

## CURS 2

### Introducere în limbajul PROLOG

#### Cuprins

Bibliografie .....	1
1. Limbajul Prolog .....	1
1.1 Elemente de bază ale limbajului SWI-Prolog .....	4
1.2 “Matching”. Cum își primesc valori variabilele? .....	6
1.3 Modele de flux .....	7
1.4 Sintaxa regulilor .....	7
1.5 Operatori de egalitate .....	8
1.6 Operatori aritmetici .....	8
2. Exemplu – predicatul “factorial” .....	10

#### Bibliografie

Capitolul 11, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

#### 1. Limbajul Prolog

- Limbajul Prolog (PROgrammation en LOGique) a fost elaborat la Universitatea din Marsilia în jurul anului 1970, ca instrument pentru programarea și rezolvarea problemelor ce implicau reprezentări simbolice de obiecte și relații dintre acestea.
- Prolog are un câmp de aplicații foarte larg: baze de date relaționale, inteligență artificială, logică matematică, demonstrarea de teoreme, sisteme expert, rezolvarea de probleme abstracte sau ecuații simbolice, etc.
- Există standardul ISO-Prolog.
- Nu există standard pentru programare orientată obiect în Prolog, există doar extensii: TrincProlog, SWI-Prolog.
- Vom studia implementarea SWI-Prolog – sintaxa e foarte apropiată de cea a standardului ISO-Prolog.
  - Turbo Prolog, Visual Prolog, GNUProlog, Sicstus Prolog, Parlog, etc.
- SWI-Prolog – 1986
  - oferă o interfață bidirecțională cu limbajele C și Java

- folosește XPCP – un sistem GUI orientat obiect
- *multithreading* – bazat pe suportul *multithreading* oferit de limbajul standard C.

## Program Prolog

- caracter descriptiv: un program Prolog este o colecție de definiții ce descriu relații sau funcții de calculat – reprezentări simbolice de obiecte și relații între obiecte. Soluția problemelor nu se mai vede ca o execuție pas cu pas a unei secvențe de instrucțiuni.
  - program – colecție de declarații logice, fiecare fiind o clauză Horn de forma  $p, p \rightarrow q, p_1 \wedge p_2 \dots \wedge p_n \rightarrow q$
  - **concluzie** de demonstrat – de forma  $p_1 \wedge p_2 \dots \wedge p_n$
- **Structura de control** folosită de interpretorul Prolog
  - Se bazează pe declarații logice numite **clauze**
    - **fapt** – ceea ce se cunoaște a fi adevărat
    - **regulă** - ce se poate deduce din fapte date (indică o concluzie care se știe că e adevărată atunci când alte concluzii sau fapte sunt adevărate)
  - **Concluzie** ce trebuie demonstrată - GOAL
    - Prolog folosește *rezoluția* (liniară) pentru a demonstra dacă concluzia (teorema) este adevărată sau nu, pornind de la ipoteza stabilită de faptele și regulile definite (axiome).
    - Se aplică raționamentul înapoi pentru a demonstra concluzia
    - Programul este citit de sus în jos, de la dreapta la stânga, căutarea este în adâncime (*depth-first*) și se realizează folosind **backtracking**.
- $p \rightarrow q ( \neg p \vee q )$  se transcrie în Prolog folosind clauza  $q :- p.$  ( $q$  if  $p.$ )
- $\wedge$  se transcrie în Prolog folosind ",",
  - $p_1 \wedge p_2 \dots \wedge p_n \rightarrow q$  se transcrie în Prolog folosind clauza  $q :- p_1, p_2, \dots, p_n.$
- $\vee$  se transcrie în Prolog folosind ";" sau o clauză separată.
  - $p_1 \vee p_2 \rightarrow q$  se transcrie în Prolog
    - folosind clauza  $q :- p_1; p_2.$
    - sau
    - folosind 2 clauze separate
      - $q :- p_1.$
      - $q :- p_2.$

## Exemple

- Logică**

$$\forall x p(x) \wedge q(x) \rightarrow r(x)$$

$$\forall x w(x) \vee s(x) \rightarrow p(x)$$

**(SWI-)Prolog**

$$r(X) : -p(X), q(X).$$

$$p(X) : -w(X).$$

$$p(X) : -s(X).$$

$w \vee s \rightarrow p$	$\neg(w \vee s) \vee p$	$(\neg w \wedge \neg s) \vee p$	$(\neg w \vee p) \wedge (\neg s \vee p)$	$(w \rightarrow p) \wedge (s \rightarrow p)$
--------------------------	-------------------------	---------------------------------	--	--

$$\forall x t(x) \rightarrow s(x) \wedge q(x)$$

$$s(X) : -t(X).$$

$$q(X) : -t(X).$$

$$t(a)$$

$$t(a).$$

$$w(b)$$

$$w(b).$$

**Concluzie**

$$r(a)$$

**Goal**

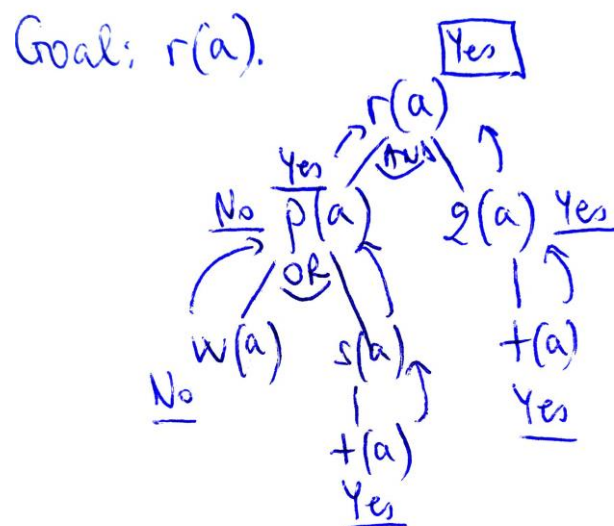
$$? r(a).$$

true

$$q(b)$$

$$? q(b).$$

**false**



- Logică**

$$\forall x s(x) \rightarrow p(x) \vee q(x)$$

**Prolog**

????

$s \rightarrow p \vee q$	... ..	... ..	$(s \wedge \neg p) \rightarrow q$
--------------------------	--------	--------	-----------------------------------

- [Lawrence C Paulson, \*Logic and Proof\*, University of Cambridge, 2000](#): rezoluție (subcapitolul 7.3), unificare (capitolul 10), Prolog

## 1.1 Elemente de bază ale limbajului SWI-Prolog

### 1. Termen

#### ▪ SIMPLU

##### a. constantă

- simbol (*symbol*)
  - secvență de litere, cifre, \_
  - începe cu **literă mică**
- număr =întreg, real (*number*)
- șir de caractere (*string*): ‘text’ (caracter: ‘c’, ‘\t’,...)

ATOM = SIMBOL + STRING +ȘIR-DE-CARACTERE-SPECIALE + [] (lista vidă)

- caractere speciale + \* / < > = : . & \_ ~

##### b. variabilă

- secvență de litere, cifre, \_
- începe cu **literă mare**
- variabila anonimă este reprezentată de caracterul underline (\_).

#### ▪ COMPUS (a se vedea Secțiunea 13).

- listele (*list*) sunt o clasă specială de termeni compuși

### 2. Comentariu

% Acesta este un comentariu

/\* Acesta este un comentariu \*/

### 3. Predicat

a). standard (ex: **fail**, **number**, ...)

b). utilizator

- $\text{nume} [(\text{obiect}[, \text{obiect}....])]$   
 ↓  
 numele simbolic al relației

*Tipuri*

1. **number** (integer, real)
2. **atom** (symbol, string, șir-de-caractere-speciale)

### 3. **list** (secvență de elemente) specificat ca **list=tip\_de\_bază\***

ex. listă (omogenă) formată din numere întregi [1,2,3]

% definire tip:            el=integer        list=el\*

**!!! lista vidă [] este singura listă care e considerată în Prolog atom.**

### Convenții.

- În SWI-Prolog nu există declarații de predicate, nici declarații de domenii/tipuri (ex. ca în Turbo-Prolog).
- *Specificarea unui predicat*
  - % definire tipuri, dacă e cazul
  - % *nume* [(param<sub>1</sub>:tip<sub>1</sub>[,param<sub>2</sub>:tip<sub>2</sub>...)]
  - % modelul de flux al predicatului (i, o, ...) - vezi Secțiunea 4
  - % param<sub>1</sub> - semnificația parametrului 1
  - % param<sub>2</sub> - semnificația parametrului 2
  - ....

### 4. Clauza

- fapt
  - relație între obiecte
  - *nume\_predicat* [(obiect [, obiect....)]
- regula
  - permite deducere de fapte din alte fapte

#### Exemplu:

fie predicatele

**tata**(X, Y) reprezentând relația “Y este tatăl lui X”

**mama**(X, Y) reprezentând relația “Y este mama lui X”

și următoarele fapte corespunzătoare celor două predicate:

mama(a,b).

mama(e,b).

tata(c,d).

tata(a,d).

**Se cere:** folosind definițiile anterioare să se definească predicatele

**parinte**(X, Y) reprezentând relația “Y este părintele lui X”

**frate**(X, Y) reprezentând relația “Y este fratele lui X”

### **Clauze în SWI-Prolog**

```

parinte(X,Y) :-tata(X,Y).
parinte(X,Y) :-mama(X,Y).
% “\=” reprezintă operatorul “diferit” – a se vedea Secțiunea 1.5
frate(X,Y) :- parinte(X,Z),
               parinte(Y,Z),
               X \= Y.

```

## 5. Intrebare (goal)

- e de forma  $\text{predicat}_1 [(obiect [, obiect....)], \text{predicat}_2 [(obiect [, obiect....)]....]$ .
- **true, false**
- **CWA – Closed World Assumption**

Folosind definițiile anterioare, formulăm următoarele întrebări:

?- parinte(a,b).	?- parinte(a,X).
<b>true.</b>	X=d;
	X=b.
? - parinte(a,f).	
<b>false.</b>	
?- frate(a,X).	?- frate(a,_).
X=c;	<b>true.</b>
X=e.	

## 1.2 “Matching”. Cum își primesc valori variabilele?

Prolog nu are instrucțiuni de atribuire. Variabilele în Prolog își primesc valorile prin **potrivire** cu constante din fapte sau reguli.

Până când o variabilă primește o valoare, ea este **liberă** (free); când variabila primește o valoare, ea este **legată** (bound). Dar ea stă legată atâta timp cât este necesar pentru a obține o soluție a problemei. Apoi, Prolog o dezleagă, face backtracking și caută soluții alternative.

**Observație.** Este important de reținut că nu se pot stoca informații prin atribuire de valori unor variabile. Variabilele sunt folosite ca parte a unui proces de potrivire, nu ca un tip de stocare de informații.

## Ce este o potrivire?

Iată câteva reguli care vor explica termenul 'potrivire':

1. Structuri identice se potrivesc una cu alta
  - $p(a, b)$  se potrivește cu  $p(a, b)$
2. De obicei o potrivire implică variabile libere. Dacă X e liberă,

- $p(a, X)$  se potrivește cu  $p(a, b)$
  - $X$  este legat la  $b$ .
3. Dacă  $X$  este legat, se comportă ca o constantă. Cu  $X$  legat la  $b$ ,
- $p(a, X)$  se potrivește cu  $p(a, b)$
  - $p(a, X)$  NU se potrivește cu  $p(a, c)$
4. Două variabile libere se potrivesc una cu alta.
- $p(a, X)$  se potrivește cu  $p(a, Y)$

**Observație.** Mecanismul prin care Prolog încearcă să ‘potrivească’ partea din întrebare pe care dorește să o rezolve cu un anumit predicat se numește **unificare**.

### 1.3 Modele de flux

În Prolog, legările de variabile se fac în două moduri: la intrarea în clauză sau la ieșirea din clauză. Direcția în care se leagă o valoare se numește ‘**model de flux**’. Când o variabilă este dată la intrarea într-o clauză, aceasta este un parametru de intrare (i), iar când o variabilă este dată la ieșirea dintr-o clauză, aceasta este un parametru de ieșire (o). O anumită clauză poate să aibă mai multe modele de flux. De exemplu clauza

factorial (N, F)

poate avea următoarele modele de flux:

- (i,i) - verifică dacă  $N! = F$ ;
- (i,o) - atribuie  $F := N!$ ;
- (o,i) - găsește acel  $N$  pentru care  $N! = F$ .

**Observație.** Proprietatea unui predicat de a funcționa cu mai multe modele de flux depinde de abilitatea programatorului de a programa predicatul în mod corespunzător.

### 1.4 Sintaxa regulilor

Regulile sunt folosite în Prolog când un fapt depinde de succesul (veridicitatea) altor fapte sau succesiuni de fapte. O regulă Prolog are trei părți: capul, corpul și simbolul if (:-) care le separă pe primele două.

Iată sintaxa generică a unei reguli Prolog:

capul regulii :-  
           subgoal,  
           subgoal,  
           ...,  
           subgoal.

Fiecare subgoal este un apel la un alt predicat Prolog. Când programul face acest apel, Prolog testează predicatul apelat să vadă dacă poate fi adevărat. Odată ce subgoal-ul curent a fost satisfăcut (a fost găsit adevărat), se revine și procesul continuă cu următorul subgoal. Dacă

procesul a ajuns cu succes la punct, regula a reușit. Pentru a utiliza cu succes o regulă, Prolog trebuie să satisfacă toate subgoal-urile ei, creând o mulțime consistentă de legări de variabile. Dacă un subgoal eșuează (este găsit fals), procesul revine la subgoal-ul anterior și caută alte legări de variabile, și apoi continuă. Acest mecanism se numește **backtracking**.

## 1.5 Operatori de egalitate

$X=Y$  verifică dacă  $X$  și  $Y$  pot fi unificate

- Dacă  $X$  este variabilă liberă și  $Y$  legată, sau  $Y$  este variabilă liberă și  $X$  e legată, propoziția este satisfăcută unificând pe  $X$  cu  $Y$ .
- Dacă  $X$  și  $Y$  sunt variabile legate, atunci propoziția este satisfăcută dacă relația de egalitate are loc.

?-  $[a,b]=[a,b]$ .  
**true.**

?-  $[X,Y]=[a,b]$ .  
 $X = a,$   
 $Y = b.$

?-  $[a,b]=[X,Y]$ .  
 $X = a,$   
 $Y = b.$

$X \neq Y$  verifică dacă  $X$  și  $Y$  nu pot fi unificate  
 $\neq X=Y$

?-  $[X,Y,Z] \neq [a,b]$ .  
**true.**

?-  $[X,Y] \neq [a,b]$ .  
**false.**

?-  $[a,b] \neq [X,Y]$ .  
**false.**

?-  $\neq a=a$ .  
**false.**

?-  $\neq [X,Y]=[a,b]$ .  
**false.**

?-  $\neq [a,b]=[X,Y,Z]$ .  
**true.**

$X == Y$  verifică dacă  $X$  și  $Y$  sunt legate la aceeași valoare.

?-  $[2,3]==[2,3]$ .  
**true.**

?-  $a==a$ .  
**true.**

?-  $R==1$ .  
**false.**

$X \neq Y$  verifică dacă  $X$  și  $Y$  nu au fost legate la aceeași valoare.

?-  $[2,3] \neq [3,2]$ .  
**true.**

?-  $a \neq a$ .  
**false.**

?-  $R \neq 1$ .  
**true.**

## 1.6 Operatori aritmetici

### !!! Important

- $2+4$  e doar o structură, utilizarea sa nu efectuează adunarea
- Utilizarea  $2+4$  nu e aceeași ca utilizarea lui 6.

### Operatori aritmetici

$=, \neq, ==, \neq$  A se vedea Secțiunea 1.5.



?- 2+4=6.      ?- 2+4\=6.      ?- 6==6.      ?- 6\=7.      ?- 6==2+4.  
**false.**      **true.**      **true.**      **true.**      **false.**

?- 2+4=2+4.      ?- 2+4=4+2.      ?- X= 2+4-1.  
**true.**      **false.**      X=2+4-1.

**==**

- testează egalitatea aritmetică
- forțează evaluarea aritmetică a ambelor părți
- operanzii trebuie să fie numerici
- variabilele sunt LEGATE

**=\=**      testează operatorul aritmetic "diferit"

?- 2+4==6.      ?- 2+4=\=7.      ?- 6==6.  
**true.**      **true.**      **true.**

**is**

- partea dreaptă este LEGATĂ și numerică
- partea stângă trebuie să fie o variabilă
- dacă variabila este legată, verifică egalitatea numerică (ca și ==)
- dacă variabila nu este legată, evaluează partea dreaptă și apoi variabila este legată de rezultatul evaluării

?- X is 2+4-1.      ?- X is 5.  
X=5      X=5

## Inegalități

<	mai mic
=<	mai mic sau egal
>	mai mare
>=	mai mare sau egal

- evaluează ambele părți
- variabile LEGATE

?- 2+4=<5+2.      ?- 2+4=\=7.      ?- 6==6.  
**true.**      **true.**      **true.**

## Câteva funcții aritmetice predefinite SWI-Prolog

X mod Y      întoarce restul împărțirii lui X la Y  
mod(X, Y)

X div Y      întoarce câtul împărțirii lui X la Y  
div(X, Y)

abs(X)	întoarce valoarea absolută a lui X
sqrt(X)	întoarce rădăcina pătrată a lui X
round(X)	întoarce valoarea lui X rotunjită spre cel mai apropiat întreg (round(2.56) este 3, round (2.4) este 2)
...	

## 2. Exemplu – predicatul “factorial”

Dându-se un număr natural  $n$ , să se calculeze factorialul numărului.

$$fact(n) = \begin{cases} 1 & \text{daca } n = 0 \\ n \cdot fact(n-1) & \text{altfel} \end{cases}$$

Conform cerinței probleme, am dori să definim predicatul **fact(integer, integer)** în modelul de flux (**i, o**). Vom vedea că predicatul definit în acest model de flux funcționează și în modelul de flux (**i, i**).

În Varianta 2, ! reprezintă predicatul “cut” (tăietura roșie, în acest context), folosit pentru a preveni luarea în calcul a subgoal-urilor alternative (backtracking-ul la următoarea clauză).

### 1. Varianta 1

```
% fact1(N:integer, F:integer)
% (i, i) (i, o)
fact1(0, 1).
fact1(N, F) :- N > 0,
               N1 is N-1,
               fact1(N1, F1),
               F is N * F1.
```

```
go1 :- fact1(3, 6).
```

### 2. Varianta 2

```
% fact1(N:integer, F:integer)
% (i, i) (i, o)
fact2(0, 1) :- !.
fact2(N, F) :- N1 is N-1,
               fact2(N1, F1),
               F is N * F1.
```

```
go2 :- fact2(3, 6).
```

```
SWI-Prolog (Multi-threaded, version 6.6.6)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.6)
Copyright (c) 1990-2013 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 ?-
% d:/Docs/Didactice/Cursuri/2014-15/pfl/teste/fact.pl compiled 0.00 sec, 7 clauses
1 ?- fact1(3,6).
true ;
false.

2 ?- fact1(3,X).
X = 6 ;
false.

3 ?- go1.
true ;
false.

4 ?- fact2(3,6).
true.

5 ?- fact2(3,X).
X = 6.

6 ?- go2.
true.

7 ?- █
```

### 3. Varianta 3

**% fact3(N:integer, F:integer)**

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

```
fact3(N, F) :- N > 0,
               N1 is N-1,
               fact3(N1, F1),
               F is N * F1.
```

```
fact3(0, 1).
```

```
?- fact3(3, N).
```

```
N = 6.
```

## CURS 3

### Predicate deterministe și nedeterministe. Exemple

#### Cuprins

1. Predicate deterministe și nedeterministe.....	1
1.1 Predicate predefinite .....	1
1.2 Predicatul „findall“ (determinarea tuturor soluțiilor) .....	2
1.3 Negatie - “not”, “\+” .....	2
1.4 Liste și recursivitate .....	2
1.4.1 Capul și coada unei liste (head&tail) .....	3
1.4.2 Procesarea listelor .....	3
1.4.3 Utilizarea listelor.....	3
2. Exemple .....	4

#### Bibliografie

Capitolul 14, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

### 1. Predicate deterministe și nedeterministe

Tipuri de predicate

- **deterministe**
  - un predicat determinist are o singură soluție
- **nedeterministe**
  - un predicat nedeterminist are mai multe soluții

**Observație.** Un predicat poate fi determinist într-un model de flux, nedeterminist în alte modele de flux.

#### 1.1 Predicate predefinite

var(X)	= adevărat dacă X e liberă, fals dacă e legată
number(X)	= adevărat dacă X e legată la un număr
integer(X)	= adevărat dacă X e legată la un număr întreg
float(X)	= adevărat dacă X e legată la un număr real
atom(X)	= adevărat dacă X e legată la un atom
atomic(X)	= atom(X) or number(X)
....	

## 1.2 Predicatul „findall” (determinarea tuturor soluțiilor)

Prolog oferă o modalitate de a găsi toate soluțiile unui predicat în același timp: predicatul **findall**, care colectează într-o listă toate soluțiile găsite.

**findall** (arg1, arg2, arg3)

Acesta are următoarele argumente:

- primul argument specifică argumentul din predicatul considerat care trebuie colectat în listă;
- al doilea argument specifică predicatul de rezolvat;
- al treilea argument specifică lista în care se vor colecta soluțiile.

### EXEMPLU

p(a, b).  
p(b, c).  
p(a, c).  
p(a, d).  
toate(X, L) :- forall(Y, p(X, Y), L).

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

## 1.3 Negație - “not”, “\+”

**not**(subgoal(Arg1, ..., ArgN))

adevărat dacă *subgoal* eșuează (nu se poate demonstra că este adevărat)

**\+** subgoal(Arg1, ..., ArgN)

?- \+ (2 = 4).  
**true.**

?- not(2 = 4).  
**true.**

## 1.4 Liste și recursivitate

În Prolog, o listă este un obiect care conține un număr arbitrar de alte obiecte. Listele în SWI-Prolog sunt eterogene (elementele componente pot avea tipuri diferite). Listele se construiesc folosind parantezele drepte. Elementele acestora sunt separate de virgulă.

Iată câteva exemple:

[1, 2, 3]  
[dog, cat, canary]  
[“valerie ann”, “jennifer caitlin”, “benjamin thomas”]

Dacă ar fi să declarăm tipul unei liste (omogenă) cu elemente numere întregi, s-ar folosi o declarație de domeniu de tipul următor

element = integer  
list = element\*

### 1.4.1 Capul și coada unei liste (head&tail)

O listă este un obiect realmente recursiv. Aceasta constă din două părți: **capul**, care este primul element al listei și **coada**, care este restul listei. Capul listei este element, iar coada listei este listă.

Iată câteva exemple:

Capul listei [a, b, c] este a

Coada listei [a, b, c] este [b, c]

Capul listei [c] este c

Coada listei [c] este []

Lista vidă [] nu poate fi împărțită în cap și coada.

### 1.4.2 Procesarea listelor

Prolog oferă o modalitate de a face explicite capul și coada unei liste. În loc să separăm elementele unei liste cu virgule, vom separa capul de coadă cu caracterul '|'.

De exemplu, următoarele liste sunt echivalente:

[a, b, c]      [a | [b, c]]      [a | [b | [c]]]      [a | [b | [c | []]]]

De asemenea, înaintea caracterului '|' pot fi scrise mai multe elemente, nu doar primul. De exemplu, lista [a, b, c] de mai sus este echivalentă cu

[a | [b, c]]      [a, b | [c]]      [a, b, c | []]

- În urma unificării listei [a, b, c] cu lista [H | T] (H, T fiind variabile libere)
  - H se leagă la a; T se leagă la [b, c]
- În urma unificării listei [a, b, c] cu lista [H | [H1 | T]] (H, H1, T fiind variabile libere)
  - H se leagă la a; H1 se leagă la b; T se leagă la [c]

### 1.4.3 Utilizarea listelor

Deoarece o listă (înlănțuită) este o structură de date recursivă, pentru procesarea ei este nevoie de algoritmi recursivi. Modul de bază de procesare a listei este acela de a lucra cu ea, executând anumite operații cu fiecare element al ei, până când s-a atins sfârșitul.

Un algoritm de acest tip are nevoie în general de două clauze. Una dintre ele spune ce să se facă cu o listă vidă. Cealaltă spune ce să se facă cu o listă nevidă, care se poate descompune în cap și coadă.

## 2. Exemple

**EXEMPLU 2.1** Dându-se un număr natural  $n$  nenul, se cere să se calculeze  $F=n!$ . Se va simula procesului iterativ de calcul.

```
 $i \leftarrow 1$   
 $P \leftarrow 1$   
Cât $Timp$   $i < n$  execută  
     $i \leftarrow i + 1$   
     $P \leftarrow P * i$   
SfCât $Timp$   
 $F \leftarrow P$ 
```

$fact(n) = fact\_aux(n, 1, 1)$

$$fact\_aux(n, i, P) = \begin{cases} P & \text{daca } i = n \\ fact\_aux(n, i + 1, P * (i + 1)) & \text{altfel} \end{cases}$$

- descrierea nu este direct recursivă, se folosesc variabile colectoare ( $i, P$ )

```
% fact(N:integer, F:integer)  
% (i, i), (i, o) - determinist  
fact(N, F) :- fact_aux(N, F, 1, 1).
```

```
% fact_aux(N:integer, F:integer, I:integer, P:integer)  
% (i, i, i, i), (i, o, i, i) - determinist  
fact_aux(N, F, N, F) :- !. % fact_aux(N, F, I, P) :- I is N, F is P, !.  
fact_aux(N, F, I, P) :- I1 is I+1,  
    P1 is P*I1,  
    fact_aux(N, F, I1, P1).
```

Rezultatul este o recursivitate de coadă (a se vedea **Cursul 6**). Toate variabilele de ciclare au fost introduse ca argumente ale predicatului **fact\_aux**.

**TEMĂ** Scrieți un predicat factorial ( $N, F$ ) care să funcționeze în toate cele 3 modele de flux  $(i, i)$ ,  $(i, o)$  și  $(o, i)$ .

## **EXEMPLU 2.2** Verificați apartenența unui element într-o listă.

% (i, i)

Pentru a descrie apartenența la o listă vom construi predicatul `member(element, list)` care va investiga dacă un anumit element este membru al listei. Algoritmul de implementat ar fi (din punct de vedere declarativ) următorul:

1. E este membru al listei L dacă este capul ei.
2. Altfel, E este membru al listei L dacă este membru al cozii lui L.

Din punct de vedere procedural,

1. Pentru a găsi un membru al listei L, găsește-i capul;
2. Altfel, găsește un membru al cozii listei L.

$$\text{member}(E, l_1 l_2 \dots l_n) = \begin{cases} \text{fals} & \text{daca } l \text{ e vida} \\ \text{adevarat} & \text{daca } l_1 = E \\ \text{member}(E, l_2 \dots l_n) & \text{altfel} \end{cases}$$

### **1. Varianta 1**

```
% member(e:element, L:list)
% (i, i) - determinist, (o, i) - nedeterminist
member1(E,[E|_]).
member1(E,[_|L]) :- member1(E,L).

go1 :- member1(1,[1,2,1,3,1,4]).
```

### **2. Varianta 2**

```
% member(e:element, L:list)
% (i, i) - determinist, (o, i) - nedeterminist
member2(E,[E|_]) :- !.
member2(E,[_|L]) :- member2(E,L).

go2 :- member2(1,[1,2,1,3,1,4]).
```



### 3. Varianta 3

% member(e:element, L:list)

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

member3(E,[\_|L]) :- member3(E,L).

member3(E,[E|\_]).

?- member3(E,[1,2,3]).

E=3 ;

E=2 ;

E=1.

?-member3(4, [1,2,3]).

false.

?- member3(2,[1,2,3]).

true ;

false.

După cum se observă, predicatul **member** funcționează și în modelul de flux (o, i), în care este **nedeterminist**.

Pentru a descrie predicatul *member* în modelul de flux (o, i) – o soluție să fie câte un element al listei -

?- member3(E,[1,2,3]).

E=1;

E=2;

E=3.

$member(l_1, l_2, \dots, l_n) =$

1.  $l_1$             *daca l e nevida*

2.  $member(l_2, \dots, l_n)$

### **EXEMPLU 2.3** Adăugarea unui element la sfârșitul unei liste

? adaug(3, [1, 2], L).

L = [1, 2, 3]

Formula recursivă:

$$adaug(e, l_1 l_2 \dots l_n) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus adaug(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

#### Varianta 1

% adaug(e:element, L:list, LRez: list)

% (i, i, o) - determinist

adaug(E, [], [E]). % adaug(E, [], Rez) :- Rez = [E].

adaug(E, [H | T], [H | Rez]) :-

    adaug(E, T, Rez).

% adaug(E, [H | T], Rez) :-

    adaug(E, T, L), Rez = [H | L].

#### Varianta 2

% adaug(L:list, e:element, LRez: list)

% (i, i, o) - determinist

adaug([], E, [E]).

adaug([H | T], E, [H | Rez]) :-

    adaug(T, E, Rez).

% adaug([H | T], E, Rez) :-

    adaug(T, E, L), Rez = [H | L].

Complexitatea timp a operației de adăugare a unui element la sfârșitul unei liste cu  $n$  elemente este  $\theta(n)$ .

### **EXEMPLU 2.4** Să se determine inversa unei liste.

**Varianta A** (direct recursiv)

$$\text{invers}(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ \text{invers}(l_2 \dots l_n) \oplus l_1 & \text{altfel} \end{cases}$$

% invers(L:list, LRez: list)

% (i, o) - determinist

invers([], []).

invers([H | T], Rez) :-

invers(T, L), adaug(H, L, Rez).

Complexitatea timp a operației de inversare a unei liste cu  $n$  elemente (folosind adăugarea la sfârșit) este  $\theta(n^2)$ .

**Varianta B** (cu variabilă colectoare)

Se va folosi o variabilă colectoare **Col**, pentru a colecta inversa listei, pas cu pas.

L	Col
[1, 2, 3]	$\emptyset$
[2, 3]	[1]
[3]	[2, 1]
$\emptyset$	[3, 2, 1]

$$\text{invers\_aux}(l_1 l_2 \dots l_n \text{ Col}) = \begin{cases} \text{Col} & \text{daca } l \text{ e vida} \\ \text{invers\_aux}(l_2 \dots l_n, l_1 \oplus \text{Col}) & \text{altfel} \end{cases}$$

$$\text{invers}(l_1 l_2 \dots l_n) = \text{invers\_aux}(l_1 l_2 \dots l_n, \emptyset)$$

% invers(L:list, LRez: list)

% (i, o) – determinist

invers(L, Rez) :- invers\_aux([], L, Rez).

% invers\_aux(Col:list, L:list, LRez: list) – primul argument e colectoarea

% (i, i, o) – determinist

invers\_aux(Col, [], Col). % invers\_aux(Col, [], Rez) :- Rez = Col.

invers\_aux(Col, [H|T], Rez) :-

invers\_aux([H|Col], T, Rez).

Complexitatea timp a operației de inversare a unei liste cu  $n$  elemente (folosind o variabilă colectoare) este  $\theta(n)$ .

**Observație.** Folosirea unei variabile colectoare nu reduce complexitatea, în toate cazurile. Sunt situații în care folosirea unei variabile colectoare crește complexitatea (ex: adăugarea în colectoare se face la sfârșitul acesteia, nu la început).

**EXEMPLU 2.5** Să se determine lista elementelor pare dintr-o listă (se va păstra ordinea elementelor din lista inițială).

**Varianta A** (direct recursiv)

$$pare(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ l_1 \oplus pare(l_2 \dots l_n) & \text{daca } l_1 \text{ par} \\ pare(l_2 \dots l_n) & \text{altfel} \end{cases}$$

% pare(L:list, LRez: list)

% (i, o) – determinist

pare([], []).

pare([H|T], [H |Rez]) :-

H mod 2 =:= 0,

!,

pare(T, Rez).

pare([\_|T], Rez) :-

pare(T, Rez).

Complexitatea timp a operației este  $\theta(n)$ ,  $n$  fiind numărul de elemente din listă.

$$T(n) = \begin{cases} 1 & \text{daca } n = 0 \\ T(n-1) + 1 & \text{altfel} \end{cases}$$

**Varianta B** (cu variabilă colectoare)

$$pare\_aux(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & \text{daca } l \text{ e vida} \\ pare\_aux(l_2 \dots l_n, Col \oplus l_1) & \text{daca } l_1 \text{ par} \\ pare\_aux(l_2 \dots l_n, Col) & \text{altfel} \end{cases}$$

$$pare(l_1 l_2 \dots l_n) = pare\_aux(l_1 l_2 \dots l_n, \emptyset)$$

% pare(L:list, LRez: list)

% (i, o) – determinist

pare(L, Rez) :-

pare\_aux(L, Rez, []).

% pare(L:list, LRez: list, Col: list)

```

% (i, o, i) – determinist
pare_aux([], Rez, Rez).
pare_aux([H|T], Rez, Col) :-
    H mod 2 =:= 0,
    !,
    adaug(H, Col, ColN), % adăugare la sfârșit
    pare_aux(T, Rez, ColN).
pare_aux([_|T], Rez, Col) :-
    pare_aux(T, Rez, Col).

```

Complexitatea timp în caz defavorabil este  $\theta(n^2)$ ,  $n$  fiind numărul de elemente din listă.

**EXEMPLU 2.6** Dându-se o listă numerică, se cere un predicat care determină lista perechilor de elemente strict crescătoare din listă.

```

? perechi([2, 1, 3, 4], L)
L = [[2, 3], [2, 4], [1, 3], [1, 4], [3, 4]]

? perechi([5, 4, 2], L)
false

```

Vom folosi următoarele predicate:

- predicatul nedeterminist **pereche**(element, lista) (model de flux (i, o)), care va produce perechi în ordine crescătoare între elementul dat și elemente ale listei argument
 

```

? pereche(2, [1, 3, 4], L)
L = [2, 3]
L = [2, 4]

```

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

1.  $(e, l_1) \quad e < l_1$
2.  $pereche(e, l_2, \dots, l_n)$

```

% pereche(E: element, L:list, LRez: list)
% (i, i, o) – nedeterminist
pereche(A, [B|_], [A, B]) :-
    A < B.
pereche(A, [_|T], P) :-
    pereche(A, T, P).

```

- predicatul nedeterminist **perechi**(lista, lista) (model de flux (i, o)), care va produce perechi în ordine crescătoare între elementele listei argument
 

```

? perechi([2, 1, 4], L)
L = [2, 4]
L = [1, 4]

```

$perechi(l_1, l_2, \dots, l_n) =$

1.  $pereche(l_1, l_2, \dots, l_n)$
2.  $perechi(l_2, \dots, l_n)$

**% perechi(L:list, LRez: list)**

**% (i, o) – nedeterminist**

perechi([H|T], P) :-

    pereche(H, T, P).

perechi([\_|T], P) :-

    perechi(T, P).

- predicatul principal **toatePerechi**(lista, listad) (model de flux (i, o)), care va colecta toate soluțiile predicatului nedeterminist **perechi**.

**% toatePerechi(L:list, LRez: list)**

**% (i, o) –determinist**

toatePerechi(L, LRez) :-

    findall(X, perechi(L, X), LRez).

## CURS 4

### Controlarea backtrackingului. Liste eterogene

#### Cuprins

1. Controlarea backtracking-ului. Predicatele “cut” (tăietura) și fail.....	1
1.1 Predicatul ! (cut) – “tăietura” .....	1
1.2 Predicatul “fail” .....	4
2. Obiecte simple și obiecte compuse. ....	6
3. Exemple .....	8

**Exemplu** Fie următoarele definiții de predicate.

1) Care este efectul următoarelor interogări? Apare **false** la finalul căutării?

? g([1, 3, 4], 2, P).

? f([1,2,3,4], P).

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

% (i, i, o) – nedeterminist

g([H|\_], E, [E,H]).

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

    g(T, E, P).

% f(L:list, LRez: list)

% (i, o) – nedeterminist

f([H|T],P):-

    g(T, H, P).

f([\_|T], P):-

    f(T, P).

2) Funcționează predicatele în alte modele de flux?

?- g([1,2,3], E, [4,1]).

### 1. Controlarea backtracking-ului. Predicatele “cut” (tăietura) și fail

#### 1.1 Predicatul ! (cut) – “tăietura”

Limbajul Prolog conține predicatul cut (!) folosit pentru a preveni backtracking-ul. Când se procesează predicatul !, apelul reușește imediat și se trece la subgoalul următor. O dată ce s-a trecut peste o tăietură, nu este posibilă revenirea la subgoal-urile plasate înaintea ei și nu este posibil backtracking-ul la alte reguli ce definesc predicatul în execuție.

Există două utilizări importante ale tăieturii:

1. Când știm dinainte că anumite posibilități nu vor duce la soluții, este o pierdere de timp să lăsăm sistemul să lucreze. În acest caz, tăietura se numește **tăietură verde**.
2. Când logica programului cere o tăietură, pentru prevenirea luării în considerație a subgoal-urilor alternative, pentru a evita obținerea de soluții eronate. În acest caz, tăietura se numește **tăietură roșie**.

### Prevenirea backtracking-ului la un subgoal anterior

În acest caz tăietura se utilizează astfel:  $r1 :- a, b, !, c.$

Aceasta este o modalitate de a spune că suntem mulțumiți cu primele soluții descoperite cu subgoal-urile a și b. Deși Prolog ar putea găsi mai multe soluții prin apelul la c, nu este autorizat să revină la a și b. De asemenea, nu este autorizat să revină la altă clauză care definește predicatul r1.

### Prevenirea backtracking-ului la următoarea clauză

Tăietura poate fi utilizată pentru a-i spune sistemului Prolog că a ales corect clauza pentru un predicat particular. De exemplu, fie codul următor:

```
r(1) :- !, a, b, c.  
r(2) :- !, d.  
r(3) :- !, e.  
r(_) :- write("Aici intră restul apelurilor").
```

Folosirea tăieturii face predicatul r determinist. Aici, Prolog apelează predicatul r cu un argument întreg. Să presupunem că apelul este r(1). Prolog caută o potrivire a apelului. O găsește la prima clauză. Faptul că imediat după intrarea în clauză urmează o tăietură, împiedică Prolog să mai caute și alte potriviri ale apelului r(1) cu alte clauze.

**Observație.** Acest tip de structură este echivalent cu o instrucțiune de tip *case* unde condiția de test a fost inclusă în capul clauzei. La fel de bine s-ar fi putut spune și

```
r(X) :- X = 1, !, a, b, c.  
r(X) :- X = 2, !, d.  
r(X) :- X = 3, !, e.  
r(_) :- write("Aici intra restul apelurilor").
```

**Notă.** Deci, următoarele secvențe sunt echivalente:

case X of	
v1: corp1;	predicat(X) :- X = v1, !, corp1.
v2: corp2;	predicat(X) :- X = v2, !, corp2.
...	...
else corp_else.	predicat(X) :- corp_else.

De asemenea, următoarele secvențe sunt echivalente:



```

if cond1 then
    corp1
else if cond2 then
    corp2
...
else
    corp_else.

```

```

predicat(...) :-
    cond1, !, corp1.
predicat(...) :-
    cond2, !, corp2.
...
predicat(...) :-
    corp_else.

```

## EXAMPLE

### 1. Fie următorul cod Prolog

<pre> % p(E: integer) % (o) – nedeterminist p(1). p(2). % q(E: integer) % (o) – nedeterminist q(3). q(4). </pre>	<pre> % r(E1: integer, E2: integer) % (o, o) – nedeterminist % V1 r(X, Y) :- !, p(X), q(Y). % V2 r(X, Y) :- p(X), !, q(Y). % V3 r(X, Y) :- p(X), q(Y), !. </pre>
--	--

Care sunt soluțiile furnizate la goal-ul

? r(X,Y).

în cele 3 variante (V1, V2, V3) de definire a preducaturii r?

**R:**

- V1 - sunt 4 soluții:  
X=1 Y=3  
X=1 Y=4  
X=2 Y=3  
X=2 Y=4
- V2 - sunt 2 soluții:  
X=1 Y=3  
X=1 Y=4
- V3 – e 1 soluție:  
X=1 Y=3

### 2. Fie următorul cod Prolog

```

% p(L: list, S:integer)
% (i, o) –determinist
p([], 0).
p([H|T], S) :-
    H > 0,
    !,
    p(T, S1),
    S is S1 + H.

```

```

? p([1, 2, 3, 4], S).
S=10.

```

```

? p([1, -1, 2, -2], S).
S=3.

```

$p([\_|T], S) :- p(T, S).$

Ce se întâmplă dacă se mută tăietura înaintea condiției, adică dacă în cea de-a doua clauză se mută tăietura înaintea condiției?

**% p(L: list, S:integer)**

**% (i, o) –determinist**

$p([], 0).$

$p([H|T], S) :-$

**!,**

$H > 0,$

$p(T, S1),$

$S \text{ is } S1 + H.$

$p([\_|T], S) :- p(T, S).$

$? p([1, 2, 3, 4], S).$

$S=10.$

$? p([1, -1, 2, -2], S).$

**false**

## 1.2 Predicatul “fail”

Valoarea lui *fail* este eșec. Prin aceasta el încurajează backtracking-ul. Efectul lui este același cu al unui predicat imposibil, de genul  $2 = 3$ . Fie următorul exemplu:

$\text{predicat}(a, b).$

$\text{predicat}(c, d).$

$\text{predicat}(e, f).$

$\text{toate} :-$

$\text{predicat}(X, Y),$

$\text{write}(X), \text{write}(Y), \text{nl},$

$\text{fail}.$

$\text{toate1} :-$

$\text{predicat}(X, Y),$

$\text{write}(X), \text{write}(Y), \text{nl}.$

Predicatele **toate** și **toate1** sunt fără parametri, și ca atare sistemul va trebui să răspundă dacă există  $X$  și  $Y$  astfel încât aceste predicate să aibă loc.

$?- \text{toate}.$

$ab$

$cd$

$ef$

**false.**

$?- \text{toate1}.$

$ab$

**true;**

$cd$

**true;**

$ef$

**true.**

$?- \text{predicat}(X, Y).$

$X = a,$

$Y = b ;$

$X = c,$

$Y = d ;$

$X = e,$

$Y = f.$

Faptul că apelul predicatului **toate** se termină cu *fail* (care eșuează întotdeauna) obligă Prolog să înceapă backtracking prin corpul regulii **toate**. Prolog va reveni până la ultimul apel care poate oferi mai multe soluții. Predicatul **write** nu poate da alte soluții, deci revine la apelul lui **predicat**.

### Observații.

- Acel **false** de la sfârșitul soluțiilor semnifică faptul că predicatul ‘toate’ nu a fost satisfăcut.
- După *fail* nu are rost să puneti nici un predicat, deoarece Prolog nu va ajunge să-l execute niciodată.

**Notă.** Secvențele următoare sunt echivalente:

cât timp *condiție* execută  
*corp*

predicat :-  
*condiție*,  
*corp*,  
*fail*.

### EXEMPLU

Fie următorul cod Prolog

```
% q(E: integer)
% (o) – nedeterminist
q(1).
q(2).
q(3).
q(4).
p :- q(I), I<3, write (I), nl, fail.
```

Care este rezultatul următoarei întrebări **?p.**, în condițiile în care apare/nu apare fail la finalul ultimei clauze?

- cu **fail** la finalul clauzei, soluțiile sunt  
1  
2  
**false**
- fără **fail** la finalul clauzei, soluțiile sunt  
1  
true;  
2  
true;  
**false**

## 2. Obiecte simple și obiecte compuse.

### Obiecte simple

Un obiect simplu este fie o variabilă, fie o constantă. O constantă este fie un caracter, fie un număr, fie un atom (simbol sau string).

Variabilele Prolog sunt locale, nu globale. Adică, dacă două clauze conțin fiecare câte o variabilă numită X, cele două variabile sunt distincte și, de obicei, nu au efect una asupra celeilalte.

### Obiecte compuse și functori

**Obiectele compuse** ne permit să tratăm mai multe informații ca pe un singur element, într-un astfel de mod încât să-l putem utiliza și pe bucăți. Fie, de exemplu, data de *2 februarie 1998*. Constă din trei informații, ziua, luna și anul, dar e util să o tratăm ca un singur obiect cu o structură arborescentă:

```
DATA
 /  |  \
2 februarie 1998
```

Acest lucru se poate face scriind obiectul compus astfel:

```
data(2, "februarie", 1998)
```

Aceasta seamănă cu un fapt Prolog, dar nu este decât un obiect (o dată) pe care îl putem manevra la fel ca pe un simbol sau număr. Din punct de vedere sintactic, începe cu un nume (sau **functor**, în acest caz cuvântul **data**) urmat de trei argumente.

**Notă.** Functorul în Prolog nu este același lucru cu funcția din alte limbaje de programare. Este doar un nume care identifică un tip de date compuse și care ține argumentele laolaltă.

Argumentele unei date compuse pot fi chiar ele compuse. Iată un exemplu:

```
naștere(persoana("Ioan", "Popescu"), data(2, "februarie", 1918))
```

### Unificarea obiectelor compuse

Un obiect compus se poate unifica fie cu o variabilă simplă, fie cu un obiect compus care se potrivește cu el. De exemplu,

```
data(2, "februarie", 1998)
```

se potrivește cu variabila liberă X și are ca rezultat legarea lui X de data(...). De asemenea, obiectul compus de mai sus se potrivește și cu

```
data(Zi, Lu, An)
```

și are ca rezultat legarea variabilei Zi de valoarea 2, a variabilei Lu de valoarea “februarie” și a variabilei An de valoarea 1998.

### Observatii

- Convenim să folosim următoarea declarație pentru *specificarea* unui domeniu cu alternative  
    % domeniu = alternativa<sub>1</sub>(dom, dom, ..., dom);  
    %           alternativa<sub>2</sub>(dom, dom, ..., dom);  
    %           ...  
▪ Functorii pot fi folosiți pentru controla argumentele care pot avea tipuri multiple  
    % element = i(integer); r(real); s(string)

### Liste eterogene

În SWI-Prolog listele sunt eterogene, elementele componente pot fi de tipuri diferite. Pentru a se determina tipul unui element al listei, se folosesc predicatele predefinite în SWI (**number**, **is\_list**, etc)

Varianta 1. Se dă o listă eterogenă formată din numere, simboluri și/sau liste de numere. Se cere să se determine suma numerelor din lista eterogenă.

```
?- suma([1,a,[1,2,3],4],S).  
S = 11.
```

```
?- suma([a,b,[],],S).  
S=0.
```

```
%(L:list of numbers, S: number)  
% (i,o) - determ  
sumalist([],0).  
sumalist([H|T],S) :- sumalist(T,S1),  
                  S is S1+H.
```

```
%(L:list, S: number)  
% (i,o) - determ  
suma([],0).  
suma([H|T],S):-number(H),  
              !,  
              suma(T,S1),  
              S is H+S1.  
suma([H|T],S):-is_list(H),  
              !,  
              sumalist(H,S1),  
              suma(T,S2),  
              S is S1+S2.  
suma([_|T],S):-suma(T,S).
```

Varianta 2. O altă posibilitate (mai generală) de a gestiona liste eterogene este folosind obiecte compuse/functori.

Se dă o listă de numere întregi sau/și simboluri. Se cere suma numerelor întregi pare din listă.

?- suma([i(1),s(a),i(2)],S).

S = 2.

?- suma([s(a),s(b),i(1)],S).

S=0.

% suma(L: heterogeneous list, S: number)

% (i,o) - determ

suma([],0).

sumalist([i(H)|T], S) :-

H mod 2 =:=0, !,

suma(T,S1),

S is S1+H.

% atenție la o clauză de forma suma ([i(\_)|T], S) :- suma(T,S).

suma ([\_|T], S) :- suma(T,S).

### 3. Exemple

**EXEMPLU 3.1** Să se scrie un predicat care concatenează două liste.

? concatene([1, 2], [3, 4], L).

L = [1, 2, 3, 4].

Pentru combinarea listelor L1 si L2 pentru a forma lista L3 vom utiliza un algoritm recursiv de genul:

1. Dacă L1 = [] atunci L3 = L2.
2. Altfel, capul lui L3 este capul lui L1 și coada lui L3 se obține prin combinarea cozii lui L1 cu lista L2.

Să explicăm puțin acest algoritm. Fie listele L1 = [a<sub>1</sub>, ..., a<sub>n</sub>] și L2 = [b<sub>1</sub>, ..., b<sub>m</sub>]. Atunci lista L3 va trebui să fie L3 = [a<sub>1</sub>, ..., a<sub>n</sub>, b<sub>1</sub>, ..., b<sub>m</sub>] sau, dacă o separăm în cap și coadă, L3 = [a<sub>1</sub> | [a<sub>2</sub>, ..., a<sub>n</sub>, b<sub>1</sub>, ..., b<sub>m</sub>]]. De aici rezultă că:

1. capul listei L3 este capul listei L1;
2. coada listei L3 se obține prin concatenarea cozii listei L1 cu lista L2.

Mai departe, deoarece recursivitatea constă din reducerea complexității problemei prin scurtarea primei liste, rezultă că ieșirea din recursivitate va avea loc odată cu epuizarea listei L1, deci când L1 este []. De remarcat că condiția inversă, anume dacă L2 este [] atunci L3 este L1, este inutilă.

Programul SWI-Prolog este următorul:

## Model recursiv

$$\text{concatenare}(l_1 l_2 \dots l_n, l'_1 \dots l'_m) = \begin{cases} l'_1 & \text{daca } l = \emptyset \\ l_1 \oplus \text{concatenare}(l_2 \dots l_n, l'_1 \dots l'_m) & \text{altfel} \end{cases}$$

*% (concatenare(L1: list, L2: list, L3: list))*

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

*concatenare([], L, L).*

*concatenare([H|L1], L2, [H|L3]) :-*

*concatenare(L1, L2, L3).*

Se observă că predicatul **concatenare** descris mai sus funcționează cu mai multe modele de flux, fiind nedeterminist în unele modele de flux, determinist în altele.

De exemplu, pentru întrebările

? concatenare (L1, L2, [1, 2, 3]).

*/\* model de flux (o,o,i) - nedeterminist \*/*

L1=[] L2=[1, 2, 3]

L1=[1] L2=[2, 3]

? concatenare (L, [3, 4], [1, 2, 3, 4]).

*/\* model de flux (o,i,i) sau (i,o,i) - determinist\*/*

L=[1, 2]

**EXEMPLU 3.2** Să se scrie un predicat care determină lista submulțimilor unei liste reprezentate sub formă de mulțime.

? submulțimi([1, 2], L)

va produce L = [[], [2], [1], [1, 2]]

**Observație:** Dacă lista e vidă, submulțimea sa e lista vidă. Pentru determinarea submulțimilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

- i. determină o submulțime a listei L
- ii. plasează elementul E pe prima poziție într-o submulțime a listei L

$$\text{subm}(l_1, l_2, \dots, l_n) =$$

1.  $\emptyset$  *daca l e vida*

2.  $\text{subm}(l_2, \dots, l_n)$

3.  $l_1 \oplus \text{subm}(l_2, \dots, l_n)$

Vom folosi predicatul nedeterminist **subm** care va genera submulțimile, după care vor fi colectate toate soluțiile acestuia, folosind predicatul **findall**.

Codul SWI-Prolog este indicat mai jos

```
% subm(L: list, Subm:list)
% (i, o) - nedeterminist
subm([],[]).
subm([_|T], S) :- subm(T, S).
subm([H|T], [H|S]) :- subm(T, S).

% submultimi(L: list, LRez:list of lists)
% (i, o) - determinist
submultimi(L, LRez):-findall(S, subm(L,S), LRez).
```

### **EXEMPLU 3.3**

<pre>% sP(L:list of numbers, L: list of number) % (i,o) – nondeterm sP([],[]). sP ([_ T],S):-sP(T,S). sP ([H T],[H S]):- H mod 2 =:=0,                     !,                     sP(T,S). sP ([H T],[H S]):-sI(T,S).</pre>	<pre>% sI(L:list of numbers, L: list of number) % (i,o) – nondeterm  sI([H],[H]):-H mod 2 =\=0, !. sI([_ T],S):-sI(T,S). sI([H T],[H S]):-H mod 2 =:=0,                     !,                     sI(T,S). sI([H T],[H S]):-sP(T,S).</pre>
---	---

? sP([1, 2, 3], S).

[]  
[2]  
[1, 3]  
[1, 2, 3]

?sI([1, 2, 3], S).

[3]  
[2, 3]  
[1]  
[1, 2]  
**false**

**EXEMPLU 3.4** Fie următoarele definiții de predicate. Care este efectul următoarei interogări?

?- det([1,2,1,3,1,7,8],1,L).

```
% det( L:list of elements, E:element, LRez: list of numbers)
%(i, i, o) - determinist
det(L, E, LRez):- det_aux(L, E, LRez, 1).
```

```
% det_aux( L:list of elements, E:element, LRez: list of numbers, P:intreg)
% (i, i, o, i) - determinist
```



```

det_aux([], _, [], _).
det_aux([E|T], E, [P|LRez], P) :-
    !,
    P1 is P+1,
    det_aux(T, E, LRez, P1).
det_aux([_|T], E, LRez, P) :-
    P1 is P+1,
    det_aux(T, E, LRez, P1).

```

**Solutia pentru evitarea apelului repetat** (marcat cu albastru)

- folosirea unui predicat auxiliar

```

det_aux([], _, [], _).
det_aux([H|T], E, LRez, P) :-
    P1 is P+1,
    det_aux(T, E, L, P1),
    prel(H, E, P, L, LRez).

```

```

% prel(H: element, E:element, P: number, L:list, LRez: list of numbers)
% (i, i, i, i, o) - determinist
prel(E, E, P, L, [P|L]) :- !.
prel(_, _, _, L, L).

```

## CURS 5

### Nedeterminism. Arbori. Evitare apeluri recursive repetate.

#### Recursivitate de coadă

#### Cuprins

1. Evitare apeluri recursive repetate.....	1
2. Arbori.....	3
3. Optimizarea prin recursivitate de coadă (tail recursion).....	5
Funcționarea recursivității de coadă .....	5
Utilizarea tăieturii pentru păstrarea recursivității de coadă .....	6
4. Exemple predicate nedeterminate (continuare).....	7

### 1. Evitare apeluri recursive repetate

**EXEMPLU 1.1** Să se calculeze minimul unei liste de numere întregi.

```
% minim(L: list of integer, M:integer)
% (i, o) - determinist
minim([A], A).
minim([H|T], Rez) :-
    minim(T, M),
    H =< M, !, Rez=H.
minim([_|T], M) :-
    minim(T, M).
```

#### **Soluția 1**

```
minim([A], A).
minim([H|T], M) :-
    minim(T, M),
    H > M, !.
minim([H|_], H).
```

**Soluția 2.** Se folosește un predicat auxiliar, pentru a evita apelul (recursiv) repetat din clauzele 2 și 3.

```
minim([A], A).
minim([H|T], Rez) :-
    minim(T, M), aux(H, M, Rez).
```

```

% aux(H: integer, M:integer, Rez:integer)
% (i, i, o) - determinist
aux(H, M, Rez) :-
    H =< M, !, Rez=H.
aux(_, M, M).

```

**EXEMPLU 1.2** Se dă o listă numerică. Să se dea o soluție pentru evitarea apelului recursiv repetat. *Nu se vor redefini clauzele.*

```

% f(L:list of numbers, E: number)
% (i,o) – determinist
f([E],E).
f([H|T],Y):- f(T,X),
    H=<X,
    !,
    Y=H.
f([_|T],X):- f(T,X).

```

**Soluție.** Se folosește un predicat auxiliar. Soluție nu presupune înțelegerea semanticii.

```

f([E],E).
f([H|T],Y):- f(T,X), aux(H, X, Y).
% aux(H: integer, X:integer, Y:integer)
% (i, i, o) - determinist
aux(H, X, Y) :-
    H=<X,
    !,
    Y=H.
aux(_, X, X).

```

**EXEMPLU 1.3** Să se dea o soluție pentru evitarea apelului recursiv repetat.

```

% f(K:number, X:number)
% (i,o) – determinist
f(1,1):-!.
f(2,2):-!.
f(K,X):- K1 is K-1,
    f(K1, Y),
    Y>1,
    !,
    K2 is K-2,
    X=K2.
f(K,X):- K1 is K-1,
    f(K1, X).

```

**Solutie.** Se folosește un predicat auxiliar.

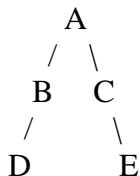
```
f(1,1):-!.  
f(2,2):-!.  
f(K,X) :- K1 is K-1,  
           f(K1, Y),  
           aux(K, Y, X).  
% aux(K: integer, Y:integer, X:integer)  
% (i, i, o) - determinist  
aux(K, Y, X) :-  
    Y>1,  
    !,  
    K2 is K-2,  
    X=K2.  
aux(_, Y, Y).
```

## 2. Arbori

Folosind obiecte compuse, se pot defini și prelucra în Prolog diverse structuri de date, precum arborii.

```
% domeniul corespunzător AB – domeniu cu alternative  
% arbore=arb(integer, arbore, arbore);nil  
% Functorul nil îl asociem arborelui vid
```

De exemplu, arborele



se va reprezenta astfel

```
arb(A, arb(B,  
            arb(D, nil, nil),  
            nil  
        ),  
    arb(C, nil,  
        arb(E, nil, nil)  
    )  
)
```

Se dă o listă numerică. Se cere să se afișeze elementele listei în ordine crescătoare. Se va folosi sortarea arborescentă (folosind un ABC).

**Indicație.** Se va construi un ABC cu elementele listei. Apoi, se va parcurge ABC în inordine.

% domeniul corespunzător ABC – domeniu cu alternative  
 % **arbore=arb(integer, arbore, arbore);nil**  
 % Functorul **nil** îl asociem arborelui vid

$$inserare(e, arb(r, s, d)) = \begin{cases} arb(e, \emptyset, \emptyset) & \text{daca } arb(r, s, d) \text{ e vid} \\ arb(r, inserare(e, s), d) & \text{daca } e \leq r \\ arb(r, s, inserare(e, d)) & \text{altfel} \end{cases}$$

% (integer, arbore, arbore) – (i,i,o) determinist

% insereaza un element într-un ABC

inserare(E, nil, arb(E, nil, nil)).

inserare(E, arb(R, S, D), arb(R, SNou, D)) :-

E =< R,

!,

inserare(E, S, SNou).

inserare(E, arb(R, S, D), arb(R, S, DNou)) :-

inserare(E, D, DNou).

% (arbore) – (i) determinist

% afișează nodurile arborelui în inordine

inordine(nil).

inordine(arb(R,S,D)) :-

inordine(S),

write(R),

nl,

inordine(D).

$$creeazaArb(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ inserare(l_1, creeazaArb(l_2 \dots l_n)) & \text{altfel} \end{cases}$$

% (arbore, list) – (i,o) determinist

% creează un ABC cu elementele unei liste

creeazaArb([], nil).

creeazaArb([H|T], Arb) :-

creeazaArb(T, Arb1),

inserare(H, Arb1, Arb).

% (list) – (i) determinist

% afișează elementele listei în ordine crescătoare (folosind sortare arborescentă)

sortare(L) :-

creeazaArb(L, Arb),

inordine(Arb).

### 3. Optimizarea prin recursivitate de coadă (tail recursion)

Recursivitatea are o mare problemă: consumă multă memorie. Dacă o procedură se repetă de 100 ori, 100 de stadii diferite ale execuției procedurii (cadre de stivă) sunt memorate.

Totuși, există un caz special când o procedură se apelează pe ea fără să genereze cadru de stivă. Dacă procedura apelatoare apelează o procedură ca ultim pas al sau (după acest apel urmează punctul). Când procedura apelată se termină, procedura apelatoare nu mai are altceva de făcut. Aceasta înseamnă că procedura apelatoare nu are sens să-și memoreze stadiul execuției, deoarece nu mai are nevoie de acesta.

#### Funcționarea recursivității de coadă

Iată două reguli depre cum să faceți o recursivitate de coadă:

1. Apelul recursiv este ultimul subgoal din clauza respectivă.
2. Nu există puncte de backtracking mai sus în acea clauză (adică, subgoal-urile de mai sus sunt deterministe).

Iată un exemplu:

```
tip(N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    tip(Nou).
```

Această procedură folosește recursivitatea de coadă. Nu consumă memorie, și nu se oprește niciodată. Eventual, din cauza rotunjirilor, de la un moment va da rezultate incorecte, dar nu se va opri.

#### Exemple greșite de recursivitate de coadă

Iată cateva reguli despre cum să NU faceți o recursivitate de coadă:

1. Dacă apelul recursiv nu este ultimul pas, procedura nu folosește recursivitatea de coadă.

Exemplu:

```
tip (N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    tip (Nou),  
    nl.
```

2. Un alt mod de a pierde recursivitatea de coadă este de a lăsa o alternativă neîncercată la momentul apelului recursiv.

Exemplu:

```
tip(N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    tip(Nou).  
tip(N) :-  
    N < 0,  
    write('N este negativ.').
```

Aici, prima clauză se apelează înainte ca a doua să fie încercată. După un anumit număr de pași intră în criză de memorie.

3. Alternativa neîncercată nu trebuie neaparat să fie o clauza separată a procedurii recursive. Poate să fie o alternativă a unei clauze apelate din interiorul procedurii recursive.

Exemplu:

```
tip (N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    verif(Nou),  
    tip(Nou).  
verif(Z) :- Z >= 0.  
verif(Z) :- Z < 0.
```

Dacă N este pozitiv, prima clauză a predicatului **verif** a reușit, dar a doua nu a fost încercată. Deci, **tip** trebuie să-și pastreze o copie a cadrului de stivă.

### Utilizarea tăieturii pentru păstrarea recursivității de coadă

A doua și a treia situație de mai sus pot fi înlăturate dacă se utilizează tăietura, chiar dacă există alternative neîncercate.

Exemplu la situația a doua:

```
tip (N) :-  
    N >= 0,  
    !,  
    write(N),  
    nl,  
    Nou = N + 1,  
    tip(Nou).  
tip(N) :-  
    N < 0,  
    write("N este negativ.").
```

Exemplu la situația a treia:

```

tip(N) :-
    write(N),
    nl,
    Nou = N + 1,
    verif(Nou),
    !,
    tip(Nou).
verif(Z) :- Z >= 0.
verif(Z) :- Z < 0.

```

## 4. Exemple predicate nedeterminate (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:

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

Modelul recursiv pentru generare este:

$$\begin{aligned}
 \text{comb}(l_1 \ l_2 \ \dots \ l_n, k) = & \\
 1. \quad & (l_1) \qquad \qquad \qquad \text{daca } k = 1 \\
 2. \quad & \text{comb}(l_2 \ \dots \ l_n, k) \\
 3. \quad & l_1 \oplus \text{comb}(l_2 \ \dots \ l_n, k - 1) \quad \text{daca } k > 1
 \end{aligned}$$

Vom folosi predicatul nedeterminist **comb** care va genera toate combinațiile. Dacă se dorește colectarea combinațiilor 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 preducatul 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)

**TEMA** 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];

# CURS 6

## Backtracking.

### Cuprins

1. Backtracking .....	1
-----------------------	---

% depanare cod Prolog

% trace.

% notrace.

## 1. 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(K, K, S, LC) :-
    findall(C, combSuma(L, K, S, C), LC).

```

**VARIANTA 2** Se generează combinațiile cu  $k$  elemente și apoi se verifică dacă suma unei combinații este  $S$ . Această soluție este ineficientă, având în vedere că se generează (în plus) combinații care e posibil să nu fie de sumă  $S$ .

Se vor folosi următoarele predicate:

- predicatul **comb** pentru generarea unei combinații cu  $k$  elemente dintr-o listă, predicat descris în **EXEMPLU 3.1**, Cursul 5.
- predicatul **suma** (L:list of numbers, S:integer), model de flux (i, i) care verifică dacă suma elementelor listei L este egală cu 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).

```

**VARIANTA 3** 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* ( $l_1 l_2, \dots, l_n$ ) =

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) :-
    candidat(E, L),
    E <= S,
    combaux(L, K, S, C, 1, E, [E]).
```

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

*combaux*( $l, k, s, lg, sum, col$ ) =

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

2.
combaux(l,k,s,lg + 1,sum + e,e U col)   daca (sum ≠ s sau k ≠ lg) și (lg < k) și
                                           (e = candidat( $l_1, l_2, \dots, l_n$ )) și (e < col1) și (sum + e ≤ 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 is Sum+E,
    Sum1 <= S,
    Lg1 is Lg+1,
    combaux(L, K, S, C, Lg1, Sum1, [E|[H|T]]).
```

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

```
% 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 is Sum + E,
Sum1 =< S.
combaux(L, K, S, C, Lg, Sum, [H|T]):-
    conditie(L, K, S, Lg, Sum, H, E),
    Lg1 is Lg+1,
    Sum1 is Sum + E,
    combaux(L, K, S, C, Lg1, Sum1, [E|[H|T]]).

```

**EXEMPLU 1.2** 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 1.3** 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) dacă  $n > 1$*

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, [[H|T]]).
```

**EXEMPLU 1.4** 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 colectează soluția, al cincilea argument  
 % reprezintă 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 1.5** 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).

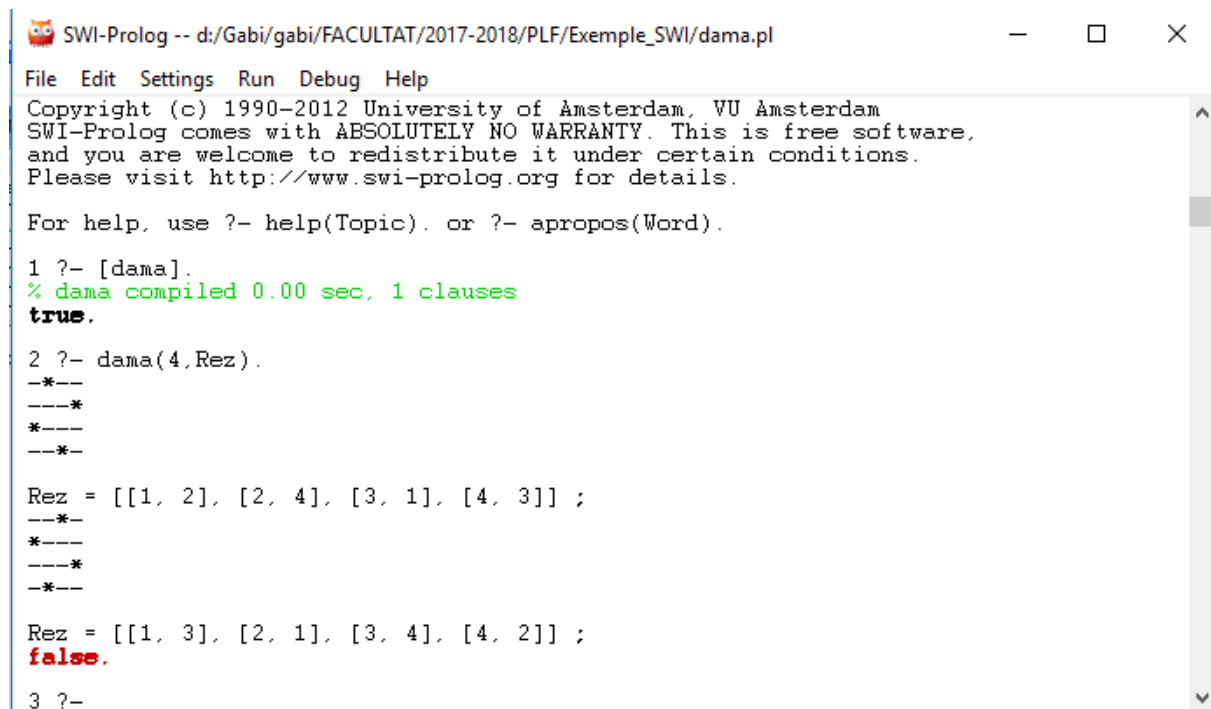
### PROBLEMA CELOR 5 CASE.

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 1.6** 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).
```

### TEMĂ

1. Se dă o listă cu elemente numere întregi distincte. Să se genereze toate submulțimile cu elemente în ordine strict crescătoare.
2. Se dă o listă cu elemente numere întregi distincte. Să se genereze toate submulțimile cu  $k$  elemente în progresie aritmetică.