CS205-2022Fall Project 2

A Better Calculator

Name: 陈康睿

SID: 12110524

Part 1 - Analysis & Implementation

To implement a much more complex calculator than project 1 that is capable to calculate expressions with different operators, functions as well as symbols, this program should be able to analyse string as calculation expressions and store statistics. Also, for precision purpose, it needs dynamic data structures.

(Note: only core codes are elaborated in the report, the whole project is available in My GitHub Repository)

Typedef & Namespace

```
typedef long long ll;
typedef unsigned long long ull;
```

```
using std::vector;
using std::string;
using std::map;
using std::pair;
using std::to_string;
using std::min;
using std::max;
using std::swap;
using std::make_pair;
using std::stack;
using std::cin;
using std::cout;
using std::endl;
```

Arbitrary precision

Instead of using primary limited floating data types, I used an self-degined structure `number` to store data in the project, which used a `std::vector<short>` to store precise digits, a `bool` to record whether the number is negative, a `long long` to maintain the exponent and a `char` for posible operators when analysing expressions. It can be initialized through different ways. The most important one is to parse a `string`. It also use a function `simplify()` to standardize the storage of data. Another useful one is `limit_precision()`, which is used to limit the number numbers' precise digits to fit the global percision setting. Not the least, `to_s()` converts number to a string for output.

```
struct number {
  char op = 0;
  bool negative = false;
  ll exp = 0;
  vector<short> digit;  // use short to save memory
}

number::number(const string &s) {
    if (s == "NaN" || s == "Error") {  // Not a number or syntax error
        digit.push_back(s == "NaN" ? -1 : -2);
        return;
    }
    if (s[0] == '-') negative = true;
    size_t dot = -1, e = -1, ms = -1;
    ll temp = 0;
    for (size_t i = (s[0] == '-' || s[0] == '+'); i < s.length(); i++)
```

```cpp
        switch (s[i]) {
            case '.':
                dot = i;
                break;
            case 'E':
            case 'e':
                e = i;
                break;
            case '-':
                ms = i;
                break;
            default:
                if (~e) temp = temp*10+(s[i]-'0');  // exponent
                else {
                    digit.push_back(s[i]-'0');  // percise digits
                    if (~dot) --exp;
                }
                break;
        }
    if (~ms) temp *= -1;
    exp += temp;
    reverse(digit.begin(), digit.end());
    simplify();
}

/*
 * Delete prefix & postfix zeros
 * For efficiency, reload the whole vector instead of erasing iterators
 */
void number::simplify() {
    if (digit.empty()) {  // clear to 0
        negative = false;
        exp = 0;
        return;
    }

    // find intevals of pre/postfix 0
    int tail = 0, head = digit.size()-1;
    while (tail < digit.size() && !digit[tail]) ++tail;
    while (~head && !digit[head]) --head;
    if (!tail && head == digit.size()-1) return;
    if (head < tail) {
        digit.clear();
        negative = false;
        exp = 0;
    } else {
        vector<short> temp;
        for (size_t i = tail; i <= head; ++i)
            temp.push_back(digit[i]);
        exp += tail;
        swap(digit, temp);
    }
}

// limit the percision of the number
```

```
void number::limit_percision(const size_t &limit) {
    reverse(digit.begin(), digit.end());
    while (digit.size() > limit) {  // limit percision
        digit.pop_back();
        ++exp;
    }
    reverse(digit.begin(), digit.end());
    simplify();
}

/* convert number to string for output
 * try to make the output more friendly
 * instead of just putting all the '0' out
 * or using scientific notation
 */
string number::to_s() const {
    if (digit.empty()) return "0";  // 0 alone
    if (digit[0] == -2) return "Syntax Error!";
    if (!~digit[0]) return "NaN";  // not a number
    string ret;
    if (op) return ret += op;
    if (negative) ret += "-";
    if (-exp == digit.size()) {  // 0.
        ret += "0.";
        for (int i = digit.size()-1; ~i; --i)
            ret += to_string(digit[i]);
    } else {
        ret += to_string(digit[digit.size()-1]);
        if (exp >= 0) {  // positive exponent
            if (exp <= digit.size()) {  // use postfix 0
                for (int i = digit.size()-2; ~i; --i)
                    ret += to_string(digit[i]);
                for (int i = 0; i < exp; ++i) ret += "0";
            } else {  // use scientific notation
                if (digit.size() > 1) {
                    ret += ".";
                    for (int i = digit.size()-2; ~i; --i)
                        ret += to_string(digit[i]);
                }
                ret += "e" + to_string(exp+(int)digit.size()-1);
            }
        }
        else {  // negative exponent
            if (-exp < digit.size()) {  // radix point in middle
                for (int i = digit.size()-2; i >= -exp; --i)
                    ret += to_string(digit[i]);
                ret += ".";
                for (int i = -exp-1; ~i; --i)
                    ret += to_string(digit[i]);
            } else {  // use scientific notation
                if (digit.size() > 1) {
                    ret += ".";
                    for (int i = digit.size()-2; ~i; --i)
                        ret += to_string(digit[i]);
                }
```

```
                    ret += "e"+to_string(exp+(int)digit.size()-1);
                }
            }
        }
        return ret;
}
```

Calculator Setting

I used another structure `calculator` to represent the calculator it has a `size_t precision` to store precision setting, a `const map<string, size_t> fun` predetermined function informations `(string name, size_t number of parameters)`, and a `map<string, number> var` to store variables. I implemented most of the calculation in calculator's member functions.

```
struct calculator {
    size_t precision = 4;  // used in division & sqrt, 4 digit more by default
    const map<string, size_t> fun = {{"abs", 1}, {"opp", 1}, {"sqrt", 1},
                                     {"pow", 2}, {"random", 0}};
    map<string, number> var;
}
calculator::calculator() {
    srand(time(NULL));  // used in random()
}
```

Arithmetic Operations

Comparison

Compare the value of two numbers, used a lot in other functions. First compare sign, then `0` as special case, then using exponent+size, and finally each bit.

```
/*
 * compare two numbers
 * -1: a < b
 *  0: a = b
 *  1: a > b
 */
int calculator::compare(const number &a, const number &b) const {
    // compare by sign
    if (a.negative && !b.negative) return -1;
    if (!a.negative && b.negative) return 1;

    // when a or b is 0
    if (a.digit.empty() && b.digit.empty()) return 0;
    if (a.digit.empty()) return (b.negative ? 1 : -1);
    if (b.digit.empty()) return (a.negative ? -1 : 1);

    // compare by size
    int coe = (a.negative ? -1 : 1);
    if (a.exp+(ll)a.digit.size() > b.exp+(ll)b.digit.size()) return coe;
    if (a.exp+(ll)a.digit.size() < b.exp+(ll)b.digit.size()) return -coe;

    // compare by each bit
    for (size_t ia = a.digit.size()-1, ib = b.digit.size()-1; ~ia && ~ib; --ia,
--ib) {
        if (a.digit[ia] > b.digit[ib]) return coe;
        if (a.digit[ia] < b.digit[ib]) return -coe;
    }
    if (a.digit.size() > b.digit.size()) return coe;
    if (a.digit.size() < b.digit.size()) return -coe;
    return 0;
}
```

Addition

For I was using floating method to store number, the first step of addition is to align their percise digits (i.e. same exponent after complementing with `0`), and then add up each digit and deal with the carriers.

```
// addition and substraction
number calculator::add(const number &a, const number &b) const {
    if (a.error() || b.error()) return number("Error");
    if (a.nan() || b.nan()) return number("NaN");

    number ret;
    if (a.negative == b.negative) {  // same sign, true addition
        ret.negative = a.negative;

        // initialize the length of the result
        int low = min(a.exp, b.exp),
            high = max(a.exp+(ll)a.digit.size(), b.exp+(ll)b.digit.size())-1;
        ret.exp = low;

        // digit addition
        size_t idx;
        for (int i = low; i <= high; ++i) {
            idx = i-low;
            if (ret.digit.size() == idx) ret.digit.push_back(0);
            ret.digit[idx] += ((i>=a.exp && i < a.exp+(ll)a.digit.size()) ?
                               a.digit[i-a.exp] : 0)+
                              ((i>=b.exp && i < b.exp+(ll)b.digit.size()) ?
                               b.digit[i-b.exp] : 0);
            if (ret.digit[idx] >= 10) {  // carry
                ret.digit[idx] -= 10;
                ret.digit.push_back(1);
            }
        }
    } else {  // oposite sign, substraction actually
        ...
    }
    ret.simplify();
    return ret;
}
```

Substraction

I also implemented substraction in `add()` and consider it as two opposite number added up. To make the process of borrowing easy, I force the one whose absolute value is larger to be the minuend, and then special process the sign.

```
// addition and substraction
number calculator::add(const number &a, const number &b) const {
    if (a.error() || b.error()) return number("Error");
    if (a.nan() || b.nan()) return number("NaN");

    number ret;
    if (a.negative == b.negative) {  // same sign, true addition
        ...
    } else {  // oposite sign, substraction actually
        // make sure it's big - small
        number x = abs(a), y = abs(b);
        int comp = compare(x, y);
        if (!comp) return ret;
        if (!~comp) {  // abs(b) > abs(a)
            ret.copy(b);
            swap(x, y);
        } else ret.copy(a);  // abs(a) > abs(b)

        // digit substraction
        vector<short> temp;
        if (x.exp > y.exp) {
            for (int i = x.exp-y.exp; i > 0; --i) {
                temp.push_back(0);
                --ret.exp;
            }
            for (size_t i = 0; i < ret.digit.size(); ++i)
                temp.push_back(ret.digit[i]);
            swap(ret.digit, temp);
        } else if (y.exp > x.exp) {
            for (int i = y.exp-x.exp; i > 0; --i)
                temp.push_back(0);
            for (size_t i = 0; i < y.digit.size(); ++i)
                temp.push_back(y.digit[i]);
            swap(y.digit, temp);
        }
        for (size_t i = 0; i < y.digit.size(); ++i)
            ret.digit[i] -= y.digit[i];

        // borrow
        for (size_t i = 0; i < ret.digit.size(); ++i)
            if (ret.digit[i] < 0) {
                ret.digit[i] += 10;
                ret.digit[i+1] -= 1;
            }
    }
    ret.simplify();
    return ret;
}
```

Multiplication

This is the part implemented in the last projects. I first use `0` to fill answer's `vector<short> digit` to enough size for convenience, then iterating idx to add up and dealing with carryer is easier to handle, and finally add up their exponents.

```
number calculator::multiply(const number &a, const number &b) const {
    if (a.error() || b.error()) return number("Error");
    if (a.nan() || b.nan()) return number("NaN");

    number ret;
    if (a.digit.empty() || b.digit.empty()) return ret;
    ret.negative = a.negative^b.negative;
    ret.exp = a.exp+b.exp;
    for (size_t i = a.digit.size()+b.digit.size(); ~i; --i)
        ret.digit.push_back(0);  // initiallize possible length
    for (size_t i = 0, idx; i < a.digit.size(); ++i) {
        if (!a.digit[i]) continue;
        for (size_t j = 0; j < b.digit.size(); ++j) {
            idx = i+j;
            ret.digit[idx] += a.digit[i]*b.digit[j];
            if (ret.digit[idx] >= 10) {  // carry
                ret.digit[idx+1] += ret.digit[idx]/10;
                ret.digit[idx] %= 10;
            }
        }
    }
    ret.simplify();
    return ret;
}
```

Division

Division is the hardest part of arithmetic operations because percision issues. I implemented the function much like we do as human: finding the differnece of their exponents, then trying out each bit of the quotient, which uses all other operations implemented before. I used binary answer to accelerate the process. Mention worthy that when the divisor is `0`, the answer is `NaN` (Not a Number).

```
// division
number calculator::divide(const number &a, const number &b) const {
    if (a.error() || b.error()) return number("Error");
    if (a.nan() || b.nan()) return number("NaN");
    number ret;
    // speacial cases
    if (b.digit.empty() || !~b.digit[0] ||
        (!a.digit.empty() && !~a.digit[0])) {  // NaN
        ret.digit.push_back(-1);
        return ret;
    }
    if (a.digit.empty()) return ret;  // 0

    ret.negative = a.negative^b.negative;
    ret.exp = a.exp-b.exp+1;

    // initialize dividend x and divisor y
    number x, y;
    size_t ia = a.digit.size()-1;
    for (size_t ib = 0; ib < b.digit.size(); ++ib) {
        if (~ia) x.digit.push_back(a.digit[ia--]);
        else {
            ++x.exp;
            --ret.exp;
        }
        y.digit.push_back(b.digit[ib]);
    }
    reverse(x.digit.begin(), x.digit.end());
    x.simplify();
    if (!~compare(x, y)) {  // make sure x >= y
        ret.digit.push_back(0);
        if (~ia)
            x = add(multiply(x, number(10)), number(a.digit[ia--]));
        else {
            ++x.exp;
            --ret.exp;
        }
        x.simplify();
    }

    // calculate each bit of quotient
    short l, r, mid;
    for (size_t i = 0;
         i < max(a.digit.size(), b.digit.size())+precision; ++i) {
        if ((ll)x.digit.size()+x.exp < (ll)y.digit.size()+y.exp ||
            !~compare(x, y)) {  // skip 0s
            ret.digit.push_back(0);
            if (~ia)
                x = add(multiply(x, number(10)), number(a.digit[ia--]));
            else {
                ++x.exp;
                --ret.exp;
            }
            x.simplify();
            continue;
        }
        l = 2, r = 9;
        while (l <= r) {  // try possible digits, with binary optimization
            mid = l+r>>1;
            if (!~compare(add(x, multiply(y, number(-mid))), number()))
```

```
                r = mid-1;
            else l = mid+1;
        }
        ret.digit.push_back(r);
        x = add(x, multiply(y, number(-r)));
        if (~ia) x = add(multiply(x, number(10)), number(a.digit[ia--]));
        else {
            ++x.exp;
            --ret.exp;
        }
        x.simplify();
    }
    reverse(ret.digit.begin(), ret.digit.end());
    ret.simplify();
    return ret;
}
```

Functions

Absolute

Calculate the absolute value of a number, simply make `bool negative` to `false`.

```
// return the absolute number of x
number calculator::abs(const number &x) const {
    if (x.error()) return number("Error");
    if (x.nan()) return number("NaN");

    number ret;
    ret.copy(x);
    ret.negative = false;
    return ret;
}
```

Opposite

Calculate the opposite value of a number, simply invert `bool negastive`.

```
// return the opposite number of x
number calculator::opp(const number &x) const {
    if (x.error()) return number("Error");
    if (x.nan()) return number("NaN");

    number ret;
    ret.copy(x);
    ret.negative = !x.negative;
    return ret;
}
```

Square Root Calculation

Calculate the square root of a number, a function need some mathematics techniques. I chose Newton's Method $(x_{n+1} = \frac{x_n^2+a}{2x_n})$ to find the root, which converge very fast. Repeat the iteration until the error is lower than eps (a constant set according to global precision setting). I also limite the intermediate precision to accelerate.

```
// calculate the sqrt of x
number calculator::sqrt(const number &a) const {
    if (a.error()) return number("Error");
    if (a.nan()) return number("NaN");

    number x, ret, temp, eps(1);

    //minimize abs(exp) of x, add it up later
    ll exp = a.exp/2;
    x.copy(a);
    x.exp = a.exp%2;
    if (x.negative) {
        ret.digit.push_back(-1);
        return ret;
    }
    temp.copy(x);
    size_t limit = x.digit.size()+precision<<1;
    eps.exp = -precision<<1;
    int i = 0;
    // Newton's method
    while (compare(abs(add(ret, opp(temp))), eps) > 0) {
        ret.copy(temp);
        temp = divide(add(multiply(ret, ret), x), multiply(ret, number(2)));
        temp.limit_precision(limit);  // limit temp's precise digits
    }
    ret.exp += exp;
    ret.limit_precision(x.digit.size()+precision);
    return ret;
}
```

Power

This calculator can calculate a number's n^{th} power $(n \in \mathbb{N})$, using binary power to accelerate.

```
// calculate the N-th pow of x
number calculator::pow(const number &x, const number &y) const {
    // check the validity of y
    if (x.error() || y.error() || y.negative || y.exp < 0 ||
        compare(y, number(__LONG_LONG_MAX__)) > 0)
        return number("Error");
    if (x.nan() || y.nan()) return number("NaN");

    size_t t = y.to_t();  // convert y into integer
    number ret = number(1), temp;
    temp.copy(x);
    while (t) {  // binary accelerate
        if (t&1) ret = multiply(ret, temp);
        temp = multiply(temp, temp);
        t >>= 1;
    }
    return ret;
}
```

Random

Generate a number with the global precision digits, random exponent and sign.

```
number calculator::random() const {
    number ret;
    for (int i = 0; i <= precision; ++i) ret.digit.push_back(rand()%10);
    // make sure the MSB is not 0
    while (!ret.digit.back()) ret.digit.back() = rand()%10;
    ll exp = (rand()&1 ? rand() : -rand());
    ret.exp = exp%precision-(exp < 0)*precision;
    ret.negative = rand()&1;
    ret.simplify();
    return ret;
}
```

Variable Definition

Use `map<string, number> var` to store variables, but check the validity of names in advance. Globle precision setting is dealed here as well.

```
// define or modify variables
void calculator::assign(const string &s, const number &x) {
    if (s == "precision") {  // modify precision setting
        if (x.nan() || x.error() || x.negative || x.exp < 0 ||
            compare(x, number(__LONG_LONG_MAX__)) > 0) {
            cout << "Invalid precision";
            return;
        }
        precision = x.to_t();
        cout << "Precision is set to " << x.to_t();
        return;
    }
    if (fun.count(s)) {
        cout << "Can not define names of functions as variables";
        return;
    }
    if (isdigit(s[0])) {
        cout << "Variable names can not start with a number";
        return;
    }
    for (int i = 0; i < s.length(); ++i)
        if (!validch(s[i])) {
            cout << "Unsupported character(s) for variable names";
            return;
        }
    var.insert_or_assign(s, x);
    cout << "Assign " << x.to_s() <<" to " << s;
}
```

Expression Analysis

My implementation is to classify the expression to assignment or calculation. If `'='` appears, I consider it as assignment, otherwise calculation. For assignment, I count `'='` for validity , then see the left as the name, the right side as a calculation expression , and at last check the name, calculate the value, then insert into or modify `map<string, number> var`. For calculation, my method is to get data and operators in turns, and use stack to convert the indix expression to postfix, and calculate each data element through recursion. There are too many stuff for string processing and validity checking, especially when choping up function's parameters and designing stack operators. See the code below for details. I put down many other functions to support this function.

```
// check whether c can be used in a variable name
bool validch(char c);

// define the priority of signs
int order(char c);

/*
 * return (data_type, length)
 * data_type:
 * -1: error
 *  0: number
 *  1: functions
 */
pit calculator::get_a_data(const string &s, size_t begin) const {
    size_t len = 0;
    if (isdigit(s[begin])) {
        bool dot = false;  // a flag for decimal
        while (begin+len < s.length() &&
                (isdigit(s[begin+len]) || s[begin+len] == '.')) {
            if (s[begin+len] == '.') {
                if (dot) return make_pair(-1, 0);
                else dot = true;
            }
            ++len;
        }
        return make_pair(0, len);
    } else {  // a variable or a function
        while (begin+len < s.length() && validch(s[begin+len])) ++len;
        string name = s.substr(begin, len);
        if (var.count(name)) return make_pair(0, len);
        if (fun.count(name)) return make_pair(1, len);
    }
    return make_pair(-1, 0);
}
```

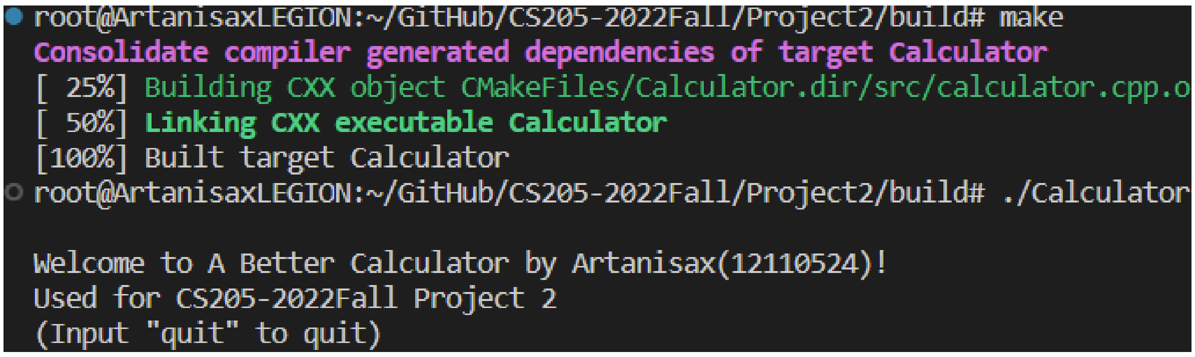
```
// for "fun(s)", split s into seperate parameters
vector<string> split(const string &s) {
    vector<string> ret;
    size_t cnt = 0, begin = 0;
    for (size_t i = 0; i < s.length(); ++i) {
        cnt += (s[i] == '(')-(s[i]==')');
        if (!cnt && s[i] == ',') {  // split using ','
            ret.push_back(s.substr(begin, i-begin));
            begin = i+1;
        }
    }
    // add the final substring or make it as
    if(!s.empty()) ret.push_back(s.substr(begin, s.length()-begin));
    return ret;
}

// calculate two numbers
number calculator::bin_calc(const number &a, const number &b, const char &c)
const {
    switch (c) {
        case '+': return add(a, b);
        case '-': return add(a, opp(b));
        case '*': return multiply(a, b);
        case '/': return divide(a, b);
        default: return number("Error");
    }
}

number calculator::fun_calc(const string &name, const vector<string> &parameter)
const {
    if (parameter.size() != fun.find(name)->second) return number("Error");
    if (name == "abs") return abs(calculate(parameter[0]));
    if (name == "opp") return opp(calculate(parameter[0]));
    if (name == "sqrt") return sqrt(calculate(parameter[0]));
    if (name == "pow") return pow(calculate(parameter[0]),
calculate(parameter[1]));
    if (name == "random") return random();
    return number("Error");
}

// calculate an expression
number calculator::calculate(const string &s) const {
    stack<char> op;
    stack<number> data;
    bool flag = true;
    for (size_t i = 0, len; i < s.length();
         i += len, flag = !flag) {
        if (flag) { // expect a data or '('s
            while (s[i] == '(') {
                op.push('(');
                if (++i == s.length()) return number("Error");
            }
            //data may start with a sign
            bool sign = (s[i] == '+' || s[i] == '-');
            // try to get a data (type, len)
            pit temp = get_a_data(s, i+sign);
            int type = temp.first;
            len = temp.second;

            //deal with different type of data
            string sub;
            switch (type) {
                case -1: return number("Error");
                case 0:  // a number
                    len += sign;
                    sub = s.substr(i, len);
                    if (isdigit(sub[0]) ||
                        sub[0] == '+' ||sub[0] == '-')
                        data.push(number(sub));
                    else data.push(var.find(sub)->second);
                    break;
                case 1:  // a function
                    i += sign;
                    if (i+len == s.length() || s[i+len] != '(')
                        return number("Error");
                    size_t cnt = 1, j;
                    // find the paired ')'
                    for (j = i+len+1; j < s.length() && cnt; ++j)
                        cnt += (s[j] == '(')-(s[j]==')');
                    if (cnt) return number("Error");
                    // split parameters and calculate the function
                    number result = fun_calc(s.substr(i, len),
                                    split(s.substr(i+len+1, j-i-len-2)));
                    if (result.error()) return number("Error");
                    data.push((sign && s[i-1] == '-') ?
                              opp(result) : result);
                    len = j-i;
                    break;
            }
        } else {  // expect a operator or a ')'
            while (i < s.length() && s[i] == ')') {
                while (!op.empty() && op.top() != '(') {
                    data.push(number(op.top()));
                    op.pop();
                }
                if (op.empty()) return number("Error");
                else op.pop();
                ++i;
            }
            if (i == s.length()) break;
            if (!~order(s[i])) return number("Error");
            else {
                if (!op.empty() && order(op.top()) >= order(s[i])) {
                    data.push(number(op.top()));
                    op.pop();
                }
                op.push(s[i]);
            }
```

```cpp
                len = 1;
            }
        }

        while (!op.empty()) {  // convert into reversed postfix expression
            if (op.top() == '(') return number("Error");
            data.push(number(op.top()));
            op.pop();
        }

        // cnt(operator) must be cnt(number)-1 in postfix expression
        if (!(data.size()&1)) return number("Error");

        // calculate postfix expression
        stack<number> temp;
        while (!data.empty()) {  // generate a postfix expression
            temp.push(data.top());
            data.pop();
        }
        number cur, a, b;
        while (!temp.empty()) {  // calculate the postfix expression
            cur = temp.top();
            if (cur.op) {
                b = data.top();
                data.pop();
                a = data.top();
                data.pop();
                data.push(bin_calc(a, b, cur.op));
            } else data.push(cur);
            temp.pop();
        }
        return data.top();
    }

    // define or modify variables
    void calculator::assign(const string &s, const number &x);

    /*
     * check the validity of the expression
     * classify expression to variable assignment or calculation
     */
    string calculator::analyse(string &s) {
        // remove ' '
        if (count(s.begin(), s.end(), ' ')) {
            string temp;
            for (size_t i = 0; i < s.length(); ++i)
                if (s[i] != ' ') temp.push_back(s[i]);
            swap(s, temp);
        }

        // quit
        if (s == "quit") {
            cout << "\nThanks for using. Good bye!\n" << endl;
            exit(0);
        }
        // assignment
        size_t cnt = count(s.begin(), s.end(), '=');
        if (cnt)
            if (cnt == 1) {
                size_t idx = s.find("=");
                string name = s.substr(0, idx);
                number value = calculate(s.substr(idx+1, s.length()-idx-1));
                if (value.error()) return "Syntax Error!";
                assign(name, value);
                return "";
            } else return "Syntax Error!";

        // calculate
        number ans = calculate(s);
        if (ans.error()) return "Syntax Error!";
        return ans.to_s();
    }
```

Part 2 - Results & Verification

Welcome

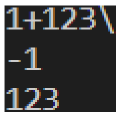
To show devs warmth, a welcome is indispensable.



```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project2/build# make
Consolidate compiler generated dependencies of target Calculator
[ 25%] Building CXX object CMakeFiles/Calculator.dir/src/calculator.cpp.o
[ 50%] Linking CXX executable Calculator
[100%] Built target Calculator
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project2/build# ./Calculator

Welcome to A Better Calculator by Artanisax(12110524)!
Used for CS205-2022Fall Project 2
(Input "quit" to quit)
```

Multiple Line Input

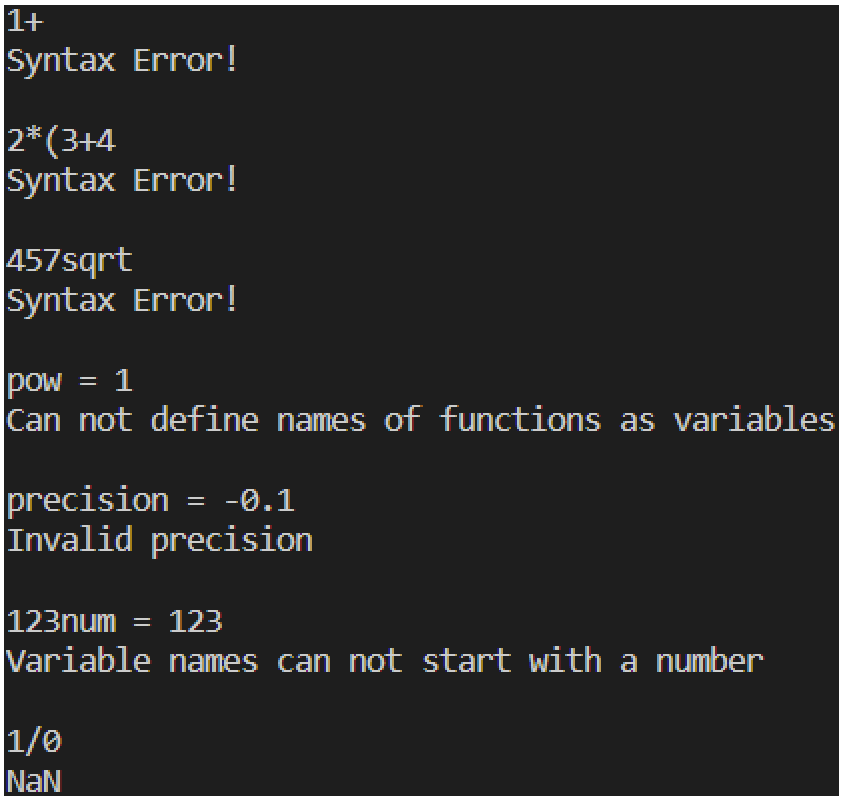
User can put a `'\'` at the end of the line, so the program will wait till the next line is input and append it to the original expression for analysis.



```
1+123\
-1
123
```

Invalid Cases

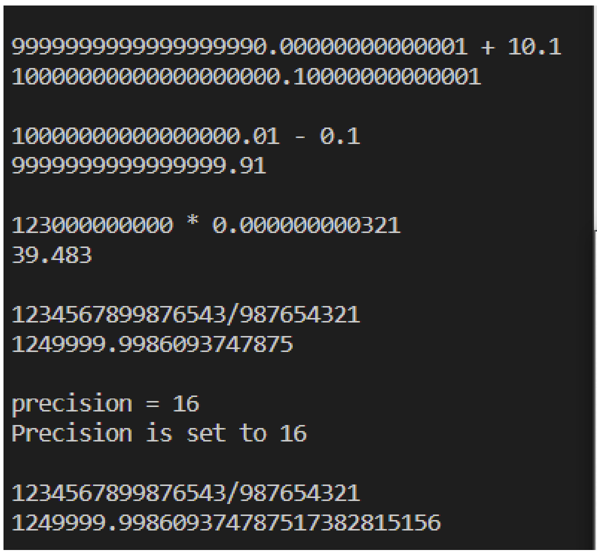
Invalid cases include syntax error, not a number, invalid name, name conflict ...



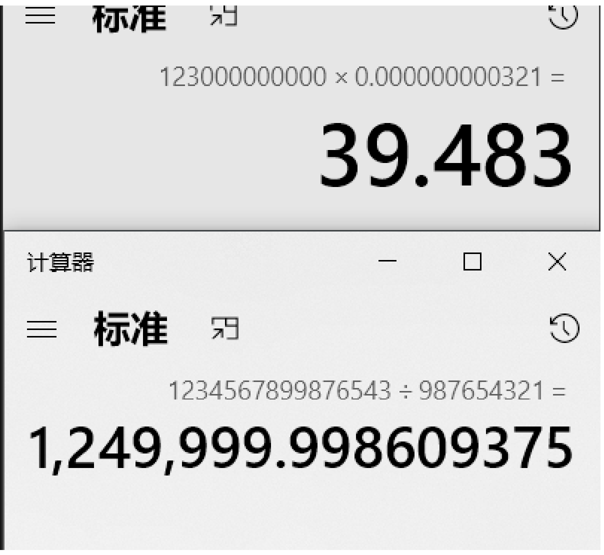
```
1+
Syntax Error!

2*(3+4
Syntax Error!

457sqrt
Syntax Error!

pow = 1
Can not define names of functions as variables

precision = -0.1
Invalid precision

123num = 123
Variable names can not start with a number

1/0
NaN
```

Valid Cases

Precision & Setting

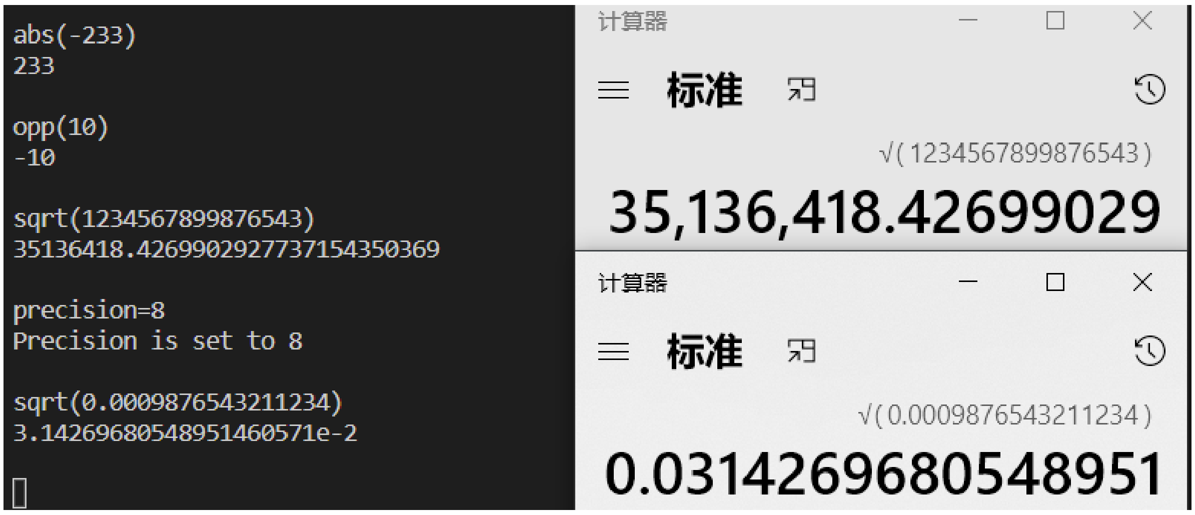


```
9999999999999999990.00000000000001 + 10.1
10000000000000000000.10000000000001

10000000000000000.01 - 0.1
9999999999999999.91

123000000000 * 0.000000000321
39.483

1234567899876543/987654321
1249999.9986093747875

precision = 16
Precision is set to 16

1234567899876543/987654321
1249999.9986093747875173828l5156
```



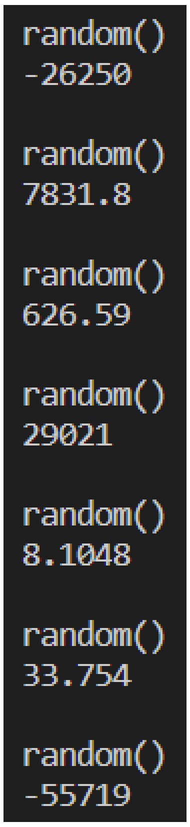
```
123000000000 × 0.000000000321 =
39.483
```

```
1234567899876543 ÷ 987654321 =
1,249,999.998609375
```



```
abs(-233)
233

opp(10)
-10

sqrt(1234567899876543)
35136418.4269902927737154350369

precision=8
Precision is set to 8

sqrt(0.0009876543211234)
3.14269680548951460571e-2
```

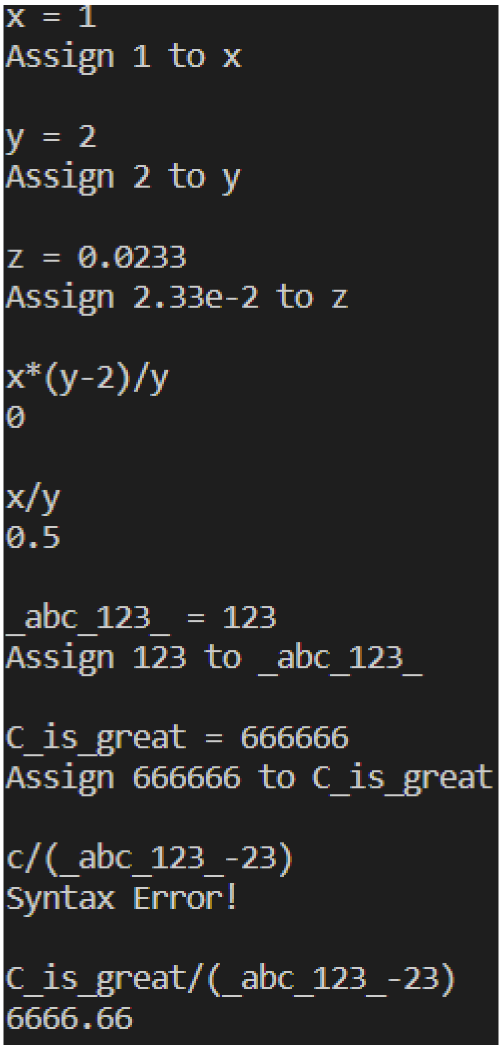
```
√( 1234567899876543 )
35,136,418.42699029

√( 0.0009876543211234 )
0.0314269680548951
```



```
random()
-26250

random()
7831.8

random()
626.59

random()
29021

random()
8.1048

random()
33.754

random()
-55719
```

Variables

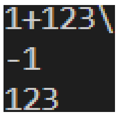
The valid varible names only contain alphabet letters and numbers plus '_', and it can't start with numbers (see invalid cases above).



```
x = 1
Assign 1 to x

y = 2
Assign 2 to y

z = 0.0233
Assign 2.33e-2 to z

x*(y-2)/y
0

x/y
0.5

_abc_123_ = 123
Assign 123 to _abc_123_

C_is_great = 666666
Assign 666666 to C_is_great

c/(_abc_123_-23)
Syntax Error!

C_is_great/(_abc_123_-23)
6666.66
```

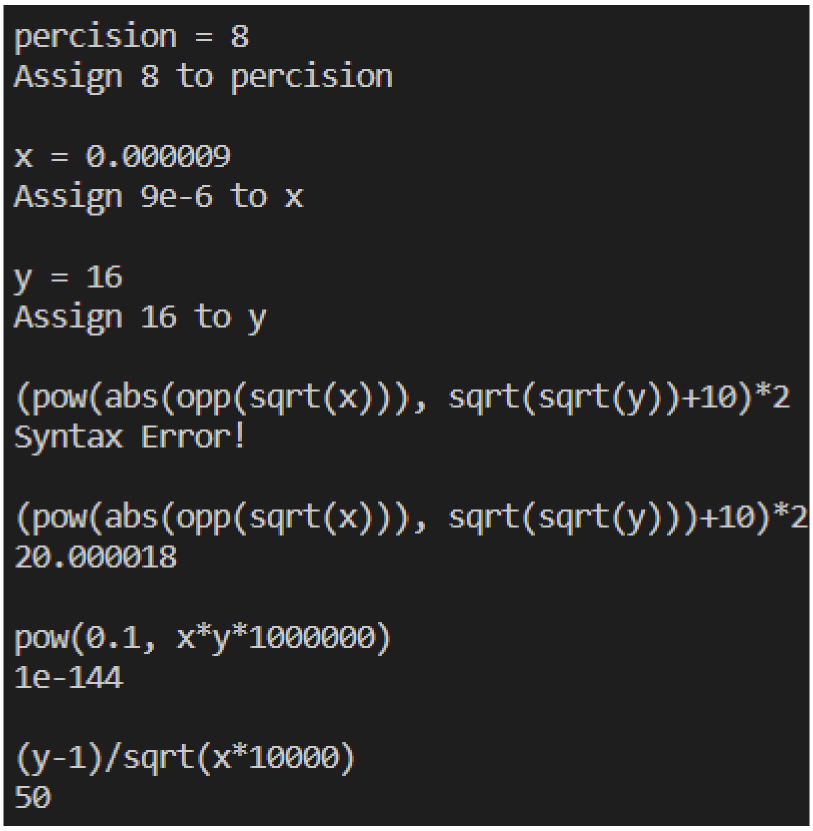
Multiple Line Input

User can put a `'\'` at the end of the line, so the program will wait till the next line is input and append it to the original expression for analysis.



```
1+123\
-1
123
```

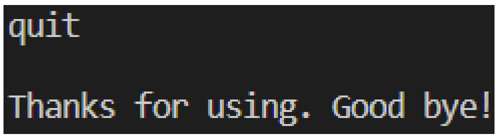
Combanition



```
percision = 8
Assign 8 to percision

x = 0.000009
Assign 9e-6 to x

y = 16
Assign 16 to y

(pow(abs(opp(sqrt(x))), sqrt(sqrt(y))+10)*2
Syntax Error!

(pow(abs(opp(sqrt(x))), sqrt(sqrt(y)))+10)*2
20.000018

pow(0.1, x*y*1000000)
1e-144

(y-1)/sqrt(x*10000)
50
```

Farewell

Thank you for reading this boring report and (perhaps) testing my calculator.



```
quit

Thanks for using. Good bye!
```

Part 3 - Self-review

Due to limited time and experience, there're some parts of this projects I didn't implement them in a simple and efficiency way, leading to poor code readability and program performance. But I've search for materials online and come up some ways to do better the next time.

1. My program run significantly slow when multiplication and division are used. Here's some algorithm that can make them perform better.
   - FFT Polynomial Acceleration (Using knowledge of convolution and complex number)
   - Variable-shifting SRT Division (It seems more easy to implement if I use binary bits instead of decimal digits)
2. In my project, I implement basic arithmetic operations in calculator.cpp, and didn't use overloaded operators, making function implements much more hard to read and fallible, especially when the expressions went complex.

   For example, in my `sqrt()`, the Newton's Method is expressed by:

   ```
   temp = divide(add(multiply(ret, ret), x), multiply(ret, number(2)));
   ```

   But if I've overloaded the operation, it would be simply:

```
temp = (ret*ret+x)/(ret*number(2))
```

(Note: like changing from prefix expression to infix expression)

3. One thing really confuse me is that when try to get the function names from the input string and call the correct ones in the program for I have to enumerate them in my program. But later I found pointers can point to functions in C++, so I wonder if I can use this feature to simplify coding. I'm willing to read more professional and mature source codes to figue it out.
4. Some validity checking is annoying, maybe regex is helpful.

Part 4 - Codes & Comments

Please See My GitHub Repository