shapes/2

You get an NxM (N and M differ from 0) matrix filled with the colours black and white. The blacks are background; a connected component of whites form a shape. You are to determine the number of shapes in the matrix. Two white entries of the matrix belong to the same shape if there is a path from one to the other over whites; a path consist of successive points with coordinates (i,j) and (k,l) such that |i-k| =< 1 and |j-l| =< 1.

Your program cannot use any of the following symbols

```
arg   =..   functor   name
```

You can check whether your program passes the criterium (no forbidden symbols) with the check_arith program in your home directory: consult it and then run the query ?- iclp97_check_shapes(<file>). where <file> is the filename of your program. The organisation uses the same program.

The matrix is given as a list of lists which contains elements equal to black or white - this should be obvious from the examples - as the first argument of the predicate shapes/2 that you have to write. The second argument should be unified with the number of shapes in the first argument. Here are some example calls:

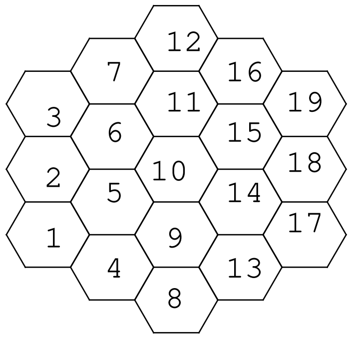
```
?- shapes([[black,white,black],
           [white,black,black],
           [white,black,white]],N).
N = 2

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

N = 1
```

A hint: you need no arithmetic at all :-)

Acknowledgement

The problems above are variations on folklore and/or partly new as far as the organisation knows, but we claim no originality. Sincere thanks to Phuong Lan Nguyen for checking the questions and the feasability of the problems.

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

?- hex(3,1,1,L).
L = [2,4,5]

?- hex(3,1,2,L).
L = [3,6,8,9,10]

?- hex(3,1,3,L).
L = [7,11,13,14,15]

?- hex(3,1,4,L).
L = [12,16,17,18,19]

?- hex(3,1,5,L).
L = []
```



codegen/3

You have a machine with registers r1 up to rn, which are organised in a ring, so that you can have only register moves from ri to r(i+1) for 1 <= i < n and from rn to r1, with move(i) (1 <= i < n) and move(n). And you can also swap the contents of any two registers, with the swap(i,j) instruction where i < j.

You are give the initial contents of all registers (but 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 into the final state, or fails if this is impossible.

The predicate you write is codegen/3: first argument is a list of the initial contents of the registers, second argument the final contents; the sequence of instructions that effectuate the transition is returned in argument three, 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 *: in the initial state, the meaning is "don't know the contents"; in the final state, it means "don't care about the contents". Accordingly:

```
?- codegen([a,*,c],[c,a,*],L).
L = [swap(1,2),swap(1,3)]
or
L = [move(1),move(3)]
or many other correct solutions

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

index/2

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 occurences 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
```

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

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

You will write a predicate index/2 which given a clause as first argument, produces its readability index. Clauses are always given as (Head :- Body) (facts have Body == true). Don't worry about higher order arguments. The depth of an argument is defined as follows:

```
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:

	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

hex/4

You get a large hexagonal composed of smaller hexagonals, numbered as in the picture. You are asked to generate the set of small hexagonals at distance N (for a given N) to a given hexagonal H. You will write the predicate hex/4, which has first argument the 'size' S of the large hexagonal (i.e. the number of hexagonals in its first column), as second argument the number of the hexagonal H, as third argument the distance N. Hex/4 should unify the fourth argument with a list of all numbers of hexagonals which are at distance N from H. This list must be sorted.

You can earn two points for hex: one point if it works correctly for small S (up to 20); one extra point if it works also for large S - in the order of 10000 - and small N up to 10. You can hand in a second solution to earn this point (it must also pass all given test cases !)

Example calls and answers:

There are 5 problems: their name is also the name/arity of the predicate you should write.

For every problem, there are test cases in your home directory, in files that have the names of the problems with prefix test_ (how to sue them should be evident).You should hand in solutions only after testing the test cases you are given: if you hand in a solution for problem X which does not pass all given test cases, then you will not be permitted to hand in another solution to X later !

Your solution should not contain predicates with a name that starts with iclp97 or test, because these might clash with predicates in the test suite.

Efficiency of your programs is not important (except for the extra point you can earn in problem hex/4), but if your program fails to finish in a reasonable time, it will be considered incorrect. Test cases will contain an indication of the run+gctime the organisation deems reasonable.

snake/3

The Belgian snake is an animal whose body has a repeated pattern but 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. Also, the Belgian snake is an animal that likes to lie around 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 picture. We want you to write a predicate snake/3 which displays such a coiled up Belgian snake. That predicate will be called with the following three arguments:

```
a list of atoms representing a pattern
a list whose length = the number of rings in one row (not [])
a list whose length = the number of rings in one column (not [])
```

Your snake/3 should produce output on the screen; as an example:

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

(the stretched out snake would look like: a b c d a b c d a b c d a b c)

should produce:

```
a b c d a
b a d c b
c d a b c
```

Before you start implementing this, you should know that Belgian snakes dislike arithmetic computations very much: your solution program is not allowed to use any of the arithmetic operators; in particular, your program will be rejected if it contains any of the symbols

```
is   <   >   >=   =<   +   -   *   name   arg   functor   =..
```

You can check whether your program passes the criterium (no forbidden symbols) with the check_arith program in your home directory: consult it and then run the query ?- iclp97_check_snake(<file>). where <file> is the filename of your program. The organisation uses the same program.