```
/* In baza de cunostinte de la primul laborator, pentru ciurul lui Eratostene, am filtrat la fiecare
pas lista eliminand din coada ei multiplii primului sau element, adica stergand din coada listei
elementele care se divid cu capul listei.
Sa efectuem la fiecare pas filtrarea listei [H|T] si prin stergerea din coada T a elementelor din H
in H, adica stergerea elementelor de pe pozitiile H, 2*H, 3*H s.a.m.d. din coada T.
Marcam cut(!)-urile optionale, care au ca rol doar intreruperea executiei imediat dupa gasirea
unicei solutii, astfel ca Prolog-ul sa nu mai lase utilizatorul sa ceara alte solutii (cu ;/Next),
apoi sa intoarca false, indicand faptul ca nu mai exista alte solutii: */

ciur(N,LP) :- lista(2,N,L), ciuruire(L,LP).

lista(K,N,[]) :- K>N, !.
lista(K,N,[K|T]) :- SK is K+1, lista(SK,N,T).

ciuruire([],[]).   %%% Optional: ciuruire([],[]) :- !.
ciuruire([H|T],[H|LP]) :- filtreaza(H,T,L), ciuruire(L,LP).

filtreaza(X,L,LfaraMX) :- auxfilt(X,X,L,LfaraMX).

auxfilt(_,_,[],[]).   %%% Optional: auxfilt(_,_,[],[]) :- !.
auxfilt(X,1,[_|T],L) :- auxfilt(X,X,T,L), !.
auxfilt(X,Contor,[H|T],[H|L]) :- PC is Contor-1, auxfilt(X,PC,T,L).

/* Interogati:
?- ciur(15,L).
?- ciur(1500,L), write(L).
La ultima interogare, lista L rezultata este lunga, asadar Prolog-ul o afiseaza truchiat ca valoare
de variabila, asadar o afisam explicit, cu predicatul predefinit write. */

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
/* Varianta pentru a doua clauza din definitia predicatului ciuruire si definitia predicatului
filtreaza, avand in vedere ca predicatul filtreaza de mai sus doar apeleaza un predicat auxiliar cu
un singur argument in plus, anume contorul care se reseteaza dupa fiecare stergere a unui element:

ciuruire([H|T],[H|L]) :- filtreaza(H,H,T,M), ciuruire(M,L).

filtreaza(_,_,[],[]).
filtreaza(X,1,[_|T],L) :- filtreaza(X,X,T,L).
filtreaza(X,Contor,[H|T],[H|L]) :- Contor>1, PC is Contor-1, filtreaza(X,PC,T,L).
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Si varianta pe care am scris-o initial la acest laborator, mai dezavantajoasa:

ciuruire([H|T],[H|LP]) :- filtreaza(H,T,L), ciuruire(L,LP).

filtreaza(_,[],[]) :- !.
filtreaza(X,L,M) :- elimalXlea(X,L,[],P,S), filtreaza(X,S,Sfiltrata), append(P,Sfiltrata,M).

elimalXlea(_,[],P,P,[]).
elimalXlea(1,[_|T],P,P,T).
elimalXlea(X,[H|T],Pred,P,S) :- X>1, append(Pred,[H],PredH),
                                PX is X-1, elimalXlea(PX,T,PredH,P,S).

Pentru a vedea inclusiv diferenta intre timpii de executie ai acestor doua variante, cea cu
filtrarea listei [H|T] prin stergerea tuturor elementelor din coada T din H in H direct cu
predicatul de filtrare versus stergerea succesiva a celui de-al H-lea, apoi apelarea recursiva a
predicatului de filtrare, interogati, pentru fiecare dintre aceste variante:
?- Init is cputime, ciur(10000,L), Fin is cputime, write(L), nl, Dif is Fin-Init, write(Dif),
tab(1), write(secunde).
```

```
Veti constata ca aceasta varianta cu apel recursiv este de cam 100 de ori mai lenta decat cea in
care predicatul de filtrare face toata stergerea dintr-o data, resetand doar valoarea contorului
dupa fiecare pas de stergere.*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Sa folosim genul de concatenare de mai sus pentru a pune un element la sfarsitul unei liste
pentru a inversa o lista; un predicat predefinit care face acelasi lucru este reverse: */

inversa([],[]).
inversa([H|T],L) :- inversa(T,M), append(M,[H],L).

/* Interogati:
?- inversa([1,2,3,4,5],CeLista).
?- inversa(CareiListe,[1,2,3,4,5]).
La ultima dintre aceste doua interogari, nu cereti inca o solutie cu ;/Next, pentru ca in acest caz
predicatul produce ciclare infinita. */

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Predicatul zeroar predefinit fail intotdeauna esueaza, adica intoarce false.
Spre deosebire de predicatul predefinit member, urmatorul predicat detecteaza doar prima aparitie a
unui element intr-o lista: */

apartine(_,[]) :- fail.
apartine(H,[H|_]) :- !.
apartine(X,[_|T]) :- apartine(X,T).

% Acum sa scriem un predicat echivalent cu predicatul predefinit member:

membru(_,[]) :- fail.
```

```
membru(H,[H|_]).      %%% Regula echivalenta cu acest fapt: membru(X,[H|_]) :- X=H.
membru(X,[_|T]) :- membru(X,T).

/* Interogati:
?- apartine(4,[1,2,3,4,5]).
?- apartine(10,[1,2,3,4,5]).
?- apartine(Cine,[1,2,3,4,5]).
?- membru(4,[1,2,3,4,5]).
?- membru(10,[1,2,3,4,5]).
?- membru(Cine,[1,2,3,4,5]).
si dati ;/Next pentru obtinerea tuturor solutiilor la aceasta ultima interogare.
   Dati interogarile de mai sus si cu predicatul predefinit member in locul lui apartine sau membru.
   Pentru oprirea executiei la intalnirea capatului listei, la fel ca in cazul lui member, in locul
cautarii unei alte solutii si intoarcerea lui false, indicand ca nu mai exista alte solutii, am
putea defini predicatul astfel: */

testmembru(X,[X]) :- !.
testmembru(X,[H|T]) :- T\=[], (X=H; testmembru(X,T)).

/* Acum sa determinam aparitiile literal identice ale unui element intr-o lista, inlocuind, in
predicatul member, unificarea cu literal identitatea: */

membrulitid(_,[]) :- fail.
membrulitid(X,[H|_]) :- X==H.
membrulitid(X,[_|T]) :- membrulitid(X,T).

/* Predicatul predefinit == pentru testarea literal identitatii face diferenta chiar si intre
variabilele cu denumiri diferite. Interogati:
?- membrulitid(4,[1,2,3,4,5]).
?- membrulitid(10,[1,2,3,4,5]).
?- membrulitid(Cine,[1,2,3,4,5]).
```

```
?- membrulitid(Cine,[1,X,2,V,3,4,5]).
?- membrulitid(Cine,[Cine,1,X,Cine,2,V,Cine,3,4,5,Cine]).
si, la ultima interogare, dati ;/Next pentru a obtine toate cele 4 solutii.
   Amintesc ca:
negatia predicatului = de testare a unificarii este \=
negatia predicatului == de testare a literal identitatii este \==
negatia predicatului =:= care produce efectuarea calculului aritmetic in ambii sai membri, apoi
unificarea constantelor numerice astfel obtinute, este =\=
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Metapredicate care colecteaza solutiile altor predicate:
   setof(Termen, Conditie, Lista)=true <=> Lista este lista termenilor de forma Termen care satisfac
conditia Conditie, ca lista fara duplicate, si intorcand false cand nu exista termeni de forma
Termen care sa satisfaca scopul Conditie;
   bagof(Termen, Conditie, Lista) <=> Lista este lista termenilor de forma Termen care satisfac
conditia Conditie, ca lista cu duplicate, si intorcand false cand nu exista termeni de forma Termen
care sa satisfaca scopul Conditie;
   findall(Termen, Conditie, Lista) <=> Lista este lista termenilor de forma Termen care satisfac
conditia Conditie, ca lista cu duplicate, si intorcand Lista=[] cand nu exista termeni de forma
Termen care sa satisfaca scopul Conditie.
   findall mai difera fata de bagof si in tratarea conditiilor Conditie in care apar variabile care
nu apar in termenul Termen.
Interogati:
?- setof(X, member(X,[a,1,b,0,a,b,c,a,1,1]), LfaraDuplicate).
?- setof((X,Y,Z), (member(X,[false,true]), member(Y,[false,true]), member(Z,[false,true])),
ListaTripleteValoriBooleene), write(ListaTripleteValoriBooleene).
?- setof(X, member((X,Y),[(a,1),(b,1),(a,2),(c,3),(c,3),(d,3)]), L).
?- bagof(X, member((X,Y),[(a,1),(b,1),(a,2),(c,3),(c,3),(d,3)]), L).
?- findall(X, member((X,Y),[(a,1),(b,1),(a,2),(c,3),(c,3),(d,3)]), L).
```

```
?- bagof(X, Y^member((X,Y),[(a,1),(b,1),(a,2),(c,3),(c,3),(d,3)]), L).
?- setof(X, Y^member((X,Y),[(a,1),(b,1),(a,2),(c,3),(c,3),(d,3)]), L).
Sintaxa de mai sus, care nu este acceptata de findall, inseamna cuantificare existentiala pentru
variabila Y. Penultimele doua interogari de mai sus intorc acelasi lucru.
Observati ca, fara aceasta cuantificare existentiala pentru variabila Y care apare in conditia din
al doilea argument, dar nu apare in termenul din primul argument, setof si bagof intorc cate o
valoare a listei L pentru fiecare valoare distincta a lui Y, pe cand findall grupeaza in aceeasi
lista L toate valorile lui X pentru toate aceste valori ale lui Y. */

% Afisarea unei liste cu fiecare element pe alt rand:

afislista([]).
afislista([H|T]) :- write(H), nl, afislista(T).

/* Interogati:
?- setof((X,Y,Z), (member(X,[false,true]), member(Y,[false,true]), member(Z,[false,true])),
ListaTripleteValoriBooleene), afislista(ListaTripleteValoriBooleene).
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Am vazut ca setof poate fi folosit pentru eliminarea duplicatelor dintr-o lista. Sa eliminam
duplicatele si prin scrierea unui predicat recursiv, care pastreaza ultima aparitie a fiecarui
element in lista: */

elimdup([],[]).
elimdup([H|T],L) :- member(H,T), !, elimdup(T,L).
elimdup([H|T],[H|L]) :- elimdup(T,L).

% Stergerea tuturor aparitiilor unui element dintr-o lista:
```

```
sterge(_,[],[]).
sterge(H,[H|T],L) :- !, sterge(H,T,L).
sterge(X,[H|T],[H|L]) :- sterge(X,T,L).

% Eliminarea duplicatelor cu pastrarea primei aparitii a fiecarui element in lista:

elimdupl([],[]).
elimdupl([H|T],[H|L]) :- sterge(H,T,M), elimdupl(M,L).

/* Asadar, la interogarea:
?- elimdup([a,2,1,0,b,a,2,1,1],L).
vom primi raspunsul: L=[0,b,a,2,1].
Iar la interogarea:
?- elimdupl([a,2,1,0,b,a,2,1,1],L).
vom primi raspunsul: L=[a,2,1,0,b].
*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Produsul cartezian (de multimi, i.e. intors fara duplicate chiar daca listele au duplicate): */

prodmult(L,M,LxM) :- setof((X,Y), (member(X,L), member(Y,M)), LxM), !.
prodmult(_,_,[]).

% Produsul cartezian de liste (intors cu duplicate):

prodlist(L,M,LxM) :- bagof((X,Y), (member(X,L), member(Y,M)), LxM), !.
prodlist(_,_,[]).

% Tot produsul cartezian de liste (intors cu tot cu duplicatele din acestea):
```

```
prodliste(L,M,LxM) :- findall((X,Y), (member(X,L), member(Y,M)), LxM).

% Putem defini produsul cartezian de liste si recursiv, fara a apela la metapredicate:

prodcart([],_,[]).
prodcart([H|T],L,P) :- prodsgl(H,L,Q), prodcart(T,L,R), append(Q,R,P).

prodsgl(_,[],[]).
prodsgl(H,[K|T],[(H,K)|U]) :- prodsgl(H,T,U).

% Pentru a obtine produsul cartezian fara duplicate, putem scrie:

prodcartmult(L,M,LxM) :- prodcart(L,M,P), elimdupl(P,LxM).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Sa demonstram semantic, adica prin calcul cu valori de adevar, distributivitatea reuniunii fata
de intersectie, adica proprietatea ca, pentru orice multimi A,B,C, are loc egalitatea:
   A U (B ^ C) = (A U B) ^ (A U C),
unde am notat cu ^ operatia de intersectie intre multimi.
   Sa fixam trei multimi arbitrare A,B,C.
   Fie x (se subintelege ca arbitrar, fixat).
   Notam cu variabilele A,B,C (le puteti nota _a,_b,_c in cazul in care considerati ca e pericol de
confuzie intre aceste variabile booleene si multimile A,B,C) urmatoarele proprietati:
A:  x apartine lui A
B:  x apartine lui B
C:  x apartine lui C
   Avem de demonstrat egalitatea AU(B^C)=(AUB)^(AUC), adica faptul ca multimile AU(B^C) si
(AUB)^(AUC) au aceleasi elemente, adica faptul ca, pentru x-ul arbitrar fixat mai sus, are loc: x
apartine lui AU(B^C) <=> x apartine lui (AUB)^(AUC), adica:
   [x apartine lui A sau (x apartine lui B si x apartine lui C)] <=>
```

```
   [(x apartine lui A sau x apartine lui B) si (x apartine lui A sau x apartine lui C)].
   Asadar distributivitatea reuniunii fata de intersectie se transcrie in distributivitatea
disjunctiei fata de conjunctia logica.
   Demonstram ca orice triplet de valori booleene satisface aceasta echivalenta aratand ca nu exista
triplet de valori booleene care sa nu satisfaca aceasta echivalenta, folosind predicatul member si
faptul ca, pentru a satisface not(Scop), Prolog-ul incearca mai intai sa satisfaca Scop, iar, daca
nu gaseste satisfaceri ale acestuia, intoarce true pentru not(Scop). Pentru a verifica faptul ca
Prolog-ul trece prin toate tripletele de valori booleene incercand sa satisfaca argumentul not-ului
cel mai exterior de mai jos, vom si afisa, la fiecare pas, tripletul curent.
   Atentie la perechile suplimentare de paranteze necesare pentru ca Prolog-ul sa nu confunde
conjunctiile cu separatori de argumente! */

% Sa definim conectorii logici de implicatie si echivalenta:

implica(P,Q) :- not(P);Q.
echiv(P,Q) :- implica(P,Q), implica(Q,P).

% Acum sa demonstram distributivitatea de mai sus:

membrulstg(A,B,C) :- A ; B,C.
membruldr(A,B,C) :- (A;B) , (A;C).

distrib(A,B,C) :- echiv(membrulstg(A,B,C),membruldr(A,B,C)).

demdistrib :- not((member(A,[false,true]), member(B,[false,true]), member(C,[false,true]),
        write((A,B,C)), nl, not(distrib(A,B,C)))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* Exercitiul 4 din Seminarul I, partea 1:
(1)     (a si b) => (c xor d)
```

```
(2)     (b si c) => [(a si d) sau (non a si non d)] <=> non(a xor d)
(3)     (non a si non b) => (non c si non d)
Sa demonstram:
(I)     (non a si non b) => non c
(II)    non(a si b si c)
in ipotezele de mai sus, adica faptul ca ipotezele (1),(2) si (3) implica (I) si (II). */

/* Sa definim conectorul logic sau exclusiv; vedeti, in baza de cunostinte de la Cursul IV, si
definitia sa ca operator infixat: */

xor(P,Q) :- P, not(Q) ; Q, not(P).

% Acum sa rezolvam exercitiul de mai sus:

ipoteza1(A,B,C,D) :- implica((A,B), xor(C,D)).
ipoteza2(A,B,C,D) :- implica((B,C), (A,D ; not(A),not(D))).
ipoteza3(A,B,C,D) :- implica((not(A),not(B)), (not(C),not(D))).

ipoteza(A,B,C,D) :- ipoteza1(A,B,C,D), ipoteza2(A,B,C,D), ipoteza3(A,B,C,D).

concluziaI(A,B,C) :- implica((not(A),not(B)), not(C)).
concluziaII(A,B,C) :- not((A,B,C)).

cerintaI(A,B,C,D) :- implica(ipoteza(A,B,C,D), concluziaI(A,B,C)).
cerintaII(A,B,C,D) :- implica(ipoteza(A,B,C,D), concluziaII(A,B,C)).

demcerintaI :- not((member(A,[false,true]), member(B,[false,true]),
                member(C,[false,true]), member(D,[false,true]), write((A,B,C,D)), nl,
                not(cerintaI(A,B,C,D)))).

demcerintaII :- not((member(A,[false,true]), member(B,[false,true]),
```

```
                member(C,[false,true]), member(D,[false,true]), write((A,B,C,D)), nl,
                not(cerintaII(A,B,C,D)))).

% Varianta:

concluzia(A,B,C) :- concluziaI(A,B,C), concluziaII(A,B,C).

cerinta(A,B,C,D) :- implica(ipoteza(A,B,C,D), concluzia(A,B,C)).

demcerinta :- not((member(A,[false,true]), member(B,[false,true]),
                member(C,[false,true]), member(D,[false,true]), write((A,B,C,D)), nl,
                not(cerinta(A,B,C,D)))).
```