

CURS 3

Predicate deterministe și nedeterministe. Exemple

Cuprins

Bibliografie	1
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) :- findall(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 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 $\theta(n)$.

- Alte modele de flux?

% (i, i, i) - determinist ? adaug(1, [2, 3], [2, 3, 1]). true.	% (o, i, i) - determinist ? adaug(E, [2, 3], [2, 3, 1]). E=1.	% (i, o, i) - determinist ? adaug(1, L, [2, 3, 1]). L=[2, 3]; false.	% (o, o, i) - determinist ? adaug(E, L, [2, 3, 1]). E=1 L=[2, 3]; false.
---	---	--	---

EXEMPLU 2.2 Verificați apartenența unui element într-o listă.

% (i, i) ? member(3, [1, 2, 3]).

Pentru a descrie apartenența la o listă vom contrui 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.

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

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]).
```

```
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/member.pl compiled 0.00 sec, 7 clauses
1 ?- member1(1,[1,2,1,3,1,4]).
true ;
true ;
true ;
false.

2 ?- member1(X,[1,2,1,3,1,4]).
X = 1 ;
X = 2 ;
X = 1 ;
X = 3 ;
X = 1 ;
X = 4 ;
false.

3 ?- member1(1,[2,1,3]).
true ;
false.

4 ?- member1(5,[2,1,3]).
false.

5 ?- go1.
true ;
true ;
true ;
false.

6 ?- member2(1,[1,2,1,3,1,4]).
true.

7 ?- member2(X,[1,2,1,3,1,4]).
X = 1.

8 ?- member2(1,[2,1,3]).
true.

9 ?- member2(5,[2,1,3]).
false.

10 ?- go2.
true.
```

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

$i \leftarrow 1$
 $P \leftarrow 1$
CâtTimp $i < n$ execută
 $i \leftarrow i + 1$
 $P \leftarrow P * i$
SfCâtTimp
 $F \leftarrow P$

$fact(n) = fact_aux(n, 1, 1)$

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

- 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 2.4 Să se determine inversa unei liste.

Varianta A (direct recursiv)

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

```
% 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]

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

$$invers(l_1 l_2 \dots l_n) = invers_aux(l_1 l_2 \dots l_n, \emptyset)$$

% 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, ..., *l*_{*n*}) =

1. (*e*, *l*₁) *e* < *l*₁
2. *pereche*(*e*, *l*₂, ..., *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, ..., *l*_{*n*}) =

1. *pereche*(*l*₁, *l*₂, ..., *l*_{*n*})
2. *perechi*(*l*₂, ..., *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*).

EXEMPLU 2.7 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.

Observatie. 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)

```
%(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).
```