Prolog CheatSheet

Laborator 11

Liste și funcții utile pe liste

```
[]                  - lista vida
[a,b,c]             - elementele a, b, c
[Prim|Rest]         - element concatenat la rest
[X1,X2,...,XN|Rest] - n elemente concatenate la restul listei

append(?List1, ?List2, ?List1AndList2) % List1AndList2 este
    concatenarea între List1 și List2
?- append([1,2,3], [4,5], X).
X = [1, 2, 3, 4, 5].
?- append([1,2,3], X, [1,2,3,4]).
X = [4].
?- append([1,2,3], X, [2,3,4]).
false.

member(?Elem, ?List) % Adevărat dacă Elem esre prezent în List
?- member(1, [1,2,3]).
true ;
false.
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

length(?List, ?Int) % Adevărat dacă Int reprezintă numărul de
    elemente din List
?- length([1,2,3], X).
X = 3.

reverse(?List1, ?List2) % Adevărat atunci când elementele din
    List2 sunt în ordine inversă față de List1
?- reverse([1, 2, 3], X).
X = [3, 2, 1].

sort(+List, -Sorted) % Adevărat atunci când Sorted conține toate
    elementele din List (fără duplicate), sortate în ordinea
    standard
?- sort([2, 3, 1, 2], X).
X = [1, 2, 3].
```

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

findall/3

```
findall(+Template, +Goal, -Bag)
```

Predicatul findall creează o listă de instanțieri ale lui Template care satisfac Goal și apoi unifică rezultatul cu Bag

```
higherThan(Numbers, Element, Result):-
    findall(X, (member(X, Numbers), X > Element), Result).
?- higherThan([1, 2, 7, 9, 11], 5, X).
X = [7, 9, 11]

?- findall([X, SqX], (member(X, [1,2,7,9,15]), X > 5, SqX is X **
     2), Result). % în argumentul Template putem construi
    structuri mai complexe
Result = [[7, 49], [9, 81], [15, 225]].
```

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

bagof/3

```
bagof(+Template, +Goal, -Bag)
```

Predicatul bagof este asemănător cu predicatul findall, cu excepția faptului că predicatul bagof construiește câte o listă separată pentru fiecare instanțiere diferită a variabilelor din Goal (fie că ele sunt numite sau sunt înlocuite cu underscore.

```
are(andrei,laptop,1). are(andrei,pix,5). are(andrei,ghiozdan,2).
are(radu,papagal,1). are(radu,ghiozdan,1). are(radu,laptop,2).
are(ana, telefon, 3). are(ana, masina, 1).

?- findall(X, are(_, X, _), Bag).
Bag = [laptop, pix, ghiozdan, papagal, ghiozdan, laptop, telefon,
     masina]. % laptop și ghiozdan apar de două ori pentru că
    sunt două posibile legări pentru persoană și pentru cantitate

?- bagof(X, are(andrei, X, _), Bag).
Bag = [laptop] ;
Bag = [ghiozdan] ;
Bag = [pix].
% bagof creează câte o soluție pentru fiecare posibilă legare
    pentru cantitate.  Putem aici folosi operatorul existențial ^
?- bagof(X, C^are(andrei, X, C), Bag).
Bag = [laptop, pix, ghiozdan]. % am cerut lui bagof să pună toate
    soluțiile indiferent de legarea lui C în același grup

?- bagof(X, C^are(P, X, C), Bag).
P = ana, Bag = [telefon, masina] ;
P = andrei, Bag = [laptop, pix, ghiozdan] ;
P = radu, Bag = [papagal, ghiozdan, laptop].
```

Dacă aplicăm operatorul existențial pe toate variabilele libere din scop, rezultatul este identic cu cel al lui `findall`.

```
?- bagof(X, X^P^C^are(P, X, C), Bag).
Bag = [laptop, pix, ghiozdan, papagal, ghiozdan, laptop, telefon,
     masina].
```

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

setof/3

```
setof(+Template, +Goal, -Bag)
```

Predicatul setof este asemănător cu bagof, dar sortează rezultatul (și elimină duplicatele) folosind sort/2.

```
?- setof(X, C^are(P, X, C), Bag).
P = ana, Bag = [masina, telefon] ; %se observă sortarea
P = andrei, Bag = [ghiozdan, laptop, pix] ;
P = radu, Bag = [ghiozdan, laptop, papagal].

?- setof(X, P^C^are(P, X, C), Bag).% setof elimină duplicatele
Bag = [ghiozdan, laptop, masina, papagal, pix, telefon].
```

Backtracking când se cunoaște lungimea căii către soluție

Regulă: Atunci când calea către soluţie respectă un anumit template (avem de instanţiat un număr finit, predeterminat, de variabile), este eficient să definim un astfel de template în program.
De exemplu, pentru problema celor opt regine putem scrie astfel:

```
template([1/_, 2/_, 3/_, 4/_, 5/_, 6/_, 7/_, 8/_]).

correct([]):-!.
correct([X/Y|Others]):-
        correct(Others),
        member(Y, [1, 2, 3, 4, 5, 6, 7, 8]),
        safe(X/Y, Others). % predicat care verifică faptul că
            reginele nu se atacă între ele

solve_queens(S):-template(S), correct(S).
```

Backtracking când calea are un număr nedeterminat de stări

În această situaţie nu este posibil să definim un template care descrie forma soluţiei problemei. Vom defini o căutare mai generală, după modelul următor:

```
solve(Solution):-
        initial_state(State),
        search([State], Solution).
```

search(+StăriVizitate,-Soluţie) definește mecanismul general de căutare astfel:

- căutarea începe de la o stare inițială dată (predicatul initial_state/1)
- dintr-o stare curentă se generează stările următoare posibile (predicatul next_state/2)
- se testează că starea în care s-a trecut este nevizitată anterior (evitând astfel traseele ciclice)
- căutarea continuă din noua stare, până se întâlnește o stare finală (predicatul final_state/1)

```
search([CurrentState|Other], Solution):-
        final_state(CurrentState), !,
        reverse([CurrentState|Other], Solution).

search([CurrentState|Other], Solution):-
        next_state(CurrentState, NextState),
        \+ member(NextState, Other),
        search([NextState,CurrentState|Other], Solution).
```

Mecanism BFS

bfs(+CoadaStărilorNevizitate,+StăriVizitate,-Soluţie) va defini mecanismul general de căutare în lăţime, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spaţiul stărilor (StartNode cu părintele nil)
- se generează toate stările următoare posibile
- se adaugă toate aceste stări la coada de stări încă nevizitate
- căutarea continuă din starea aflată la începutul cozii, până se întâlneşte o stare finală

```
bfs([(StartNode,nil)], [], Discovered).
```

Mecanism A*

astar(+End, +Frontier, +Discovered, +Grid, -Result) va defini mecanismul general de căutare A*, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spaţiul stărilor (StartNode cu părintele nil) și distanța estimată de la acesta până la nodul de final printr-o euristică (exemplu: distanța Manhattan)
- se generează toate stările următoare posibile și se calculează costul acestora adăugând costul acțiunii din părinte până în starea aleasă cu costul real calculat pentru a ajunge în părinte (costul părintelui în Discovered)
- dacă starea aleasă nu este în Discovered sau dacă noul cost calculat al acesteia este mai mic decât cel din Discovered, se adaugă în acesta, apoi va fi introdusă în coada de priorități (Frontier) cu prioritatea fiind costul cu care a fost adaugată în Discovered + valoarea dată de euristică din starea curentă până în cea finală
- căutarea continuă din starea aflată la începutul cozii, până se întâlneşte o stare finală

```
astar_search(Start, End, Grid, Path) :-
  manhattan(Start, End, H),
  astar(End, [H:Start], [Start:("None", 0)], Grid, Discovered),
  get_path(Start, End, Discovered, [End], Path).
```