Elementary Watson

Dear old Dr. Watson is secretly envious of his friend Sherlock's awesome detective skills. Watson has been working on becoming better at observing minute details and collecting evidence, and has achieved some modest successes there. Sadly, his capacity for logical reasoning is still lacking. Recently Watson has learned that modern computational engines can do something that's called "automated reasoning," and he thinks that this is the solution to his conundrum.

Watson designs a system for automated logical reasoning that he is calling, with becoming modesty, WatLog, and he wants you to implement it. You try to explain him that operating this system would require pretty much the same skills as doing the reasoning on his own, but the good old doctor is adamant. If only to convince him of the error of his ways, you decide to accede to his request.

WatLog Syntax & Operation

Names in WatLog consist of one or more alphanumeric characters and dashes. The first character of a name must be alphabetic. Names are case-sensitve.

Comments may include any characters, except for newline.

The rest of the grammar for the input language of a WatLog system (assuming `<name>` and `<comment>` as described informally above) is as follows.

```
<variable> ::= "#" <name>
<relational-term> ::= "[" <name> ": " <simple-term> <simple-terms> "]"
<simple-term> ::= <name> | <variable> | <relational-term>
<simple-terms> ::= "" | ", " <simple-term> <simple-terms>
<equality-assertion> ::= "<" <simple-term> " = " <simple-term> ">"
<non-equality-assertion> ::= "<" <simple-term> " /= " <simple-term> ">"
<complex-term> ::= <simple-term> | <equality-assertion> | <non-equality-assertion>
<complex-terms> ::= "" | <complex-term> <complex-terms-1>
<complex-terms-1> ::= "" | ", " <complex-term> <complex-terms-1>
<rule> ::= <simple-term> "." | "{(" <complex-terms> ") => " <simple-term> "}."
<query> ::= "(" <complex-terms> ")?"
<command> ::= "quit!"
<no-op> ::= "" | "%" <comment>
<op> ::= <rule> | <query> | <command> | <no-op>
<input-line> ::= <op> <EOL>
```

Note that the grammar is strict with respect to whitespace. Superfluous whitespace is not allowed, and space characters may not be omitted if present in a certain position in the grammar.

A WatLog system should read input lines and react to them.

1. Upon reading a `<rule>` the system should add this rule to its knowledge base and output a line with the text "`Ok.`"
2. Upon reading a `<query>` the system should verify whether this query is satisfiable using the current knowledge base, output the results (including variable assignments, where applicable) for *all* satisfying derivations, and output "`Ready.`" on a separate line.
3. Upon reading a `<command>` the system should output a line with the text "`Bye.`" and terminate immediately (as the only command is "`quit!`").
4. Upon reading a `<no-op>` the system should do nothing and proceed to the next input line.

WatLog Semantics

Facts & Rules

If a rule is a `<simple-term>` (a "fact"), it should be interpreted as stating that the term is true for all variable assignments. Note that the fact syntax is merely syntax sugar for a rule with empty list of premises. There are no semantic differences between `FACT.` and `{() => FACT}.`

A `<rule>` should be interpreted as an inference rule. The part in the parentheses lists the premises, and the part after the double arrow (`=>`) is the conclusion. The premises may be `<simple-term>`s or `<complex-term>`s, while the conclusion is always a `<simple-term>`. As with facts, all variables in a rule are considered to be universally quantified. The rule states that if all premises can be proven true for some variable assignment, then the conclusion is also true for that variable assignment.

Queries

A query is a conjunction of `<complex-term>`s, with all variables existentially quantified. The goal of the WatLog system is to find all derivations and variable assignments such that the query holds true, using known facts and rules of inference.

To ensure well-specified evaluation order, query terms should be resolved left-to-right, and rules for resolution should be attempted in the same order they were entered into the system.

Equality

1. A name is equal to itself, but not equal to any other name or to any relational term.
2. A relational term is equal to another relational term if they have the same name, the same number of sub-terms, and all sub-terms are pairwise equal. If any of the conditions do not hold, the two relational terms are not equal.
3. A variable is always equal to itself. A variable may or may not be equal to another variable, but the conclusion cannot be made until both variables are (at least partially) assigned. Asserting equality of a variable to another term effectively assigns this variable.
4. A variable may never be equal to a relational term involving the same variable (Watson doesn't like infinite terms). Implement the occurs check to ensure this.

WatLog Query Output

If a query is unsatisfiable, the system should simply output "`UNSAT`" on a separate line.

For each satisfying derivation of a query with no variables, the system should output "`SAT`" on a separate line.

For each satisfying derivation of a query with variables, the system should output these two lines:

```
SAT:
=====
```

...followed by assignments of all variables in the query. Variable assignments should be listed in lexicographical order of variable names. All variables must be assigned to closed terms (no variables may be on the right-hand side of the assignment). Variable assignment format consists of a hasmark followed by a variable name, followed by "`:=`", followed by assigned closed simple term (in syntax described by `<simple-term>` - but note the requirement above to have no variables on the right-hand side).

Examples

Example 1: Simple Queries

Input:

```
% WatLog test case: simple
% verifies the basic reasoning

[r: a].
{([r: #z]) => [p: [r: #z], #z]}.
([p: [r: a], a])?
([p: #a, a])?
([p: #a, b])?
([p: [r: a], #a])?
([p: #a, #b])?
([p: #c, #c])?
([p: #a, #b], <#a = #b>)?
([p: #a, #b], <#a /= #b>)?
([p: #a, #b], <a /= #b>)?

quit!
```

The correct output is:

```
Ok.
Ok.
SAT
Ready.
SAT:
=====
#a := [r: a]
Ready.
UNSAT
Ready.
SAT:
=====
#a := a
Ready.
SAT:
=====
#a := [r: a]
#b := a
Ready.
UNSAT
Ready.
UNSAT
Ready.
SAT:
=====
#a := [r: a]
#b := a
Ready.
UNSAT
Ready.
Bye.
```

This test case includes one fact and one rule.

Query 1: The proposition is true, as the inference rule allows to make this conclusion from the single known fact.

Query 2: Same as with the previous one, if we assign `#a` to `[r: a]`.

Query 3: This fact cannot be deduced from known facts and inference rules. The query is unsatisfiable.

Query 4: As with the second query, but if we assign `#a` to `a`.

Query 5: Once again, the fact and the rule allow us to make a single inference (that `[p: [r: a], a]`), which satisfies this query for `#a := [r: a]` and `#b := a`.

Query 6: Not satisfiable, since `[r: a] /= a`.

Query 7: Ditto.

Query 8: Same as Query 5, the inequality assertion does not constrain the solution space here.

Query 9: Not satisfiable, since the only assignment satisfying the first term requires that `#b := a`, which contradicts the non-equality assertion that follows.

Example 2: Murder Mystery

Dr. Watson prepared an example illustrating how he expects to use WatLog, describing a case that he's been investigating.

```
% WatLog test case: Mystery (Simple)
% Col. Travis was poisoned...

{([means: #x], [motive: #x], [opportunity: #x]) => [suspect: #x]}.

{([doctor: #x], [poisoned: #y]) => [means: #x]}.
{([nurse: #x], [poisoned: #y]) => [means: #x]}.
{([veterinarian: #x], [poisoned: #y]) => [means: #x]}.
{([owns-firearm: #x], [shot: #y]) => [means: #x]}.
{([strong: #x], [strangled: #y]) => [means: #x]}.

{([jealous-of: #x, #y], [victim: #y]) => [motive: #x]}.
{([inherits-from: #x, #y], [financial-trouble: #x], [victim: #y]) => [motive: #x]}.

{([witnessed-by-at-on: #x, #z, #s, #t], [crime-scene: #s], [time-of-death: #t]) => [opportunity: #x]}.
{([has-no-alibi-for: #x, #t], [time-of-death: #t]) => [opportunity: #x]}.

{([poisoned: #x]) => [victim: #x]}.
{([shot: #x]) => [victim: #x]}.
{([strangled: #x]) => [victim: #x]}.

{([loves: #x, #y], [loves: #z, #y]) => [jealous-of: #x, #z]}.

{([son-of: #x, #y]) => [inherits-from: #x, #y]}.
{([daughter-of: #x, #y]) => [inherits-from: #x, #y]}.
{([spouse-of: #x, #y]) => [inherits-from: #x, #y]}.

[crime-scene: CountryHouse].
[time-of-death: Wednesday].
%[time-of-death: Tuesday].
[poisoned: ColTravis].
%[shot: ColTravis].

% What if the Colonel was poisoned on Tuesday?
% What if he was shot rather than poisoned?

ColTravis.
[spouse-of: ColTravis, Martha].
[loves: ColTravis, Martha].
[soldier: ColTravis].
[doctor: ColTravis].
[owns-firearm: ColTravis].
[strong: ColTravis].
[witnessed-by-at-on: ColTravis, Martha, CountryHouse, Tuesday].
[witnessed-by-at-on: ColTravis, Martha, CountryHouse, Wednesday].

Martha.
[spouse-of: Martha, ColTravis].
[nurse: Martha].
[has-no-alibi-for: Martha, Wednesday].

Jeffrey.
[son-of: Jeffrey, ColTravis].
[son-of: Jeffrey, Martha].
[golden-youth: Jeffrey].
[owns-firearm: Jeffrey].
[strong: Jeffrey].
[financial-trouble: Jeffrey].
[witnessed-by-at-on: Jeffrey, Mordred, CountryHouse, Wednesday].

Susan.
[daughter-of: Susan, ColTravis].
[daughter-of: Susan, Martha].
```

```
[doctor: Susan].
[witnessed-by-at-on: Susan, Martha, CountryHouse, Tuesday].

Mordred.
[loves: Mordred, Martha].
[veterinarian: Mordred].
[witnessed-by-at-on: Mordred, Susan, CountryHouse, Tuesday].
[has-no-alibi-for: Mordred, Wednesday].

% Is Jeffrey a suspect?
([suspect: Jeffrey])?

% Is Susan a suspect?
([suspect: Susan])?

% List all known suspects.
([suspect: #X])?

% Curiously, Col. Travis is a suspect in his own murder...
% While suicides certainly do happen, it doesn't seem plausible in this case.
% What motive could Col. Travis possibly have to take his own life?
% Perhaps, good old Watson didn't model the situation too well?

quit!
```

The correct output is:

```
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
```

```
Ok.
Ok.
Ok.
UNSAT
Ready.
UNSAT
Ready.
SAT:
=====
#X := ColTravis
SAT:
=====
#X := Mordred
Ready.
Bye.
```

Example 3: Insertion Sort

You know that WatLog is much more general than Dr. Watson guesses, so you implemented an insertion sort for cons cell lists over a small finite domain equipped with a full ordering relationship.

```
% WatLog test case: Insertion Sort

[lt-n: a, b].
[lt-n: b, c].
[lt-n: c, d].
[lt-n: d, e].

% transitive closure on lt-n relation
{([lt-n: #x, #y]) => [lt: #x, #y]}.
{([lt-n: #x, #y], [lt: #y, #z]) => [lt: #x, #z]}.
{([lt: #x, #y]) => [ge: #y, #x]}.
{(<#x = #y>) => [ge: #x, #y]}.

% test
([lt: #x, #y])?

% insertion sort
[sorted: nil, nil].
{([sorted: #t, #y], [insert: #h, #y, #z]) => [sorted: [cons: #h, #t], #z]}.
[insert: #h, nil, [cons: #h, nil]].
{([lt: #h, #h2]) => [insert: #h, [cons: #h2, #t], [cons: #h, [cons: #h2, #t]]]}.
{([ge: #h, #h2], [insert: #h, #t, #z]) => [insert: #h, [cons: #h2, #t], [cons: #h2, #z]]}.

% test
([sorted: [cons: a, [cons: b, nil]], #z])?
([sorted: [cons: b, [cons: d, [cons: a, [cons: b, [cons: c, [cons: e, nil]]]]]], #z])?

% all done here.
quit!
```

The correct output is:

```
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
Ok.
SAT:
=====
#x := a
#y := b
SAT:
=====
#x := b
#y := c
SAT:
=====
```

```
#x := c
#y := d
SAT:
=====
#x := d
#y := e
SAT:
=====
#x := a
#y := c
SAT:
=====
#x := a
#y := d
SAT:
=====
#x := a
#y := e
SAT:
=====
#x := b
#y := d
SAT:
=====
#x := b
#y := e
SAT:
=====
#x := c
#y := e
Ready.
Ok.
Ok.
Ok.
Ok.
Ok.
SAT:
=====
#z := [cons: a, [cons: b, nil]]
Ready.
SAT:
=====
#z := [cons: a, [cons: b, [cons: b, [cons: c, [cons: d, [cons: e, nil]]]]]]
Ready.
Bye.
```

Test Case Notes

All of the input files used in test cases are shorter than 25K. No individual input line is longer than 2.5K. The required output is never larger than 15K.

The search space will be finite for all the queries used in the test cases, if you prune it by enforcing assertions at the earliest possible moment.

Summary Notes

Ordering is important for your implementation to be considered correct. To ensure your answers are in the correct order, follow these rules:

1. Process query terms left to right.
2. Facts and rules should be prioritized according to the order in which they have been entered into the system. Earlier rules and facts should always be tried first.
3. Output the variable assignments in lexicographical order of variable names.

Note that there may be several derivations leading to the same variable assignment. You should output all derivations, without attempting to filter out duplicate assignments.

You may need to do something about variable names in facts and rules, otherwise you might accidentally

capture variables.

All variables must be assigned to terms with no free variables.

The grammar is strict with respect to whitespace. Spaces are required after commas and colons, as well as around `=>`, `=` and `/=`, and prohibited in all other places.