Problem G: Function Overloading

Source: `overload.{c,cpp,java}`

Many programming languages (including C, C++ and Java) allow the programmer to define overloaded functions, i.e., several functions that have the same name but different parameter lists. However, in some languages (such as Ada) it is possible to overload by return type as well. That is, it is possible for two functions to have the same name and parameter list, but different return types. For example:

```
char f(float x, int y)
char f(float x, float y)
float f(float x, float y)
float g(float x, int y)
float g(int x, float y)
```

Given these function declarations, suppose the program contains the following variable declarations and function call:

```
float a = 1.0, b = 2.0;
int c = 3;
float d = g(c, f(a, b));
```

The first two declarations of f would not work here, but the third one does: **`f(<float>, <float>)`** returns **`<float>`**, which reduces the function call to **`g(<int>, <float>)`**, and the second declaration of **`g`** will return the **`<float>`** that can be assigned to variable **`d`**. Since we used the 3rd version of **`f`** and the 2nd version of **`g`**, we say that the given function call is resolved by

```
d = g2(c, f3(a, b)).
```

Using the same declarations, the function call **`c = g(a, f(a, c))`** cannot be resolved. As a final example, consider the function declarations

```
float x(float w)
int x(float w)
char y(float v)
char y(int v)
```

and the variable declaration and function call

```
float a = 1.0;
char c = y(x(a));
```

In this case, we see that the resolution of the given function call is ambiguous.

Input

The input will consist of a list of function declarations (one per line). Each function declaration in the input will have the form

```
name num_params param(1) param(2) ... param(num_params) rettype
```

where **name** is the function name, **param(i)** is the data type of the i-th parameter, and rettype is the data type of the return value (this problem does not deal with 'void' functions); **num_params** is at least 1 and at most 9. Note that the parameters do not have names, it is only their data types that matter. Function names are single lower case letters, while data types are single upper case letters. Different functions with the same name will appear consecutively in the input, and there are at most 500 different functions for each function name. No two function declarations will be exactly the same. Each function declaration carries with it, implicitly, a 'serial number'. The serial numbers start out at 1 and increase until a new function name is encountered; then they start out at 1 again.

The list of function declarations in the input will be concluded by a line containing only a pound sign ('#'). Thereafter will come a list of function calls (one per line).

The structure of each function call can be defined by the grammar:

```
<function_call> := <data_type> = <right_hand_side>
<right_hand_side> := <fname> <num_params> <param_list>
<param_list> := <param> | <param_list> <param>
<param> := <data_type> | <right_hand_side>
<data_type> := <upper_case_letter>
<num_params> := '1' | '2' | ... | '9'
<fname> := <lower_case_letter>
```

Here the symbols **:=** and **|** are used to define the grammar, they will not show up in the actual function call. Furthermore, in any function call the specified num_params will match exactly the number of parameters given in the parameter list. Each function call will contain no more than 500 function names (not necessarily distinct). The end of the list of function calls will be marked by a line containing only a pound sign. In each function declaration, as well as in each function call in the input, adjacent tokens (lower or upper case letters, digits, equals sign) will be separated by exactly one blank space.

Output

For each function call in the input, there will be one line of output. If the function call can be resolved uniquely, the output will be the same as the input function call, but each function name will have a serial number appended to it, to indicate which version of the function was used there. Otherwise, the output will be either 'impossible' or 'ambiguous', as explained above. If it is ambiguous, also output the number of ways the function call can be resolved, or print '> 1000' if there are more than 1000 ways.

Sample Input

```
f 2 A B C
f 2 A A C
f 2 A A A
g 2 A B A
g 2 B A A
x 1 A A
x 1 A B
y 1 A C
y 1 B C
h 2 A B E
h 2 C D F
k 2 E F A
#
A = g 2 B f 2 A A
B = g 2 A f 2 A B
C = y 1 x 1 A
A = k 2 h 2 A B h 2 C D
#
```

Sample Output

```
A = g2 2 B f3 2 A A
impossible
ambiguous 2
A = k1 2 h1 2 A B h2 2 C D
```