

## 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 XPCe – 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**

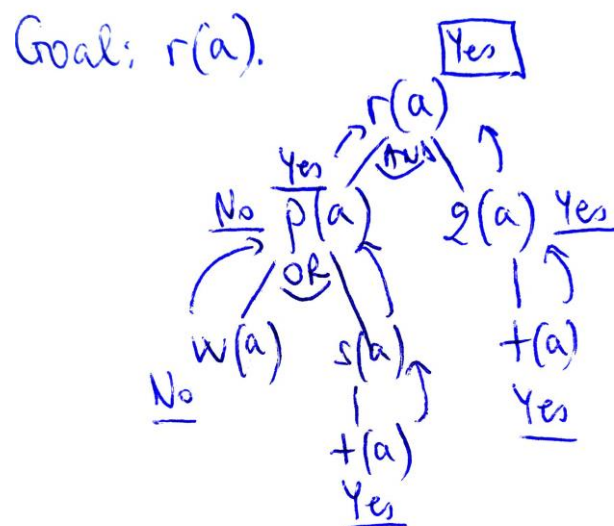
$$? r(a).$$

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 **Curs 4**).

- 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 1.3**
  - % 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). <b>true.</b>	?- parinte(a,X). X=d; X=b.
? - parinte(a,f). <b>false.</b>	
?- frate(a,X). X=c; X=e.	?- frate(a,_). <b>true.</b>

## 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) – determină  $F = N!$ ;
- (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.**

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

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

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

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

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

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

?-  $\backslash+ a=a$ .  
**false.**

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

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

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

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

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

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

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

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

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

?-  $R \backslash== 1$ .  
**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

=, \=, ==, \==

A se vedea Secțiunea 1.5.

?- 2+4=6. <b>false.</b>	?- 2+4\=6. <b>true.</b>	?- 6==6. <b>true.</b>	?- 6\=7. <b>true.</b>	?- 6==2+4. <b>false.</b>
----------------------------	----------------------------	--------------------------	--------------------------	-----------------------------

?- 2+4=2+4. <b>true.</b>	?- 2+4=4+2. <b>false.</b>	?- X= 2+4-1. 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. <b>true.</b>	?- 2+4\=\=7. <b>true.</b>	?- 6\:=6. <b>true.</b>
-----------------------------	------------------------------	---------------------------

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=5	?- X is 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. <b>true.</b>	?- 2+4=\=7. <b>true.</b>	?- 6\:=6. <b>true.</b>
------------------------------	-----------------------------	---------------------------

## Câteva funcții aritmetice predefinite SWI-Prolog

X mod Y                      întoarce restul împărțirii lui X la Y

X div Y	întoarce câtul împărțirii lui X la 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.
```