1988 ACM Scholastic Programming Contest

Problem A: Air Traffic Control

A service offered by the FAA (Francine's Airplane Alarms) uses radar to tell airplane pilots when they are in danger of passing too closely to another airplane in flight. For this problem you are to write a program to simulate this radar.

To simplify things, you may assume that the airspace under consideration is above a planar Earth, centered on an origin with coordinates (0, 0). The positive y-axis lies due north, the positive x-axis to the east. All coordinates are in miles, all airspeeds in miles per hour, and all altitudes in feet (one mile = 5280 feet). You may also assume that airspeed equals ground speed (meaning no wind).

The input to the program will consist of up to twenty lines, one line for each airplane in the airspace. Each line contains numbers, separated by one or more blank spaces, giving the following values in this order:

- flight number (integer)
- altitude (real)
- flight heading (real, in degrees, 0 = north, 90 = east, etc.)
- position of the airplane at the start of the simulation (two real numbers, the x-coordinate and then the y-coordinate)
- airspeed (real)

You may assume that all planes maintain constant airspeed, altitude, and heading throughout the simulation.

A *near miss* is defined to occur when two planes are less than or equal to five miles from each other, measured only horizontally (ignore altitude in computing this distance), *and* simultaneously differ in altitude by less than or equal to 1000 feet. Your program should detect all near misses between any two airplanes. For each near miss, you should print the time (in seconds since the beginning of the simulation) when you detected the near miss, and the flight numbers, positions (coordinates), and altitudes of the two planes involved.

Your radar antenna rotates once very 20 seconds, so it is possible that two planes would be within the near miss limits for just a few seconds while your antenna is pointing the other way. *This is acceptable!* The data set provided was designed to have only near misses that continue for at least 20 seconds, so it doesn't matter which way your antenna is pointed when the simulation begins. Note also that the time you report for a near miss does *not* have the be the exact second of the beginning of the near miss; it might be as much as 20 seconds later and still be correct. Similarly, the location you report may be anywhere within the near miss zone, depending on when you detect the incident.

Each near miss should be reported only once. It is possible for the two planes still to be within the near miss limits 20 seconds later when your radar sees them again. Do not report the same two planes again.

Your simulation should end after three hours.

1988 ACM Scholastic Programming Contest

Problem B: Candlepin Bowling Scoring

Candlepin bowling, although using the same number of pins (10) and same length and width lanes as Ten Pin bowling, differs in that the pins are shaped differently (wider in the middle and tapered to a relatively narrow top and bottom), the balls are much smaller (about the size of a grapefruit), and it is much more difficult to attain high scores than it is in Ten Pin bowling (championship class Ten Pin bowlers often average around 210 per game, whereas championship class Candlepin bowlers usually average around 130 per game). This latter difference might be a particularly surprising one to the uninitiated, given that a Candlepin bowler is allowed three balls per frame, whereas in Ten Pin bowling only two balls per frame are allowed.

For this problem, you are to produce a program that will compute the score of Candlepin bowling games. Each frame consists of a maximum of three balls. Frames are scored according to the following rules:

1. If the first ball thrown in a frame knocks down all ten pins (called a strike and marked as an X), then the score for the frame is equal to 10 plus the number of pins knocked down by the next two balls thrown in the following frame(s). When a strike is thrown, no other balls are thrown in that frame, since all the pins have already been knocked down (see #4 below for final frame exception).
2. If all ten pins are not knocked down by the first ball, but are all knocked down by the combination of first and second balls thrown in a frame (called a spare and marked as a /), then the score for the frame is 10 plus the number of pins knocked down by the next ball thrown (that is, the first ball thrown in the next frame). When a spare is made, no third ball is thrown in the frame, since all the pins have already been knocked down (see #4 below for final frame exception).
3. If all ten pins are not knocked down as a result of the first bali (a strike) or first and second balls (a spare), a third ball is thrown, and the score for the frame is simply the number of pins knocked down by all three balls.
4. Since the 10th frame is the final frame, it is impossible to use pin counts from subsequent frames to complete strikes or spares rolled in the 10th frame. Therefore, the 10th frame always consists of 3 balls thrown, wherein:
   a. if the first ball of the frame is a strike, the pins are reset (ten new pins are set up, as at the beginning of each frame) and the bowler throws two more balls; in the case of another strike, the pins are reset again and the bowler throws one more ball at them; the score for the frame is 10 plus the sum of the pins knocked down by the following two balls;
   b. if the first ball is not a strike, but the combination of the first two balls is a spare, the pins are reset and the bowler throws one ball at them; the score for the frame is 10 plus the number of pins knocked down by the third ball;
   c. if neither a strike nor a spare is rolled in the 10th frame, the score for the frame is simply the number of pins knocked down by all three balls.

Your program is to read from a data file the results of each ball thrown in an arbitrary number of bowling games and print the score for each game. Each data line will represent the results of each ball thrown for a single game. Data items representing a game's results of balls thrown are in even-numbered columns, separated by blanks in odd-numbered columns. A data line with ** in columns 1 and 2 indicates the end of the data file.

Valid data items, representing the results of the balls thrown in a game are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, /, and X. (Note that this programs accepts data as it would be entered on a bowling scoresheet; that is, if the first ball in a frame were a strike, an X, not 10, would be marked; if the first and second balls were 8 and 2 for a spare, 8 / would be marked, not 8 2.) One slight variation will be employed, however, so that a single character will be capable of representing any possible scoring circumstance, that being an A will be the scoring of a 10 count on the third ball of a frame in which the first two balls had 0 pin counts. (Note that if the first ball of a frame had a 0 pin count, and all ten pins are knocked down with the second ball, 0 /, indicating a spare, is scored.)

A sample of a data line representing a valid game would be:

```
X 8 / 7 2 1 3 5 0 6 / 7 / 5 2 2 X 9 / X X 8
```

Translating into frames, and scoring would yield:

```
 X     8 /     7 2 1    3 5 0    6 /    7 /    5 2 2    X     9 /     X X 8

20     17      10        8       17     15       9     20     20       28
```

or a cumulative scoring of:

```
20     37      47       55       72     87      96     116    136     164
```

Assuming that the first data line to be processed was represented by the sample above, and that there were n games scored, your output should be presented as follows:

```
GAME #1:   164
GAME #2:   xxx
   .
   .
   .
GAME #3:   xxx

END OF SCORING
```

You may assume that each line to be scored is a valid combination of entries.

1988 ACM Scholastic Programming Contest

Problem C: Monkey Business

The syntax of the *monkey* language is quite simple, yet only monkeys can speak it without making mistakes. The alphabet of the language is {a, b, c, d, #} where # represents 1 space. The complete grammar of the monkey language is:

```
<sentence>   ::= <word> | <sentence> # <word>
<word>       ::= <syllable> | <syllable> <word> <syllable>
<syllable>   ::= <plosive> | <plosive> <stop> | a <plosive> | a <stop>
<plosive>    ::= c <stop> a
<stop>       ::= b | d
```

As director of monkey intelligence, you have just been given a machine-readable version of a transcript of a conversation among several important monkeys. There is at least one and possibly more spies in this group of monkeys and you have been assigned the task of determining which monkeys are spies and which are not. You are going to do this by detecting which monkeys are not speaking proper monkey language.

The lines of dialogue from the monkey conversation will be in the input data file. Each line of dialogue will have the name of the monkey speaker (from 1 to 15 characters long), a colon (':'), and then the sentence spoken by that monkey. Each line will have the monkey name start in column 1. There will be no blanks (spaces) on a line except possibly in the spoken sentence. There will be at most 60 characters in a monkey's sentence. Each monkey's sentence will terminate with a non-blank character (thus no input line has trailing blanks). There will be at most 25 unique monkeys in the conversation, and, of course, each monkey may speak more than once. The end of the transcription will be designated by end of file.

Your output should contain a list of the "spies" in their order of appearance in the conversation, as well as the first sentence they spoke that gave them away. After you list the spies, then list the "ok" monkeys in their order of appearance in the conversation. The format of your output is unimportant as long as the two lists, the names, and the sentences needed are clearly identified.

1988 ACM Scholastic Programming Contest

Problem D: Multicomputer Processor Status

Assume that you are the system administrator for an MIMD (multiple instruction, multiple data) multicomputer composed of 50 processing elements. Each processor (node) within the architecture is limited to executing a single job (i.e. no multiprocessing) but may spawn tasks for neighboring processors to solve.

Support software is practically nonexistent on this system. However, a node may be queried to determine the particular job it is executing and whether or not the job was spawned from another node. Your assignment is to write a simple utility program that will display the current status of the system, indicating what jobs are running and the nodes on which they are executing. Any tasks spawned by the job should also be indicated.

The input to your program will consist of several lines, each containing a pair of integer values and a character string. There will be a single blank separating the first integer from the second, and a single blank separating the second integer from the character string. The first integer value indicates which node is responding to the system status query. The second integer value indicates which node (if any) spawned the task currently executing on the node indicated by the first integer value. Parent nodes and idle nodes will respond with a value of 0 for the second integer value. The character string (with a maximum length of 35 characters and a minimum length of 1 character) is the name of the executing job. The name will be blank on responses from spawned nodes and will contain at least one but no more than 35 spaces. You may assume all input is correct. Nodes will respond in a random order.

The output from your program will consist of two single spaced lines for each job currently executing in the system, followed by one blank line. Jobs should be printed in increasing order of root node numbers. The first line will be the name of the job and the second line will be the hierarchy of nodes currently used by the job. The root node should be printed first.

If a root node spawned any children, then the root node should be followed by an arrow --> (two hyphens and a greater than sign) and the children spawned should be enclosed within braces { } and separated by semicolons. Child nodes should be printed in increasing order among other children at the same level. If a child node spawned any children, then its children should be printed in the same manner. Exact spacing of output is *not* required.

For example, given the following system configuration of executing jobs and spawned tasks, your program must produce the desired results as indicated from the specified input. (Note that the exact order in which each nodes' status is received is undetermined.)

```
       Job 1:                      Job 2:                     Job 3:
Particle and Cell Sim     Flux Corrected Transport             IDLE

          1                          5                          12
         / \                       / / \ \
        3   4                     7 2  11 6
       /   / \
      8   9   10
```

Input

```
9 4
2 5
12 0 IDLE
3 1
7 5
10 4
1 0 Particle and Cell Sim
8 3
6 5
5 0 Flux Corrected Transport
4 1
11 5
```

Desired Output

```
Particle and Cell Sim
1 --> { 3 --> { 8 }; 4 --> { 9; 10 }

Flux Corrected Transport
5 --> { 2; 6; 7; 11 }

IDLE
12
```

Problem E: Pluviometrics

After the bottom dropped out of the hand-blown bud vase market, your employer purchased 100,000 of them for a pittance, intending to add a line of designer rain gauges to its gegaw catalog. After finding out that each vase has a shape uniquely its own, they handed you the problem of how to place the scale marks on them.

The marks on the rain gauge should be at the correct locations to indicate 0.1 inch, 0.2 inch, etc. Each 0.1 inch represents a volume of rainfall equal to 0.1 inch times the area of the opening at the top of the gauge.

The vases all have a single bulge at the base and flare toward the lip end. Since they were spun during blowing, their insides are surfaces of revolution about an axis perpendicular to a flat base. A tedious empirical study (in which, of course, least squares gave you fits) produced the result that the interior surfaces of the bud vases can be described closely enough by a function of four parameters that are easily determined for each vase. The parameters are:

h	the height in inches measured from the inner surface of the bottom
d	the diameter in inches of that round bottom
a, b	($0 < a < 1, 0 < b$) parameters that describe the shape of the vase through the function $r(x)$, which gives the radius of the vase x inches above the inner bottom:

$$r(x) = \frac{d}{2}\sqrt{1 + a\sin\frac{2\pi x}{h} + b\frac{x}{h}}$$

Values of these four parameters will be determined for each vase and placed in a single record in the order h, d, a, b as fixed point quantities separated by at least one blank. For each such record, the required program must print these parameters (suitably identified) and a table showing the height (in inches above the inside base) at which each tenth of an inch gradation in rainfall amount must be marked. The calculated height must be given to the nearest hundredth of an inch. Meniscus effects and refraction may be ignored. To avoid confusing the mark makers, no height exceeding h may appear.

Possibly helpful facts:

$\pi = 3.14159265$

Volume of a cylinder of radius r and height h is $\pi r^2 h$

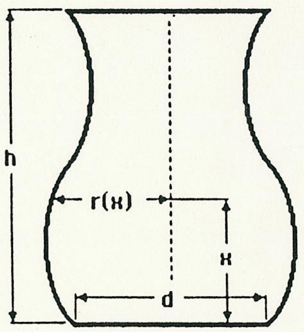
$$\int \sin t\, dt = -\cos t + c$$

Example:

```
h = 3.375    d = 0.715    a = 0.520    b = 0.133

  Rain      Line
  0.1       0.11
  0.2       0.21
  0.3       0.30
  0.4       0.38
  0.5       0.46
  0.6       0.54
  0.7       0.62
  0.8       0.69
  0.9       0.77
  1.0       0.84
  1.1       0.91
```

etc.



Problem F: Pretty Printing

The declaration section of many procedural programming languages is easier to read if the information is nicely formatted into columns, but it is difficult for a programmer to enter the program in this format. For this problem you are to write a program that will format a specific kind of declarations

The input data will consist of up to 50 lines, each of which contains (in this order):

- an identifier (the name of a variable)
- a colon character (:)
- a type specification, which may be a single identifier or an expression such as `array (1..10) of integer`
- a semicolon character (;)
- an optional comment (not all lines will have comments), which begins with two consecutive hyphens (--) and continues to the end of the line

Blank spaces may or may not be present between any two of the above items. Blank spaces may or may not be present between the comment delimiter (--) and the first word of the comment. Identifiers may contain uppercase or lowercase alphabetic characters, numerals, and the underscore character (_). A type specification will not contain a semicolon or a comment delimeter.

The output from your program should be the same declarations, formatted as follows:

- All variable name identifiers begin in column 3 of an output line.
- The colons are all aligned vertically in the second column after the longest variable identifiers.
- All types begin in the second column after the colons and are formatted exactly as given in the input.
- All semicolons follow immediately after the types, without intervening blank spaces.
- All comment delimiters (--) are aligned vertically in the third and fourth columns after the semicolon after the longest type specification.
- If the reformatting causes a comment to extend beyond column 80 on the line, the comment must be broken into more than one line. The line breaks cannot occur in the middle of words unless a single word is too long for the allotted space (in which case the line break my appear anywhere in that word). Each additional comment line must have its own delimiter (--) aligned with all the others. No printed line can go beyond 80 columns.
- All comment delimiters are followed by exactly one blank space and then the first (or next) word of the comment. The only exception is a comment that is totally empty (comment delimiter followed by no other characters).

For example, below are shown a declaration section before and after pretty printing.

```
 code : codeblock;
ident:type2;--here is a nice long comment to cause some continuation lines here.
 lines : integer;
 linenum : integer;--line number counter for loop
                                                       maxidentlen : integer;
 dummy :                                                                  integer;
list   :    array [ 1..100 ] of real   ; --this is a big list of things
```

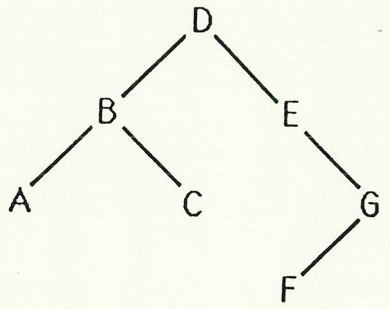
```
  code        : codeblock;
  ident       : type2;                 -- here is a nice long comment to
                                       -- cause some continuation lines
                                       -- here.

  lines       : integer;
  linenum     : integer;               -- line number counter for loop
  maxidentlen : integer;
  dummy       : integer;
  list        : array [ 1..100 ] of real;  -- this is a big list of things
```

1988 ACM Scholastic Programming Contest

Problem G: Tree Recovery

Three standard traversals of binary trees are preorder, inorder, and postorder. Assuming all the nodes of a binary tree contain unique data, an inorder traversal and a postorder traversal completely determine the shape and information in a tree. For example, an inorder traversal of ABCDEFG and a postorder traversal of ACBFGED correspond to the tree below.



Write a program that has as input the inorder and postorder traversals of binary trees and produces the preorder traversals of the trees. Each node in the tree contains a single printable nonblank character. A traversal can therefore be represented as a character string (as in the first paragraph above).

Input for your program consists of a text file organized into pairs of lines. The first line of each pair contains the postorder traversal of a tree and the second line contains the inorder traversal of the same tree. You may assume that there are an even number of lines in the file and that each line contains no blank or nonprintable characters. You may also assume that the pairs of lines are consistent, i.e. given the *n*th line, where *n* is odd, the *n+1*st line contains the inorder traversal of the tree whose postorder traversal is on the preceding line.

Output should consist of an echo of the input as well as the preorder traversal of each tree. All output must be well identified and easy to read.

For example, the tree shown above would be represented in the data as:

```
ACBFGED
ABCDEFG
```

For this data, your program should print:

```
Postorder traversal:
        ACBFGED
Inorder traversal:
        ABCDEFG
Preorder traversal:
        DBACEGF
```

1988 ACM Scholastic Programming Contest

Problem H: Window Manager

A *window manager* is a set of routines that control the appearance of multiple windows on a display screen. For this problem you will simulate a few of the capabilities of a window manager. The specifications for the simulation are described below.

The screen and the windows are measured in characters rather than pixels. The screen is 80 characters wide and 24 characters high, with rows numbered 1 through 24 from the top and columns numbered 1 through 80 from the left.

A window is simulated by putting its code character into every character position of the window.

Windows may overlap each other. Only those parts of a window not obscured by a higher window (meaning one conceptually in front of another) will show the code character. It is possible for a window to be completely hidden by one or more other windows.

When a new window is created, it is the highest or frontmost window, so it is completely visible.

When an existing window is deleted, any parts of windows previously obscured by that window become visible again.

When the user clicks the mouse at any visible point in a window, that entire window is moved to the highest (frontmost) level. Other windows stay in their same order relative to each other.

When the user gives a *cycle* command, the bottom or backmost window is moved to the top (frontmost), even if that window was previously totally obscured. Other windows stay in their same order relative to each other.

The simulation begins with no windows, so the entire simulated screen is blank. However, for purposes of this simulation, when the screen is displayed, print a period character (".") for each blank on the screen. This will make it easier to see where the windows actually are relative to the whole screen.

Your program should read commands from the input file and perform the requested operation. There are four kinds of commands:

1. *New Window Command*: This command contains a w in the first character position on the input line, followed by four integers representing, respectively, the column of the upper left corner of the new window, the row of the upper left corner, the number of characters in a row of the new window, the number of characters in a column. Following these integers is one or more blanks and then a non-blank code character, which is the character to display for this window. Upon receipt of this command, your program should create the new window.
2. *Mouse Click Command*: This command contains a m in the first character position on the input line, followed by the column and row, respectively, of the mouse position at the time of the click. Upon receipt of this command, your program should perform the action described above, provided that the mouse position is within a window.
3. *Delete Window Command*: This command contains a d in the first character position, then one or more blanks, then a non-blank character that is the code character of the window to be deleted. Upon receipt of this command, your program should delete the window, as described above.
4. *Cycle Windows Command*: This command contains a c in the first character position. Upon receipt of this command, your program should cycle the windows as described above.

Each command, including its various parameters, constitutes one line of the input data. At least one blank separates each two adjacent parameters. The data will contain only valid commands. It will not try to put two windows with the same code character on the screen at the same time. It will not give invalid locations or sizes for new windows. It will not give invalid coordinates for a mouse click. It will not ask you to delete a nonexistent window.

Your program should display the entire screen *only once*, after all commands have been executed.