Languages like Java have Big Decimal libraries supporting basic arithmetic operations like additions, subtractions, multiplications and divisions. However, scientific problems usually requires mathematical functions like sin, cos, etc. In this problem, you're to write a Big Decimal Calculator.

There are 15 commands:

- Function with two arguments: `add`, `sub`, `mul`, `div`, `pow`, `atan2`
- Functions with only one argument: `exp`, `ln`, `sqrt`, `asin`, `acos`, `atan`, `sin`, `cos`, `tan`

For trigonometric functions, angles are always in radians.

Input

There will be at most 100 lines. Each line begins with the function name, followed by arguments, then the precision p ($1 \le p \le 50$). Each argument is formatted as one or more digits, followed by a dot '`.`', then by one or more digits. The integer part cannot be omitted, but the last two parts can be omitted together. There can be an optional negative sign before an argument. Each input number contains at most 20 digits. In function '`pow`', '`exp`', '`ln`' and '`sqrt`', all the arguments are strictly positive; in function '`asin`' and '`acos`', the integer part of the arguments are always zero.

Output

For each line, print the answer, rounded to p decimal places (Don't use scientific notation!). It is guaranteed that the result will be a finite number, and its integer part will not exceed 10 digits.

Note:

You may notice that this problem is not language-neutral. I mean, some programming languages have advantages over some others. This is intentional: real-world software development is like this. Choosing programming languages, libraries and the overall architectures can be vital.

There are quite a lot of literatures on this topic (for example, Fast multiprecision evaluation of series of rational numbers by Bruno Haible , Thomas Papanikolaou), but that's overkill for this problem. The time limit for this problem is rather large, and the test cases are quite gentle: the goal of this problem is to write a working program, not a perfect one, so try to write a concise code, which is usually faster to write and easier to debug.

For a much more practical literature, look at this: `http://www.tc.umn.edu/.ringx004/sidebar.html`. That small article presents how to translate arguments to make the series expansion converge faster. Though you will not find the whole solution, you'll have some nice ideas.

Sample Input

```
add 1.357 4.6279 10
sub 1.357 4.6279 10
mul 1.357 4.6279 10
div 1 103 30
pow 12.2 12.15 20
atan2 2.45 1.77 30
exp 10.98 50
ln 21.065 50
sqrt 2 40
asin 0.81 30
acos 0.47 35
atan 0.618 40
sin 3.1415 25
cos 2.0113 50
tan 1.78987 30
```

Sample Output

```
5.9849000000
-3.2709000000
6.2800603000
0.009708737864077669902912621359
15822384813181.61872382001683484036
0.945162277467215967394902628052
58688.55427461755601946329091442988532551237342326651423
3.04761289543097985660178308429069456872534888053139
1.4142135623730950488016887242096980785697
0.944152115154155950477697775653
1.08150554878078090500864808815790029
0.5535497640327316544572642343482671646331
0.0000926535896606714405662
-0.42639511018918176703311006536787403871085921161347
-4.491415179046604916096895094786
```