UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Programming Contest
Fall 1996

P. N. Hilfinger

1996 Programming Problems

To set up your account, execute

```
source ~ctest/contest/bin/setup
```

in all shells that you are using. (This is for those of you using the C-shell. Others will have to examine this file and do the equivalent for their shells.)

This booklet should contain eight problems on **??** pages. You have 5 hours in which to solve as many of them as possible. Put each complete C solution into a file N`.c` file and each complete C++ solution into a file N`.C` or N`.cc`, where N is the number of the problem. Each program must reside entirely in a single file. Each file should start with the line

```
#include "contest.h"
```

and must contain no other `#include` directives. Upon completion, each program *must* terminate by calling `exit(0)`.

Aside from files in the standard system libraries and those we supply, you may not use any pre-existing computer-readable files to supply source or object code; you must type in everything yourself. Selected portions of the standard `g++` class library are included among of the standard libraries you may use: specifically, the headers `string`, `vector`, `iostream.h`, `iomanip.h`, and `fstream.h`. Likewise, you can use the standard C IO libraries (in either C or C++), and the math library (header `math.h`). You may not use utilities such as `yacc`, `bison`, `lex`, or `flex` to produce programs. Your programs may not create other processes (as with the `system`, `popen`, `fork`, or `exec` series of calls). You may use any inanimate reference materials you desire, but no people. You can be disqualified for breaking these rules.

When you have a solution to problem number N that you wish to submit, use the command

`submit` N

from the directory containing N`.c`, N`.C`, or N`.cc`. Before actually submitting your program, `submit` will first compile it and run it on one sample input file. No submission that is sent after the end of the contest will count. You should be aware that `submit` takes some time before it actually sends a program. In an emergency, you can use

`submit -f` N

which submits problem N without any checks.

You will be penalized for incorrect submissions that get past the simple test administered by `submit`, so be sure to test your programs (if you get a message from `submit` saying that it failed, you will *not* be penalized). All tests will use the compilation command

`contest-gcc` N

followed by one or more execution tests of the form (Bourne shell):

N `<` *test-input-file* `2>` *junk-file*

which sends normal output to *test-input-file* and error output to *junk-file.* The output from running each input file is then compared with a standard output file. In this comparison, leading and trailing blanks are ignored and sequences of blanks are compressed to single blanks. Otherwise, the comparison is literal; be sure to follow the output formats *exactly.* It will do no good to argue about how trivially your program's output differs from what is expected; you'd be arguing with a program. Make sure that the last line of output ends with a newline. Your program must not send any output to `stderr`; that is, the temporary file *junk-file* must be empty at the end of execution. Each test is subject to a time limit of about 15 seconds. You will be advised by mail whether your submissions pass.

The command `contest-gcc [-g]` N, where N is the number of a problem, is available to you for developing and testing your solutions (as usual, the optional `-g` is for debugging information). It is equivalent to

`gcc -Wall -o` N `-O [-g] -I`*our-includes* N`.* -lstdc++ -lm`

The *our-includes* directory contains `contest.h`, which also supplies the standard header files. The files in `~ctest/submission-tests/`N, where N is a problem number, contain the input files and standard output files that `submit` uses for its simple tests.

All input will be placed in `stdin`. You may assume that the input conforms to any restrictions in the problem statement; you need not check the input for correctness. Consequently, you are free to use `scanf` to read in numbers and strings.

Scoring. Scoring will be according to the ACM Contest Rules. You will be ranked by the number of problems solved. Where two or more contestants complete the same number of problems, they will be ranked by the *total time* required for the problems solved. The total time is defined as the sum of the *time consumed* for each of the problems solved. The time consumed on a problem is the time elapsed between the start of the contest and successful submission, plus 20 minutes for each unsuccessful submission, and minus the time spent judging your entries. Unsuccessful submissions of problems that are not solved do not count. As a matter of strategy, you can derive from these rules that it is best to work on the problems in order of increasing expected completion time.

Protests. Should you disagree with the rejection of one of your problems, first prepare a file containing the explanation for your protest, and then use the `protest` command (without arguments). It will ask you for the problem number, the submission number (submission 1

is your first submission of a problem, 2 the second, etc.), and the name of the file containing your explanation. Do not protest without first checking carefully; groundless protests will be result in a 5-minute penalty (see Scoring above).

Terminology. The term *free-form input* indicates that input numbers, words, or tokens are separated from each other by arbitrary whitespace characters. By standard C/UNIX convention, a whitespace character is a space, tab, return, newline, formfeed, or vertical tab character.

1. Cribbage is a card game that uses a special board for scoring. Each player has a four-card hand. The dealer gets an additional four-card *crib*, and there is one additional (shared) card, called the *starter*. The cards rank ace (low), 2–10, jack, queen, king (high). Cards also have *count values:* ace counts 1; 2–10 count their face values; and jack, queen, and king each count 10.

Part of each turn consists of *showing* one's hand, which combines with the starter card to score points according to the following rules:

- Score 2 points for each distinct combination of cards whose counts add up to 15.
- Score 2 points for each distinct pair of cards with the same rank (each pair of aces, twos, jacks, etc.). Thus, three of a kind scores 6, since there are three distinct ways to pair up three cards, and four of a kind scores 12.
- Score one point for each card in each distinct maximal run of three or more cards in sequence by rank (suit is irrelevant here). I say "maximal" here to indicate that one counts a run such as 2-3-4-5 as four points; you don't get an additional three points for 2-3-4 and for 3-4-5.
- Score 1 point if the hand contains the jack of the same suit as the starter (this jack is called *his nobs* or *the right jack.*)
- (In real cribbage, flushes count also, but we won't worry about scoring them here.)

The dealer scores his hand and then his crib. Scoring the crib follows the same rules, except for the last: four-card flushes don't score.

Write a program that reads in a sequence of data sets—each consisting of an opponent's hand, dealer's hand, crib, and starter card, in that order—and prints out, for each set, the opponent's hand's score, dealer's hand's score, and the crib's score in the format shown in the example. The hands and the crib each consists of four cards. A card has the form RS (no intervening space), where R is a rank—one of `A`, `2`, `3`, `4`, `5`, `6`, `7`, `9`, `10`, `J`, `Q`, or `K`—and S is a suit—one of `S`, `H`, `D`, or `C`. The input is in free format.

Thus, the hand consisting of `4D`, `6D`, `5S`, and `9H`, with a starter of `6H` scores 16: `4D`, `5S`, and `6H` add up to 15 (2 points) and also constitute a run of three cards (3 points); `4D`, `5S`, and `6D` likewise add up to 15 and constitute a run; `6H` and `9H` add up to 15; `6D` and `9H` add up to 15; and the two 6's make a pair (2 points). To indicate that a hand scores no points, it is traditional to say that it "scores 19," since no cribbage hand can score 19 points. Additional examples follow on the next page.

Example:

Input	Output
4D 6D 5S 9H 3S 3H 3C 6S KD AS 8S 4D 6H KD 10D JD AS 2S 2H 3H 3S 3D 8D 9D QD AD AD AS AC 8D 2S 3S 4D 5C 3D 8D 9D QD AH	Hand 0: Opponent's hand scores 16 Dealer's hand scores 18 Crib scores 4 Hand 1: Opponent's hand scores 3 Dealer's hand scores 16 Crib scores 19 Hand 2: Opponent's hand scores 12 Dealer's hand scores 7 Crib scores 19

2. A *linker* is a program that combines compiled object files of a program into an executable file, linking unresolved external name references (to subprograms or global variables) in one object file to their definitions in another. Most linkers have a facility for dealing with *libraries*—collections of object files—selecting a minimal set of object files out of a library so as to define all the outstanding unresolved names. The individual object files in the library may themselves reference unresolved names, so when the linker includes one object file from a library, it may have to include others as well. For this problem, you will write a program to determine what the minimal set is.

The input to the this program will consist of a set of *root names* (strings separated by whitespace) for which definitions are to be found, followed by an isolated semicolon (i.e., a semicolon that is surrounded by whitespace), followed by a description of the contents of a library. This description consists of a sequence of *object file directories,* each having the form

filename type1 name1 type2 name2 $\cdots$;

The *filename* and all the `name`s are strings. Each *type* is either `U`, indicating that the following name is used in *filename*, but not defined, or `D`, indicating that the following name is defined in *filename.* Each object file directory ends with an isolated ';'. The input is in free format. You may assume file names are limited to 256 characters. You may assume that each required name is defined in exactly one library file.

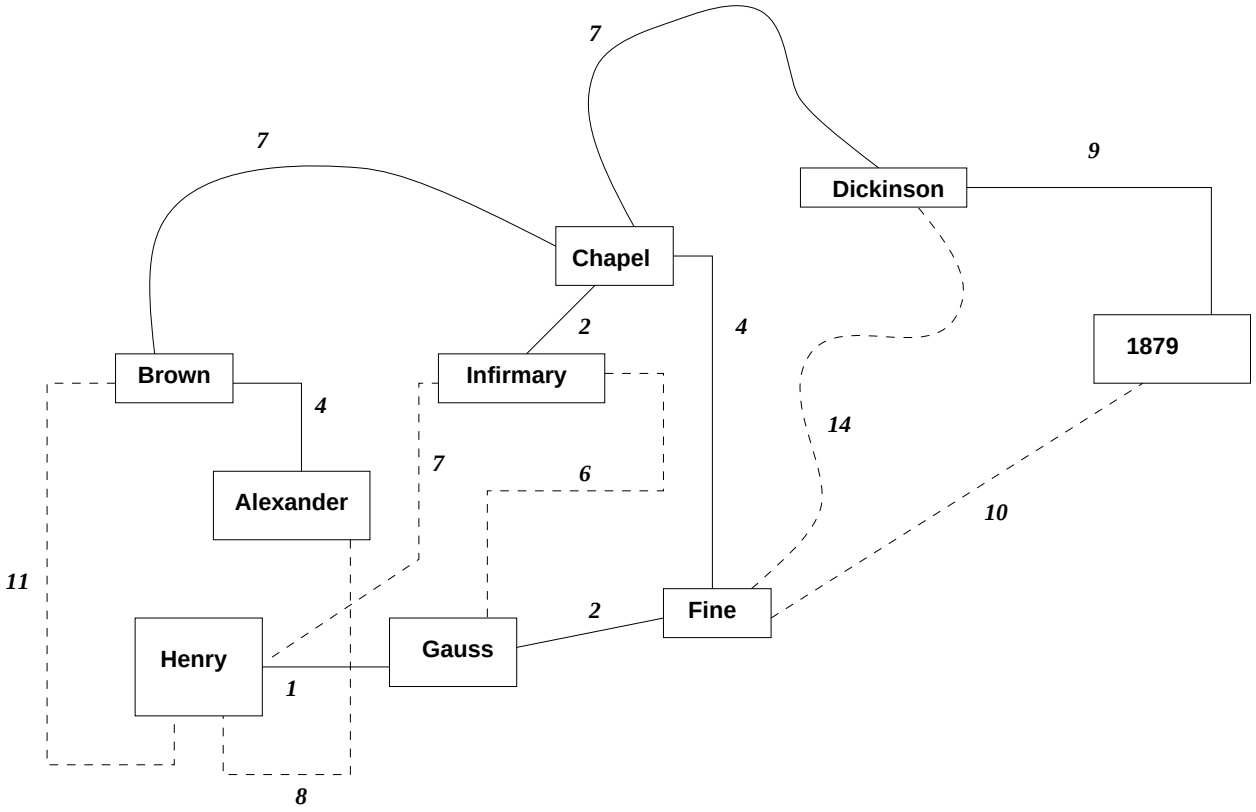
The output consists of a list of the minimal set, S, of filenames from the library such that the root names and the type 'U' names from the files in S all have definitions (type 'D' names) among the files in S. List the result in the format shown in the example on the next page, in the order in which the file names appear in the description of the library.

Example:

Input	Output
`print sin cos read ;`	`files.o`
`files.o D close D open ;`	`math.o`
`math.o  D sin D cos U error ;`	`diagnose.o`
`diagnose.o D error ;`	`input.o`
`format.o D printf D scanf ;`	`output.o`
`input.o D read U close U`	
`error D seek ;`	
`output.o D print U error U close ;`	

3. Wolverine College, in Michigan's Upper Peninsula, is often covered with snow. Its grounds crew tends to get somnolent in the winter, however, and tries to do as little work as possible in clearing the sidewalks. Specifically, once they know what buildings will be used on any given day (the faculty isn't so energetic either), they clear off the shortest combination of sidewalks possible that leave some (possibly long) path connecting all the open buildings. Your task here is to write a program that determines this set of sidewalks.

For example, given the diagrammatic map below, the solid lines represent a minimum-length set of clear sidewalks, the dashed lines represent the snow-covered walks, and the numbers are lengths of the walks.



Each set of input to your program will consist of a sequence of *sidewalk descriptions,* all in free form, followed by an isolated semicolon (i.e., surrounded by whitespace). A sidewalk description consists of two building names and a distance (a positive integer). Building names consist of up to 128 letters, digits, and underscores—no blanks or punctuation. You may assume that all buildings are connected by some path to all other buildings (so that, in particular, all building names are included in at least one sidewalk description). You may make no assumptions about the number of buildings or sidewalks. You may assume that no walk will be longer than 10^9 long. You may also assume that there will be a unique solution.

For each set of input, the output is to have the form illustrated in the sample input on the next page. Output the sidewalks in the same order as their descriptions were input.

Input	Output
Brown Henry 11 Henry Alexander 8 Gauss Henry 1 Infirmary Henry 7 Brown Chapel 7 Alexander Brown 4 Infirmary Chapel 2 Gauss Infirmary 6 Gauss Fine 2 Fine Dickinson 14 1879 Fine 10 Dickinson 1879 9 Chapel Dickinson 7 Chapel Fine 4 ; Alexander Brown 4 Brown Henry 11 Henry Alexander 8 ;	Set 0: Clear Gauss to Henry Clear Brown to Chapel Clear Alexander to Brown Clear Infirmary to Chapel Clear Gauss to Fine Clear Dickinson to 1879 Clear Chapel to Dickinson Clear Chapel to Fine Set 1: Clear Alexander to Brown Clear Henry to Alexander

4. You've probably seen those puzzles in which you are given clues such as "The plumber lives in the green house" and "Joe is not the archdeacon" and you are supposed to figure out everyone's occupation and taste in exterior latex enamel. In this problem, you are to write a general-purpose solver for this type of problem.

The input consists of sets of specifications, each set terminated by the question "`So?`". At the beginning of each set, there is an integer number, N, giving the number of people, occupations, and houses. Each of the specifications that follows it has one of the following forms:

Name *name*
Occupation *occupation*
House *color*
LivesIn *name color*
NotLivesIn *name color*
IsA *name occupation*
IsNotA *name occupation*
JobOf *color occupation*
NotJobOf *color occupation*

The first three simply declare that certain people, occupations, and house colors exist, without giving any further information ("`Occupation plumber`" means "There is a plumber."). The next two tell about what color house a certain person lives or does not live in ("`LivesIn Paul white`" means "Paul lives in the white house.") The next two tell about a person's job ("`IsA John biochemist`" means "John is the biochemist.") The last two refer to what job the person occupying a house of a certain color does or does not have ("JobOf blue actuary" means "The person in the blue house is the actuary.") Each *name, occupation,* and *color* consists of up to 128 alphabetic characters. The input is free-form. You may assume that all N names, colors, and occupations will be mentioned in the input. Each person occupies exactly one house, distinct from everyone else's, and has one job, distinct from everyone else's.

The output consists of a sequence of assignments of names to occupations and houses, each having the form

`The` *occupation, name,* `lives in the` *color* `house.`

(as in "The plumber, John, lives in the yellow house.") The output is sorted according to the order in which the *names* are mentioned in the input. If the sentences are contradictory, the output consists of the single sentence,

```
This is impossible.
```

If the sentences admit of more than one solution, the output consists of the single sentence,

```
I don't know.
```

There are examples on the next page.

Example:

Input	Output
1 Name Sue House brown Occupation professor So? 3 IsNotA John carpenter JobOf blue plumber House green LivesIn John yellow NotLivesIn Mary blue Name Tom Occupation architect So? 2 LivesIn Jack blue NotLivesIn Mary blue Occupation mechanic House red Occupation architect So? 2 LivesIn Jack white IsA Jack carpenter IsA Jill clerk JobOf yellow carpenter So?	Set 0: The professor, Sue, lives in the brown house. Set 1: The architect, John, lives in the yellow house. The carpenter, Mary, lives in the green house. The plumber, Tom, lives in the blue house. Set 2: I don't know. Set 3: This is impossible.

5. It's not necessary to use purely non-negative digits to compose numerals. Indeed, the denizens of the planet Gorgonea Quarta IV, perhaps due to the rather unusual configuration of their "hands," use a number system in which there are three digits (called *trits*): '`0`', '`1`', and '`-`', having values 0, 1, and −1, respectively. In this number system, each position has a value 3 times that of the position to its right. Thus, the numeral '`10-`' represents the number 8 (since $8 = 1\times9+0\times3+-1\times1$) and '`-1`' represents the number -2 (since $-2 = -1\times3+1\times1$).

Write a program that reads in a sequence of integers in the range -2^{31} to $2^{31}-1$, and prints out their equivalents in Gorgonea Quartan (GQ) notation, using the format shown in the examples below.

Example:

Input	Output
`10`	`10 = 101 GQ`
`2`	`2 = 1- GQ`
`-17`	`-17 = -101 GQ`
`42`	`42 = 1---0 GQ`
`1024`	`1024 = 111-0-1 GQ`
`-2147483648`	`-2147483648 = -10110100011---1-1--1 GQ`

6. Project SCRUB (Simple Cleaning Robots Using Brushes) aims to provide every household with a collection of simple-minded robots for keeping floors clean and free of litter. Each robot has detectors that inform it of rubbish or other robots in its vicinity. The robots' behavior follows these rules:

- Each robot has a distinct number by which it may be identified.
- Robots move at a constant velocity.
- Once per second, each robot—call it robot R—consults its detectors and changes its direction, if necessary, as follows:
  - If there is another robot within a given *avoidance distance* of R's center, then R changes direction so as to be moving in the direction exactly opposite from the other robot's center. When there is more than one robot within the avoidance distance, R ignores all but the lowest-numbered robot.
  - If there are no robots within the avoidance distance and there is still dirt or rubbish, R turns in the direction of the nearest pile of it.
  - Otherwise, R stops moving and turns itself off. Robots never restart once they are off.
- All robots that haven't yet turned themselves off are synchronized; they all make their (instantaneous) changes in direction together each second.
- Once a piece of dirt or rubbish gets within a certain *cleaning distance* of a robot, it gets sucked up into the robot's receptacle, and no longer registers on any robot's sensors. This happens at most once a second, *before* the robots look around for dirt. If more than one robot is within the cleaning distance, the lower-numbered robot gets the dirt.

To evaluate the effectiveness of this design, you are to provide a simulation program. The input to the program consists of an integer, N, giving the maximum simulation time in seconds, followed by a floating-point number V, giving the velocity of each robot (in units moved per second), followed by a floating-point number A, giving the avoidance distance, followed by a floating-point number C, giving the cleaning distance, followed finally by any number of robot and dirt placement descriptions in free form. Each of these, in turn, has the form

$$T \quad x \quad y$$

Here, T is either '`R`' to indicate a robot or '`D`' to indicate a pile of dirt. The coordinates x and y are both real numbers (C/C++ type `double`). After reading in this input, your program is to simulate the robots' movements for N seconds, reporting each time a robot either sweeps up some dirt, runs into another robot, or stops moving, using the format in the example. At time 0, the robots sweep up any nearby dirt, decide on a direction and start moving, in accordance with the rules above.

The robot whose description occurs first in the input is robot number 0, the next is 1, and so forth. At each time that something happens, report dirt encounters first, then collisions,

then robots that stop. In all cases, list robots in numerical order. Identify piles of dirt by number (starting with 0) in the order in which their descriptions are read in. Report up to and including time N. That is, report collisions, dirt encounters, and robots halting that occur at time N.

Example 1: The last columns in these examples show the position of the robot being reported on in the middle column. It's just for your convenience; your program *must not* print the information in the last column!

Input	Output	Robot's Position
`100 1 2.5 0.5`	`Robot #0 picked up dirt pile #0 at time 2.`	
`R 0 0`	`Robot #0 turns to avoid robot #1 at time 2.`	*(0, 2)*
`D 0 1.9`	`Robot #1 turns to avoid robot #0 at time 2.`	*(0, 2)*
`R 0 5`	`Robot #0 stops at time 3.`	*(0, 3)*
	`Robot #1 stops at time 3.`	*(0, 1)*
		(0, 4)

Example 2:

Input	Output	Robot's Position
`100 1 1.0 0.5`	`Robot #1 picked up dirt pile #1 at time 1.`	
	`Robot #0 picked up dirt pile #0 at time 2.`	*(0.71, 4.29)*
`R 0 0 R 0 5`	`Robot #0 picked up dirt pile #2 at time 8.`	*(0.21, 1.99)*
`D 0.2 1.9 D 1 4 D -2`	`Robot #0 stops at time 8.`	*(-1.87, -3.64)*
`-4`	`Robot #1 stops at time 8.`	*(-1.87, -3.64)*
		(-1.44, -2.36)

Example 3:

Input	Output	Robot's Position
`5 1 1.0 0.4`	`Robot #0 turns to avoid robot #1 at time 1.`	
	`Robot #1 turns to avoid robot #0 at time 1.`	*(0, 1)*
`R 0 0`	`Robot #0 turns to avoid robot #1 at time 3.`	*(0, 1.9)*
`R 0 2.9`	`Robot #1 turns to avoid robot #0 at time 3.`	*(0, 1)*
`D 0 0.5`	`Robot #0 turns to avoid robot #1 at time 5.`	*(0, 1.9)*
	`Robot #1 turns to avoid robot #0 at time 5.`	*(0, 1)*
		(0, 1.9)

7. Consider a string of binary digits, such as '10110111010'. One way to describe this is to write it as '$10(1101)^2 0$.' Here, the notation '$(s)^n$' means "the string s repeated n times." We could also write it as '$(101)^2 11010$' or '$10110(1)^3 010$'. However, these latter two renderings require more binary digits than the first did (8 and 9, respectively, versus 7).

Write a program that, given a sequence of non-null binary strings, prints out a representation for each in the form

$$\alpha\texttt{(}\beta\texttt{)[}n\texttt{]}\gamma$$

where α, β, and γ stand for strings of binary digits, and $n \geq 1$ is an integer. The strings α and γ may be empty (but β never is). For each one, find a representation that minimizes the combined length of '$\alpha\beta\gamma$' (i.e., ignoring n). Where more than one such minimal representation is possible, choose the one that minimizes the length of α. Where there is more than one minimal representation that minimizes the length of α, choose the one that minimizes the length of γ. See the examples for the precise format.

The input will consist of binary strings of up to 1024 binary digits in free form.

Example:

Input	Output
`10110111010`	`10110111010 = 10(1101)[2]0`
`0  111111 101`	`0 = (0)[1]`
`10101`	`111111 = (1)[6]`
	`101 = (101)[1]`
	`10101 = (10)[2]1`

8. The phrase *propositional calculus* is just a pretentious term for a logic involving simple logical formulae containing variables and the logical connectives '`&`' (and), '`|`' (or), '`-`' (not), and `>` (implies). For convenient input, we can write a formula such as '`(x&y) > (-x | y)`' in the One True Syntax like this:

```
(> (& x y) (| (- x) y))
```

that is, each (sub)expression is either a simple variable or is a parenthesized expression with the operator first, followed by one or two operand subexpressions in the same format (recursively). For simplicity, assume that all logical variables are one-character lower-case letters. The presence or absence of whitespace has no effect on the meaning of the expression.

An *axiom schema* has the same format, but may also contain upper-case letters, which we'll call *pattern variables:*

```
(> A (> B A))
```

An axiom schema *matches* a formula $\mathcal{F}$ if one can substitute for all its pattern variables so as to get $\mathcal{F}$. Thus, '`(> A (> B A))`' matches '`(> x (> x x))`' or '`(> (& x z) (> y (& x z)))`', but it does not match '`(> x (& x x))`' or '`(> x (> x y))`'. A formula (i.e., not a schema) matches another formula if they are identical. We never match one schema to another.

A *proof with k givens* is a sequence of axiom schemata and formulas, $X_1 \cdots X_n$, such that for all $1 \le i \le n$, either

- $i \le k$ (i.e., "X_i is given."), or
- There is some $j < i$ such that X_j matches X_i (X_i must be a formula, not a schema), or
- For some $j < i$ and $k < i$, X_j is the formula `(>` X_k X_i`)`. (This is the inference rule known as *modus ponens:* if A is true and A implies B, then infer B is true.)

Write a program that reads in a sequence of formulae and indicates whether it is a proof according to these rules. The givens are separated from the rest of the proof by empty parentheses, '`()`', which will always appear. The output consists of an echo of the proof up to either its end, if it is valid, or the first erroneous line, if it is not, with a message about whether it checks. Use the format illustrated in the examples. [Hint: The conventional meanings of the connectives is irrelevant in this problem.]

Example 1:

Input	Output

`(> A (> B A))`
`x`
`()`
`(> x`
`  (> y x))`
`(> y`
`x)`
	`(> A (> B A))`
	`x`
	`()`
	`(> x (> y x))`
	`(> y x)`
	`*Proof OK*`

Example 2:

Input	Output

`(> A (- A))`
`x`
`()`
`(> x (- y))`
`(- y)`
	`(> A (- A))`
	`x`
	`()`
	`(> x (- y))`
	`*Error in proof*`

Example 3:

Input	Output

`(> A (> B A))`
`(> (> A B)`
`  (> (> A (> B C))`
`  (> A C)))`
`()`
`(> x (> x x))`
`(> (> x (> x x))`
`  (> (> x (> (> x x) x))`
`  (> x x)))`
`(> (> x (> (> x x) x))`
`  (> x x))`
`(> x (> (> x x) x))`
`(> x x)`
	`(> A (> B A))`
	`(> (> A B) (> (> A (> B C)) (> A C)))`
	`()`
	`(> x (> x x))`
	`(> (> x (> x x)) (> (> x (> (> x x) x)) (> x x)))`
	`(> (> x (> (> x x) x)) (> x x))`
	`(> x (> (> x x) x))`
	`(> x x)`
	`*Proof OK*`