x

A Simple Calculator

项目编号：project1

学　　号：	**12210158**
姓　　名：	吴同馨
指导老师：	于仕琪
完成时间：	2024-03-09

目 录

第 1 章 项目要求分析……1
1.1 概述……1
1.2 项目要求依据……1
1.3 开发语言要求……1
1.4 实现功能要求……1
1.5 项目提交内容……1
1.6 项目提交时间……2
第 2 章 难点与解决方法……3
2.1 难点与解决方法……3
2.2 解决方法的试验……5
第 3 章 开发环境及工具……6
3.1 开发环境……6
3.2 开发工具……6
3.3 开发工具安装……7
3.4 编译指令……8
第 4 章 功能结构及实现……9
4.1 命令行模式……10
4.2 非命令行模式……11
4.3 大数计算……11
4.4 复杂的输入……12
4.5 错误处理……12
第 5 章 源代码及阐述……14
5.1 main.c……14
5.2 lexer.l……17
5.3 parser.y……19
第 6 章 项目总结……24

第1章　项目要求分析

1.1 概述

根据项目作业要求“Project1.md”文件，本项目的主要要求是用 C 语言实现简单计算器（Simple Calculator），功能包括 2 个数字的加、减、乘、除的运算。

由于这是第一个 C 语言项目，希望通过这个项目接触并熟悉 C 语言开发的各种工具和函数库，以便于在后面的项目中能够更好地使用。

1.2 项目要求依据

Project1.md 文档。

1.3 开发语言要求

用 C 语言实现，满足基本的 C 语言语法，不能采用 C++。

1.4 实现功能要求

1. 实现 2 个数字的加、减、乘、除的运算；
2. 能够对无法执行的运算给出提示信息；
3. 运算符两边应该满足是数字，不是数字时给出提示信息；
4. 能够对大数进行计算；
5. 当不是命令行输入进行计算时，进入一般输入模式，详细见后面章节说明；
6. 扩展功能：参考<https://www.gnu.org/software/bc/manual/html_mono/bc.html>。

1.5 项目提交内容

1. 源代码: .c 文件格式
2. report: pdf 文件格式

1.6 项目提交时间

2024 年 3 月 10 日

第2章　难点与解决方法

2.1 难点与解决方法

项目要求中有 3 个难点。针对这些难点如果都自己从头编程，实现复杂，周期很长，时间不允许。因此解决方法是考虑利用已有的 C 程序工具和函数库。

具体编程难点如下。

1. 输入内容需要判定是否是数字型

   根据项目要求命令行输入的形式如下：

```
$./calculator 2 + 3
$./calculator 2.3 + 3
$./calculator 1.0e200 * 1.0e200
```

   对于上述输入，需要加减乘除符号两边是数字型，用正则表达式是最好的判断方法。通过查阅资料，C 语言可以通过 POSIX 库的正则表达式函数来处理数字输入的判定问题，POSIX 库正则表达式库<regex.h>提供了一系列函数，主要如下：

```
<regex.h>
```

- 编译函数：编译正则表达式字符串为内部格式。
- int regcomp(regex_t *restrict preg, const char *restrict pattern, int cflags);

注意：cflags 使用 REG_EXTENDED，否则会影响匹配函数的作用。

- 匹配函数：在给定字符串上执行已编译的正则表达式匹配。

int regexec(const regex_t *restrict preg, const char *restrict string, size_t nmatch, regmatch_t pmatch[], int eFlags);

- 释放函数： 释放由 regcomp() 分配的正则表达式结构。

```
void regfree(regex_t *preg);
```

使用 chatGPT 输入指令“满足大数，小数，正数，负数，科学计数法的正则表达式”最终得到相应正则表达式。

2. 大数计算

a)使用 GMP 库

如果从头开发大数计算相当复杂，估计在项目周期内最多只能完成一个大数加法运算，因此考虑采用免费、开源的任意精度算术库。

任意精度算术库的英文全称是 GNU Multiple Precision Arithmetic Library，简称 GMP。

GMP 库的特点包括：

任意精度：GMP 支持任意大小的整数、有理数和浮点数运算。

高效性：GMP 使用了多种优化技术以提高运算速度。

易用性：GMP 提供了简洁明了的 API 接口，方便开发者使用。

使用 GMP 库可以降低编程难度，编程进度也可以提速。

b)自定义大数计算函数

使用 char*存放大数进行运算，由于较为复杂，最后只实现了大数正整数乘法。最终使用 GMP 库进行各种操作。

3. 非命令行模式

根据项目要求，非命令行模式要求能连续输入计算式，每输入一次计算式，能打印计算结果。并且根据项目要求所给的网站：

https://www.gnu.org/software/bc/manual/html_mono/bc.html

还要提供在 2 个数加减乘除的运算以外，可以扩展更多更好的计算功能，最少要能提供不止 2 个数的运算功能，比如：

```
(1 + 2)*(2+3)
```

这样的计算功能。这样复杂的表达式需要用词法分析和语法树来进行处理。如果自己做这方面程序可能根本完成不了项目。通过查阅资料，C 语言中有很好的词法分析和自动生成语法树的工具：

```
Flex
Bison
```

Flex 简介：Flex 是一个快速词法分析器生成器（Fast Lexical Analyzer Generator），也称为词法扫描器或词法分析器。它的主要作用是将输入的文本字符串转换成一系列的标记（tokens）。Flex 使用正则表达式来描述这些标记的模式，然后生成相应的 C 语言代码来识别和处理这些模式。

Bison 简介：Bison 是一个语法分析器生成器，它是一个强大的工具，可以用于开发各种语言解析器，从用于简单台式计算器的语言解析器到复杂的编程语言解析器。它提供了一个框架，使开发人员能够更容易地创建高效、准确的语法分析器，而无需从头开始编写大量的代码。

2.2 解决方法的试验

上面虽然考虑了难点及其解决方法，但是需要对可行性进行试验。因此，在项目编程过程中，首先对上面的方法进行了试验。在试验过程中，正则表达式库比较容易用，但 GMP 和 Flex、Bison 混合在一起使用时有些麻烦，主要是因为 GMP 的计算在 flex 和 bison 中和常规的计算不一样引起的。最终，还是解决了 GMP 和 flex、bison 的混合使用。

第3章　开发环境及工具

3.1 开发环境

根据本项目的要求，将在 ubuntu 上实施开发。

1. 操作系统：ubuntu
2. 开发语言：C 语言

3.2 开发工具

由于本项目是在 ubuntu 下开发，因此充分考虑利用 ubuntu 下集成或安装的开发工具，使得开发更加贴近单纯的 linux 操作系统，从而熟悉 linux 操作系统下标准开发工具包括哪些，以及加深对 windows 下集成开发环境的进一步理解：

1. 编辑器：vi

   vi 编辑器具有很强的文字处理能力，并且提供了丰富的快捷方式高效地进行文本编辑，其编辑效率超过了绝大数图形界面的编辑工具。

2. 编译器：gcc

   由于采用 C 语言，因此编译器使用 gcc

3. 调试器：gdb

   gdb 可以在 ubuntu 字符终端下对编译的 c 语言执行程序进行良好地调试，包括设置断点、单步跟踪、查看 runtime 的变量值等全部调试功能；同时可以通过配置 gdb 的资源文件达到自定义查看功能，因此可以更加理解调试的内涵。

4. GMP

   任意精度算术库。

5. 词法语法分析：flex 和 bison

本质上，算术式的解析可以看成是语法和词法分析，所以可以利用 flex 和 bison 来进行算术式的解析，从而能够实现更加复杂的计算功能，从而满足项目要求中的扩展功能的要求。

3.3 开发工具安装

1. vi

   ubuntu 自带 vi，无需安装。但是为了使得在打开.c 文件时自动的使用 C 语言的编辑风格，对~/.vimrc 文件做如下配置：

```
autocmd BufRead,BufNewFile *.c,*.cpp setlocal filetype=c
autocmd BufRead,BufNewFile *.c,*.cpp setlocal autoindent
autocmd BufRead,BufNewFile *.c,*.cpp setlocal smartindent
autocmd BufRead,BufNewFile *.c,*.cpp setlocal cindent
autocmd BufRead,BufNewFile *.c,*.cpp setlocal shiftwidth=4
autocmd BufRead,BufNewFile *.c,*.cpp setlocal tabstop=4
syntax on
set tabstop=4
set shiftwidth=4
set number
```

配置好.vimrc 文件后，打开.c 文件编辑，就完全是 C 的文本编辑风格比如自动缩进。

2. gcc

```
执行：sudo apt-get install build-essential
```

3. gdb

```
执行：sudo apt-get install gdb
如果要增加 gdb 的指令，可以编辑~/.gdbinit 文件。
```

4. gmp

```
sudo apt-get install libgmp-dev
```

5. flex 和 bison

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

3.4 编译指令

编译指令：

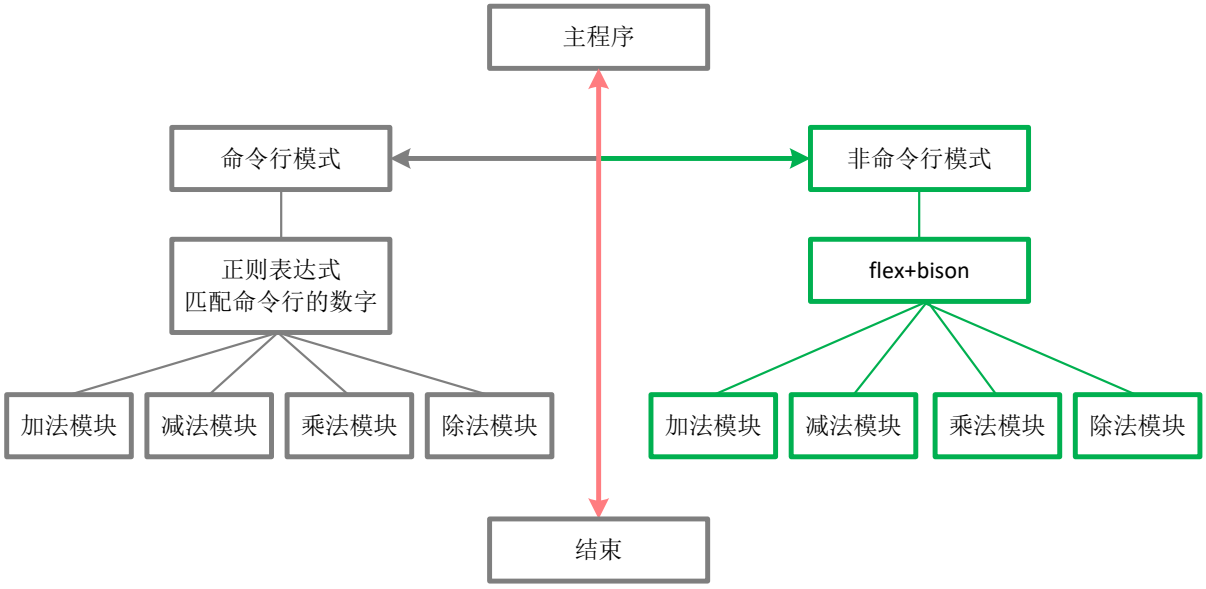
flex lexer.l

bison -d parser.y

gcc -o calculator lex.yy.c parser.tab.c main.c -lgmp

第4章　功能结构及实现

根据项目要求，软件功能有如下的逻辑结构：



软件的功能结构

项目基本功能的要求：

① 完成 2 个数字的加、减、乘、除的运算；

② 完成大数的加减乘除运算；

③ 错误输入的处理；

④ 扩展计算功能参考 bc，比如能进行如下运算：

示例 1： 增加带括号运算

$./calculator

(1+2)*4

(1+2)*4=12.000000

示例 2： 增加变量赋值运算（以分号为隔断，对字母进行赋值后进行字母的加减乘除）

$./calculator

```
a=1;b=2;c=a-b;c

a=1;b=2;c=a-b;c=-1.000000
```

示例 3： 增加数字平方运算

```
$./calculator

10000000000000000000000000000000000000000000000000000000000000000
000000000^
100000000000000000000000000000000000000000000000000000000000000000000
00000^=10000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000.000000
```

4.1 命令行模式

命令行模式下的计算器功能如下所示：

```
```bash
$./calculator 2 + 3
2 + 3 = 5
$./calculator 2 - 3
2 - 3 = -1
$./calculator 2 * 3
2 * 3 = 6
$./calculator 2 / 3
2 / 3 = 0.66666667
```

命令行模式下，要处理加减乘除四则运算，应考虑以下 2 个问题：

1. 运算数都是整数时的输出应该也是整数；
2. 运算数有一个是小数是输出应该是小数；
3. 进行除法运算时两个整数相除结果可能为小数；
```

4. 关于乘法，由于*是转义符，因此运行程序时（命令行式）需要使用*表示乘法；

4.2 非命令行模式

程序内输入模式下的计算器功能如下所示：

```bash
./calculator
2 + 3 # the input
2 + 3 = 5 # the output
2 \* 3 # input another expression
2 * 3 = 6 # the output
quit # the command to quit
```

程序内输入模式下，要处理加减乘除四则运算，应考虑以下 2 个问题：

5. 运算数都是整数时的输出应该也是整数；
6. 运算数有一个是小数是输出应该是小数；

由于 3.1 和 3.2 节中的加减乘除运算功能完全一样，可以调用共同的加减乘除运算函数来实现。

由于*在 ubuntu 中是通配符，因此需要采用\进行转义，最终使用*进行乘法运算。

4.3 大数计算

```bash
./calculator 987654321 \* 987654321
987654321 * 987654321 = 975461057789971041
./calculator 1.0e200 \* 1.0e200
100000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000.000000 *
```

```
10000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000.000000 =
10000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000.000000
```

大数计算模式下，如果要在一个项目周期内完成加减乘除的运算，完全用自己的算法来实现，开发时间上是有难度的。因此，采用自己做一部分，更多的是调用任意精度算术库通常是 GNU Multiple Precision Arithmetic Library (GMP)。

4.4 复杂的输入

根据作业要求，参考 bc 的输入应该有如下的输出：

```
(1+2)*2
(1+2)*2=6
```

对于复杂的表达式，涉及到输入的词法和语法分析，因此采用 flex 和 bison 来进行词法和语法分析，从而达到能够进行复杂计算的效果。

4.5 错误处理

有以下几种出错处理(命令行式)：

It can tell the reason why the operation cannot be carried out.

```bash
./calculator 3.14 / 0
A number cannot be divied by zero.
```

It can tell the reason when the input is not a number.

```bash
./calculator a \* 2
The input cannot be interpret as numbers!
```

对于出现上述的错误输入，可以编写错误处理函数对上述错误输入的提示进行集中管理。

第5章　源代码及阐述

关于源代码，重点阐述了：

1. 正则匹配
2. GMP 的使用
3. flex 和 bison 的使用

5.1 main.c

```c
int is_number(const char *str) {
    //匹配整数、小数(正负数)和科学计数法
    const char *pattern = "^[-+]?[0-9]*\\.?[0-9]+([eE][-+]?[0-9]+)?$";
    regex_t regex;
    int ret;
    regmatch_t match[10];
    // 编译正则表达式
    ret = regcomp(&regex, pattern,  REG_EXTENDED);
    if(ret){
        return 0;
    }
    // 执行正则表达式匹配
    ret = regexec(&regex, str, 10, match, 0);
    if(!ret){
        // 匹配成功
        regfree(&regex);
        return 1;
    }else if(ret == REG_NOMATCH){
        // 匹配失败
        regfree(&regex);
        return 0;
    }else{
        // 正则表达式匹配时发生错误
        char msgbuf[100];
        regerror(ret, &regex, msgbuf, sizeof(msgbuf));
        fprintf(stderr, "Match failed: %s\n", msgbuf);
        regfree(&regex);
        return 0;
    }
}
```

该函数使用正则表达式，判断两个数字的类型是否符合正数，负数，整数，小数，科学计数法。

addBig，addSub，addMul，addDiv 为实现命令行式加减乘除函数，使用 gmp 库进行运算，可以运算超过 c 语言本身 long long 型和 double 型大数。

以 addBig 为例，加法分为整数加法和含小数的加法。整数加法使用 mpz_t 型，

小数加法使用 mpf_t 型。(如下)

```c
void addBig(char * argv[], int b){//大数
    char operator = argv[2][0];
    // 创建GMP整数类型变量
    if(b == 0){
        mpz_t num1,num2,result;
        //初始化
        mpz_init(num1);
        mpz_init(num2);
        mpz_init(result);
        // 从命令行参数读取大整数值
        mpz_set_str(num1, argv[1], 10);
        mpz_set_str(num2, argv[3], 10);
        mpz_add(result,num1,num2);
        // 打印大整数的值
        gmp_printf("%Zd %c %Zd = %Zd\n",num1,operator,num2,result);
        // 清理
        mpz_clear(num1);
        mpz_clear(num2);
        mpz_clear(result);
    }else if(b == 1){
        mpf_t num1,num2,result;
        mpf_init(num1);
        mpf_init(num2);
        mpf_init(result);
        mpf_set_str(num1, argv[1], 10);
        mpf_set_str(num2, argv[3], 10);
        mpf_add(result,num1,num2);
        gmp_printf("%Ff %c %Ff = %Ff\n",num1,operator,num2,result);
        // 清理
        mpf_clear(num1);
        mpf_clear(num2);
        mpf_clear(result);
    }
}
```

不调用 gmp 库实现的大数整数乘法，相比 gmp 复杂很多，功能也有限，在 main 函数中的调用被注释了。(如下)

```c
//不依赖gmp框架进行大数的正整数乘法
char * mul(char * argv[]) {
    char * a = argv[1];
    char * b = argv[3];
    int len1 = strlen(a);
    int len2 = strlen(b);
    //初始化
    char * result = (char*) malloc((len1 + len2 + 1) * sizeof(char));
    memset(result, '0', len1 + len2 + 1);
    result[len1+len2] = '\0';
    for (int i = len1 - 1; i >= 0; i--) {
        int carry = 0;
        for (int j = len2 - 1; j >= 0; j--) {
            int temp = (a[i] - '0') * (b[j] - '0') + carry + (result[i+j+1] - '0');
            carry = temp / 10;
            result[i+j+1] = (temp % 10) + '0';
        }
        result[i] += carry;
    }
//从结果数组中删除不需要的"0"
    int i = 0;
    while (result[i] == '0') {
        i++;
    }
    if (i == len1 + len2) {
        return "0";
    } else {
        return result + i;
    }
    free(result);
}
void mulPrint(char * argv[]){
    char * result = mul(argv);
    char * a = argv[1];
    char * b = argv[3];
    char operator = argv[2][0];
    printf("%s %c %s = %s\n",a,operator,b,result);
}
```

以下是单独调用 mulPrint 的运行结果。

```bash
./calculator 987654321 \* 987654321

987654321 * 987654321 = 975461057789971041
```

```c
//主函数
int main(int argc, char *argv[])
{
    if (argc > 4) {
        exit(EXIT_FAILURE);
    }

    if(argc>1){//命令行
        //判断非法情况
        char * num1 = argv[1];
        char * num2 = argv[3];
        char operator = argv[2][0];
        int a1 = 0;//判断是否为合法字符(含科学计数法)
        int a2 = 0;
        int b = 0;//判断是否是整数
        int c = 0;//判断被除数是否有0
        //判断以上两种情况
        char * ptr1 = num1;
        char * ptr2 = num2;
        while(*ptr1 != '\0') {
            if(*(ptr1 + 1) != '\0'){
                if(*ptr1 == '.'){
                    b = 1;
                }
            }
            ptr1++;
        }
        while(*ptr2 != '\0') {
            if(*(ptr2 + 1) != '\0'){
                if(*ptr2 == '.'){
                    b = 1;
                }
            }
            if(*ptr2 != '0'){
                c = 1;
            }
            ptr2++;
        }
        a1 = is_number(argv[1]);
        a2 = is_number(argv[3]);

        if(a1 == 1 && a2 == 1){
            switch (operator) {
                case '+':
                    addBig(argv,b);
                    break;
                case '-':
                    subBig(argv,b);
                    break;
                case '*'://打印时由于*是通配符需要使用\*作为乘法符号
                    mulBig(argv,b);
                    //mulPrint(argv);//不依赖gmp框架进行大数的正整数乘法调用
                    break;
                case '/':
                    divBig(argv,c);
                    break;
                default:
                    fprintf(stderr, "Error: Unsupported operator '%c'\n", operator);
                }
            }else {//非法
                printf("The input cannot be interpret as numbers!\n");
            }
        }else{
            while(1){//非命令行模式,使用flex和bison
                yyparse();
            }
        }
    return 0;
}
```

5.2 lexer.l

lexer.l 用于实现非命令行模式，定义了本项目词法分析器，用于实现如 1+1=2 的基本功能。使用 flex 编译。

使用 chatGPT 学习 flex 和 bison 语法并得到框架，之后完善相关计算方法。

NUM 代表数字。

VAR 代表变量；LETTER 为字母用于变量的构成。

“OB”“CB”表示小括号。

“+”“-”“*”“/”分别代表加减乘除运算。

增加字母赋值“=”“;”。

增加“^”代表平方运算。

quit 用于退出非命令行模式

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gmp.h>
#include "parser.tab.h"
%}

%option noyywrap
%option header-file="lex.yy.h"

NUM     -+?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
LETTER  [a-zA-Z]
VAR     {LETTER}({LETTER}|[0-9])*

%%
"("         { printf("%s", yytext);return OB;  }
")"         { printf("%s", yytext);return CB;  }
"+"         { printf("%s", yytext);return ADD; }
"-"         { printf("%s", yytext);return SUB; }
"*"         { printf("%s", yytext);return MUL; }
"/"         { printf("%s", yytext);return DIV; }
"="         { printf("%s", yytext);return ASSIGN; }
";"         { printf("%s", yytext);return SEMI; }
"^"         { printf("%s", yytext);return SQU; }
{NUM}       {
                mpf_init2(yylval.mpf_value,256);
                mpf_set_str(yylval.mpf_value,yytext,10);
                printf("%s", yytext);
                return NUM;
            }
"quit"      {
                return QUIT;
            }
{VAR}       {
                if(!strcmp(yytext,"quit")){
                    return QUIT;
                } else{
                    yylval.str_value = strdup(yytext);
                    printf("%s", yytext);
                    return VAR;
                }
            }
\n          { return EOL; }
[ \t]       { }
.           { printf("%s", yytext);return yytext[0]; }
%%
```

5.3 parser.y

parser.y 文件为本项目语法分析。

相关定义如下图：

其中% {　%}中定义了 c 语言变量表结构和变量计数器 symbol_count。

symbol_t 定义了变量表的基本结构，包括了变量名，变量值。

symbo_table[25600]定义表量表，本程序支持最大是 25600 个变量。

%union 定义了语法符号的语义值，不同的符号可以有不同的数据类型。其中 mpf_value 用于大数，str_value 用于变量名。

%token 指令用于定义终端符号（terminals）和它们的类型。这些符号通常是语法中的基本元素，如关键字、标识符或字面量。其中%token NUM 和%token VAR 需要定义变量类型。

%type 指令用于为语法规则中的非终端符号（nonterminals）指定类型。这个类型必须是之前通过 %union 定义的 union 类型中的一个成员。

```
%{
#include <stdio.h>
#include <gmp.h>
#include "lex.yy.h"

extern int yylex();
void yyerror(char * e);
mpf_t* find_variable_table(char* name);

int symbol_count = 0;

typedef struct {
    char* name;
    mpf_t mpf_value;
}symbol_t;

symbol_t symbol_table[25600];
mpf_t result4cunit;//用于cunit的结果assert
%}

%union {
    mpf_t mpf_value;
    char* str_value;
}

%token <mpf_value> NUM
%token <str_value> VAR
%token ADD SUB MUL DIV
%token SQU
%token ASSIGN
%token OB CB
%token SEMI
%token EOL
%token QUIT
%type <mpf_value> expr factor term assignment
```

%%　%%中定义语法。

```
%%
program:
    statement_list EOL
    ;

statement_list:
    statement
    |statement_list statement
    ;

statement:
    assignment SEMI
    | expr SEMI
    ;

assignment:
    VAR ASSIGN expr {
        mpf_init2($$,256);
        for( int i=0; i < symbol_count; i++) {
            if( !strcmp(symbol_table[i].name, $1)) {
                mpf_init2(symbol_table[i].mpf_value,256);
                mpf_set(symbol_table[i].mpf_value,$3);
                mpf_set($$,symbol_table[i].mpf_value);
                mpf_clear($3);
                free($1);
                return 1;
            }
        }
        symbol_table[symbol_count].name = strdup($1);
        mpf_init2(symbol_table[symbol_count].mpf_value,256);
        mpf_set(symbol_table[symbol_count].mpf_value,$3);
        mpf_set($$,symbol_table[symbol_count].mpf_value);
        mpf_clear($3);
        symbol_count++;
    }
    ;
```

在 parser.y 中实现相关功能。语法由一个声明列表和回车构成。其中 EOL 为进行回车指令后打印相关结果。

NUM 是 mpf_t 类型，存储在 mpf_value 里。$1，$3 为 NUM，$$为输出结果。

NUM NUM 解决了负数减法问题。

statement_list 由 statement 构成，statement 由计算式和赋值语句（assignment）构成。

以加法减法为例，计算式包括加减运算。

expr 表示加减和因子本身。

factor 表示乘除和因子本身。

term 表示括号，变量和因子本身。

```
expr:
    factor {
        mpf_init2($$,256);
        mpf_set($$,$1);
        mpf_clear($1);
    }
    | expr ADD factor {
        mpf_init2($$,256);
        mpf_add($$,$1,$3);
        mpf_clear($1);
        mpf_clear($3);
    }
    | NUM NUM { //减法
        mpf_init2($$,256);
        mpf_add($$,$1,$2);
        mpf_clear($1);
        mpf_clear($2);
    }
    | expr SUB factor { //减法(用于字母赋值运算)
        mpf_init2($$,256);
        mpf_sub($$,$1,$3);
        mpf_clear($1);
        mpf_clear($3);
    }
    | expr EOL {
        gmp_printf("=%Ff\n",$$);
        mpf_init2(result4cunit,256);
        mpf_set(result4cunit,$$);
        mpf_clear($$);
        YYABORT;
    }
    | QUIT {
        exit(0);
    }
    ;
```

```
term: NUM{
        mpf_init2($$,256);
        mpf_set($$,$1);
        mpf_clear($1);
    }
    | OB expr CB {//括号
        mpf_init2($$,256);
        mpf_set($$,$2);
        mpf_clear($2);
    }
    | VAR{
        //查找变量，如果没有该变量，则初始化
        mpf_t * vari_value;
        vari_value = find_variable_table($1);

        //结果输入到$$，即保存为当前语法栈的结果值
        mpf_init2($$,256);
        mpf_set($$,*vari_value);
        mpf_clear(*vari_value);
        free(*vari_value);
        free($1);
    }
    ;
%%
```

```
factor: term{
        mpf_init2($$,256);
        mpf_set($$,$1);
        mpf_clear($1);
    }
    | factor MUL term{
        mpf_init2($$,256);
        mpf_mul($$,$1,$3);
        mpf_clear($1);
        mpf_clear($3);
    }
    | factor DIV term{
        mpf_init2($$,256);
        if(mpf_cmp_si($3,0) == 0){
            printf("A number cannot be divied by zero.\n");
            break;
        }else{
            mpf_div($$,$1,$3);
        }
        mpf_clear($1);
        mpf_clear($3);
    }
    | factor SQU {//平方
        mpf_init2($$,256);
        mpf_pow_ui($$,$1,2);
        mpf_clear($1);
    }
    ;
```

以下是相关运算结果。

```
./calculator

2+3

2+3=5.000000

2-3

2-3=-1.000000

2*3

2*3=6.000000

2/3

2/3=0.666667

(2+3)*5
```

(2+3)*5=25.000000

2+3*5

2+3*5=17.000000

a=1;b=-2;c=a+b;c

a=1;b=-2;c=a+b;c=-1.000000

100000000000^

100000000000^=1000000000000000000000.000000

1.0e200*1.0e200

1.0e200*1.0e200=100.000000

quit

第6章　项目总结

通过本项目，学习了 C 语言基本编程方法，也学习了一些工具，包括：任意精度算术库 GMP、正则表达式处理库、flex 和 bison 以及词法分析和语法分析初步、cunit 测试方法初步等等，同时熟悉了 ubuntu 上编辑器 vi、编译器 gcc、调试器 gdb。

在项目的编程过程中，由于时间的限制，上述的工具很多是浅尝则止，许多更深的功能还没有来得及使用，而且即便已经使用的部分也还是没有达到熟练程度。比如在 bison 的使用中，只是实现了简单的算术表达式，距离
https://www.gnu.org/software/bc/manual/html_mono/bc.html
所描述的功能有巨大的差距。

通过本项目的实践，发现对于项目中复杂的功能需要大量利用已有成熟的工具去完成，这样才能在有限的时间里开发出相对完善、正确的功能，并且这些成熟的工具对于拓宽视野、开阔思路也非常有帮助。