Prolog Programming Contest: 9 September, Porto

Prologue

There are 6 problems. The name of the file that contains your submitted solution must be the same as given in the header of the problem - this file must be readable for the organisers. So, for the first problem, you make a file named `carpet.pl`.
The *input* to your program is in the form of one or more arguments to the predicate you must write, or as some given facts: do not put such given facts in any submission.
Each correct submission (at most one for each problem) earns you one point.
Efficiency of your programs is not important, but if your program fails to finish in a reasonable time, it will be considered incorrect. You can earn a time bonus for problems bitrev and optisort - it is not cpu performance related: see the explanation there.
Do not start predicate names with iclp07 or test - never use the dynamic database or other similar global stuff built-in predicates!

1 Carpet *(carpet.pl)*

Sierpinski invented many beautiful fractal drawings. We can write programs to reproduce them. We start with the Sierpinski Carpet. It is based on the following rewriting rules:

```
     000            XXX
0 -> 000       X -> X0X
     000            XXX
```

where 0 actually stands for a space. Starting from a single X, and applying rewrite rules 1,2 and 3 times, gives us

```
XXX
X X
XXX
```

```
XXXXXXXXX
X XX XX X
XXXXXXXXX
XXX   XXX
X X   X X
XXX   XXX
XXXXXXXXX
X XX XX X
XXXXXXXXX
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXX
X XX XX XX XX XX XX XX XX X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX   XXXXXX   XXXXXX   XXX
X X   X XX X   X XX X   X X
XXX   XXXXXX   XXXXXX   XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
X XX XX XX XX XX XX XX XX X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXX         XXXXXXXXX
X XX XX X         X XX XX X
XXXXXXXXX         XXXXXXXXX
XXX   XXX         XXX   XXX
X X   X X         X X   X X
XXX   XXX         XXX   XXX
XXXXXXXXX         XXXXXXXXX
X XX XX X         X XX XX X
XXXXXXXXX         XXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
X XX XX XX XX XX XX XX XX X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX   XXXXXX   XXXXXX   XXX
X X   X XX X   X XX X   X X
XXX   XXXXXX   XXXXXX   XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
X XX XX XX XX XX XX XX XX X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
```

That is actually the result displayed on the screen by the queries *?- carpet(1).*, *?- carpet(2).* and *?- carpet(3).* respectively.
If we start from a different set of rewrite rules, e.g.,

```
          000                XXX
     0 -> 000           X -> 0X0
          000                XXX
```

we get:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXX
 X  X  X  X  X  X  X  X  X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXX      XXX      XXX
    X        X        X
   XXX      XXX      XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
 X  X  X  X  X  X  X  X  X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
         XXXXXXXXX
          X  X  X
         XXXXXXXXX
            XXX
             X
            XXX
         XXXXXXXXX
          X  X  X
         XXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
 X  X  X  X  X  X  X  X  X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXX      XXX      XXX
    X        X        X
   XXX      XXX      XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
 X  X  X  X  X  X  X  X  X
XXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Your program must accept all possible rewrite rules of the form above: they are specified in Prolog as two facts. For the Sierpinski carpet, those would be:

```
rewrite('X',[['X','X','X'],['X',0,'X'],['X','X','X']]).
rewrite(0,[[0,0,0],[0,0,0],[0,0,0]]).
```

For the second drawing, they would be:

```
rewrite('X',[['X','X','X'],[0,'X',0],['X','X','X']]).
rewrite(0,[[0,0,0],[0,0,0],[0,0,0]]).
```

Write a program that has a predicate carpet/1 to be called as already shown and with the desired effect, depending on the rewrite rules given to you. The argument to carpet/1 shall always be a strictly positive integer. And in case you wonder ... the second argument of rewrite/2 is a list of length N of lists of length N: we have shown only the case N = 3, but of course, your program must work (sensibly !) for any non-negative N.

2 Bitrev *(bitrev.pl)*

Write a predicate bitrev/2 which accepts as first argument a list L of length 2^N for some N, and which unifies the second argument with a permutation P of L such that if x is the ith element in L, then the $bitrev(N,i)^{th}$ element of P is this very x. We start counting positions from 0.
For a given i, $bitrev(N,i)$ is the number you get by reading the N bits of the binary representation of i in reverse, so,

$bitrev(4, 12) = rev(1100_2) = 0011_2 = 3$ and $bitrev(5, 12) = rev(01100_2) = 00110_2 = 6$.

The nature of the elements is unspecified: atoms, integers, terms or variables ... There can be duplicates in the list L.

There is a bonus for a pure Prolog program that works for any instantiation of the query. This bonus consists in the subtraction of 20 minutes from your total submission time. Be careful not to use impure built-in or library predicates ! Ah, small detail: you can't use arithmetic even if you are not going for the bonus ! In particular, your can't use any number in your program.

Examples:

```
?- bitrev([a,b],L).
L = [a,b]

?- bitrev([a,b,c,d],L).
L = [a,c,b,d]

?- bitrev(L,[1,2,3,4,5,6,7,8]).
L = [1,5,3,7,2,6,4,8]

?- bitrev(In,Out).
In = []
Out = [] ;

In = [_G252]
Out = [_G252] ;

In = [_G252, _G255]
Out = [_G252, _G255] ;

In = [_G252, _G255, _G273, _G279]
Out = [_G252, _G273, _G255, _G279]
```

Thanks to Thomas Conway for suggesting this problem for the 1996 contest.

3 More sums than differences *(sumdiff.pl)*

With a (finite) set S of integers, we can form the set of sum(S) of sums of 2 elements of S and the set of differences diff(S) of 2 elements of S. More formally:

$$sum(S) = \{i + j | i, j \in S\} \text{ and } diff(S) = \{i - j | i, j \in S\}$$

Since for $i \neq j$ we have that $(i-j) \neq (j-i)$ but $(i+j) = (j+i)$, we might expect that $|sum(S)| \leq |diff(S)|$, but for many sets $|sum(S)| > |diff(S)|$. In the literature, such a set is called MSTD (More Sums Than Differences)[1].

The property of being MSTD is invariant under affine transformation, i.e., if S is MSTD, than also the sets x*S+y for all integer $x \neq 0$ and any y are MSTD. So it makes sense to consider only MSTDs which contain 0 and positive numbers, like {0,2,3,4,7,11,12,14}. In Prolog, we represent such a set as the list: [0,2,3,4,7,11,12,14].

The length of an MSTD is the size of the set, and the weigth of an MSTD is its highest element. The above MSTD has length 8 and weigth 14.

The following can then be investigated: for given N, compute an MSTD with length N and with smallest possible weigth. Yes, yes, do not be impatient, you are allowed to write a Prolog program that can solve this question. Here an example of a correct query:

```
?- sumdiff(8,L).
L = [0,2,3,4,7,11,12,14]
```

For some values of N, the query sumdiff(N,L) has no answer (e.g., for N = 3): your predicate is allowed to loop in such a case. If there is more than one answer, please give us just one.

4 Juggle sequence *(juggle.pl)*

We focus for now on two-handed ball juggling in which hands throw alternatingly one ball and only on every beat of a regular device (like a clock). Every time a ball is thrown, we can note down the number of beats before it is picked up by a hand and thrown immediately again - note that this might be the same hand it was thrown from before. So, the sequence 3333333 denotes the classical juggling with 3 balls in which each hand alternatingly throws a ball to the other hand: each ball stays in the air for 3 beats. 22222 denotes juggling with two balls, where each hand throws one ball and catches the same ball over and over again. Now a slightly more complicated one: 151515151515. This one can be performed as follows with three balls named A, B and C:

```
beat 1: throw A with left hand to right hand: stays 1 beat in the air
beat 2: throw A with right hand to left hand:  5 beats
beat 3: throw B with left hand to right hand:  1 beat
beat 4: throw B with right hand to left hand:  5 beats
beat 5: throw C with left hand to right hand:  1 beat
beat 6: throw C with right hand to left hand:  5 beats
beat 7: A (thrown at beat 2) arrives in left hand: do as in beat 1
...
```

[1]Surely you think we are pulling your leg, but we are not.

Every pattern of juggling has its own sequence. Such a sequence might repeat and it is costumary to denote only the shortest repeating sequence, so the above is just as well described by 3 and 2 and 15 (or 51).

Not every sequence of integers is a juggling sequence: e.g., 32 (or 32323232) cannot be a juggling sequence because the balls thrown at beats 1 and 2, arrive at the same moment. Similarly, 5939 cannot be a juggling sequence because the balls thrown at beats 1 and 3 arrive at the same moment in the same hand.

You shall write a predicate juggle/2 which given a sequence (in the form of a list, e.g., [3,2]) succeeds if and only if that sequence is a juggling sequence.
Examples:

```
?- juggle(2,[3]). Yes
?- juggle(2,[2]). Yes
?- juggle(2,[5,1]). Yes
?- juggle(2,[2,3]). No
```

You wonder what the first argument is for of course: it is the number of hands of the juggler. Above we were explaining everything under the assumption that the juggler has just two hands, but you must generalise appropriately to any number (> 1) of hands: the general idea is that if there n hands, they throw in the order 1,2,3,...,n,1,2,... of course, but never at the same moment.

5 Optimal sort *(optisort.pl)*

Page 184 in D. Knuth, *Sorting and Searching* shows how 5 (different) items can be sorted with at most 7 comparisons (and that less than 7 is not enough): this constitutes minimal comparison sorting. Clearly, one can devise an algorithm to find the optimal sequence of comparisons for any number of items. You will do almost that: you will write a predicate optisort/3 which is called with as first argument a list of different items (ground terms) that need to be sorted, as second argument a list of comparisons between some items that are known to hold, and a free third argument.

Your predicate must unify the third argument with a *good* decision tree of comparisons, so that together with the known comparisons, the item list can be sorted.
A decision tree has the type:

```
decisiontree --> done | compare(item,item,decisiontree,decisiontree)
```

The third argument of compare/4 is the decision tree in case the comparison between the first two item arguments yields $<$; and opposite for the fourth argument.
Some examples:

```
?- optisort([a,b],[],D).
D = compare(a,b,done,done).
```

```
?- optisort([a,b,c],[],D).
D = compare(a,b,compare(a,c,compare(b,c,done,done),
                            done),
                compare(a,c,done,
                            compare(b,c,done,done)))

?- optisort([a,b,c],[a<b,b<c],D).
D = done

?- optisort([a,b,c],[b<c],D).
D = compare(a,b,done,compare(a,c,done,done))
```

You see that we use $x < a$ to denote that in the order of the items, x comes before a. Of course, you should not use the Prolog $<$ on the items, neither the $@-compare$ family of term comparison predicates: the order is completely fictitious.

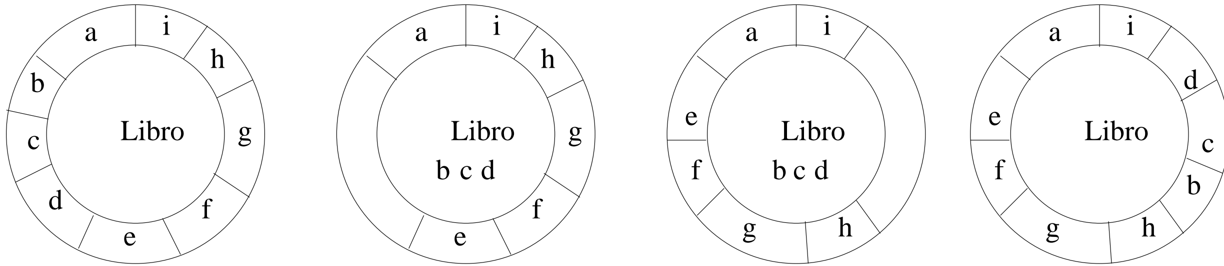
Here is the criterium for accepting your solution: it should be at least as good as our own solution (which we know is not optimal) on all test cases. A solution is better when the decision tree is less high.

You can even earn a time bonus: for a set of 5 test cases the differences are made between the height of your decision tree and ours, and then summed. This will yield a number $D \geq 0$. Your bonus is $10 * D + 10$ minutes.

6 Circular book shelf move *(bookmove.pl)*

Libro, the university librarian, likes to keep the books on his shelves sorted by the id on them (btw, a book id is not necessarily a number). Students tend to put books back in the wrong order, but on the right shelf, so every morning Libro sorts the books just before the opening of the library. It is important to know that the shelves are round - see the picture later - and that Libro has a peculiar way of sorting books (also see later), in particular, he does not care where the book with the lowest id is placed on the shelf and he doesn't care whether the books are sorted in ascending or descending order. We first define what it means for such a circular book shelf to be sorted. There are two distinguished books, Min and Max: no book has an id lower than Min and no book has an id higher than Max. A book shelf is sorted if it is left sorted or right sorted. It is left sorted if the id of a book is always smaller than the id of its left neighbour, except for Max which has as left neighbour Min. You certainly can make up the definition of right sorted now, can't you? Left and right mean different things when you are inside or outside the circle: Libro stands inside: see the picture.

The way Libro sorts a circular book shelf goes as follows: he makes *book moves*. One book move consists in: he takes from the shelf N books that happen to be next to each other on the shelf, shifts M books to the left or the right and inserts the N books in the new hole - in the same order they were in when he took them.



The figure shows 9 books (a up to i) on one circular shelf. Initially, the books are in the order a,b,c,d,e,f,g,h,i. Libro stands in the middle. He takes out books b,c,d and keeps them on a little table with him. He then puts his hand between books i and h, and shifts books e,f,g,h to the right, until book e touches book a. Finally, he puts books b,c,d back in the now open slot.

We represent a full book shelf in Prolog by a list of book ids, starting with an arbitrary one and then its left (for Libro) neighbour etc. So a correct representation of the initial shelf in the picture is [a,b,c,d,e,f,g,h,i], but also [g,h,i,a,b,c,d,e,f] would do.

The end configuration in the picture was obtained by one book move. We are concerned here with the following sanity check: given two book configurations, can the second one be obtained by one book move from the first one ? You must write a predicate bookmove/2 which when called with two lists of book ids, succeeds when they are one book move apart, and otherwise fails. Here are some examples:

```
?- bookmove([a,b,c,d,e,f,g,h,i],[e,b,c,f,g,h,i,a,d]).
yes

?- bookmove([1,2,3,4,5],[4,5,3,1,2]).
yes

?- bookmove([1,2,3,4,5,6],[6,5,4,3,2,1]).
no
```

Libro manages a good library, so there is more than one copy of some books. Libro never saw a good reason to give duplicates a different id.