Curs 6

2019-2020

Programare Logică

Cuprins

Bibliografie:
L.S. Sterling and E.Y. Shapiro, The Art of Prolog
`https://mitpress.mit.edu/books/art-prolog-second-edition`

Liste și recursie

Listă [t1,...,tn]

- O listă în Prolog este un șir de elemente, separate prin virgulă, între paranteze drepte:

```
[1,cold, parent(jon),[winter,is,coming],X]
```

- O listă poate conține termeni de orice fel.
- Ordinea termenilor din listă are importanță:

```
?- [1,2] == [2,1] .
false
```

- Lista vidă se notează [].
- Simbolul | desemnează coada listei:

```
?- [1,2,3,4,5,6] = [X|T].
X = 1, T = [2, 3, 4, 5, 6].

?- [1,2,3|[4,5,6]] == [1,2,3,4,5,6].
true.
```

Liste

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

```
is_list([]).
is_list([_|_]).
```

- Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

```
head([X|_],X).


last([X],X).
last([_|T],Y):- last(T,Y).


tail([],[]).
tail([_|T],T).
```

Liste

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
member(H, [_|T]) :- member(H,T).
```

- Definiți un predicat append/3 care verifică dacă o listă se obține prin concatenarea altor două liste.

```
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).
```

Există predicatele predefinite `member/2` și `append/3`.

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([],[]). perm([X|T],L) :- elim(X,L,R), perm(R,T).
```

Predicatele predefinite `select/3` și `permutation/2` au aceeași funcționalitate.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).  word(early).  word(layer).`

Predicat util:

```
?- name(relay,L). % conversie între atomi şi liste
L = [114, 101, 108, 97, 121]
```

Două abordări posibile:

- se generează o posibilă, soluție apoi se testează dacă este în KB.
- se parcurge KB și pentru fiecare termen se testează dacă e soluție.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).  word(early).  word(layer).`

```
anagram1(A,B) :- name(A,L), permutation(L,W),
                 name(B,W), word(B).
anagram2(A,B) :- name(A,L), word(B),
                 name(B,W), permutation(L,W).
```

```
?- anagram1(layre,X).
X = layer ;
X = relay ;
X = early ;
false.
```

```
?- anagram2(layre,X).
X = relay ;
X = early ;
X = layer ;
false.
```

Recursie

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

```
rev([],[]).
rev([X|T],L) :- rev(T,R),append(R,[X],L).
```

Soluția de mai sus este corectă, dar foarte costisitoare computațional, datorită stilului de programare declarativ.

Cum putem defini o variantă mai rapidă?

O metodă care prin care recursia devine mai rapidă este folosirea acumulatorilor, în care se păstrează rezultatele parțiale.

Recursie cu acumulatori

- Varianta inițială:

```
rev([],[]).
rev([X|T],L) :- rev(T,R),append(R,[X],L).
```

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul iniţial nu am acumulat nimic.

revac([], R, R).
% cand lista iniţială a fost consumată,
% am acumulat rezultatul final.

revac([X|T], Acc, R) :- revac(T,[X|Acc],R).
% Acc conţine inversa listei care a fost deja parcursă.
```

- Complexitatea a fost redusă de la $O(n^2)$ la $O(n)$, unde n este lungimea listei.

Recursie

- Multe implementări ale limbajului Prolog aplică "*last call optimization*" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (*tail recursion*).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (*tail recursion*).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul time/1.

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.

biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

- Predicatul cu recursie la coadă:

```
biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).

?- time(biglist_tr(50000,X)).
100,000 inferences, 0.000 CPU in 0.007 seconds
(0% CPU, Infinite Lips)
X = [50000, 49999, 49998|...]
```

Liste append/3

- Reamintim definiția funcției append/3:

```
?- listing(append/3).
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).

?- append(X,Y,[a,b,c]).
X = [],
Y = [a, b, c] ;
X = [a],
Y = [b, c] ;
X = [a, b],
Y = [c] ;
X = [a, b, c],
Y = [] ;
false
```

- Funcția astfel definită poate fi folosită atât pentru verificare, cât și pentru generare.

Liste

```
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).
```

Exercițiu

Definiți `prefix/2` și `suffix/2` folosind append.

```
prefix(P,L) :- append(P,_, L).
suffix(S,L) :- append(_,S,L).
```

Observăm că funcția append parcurge prima listă.

Am putea rescrie această funcție astfel încât legătura să se facă direct, așa cum putem face în programarea imperativă?

Problema poate fi rezolvată scriind listele ca diferențe, o tehnică utilă în limbajul Prolog.

Liste ca diferențe

- Ideea: lista `[t1,...,tn]` va fi reprezentată printr-o pereche

  `([t1,...,tn|T], T)`

  Această pereche poate fi notată `[t1,...,tn|T]- T`, dar notația nu este importantă.
- Vrem să definim append/3 pentru liste ca diferențe:

  `dlappend((X1,T1),(X2,T2),(R,T)) :- ?.`

```
?- dlappend(([1,2,3|P],P),([4,5|T],T),RD).
P = [4, 5|T],
RD = ([1, 2, 3, 4, 5|T], T).
```

Liste ca diferențe ([t1,...,tn|T], T)

```
dlappend((X1,T1),(X2,T2),(R,T)) :- ?.
```

- Dacă [t1,..., tn] este diferența (X1,T1), iar [q1,..., qk] este diferența (X2,T2) observăm că diferența (R,T) trebuie să fie [t1,...,tn,q1..., qk].
- Obținem R=[t1,...,tn,q1..., qk|T], deci (X1,T1) = (R, P) și (X2,T2) = (P,T) unde P =[q1,...,qk|T]).
- Definiția este:

```
dlappend((R,P),(P,T),(R,T)).
```

```
?- dlappend(([1,2,3|P],P),([4,5|T],T),RD).
P = [4, 5|T],
RD = ([1, 2, 3, 4, 5|T], T).
```

- dlappend este foarte rapid, dar nu poate fi folosit pentru generare, ci numai pentru verificare.

Tipuri de date compuse

Termeni compuși `f(t1,..., tn)`

- Termenii sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
  - Constante: `23, sansa, 'Jon Snow'`
  - Variabile: `X, Stark, _house`
  - Termeni compuși:
    - predicate
    - termeni prin care reprezentăm datele

Exemplu

- `born(john, date(20,3,1977))`
  - `born/2` și `date/3` sunt *functori*
  - `born/2` este un predicat
  - `date/3` definește date compuse

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
  - `void` este arbore
  - `tree(X,A1,A2)` este arbore, unde `X` este un element, iar `A1` și `A2` sunt arbori

`tree(X,A1,A2)` este un termen compus, dar nu este un predicat!

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

```
def(arb, tree(a, tree(b,
                      tree(d,void,void),
                      void),
                 tree(c, void,
                      tree(e,void,void)))).
```

Deoarece în Prolog nu avem declarații explicite de date, pentru a defini arborii vom scrie un predicat care este adevărat atunci când argumentul său este un arbore.

Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right),
                                         element_binary_tree(Element)

element_binary_tree(X):- integer(X). /* de exemplu */


test:- def(arb,T), binary_tree(T).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

```
tree_member(X,tree(X,Left,Right)).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că doi arbori binari sunt izomorfi (fiecare nod are aceași copii, dar ordinea nu contează).

```
isotree(void,void).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
            isotree(Left1,Left2), isotree(Right1,Right2).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
             isotree(Left1,Right2), isotree(Right1,Left2).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

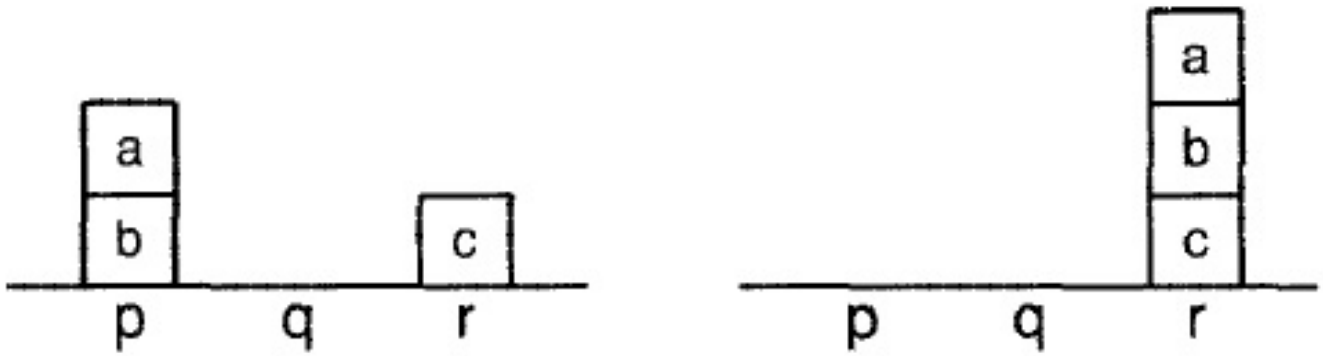
```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).

test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).

?- test(T,P).
T = tree(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
P = [a, b, d, c, e]
```

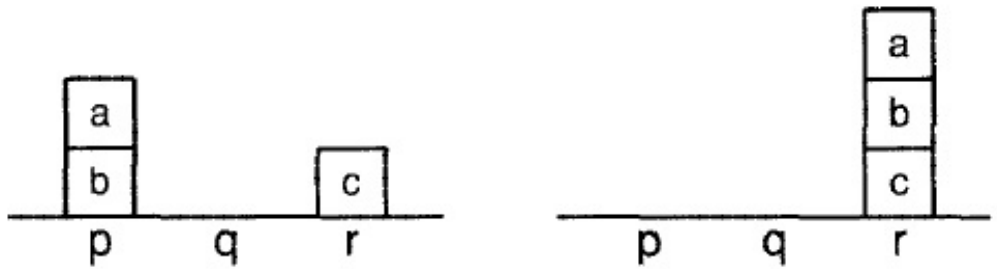
Planning în Prolog

Problemă: Lumea blocurilor



- Lumea blocurilor este formată din:
  - trei blocuri: `a`,`b`, `c`
  - trei poziții: `p`,`q`, `r`
  - un bloc poate sta peste un alt bloc sau pe o poziție
- Un bloc poate fi mutat pe o poziție liberă sau pe un alt bloc.
- Problema este de a găsi un șir de mutări astfel încât dintr-o stare inițială să se ajungă într-o stare finală

Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:

```
block(a). block(b). block(c).
place(p). place(q). place(r).
```

```
initial_state([on(a,b), on(b,p),on(c,r)]).
final_state([on(a,b),on(b,c),on(c,r)]).
```

Observați că `on(a,b)`, `on(b,c)`, etc. sunt date compuse.

- O stare este o listă de termenii de tipul `on(X,Y)`.
Într-o listă care reprezintă o stare, termenii `on(X,Y)` sunt ordonați după prima componentă.

Lumea blocurilor

- Predicatul `transform(State1,State2,Plan)` va **genera** în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

```
transform(State1,State2,Plan) :-
                    transform(State1,State2,[State1],Plan).
```

Predicatele `transform/3` și `transform/4` sunt diferite. În modelarea noastră, `transform/4` este un predicat auxiliar, cu ajutorul căruia reținem stările "vizitate".

```
transform(State,State,Visited,[],_).

transform(State1,State2,Visited,[Action|Actions]) :-
                                legal_action(Action,State1),
                                update(Action,State1,State),
                                   \+ member(State,Visited),
            transform(State,State2,[State|Visited],Actions).
```

Lumea blocurilor

- Predicatul `transform(State1,State2,Plan)` va **genera** în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

```
transform(State1,State2,Plan) :-
                    transform(State1,State2,[State1],Plan).

transform(State,State,Visited,[],_).

transform(State1,State2,Visited,[Action|Actions]) :-
                                  legal_action(Action,State1),
                                  update(Action,State1,State),
                                     \+ member(State,Visited),
          transform(State,State2,[State|Visited],Actions).
```

- Căutare de tip depth-first.

Lumea blocurilor

- Predicatul `legal_action(Action,State)` va instanția `Action` cu o mutare care poate fi efectuată în starea `State`. Există două mutări posibile: *mutarea pe un bloc* și *mutarea pe o poziție*.

```
legal_action(to_block(Block1,Block2),State) :-
                        block(Block1), clear(Block1,State),
                          block(Block2), Block1 \== Block2,
                                        clear(Block2,State).


clear(X,State) :- \+ member(on(_,X),State).


legal_action(to_place(Block,Place),State) :-
                          block(Block), clear(Block,State),
                           place(Place), clear(Place,State).
```

Lumea blocurilor

- Predicatul `update(Action,State,State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

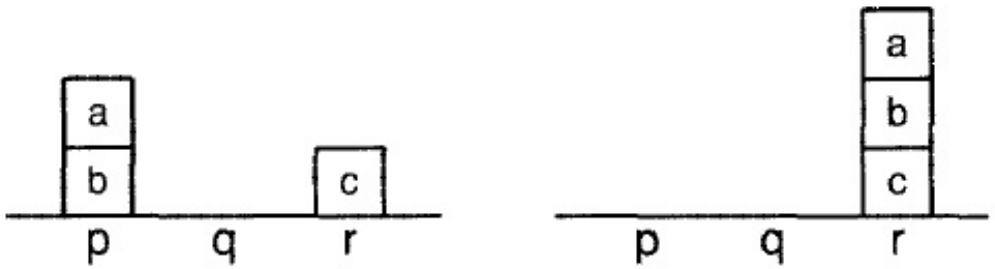
```
update(to_block(X,Z),State,State1) :-
                substitute(on(X,_),on(X,Z),State,State1).

update(to_place(X,Z),State,State1) :-
                substitute(on(X,_),on(X,Z),State,State1).
```

- `substitute(X,Y,L,R)` substituie `X` cu `Y` în lista `L`, rezultatul fiind `R`.

```
substitute(X,Y,[X|Xs],[Y|Xs]).
substitute(X,Y,[X1|Xs],[X1|Ys]) :- X \== X1,
                                   substitute(X,Y,Xs,Ys).
```

Lumea blocurilor



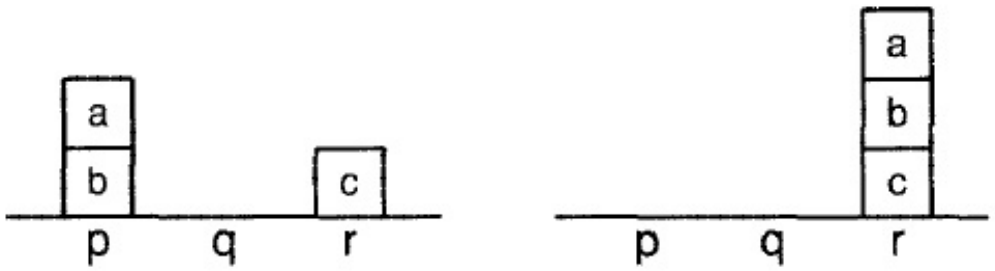

```
block(a). block(b). block(c).
place(p). place(q). place(r).


initial_state([on(a,b), on(b,p),on(c,r)]).
final_state([on(a,b),on(b,c),on(c,r)]).

test_plan(Plan) :- initial_state(I), final_state(F),
                                        transform(I,F,Plan).


?- test(Plan).
```

Lumea blocurilor




```
test_plan(Plan) :- initial_state(I), final_state(F),
                                        transform(I,F,Plan).
```

```
?- test(Plan).
Plan = [to_place(a, b, q), to_block(a, q, c),
to_place(b, p, q), to_place(a, c, p), to_block(a, p, b),
to_place(c, r, p), to_place(a, b, r), to_block(a, r, c),
to_place(b, q, r), to_place(a, c, q), to_block(a, q, b),
to_place(c, p, q), to_place(a, b, p), to_block(b, r, a),
to_place(c, q, r), to_block(b, a, c), to_place(a, p, q),
to_block(a, q, b)]
```

Lumea blocurilor

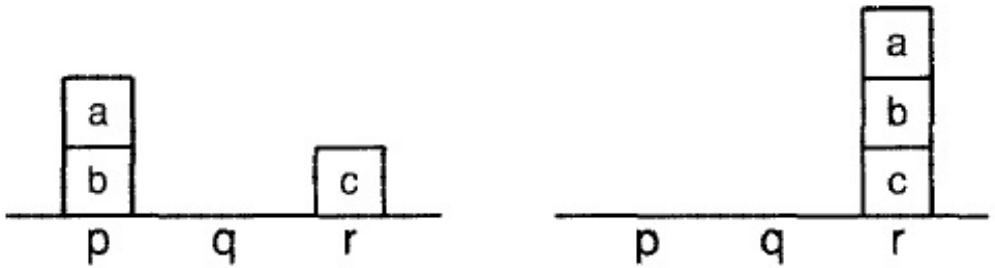
Pentru a obține o soluție mai simplă, putem limita numărul de mutări!

```
transform(State1,State2,Plan, N) :-
                    transform(State1,State2,[State1],Plan,N).


transform(State,State,Visited,[],_).

transform(State1,State2,Visited,[Action|Actions],N) :-
                                  legal_action(Action,State1),
                                  update(Action,State1,State),
        \+ member(State,Visited), length(Visited,M), M < N,
          transform(State,State2,[State|Visited],Actions,N).
```

Lumea blocurilor




```
test_plan(Plan,N) :- initial_state(I), final_state(F),
                                        transform(I,F,Plan,N).
```

```
?- test(Plan,3).
false
?- test(Plan,4).
Plan = [to_place(a, b, q), to_block(b, p, c), to_block(a, q, b)]
```

În general

- Predicatul `transform(State1,State2,Plan)` generează, printr-o căutare de tip depth-first, în variabila `Plan` un șir de mutări permise care transformă starea `State1` în starea `State2`.

```
transform(State1,State2,Plan) :-
                    transform(State1,State2,[State1],Plan).

transform(State,State,Visited,[],_).

transform(State1,State2,Visited,[Action|Actions]) :-
                                legal_action(Action,State1),
                                update(Action,State1,State),
                                   \+ member(State,Visited),
            transform(State,State2,[State|Visited],Actions).
```

- Reprezentarea stărilor, a acțiunilor, a soluției depinde de problema concretă pe care o rezolvăm.

Pe săptămâna viitoare!