

## CURS 5

### Nedeterminism. Backtracking

#### Cuprins

1. Exemple predicate nedeterministe (continuare).....	1
2. Backtracking .....	5

% depanare cod Prolog  
% trace.  
% notrace.

#### 1. Exemple predicate nedeterministe (continuare)

**EXEMPLU 3.1** Să se scrie un predicat nedeterminist care generează combinații cu  $k$  elemente dintr-o mulțime nevidă reprezentată sub forma unei liste.

```
? comb([1, 2, 3], 2, C). /*model de flux (i, i, o) - nedeterminist*/  
C = [2, 3];  
C = [1, 2];  
C = [1, 3].
```

**Observație:** Pentru determinarea combinațiilor unei liste  $[E|L]$  (care are capul  $E$  și coada  $L$ ) luate câte  $K$ , sunt următoarele cazuri posibile:

- dacă  $K=1$ , atunci o combinație este chiar  $[E]$
- determină o combinație cu  $K$  elemente a listei  $L$ ;
- plasează elementul  $E$  pe prima poziție în combinațiile cu  $K-1$  elemente ale listei  $L$  (dacă  $K>1$ ).

Modelul recursiv pentru generare este:

$comb(l_1 l_2 \dots l_n, k) =$

- $(l_1)$   $daca\ k = 1$
- $comb(l_2 \dots l_n, k)$
- $l_1 \oplus comb(l_2 \dots l_n, k - 1)$   $daca\ k > 1$

Vom folosi predicatul nedeterminist **comb** care va genera toate combinările. Dacă se dorește colectarea combinărilor într-o listă, se va putea folosi predicatul **findall**.

Programul SWI-Prolog este următorul:

```
% comb(L: list, K:integer, C:list)
% (i, i, o) - nedeterminist
comb([H|_], 1, [H]).
comb([_|T], K, C) :-
    comb(T, K, C).
comb([H|T], K, [H|C]) :-
    K > 1,
    K1 is K-1,
    comb(T, K1, C).
```

**EXEMPLU 3.2** Să se scrie un predicat nedeterminist care inserează un element, pe toate pozițiile, într-o listă.

```
? insereaza(1, [2, 3], L).    /*model de flux (i, i, o) - nedeterminist*/
L = [1, 2, 3];
L = [2, 1, 3];
L = [2, 3, 1].
```

### Model recursiv

$insereaza(e, l_1 l_2 \dots l_n) =$

1.  $e \oplus l_1 l_2 \dots l_n$
2.  $l_1 \oplus insereaza(e, l_2 \dots l_n)$

```
% insereaza(E: element, L:List, LRez:list)
% (i, i, o) - nedeterminist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
    insereaza(E, T, Rez).
```

Observăm că, pe lângă modelul de flux (i, i, o) descris anterior, predicatul **insereaza** funcționează cu mai multe modele de flux (în unele modele de flux preducatorul fiind determinist, în altele nedeterminist).

- $insereaza(E, L, [1, 2, 3])$ , cu modelul de flux (o, o, i) și soluțiile  
 $E=1, L = [2, 3]$   
 $E=2, L = [1, 3]$

- E=3, L = [1, 2]
- insereaza(1, L, [1, 2, 3]), cu modelul de flux (i, o, i) și soluția  
L = [2, 3]
  - insereaza(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) și soluția  
E = 2

**EXEMPLU 3.3** Să se scrie un predicat nedeterminist care șterge un element, pe rând, de pe toate pozițiile pe care acesta apare într-o listă.

```
? elimin(1, L, [1, 2, 1, 3]).  /*model de flux (i, o, i) - nedeterminist*/
L = [2, 1, 3];
L = [1, 2, 3];
```

$elimin(e, l_1 l_2 \dots l_n) =$

1.  $l_2 \dots l_n$       *daca*  $e = l_1$
2.  $l_1 \oplus elimin(e, l_2 \dots l_n)$

% elimin(E: element, LRez:list, L:list)

% (i, o, i) – nedeterminist

elimin(E, L, [E|L]).

elimin(E, [A|L], [A|X]) :-

elimin(E, L, X).

Observăm că predicatul **elimin** funcționează cu mai multe modele de flux. Astfel, următoarele întrebări sunt valide:

- elimin(E, L, [1, 2, 3]), cu modelul de flux (o, o, i) și soluțiile  
E=1, L = [2, 3]  
E=2, L = [1, 3]  
E=3, L = [1, 2]
- elimin(1, [2, 3], L), cu modelul de flux (i, i, o) și soluțiile  
L = [1, 2, 3]  
L = [2, 1, 3]  
L = [2, 3, 1]
- elimin(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) și soluția  
E = 2

**EXEMPLU 3.4** Să se scrie un predicat nedeterminist care generează permutările unei liste.

```
? perm([1, 2, 3], P).  /*model de flux (i, o,) - nedeterminist*/
P = [1, 2, 3];
P = [1, 3, 2];
P = [2, 1, 3];
P = [2, 3, 1];
P = [3, 1, 2];
P = [3, 2, 1]
```

Cum obținem permutările listei [1, 2, 3] dacă știm să generăm permutările sublistei [2, 3] (adică [2, 3] și [3, 2])?

Pentru determinarea permutărilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

1. determină o permutare L1 a listei L;
2. plasează elementul E pe toate pozițiile listei L1 și produce în acest fel lista X care va fi o permutare a listei inițiale [E|L].

Modelul recursiv este:

$perm(l_1 l_2 \dots l_n) =$

1.  $\emptyset$  *daca l e vida*
2. *insereaza*( $l_1, perm(l_2 \dots l_n)$ ) *altfel*

% perm(L:list, LRez:list)

% (i, o) – nedeterminist

perm([], []).

perm([E|T], P) :-

perm(T, L),

insereaza(E, L, P). % (i, i, o)

% alternativa pentru clauza 2

perm(L, [H|T]) :-

elimin(H, Z, L), % (o o, i)

perm(Z, T).

% alternativa pentru clauza 2

perm(L, [H|T]) :-

insereaza(H, Z, L), % (o o, i)

perm(Z, T).

% alternativa pentru clauza 2

perm([E|T], P) :-

perm(T, L),

elimin(E, L, P), % (i, i, o)

**TEMĂ** Să se scrie un predicat nedeterminist care generează aranjamente cu  $k$  elemente dintr-o mulțime nevidă reprezentată sub forma unei liste.

? aranj([1, 2, 3], 2, A). /\*model de flux (i, i, o) - nedeterminist\*/

A = [2, 3];

A = [3, 2];

A = [1, 2];

A = [2, 1] ;

$A = [1, 3];$   
 $A = [3, 1];$

## 2. Backtracking

Metoda **backtracking** (căutare cu revenire)

- aplicabilă, în general, unor probleme ce au mai multe soluții.
- permite generarea tuturor soluțiilor unei probleme.
- căutare în adâncime limitată (*depth limited search*) în spațiul soluțiilor problemei
- exponențială ca și timp de execuție
- metodă generală de rezolvare a problemelor din clasa *Constraint Satisfaction Problems*, (CSP)
- Prolog-ul este potrivit pentru rezolvarea problemelor din clasa CSP
  - **structura de control** folosită de interpretorul Prolog se bazează pe backtracking

### Formalizare

- soluția problemei este un vector/listă  $(x_1 x_2 \dots x_n)$  ,  $x_i \in D_i$
- vectorul soluție se generează incremental
- notăm cu *col* lista care colectează o soluție a problemei
- notăm cu **condiții-finale** funcția care verifică dacă lista *col* e o soluție a problemei
- notăm cu **condiții-continuare** funcția care verifică dacă lista *col* poate conduce la o soluție a problemei

Pentru determinarea unei soluții a problemei, sunt următoarele cazuri posibile:

- i. dacă *col* verifică **condiții-finale**, atunci e o soluție a problemei;
- ii. (altfel) se completează *col* cu un element *e* (pe care îl vom numi **candidat**) astfel încât  $col \cup e$  să verifice **condiții-continuare**

### **Observație**

- dacă generarea elementului *e* cu care se va complete colectoarea *col* la punctul ii. este nedeterministă, atunci se vor putea genera toate soluțiile problemei.
- de menționat faptul că sunt probleme în care nu se impun **condiții-finale**

Modelul recursiv general pentru generarea soluțiilor este:

*solutie(col) =*

1. *col* **daca conditii – finale(col)**

2. *solutie(col  $\cup$  e)* **daca conditii – continuare(col  $\cup$  e), e fiind un posibil candidat**

**EXEMPLU 1.1** Să se scrie un predicat nedeterminist care generează combinații cu  $k \neq 1$  elemente dintr-o mulțime nevidă ale cărei elemente sunt numere naturale nenule pozitive, astfel încât suma elementelor din combinație să fie o valoare  $S$  dată.

```
? combSuma([3, 2, 7, 5, 1, 6], 3, 9, C).    /* model de flux (i, i, i, o) – nedeterminist */
C = [2, 1, 6];                             /* k=3, S=9 */
C = [3, 5, 1]
```

!!! **false** la final?

```
? toateCombSuma([3, 2, 7, 5, 1, 6], 2, 9, LC).
LC=[[2, 1, 6], [3, 5, 1]].
```

Pentru rezolvarea acestei probleme, vom da 3 variante de rezolvare, ultima dintre ele fiind bazată pe metoda backtracking descrisă anterior.

**VARIANTA 1** Generăm soluțiile problemei direct recursiv

Pentru determinarea combinațiilor unei liste  $[H|L]$  (care are capul  $H$  și coada  $L$ ) luate câte  $K$ , de sumă dată  $S$  sunt următoarele cazuri posibile:

- i. dacă  $K=1$  și  $H$  este egal cu  $S$ , atunci o combinație este chiar  $[H]$
- i. determină o combinație cu  $K$  elemente a listei  $L$ , având suma  $H$ ;
- ii. plasează elementul  $H$  pe prima poziție în combinațiile cu  $K-1$  elemente ale listei  $L$ , de sumă  $S-H$  (dacă  $K>1$  și  $S-H>0$ ).

Modelul recursiv pentru generare este:

```
combSuma( $l_1 l_2 \dots l_n, k, S$ ) =
1. ( $l_1$ )                                daca  $k = 1$  și  $l_1 = S$ 
2.  $comb(l_2 \dots l_n, k, S)$ 
3.  $l_1 \oplus comb(l_2 \dots l_n, k-1, S-l_1)$   daca  $k > 1$  și  $S-l_1 > 0$ 
```

Vom folosi predicatul nedeterminist **combSuma** care va genera toate combinațiile. Dacă se dorește colectarea combinațiilor într-o listă, se va putea folosi predicatul **findall**.

Programul SWI-Prolog este următorul:

```
% combSuma(L: list, K:integer, S: integer, C:list)
% (i, i, i, o) - nedeterminist
combSuma([H|_], 1, H, [H]).
combSuma([_|T], K, S, C) :-
    combSuma(T, K, S, C).
combSuma([H|T], K, S, [H|C]) :-
    K>1,
    S1 is S-H,
    S1>0,
```

K1 is K-1,  
combSuma(T, K1, S1, C).

```
% toateCombSuma (L: list, K:integer, S: integer, LC:list of lists)
```

## % (i, i, i, o) - determinist

toateCombSuma(L, K, S, LC) :-

$$\text{findall}(C, \text{combSuma}(L, K, S, C), LC).$$

**VARIANTA 2** Folosim un predicat neterminist

**candidat**(E:element, L:list) - model de flux **(o,i)**

care generează, pe rând, câte un element al unei liste. O soluție a acestui predicat va fi un element care poate fi adăugat în soluție.

? candidat(E, [1, 2,3]).

E=1;

$$E=2;$$

E=3

$$candidat(l_1l_2, \dots, l_n) =$$

1.  $l_1$  *daca  $l$  e nevida*

2. *candidat*( $l_2 \dots l_n$ )

```
% candidat(E: element, L:list)
```

% (o, i) - nedeterminist

candidat(E,[E|\_]).

candidat(E,[\_|T]) :-

candidat(E,T).

Predicatul de bază va genera un candidat E și va începe generarea soluției cu acest element.

```
% combSuma(L:list, K:integer, S:integer, C: list)
```

% (i, i, i, o) - nedeterminist

combSuma(L, K, S, C) :-

$$\text{candidat}(\mathbf{E}, \mathbf{L}),$$
$$E < S,$$

combaux(L, K, S, C, 1, E, [E]).

Predicatul auxiliar nedeterminist **comb\_aux** va genera câte o combinație de sumă dată.

$$\text{combaux}(l, k, s, lg, \text{sum}, \text{col}) =$$

1. *col* *daca* (*sum* = *s* și *k* = *lg*)

2.

$combaux(l, k, s, lg + 1, sum + e, e \cup col)$     *daca*  $(sum \neq s \text{ sau } k \neq lg)$  și  $(lg < k)$  și

$(e = candidat(l_1, l_2, \dots, l_n))$  și  $(e < col_1)$  și  $(sum + e \leq s)$

%  $combaux(L:list, K:integer, S:integer, C: list, Lg:integer, Sum:integer, Col:list)$

% (i, i, i, o, i, i, i) - *nedeterminist*

$combaux(\_, K, S, C, K, S, C) :- !.$

$combaux(L, K, S, C, Lg, Sum, [H|T]) :-$

$Lg < K,$

$candidat(E, L),$

$E < H,$

$Sum1 \text{ is } Sum + E,$

$Sum1 \leq S,$

$Lg1 \text{ is } Lg + 1,$

$combaux(L, K, S, C, Lg1, Sum1, [E|H|T]).$

% *alternativa la clauza II – refactorizare - un predicat auxiliar pentru verificarea conditiei*

%  $conditie(L:list, K:integer, S:integer, Lg:integer, Sum:integer, H:integer, E:integer)$

% (i, i, i, i, i, i, o) - *nedeterminist*

$conditie(L, K, S, Lg, Sum, H, E) :-$

$Lg < K,$

$candidat(E, L),$

$E < H,$

$Sum1 \text{ is } Sum + E,$

$Sum1 \leq S.$

$combaux(L, K, S, C, Lg, Sum, [H|T]) :-$

$conditie(L, K, S, Lg, Sum, H, E),$

$Lg1 \text{ is } Lg + 1,$

$Sum1 \text{ is } Sum + E,$

$combaux(L, K, S, C, Lg1, Sum1, [E|H|T]).$

**VARIANTA 3** Se generează combinările cu  $k$  elemente și apoi se verifică dacă suma unei combinări este  $S$ . Această soluție este varianta **brute force** (GTS – *generate and test*), având în vedere că se generează (în plus) combinări care e posibil să nu fie de sumă  $S$ .

%  $combSuma(L: list, K:integer, S: integer, C:list)$

% (i, i, i, o) - *nedeterminist*

$combSuma(L, K, S, C) :-$

$comb(L, K, C),$

$suma(C, S).$

- predicatul **comb** pentru generarea unei combinări cu  $k$  elemente dintr-o listă.
- predicatul **suma** ( $L$ :list of numbers,  $S$ :integer), model de flux (i, i) care verifică dacă suma elementelor listei  $L$  este egală cu  $S$ .

**NOTĂ** Această variantă nu e acceptată la examen.



**EXEMPLU 1.2** Se dă o listă cu elemente numere întregi distincte. Să se genereze toate submulțimile având cel puțin 2 elemente, cu elemente în ordine strict crescătoare.

```
? submCresc([2, 1, 3], S).    /* model de flux (i, o) – nedeterminist */
S = [1, 3];
S = [2, 3];
S = [1, 2];
S = [1, 2, 3];
```

**EXEMPLU 1.3** Dându-se o mulțime reprezentată sub formă de listă, se cere să se genereze submulțimi de sumă pară formate doar din numere impare.

```
? submSP([1, 2, 3, 4, 5], S).    /* model de flux (i, o) – nedeterminist */
S = [1, 3];
S = [1, 5];
S = [3, 5];
```

**EXEMPLU 1.4** Dându-se două valori naturale  $n$  ( $n > 2$ ) și  $v$  ( $v$  nenul), se cere un predicat Prolog care determină permutările elementelor  $1, 2, \dots, n$  cu proprietatea că orice două elemente consecutive au diferența în valoare absolută mai mare sau egală cu  $v$ .

```
? permutari(4, 2, P)           (n=4, v=2)
P = [2, 4, 1, 3];
P = [3, 1, 4, 2];
....
```

```
? candidat(4, I).
I=4;
I=3;
I=2;
I=1
```

### Modele recursive:

*candidat*( $n$ ) =

1.  $n$
2. *candidat*( $n - 1$ ) *daca*  $n > 1$

Se vor folosi următoarele predicate

- predicatul nedeterminist **candidat**( $N, I$ ) (**i, o**) generează un candidat la soluție (o valoare între 1 și  $N$ );
- predicatul nedeterminist **permutari\_aux**( $N, V, LRezultat, Lungime, LColectoare$ ) (**i, i, o, i, i**) colectează elementele unei permutări în **LColectoare** de lungime **Lungime**. Colectarea se

va opri atunci când numărul de elemente colectate (**Lungime**) este N. În acest caz, **LColectoare** va conține o permutare soluție, iar **LRezultat** va fi legată de **LColectoare**.

- predicatul nedeterminist **permutari**(N, V, L) (**i, i, o**) generează o permutare soluție;
- predicatul **apare**(E, L) care testează apartenența unui element la o listă (pentru a colecta în soluție doar elementele distincte).

Folosim un predicat neterminist **candidat**(N:intreg, I:intreg), model de flux (i,o), care generează, pe rând, elementele N, N-1,...,1.

```
% candidat(N:integer, I:integer)
% (i,o) - nedeterminist
candidat(N, N).
candidat(N, I) :-
    N>1,
    N1 is N-1,
    candidat(N1,I).
% permutari(N:integer, V:integer, L:list)
% (i,i,o) - nedeterminist
permutari(N, V, L) :-
    candidat(N, I),
    permutari_aux(N, V, L, 1, [I]).
% permutari_aux(N:integer, V:integer, L:list, Lg:integer, Col:list)
% (i,i,o,i,i) - nedeterminist
permutari_aux(N, _, Col, N, Col) :- !.
permutari_aux(N, V, L, Lg, [H|T]) :-
    candidat(N, I),
    abs(H-I)>=V,
    \+ apare(I, [H|T]), % (i,i)
    Lg1 is Lg+1,
    permutari_aux(N, V, L, Lg1, [I|H|T]).
```

**EXEMPLU 1.5** Se dă o mulțime de numere naturale nenule reprezentată sub forma unei liste. Să se determine toate posibilitățile de a scrie un număr N dat sub forma unei sume a elementelor din listă.

```
% list=integer*
% candidat(list, integer) (i, o) - nedeterminist
% un element posibil a fi adaugat in lista solutie

candidat([E|_],E).
candidat([_|T],E):-candidat(T,E).

% solutie(list,integer,list) (i,i,o) nedeterminist
% solutie_aux(list,integer,list,list,integer) (i,i,o,i,i) nedeterminist
% al patrulea argument colecteaza solutia, al cincilea argument
```

% reprezinta suma elementelor din colectoare

```
solutie(L, N, Rez) :-  
    candidat(L, E),  
    E =< N,  
    solutie_aux(L, N, Rez, [E], E).
```

```
solutie_aux(_, N, Rez, Rez, N):-!.  
solutie_aux(L, N, Rez, [H | Col], S) :-  
    candidat(L, E),  
    E < H,  
    S1 is S+E,  
    S1 =< N,  
    solutie_aux(L, N, Rez, [E | [H | Col]], S1).
```

### EXEMPLU 1.6 PROBLEMA CELOR 3 CASE.

1. Englezul locuiește în prima casă din stânga.
2. În casa imediat din dreapta celei în care se află lupul se fumează Lucky Strike.
3. Spaniolul fumează Kent.
4. Rusul are cal.

Cine fumează LM? Al cui este câinele?

Se observă că problema are două soluții:

I.	englez	câine	LM
	spaniol	lup	Kent
	rus	cal	LS
II.	englez	lup	LM
	rus	cal	LS
	spaniol	câine	Kent

Este o problemă de satisfacere a limitărilor (**constraint satisfaction**).

Codificăm datele problemei și observăm că o soluție e formată din triplete de forma (N, A, T) unde:

N aparține mulțimii	[eng, spa, rus]
A aparține mulțimii	[caine, lup, cal]
T aparține mulțimii	[lm, ls, ken]

Vom folosi următoarele predicate:

- predicatul nedeterminist **rezolva**(N, A, T) (**o, o, o**) care generează o soluție a problemei
- predicatul nedeterminist **candidati**(N, A, T) (**o, o, o**) care generează toți candidații la soluție
- predicatul determinist **restricții**(N, A, T) (**i, i, i**) care verifică dacă un candidat la soluție satisface restricțiile impuse de problemă
- predicatul nedeterminist **perm**(L, L1) (**i, o**) care generează permutările listei L

```
SWI-Prolog -- d:/Gabi/gabi/FACULTAT/2014-2015/PLF/DOC/Exemple_SWI/exemple.pl
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 29 clauses
% c:/users/istvan/appdata/roaming/swi-prolog/pl.ini compiled 0.00 sec, 1 clauses
% d:/Gabi/gabi/FACULTAT/2014-2015/PLF/DOC/Exemple_SWI/exemple.pl compiled 0.00 sec, 95 clauses
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.2.0)
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- rezolva(N,A,T).
N = [eng, spa, rus],
A = [caine, lup, cal],
T = [lm, kent, ls] ;
N = [eng, rus, spa],
A = [lup, cal, caine],
T = [lm, ls, kent] ;
false.

2 ?- █
```

**% rezolva - (o,o,o)**

**% candidati - (o,o,o)**

**% restrictii - (i,i,i)**

rezolva(N, A, T) :-

    candidati(N, A, T),

    restrictii(N, A, T).

candidati(N, A, T) :-

    perm([eng, spa, rus], N),

    perm([caine, lup, cal], A),

    perm([lm, kent, ls], T).

restrictii(N, A, T) :-

    aux(N, A, T, eng, \_, \_, 1),

    aux(N, A, T, \_, lup, \_, Nr),

    dreapta(Nr, M),

    aux(N, A, T, \_, \_, ls, M),

    aux(N, A, T, spa, \_, kent, \_),

    aux(N, A, T, rus, cal, \_, \_).

**% dreapta - (i,o)**

dreapta(I,J) :- J is I+1.

**% aux (i,i,i,o,o,o,o)**

aux([N1, \_, \_], [A1, \_, \_], [T1, \_, \_], N1, A1, T1, 1).

aux([\_, N2, \_], [\_, A2, \_], [\_, T2, \_], N2, A2, T2, 2).

aux([\_, \_, N3], [\_, \_, A3], [\_, \_, T3], N3, A3, T3, 3).

**% insereaza (i,io)**

insereaza(E, L, [E|L]).

insereaza(E, [H|L], [H|T]) :- insereaza(E,L,T).

**% perm (i,o)**

perm([], []).

perm([H|T], L):-perm(T, P),insereaza(H, P, L).

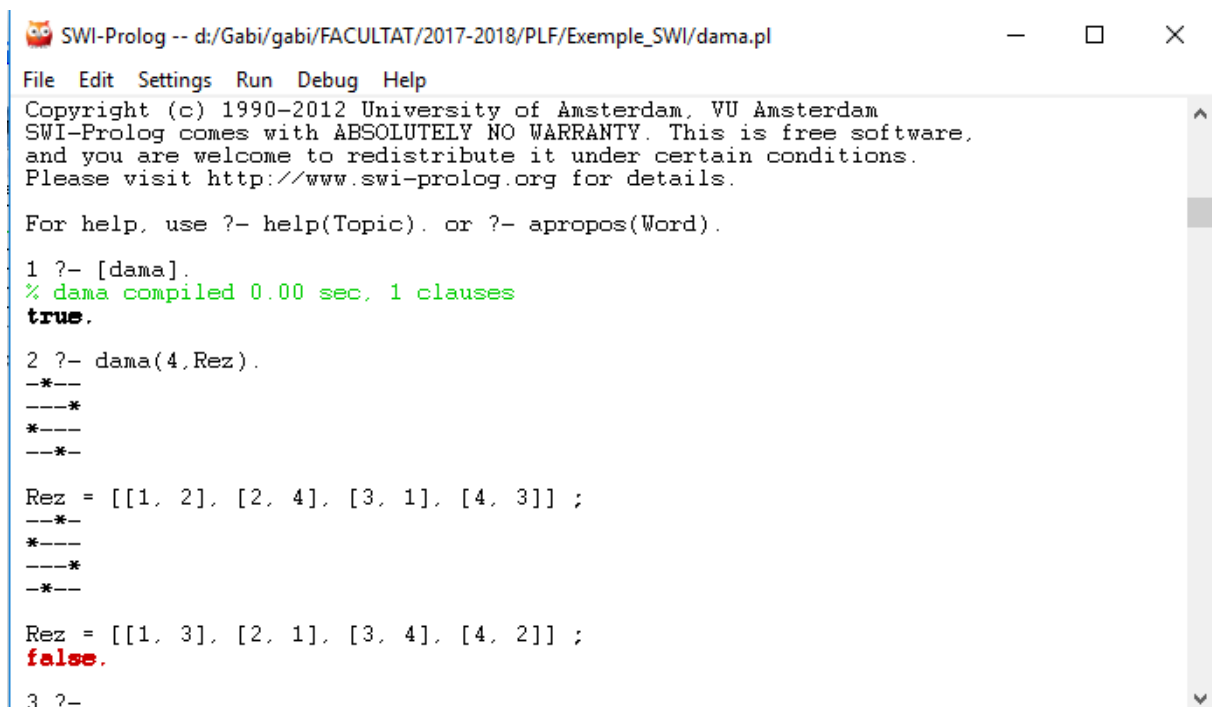
### PROBLEMA CELOR 5 CASE.

5 people live in the five houses in a street. Each has a different profession, animal, favorite drink, and each house is a different color.

1. The Englishman lives in the red house
2. The Spaniard owns a dog
3. The Norwegian lives in the first house on the left
4. The Japanese is a painter
5. The green house is on the right of the white one
6. The Italian drinks tea
7. The fox is in a house next to the doctor
8. Milk is drunk in the middle house
9. The horse is in a house next to the diplomat
10. The violinist drinks fruit juice
11. The Norwegians house is next to the blue one
12. The sculptor breeds snails
13. The owner of the green house drinks coffee
14. The diplomat lives in the yellow house

Who owns the zebra? Who drinks water?

**EXEMPLU 1.7** Să se dispună N dame pe o tablă de șah NxN, încât să nu se atace reciproc.



```
SWI-Prolog -- d:/Gabi/gabi/FACULTAT/2017-2018/PLF/Exemple_SWI/dama.pl
File Edit Settings Run Debug Help
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- [dama].
% dama compiled 0.00 sec, 1 clauses
true.

2 ?- dama(4,Rez).
*--
*--
*--
*--
Rez = [[1, 2], [2, 4], [3, 1], [4, 3]] ;
*--
*--
*--
*--
Rez = [[1, 3], [2, 1], [3, 4], [4, 2]] ;
false.

3 ?-
```

**% (integer,list\*) – (i, o) nedeterm.**

```
dama(N, Rez) :-  
    candidat(E, N),  
    dama_aux(N, Rez, [[N, E]], N),  
    tipar(N, Rez).
```

**% (integer,integer) – (o, i) nedeterm.**

```
candidat(N, N).  
candidat(E, I) :-  
    I>1,  
    I1 is I-1,  
    candidat(E, I1).
```

**% (integer,list\*,list\*,integer) – (i,o, i, i) nedeterm.**

```
dama_aux(_, Rez, Rez, 1) :- !.  
dama_aux(N, Rez, C, Lin) :-  
    candidat(Col1, N),  
    Lin1 is Lin-1,  
    valid(Lin1, Col1, C),  
    dama_aux(N, Rez, [[Lin1, Col1] | C], Lin1).
```

**% (integer,integer,list\*) – (i,i, i) determ.**

```
valid(_, _, []).  
valid(Lin, Col, [[Lin1,Col1] | T]) :-  
    Col =\= Col1,  
    DLin is Col-Col1,  
    DCol is Lin-Lin1,  
    abs(DLin) =\= abs(DCol),  
    valid(Lin, Col, T).
```

**% (integer, list\*) – (i,i) determ.**

```
tipar(_, []) :- nl.  
tipar(N, [[_, Col] | T]) :-  
    tipLinie(N, Col),  
    tipar(N, T).
```

**% (integer, char) – (i,o) determ.**

```
caracter(1, '*') :- !.  
caracter(_, '-').
```

**% (integer, list\*) – (i,i) determ.**

```
tipLinie(0, _) :- nl, !.  
tipLinie(N, Col) :-  
    caracter(Col, C),  
    write(C),  
    N1 is N-1,  
    Col1 is Col-1,  
    tipLinie(N1, Col1).
```