0;
                }
                else
                    return log(ex(p->opr.op[0]));
case SQRT:      if (ex(p->opr.op[0]) < 0)
                {
                    printf("\t\tError: Must be a non-negative number\n");
                    return 0;
                }
                else
                    return sqrt(ex(p->opr.op[0]));
case CUBE:      return cube(ex(p->opr.op[0]));
case SIN:       return sin(ex(p->opr.op[0]) * pi / 180);
case COS:       return cos(ex(p->opr.op[0]) * pi / 180);
case TAN:       return tan(ex(p->opr.op[0]) * pi / 180);
case CEIL:      return ceil(ex(p->opr.op[0]));
case FLOOR:     return floor(ex(p->opr.op[0]));
```

Code added to compiler.c

```
case ABS:
    ex(p->opr.op[0]);
    printf("\tABS\n");
    break;
case EXP:
    ex(p->opr.op[0]);
    printf("\tEXP\n");
    break;
case LOG:
    ex(p->opr.op[0]);
    printf("\tLOG\n");
    break;
case LN:
    ex(p->opr.op[0]);
    printf("\tLN\n");
    break;
case SQRT:
    ex(p->opr.op[0]);
    printf("\tSQRT\n");
    break;
case CUBE:
    ex(p->opr.op[0]);
    printf("\tCUBE\n");
    break;
```

```
case SIN:
    ex(p->opr.op[0]);
    printf("\tSIN\n");
    break;
case COS:
    ex(p->opr.op[0]);
    printf("\tCOS\n");
    break;
case TAN:
    ex(p->opr.op[0]);
    printf("\tTAN\n");
    break;
case CEIL:
    ex(p->opr.op[0]);
    printf("\tCEIL\n");
    break;
case FLOOR:
    ex(p->opr.op[0]);
    printf("\tFLOOR\n");
    break;
```

Code added to graph.c

```
case ABS:       s = "[ABS]"; break;
case EXP:       s = "[EXP]"; break;
case LOG:       s = "[LOG]"; break;
case LN:        s = "[LN]"; break;
case SQRT:      s = "[SQRT]"; break;
case CUBE:      s = "[CUBE]"; break;
case SIN:       s = "SIN"; break;
case COS:       s = "COS"; break;
case TAN:       s = "TAN"; break;
case CEIL:      s = "[CEIL]"; break;
case FLOOR:     s = "[FLOOR]"; break;
```

Sample Output:

Interpreter:

```
x = abs(-2);
print x;
                Output: 2.000000
```

Compiler:

```
x = abs(-2);
        push    2.000000
        neg
        ABS
        pop     x
```

Graph:

```
x = abs(-2);


Graph 0:

      [=]
       |
   |------|
   |      |
 id(X)  [ABS]
          |
          |
          |
        [_]
         |
         |
         |
       c(2.000)
```

6. (PRINT SERIES) Function

Function performance

print : print the variable's value as normal
print_bin : print the value in binary representation
print_oct : print the value in octal representation
print_hex : print the value in hexadecimal representation
print_all : print all the used variable names and their corresponding values
print_allf : print all the used variable names and their corresponding value in decimal, hexadecimal, octal, and binary

Code added to calc.l

```
"print"             return PRINT;
"print_all"         return W;
"print_bin"         return X;
"print_oct"         return Y;
"print_hex"         return Z;
"print_allf"        return V;
```

Code added to calc.y

```
%token PRINT V W X Y Z
```

```
stmt:
    | PRINT expr ';'          { $$ = opr(PRINT, 1, $2); }
    | W ';'                   { $$ = opr(W, 0); }
    | V ';'                   { $$ = opr(V, 0); }
    | X expr ';'              { $$ = opr(X, 1, $2); }
    | Y expr ';'              { $$ = opr(Y, 1, $2); }
    | Z expr ';'              { $$ = opr(Z, 1, $2); }
```

Code added to interpreter.c

```
case PRINT:     printf("\t\tOutput: %lf\n", ex(p->opr.op[0]));
                return 0;
case V:         printallf();
                return 0;
case W:         printall();
                return 0;
case X:         printf("\t\tOutput: ");
                ftob(ex(p->opr.op[0]));
                printf("\n");
                return 0;
case Y:         printf("\t\tOutput: ");
                ftoo(ex(p->opr.op[0]));
                printf("\n");
                return 0;
case Z:         printf("\t\tOutput: ");
                ftoh(ex(p->opr.op[0]));
                printf("\n");
                return 0;
```

Code added to compiler.c

```
case PRINT:
    ex(p->opr.op[0]);
    printf("\tprint\n");
    break;
case V:
    printf("\tprint_all\n");
    break;
case W:
    printf("\tprint_all_form\n");
    break;
case X:
    ex(p->opr.op[0]);
    printf("\tprint_bin\n");
    break;
case Y:
    ex(p->opr.op[0]);
    printf("\tprint_oct\n");
    break;
case Z:
    ex(p->opr.op[0]);
    printf("\tprint_hex\n");
    break;
```

Code added to graph.c

```
case PRINT:     s = "print"; break;
case V:         s = "print all variable in all form\n"; break;
case W:         s = "print all variable\n"; break;
case X:         s = "print_bin"; break;
case Y:         s = "print_oct"; break;
case Z:         s = "print_hex"; break;
```

Output:

Interpreter:

```
x = b1101111;
y = o12345;
z = hABCD;
print x;
                Output: 111.000000
print_bin x;
                Output: bx1101111
print_oct x;
                Output: ox157
print_hex x;
                Output: hx6F
print_all;
        Output:
                x = 111.000000
                y = 5349.000000
                z = 43981.000000
print_allf;
        Output:
        x = 111.00      hx6F        ox157       bx1101111
        y = 5349.00     hx14E5      ox12345     bx1010011100101
        z = 43981.00    hxABCD      ox125715    bx1010101111001101
```

Compiler：

```
x = b1101111;
        push    111.000000
        pop     x
y = o12345;
        push    5349.000000
        pop     y
z = hABCD;
        push    43981.000000
        pop     z
print x;
        push    x
        print
print_bin x;
        push    x
        print_bin
print_oct x;
        push    x
        print_oct
print_hex x;
        push    x
        print_hex
print_all;
        print_all_form
print_allf;
        print_all
```

Graph：

```
Graph 3:

 print
   |
   |
   |
 id(X)
print_bin x;


Graph 4:

 print_bin
     |
   |--
   |
 id(X)
print_oct x;


Graph 5:

 print_oct
     |
   |--
   |
 id(X)
print_hex x;
```

```
Graph 6:

 print_hex
     |
   |--
   |
 id(X)
print_all;


Graph 7:

 print all variable

print_allf;


Graph 8:

 print all variable in all form
```

2.3 Combination of Different Functions

```
x=20;
while (x>=0)
{
        print_bin x;
        x=x--;
}
                Output: bx10100
                Output: bx10011
                Output: bx10010
                Output: bx10001
                Output: bx10000
                Output: bx1111
                Output: bx1110
                Output: bx1101
                Output: bx1100
                Output: bx1011
                Output: bx1010
                Output: bx1001
                Output: bx1000
                Output: bx111
                Output: bx110
                Output: bx101
                Output: bx100
                Output: bx11
                Output: bx10
                Output: bx1
                Output: bx
```

```
x=100;
a=7;
b=a;
y=a!*b+x;
print y
;
                Output: 35380.000000
x=exp(5)+abs(-1)+sqrt(2);
print x;
                Output: 150.827373
y=log(10)+log(2)*sqrt(4)+cube(3)-3%1;
print y;
                Output: 28.602060
y=5^6+3%2+2--;
print y;
                Output: 15627.000000
x=--1+ceil(3.3333)-floor(-4.44444)+8!;
print x;
                Output: 40329.000000
z=sin(2)+cos(30)-tan(45)+ln(e);
print z;
                Output: 0.900925
```

```
x = b1101111;
y = o12345;
z = hABCD;
print x;
                Output: 111.000000
print_bin x;
                Output: bx1101111
print_oct x;
                Output: ox157
print_hex x;
                Output: hx6F
print_all;
        Output:
                x = 111.000000
                y = 5349.000000
                z = 43981.000000
print_allf;
        Output:
        x = 111.00      hx6F        ox157       bx1101111
        y = 5349.00     hx14E5      ox12345     bx1010011100101
        z = 43981.00    hxABCD      ox125715    bx1010101111001101
```

```
x = 5;
y = 2;
print y^x!;
```

```
Output: 1329227995784915872903807060280344576.000000
```

```
y=cube(x)++;
print y;
```

```
Output: 730.000000
```

```
y=x^x*2;
print y;
```

```
Output: 774840978.000000
```

```
x=100;
a=7;
b=a;
y=a!*b+x;
print y
;
```

```
Output: 35380.000000
```

2.4 Manipulation Manual

This calculator can produce 3 types of output: interpreter, compiler, and graph.

1. Open the terminal and go to the folder where these files are
2. To produce an interpreter program, type the command: cc calc.h lex.yy.c calc.tab.c interpreter.c
3. To produce a compiler program, type the command: cc calc.h lex.yy.c calc.tab.c compiler.c
4. To produce a graph program, type the command: cc calc.h lex.yy.c calc.tab.c graph.c
5. Run the a.out file using command: ./a.out
6. You can use different functions/operators given in manual
7. All the statement must follow by a semicolon
8. If u write something other than functions as specified in manual it will give syntax error
9. In order to get out of the calculator type "exit"

Calculator Functions and Instructions :

C like statement

if (expression) statement;
if (expression) statement else statement;

All lower case character (a to z) are reserved as variable name to store variable

CONSTANT

PI = 3.141592654
E = 2.718281828
Example : x = PI;

Trigonometric Function, inputs are recognise as in degree.

sin() : sin function
cos() : cos function
tan() : tan function
Example : x = sin(30);

Logarithmetic Function

log() or LOG() : logarithmic function used for calculating log value in base 10
ln() or LN() : logarithmic function used for calculating log value in base e
Example : x = log(30);

Arithmetric operator, input the arithmetric expression as usual

"+" : adding 2 numbers
"-" : substracting 2 numbers
"*" : multiplying 2 numbers
"/" : dividde 2 numbers
"%" : mod function, return the quotient when dividing 2 numbers
"^" : power function, return the left value to the power of the right value
'+' and '-' are in the same precedence, left associative
'*' and '/' are in the same precedence, left associative
'^' and '%' are in the same precedence, left associative
'*' and '/' are higher precedence than '+' and '-'
'^' and '%' are higher precedence than '*' and '/'
use open and close brackets '(' ')' when necessary
Example : x = y + 5 * 4 ^ 2 / 3;

Other Function

abs() : return the absolute value of the number
sqrt() : square root function, input value must greater than or equal to zero
ceil() : ceiling function, return the nearest integer greater then or equal to the number
floor() : floor function, return the nearest integer smaller then or equal to the number
cube() : cube function, return the value power of 3
exp() or EXP() : exponential function, return the exponential e power of value input
Example : x = abs(y);

Other Operator

"!" : factorial function, input must be an integer
"++" : increment the value by 1
"--" : decrement the value by 1
'!', '++', and '--' have the same precedence, all higher than arithmetic operator
Example : x = 2 ^ 5!;

Print Function

print : print the variable's value as normal
print_bin : print the value in binary representation
print_oct : print the value in octal representation
print_hex : print the value in hexadecimal representation
Example : print x;
print_all : print all the used variable names and their corresponding values
print_allf : print all the used variable names and their corresponding value in decimal, hexadecimal, octal, and binary

For binary, octal, and hexadecimal, this calculator only print integer part of the value by discarding the floating point part.

Example : print_all;

Input Recognizaiton

can recognize decimal, binary, octavalue, and hexadicimal number
max and min value for input is 2,147,483,647 and -2,147,483,648
all the calculations give at least 15 decimal value of precision
decimal : input the values as usual
bianry : input the values with prefix 'b', maximum 32 bits of '1' and '0'
octal : input the values with prefix 'o', maximum 12 characters [0-7]
hexadicimal : input the values with prefix 'h', use upper case letter (A-F) and [0-9], maximum 8 characters

All the input for binary, octal, and hexadecimal must be an integer

Other

exit : terminate the program

reset : reset all the variables' value to zero

3. Discussion

During the implementation of new functions, many problems arose while coding. In addition to spelling errors, there were also some simple errors, such as the lack of function library in the part of C language code, or the lack of files. Additionally, there were many complex problems, such as in factorial operation, float number will cause problems, as well as some calculation errors. Hence, in this part, we will focus on these problems we encountered and the corresponding solutions.

Problem A

- *Description*

After adding the mathematical functions used in the calculation to the interpreter.c file, and on the premise of ensuring all the codes were correct, we got the following error prompt after running.

- *Error*

```
interpreter.c: In function ‘ex’:
interpreter.c:25:37: warning: implicit declaration of function ‘log10’ [-Wimplicit-function-declaration]
    case LOG:               {int d = log10(ex(p->opr.op[0]));
                                     ^
interpreter.c:25:37: warning: incompatible implicit declaration of built-in function ‘log10’
interpreter.c:25:37: note: include ‘<math.h>’ or provide a declaration of ‘log10’
interpreter.c:42:22: warning: implicit declaration of function ‘sin’ [-Wimplicit-function-declaration]
    case SIN:  return sin(ex(p->opr.op[0]) * PI / 180);
                      ^
interpreter.c:42:22: warning: incompatible implicit declaration of built-in function ‘sin’
interpreter.c:42:22: note: include ‘<math.h>’ or provide a declaration of ‘sin’
interpreter.c:43:22: warning: implicit declaration of function ‘cos’ [-Wimplicit-function-declaration]
    case COS:  return cos(ex(p->opr.op[0]) * PI / 180);
                      ^
interpreter.c:43:22: warning: incompatible implicit declaration of built-in function ‘cos’
interpreter.c:43:22: note: include ‘<math.h>’ or provide a declaration of ‘cos’
interpreter.c:44:22: warning: implicit declaration of function ‘tan’ [-Wimplicit-function-declaration]
    case TAN:  return tan(ex(p->opr.op[0]) * PI / 180);
                      ^
interpreter.c:44:22: warning: incompatible implicit declaration of built-in function ‘tan’
interpreter.c:44:22: note: include ‘<math.h>’ or provide a declaration of ‘tan’
```

- *Solution*

In the file of interpreter.c, we must add the <math.h> file in the standard library required for mathematical operation. The <math.h> header file declares some common mathematical operations, such as the multiplication and the square root, so we can realize the mathematical calculation we need.

Problem B

- *Description*

When we tried to add functions related to binary numbers, it was found that two’s complement numbers could not be recognized. For two’s complement numbers, the first bit indicates its negativity.

- *Solution*

To solve such problem, we decided to think from another angle that we may just simply use – to represent negative binary numbers. And to make sure that users can correctly use such technique, manual was updated to include such information.

Problem C

- *Description*

When we added functions for manipulating **double** numbers, we suddenly found that two types of numbers, integer and floating-point numbers could not be stored at the same time. A lot of codes would be needed changing and there would be plenty of errors to crash the program. And of course, the user-experience would be influenced.

- *Solution*

To solve such program, the easiest and simplest way was to convert data type integer to double. Hence, all the functions with parameter of integer type became functions aimed at floating-point numbers of **double** type.

However, since functions **'!'** and **'%'** are only valid while the operands are integers, another technique needed to be included. That is, we would check the operand before the operations were made. If the floating part of one number is multiples of 0, for example, a number 5.00000, we could directly extract the integral part (which is 5 in the example) and manipulate only on the integral part.

Problem D

- *Description*

Mathematically speaking, the second operand of the MOD function cannot be 0. The program recognized the 0 operand in MOD function as floating point.

- *Error*

```
y=2%2;
x=2%1;
z=3%3;
s=5%4;
a=2%0;
Floating point exception (core dumped)
```

- *Solution*

New case is added to the switch block to give hints to users.

Original code:

```
case MOD:       if ((ex(p->opr.op[0]) - (int)ex(p->opr.op[0]) != 0) ||
                    (ex(p->opr.op[1]) - (int)ex(p->opr.op[1]) != 0))
                {
                    printf("\t\tError: Operand must be an integer\n");
                    return 0;
                }
                else
                    return ((int)(ex(p->opr.op[0])) % ((int)ex(p->opr.op[1])));
```

Modified code:

```
case MOD:       if ((ex(p->opr.op[0]) - (int)ex(p->opr.op[0]) != 0) ||
                    (ex(p->opr.op[1]) - (int)ex(p->opr.op[1]) != 0))
                {
                    printf("\t\tError: Operand must be an integer\n");
                    return 0;
                }
            if (ex(p->opr.op[1]) == 0)
                {
                    printf("\t\tError: The second operator can not be zero\n");
                    return 0;
                }
                else
                    return ((int)(ex(p->opr.op[0])) % ((int)ex(p->opr.op[1])));
```

Correct output:

```
y=2%1;
z=2%4;
print x;
                Output: 0.000000
print z;
                Output: 2.000000
s=2%0;
                Error: The second operator can not be zero
```

Problem E

- *Description*

For operation **"++"** and **"--"**, a problem arose when we implemented it. In C programming language, "printf ("%d", ++x)" or "printf ("%d", --x)" not only gives the output, but also means the value of x will be updated and stored. Some technique must be designed to update the value of operand automatically.

- *Solution*

To solve the problem discussed above, assignment to the operand could be used, that we could assign the value to the operand right after the operation **"++"** or **"--"**.

Correct output:

```
x = 1;
print x;
                Output: 1.000000
x++;
print x;
                Output: 1.000000
x = x++;
print x;
                Output: 2.000000
```

Problem F

- *Description*

The original code only allows the postfix increment and decrement. The prefix increment and decrement is recognized as syntax error.

Original output:

```
x=1--;
y=2++;
print x;
                Output: 0.000000
print y;
                Output: 3.000000
x=--1;
syntax error
```

- *Solution*

Without change the previous code, the following two lines is added to the calc.y file to implement prefix increment and decrement.

Original code:

```
expr
    | expr INCR           { $$ = opr(INCR, 1, $1); }
    | expr DECR           { $$ = opr(DECR, 1, $1); }
```

Modified code:

```
expr
    | expr INCR             { $$ = opr(INCR, 1, $1); }
    | expr DECR             { $$ = opr(DECR, 1, $1); }
    | INCR expr             { $$ = opr(INCR, 1, $2); }
    | DECR expr             { $$ = opr(DECR, 1, $2); }
```

Correct output:

```
x=--1;
y=++2;
print x;
                Output: 0.000000
print y;
                Output: 3.000000
```

Problem G

- *Description*

Negative inputs were not considered when creating the SQRT function, Then the output is not a recognizable symbol in this case.

- *Error*

```
x=sqrt(1);
y=sqrt(0);
z=sqrt(-1);
print x;
                Output: 1.000000
print y;
                Output: 0.000000
print z;
                Output: -nan
```

- *Solution*

The conditional statement to judge the sign of the input is added to the original code to solve the problem.

Original code:

```
case SQRT:      return sqrt(ex(p->opr.op[0]));
```

Modified code:

```
case SQRT:      if (ex(p->opr.op[0]) < 0)
                {
                    printf("\t\tError: Must be a non_negative number\n");
                    return 0;
                }
                else
                    return sqrt(ex(p->opr.op[0]));
```

Correct output:

It can be observed that the negative input can be recognized and the corresponding hint is presented.

```
x=sqrt(10);
y=sqrt(-1);
                 Error: Must be a non_negative number
print x;
                 Output: 3.162278
```

Since the independent variable of the logarithmic function must be mathematically positive, the same problem occurs in the LN function and the LOG function. We modify the LN function and LOG function in the same way, and finally got the correct output.

Original output:

```
x=log(0);
print  x;
                Output: -inf
x=ln(0);
print x;
                Output: -inf
```

Correct output:

```
x=ln(-1);
                 Error: Must be a positive number
y=log(-1);
                 Error: Must be a positive number
x=ln(0);
                 Error: Must be a positive number
y=log(0);
                 Error: Must be a positive number
```

Problem H

- *Description*

Initially, the program cannot recognize the mathematical constant number PI and E.

- *Error*

```
x=ln(E);
                 Warning: Unknown character
syntax error
```

```
x=ln(PI);
                 Warning: Unknown character
                 Warning: Unknown character
syntax error
```

- *Solution*

We define the two constant number in advance

Code added to calc.l:

```
PI          {
                yylval.iValue = 3.141592654;
                return DOUBLE;
            }

E           {
                yylval.iValue = 2.718281828;
                return DOUBLE;
            }
```

Correct Output：

```
x=ln(PI);
print x;
                 Output: 1.144730
```

Question I

- *Description*

The wrong setting of the priority in the calc.y file, It causes the input operators with different priorities the wrong output of graph.c file.

- *Error*

- *Solution*

Set the priority of UMINUS to a higher level than pow function, and graph.c can get the correct syntax tree.

Original Code：

```
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%left POW MOD
```

Modified Code：

```
%left '+' '-'
%left '*' '/'
%left POW MOD
%nonassoc UMINUS
```

Correct Output：

4. Conclusion

By referring to the detailed tutorial given by Tom Niemann, the methods and instructions for building a basic compiler was mastered by us and a calculator was implemented subsequently. To improve the robustness of the calculator, we referred to a lot of materials and tutorials to learn the methods and ways for adding functions, and then we discussed, debugged and suggested together with trial and error.

Many difficulties arose during the process of we adding codes, but hard work paid off, an advanced “fresh baked” calculator came out with an official manual directing users to use our amended and added functions. In the whole process, we did not only learn more about how to work together for a common goal and how to explain and demonstrate clearly what we had accomplished, but also improved the understanding of knowledge related to compiler. For instance, the Lexical Analysis and Syntax Analysis became clearer to us, these processes seemed far from us if we just read the definitions of them.

It could be said that, we gained knowledge, programming ability, team spirit and also happiness from this lab project.

5. Recommendation

Although the advanced calculator we built is able to function a lot of operations, many complex operations have still not been involved yet. Hence, we would also like to go a little bit further from what we have achieved now.

1. For future advancement, complex number, integral and derivatives may be included for calculation.
2. Error messages for syntax tree drawing may also be included for robustness.
3. Besides, when too many operators are put together, some errors may arise, how to perfect the codes also remains to be solved.
4. Additionally, when the operand overflows, that is, it outranges the **MAX** or **MIN**, the calculator may crash, so a bigger range may be declared.

6. References

[1] Bison for Semantic Analysis. Retrieved from https://pandolia.net/tinyc/ch13_bison.html.

[2] Bison The YACC-compatible Parser Generator. Retrieved from http://dinosaur.compilertools.net/bison/index.html.

[3] Flex, version 2.5. Retrieved from http://dinosaur.compilertools.net/flex/index.html.

[4] Lesk, M. E. and E. Schmidt. [1975]. Lex - A Lexical Analyzer Generator. Retrieved from http://dinosaur.compilertools.net/lex/index.html.

[5] The Lex & Yacc Page. Retrieved from http://dinosaur.compilertools.net/#lex.

[6] What is Lex? What is Yacc? Retrieved from https://luv.asn.au/overheads/lex_yacc/index.html.

[7] Yacc: Yet Another Compiler. Johnson, Stephen C. [1975]. Retrieved from http://dinosaur.compilertools.net/yacc/index.html.

[8] Lex & Yacc Tutorial. Niemann, Tom. [2015].

7. Appendix I - Codes

calc.h

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct
{
    double value;                   /* value of constant */
} conNodeType;

/* identifiers */
typedef struct
{
    int i;                      /* subscript to sym array */
} idNodeType;

/* operators */
typedef struct
{
    int oper;                   /* operator */
    int nops;                   /* number of operands */
    struct nodeTypeTag *op[1];  /* operands, extended at runtime */
} oprNodeType;

typedef struct nodeTypeTag
{
    nodeEnum type;              /* type of node */
    union
    {
        conNodeType con;        /* constants */
        idNodeType id;          /* identifiers */
        oprNodeType opr;        /* operators */
    };
} nodeType;

extern double sym[26];
```

calc.l

```
%{
    #include <stdlib.h>
    #include <string.h>
    #include "calc3.h"
    #include "calc3.tab.h"
    #define MAX 2147483647
    #define MIN -2147483648
    void yyerror(char *);
    int check_overflow(double);
    double btof(char [], int);
    double otof(char [], int);
    double htof(char [], int);
    double value;
    int size;

%}


%%

[a-z]       {
                yylval.sIndex = *yytext - 'a';
                return VARIABLE;
            }

0           {
                yylval.iValue = atof(yytext);
                return DOUBLE;
            }

[0-9]+(\.[0-9]+)?   {
                size = strlen(yytext);
                value = atof(yytext);
                if (check_overflow(value))
                    yylval.iValue = value;
                else
                {
                    yylval.iValue = 0;
                    yyerror("\t\tWarning: Overflow, value ignored");
                }
                return DOUBLE;
            }

PI          {
                yylval.iValue = 3.141592654;
                return DOUBLE;
            }

E           {
                yylval.iValue = 2.718281828;
                return DOUBLE;
            }

b[0|1]+     {
                size = strlen(yytext);
                value = btof(yytext, size);
                if (check_overflow(value))
                    yylval.iValue = value;
                else
                {
                    yylval.iValue = 0;
                    yyerror("\t\tWarning: Overflow, value ignored");
                }
                return DOUBLE;
            }

o[0-7]+     {
                size = strlen(yytext);
                value = otof(yytext, size);
                if (check_overflow(value))
                    yylval.iValue = value;
                else
                {
                    yylval.iValue = 0;
                    yyerror("\t\tWarning: Overflow, value ignored");
                }
                return DOUBLE;
            }

h[0-9A-F]+  {
                size = strlen(yytext);
                value = htof(yytext, size);
                if (check_overflow(value))
                    yylval.iValue = value;
                else
                {
                    yylval.iValue = 0;
                    yyerror("\t\tWarning: Overflow, value ignored");
                }
                return DOUBLE;
            }

"++"            return INCR;
"--"            return DECR;

[-()<>=+*/!;{}.]    return *yytext;

">="            return GE;
"<="            return LE;
"=="            return EQ;
"!="            return NE;
"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;
"abs"           return ABS;
"exit"          return EXIT;
"sin"           return SIN;
"cos"           return COS;
"tan"           return TAN;
"print_all"     return W;
"print_bin"     return X;
"print_oct"     return Y;
"print_hex"     return Z;
"print_allf"    return V;
"reset"         return RESET;
"sqrt"|"SQRT"   return SQRT;
"^"             return POW;
"%"             return MOD;
"exp"|"EXP"     return EXP;
"cube"          return CUBE;
"log"|"LOG"     return LOG;
"ln"|"LN"       return LN;
"ceil"          return CEIL;
"floor"         return FLOOR;

[ \t\n]+        ;   /* ignore whitespace */

.               yyerror("\t\tWarning: Unknown character");

%%

double btof(char c[], int s)
{
    int i, j;
    double temp = 1;
    double result = 0;
    for (i = s-2; i >= 0; i--)
    {
        if (c[s-i-1] == '1')
        {
            for (j = 0; j < i; j++) temp *= 2;
            result += temp;
            temp = 1;
        }
    }
    return result;
}

double otof(char c[], int s)
{
    int i, j, k;
    double temp = 1;
    double result = 0;
    for (i = s-2; i >= 0; i--)
    {
        switch (c[s-i-1])
        {
            case '0':   k = 0; break;
            case '1':   k = 1; break;
            case '2':   k = 2; break;
            case '3':   k = 3; break;
            case '4':   k = 4; break;
            case '5':   k = 5; break;
            case '6':   k = 6; break;
            case '7':   k = 7; break;
            default: break;
        }
        for (j = 0; j < i; j++) temp *= 8;
        result += k * temp;
        temp = 1;
    }
    return result;
}

double htof(char c[], int s)
{
    int i, j, k;
    double temp = 1;
    double result = 0;
    for (i = s-2; i >= 0; i--)
    {
        switch (c[s-i-1])
        {
            case '0':   k = 0; break;
            case '1':   k = 1; break;
            case '2':   k = 2; break;
            case '3':   k = 3; break;
            case '4':   k = 4; break;
            case '5':   k = 5; break;
            case '6':   k = 6; break;
            case '7':   k = 7; break;
            case '8':   k = 8; break;
            case '9':   k = 9; break;
            case 'A':   k = 10; break;
            case 'B':   k = 11; break;
            case 'C':   k = 12; break;
            case 'D':   k = 13; break;
            case 'E':   k = 14; break;
            case 'F':   k = 15; break;
            default: break;
        }
        for (j = 0; j < i; j++) temp *= 16;
        result += k * temp;
        temp = 1;
    }
    return result;
}

int check_overflow(double a)
{
    if (a > MAX || a < MIN)
        return 0;
    else
        return 1;
}

int yywrap(void)
{
    return 1;
}
```

calc.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <stdarg.h>
    #include "calc3.h"

    /* prototypes */
    nodeType *opr(int oper, int nops, ...);
    nodeType *id(int i);
    nodeType *con(double value);
    void freeNode(nodeType *p);
    double ex(nodeType *p);
    int yylex(void);

    void yyerror(char *s);
    double sym[26];                 /* symbol table */
%}

%union
{
    double iValue;                  /* integer value */
    char sIndex;                    /* symbol table index */
    nodeType *nPtr;                 /* node pointer */
};

%token <iValue> DOUBLE
%token <sIndex> VARIABLE
%token WHILE IF PRINT EXIT V W X Y Z RESET
%token ABS SIN COS TAN SQRT CUBE EXP LOG LN CEIL FLOOR
%nonassoc IFX
%nonassoc ELSE

%right '='
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%left POW MOD
%nonassoc UMINUS
%left '!' INCR DECR

%type <nPtr> stmt expr stmt_list

%%

program:
    function EXIT               { printf("THANK YOU!\n"); exit(0); }
    ;

function:
    function stmt               { ex($2); freeNode($2); }
    | /* NULL */
    ;

stmt:
    ';'                         { $$ = opr(';', 2, NULL, NULL); }
    | expr ';'                  { $$ = $1; }
    | PRINT expr ';'            { $$ = opr(PRINT, 1, $2); }
    | W ';'                     { $$ = opr(W, 0); }
    | V ';'                     { $$ = opr(V, 0); }
    | RESET ';'                 { $$ = opr(RESET, 0); }
    | X expr ';'                { $$ = opr(X, 1, $2); }
    | Y expr ';'                { $$ = opr(Y, 1, $2); }
    | Z expr ';'                { $$ = opr(Z, 1, $2); }
    | VARIABLE '=' expr ';'     { $$ = opr('=', 2, id($1), $3); }
    | WHILE '(' expr ')' stmt   { $$ = opr(WHILE, 2, $3, $5); }
    | IF '(' expr ')' stmt %prec IFX    { $$ = opr(IF, 2, $3, $5); }
    | IF '(' expr ')' stmt ELSE stmt    { $$ = opr(IF, 3, $3, $5, $7);
    | '{' stmt_list '}'         { $$ = $2; }
    ;

stmt_list:
    stmt                        { $$ = $1; }
    | stmt_list stmt            { $$ = opr(';', 2, $1, $2); }
    ;

expr:
    DOUBLE                      { $$ = con($1); }
    | VARIABLE                  { $$ = id($1); }
    | '-' expr %prec UMINUS     { $$ = opr(UMINUS, 1, $2); }
    | expr '+' expr             { $$ = opr('+', 2, $1, $3); }
    | expr '-' expr             { $$ = opr('-', 2, $1, $3); }
    | expr '*' expr             { $$ = opr('*', 2, $1, $3); }
    | expr '/' expr             { $$ = opr('/', 2, $1, $3); }
    | expr '<' expr             { $$ = opr('<', 2, $1, $3); }
    | expr '>' expr             { $$ = opr('>', 2, $1, $3); }
    | expr GE expr              { $$ = opr(GE, 2, $1, $3); }
    | expr LE expr              { $$ = opr(LE, 2, $1, $3); }
    | expr NE expr              { $$ = opr(NE, 2, $1, $3); }
    | expr EQ expr              { $$ = opr(EQ, 2, $1, $3); }
    | '(' expr ')'              { $$ = $2; }
    | ABS '(' expr ')'          { $$ = opr(ABS, 1, $3); }
    | expr '!'                  { $$ = opr('!', 1, $1); }
    | SIN '(' expr ')'          { $$ = opr(SIN, 1, $3); }
    | COS '(' expr ')'          { $$ = opr(COS, 1, $3); }
    | TAN '(' expr ')'          { $$ = opr(TAN, 1, $3); }
    | expr POW expr             { $$ = opr(POW, 2, $1, $3); }
    | expr MOD expr             { $$ = opr(MOD, 2, $1, $3); }
    | SQRT '(' expr ')'         { $$ = opr(SQRT, 1, $3); }
    | CUBE '(' expr ')'         { $$ = opr(CUBE, 1, $3); }
    | LOG '(' expr ')'          { $$ = opr(LOG,1, $3); }
    | EXP '(' expr ')'          { $$ = opr(EXP, 1, $3); }
    | LN '(' expr ')'           { $$ = opr(LN, 1, $3); }
    | expr INCR                 { $$ = opr(INCR, 1, $1); }
    | expr DECR                 { $$ = opr(DECR, 1, $1); }
    | INCR expr                 { $$ = opr(INCR, 1, $2); }
    | DECR expr                 { $$ = opr(DECR, 1, $2); }
    | CEIL '(' expr ')'         { $$ = opr(CEIL, 1, $3); }
    | FLOOR '(' expr ')'        { $$ = opr(FLOOR, 1, $3); }
    ;

%%

#define SIZEOF_NODETYPE     ((char *)&p->con - (char *)p)

nodeType *con(double value)
{
    nodeType *p;

    /* allocate node */
    if ((p = malloc(sizeof(nodeType))) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeCon;
    p->con.value = value;

    return p;
}

nodeType *id(int i)
{
    nodeType *p;

    /* allocate node */
    if ((p = malloc(sizeof(nodeType))) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeId;
    p->id.i = i;

    return p;

}

nodeType *opr(int oper, int nops, ...)
{
    va_list ap;
    nodeType *p;
    int i;

    /* allocate node, extending op array */
    if ((p = malloc(sizeof(nodeType) +
        (nops-1) * sizeof(nodeType *))) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeOpr;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}

void freeNode(nodeType *p)
{
    int i;

    if (!p) return;
    if (p->type == typeOpr)
    {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free(p);
}

void yyerror(char *s)
{
    fprintf(stdout, "%s\n", s);
}

int main(void)
{
    yyparse();
    return 0;
}
```

interpreter.c

```c
#include <stdio.h>
#include <math.h>
#include "calc3.h"
#include "calc3.tab.h"
#define pi 3.141592654

int tag[26] = {0};

double factorial(double num)
{
    if (num == 0 || num == 1)
        return 1;
    else
        return num * factorial(num-1);
}

void ftob(double a)
{
    long b = (long)a;
    int neg_flag = 0;
    if (a < 0)
    {
        b *= -1;
        neg_flag = 1;
    }
    int c[32];
    char d[32];
    int i, j;
    for (i = 0; b > 0; i++, b /= 2)
        c[i] = b % 2;
    i--;
    for (j = 0;i >= 0; i--,j++)
    {
        if (c[i] == 1) d[j] = '1';
        else d[j] = '0';
    }
    d[j] = '\0';
    if (neg_flag)
        printf("-bx%-32s", d);
    else
        printf("bx%-33s", d);
}

void ftoo(double a)
{
    long b = (long)a;
    int neg_flag = 0;
    if (a < 0)
    {
        b *= -1;
        neg_flag = 1;
    }
    int c[12];
    char d[12];
    int i, j;
    for (i = 0; b > 0; i++, b /= 8)
        c[i] = b % 8;
    i--;
    for (j = 0;i >= 0; i--,j++)
    {
        switch (c[i])
        {
            case 0: d[j] = '0'; break;
            case 1: d[j] = '1'; break;
            case 2: d[j] = '2'; break;
            case 3: d[j] = '3'; break;
            case 4: d[j] = '4'; break;
            case 5: d[j] = '5'; break;
            case 6: d[j] = '6'; break;
            case 7: d[j] = '7'; break;
        }
    }
    d[j] = '\0';
    if (neg_flag)
        printf("-ox%-12s", d);
    else
        printf("ox%-13s", d);
}

void ftoh(double a)
{
    long b = (long)a;
    int neg_flag = 0;
    if (a < 0)
    {
        b *= -1;
        neg_flag = 1;
    }
    int c[8];
    char d[8];
    int i, j;
    for (i = 0; b > 0; i++, b /= 16)
        c[i] = b % 16;
    i--;
    for (j = 0;i >= 0; i--,j++)
    {
        switch (c[i])
        {
            case 0: d[j] = '0'; break;
            case 1: d[j] = '1'; break;
            case 2: d[j] = '2'; break;
            case 3: d[j] = '3'; break;
            case 4: d[j] = '4'; break;
            case 5: d[j] = '5'; break;
            case 6: d[j] = '6'; break;
            case 7: d[j] = '7'; break;
            case 8: d[j] = '8'; break;
            case 9: d[j] = '9'; break;
            case 10: d[j] = 'A'; break;
            case 11: d[j] = 'B'; break;
            case 12: d[j] = 'C'; break;
            case 13: d[j] = 'D'; break;
            case 14: d[j] = 'E'; break;
            case 15: d[j] = 'F'; break;
        }
    }
    d[j] = '\0';
    if (neg_flag)
        printf("-hx%-8s", d);
    else
        printf("hx%-9s", d);
}

void printall(void)
{
    int i, flag = 1;
    printf("\tOutput:\n");
    for (i = 0; i < 26; i++)
    {
        if (tag[i] == 1)
        {
            printf("\t\t%c = %lf\n", 'a' + i, sym[i]);
            flag = 0;
        }
    }
    if (flag)
        printf("\t\tNo data found\n");
}

void printallf(void)
{
    int i, flag = 1;
    printf("\tOutput:\n");
    for (i = 0; i < 26; i++)
    {
        if (tag[i] == 1)
        {
            printf("\t%c = %-10.2lf", 'a' + i, sym[i]);
            printf(" "); ftoh(sym[i]);
            printf(" "); ftoo(sym[i]);
            printf(" "); ftob(sym[i]);
            printf("\n");
            flag = 0;
        }

    }
    if (flag)
        printf("\t\tNo data\n");
}
```

```
161     void reset(void)
162     {
163         int i;
164         for (i = 0; i < 26; i++)
165         {
166             sym[i] = 0;
167             tag[i] = 0;
168         }
169     }
170
171     double cube(double num)
172     {
173         return num * num * num;
174     }
175
176     double ex(nodeType *p)
177     {
178         if (!p) return 0;
179         switch (p->type)
180         {
181             case typeCon:       return p->con.value;
182             case typeId:        return sym[p->id.i];
183             case typeOpr:
184             switch (p->opr.oper)
185             {
186                 case WHILE:     while(ex(p->opr.op[0]))
187                                     ex(p->opr.op[1]);
188                                 return 0;
189                 case IF:        if (ex(p->opr.op[0]))
190                                     ex(p->opr.op[1]);
191                                 else if (p->opr.nops > 2)
192                                     ex(p->opr.op[2]);
193                                 return 0;
194                 case PRINT:     printf("\t\tOutput: %lf\n", ex(p->opr.op[0]));
195                                 return 0;
196                 case V:         printallf();
197                                 return 0;
198                 case W:         printall();
199                                 return 0;
200                 case X:         printf("\t\tOutput: ");
201                                 ftob(ex(p->opr.op[0]));
202                                 printf("\n");
203                                 return 0;
204                 case Y:         printf("\t\tOutput: ");
205                                 ftoo(ex(p->opr.op[0]));
206                                 printf("\n");
207                                 return 0;
208                 case Z:         printf("\t\tOutput: ");
209                                 ftoh(ex(p->opr.op[0]));
210                                 printf("\n");
211                                 return 0;
212                 case RESET:     reset();
213                                 printf("\t\tAll varaibles cleared\n");
214                                 return 0;
215                 case ABS:       if (ex(p->opr.op[0]) >= 0)
216                                     return ex(p->opr.op[0]);
217                                 else
218                                     return -ex((p->opr.op[0]));
219                 case SIN:       return sin(ex(p->opr.op[0]) * pi / 180);
220                 case COS:       return cos(ex(p->opr.op[0]) * pi / 180);
221                 case TAN:       return tan(ex(p->opr.op[0]) * pi / 180);
222                 case ';':       ex(p->opr.op[0]);
223                                 return ex(p->opr.op[1]);
224                 case '=':       tag[p->opr.op[0]->id.i] = 1;
225                                 return sym[p->opr.op[0]->id.i] = ex(p->opr.op[1]);
226                 case UMINUS:    return -ex(p->opr.op[0]);
227                 case '+':       return ex(p->opr.op[0]) + ex(p->opr.op[1]);
228                 case '-':       return ex(p->opr.op[0]) - ex(p->opr.op[1]);
229                 case '*':       return ex(p->opr.op[0]) * ex(p->opr.op[1]);
230                 case '/':       if (ex(p->opr.op[1]) == 0)
231                                 {
232                                     printf("\t\tError: Connot divide by 0\n");
233                                     return 0;
234                                 }
235                                 else
236                                     return ex(p->opr.op[0]) / ex(p->opr.op[1]);
237                 case '<':       return ex(p->opr.op[0]) < ex(p->opr.op[1]);
238                 case '>':       return ex(p->opr.op[0]) > ex(p->opr.op[1]);
239                 case GE:        return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
240                 case LE:        return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
241                 case NE:        return ex(p->opr.op[0]) != ex(p->opr.op[1]);
242                 case EQ:        return ex(p->opr.op[0]) == ex(p->opr.op[1]);
243                 case '!':       if (ex(p->opr.op[0]) - (int)ex(p->opr.op[0]) != 0)
244                                 {
245                                     printf("\t\tError: Must be an integer\n");
246                                     return 0;
247                                 }
248                                 else
249                                     return factorial(ex(p->opr.op[0]));
250                 case POW:       return pow(ex(p->opr.op[0]),ex(p->opr.op[1]));
```

```
251             case MOD:       if ((ex(p->opr.op[0]) - (int)ex(p->opr.op[0]) != 0) ||
252                                 (ex(p->opr.op[1]) - (int)ex(p->opr.op[1]) != 0))
253                             {
254                                 printf("\t\tError: Operand must be an integer\n");
255                                 return 0;
256                             }
257                             else
258                                 return ((int)(ex(p->opr.op[0])) % ((int)ex(p->opr.op[1])));
259             case EXP:       return exp(ex(p->opr.op[0]));
260             case LOG:       if (ex(p->opr.op[0]) <= 0)
261                             {
262                                 printf("\t\tError: Must be a positive number\n");
263                                 return 0;
264                             }
265                             else
266                                 return log10(ex(p->opr.op[0]));
267             case LN:        if (ex(p->opr.op[0]) <= 0)
268                             {
269                                 printf("\t\tError: Must be a positive number\n");
270                                 return 0;
271                             }
272                             else
273                                 return log(ex(p->opr.op[0]));
274             case SQRT:      if (ex(p->opr.op[0]) < 0)
275                             {
276                                 printf("\t\tError: Must be a non-negative number\n");
277                                 return 0;
278                             }
279                             else
280                                 return sqrt(ex(p->opr.op[0]));
281             case CUBE:      return cube(ex(p->opr.op[0]));
282             case INCR:      return ex(p->opr.op[0]) + 1;
283             case DECR:      return ex(p->opr.op[0]) - 1;
284             case CEIL:      return ceil(ex(p->opr.op[0]));
285             case FLOOR:     return floor(ex(p->opr.op[0]));
286         }
287     }
288     return 0;
289 }
290
```

compiler.c

```
1   #include <stdio.h>
2   #include "calc3.h"
3   #include "calc3.tab.h"
4
5   static int lbl;
6
7   double ex(nodeType *p)
8   {
9       int lbl1, lbl2;
10
11      if (!p) return 0;
12      switch (p->type)
13      {
14          case typeCon:       printf("\tpush\t%lf\n", p->con.value);
15                              break;
16          case typeId:        printf("\tpush\t%c\n", p->id.i + 'a');
17                              break;
18          case typeOpr:
19          switch(p->opr.oper)
20          {
21              case WHILE:
22                  printf("L%03d:\n", lbl1 = lbl++);
23                  ex(p->opr.op[0]);
24                  printf("\tjz\tL%03d\n", lbl2 = lbl++);
25                  ex(p->opr.op[1]);
26                  printf("\tjmp\tL%03d\n", lbl1);
27                  printf("L%03d:\n", lbl2);
28                  break;
29              case IF:
30                  ex(p->opr.op[0]);
31                  if (p->opr.nops > 2)
32                  {
33                      /* if else */
34                      printf("\tjz\tL%03d\n", lbl1 = lbl++);
35                      ex(p->opr.op[1]);
36                      printf("\tjmp\tL%03d\n", lbl2 = lbl++);
37                      printf("L%03d:\n", lbl1);
38                      ex(p->opr.op[2]);
39                      printf("L%03d:\n", lbl2);
40                  }
41                  else
42                  {
43                      /* if */
44                      printf("\tjz\tL%03d\n", lbl1 = lbl++);
45                      ex(p->opr.op[1]);
46                      printf("L%03d:\n", lbl1);
47                  }
48                  break;
49              case PRINT:
50                  ex(p->opr.op[0]);
```

```
                printf("\tprint\n");
                break;
            case V:
                printf("\tprint_all\n");
                break;
            case W:
                printf("\tprint_all_form\n");
                break;
            case X:
                ex(p->opr.op[0]);
                printf("\tprint_bin\n");
                break;
            case Y:
                ex(p->opr.op[0]);
                printf("\tprint_oct\n");
                break;
            case Z:
                ex(p->opr.op[0]);
                printf("\tprint_hex\n");
                break;
            case RESET:
                printf("\tAll varaibles cleared\n");
                break;
            case ABS:
                ex(p->opr.op[0]);
                printf("\tABS\n");
                break;
            case SIN:
                ex(p->opr.op[0]);
                printf("\tSIN\n");
                break;
            case COS:
                ex(p->opr.op[0]);
                printf("\tCOS\n");
                break;
            case TAN:
                ex(p->opr.op[0]);
                printf("\tTAN\n");
                break;
            case '=':
                ex(p->opr.op[1]);
                printf("\tpop\t%c\n", p->opr.op[0]->id.i + 'a');
                break;
            case UMINUS:
                ex(p->opr.op[0]);
                printf("\tneg\n");
                break;
            case '!':
                ex(p->opr.op[0]);
                printf("\tfactorial\n");
                break;
            case EXP:
                ex(p->opr.op[0]);
                printf("\tEXP\n");
                break;
            case SQRT:
                ex(p->opr.op[0]);
                printf("\tSQRT\n");
                break;
            case CUBE:
                ex(p->opr.op[0]);
                printf("\tCUBE\n");
                break;
            case LOG:
                ex(p->opr.op[0]);
                printf("\tLOG\n");
                 break;
             case LN:
                 ex(p->opr.op[0]);
                 printf("\tLN\n");
                 break;
             case INCR:
                 ex(p->opr.op[0]);
                 printf("\tpush\t%lf\n", 1.0);
                 printf("\tadd\n");
                 break;
             case DECR:
                 ex(p->opr.op[0]);
                 printf("\tpush\t%lf\n", 1.0);
                 printf("\tsub\n");
                 break;
             case CEIL:
                 ex(p->opr.op[0]);
                 printf("\tCEIL\n");
                 break;
             case FLOOR:
                 ex(p->opr.op[0]);
                 printf("\tFLOOR\n");
                 break;
             default:
                 ex(p->opr.op[0]);
                 ex(p->opr.op[1]);
                 switch (p->opr.oper)
                 {
                     case '+':   printf("\tadd\n"); break;
                     case '-':   printf("\tsub\n"); break;
                     case '*':   printf("\tmul\n"); break;
                     case '/':   printf("\tdiv\n"); break;
                     case '<':   printf("\tcompLT\n"); break;
                     case '>':   printf("\tcompGT\n"); break;
                     case GE:    printf("\tcompGE\n"); break;
                     case LE:    printf("\tcompLE\n"); break;
                     case NE:    printf("\tcompNE\n"); break;
                     case EQ:    printf("\tcompEQ\n"); break;
                     case POW:   printf("\tpow\n"); break;
                     case MOD:   printf("\tmod\n"); break;
                 }
         }
     }
     return 0;
 }

```

graph.c

```c
/* source code courtesy of Frank Thomas Braun */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include "calc3.h"
#include "calc3.tab.h"

int del = 1;    /* distance of graph columns */
int eps = 3;    /* distance of graph lines */

/* interface for drawing (can be replaced by "real" graphic using GD or other) */
void graphInit (void);
void graphFinish();
void graphBox (char *s, int *w, int *h);
void graphDrawBox (char *s, int c, int l);
void graphDrawArrow (int c1, int l1, int c2, int l2);

/* recursive drawing of the syntax tree */
void exNode (nodeType *p, int c, int l, int *ce, int *cm);

/*****************************************************/

/* main entry point of the manipulation of the syntax tree */
int ex (nodeType *p)
{
    int rte, rtm;
    graphInit();
    exNode (p, 0, 0, &rte, &rtm);
    graphFinish();
    return 0;
}


/*c----cm---ce---->                       drawing of leaf-nodes
l leaf-info
*/

/*c---------------cm--------------ce---->   drawing of non-leaf-nodes
l             node-info
*               |
*     ------------     ...----
*     |       |              |
*     v       v              v
*  child1  child2  ...     child-n
*         che     che            che
*cs       cs      cs             cs
*
*/

void exNode
    (
        nodeType *p,
        int c, int l,       /* start column and line of node */
        int *ce, int *cm    /* resulting end column and mid of node */
    )
{
    int w, h;       /* node width and height */
    char *s;        /* node text */
    int cbar;       /* "real" start column of node (centred above subnodes) */
    int k;          /* child number */
    int che, chm;   /* end column and mid of children */
    int cs;         /* start column of children */
    char word[20];  /* extended node text */

    if (!p) return;

    strcpy(word, "???"); /* should never appear */
    s = word;
    switch (p->type)
    {
        case typeCon:   sprintf (word, "c(%.3f)", p->con.value); break;
        case typeId:    sprintf (word, "id(%c)", p->id.i + 'A'); break;
        case typeOpr:
            switch (p->opr.oper)
            {
                case WHILE:     s = "while"; break;
                case IF:        s = "if"; break;
                case PRINT:     s = "print"; break;
                case V:         s = "print all variable in all form\n"; break;
                case W:         s = "print all variable\n"; break;
                case X:         s = "print_bin"; break;
                case Y:         s = "print_oct"; break;
                case Z:         s = "print_hex"; break;
                case RESET:     s = "all variable reset"; break;
                case SIN:       s = "SIN"; break;
                case COS:       s = "COS"; break;
                case TAN:       s = "TAN"; break;
                case ';':       s = "[;]"; break;
                case '=':       s = "[=]"; break;
                case UMINUS:    s = "[_]"; break;
                case '+':       s = "[+]"; break;
                case '-':       s = "[-]"; break;
                case '*':       s = "[*]"; break;
                case '/':       s = "[/]"; break;
                case '<':       s = "[<]"; break;
                case '>':       s = "[>]"; break;
                case GE:        s = "[>=]"; break;
                case LE:        s = "[<=]"; break;
```

```
                    case NE:        s = "[!=]"; break;
                    case EQ:        s = "[==]"; break;
                    case ABS:       s = "[ABS]"; break;
                    case '!':       s = "[!]"; break;
                    case POW:       s = "[^]"; break;
                    case MOD:       s = "[%]"; break;
                    case SQRT:      s = "[SQRT]"; break;
                    case CUBE:      s = "[CUBE]"; break;
                    case LOG:       s = "[LOG]"; break;
                    case EXP:       s = "[EXP]"; break;
                    case LN:        s = "[LN]"; break;
                    case INCR:      s = "[++]"; break;
                    case DECR:      s = "[--]"; break;
                    case CEIL:      s = "[CEIL]"; break;
                    case FLOOR:     s = "[FLOOR]"; break;
                }
                break;
        }
        /* construct node text box */
        graphBox (s, &w, &h);
        cbar = c;
        *ce = c + w;
        *cm = c + w / 2;

        /* node is leaf */
        if (p->type == typeCon || p->type == typeId || p->opr.nops == 0)
        {
            graphDrawBox(s, cbar, l);
            return;
        }

        /* node has children */
        cs = c;
        for (k = 0; k < p->opr.nops; k++)
        {
            exNode (p->opr.op[k], cs, l+h+eps, &che, &chm);
            cs = che;
        }

        /* total node width */
        if (w < che - c)
        {
            cbar += (che - c - w) / 2;
            *ce = che;
            *cm = (c + che) / 2;
        }

        /* draw node */
        graphDrawBox(s, cbar, l);

        /* draw arrows (not optimal: children are drawn a second time) */
        cs = c;
        for (k = 0; k < p->opr.nops; k++)
        {
            exNode(p->opr.op[k], cs, l+h+eps, &che, &chm);
            graphDrawArrow(*cm, l+h, chm, l+h+eps-1);
            cs = che;
        }
    }

    /* interface for drawing */

    #define lmax 200
    #define cmax 200

    char graph[lmax][cmax]; /* array for ASCII-Graphic */
    int graphNumber = 0;

    void graphTest(int l, int c)
    {
        int ok;
        ok = 1;
        if (l < 0) ok = 0;
        if (l >= lmax) ok = 0;
        if (c < 0) ok = 0;
        if (c >= cmax) ok = 0;
        if (ok) return;
        printf("\n+++error: l=%d, c=%d not in drawing rectangle 0, 0 ... %d, %d",
            l, c, lmax, cmax);
        exit (1);
    }

    void graphInit(void)
    {
        int i, j;
        for (i = 0; i < lmax; i++)
        {
            for (j = 0; j < cmax; j++)
            {
                graph[i][j] = ' ';
            }
        }
    }

    void graphFinish()
    {
        int i, j;
        for (i = 0; i < lmax; i++)
        {
            for (j = cmax-1; j > 0 && graph[i][j] == ' '; j--);
```

```
            graph[i][cmax-1] = 0;
            if (j < cmax-1) graph[i][j+1] = 0;
            if (graph[i][j] == ' ') graph[i][j] = 0;
        }
        for (i = lmax-1; i > 0 && graph[i][0] == 0; i--)
            ;
        printf ("\n\nGraph %d:\n", graphNumber++);
        for (j = 0; j <= i; j++)
            printf ("\n%s", graph[j]);
        printf("\n");
    }

    void graphBox(char *s, int *w, int *h)
    {
        *w = strlen (s) + del;
        *h = 1;
    }

    void graphDrawBox(char *s, int c, int l)
    {
        int i;
        graphTest(l, c+strlen(s)-1+del);
        for (i = 0; i < strlen (s); i++)
        {
            graph[l][c+i+del] = s[i];
        }
    }

    void graphDrawArrow(int c1, int l1, int c2, int l2)
    {
        int m;
        graphTest (l1, c1);
        graphTest (l2, c2);
        m = (l1 + l2) / 2;
        while (l1 != m)
        {
            graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
        }
        while (c1 != c2)
        {
            graph[l1][c1] = '-'; if (c1 < c2) c1++; else c1--;
        }
        while (l1 != l2)
        {
            graph[l1][c1] = '|'; if (l1 < l2) l1++; else l1--;
        }
        graph[l1][c1] = '|';
    }
```

8. Appendix II - Minutes of Meetings

Minutes of the 1st Meeting

Date	:	27th November 2019
Venue	:	Canteen D6#1st Floor
Time	:	08:30 p.m.-10:00 p.m.
Attendees	:	ALL MEMBERS *except*
Absent with Apology	:	——

No.	Item
1.	**Call to Order** The meeting was called to order at 8.30 p.m. after our 5-member group had been formed.
2.	**Read the attached file (LexAndYacc.pdf)** Before this meeting, each group members read the file firstly to be familiar with Lex and Yacc. In order to work more efficiently, we decide to use Ubuntu to run the code.
3.	**Install the tools that we need in Ubuntu** After referring to a lot of materials, we installed the Flex and Bison tools in Ubuntu.
4	**Learning the basic knowledge of Lex and Yacc** To make a better improvement, all the group members searched for some basic knowledge of the Lex and Yacc. Then we tried to run the original code

	which generated a basic calculator.
5	**Assigning post-meeting tasks** As the time for meeting is limited, the post-meeting tasks were assigned to all group members. Moreover, it was decided that the group members added different functions of our calculator separately and discussed it in the next meeting.
6	**Adjournment of Meeting** The meeting was adjourned at 10.00 p.m. as the objectives of this meeting had been achieved. The tentative date of the next meeting will be 8:20pm, December 4.

XIAMEN UNIVERSITY MALAYSIA

Minutes of the 2nd Meeting

Date	:	4th December 2019
Venue	:	Canteen D6#1st Floor
Time	:	08:20 p.m.-11:00 p.m.
Attendees	:	ALL MEMBERS *except*
Absent with Apology	:	——

No.	**Item**
1.	**Call to Order** The meeting was called to order at 8.20 p.m. at canteen 1st floor in WeChat group.
2.	**Combine the different functions to our calculator** At the last meeting, it was decided that the group members added the additional functions separately. Hence at the beginning of this meeting, each of the group members shared the improvement that they made in the past few days.
3.	**Write down the outline of the report** After deciding the company to analyze, the group members discussed and wrote down the outline of the report together.
4	**Decide the format of our lab report** To make a better lab report, a plan was made by discussing among the team members. It was decided to divide the report mainly into four parts.

	• Introduction of Lax and Yacc. • Implementation of the calculator. • Discussion of the errors that we met in the process. • Conclusion
5	**Assign post-meeting tasks** As the time for meeting is limited, the post-meeting tasks were assigned to all group members. It was decided each group members try to add more functions to our calculator, amend and simplify the code and begin to write the report.
6	**Adjournment of Meeting** The meeting was adjourned at 11.00 p.m. as the objectives of this meeting had been achieved. The tentative date of the next meeting was 8:00pm, December 11.

XIAMEN UNIVERSITY MALAYSIA

Minutes of the 3rd Meeting

Date	:	11th December 2019
Venue	:	Canteen D6#1st Floor
Time	:	08:00 p.m.-10:00 p.m.
Attendees	:	ALL MEMBERS *except*
Absent with Apology	:	——

No.	**Item**
1.	**Call to Order** The meeting was called to order at 8.00 p.m. at canteen 1st floor in WeChat group.
2.	**Discuss the problems that we meet** In order to make a better improvement, the group members shared the problems that they meet in the process and explained how to solve it.
3.	**Combine the part of report that have been written** At the last meeting, we executed the plan we made. And we made a manual of our calculator together. The group members also came up with suggestions of the lab report of other's parts.
4	**Assign post-meeting tasks** It was decided that references should also be collected and more appendix should be included for better understanding to users. Besides, some sample lab reports should be read to improve our own one.

5	**Adjournment of Meeting** The meeting was adjourned at 10.00 p.m. as the objectives of this meeting had been achieved. The tentative date of the next meeting was 7:00pm, December 15.

XIAMEN UNIVERSITY MALAYSIA

Minutes of the 4rd Meeting

Date	:	15th December 2019
Venue	:	Canteen D6#1st Floor
Time	:	07:00 p.m.-10:00 p.m.
Attendees	:	ALL MEMBERS *except*
Absent with Apology	:	——

No.	**Item**
1.	**Call to Order** The meeting was called to order at 7.00 p.m. at canteen 2nd floor in WeChat group.
2.	**Argue the problems in report** The group members argued the problems we encountered during the writing process. The group members gave our opinions and we reached an agreement at the end.
3.	**Post-mortem** We looked back at the whole process and reflected on our successes and failures. Every group member listened, responded, put forward their thoughts and felt content for the achievement. We built strong relationships with each other and decided to have midnight snack together.
4	**Adjournment of Meeting**

	The meeting was adjourned at 10.00 p.m. as the objectives of this meeting had been achieved.