7739 Lambda Calculus

The **lambda calculus** is a simple language that is often used to study the theory of programming languages. This language only consists of variable references, functions that take a single argument, and applications of functions.

The lambda calculus consists of a language of **lambda terms**. The syntax of the lambda calculus defines that all valid lambda terms can be built by applying the following three rules:

1. a **variable** x, which is a nonempty string other than lambda, is itself a lambda term;
2. if t is a valid lambda term, and x is a variable, then "($lambda$ (x) t)" is a lambda term (called a **lambda abstraction**), here x has at least one occurrence in t;
3. if t and s are lambda terms, then "(t s)" is a lambda term (called an **application**).

Nothing else is a lambda term. Thus a lambda term is valid if and only if it can be obtained by repeated application of these three rules. Note that the parentheses in the second and the third rule can **NOT** be omitted.

A **lambda abstraction** "($lambda$ (x) t)" is a definition of an anonymous function that is capable of taking a single input x and substituting it into the term t. The abstraction **binds** the variable x in term t, so we call it a **lambda binding** of variable x.

An **application** "(t s)" represents the application of a function t to an input s, that is, it represents the act of calling function t on input s to produce $t(s)$.

To see how this works, consider the lambda calculus extended with arithmetic operators. In that language, a lambda abstraction "($lambda$ (x) x + 5)", in which x is bound, defines a function $f(x) = x + 5$. And an application "(($lambda$ (x) x + 5) 3)" applies function $f(x) = x + 5$ to input 3 and produces $f(3) = 3 + 5 = 8$.

We say that a variable **occurs free** in an lambda term t if it has some occurrence in t that is not inside some lambda binding of the same variable.

For example,

- x occurs free in x;
- x does not occur free in y;
- x does not occur free in ($lambda$ (x) (x y));
- x occurs free in ($lambda$ (y) (x y));
- x occurs free in (($lambda$ (x) x) (x y)), which is an application that produces (x y) in which x occurs free;
- x occurs free in ($lambda$ (y) ($lambda$ (z) (x (y z)))).

Now you are given a lambda term t, you need to find all distinct variables that occur free in t.

Input

The first line of the input contains a positive integer T denoting the number of test cases. Then T lines follow, each line contains a nonempty string denoting a lambda term.

Each variable in the the lambda terms contains only latin letters and the hyphen, and is no longer than 20 characters.

It is guaranteed that all lambda terms in the input are valid, and the total length of all lambda terms will not exceed 10^7.

There might be some extra spaces in the input as long as they do not cause any misunderstanding.

Output

For each test case, output first the case number and then all distinct variables that occur free in the lambda term in a single line. These variables should be printed in lexicographic order, and there should be a single space between each two adjacent variables.

There should be a space after the colon in each test case.

Hint The free variables of a term are those variables not bound by a lambda abstraction. The set of free variables of an expression is defined inductively:

1. The free variables of x are just x;
2. The set of free variables of $(lambda\ (x)\ t)$ is the set of free variables of t, but with x removed;
3. The set of free variables of $(t\ s)$ is the union

We can define it with the context-free grammar:

$$\begin{aligned} LcExp &::= Variable \\ &::= (lambda\ (Variable)\ LcExp) \\ &::= (LcExp\ LcExp) \end{aligned}$$

where a variable is any string other than '*lambda*'.

Sample Input

```
6
x
y
(lambda (x) (x y))
(lambda (y) (x y))
( (lambda (x) x) (x y) )
( lambda (y) ( lambda (z) (x (y z)) ) )
```

Sample Output

```
Case #1: x
Case #2: y
Case #3: y
Case #4: x
Case #5: x y
Case #6: x
```