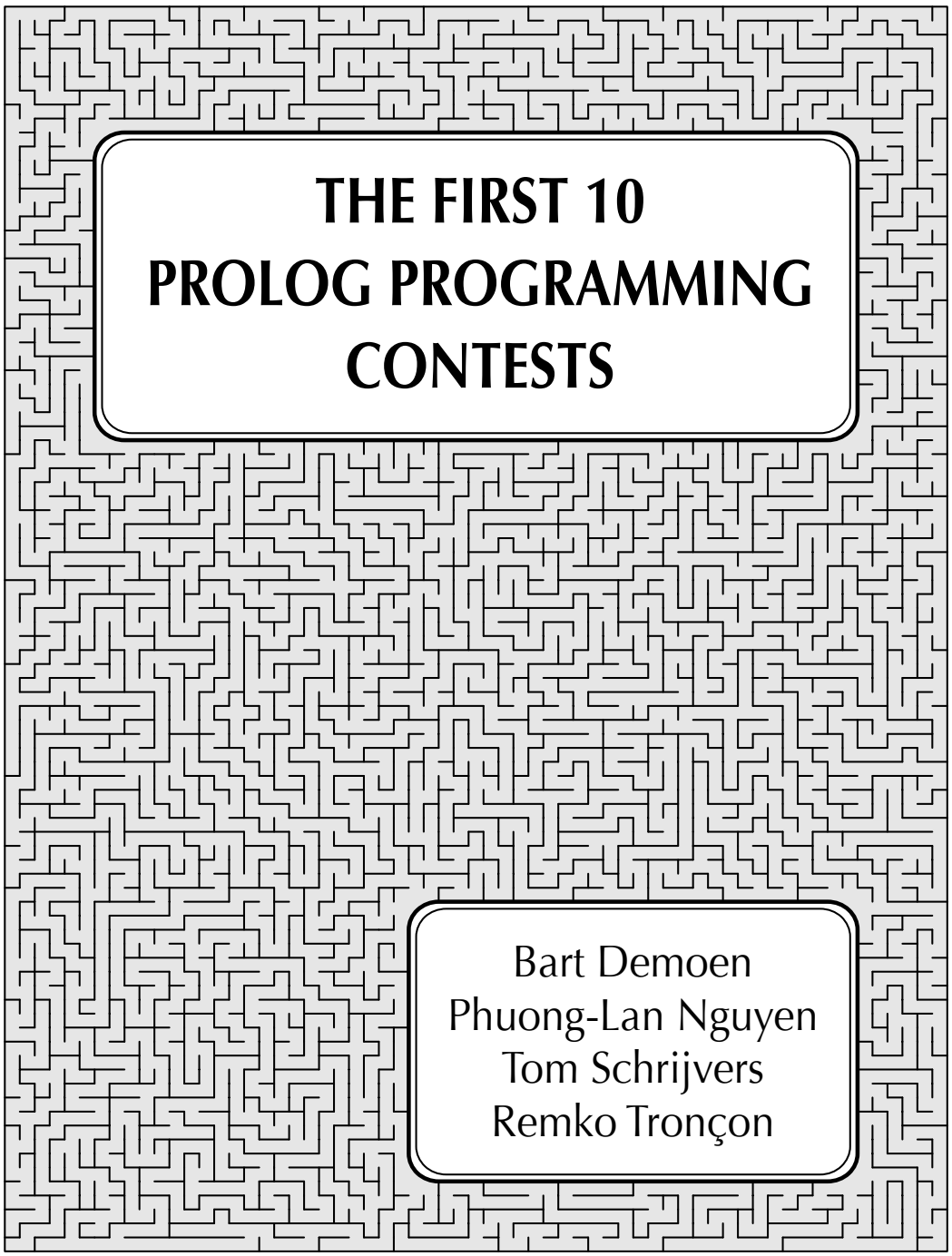

THE FIRST 10
PROLOG PROGRAMMING
CONTESTS
Bart Demoen
Phuong-Lan Nguyen
Tom Schrijvers
Remko Tronçon

The First 10 Prolog Programming Contests

Bart Demoen
K.U.Leuven, Belgium

Phuong-Lan Nguyen
UCO, Angers, France

Tom Schrijvers
K.U.Leuven, Belgium

Remko Tronçon
K.U.Leuven, Belgium

Editor Bart Demoen
Cover Design Remko Tronçon

ISBN-10: 9090197826
ISBN-13: 9789090197821

Cover Image: *Perfect Maze* by Remko Tronçon, generated with Prolog, rendered with METAPOST.

Printed in Belgium - 2005

About the authors

Bart Demoen (23-8-1953) has been involved in Prolog implementation since 1984 when he joined BIM (Everberg, Belgium), a software house with the mission to market its own Prolog system. Since 1987 he is at the Department of Computer Science, K.U.Leuven, Belgium where he became a professor in 1992. He enjoys research, teaching and implementation, and occasionally interferes with `comp.lang.prolog`. He co-designed and/or co-implemented BIM-Prolog, dProlog, ilProlog and hProlog, and contributed to XSB, HAL, SWI-Prolog and ISO Prolog.

Phuong-Lan Nguyen (7-10-1956) worked on Prolog during her Master's Thesis (1988) in Grenoble and ever since. She is now a *Professeur Associé* at the Institut de Mathématiques Appliquées in Angers, France, and heavily involved in teaching. She has published papers on type systems, efficient emulators and garbage collection ... all for Prolog.

Tom Schrijvers (10-6-1978) is currently a post-doc at the Department of Computer Science, K.U.Leuven, Belgium. The subject of his doctoral thesis was the implementation, analysis and optimization of Constraint Handling Rules and he continues to be active in this domain. Tom's favorite implementation language is Prolog and he has contributed several libraries to SWI-Prolog.

Remko Tronçon (21-1-1980) is currently in the final phase of his Ph.D. at the Department of Computer Science, K.U.Leuven, Belgium. He is working on the efficient implementation of query evaluation in the context of inductive learning and the Prolog system hipP. He is also the current maintainer of the `comp.lang.prolog` FAQ. As a member of the Jabber Software Foundation, he is also involved in the development of instant messaging systems.

Preface

Prolog is an excellent programming contest language: Prolog is close enough to the ultimate specification language (logic), so that the distance between problem and solution is not too big. This means that, even if you don't have a clue about a good algorithm, you might still end up with a nice program that computes a useful result (at least for small problem sizes). Prolog is also a real programming language, which means you can express your favorite nifty optimal solution strategy in it.

Programming is enormously fun. Fun was indeed our main motivation for starting the series of Prolog Programming Contests in 1994. Ask anybody who ever participated in a Prolog Programming Contest: they loved it! There is a thrill in trying to solve five problems in two hours, crammed in a small room filled with ten or more sweaty teams doing the same. And there is a great satisfaction, unmatched by any other daytime scheduled conference event, when the attempt is even partially successful.

This book shows solutions to problems that were in the first 10 Prolog Programming Contests. The solutions in this book were not constructed by participants during the contest, since the contest rules always prevented that. However, many of our solutions could have been constructed during the contest under extreme time pressure, and so you will find many solutions using the generate and test strategy, together with a higher than usual deployment of `member/2`, `append/3`, `findall/3` and even `reverse/2`. On the other hand, we have avoided dynamic predicates (except in the solution of two problems), and we have often preferred the Prolog if-then-else and `once/1` predicate over the use of the `!`. We have also avoided comments in the programs.

Mind you: this book does not attempt to teach you how to program in Prolog. For that, you will need to read one of the excellent books on Prolog, or go through one of the on-line Prolog tutorials. You can find them in the `comp.lang.prolog` FAQ[1]. Keep in mind that reading the books is not enough: you must do lots of exercises!

[1] Currently at `http://www.cs.kuleuven.be/~remko/prolog/faq/`

So we assume that you have already some basic Prolog programming skills. Then, how should you use this book? We suggest that you do not try to digest all questions and answers at once. Read one problem statement, skip the hints, and solve the problem, preferably with the clock ticking at your side. Oh well, if you really want, you can read the hints anyway. When you are finished programming, look at our solution, and compare it with yours. You might be inclined to make a judgement like 'My solution is better than yours', or the other way around. Don't just stop there: learn from the differences.

In case you are particularly proud of your solution, or you think your program is better than ours for some reason, please send it to us. We intend to make the book available electronically very soon, and your solution might find its place there - with proper credits to you of course. As an alternative, consider sending an elaborated version of your program to the Logic Programming Pearls section of the Journal of Theory and Practice of Logic Programming.

We have made sure that all the solutions in this book run as is in SWI-Prolog. This means that, together with the `lists` library and a small *contest library* (see Page 145), you always have a working program. Most solutions also run in Ciao, SICStus Prolog and YAP. When they don't, the reason is usually a small difference in the `lists` library, or a missing common predicate.

Some of the problems lend themselves better to being solved in a different LP language, such as CLP, CHR, XSB, ASP, ... Have a go at it and spread the word!

This book is meant to be sold out quickly, so only a small number of copies were printed. Soon you will be able to download the book for free. The web version of the book will contain all our programs in a form that you can directly consult, a set of test cases and, hopefully with your help, new solutions in Prolog and other logic languages. You can find this material and further contact information at `http://www.cs.kuleuven.be/~dtai/ppcbook/`.

Finally, before you dive into the problems, consider this quote from an honest participant after the first contest: *'I did my best to write a completely declarative program, but I now realize this was absolutely unnecessary'*. We hope you enjoy the book!

September 2005
Leuven, Belgium — Bart Demoen, Phuong-Lan Nguyen,
Angers, France — Tom Schrijvers, Remko Tronçon

The History of the Prolog Programming Contests

During the ILPS'93 program committee meeting in Philadelphia, it transpired that Leon Sterling and Bart Demoen had independently been contemplating the organization of a Prolog Programming Contest (PPC) during major Logic Programming conferences. So, we decided to go for it. ILPS'93 was too close, and the first upcoming opportunity seemed PAP'94 in London. Leon had the credibility with Al Roth (general chair of PAP'94) to have the first Prolog Programming Contest there.

The first set of problems was put together by Bart. Leon wanted to be involved, but unfortunately he was too busy. At that time there were two well-known programming contests: the ACM Programming Contest, and the Duke Internet Programming Contest. We looked at the problem statements of both contests, and found out they were geared at imperative programming languages: most of their problems were no fun to do in Prolog. Still, the first two PPCs contain problems that were adapted from the Internet Programming Contest. We are therefore grateful to Vivek Khera (and his colleagues) for making the IPC problem set publicly available. Our first set of problems was tried out at the Department of Computer Science at the K.U.Leuven by several members of the DTAI research group: Maurice Bruynooghe, Alain Callebaut, Marc Denecker, Veronique Dumortier, Gerda Janssens, André Mariën, Bern Martens, Anne Mulkers, Stef Renkens, and possibly others. We can't remember who won.

Back to PAP'94. Chris Moss reserved a computer lab at Imperial College for the contest. The lab was huge: never again did the contest take place in such a large lab! However, not a single contestant showed up ...

In the Ithaca, USA, ILPS'94 PPC, the conditions were perfect, and we now consider this to be the first PPC. The conference hotel was in the middle of nowhere, and the fact that the contestants needed to walk to a Cornell Uni computer lab was no problem. The lab was not too big, and that has always proven to be good

for the atmosphere during the contest. Anne Louise Gockel and Orlando Johnson made sure that the lab was in good order software wise, and Wiktor Marek gave the contest a good slot in the conference program. Evan Tick joined the organization effort at some point. Elsevier provided the price for the winning team: the special "Ten Years of Logic Programming" double-volume issue of the Journal of Logic Programming for each member of the winning team. SICS provided a SICStus Prolog license for the duration of the contest; SICStus Prolog remained the sole contest Prolog until 1999. We started the tradition of offering a Belgian *praline* to every team member as soon as the team handed in its first solution.

The second Prolog Programming Contest was during ILPS'95, and took place at a lab in Portland State University, Washington. Portland has a weather that inspires taking part in a contest, so there were more people than the year before. Evan Tick helped in the organization locally and also during the contest. Hannah Linder and John Jendro from PSU helped in the practical local issues. MIT Press provided books as prices.

The third Prolog Programming Contest took place in 1996 in Bad Honnef (near Bonn, Germany) during JICSLP'96. The University of Bonn, Rainer Manthey, and Lutz Plumer helped getting things off the ground locally, and especially Thomas Fuchs was wonderful in his system support during the contest when machines broke down and an expert was needed. One participant complained that *'This time, you had to actually read the questions before solving them!'*

Up to that point, the problems were concocted solely by Bart, and it was time to enlarge the permanent team. Phuong-Lan Nguyen joined, and from 1997 on, the problems were molded into their final shape by the duo. In 1997, during ICLP in Leuven, Belgium, the local support came from Jean Huens and his systems support group. Lee Naish and Gerda Janssens were instrumental in making space and time available in the program. From this year on, the prizes were sponsored by the conference, i.e. indirectly by the Association of Logic Programming. The prize was a box of Belgian chocolates and some JLP issues.

In 1998, JICSLP was in Manchester, England. Ian Pratt and Kung Kiu-Lau provided help locally. This contest is remembered by the participants because halfway, the fire alarm went off, and we had to clear the building for 15 minutes. The prize was a University of Manchester mug. Clearly, participating is more important than winning!

ICLP'99 was in Las Cruces, New Mexico, and was organized by Gopal Gupta. He and his colleagues Ivan Strnad, Sonja Mendoza and Enrico Pontelli provided local assistance. The problems could be solved in *any reasonable* LP language, and solutions were submitted in XSB, SWI-Prolog, B-Prolog, BinProlog, SICStus Prolog and even Oz. A novelty during this year's contest was that we provided the teams with a small test suite. On the other hand, we didn't provide any immediate

feedback on the submissions. This was probably what caused this contest to have no winner. The contest rules indeed stipulated that the winning team must submit at least three correct solutions, and this was not the case. This rule was relaxed in later years.

A millennium bug prevented a PPC in 2000.

ICLP2001 was organized in Paphos, Cyprus, by Tony Kakas. He gave the Prolog Programming Contest an excellent slot in the program, and a very nice room to hold it in. Maria Tsolakis installed several Prolog systems on a number of machines. It was the first time that the contest was not held on a set of interconnected machines. In fact, many teams used their laptop, so this contest is known to us as the first laptop contest. Solutions were handed in on a floppy. A choice of Prolog systems could be used: SWI-Prolog, SICStus, GNU, XSB, ECL$^{\text{i}}$PS$^{\text{e}}$ and Ciao. Everything went amazingly smooth. For the first time (and ever since), we also held a web version of the contest. It starts at the same time as the contest at the conference, has the same questions, but lasts 24 hours. From now on, the prize for the winners became a t-shirt with the conference logo and in capital letters the text *'Winner'*.

The eighth Prolog Programming Contest was at the occasion of ICLP2002 in Copenhagen, Denmark. Henning Christiansen made sure we had a good room and Jens Peter Secher was very helpful in making sure the machines were running well. SICStus Prolog was the contest Prolog.

Mumbay, India was the place of the ninth Prolog Programming Contest, during ICLP2003. R.K. Shyamasundar got us a room full of brand new machines through sponsoring by RedHat India. Anbu and Sachin provided technical support and installed the software. SWI-Prolog became the official contest Prolog: Jan Wielemaker helped installing it JIT! The questions were easier than ever: the results showed it, and the participants were happier than ever.

And finally, the tenth Prolog Programming Contest was in Saint-Malo, France, at the occasion of ICLP2004. Bart felt he couldn't combine organizing the contest with being a program co-chair, so two Ph.D. students working with him, Tom Schrijvers and Remko Tronçon, took over. Bart Demoen and Vladimir Lifschitz provided a time slot to hold the contest, while Mireille Ducassé made the local arrangements to make the contest possible. Before and during the contest, we could also count on Benjamin Sigonneau for support on site. This year's formula gave the contestants the possibility again to choose their favorite system from a specific selection: Ciao, GNU Prolog, SWI-Prolog, XSB, and YAP.

This book ends with the 2004 Prolog Programming Contest, but the contests go on: the one at ICLP2005 in Sitges, Spain is being prepared while this book is written. We wish you can participate, either at the conference or over the net!

The Prolog Programming Contest Hall of Fame

The true spirit of the Prolog Programming Contests is that the only losers in this contest are the people who did not participate. And there are winners of course, agile Prolog programmers whose name is worth remembering. Here is a complete list of the winners of the contests at the conferences:

ILPS 1994	Fergus Henderson, Peter Stuckey
ILPS 1995	Thomas Conway, Fergus Henderson, Peter Stuckey
JICSLP 1996	Slim Abdenader, Francesco Bugliesi, Thomas Frühwirth
ICLP 1997	Lee Naish, Peter Schachte, Peter Stuckey
JICSLP 1998	Tony Kusalik, Konstantinos Sagonas, David S. Warren
ICLP 2001	Henk Vandecasteele, Bert Van Nuffelen, Jan Wielemaker
ICLP 2002	Konstantinos Sagonas, Christian Schulte, Peter Stuckey
ICLP 2003	Vitaly Lagoon, Christian Schulte, Peter Stuckey
ICLP 2004	Bart Demoen, Konstantinos Sagonas, Peter Stuckey

The winners of the web contests also deserve eternal glory:

2001	Bernhard Pfahringer
2002	Warwick Harvey, Neil Yorke-Smith
2003	Bernhard Pfahringer
2004	Pierre Alexandre Favier, Jean Michel Leconte

Contents

Contest I: 1994 Ithaca, USA

1. Triplets

Write a predicate `triplets/1` which by backtracking unifies its argument with all triplets $[X, Y, Z]$ that satisfy the following two conditions: (1) X, Y and Z are different integers between 0 and 9 (both included) (2) $\frac{10*X+Y}{10*Y+Z} = \frac{X}{Z}$ with infinite precision.

E.g., suppose that [3, 5, 9] and [3, 1, 6] are the only solutions, then the output looks as follows:

```
?- triplets(Triplets).

Triplets = [3, 5, 9] ;

Triplets = [3, 1, 6] ;

No
```

The order of the solutions is not important.

Hints *This is a very small problem, so one can write it most quickly by a simple generate and test. Reordering the goals makes it faster, but who cares. The infinite precision condition is just there to slow you down, thinking for a while.*

Solution

```
:- use_module(contestlib,[for/3]).

triplets([X,Y,Z]) :-
        digit(X),
        digit(Y),
        digit(Z),
        Z > 0,
        X \== Y,
        X \== Z,
        Y \== Z,
        (10*X + Y)*Z =:= (10*Y + Z)*X.

digit(Digit) :- for(Digit,0,9).
```

2. Spiral

Write a predicate `spiral/2`, which will be called with two positive integers N and M as arguments. Such a call should draw a rectangle with height N and width M on the screen. This rectangle must contain the numbers from 1 to $N * M$ in a spiraling fashion. For example:

```
?- spiral(4,3).

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

The columns must be properly aligned to the right and occupy $Width+1$ positions, where $Width$ is the number of digits in the decimal representation of $N * M$.

Hints *It is easy to print out a matrix row by row. The tricky part is computing the matrix elements. In this case, the value of a matrix entry at position (I,J) is the distance (+1) from the entry (1,1) along the spiral.*

So, the solution is based on the predicate `distance(N,M,I,J,D)`*, which gives in a N×M rectangle the distance between point (1,1) and point (I,J) along the spiral. This distance is defined recursively, as illustrated by Figure 1.1.*

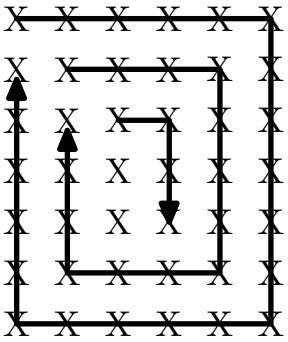


Figure 1.1: The recursive nature of distance.

The other issue in this problem is that you need to put just enough spaces between the matrix elements. We use `int_width/2` *to compute the number of decimal positions of a number, and use this number to insert the spaces. The predicate* `int_width/2` *is also used in some other solutions, so it is in the contest library.*

Solution

```
:- use_module(contestlib, [writeN/2, for/3, int_width/2, write_int/2]).

spiral(N,M) :-
        NM is N*M,
        int_width(NM,Width),
        Width1 is Width + 1,
        for(I,1,N),
              nl,
              for(J,1,M),
                    distance(N,M,I,J,Distance),
                    write_int(Distance,Width1),
        fail.
spiral(_,_).

distance(_,_,1,J,D) :- !, D is J - 1 + 1.
distance(_,M,I,M,D) :- !, D is M + I - 2 + 1.
distance(N,M,N,J,D) :- !, D is N + 2*M - J - 2 + 1.
distance(N,M,I,1,D) :- !, D is 2*N + 2*M - I - 3 + 1.
distance(N,M,I,J,D) :-
        N1 is N - 2,
        M1 is M - 2,
        I1 is I - 1,
        J1 is J - 1,
        distance(N1,M1,I1,J1,D1),
        D is 2*N + 2*M + D1 - 4.
```

3. Domino

Domino is played as follows: each participant gets a number of stones. Domino stones have two numbers of them, represented by two sets of dots, as depicted in Figure 1.2.

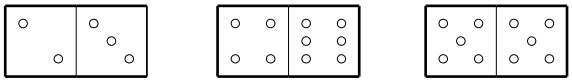


Figure 1.2: Some domino stones.

At the beginning of the game, a stone is put on the table, and each player in turn must either add one of his stones to the ones already on the table, or pass. The player who first gets rid of all his stones, wins. A stone can only be added if there is another stone on the table with a free side containing number N, and if the stone is added with its N-side touching a free N-side on the table. A stone with two different numbers has two free sides initially (one for each number). A stone with two equal numbers on it has three free sides. When a stone is put with a free side touching a free side of another stone, both free sides become occupied.

We will consider a variant called *domino solitaire*: you get a set of stones and you are to finish the game all by yourself. So, you start with one of your stones on the table (you choose which one), and continue adding stones until you have no more stones. This can fail of course.

Write a predicate `domino/1` that, given a set of stones in the form of `stone/2` facts, returns the order of the stones to be added incrementally to the table, provided that the rules of the game are satisfied at all times. Only one solution should be returned.

For example, given the facts

```
stone(2,2).
stone(4,6).
stone(1,2).
stone(2,4).
stone(6,2).
```

the following output would be correct:

```
?- domino(Stones).

Stones = [stone(1,2),stone(2,2),stone(2,4),stone(4,6),stone(6,2)]
```

Note that the solution list can represent more than one end configuration on the table, as can be seen in Figure 1.3.

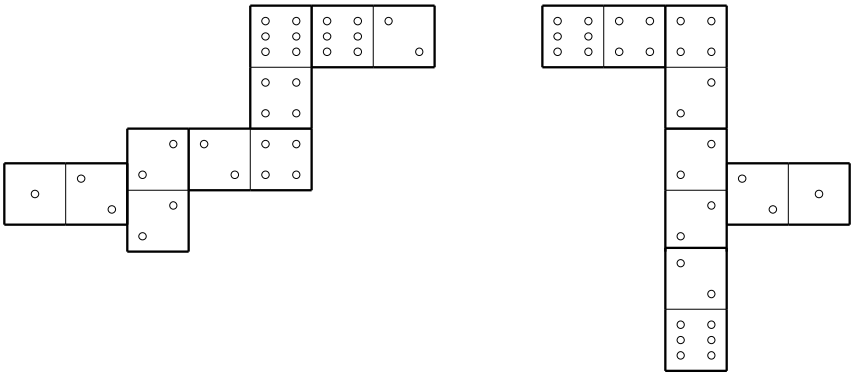


Figure 1.3: Two configurations resulting from the solution.

The query would for instance fail on the following set of stones:

```
stone(1,2).
stone(2,3).
stone(4,5).
stone(5,6).
```

Hints *When you play solitaire domino, it does not matter which stone you put on the table first: the end result might look different, but whether you can use up all stones does not depend on the first choice. So, the program starts by collecting all the stones in a list with* `findall/3`*, and the first element of this list is put on the table as the first stone. A list with free sides (i.e. where you can put new stones) is initialized and maintained throughout.*

Solution

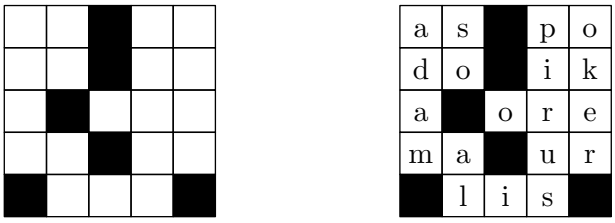
```
:- use_module(library(lists), [select/3]).

domino(Chain) :-
        findall(stone(X,Y),stone(X,Y),[FirstStone|RestStones]),
        Chain = [FirstStone|RestChain],
        init_freesides(FirstStone,FreeSides),
        once(chain(FreeSides,RestStones,RestChain)).

init_freesides(stone(A,A),FreeSides) :- !, FreeSides = [A,A,A].
init_freesides(stone(A,B),FreeSides) :- FreeSides = [A,B].

chain(        _,    [],           []).
chain(FreeSides,Stones,[Stone|Chain]) :-
        select(Stone,Stones,RestStones),
        add_stone(Stone,FreeSides,RestStones,Chain).

add_stone(stone(A,A),FreeSides,RestStones,Chain) :- !,
        once(select(A,FreeSides,RestFreeSides)),
        chain([A,A|RestFreeSides],RestStones,Chain).
add_stone(stone(A,B),FreeSides,RestStones,Chain) :-
        (
          once(select(A,FreeSides,RestFreeSides)),
          chain([B|RestFreeSides],RestStones,Chain)
        ;
          once(select(B,FreeSides,RestFreeSides)),
          chain([A|RestFreeSides],RestStones,Chain)
        ).
```

4. Crossword

You get a set of facts as shown below:

```
size(5).
black(1,3).
black(2,3).
black(3,2).
black(4,3).
black(5,1).
black(5,5).
```

These facts represent the empty crossword puzzle of Figure 1.4(a).



(a) Empty puzzle (b) Solved puzzle

Figure 1.4: Empty and solved puzzle.

You also get one fact `words/1` with a list of words, for instance:

```
words([do,ore,ma,lis,ur,as,po,so,pirus,oker,al,adam,ik]).
```

All words have at least two characters. Fill the puzzle with all the words exactly once. If that is impossible, fail. The solved puzzle is shown in Figure 1.4(b). Write a predicate `crossword/1`, which unifies its argument with a list of words. The order in this list indicates how these words can be used to fill out the puzzle. As an example:

```
?- crossword(Puzzle).

Puzzle = [as,po,do,ik,ore,ma,ur,lis,adam,so,al,pirus,oker]
```

i.e. first all horizontal words, row by row and in a row as they occur from left to right, and then all the vertical words, column by column, from high to low.

If there is more than one solution, your program should produce them all by backtracking. Every non-black square belongs to a word of two or more characters.

Hints *The idea behind our solution is this: we transform the list of words into a list of character sequences that represent the words, i.e. if the word list is [abc,de,fghi], the transformed list is: [[a,b,c],[d,e],[f,g,h,i]]. The predicate* `word2chars/2` *takes care of that. We also transform the empty crossword puzzle into a list of open spaces, where each open space represents a number of consecutive white squares as a list of variables: that is done by* `make_empty_words/1`*. Figure 1.5 should clarify that.*

A	B	■	C	D
E	F	■	G	H
I	■	J	K	L
M	N	■	O	P
■	Q	R	S	■

[[A,B], [C,D], [E,F], [G,H], [J,K,L], [M,N], [O,P], [Q,R,S], [A,E,I,M], [B,F], [N,Q], [C,G,J,O,S], [D,H,L,P]]

Figure 1.5: Left: the puzzle with a variable in each white square; right: the list of open spaces for words to fill in.

Finally, these two lists (characters and open spaces) are matched. You can obtain a very efficient program by carefully choosing the order in which these two lists match up.

This problem is discussed in Pascal Van Hentenryck, Mehmet Dincbas: Forward Checking in Logic Programming. Proceedings of ICLP'87, 229-256.

Solution

```
:- use_module(library(lists),[nth1/3, select/3]).

crossword(Puzzle) :-
        words(WordList),
        word2chars(WordList,CharsList),
        make_empty_words(EmptyWords),
        fill_in(CharsList,EmptyWords),
        word2chars(Puzzle,EmptyWords).

word2chars([],[]).
word2chars([Word|RestWords],[Chars|RestChars]) :-
        atom_chars(Word,Chars),
        word2chars(RestWords,RestChars).

fill_in([],[]).
fill_in([Word|RestWords],Puzzle) :-
        select(Word,Puzzle,RestPuzzle),
        fill_in(RestWords,RestPuzzle).

make_empty_words(EmptyWords) :-
        size(Size),
        make_puzzle(Size,Puzzle),
        findall(black(I,J),black(I,J),Blacks),
        fillblacks(Blacks,Puzzle),
        empty_words(Puzzle,EmptyWords).

make_puzzle(Size,Puzzle) :-
        length(Puzzle,Size),
        make_lines(Puzzle,Size).

make_lines([],_).
make_lines([L|Ls],Size) :-
        length(L,Size),
        make_lines(Ls,Size).

fillblacks([],_).
fillblacks([black(I,J)|Blacks],Puzzle) :-
        nth1(I,Puzzle,LineI),
        nth1(J,LineI,black),
        fillblacks(Blacks,Puzzle).
```

```
empty_words(Puzzle,EmptyWords) :-
        empty_words(Puzzle,EmptyWords,TailEmptyWords),
        size(Size),
        transpose(Size,Puzzle,[],TransposedPuzzle),
        empty_words(TransposedPuzzle,TailEmptyWords,[]).

empty_words([],Es,Es).
empty_words([L|Ls],Es,EsTail) :-
        empty_words_on_one_line(L,Es,Es1),
        empty_words(Ls,Es1,EsTail).

empty_words_on_one_line([],Tail,Tail).
empty_words_on_one_line([V1,V2|L],[[V1,V2|Vars]|R],Tail) :-
        var(V1), var(V2), !,
        more_empty(L,RestL,Vars),
        empty_words_on_one_line(RestL,R,Tail).
empty_words_on_one_line([_|RestL],R,Tail) :-
        empty_words_on_one_line(RestL,R,Tail).

more_empty([],[],[]).
more_empty([V|R],RestL,Vars) :-
        ( var(V) ->
            Vars = [V|RestVars],
            more_empty(R,RestL,RestVars)
        ;
            RestL = R,
            Vars = []
        ).

transpose(N,Puzzle,Acc,TransposedPuzzle) :-
        ( N == 0 ->
            TransposedPuzzle = Acc
        ;
            nth_elements(N,Puzzle,OneVert),
            M is N - 1,
            transpose(M,Puzzle,[OneVert|Acc],TransposedPuzzle)
        ).

nth_elements(_,[],[]).
nth_elements(N,[X|R],[NthX|S]) :-
        nth1(N,X,NthX),
        nth_elements(N,R,S).
```

5. Loops

You get a directed graph in the form of `arrow/2` facts as in:

```
arrow(a,b).
arrow(b,c).
arrow(c,c).
arrow(a,d).
arrow(d,a).
```

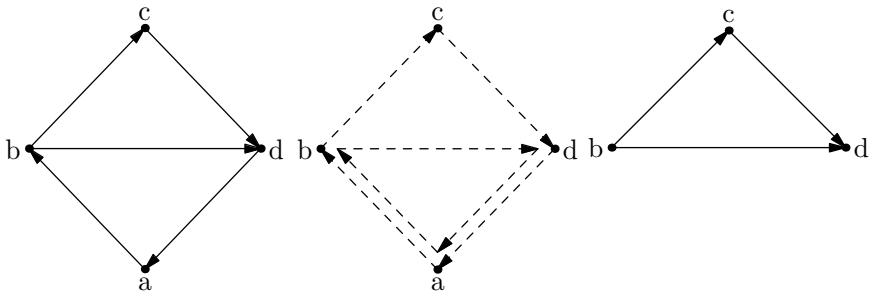
with obvious meaning. Write a predicate `loops/1` that unifies its argument with a (possibly empty) list of all the minimal cycles in the graph. A cycle is represented by a list containing all the nodes in the cycle in the order in which they are linked by the arrows, and starts and ends with the same node. A minimal cycle does not properly contain any other cycle. Every minimal cycle of the graph should be given exactly once. For the example, your program could give any of the following lists as solution:

```
[[c,c],[a,d,a]]
[[c,c],[d,a,d]]
[[a,d,a],[c,c]]
[[d,a,d],[c,c]]
```

The order in the list of cycles is irrelevant, and so is the actual start node of a cycle.

Hints *A minimal cycle is a cycle that does not contain the same node more than once. Therefore, to collect all the minimal cycles, we can pick any node and collect the minimal cycles starting at that node; then, we remove the selected node from the graph, and restart the procedure.*

Figure 1.6(a) shows a graph. Initially, node **a** *is picked, and its minimal cycles are constructed. There are two of them, and they are indicated in Figure 1.6(b). Then, the arrows that involve node* **a** *are removed, leading to the graph in Figure 1.6(c). At that point, the recursive call to* `findloops/3` *in the program is made.*



(a) Initial Graph

(b) Loops involving node **a**

(c) The graph after removing arrows from/to node **a**

Figure 1.6: The three basic steps in the program.

Solution

```
:- use_module(library(lists), [append/3, member/2]).

loops(Loops) :-
        findall(arrow(A,B),arrow(A,B),AllArrows),
        findloops(AllArrows,[],Loops) .

findloops([],    AllLoops,AllLoops).
findloops(Arrows,AccLoops,AllLoops) :-
        Arrows = [arrow(Start,_)|_],
        findall([Start|Path],
                path(Start,Start,Arrows,[],Path),LoopsFromStart),
        append(LoopsFromStart,AccLoops,NewAccLoops),
        delete_node(Arrows,Start,RestArrows),
        findloops(RestArrows,NewAccLoops,AllLoops) .

path(Start,Current,Arrows,Visited,Loop) :-
        member(arrow(Current,Next),Arrows),
        \+ member(Next,Visited),
        Loop = [Next|RestLoop],
        ( Next == Start ->
            RestLoop = []
        ;
            path(Start,Next,Arrows,[Next|Visited],RestLoop)
        ).

delete_node(   [],   _, []).
delete_node([A|As],Node,Arrows) :-
        ( (A = arrow(Node,_) ; A = arrow(_,Node)) ->
            delete_node(As,Node,Arrows)
        ;
            Arrows = [A|RestArrows],
            delete_node(As,Node,RestArrows)
        ).
```

6. Path

Consider a square board with a start point and a final point. The start point will always be $(1, 1)$. The final point is given by a fact like `goto(1,4,final)`, and the size of the board is given by a `size/1` fact. There are also facts for `goto/3`, as in the following example:

```
goto(1,1,up).
goto(1,3,right).
goto(3,3,down).
goto(3,2,left).
```

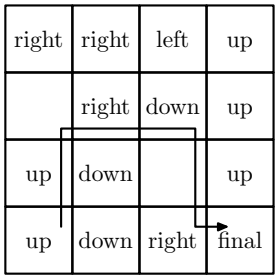
The above facts mean: when your position is $(1, 1)$, then you must go up; when your position is $(1, 3)$, go right; ... By following these `goto/3` instructions, it is easy to construct a path from the start point to the final point. Too easy in fact.

Luckily, a mean demon has removed some information from the board: it has taken away some `goto/3` facts, such that the path is no longer completely represented. Still, it was not that mean, since no two neighboring squares in the path have their information removed, neither is the final point, and at most two neighbors of a removed square belong to the correct path (diagonally touching squares are not considered neighbors). It has become a bit more difficult, but you can still easily find the path.

However, a second, much meaner demon has introduced information for every square *not* on the path and in such a way that, if you follow this information, you will either run around in circles or fall off the board (in particular, you will not meet emptied squares). Figure 1.7(a) shows a mutilated board:

right	right	left	up
	right	down	up
up	down		up
up	down	right	final

(a) Given board



(b) Same board with path

Figure 1.7: An example.

That board corresponds to the set of facts:

```
goto(1,4,right). goto(2,4,right). goto(3,4,left).  goto(4,4,up).
                 goto(2,3,right). goto(3,3,down).  goto(4,3,up).
goto(1,2,up).    goto(2,2,down).                   goto(4,2,up).
goto(1,1,up).    goto(2,1,down).  goto(3,1,right). goto(4,1,final).

size(4).
```

Write a predicate `path/1` which delivers the correct path as a list, starting with (1,1), ending in the final point, and ordered along the path. So, in the above example, we would get:

```
?- path(X).

X = [(1,1),(1,2),(1,3),(2,3),(3,3),(3,2),(3,1),(4,1)]
```

That path is shown in Figure 1.7(b).

Hints *Compared to the usual path search in a maze, there are two snags: the information* `goto/3` *is missing for some points, and some points lead you in a bad direction (one that leads to a loop or off the board). One doesn't really need to distinguish the two possibilities as long as one always checks for loops and stays on the board. The only thing to take care of is that, when there is no* `goto/3` *fact, all four directions are tried.*

Solution

```
:- use_module(library(lists),[member/2]).

path([StartPoint|RestPath]) :-
        StartPoint = (1,1),
        Visited = [StartPoint],
        complete_path(1,1,Visited,RestPath).

complete_path(I,J,_,ResultPath) :-
        goto(I,J,final),
        !,
        ResultPath = [].
complete_path(I,J,Visited,ResultPath) :-
        ( goto(I,J,Direction) ->
            true
        ;
            true
        ),
        next_square(Direction,I,J,NewI,NewJ),
        \+ bad_move(NewI,NewJ,Visited),
        ResultPath = [(NewI,NewJ)|TailResultPath],
        complete_path(NewI,NewJ,[(NewI,NewJ)|Visited],TailResultPath).

bad_move(I,J,Visited) :- member((I,J),Visited).
bad_move(I,J,      _) :- (I < 1 ; J < 1).
bad_move(I,J,      _) :- size(N), (I > N ; J > N).

next_square(   up, I,J,    I,NewJ) :- NewJ is J + 1.
next_square( down, I,J,    I,NewJ) :- NewJ is J - 1.
next_square( left, I,J, NewI,   J) :- NewI is I - 1.
next_square(right, I,J, NewI,   J) :- NewI is I + 1.
```

Contest II: 1995 Portland, USA

1. Triangle

Write a predicate `triangle/1`, which is called with its argument N instantiated to a non-negative integer, and which draws a triangle of size N on the screen. For example, a triangle of size 5 looks like this:

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

Note that there is a space between every two stars on a horizontal line. Between the top of a triangle of size N and the left side of the screen, there should not be more than (N+2) spaces.

Hints *In the following picture, the significant leading spaces are indicated by the symbol ␣. Observe that*

```
␣␣␣␣␣␣␣*              ␣␣␣␣␣␣␣      *
␣␣␣␣␣␣␣* *            ␣␣␣␣␣␣␣      * *
␣␣␣␣␣␣␣* * *          ␣␣␣␣␣␣␣  +   * * *
␣␣␣␣␣* * * *          ␣␣␣␣␣␣␣      * * * *
␣␣␣␣* * * * *         ␣␣␣␣␣␣␣      * * * * *
␣␣␣* * * * * *        ␣␣␣␣␣␣␣      * * * * * *
```

So, each line in the picture is a conjunction of a decreasing number of spaces, an increasing number of stars, and a newline.

Solution

```
:- use_module(contestlib, [writeN/2]).

triangle(N) :-
        Stars = 1,
        triangle(N,Stars).
triangle(_).

triangle(Spaces,Stars) :-
        Spaces > 0,
        writeN(Spaces,' '),
        writeN(Stars,'* '),
        nl,
        Spaces1 is Spaces - 1,
        Stars1 is Stars + 1,
        triangle(Spaces1,Stars1).
```

2. Cycle

Write a predicate `cycle/3` that is to be called with positive (non-zero) integers as the first two arguments and a free third argument. Such a call must succeed exactly once and unify the third argument with a list of integers (from 0 to 9) that represents the decimal cycle you get when dividing the first argument by the second. Some examples:

```
?- cycle(3,4,C) .

C = [0]             % since 3/4 = 0.250000...

?- cycle(4,3,C) .

C = [3]             % since 4/3 = 1.333...

?- cycle(1,7,C) .

C = [1,4,2,8,5,7]   % since 1/7 = 0.142857142857...
```

In the last example, C = [4,2,8,5,7,1] (and any other rotation of it) is also a correct answer.

Hints *Our solution builds up a list of subsequent remainders and quotients (in the form Remainder $*$ Quotient). As soon as the same remainder shows up, the corresponding Quotients form the repeating fraction.*

Solution

```
cycle(Dividend,Divisor,Repetition) :-
        Remainder is Dividend mod Divisor,
        RemQuotList = [],
        divide(Remainder,Divisor,RemQuotList,Repetition).

divide(Remainder,_,RemQuotList,Repetition) :-
        find_repetition(RemQuotList,Remainder,[],Repetition),
        !.
divide(Remainder,Divisor,RemQuotList,Repetition) :-
        Remainder10 is 10 * Remainder,
        Quot is Remainder10 // Divisor,
        NewRemainder is Remainder10 - Divisor*Quot,
        divide(NewRemainder,Divisor,
               [Remainder*Quot|RemQuotList],Repetition).

find_repetition([X*Quot|RemQuotList],Remainder,In,Out) :-
        ( Remainder == X ->
            Out = [Quot|In]
        ;
            find_repetition(RemQuotList,Remainder,[Quot|In],Out)
        ).
```

3. Powers

Write a predicate `powers/3`, which is called with as first argument a list of strictly positive integers, as second argument a strictly positive integer N, and a free third argument. Such a call must succeed exactly once and unify the third argument with the list that contains the smallest N integers (in ascending order) that are a positive (non-zero) power of one of the elements of the first argument. Some examples:

```
?- powers([3,5,4],17,Powers) .

Powers = [3,4,5,9,16,25,27,64,81,125,243,256,625,729,1024,2187,3125]


?- powers([2,9999999,9999999],20,Powers) .

Powers = [2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
          65536,131072,262144,524288,1048576]


?- powers([4,2],6,Powers) .

Powers = [2,4,8,16,32,64]
```

Hints *This is a variant of the Hamming problem. We can solve it by simply generating the first N powers of all the given numbers, collect them with* `setof/3`*, and then take the smallest N. However, just as for the Hamming problem, we can do better.*

Suppose the query is `?- powers([3,2,5],7,Powers)`*. We start by constructing the list of pairs [(2,2),(3,3),(5,5)], which is sorted and which has no two pairs with the same first component. In a pair (P,F), P is the smallest power of F that is not in the solution list yet. So, the first component of the first element of the pair-list is the next element in the output we are constructing. We remove this pair, compute the next power of F (i.e. P∗F) and insert the pair (P∗F,F) into the pair-list, respecting the invariants.*

You should think about why `remove_power/3` *can actually remove pairs without taking the F-component of the pairs into account. This is crucial to not generating duplicates in the output, but it can also reduce the pair-list, as you can check if you follow the execution of the query* `?- powers([2,4],6,Powers)`*.*

Solution

```
powers(Factors,N,Powers) :-
        sort(Factors,SFactors),
        pair(SFactors,Pairs),
        first_powers(N,Pairs,Powers).

pair([],[]).
pair([X|R],[(X,X)|S]) :- pair(R,S).

first_powers(N,[(Power,Factor)|PFs],[Power|Powers]) :-
        ( N == 1 ->
            Powers = []
        ;
            N1 is N - 1,
            remove_power(Power,PFs,PFs1),
            Power1 is Power * Factor,
            sorted_insert(PFs1,(Power1,Factor),PFs2),
            first_powers(N1,PFs2,Powers)
        ).

remove_power(Power,PFsIn,PFsOut) :-
        ( PFsIn = [(Power,_)|RestPFsIn] ->
            remove_power(Power,RestPFsIn,PFsOut)
        ;
            PFsOut = PFsIn
        ).

sorted_insert([],X,[X]).
sorted_insert([A|R],X,Out) :-
        ( A @< X ->
            Out = [A|RestOut],
            sorted_insert(R,X,RestOut)
        ;
            Out = [X,A|R]
        ).
```

4. Exit

Consider a maze with the following properties: every location in the maze is characterized by two coordinates (X,Y), both of which are integers; the maze has one entry (by convention on position (1,1)), and one exit. You guessed it: you will have to find the coordinates of the exit. You can interrogate the maze by means of two predicates: `move/2` and `at_exit/0`. The goal `at_exit` succeeds if and only if your current position is the one of the exit. The action of the goal `move/2` is a bit more complicated: you call it with ground arguments representing a position (e.g. `?- move(3,4)`); if the maze is such that you can go in one step from your current position to the position (3,4), the call succeeds and the move is made, i.e. your current position becomes the location (3,4). If you cannot go in one step from your current position to (3,4) the goal fails, and you are thrown back to the entry. In other words: whenever you try to make an impossible step, your current position becomes (1,1) again. The same happens if you call `move/2` with free or otherwise *bad* arguments. To limit the possibilities: a direct step is only possible to an adjacent position (X or Y coordinates differ by one).

Write a predicate `exit/1`, which, when called with a free argument unifies that argument with the coordinates of the exit of the maze. The size and shape of the maze is not given (`move/2` fails if it is called with a non-existing position, and throws you back to the entry).

Hints *There are three quirks in this problem: (1) you cannot really test your solution, unless you simulate a maze; (2) you have no idea how large the maze is, neither do you know its global shape; and (3), if a move fails, you cannot just backtrack over the attempt, because you are thrown back to the entry of the maze! It is essential that this failure to move is remembered, and that a new walk starting from (1,1) is initiated with the additional information. The temptation to use `assert/1` might feel irresistible, but we can do perfectly without it. Our program keeps track of the walls in the maze, and also of the places visited in the most recent walk starting from the origin. Note how the failure of a move (and being thrown back to the entry) does not mean we have to backtrack in the Prolog program.*

Also note how a move must be undone after the recursive call to `walk/4` has failed: upon backtracking, we indeed need to retrace our steps, otherwise we might attempt new moves from a position we are not at.

Solution

```
:- use_module(library(lists), [member/2]).

exit(Exit) :-
        Walls = [],
        BeenTheres = [],
        StartPoint = (1,1),
        once(walk(StartPoint,Walls,BeenTheres,Exit)).

walk(Point,    _,         _,Point) :- at_exit.
walk(Point,Walls,BeenTheres,Exit) :-
        next_point(Point,NextPoint),
        \+ member(NextPoint,BeenTheres),
        Wall = wall(Point,NextPoint),
        \+ member(Wall,Walls),
        ( move(NextPoint) ->
            (
              walk(NextPoint,Walls,[NextPoint|BeenTheres],Exit)
            ;
              move(Point),
              fail
            )
        ;
            walk((1,1),[Wall|Walls],[],Exit)
        ).

move((I,J)) :- move(I,J).

next_point((I,J),(NextI,NextJ)) :- NextI = I, NextJ is J + 1.
next_point((I,J),(NextI,NextJ)) :- NextI = I, NextJ is J - 1.
next_point((I,J),(NextI,NextJ)) :- NextJ = J, NextI is I + 1.
next_point((I,J),(NextI,NextJ)) :- NextJ = J, NextI is I - 1.
```

5. Palindrome Prolog Program

A Prolog program (PP) can be executed under the usual Prolog strategy, and also under the *reverse Prolog strategy*. The reverse Prolog strategy selects clauses in reverse lexical order and literals in a clause from right to left. We will only consider PPs without cut, negation, if-then-else and disjunction in the body. A PP is a palindrome with respect to a query `?- run(X).` if this query computes exactly the same solutions (order, multiplicity) when executed under the Prolog strategy as under the reverse Prolog strategy. You must write a predicate `ppp/0` that, for a given PP, decides whether the PP is palindrome (a PPP) for the goal `run(X)`. The PP is given as a set of facts of the form `pp/2`, whose first argument is the head of a clause, and whose second argument is the body of a clause (in the form that Prolog's `clause/2` would have returned it if the PP was in the dynamic database). Your predicate `ppp/0` succeeds if and only if the facts `pp/2` represent a PPP with respect to the query.

```
pp(run(1),true).
pp(run(2),true).

?- ppp .
No
```

```
pp(run(1),true).
pp(run(2),true).
pp(run(1),true).

?- ppp .
Yes
```

```
pp(run(1),true).
pp(run(2),run(2)).
pp(run(3),true).

?- ppp2.
No
```

The only built-in predicate your solution must be able to deal with, is `true/0`. You might have noticed that the property PPP is not decidable because the PP could loop. Still, even though the program in the rightmost example loops (for both strategies), it is possible to decide that it is not a PPP. You do not need to explore the whole search tree to detect this.

Start by writing a `ppp/0` that works for loop-free programs (this is relatively easy). When this is finished, write a `ppp2/0` that also works on non-PPP-programs containing a loop (for either strategy), but for which it is still decidable.

Hints *Our `ppp/0` solution solves the problem for terminating programs. It works as follows: collect the clauses in a list named Program1; reverse the program (both the order of the clauses and the order of the goals in a body) to obtain Program2; then run both programs independently, collecting all their answers with `findall/3`, and then compare the answer lists. This approach works because the program terminates.*

Our `ppp2/0` solves the problem for (some) non-terminating non-PPPs. If only Prolog had a built-in that returns the n^{th} answer of a goal ... we made it ourselves: `find_nth_answer/3`. We count the solutions by using assert/retract on a dynamic predicate `answers_to_skip/1`.

The code common to `ppp/0` and `ppp2/0` is in the first program.

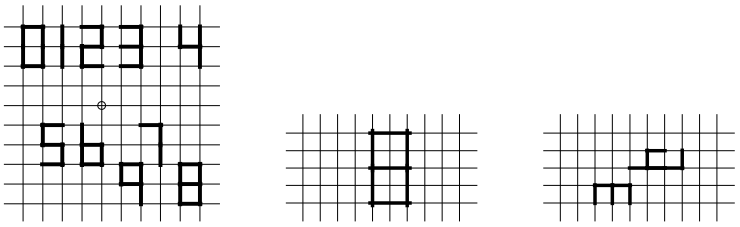
Solution

```
:- use_module(library(lists), [member/2]).

ppp :-
        findall(pp(Head,Body),pp(Head,Body),Program1),
        reverse_program(Program1,[],Program2),
        Query = run(_),
        findall(Query,exec(Query,Program1),Answers1),
        findall(Query,exec(Query,Program2),Answers2),
        is_a_copy(Answers1,Answers2).

exec((Goal1,Goal2),Program) :- !,
        exec(Goal1,Program),
        exec(Goal2,Program).
exec(true,_) :- !.
exec(Head,Program) :-
        member(Clause,Program),
        copy_term(Clause,pp(Head,Body)),
        exec(Body,Program).

reverse_program([],L,L).
reverse_program([pp(H,B)|R],In,Out) :-
        reverse_body(B,true,NewB),
        reverse_program(R,[pp(H,NewB)|In],Out).

reverse_body((A,B),In,Out) :- !, reverse_body(B,(A,In),Out).
reverse_body(A,In,(A,In)).

is_a_copy(A,B) :-
        numbervars(A,1,N),
        numbervars(B,1,N),
        A = B.
```

```
:- dynamic answers_to_skip/1.

ppp2 :-
        findall(pp(Head,Body),pp(Head,Body),Program1),
        reverse_program(Program1,[],Program2),
        repeat(0,N),
        find_nth_answer(N,Program1,Answer1),
        find_nth_answer(N,Program2,Answer2),
        ( Answer1 == nomore, Answer2 == nomore ->
            !                   % stop the repeat and succeed
        ; is_a_copy(Answer1,Answer2) ->
            fail                % ok, try next answers
        ;
            !,
            fail                % different answers found: fail
        ).
ppp2.

find_nth_answer(N,Program,Goal) :-
        Goal = run(_),
        set_answers_to_skip(N),
        exec(Goal,Program),
        answers_to_skip(M),
        ( M == 0 ->
            !
        ;
            NewM is M - 1,
            set_answers_to_skip(NewM),
            fail
        ).
find_nth_answer(_,_,nomore).

repeat(I,I).
repeat(I,J) :-
        I1 is I + 1,
        repeat(I1,J).

set_answers_to_skip(N) :-
        retractall(answers_to_skip(_)),
        assert(answers_to_skip(N)).
```

6. Numbers

Consider an infinite grid with a finite number of its segments lit up. These lit up segments are represented by the given facts `lit/4`, which have start and end point coordinates as arguments. Like on a digital display, these lit up parts can form numbers from 0 to 9. For example, a grid with every decimal number is depicted in Figure 2.1(a).



(a) All numbers

(b) Not an eight

(c) Tilted and hidden numbers

Figure 2.1: Examples of grids with numbers.

Write a predicate `numbers/1` which unifies its argument with a list (without duplicates, order is not important) of all the numbers that are recognizable in the given set of facts. For Figure 2.1(a), this would be (up to the order in the list):

```
?- numbers(L).
L = [0,1,2,3,4,5,6,7,8,9]
```

Suppose the dot in the grid of Figure 2.1(a) indicates the point with coordinates (0,0), then the number 3 in that figure is represented by the facts:

```
lit(1,2,2,2). lit(2,2,2,3). lit(2,4,2,3). lit(2,3,1,3). lit(2,4,1,4).
```

No scaling should be performed, i.e. you should not recognize the number 8 in Figure 2.1(b). On the other hand, numbers can be rotated and/or recognizable as a part of a configuration of lit up segments. E.g. in Figure 2.1(c), we have:

```
?- numbers(L).
L = [1,3,4,7]
```

The orientation of the numbers does not need to be the same, and one lit up segment can be used as part of more than one number (for instance, a 4 and a 1 is hidden in a 9, which in turn is also a rotated 6).

Hints *Each number has a description of how it can be formed. This description is a sequence of instructions up-down-left-right that, starting from a position, form the number. Numbers can also be rotated 0, 90, 180 or 270 degrees.*

Solution

```
:- use_module(library(lists), [member/2]).
:- use_module(contestlib, [map/3]).

numbers(Numbers) :- findall(Number,number_in_grid(Number),Numbers).

number_in_grid(Number) :-
        description(Number,Description),
        once(((lit(X,Y,_,_) ; lit(_,_,X,Y)),
              map(Description,rotate(_),RotatedDescription),
              matches(RotatedDescription,X,Y))).

description(0,[up,up,right,down,down,left]).
description(1,[down,down]).
description(2,[right,down,left,down,right]).
description(3,[right,down,left,right,down,left]).
description(4,[down,right,up,down,down]).
description(5,[left,down,right,down,left]).
description(6,[down,down,right,up,left]).
description(7,[right,down,down]).
description(8,[up,right,down,down,left,up,right]).
description(9,Description) :- description(6,Description).

rotate(  0,X,X).
rotate( 90,A,B) :-
        member((A,B),[(down,right),(right,up),(up,left),(left,down)]).
rotate(180,A,B) :-
        rotate(90,A,C),
        rotate(90,C,B).
rotate(270,A,B) :-
        rotate(90,B,A).

matches([],_,_).
matches([Step|Steps],X,Y) :-
        nextpoint(Step,X,Y,A,B),
        once((lit(X,Y,A,B) ; lit(A,B,X,Y))),
        matches(Steps,A,B).

nextpoint(   up, X,Y, A,B) :- A = X, B is Y + 1.
nextpoint( down, X,Y, A,B) :- A = X, B is Y - 1.
nextpoint(right, X,Y, A,B) :- A is X + 1, B = Y.
nextpoint( left, X,Y, A,B) :- A is X - 1, B = Y.
```

Contest III: 1996 Bonn, Germany

1. Nested Triangle

Write a predicate `triangle/1`, whose argument is a list L of characters (atoms of length 1). The predicate draws a set of triangles, which sit inside each other, and whose circumferences are drawn with the subsequent character from L. For instance:

```
?- triangle([a,b,c,d])

          a
         aba
        abcba
       abcdcba
      abccccba
     abbbbbbbba
    aaaaaaaaaaaaa
```

Hints *There are two separate concerns: the spaces in front of the visible text, and the visible parts themselves. The spaces follow a simple pattern: each next line has one less.*

The next observation is that the first line contains only the first character, and that up to the first time all characters appear in a line, it is as if we duplicate the middle character of the previous line, and put a new character in between. So, from a given line, we can compute the next line. This is the first part of the picture, and it is taken care of by `part1/4`.

From the moment all the characters have been used once (or more) in a line, the next lines are always made by finding the innermost character, and replacing it (all its occurrences) by its neighbor. This goes on until we can no longer find an innermost character with a neighbor. Predicate `part2/3` *implements this.*

Solution

```
:- use_module(library(lists), [reverse/2]).
:- use_module(contestlib, [write_elements/1, writeN/2]).

triangle(Chars) :-
        length(Chars,Spaces),
        Chars = [Middle|RestChars],
        End = [],
        Spaces1 is 2*Spaces,
        part1(Middle,End,RestChars,Spaces1).
triangle(Chars) :-
        length(Chars,Spaces),
        reverse(Chars,[_|Post]),
        part2(Post,5,Spaces).
triangle(_).

part1(Middle,End,RestChars,Spaces) :-
        writeN(Spaces,' '),
        reverse_write_elements(End),
        write_elements([Middle|End]),
        nl,
        Spaces1 is Spaces - 1,
        RestChars = [NewMiddle|NewRestChars],
        part1(NewMiddle,[Middle|End],NewRestChars,Spaces1).

part2([X|Cs],Len,Spaces) :-
        writeN(Spaces,' '),
        reverse_write_elements(Cs),
        writeN(Len,X),
        write_elements(Cs),
        nl,
        Spaces1 is Spaces - 1,
        Len1 is Len + 4,
        part2(Cs,Len1,Spaces1).

reverse_write_elements(L) :-
        reverse(L,RL),
        write_elements(RL).
```

2. Base

Write a predicate `base/2`, whose first argument is a ground list (we will refer to this list as the Elements), and whose second argument must be unified with the *base* of the Elements.

You get a predicate `lub/3`. This predicate can only be called with its first two arguments elements of Elements or terms you got as (ground) output (third argument) of calls to `lub/3`; its third argument will then be unified with the least upper bound of the first two arguments. `lub/3` is associative, commutative and idempotent. For example, `lub/3` could be the bitwise or, the least common multiple of two positive integers, ...

The base of a set S is the subset of all elements E of S, such that E cannot be formed by `lub`-bing two or more elements from $S \setminus \{E\}$. The order in a base is not important.

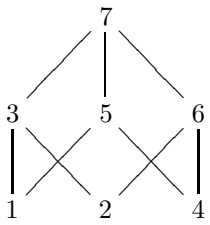
For example, suppose

```
lub(A,B,C) :- C is A \/ B .
```

then

```
?- base([1,2,3,4,5,6,7],Base) .
Base = [1,2,4]
```

as can be seen in the lattice formed by `lub/3`:



Hints *Since `lub/3` has all these nice properties, it is sufficient to compute all subsets of the input and check whether they produce an element that is not in the subset. Such an element is not in the base.*

Solution

```
:- use_module(library(lists), [member/2]).
:- use_module(contestlib, [sublist/2]).

base(Elements,Base) :- findall(X,is_base(Elements,X),Base).

is_base(Elements,X) :-
        member(X,Elements),
        \+ is_no_base(Elements,X).

is_no_base(Elements,X) :-
        sublist(Elements,Sub),
        \+ member(X,Sub),
        luball(Sub,X).

luball([X],X).
luball([X|Y],Z) :-
        luball(Y,O),
        lub(O,X,Z).
```

3. Board

You get an $N \times N$ board with a binary arithmetic operator and an integer operand on each square. You start off with 0 as current value, and you choose a path that visits each square of the board exactly once. For each visited square, you perform the operation on the square with the current value as left operand of the operation, and the number on the square as right operand, and continue with the result as the current value. At the end of the path, your current value depends on the path chosen; this is the end value of the path. You should find the minimal end value of all paths, and the number of paths that have this minimal end value.

Below, you see a board and its corresponding data structure representation:

```
[[op(*,-1), op(-,   3), op(-,555), op(-,  3)],
 [op(-, 3), op(-,2000), op(*,133), op(-,555)],
 [op(*, 0), op(*, 133), op(-,  2), op(+, 19)],
 [op(-, 3), op(-,1000), op(-,  2), op(*,  3)]]
```

$*(-1)$	-3	-555	-3
-3	-2000	$*133$	-555
$*0$	$*133$	-2	$+19$
-3	-1000	-2	$*3$

Write a predicate `board/3`, which has as first (input) argument the board, and which unifies the minimal end value with the second argument and the number of different paths having this minimal end value with the third argument. A path can start on any square, and two subsequent squares in a path must touch with a side of the squares.

Hints *We need to follow all paths without loops in this $N \times N$ board, starting from every location. In this solution, we first transform each square of the board, such that for instance* `op(*,8)` *becomes* `\Z.Z*8`.

Here, Z is a fresh variable for each square. With this representation, we access each square by using its coordinates in calls to ***`nth1/3`****. The Z variable in each square must be free, otherwise the square has been visited before. This way, we avoid loops. By unifying Z with a current value, we can compute the next value by evaluating the expression on the square. The program first generates all possible values within a* ***`setof/3`****, picks out the minimal value, uses this value in a* ***`findall/3`*** *to generate all paths with that minimal value, and then counts them with* ***`length/2`****.*

Solution

```
:- use_module(library(lists),[nth1/3]).
:- use_module(contestlib,[for/3, map/3]).

:- op(100,xfx,.).
:- op(200,fy,\).

board(InitialBoard,Min,HowMany) :-
        length(InitialBoard,Size),
        mknewboard(InitialBoard,NewBoard),
        setof(Value,NewBoard^value(NewBoard,Size,Value),[Min|_]),
        findall(Min,value(NewBoard,Size,Min),MinVals),
        length(MinVals,HowMany).

value(Board,Size,Value) :-
        for(X,1,Size),
        for(Y,1,Size),
        keepmoving(Size*Size,X,Y,Board,Size,0,Value).

keepmoving(N,X,Y,Board,Size,CurrentVal,Result) :-
        nth1(X,Board,XRow),
        nth1(Y,XRow,Lambda),
        eval(Lambda,CurrentVal,CurrentVal1),
        N1 is N - 1,
        ( N1 == 0 ->
            Result = CurrentVal1
        ;
            next(X,Y,NextX,NextY),
            0 < NextX, NextX =< Size,
            0 < NextY, NextY =< Size,
            keepmoving(N1,NextX,NextY,Board,Size,CurrentVal1,Result)
        ).

eval(\Z.Expr,In,Out) :- var(Z), Z = In, Out is Expr.

next(X,Y,NextX,NextY) :- NextX = X, NextY is Y + 1.
next(X,Y,NextX,NextY) :- NextX = X, NextY is Y - 1.
next(X,Y,NextX,NextY) :- NextX is X + 1, NextY = Y.
next(X,Y,NextX,NextY) :- NextX is X - 1, NextY = Y.

mknewboard(In,Out) :- map(In,mknewrow,Out).

mknewrow(In,Out) :- map(In,mknewsquare,Out).

mknewsquare(op(A,B),\Z.Expr) :- Expr =.. [A,Z,B].
```

4. Preprocessor

We have just invented a great new optimization for Prolog, but unfortunately it only works for programs that have neither disjunction nor if-then-else in the body, and in which there is at most one user defined predicate with more than one clause. This means we cannot even compile the *naive reverse* program. Of course, we will not force the future users of our system to program with this restriction, so we need a preprocessor that accepts almost any Prolog program and transforms it to the form we like: an equivalent Prolog program with at most one predicate with more than one clause, and of course no disjunction nor if-then-else.

Since we are still in the prototyping phase, the input Prolog programs contain no disjunction nor if-then-else, but all the built-in predicates are allowed (including cut). `predicate_property(Term,built_in)` tells you whether Term is actually a built-in goal. You may assume that the program does not contain directives or queries.

Write the predicate `pre/2`, which will be called with a program represented as a list of clauses as its first argument. For example, the code for the *naive reverse* benchmark is represented as

```
[(nrev([],[]) :- true),
 (nrev([X|R],O) :- nrev(R,O2), append(O2,[X],O)),
 (append([],L,L) :- true),
 (append([X|L1],L2,[X|L3]) :- append(L1,L2,L3))
]
```

The predicate `pre/2` must unify its second argument with an equivalent program represented similarly, but with only one predicate with more than one definition. There is of course more than one correct answer. A good way to test whether an answer is correct is by running it and checking whether it complies with the other requirements. One more thing: queries are not transformed; they should run directly in your transformed program! You may assume that no symbol in the given program starts with `my_` ; this can make it easier to invent unique new symbols.

Hints *It is all much simpler than it appears at first sight. Here is an example of what the given solution does:*

```
?- pre([ (a:-b), (a:-c), (d :- write(ok))],  NewProg).

NewProg = [ (my_pred(a):-b), (my_pred(a):-c), (my_pred(d):-write(ok)),
            (a:-my_pred(a)), (d:-my_pred(d))]
```

As you can see, there is no need to transform the clause bodies. The heads are just wrapped in a `my_pred/1` wrapper, and for every defined predicate (like `d/0`) we add one clause (like `d :- my_pred(d)`).

Solution

```
pre(ProgIn,ProgOut) :-
        transform(ProgIn,[],ProgOut).

transform([],Preds,PredCalls) :-
        sort(Preds,UniquePreds),
        makepredcalls(UniquePreds,PredCalls).
transform([(Head :- Body)|RestProgIn],Preds,[NewClause|RestProgOut]) :-
        functor(Head,Name,Arity),
        NewPreds = [Name/Arity | Preds],
        NewClause = (my_pred(Head) :- Body),
        transform(RestProgIn,NewPreds,RestProgOut).

makepredcalls([],[]).
makepredcalls([Name/Arity|R],[(Head :- my_pred(Head))|S]) :-
        functor(Head,Name,Arity),
        makepredcalls(R,S).
```

5. Warp

Consider a finite maze with a space warp. The maze is a set of connected squares. Its shape is unknown, but, apart from the warp, it is flat. The squares are possibly connected at their `south`, `north`, `west` and/or `east` side. When you are standing on a square, you can try to go to the neighboring (say) `south` square by asking (kindly) `move(south,Where)`, and you will get the name of the square you arrive at (unifying it with `Where`). This is all very straightforward, except for the space warp in this maze. The space warp connects two squares, and when you step on one of them, you arrive at the other end of the warp. It works as an instantaneous tunnel. For example, suppose the warp has begin-end squares with names *here* and *there*, you were standing on the north neighbor of *here*, and you would ask to `move(south,Where)`; then `Where` will be unified with *there* (*not* with *here* as you would expect). So, both ends of the warp have a name, and you have to figure out the name of either end (or beginning) of the warp. It does not matter which end.

You start off at a square named `start`, which is guaranteed not to be part of the warp. All names of squares are ground and different.

You probably noticed that the `move` request has no argument indicating from which square you want to move. Of course, the maze knows where you are! The maze also likes you to ask kindly to move, which means that the direction (argument 1) should be a correct direction, that the place of arrival has a name which can unify with argument 2, and that the move is possible. I.e.,if you ask to move to the north of your current position when this is impossible, this both embarrassing and unkind to the maze. Therefore, the maze will throw you to nowhere and you will not be able to move anymore! But the maze is also kind to you: you can always find out which directions are possible from the current position by asking `directions(WhereToGo)`, where `WhereToGo` will be unified with a list of directions (in no particular order) which you can go in (e.g. `[north, east, west]`). This also has to be asked kindly!

The warp works both ways and as often as you step on either side. There is guaranteed to be exactly one warp that you can reach from the start position.

To sum up: you get the predicates `move/2` and `directions/1`, and you have to write the predicate `warp/1`, which unifies its free argument with the name of one warp end.

Hints *The solution explores the maze exhaustively, i.e. from a position, every direction is tried in predicate* ***`explore/3`****. Of course we avoid loops. The predicate* ***`at_warp/2`*** *checks whether we have moved to either end of a warp, by setting a step back: if we get back where we started from, it was not a warp end.*

Solution

```
:- use_module(library(lists), [member/2]).

warp(End) :-
        Explored = [],
        Current = start,
        once(explore(Current,Explored,End)).

explore(Current,Explored,End) :-
        directions(Directions),
        member(Direction,Directions),
        move(Direction,NewPoint),
        (
          \+ member(NewPoint,Explored),
          ( at_warp(Direction,Current) ->
              End = NewPoint
          ;
              explore(NewPoint,[Current|Explored],End)
          )
        ;
          goback(Direction),
          fail
        ).

goback(Dir) :-
        oppositedir(Dir,Opposite),
        move(Opposite,_).

at_warp(Dir,Current) :-
        oppositedir(Dir,Opposite),
        directions(Dirs),
        once(member(Opposite,Dirs)),
        move(Opposite,NewCurrent),
        move(Dir,_),
        Current == NewCurrent,
        !,
        fail.
at_warp(_,_).

oppositedir(south,north).
oppositedir(north,south).
oppositedir(east,west).
oppositedir(west,east).
```

Contest IV: 1997 Leuven, Belgium

1. Snake

The body of the Belgian snake shows a repeating pattern. However, the pattern is not necessarily repeated an integral number of times. A pattern consists of a sequence of rings, and a ring has an identifier which is an atom of length 1. When the Belgian snake takes a nap, it likes to lie coiled up in a particular way: it always lies in a rectangle, its head in the upper left corner and filling the rectangle row by row (see the query below). Write a predicate `snake/3`, which displays such a coiled up Belgian snake. This predicate will be called with the following three arguments: a list of atoms representing a pattern, a list whose length is equal to the number of rings in one column and a list whose length is equal to the number of rings in one row. Your `snake/3` should draw the coiled up snake as output on the screen. For example:

```
?- snake([a,b,c,d],[_,_,_,_,_],[_,_,_]).

abcda
badcb
cdabc
```

This snake would look like `abcdabcdabcdabc` when stretched out.

There is a catch: Belgian snakes dislike arithmetic computations very much. Therefore, we urge you to avoid any arithmetic.

Hints *The snake consists of an ever repeating pattern, and one way of representing this is by a cyclic list: it contain the pattern and bites its tail. This cyclic list is used as a infinite supply of the pattern, from which we need to take pieces with the same length as the list in the second argument. This piece must be reversed for even rows.*

Solution

```
:- use_module(library(lists), [reverse/2]).
:- use_module(contestlib, [write_elements/1]).

snake(Pattern,Cols,Rows) :-
        infinite_snake(Pattern,InfSnake,InfSnake),
        produce_snake(Rows,Cols,InfSnake,Snake),
        coil_it(Snake,odd).

infinite_snake([],S,S).
infinite_snake([A|R],[A|T],S) :-
        infinite_snake(R,T,S).

produce_snake([],_,_,[]).
produce_snake([_|Rows],Cols,InfSnake,[Part|Tail]) :-
        part_of_snake(Cols,InfSnake,NewInfSnake,Part),
        produce_snake(Rows,Cols,NewInfSnake,Tail).

part_of_snake([],RestSnake,RestSnake,[]).
part_of_snake([_|R],[Ring|Rings],RestSnake,[Ring|RestRings]) :-
        part_of_snake(R,Rings,RestSnake,RestRings).

coil_it([],_).
coil_it([Line|Lines],odd) :-
        write_elements(Line), nl,
        coil_it(Lines,even).
coil_it([Line|Lines],even) :-
        reverse(Line,Line1),
        write_elements(Line1), nl,
        coil_it(Lines,odd).
```

2. Index

The readability index of a Prolog program clause is defined in terms of the following entities:

nlit: the total number of literals in the clause

narg: the total number of arguments of literals in the clause

ndisj: the total number of disjunctions

niff: the total number of if-then-elses

neq: the number of occurrences of the literal =/2

neckcut: (0 or 1) whether the clause has a ! as first goal

ncut: the total number of cuts in the clause

maxarg: the maximal depth of the arguments

Note that the head of the clause also counts as a literal. The best formula combining these entities is debatable, and therefore arbitrarily decided to be:

$$nlit + 7 * narg + 2 * neq + 3 * ndisj + niff + neckcut + 12 * ncut + maxarg$$

Write a predicate `index/2` that, given a clause as first argument, unifies its readability index with the second argument. Clauses are always given as `(Head :- Body)` and facts have a `Body` equal to `true`. Do not worry about higher order arguments. The depth of an argument equals:

if the argument is atomic or a variable: 0

else: 1 + the maximum of the depths of its arguments

For the following clauses, their characteristics are:

clause	nlit	narg	ndisj	niff	neq	neckcut	ncut	maxarg
a :- true.	2	0	0	0	0	0	0	0
a :- b,c.	3	0	0	0	0	0	0	0
a :- b;c.	3	0	1	0	0	0	0	0
a :- b → c ; d.	4	0	1	1	0	0	0	0
a :- b → c.	3	0	0	1	0	0	0	0
a([1,2,3]) :- true.	2	1	0	0	0	0	0	3
a :- !, b, !.	4	0	0	0	0	1	2	0
a :- X = 1, Y = f(X).	3	4	0	0	2	0	0	1

Hints *This is a very boring exercise, sorry. Just be systematic.*

Solution

```
:- use_module(library(lists), [member/2]).

index((H:-B),Index) :-
        count((B,H),Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth),
        Index is Nlit + 7*Narg + 2*Neq + 3*Ndisj + Niff
                 + Neckcut + 12*Ncut + MaxDepth.

count(Body,Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth) :-
        var(Body), !,
        (Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth) =
        (   1,   0,    0,   0,  0,      0,   0,       0).
count(Body,Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth) :-
        control_construct(Body,A,B,PlusDisj,PlusIff), !,
        count(A,Nlit1,Narg1,Ndisj1,Niff1,Neq1,Neckcut,Ncut1,MaxDepth1),
        count(B,Nlit2,Narg2,Ndisj2,Niff2,Neq2,      _,Ncut2,MaxDepth2),
        Nlit is Nlit1 + Nlit2,
        Narg is Narg1 + Narg2,
        Ndisj is Ndisj1 + Ndisj2 + PlusDisj,
        Niff is Niff1 + Niff2 + PlusIff,
        Neq is Neq1 + Neq2,
        Ncut is Ncut1 + Ncut2,
        MaxDepth is max(MaxDepth1,MaxDepth2).
count(Goal,Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth) :-
        Goal = (_ = _), !,
        (Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut) =
        (   1,   2,    0,   0,  1,      0,   0),
        max_depth(Goal,MaxDepth).
count(!,Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth) :- !,
        (Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut) =
        (   1,   0,    0,   0,  0,      1,   1),
        MaxDepth = 0.
count(Goal,Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut,MaxDepth) :-
        functor(Goal,_,Narg),
        (Nlit,Narg,Ndisj,Niff,Neq,Neckcut,Ncut) =
        (   1,Narg,    0,   0,  0,      0,   0),
        max_depth(Goal,MaxDepth).

control_construct((A,B),A,B,0,0).
control_construct((A;B),A,B,1,0).
control_construct((A->B),A,B,0,1).
```

```
max_depth(Term,MaxDepth) :-
        atomic(Term), !,
        MaxDepth = 0.
max_depth(Term,MaxDepth) :-
        setof(D,depth(Term,D),[M|_]),
        MaxDepth is -M - 1.

depth(Term,Depth) :-
        compound(Term), !,
        Term =.. [_|L],
        member(X,L),
        depth(X,Depth1),
        Depth is Depth1 - 1.
depth(_,0).
```

3. Hex

You get a large hexagon composed of smallhexagons, numbered as in Figure 4.1. Write a predicate `hex/4`, which has the size S of the large hexagon (i.e. the number of hexagons in its first row) as the first argument, the number of a small hexagon H as the second argument, and a distance N as its third argument. The predicate `hex/4` should unify the fourth argument with a sorted list of the numbers of the small hexagons which are at distance N from H in this hexagon.

Figure 4.1 shows a hexagon of size 3 and the small numbered hexagons..

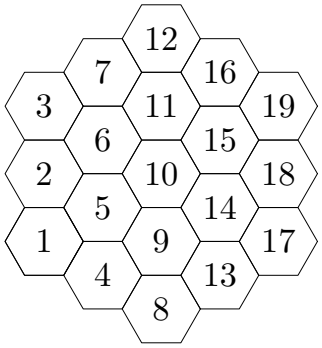


Figure 4.1: The size 3 hexagon.

The neighbors at distance 1 of 15 are [10,11,14,16,18,19]. Here are some more example calls and answers:

```
?- hex(4,11,0,L).
L = [11]

?- hex(4,11,1,L).
L = [5,6,10,12,17,18]

?- hex(4,11,5,L).
L = [22,28,33,37]

?- hex(4,19,500,L).
L = []

?- hex(4,19,3,L).
L = [1,2,3,4,5,9,10,15,16,22,23,28,29,33,34,35,36,37]
```

Hints *We visualize the hexagon as a sequence of columns, numbered from 1 (at the left) to* $2 * \texttt{HexSize} - 1$. `HexSize` *denotes the size of the first column, and is also the first argument to the query to* `hex/4`. `ColNr` *is a variable throughout the program that denotes a column number.*

The predicate `column_of_point` *figures out which is the textttColNr of the given textttPoint (the second argument of the query), and also computes the minimal value in the column. Such a minimal value is denoted by the variable textttColMin in the program. Finally, it also tells the column size textttColSize.*

Both the textttColSize and `ColNr` *for the columns of a hexagon of* `HexSize` *4 are shown in Figure 4.2.*

The column in which `Point` *is found contains two potential neighbors at distance* `Dist`. *These two are at distance* `UpD` *and* `LowD` *from the minimal element of the column of the given* `Point`. *The program maintains these up and low distances while moving from the* `Point` *column to the left or the right. Their values are shown at the bottom of Figure 4.2.*

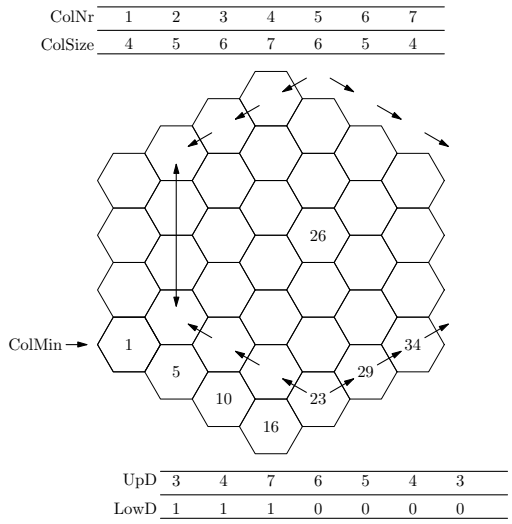


Figure 4.2: HexSize = 4, Point = 26, Distance = 3.

The program starts by moving to the left of the initial `ColNr` *(in which* `Point` *was found), and collects all the neighbors at the correct distance. This is done with an accumulating parameter, since we want the neighbors ordered in the end. To the right, this can be done with a simple output parameter.*

In between moving left and right, the neighbors in the column of `Point` *are added to the result.*

Solution

```
hex(HexSize, Point, Dist, Neighbours) :-
        column_of_point(HexSize, Point,1,ColNr,ColSize,1,ColMin),
        UpD is Point - ColMin + Dist,
        LowD is Point - ColMin - Dist,
        LeftStop is max(1,ColNr-Dist),
        RightStop is min(ColNr+Dist,2*HexSize - 1),
        addmoreleft(UpD,LowD,HexSize,ColNr,ColMin,ColSize,
                    LeftStop,Dist,Acc,Neighbours),
        add(ColMin, ColSize, UpD, LowD, Acc1, Acc),
        addmoreright(UpD,LowD,HexSize,ColNr,ColMin,ColSize,
                     RightStop,Dist,Acc1).

column_of_point(HexSize,Point,CurrentColNr,PointColNr,ColSize,
                CurrentColMin,ColMin) :-
        col_size(CurrentColNr,HexSize,CurrentColSize),
        NextColMin is CurrentColMin + CurrentColSize,
        ( Point < NextColMin ->
            PointColNr = CurrentColNr,
            ColMin = CurrentColMin,
            ColSize = CurrentColSize
        ;
            NextColNr is CurrentColNr + 1,
            column_of_point(HexSize,Point,NextColNr,PointColNr,ColSize,
                            NextColMin,ColMin)
        ).

col_size(Col,HexSize,ColSize) :-
        ( Col =< HexSize ->
            ColSize is HexSize + Col - 1
        ;
            ColSize is 3*HexSize - Col - 1
        ).
```

```
addmoreleft(UpD,LowD,HexSize,ColNr,ColMin,ColSize,Stop,Dist,Acc,Result) :-
        ( Stop == ColNr ->
            Result = Acc
        ;
            ColNr1 is ColNr - 1,
            ( ColNr > HexSize ->
                LowD1 is LowD + 1,
                UpD1 = UpD,
                ColSize1 is ColSize + 1
            ;
                UpD1 is UpD - 1,
                LowD1 = LowD,
                ColSize1 is ColSize - 1
            ),
            ColMin1 is ColMin - ColSize1,
            ( Dist == 1 ->
                addall(LowD1,UpD1,ColMin1,ColSize1,Acc,Result)
            ;
                add(ColMin1,ColSize1,UpD1,LowD1,Acc,Acc1),
                Dist1 is Dist - 1,
                addmoreleft(UpD1,LowD1,HexSize,ColNr1,ColMin1,
                            ColSize1,Stop,Dist1,Acc1,Result)
            )
        ).



addall(LowD,UpD,Min,ColSize,Tail,List) :-
        From is max(Min,Min+LowD),
        To is min(Min+ColSize-1,Min+UpD),
        interval(From,To,Tail,List).

add(ColMin,ColSize,UpD,LowD,Acc,Acc1) :-
        P1 is ColMin + UpD,
        P2 is ColMin + LowD,
        ( ColMin =< P1, P1 < ColMin + ColSize ->
            Ns = [P1|Acc]
        ;
            Ns = Acc
        ),
        ( ColMin =< P2, P2 < ColMin + ColSize, P1 \== P2 ->
            Acc1 = [P2|Ns]
        ;
            Acc1 = Ns
        ).
```

```
addmoreright(UpD,LowD,HexSize,ColNr,ColMin,ColSize,Stop,Dist,Result) :-
        ( Stop == ColNr ->
            Result = []
        ;
            ColNr1 is ColNr + 1,
            ( ColNr < HexSize ->
                LowD1 is LowD + 1,
                UpD1 = UpD,
                ColSize1 is ColSize + 1
            ;
                UpD1 is UpD - 1,
                LowD1 = LowD,
                ColSize1 is ColSize - 1
            ),
            ColMin1 is ColMin + ColSize,
            ( Dist == 1 ->
                addall(LowD1,UpD1,ColMin1,ColSize1,[],Result)
            ;
                add(ColMin1,ColSize1,UpD1,LowD1,Result1,Result),
                Dist1 is Dist - 1,
                addmoreright(UpD1,LowD1,HexSize,ColNr1,ColMin1,
                             ColSize1,Stop,Dist1,Result1)
            )
        ).



interval(From,To,Tail,List) :-
        ( From > To ->
            List = Tail
        ;
            List = [From|Rest],
            From1 is From + 1,
            interval(From1,To,Tail,Rest)
        ).
```

4. Codegen

Consider a machine with registers r_1 up to r_n, organized in a ring. This machine has only two instructions:

`move(i)` copies the contents of r_i to r_{i+1} for $1 \leq i < n$, and from r_n to r_1 for $i = n$.

`swap(i,j)` swaps the contents of r_i and r_j.

You get the initial contents of all registers (some can be wild cards) and the desired final contents of all registers (again some can be wild cards). Generate a shortest instruction sequence of `move/1` and `swap/2` that transforms the initial contents into the final contents, or fails if this is impossible.

Write the predicate `codegen/3` whose first argument is a list of the initial contents of the registers, and whose second argument is their final contents. The sequence of instructions that effectuate the transition is unified with the third argument, as in

```
?- codegen([a,b,c,d],[a,d,a,b],L).
L = [move(2),move(1),swap(2,3),swap(2,4)]
```

Wild cards are represented as a `*`. Initially, their meaning is *"don't know the contents"*; in the description of the final contents, they mean *"don't care about the contents"*. Accordingly:

```
?- codegen([a,*,c],[c,a,*],L).
L = [swap(1,2),swap(1,3)]
```

and there are five more correct answers, but one is enough.

One more example:

```
?- codegen([a,b,c],[a,a,*],L).
L = [move(1)]
```

Hints *We will do this with a simple iterative deepening generate and test: we generate longer and longer sequences of instructions, and test whether they transform the initial registers into the final registers. Before starting that, we transform the final register list, such that stars are replaced by new variables. Checking whether we have reached the desired final contents then reduces to a unification. This transformation happens through the predicate `preprocess/3`, which also collects the values in input and output. The required instruction sequence can be generated if and only if the output does not contain any new values. This check prevents generating instruction sequences forever.*

Solution

```
:- use_module(library(lists), [append/3, member/2]).

codegen(Initial,Final,Instrs) :-
        preprocess(Initial,_,Vali),
        preprocess(Final,Final1,Valf),
        \+ (member(X,Valf), \+ member(X,Vali)),
        once((length(Instrs,_),
              effectuate(Instrs,Initial,Final1))).

preprocess([],[],[]) .
preprocess([X|IR],[Y|FR],Symb) :-
        ( X = (*) ->
            preprocess(IR,FR,Symb)
        ;
            Y = X,
            Symb = [X|RS],
            preprocess(IR,FR,RS)
        ).

effectuate([],Final,Final).
effectuate([Instr|RI],Initial,Final) :-
        apply_instr(Instr,Initial,Initial2),
        effectuate(RI,Initial2,Final).

apply_instr(swap(I,J),Initial,Final) :-
        append(BeforeI,[XI|AfterI],Initial),
        append(BeforeJ,[XJ|AfterJ],AfterI),
        append(BeforeJ,[XI|AfterJ],NewAfterI),
        append(BeforeI,[XJ|NewAfterI],Final),
        length(BeforeI,I1),
        length(BeforeJ,J1),
        I is I1 + 1,
        J is I1 + J1 + 2.
apply_instr(move(From),Initial,Final) :-
        append(Front,[X|Back],Initial),
        ( Back == [] ->
            Initial = [_|Rest],
            Final = [X|Rest]
        ;
            Back = [_|Back1],
            append(Front,[X,X|Back1],Final)
        ),
        length(Front,I),
        From is I + 1.
```

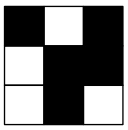
5. Shapes

You get an $N \times M$ matrix, filled with the colors black and white. The blacks are background, and a connected component of whites form a shape. You are to determine the number of shapes in the matrix. A more precise definition of a shape is: 2 whites of the matrix belong to the same shape if there is a path from one to the other over whites, where a path consist of successive points with coordinates (i,j) and (k,l) such that $|i - k| \leq 1$ and $|j - l| \leq 1$.

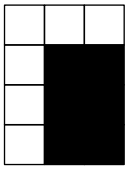
The matrix is given as a list of lists as the first argument of the predicate `shapes/2`, which you have to write. Your program should not use any of the following built-in predicates: `arg/3`, `=../2`, `functor/3`, `name/2`. It should use arithmetic either.

Here are some example calls and the matrix corresponding to the input:

```
?- shapes([[black,white,black],
           [white,black,black],
           [white,black,white]],N).
N = 2
```



```
?- shapes([[white,white,white],
           [white,black,black],
           [white,black,black],
           [white,black,black]],N).

N = 1
```



Hints *The general idea of our solution is this: replace every white in the matrix by a variable, and unify two variables that are neighbors. At the end, counting the number of different variables is done by* `numbervars`*: no other arithmetic is used. More specifically, from the matrix*

```
[[black,white,black],
 [white,black,black],
 [white,black,white]]
```

we compute

```
[[black,    white(A), black   ],
 [white(B), black,    black   ],
 [white(C), black,    white(D)]]
```

While doing that, we unify neighbors if possible, which results in

```
[[black,    white(A), black   ],
 [white(A), black,    black   ],
 [white(A), black,    white(D)]]
```

which has two variables, and hence two shapes. Note the similarity with the union-find *algorithm.*

We could have done without the `white/1` *data structure, but we would have had to use some impure predicate.*

Note how a line that is already transformed is used for taking the three neighbors of the element we are considering in the next line. These three neighbors are the `P0`*,* `P1` *and* `P2` *in the program. It is convenient to have an extra black neighbor at the left and right, and* `new_neighbor/3` *caters that. It even caters for an unlimited supply of black cells starting from the empty list, and we use that in the second argument of the top call to* `transform/3`*.*

Solution

```
shapes(Matrix,N) :-
        transform(Matrix,[],Matrix1),
        numbervars(Matrix1,0,N).

transform([],_,[]).
transform([Line|RestMatrix],PrevLine,[Line1|RestMatrix1]) :-
        new_neighbor(PrevLine,P1,PrevLine1),
        new_neighbor(PrevLine1,P2,PrevLine2),
        transformline(Line,black,P1,P2,PrevLine2,white(_),Line1),
        transform(RestMatrix,Line1,RestMatrix1).

transformline([],_,_,_,_,_,[]).
transformline([black|Xs],_,P1,P2,RP,_,[black|Ys]) :-
        new_neighbor(RP,P3,RestRP),
        transformline(Xs,P1,P2,P3,RestRP,white(_),Ys).
transformline([white|Xs],P0,P1,P2,RP,WhiteVar,[WhiteVar|Ys]) :-
        new_neighbor(RP,P3,RestRP),
        bind(P0,WhiteVar),
        bind(P1,WhiteVar),
        bind(P2,WhiteVar),
        transformline(Xs,P1,P2,P3,RestRP,WhiteVar,Ys).

bind(black,_).
bind(white(V),white(V)).

new_neighbor([],black,[]).
new_neighbor([X|R],X,R).
```

Contest V: 1998 Manchester, UK

1. Remote

I am a TV addict and I have a television in each of the four corners of my living room. They are all on all the time. Wherever I sit, I can always see (at least) one TV. I do not care what I watch, but I want all four TVs to show the same channel. Unfortunately, I can only receive five channels, named 1,2,3,4 and 5. When I wake up in the morning (after falling asleep on the TV room table while watching some late night show), the first thing I want to do is set all TVs to the same channel, not caring which channel. Of course, I use my remote control for this, but it is harder than it sounds. During the night, all TVs switch randomly to one of the five channels. And my remote control only has a *next* button with which I can choose the next channel (modulo 5). On top of that, my TVs refuse to change channel twice in a row, so, in order to change one particular TV twice, I have to change channels on at least one other TV in between. And this while I am in a hurry to make all TVs play the same channel.

Write a predicate `remote/2`, which has as its first argument a list of the channels for the four TVs (i.e. a list of four integers between 1 and 5), and which unifies its second argument with a list of TVs I should change consecutively to achieve my goals. Moreover, this list should have minimal length. One solution is enough.

For example:

```
?- remote([3,1,3,3],L).
  L = [1,2,3,2,4,2]
Yes
```

Hints *Iterative deepening with generate and test will do the trick ...*

Solution

```
:- use_module(contestlib, [for/3]).

remote(Channels,Zaps) :- once(gen_shortest(Channels,Zaps)).

gen_shortest(Channels,Zaps) :-
        LastZapped = none,
        length(Zaps,_),
        gen_allowed_zapping(Zaps,LastZapped),
        check_zaps(Zaps,Channels,[X,X,X,X]).

gen_allowed_zapping([],_).
gen_allowed_zapping([Zap|Zaps],LastZapped) :-
        for(Zap,1,5),
        Zap \== LastZapped,
        gen_allowed_zapping(Zaps,Zap).

check_zaps([],Channels,Channels).
check_zaps([Zap|Zaps],ChannelsIn,ChannelsOut) :-
        zap(Zap,ChannelsIn,Channels1),
        check_zaps(Zaps,Channels1,ChannelsOut).

zap(1,[A,B,C,D],[A1,B,C,D]) :- zap(A,A1).
zap(2,[A,B,C,D],[A,B1,C,D]) :- zap(B,B1).
zap(3,[A,B,C,D],[A,B,C1,D]) :- zap(C,C1).
zap(4,[A,B,C,D],[A,B,C,D1]) :- zap(D,D1).

zap(X,Y) :- Y is (X mod 5) + 1.
```

2. Choices

Prof. N. Ocut only writes very pure Prolog programs. She never uses any built-in predicate (in particular cut), and she even refuses to use if-then-else and disjunction. Still, she claims that all her programs are highly deterministic. More precisely, she claims that, if the Prolog implementation has full indexing on all arguments, the execution of her queries never creates a choice point. However, she recently started to doubt her ability to write such programs, and she would feel better of this were checked. Not by global analysis (good lord, no!), but by running what she believes is a typical set of queries, and observing the choicepoint stack. Help her by writing a predicate named `choices/2`: it will be called with as first argument a (typical) query, and it should unify its second argument with a list (order not important) of subgoals (instantiated as they were at the moment of the call) which require a choice point. Below, you find a program Prof. N. Ocut wrote, and some queries to `choices/2` with their answers:

```
aa :- b(X), c(X,Y), d(Y).

append([],L,L).                 b(1).          c(1,3).         d(3).
append([X|R],S,[X|T]) :-        b(1).          c(1,3).         d(4).
           append(R,S,T).       b(2).          c(2,4).         d(4).


?- choices(aa,L).                          ?- choices(b(3),L).
  L = [b(_93),d(4),c(1,_89)]                 L = []

?- choices((b(X),b(X)),L).                 ?- choices((aa,b(9)),L).
  L = [b(_86),b(1)]                          L = [b(_99),d(4),c(1,_95)]

?- choices((c(1,X),d(X)),L).               ?- choices(append(X,Y,[1]),L).
  L = [c(1,_84)]                             L = [append(_93,_94,[1])]

?- choices(append(X,Y,[1,2]),L).           ?- choices(append(X,Y,[]),L).
  L = [append(_95,_96,[1,2]),                L = []
               append(_123,_124,[2])]
```

As you see, `choices/2` does not (need to) produce an answer in its first argument, only in its second argument. Moreover, even if the first argument as a goal fails, `choices/2` should succeed. The answer should not contain duplicates.

You can assume that you can use `clause/2` on the goals you are given. You can also assume that execution of any given query terminates. Of course, you can use some of the dirty features of Prolog, like cut for instance. In fact, everything is allowed, except for the `assert/retract/record`-family of predicates. That would annoy Prof. N. Ocut too much, and we would rather please her, no?

Hints *We will solve this by writing an enhanced meta interpreter. We add one output argument to the classical 3-line meta interpreter, which we will call* `Multi`. `Multi` *is unified with a list of (a copy of) every goal that unified with more than one head during the execution of the top goal. The meta-interpreter must succeed for every goal, and can even produce a non-empty multi list for a failing goal.*

The predicate `exec/2` *produces a list of literals, so if answers from* `exec/2` *are collected with* `findall/3`*, we end up with a list of lists of literals, and we need to* `flatten/2` *it.*

In the end, the duplicates (variants in fact) are removed.

The last clause of `exec/2` *is worth explaining a bit. The invariant is that* `exec/2` *succeeds once and that its second argument is unified with a possibly empty list of subgoals that require a choice point. That means that the* `BodyMultis` *build by the* `findall` *goal can be empty only if the goal* `clause(Head,Body)` *had no solutions - in which case obviously no subgoal of* `Head` *needed a choice point. If* `BodyMultis` *is a list with only one element, clearly* `clause(Head,Body)` *did succeed only once, so the* `Head` *didn't need a choice point. If* `BodyMultis` *contains more than one element, the* `Head` *needed a choice point itself.*

Solution

```
:- use_module(library(lists), [flatten/2, member/2]).

choices(Goal,ChoiceList) :-
        exec(Goal,Multis),
        elim_dupl(Multis,ChoiceList).

exec((A,B),MultiAB) :- !,
        exec(A,MultiA),
        findall(Multi,(A,exec(B,Multi)),MultiB),
        flatten([MultiA,MultiB],MultiAB).
exec(true,[]) :- !.
exec(Head,MultiH) :-
        findall(Multi,(clause(Head,Body),exec(Body,Multi)),BodyMultis),
        (
          BodyMultis == [] ->
            MultiH = []
        ;
          BodyMultis = [MultiH] ->
            true
        ;
            copy_term(Head,HeadC),
            flatten([HeadC|BodyMultis],MultiH)
        ).

elim_dupl([],[]).
elim_dupl([X|R],Out) :-
        ( member(Y,R), variants(Y,X) ->
            elim_dupl(R,Out)
        ;
            Out = [X|NewOut],
            elim_dupl(R,NewOut)
        ).

variants(X,Y) :-
        \+ \+ (numbervars(X,1,N),numbervars(Y,1,N),X = Y).
```

3. Close Valves

My plumber has left the plumbing of my house in a deplorable state. The pipe network seems randomly welded together. Luckily, he installed valves in some pipes so that I can prevent flow (in either direction) through that pipe. This morning, one faucet was dripping, and before I left the house, I wanted to shut enough valves to prevent that faucet from dripping. Of course, I don't want to close too many valves, such that a maximum number of other faucets would still be able to give water. This looked straightforward to me. I figured out from a plan of the house which valves to close, but alas: the faucet was still dripping! It turned out that some valves do not close properly: I cannot rely on the plan. So, please help me and write a program for that. What you get is the plan of the house as a set of facts describing the pipe network:

`inlet/1` (exactly one)

`faucet/1` (at least one)

`dripping/1` (exactly one)

`pipe/2` (meaning that 2 nodes in the pipe network are connected by a pipe)

`valve/2` (a subset of `pipe/2`, meaning that a valve was installed on that pipe)

The arguments of all these facts are ground terms representing a node, the inlet, or a faucet.

You may also use a predicate `gets_water/2`, which you call with its first argument a faucet and its second argument a list of valves (i.e. terms of the same form as the facts for `valve/2`) you want closed. Such a call succeeds if and only if the closing of the valves has not prevented the flow of water from the inlet to the faucet. It is the analogue of me running up and down the stairs, closing valves (in the basement) and checking whether a faucet (upstairs) is still active, but you will not get so tired.

Write a predicate `close_valves/1`, which is called with a free variable, and unifies it with a list of valves that need to be closed such that the dripping faucet stops, and that the number of other faucets still capable of running water is maximal. Oh, my plumber might have been so bad that it is not possible to stop the dripping! Then `close_valves` can fail (finitely of course). One more thing: I am not sure all faucets are connected to the inlet. My plumber is really bad!

Hints *The problem statement is longer than the solution. We want a subset of valves which stops the dripping, and which leaves the maximal number of faucets active. This is just too easy in Prolog!*

Solution

```
:- use_module(library(lists), [last/2]).
:- use_module(contestlib, [sublist/2]).

close_valves(Valves) :-
        setof((N-Valves),valves_activefaucets(Valves,N),L),
        last(L,N-Valves).

valves_activefaucets(Valves,N) :-
        setof(valve(A,B),valve(A,B),AllValves),
        dripping(Drip),
        sublist(AllValves,Valves),
        \+ gets_water(Drip,Valves),
        findall(Faucet,(faucet(Faucet),gets_water(Faucet,Valves)),Faucets),
        length(Faucets,N).
```

4. Diamond

Write a predicate `diamond/1` which draws a diamond on the screen. The predicate `diamond/1` is called with a positive integer, giving the size of the diamond. Generalize from the examples:

```
?- diamond(2).
     1
  3     2
     4

?- diamond(3).
        1
     4     2
  7     5     3
     8     6
        9

?- diamond(8).
                       1
                    9     2
                17    10     3
             25    18    11     4
          33    26    19    12     5
       41    34    27    20    13     6
    49    42    35    28    21    14     7
 57    50    43    36    29    22    15     8
    58    51    44    37    30    23    16
       59    52    45    38    31    24
          60    53    46    39    32
             61    54    47    40
                62    55    48
                   63    56
                      64
```

Solution

```
:- use_module(contestlib, [writeN/2, int_width/2, write_int/2]).

diamond(N) :-
        N1 is N*N,
        int_width(N1,W),
        Spaces1 is N*W,
        first(N,1,1,N,Spaces1,W),
        Start is N*(N - 1) + 2,
        Spaces2 is 2*W,
        last(N,Start,N,Spaces2,W).

first(0,_,_,_,_,_) :- !.
first(I,Start,Row,N,Spaces,W) :-
        I1 is I - 1,
        Start1 is Start + N,
        Row1 is Row + 1,
        Spaces1 is Spaces - W,
        writeN(Spaces,' '),
        writeline(Row,Start,N,W),
        first(I1,Start1,Row1,N,Spaces1,W).

last(1,_,_,_,_) :- !.
last(I,Start,N,Spaces,W) :-
        I1 is I - 1,
        Start1 is Start + 1,
        writeN(Spaces,' '),
        Spaces1 is Spaces + W,
        writeline(I1,Start,N,W),
        last(I1,Start1,N,Spaces1,W).

writeline(0,_,_,_) :- !, nl.
writeline(R,S,N,W) :-
        write_int(S,W),
        writeN(W,' '),
        S1 is S - N + 1,
        R1 is R - 1,
        writeline(R1,S1,N,W).
```

5. Compress

We will be considering sequences of alphabetic letters (i.e. a to z) as input (e.g. *xcaabaabaabccadadcaabaabaabccadady*). This sequence can be compressed by a simple compression algorithm which replaces n (say 7) consecutive occurrences of a symbol (say p) by the symbol followed by n (i.e. $p7$), and $p7$ is clearly shorter than *ppppppp*. The same compression can also be applied to subsequences, i.e. *ababab* can be compressed to $(ab)3$. Note that $(ab)3$ has length 5. Hence, the long sequence shown before can be compressed to $x(c(a2b)3c2(ad)2)2y$. However, this is not the shortest compression, because of the occurrence of $(ad)2$ in it. Indeed, $(ad)2$ has a length that is one more than the length of *adad*. It is clear when brackets are needed and when not. Since digits nor brackets can occur in the input string, there is no ambiguity.

Write a predicate `compress/2` that takes as input a sequence (represented as a list of letters, i.e. atoms of length 1), and unifies its second argument with one shortest compressed equivalent list. The example below shows that there can be more than one correct answer:

```
?- compress([x,c,a,a,b,a,a,b,a,a,b,c,c,a,d,a,d,
             c,a,a,b,a,a,b,a,a,b,c,c,a,d,a,d,y],L).

L = [x,(,c,(,a,2,b,),3,c,2,a,d,a,d,),2,y] ;

L = [x,(,c,(,a,a,b,),3,c,c,a,d,a,d,),2,y]
```

Even if the repetition factor is larger than 9, it still only counts as one for computing the length:

```
?- compress([a,a,a,a,a,a,a,a,a,a,a,a],L).

L = [a, 12]
```

i.e. $a12$ means a repeated 12 times and its length is 2.

Hints *It is clear that the length of the input list is an upper bound on the length of the shortest solution. So, we start a computation which has an upper bound as an argument, and we avoid computations that lead to a solution which has a length larger (or equal) than the current upper bound. As soon as we find a shorter one, we use its length as a new upper bound in a new computation, and we do this until we cannot get a shorter solution, meaning we have found the shortest one. This is basically what happens in* `compress/4`*. Maybe* `branch-and-bound` *rings a bell.*

The strategy of compressing consists further in first trying whether the input can be divided into a number of equal sublists (which could lead to a shorter compression), and calling `compress/2` *on that repeating sublist. If that fails, we try to divide the input in two parts on which* `compress/2` *is called recursively.*

Together, these two give a correct algorithm. However, it is very inefficient because it will compute a shortest compression of sublists of the original list over and over again. So, we introduce some extra power to avoid that. Every time a call to `compress/2` *succeeds, we assert the result in the form of the predicate* `memo_compress/2` *(e.g.* `assert(memo_compress([a,a,a,a,a,a],[a,6]))`*). We use these asserted facts at the start of a new* `compress/2` *computation, by calling* `memo_compress/2` *to check whether we have already computed the result for the given input. In a Prolog system with tabling (like XSB), we would simply add the declaration* `:- table compress/2`*. The dynamic predicate* `memo_compress/2` *has initially two facts describing an optimal compression of sequences of one and two characters.*

The predicate `compress/2` *delivers only one solution, although there could be more than one shortest compressed form for a given input. Convince yourself that our introduction of memoing does not affect optimality, even though some optimal solutions are not recorded.*

Solution

```
:- use_module(library(lists), [append/3]).

:- dynamic(memo_compress/2).
memo_compress([C],[C]).
memo_compress([C1,C2],[C1,C2]).

compress(Initial,Compressed) :-
        ( memo_compress(Initial,Compressed) ->
            true
        ;
            length(Initial,LenInitial),
            CurrentBest = Initial,
            LenCurrentBest = LenInitial,
            compress(Initial,CurrentBest,LenCurrentBest,Compressed),
            assert(memo_compress(Initial,Compressed))
        ).

compress(Initial,CurrentBest,LenCurrentBest,Compressed) :-
        ( compress_with_bound(Initial,LenCurrentBest,NewBest) ->
            length(NewBest,NewLenBest),
            compress(Initial,NewBest,NewLenBest,Compressed)
        ;
            Compressed = CurrentBest
        ).

compress_with_bound(Initial,LenBound,Better) :-
        repetition_compress(Initial,LenBound,Better).
compress_with_bound(Initial,LenBound,Better) :-
        two_piece_compress(Initial,LenBound,Better).

repetition_compress(Initial,LenBound,Better) :-
        chopup(Initial,Piece,Repeated),
        ( Piece = [C] ->
            2 < LenBound,
            Better = [C,Repeated]
        ;
            compress(Piece,CompressedPiece),
            append(['('|CompressedPiece], [')',Repeated], Better),
            length(Better,LenBetter),
            LenBetter < LenBound
        ).
```

```
two_piece_compress(Initial,LenBound,Better) :-
        append(Piece1,Piece2,Initial),
        Piece1 \== [],
        Piece2 \== [],
        compress(Piece1,Compressed1),
        length(Compressed1,LenCompressed1),
        LenCompressed1 < LenBound,
        compress(Piece2,Compressed2),
        length(Compressed2,LenCompressed2),
        LenCompressed1 + LenCompressed2 < LenBound,
        append(Compressed1,Compressed2,Better).

chopup(List,Part,Repeated) :-
        append(Part,Rest,List),
        Part \== [], Rest \== [],
        count_parts(Rest,Part,1,Repeated).

count_parts([],_,I,O) :- !, O = I.
count_parts(Rest,Part,I,O) :-
        append(Part,Rest1,Rest),
        I1 is I + 1,
        count_parts(Rest1,Part,I1,O).
```

6. Exchange

Write a predicate `exchange/2`, which is called with as first argument a ground list of positive integers, and with the second argument free. It should unify the second argument with the maximal value of the first argument. The value of a list of integers is the alternating sum and difference of its elements, e.g.

$$value([9,8,2]) = 9 - 8 + 2 = 3$$

$$value([1,4,89,12]) = 1 - 4 + 89 - 12 = 74$$

Any other value of the list is the value of a list you get by exchanging numbers in the list. This operation of exchanging numbers is a bit tricky, though. First of all, it works on the decimal representation of the integers in the list. Secondly, you can exchange numbers only between neighboring places in the list. Here are some examples of exchanges between 2 neighboring elements in the list:

$$exchange(17,34) = (43,71)$$

$$exchange(123,45) = (54,321)$$

Got it? Two neighbors swap numbers, swapping the order at the same time. One more restriction: numbers cannot move further than to the neighboring place in the list, so transforming [12,34,56] into [43,65,12] is not possible, because the 12 has moved two positions in the list.

Here are some examples of expected behavior:

```
?- exchange([1,2],V).
V = 1
```

```
?- exchange([12,56,34],V).
V = 78
```

Hints *If there were no restriction on the swaps, then we would be talking about all permutations of the initial list. However, with this restriction, there are much fewer ways to transform a list, and so we do this with a simple generate and test. Convince yourself that the program never swaps an element more than once.*

Solution

```
:- use_module(library(lists), [reverse/2, last/2]).

exchange(List,Value) :-
        setof(Value,values(List,Value),AllValues),
        last(AllValues,Value).

values(List,Value) :-
        swap(List,LSwapped),
        value(LSwapped,Value).

swap([],[]).
swap([A|R],[A|RestOut]) :-
        swap(R,RestOut).
swap([A,B|R],Out) :-
        reverse_int(A,ARev),
        reverse_int(B,BRev),
        Out = [BRev,ARev|RestOut],
        swap(R,RestOut).

reverse_int(Int,IntRev) :-
        number_codes(Int,IntL),
        reverse(IntL,IntLRev),
        number_codes(IntRev,IntLRev).

value(L,V) :-
        InitialValue = 0,
        value(L,InitialValue,V).

value([],FinalValue,FinalValue).
value([X],InitialValue,FinalValue) :-
        FinalValue is InitialValue + X.
value([X,Y|R],InitialValue,FinalValue) :-
        InitialValue1 is InitialValue + X - Y,
        value(R,InitialValue1,FinalValue).
```

Contest VI: 1999 Santa Cruz, New Mexico

1. Star

Write a predicate `star/1`, which is called with a list of commands as input, and which draws on the screen the picture corresponding to the list of commands.

There are 5 different commands: `up`, `down`, `right`, `left`, `star`.

What does a sequence of these commands mean? Imagine there is a rectangular grid on the screen, and that you start off at some position. The command `up` obviously means that you move up one step in the grid on the screen. Similarly reasonable interpretations can be made for `down`, `left` and `right`. The `star` command means that you must write a `*` on the screen at the current position. It is clear that, by executing the list of commands, a picture of `*`s is drawn on the screen. However, since the starting position is not given[1], there is a problem. Indeed, we want the picture on the screen to be touching the left border of the screen/window. This means for instance that the following queries all produce the same picture:

```
?- star([right,right,right,right,right,star]).
?- star([left,left,left,left,left,star]).
?- star([left,right,left,right,right,star,left,left,right]).
?- star([down,star,left,down,right,up,star,right]).
```

namely, a `*` in the left column on the screen.

Get the idea?

To summarize: write a predicate `star/1`, with as first argument a list of commands, and which draws the picture corresponding to the commands on the screen, such that a left-most star of the picture is in the left-most column of the screen.

[1] Ancient Greek history explains why, but we have no time for that now.

Hints *The program starts by transforming the input into a list of locations (coordinates) where a star appears. We start off (arbitrarily) at (0,0), and keep track of the current location while executing the commands. Every time there is a star in the command sequence, we add a* `star(i,j)` *to that list, where (i,j) is the current location (i being the x-coordinate, and j the y-coordinate). The program then finds the lowest x-coordinate and the highest y-coordinate, so that printing on the screen can start.*

The next two problems are also related to the star problem, so we have grouped some code common to the solutions of these problems on Page 78.

Solution

```
:- use_module(library(lists), [member/2]).

star(Commands) :-
        collect_stars(Commands,Stars),
        Stars = [star(StartCol,StartLine0)|RestStars],
        max_line(RestStars,StartLine0,StartLine),
        write_stars(Stars,StartLine,StartCol).

write_stars([],_,_).
write_stars(Lines,StartLine,StartCol) :-
        Lines = [_|_],
        write1line(Lines,StartLine,StartCol),
        findall(star(A,B),(member(star(A,B),Lines),
                           B \== StartLine), RestLines),
        StartLine1 is StartLine - 1,
        write_stars(RestLines,StartLine1,StartCol).

write1line(Lines,StartLine,StartCol) :-
        findall(Col,member(star(Col,StartLine),Lines),Cols),
        sort(Cols,ColsS),
        writecols(ColsS,StartCol),
        nl.

writecols([],_).
writecols([X|R],Col) :-
        Col1 is Col + 1,
        ( Col == X ->
            write('*'),
            writecols(R,Col1)
        ;
            write(' '),
            writecols([X|R],Col1)
        ).
```

```
% code common to star, similistar and starpalindrome

collect_stars(Commands,Stars) :-
        once(collect_stars(Commands,0,0,UnorderedStars)),
        sort(UnorderedStars,Stars).

collect_stars([]               ,_,_,[]).
collect_stars([up   | Commands],I,J,Stars) :-
        J1 is J + 1,
        collect_stars(Commands,I,J1,Stars).
collect_stars([down | Commands],I,J,Stars) :-
        J1 is J - 1,
        collect_stars(Commands,I,J1,Stars).
collect_stars([left | Commands],I,J,Stars) :-
        I1 is I - 1,
        collect_stars(Commands,I1,J,Stars).
collect_stars([right| Commands],I,J,Stars) :-
        I1 is I + 1,
        collect_stars(Commands,I1,J,Stars).
collect_stars([star | Commands],I,J,[star(I,J)| Stars]) :-
        collect_stars(Commands,I,J,Stars).

max_line([],Max,Max).
max_line([star(_,L)|RestStars],MaxSoFar,Max) :-
        ( L > MaxSoFar ->
            max_line(RestStars,L,Max)
        ;
            max_line(RestStars,MaxSoFar,Max)
        ).
```

2. Similistar

Write a predicate `similistar/2`, whose arguments are as the command sequence in the `star` problem. It decides whether the two command sequences produce a similar star picture, according to the earlier star problem. A call to `similistar/2` succeeds if the two arguments produce a similar picture, and fails otherwise. For example:

```
?- similistar([left,left,star,left,left,star],
              [right,star,right,right,star]).
Yes
```

Hints *A standard way to decide whether two things are similar is to make a canonical representation, and test whether the canonical representations are equal. In the solution to the* `star` *problem, we were already close to a canonical representation: we constructed a list of coordinates where stars should occur. We now can move the picture so that all* x *are* ≥ 0*, all* y *coordinates are* ≤ 0*, and some* x *and* y *coordinate is equal to* 0*.*

Solution

```
similistar(Commands1,Commands2) :-
        collect_stars(Commands1,Stars1),
        collect_stars(Commands2,Stars2),
        canonical_stars(Stars1,Canonical1),
        canonical_stars(Stars2,Canonical2),
        Canonical1 == Canonical2.

canonical_stars([],[]).
canonical_stars(Stars,CanonicalStars) :-
        Stars = [star(StartCol,InitialLine)|RestStars],
        max_line(RestStars,InitialLine,StartLine),
        canonical_stars(Stars,StartCol,StartLine,CanonicalStars).

canonical_stars([],_,_,[]).
canonical_stars([star(X,Y)|R],StartCol,StartLine,[star(X1,Y1)|R1]) :-
        X1 is X - StartCol,
        Y1 is Y - StartLine,
        canonical_stars(R,StartCol,StartLine,R1).
```

3. Star Palindrome

A picture produced by a command sequence as in the `star` problem is a *star palindrome* if and only if it can be produced by a command sequence that is itself a palindrome. You can imagine what a palindrome command sequence is, so here is just one example:

```
star down left star up up star left down star
```

Write a predicate `starpalindrome/1`, which accepts as input a command sequence, which succeeds if the picture corresponding to the sequence is a star palindrome, and which fails otherwise.

Hints *Iterative deepening and generate and test might look an attractive approach, but when to stop if the picture is not a palindrome? We attack the problem from a different angle ...*

Suppose we have a command sequence for a picture. We can abstract it to a sequence of the stars occurring in it, where each star has its coordinates, just like in the representation in the `star` *problem. I.e. the sequence*

```
up up star right down star left
```

has as abstract form: `[star(0,2) star(1,1)]`. *Any abstract sequence can be made into a concrete command sequence, such that executing this sequence results in the same picture. We use this representation to prove there is a symmetry point in every star palindrome.*

Suppose a picture has a palindrome command sequence S. Abstract it to its abstract form A. There are two possibilities: A has odd length and is of the form A_{left}star(x,y)A_{right}, or A has even length and is of the form A_{left} A_{right}, where A_{left} and A_{right} have the same length. Rewrite A_{left}star(x,y)A_{right} to A_{left}star(x,y)star(x,y)A_{right}. This results in a palindrome of the other form, and so we can focus on the form A_{left} A_{right}.

Let star(x_i^l,y_i^l) be i-th element in the sequence A_{left} (starting with $i = 1$ at the left), and let star(x_i^r,y_i^r) be the i-th element in the sequence A_{right} (starting with $i = 1$ at the right). These are corresponding elements in the palindrome. Compute $(X_i,Y_i) = (x_i^r + x_i^l,\ y_i^r + y_i^l)$. Convince yourself that (X_i,Y_i) is independent of i. Now, $(X_i/2,Y_i/2)$ is the symmetry point for the picture.

So, every star palindrome has a symmetry point. Conversely, if a picture has a symmetry point, it is a star palindrome: you can construct an abstract palindrome sequence, and from there easily generate a palindrome command sequence.

Conclusion: there must exist a point of symmetry, and this is all our program tries to establish.

One small point: the formula for the coordinates of the midpoint of two points involves a division by 2. It is better to avoid that division.

Solution

```
:- use_module(library(lists), [member/2]).

starpalindrome(Commands) :-
        collect_stars(Commands,Stars),
        ( Stars == [] ->
            true
        ;
            check_palindrome(Stars)
        ).

check_palindrome(Stars) :-
        setof((X,Y),symmetry_point(Stars,X,Y),SymmetryPoints),
        member((X,Y),SymmetryPoints),
        \+ some_star_has_no_symmetric_star(Stars,X,Y).

some_star_has_no_symmetric_star(Stars,X,Y) :-
        member(star(A,B),Stars),
        \+ has_symmetric_star(A,B,Stars,X,Y).

has_symmetric_star(A,B,Stars,X,Y) :-
        member(star(C,D),Stars),
        X is A + C,
        Y is B + D.

symmetry_point(Stars,X,Y) :-
        member(star(X1,Y1),Stars),
        member(star(X2,Y2),Stars),
        X is X1 + X2,
        Y is Y1 + Y2.
```

4. Division

The following is a well known true *Statement* in arithmetics:

> *Statement:*
> If a number in decimal representation consists of 3 consecutive digits, then the number is divisible by 3.

Examples of such numbers are: 123 and 786. We now introduce *Statement(Base,M,Divisor)*, which has 3 parameters:

> *Statement(N,M,P):*
> If a number in base *Base* representation consists of M consecutive digits, then the number is divisible by *Divisor*.

It is clear that the above *Statement* equals *Statement(10,3,3)*.

You are to write a program that checks the instances of the *Statement(Base,M,Divisor)* for given values of *Base*, M and *Divisor*. Write a predicate `division/3`, which gets as three parameters the numbers $Base, M, Divisor$, and that succeeds or fails according to whether *Statement(Base,M,Divisor)* is true or not.

Here are some examples:

```
?- division(10,3,3).
Yes

?- division(15,4,4).
No

?- division(7,3,3).
Yes

?- division(451,9,9).
Yes
```

Hints *The first idea that comes to mind is: generate all numbers with consecutive digits of the required length and in the given base, and then test whether they all pass the division test. There is a problem with this approach. E.g. let the query be* `?- division(10,3,3)`. *Our naive program would generate and test both 876 and 786, and all permutations of the numbers 8, 7 and 6. We can avoid testing all these permutations by making the simple observation that two such permutations differ by a multiple of 9 (that is* $Base - 1$*), so if all permutations must be divisible by Divisor, also* $Base - 1$ *must be divisible by Divisor. And vice versa: if Divisor divides* $Base - 1$*, and one of the permutations is divisible by Divisor, all of them are.*

Solution

```
:- use_module(contestlib,[numlist/3]).
:- use_module(library(lists), [append/3]).

division(Base,M,Divisor) :-
        0 =:= (Base - 1) mod Divisor,
        \+ exists_non_divsible_subsequence(Base,M,Divisor).

exists_non_divsible_subsequence(Base,M,Divisor) :-
        Base1 is Base - 1,
        numlist(0,Base1,AllDigits),
        length(SubSeq,M),
        append(_,Part,AllDigits),
        append(SubSeq,_,Part),
        ModBase is Base mod Divisor,
        compute_mod(SubSeq,0,ModBase,Divisor,Result),
        Result =\= 0.

compute_mod([],Res,_,_,Res).
compute_mod([X|R],Acc,Base,Divisor,Res) :-
        NewAcc is (Acc*Base + X) mod Divisor,
        compute_mod(R,NewAcc,Base,Divisor,Res).
```

5. Möbius

You probably know what a Möbius ring is and how you make one: start off with a ring of paper, cut it somewhere, twist one end, and glue the ends together again. You can do this more than once and at different places. Now imagine you cut out a piece of paper from a book, and that this piece is exactly one line of the book. Say, for example, you have cut out the line:

h	e	l	l	o		w	o	r	l	d

At the other side of the strip of paper, there will also be text, say:

f	i	n	g	e	r		l	a	d	y

And, of course, the d on the first side lines up with the f on the other side. Now glue together both end points, but without twisting the strip. You get a ring, with on the outside side written 'hello world', and on the inside 'finger lady'. We call this the initial state.

Imagine you have a current position which is at the beginning of the text on the outside, i.e. at the h in the example. Now there are some commands you must be able to execute. The first command is `print`, which prints the characters on the band starting from the current position, moving a character to the right, one at a time, until you reach the position in which you were at the beginning of the print command. If you would execute a print in the initial state of the example, the screen would show:

```
hello world
```

The second command is `skip`, which skips the current position one character to the right. Starting from the initial state of the example, executing the sequence of commands

```
skip skip print
```

results in

```
llo worldhe
```

Then, there is the `twist` command, which cuts the band just before the current position (cutting is always between two characters), turns one end 180 degrees (a half twist of the end with the current position), and glues the ends back together. After this, the new current position is just to the right of the cut. Some letters will appear upside down now, but we will not care about that. For example, the sequence

```
skip skip twist print
```

results in

```
al regnifydllo worldhe
```

Finally, there is the command `stop`, indicating that you are no longer required to do anything more.

Write a predicate `mobius/3`, which gets as first argument a list of characters representing the outside of the initial ring, a list of characters representing the inside of the initial ring, and a list of commands as described above, ending with `stop`. Your program ought to execute all commands (in the given order of course), and then stop. For example:

```
?- mobius([h,e,l,l,o,' ',w,o,r,l,d],[f,i,n,g,e,r,' ',l,a,d,y],
          [print,skip,skip,twist,print,stop]).

hello world
al regnifydllo worldhe

?- mobius([h,e,l,l,o,' ',w,o,r,l,d],[f,i,n,g,e,r,' ',l,a,d,y],
   [print,skip,skip,twist,print,skip,twist,print,stop]).

hello world
al regnifydllo worldhe
l regnifydl
```

Hints *A good representation helps a lot. Imagine the initial band as if the paper were see through - we take two short words: abc and 123. From the outside you see the sequence of letters in abc and you see more vaguely the letters 321. First observe that the letters a and 3 stay together on the band, whatever command is executed. The image of this band represented by the open ended term* `normal(a/3,normal(b/2,normal(c/1,Tail0)))`. *The functor* `normal/2` *indicates that there was no twist at this place. Skipping one position to the right results in the representation* `normal(b/2,normal(c/1,normal(a/3,Tail1)))`. *Suppose we now execute a* `twist`*, then the representation changes to* `twisted(b/2,normal(c/1,normal(a/3,Tail1)))`. *And another* `skip` *results in* `normal(c/1,normal(a/3,twisted(b/2,Tail2)))`. *Twisting twice at the same place reinstalls the normal situation. Now how about printing ... we keep track of how many times a twist was made - actually we need only the parity: if it is* `even`*, the number of characters to be printed is equal to the lenght of one of the words. This corresponds to the fact that the band really has two sides. If the parity is* `odd`*, the band has only one side and we need to print both words. Which character of a pair to select, depends on whether there was a twist just before, and on whether we print the outside or the inside of the band.*

Solution

```
:- use_module(library(lists),[reverse/2]).

mobius(Side1,Side2,Commands) :-
        make_band(Side1,Side2,Band,TailBand),
        Twists = even,
        execute_commands(Commands,Side1,Twists,Band,TailBand).

make_band(Side1,Side2,Band,TailBand) :-
        reverse(Side2,Side3),
        bandmerge(Side1,Side3,Band,TailBand).

bandmerge([],[],Tail,Tail).
bandmerge([A|L1],[B|L2],normal(A/B,RestBand),Tail) :-
        bandmerge(L1,L2,RestBand,Tail).

execute_commands([],_,_,_,_).
execute_commands([Command|Commands],Len,Twists,Band,TailBand) :-
        execute_command(Command,Len,Twists,Band,TailBand,
                        NewTwists,NewBand,NewTailBand),
        execute_commands(Commands,Len,NewTwists,NewBand,NewTailBand).

execute_command(print,Len,Twists,Band,TailBand,Twists,Band,TailBand) :-
        write_band(Len,even,Band),
        ( Twists == odd ->
            write_band(Len,odd,Band)
        ;
            true
        ),
        nl.
execute_command(twist,_,Twists,Band,TailBand,NewTwists,NewBand,TailBand) :-
        even_odd(Twists,NewTwists),
        twist(Band,NewBand).
execute_command(skip,_,Twists,Band,TailBand,Twists,NewBand,NewTailBand) :-
        skip_character(Band,TailBand,NewBand,NewTailBand).

write_band([],_,_).
write_band([_|R],Odd_or_Even,normal(Chars,Rest)) :-
        select_char(Odd_or_Even,Chars,Char),
        write(Char),
        write_band(R,Odd_or_Even,Rest).
write_band([_|R],Odd_or_Even,twisted(Chars,Rest)) :-
        even_odd(Odd_or_Even,Even_or_Odd),
        select_char(Even_or_Odd,Chars,Char),
        write(Char),
        write_band(R,Even_or_Odd,Rest).
```

```
even_odd(odd,even).
even_odd(even,odd).

twist(normal(E,L),twisted(E,L)).
twist(twisted(E,L),normal(E,L)).

skip_character(normal(Chars,Tail),normal(Chars,NewTail),Tail,NewTail).
skip_character(twisted(Chars,Tail),twisted(Chars,NewTail),Tail,NewTail).

select_char(even, X/_, X).
select_char(odd , _/X, X).
```

6. Palm Tree

The garden of our hotel on Cyprus has beautiful palm trees, and they need water every morning. The gardener is a nice man who can follow and interpret orders. He must start every morning exactly at 6, and he has to give each palm tree the exact amount of water prescribed by the manager. She usually is very precise, but she has not realized how much freedom her orders really leave the gardener. Indeed, in an attempt to make sure that he is never idle, she has set up a set of rules he must obey. But before showing you the rules, here is how watering of the palm trees proceeds: the gardener fills his bucket with water at the well, walks to a palm tree, pours the required amount of water, walks to another palm tree, pours, walks, ... until his bucket is empty; he then walks back to the well, fills the bucket, walks to a palm tree, and so on until all palm trees are watered. He then walks back to the well and his watering job is done. These are the rules the gardener must obey:

> Going from one tree to another or to or from the well must be done in straight lines, and the gardener is to walk at a steady pace. We will model this by just giving you the time it takes to walk from one point to another.
>
> The bucket must be empty when returning to the well, except perhaps on the last return.
>
> The bucket must always be filled completely at the well, and this takes a fixed amount of time.
>
> Each tree must be visited exactly once.
>
> The gardener is not doing anything else but filling, pouring and walking.

The gardener knows that when he is finished watering the palm trees, he will have to start digging holes for new trees. He prefers watering over digging, so he exploits the rules to make the watering job as long as possible (of course within the rules set by the manager). This is not such a straightforward task, though, so he needs your program to help him. Your program can use given facts representing the bucket size, the distances between palm trees and the well, and the needs of each palm tree. Here is an example (from which you should be able to make a correct generalization):

```
% distances between palm trees
palm2palm(jasmine,sheherazade,18).
palm2palm(sheherazade,nina,12).
palm2palm(jasmine,nina,19).
palm2palm(elisa,nina,8).
palm2palm(elisa,sheherazade,20).
```

```
% distances from palm trees to well
palm2well(jasmine,13).
palm2well(sheherazade,19).
palm2well(nina,22).
palm2well(elisa,34).

% amount of water per tree
palm_needs(jasmine,2).
palm_needs(sheherazade,1).
palm_needs(nina,2).
palm_needs(elisa,4).

% bucket size
bucket(5).
```

Do not worry about non-Euclidean distances here. The garden has obstacles like swimming pools for the guests, a bar, flower beds, ..., and it is not always possible to walk from one tree to the other. You might have noticed that the gardener gives girls' names to his trees.

Write a predicate `palmtree/1`, which unifies its argument with an optimal compliant palm tree trajectory, or fails if no such trajectory exists. A palm tree trajectory is just a sequence of palm tree names, e.g. `[nina,jasmine,elisa,nina]`. A palm tree trajectory is compliant if the gardener could visit all the palm trees in the order of the trajectory, while obeying the watering prescriptions from his manager. The previous example is clearly not compliant, since it visits `nina` twice. The following is a compliant trajectory: `[sheherazade,jasmine,nina,elisa]`.

A compliant trajectory is optimal if the time it takes for the gardener to follow the trajectory during watering is maximal.

All distances, durations and water quantities are integers.

Hints *The program is of the generate and test type: all possible trajectories are generated (i.e. permutations of the palm trees), and then checked whether they are compliant, while computing their duration. All this happens inside a* `setof`*, such that we just need to pick out the largest one at the end, and we are done. Well,* `setof/3` *sorts in ascending order, so if we compute the negative of the duration, we can take the first one from the list produced by* `setof/3`*.*

The fixed amount of time it takes for filling the bucket is totally irrelevant for the problem.

Solution

```
:- use_module(library(lists),[select/3]).

palmtree(LongestVisit) :-
        setof(Tree,palm(Tree),Trees),
        setof(Duration-Visit,visit(Trees,Duration,Visit),AllVisits),
        AllVisits = [_-LongestVisit|_].

visit(Trees,Duration,Visit) :-
        permute(Trees,Visit),
        time_from_well(Visit,0,Duration).

time_from_well([],Duration,FinalDuration) :- FinalDuration is -Duration.
time_from_well([T|Trees],Duration,FinalDuration) :-
        palm2well(T,Time),
        Duration1 is Duration + Time,
        bucket(FullBucket),
        palm_needs(T,Cap),
        NewBucket is FullBucket - Cap,
        NewBucket > -1,
        time_from_tree(Trees,T,NewBucket,Duration1,FinalDuration).

time_from_tree([],Tree,_,DurationIn,DurationOut) :-
        palm2well(Tree,Time),
        DurationOut is DurationIn + Time.
time_from_tree([T|Trees],Tree,Bucket,Duration,FinalDuration) :-
        ( Bucket == 0 ->
            palm2well(Tree,Time),
            Duration1 is Duration + Time,
            time_from_well([T|Trees],Duration1,FinalDuration)
        ;
            palm_needs(T,Cap),
            NewBucket is Bucket - Cap,
            NewBucket >= 0,
            p2p(T,Tree,Time),
            Duration1 is Duration + Time,
            time_from_tree(Trees,T,NewBucket,Duration1,FinalDuration)
        ).

palm(X) :- palm2palm(X,_,_) ; palm2palm(_,X,_) ; palm2well(X,_).

p2p(X,Y,T) :- once((palm2palm(X,Y,T) ; palm2palm(Y,X,T))).

permute([],[]).
permute([X|R],O) :- permute(R,RP), select(X,O,RP).
```

Contest VII: 2001 Paphos, Cyprus

1. Spiral Cross

Write a predicate `cross/1`, which takes as input a positive integer, and draws a cross composed of . . . what the heck, just look at what is expected for some queries:

```
?- cross(3).

1   2
  5
4   3
```

```
?- cross(6).

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

You can look at the picture as a spiral of numbers.

Make sure that the columns are properly aligned. The argument to `cross/1` is not larger than 49.

Hints *Another way to look at the picture is as a set of shrinking rectangles sitting inside each other. This leads directly to an inductive definition of the picture in terms of the number that needs to be drawn in the upper left corner, the number of spaces before that number, and the number of spaces between the two numbers on the first line.*

The limitation on the input ensures that no printed number has more than two decimals.

Solution

```
:- use_module(contestlib, [writeN/2, write_int/2]).

cross(N) :-
        FirstIndent = 5,
        MiddleIndent is 2*N - (N mod 2) - 3,
        UpperLeft = 1,
        cross(N,FirstIndent,UpperLeft,MiddleIndent).

cross(0,_,_,_) :- !.
cross(1,FirstIndent,UpperLeft,_) :- !,
        writeN(FirstIndent,' '),
        write_int(UpperLeft,2),
        nl.
cross(N,FirstIndent,UpperLeft,MiddleIndent) :-
        UpperRight is UpperLeft + 1,
        write1line(FirstIndent,UpperLeft,MiddleIndent,UpperRight),
        N1 is N - 2,
        FirstIndent1 is FirstIndent + 2,
        UpperLeft1 is UpperLeft + 4,
        MiddleIndent1 is MiddleIndent - 4,
        cross(N1,FirstIndent1,UpperLeft1,MiddleIndent1),
        DownLeft is UpperLeft + 3,
        DownRight is UpperLeft + 2,
        write1line(FirstIndent,DownLeft,MiddleIndent,DownRight).

write1line(Spaces1,I1,Spaces2,I2) :-
        writeN(Spaces1,' '),
        write_int(I1,2),
        writeN(Spaces2,' '),
        write_int(I2,2),
        nl.
```

2. M-Queens

The famous N-queens problem is actually just a crude approximation of the less famous M-queens problem, which consists in finding all *maximally* safe configurations of queens on an $M \times M$ chess board. A configuration is maximally safe if no queen can be added without the configuration becoming unsafe. The usual definition of a safe configuration applies here too: no queen can beat any other queen. So, write a predicate `mqueens/2`, which, given as input a number, unifies the second argument with all maximally safe configurations through backtracking. The answers for input 1, 2 and 3 are given below:

```
?- mqueens(1,L).
L = [1]

?- mqueens(2,L).
L = [1,none]
L = [2,none]
L = [none,1]
L = [none,2]

?- mqueens(3,L).
L = [1,3,none]
L = [1,none,2]
L = [2,none,1]
L = [2,none,3]
L = [3,1,none]
L = [3,none,2]
L = [none,1,3]
L = [none,2,none]
L = [none,3,1]
```

The order in which solutions are delivered is not important. Each solution should be given only once. The convention is clear, but just to make sure: a solution `[none,3,none,1]` denotes a configuration in which there is no queen in the first column, one queen in column 2 in row 3, no queen in column 3 and one queen in row 1 of column 4.

Hints *We can adapt an ordinary N-queens program to generate (potential) solutions with empty columns as well. However, we need something to check whether or not we can put an extra queen in any of the empty columns. One way to do this is to check whether all squares on the board are attacked by the queens in the solution. For instance, the generation phase might generate* `[3,none,none]`*, but the square (3,2) is not attacked, so we can still place a queen in column three, and so* `[3,none,none]` *is rejected.*

Solution

```
:- use_module(library(lists),[member/2]).
:- use_module(contestlib,[numlist/3]).

mqueens(M,Solution) :-
        findall(sq(A,B),(numlist(1,M,Rows),
                         member(A,Rows),member(B,Rows)),Squares),
        mqueens(M,Squares,[],Solution).

mqueens(M,NotAttacked,PartialSolution,Solution) :-
        ( M == 0 ->
            NotAttacked = [],
            Solution = PartialSolution
        ;
            M1 is M - 1,
            (
              member(sq(M,X),NotAttacked),
              safe(PartialSolution,1,X),
              delete_attacked(NotAttacked,M,X,NotAttacked1)
            ;
              X = none,
              NotAttacked1 = NotAttacked
            ),
            mqueens(M1,NotAttacked1,[X|PartialSolution],Solution)
        ).

safe([],_,_).
safe([X|R],Dist,Y) :-
        ( X == none ->
            true
        ;
            X =\= Y,
            abs(X - Y) =\= Dist
        ),
        Dist1 is Dist + 1,
        safe(R,Dist1,Y).

delete_attacked([],_,_,[]).
delete_attacked([sq(A,B)|R],I,J,Squares) :-
        ( (A =:= I ; B =:= J ; A-B =:= I-J ; A+B =:= I+J) ->
            delete_attacked(R,I,J,Squares)
        ;
            Squares = [sq(A,B)|S],
            delete_attacked(R,I,J,S)
        ).
```

3. Trip

Teleportation has its limitations: teleportation gates cannot be visited by more than one object at the same time (whether this object is a Vulcan, a bulk transporting ship, or a black hole is immaterial), and not every gate is connected to every other gate. Nevertheless, teleportation is useful, and in fact heavily used for the transportation of silk. So much even that the silk transportation ships have a fixed schedule. This was only possible by the invention of the universal clock (which is really universal, because all universal clocks in the universe indicate the same time at every moment), and the convention that every teleportation trip in the universe departs on the hour. The schedule is maintained centrally by UTS (Universal Teleportation Systems). This avoids collisions, and allows for maximal use of the teleportation network. Of course, when individuals plan their holidays, they wish to use the teleportation system as well. They can submit a request to UTS for a trip from, say, the Betelgeuse gate to Rigel at a particular hour. UTS sends back the schedule the individual has to follow. The schedule consists of a sequence of gates, starting with Betelgeuse and ending with Rigel, and a starting time. In order to avoid collisions, it is sometimes necessary to wait at a gate (actually outside) for some time (always an integral number of hours of course). Hence, the schedule also contains terms of the form `wait(3)`, meaning: wait 3 hours before performing the next hop, or, to be more precise, *let the next 3 occasions for teleportation pass*.

Write the scheduling predicate `trip/6` for UTS. The gate network is given in the form of a set of facts `connection/2`, as in the example below. The fixed and already committed schedules are given in the form of `fixed_trip/2` facts, as in

```
connection(a,b).
connection(a,c).
connection(c,a).  % connections are not always symmetric!

fixed_trip(345,[a,b,wait(2),c,wait(1),z]).
fixed_trip(346,[a,wait(1),b,c,z]).
```

The first argument is the starting hour of the trip, the second has been explained before.

When you want a schedule for your trip, you should send a query to UTS like

```
?- trip(344,348,350,a,z,Trip).
```

where the first and second argument indicate the earliest and latest starting hour you find acceptable, the third indicates the latest arrival time, the fourth and fifth indicate the begin and end gate of your intended trip, and the last argument should be unified with a trip fact that avoids collisions with previously existing trips. If no such trip exists, the query fails. Only one trip is delivered as an answer, and it must be the (or rather *a*) trip with the earliest possible starting hour.

Just to be precise: no two arrivals at the same gate can be at the same moment, and neither can two departures. Teleportation itself takes 6 seconds, so an arrival and departure can be scheduled at the same hour at a particular gate. And remember that departures are on the hour.

Hints *The proposed solution simply generates all trips, then filters out the ones that conflict with the predefined ones. What remains is collected by* `setof/3`*, and the first is a trip with a earliest arrival time. Conflicts are found by extracting the departure and arrival events from a trip. The fixed predefined trips and the newly planned one should not have an event in common.*

It makes no sense to let a trip start by waiting at a gate, and a trip should not have two waits at the same gate, so the first argument if the predicate `any_trip/7` *tells whether a wait can be generated or not.*

Solution

```
:- use_module(contestlib,[for/3]).

trip(EarliestStart,LatestStart,LatestEnd,Begin,End,Trip) :-
        setof(Arrival-Trip,
              good_trip(EarliestStart,LatestStart,LatestEnd,
                        Begin,End,Trip,Arrival),
              [_ - Trip|_]).

good_trip(EarliestStart,LatestStart,LatestEnd,
          Begin,End,trip(Start,Path),Arrival) :-
        for(Start,EarliestStart,LatestStart),
        Start =< LatestEnd,
        any_trip(nowait,Start,LatestEnd,Begin,End,Path,Arrival),
        \+((Path = [Gate|Gates],
            tripevent(Gates,Gate,Start,Event),
            fixed_trip(FixedStart,[FixedGate|FixedRestGates]),
            tripevent(FixedRestGates,FixedGate,FixedStart,Event))).

tripevent([wait(N)|RGs],Gate,Start,Event) :- !,
        NewStart is Start + N,
        tripevent(RGs,Gate,NewStart,Event).
tripevent([_|_],Gate,Start,departure(Start,Gate)).
tripevent([G|_],   _,Start,arrival(Start,G)).
tripevent([G|RGs], _,Start,Event) :-
        NewStart is Start + 1,
        tripevent(RGs,G,NewStart,Event).

any_trip(   _, Start,        _,Begin,Begin,[Begin],Start).
any_trip(wait, Start,LatestEnd,Begin,End,[wait(Wait)|R],Arrival) :-
        N is LatestEnd - Start,
        for(Wait,1,N),
        NewStart is Start + Wait,
        NewStart =< LatestEnd,
        any_trip(nowait,NewStart,LatestEnd,Begin,End,R,Arrival).
any_trip(   _, Start,LatestEnd,Begin,End,[Begin|R],Arrival) :-
        connection(Begin,NewBegin),
        NewStart is Start + 1,
        NewStart =< LatestEnd,
        any_trip(wait,NewStart,LatestEnd,NewBegin,End,R,Arrival).
```

4. Tolerant Unification

Some systems enforce type declarations, or do type checking/inference. The reason is basically that some program errors are avoided if the compiler can tell you that you have misspelled the name of an identifier. Such policy is supposed to make your software more robust, under the slogan *Well-typed programs can't go wrong!*. The underlying philosophy derives from the observation that `bla = bla` succeeds, while `bla = blo` (note the misspelling of `bla` to `blo`!) fails.

A more modern vision on this phenomenon consists in not bothering the programmer with annoying type error messages at compile time, and allowing unification to succeed on a certain amount of possible misspellings at run time. Even though you misspelled `bla` by `blo` in some place, the unification `bla = blo` can still succeed. But if you later (in the forward continuation of `bla = blo`) try to unify `bla` (or `blo`) with `bli`, this must fail. Indeed, you cannot misspell inconsistently! This new unification procedure is called *tolerant unification*. In order to incorporate tolerant unification in a 3-line meta-interpreter (which is written in a non-tolerant kernel of the language) you will, as a first experiment, implement a predicate `tu/1`, which can execute a conjunction of (tolerant) unifications of the form $Term_1 = Term_2$, with $Term_i$ just terms. `tu/1` succeeds once or fails. No messages are printed. Here are a few examples:

```
?- tu(foo(X,bla) = bar(blo,X)).
X = bla     % or X = blo

?- tu(bla(blo) = blo(bli)).
Yes
```

```
?- tu((bla = blo, blo = bli)).
No

?- tu(bla(_,_) = bla(_)).
No
```

The last two examples show that tolerant unification is arity sensitive[1]. In this version of tolerant unification, we will allow almost any sort of misspelling, not just one-character misspellings. This means that `x = qwertyuio` also succeeds. However, during experiments, it was noticed that very few people misspell a list into another functor of arity two, and that misspelling an empty list by another atom is also rare, so unifications like `[a] = dot(a,[])` and `[a|nil] = [a]` should fail. Also, we do not allow numbers to be misspelled.

[1] There is still some research to be done in this area. Eventually, totally tolerant unification (TTU) will also let terms with different arities unify, such that many programs in which you have added an extra argument to some of the functor occurrences will continue running as intended. Even when a limited number of arguments are in different positions, your program will still make sense under the TTU paradigm. However, TTU is a bit hard to specify within the scope of this book. Also, first argument indexing is non-trivial.

Solution

```
:- use_module(library(lists),[member/2]).

tu(Goal) :- tu_goal(Goal,[],_).

tu_goal((T1=T2,Goal),ToleranceIn,ToleranceOut) :-
        tu_terms(T1,T2,ToleranceIn,Tol),
        tu_goal(Goal,Tol,ToleranceOut).
tu_goal(T1=T2,ToleranceIn,ToleranceOut) :-
        tu_terms(T1,T2,ToleranceIn,ToleranceOut).

tu_terms(T1,T2,ToleranceIn,ToleranceOut) :-
        ( (special(T1) ; special(T2)) ->
            T1 = T2,
            ToleranceOut = ToleranceIn
        ; (atom(T1) ; atom(T2)) ->
            atom(T1), atom(T2),
            tolerance_lookup(T1,T2,ToleranceIn,ToleranceOut)
        ; (T1 = [_|_] ; T2 = [_|_]) ->
            T1 = [A1|R1], T2 = [A2|R2],
            tu_terms(A1,A2,ToleranceIn,Tol),
            tu_terms(R1,R2,Tol,ToleranceOut)
        ;
            functor(T1,N1,A),
            functor(T2,N2,A),
            tolerance_lookup(N1/A,N2/A,ToleranceIn,Tol),
            T1 =.. [_|As1],
            T2 =.. [_|As2],
            tu_terms(As1,As2,Tol,ToleranceOut)
        ).

tolerance_lookup(X,Y,ToleranceIn,ToleranceOut) :-
        ( X = Y ->
            ToleranceOut = ToleranceIn
        ; member(X=YY,ToleranceIn) ->
            Y = YY,
            ToleranceOut = ToleranceIn
        ; member(Y=XX,ToleranceIn) ->
            X = XX,
            ToleranceOut = ToleranceIn
        ;
            ToleranceOut = [X=Y,Y=X|ToleranceIn]
        ).

special(T) :- (var(T) ; number(T) ; T == []).
```

5. Shop

Aikia finally understood her mission in life: to open a furniture shop in all major cities around the world. She started by designing a leaflet, or rather *the leaflet. Superficially, the leaflet was just a linear map of the shop, intended for customers (she preferred to name them clients) to find their way. But to Aikia (and future generations), it was more: the number of islands in the shop (she prefers to name them milieus), their contents, their interconnection and their relation to the outside world would be fixed for* all Aikia shops until the next major comet impact. We have no permission to reproduce the full leaflet here, but a reduced one can be admired in Figure 7.1.

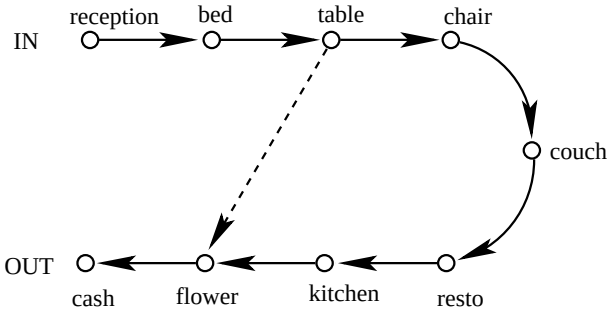


Figure 7.1: Abstract shop layout.

The idea is that the client comes in at IN, and follows the main path, indicated by the solid arrows: from the reception to the milieu with beds, then to the table milieu and so on. However, if the client is in a hurry, she[2] can use a shortcut. Our example has one such shortcut, from the table milieu to the flower milieu (indicated by a dashed arrow). Actually, the direction of the arrows is not important, as clients can retrace their steps (if not from memory, then from the objects they picked up and have put in their carriage).

Her next move was more cunning than a black adder can imagine. She had 3 zillion copies of the leaflet printed on non-recycled paper, the price of wood rose, and the furniture market was left in the shambles. It still is, but ... Aikia managed to open her shops in every worthy place on earth[3]. And, although the leaflet was a fact of life, invariable and ever lasting, the shops looked different from the outside. Figure 7.2 shows two examples of a realization of the leaflet, where both shops consist of 9 cubicles of course. To make sure that the client follows the intended path, there are solid walls between, for instance, the table milieu and the restaurant. But between the couch and chair milieu, there is no barrier (in

[2]Men get nervous breakdowns in Aikia's shops.
[3]And in some unworthy places as well.

the figure represented by a dotted line). A shortcut is realized by a flapping door (drawn as a dashed line in the picture), like between flower and table.

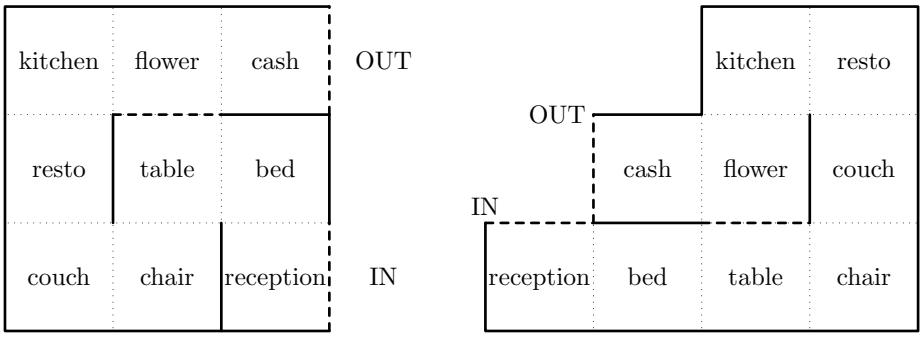


Figure 7.2: Two shop layouts in reality, but according to the leaflet.

This is where you come in[4]: given an empty shop (that is, the arrangement of its cubicles), you need to assign a milieu to each cubicle. So, let us see how the leaflet and an empty shop are specified:

> If there are N cubicles, they have integer names 1 up to N, and the fact `numberofcubes(`N`)` is given.
>
> Cubicle I is next to cubicle J (which means that one can go from one to the other directly), is represented by the fact `nextto(`I`,`J`)`.
>
> For each cubicle I that has an outer wall (and can thus serve as entrance or exit), there is a fact `outerwall(`I`)`

For the left shop in Figure 7.2, the numbering of the cubicles could be as in Figure 7.3.

1	2	3
4	5	6
7	8	9

Figure 7.3: Cubicle numbering.

The given facts for this example are:

[4]No, don't worry, not into the shop, just into the problem.

```
nextto(1,2).
nextto(1,4).
nextto(2,3).
nextto(2,5).
nextto(3,6).
nextto(4,5).
nextto(4,7).
nextto(5,6).
nextto(5,8).
nextto(6,9).
nextto(7,8).
nextto(8,9).

numberofcubes(9).

outerwall(1).
outerwall(2).
outerwall(3).
outerwall(4).
outerwall(6).
outerwall(7).
outerwall(8).
outerwall(9).
```

Every manager knows the leaflet by heart! It is specified as one fact `milieus/1` whose argument is a list of atoms, denoting the main trajectory, and a series of `shortcut(atom1, atom2)` facts.

The above leaflet would be represented by:

```
milieus([reception,bed,table,chair,couch,resto,kitchen,flower,cash]).

shortcut(table,flower).  % there can be several of them
```

Write a predicate `shop/1`, which, assuming you are given the real original leaflet and an empty shop (in the form of the facts above), unifies its argument with one (!, no pun intended) realization of the leaflet, in the form of a list of associations between a cubicle and the milieu which must go in it. For the realization in the left half of figure 7.2, this might be:

```
[1-kitchen,2-flower,3-cash,4-resto,5-table,6-bed,7-couch,8-chair,9-reception]
```

Your realization must take into account that the entrance and exit cubicles are the first and last in the milieus list (which must have an outer wall), and that there are shortcuts. The order in your answer list is not important.

Solution

```
:- use_module(library(lists),[member/2, select/3]).
:- use_module(contestlib,[numlist/3]).

shop(Shop) :-
        numberofcubes(N),
        numlist(1,N,Ns),
        milieus(Milieus),
        shop(Ns,Milieus,[],Shop),
        \+ has_obstructed_shortcut(Shop).

shop([],[],Shop,Shop) :-
        Shop = [ExitI-_|_],
        outerwall(ExitI).
shop(Ns,[Mil|Mils],ShopIn,ShopOut) :-
        select(N,Ns,NewNs),
        check_passage(ShopIn,N),
        NewShopIn = [N-Mil|ShopIn],
        shop(NewNs,Mils,NewShopIn,ShopOut).

check_passage([],I) :- outerwall(I).
check_passage([J-_|_],I) :- no_wall_between(I,J).

no_wall_between(X,Y) :- nextto(X,Y).
no_wall_between(X,Y) :- nextto(Y,X).

has_obstructed_shortcut(Shop) :-
        shortcut(MI,MJ),
        member(I-MI,Shop), member(J-MJ,Shop),
        \+(no_wall_between(I,J)).
```

Contest VIII: 2002 Copenhagen, Denmark

1. K_4

Write a predicate `kay4/1` which draws the famous K_4 graph on the screen. The input to the predicate `kay4/1` is an integral number. Below are the pictures that must appear on the screen for various input:

```
?- kay4(7).

a-----b
|\   /|
| \ / |
|  X  |
| / \ |
|/   \|
c-----d
```

```
?- kay4(6).

a----b
|\  /|
| \/ |
| /\ |
|/  \|
c----d
```

The input can be anything larger than 2.

Hints *Imagine there is an X axis parallel to the (a,b) edge, a Y axis parallel to the (a,c) edge, and let point a have coordinates (1,1). It suffices to describe the mapping from the coordinates (1,1) up to (N,N) to the character that must be written in that position. This mapping is described by the predicate* ***point_contains/4****. We generate the coordinates in the correct order, and write out the character.*

Solution

```
:- use_module(contestlib,[for/3]).

kay4(N) :-
        for(X,1,N),
        for(Y,1,N),
        point_contains(X,Y,N,C),
        write(C),
        fail.

point_contains(1,1,_,C) :- !, C = a.
point_contains(1,N,N,C) :- !, C = 'b\n'.
point_contains(1,_,_,C) :- !, C = '-'.
point_contains(N,1,N,C) :- !, C = c.
point_contains(N,N,N,C) :- !, C = 'd\n'.
point_contains(N,_,N,C) :- !, C = '-'.
point_contains(_,1,_,C) :- !, C = '|'.
point_contains(_,N,N,C) :- !, C = '|\n'.
point_contains(X,Y,N,C) :-
        (X =< N//2 ; (N - X) < N//2), !,
        ( X == Y ->
            C = '\\'
        ; X + Y =:= N + 1 ->
            C = '/'
        ;
            C = ' '
        ).
point_contains(_,Y,N,C) :-
        ( Y =:= N//2 + 1 ->
            C = 'X'
        ;
            C = ' '
        ).
```

2. Bicentered Trees

The distance between two nodes v and u in a graph is the length of a shortest path between v and u, where the length of a path is simply the number of edges in that path. The eccentricity of a node v is the maximal distance between v and any other node in the graph. A node with minimal eccentricity is named a center of the graph. There is a special class of trees which have exactly two centers; these are named ... *bicentered trees* (surprise, surprise!)

Ground Prolog terms are trees. Your task is to write a predicate `bicentered/2` that, given a ground Prolog term as first argument, unifies its second argument with a list of the two centers of the term if the Prolog term is a bicentered tree. If it is not a bicentered tree, the predicate should fail.

The Prolog term could look like: `f(g(a),h(b),i(k(c)))`. The functor (and atom) names of all the nodes in the given term are always different. This term is a bicentered tree, and the list of centers is `[f,i]` or `[i,f]`. So,

```
?- bicentered(f(g(a),h(b),i(k(c))),Centers).
```

answers either `Centers = [f,i]` or `Centers = [i,f]`, and has no other solutions.

Hints *The solution first transforms the Prolog term into a list of edges: this happens in the first goal of* `bicentered/2`*. Once that is done, we treat it as the graph problem. Note that a simple path between two nodes in a tree is also the shortest path: that is why* `path/5` *removes the traversed edges and computes the length of only one path.*

Solution

```
:- use_module(library(lists),[member/2, select/3, last/2]).

bicentered(Term,Centers) :-
        findall(Edge,is_edge(Term,Edge),Edges),
        setof(Node,is_node(Node,Edges),Nodes),
        two_centers(Edges,Nodes,Centers).

is_edge(Term,Edge) :-
        Term =.. [ParentName|Children],
        member(Child,Children),
        (
          Child =.. [ChildName|_],
          Edge = ParentName-ChildName
        ;
          is_edge(Child,Edge)
        ).

is_node(Node,Edges) :- member(Node-_,Edges) ; member(_-Node,Edges).

two_centers(Edges,Nodes,[Center1,Center2]) :-
        setof(Ecc-Node,eccentricity(Edges,Nodes,Ecc,Node),EccNodes),
        EccNodes = [Ecc-Center1,Ecc-Center2|Rest],
        (
          Rest = []
        ;
          Rest = [Ecc2 - _|_],
          Ecc2 > Ecc
        ).

eccentricity(Edges,Nodes,Ecc,Node) :-
        member(Node,Nodes),
        setof(Dist,distance(Node,Nodes,Edges,Dist),Dists),
        last(Dists,Ecc).

distance(Node,Nodes,Edges,NegDist) :-
        member(Node1,Nodes),
        path(Node,Node1,Edges,0,NegDist).

path(Node,Node,_,DistIn,DistOut) :- !, DistIn = DistOut.
path(Node,Node2,Edges,DistIn,DistOut) :-
        (select(Node-Node1,Edges,Edges1) ; select(Node1-Node,Edges,Edges1)),
        DistIn1 is DistIn + 1,
        path(Node1,Node2,Edges1,DistIn1,DistOut).
```

3. Antwerpen

The Towers of Antwerpen problem[1] is a variant of the Towers of Hanoi problem. You start off with a configuration of 3 pegs (say a, b and c), where each peg has N disks. The disks have sizes N down to 1, and a disk is never placed on a disk with strictly smaller size (equal size is OK, otherwise you could not move any disk at all). The disks on peg a are colored black, the disks on peg b are yellow, and the disks on peg c are red[2]. You can move one disk at a time, from one peg to another, but you can never place a disk on a strictly smaller one. Still, you should move the disks such that the black ones go all to peg c, the yellow ones to peg a, and the red ones to peg b. You must do this in the smallest number of moves.

The query is of the form `?- antwerpen(N,MoveList).` where `N` is the given number of disks on each peg. A move is of the form `x->y`, meaning that the topmost disk from peg x moves to peg y. So, a list of moves `[(a->b),(c->b)]` means that the top disk is moved from peg a to b, and then the top disk from c to b.

There is no need to produce a move list of minimal length, but you might want to pay attention to performance.

Hints *The paper by Minsker contains an optimal algorithm, but one cannot expect anyone to reconstruct it under time pressure. It is easy to implement an iterative deepening solution that is guaranteed to give the smallest number of moves, but it is horribly slow. What we present is not optimal, but can serve as a first step in understanding how the optimal solution by Minsker works.*

Suppose we had the following operation already implemented:

> *From the starting configuration, produce a tower on peg a with all the disks, such that the order of the equal disks is always black, yellow, red. See Figures 8.1(a) and 8.1(b) for an illustration.*

We can now switch the pegs b and c, and reverse the construction of that tower. We will then end up in the situation from Figure 8.1(c). We have just described the clause for `exchange/5`*!*

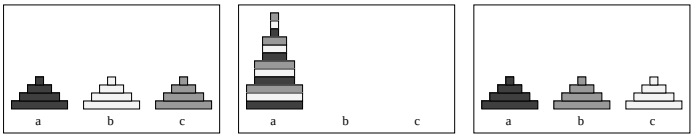
Hence, the problem of exchanging the contents of two pegs is reduced to the problem of constructing a tower with all the disks on one peg. We will break this down in smaller problems.

Suppose there is only one disk (so N equals 1) on each peg. Then, it is easy to make the full tower on a: just move the disk from b to a, and move the disk from c to a.

---

[1] S. Minsker, *The towers of Antwerpen problem*, Information Processing Letters, 38, 107-111, 1991.

[2] The colors of the Belgian flag.



(a) Initial towers

(b) All disks on peg a

(c) After switching b and c, and redistributing

Figure 8.1: Construct the tower, switch b and c, and deconstruct the tower.

Suppose you know how to make full tower with the first N-1 disks of each peg, how would you make a full tower of N disks? The series of pictures in Figure 8.2 shows how:

1. *Make the N-1 tower on peg c*
2. *Move the remaining disk from b to a*
3. *Move the N-1 tower from c to b*
4. *Move the remaining disk from c to a*
5. *Move the N-1 tower from b to a*

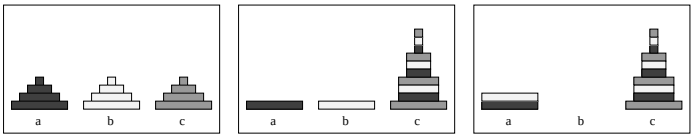
Step i above corresponds to the construction of command sequence C_i in the second clause of `make_tower/5`.

We are now left with solving the problem of moving a full N tower. If N equals 0, this is easy. Suppose we know how to do it for a full N-1 tower, then we can move a full N tower (say from a to b) as follows:

1. *Move the top full N-1 tower on a to b*
2. *Move the two top disks from a to c*
3. *Move the full N-1 tower from b to c*
4. *Move the last disk from a to b*
5. *Move the N-1 tower from c to a*
6. *Move the two disks from c to b*
7. *Move the N-1 tower from a to b*

Step i above, corresponds to the construction of command sequence C_i in the second clause of `move_tower/5`. *Figure 8.3 shows how a tower can be moved.*

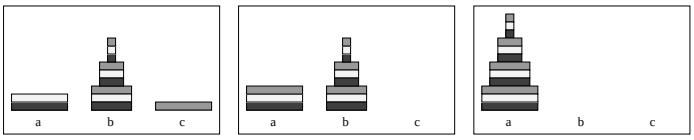
Once we know how to exchange the contents of two pegs, we can go from the initial situation to the required final one by exchanging peg b and c, and then exchanging peg a and c.



(a) Initial towers

(b) Full N-1 tower on c

(c) Moved last disk from b to a

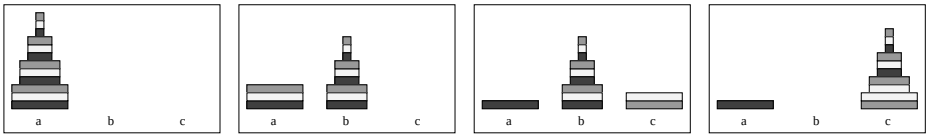


(d) Moved N-1 full tower from c to b

(e) Moved last disk from c to a

(f) Moved N-1 full tower from b to a

Figure 8.2: How to build a full tower.

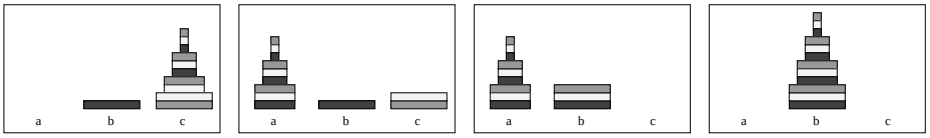


(a) Initial full N tower on a

(b) Full N-1 tower moved to b

(c) Top two disks from a to c

(d) Moved full N-1 tower from b to c



(e) Moved last disk from a to b

(f) Moved full N-1 tower from c to a

(g) Moved two disks from c to b

(h) Moved full N-1 tower from a to b

Figure 8.3: How to move a full tower from a to b.

Solution

```
:- use_module(library(lists),[append/3, flatten/2]).

antwerpen(N,Commands) :-
        exchange(N,b,c,a,Cbc),
        exchange(N,a,c,b,Cac),
        append(Cbc,Cac,Commands).

exchange(N,PegB,PegC,PegA,Commands) :-
        make_tower(N,PegA,PegB,PegC,C1),
        invert_commandsequence(C1,PegB,PegC,[],C2),
        append(C1,C2,Commands).

make_tower(1,PegA,PegB,PegC,Commands) :- !,
        Commands = [(PegB->PegA),(PegC->PegA)].
make_tower(N,PegA,PegB,PegC,Commands) :-
        N1 is N - 1,
        make_tower(N1,PegB,PegA,PegC,C1),
        C2 = [(PegC->PegA)],
        move_tower(N1,PegB,PegC,PegA,C3),
        C4 = [(PegB->PegA)],
        move_tower(N1,PegC,PegA,PegB,C5),
        flatten([C1,C2,C3,C4,C5],Commands).

move_tower(1,PegA,PegB,PegC,Commands) :- !,
        Commands = [(PegA->PegC),(PegA->PegC),
                    (PegA->PegB),(PegC->PegB),(PegC->PegB)].
move_tower(N,PegA,PegB,PegC,Commands) :-
        N1 is N - 1,
        move_tower(N1,PegA,PegB,PegC,C1),
        C2 = [(PegA->PegC),(PegA->PegC)],
        move_tower(N1,PegB,PegC,PegA,C3),
        C4 = [(PegA->PegB)],
        move_tower(N1,PegC,PegA,PegB,C5),
        C6 = [(PegC->PegB),(PegC->PegB)],
        move_tower(N1,PegA,PegB,PegC,C7),
        flatten([C1,C2,C3,C4,C5,C6,C7],Commands).

invert_commandsequence([],_,_,In,In).
invert_commandsequence([(A->B)|R],X,Y,In,Out) :-
        switch(A,X,Y,A1), switch(B,X,Y,B1),
        invert_commandsequence(R,X,Y,[(B1->A1)|In],Out).

switch(A,A,B,B) :- !.
switch(A,B,A,B) :- !.
switch(A,_,_,A).
```

4. Shunt

You are an engine driver, and your train is specified in the form $[c_1,c_2,c_3,\ldots,c_n]$, where the c_i are wagons. The locomotive of the train is at the c_1 side of the train, and is otherwise not explicitly represented. You are in a shunting station with a given train, and your task is to transform the train into a given permutation of the initial train. The station has two shunting tracks on which you can push and pop any number of wagons. See how the train `[a,b,c]` is transformed to `[b,c,a]` in the series of pictures in Figure 8.4.

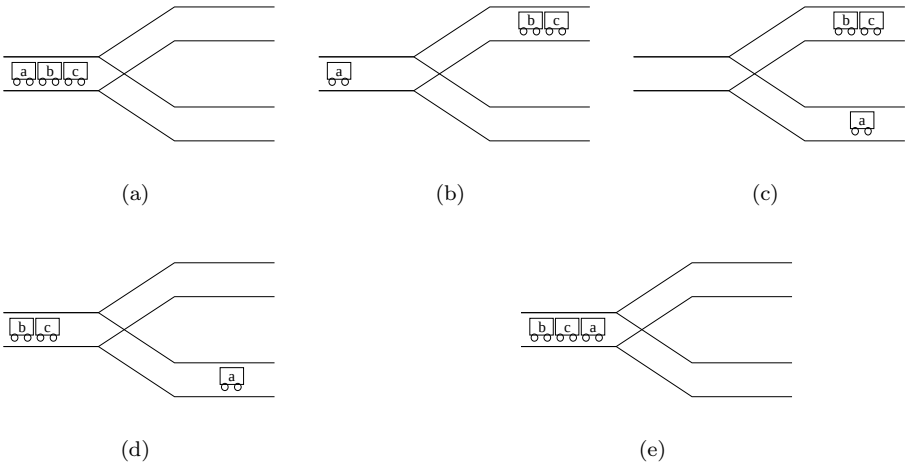


Figure 8.4: `[a,b]` to `[b,a]` in 4 moves.

This series of pushes and pops is represented by the following list:

```
[one/2, two/1, one/ -2, two/ -1]
```

This list means: push two wagons on rail `one`; push one wagon on rail `two`, pop two wagons from rail `one`; pop one wagon from rail `two`. The first wagon pushed must go to rail `one`. Other series of pushes and pops might result in the same end configuration, but we are interested in the shortest one only.

Your task is to write a predicate `shunt/3`, which can be called as

```
?- shunt([a,b,c],[b,c,a],L).
```

and which could answer with the above list, since that is a shortest sequence of pushes and pops to transform the `[a,b,c]` train into the `[b,c,a]` train.

Hints *We treat this as a graph search problem, where a node is a state of the rails (i.e. which wagons are on it), and edges between nodes are between states that can be obtained from another by a push or a pop. We use iterative deepening on the length of the paths constructed, and so the first path we find is the shortest, and we don't need to check for loops in the path.*

Note that in a shortest solution, the successive moves alternate between the tracks.

Solution

```
:- use_module(library(lists), [append/3]).

shunt(In,Out,Moves) :-
        StartNode = node(In,[],[]),
        EndNode = node(Out,[],[]),
        LastTrack = two,
        length(Moves,_),
        path(Moves,StartNode,EndNode,LastTrack),
        !.

path([],Node,Node,_).
path([OtherTrack/N|Moves],Node,EndNode,LastTrack) :-
        opposite(LastTrack,OtherTrack),
        edge(Node,NextNode,OtherTrack,N),
        path(Moves,NextNode,EndNode,OtherTrack).

edge(node(X1,A1,B), node(X2,A2,B), one, N) :- pop(A1,A2,X1,X2,N).
edge(node(X1,A1,B), node(X2,A2,B), one, N) :- push(A1,A2,X1,X2,N).
edge(node(X1,A,B1), node(X2,A,B2), two, N) :- pop(B1,B2,X1,X2,N).
edge(node(X1,A,B1), node(X2,A,B2), two, N) :- push(B1,B2,X1,X2,N).

pop(A1,A2,X1,X2,N) :-
        append(F,A2,A1),
        F \== [],
        append(X1,F,X2),
        length(F,M),
        N is -M.

push(A1,A2,X1,X2,N) :-
        append(X2,F,X1),
        F \== [],
        append(F,A1,A2),
        length(F,N).

opposite(one,two).
opposite(two,one).
```

5. Mamadee

You are the programmer for a malicious maze designer (MMD[3]) who likes designing mazes from which no escape is possible (unless by going back). But the MMD also has a handicap: he needs to use partially pre-constructed mazes (PPCM), which means that blocking walls can be put up only in predefined slots, and that the number of walls that can be used is limited. These PPCMs also have a predefined set of entrances, and one predefined exit (because the doors open differently). So, when the new PPCM model comes out, the MMD wants to place the walls such that the least number of entrances allow for finding the exit. You are to solve a part of this problem: for a given entry and exit, a given maximal number of walls and given slots you can put the walls in, determine whether it is possible to place the walls in such a way that there is no path from the entry to the exit. So, write a predicate `mamadee/5`, which will be typically called as

```
?- mamadee(5,(1,4),(3,1),7,
       [wall((0,2),(1,2)), wall((1,2),(2,2)), wall((2,2),(2,3)),
        wall((2,2),(3,2)), wall((3,2),(3,1)), wall((4,1),(5,1)),
        wall((1,3),(1,4)), wall((2,4),(2,5)), wall((2,0),(2,1))],
       BlockingListOfWalls).
```

In this query, the first argument is the size of the maze (the maze is always rectangular); the second and third argument represent the entry and the exit of the maze; the fourth argument is the maximal number of walls that can be used; the fifth argument is a list with the specification of wall slots in the maze. If it is possible to place no more of these walls than the number of allowed walls (specified in argument 4) and such that the way from the entry to the exit is blocked, then the last argument must be unified with a shortest list of walls (from argument 5) doing so. The order in argument 5 is not important, but it should not contain duplicates, nor unnecessary walls. If there is no way to prevent the existence of a path from the entry to the exit, the query above must (finitely) fail.

The maze with its wall slots specified in the above query, is shown in Picture 8.5. The entry (and exit) is given as a tuple of coordinates of the entry (or exit) square. Squares have coordinates from $(1, 1)$ to (N, N) where N is the size of the maze. Moreover, the exit is guaranteed to be different from the entry.

A wall slot always has length one and is given as `wall/2` term, which contains two tuples of coordinates, giving the begin and end point of the wall. For instance, the term `wall((0,2),(1,2))` blocks the direct passage from square $(1, 2)$ to square $(1, 3)$. You cannot count on a specific order in the arguments of the `wall/2` term.

This example query fails.

[3]Pronounced *ma-ma-dee*.

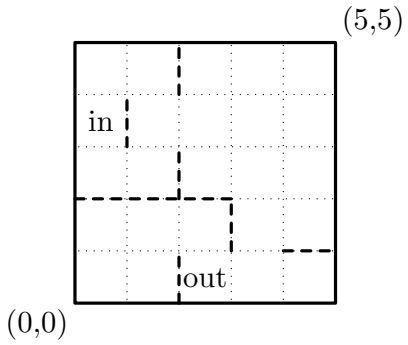


Figure 8.5: A 5x5 maze with entry, exit and possible wall slots.

Hints *There is one annoying thing about the problem statement: the naming of squares in the maze does not seem to match up with the one of the walls. So, some code is needed to compute a wall from two adjacent squares. This also happens in the predicate* `step/4`.

Instead of computing a minimal set of blocking walls (i.e. without redundant walls), we search for a smallest one: we use iterative deepening on the size of the output.

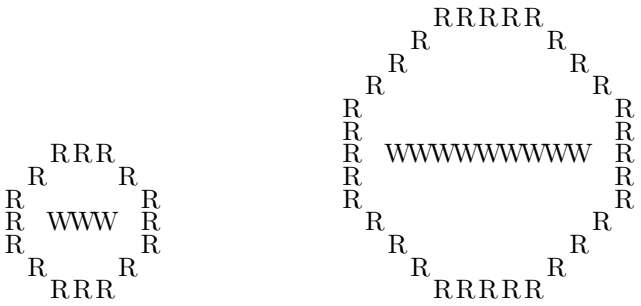
Solution

```
:- use_module(library(lists),[member/2]).
:- use_module(contestlib,[sublist/2, for/3]).

mamadee(MazeSize,Entry,Exit,MaxWalls,Walls,Blocking) :-
        for(I,1,MaxWalls),
        length(Blocking,I),
        sublist(Walls,Blocking),
        Been = [Entry],
        \+ exists_path(Entry,Exit,MazeSize,Been,Blocking),
        !.

exists_path(Entry,Exit,MazeSize,Been,Blocking) :-
        ( Exit == Entry ->
            true
        ;
            step(Entry,NewEntry,MazeSize,Blocking),
            \+ member(NewEntry,Been),
            exists_path(NewEntry,Exit,MazeSize,[NewEntry|Been],Blocking)
        ).

step((A,B),(X,Y),MazeSize,Blocking) :-
        (
          X = A, Y is B + 1,
          C is A - 1, D = B,
          U = A, V = B

        ;
          X = A, Y is B - 1,
          C is X - 1, D = Y,
          U = X, V = Y
        ;
          Y = B, X is A + 1,
          C = A, D is B - 1,
          U = A, V = B
        ;
          Y = B, X is A - 1,
          C = X, D is Y - 1,
          U = X, V = Y
        ),
        1 =< X, X =< MazeSize,
        1 =< Y, Y =< MazeSize,
        \+ member(wall((C,D),(U,V)),Blocking),
        \+ member(wall((U,V),(C,D)),Blocking).
```

Contest IX: 2003 Mumbay, India

1. Stop

Write a predicate `stop/1` which draws in the screen a stop sign. A stop sign is an 8-sided board, all red, except for a white horizontal strip in the middle. A size 3 stop sign - drawn by the query `?- stop(3).` - looks like Figure 9.1(a) on the screen.



(a) $N = 3$ (b) $N = 5$

Figure 9.1: Stop Signs.

As you can see, only the contour is red (denoted by the capital letter `R`), and the horizontal white strip is indicated by the capital letter `W`.

As another example, `?- stop(5).` produces Figure 9.1(b).

The input to `stop/1` is larger than 2 and odd.

Hints *Figure 9.2 shows a stop sign of size $N = 5$, and the distances that are important in the program.*

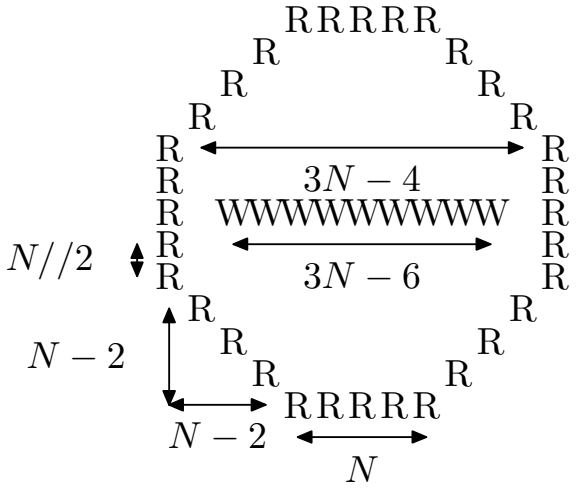


Figure 9.2: $N = 5$ and related sizes for the program.

Solution

```
:- use_module(contestlib,[writeN/2]).

stop(N) :-
        first_last_line(N),
        slope(N - 2,N - 2,N,-1),
        vertical(N//2,3*N - 4),
        mid(3*N - 6),
        vertical(N//2,3*N - 4),
        slope(N - 2,1,3*N - 6,1),
        first_last_line(N).

first_last_line(N) :-
        N1 is N - 1,
        space(N1),
        red(N),
        nl.

slope(M,SpaceBefore,SpaceMiddle,I) :-
        ( M > 0 ->
            space(SpaceBefore), red(1), space(SpaceMiddle), red(1), nl,
            M1 is M - 1,
            SpaceBefore1 is SpaceBefore + I,
            SpaceMiddle1 is SpaceMiddle - I - I,
            slope(M1,SpaceBefore1,SpaceMiddle1,I)
        ;
            true
        ).

mid(N) :- red(1), space(1), white(N), space(1), red(1), nl.

vertical(N,Space) :-
        ( N > 0 ->
            red(1), space(Space), red(1), nl,
            N1 is N - 1,
            vertical(N1,Space)
        ;
            true
        ).

red(N) :- writeN(N,'R').

space(N) :- writeN(N,' ').

white(N) :- writeN(N,'W').
```

2. Cheater

N tourists have booked a day tour which includes *one* trip in a gondola. There is only one gondola, operating all day. These tourists can board any time the gondola is at the departure spot. After the trip, everyone must leave the gondola. Each tourist has a sticker on his/her shirt, so that the gondola driver can immediately see whether someone is entitled to board the gondola. The gondola driver is payed by the number of people who have actually taken 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. However, 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 other tourists (s)he saw on the same gondola trip. The cheating tourist knows this of course, so (s)he mentions the union of the people (s)he saw on both boat trips. The result is a sequence of Prolog facts `saw/2` (which should be interpreted as their commutative closure). Examples will follow later.

Because the tourist office foresees that it will have to deal with this problem regularly, it lets you - the poor programmer - write a predicate `cheater/1`, which unifies its argument with the name of the cheater. There are a couple more things you know: (1) there is at most one cheater, and the cheater cheated just once; (2) the gondola does not make trips with just one tourist; (3) 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,christa).
saw(beata,christa).
saw(donna,eva).
?- cheater(C).
C = gondola_driver
```

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

Hints *This problem is a much simplified version of the "Berge mystery story", usually solved by the notion of interval graphs. Without mentioning the graph theory context in the problem statement above, about half of the people trying to solve this problem come up with a program that tries to find two complete subgraphs of the `saw/2` graph with exactly one node in common. The other people come up with the direct translation of the logic involved: the cheater is someone who saw two different people who did not see each other; if no such person exists, the gondola driver is the cheater.*

Solution

```
cheater(Cheater) :-
        saw_each_other(Cheater, Person1),
        saw_each_other(Cheater, Person2),
        Person1 \== Person2,
        \+ saw_each_other(Person1,Person2),
        !.
cheater(gondola_driver).

saw_each_other(Person1,Person2) :- saw(Person1,Person2).
saw_each_other(Person1,Person2) :- saw(Person2,Person1).
```

3. Spanning Spider

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 have to 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, and you may assume that the graph is connected. Three example `edge/2` sets with the query and answer are shown below:

```
edge(d,a).
edge(d,b).
edge(d,c).
edge(d,e).
edge(a,x).
edge(b,y).
edge(c,z).
edge(a,b).
edge(y,c).
?- spanspid.
Yes
```

```
edge(a,e).
edge(b,e).
edge(e,f).
edge(f,d).
edge(f,c).
?- spanspid.
No
```

```
edge(a,b).
edge(b,c).
edge(a,c).
edge(d,a).
edge(d,b).
edge(d,c).
?- spanspid.
Yes
```

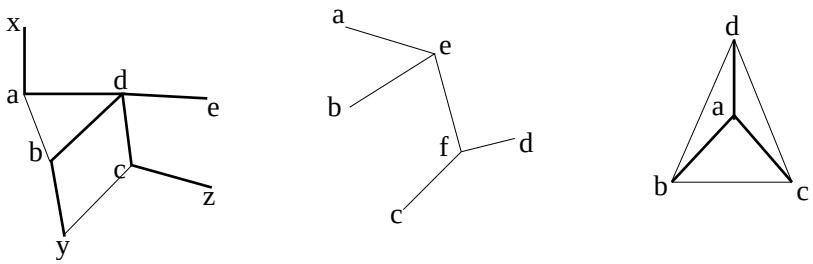


Figure 9.3: The three graphs: their spanning spider is in bold (if it exists).

Hints *The program generates all subgraphs consisting of one edge less than the number of nodes in the graph, while covering all nodes. Such a subgraph is always a spanning tree. The program then tests for each of these spanning trees whether it has at most one node with degree 3 or more.*

Solution

```
:- use_module(library(lists), [member/2]).
:- use_module(contestlib, [sublist/2]).

spanspid :-
        findall(edge(X,Y),edge(X,Y),AllEdges),
        setof(X,is_node_in(X,AllEdges),AllNodes),
        AllNodes = [_|OneLess],
        length(OneLess,Len),
        length(SubGraph,Len),
        sublist(AllEdges,SubGraph),
        setof(X,is_node_in(X,SubGraph),NodesSubGraph),
        NodesSubGraph = AllNodes,
        findall(Node,bigdegree_node(SubGraph,Node,AllNodes),BigDegrees),
        sort(BigDegrees,BigDegreesUnique),
        BigDegreesUnique \= [_,_|_],
        !.

bigdegree_node(SubGraph,Node,Nodes) :-
        member(Node,Nodes),
        setof(Connected,connected(Node,Connected,SubGraph),ListConnected),
        ListConnected = [_,_,_|_].

connected(N1,N2,Edges) :- member(edge(N1,N2),Edges).
connected(N1,N2,Edges) :- member(edge(N2,N1),Edges).

is_node_in(Node,Edges) :- connected(Node,_,Edges).
```

4. Longest Decreasing Subsequence

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 of this theorem start by '*Take a longest decreasing subsequence ...*', and this leads to the following problem: write a predicate `ld/2` that, 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 (and its answer) is:

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

Another correct answer would be `[6,4,1]` and there are some more.

The input list is never empty.

Hints *The optimal algorithm for this problem is quadratic in the length of the input (try it!). Our solution is a terribly inefficient, yet simple and quick to implement generate and test method. It is exactly the thing one would write under time pressure, and it makes good use of the libraries.*

Solution

```
:- use_module(contestlib,[sublist/2]).
:- use_module(library(lists),[reverse/2, last/2]).

ld(List,LongestDecrSub) :-
        setof(Len-DecrSub,decreasing_sublist(List,Len,DecrSub),AllDecrSubs),
        last(_-LongestDecrSub,AllDecrSubs).

decreasing_sublist(List,Len,DecrSub) :-
        sublist(List,DecrSub),
        sort(DecrSub,DecrSubSorted),
        reverse(DecrSubSorted,DecrSub),
        length(DecrSub,Len).
```

5. Cellular Automaton

An elementary cellular automaton (e-celauton) consists of an infinite array (both directions) of cells each having one of two possible colors (black and 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 "A New Kind of Science" by S. Wolfram .

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:

```
oxo      oxx
 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 neighbor remains black.

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

```
ooo
 o
```

The computing power of e-celautons is amazing (well, that is, if you amaze easily, e.g. by universal Turing machines). 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:

```
  oxo
 oxoxo
oxoxoxo
```

The first line (or generation) contains only one black cell. Because of (*) and the bleak rule, we do not need to represent the infinite white parts at the left and the right. In this picture, 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 ...

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). For example:

```
[    [o,x,o],
   [o,x,o,x,o],
 [o,x,o,x,o,x,o]]
```
(9.1)

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

```
oxo      oox      xoo      xox
 o        x        x        x
```

plus the bleak rule and some 11 other rules which are of no consequence for the evolution shown.

These 4 rules + the bleak rule are represented as follows:

```
[[o,o,o,o],[o,x,o,o], [o,o,x,x], [x,o,o,x], [x,o,x,x]]        (9.2)
```

The order of the rules does not matter here.

The input to the predicate you have to write is a list as in 9.1. The output is a list of rules as in 9.2.

For 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 and so is any permutation of the `Rules` list.

The `Rules` list must be the minimal required set of rules needed to cause the given sequence of generations. 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 finitely fail. For instance:

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

fails.

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

As before, a generate and test approach will do. The program collects every locally used rule in a list, and if they are not in conflict, that is the result. Conflict testing (predicate `consistent/1`*) is made easy by first sorting the locally used rules (which also removes duplicates).*

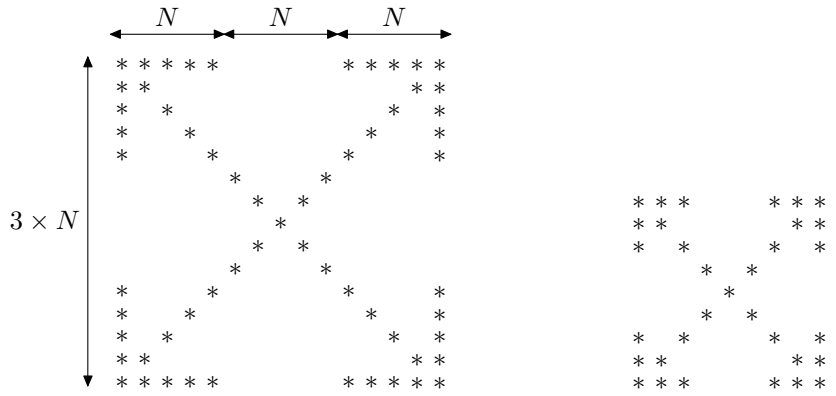
Solution

```
:- use_module(library(lists), [append/3]).

celauton(Evolution,Rules) :-
        setof(Rule, needed_rule(Evolution,Rule), Rules),
        consistent(Rules).

needed_rule([T1,Time2|_],Rule) :-
        append([o,o|T1],[o,o],Time1),
        needed_rule(Time2,Time1,Rule).
needed_rule([_|Evolution],Rule) :-
        needed_rule(Evolution,Rule).

needed_rule([X|_],[A,B,C|_],[A,B,C,X]).
needed_rule([_|Time2],[_|Time1],Rule) :-
        needed_rule(Time2,Time1,Rule).

consistent([_]).
consistent([[A1,B1,C1,_],[A2,B2,C2,_]|R]) :-
        [A1,B1,C1] \== [A2,B2,C2],
        consistent([[A2,B2,C2,_]|R]).
```

Contest X: 2004 St-Malo, France

1. Cross

Write a predicate `cross/1`, for which `cross(N)` draws a cross figurer.



(a) $N = 5$ (b) $N = 3$

Figure 10.1: Example output of `cross/1`

In Figure 10.1(a) you see the cross for N=5 with arrows indicating the size, and in Figure 10.1(b) you see the cross for N=3.

Your program should work for any odd number `N` between 3 and 23. Do not add any redundant space characters.

Hints *The structural decomposition shown in Figure 10.2 for the cross of size* 5 *is reflected in the solution.*

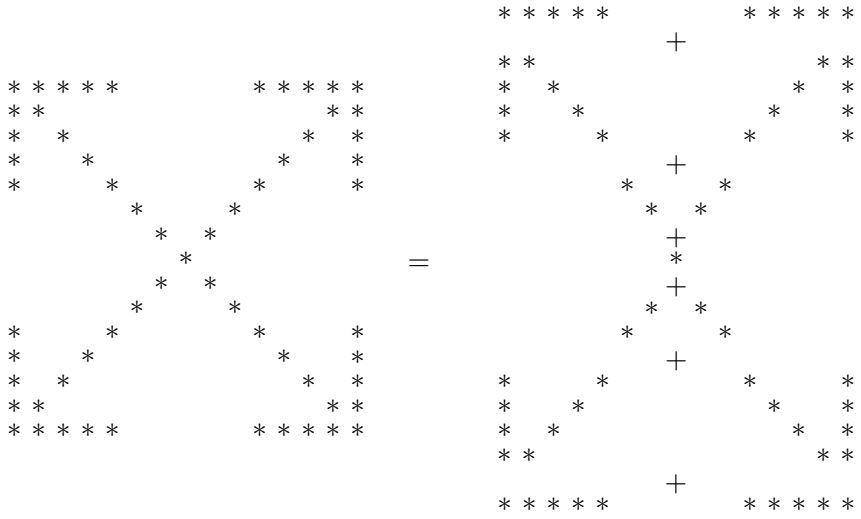


Figure 10.2: Structural decomposition of the cross.

Solution

```
:- use_module(contestlib, [writeN/2]).

cross(N) :-
        hor_line(N),
        N1 is N - 1,
        B1 is N1 // 2,
        Middle is N1 + B1,
        down(N,'*',0,N1),
        down(N,' ',N1,B1),
        blanks(1), blanks(Middle), stars(1), nl,
        up(N,' ',Middle,B1),
        up(N,'*',N1,N1),
        hor_line(N).

hor_line(N) :- stars(N), blanks(N), stars(N), nl.

down(_,_,_,0) :- !.
down(N,Delimiter,Outer,Lines) :-
        write_line(Delimiter,Outer,N),
        Outer1 is Outer + 1,
        Lines1 is Lines - 1,
        down(N,Delimiter,Outer1,Lines1).

up(_,_,_,0) :- !.
up(N,Delimiter,Outer,Lines) :-
        Outer1 is Outer - 1,
        write_line(Delimiter,Outer1,N),
        Lines1 is Lines - 1,
        up(N,Delimiter,Outer1,Lines1).

write_line(Delimiter,OuterSpace,N) :-
        write(Delimiter),
        blanks(OuterSpace),
        stars(1),
        InnerSpace is 3*N - 4 - 2*OuterSpace,
        blanks(InnerSpace),
        stars(1),
        blanks(OuterSpace),
        write(Delimiter),
        nl.

blanks(N) :- writeN(N,' ').

stars(N)  :- writeN(N,'*').
```

2. Station

Belgium is a railway country. On the 5th of May 1835, the first train on the mainland of Europe connected the capital Brussels with the provincial city of Mechelen. Many more lines and tracks have followed, and turned Belgium into a country with a very dense railway net. Trains are always late, and there is nothing we can do about that.

However, we *can* live in any part of the country we like and still be able to go to work in any other part without wasting more time than, say, a commuter on Long Island who works in Manhattan, or a Ph.D. student who lives at the outskirts of Melbourne and has an office at Melbourne Uni.

Since proximity to public transport is not an issue, other preferences determine where to live in Belgium. Write a predicate `station/1` that unifies its argument with the best station (and hence the best city, because every city has just one station). The predicate only has to succeed once if there is a best station, or fail finitely if there is none.

Small cities are cities with at most two direct connections to other cities. Conversely, big cities have more than two direct connections.

A good city, is a small city that is equally close to the closest two big cities, each on a different line leaving the city. This is for example convenient for shopping. The best city is the good city that is closest (among all good cities) to its two closest big cities, as we do not want to go too far for our shopping sprees.

The railway network is given as `rail/2` facts. All rails are bi-directional and equally long. The distance between two cities is the length of the shortest path between the cities.

Here are some examples:

```
rail(brussel,mechelen).
rail(brussel,antwerpen).
rail(brussel,gent).
rail(antwerpen,mechelen).
rail(antwerpen,gent).
rail(gent,brugge).
```

```
?- station(X).
X = mechelen
```

```
rail(brussel,charleroi).
rail(brussel,haacht).
rail(haacht,mechelen).
rail(mechelen,berchem).
rail(berchem,antwerpen).
rail(brussel,boom).
rail(boom,antwerpen).
rail(antwerpen,turnhout).
```

```
?- station(X).
X = boom
```

Hints *The solution uses a straightforward generate and test approach. It does not fully exploit the properties of big and small cities.*

Solution

```
:- use_module(library(lists), [member/2]).

station(Station) :-
        setof(Distance-City,good_city(City,Distance),[_-Station|_]).

good_city(City,Distance) :-
        small_city(City),
        setof(big_city(Distance,BigCity,OutRail),
                reachable_big_city(City,BigCity,Distance,OutRail),BigCities),
        BigCities = [big_city(Distance,City1,OutRail1)|Rest],
        Rest = [big_city(Distance,City2,OutRail2)|_],
        City1 \== City2,
        OutRail1 \== OutRail2.

reachable_big_city(City,BigCity,Distance,OutRail) :-
        rail_c(City,OutRail),
        Visited = [City],
        reachable(OutRail,BigCity,Visited,Distance),
        big_city(BigCity).

reachable(City,City,Visited,Distance) :- length(Visited,Distance).
reachable(City,TargetCity,Visited,Distance) :-
        rail_c(City,City1),
        \+ member(City1,Visited),
        reachable(City1,TargetCity,[City|Visited],Distance).

big_city(City) :-
        rail_c(City,CityA),
        rail_c(City,CityB),
        rail_c(City,CityC),
        CityA \== CityB,
        CityA \== CityC,
        CityB \== CityC.

small_city(City) :-
        rail_c(City,_),
        \+ big_city(City).

rail_c(From,To) :- rail(From,To).
rail_c(From,To) :- rail(To,From).
```

3. Turtle

Logo and Basic both have their turtle graphics drawing instructions. It was time that Prolog becomes more accessible for children, so we implemented a similar turtle system in Prolog. The system has the following specifications:

> Initially the turtle starts at coordinate $(0, 0)$ in a 2-D plane, facing $(0, \infty)$.
>
> Wherever the turtle goes, it leaves a trail of ink.
>
> Since Prolog is a declarative language, the turtle follows only simple commands:
>
> - `step`: the turtle performs a unit step in the direction it is facing.
> - `rotate`: the turtle turns 90 clockwise.
>
> The turtle receives all its commands in a list and processes them sequentially from the first to the last.

Now, we want to encourage our little users to become compiler writers, so we want to instill them with a notion of optimality.

Write a predicate `turtle/2`, which will be called with a list of commands as the first argument, and which unifies the second argument with the shortest possible list of commands that would have the turtle make the same drawing as the input list.

Hints *The program can be solved using generate and test. Programs of increasing length are generated, until one is found that draws the same figure as the original program. A canonical representation is handy for comparing equality of drawings.*

Solution

```
turtle(Program,ShortProgram):-
        Coordinate = (0,0),
        Orientation = (1,0),
        canonical_drawing(Program,Coordinate,Orientation,Drawing),
        length(ShortProgram,_),
        canonical_drawing(ShortProgram,Coordinate,Orientation,Drawing),
        !.

canonical_drawing(Program,Coordinate,Orientation,Drawing) :-
        generate(Program,Coordinate,Orientation,ADrawing),
        sort(ADrawing,Drawing).

generate([],_,_,[]).
generate([rotate|Program],Coordinate,(DX,DY),O) :-
        generate(Program,Coordinate,(DY,-DX),O).
generate([step|Program],(X,Y),(DX,DY),Drawing) :-
        X1 is X + DX,
        Y1 is Y + DY,
        Drawing = [(X,Y)-(X1,Y1),(X1,Y1)-(X,Y)|RestDrawing],
        generate(Program,(X1,Y1),(DX,DY),RestDrawing).
```

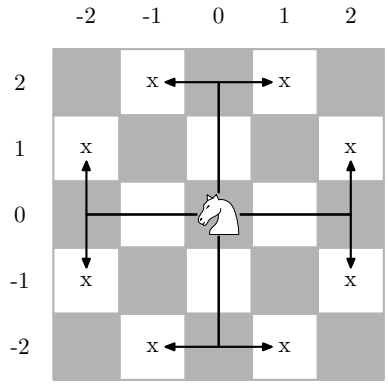
4. Knights

The N-queens problem is a well-established programming problem: fit N queens on an $N \times N$ board without any one queen attacking another. Also the M-queens problem has been tackled before on page 95: fit as many queens as possible on an $M \times M$ board.

Enough of those stuffy old queens! Time for some action! Time for whinnying horses, the stench of blood, battle clamor and steel against steel... Time for the knights to ride out!

Write a predicate `knights/2`, which will be called with the first argument the size N of the board, and which unifies its second argument with the maximum number of knights that fit on a $N \times N$ chess board, without any knight attacking another.

Remember, a knight attacks any position that is two steps in one direction and one step to the side, as illustrated in the following figure:



Hints *The program recursively computes for each square the maximum of assignable knights for the remaining squares, where the square is either empty or contains a knight (if it is not attacked).*

Do not spoil the fun by simply implementing the formula:

$$knights(N) = \begin{cases} N^2 & , N \leq 2 \\ \left\lceil \frac{N^2}{2} \right\rceil & , N > 2 \end{cases}$$

Solution

```
:- use_module(library(lists), [member/2]).
:- use_module(contestlib, [for/3]).

knights(N,Max) :-
        PlacedKnights = [],
        findall((X,Y),(for(X,1,N),for(Y,1,N)),Squares),
        knights(Squares,PlacedKnights,Max).

knights([],PlacedKnights,Max) :-
        length(PlacedKnights,Max).
knights([(X,Y)|Squares],PlacedKnights,Max) :-
        ( is_attacked(X,Y,PlacedKnights) ->
            knights(Squares,PlacedKnights,Max)
        ;
            knights(Squares,[knight(X,Y)|PlacedKnights],Max1),
            knights(Squares,PlacedKnights,Max2),
            Max is max(Max1,Max2)
        ).

is_attacked(X,Y,PlacedKnights) :-
        ( NX is X - 1, NY is Y - 2
        ; NX is X - 1, NY is Y + 2
        ; NX is X + 1, NY is Y - 2
        ; NX is X + 1, NY is Y + 2
        ; NX is X - 2, NY is Y - 1
        ; NX is X - 2, NY is Y + 1
        ; NX is X + 2, NY is Y - 1
        ; NX is X + 2, NY is Y + 1
        ),
        member(knight(NX,NY),PlacedKnights).
```

5. 3-D

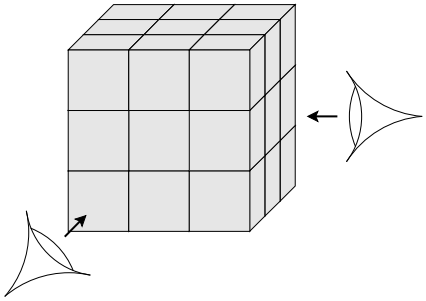
The last question is a practical experiment in genetic evolution. We ask ourselves whether programmers who work on a flat screen all day long still retain the three dimensional visual skills that were essential for their mammoth-hunting ancestors. Or will we just find out who has been playing too much 3D-tetris?

The problem at hand is fairly simple. Consider a large cube consisting of $3 \times 3 \times 3$ small cubes. Every small cube has just one color, but different small cubes can have different colors. Now we take away zero or more of the small cubes and present you with two *adjacent* side views. With these two side views only, you cannot possibly compute how many small cubes remain in the large cube, but you can make a conservative guess.

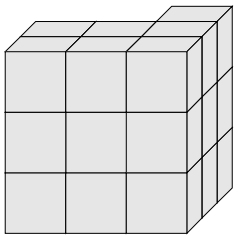
Write a predicate `threed/3` that, given the two views as first arguments, unifies the third argument with the maximum number of small cubes that can be in the large cube. A view is a list of rows, and a row is a list of color of small cubes. A color is simply represented by an atom. The atom `*` is used if you can see straight through the large cube. The first view's rightmost column touches the second view's leftmost column.

For example:

```
?- threed(
        [[a,a,a],
         [a,a,a],
         [a,a,a]],
        [[a,a,a],
         [a,a,a],
         [a,a,a]], MaxCubes).
MaxCubes = 27
```

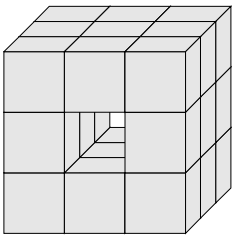


An example of a non-maximal cube for this query (containing only 25 small cubes) looks like this:

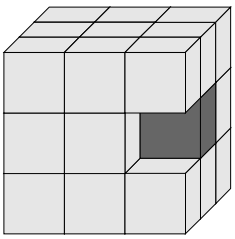


Two more examples:

```
?- threed(
        [[a,a,a],
         [a,*,a],
         [a,a,a]],
        [[a,a,a],
         [a,a,a],
         [a,a,a]], MaxCubes).
MaxCubes = 24
```



```
?- threed(
        [[a,a,a],
         [a,a,b],
         [a,a,a]],
        [[a,a,a],
         [a,b,a],
         [a,a,a]], MaxCubes).
MaxCubes = 26
```



Hints *The problem can be solved for each horizontal slice separately. When looking at one slice from above, you see the small cubes as*

```
A B C
D E F
G H I
```

and the two views (arguments to `threed_slice`*) are from the* `GHI` *and* `IFC` *side. We name the sequence* `GDA` *a* tube*: either* `G` *has a color (and* `DA` *are not visible) or* `G` *is not there (so you see through and then we must consider what happens to the tube* `DA`*). The names of the small cubes (*`A` *up to* `I`*) are used in the program as Prolog variables: they can be unified with a color (which can be any atom), or with the term* `color(none)`*, or remain free if their color is immaterial (because the cube is not visible).*

Solution

```
:- use_module(library(lists),[member/2, last/2]).

threed([View1Slice1,View1Slice2,View1Slice3],
       [View2Slice1,View2Slice2,View2Slice3],Max) :-
        threed_slice(View1Slice1,View2Slice1,Max1),
        threed_slice(View1Slice2,View2Slice2,Max2),
        threed_slice(View1Slice3,View2Slice3,Max3),
        Max is Max1 + Max2 + Max3.

threed_slice(View1,View2,Max) :-
        findall(M,consistent_slice(View1,View2,M),Ms),
        sort(Ms,L),
        last(L,Max).

consistent_slice([Color1,Color2,Color3],[Color4,Color5,Color6],M) :-
        visible_color([G,D,A],Color1),
        visible_color([H,E,B],Color2),
        visible_color([I,F,C],Color3),
        visible_color([I,H,G],Color4),
        visible_color([F,E,D],Color5),
        visible_color([C,B,A],Color6),
        count_cubes([A,B,C,D,E,F,G,H,I],0,M).

visible_color(Cubes,Color) :-
        ( Color == (*) ->
            see_through(Cubes)
        ;
            first_visible_is(Cubes,Color)
        ).

see_through([]).
see_through([color(none)|R]) :- see_through(R).

first_visible_is([Color|_],Color).
first_visible_is([color(none)|R],Color) :- first_visible_is(R,Color).

count_cubes(Cubes,_,N) :-
        findall(C,(member(C,Cubes), C \== color(none)),Cs),
```

The Contest Library

This appendix contains a few predicates that are used in more than one solution: depending on the Prolog system you use, some of them can be found in the libraries.

```
:- module(contestlib,
          [sublist/2,
           write_elements/1,
           writeN/2,
           for/3,
           numlist/3,
           int_width/2,
           write_int/2,
           map/3]).

sublist([],[]).
sublist([X|R],[X|S]) :- sublist(R,S).
sublist([_|R],S) :- sublist(R,S).

write_elements([]).
write_elements([X|R]) :- write(X), write_elements(R).

writeN(N,C) :-
        ( N > 0 ->
            M is N - 1,
            write(C),
            writeN(M,C)
        ;
            true
        ).

for(I,I,J) :- I =< J.
for(K,I,J) :- I < J,
        I1 is I + 1,
        for(K,I1,J).
```

```
numlist(I,J,List) :-
        ( I =< J ->
            List = [I|Rest],
            I1 is I + 1,
            numlist(I1,J,Rest)
        ;
            List = []
        ).

int_width(N,Width) :-
        ( N > 9 ->
            M is N // 10,
            int_width(M,WidthM),
            Width is WidthM + 1
        ;
            Width = 1
        ).

write_int(Int,Width) :-
        int_width(Int,WidthInt),
        Spaces is Width - WidthInt,
        writeN(Spaces,' '),
        write(Int).


map([],_,[]).
map([X|R],C,[CX|CR]) :-
        call(C,X,CX),
        map(R,C,CR).
```