18th Prolog Programming Contest
July 2011, Lexington

The contest consists of 5 challenges. Each correct submission (at most one for each challenge) earns you one point. Incorrect submissions cost nothing, except that you might be mentioned in the award speech.

Efficiency of your programs is usually not important, but if your program fails to finish in a reasonable time, it will be considered incorrect. However, the efficiency of your submission to challenge *war.pl* will decide in case of a draw.

Submit a file for each solution. The file name must be the same as given in the header of the challenge - this file must be readable for the organizers. For instance, for the first problem, you make a file named `chicken.pl`.

The file must not contain predicates whose names start with `iclp11` or `test` - do not use the dynamic database or other similar global stuff built-in predicates!

1 Chicken *(chicken.pl)*

Chickens are a $900 million industry in Kentucky. They are all over the place, even in the Prolog programming contest. Write a `chicken/1` predicate that draws `N` > 0 chickens on the screen for the goal `?-chicken(N).` Do not add extra spaces on the left!

Here are the results for `N` = 1, 2 and 3:

```
?- chicken(1).
   \\
   (o>
\\_//)
 \_/_)
___|__
```

```
?- chicken(2).
       //
      <o)
   \\  (\\_//  //
   (o> (_\_/  <o)
\\_//) __|___ (\\_//
 \_/_)|      |(_\_/
___|__|      |__|___
```

```
?- chicken(3).
                 \\
                 (o>
         \\  \\_//)  //
         (o>  \_/_) <o)
   \\  \\_//) ___|__ (\\_//  //
   (o>  \_/_)|      |(_\_/  <o)
\\_//) ___|__|      |__|___ (\\_//
 \_/_)|                    |(_\_/
___|__|                    |__|___
```

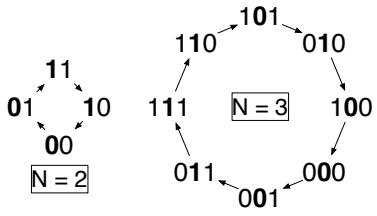
The top chicken leans right (left) with odd (even) `N`.

2 Bit Pattern *(pattern.pl)*

Take all the distinct bit patterns of length N and arrange them in a cycle such that each pattern overlaps with its two neighbours in $N-1$ bits. Then flatten the cycle into a Prolog list.

Write a predicate `pattern(`N`,`L`)` that returns a list of bits that comprises all the bit patterns of length $N > 0$ following the above rules. Note, there are multiple possible solutions. You need only produce one.

```
?- pattern(1, L).
L = [0,1]
?- pattern(2, L).
L = [0,0,1,1]
?- pattern(3, L).
L = [0,0,0,1,1,1,0,1]
```



3 The Leaper Race *(race.pl)*

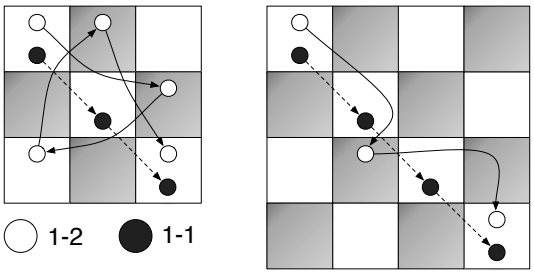
After midnight,[1] strange things happen on the board of chess. Kings, queens, knigths, ... have long gone to bed, and out come the *fairy* pieces, strange aberrations of the familiar pieces.

The knight has a whole family of fairy cousins: the *leapers*. Each leaper moves according to a particular vector. For instance, the knight himself is a `1-2` leaper. Some of his many cousins are the *fers* (`1-1`), the camel (`1-3`) and the zebra (`2-3`).

Leapers love nothing better than racing. They square off (actually it's sometimes a rectangle) part of a chess board, and race from one corner to the opposite. Obviously, most leapers cannot run in a straight line. Nevertheless, each leaper takes the least number of steps to reach the finish. Sometimes though, to spite its brethren, a fairy chooses a rectangle that none of the contestants can cross. That's fairies for you.

Write a predicate *race(Leapers,N,M,Winner)* that determines, on an $N \times M$ chess board, which of the *Leapers* reaches in the least number of steps square (N,M), starting in square (1,1). If none of them does, the predicate should fail.

```
?- race([1-1,1-2],3,3,Winner).
Winner = 1-1.
?- race([1-1,1-2],4,4,Winner).
Winner = 1-2.
```



[1] and many a glass of bourbon

4 Fast Food War *(war.pl)*

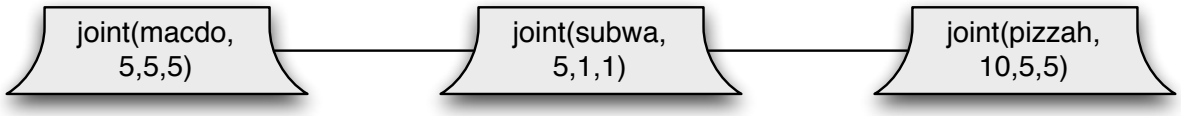
Colonel Sanders is going to war. Various fast food chains have taken over his beloved Kentucky, and fried chickens have had to make way for pizzas, subs and the almighty burger. This has come to an end, now. The Colonel is counting on your strategic insight to make up his battle plans.
Your mission, should you accept it, is to write a predicate

```
war(Joints,Streets,NbChickenWarriors)
```

that determines the *minimum* number of his (fried) chicken warriors needed to take over a city, with given list of fast food `Joints`, and `Streets` connecting them. Sanders' host of warriors will overrun the joints one by one, and traveling between them along the available streets.
A *fast food joint* is a term `joint(Name,Needed,Indigested,Defenders)`, where:

`Name` is a unique name,
`Needed` is the number of chicken warriors needed to overrun the joint and eat all the food,
`Indigested` is the number of warriors incapacitated after the raid, useless for further warfare,
`Defenders` is the number of warriors that need to be left behind in the joint to defend it against enemy troops.

Any remaining warriors after a raid that are not indigested or left behind to defend, can move on to conquer the next joint.
A *street* connects two joints, and is represented by a term `street(Name1,Name2)`. You can assume that the joints and streets form a connected graph.
In your plan, you are free to start at any joint, but you should travel at most once in either direction along a street to avoid ambushes.

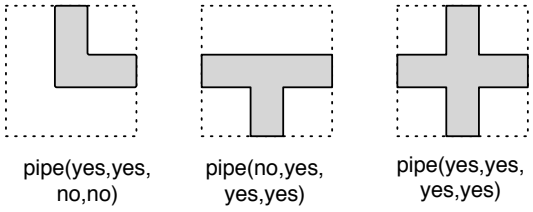



```
?- war([joint(macdo,5,5,5), joint(pizzah,10,5,5), joint(subwa,5,1,1)]
      ,[street(macdo,subwa),street(pizzah,subwa)], MinWarriors]).

MinWarriors = 22.
```

5 Joe the Plumber *(plumber.pl)*

You are Joe the plumber. Your politician friend is in a fix, and has called you for help. He's tried to fix his plumbing, but after a while discovered that some jobs are best left to professionals. Of course, now your job is harder than it was before he started. He's taken apart the piping, put back a few pieces and now has no clue whatsoever on how to put back the remaining pieces.
This is your job: put back the remaining pieces into the grid to create a closed piping system, respecting the pieces already present in the grid. (Fortunately, you have a knack for sudoku puzzles, and you realize politicians can afford twice your usual rate.)
Write a predicate `plumber(`*Grid*`,`*Pieces*`)`. *Grid* is a list of lists, representing a square grid of pipe pieces. Free slots are represented by free variables. *Pieces* is a list of available pieces to add to the grid.



A piece of piping is a term `pipe(`*Up*`,`*Right*`,`*Down*`,`*Left*`)`, Where each of the directions is either the atom `yes` or the atom `no`. Each of the directions indicates whether the pipe has an outgoing/incoming part in that direction or not.
Your predicate should unify the free slots in the grid with the available pieces of piping. Use each available piece at most once.

```
?- Grid = [[_,_],[pipe(no,yes,no,yes),_],
   Pieces = [pipe(yes,no,no,yes),pipe(no,yes,yes,no),pipe(no,no,yes,yes)],
   plumber(Grid,Pieces).

Grid = [[pipe(no,yes,yes,no),pipe(no,no,yes,yes)]
       ,[pipe(no,yes,no,yes),pipe(yes,no,no,yes)]].
```