Problem A
Factorial Frequencies

In an attempt to bolster her sagging palm-reading business, Madam Phoenix has decided to offer several numerological treats to her customers. She has been able to convince them that the frequency of occurrence of the digits in the decimal representation of factorials bear witness to their futures. Unlike palm-reading, however, she can't just conjure up these frequencies, so she has employed you to determine these values.

Recall that the definition of n! (that is, n factorial) is just 1×2×3×...×n. As she expects to use the day of the week, the day of the month, or the day of the year as the value of n, you must be able to determine the number of occurrences of each decimal digit in numbers as large as 366 factorial (366!), which has 781 digits.

The input data for the program is simply a list of integers for which the digit counts are desired. All of these input values will be less than or equal to 366 and greater than 0, except for the last integer, which will be zero. Don't bother to process this zero value; just stop your program at that point. The output format isn't too critical, but you should make your program produce results that look similar to those shown below.

Madam Phoenix will be forever (or longer) in your debt; she might even give you a trip if you do your job well!

Sample Input

```
3
8
100
0
```

Expected Output

```
3! --
   (0)    0    (1)    0    (2)    0    (3)    0    (4)    0
   (5)    0    (6)    1    (7)    0    (8)    0    (9)    0
8! --
   (0)    2    (1)    0    (2)    1    (3)    1    (4)    1
   (5)    0    (6)    0    (7)    0    (8)    0    (9)    0
100! --
   (0)   30    (1)   15    (2)   19    (3)   10    (4)   10
   (5)   14    (6)   19    (7)    7    (8)   14    (9)   20
```

Problem B
Identifying Legal Pascal Real Constants

Pascal requires that real constants have either a decimal point, or an exponent (starting with the letter e or E, and officially called a scale factor), or both, in addition to the usual collection of decimal digits. If a decimal point is included it must have at least one decimal digit on each side of it. As expected, a sign (+ or -) may precede the entire number, or the exponent, or both. Exponents may not include fractional digits. Blanks may precede or follow the real constant, but they may not be embedded within it. Note that the Pascal syntax rules for real constants make no assumptions about the range of real values, and neither does this problem.

Your task in this problem is to identify legal Pascal real constants. Each line of the input data contains a candidate which you are to classify. For each line of the input, display your finding as illustrated in the example shown below. The input terminates with a line that contains only an asterisk in column one.

Sample Input

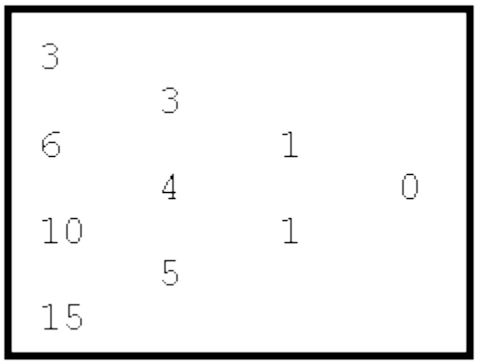
```
1.2
   1.
  1.0e-55
  e-12
    6.5E
     1e-12
  +4.1234567890E-99999
   7.6e+12.5
99
*
```

Expected Output

```
1.2 is legal.
1. is illegal.
1.0e-55 is legal.
e-12 is illegal.
6.5E is illegal.
1e-12 is legal.
+4.1234567890E-99999 is legal.
7.6e+12.5 is illegal.
99 is illegal.
```

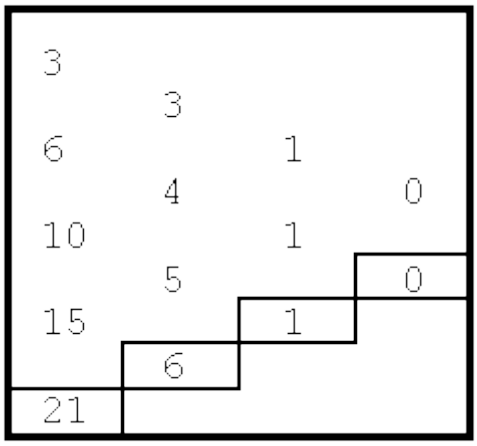
Problem C
Extrapolation Using a Difference Table

A very old technique for extrapolating a sequence of values is based on the use of a difference table. The difference table used in the extrapolation of a sequence of 4 values, say 3, 6, 10, and 15, might be displayed as follows:



The original sequence of values appears in the first column of the table. Each entry in the second column of the table is formed by computing the difference between the adjacent entries in the first column. These values (in the second column) are called first differences. Each entry in the third column is similarly the difference between the adjacent entries in the second column; the third column entries are naturally called second differences. Computation of the last column in this example should now be obvious (but beware that this value is not always zero). Note that the last column will always contain only a single value. If we begin with a sequence of n values, the completed difference table will have n columns, with the single value in column n representing the single n-1st difference.

To extrapolate using a difference table we assume the n-1st differences are constant (since we have no additional information to refute that assumption). Given that assumption, we can compute the next entry in the n-2nd difference column, the n-3rd difference column, and so forth until we compute the next entry in the first column which, of course, is the next value in the sequence. The table below shows the four additional entries (in boxes) added to the table to compute the next entry in the example sequence, which in this case is 21. We could obviously continue this extrapolation process as far as desired by adding additional entries to the columns using the assumption that the n-1st differences are constant.



The input for this problem will be a set of extrapolation requests. For each request the input will contain first an integer n, which specifies the number of values in the sequence to be extended. When n is zero your program should terminate. If n is non-zero (it will never be larger than 10), it will be followed by n integers representing the given elements in the sequence. The last item in the input for each extrapolation request is k, the number of extrapolation operations to perform; it will always be at least 1. In effect, you are to add k entries to each column of the difference table, finally reporting the last value of the sequence computed by such extrapolation. More precisely, assume the first n entries (the given values) in the sequence are numbered 1 through n. Your program is to determine the n+kth value in the sequence by extrapolation of the original sequence k times. (Hint: no upper limit is given for k, and you might not be able to acquire enough memory to construct a complete difference table.)

Sample Input

```
4  3  6  10  15  1
4  3  6  10  15  2
3  2  4  6  20
6  3  9  12  5  18  -4  10
0
```

Expected Output

```
Term 5 of the sequence is 21
Term 6 of the sequence is 28
Term 23 of the sequence is 46
Term 16 of the sequence is 841
```

Problem D
Evaluating Simple C Expressions

The task in this problem is to evaluate a sequence of simple C expressions, but you need not know C to solve the problem! Each of the expressions will appear on a line by itself and will contain no more than 80 characters. The expressions to be evaluated will contain only simple integer variables and a limited set of operators; there will be no constants in the expressions. There are 26 variables which may appear in our simple expressions, namely those with the names a through z (lower-case letters only). At the beginning of evaluation of each expression, these 26 variables will have the integer values 1 through 26, respectively (that is, a = 1, b = 2, ...). Each variable will appear at most once in an expression and many variables may not be used at all.

The operators that may appear in expressions include the binary (two-operand) + and -, with the usual interpretation. Thus the expression `a + c - d + b` has the value 2 (computed as `1 + 3 - 4 + 2`). The only other operators that may appear in expressions are `++` and `--`. These are unary (one-operand) operators, and may appear before or after any variable. When the `++` operator appears before a variable, that variable's value is incremented (by one) before the variable's value is used in determining the value of the entire expression. Thus the value of the expression `++ c - b` is 2, with c being incremented to 4 prior to evaluation the entire expression. When the `++` operator appears after a variable, that variable is incremented (again, by one) after its value is used to determine the value of the entire expression. Thus the value of the expression `c ++ - b` is 1, but c is incremented after the complete expression is evaluated; its value will still be 4. The `--` operator can also be used before or after a variable to decrement (by one) the variable; its placement before or after the variable has the same significance as for the `++` operator. Thus the value of the expression `--` c + b -- has the value 4, with variables b and c having the values 1 and 2 following the evaluation of the expression.

Here's another, more algorithmic, approach to explaining the `++` and `--` operators. We'll consider only the `++` operator, for brevity:

1. Identify each variable that has a ++ operator before it. Write a simple assignment statement that increments the value of each such variable, and remove the ++ from before that variable in the expression.
2. In a similar manner, identify each variable that has a ++ operator after it. Write a simple assignment statement that increments the value of each of these, and remove the ++ operator from after that variable in the expression.

3. Now the expression has no ++ operators before or after any variables. Write the statement that evaluates the remaining expression after those statements written in step 1, and before those written in step 2.
4. Execute the statements generated in step 1, then those generated in step 3, and finally the one generated in step 2, in that order.

Using this approach, evaluating the expression `++ a + b ++` is equivalent to computing `a = a + 1` (from step 1 of the algorithm) `expression = a + b` (from step 3) `b = b + 1` (from step 2) where expression would receive the value of the complete expression.

Your program is to read expressions, one per line, until a totally blank (or empty) line is read. Display each expression exactly as it was read, then display the value of the entire expression, and on separate lines, the value of each variable after the expression was evaluated. Do not display the value of variables that were not used in the expression. The samples shown below illustrate the desired output format.

Blanks are to be ignored in evaluating expressions, and you are assured that ambiguous expressions like `a+++b` (ambiguous because it could be treated as `a++ + b` or `a + ++b`) will not appear in the input. Likewise, `++`or `--` operators will never appear both before and after a single variable. Thus expressions like `++a++` will not be in the input data.

Sample Input

```
a + b
b - z
a+b--+c++
c+f--+--a
   f-- + c-- + d-++e
```

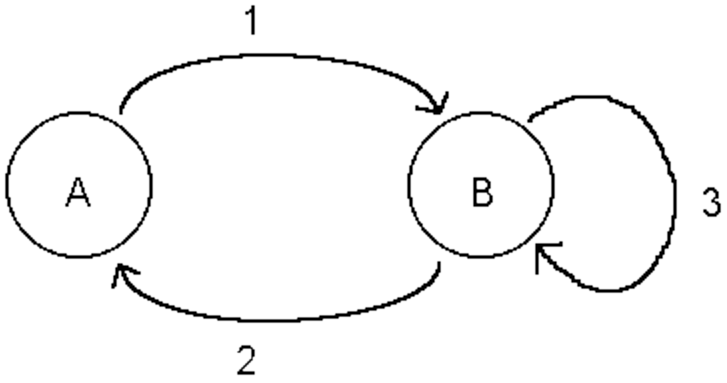
Expected Output

```
Expression: a + b
    value = 3
    a = 1
    b = 2
Expression: b - z
    value = -24
    b = 2
    z = 26
Expression: a+b--+c++
    value = 6
    a = 1
    b = 1
    c = 4
Expression: c+f--+--a
    value = 9
```

```
    a = 0
    c = 3
    f = 5
Expression:    f-- + c-- + d-++e
    value = 7
    c = 2
    d = 4
    e = 6
    f = 5
```

Problem E
The Finite State Text-Processing Machine

A finite state machine (FSM) is essentially a directed graph. Each node in the graph is called a state; at any point during execution of the FSM, one of the states is said to be the current state. Each directed edge between two states is called a transition. When the conditions are right, one of the transitions from the current state is said to occur, and the current state changes to a new state as determined by the transition. Consider the following very simple example.



There are two states in this FSM, labeled A and B, and three transitions, labeled 1, 2, and 3. If the current state is A, and the conditions for transition 1 are met, then the current state becomes B. When the current state is B, and the conditions for transition 2 are met, the current states becomes A again. If the current state is B and the conditions for transition 3 are met, the current state remains B.

In this problem the input will be descriptions of several FSMs. Each transition in these FSMs has an associated set of characters called the input set, and a string called the output string. A transition can occur when the current input data character is in the transition's input set. When the transition occurs, the output string is printed.

Input Description

The input consists of a sequence of pairs {FSM description, input for the FSM}. A FSM is described by the following sequence of items separated by whitespace (blanks, tabs, and end of line characters):

- An integer specifying the number of states in the FSM. for each of these states there will be the following items, in order:

- A one to eight character name for the state. State names are case significant, and unique
- An integer specifying the number of transitions that leave the current state. For each of these transitions there will be the following data items, in order:
  - The set of input characters that enable the transition (see below). the name of the new current state when this transition occurs
  - The output string (see below).

The input set and the output string are given as sequences of printable characters with no imbedded whitespace. Several special constructions may appear in these, however. When \b appears it is to be interpreted as a blank. Treat \n as an end of line, and \\ as a single backslash. The construction \0 (that is, backslash followed by zero) will appear only as an output string, and means to print nothing when the transition occurs. The construction \c appearing as an input set matches anything. That is, if none of the other transitions are enabled and a transition has \c specified as its input set, then it is enabled. When \c appears in an output string, it means to print the current input character. this could appear several times in the same output string.

After the last item in the description of and FSM has been read begin "executing" the FSM using the characters that start on the first complete line following the description. the beginning state will always be called START. The final state will always e called END, but will not appear as a state in the description of a FSM. All input is guaranteed to be correct.

Output Description and Examples

For each input your program should display the FSM number (1, 2, ...) and, beginning on the next line, the output that results from those transitions that occur. The examples below illustrate this.

The first example FSM replaces each vowel in a single line of input with an asterisk. The second example will delete each vowel that follows a lower or upper case X, again processing only a single line of input. The final example changes the case of each odd-numbered vowel in the input; processing stops when an exclamation point is encountered, and the remainder of the input line is ignored.

Sample Input

```
1
START 3
      AEIOUaeiou  START *
      \n          END   \n
      \c          START \c
This is input for example one.
2
START 3
      \c    START \c
      Xx    SKIP  \c
      \n    END   \n
SKIP  4
      AEIOU START \0
      aeiou START \0
      Xx    SKIP  \c
      \n    END   \n
XaXxe     Test   XIo  iXixO
3
START 12
   \c START \c    !     FINIS \0
   A  TWO   a     E     TWO   e
   I  TWO   i     O     TWO   o
   U  TWO   u     a     TWO   A
   e  TWO   E     i     TWO   I
   o  TWO   O     u     TWO   U
TWO   4
  \c  TWO   \c    AEIOU START \c
   aeiou  START \c  !  FINIS  \0
FINIS 2
   \c  FINIS \0   \n    END   \n
This is some data for FSM number 3.
!    IGNORED
0
```

Expected Output

```
Finite State Machine 1:
Th*s *s *np*t f*r *x*mpl* *n*.
Finite State Machine 2:
XXx     Test   Xo  iXx
Finite State Machine 3:
```

Problem F
PostScript Emulation

PostScript ® is widely used as a page description language for laser printers. The basic unit of measurement in PostScript is the point, and there are assumed to be exactly 72 points per inch. The default coordinate system used at the beginning of a PostScript program is the familiar Cartesian system, with the origin (the point with x and y coordinates both equal to zero) at the lower left corner of the page.

In this problem you are required to recognize a small part of the PostScript language. Specifically, the commands listed below must be recognized. All commands will be given in lower-case letters. When the word number appears in the command description, a floating point value, with an optional sign and decimal point will be present in the input data. When two numbers appear in a command, the first refers to the x coordinate, and the second to the y coordinate. For simplicity, each command will appear on a line by itself, and a single blank will separate each component of the command. The end of line character will immediately follow each command, and a line with a single asterisk in column one will terminate the input.

`number rotate`

> **`number`** represents the measure of an angle. This command rotates the coordinate system that may degrees counter clockwise about the current origin. his does not affect any lines that have already been drawn on the page. Example: **`90 rotate`** will rotate the axes 90 degrees counterclockwise, so the positive y axis now points to the left, and the positive x axis points up.

`number number translate`

> Moves the origin of the coordinate system to the position specified. The values are interpreted relative to the current origin. Example: **`612 792 translate`** would move the origin of the coordinate system to the upper right corner of the page. Now only points with negative x and y components will correspond to points on the physical page (assuming an 8.5" by 11" page in the portrait orientation).

`number number scale`

> This command applies the scaling factors given to the x and y coordinates. In effect, the actual x and y sizes of objects are multiplied by the respective scale factors. Example: **`3 2 scale`** would cause a line drawn from (0,0) to (72,72) to appear as if it was drawn from the lower left corner of the page to a point three inches to the right of the left edge of the paper and two inches up from the bottom edge of the paper, assuming the original coordinate system was in effect just before the command.

`number number moveto`

The current point is moved to the coordinates specified. Example: **`72 144 moveto`** would move the current point to one inch from the left edge of the paper, and two inches up from the bottom edge, assuming the original coordinate system was in effect.

`number number rmoveto`

This command is like **`moveto`**, except the numbers specified give the coordinates of the new current point relative to the current position. Example: **`144 -36 rmoveto`** would move the current point set by the previous example two inches further to the right and one-half inch lower on the page. Thus the coordinates of the current point become (216,108). Notice that numbers can be negative!

`number number lineto`

Draws a line from the current position to the position specified by the numbers. The current position becomes the position specified in the command. Example: (216,144), would draw a line from the current position to the point (216,144), and leave the current point there. If we assume the current position from the previous example, this would be a line from (216,108) to (216,144), or a half-inch vertical line.

`number number rlineto`

This is similar to **`lineto`**, but the coordinates given in the command are relative o the current position. Again, the end of the line that is drawn becomes the new current position. Example: **`0 144 rlineto`** will draw a two inch horizontal line from the current position two inches to the right. Using the current position left in the previous example, this draws a line from (216,144) to (216,288), and leaves the current point at (216,288).

Your task is to read one of these small PostScript programs and to display a program that will produce the same effect *without* using the **`rotate`**, **`translate`**, or **`scale`** commands. That is, each **`moveto`**, **`rmoveto`**,**`lineto`**, and **`rlineto`** command in the original (input) program should appear in your output (most likely with modified numbers), but the **`rotate`**, **`translate`**, and **`scale`** commands in the input must not appear in the output. Assume the original coordinate system is in effect at the beginning of execution. The numbers used with the commands in the program you produce must be accurate to at least two fractional digits.

Sample Input

```
300 300 moveto
0 72 rlineto
2 1 scale
36 0 rlineto
1 -4 scale
0 18 rlineto
1 -0.25 scale
0.5 1 scale
```

```
300 300 translate
90 rotate
0 0 moveto
0 72 rlineto
2 1 scale
36 0 rlineto
1 -4 scale
0 18 rlineto
*
```

Expected Output

```
300 300 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
300 300 moveto
-72 0 rlineto
0 72 rlineto
72 0 rlineto
```