Liste

Una lista è una sequenza finita di elementi.

Esempi:

[antonio, vittorio, tommaso]
[antonio, 2, 3.0]
[antonio, padre(antonio), X, Y]
[] (la lista vuota)
[antonio, [2, [b,c]], f(X), []]

Una lista non vuota si può vedere come composta di due parti:

- testa: primo elemento della lista
- coda: la lista che si ottiene eliminando il primo elemento

[antonio, [2, [b,c]], f(X), []]
testa: antonio
coda: [[2, [b,c]], f(X), []]

L'operatore predefinito |

L'operatore | si può usare per decomporre una lista nella sua testa e coda

```
?- [Testa|Coda] = [antonio, vittorio, tommaso].
Testa = antonio,
Coda = [vittorio, tommaso].

?- [Testa|Coda] = [antonio, [2, [b,c]], f(X), [] ].
Testa = antonio,
Coda = [[2, [b, c]], f(X), []].

?- [Testa|Coda] = [].
false.

?- [Uno,Due|Resto] = [antonio, [2, [b,c]], f(X), [] ].
Uno = antonio,
Due = [2, [b, c]],
Resto = [f(X), []].
```

Predicati predefiniti sulle liste: member

Guardare sul manuale SWI: library(lists): List Manipulation:
http://www.swi-prolog.org/pldoc/man?section=lists

member(?Elem, ?List): True if Elem is a member of List.

Notation of Predicate Descriptions

- \+ Argument must be fully instantiated. Think of the argument as input.
- \- Argument must be unbound. Think of the argument as output.
- ? Think of the argument as either input or output or both input and output.

```
%% mem = member, l'underscore _ e' la "variabile muta"
mem(X,[X|_]).
mem(X,[_|Coda]) :- mem(X,Coda).

?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c.
```

Predicati predefiniti sulle liste: length

length(?List, ?Int) True if Int represents the number of elements in List.

```
%% lung = length (pressappoco)
lung([],0).
lung([_|Coda],N) :- length(Coda,M), N is M+1.
```

Attenzione: length è un predicato

```
/* sum_of_lung(+L,?N) = L e' una lista di liste e N e'
          la somma delle lunghezze delle liste  in L */
sum_of_lung([],0).
sum_of_lung([Prima|Resto],N) :-
        length(Prima,P), sum_of_lung(Resto,R), N is P+R.
```

length è reversibile, ma attenzione:

```
?- member(a,L), length(L,3).
L = [a, _G330, _G333] ;
L = [_G327, a, _G333] ;
L = [_G327, _G330, a] ;
ERROR: Out of global stack
```

Predicati predefiniti sulle liste: append

append(?List1, ?List2, ?List1AndList2): List1AndList2 is the concatenation of List1 and List2

```
%%%concat = append
concat([],X,X).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

Notare l'uso dell'unificazione nella regola: avremmo potuto scriverla così (ma non sarebbe Prolog style!):

```
concat([X|L1],L2,Z) :- concat(L1,L2,L3), Z=[X|L3].
```

per calcolare la concatenzione Z di [X|L1] e L2, calcolare la concatenazione L3 di L1 e L2. Z è allora [X|L3].

Ma tanto vale unificare direttamente nella testa della clausola il "risultato" con [X|L3]

Reversibilità di append

```
?- append([a,b],[1,2,3],X).
X = [a, b, 1, 2, 3].

?- append(X,[1,2,3],[a,b,1,2,3]).
X = [a, b]

?- append(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3] ;
X = [1, 2],
Y = [3] ;
X = [1, 2, 3],
Y = [] ;
false.
```

Uso di append

```
%%% ultimo(L,X) = X e' l'ultimo elemento di L
ultimo(L,X) :-
        append(_,[X],L).
```

Attenzione: append è un predicato

```
/* append3(L1,L2,L3,Result) = Result e' la concatenazione
                 di L1, L2 e L3 */
append3(L1,L2,L3,Result) :-
        append(L1,L2,L),
        append(L,L3,Result).
```

Esercizio: vedere tutte le soluzioni fornite dal Prolog per il goal append3(X,Y,Z,[a,b,c]), osservare l'ordine in cui sono generate e meditare su come funziona il backtracking.

Accesso alla definizione dei predicati

```
?- listing(ultimo).
ultimo(B, A) :-
        append(_, [A], B).
true.

?- listing(nextto).
lists:nextto(A, B, [A, B|_]).
lists:nextto(A, B, [_|C]) :-
        nextto(A, B, C).
true.
```

listing senza argomenti: elenca tutte le clausole del programma definite dall'utente.

Notare: si può usare lo stesso nome per due predicati che hanno un diverso numero di argomenti (arità), il Prolog non li confonde, ma li considera predicati distinti.