```
Problems Index                Wed Oct 19 02:44:05 EDT 2005


Easy Problems:
--------------

    problems/isa
        What are you?

    problems/exchanged
        Typing errors.

    problems/stringhash
        Randomizing the input.

    problems/pseudopi
        Throwing chalk.

Medium Difficulty Problems:
---------------------------

    problems/faireye
        Standing exactly in between.

    problems/overlapgame
        Sudden death by words.

Difficult Problems:
-------------------

    problems/chromatic
        The Computer Science coloring book.

    problems/prooflabel
        All you need to know about ‘why?’.
```

Is It or Isn’t It?
-- -- -- ----- ---

The ‘Is A’ relation follows some very simple rules. For example, if

```
        x is a Y
        Y is a Z
then
        x is a Z
```

Here x is a proper noun, like ‘Jill’ or ‘Jack’, denoting a particular object, and Y and Z are generic nouns, like ‘mammal’ or ‘animal’, denoting properties. So the above is like

```
        Jack is a mammal
        mammal is a animal
```

therefore

```
        Jack is a animal
```

all of which, of course, needs further editing to be good English, but is good enough for the internal thoughts of a computer.

You are given some data consisting of nothing but ‘is a’ relations in which all objects are named by single lower case letters and all properties are named by single upper case letters. You are then asked questions like

```
        x is a Z?
```

which according to the above data has the answer ‘true’. However, if you cannot deduce that something is true, then it is not necessarily false, so given the above data if you are asked

```
        x is a W?
```

the answer should be ‘unknown’, and NOT ‘false’.

Input

A sequence of test cases.

Each test begins with a sequence of data lines, each of the form

```
        x is a Y
```

or the form

```
        X is a Y
```

where x can be replaced by any lower case letter and X and Y can be replaced by any upper case letters.

Following the data lines is a sequence of question lines, each of the form

```
        x is a Y?
```

or the form

```
        X is a Y?
```

where x can be replaced by any lower case letter and X and Y can be replaced by any upper case letters.

The question lines are followed by a single end line containing just ‘.’, which ends the test case

The input terminates with an end of file.

To make input easy, each data line is exactly 8 characters, each question line exactly 9 characters, characters 2 through 7 of each line are ' is a ', characters 1 and 8 of each line are letters, and character 9 of each question line is '?'. Character 8 must be upper case, while character 1 may be lower or upper case.

Also to make the algorithm easier, each test case will be such that if you can deduce that 'X is a Y' is true, then you CANNOT also deduce that 'Y is a X' is true. Thus there are no 'loops' in the deductions.

Example Input
------- -----

```
x is a P
P is a Q
Q is a R
R is a S
P is a M
M is a N
x is a P?
x is a S?
R is a P?
Q is a N?
.
B is a C
B is a D
B is a E
x is a D
C is a E?
E is a B?
x is a B?
.
```

Output

For each test case, a sequence of answer lines that correspond to the test case question lines, followed by a single end line containing nothing but '.'.

An answer line contains nothing but 'true' or 'unknown', according to whether or not the statement in the corresponding question line can be deduced from the data or not.

Example Output
------- ------

```
true
true
unknown
unknown
.
unknown
unknown
unknown
.
```

File: isa.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Mon Oct 17 00:00:54 EDT 2005

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $

Exchanged Compare

Often times people will mistype a word by exchanging adjacent letters. You are asked to write a compare function which returns true if and only if a first word equals a second word after zero or more pairs of non-overlapping letters in the second word are exchanged.

Input

A sequence of test cases, each a single line containing two words. The words are separated by spaces and tabs. The input ends with an end of file.

The words contain only lower case letters, and no line is longer than 80 characters.

Output

For each test case, one line containing ‘true’ if the two input words are equal after non-adjacent letter pair exchanges, and ‘false’ otherwise.

Example Input

```
hello hello
hello helol
hello heoll
hello ehllo
hello ehlol
hello hleol
hello helo
```

Example Output

```
true
true
false
true
true
true
false
```

File: exchanged.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Tue Oct 18 10:05:24 EDT 2005

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $

```
String Hashing
------ -------

It is often desirable to compute a hash code from a
character string.  One good way of doing this is to use
the function:

    hash = hash(N)
    hash(n) = P * hash(n-1) + c[n-1] mod M if n > 0
    hash(0) = 0

where

    N is the number of characters in the string
    c[0], c[1], ..., c[N-1] are the characters of the
        string
    0 <= c[i] < 256 for all 0 <= i < N
    M is an integer > 0, the modulus of computation
    P is a number prime to M
    hash(n) is the hash code of the first n characters
        of the string
    hash is the hash code of the entire string

Good values of P and M are

    M = 2**32
    P = 33 or 65599

Computing modulo 2**32 is fast because it is just trun-
cating to 32 bits.  Multiplying by 33 = 2**5 + 2**0 can
be done quickly by one shift and one addition.  Multi-
plying by 65599 = 2**16 + 2**6 - 2**0 can be done by 2
shifts, one addition, and one subtraction.

You have been asked to compute hash values for some
strings.  However, to be absolutely sure there are no
arithmetic overflow problems, we are simplifying the
problem by requiring

        0 < P < M < 2**15

Also, we do NOT require that M and P be relatively
prime.

Input
-----

For each test case, one line containing

        M P STRING

in the given order.  M, P, and STRING are separated by
whitespace consisting of spaces and tabs.  M and P are
integers, and STRING is a sequence of at most 80 non-
whitespace characters.

The input terminates with an end of file.


Output
------

For each test case, one line containing

        M P STRING HASH

which copies M, P, and STRING from the input and outputs
the 'hash' value computed for the STRING character
string using M and P.
```

```
Example Input
------- -----

10000 100 A
10000 100 B
10000 100 C
10000 100 D
10000 100 AB
10000 100 CD
32000 33  AB
32000 33  CD
32000 33  ABCDEFGHIJ
32000 33  BACDEFGHIJ
32000 33  %^@abc++=903#?..."

Example Output
------- ------

10000 100 A 65
10000 100 B 66
10000 100 C 67
10000 100 D 68
10000 100 AB 6566
10000 100 CD 6768
32000 33 AB 2211
32000 33 CD 2279
32000 33 ABCDEFGHIJ 5207
32000 33 BACDEFGHIJ 12919
32000 33 %^@abc++=903#?..." 11238
```


```
File:       stringhash.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Wed Oct 19 07:16:00 EDT 2005
```



```
RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $
```

Pseudo-Random Computation of PI

One of the classic demonstrations of probability is the following. The professor draws a large square on the blackboard, and draws its inscribed circle. Then standing with her back to the board, she throws pieces of chalk at the square. After this she counts the number M of hits in the circle and the number N >= M of hits in the square (including those in the circle), and demonstrates that M/N is about PI/4. This is because PI/4 is the area of the inscribed circle divided by the area of the square, and the probability of hitting any small part of the square is roughly identical to hitting any other small part of the square.

You have been asked to simulate the demonstration in the computer. The square is to be simulated by the unit square in the XY-plane, [0,1]x[0,1], which has (0,0) as its lower left corner and (1,1) as its upper right corner. To simulate throwing the chalk, two random integers X and Y are 'drawn uniformly' (see below for details) from the range 0 .. S-1, where S > 0 is some integer. Then the coordinates where the chalk strikes are set at ((X + 0.5)/S, (Y + 0.5)/S). These are inside the square, so all our 'throws' count toward N. They are inside the circle, and count toward M, if and only if the chalk strikes at a distance of 0.5 or less from the center of the circle, (0.5, 0.5).

Thus if S = 100 and the first two random integers drawn are 37 and 69, the chalk point is (0.375,0.695) which is distance 0.23 from (0.5,0.5), and is therefore in the circle and counts toward both M and N.

Drawing Random Numbers

You are asked to draw pseudo-random numbers according to the equation:

```
RANDOM = ( RANDOM * MULTIPLIER ) mod MODULUS
```

where RANDOM is the value of the pseudo-random number, the equation steps from the the last pseudo-random number to the next pseudo-random number, and MULTIPLIER and MODULUS are fixed values that determine the pseudo-random number sequence.

To get started, RANDOM is initialized to a value called SEED. The first pseudo-random number in the sequence is not SEED, but the first number after SEED in the sequence.

If MULTIPLIER and MODULUS have good values for this purpose, the resulting sequence of numbers appears when tested to be truly random and uniformly distributed in the range from 1 through MODULUS - 1. Uniformly distributed means all values in this range are equally probable. The choices

```
MULTIPLIER = 7**5 = 16807
MODULUS = 2**31 - 1 = 2147483647
```

are very good for this purpose.

For example, if MULTIPLIER and MODULUS are as just given, and the SEED is 374332679, then the first two random numbers are 1429733890 and 1342962947.

A remaining difficulty is how to convert uniformly distributed integers from 1 through MODULUS - 1 to uniformly distributed integers from 0 through S-1. An easy solution, which we will adopt, is to set

```
        S = MODULUS - 1
```

and subtract 1 from each value of RANDOM. Thus ‘a chalk throw’ is simulated by executing

```
    RANDOM = ( MULTIPLIER * RANDOM ) mod MODULUS
    X = RANDOM - 1
    X = (X + 0.5 ) / S
    RANDOM = ( MULTIPLIER * RANDOM ) mod MODULUS
    Y = RANDOM - 1
    Y = (Y + 0.5 ) / S
```

to yield (X,Y) in the unit square.

Implementation of the above algorithm requires integers longer than 32 bits. In C or C++ you can use doubles and the fmod function. Or you can use ‘long long’s and the % operator. In JAVA you can use ‘long’s and the % operator. Remember, ‘long’s are only 32 bits in C and C++, but are 64 bits in JAVA. ‘long long’s are 64 bits in C and C++.

Input

For each of several test cases, one line containing four numbers in the order:

```
        N MULTIPLIER MODULUS SEED
```

The numbers may be separated by spaces or tabs. All input numbers are positive integers below 2**31 (but some products computed by intermediate computations will be larger).

Input ends with an end of file.

The simulation is to be done with RANDOM initialized to SEED (SEED is NOT the first pseudo-random number) and S = MODULUS - 1.

Output

For each test case one line containing five numbers in the order:

```
        N MULTIPLIER MODULUS SEED PI_ESTIMATE
```

where the first four numbers are copied from the input, and PI_ESTIMATE equals 4*M/N expressed as a decimal number with exactly 5 decimal places.

Example Input
------- -----

```
100      16807 2147483647 374332679
1000     16807 2147483647 374332679
10000    16807 2147483647 374332679
100000   16807 2147483647 374332679
1000000  16807 2147483647 374332679
```

Example Output
------- ------

```
100 16807 2147483647 374332679 3.20000
1000 16807 2147483647 374332679 3.13600
10000 16807 2147483647 374332679 3.15960
100000 16807 2147483647 374332679 3.13888
1000000 16807 2147483647 374332679 3.14167
```

File: pseudopi.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 19 07:19:20 EDT 2005

RCS Info (may not be true date or author):

```
    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $
```

Fair Eye’s Secret

Fair Eye, the referee, is known for making correct calls. Scooper, the news reporter, thinks he has figured out the secret of Fair Eye’s success. Just before making a call, Fair Eye carefully positions himself at an equal distance from any place where an event that needs to be called may occur.

To test his theory Scooper uses Sky Cam to measure the location F of Fair Eye and the locations P1, P2, P3 of places where events that may need to be called occur. For some reason there are almost always three such places, and Scooper ignores the cases where there are not three. To test his theory Scooper wants to compute for every three places P1, P2, and P3 the exact location of the point C equidistant from these three places, so that C may be compared to F, where Fair Eye positions himself.

Mathematicians call C the circumcenter of P1, P2, and P3, or of the triangle whose vertices are P1, P2, and P3. It is the center of the circumscribed circle of that triangle.

Input

For each case, a single line containing the 6 numbers

```
        x1 y1 x2 y2 x3 y3
```

defining three points: P1 = (x1,y1), P2 = (x2,y2), and P3 = (x3,y3). The 6 numbers are real numbers (and may be negative).

An end of file terminates the input.

Output

For each case, a single line containing the 2 numbers

```
        x y
```

defining the circumcenter C = (x,y) of the three points P1, P2, P3.

Both x and y must be printed with exactly 3 decimal places.

You may assume that double precision floating point numbers will suffice to compute C with adequate precision, and that the three points are not so close to being co-linear that there will be computation problems.

Example Input

```
0.0 0.0 1.0 0.0 0.0 1.0
0.0 0.0 1.0 1.0 0.0 1.0
0.0 0.0 1.0 2.0 2.0 0.0
```

Example Output

```
0.500 0.500
0.500 0.500
1.000 0.750
```

```
File:      faireye.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Tue Oct 18 11:29:13 EDT 2005

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $
```

The Overlap Game

The Overlap Game is played with a set of words that are used to create a string of letters. The first player picks a word from the set and puts it on the board, creating the string. Thereafter each player picks a word and puts it at the end of the string, BUT, the beginning of the word must overlap the end of the string by one or more letters. The word is placed so its overlapping letters actually overlap those at the end of the string. Each move must increase the number of letters on the board.

Thus if a game starts with the words

```
        THE     EATEN   ENCHANTMENT
```

the first move could result in

```
        THE
```

the second move could result in

```
        THEATEN
```

and the third move could result in

```
        THEATENCHANTMENT
```

However, a second move resulting in

```
        THENCHANTMENT
```

is also possible, and then there could be no third move.

There are only two players in this game, and the person to move last loses. Play stops only when no more words can be added to the string.

You have been asked to assist a player by determining winning moves.

Input

For each case, a list of words followed by the mark ‘*’. The words and marks are separated by whitespace, where any combination of spaces, tabs, and line ends are considered to be whitespace.

All words consist of just upper case letters. There are at most 100 words in a case, and each word is at most 20 letters.

An end of file terminates the input.

Output

For each case, a single line containing just the (upper case) words the first player can play first to force a win. The words are separated by spaces.

If there are no words that will result in a win, the single line should instead contain just lower case ‘lose’. This indicates the first player must lose if the second player plays optimally.

```
Example Input
------- -----

THE EATEN ENCHANTMENT *
THE EATEN ENCHANTMENT ENTICES *
THIS HISTORY IS YIPPING SILLY *
AB BC CD DE EA *
HISTORY YES SENSIBLE YEOWL ELOQUE LONG *
THE ENVELOP OPERATES TESTING GREAT
    EATERIES POST *


Example Output
------- ------

lose
EATEN ENCHANTMENT
HISTORY SILLY
lose
SENSIBLE YEOWL
THE ENVELOP OPERATES
```


File: overlapgame.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Wed Oct 19 07:24:25 EDT 2005

RCS Info (may not be true date or author):

```
    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $
```

```
Chromatic Polynomials
--------- -----------

Given an undirected graph G (a set of vertices and
edges) let P(G,n) be the number of ways to color G with
n colors so no adjacent vertices have the same color.

If e is an edge of G between vertices x and y, then

    P(G,n) = number of ways to color G such that no 2
             adjacent vertices OTHER THAN x or y have
             the same color

           - number of ways to color G such that no 2
             adjacent vertices OTHER THAN x or y have
             the same color, AND x and y do have the
             same color

           = P(G',n) - P(G'',n)

where G' is the graph made from G by deleting edge e,
and G'' is the graph made from G by merging x and y,
deleting e, and deleting any duplicates of edges in
the resulting graph.  That is, if in G e1 is an edge
from x to z and e2 is an edge from y to z then in G''
x and y become the same vertex so e1 and e2 are now the
same edge and one of these must be deleted to avoid
duplicate edges.

G' and G'' both have fewer edges than G.  By repeating
this process you can reduce the problem to computing
the number of ways of coloring a graph with no edges
with n colors.  This is just n**d, where d is the number
of vertices in the graph with no edges.

Therefore P(G,n) is a polynomial in n of degree |G|,
where |G| is the number of vertices in G.  You are asked
to compute P(G,n) for various G.


Input
-----

For each of several test cases, a specification of a
graph G as follows:

   A line containing the number V of vertices.
   1 <= V <= 10.

   V lines each containing V binary digits
     ('0's and '1's).

Vertices are identified by integers i, 1 <= i <= V.
Lines of digits are numbered 1, 2, 3, from the first
line to the last line.  Digits in a line are numbered
1, 2, 3, from left to right.

For 1 <= i,j <= V, digit j of line i is '1' if vertex
i is adjacent to vertex j, and '0' otherwise.  Digit
j of line i equals digit i of line j, and digit i of
line i is '0' (a vertex is NOT adjacent to itself).

No lines contain any spaces.  The input terminates
with an end of file.

Output
------

For each case, a single line containing V+1 integers,
which are the coefficients of P(G,n) from high order
to low order.
```

```
Example Input
------- -----

3
000
000
000
3
011
101
110
5
01000
10100
01010
00101
00010
5
01001
10100
01010
00101
10010


Example Output
------- ------

1 0 0 0
1 -3 2 0
1 -4 6 -4 1 0
1 -5 10 -10 4 0
```

```
File:       chromatic.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Mon Oct 17 03:05:49 EDT 2005
```



```
RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $
```

Proof Labeling

The following is an example of a proof in a limited logic language with just the implication operator => and propositional variables denoted by single upper case letters.

```
z   ((A=>B)=>B)                         assumption
y   (B=>A)                              assumption
x   (A=>B)                              assumption
1   B                                   modus ponens z x
2   A                                   modus ponens y 1
3   ((A=>B)=>A)                         discharge x 2
a   (((A=>B)=>A)=>A)                    axiom
4   A                                   modus ponens a 3
5   ((B=>A)=>A)                         discharge y 4
6   (((A=>B)=>B)=>((B=>A)=>A))          discharge z 5
```

Here each line is an inference. Inferences are named by lower case letters or integers. After the name comes the logical formula. After that, the reason the formula is a valid inference (e.g., ‘axiom’ or ‘discharge z 5’).

Assumptions and axioms are named by lower case letters, while other inferences are named by integers. Different inferences have distinct names.

A logical formula is either an atom, denoted by a single upper case letter, or an implication, which has the form (F1=>F2), where F1 and F2 are any logical formulae.

There are four kinds of inferences.

Axioms. These are just named with a lower case letter. The only axiom needed by our limited logic language is Pierce’s axiom, which is any formula of the form (((F1=>F2)=>F1)=>F1). (F1=>F2) is used as a stand-in for ‘not F1’, since negation is not in our limited language, so Pierce’s axiom just says that if you can prove F1 from ‘not F1’, then F1 is true.

Assumptions. These are named by a lower case letter. They must be discharged. Avoiding the use of undischarged assumptions in a proof is a subtle point that will be elaborated on in the notes below.

Modus Ponens. This is the rule of logic that says given (F1=>F2) and F1 you can infer F2. A modus ponens inference is named by an integer. Let F2 be the logical formula of the inference. Then the reason of the inference must have the form ‘modus ponens N1 N2’ where N1 names a previous inference whose logical formula has the form (F1=>F2) for some logical formula F1, and N2 names a previous inference whose logical formula is F1.

Discharge. This is how you discharge assumptions. A discharge inference has an integer name and a logical formula of the form (F1=>F2). Its reason has the form ‘discharge N1 N2’ where N1 names a previous inference that is an assumption with logical formula F1, and N2 names a previous inference with logical formula F2.

Inferences in a proof can be given labels that completely describe how the inference was arrived at. When labels are added on a line after each inference in the above example, the example looks like this:

```
z   ((A=>B)=>B)                          assumption
    z
y   (B=>A)                               assumption
    y
x   (A=>B)                               assumption
    x
1   B                                    modus ponens z x
    (zx)
2   A                                    modus ponens y 1
    (y(zx))
3   ((A=>B)=>A)                          discharge x 2
    (\x.(y(zx)))
a   (((A=>B)=>A)=>A)                     axiom
    a
4   A                                    modus ponens a 3
    (a(\x.(y(zx))))
5   ((B=>A)=>A)                          discharge y 4
    (\y.(a(\x.(y(zx)))))
6   (((A=>B)=>B)=>((B=>A)=>A))           discharge z 5
    (\z.(\y.(a(\x.(y(zx))))))
```

Note that inference names and inference labels are different things, though for assumptions and axioms they happen to be equal. Labels are computed as follows:

Axiom. The label of an axiom inference is the name of the inference, a lower case letter.

Assumption. The label of an assumption inference is the name of the inference, a lower case letter.

Modus Ponens. The label of a ‘modus ponens N1 N2’ inference is (XY) where X is the label of inference N1 and Y is the label of inference N2.

Discharge. The label of a ‘discharge N1 N2’ inference is (\X.Y) where X is the label of the inference N1, and is always a lower case letter, as N1 is an assumption, while Y is the label of the inference N2.

We have told you everything you need to know to do this problem, but there is more interesting stuff in the notes at the end.

Input

A sequence of inferences, one per line, without any labels. Each inference consists of a name, a logical formula, and a reason. There are no spaces inside the logical formula. Any amount of whitespace may be used to separate the name, the logical formula, the reason, and the separate parts of the reason.

An end of file terminates the input.

No two inferences have the same name. Names are all lower case letters or integers in the range from 1 through 1000. No inference line is longer than 80 characters.

Output

For each inference print the exact input line containing the inference followed by one additional line containing the label of the inference indented by 4 spaces. The label must not contain spaces.

However, you must perform checks on modus ponens and discharge inferences. If the checks do not pass, you must output '$' as the label of the inference. This label may then propagate into the labels of other inferences that use the inference which did not check.

The check for a 'modus ponens N1 N2' inference with logical formula F2 is that inference N1 has a logical formula of the form (F1=>F2) and inference N2 has the logical formula F1.

The check for a 'discharge N1 N2' inference is that it has a logical formula of the form (F1=>F2), inference N1 is an assumption (check this) with logical formula F1, and inference N2 has logical formula F2.

The input data will be such that no label will be longer than 76 characters (so no output line will be longer than 80 characters). The N1 and N2 above will always name previous inferences (though not necessarily those that will pass the checks for formula or reason).

```
Example Input
------- -----

z   ((A=>B)=>C)                             assumption
y   ((B=>A)=>C)                             assumption
x   (C=>B)                                  assumption
w   (B=>A)                                  assumption
1   C                                       modus ponens y w
2   B                                       modus ponens x 1
3   ((B=>A)=>B)                             discharge w 2
a   (((B=>A)=>B)=>B)                        axiom
4   B                                       modus ponens a 3
v   A                                       assumption
5   (A=>B)                                  discharge v 4
6   C                                       modus ponens z 5
7   ((C=>B)=>C)                             discharge x 6
b   (((C=>B)=>C)=>C)                        axiom
8   C                                       modus ponens b 7
9   (((B=>A)=>C)=>C)                        discharge y 8
10  (((A=>B)=>C)=>(((B=>A)=>C)=>C))         discharge z 9
11  (A=>A)                                  discharge v v
12  C                                       modus ponens v 11
13  B                                       modus ponens x 12
14  (A=>B)                                  discharge v 13
15  ((((B=>A)=>B)=>B)=>B)                   discharge a 13


Example Output
------- ------

z   ((A=>B)=>C)                             assumption
    z
y   ((B=>A)=>C)                             assumption
    y
x   (C=>B)                                  assumption
    x
w   (B=>A)                                  assumption
    w
```

```
1   C                                      modus ponens y w
    (yw)
2   B                                      modus ponens x 1
    (x(yw))
3   ((B=>A)=>B)                            discharge w 2
    (\w.(x(yw)))
a   (((B=>A)=>B)=>B)                       axiom
    a
4   B                                      modus ponens a 3
    (a(\w.(x(yw))))
v   A                                      assumption
    v
5   (A=>B)                                 discharge v 4
    (\v.(a(\w.(x(yw)))))
6   C                                      modus ponens z 5
    (z(\v.(a(\w.(x(yw))))))
7   ((C=>B)=>C)                            discharge x 6
    (\x.(z(\v.(a(\w.(x(yw)))))))
b   (((C=>B)=>C)=>C)                       axiom
    b
8   C                                      modus ponens b 7
    (b(\x.(z(\v.(a(\w.(x(yw))))))))
9   (((B=>A)=>C)=>C)                       discharge y 8
    (\y.(b(\x.(z(\v.(a(\w.(x(yw)))))))))
10  (((A=>B)=>C)=>(((B=>A)=>C)=>C))        discharge z 9
    (\z.(\y.(b(\x.(z(\v.(a(\w.(x(yw))))))))))
11  (A=>A)                                 discharge v v
    (\v.v)
12  C                                      modus ponens v 11
    $
13  B                                      modus ponens x 12
    (x$)
14  (A=>B)                                 discharge v 13
    (\v.(x$))
15  ((((B=>A)=>B)=>B)=>B)                  discharge a 13
    $
```

Notes

An assumption name X is discharged in a label if it only occurs inside subexpressions of the label that have the form (\X...). That is, the \X discharges all X's in the subexpression it begins.

Thus z is discharged in (a(\z.(xz))) but is not discharged in (z(\z.(xz))) as in the latter the first z is outside any (\z...)

In a valid proof of a theorem, all assumptions must be discharged. Notice we did NOT ask you to check this.

It is possible to prove the following:

If a formula F has a proof with label ((\X.Y)Z) then it has a proof with label Y[X=Z], which denotes the label Y with all undischarged X's in it replaced by Z, provided that Z has no undischarged assumption names that become discharged when Z is inserted into Y. Thus the label of a proof can be 'reduced' by the 'reduction rule' ((\X.Y)Z) --> Y[X=Z].

Inference labels as we have introduced them have exactly the same syntax as formula's in lambda calculus, where we have used the backslash \ in place of the Greek letter 'lambda'. Furthermore, the reduction rule we have just introduced for inference labels is exactly the main reduction rule for the lambda calculus, beta reduction, and our word 'discharged' corresponds exactly to the lambda calculus word 'bound'.

It can also be shown that the other reduction rules for the lambda calculus, alpha reduction and eta reduction, are valid for inference labels. Thus there is an exact 1-1 correspondence between inference labels and lambda calculus. This is called the Curry-Howard Isomorphism.

It further turns out that the relation between the logical formula of an inference and the label of the inference is exactly the same as the relation between the type of a lambda calculus formula and the formula. Thus logical formula can be read as the types of the labels of their proofs.

File: prooflabel.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Tue Oct 18 11:33:00 EDT 2005

RCS Info (may not be true date or author):

```
    $Author: walton $
    $Date: 2005/10/24 07:36:22 $
    $RCSfile: problems-bospre2005.ps,v $
    $Revision: 1.1 $
```