If we know that one is of type int and id is of type forall[a] a -> a, we can infer that id(one) is of type int.

A function fun x y -> x has a generic type of forall[a b] (a, b) -> a.

Let's write a program to help us infer the type of expression in a given environment!

First, we define the syntax of expression:

```plaintext
ident : [A-Za-z][A-Za-z0-9]*     // variable names
expr : "let " ident" = " expr in " expr // variable definition
     | "fun " argList" -> " expr       // function definition
     | simpleExpr
argList : { 0 or more ident seperated by ' ' }
simpleExpr : (' expr')
            | ident
            | simpleExpr('/' paramList')      // function calling
            | simpleTy
paramList : { 0 or more expr seperated "," }
```

Then, we define the syntax of type:

```plaintext
ty : '(' -> " ty     // function without arguments
    | '(' tyList')" -> " ty     // uncurry function
    | "forall"[" argList"]" ty  // generic type
    | simpleTy" -> " ty     // curry function
    | simpleTy
tyList : { 1 or more ty seperated by " , " }
simpleTy : (' ty')
          | ident
          | simpleTy'/" tyList"     // such as list[int]
```

Hint in parsing:

- Spacing is strict.
- Pay attention to avoid dead loop.

Type of given expression should be inferred in an environment. The environment is consisted of a set of functions with types:

```plaintext
head: forall[a] list[a] -> a
tail: forall[a] list[a] -> list[a]
nil: forall[a] list[a]
cons: forall[a] (a, list[a]) -> list[a]
cons_curry: forall[a] a -> list[a] -> list[a]
map: forall[a b] (a -> b, list[a]) -> list[b]
map_curry: forall[a b] (a -> b) -> list[a] -> list[b]
one: int
zero: int
succ: int -> int
plus: (int, int) -> int
eq: forall[a] (a, a) -> bool
eq_curry: forall[a] a -> a -> bool
not: bool -> bool
true: bool
false: bool
```
Sample Input #00

let x = id in x

Sample Output #00

forall[a] a -> a

Explanation #00:

x is just id in the environment.

Sample Input #01

fun x -> let y = fun z -> z in y

Sample Output #01

forall[a b] a -> b -> b

Explanation #01:

Function with variables which are not bounded in the environment should be generic function. The variable names appear in forall[] should be from a to z subject to their appearance order in type body.

Sample Input #02

choose(fun x y -> x, fun x y -> y)

Sample Output #02

forall[a] (a, a) -> a

Explanation #02:

The type of choose is forall[a] (a, a) -> a. So x and y should be of the same type.

Sample Input #03

fun f -> let x = fun g y -> let _ = g(y) in eq(f, g) in x

Sample Output #03

forall[a b] (a -> b) -> (a -> b, a) -> bool

Explanation #03:
The longest test case.

**Final note:**
All given expression are valid, non-recursive and can be inferred successfully in given environment. But an *optional* requirement is that your program should *fail* on incomplete uncurry version function calling. For example, `choose_curry(one)` should be inferred as `int -> int` but `choose(one)` just *fail* in infering.

*Tested by Bo You*