RMC Western Montana High School Programming Contest

Morning Session, March 16, 2013

Self Intersecting Path

The “Snake” video game is an early classic game where a player drives a snake around the screen eating food to survive. The challenge of the game is that as the snake eats more it grows and grows and it becomes more and more difficult to navigate the snake around the screen without having the snake’s head run into a different part of its body.

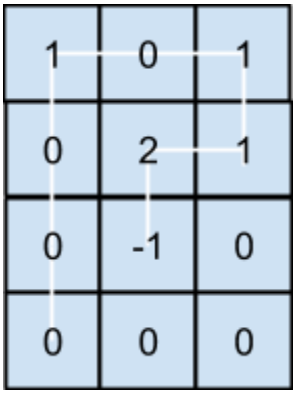
At the end of the day, the crucial algorithm is one that examines a path and determines whether that path is self intersecting or not. Your job is to write such a program. Each path is made up of a finite number of steps. There are three different kinds of steps: one where the path goes one unit forward, one where the path turns 90 degrees to the right and then goes one unit forward, and one where the path turns 90 degrees to the left and then goes one unit forward. Note: turn first, then move.

Your input will be a sequence of integers that ends in -1. Each input specifies the direction of the next step using the following key:

0 = go forward
1 = turn right
2 = turn left

(note “turn around” would cause instant death, so it’s not an option). After reading in the sequence, your program needs to either print out “path self-intersects” or “path doesn’t self-intersect”.

Examples:



Input: 0 0 0 1 0 1 1 2 -1
Output: “path doesn’t self-intersect”

The picture above shows what the path would look like if it started in the bottom left corner of the board with an initial direction heading north. **note: the problem only asks whether the path intersects itself or not; the starting direction and starting location are irrelevant to answering this question.**

On the other hand, you can see that the path is quite close to intersecting itself in this picture and if it takes a few more steps, it will in fact self-intersect.

Input: 0 0 0 1 0 1 1 2 1 0 0 0
Output: “path self-intersects”

Downton Abbey's Seating Chart

One of the more complicated challenges for the head of a major house at the start of the 20th Century was the amazingly complex rules of precedence: every individual had a defined role in society and it was important to make sure that each individual was acknowledged at exactly the right level. However, those rules tended to rank individuals relative to others in a similar field (for example, elevating bishops above ministers above deacons) instead of between fields (for example, ambiguities between the relative importance of a minister versus a physician).

To simplify this problem, we plan to use our new time travel technology to send a computer back to Downton to help Lord Grantham test his seating arrangements to make sure that they are socially acceptable, thus helping him avoid this veritable pit of despair. Your job is to write the program that we will preload on the computer (since the internet after World War I was really really slow). Your program will be given the following information: First the number of precedence pairs will be given. Secondly each precedence pair will be given as *name1 name2* with a single space in between the two names. Finally, a proposed seating arrangement will be given. All names in the input will be a single word long. Your program must then print out any precedence rules that are violated, or "it is acceptable" if there are no rules violated.

Example data set 1:

input:

```
3
Abby Bob
Bob Charlie
Abby Doug
Charlie Doug Bob George Abby
```

output:

```
error: Abby must come before Bob.
error: Bob must come before Charlie.
error: Abby must come before Doug.
```

Example data set 2:

input:

```
4
Abby Bob
Bob Charlie
Abby Doug
George Doug
Earl Abby Bob Doug Charlie
```

output:

```
it is acceptable
```

Evaluating Simple Java Expressions

The task in this problem is to evaluate a sequence of simple Java expressions, but you really don't even need to know Java to do this. Each of the expressions will appear on a line by itself and will contain no more than 80 characters. The expressions to be evaluated will contain only simple integer variables and a limited set of operators. There are 26 different variables that could appear, namely those with the names 'a' through 'z'. 'a' is initialized to 1, 'b' to 2 and so on up to 'z' which is initialized to 26. Each variable is separated from its neighboring operator by exactly one space.

The operators that may appear in expressions include the binary operators +, -, and * with the usual interpretation. That is, multiplication must be done before addition or subtraction, and once all the multiplications have been done, the additions and subtractions are evaluated from left to right. Thus the expression "a + b * c - d + b" is equal to "1 + 2 * 3 - 4 + 2" which is "1 + 6 - 4 + 2" = "7 - 4 + 2" = "3 + 2" = "5"

Your program must read in a single expression and print out its value.

Example 1:

input: `a + b * c - d + b`
output: `5`

Example 2:

input: `f - c * b * a + f * e`
output: `30`

Triple LCM

A fundamental step in adding and subtracting fractions is finding the least common multiple. Doing this by hand is difficult for some children (and some adults). You will write a program that finds the least common multiple of three positive integers. The least common multiple of three integers **a**, **b**, and **c** is the smallest positive integer that is divisible by all three of them.

The first line of your input will be a positive integer, **n**, indicating the number of problem sets. Each problem set consists of three positive integers separated by exactly one space. There are no blank lines in the input. For each problem set, print the least common multiple of the three positive integers. Print each least common multiple on a separate line.

Example 1:

input:

```
4
15 21 35
33 22 11
9 10 7
1 1 1
```

output:

```
105
66
630
1
```

Diamond Days

Your job is to draw a large character art diamond using * and space characters. The input to your program will be a single integer (larger than 2) which tells you how many *s are on each side of the diamond.

Example 1:

input: 3

output:

```
   *
  * *
 *   *
  * *
   *
```

Example 2:

input: 5

output:

```
     *
    * *
   *   *
  *     *
 *       *
  *     *
   *   *
    * *
     *
```

Cracking the Caesar

The Caesar cypher was a method of encoding text that was used by Julius Caesar. In his time, it was considered state of the art and in fact was a major military advantage that the Roman troops had over their adversaries. The cypher was simple. First, the encoder and decoder needed to agree on a shift which was a number between 1 and 25. Then the cypher took a "*plaintext*" message and replaced each letter of the alphabet with the letter "shift" places after it in the alphabet (wrapping around if necessary). This encoded string is called a *cyphertext*. To decode, the process is reversed, replacing each letter of the cyphertext with the letter that comes "shift" places before it in the alphabet (wrapping around if necessary). Non-letter characters (spaces, punctuation, etc.) are left untouched. So for example, a message of "hi there!!!" becomes "jk vjgtg!!!"

Nowadays, computers can easily break Caesar cyphers. Probably the easiest way for a computer to do so is to try each of the 25 possible shifts in turn, checking each potential solution to see if the "decoded" words are all in the dictionary. Your program will do the same.

Your program will first be given a positive integer n that corresponds to the number of words in the dictionary. Next, the n words are given each on a line by themselves. Finally, the last line given to your program will be the cyphertext. Your program should print out the decoded plaintext.

For simplicity, all letters that appear in this problem are guaranteed to be lower case letters. Also, the data is guaranteed to be decodable by the provided dictionary in exactly one way.

Input 1:

```
5
cat
dog
fleas
has
my
pb, pb grj kdv iohdv!!!
```

Output 1:

```
my, my dog has fleas!!!
```

Identifying Legal Pascal Real Constraints

The computer language Pascal was the first language that was taught to programming students for most of the 1980s and 1990s. Students who have learned to program more recently may find some of its syntax a bit strange because it is most directly descended from the language ALGOL instead of many currently popular languages which borrow most of their syntax from the language C.

One such strangeness is in the definition of constants. Pascal requires that real constants have either a decimal point, or an exponent (starting with the letter e or E, and officially called a scale factor), or both, in addition to the usual collection of decimal digits. If a decimal point is included it must have at least one decimal digit on each side of it. As expected, a sign (+ or -) may precede the entire number, or the exponent, or both. Exponents may not include fractional digits. Blanks may precede or follow the real constant, but they may not be embedded within it. Note that the Pascal syntax rules for real constants make no assumptions about the range of real values, and neither does this problem.

Your task in this problem is to identify legal Pascal real constants. Each line of the input data contains a candidate which you are to classify. For each line of the input, display your finding as illustrated in the example shown below. The input terminates with a line that contains only an asterisk in column one.

Example input:
6.02E23
+3E-4
-32.143
3.
+3.2E--5
.125E4
3
6.0.2 E23.1
*
Example output:
valid
valid
valid
invalid
invalid
invalid
invalid
invalid

Note #4 is invalid because it doesn't have a digit to the right of the decimal place, #5 is invalid because it has two - signs in the scale factor, #6 is invalid because there's no digit to the left of the decimal place, #7 is invalid because it has neither a decimal place nor a scale factor, and #8 is invalid because

there are two decimal points in the left portion, and also because there is a decimal place in the scale factor.