CS205 C/ C++ Programming-Calculator

Name: 吴宇贤

SID: 12212614

Part 1 - Analysis

1. 阅读需求

首先Project文档中提供了最简单的例子：

```
$./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
```

不难发现，这个项目的核心就是通过命令行参数传入表达式，进行两数的四则运算。

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

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

命令行参数以字符串的形式传入，所以要对于参数进行错误检测，如除零、数字不合法等。

```
./calculator 987654321 * 987654321

./calculator 1.0e200 * 1.0e200
```

能够完成大数计算，甚至可以进行科学计数法的运算。

```
./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
```

然后是这个计算器的拓展功能，不传递参数的时候能够进入计算模式一直读取用户键入的表达式进行计算直到接收quit指令。

那么如何同时完成以上功能呢？我经历了以下几种思路历程：

1. 解析命令行传入的参数后，简单运行程序的+、-、*、/符号计算结果并输出，但是显然，这无法处理大数以及科学计数法。
2. 将参数解析成double类型，但测试之后发现范围过大的数字精度损失比较多，不太符合要求。
3. 四则运算都可以使用高精度完成。

所以我需要实现一个能够支持高精度四则运算的计算器，并且对于小数位数过多的数字，用户可以自行选择保留小数的位数。

2. 需求实现

> 下面图片引用自oi-wiki.org

2.1 前期准备

C语言本身并不支持高精度，最好想到的是用一个数组来表示大数，其每一位表示大数的一个数字，但是使用数组并不方便表示小数，对于科学计数法也无计可施，所以我写了一个BigNum的结构体来表示大数，其中array成员存储数字数据，但是这里我用了一个小技巧，下标最小的位置存放的是数字的 **最低位**，即存储反转的字符串，这么做的好处是，数字的长度可能发生变化，但我希望同样权值位始终保持对齐（例如，希望所有的个位都在下标 `[0]`，所有的十位都在下标 `[1]`……）；同时，加、减、乘的运算一般都从个位开始进行，这都给了**反转存储**以充分的理由。利用这一个结构体可以完整的表示大数、小数、正负数，当然也可以表示成科学计数法

```
typedef struct
{
    int *array;          // 数字，倒序存储
    int sign;            // 符号
    int length;          // 数字长度
    int decimal_length; // 小数位数
    int exponent;        // 指数部分
} BigNum;
```

但是用户键入的是字符串，并不能直接转化为BigNum对象，并且程序无法直接输出BigNum对象，于是我为BigNum提供了以下功能函数以解析字符串转化为BigNum对象、管理内存、打印BigNum。

- BigNum BigNumConstructor(char *str);（调用函数前str字符串一定通过了错误检测）
  - 返回BigNum型的bigNum
  - 检查首位是否有符号，初始化bigNum.sign
  - 遍历字符串检查是否有关键字符`e`，若有，则含指数，根据`e`后面的数字初始化bigNum.exponent；若没有，则将bigNum.exponent初始化为0
  - 对于`e`前面的数字部分，检查是否有小数点，若有统计小数位数后将数字逆序存至bigNum.array；若没有，则直接将数字逆序存至bigNum.array
- void freeBigNum(BigNum *bigNum);
  - 释放分配的内存
- void printBigNum(BigNum *bigNum, int scale, int isScientifNotation);
  - scale：保留的小数位数；isScientifNotation：是否用科学计数法表示
  - 若使用科学计数法输出，则需要先调整这个bigNum为标准的科学计数法，比如把$12e9$调整为$1.2e10$。
    - 这里我写了一个函数void toScientificNotation(BigNum *bigNum); 用于调整bigNum的小数位数和指数部分大小以达到要求
  - 随后分别打印整数部分(若有)、小数点、小数部分和指数部分即可

2.2 错误检测

首先用户键入的运算式需要满足以下格式`<operand1> <operator> <operand2> [-s n]`，其中`[-s n]`表示要保留的`n`位小数，若不输入则保留默认位数，加减乘法保留原始位数，除法默认保留六位小数。

按照以上规则，解析完键入的运算式后，字符串个数应该满足3个或者5个，否则提示`Invalid expression format`。

对于操作数，用int isValidBigNum(char *str)函数检测其是否满足要求。若满足返回1；不满足则返回0。

- 检查符号位
- 小数点只允许出现一次，且不在指数部分
- 'e'或'E'表示指数，只允许出现一次，且不能是第一个字符
- 检查指数后是否有符号位，且指数部分不能出现小数点
- 最后一个字符不能是 'e' 或 'E'

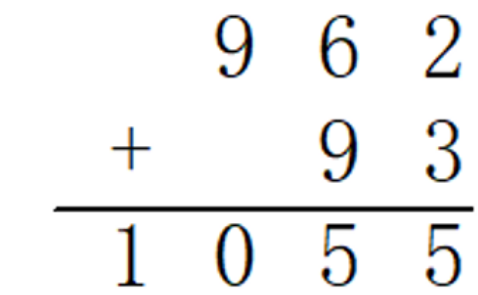
对于操作符op，只允许+、-、*、/、%，用`strchr("+-*/%", op)`检测是否满足要求

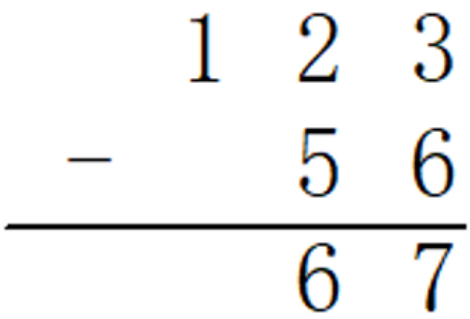
若用户输入了保留小数的位数，则检测第四个字符串是否为`-s`或者`-scale`，不满足则提示不合法。

最后对于除零的检测，用int isBigNumZero(const BigNum *num)函数检测大数是否为0，若为0且操作符为`/`或者`%`则提示`A number cannot be divied by zero!`

2.3 高精度加减法

高精度加减法实际上就是竖式加减法：





对于加法而言，就是从最低位开始，将两个加数对应位置上的数码相加，并判断是否达到或超过10。如果达到，那么处理进位：将更高一位的结果上增加1，当前位的结果减少10。

```
// 从低位开始，逐位相加，carry表示进位
int carry = 0;
for (int i = 0; i < maxLength - 1; i++)
{
    int aDigit = i < a->length ? a->array[i] : 0;
    int bDigit = i < b->length ? b->array[i] : 0;
    int sum = aDigit + bDigit + carry;
    result.array[i] = sum % 10;
    carry = sum / 10;
    result.length++;
}
if (carry)
{
    result.array[result.length] = carry;
    result.length++;
}
```

对于减法而言，从个位起逐位相减，遇到负的情况则向上一位借 1，整体思路与加法完全一致。

```
// 从低位开始，逐位相减，borrow表示借位
int borrow = 0;
for (int i = 0; i < a->length; i++)
{
    int aDigit = a->array[i];
    int bDigit = i < b->length ? b->array[i] : 0;
    int diff = aDigit - bDigit - borrow;
    if (diff < 0)
    {
```

```
        diff += 10;
        borrow = 1;
    }
    else
    {
        borrow = 0;
    }
    result.array[i] = diff;
    if (diff != 0)
        result.length = i + 1;
}
```

在代码的层面来说就是利用数组模拟竖式计算从低位依次计算到高位，而由于BigNum的array成员是倒序存储数字，故而对于整数而言，我们可以直接利用array成员进行运算。但是由于我们的BigNum支持小数运算，实际上的对齐低位可以描述为对齐小数点，小数位数少的用0补齐，于是我写了一个函数用于补全小数位数。

> 若当前大数小数位数大于等于期望位数时则直接return

```
// 对小数位数进行补全
void padDecimalPlaces(BigNum *bigNum, int newDecimalLength);
```

这样补全小数后，两个数从低位开始又一一对应，只需继续竖式计算即可，结果小数位数即为补全后的小数位数。

而另一层面如果是对于使用科学计数法的大数，我们需要运用到类似于补全小数的原理，先让指数部分相同（方便起见，把指数大的化小），随后再对有效数部分进行加减运算，当然指数相同后，还需要对指数前的数进行小数补全，随后两个使用科学计数法的数字指数部分不变，其前面的数字部分进行加减法即可。

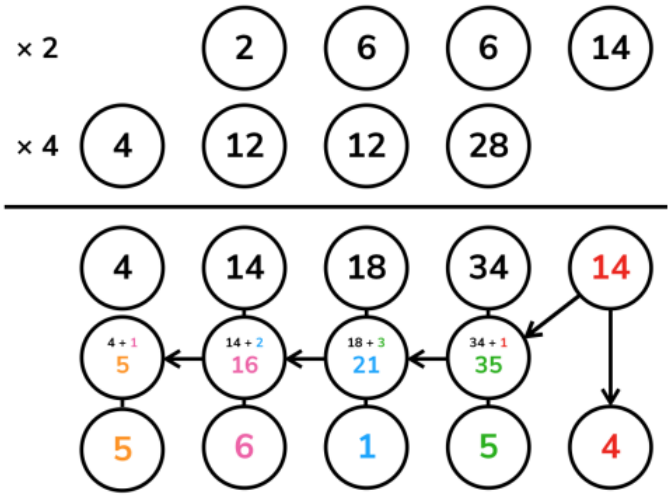
```
// 调整指数大小 把指数大的调整为和指数小的一致
void adjustExponent(BigNum *bigNum, int newExponent);
```

核心代码：(对于两个操作数BigNum *a, BigNum *b而言)

```
// 若指数不一致，把指数较大的调整为和指数较小的一致
adjustExponent(a, b->exponent);
adjustExponent(b, a->exponent);
// 若小数点后位数不一致，补全位数少的
padDecimalPlaces(a, b->decimal_length);
padDecimalPlaces(b, a->decimal_length);
```

2.4 高精度乘法

从本质上来说，高精度乘法还是竖式乘法，以下是1337 * 42的竖式计算过程



对于$a * b$得到的$result$来说， 忽略进位时，假设a和b是十进制下的两个数字，它们的每一位分别为a_j和b_k，其中j和k表示数字的位数（从0开始计数）。那么，$result$的第i位可以近似地表示为所有$j + k = i$时的$a_j * b_k$和。

$$result[i] = \sum_{j+k=i} a_j * b_k$$

利用这个特性，我们先算出结果每一位上的数字之后，再从低位到高位统一进位：

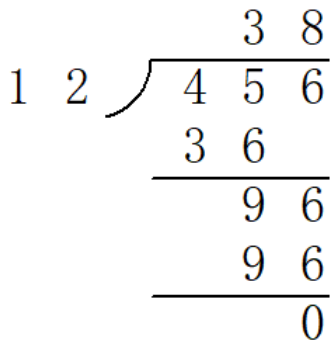
```
// 先计算result的每一位数字
for (int i = 0; i < a->length; i++)
{
    for (int j = 0; j < b->length; j++)
    {
        result.array[i + j] += a->array[i] * b->array[j];
    }
}

// 统一处理进位
for (int i = 0; i < maxLength - 1; i++)
{
    if (result.array[i] >= 10)
    {
        result.array[i + 1] += result.array[i] / 10;
        result.array[i] %= 10;
    }
}
```

对于小数而言，$result$小数位数即为a和b小数位数之和；对于科学计数法而言，指数部分相加即可。

2.5 高精度除法

高精度除法还是可以利用除法的竖式来解决，不过其中处理有点复杂



竖式长除法实际上可以看作一个逐次减法的过程。例如上图中商数十位的计算可以这样理解：将45减去三次12后变得小于12，不能再减，故此位为3，随后计算下一位。

这里我选择把逆序存储的数字倒转成正序存储的数组进行操作，符合竖式思维

```
// BigNum转数组顺序存储
void BigNumToArray(int *array, BigNum *bigNum)
{
    for (int i = 0; i < bigNum->length; i++)
    {
        array[i] = bigNum->array[bigNum->length - 1 - i];
    }
}
```

这里实现了一个函数 `canSubtract()` 用于判断被除数以下标 `last_dg` 为最低位，是否可以再减去除数而保持非负。此后对于商的每一位，不断调用 `canSubtract()`，并在成立的时候用高精度减法从余数中减去除数，也即模拟了竖式除法的过程。

```
// 模拟竖式除法 divisor除数、quotient商、remainder余数(初始等于被除数)
for (int i = b->length - 1; i < dividend_length; i++)
{
    // 计算商的第i位
    while (canSubtract(remainder, divisor, i, b->length))
    {
        // 高精度减法
        for (int j = 0; j < b->length; j++)
        {
            remainder[i - j] -= divisor[b->length - 1 - j];
            if (remainder[i - j] < 0)
            {
                remainder[i - j] += 10;
                remainder[i - j - 1] -= 1;
            }
        }
        quotient[i]++;
    }
}
```

随后再根据需要把顺序存储的数组转化为BigNum：

```c
// 顺序存储的数组转BigNum
void ArrayToBigNum(int *array, BigNum *bigNum, int start, int len)
{
    bigNum->length = len - start;
    bigNum->array = (int *)malloc(bigNum->length * sizeof(int));
    for (int i = 0; i < bigNum->length; i++)
    {
        bigNum->array[i] = array[len - 1 - i];
    }
}
```

对于小数除法，通过被除数和除数同时扩大的方式确保除数为整数，随后记录好小数点的位置即可；如果想要保证更高的精度，可以直接给被除数后面加零，在计算完成之后，移动对应位数的小数点即可。

对于科学计数法，指数部分前面的数字正常计算，指数部分相减即可。

> 而这里remainder即为剩下的余数，于是我们可以拓展出取模操作（对整数而言），允许%运算符

2.6 无参运行程序

每次通过命令行参数传入表达式的方法只计算一个表达式有些局限，我们希望能够无参运行程序获得更多模式。

首先第一个模式在计算两数四则运算的基础上，能够不断地从控制台接收并计算，直到用户输入quit指令退出该模式，我把这个模式命名为`Standard Mode`。

其次，个人认为只对于两个数进行四则运算也过于局限，由于我们已经有BigNum结构和计算加减乘除的函数，这里完全利用这些加之栈的结构可以拓展出一个模式来不断接收并计算完整的表达式，这个模式为`Expression Mode`。

对于模式的选择，程序提醒用户输入`1`、`2`或者`quit`分别表示Standard Mode、Expression Mode或者退出程序

如图：

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out
Welcome to smart calculator!
Enter '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> 1
Standard Mode!
Enter expressions in the format <operand1> <operator> <operand2>.
Type 'quit' to back to main menu.
<Standard Mode> quit
Main menu!
Type '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> 2
Expression Mode!
Enter your expression (type 'quit' to back to main menu):
<Expression Mode> quit
Main menu!
Type '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> 3
Invalid input. Please enter '1', '2', or 'quit'.
Enter '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> quit
Exiting...
```

Standard Mode

使用while循环保证程序一直运行，每次循环迭代都会提示用户输入一个新的表达式，然后使用`fgets`从标准输入读取这个表达式。随后若通过错误检测则进行运算，若没有通过则提示用户重新输入。

Expression Mode(拓展)

在这个模式之下，用户可以自由输入表达式，如`(1 + 2) *3 /(4 - 5)`随后程序计算出结果输出。

我选择使用的方法是使用两个栈：一个用于数字（操作数），另一个用于运算符（包括括号）。以下是详细的步骤：

1. **解析和预处理输入**

   清理和验证输入表达式，例如去除不必要的空格以方便后续读取表达式。

2. **使用栈实现的表达式算法**

- 操作数栈：用于存储操作数。

```
typedef struct
{
    BigNum items[STACK_SIZE];
    int top;
} BigNumStack;
```

- 运算符栈：用于存储运算符和括号。

```
typedef struct
{
    char items[STACK_SIZE];
    int top;
} OperatorStack;
```

基本步骤：

- 遍历表达式中的每个标记（数、运算符、括号）
  - 如果是数字，直接压入操作数栈。
  - 如果是运算符，比较其与运算符栈栈顶运算符的优先级：
    - 如果当前运算符优先级更高，将其压入运算符栈。
    - 否则，从运算符栈中弹出运算符，并从操作数栈中弹出相应数量的操作数进行计算，然后将结果压回操作数栈。重复此过程，直到可以安全地将当前运算符压入运算符栈。
  - 如果是左括号`(`，直接压入运算符栈。
  - 如果是右括号`)`，则反复从运算符栈中弹出运算符，并进行计算，直到遇到左括号`(`。左括号`(`只是用来标记括号的开始，一旦完成计算就应该从栈中移除。
- 遍历完所有标记后，如果运算符栈中仍有运算符，继续进行弹出和计算操作，直到运算符栈为空。

3. **处理高精度数值**

- 上述过程中提到的“数字”使用之前定义的`BigNum`结构体来表示和存储，确保高精度。
- 所有的计算操作（加、减、乘、除）都需要使用相应的高精度算法实现。

4. **最终结果**

- 算法完成后，操作数栈中只剩下一个元素，即整个表达式的计算结果。

3 项目特色

1. 高精度支持：四则运算都支持高精度整数和小数，除法也不会出现类似于double的精度缺失
2. 错误提示：若用户输入数据有误能够给出准确提示。
3. 科学计数法：支持用户输入用科学计数法表示的数字，以及能够用科学计数法输出。
4. 支持长表达式运算：用户输入长表达式，程序能够按照正确的顺序计算并输出答案。

Part 2 - Result & Verification

1. 基本功能

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 2 + 3
2 + 3 = 5
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 2 - 3
2 - 3 = -1
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 2 \* 3
2 * 3 = 6
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 2 / 3
2 / 3 = 0.666667
```

2. 错误提示

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 3.14 / 0
A number cannot be divied by zero!
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out a / 2
The input cannot be interpreted as numbers! Please enter a valid operand (e.g., -123.45).
```

3. 高精度和科学计数法

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 987654321 \* 987654321
987654321 * 987654321 = 975461057789971041
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 1.0e200 \* 1.0e200
1.0e200 * 1.0e200 = 1e400
```

4. 保留小数位数

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 12 / 15 -s 10
12 / 15 = 0.8000000000
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 12 / 17
12 / 17 = 0.705882
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 12 / 17 -s 10
12 / 17 = 0.7058823529
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out 12 / 17 -s 3
12 / 17 = 0.706
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out
Welcome to smart calculator!
Enter '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> 1
Standard Mode!
Enter expressions in the format <operand1> <operator> <operand2>.
Type 'quit' to back to main menu.
<Standard Mode> 12.36658 * 7.3225
12.36658 * 7.3225 = 90.55428205
<Standard Mode> 12.36658 * 7.3225 -s 5
12.36658 * 7.3225 = 90.55428
<Standard Mode> 12.36658 * 7.3225 -s 4
12.36658 * 7.3225 = 90.5543
<Standard Mode> quit
Main menu!
Type '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> quit
Exiting...
```

4. Standard Mode

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out
Welcome to smart calculator!
Enter '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> 1
Standard Mode!
Enter expressions in the format <operand1> <operator> <operand2>.
Type 'quit' to back to main menu.
<Standard Mode> 1 + 2
1 + 2 = 3
<Standard Mode> -1 + 2
-1 + 2 = 1
<Standard Mode> 1 - 2
1 - 2 = -1
<Standard Mode> 123.6 + 1.36
123.6 + 1.36 = 124.96
<Standard Mode> 1.36 - 123.6
1.36 - 123.6 = -122.24
<Standard Mode> 123.6 * 1.36
123.6 * 1.36 = 168.096
<Standard Mode> 123.6 / 1.36
123.6 / 1.36 = 90.882353
<Standard Mode> 12345686541345687823154 - 0.2254785
12345686541345687823154 - 0.2254785 = 12345686541345687823153.7745215
<Standard Mode> 132456887645122125 * 12356658785211254.66358
132456887645122125 * 12356658785211254.66358 = 1636724564381838403517442257944889.7075
<Standard Mode> 3.6e10 + 1.9e11
3.6e10 + 1.9e11 = 2.26e11
<Standard Mode> 5.5e8 * 6.7e5
5.5e8 * 6.7e5 = 3.685e14
<Standard Mode> 123.6 / 1.36 -s 10
123.6 / 1.36 = 90.8823529412
<Standard Mode> 123.6 / 1.36 -s 1
123.6 / 1.36 = 90.9
<Standard Mode> -123.6 * 1.36
-123.6 * 1.36 = -168.096
<Standard Mode> a * 2
Invalid expression format.
Please follow the format <operand1> <operator> <operand2>.
<Standard Mode> quit
Main menu!
Type '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> quit
Exiting...
wyx@WYX:~/cs203_projects/projects/project1 caculator$
```

5. Expression Mode

```
wyx@WYX:~/cs203_projects/projects/project1 caculator$ ./calculator.out
Welcome to smart calculator!
Enter '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> 2
Expression Mode!
Enter your expression (type 'quit' to back to main menu):
<Expression Mode> (1 + 2) *3 /(4 - 5)
(1 + 2) *3 /(4 - 5) = -9.000000
<Expression Mode> 1+3*7-10*5
1+3*7-10*5 = -28
<Expression Mode> 12 + 2*8-44
12 + 2*8-44 = -16
<Expression Mode> 100+2e2*6
100+2e2*6 = 1.3e3
<Expression Mode> (3e10+2e9)*3 - 5e11
(3e10+2e9)*3 - 5e11 = -4.04e11
<Expression Mode> (1+3)*125.335 - (5-3*8)+12*99.3
(1+3)*125.335 - (5-3*8)+12*99.3 = 1711.94
<Expression Mode> quit
Main menu!
Type '1' for standard mode, '2' for expression mode, or 'quit' to exit:
> quit
Exiting...
```

Part 3 - Difficulties & Solutions

怎样表示大数

我是c语言初学者，此前只接触过java，最开始看到project题目的时候所有的思考都是从java的角度出发的，比如写一个BigNum类表示大数，但到c语言这边完全不知道如何下手，于是恶补c语言一些基本的结构和语法后才用结构体解决这个问题。

小数大数

最开始BigNum结构体只能够表示整数的大数，在写完整数大数的四则运算之后，怎么样用最小的修改成本能够让现有的结构体表示小数呢?

我发现小数的四则运算和整数完全没有区别，不过是小数点需要做处理，于是我给BigNum结构体加了一个成员decimal_length表示小数的位数，0表示没有小数。

这样即使是想要移动小数点也非常方便，只需要改变decimal_length的位数即可。

指数和科学计数法

BigNum能够表示小数之后，如何表示指数和科学计数法呢，于是再次加了一个成员exponent表示指数大小，这样一个正常的大数只需要改变两个成员变量decimal_length和exponent就能够直接达成目的。

解析长表达式

这个拓展功能花了很多时间，首先是一个输入一个长表达式，我的程序该如何精确的识别数字和符号呢？最后我选择先处理掉多余的空格，然后遍历字符串一一匹配的方法来解析这个表达式。

其次如何实现这个算法，把上学期的dsaa中的栈知识复习一遍之后，最开始我想的是把中缀表达式转化成后缀表达式然后计算结果，但是我不知道一个栈怎么如和同时存储两种不同类型的变量（大数BigNum、 操作符char），然后我发现我们其实可以选择用两个栈，一个存储大数，一个存储操作符，然后同时对栈进行处理即可。

Part 4 - Conclusion

1. 通过这次的探索和实践，学到了很多新知识，尤其是关于`BigNum`结构体的使用。这不仅加深了我对C语言中结构体的理解，也让我对如何处理和表示大数有了全新的认识。在构建`BigNum`这一过程中，学会了如何设计一个能够存储和操作大数的数据结构，包括如何在结构体中安排各种字段来存储数值的各个部分、符号、长度、小数位数，以及指数部分来支持科学计数法。

   还学习到了如何通过动态内存分配来给大数分配空间，涉及到对`malloc`的使用，以及如何在操作完成后正确地释放内存来避免内存泄露，而这对于我深入理解C语言中的内存管理机制比较有帮助。

   此外，还掌握了字符串处理的技巧，例如如何去除字符串中的空格、如何从字符串中解析出大数的各个组成部分，以及如何处理科学计数法的表示。学会了使用`atoi`函数将字符串转换为整数，并理解了指针运算在字符串处理中的应用。

   在实现`BigNum`的各种操作过程中，比如加、减、乘、除以及指数的调整，深入理解了基本的算术运算原理，学习到了如何在c语言中，尤其是在处理高精度运算时实现这些原理。

2. 写代码就像建房子，而房子不是一下子就能建起来的，而是一步一步地构建。对于`BigNum`这个结构体来说，我是一步一步地给它添加功能，但实际上我写代码的时候非常痛苦，因为新功能有的地方没办法和旧功能兼容。每当我尝试引入一个新特性时，往往需要重新考虑和调整既有的代码结构和逻辑，以确保新旧功能之间的共存。这个过程中，代码的复杂度逐渐增加，有时候甚至需要对原有的设计进行根本性的重构，这无疑增加了开发的难度和工作量。正如在建筑过程中可能需要重新

布线或改管，软件开发也需要不断地迭代和改进，以适应功能的增加和需求的变化。但我们其实可以通过从一开始就构建好框架，尽量避免这种情况的发生，比如如果一开始我就同时考虑到整数、小数以及指数的表示，那么在写代码的过程中无论是哪个部分，我都可以直接考虑到兼容的情况。这个project警示了我在写项目之前要先规划好一个大体的框架，而不是在写代码的过程中不断推翻之前的框架去建立新框架。

Part 5 - Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>

#define STACK_SIZE 100

typedef struct
{
    int *array;          // 数字，倒序存储
    int sign;            // 符号
    int length;          // 数字长度
    int decimal_length; // 小数位数
    int exponent;        // 指数部分的值
} BigNum;

// 检查字符串是否合法
int isValidBigNum(char *str);
// 解析字符串转化成BigNum类型
BigNum BigNumConstructor(char *str);
// 释放内存
void freeBigNum(BigNum *bigNum);
// 打印BigNum类型整数
void printBigNum(BigNum *bigNum, int scale, int isScientifNotation);

BigNum add(BigNum *a, BigNum *b);
// a>b的减法
BigNum subtract(BigNum *a, BigNum *b, int com);

BigNum multiply(BigNum *a, BigNum *b);
// 对小数位数进行补全
void padDecimalPlaces(BigNum *bigNum, int newDecimalLength);
// 调整指数大小 把指数大的调整为和指数小的一致
void adjustExponent(BigNum *bigNum, int newExponent);
BigNum addBigNum(BigNum *a, BigNum *b);
BigNum subtractBigNum(BigNum *a, BigNum *b);
BigNum multiplyBigNum(BigNum *a, BigNum *b);
// 计算a/b ，c表示商，d表示余数
void divideBigNum(BigNum *a, BigNum *b, BigNum *c, BigNum *d, int scale);

void calculate(char *operator1, char *operator, char * operator2, int
scale);

// 以下函数计算表达式的值

```

```c
// BigNum 栈
typedef struct
{
    BigNum items[STACK_SIZE];
    int top;
} BigNumStack;

// 运算符栈
typedef struct
{
    char items[STACK_SIZE];
    int top;
} OperatorStack;
// 初始化 BigNum栈
void initBigNumStack(BigNumStack *stack);
// 初始化运算符栈
void initOperatorStack(OperatorStack *stack);
// BigNum入栈操作
void pushBigNum(BigNumStack *stack, BigNum num);
// BigNum弹栈操作
BigNum popBigNum(BigNumStack *stack);
// 运算符入栈操作
void pushOperator(OperatorStack *stack, char op);
// 运算符弹栈操作
char popOperator(OperatorStack *stack);
// 访问运算符栈栈顶元素
char peekOperator(OperatorStack *stack);
// 判断运算符栈是否为空
int isOperatorStackEmpty(OperatorStack *stack);
// 返回运算符优先级，数字越大优先级越高
int precedence(char op);
// 对BigNum进行运算(和calculate()类似，参数不同)
BigNum performOperation(BigNum *a, BigNum *b, char op);
// 去除表达式中的空格
void removeSpaces(char *str);
// 处理表达式
void evaluateExpression(char *expression);
// 标准模式
void standardMode();
// 表达式模式
void expressionMode();

int main(int argc, char *argv[])
{
    if (argc != 4 && argc != 1 && argc != 6)
    {
        printf("Usage: <operand1> <operator> <operand2>\n");
        printf("Or just run the program without any arguments to choose more mode.\n");
        return 1;
    }

    if (argc == 1)
    {
        char input[100]; // 存储用户输入

```

```
        printf("Welcome to smart calculator!\nEnter '1' for standard mode, '2' for expression mode, or 'quit' to exit:\n");
        while (1)
        {
            printf("> ");
            if (!fgets(input, sizeof(input), stdin))
            {
                continue;
            }

            input[strcspn(input, "\n")] = 0;

            if (strcmp(input, "1") == 0)
            {
                // 进入标准模式
                standardMode();
                printf("Main menu!\nType '1' for standard mode, '2' for expression mode, or 'quit' to exit:\n");
            }
            else if (strcmp(input, "2") == 0)
            {
                // 进入表达式模式
                expressionMode();
                printf("Main menu!\nType '1' for standard mode, '2' for expression mode, or 'quit' to exit:\n");
            }
            else if (strcmp(input, "quit") == 0)
            {
                // 退出程序
                break;
            }
            else
            {
                printf("Invalid input. Please enter '1', '2', or 'quit'.\n"
                       "Enter '1' for standard mode, '2' for expression mode, or 'quit' to exit:\n");
            }
        }

        printf("Exiting...\n");
        return 0;
    }
    else
    {
        // 检查操作数和操作符
        if (!isValidBigNum(argv[1]))
        {
            printf("The input cannot be interpreted as numbers! Please enter a valid operand (e.g., -123.45).\n");
            return 1;
        }

        char op = argv[2][0];
        if (strchr("+-*/%", op) == NULL || argv[2][1] != '\0')
        {
```

```
151             printf("Invalid operator. Only +, -, *, / and %% are
allowed.\n");
152             return 1;
153         }
154 
155         if (!isValidBigNum(argv[3]))
156         {
157             printf("The input cannot be interpreted as numbers! Please
enter a valid operand (e.g., -123.45).\n");
158             return 1;
159         }
160 
161         if (argc == 4)
162         {
163             calculate(argv[1], argv[2], argv[3], -1);
164         }
165         else if (argc == 6)
166         {
167             if (!(strcmp(argv[4], "-s") || strcmp(argv[4], "-scale")))
168             {
169                 printf("Invalid format for extension argument. Please
enter valid arguments to keep n bits after the decimal: -s(-scale) n.\n");
170                 return 1;
171             }
172 
173             int scale;
174             sscanf(argv[5], "%d", &scale); // 把argv[5]转换为整形
175 
176             calculate(argv[1], argv[2], argv[3], scale);
177         }
178     }
179 
180     return 0;
181 }
182 
183 int isValidBigNum(char *str)
184 {
185     int hasDecimalPoint = 0;
186     int hasExponent = 0;
187     int len = strlen(str);
188     int start = 0;
189 
190     // 检查符号位
191     if (str[0] == '+' || str[0] == '-')
192     {
193         start = 1;
194     }
195 
196     for (int i = start; i < len; i++)
197     {
198         if (str[i] >= '0' && str[i] <= '9')
199         {
200             continue;
201         }
202         else if (str[i] == '.' && !hasDecimalPoint && !hasExponent)
203         {
```

```
            // 小数点只允许出现一次，且不在指数部分
            hasDecimalPoint = 1;
        }
        else if ((str[i] == 'e' || str[i] == 'E') && !hasExponent && i != start)
        {
            // 'e'或'E'表示指数，只允许出现一次，且不能是第一个字符
            hasExponent = 1;
            hasDecimalPoint = 1; // 避免指数部分出现小数点
            // 检查指数后是否有符号位
            if (str[i + 1] == '+' || str[i + 1] == '-')
            {
                i++; // 跳过指数符号位
            }
        }
        else
        {
            // 非法字符
            return 0;
        }
    }

    // 最后一个字符不能是 'e' 或 'E'
    if (len > 0 && (str[len - 1] == 'e' || str[len - 1] == 'E'))
    {
        return 0;
    }

    // 合法
    return 1;
}

BigNum BigNumConstructor(char *str)
{
    BigNum bigNum;
    int strLength = strlen(str);
    int startIndex = 0;

    // 检查是否为负数
    if (str[0] == '-')
    {
        bigNum.sign = -1;
        startIndex = 1; // 跳过负号
    }
    else
    {
        bigNum.sign = 1;
    }

    // 检查是否有指数，并初始化exponent
    int eIndex = 0;
    for (; eIndex < strLength; eIndex++)
    {
        if (str[eIndex] == 'e' || str[eIndex] == 'E')
        {
            break;
```

```
        }
    }
    bigNum.exponent = atoi(str + eIndex + 1);
    if (eIndex == strLength)
    {
        bigNum.exponent = 0;
    }

    int isDecimal = 0;
    bigNum.decimal_length = 0;
    // 统计小数位数
    for (int i = startIndex; i < eIndex; i++)
    {
        if (isDecimal)
        {
            bigNum.decimal_length++;
        }
        if (str[i] == '.')
        {
            isDecimal = 1;
        }
    }

    bigNum.length = eIndex - startIndex - isDecimal;
    // 申请并分配内存
    bigNum.array = (int *)malloc(bigNum.length * sizeof(int));

    // 低位在前，从字符串末尾开始转换
    if (isDecimal)
    {
        for (int i = 0; i < eIndex; i++)
        {
            int index = eIndex - 1 - i;
            if (i < bigNum.decimal_length)
            {
                bigNum.array[i] = str[index] - '0';
            }
            else if (i > bigNum.decimal_length)
            {
                bigNum.array[i - 1] = str[index] - '0';
            }
        }
    }
    else
    {
        for (int i = 0; i < bigNum.length; i++)
        {
            bigNum.array[i] = str[eIndex - 1 - i] - '0';
        }
    }

    return bigNum;
}

void freeBigNum(BigNum *bigNum)
{
```

```
    free(bigNum->array);
    bigNum->array = NULL;
    bigNum->length = 0;
    bigNum->sign = 0;
    bigNum->decimal_length = 0;
}

void toScientificNotation(BigNum *bigNum)
{
    if (bigNum->decimal_length == bigNum->length - 1) // 只有小数部分
    {
        int index = bigNum->length - 1;
        for (; index >= 0; index--)
        {
            if (bigNum->array[index] != 0) // 从高位开始找到第一个非零数字索引
            {
                break;
            }
        }

        // 用newLength存储科学计数法的数字部分
        int newLength = index + 1;
        int *newArray = (int *)malloc(newLength * sizeof(int));
        for (int i = 0; i <= index; i++)
        {
            newArray[i] = bigNum->array[i];
        }
        free(bigNum->array);
        bigNum->array = newArray;
        bigNum->exponent -= bigNum->length - index - 1;
        bigNum->length = newLength;
        bigNum->decimal_length = bigNum->length - 1;
    }
    else
    {
        bigNum->exponent += bigNum->length - 1 - bigNum->decimal_length;
        bigNum->decimal_length = bigNum->length - 1;
    }
}

void printBigNum(BigNum *bigNum, int scale, int isScientifNotation)
{
    if (bigNum->length == 0)
    {
        printf("0\n");
        return;
    }

    // 指定用科学计数法表示或者指数部分不为0，则调整为合法的科学计数法
    if (isScientifNotation || bigNum->exponent != 0)
    {
        toScientificNotation(bigNum);
    }

    if (bigNum->sign == -1)
        printf("-");
```

```
371
372     int startIndex = 0;
373
374     if (scale == -1) // 未指定保留小数位数，舍弃末尾多余的0
375     {
376         while (bigNum->array[startIndex] == 0 && startIndex < bigNum-
>decimal_length)
377         {
378             startIndex++;
379         }
380     }
381     else
382     {
383         padDecimalPlaces(bigNum, scale);
384         int index = bigNum->decimal_length - scale - 1;
385         if (bigNum->array[index] >= 5)
386         {
387             while (index + 1 < bigNum->length && bigNum->array[index + 1]
== 9)
388             {
389                 bigNum->array[index + 1] = 0;
390                 index++;
391             }
392
393             if (index + 1 >= bigNum->length)
394             {
395                 printf("1");
396             }
397             else
398             {
399                 bigNum->array[index + 1]++;
400             }
401         }
402         startIndex = bigNum->decimal_length - scale;
403     }
404
405     // 打印整数部分
406     for (int i = bigNum->length - 1; i >= bigNum->decimal_length; i--)
407     {
408         printf("%d", bigNum->array[i]);
409     }
410     // 打印小于1的数小数点前的0
411     if (bigNum->length <= bigNum->decimal_length)
412     {
413         printf("0");
414     }
415
416     // 打印小数点
417     if (bigNum->decimal_length > startIndex)
418     {
419         printf(".");
420     }
421     // 打印小数部分
422     for (int i = bigNum->decimal_length - 1; i >= startIndex; i--)
423     {
424         printf("%d", bigNum->array[i]);
```

```
    }

    // 打印指数部分
    if (bigNum->exponent != 0)
    {
        printf("e%d", bigNum->exponent);
    }

    printf("\n");
}

// 比较两个数字大小
int compare(const BigNum *a, const BigNum *b)
{
    // 检查长度(检查长度前已经对小数进行补全)
    if (a->length > b->length)
        return 1;
    if (a->length < b->length)
        return -1;

    // 长度相同，逐位比较
    for (int i = a->length - 1; i >= 0; i--)
    {
        if (a->array[i] > b->array[i])
            return 1;
        else if (a->array[i] < b->array[i])
            return -1;
    }

    // 相等
    return 0;
}

BigNum add(BigNum *a, BigNum *b)
{
    BigNum result;
    // 最大可能长度
    int maxLength = (a->length > b->length ? a->length : b->length) + 1;
    result.array = (int *)malloc(maxLength * sizeof(int));
    result.length = 0;
    result.decimal_length = a->decimal_length;
    result.exponent = a->exponent;

    // 从低位开始，逐位相加，carry表示进位
    int carry = 0;
    for (int i = 0; i < maxLength - 1; i++)
    {
        int aDigit = i < a->length ? a->array[i] : 0;
        int bDigit = i < b->length ? b->array[i] : 0;
        int sum = aDigit + bDigit + carry;
        result.array[i] = sum % 10;
        carry = sum / 10;
        result.length++;
    }
    if (carry)
    {
```

```
        result.array[result.length] = carry;
        result.length++;
    }

    return result;
}

BigNum subtract(BigNum *a, BigNum *b, int com)
{
    BigNum result;
    result.decimal_length = a->decimal_length;
    result.exponent = a->exponent;
    // 若a < b， 则将a，b位置交换再调用函数
    if (com == -1)
    {
        result = subtract(b, a, 1);
        result.sign = -1;
    }
    else
    {
        result.array = (int *)malloc(a->length * sizeof(int));
        result.length = 0;
        result.sign = 1;

        // 从低位开始，逐位相减，borrow表示借位
        int borrow = 0;
        for (int i = 0; i < a->length; i++)
        {
            int aDigit = a->array[i];
            int bDigit = i < b->length ? b->array[i] : 0;
            int diff = aDigit - bDigit - borrow;
            if (diff < 0)
            {
                diff += 10;
                borrow = 1;
            }
            else
            {
                borrow = 0;
            }
            result.array[i] = diff;
            if (diff != 0)
                result.length = i + 1;
        }
    }
    return result;
}

void padDecimalPlaces(BigNum *bigNum, int newDecimalLength)
{
    if (newDecimalLength <= bigNum->decimal_length)
        return; // 不需要补位

    int newLength = bigNum->length + (newDecimalLength - bigNum->decimal_length);
    int *newArray = (int *)malloc(newLength * sizeof(int));
```

```
536
537     for (int i = 0; i < newDecimalLength - bigNum->decimal_length; i++)
538     {
539         newArray[i] = 0;
540     }
541     for (int i = 0, j = newDecimalLength - bigNum->decimal_length; i <
bigNum->length; i++, j++)
542     {
543         newArray[j] = bigNum->array[i];
544     }
545
546     free(bigNum->array);
547     bigNum->array = newArray;
548     bigNum->decimal_length = newDecimalLength;
549     bigNum->length = newLength;
550 }
551
552 void adjustExponent(BigNum *bigNum, int newExponent)
553 {
554     if (newExponent >= bigNum->exponent)
555         return; // 不需要调整
556
557     int exponentChange = bigNum->exponent - newExponent;
558     if (exponentChange > bigNum->decimal_length)
559     {
560         int newLength = bigNum->length + (exponentChange - bigNum-
>decimal_length);
561         int *newArray = (int *)malloc(newLength * sizeof(int));
562
563         for (int i = 0; i < exponentChange - bigNum->decimal_length; i++)
564         {
565             newArray[i] = 0;
566         }
567         for (int i = 0, j = exponentChange - bigNum->decimal_length; i <
bigNum->length; i++, j++)
568         {
569             newArray[j] = bigNum->array[i];
570         }
571
572         free(bigNum->array);
573         bigNum->array = newArray;
574         bigNum->decimal_length = 0;
575         bigNum->length = newLength;
576         bigNum->exponent = newExponent;
577     }
578     else
579     {
580         bigNum->decimal_length -= exponentChange;
581         bigNum->exponent = newExponent;
582     }
583 }
584
585 BigNum addBigNum(BigNum *a, BigNum *b)
586 {
587     if (a->exponent != 0 || b->exponent != 0)
588     {
```

```
        toScientificNotation(a);
        toScientificNotation(b);
    }

    // 若指数不一致，把指数较大的调整为和指数较小的一致
    adjustExponent(a, b->exponent);
    adjustExponent(b, a->exponent);
    // 若小数点后位数不一致，补全位数少的
    padDecimalPlaces(a, b->decimal_length);
    padDecimalPlaces(b, a->decimal_length);

    BigNum result;
    if (a->sign == 1 && b->sign == 1) // a > 0, b > 0
    {
        result = add(a, b);
        result.sign = 1;
    }
    else if (a->sign == -1 && b->sign == -1) // a < 0, b < 0
    {
        result = add(a, b);
        result.sign = -1;
    }
    else if (a->sign == 1) // a > 0, b < 0
    {
        result = subtract(a, b, compare(a, b));
    }
    else
    { // a < 0, b > 0
        result = subtract(b, a, compare(b, a));
    }

    return result;
}

BigNum subtractBigNum(BigNum *a, BigNum *b)
{
    if (a->exponent != 0 || b->exponent != 0)
    {
        toScientificNotation(a);
        toScientificNotation(b);
    }

    // 若指数不一致，把指数较大的调整为和指数较小的一致
    adjustExponent(a, b->exponent);
    adjustExponent(b, a->exponent);

    // 若小数点后位数不一致，补全位数少的
    padDecimalPlaces(a, b->decimal_length);
    padDecimalPlaces(b, a->decimal_length);

    BigNum result;
    if (a->sign == 1 && b->sign == 1) // a > 0, b > 0
    {
        result = subtract(a, b, compare(a, b));
    }
    else if (a->sign == -1 && b->sign == -1) // a < 0, b < 0
```

```
    {
        result = subtract(b, a, compare(b, a));
    }
    else if (a->sign == 1) // a > 0, b < 0
    {
        result = add(a, b);
        result.sign = 1;
    }
    else
    { // a < 0, b > 0
        result = add(a, b);
        result.sign = -1;
    }
    return result;
}

BigNum multiplyBigNum(BigNum *a, BigNum *b)
{
    BigNum result;
    int maxLength = a->length + b->length;
    result.array = (int *)malloc(maxLength * sizeof(int));
    memset(result.array, 0, maxLength * sizeof(int));
    result.length = 0;
    result.sign = a->sign * b->sign;
    result.decimal_length = a->decimal_length + b->decimal_length;
    result.exponent = a->exponent + b->exponent;

    // 先计算result的每一位数字
    for (int i = 0; i < a->length; i++)
    {
        for (int j = 0; j < b->length; j++)
        {
            result.array[i + j] += a->array[i] * b->array[j];
        }
    }

    // 统一处理进位
    for (int i = 0; i < maxLength - 1; i++)
    {
        if (result.array[i] >= 10)
        {
            result.array[i + 1] += result.array[i] / 10;
            result.array[i] %= 10;
        }
    }

    for (int i = maxLength - 1; i >= 0; i--)
    {
        if (result.array[i] != 0)
        {
            result.length = i + 1;
            break;
        }
    }

    return result;
```

```
}

// BigNum转数组顺序存储
void BigNumToArray(int *array, BigNum *bigNum)
{
    for (int i = 0; i < bigNum->length; i++)
    {
        array[i] = bigNum->array[bigNum->length - 1 - i];
    }
}

// 顺序存储的数组转BigNum
void ArrayToBigNum(int *array, BigNum *bigNum, int start, int len)
{
    bigNum->length = len - start;
    bigNum->array = (int *)malloc(bigNum->length * sizeof(int));
    for (int i = 0; i < bigNum->length; i++)
    {
        bigNum->array[i] = array[len - 1 - i];
    }
}

// last_dg表示最低位, len 表示除数的长度
bool canSubtract(int *a, int *b, int last_dg, int len)
{
    // 被除数剩余的部分比除数长
    if (last_dg - len >= 0 && a[last_dg - len] != 0)
        return true;
    // 从高位到低位，逐位比较
    for (int i = last_dg - len + 1, j = 0; j < len; i++, j++)
    {
        if (a[i] > b[j])
            return true;
        if (a[i] < b[j])
            return false;
    }
    // 相等
    return true;
}

// 判断 BigNum 是否为0
int isBigNumZero(const BigNum *num)
{
    // 如果长度为0，则认为是0
    if (num->length == 0)
        return 1;

    // 检查每一位是否都是0
    for (int i = 0; i < num->length; i++)
    {
        if (num->array[i] != 0)
            return 0; // 发现非0位，不是0
    }

    // 所有位都是0
    return 1;
```

```
}

void divideBigNum(BigNum *a, BigNum *b, BigNum *c, BigNum *d, int scale)
{
    // 计算到保留位数的后一位
    int dividend_length = scale - (a->decimal_length - b->decimal_length)
+ 1 + a->length;
    int *dividend = (int *)malloc(dividend_length * sizeof(int));
    memset(dividend, 0, dividend_length * sizeof(int));
    BigNumToArray(dividend, a);

    int *divisor = (int *)malloc(b->length * sizeof(int));
    BigNumToArray(divisor, b);

    int *quotient = (int *)malloc(dividend_length * sizeof(int));
    memset(quotient, 0, dividend_length * sizeof(int));

    int *remainder = (int *)malloc(dividend_length * sizeof(int));
    for (int i = 0; i < a->length; i++)
    {
        remainder[i] = dividend[i];
    }

    // 模拟竖式除法
    for (int i = b->length - 1; i < dividend_length; i++)
    {
        // 计算商的第i位
        while (canSubtract(remainder, divisor, i, b->length))
        {
            // 高精度减法
            for (int j = 0; j < b->length; j++)
            {
                remainder[i - j] -= divisor[b->length - 1 - j];
                if (remainder[i - j] < 0)
                {
                    remainder[i - j] += 10;
                    remainder[i - j - 1] -= 1;
                }
            }
            quotient[i]++;
        }
    }

    int start = 0;
    for (; start < dividend_length - scale - 1; start++)
    {
        if (quotient[start] != 0)
        {
            break;
        }
    }
    c->decimal_length = scale + 1;
    c->sign = a->sign * b->sign;
    c->exponent = a->exponent - b->exponent;
    ArrayToBigNum(quotient, c, start, dividend_length);

```

```
    start = 0;
    while (remainder[start] == 0)
    {
        start++;
    }
    d->decimal_length = 0;
    d->sign = 1;
    d->exponent = 0;
    ArrayToBigNum(remainder, d, start, dividend_length);
}
// 判断是否含e
int hasENotation(char *operator)
{
    for (size_t i = 0; i < strlen(operator); i++)
    {
        if (operator[i] == 'e' || operator[i] == 'E')
        {
            return 1;
        }
    }
    return 0;
}

void calculate(char *operator1, char *operator, char * operator2, int
scale)
{
    BigNum a = BigNumConstructor(operator1);
    BigNum b = BigNumConstructor(operator2);
    BigNum result;
    BigNum temp;

    if ((*operator== '/' || *operator== '%') && isBigNumZero(&b))
    {
        printf("A number cannot be divied by zero!\n");
        freeBigNum(&a);
        freeBigNum(&b);
        return;
    }

    printf("%s %s %s = ", operator1, operator, operator2);

    switch (*operator)
    {
    case '+':
        result = addBigNum(&a, &b);
        break;
    case '-':
        result = subtractBigNum(&a, &b);
        break;
    case '*':
        result = multiplyBigNum(&a, &b);
        break;
    case '/':
        // 除法默认保留六位小数
        if (scale == -1)
        {
```

```
            scale = 6;
        }
        divideBigNum(&a, &b, &result, &temp, scale);
        freeBigNum(&temp);
        break;
    case '%':
        divideBigNum(&a, &b, &temp, &result, -1);
        freeBigNum(&temp);
        break;
    default:
        break;
    }

    printBigNum(&result, scale, hasENotation(operator1) ||
hasENotation(operator2));

    freeBigNum(&a);
    freeBigNum(&b);
    freeBigNum(&result);
}

void initBigNumStack(BigNumStack *stack)
{
    stack->top = -1;
}

void initOperatorStack(OperatorStack *stack)
{
    stack->top = -1;
}

void pushBigNum(BigNumStack *stack, BigNum num)
{
    if (stack->top < STACK_SIZE - 1)
    {
        stack->items[++stack->top] = num;
    }
    else
    {
        printf("BigNum Stack overflow\n");
    }
}

BigNum popBigNum(BigNumStack *stack)
{
    if (stack->top >= 0)
    {
        return stack->items[stack->top--];
    }
    else
    {
        printf("BigNum Stack underflow\n");
        return (BigNum){NULL, 0, 0}; // 返回一个空的BigNum作为错误指示
    }
}

```

```c
void pushOperator(OperatorStack *stack, char op)
{
    if (stack->top < STACK_SIZE - 1)
    {
        stack->items[++stack->top] = op;
    }
    else
    {
        printf("Operator Stack overflow\n");
    }
}

char popOperator(OperatorStack *stack)
{
    if (stack->top >= 0)
    {
        return stack->items[stack->top--];
    }
    else
    {
        printf("Operator Stack underflow\n");
        return '\0';
    }
}

char peekOperator(OperatorStack *stack)
{
    if (stack->top >= 0)
    {
        return stack->items[stack->top];
    }
    else
    {
        return '\0';
    }
}

int isOperatorStackEmpty(OperatorStack *stack)
{
    return stack->top == -1;
}

int precedence(char op)
{
    switch (op)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    default:
        return -1;
    }
}
```

```
978
979 BigNum performOperation(BigNum *a, BigNum *b, char op)
980 {
981     BigNum result;
982     BigNum temp;
983
984     switch (op)
985     {
986     case '+':
987         result = addBigNum(a, b);
988         break;
989     case '-':
990         result = subtractBigNum(a, b);
991         break;
992     case '*':
993         result = multiplyBigNum(a, b);
994         break;
995     case '/':
996         divideBigNum(a, b, &result, &temp, 6);
997         freeBigNum(&temp);
998         break;
999     default:
1000         printf("Unsupported operation: %c\n", op);
1001         break;
1002     }
1003
1004     freeBigNum(a);
1005     freeBigNum(b);
1006     return result;
1007 }
1008
1009 void removeSpaces(char *str)
1010 {
1011     int i = 0, j = 0;
1012     while (str[i])
1013     {
1014         if (str[i] != ' ')
1015         {
1016             str[j++] = str[i]; // 复制非空格字符
1017         }
1018         i++;
1019     }
1020     str[j] = '\0'; // 添加字符串结束符
1021 }
1022
1023 void evaluateExpression(char *expression)
1024 {
1025     char initExpression[strlen(expression)];
1026     strcpy(initExpression, expression);
1027     removeSpaces(expression);
1028     BigNumStack numbers;     // 存放BigNum的栈
1029     OperatorStack operators; // 存放运算符的栈
1030     initBigNumStack(&numbers);
1031     initOperatorStack(&operators);
1032
1033     for (int i = 0; i < strlen(expression); ++i)
```

```
    {
        if (isdigit(expression[i]))
        {
            // 处理数字
            char numStr[64]; // 存储数字字符串
            int len = 0;
            while (i < strlen(expression) && (isdigit(expression[i]) ||
expression[i] == '.' || expression[i] == 'e' || expression[i] == 'E'))
            {
                numStr[len++] = expression[i++];
            }
            numStr[len] = '\0';
            BigNum num = BigNumConstructor(numStr);
            pushBigNum(&numbers, num);
            --i;
        }
        else if (expression[i] == '(')
        {
            // 遇到左括号直接压栈
            pushOperator(&operators, expression[i]);
        }
        else if (expression[i] == ')')
        {
            // 遇到右括号，弹出运算符并计算，直到遇到左括号
            while (!isOperatorStackEmpty(&operators) &&
peekOperator(&operators) != '(')
            {
                BigNum b = popBigNum(&numbers);
                BigNum a = popBigNum(&numbers);
                char op = popOperator(&operators);
                if ((op == '/' || op == '%') && isBigNumZero(&b))
                {
                    printf("A number cannot be divied by zero!\n");
                    freeBigNum(&a);
                    freeBigNum(&b);
                    return;
                }
                BigNum result = performOperation(&a, &b, op);
                pushBigNum(&numbers, result);
                freeBigNum(&a);
                freeBigNum(&b);
            }
            popOperator(&operators); // 弹出左括号
        }
        else if (strchr("+-*/", expression[i]) != NULL)
        {
            // 处理运算符，考虑优先级
            while (!isOperatorStackEmpty(&operators) &&
precedence(peekOperator(&operators)) >= precedence(expression[i]))
            {
                BigNum b = popBigNum(&numbers);
                BigNum a = popBigNum(&numbers);
                char op = popOperator(&operators);
                if ((op == '/' || op == '%') && isBigNumZero(&b))
                {
                    printf("A number cannot be divied by zero!\n");
```

```
                    freeBigNum(&a);
                    freeBigNum(&b);
                    return;
                }
                BigNum result = performOperation(&a, &b, op);
                pushBigNum(&numbers, result);
                freeBigNum(&a);
                freeBigNum(&b);
            }
            pushOperator(&operators, expression[i]);
        }
    }

    // 表达式遍历完成后，处理剩余的运算符
    while (!isOperatorStackEmpty(&operators))
    {
        BigNum b = popBigNum(&numbers);
        BigNum a = popBigNum(&numbers);
        char op = popOperator(&operators);
        if ((op == '/' || op == '%') && isBigNumZero(&b))
        {
            printf("A number cannot be divied by zero!\n");
            freeBigNum(&a);
            freeBigNum(&b);
            return;
        }
        BigNum result = performOperation(&a, &b, op);
        pushBigNum(&numbers, result);
        freeBigNum(&a);
        freeBigNum(&b);
    }

    // numbers 栈顶的BigNum元素即为最终结果
    BigNum finalResult = popBigNum(&numbers);
    printf("%s = ", initExpression);
    printBigNum(&finalResult, finalResult.decimal_length >= 6 ? 6 : -1,
0);
    freeBigNum(&finalResult);
}

void standardMode()
{
    printf("Standard Mode!\n");
    printf("Enter expressions in the format <operand1> <operator>
<operand2>.\n");
    printf("Type 'quit' to back to main menu.\n");

    char input[256];
    while (1)
    {
        printf("<Standard Mode> ");
        if (!fgets(input, sizeof(input), stdin))
        {
            break;
        }

```

```
        // 移除末尾的换行符
        input[strcspn(input, "\n")] = 0;

        // 检查是否退出
        if (strcmp(input, "quit") == 0)
        {
            break; // 退出循环
        }

        // 解析输入的表达式
        char operand1[128], operand2[128], op[2], str[10], extra[2];
        int scale = -1; // 默认的scale值
        int numParsed = sscanf(input, "%s %s %s %s %d %s", operand1, op, operand2, str, &scale, extra);
        if (numParsed == 3)
        {
            if (!isValidBigNum(operand1) || !isValidBigNum(operand2) || (strchr("+-*/%", op[0]) == NULL || op[1] != '\0'))
            {
                printf("Invalid expression format.\nPlease follow the format <operand1> <operator> <operand2>.\n");
                continue;
            }
            calculate(operand1, op, operand2, scale);
        }
        else if (numParsed == 5)
        {
            if (!isValidBigNum(operand1) || !isValidBigNum(operand2) || (strchr("+-*/%", op[0]) == NULL || op[1] != '\0'))
            {
                printf("Invalid expression format.\nPlease follow the format <operand1> <operator> <operand2>.\n");
                continue;
            }
            if (!(strcmp(str, "-s") || strcmp(str, "-scale")))
            {
                printf("Invalid format for extension argument. Please enter valid arguments to keep n bits after the decimal: -s(-scale) n.\n");
                return;
            }
            calculate(operand1, op, operand2, scale);
        }
        else
        {
            printf("Invalid input format. Please follow the format <operand1> <operator> <operand2>.\n");
        }
    }
}

void expressionMode()
{
    printf("Expression Mode!\n");
    printf("Enter your expression (type 'quit' to back to main menu):\n");

    char expression[1024]; // 表达式最大长度1024
```

```
    while (1)
    {
        printf("<Expression Mode> ");
        if (!fgets(expression, sizeof(expression), stdin))
        {
            break;
        }

        expression[strcspn(expression, "\n")] = 0;

        if (strcmp(expression, "quit") == 0)
        {
            break;
        }
        // 函数处理并计算表达式
        evaluateExpression(expression);
    }
}
```