Project1:Simple Calculator

Name: 周益贤(Zhou Yixian)
SID: 12211825

Part1 项目分析

功能实现

项目要求实现一个支持两数字四则运算的简易计算器，有标准输入和命令行参数输入两种模式。

本项目的基础功能：

1. 支持高精度及大范围数字的四则运算。
2. 支持错误计算处理。

在此基础上，本项目还实现了以下功能：

1. 更加全面的错误处理：NaN, Inf 的处理，表达式错误位置的分析等。
2. 支持含括号、比较运算在内的多元复合运算。
3. 支持无空格分隔的表达式，由计算器自行分词。
4. 支持变量定义、赋值参与运算。
5. 支持预定义的函数调用。

详细的功能介绍在 Part3.

实现思路

0. **妙妙工具**

   为了实现程序中频繁使用的变长数组、栈等数据结构，该项目先实现了一个泛型 vector，使用 void* + size 适应不同类型，记录复制函数和析构函数以确保保存的对象被正确释放。当然由于没有真正意义的泛型，只能从中读出 void* 指针，或直接将对应值拷贝到指定对象中，而不能返回实际对象。

   同时也封装了 vector 的一个 char 类型的特化用作 string.

   ~~事实上用const char*也完全可以，不过我对这方面的函数确实不熟悉，所以多造了一些轮子~~

1. **高精度运算**

   由于不确定动态整数位的定点数和浮点数哪个效果更好，两个版本我都实现了一次，最后根据表现，还是选择了浮点数。

   本次最终提交的项目实现的为浮点数，由符号位，指数位，有效数字位构成，支持至多 10^5 位有效数字，指数至多 10^8, 即绝对值超过 10^{10^8} 的值会被认为是 $\inf$，小于 10^{-10^8} 则会变为数值 0. 可以通过函数设置有效位数，输出精度，以及比较的精度。

   - 加减法：

     对齐指数位后模拟竖式加减法即可。

     为了减少不必要的拷贝，代码中实际是直接计算偏移，而不是位移对齐。

   - 乘法：

     朴素 $O(n^2)$ 乘法，为了减少取模先一次性算出每一位的值再一次性计算进位，实际表现还算可以接受。

为了避免小数字被过多的有效位数拖累，会预处理跳过零位。

- 除法：

  试除法。每次尝试减去一次除数，若成功则给对应位加一。

- 乘方：

  好吧，项目里其实没有这个部分，因为我在最后几天感冒倒下了。

  所以这方面的思考并没有变成代码。

  考虑使用 $\exp$ 和 $\ln$ 间接计算乘方：$a^b = \exp(b \ln a)$

  $\exp$ 和 $\ln$ 在大范围内的泰勒展开都是收敛很慢（或不收敛）的，因而对 $\exp$，考虑用快速幂计算整数部分，仅展开小数部分。

  $\ln(a \cdot 10^b) = b \ln 10 + \ln a$，对浮点数，$a < 1$ 天然成立，可以一定程度上加快收敛。

- 比较：

  将两数做差，若结果绝对值小于 eps 则认为相等，否则直接根据做差结果符号判断。

略去了复杂但没什么技术含量的 inf 和 NaN 处理。

2. **表达式计算**

   这部分并没有采用大家喜闻乐见的后缀表达式，因为很多混乱的表达式都可以被正常地转换为后缀表达式，即使真的发现的错误也很难分析出这部分在原式中是什么错误。

   为了更好地分析错误表达式，这次使用了递归计算的方法。

   具体方法为：预先匹配括号，若括号包含整个算式，则去除该括号，否则跳过所有括号选择优先级最低的一个运算符，根据其操作数多少递归计算剩余表达式。括号本身作为运算符，也解决了函数调用的问题。

   为了保证函数参数顺序出栈，需将其倒序入栈，因而参数列表中的表达式为倒序计算的。事实上该顺序是不保证的，不应期待参数的求值有固定顺序。

   为了更好地表示运算符结合性的问题，当运算符位于左侧和右侧时，为它分配不同的优先级，这不仅可以轻易解决左结合与右结合的区别，也保证了整个表达式有唯一的优先级最低的运算符。

3. **变量**

   变量本身没有类型，它的类型由它保存的值决定。所有变量被保存在全局变量栈中，实际是 $O(n)$ 查找的。每个变量名在运算时会被翻译为对应引用，若为允许定义（如等号左侧）的语境，则会试图创建为定义的变量，否则不允许使用未定义的变量。引用在需求数值的语境会被解引用为数值，否则保持为引用（如自增号附近）。

4. **错误处理**

   在单函数中，使用 goto 跳转到异常处理模块。

   在递归函数中，难以跳回到指定位置，故使用 longjmp 直接重置栈空间。为了避免局部变量无法释放，该策略仅限求值函数中使用，此函数中间变量均保存在全局操作数栈中，清空栈即可释放空间。

   错误处理分为四个阶段：

   - 分词阶段：遇到不合法的 token，如不合法数字、变量名、未知符号时直接结束并返回错误。
   - 预处理阶段：求值前会预先检查括号匹配情况。
   - 求值阶段：递归计算遇到不可计算的部分使用 longjmp 结束计算。
   - 运算阶段：运算时出现的错误（除零等）直接设置 NaN 等错误符号，不中断求值进程。

   异常不会撤回表达式已经进行的部分。

Part2 Code

1.vector

该部分定义了 vector 及其相关函数。

仅展示一个函数以示例用法。

```
typedef void *ObjectPtr;
typedef void (*Destructor)(ObjectPtr);
typedef void (*CopyConstructor)(ObjectPtr, ObjectPtr, size_t);

typedef struct{

    ObjectPtr data;
    size_t width;

    int size;
    size_t capacity;

    Destructor destroy;
    CopyConstructor copy;

}Vector;

ObjectPtr get_addr(Vector vec, int idx){
    return (char *)vec.data + vec.width * idx;
}
```

每当实际容量达到预分配空间时，重新分配二倍的空间。

```
void vec_recapacity(VectorPtr vec, size_t new_size){
    size_t new_capacity = vec->capacity;

    while (new_capacity < new_size)
        new_capacity *= 2;

    if(new_capacity != vec->capacity){
        vec->capacity = new_capacity;
        vec->data = realloc(vec->data,new_capacity * vec->width);
    }
}
```

为了便于遍历，这部分封装了 `FOR_EACH` 宏，可以用指针遍历 vector 的每一个元素，需要与 `END_FOR` 成对使用。

```
#define FOR_RANGE(type, name, vec, l, r)\
    do{\
        type *name;\
        for(int __i = (l); __i < (r); ++__i){\
            name = (type *)get_addr(vec, __i);

#define FOR_EACH(type, name, vec) FOR_RANGE(type, name, vec, 0, vec.size)

#define END_FOR() }}while(0)

#define CUR_IDX __i
```

string 仅为少量封装，不做赘述。

2.Number

该部分定义了高精度类及计算、比较等方法。

NumFlag 表示数字的符号或异常属性，除 NaN 外均可直接进行比较（-INF < NEG < POS < INF）

数字记录有效位数是为了在默认有效位数发生变化时及时扩展。

```
enum _numFlag{
    NUM_NINF = -2 ,NUM_NEG = -1, NUM_NAN, NUM_POS = 1, NUM_PINF
};
typedef enum _numFlag NumFlag;

typedef struct{
    char *digits;
    int place;
    int exp;
    int sign;
}Number;
typedef Number *NumberPtr;
```

加减法：

加法与减法会首先处理 inf 和 NaN，然后根据符号选择对应的无符号加减法，最后判断结果超限情况。

```
Number num_add(NumberPtr num1, NumberPtr num2){
    Number res;
    if(num_is_nan(num1) || num_is_nan(num2))
        return create_num_of_flag(NUM_NAN);
    if(num_is_inf(num1) || num_is_inf(num2)){
        if(num1->sign == num2->sign)
            return create_num_of_flag(num1->sign);
        else return create_num_of_flag(NUM_NAN);
    }
    num_adjust(num1);
    num_adjust(num2);
    if(num1->sign == num2->sign)
        res = num_add_u(num1, num2);
    else res = num_sub_u(num1, num2);
    res.sign *= num1->sign;
    num_check(&res);
```

```
    return res;
}
```

无符号加减法实现类似，这里仅展示减法。如果要硬性对齐操作数指数位，就免不了要先复制再位移，这里通过预先计算偏移避免了这一点。发生截断时会对末尾进行四舍五入，然后逐位模拟竖式减法。

```
Number num_sub_u(NumberPtr num1, NumberPtr num2){
    Number res = create_num();
    int cmp = num_cmp_u(num1, num2);
    if(cmp == 0)
        return res;
    if(cmp < 0){
        swap_numptr(&num1, &num2);
        res.sign = -1;
    }
    res.exp = num1->exp;

    int borrow = 0, delta = num1->exp - num2->exp, place = num1->place;
    //rounding correction
    if(delta > 0 && delta <= place)
        borrow = num2->digits[place - delta] < 5 ? 0 : 1;

    for(int i = place - 1; i >= 0; --i){
        res.digits[i] = num1->digits[i] - borrow;
        if(i >= delta)
            res.digits[i] -= num2->digits[i - delta];
        if(res.digits[i] < 0){
            res.digits[i] += 10;
            borrow = 1;
        }else{
            borrow = 0;
        }
    }
    num_rm_leading_zeros(&res);
    return res;
}
```

乘法：

乘法会预处理出操作数非零位的位置，然后用朴素乘法计算出每一位的值。为了避免取模的性能开销，这里先用 int 存储了每一位不取模的结果，最后一次性处理进位。

```
Number num_mul(NumberPtr num1, NumberPtr num2){
    Number res = create_num();

    if(num_is_nan(num1) || num_is_nan(num2))
        return create_num_of_flag(NUM_NAN);

    if(num_is_inf(num1) || num_is_inf(num2)){
        if(num_is_zero(num1) || num_is_zero(num2))
            return create_num_of_flag(NUM_NAN);
        if(num1->sign != num2->sign)
            return create_num_of_flag(NUM_NINF);
        else return create_num_of_flag(NUM_PINF);
    }
```

```
    num_adjust(num1);
    num_adjust(num2);

    int place = get_dec_place();
    res.sign = num1->sign * num2->sign;
    res.exp += num1->exp + num2->exp;

    int *digits = calloc(place * 2, sizeof(int));

    Vector idx = create_vec_default(sizeof(int));
    for(int j = 0; j < place; ++j)
        if(num2->digits[j] != 0)
            push_back(&idx, &j);


    for(int i = 0; i < place; i++)
        if(num1->digits[i] != 0)
            FOR_EACH(int, j, idx)
                digits[i + *j] += num1->digits[i] * num2->digits[*j];
            END_FOR();

    int carry = 0;
    for(int i = place * 2 - 1; i > place; i--)
        carry = (carry + digits[i]) / 10;

    //rounding correction
    carry += digits[place];
    carry = (carry / 10) + (carry % 10 >= 5);

    for(int i = place - 1; i >= 0; i--){
        carry +=  digits[i];
        res.digits[i] = carry % 10;
        carry /= 10;
    }

    while(carry > 0){
        num_rshift(&res, 1);
        res.digits[0] = carry % 10;
        res.exp++;
        carry /= 10;
    }

    destroy_vec(&idx);
    free(digits);
    num_check(&res);
    return res;
}
```

除法：

模拟竖式乘法，从高位到低位依次试图减去除数，成功则给结果当前位加一，否则跳到下一位。

```
Number num_div(NumberPtr num1, NumberPtr num2){
    Number res;
```

```
    if(num_is_nan(num1) || num_is_nan(num2))
        return create_num_of_flag(NUM_NAN);

    if(num_is_inf(num1) && num_is_inf(num2))
        return create_num_of_flag(NUM_NAN);

    if(num_is_inf(num1))
        return create_num_of_flag(num2->sign * num1->sign);

    if(num_is_inf(num2))
        return create_num();

    if(num_is_zero(num2)){
        if(num_is_zero(num1))
            return create_num_of_flag(NUM_NAN);
        return create_num_of_flag(NUM_PINF * num1->sign);
    }

    num_adjust(num1);
    num_adjust(num2);

    int place = num1->place;

    char *tmp = calloc(place * 2 + 1, sizeof(char));
    char *digits = calloc(place + 1 , sizeof(char));
    memcpy(tmp, num1->digits, place * sizeof(char));

    res = create_num();

    for(int i = 0; i <= place; i++){
        while(true){
            bool flag = true;
            for(int j = 0; j < place; j++){
                if(tmp[i + j] != num2->digits[j]){
                    flag = tmp[i + j] > num2->digits[j];
                    break;
                }
            }
            if(!flag)
                break;
            digits[i]++;
            for(int j = place - 1; j >= 0; --j){
                tmp[i + j] -= num2->digits[j];
                if(tmp[i + j] < 0){
                    tmp[i + j] += 10;
                    tmp[i + j - 1]--;
                }
            }
        }
        tmp[i + 1] += tmp[i] * 10;
    }

    if(digits[0] == 0){
        res.exp--;
        memcpy(res.digits, digits + 1, place * sizeof(char));
```

```
    }else{
        memcpy(res.digits, digits, place * sizeof(char));
    }

    res.sign = num1->sign * num2->sign;
    res.exp += num1->exp - num2->exp;
    num_check(&res);
    free(digits);
    free(tmp);
    return res;
}
```

比较：

做差后与零比较即可。

该函数并不判断 NaN，这部分处理封装在 `num_lt` 等比较函数中。

```
int num_cmp_eps(NumberPtr num1, NumberPtr num2){
    if(num_is_inf(num1) || num_is_inf(num2))
        return num1->sign - num2->sign;

    Number delta = num_sub(num1, num2);
    int ret = num_cmp_zero(&delta);
    destroy_num(&delta);
    return ret;
}
```

取整较容易，计算小数点位置，将其后位置全部置零即可。

这部分也提供了字符串与高精度数的互相转换，以便输入输出。

3. value & variable

“值”由一个类型和一个值指针组成，类型将决定它如何解析值指针。变量本身不具有类型，可以保存任意类型的值。

程序预定义的变量具有常量属性，只能由程序内部更新。

```
enum _varType{
    VAR_FUN, VAR_NUM, VAR_REF, VAR_BOOL, VAR_VOID, VAR_ERR, VAR_ANY
};

typedef enum _varType VarType;

typedef struct{
    VarType type;
    ObjectPtr value;
}Value;

typedef Value *ValuePtr;

typedef struct{
    Value value;
    String name;
    bool is_const;
```

```
}Variable;

typedef Variable *VarPtr;
typedef Value (*BuiltinFunc)(Vector, char*);
```

4. tokenizer

分词具体逻辑为：判断当前字符能否拼接到最后一个 token 中，即靠前串尽可能长的贪心策略。

过程较繁琐，这里只展示数字部分，其他部分都是简单的判断，不做说明。

```
if(number_flag == NUM_BEGIN){
    if(isdigit(c)){
        push_char(&token.cont, c);
    }else if(c == '.'){
        number_flag = NUM_DOT;
        push_char(&token.cont, c);
    }else if(c == 'e' || c == 'E'){
        number_flag = NUM_EXP;
        push_string(&token.cont, ".0E");
    }else{
        push_token(&res, &token, &number_flag);
        continue;
    }
}else if(number_flag == NUM_DOT){
    if(isdigit(c)){
        push_char(&token.cont, c);
    }else if(c == 'e' || c == 'E'){
        number_flag = NUM_EXP;
        push_string(&token.cont, "0E");
    }else{
        push_token(&res, &token, &number_flag);
        continue;
    }
}else if(number_flag == NUM_EXP){
    if(isdigit(c)){
        number_flag = NUM_FINISH;
        push_char(&token.cont, '+');
        push_char(&token.cont, c);
    }else if(c == '+' || c == '-'){
        number_flag = NUM_SIGN;
        push_char(&token.cont, c);
    }else{
        goto error; // incomplete number
    }
}else if(number_flag == NUM_SIGN){
    if(isdigit(c)){
        number_flag = NUM_FINISH;
        push_char(&token.cont, c);
    }else{
        goto error; // incomplete number
    }
}else if(number_flag == NUM_FINISH){
    if(isdigit(c)){
        push_char(&token.cont, c);
```

```
    }else{
        push_token(&res, &token, &number_flag);
        continue;
    }
}
```

它依据是否出现了小数点、是否出现指数符号 e/E，指数符号后是否有正负号将数字分为若干阶段，每一阶段的数字能接受不同的后继字符。

另外还需要处理两个信息。

处理1：括号匹配

递归计算表达式需要预知每个括号的匹配位置。

遇到前括号将其入栈，遇到后括号取出栈顶作为匹配即可。

```
void match_barket(Vector tokens){
    Vector stk=create_vec_default(sizeof(int));

    FOR_EACH(Token, token, tokens)
        if(cont_eq(*token,"(")){
            push_back(&stk, &CUR_IDX);
        }else if(cont_eq(*token,")")){
            int top;
            if(stk.size == 0)
                goto error;
            pop_back(&stk, &top);
            TokenPtr left_bk = (TokenPtr) get_addr(tokens, top);
            left_bk->match_pos = CUR_IDX;
            token->match_pos = top;
        }
    END_FOR();

    if(stk.size > 0)
        goto error;

    destroy_vec(&stk);
    return;

error:
    destroy_vec(&stk);
    throw_exception(ERR_INVALID_EXPR, create_str("barket mismatch"));
}
```

处理2：一元运算符检查

主要是为了区分正号和加法。逻辑为：前面有数字、后括号、自增自减（即能构成表达式得出结果），则判断为二元运算，否则认为是一元运算。

```
void unary_check(Vector tokens){
    bool last_flag = true;
    FOR_EACH(Token, token, tokens)
        if(token->type == TOKEN_SYMBOL){
            if(cont_eq(*token, "+") || cont_eq(*token, "-")
             ||cont_eq(*token, "++") || cont_eq(*token, "--"))
```

```
                if(last_flag)
                    push_char(&token->cont, '!');
            last_flag = !cont_eq(*token,")") && !cont_eq(*token, "++") &&
!cont_eq(*token, "--");
        }else{
            last_flag = false;
        }
    END_FOR();
}
```

5. operators

这部分定义了有效运算符和它们的优先级。

运算符函数也定义在此处，操作符函数在 `evaluator` 中被链接到对应符号中，内建函数在 `global_init` 中被链接到对应变量。

```
Value opfunc_add(ValuePtr val1, ValuePtr val2, char* err);
Value opfunc_neg(ValuePtr val, char* err);
Value builtin_func_max(Vector args, char* err);
```

这里提供了一个二元运算符，一个一元运算符和一个函数的定义作为例子。

它们只需计算出结果或将异常信息写入 `err` 即可，不需关心表达式的情况。

6. evaluator

跳过括号，找到优先级最低的运算符，若为数字则压入栈中，若为变量则查找引用压栈，若为括号则试图调用函数，否则使用 `SET_BIN_OP` 链接到对应的函数。

非函数调用的括号并不会触发上述机制，因为这种括号一定在某一次递归中包括了整个表达式，而求值函数中会直接剥离这种括号。

```
bool sub_eval(VectorPtr token, int l, int r, VarType expect_type){

    if(l > r)
        return false;

    TokenPtr root = (TokenPtr)get_addr(*token, l);
    if(root->match_pos == r){
        if(!sub_eval(token, l + 1, r - 1, expect_type))
            throw_exception(ERR_INVALID_EXPR,create_str("expect expression inside
'()'."));
        return true;
    }

    int p = l;
    for(int i = l; i <= r; ++i){
        TokenPtr cur = get_addr(*token, i);
        if(get_l_level(*root) > get_r_level(*cur)){
            root = cur;
            p = i;
        }
        if(cur->match_pos != -1)
            i = cur->match_pos;
```

```
    }

    char err_info[50] = {""};
    if(root->type == TOKEN_NUMBER){
        if(p != l){
            throw_exception(ERR_INVALID_EXPR, except_info(token, l, r, "Too many
expression:"));
        }
        Value val = create_val(str_to_num(&root->cont));
        push_arg_num(&val);
        destroy_val(&val);
        return true;
    }

    if(root->type == TOKEN_NAME){
        if(p != l){
            throw_exception(ERR_INVALID_EXPR, except_info(token, l, r, "Too many
expression:"));
        }
        if(expect_type == VAR_REF){
            VarPtr v = lookup_var(root->cont);
            Value val = {VAR_REF, &v->value};
            push_arg(&val);
        }else if(expect_type == VAR_ANY){
             VarPtr v = lookup_var_notnull(root->cont);
            Value val = {VAR_REF, &v->value};
            push_arg(&val);
        }else{
            VarPtr v = lookup_var_notnull(root->cont);
            push_arg(&v->value);
        }
        return true;
    }

    Value val_l, val_r;

    SET_BIN_OP("+", VAR_NUM, VAR_NUM, opfunc_add);
    //省略其他操作符。

    if(cont_eq(*root,"(")){
        if(!sub_eval(token, l, p - 1, VAR_FUN))
            throw_exception(ERR_INVALID_EXPR, except_info(token, l, r,"Expect
more operand in the expression: "));
        Vector args = args_eval(token, p + 1, root->match_pos - 1);
        Value func = pop_arg();
        Value ret = opfunc_call(&func, args, err_info);
        if(ret.type == VAR_ERR){
            destroy_val(&func);
            destroy_vec(&args);
            throw_exception(ERR_ARITHMETIC, except_info(token, l, r, err_info));
        }
        if(ret.type == VAR_VOID){
            if(expect_type != VAR_VOID)
                throw_exception(ERR_ARITHMETIC, except_info(token, l, r, "This
function has no return value: "));
        }else{
```

```
            push_arg(&ret);
        }
        destroy_val(&ret);
        destroy_val(&func);
        destroy_vec(&args);
        return true;
    }

    throw_exception(ERR_INVALID_EXPR, except_info(token, l, r,"Unknown operator:
"));
    return false;
}
```

`SET_BIN_OP` 为设置二元操作符的宏，包括了子表达式求值、操作数出栈、链接运算符函数、异常处理、操作数析构等一系列流程，`SET_UNARY_OP` 同理。

```
#define SET_BIN_OP(sym, type_l, type_r, func)\
    do{\
        if(cont_eq(*root, sym)){\
            PREPARE_BIN_OP(type_l, type_r);\
            if(type_l == VAR_NUM)\
                NUM_CHECK(val_l);\
            if(type_r == VAR_NUM)\
                NUM_CHECK(val_r);\
            Value ret = func(&val_l, &val_r, err_info);\
            if(ret.type == VAR_ERR || strlen(err_info) > 0){\
                PREPARE_BIN_END;\
                if(strlen(err_info) > 0 && ret.type == VAR_NUM)\
                    destroy_val(&ret);\
                throw_exception(ERR_ARITHMETIC, except_info(token, l, r,
err_info));\
                return true;\
            }\
            push_arg(&ret);\
            destroy_val(&ret);\
            PREPARE_BIN_END;\
            return true;\
        }\
    }while(0)
```

7.context

全局信息保存在此处。

这部分也封装了简单的异常模块。

```
void throw_exception(ErrCode code, String desc){
    Exception e = {code, desc};
    push_back(&global.ex_info, &e);
    destroy_vec(&desc);
    longjmp(global.buf, code);
}
```

事实上只是对 `longjmp` 的包装，减少了局部变量析构不完全的风险。

预定义函数也在此处链接到对应名称：

```
void link_function(const char* name, BuiltinFunc func){
    builtin_create_var(name);
    Value func_val = create_val_func(func);
    builtin_set_var(name,&func_val);
}
```

8. input/main

这部分实现计算器的主体逻辑。

在循环开始处进行 setjump，每次抛出异常后会从此处返回错误码。

刷新全局信息，读取输入，分词，计算，输出结果即可。

```
void cmd_run(){
    while(1){
        String str;
        Vector tokens;
        bool init_flag = false;

        ErrCode err_code = setjmp(global.buf);

        if(err_code == NO_EXCEPTION){
            global_flash();
            printf(">>> ");
            str = get_line();
            char *ptr = str.data;
            tokens = tokenize(1, &ptr);

            if(tokens.size == 0){
                destroy_vec(&tokens);
                destroy_vec(&str);
                continue;
            }

            init_flag = true;

            evaluate(tokens);
        }

        //long jump will jump to here!
        print_result();

        destroy_vec(&str);
        if(init_flag)
            destroy_vec(&tokens);
    }
}
```

Part3 结果展示

命令行调用不能设置精度等参数，一次运行定义的变量也不会保留到下次，因此部分功能将不演示命令行调用。

特殊指令（如退出程序等）也由函数实现，参见“6.预定义变量”部分。

计算项目的正确性均已使用 python 验证。

0. **数字格式**

   支持定点数和科学计数两种形式，小数点前后为 0 则可省略，指数为正则正号也可省略。但一旦出现指数标志 (e/E) 则必须填写指数。

   ~~是的，小数点前后全省略也不是不行。~~

```
>>> 114
114 = 114.000000000000000
>>> 114.
114. = 114.000000000000000
>>> .14
.14 = 0.140000000000000
>>> .2e10
.2e10 = 2000000000.000000000000000
>>> 1.e-10
1.e-10 = 0.000000000100000
>>> 1E+2
1E+2 = 100.000000000000000
>>> .e
ERROR:
[INVALID TOKEN]: .e
>>> 1e
ERROR:
[INVALID TOKEN]: 1e
>>> .
. = 0.000000000000000
```

1. **二元四则运算**

```
>>> 114+514
114+514 = 628.00000
>>> 114-514
114-514 = -400.00000
>>> 114*514
114*514 = 58596.00000
>>> 114/514
114/514 = 0.22179
```

   比较运算返回布尔值，可以隐式转换为数值1或0

```
>>> 1/3*3==1
1/3*3==1 = true
>>> 0.99999==1
0.99999==1 = false
```

   命令行调用：

```
root@chtholly:~/cpp/proj1# ./calculator 114+514
114+514 = 628.000000
root@chtholly:~/cpp/proj1# ./calculator 114-514
114-514 = -400.000000
root@chtholly:~/cpp/proj1# ./calculator 114*514
114*514 = 58596.000000
root@chtholly:~/cpp/proj1# ./calculator 114/514
114/514 = 0.221790
root@chtholly:~/cpp/proj1# ./calculator 114 / 514
114/514 = 0.221790
```

2. **混合表达式**

```
>>> 114+514*1919-8/10
114+514*1919-8/10 = 986479.200000000000000
>>> (114+514)*(1919-8)/10
(114+514)*(1919-8)/10 = 120010.800000000000000
>>> 114*514<=1919*10
114*514<=1919*10 = false
>>> (-30>0)*2-1
(-30>0)*2-1 = -1.000000000000000
>>> 2+ +1- -3
2++1--3 = 6.000000000000000
```

命令行调用

```
root@chtholly:~/cpp/proj1# ./calculator '1-2-3-4+5/6/7/8'
1-2-3-4+5/6/7/8 = -7.985119047619048
root@chtholly:~/cpp/proj1# ./calculator '1<2<3'
1<2<3 = true
```

（事实上1<2<3并不是数学上的连续比较，只是true=1恰好小于3罢了）

3. **高精度计算**

```
>>> 13424012408129048129048120481247171209412094712094*9301024192481029481024981048190417017501724
13424012408129048129048120481247171209412094712094*930102419248102948102498104819041701750172
4 = 124857064168173799828164128631626887916595718403453942591370192814715155521181487110928650056.000000000000000000000
>>> 13424012408129048129048120481247171209412094712094/9301024192481029481024981048190417017501724
13424012408129048129048120481247171209412094712094/930102419248102948102498104819041701750172
4 = 1443283.24820196170684183099
>>> 14980124124810481240812049 8+14701274017417401740 1204
14980124124810481240812049 8+14701274017417401740 1204 = 14994825398827898642552 1702.000000000000000000
>>> 7189341240712424012740192740174019740172401-189471951264912641481548798251745567465124750 91
7189341240712424012740192740174019740172401-189471951264912641481548798251745567465124750 91 =
-189400057852505517241421396324343827267723026 90.00000000000000000000
>>>
```

4. **函数调用**

项目提供了 ceil, floor, round, max, min 等五个数学函数和打印函数 print。

```
>>> max(1+2,3*4,min(114,514))
max(1+2, 3*4, min(114, 514)) = 114.000000000000000
>>> floor(1.2)
floor(1.2) = 1.000000000000000
>>> ceil(-4.5)
ceil(-4.5) = -4.000000000000000
>>> floor(-1.2)
floor(-1.2) = -2.000000000000000
>>> round(1.732/0.6)
round(1.732/0.6) = 3.000000000000000
```

命令行调用：

```
root@chtholly:~/cpp/proj1# ./calculator 'max(1, 1.4, 5, -1, 4)'
max(1, 1.4, 5,-1, 4) = 5.000000
root@chtholly:~/cpp/proj1# ./calculator 'print(max, max(3,2,1), round(2.718), 114*514, 1e20000)'
function 3.000000 3.000000 58596.000000 1.000000e20000
root@chtholly:~/cpp/proj1#
```

4. **变量定义及赋值、修改**

等号具有特殊优先级：它对左侧有较高优先级（低于括号），对右侧有最低优先级。

给一个未定义的变量赋值会创建改变量，使用未定义的变量则会出错。

如 `a=1+b=1+1` 会被解释为 `a=(1+(b=(1+1)))`

```
>>> a=b=1
a=b=1 = 1.000000000000000
>>> a++
a++ = 1.000000000000000
>>> ++b
++b = 2.000000000000000
>>> a
a = 2.000000000000000
>>> b
b = 2.000000000000000
>>> c=1+d=1+1
c=1+d=1+1 = 3.000000000000000
>>> c
c = 3.000000000000000
>>> d
d = 2.000000000000000
```

变量本身没有类型，它可以保存任何值，比如函数。

```
>>> a=max
a=max = function
>>> a(1,2)
a(1, 2) = 2.000000000000000
```

5. **变量的复合运算**

   变量可以像一般数值一样参与计算。

```
>>> a=114
a=114 = 114.000000000000000
>>> b=514
b=514 = 514.000000000000000
>>> a*b
a*b = 58596.000000000000000
>>> a/b
a/b = 0.221789883268482
>>> a/11+b*12+--a
a/11+b*12+--a = 6291.363636363636364
```

6. **预定义变量**

   计算器中预定义了一些数学函数，在函数调用部分已经介绍。

   除此以外还有一些功能函数：

   `set_eps(int eps)` 可以设置比较精度为 10^{-eps}。

   `set_prec(int prec)` 设置输出小数位数（科学计数下为有效数字的小数位数）

   `set_place(int place)` 为设置实际存储的有效位数，会影响计算精度和效率。

   `set_fixflag(bool flag)` 为设置输出是否使用定点表示，设为否则总采用科学计数，否则仅对 10^{5000} 以上的数使用科学计数。

```
>>> set_place(20)
>>> set_eps(5)
>>> 0.99999==1
0.99999==1 = false
>>> 0.999999==1
0.999999==1 = true
>>> set_fixflag(false)
>>> 1e200 * 1e200
1e200*1e200 = 1.000000000000000e400
>>> set_prec(5)
>>> 1e200 * 1e200
1e200*1e200 = 1.00000e400
>>> set_fixflag(true)
>>> 14*15
14*15 = 210.00000
```

   `clear(void)` 清空输出

   `exit(void)` 退出程序

   （clear 和 exit 大概没办法演示）

此外，变量`last`表示上一次运算的结果，初始为 0，运算无结果则不变，变量`true/false`为布尔值真/假。

```
>>> 2*2
2*2 = 4.000000000000000
>>> last
last = 4.000000000000000
>>> last*last
last*last = 16.000000000000000
>>> last*last
last*last = 256.000000000000000
```

7. **异常处理**

   数值异常：

   与一般 inf 和 NaN 计算方法相同。

```
>>> nan=0/0
nan=0/0 = Not a Number
>>> inf=1/0
inf=1/0 = Infinity
>>> -1/0
-1/0 = -Infinity
>>> inf+inf
inf+inf = Infinity
>>> inf-inf
inf-inf = Not a Number
>>> inf*0
inf*0 = Not a Number
>>> inf*-inf
inf*-inf = -Infinity
>>> inf>1e200
inf>1e200 = true
>>> -1e200>-inf
-1e200>-inf = true
>>> nan==nan
nan==nan = false
>>> inf==inf
inf==inf = true
```

   类型异常：

```
>>> min(min)
ERROR:
[ARITHMETIC ERROR]: argument should be number: 'min(min)'
>>> set_eps(1,2)
ERROR:
[ARITHMETIC ERROR]: expect exactly 1 argument: 'set_eps(1, 2)'
>>> set_fixflag(1)
ERROR:
[ARITHMETIC ERROR]: argument should be bool: 'set_fixflag(1)'
```

   表达式错误：

```
>>> (114
ERROR:
[INVALID EXPRESSION]: barket mismatch
>>> ?
ERROR:
[INVALID TOKEN]: ?
>>> 1e
ERROR:
[INVALID TOKEN]: 1e
>>> print(1,,2)
ERROR:
[INVALID EXPRESSION]: expect expression in argument list: '1,, 2'
>>> 1++2
ERROR:
[INVALID EXPRESSION]: too many arguments for unary operator:  '1++2'
>>> 1+print(1)
1.000000000000000
ERROR:
[ARITHMETIC ERROR]: This function has no return value:  'print(1)'
```

变量错误：

```
>>> z
ERROR:
[UNKNOW NAME]: variable 'z' is undefined
>>> last=1
ERROR:
[ARITHMETIC ERROR]: Can not modify a const variable in expression: 'last=1'
>>> 1=1
ERROR:
[ARITHMETIC ERROR]: Can not modify a rvalue in expression: '1=1'
```

Part4 疑难与解决

- 一元、二元、括号复合的运算，以及赋值号的特殊优先级难以处理。

  当运算符位于左侧和右侧时，为它分配不同的优先级，只要优先级设置正确，选择优先级最低的运算符总能获得正确结果。

- 内层发生异常时层层退出代码极其繁琐。

  使用 goto 和 longjmp 一步到位，处理好局部变量的析构即可。

- 处理运算逻辑涉及大量重复代码。

  使用宏定义封装，每次添加运算符只需要将符号链接到处理函数即可。

- 命令行参数调用时，单独的乘号以及一些其他特殊符号会被翻译为通配符。

  这个真没办法，请您输入复杂命令行参数时用引号包括表达式。

- 迭代器失效。

  是的，既然是vector，就免不了遇到这个问题。这个问题主要来自于代码中变量引用的实现，它直接保存`global.var_table`中的地址，这显然是十分危险的。如果一次性定义多个变量，恰好在某一个表达式计算到一半时这个 vector 重新分配空间，则之前的引用会失效。

  解决方案是简单的，改为保存下标而非地址即可。

Part5 总结与感想

很早就开始，很晚才结束，经过两次重构，总算还是在 ddl 之前端了上来。

其实还有很多想写：控制语句、更复杂的数学函数…… 但是这个单文件项目已经2400+行了，身体也不是很能撑住，就这样打住吧。

写 proj 是个奇妙的过程，每天都会有新的奇思妙想蹦出来，项目永远有优化不完的地方，这个项目的大小也一路飙升，逐渐刹不住车。

本来以为 C 和 C++ 区别不大，实际写起来才发现举步维艰，写的过程也是一个不断学习的过程，收获了很多奇妙的用法。

有人说没有找到更有意义的事才会写这么多，我不敢苟同。有意义的事有很多，并不是“最有意义的”那一件事才值得做。每天写一点代码事实上也没有耽误我做别的事，决定要写而又不尽心尽力地写，才是真正的没有意义吧。

我不觉得这样有什么不好，就保持这样的热情，继续下去吧！

Part6 第三方内容及 AIGC 声明

本项目没有使用 AI 直接生成代码框架、代码片段、报告内容，但是我使用的代码补全插件有 AI 辅助，所以不能说是完全没有 AI 成分的（虽然事实上只是省下了一些 ctrl cv 的操作）。

本项目没有直接使用使用第三方库或代码片段。高精度的实现细节有参考 OI Wiki 和 GMP 库，vector 的实现有参考标准库的 std::vector。