LAB 10

Monday, May 9, 2022 12:58 PM

https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/

① FACTS

= relații implicite între obiecte / proprietăți ale obiectelor
• sunt necondiționat / mereu adevărate

ex: fruct(măr).
băiat(andrei).
leneș(mircea).
mănâncă(Alexandra, paste).
are_culoarea(păr, negru).

Sintaxa:

relation(obiect1, obiect2, ...).

relation = numele relației; obiecte = obiecte

② RULES

= relații explicite între obiecte
• sunt condiționat adevărate

ex: Alexandra este fericită dacă dansează.

fericită(alexandra) :- dansează(alexandra)
prieteni(alexandra, mihai) :- îi_place_mâncarea(alexandra), îi_place_mâncarea(mihai)
= conjuncție

Sintaxă:

S :- S1, S2, ... Sm

S = head; :- = neck; S1, S2, ... Sm = body; , = conjunction

=> când body este adevărat => head este adevărat

S1, S2 .. Sm sunt (A) => S este (A)

sau S este (A) dacă S1, S2 .. Sm sunt (A)

③ QUERIES

2) întrebări despre fapte sau relații

ex: Este mărul un fruct?
Alexandra mănâncă paste?
Sunt Alexandra și cu Mihai prieteni?

?- fruct(măr).
?- mănâncă(alexandra, paste).
?- prieteni(alexandra, mihai)

KNOWLEDGE BASE

= colecția de facts și rules
• atunci când facem un query, Prolog caută în KB să vadă dacă are informația pe care o dorim

④ VARIABLES

Până acum am văzut că obiectele pe care le-am folosit în fapte/reguli/queries se scriu cu literă mică.

Variabile -> literă mare / _

ex:

```
% X and Y are frients if X likes Y and Y likes X
friends(X,Y) :- likes(X,Y), likes(Y,X).
```

VARIABILE ANONIME : _

ex: is_parent(X):- father_of(X,_).
is_parent(X):- mother_of(X,_).

• nu ne interesează mama cui este X ca să putem spune că X este părinte

⑤ OPERATORI

Operator	Meaning
X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the X and Y values are equal
X =\= Y	the X and Y values are not equal

⑥ LISTE

listă vidă: []

listă cu elem. a,b,c: [a,b,c]

listă nevidă: [Head|Tail]

a ↙ ↓ [b,c]

OPERAȚII COMUNE PE LISTE

(vezi codul din demo)

① is_member(X, L) ...

②

```
% lungimea listei vide este 0
length_list([], 0).
% lungimea listei nevide este N daca lungimea lui Tail este N1 si N = N1 + 1
length_list([_|Tail], N) :- length_list(Tail, N1), N is N1 + 1.
```

R1 R2 G1 G2

is → acesta este un predicat!

Number is Expression

⇒ verifică dacă Number = valoarea la care se evaluează Expr

❗ Acesta nu reprezintă legarea unei variabile !! (ca în C, Java, ...)

ex: "N is N+1" nu va da fail, pentru că nu există niciun număr cu care să îl înlocuim pe N a.î. el să fie egal cu N+1 (N+1 nu se va evalua niciodată la N)

```
[trace] 4 ?- length_of([], X).
   Call: (10) length_of([], _4200) ? creep
   Exit: (10) length_of([], 0) ? creep
X = 0.
```

→ X se înlocuiește cu o var. unică _4200

⇒ pentru ca acest predicat să fie adev, _4200 trb. să fie 0.

trebe. să fie 0.

```
[trace] 26 ?- length_of([1, 2, 3], X).
   Call: (10) length_of([1, 2, 3], _1568) ? creep
   Call: (11) length_of([2, 3], _2798) ? creep
   Call: (12) length_of([3], _3554) ? creep
   Call: (13) length_of([], _4310) ? creep
   Exit: (13) length_of([], 0) ? creep
   Call: (13) _3554 is 0+1 ? creep
   Exit: (13) 1 is 0+1 ? creep
   Exit: (12) length_of([3], 1) ? creep
   Call: (12) _2798 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Exit: (11) length_of([2, 3], 2) ? creep
   Call: (11) _1568 is 2+1 ? creep
   Exit: (11) 3 is 2+1 ? creep
   Exit: (10) length_of([1, 2, 3], 3) ? creep
X = 3.
```

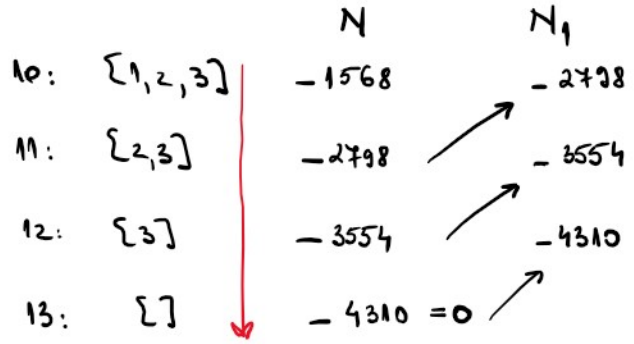
avans în recursivitate până la R_1

R_2, G

$\Rightarrow R_1$ succes

④

```
[trace] 6 ?- X is 2+1.
X = 3.
```



length([], -4310). => -4310 = 0

length([3], -3554) :- length([], -4310), -3554 is -4310 + 1.

(N) (N1) (N) (N1)

③ concat / append

[1,2,3] [4,5] => [3|[4,5]] -> [2|[3,4,5]] -> [1|[2,3,4,5]]

```
[trace] 9 ?- list_concat([1, 2], [3, 4], L3).
   Call: (10) list_concat([1, 2], [3, 4], _248) ? creep
   Call: (11) list_concat([2], [3, 4], _1508) ? creep
   Call: (12) list_concat([], [3, 4], _2272) ? creep
   Exit: (12) list_concat([], [3, 4], [3, 4]) ? creep
   Exit: (11) list_concat([2], [3, 4], [2, 3, 4]) ? creep
   Exit: (10) list_concat([1, 2], [3, 4], [1, 2, 3, 4]) ? creep
L3 = [1, 2, 3, 4].
```

```
% list_concat(L1, L2, L3) - concatenarea listelor L1 si L2 este L3
% Rezultatul concatenarii unei liste vide cu o lista L este L
list_concat([],L,L).
% Concatenam recursiv tail-ul lui L1 la L2 si salvam rezultatul in L3 (ca in Haskell)
list_concat([X1|T1],L2,[X1|T3]) :- list_concat(T1,L2,T3).
```

① Cum merg în recursivitate?

=> ca în Haskell => pe fail => p([H|T]) :- p(T)

② list_concat([X1|T1]....) :- list_concat(T1)

② Cum construiesc rezultatul?

L_1 : [1,2] -> [2] -> []

L_3 : [1,2,3,4] <- [2,3,4] <- [3,4] = Res

concat([X1|T1], L2, ...?) :- concat(T1, L2, Res)

(punem rezultatul în Res)

=> aici pun rezultatul

↓

[X1 | Res]

---

(4)

L_1 : [1,2,3] -> [2,3] -> [3] -> []

Res : [1,3] <- [3] <- [3] <- []

remove(X, [], []).

remove(X, [X1|T1], ...?) :- remove(X, T1, T2).

⇓

if X == X1 then T2

else [X1|T2]

=> remove(X, [X|T1], T2) :- remove(X, T1, T2). <=> remove(X, [X|T1], T1).

remove(X, [X1|T1], [X1|T2]) :- remove(X, T1, T2).

---

(5) reverse

L_1 : [1,2,3] -> [2,3] -> [3] -> []

Res [3,2,1] <- [3,2] <- [3] <- []

op : [3,2]++[1] [3]++[2] []++[3]

=> Res ++ [head] (=) list_concat(Res, [X1], R)

"X1

(R = rezultatul concatenării)

reverse([], []).

reverse([X1|T1], ...) :- reverse(T1, Res)

⇓

list_concat(Res, [Head], R)

=> reverse ([X1|T1], R) :- reverse (T1, Res), list_concat (Res, [X1], R).

INDICAȚIE :

① întâi gândiți-vă cum se face apelul în recursivitate (ex: [X1|T1] -> T1)

② scrieți regula fără să completați rezultatul în stânga (ex: r([X1|T1], ...) :- ...)

③ dați un nume rezultatului din dreapta (ex: r([X1|T1], ...) :- r(T1, Res))

④ modelați cum se construiește rezultatul din stânga (ex: r([X1|T1], [X1|Res]) :- r(T1, Res))

FACTORIAL

```
fact 0 = 1
fact n = n * fact(n-1)
```

=> fact (0,1). -> rezultatul
fact (N, NFact) :-