C++ Project 2

Name 康耀中(KANG Yaozhong)

SID 12110225

Part 1 - Analysis

1.1 Read Inputs

1.1.1 Input Classification

When the program reads the input, it has to decide which kind of input it is. There are mainly four kinds of inputs:

Input Class	Input form	Process
Calculation	without `=`	calculate the expression
Def Variable	with `=`	store the variable for further usage
Built-In Functions	specific	calculate the expression
Quit	q	quit the program

First, the program judges if the input correspond to Built-in Functions or Quit. If not, the program reads the input as `vector<char>` , and check each element for `=` . If there are more `=` , the program will throw `Input Error: You entered more than one =`

1.1.2 Waiting for Input

The program has to keep on reading input until they meet an error or the user input `q`.

My solution is to use a `while(true)` statement, and only break when getting a Quit command.

1.2 Saving Data and Sequence Decision

1.2.1 Saving Data: `struct nice_number`

In project 1, to implement high precision multiplication, I separated each input to three parts: `positive` `critical` `exponential`. After learning lecture 4, I figure out that I can combine these three into a structure class `nice_number`.

```
struct nice_number
{
    bool valid;
    bool positive;
    string critical;
    long long exponential;
};
```

However, you cannot directly turn the input into a `nice_number` because the input is not a single number. The inputs has to be stored as vectors first.(For how to store vectors, I will explain later this section) Thus, I wrote a generator to create `nice_number` from vector.

```
nice_number niceNumberGenerator(vector<char> input_stream ,map<string,
nice_number> variable_map);
```

This method judge the legality of the input stream and generate it into `nice_number`.

You may noticed that there is a map input in the generator. That's what I'm going to talk about next.

1.2.2 Saving Data: `map`

When the program judges that you are defining variables. It will generate the right hand side of the `=` into `nice_number` using `niceNumberGenerator`, and store it into the map for `<key> = string` `<value> = nice_number`.

Every time you want to generate a `nice_number`, the `niceNumberGenerator` will traverse the map and return if corresponds.

1.2.3 Sequence Decision

To save the data of the input as `vector<vector<char>>`, you have to dicide the sequence of the input.

---

1. Separation

We define a good expression is a mathmatical expression without round bracket. Thus, in a good expression, you have to separate the operation symbols and numbers.

For example `a/b-c*d` will be separated:

Numbers	Operations
a	/
b	-
c	*
d	Null

We can easily see that the `numbers` are always one position bigger than `operations`.

However, the input is read char by char. So when the program read `123e-100` or `412.23e+12`, it will be careful to not separating it into `123e` `-` `100` and `412.23e` `+` `12`.

---

2. Do the `add` and `minus` later

There are several kinds of operations. In math, we can classify them into to kinds.

level 1	level 2

level 1	level 2
+	*
-	/
	^
	%

We do the level 2 operations first, and then do level 1.

Thus, when the program reads an level 2 operation, it will create two vectors to do the operation first. It the next input is also an level 2 input, it will put them into the same vector. When the next input is not an level 2 operation or you've reached the end of the input, the program will do the level 2 operations using the data in the two vectors and put the result into the main vector `vector<char> backet`. This vector is to store the result in level 1.

For example: `a*b/c%d*e+f-g*h-i*j`

First, two vectors will be created:

number	operation
a	*
b	/
c	%
d	*
e	NULL

The program detected that the next operation is `+` , which is a level one operation. Thus it put `+` into the adds and minuses operations vector. And it start to calculate.

basket	operation
NULL	+

We do the operations one by one, and get the result `n1`.

basket	operation
n1	+

Next, it reads that the next operation is `+` , which is level 1 operation. Thus the program will put the number into the basket and so does operation

basket	operation
n1	+

basket	operation
f	-

After that, the program detected that the next operation is `*` , it starts another two vectors.

number	operation
g	*
h	NULL

The next number is `-` , a level 1 operation. Thus it calculate the result `n2`.

basket	operation
n1	+
f	-
n2	-

Finally, it reads `*` and then it find that the `*` is the last operation.

number	operation
i	*
j	NULL

The program get the result `n3` and put it back. This time no operation.

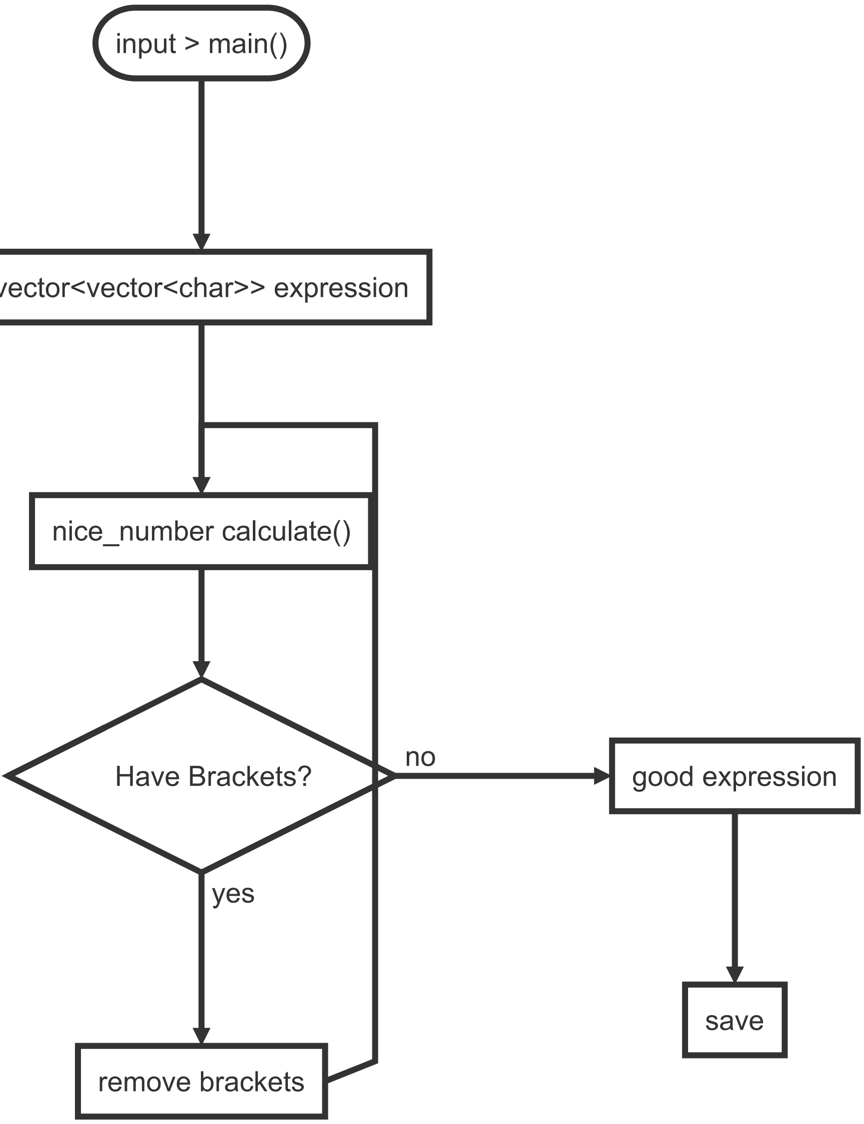
basket	operation
n1	+
f	-
n2	-
n3	NULL

The program at last will do the operation one by one and get the final result.

---

3. Round Bracket

Round brackets emphasize the expressions. When the program detected the appearance of front bracket, it will keep on reading until the number of back brackets cancel out the frount brackets and put them into vector. Then remove the outer brackets and put the vector into `calculate`.



1.3 Operations Implementation

My roommate used `mpfr.h` to solve the high-pricision calculation. However, it was too late because I've made my own rolls. Next time I would probably use `mpfr.h` to do the calculation because making high-pricision calculator by myself is inefficient and can result in devastating huge amount of work.

1.3.1 Add

The classic adding algorithm. There are two `nice_number` as input. One `nice_number` will add `0` in the back if the other's exponential part is negative or itself's exponential part is positive.

For example:

`123e-3 + 42.3253e+3` would become `123e-3` and `42000000e-3`

Thus, the result is `42325423e-3`

1.3.2 Minus

The same as add. The program will add zeros and do the classic algorithm.

1.3.3 Multiplication

Implemented in Project 1.

1.3.4 Division

The critical part of the dividend will automatically add `0` in the end for 16 times. The turn the first element into `nice_number` and do the minus operation with the divisor times one. If the result is positive, it will do the minus operation with the divisor times two and so. Until the result is negative, it will save the previous result and `push_back` the next element of the dividend and do the torturing process again and again.

1.3.5 Mod

Doing minus multiple times until the result is negative. Then return the previous result.

1.3.6 Power

Do the multiplication multiple times.

1.3.7 Square Root

At first I want to make a high-pricision square root calculator by my favourite `nice_number`. I chose to use Newton's Iteration method to do any power root.

$$x_k = \frac{a - x_{k-1}^n}{n x_{k-1}^{n-1}} + x_{k-1}$$

Where a is the input, n is the power, k is the iteration times and x is the result. Which in cpp is:

```
x_hat =
add(dividePrecise(minus(number,power(x_hat,exponential)),multiply(exponential,po
wer(x_hat,minus(exponential,nice_one))),variable_map),x_hat);
y_hat = power(x_hat,exponential);
```

However, time is limited and there are way too many bugs for me to implement the calculation.

Thus, I used `atof()`, and do the calculation. (Sad)

1.4 Functions Implementation

The program support `sqrt` `abs` function.

Part 2 - Code

My Github Repo

`calculate.cpp`

```
nice_number calculate(vector<char> input_stream, map<string, nice_number>
variable_map)
{
```

```
    regex squareRoot_regex("[s][q][r][t][(](.+)[)]");
    regex minus_regex("[-](.+)");
    regex abs_regex("[a][b][s][(](.+)[)]");

    nice_number not_valid_number = {false, true, "", 0};
    vector<vector<char>> expression_vector;
    vector<char> action_vector;
    vector<int> expression_positions = {0};
    vector<char> minus_ion = {'-','1'};

    string input_string;
    //turn to string to do the judges
    for (int i = 0; i < input_stream.size(); i++)
    {
        input_string.push_back(input_stream[i]);
    }
    //square root
    if (regex_match(input_string, squareRoot_regex))
    {
        char temp[40];
        for(int i = 5; i < input_string.length()-1; i++)
        {
            temp[i-5] = input_string[i];
        }
        double temp_double = atof(temp);
        double result = sqrt(temp_double);
        string result_string = std::to_string(result);
        vector<char> temp_char;
        for(int i = 0; i < result_string.length(); i++)
        {
            temp_char.push_back(result_string[i]);
        }

        return niceNumberGenerator(temp_char,variable_map);
    }
    // if the input is minus expression
    if(regex_match(input_string,minus_regex))
    {
        vector<char> temp_char;
        for(int i = 1; i < input_stream.size();i++)
        {
            temp_char.push_back(input_stream[i]);
        }
        nice_number temp_number = calculate(temp_char,variable_map);
        temp_number.positive ? temp_number.positive = false:temp_number.positive
= true;
        return temp_number;
    }
    //abs
    if(regex_match(input_string,abs_regex))
    {
        vector<char> temp_char;
        for(int i = 4; i < input_stream.size()-1;i++)
        {
```

```
            temp_char.push_back(input_stream[i]);
        }
        nice_number temp_number = niceNumberGenerator(temp_char,variable_map);
        temp_number.positive = true;
        return temp_number;
    }
    //check pure number
    bool pure_number = false;
    for(int i = 0; i < input_stream.size(); i++)
    {
        if(operationCheck(input_stream[i]))
        {
            if(i > 0 && input_stream[i-1] == 'e')
            {
                continue;
            }
            break;
        }
        if(i == input_stream.size()-1) pure_number=true;
    }
    if(pure_number)
    {
        return niceNumberGenerator(input_stream,variable_map);
    }
    //read the input
    for (int i = 0; i < input_stream.size(); i++)
    {
        //brackets detected, prepare to iterate
        if (input_stream[i] == '(')
        {
            int round_bracket = 1;
            expression_positions[expression_vector.size()] = i + 1;
            bool minus = false;
            for (i = i + 1; i < input_stream.size(); i++)
            {
                if (input_stream[i] == ')')
                {
                    round_bracket -= 1;
                    assert(round_bracket >= 0);
                }
                if (input_stream[i] == '(')
                {
                    round_bracket += 1;
                }
                if (input_stream[i] == ')' && round_bracket == 0)
                {
                    vector<char> temp_expression;
                    for (int k = expression_positions[expression_vector.size()];
k < i; k++)
                    {
                        temp_expression.push_back(input_stream[k]);
                    }
                    //very important, iteration
                    nice_number temp_nice_number = calculate(temp_expression,
variable_map);
```

```
                    vector<char> temp_expression_2 =
niceNumberToVector(temp_nice_number);
                    expression_vector.push_back(temp_expression_2); // push_back
can avoid index;
                    break;
                }
            }
            assert(round_bracket == 0);
            continue;
        }
        //check that the element is an operator
        if (operationCheck(input_stream[i]))
        {
            if ((input_stream[i] == '-' || input_stream[i] == '+') &&
input_stream[i - 1] == 'e' )
            {
                continue;
            }
            action_vector.push_back(input_stream[i]);
            //check that it is the end of the expression
            vector<char> temp_expression;
            if(input_stream[i-1] != ')')
            {
                for (int j = expression_positions[expression_vector.size()]; j <
i; j++)
                {
                    temp_expression.push_back(input_stream[j]);
                }
                expression_vector.push_back(temp_expression);
            }
            //store the position of an expression, in order to split
            expression_positions.push_back(i + 1);
            continue;
        }
        //store the expression into expression_vector
        if (i == input_stream.size() - 1)
        {
            assert(!operationCheck(input_stream[i]));
            vector<char> temp_expression;
            for (int j = expression_positions[expression_vector.size()]; j <= i;
j++)
            {
                temp_expression.push_back(input_stream[j]);
            }
            expression_vector.push_back(temp_expression);
            break;
        }
    }
    vector<char> add_minus;
    vector<nice_number> basket;
    // do the calculation
    for (int i = 0; i < action_vector.size(); i++)
    {
        // is + or -
        if (action_vector[i] == '+' || action_vector[i] == '-')
```

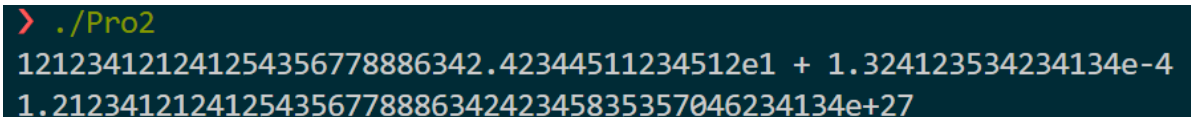
```
        {
            if (i == 0)
            {
                basket.push_back(niceNumberGenerator(expression_vector[i],
variable_map));
            }
            if (i == action_vector.size() - 1)
            {
                basket.push_back(niceNumberGenerator(expression_vector[i + 1],
variable_map));
            }
            if (action_vector[i + 1] == '+' || action_vector[i + 1] == '-')
            {
                basket.push_back(niceNumberGenerator(expression_vector[i +
1],variable_map));
            }
            add_minus.push_back(action_vector[i]);
            continue;
        }
        // is not + or -
        if (i == 0 || (action_vector[i - 1] == '+' || action_vector[i - 1] == '-
'))
        {
            vector<nice_number> temp_number;
            vector<char> temp_action;

 temp_number.push_back(niceNumberGenerator(expression_vector[i],variable_map));
            if (!niceNumberGenerator(expression_vector[i],variable_map).valid)
            {
                return not_valid_number;
            }
            temp_number.push_back(niceNumberGenerator(expression_vector[i +
1],variable_map));
            if (!niceNumberGenerator(expression_vector[i +
1],variable_map).valid)
            {
                return not_valid_number;
            }
            temp_action.push_back(action_vector[i]);

            //continue to store the operations that is not + and -
            for (i = i + 1; i < action_vector.size(); i++)
            {
                if (!(action_vector[i] == '+' || action_vector[i] == '-'))
                {

 temp_number.push_back(niceNumberGenerator(expression_vector[i+1],variable_map))
;
                    if
(!niceNumberGenerator(expression_vector[i+1],variable_map).valid)
                    {
                        return not_valid_number;
                    }
                    temp_action.push_back(action_vector[i]);
```

```
                    continue;
                }
                i--;
                break;
            }
            //do the calculation of the specific block
            for (int j = 0; j < temp_action.size(); j++)
            {
                switch (temp_action[j])
                {
                case '*':
                    temp_number[j + 1] = multiply(temp_number[j], temp_number[j
+ 1]);
                    break;
                case '/':
                    temp_number[j + 1] = dividePrecise(temp_number[j],
temp_number[j + 1],variable_map);
                    break;
                case '^':
                    temp_number[j + 1] = power(temp_number[j], temp_number[j +
1]);
                    break;
                case '%':
                    temp_number[j + 1] = mod(temp_number[j], temp_number[j +
1]);
                    break;
                }
            }
            basket.push_back(temp_number[temp_number.size() - 1]);
        }
    }
    //there's no pluses or minuses
    if (basket.size() == 0)
    {
        return basket[0];
    }
    //do the pluses and minuses in the basket
    for (int i = 0; i < add_minus.size(); i++)
    {
        switch (add_minus[i])
        {
        case '-':
            basket[i + 1] = minus(basket[i], basket[i + 1]);
            break;
        case '+':
            basket[i + 1] = add(basket[i], basket[i + 1]);
            break;
        }
    }
    //result
    return basket[basket.size() - 1];
}
```

Part 3 - Result & Verification

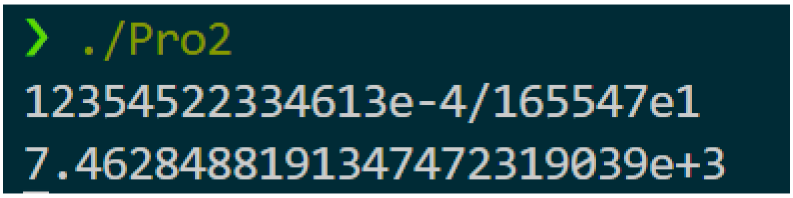
Test case #1

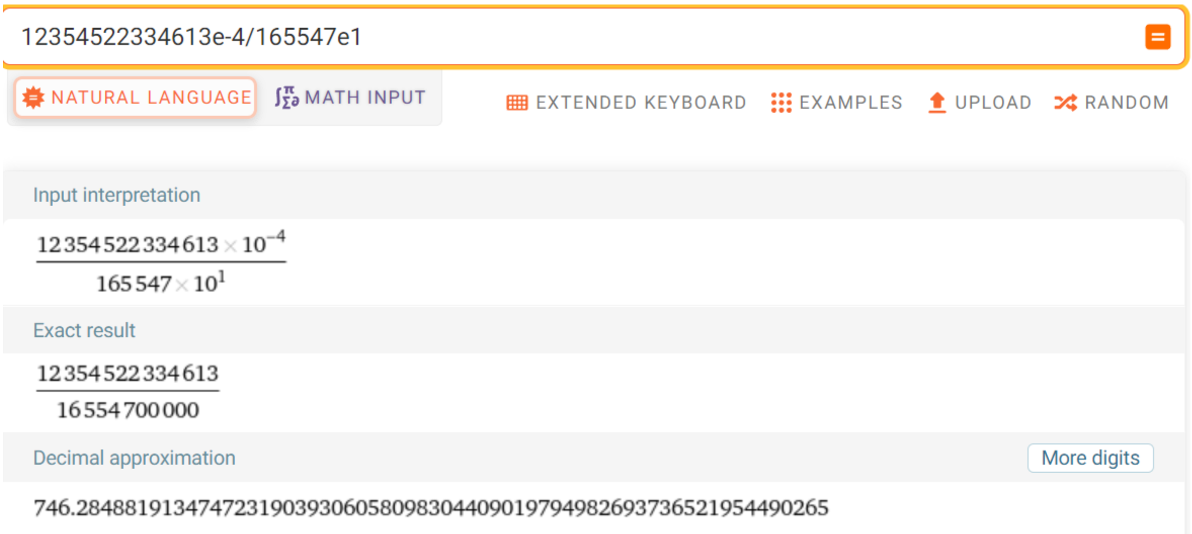
```
Input: 121234121241254356778886342.42344511234512e1 + 1.324123534234134e-4
Output : 1.2123412124125435677888634242345835357046234134e-27
```



Test case #2

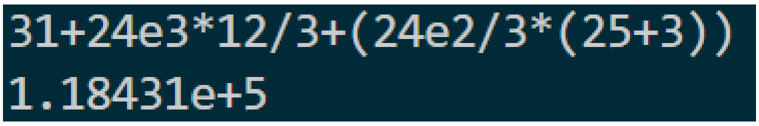
```
Input: 12354522334613e-4/165547e1
Output: 7.46284881913474723190 39e+3
```

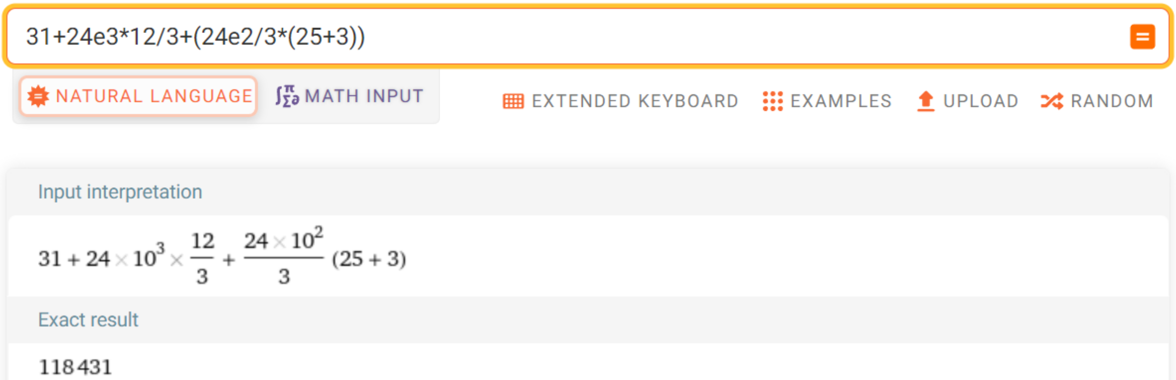




Test case #3

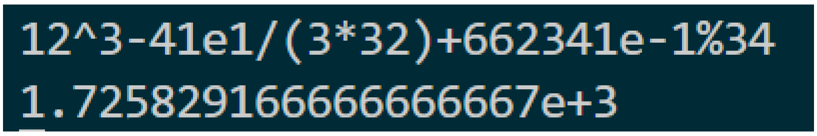
```
Input: 31+24e3*12/3+(24e2/3*(25+3))
Output: 1.18431e+5
```

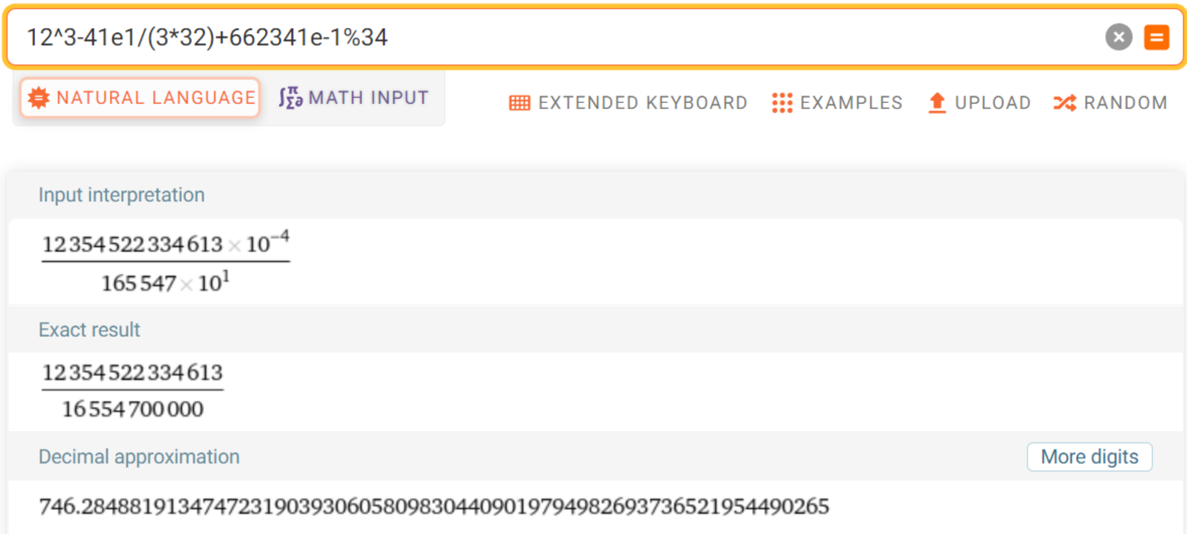




Test case #4

```
Input: 12^3-41e1/(3*32)+662341e-1%34
Output: 1.725829166666666667e+3
```





Test case #5

```
Input:
    x_hat = 5
    y_hat = 6e-1
    loss = 3e-10
    (x_hat * y_hat^2)/23 + 142*loss
Output: 3.91304389426086956e-1
```

```
x_hat = 5
y_hat = 6e-1
loss = 3e-10
(x_hat * y_hat^2)/23 + 142*loss
3.9130438942608695 6e-1
```

((5*6e-1)^2)/23 + 142*3e-10

NATURAL LANGUAGE MATH INPUT EXTENDED KEYBOARD EXAMPLES UPLOAD RANDOM

Input interpretation

$$\frac{1}{23}(5\times 6\times 10^{-1})^2 + 142\times 3\times 10^{-10}$$

Exact result

$$\frac{45000004899}{115000000000}$$

Decimal approximation More digits

0.391304390426086956521739130434782608695652173913043478260869565 2

Test case #6

```
Input:
sqrt(512e1)
abs(-100e10)
Output:
71.554175
1.0e+12
```

```
sqrt(512e1)
71.554175
```

```
abs(-100e10)
1.0e+12
```

Part 4 - Difficaulties & Solutions

The first difficulty is to solve scientific calculation. My solution is to construct a `nice_number` to contain the data. The second difficulty is to know the sequence of the calculation. My solution is to use iteration. The third difficulty is to write your own calculations using `nice_number`. My solution is to get tortured one by one, however bugs still exist. The fourth difficulty is to read variables. My solution is to use a map to contain the variables.