

Prolog Programming Contest 9 December 2003 ICLP'03

Prologue

There are 5 problems. The name of the file that contains the solution, is given to you in the header of the problem. On submitting a solution, this file must be readable for the organisers. So, for the first problem, you make a file named `stop.pl`.

The *input* to your program will be in the form of Prolog facts, or as an argument to the predicate you must write.

Efficiency of your programs is not important (except in the `ld` problem), but if your program fails to finish in a reasonable time, it will be considered incorrect. You can earn a bonus for problem `ld`; this means that in case of a tie between two (or more) teams, the speed of the solution to `ld` is decisive.

Do not start predicate names with `iclp03` - do not use the dynamic database!

1 Stop. (*stop.pl*)

This is the traditional *write-something-to-the-screen* problem, which every team should be able to get right, so that you all have something to be proud off :-)

This time, you will output to the screen some figure that looks like a stop sign (depending on which country you live in): it is an 8-sided board, which is all red except for a white horizontal band in the middle. On the screen, a size 3 stop sign on the screen looks like the left part of the picture below:

```

  RRR
R  R
R  R
R WWW R
R  R
R  R
  RRR

          RRRRR
          R  R
          R  R
          R  R
          R  R
          R  R
          R WWWWWWWW R
          R  R
          R  R
          R  R
          R  R
          R  R
          R  R
          RRRRR
```

You see that only the contour is red (the capital letter R), and the horizontal white band is indicated by the capital letter W.

What you write is a predicate `stop/1`, which will be called with an odd number ($N > 1$) and which produces the stop sign of size N . So, as another example `?- stop(5)`. produced the right part of the picture above.

2 The cheater. (*cheater.pl*)

N tourists have booked a day tour which includes ONE trip in a gondola. There is only one gondola and it operates all day. These tourists can board any time the gondola is at the only departure spot and after the trip, everyone must leave the gondola. Each tourist gets a sticker on his/her shirt, so that the gondola driver can see immediately whether someone is entitled to board the gondola. The gondola driver is payed by the number of people who have actually take the trip, so he counts how many people enter his gondola and at the end of the day reports that number to the tourist office, but ... at the end of the day, the gondola driver reports that $(N + 1)$ tourists have taken the trip and while he expects to be payed for $(N + 1)$ tourists, the tourist office is decided to find out who cheated and took the gondola trip twice. The only thing they can rely on, is the memory of the N tourists themselves: every (non-cheating) tourist tells exactly which tourists (s)he saw on the same gondola trip. The cheating tourist knows this of course, so he mentions the union of the people he saw on both boat trips. The result is a sequence of Prolog facts `saw/2` (which should be interpreted as their commutative closure) - see later for some examples.

Because the tourist office forsees that it will have to deal often with this problem, it lets you - the poor programmer - write a predicate `cheater/1` which unifies its argument with the name of the cheater. There are two more things you know:

```
there is at most one cheater and the cheater cheated just once
the gondola does not make trips with just one tourist
maybe it is the gondola driver who cheated ...
```

The latter means that your predicate `cheater/1` might also have to unify its argument with `gondola_driver` !

Here are some sample `saw/2` sets and the answer that goes with them:

```
saw(anna,beata).          saw(anna,beata).
saw(anna,christa).       saw(anna,christa).
saw(beata,christa).      ?- cheater(C).
saw(donna,eva).          C = anna
?- cheater(C).
C = gondola_driver
```

3 Spanning spider. (*spanspid.pl*)

A spanning spider of a graph is a spanning tree (a subgraph that is a tree and contains all vertices) that is also a spider: a spider is a graph with at most one vertex whose degree is 3 or more. Not every graph has a spanning spider and you will write a predicate `spanspid/0` which succeeds once if a given graph has a spanning spider and finitely fails otherwise. The graph is given as a predicate consisting of facts `edge/2` which have as arguments nodes that are connected by an edge. We are dealing with undirected graphs here. Three example `edge/2` sets with the query and answer are shown below:

<code>edge(d,a).</code>	<code>edge(a,e).</code>	<code>edge(a,b).</code>
<code>edge(d,b).</code>	<code>edge(b,e).</code>	<code>edge(b,c).</code>
<code>edge(d,c).</code>	<code>edge(e,f).</code>	<code>edge(a,c).</code>
<code>edge(d,e).</code>	<code>edge(f,d).</code>	<code>edge(d,a).</code>
<code>edge(a,x).</code>	<code>edge(f,c).</code>	<code>edge(d,b).</code>
<code>edge(b,y).</code>		<code>edge(d,c).</code>
<code>edge(c,z).</code>		
<code>?- spanspid.</code>	<code>?- spanspid.</code>	<code>?- spanspid.</code>
Yes	No	Yes

4 Longest decreasing subsequence. (*ld.pl*)

The Erdős-Szekeres theorem says:

Every sequence of $n * m + 1$ different real numbers contains a decreasing subsequence of length $m + 1$ or an increasing subsequence of length $n + 1$

Some proofs start by “*Take a **longest decreasing subsequence** ...*” and this leads to the following problem (for you): write a predicate `ld/2` which given a sequence of different numbers (as first argument), determines a longest decreasing subsequence, by unifying such a subsequence with the second argument. Sequences are represented by lists, so a typical query might be:

```
?- ld([3,6,7,4,5,1,2],L).  
with answer  
L = [7,5,2]
```

Also, `[6,4,1]` is a correct answer (and some others).

The input list is never empty.

You can earn a bonus for a faster program, but remember: it is usually better to hand in quickly a slow solution than slowly a quick solution !

5 Celauton. (*celauton.pl*)

An elementary cellular automaton (e-celauton) consists of an infinite array (both directions) of cells each having one of two possible colors - black or white - and rules that describe how one array is transformed into another one; the new color of a cell depends only on its old color and the old color of its two neighbors. We will use o for white and x for black for simplicity. Such e-celautons are described in details for instance in the book by S. Wolfram (*A new kind of science*).

We will assume that (initially) there are only a finite number of black cells (*). The rules of an e-celauton can be described by pictures of the kind:

```
    o x o          o x x
      o              x
```

meaning that a black cell with two white neighbors turns white and a black cell with a left white neighbor and a black right neighbors, remains black.

We will always assume that the e-celauton has the **bleak** rule:

```
   o o o
     o
```

The computing power of e-celautons is amazing (well, that is, if you amaze easily, e.g. by universal Turing machines) and starting from an initial array of black and white cells (obeying *) and because we have the bleak rule, we can watch the evolution of an e-celauton, as in the next example:

```
   o x o
  o x o x o
o x o x o x o
```

The first line - or generation - contains one black cell only and because of (*) and the bleak rule, we don't need to represent the infinite white parts at the left and the right. You see three generations in total.

One natural problem in this context is *the cellular automaton inverse problem*: given a sequence of generations, decide whether there exists an e-celauton that is responsible for this sequence. That is the problem you will solve !

Because we start with a finite number of blacks and because of the presence of the bleak rule, we can represent a generation by a finite list of o's and x's. A sequence of generations will be represented by a list of generations. Just to make sure that it is clear how to fit two generations under each other, the generation at time $t(i+1)$ has two more cells explicitly represented than the generation at time $t(i)$: one cell at the left and one cell at the right - even if that means that there are redundant white cells at the sides. As an example:

$$\begin{aligned} & [\quad [o,x,o], \\ & \quad [o,x,o,x,o], \\ & [o,x,o,x,o,x,o]] \quad (1) \end{aligned}$$

is the representation of the sequence of generations above.
and can be generated by the rules

$$\begin{array}{cccc} \text{oxo} & \text{oox} & \text{xoo} & \text{xox} \\ \text{o} & \text{x} & \text{x} & \text{x} \end{array}$$

plus some 12 other rules which are of no consequence for the evolution shown.
The representation of these 4 rules (+ the bleak rule) is the list

$$[[o,o,o,o],[o,x,o,o], [o,o,x,x], [x,o,o,x], [x,o,x,x]] \quad (2)$$

in which the order of the rules does not matter.

The input to the predicate you will write is a list as in (1). The output is a list of rules as in (2).

As an example:

```
?- celauton([[o,x,o],[o,x,o,x,o],[o,x,o,x,o,x,o]],Rules).
Rules = [[o,o,o,o],[o,x,o,o], [o,o,x,x], [x,o,o,x], [x,o,x,x]]
```

is correct - or with any other order in the Rules list.

The Rules list must be the minimal required set of rules needed to cause the given sequence of generations, or to put it another way, every e-celauton that causes this particular sequence of generations, must have these rules. It is possible that no e-celauton exists for the given sequence of generations: in that case, the query must fail. For instance:

```
?- celauton([ [o,x,o,x,o],
               [o,o,o,o,x,o,o] ],L).
```

fails.

(aspects of this problem are studied in more depth for cyclic, finite one-dimensional automata in "A cellular automaton inverse problem", PhD. thesis by Billie J. Rinaldi, August 2003, and with other restrictions by A. Adamatzky in one of his books)