Task: Refactoring of a Simplified Makefile

Consider a slightly simplified version of the well-known UNIX `make` utility. Such simplified `make` writes to standard output (instead o execution) a sequence of commands that are necessary to achieve the first target in Makefile according to rules inside Makefile as describec below. Makefile is represented by standard input. Every rule has the following format:

target: subtarget$_1$ *subtarget*$_2$ ...
<TAB>command$_1$
<TAB>command$_2$
...

Each rule begins with a textual dependency line which defines *target* followed by a colon (:) and optionally an enumeration of *subtarget* on which the *target* depends. The dependency line is arranged so that the *target* depends on *subtarget*s. After each dependency line, a sequence of command lines may follow which define how to transform the *subtarget*s into the target. The sequence of command lines o the *target* has been written after all *subtarget*s of the *target* had been written. The *subtarget*s are written by the same algorithm as the *target*. Each command line must begin with a tab character to be recognized as a command. The tab is a whitespace character, but the space character does not have the same special meaning. *Subtarget*s are separated by space and the ordering, in which they are written t the standard output, is not defined. For every target holds that it need not contain either subtargets or command lines.

Your task is to write a program for refactoring of Makefile for the above described `make` utility. This refactoring program must identify al "dead code" and mark it as comment.

The input of this program will be the same as that of our simplified `make`. The output will be almost the same as input with only on difference, that all lines that are not necessary to achieve of the first target of the input, will be mark as comment. The comment line is the line with added # character at the beginning of that line.

If our simplified `make` utility could not be done due to unsatisfiability of one of its target, then the refactoring program writes only one lin output of the following form:

```
ERROR
```

You can assume that the format of the input is syntactically correct. Please note that the input can be huge, as large as hundreds of megabytes.

Example:

Input:

```
all: hello
hello: main.o hello.o
	g++ main.o hello.o -o hello
main.o: main.cpp functions.h
	g++ -c main.cpp
hello.o: hello.cpp functions.h
	g++ -c hello.cpp
hello2.o: hello2.cpp functions.h
	g++ -c hello2.cpp
hello.cpp:
	echo '#include ' > hello.cpp
	echo '#include "functions.h"' >> hello.cpp
	echo 'void print_hello(void){ cout << "Hello World!"; }' >> hello.cpp
main.cpp:
	echo '#include ' > main.cpp
	echo '#include "functions.h"' >> main.cpp
	echo 'int main() { print_hello();' >> main.cpp
	echo 'cout << endl; return 0; } ' >> main.cpp
hello2.cpp:
	echo '#include ' > hello2.cpp
	echo '#include "functions.h"' >> hello2.cpp
	echo 'void print_hello(void){ cout << "Hello All!"; }' >> hello2.cpp
functions.h:
	echo 'void print_hello(void);' > functions.h
```

Output:

```
all: hello
```

```
hello: main.o hello.o
	g++ main.o hello.o -o hello
main.o: main.cpp functions.h
	g++ -c main.cpp
hello.o: hello.cpp functions.h
	g++ -c hello.cpp
#hello2.o: hello2.cpp functions.h
#	g++ -c hello2.cpp
hello.cpp:
	echo '#include ' > hello.cpp
	echo '#include "functions.h"' >> hello.cpp
	echo 'void print_hello(void){ cout << "Hello World!"; }' >> hello.cpp
main.cpp:
	echo '#include ' > main.cpp
	echo '#include "functions.h"' >> main.cpp
	echo 'int main() { print_hello();' >> main.cpp
	echo 'cout << endl; return 0; } ' >> main.cpp
#hello2.cpp:
#	echo '#include ' > hello2.cpp
#	echo '#include "functions.h"' >> hello2.cpp
#	echo 'void print_hello(void){ cout << "Hello All!"; }' >> hello2.cpp
functions.h:
	echo 'void print_hello(void);' > functions.h
```

Here is the output of the simplified `make` utility:

```
echo '#include ' > main.cpp
echo '#include "functions.h"' >> main.cpp
echo 'int main() { print_hello();' >> main.cpp
echo 'cout << endl; return 0; } ' >> main.cpp
echo 'void print_hello(void);' > functions.h
g++ -c main.cpp
echo '#include ' > hello.cpp
echo '#include "functions.h"' >> hello.cpp
echo 'void print_hello(void){ cout << "Hello World!"; }' >> hello.cpp
g++ -c hello.cpp
g++ main.o hello.o -o hello
```