ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog – Liste

Federico Chesani

DISI

Department of Informatics – Science and Engineering

Disclaimer & Further Reading

- These slides are largely based on previous work by Prof. Paola Mello

Liste

- Le liste sono una delle strutture dati primitive più diffuse nei linguaggi di programmazione per l'elaborazione simbolica (es: Lisp)
- In Prolog le liste sono dei termini costruiti a partire da uno speciale atomo (che denota la lista vuota) e utilizzando un particolare operatore funzionale (l'operatore ".").

La definizione di **lista** può essere data ricorsivamente nel modo seguente:

- l'atomo `[]` rappresenta la lista vuota
- il termine `.(T,Lista)` è una lista se `T` è un termine qualsiasi e `Lista` una Lista. `T` prende il nome di **TESTA** della lista e `Lista` di **CODA**

Liste – esempi

- I seguenti termini Prolog sono liste:

```
(1)     []
(2)     .(a, [])
(3)     .(a, .(b,[]))
(4)     .(f(g(X)), . (h(z), .(c, [])))
(5)     .([], [])
(6)     .(.(a,[]), .(b, []))
```

Il Prolog fornisce una notazione semplificata per la rappresentazione delle liste: la lista `.(T,LISTA)` può essere rappresentata anche come

```
[T | LISTA]
```

- Tale rappresentazione può essere paragonata all'applicazione della funzione "cons" del Lisp. La testa (head) T e la coda (tail) LISTA della lista non sono altro che i risultati dell'applicazione delle funzioni Lisp "car" e "cdr" alla lista stessa.

Liste – esempi

- Le liste nell'esempio precedente possono essere rappresentate nel modo seguente:

```
(1)     []
(2)     [a | []]
(3)     [a | [b | []]]
(4)     [f(g(X)) | [h(z) | [c | []]]]
(5)     [[] | []]
(6)     [[a | []] | [b | []]]
```

Ulteriore semplificazione; la lista `[a| [b| [c]]]` può essere rappresentata nel modo seguente: `[a, b, c]`

Liste – esempi

- Le liste nell'esempio precedente possono essere rappresentate
  nel modo seguente:

```
(1)      []
(2)      [a | []]
(3)      [a | [b | []]]
(4)      [f(g(X)) | [h(z) | [c | []]]]
(5)      [[] | []]
(6)      [[a | []] | [b | []]]

(1)      []
(2)      [a]
(3)      [a,b]
(4)      [f(g(X)),h(z),c]
(5)      [[]]
(6)      [[a],b]
```

Unificazione sulle Liste

- L'unificazione (combinata con le varie notazioni per le liste) è un potente meccanismo per l'accesso alle liste:

```
p([1,2,3,4,5,6,7,8,9]).
```

```
:-p(X).
yes X=[1,2,3,4,5,6,7,8,9]
:-  p([X|Y]).
yes X=1    Y=[2,3,4,5,6,7,8,9]
:-  p([X,Y|Z]).
yes X=1 Y=2 Z=[3,4,5,6,7,8,9]
:-  p([_|X]).
yes X=[2,3,4,5,6,7,8,9]
```

Operazioni sulle Liste

Le procedure che operano su liste sono definite come procedure ricorsive basate sulla definizione ricorsiva di lista

- **Verificare se un termine è una lista**

```
% is_list(T)        true    se T è una lista
                    false   se T non è una lista

is_list([]).
is_list([X|L])  :-  is_list(L).

% Esempi:
:-  is_list([1,2,3]).
yes

:-  is_list([a|b]).
no
```

Operazioni sulle Liste

- **Verificare se un termine appartiene ad una lista**

```
% member(T,L) "T è un elemento della lista L"

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

% Esempi:
:-  member(2, [1,2,3]).
yes

:-  member(1, [2,3]).
no

:-  member(X, [1,2,3]).
yes X=1;
    X=2;
    X=3;
no
```

*La relazione **member** può quindi essere utilizzata in più di un modo (per la verifica di appartenenza di un elemento ad una lista o per individuare gli elementi di una lista).*

Esercizio

Si scriva un predicato che determini l'ultimo elemento di una lista.

```
% last(L,X) "X è l'ultimo elemento della lista L"

% Esempi:
?-  last([a,b,c],N).
X = c;
no

?-  last([a,b,[c,d,e]],X).
X = [c,d,e];
no
```

Esercizio – Soluzione

```
% last(L,X) "X è l'ultimo elemento della lista L"

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

% Esempi:
?-last([b],b).
yes

?-last([a],b).
no

?-last([],b).
no
```

Operazioni sulle Liste

- **Determinare la lunghezza di una lista**

```
% length(L,N) "la lista L ha N elementi"

  % Versione Ricorsiva:
  lenght([],0).
  length([_|L],N) :- length(L,N1),
                     N is N1 + 1.

  % Versione Iterativa:
  length1(L,N)  :-  length1(L, 0, N).
  lenght1([], ACC, ACC).
  length1([_|L], ACC, N):- ACC1 is ACC+1,
                           length1(L, ACC1, N).
```

Operazioni sulle Liste – Osservazione

Molte operazioni su lista seguono lo stesso pattern:

- Caso base (lista vuota)
- Caso ricorsivo (isolo Testa, e ripeto su Resto)

Esempi:

```
length([],0).
length([_|L],N)   :-  length(L,N1),
                      N is N1 + 1.

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

Esercizio

Si scriva un predicato che, dati un termine T e una lista L, conti le occorrenze di T in L.

```
% conta(T,L,N) "N è il numero di occorrenze del
  termine T nella lista L"
```

Esempio:

```
?-  conta(a,[b,a,a,b,c,a],N).
N = 3;
no

?-  conta(a,[b,a,a,b,c,a],3).
yes
```

Esercizio – Soluzione ricorsiva non tail

```
conta(_,[],0).
conta(T, [T|R], N)  :- conta(T,R,N1),
                       N is N1+1.
conta(T, [H|R], N)  :- T\= H,
                       conta(T,R,N).

% Esempio:
?-  conta(a,[a,a,b,c,a],N).
N = 3;
no
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Esercizio – Soluzione ricorsiva tail

```
conta(T,L,N):- c(T,L,0,N). % proxy

% il terzo parametro è l'accumulatore parziale
c(_,[],N,N).
c(T,[T|R],Nin,Nout):- N1 is Nin + 1,
                      c(T,R, N1, Nout).
c(T,[H|R],Nin,Nout):- T \= H,
                      c(T,R,Nin,Nout).
```

Esercizio – sum_list

Si scriva un predicato **`sum_list`** che, data in ingresso una lista di interi come primo argomento, ha successo se il suo secondo argomento è la somma dei valori in tale lista.

```
% sum_list(L,S) "S è la somma degli elementi della lista L
    data"

% Esempi:
?-  sum_list([1,2,3], 6).
Yes

?- sum_list([1,2,3], X).
Yes X=6
```

Esercizio – sum_list – versione ricorsiva

```
sum_list([], 0).
sum_list([H|T], R):-
                sum_list(T, R1),
                R is R1+H.
```

Operazioni sulle Liste

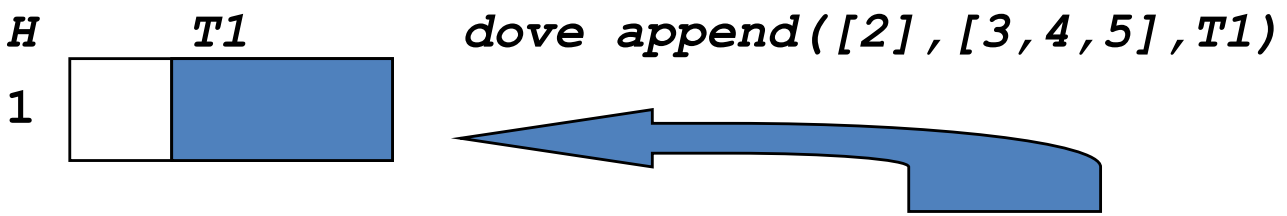
- ## Concatenazione di due liste

```
% append(L1,L2,L3) "L3 è il risultato della concatenazione
di L1 e L2"

append([],L,L).
append([H|T],L2,[H|T1]):- append(T,L2,T1).

% Esempi:
:- append([1,2],[3,4,5],L).
yes  L = [1,2,3,4,5]

:- append([1,2],L2,[1,2,4,5]).
yes  L2 = [4,5]

:- append([1,3],[2,4],[1,2,3,4]).
no
```

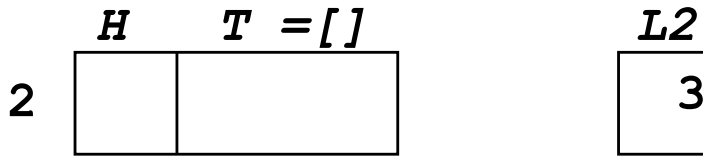
append/3 – Evoluzione della computazione

- In seguito alla valutazione del goal:

```
:-append([1,2],[3,4,5],L).
```

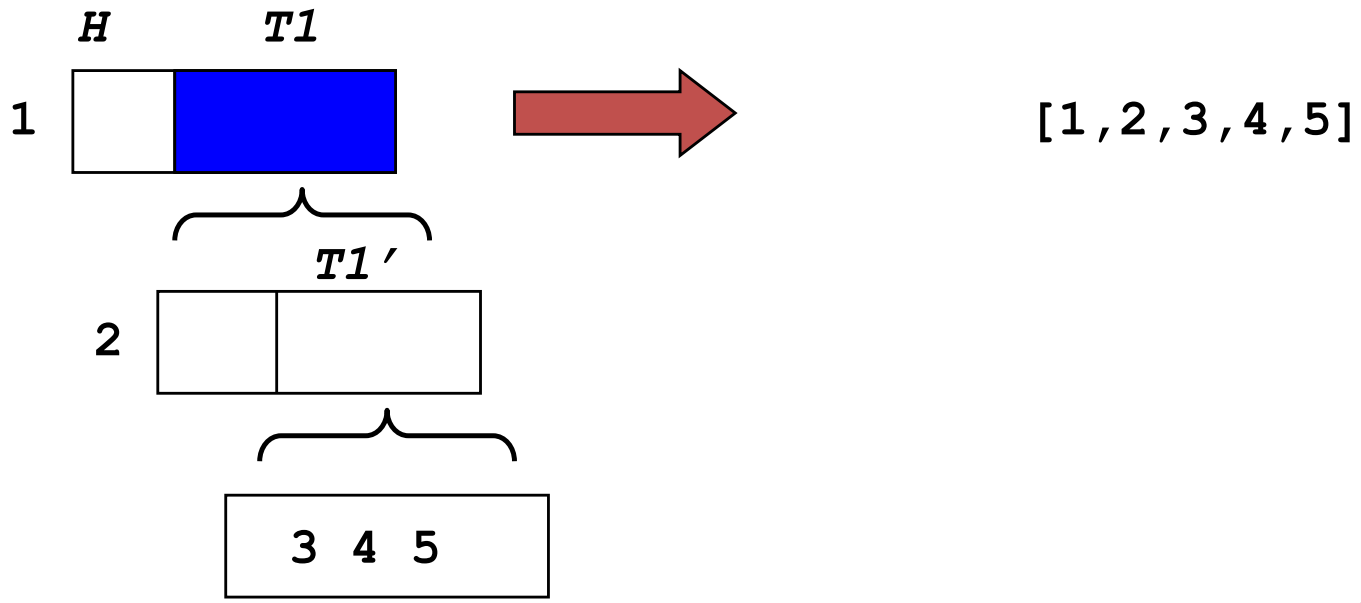


  - Viene generato il sottogoal `:- append([2],[3,4,5],T1)`



append/3 – Evoluzione della computazione

- Viene generato il sottogoal `:- append([],[3,4,5],T1')`
- Utilizzando la prima clausola si ha che `T1' = [3,4,5]`



append/3 – Esempi

```
:-  append(L1, L2, [a,b,c,d]).
 yes L1=[]          L2=[a,b,c,d];
     L1=[a]         L2=[b,c,d];
     L1=[a,b]       L2=[c,d];
     L1=[a,b,c]     L2=[d];
     L1=[a,b,c,d]  L2=[]

 :-  append(L1, [c,d], L).
 yes L1=[]          L=[c,d];
     L1=[_1]        L=[_1,c,d];
     L2=[_1,_2]     L=[_1,_2,c,d];
     (infinite soluzioni)

 :-  append([a,b], L1, L).
 yes L1=_1      L=[a,b | _1]
 :-  append(L1, L2, L3).
 yes L1=[]          L2=_1        L3=_1;
     L1=[_1]        L2=_2,       L3=[_1 | _2];
     L1=[_1,_2]     L2=_3    L3=[_1,_2 | _3];
     (infinite soluzioni)
```

Liste di Liste e Liste Ibride

Le liste sono termini e quindi possono essere elementi di liste (**liste di liste o liste ibride**).

E' l'unificazione, di nuovo, il meccanismo con cui si fanno corrispondere i termini delle teste delle clausole (parametri formali) ai termini che compaiono nelle chiamate o sottogoal (parametri attuali)

```
?- length([b, a, b, [c,a], d],X).

?- member(X, [[1,2,3],[a], [d,c]]).

?- member(1, [[1,2,3],[a], [d,c]]).
```

Esercizio

Si scriva un predicato **depth_member/2** che ha successo se il suo primo argomento appartiene alla lista data come secondo argomento o, ricorsivamente, ad una lista elemento di tale lista.

```
% depth_member(T,L)       "T appartiene alla lista L o,
   ricorsivamente, ad una lista che appartiene a L"


% Esempi:
?-  depth_member(b,[a,b,[c,a],d]).
Yes
?-  depth_member([c,a],[a,b,[c,a],d]).
Yes
?-  depth_member(X,[a,b,[c,a],d]).
X=a  ;
X=b  ;
...
```

Esercizio – Soluzione

```
depth_member(T,[T|_]).
depth_member(T,[L|_]) :- is_list(L),
                         depth_member(T,L).
depth_member(T,[_|L]) :- depth_member(T,L).
```

Operazioni sulle Liste

- Cancellazione di uno o più elementi dalla lista

```
% delete1(El,L,L1) "la lista L1 contiene gli elementi di L
                    tranne il primo termine unificabile con El"

    delete1(El,[],[]).
    delete1(El,[El|T],T).
    delete1(El,[H|T],[H|T1]):-  delete1(El,T,T1).

% oppure:
% delete(El,L,L1) "la lista L1 contiene gli elementi di L
                  tranne tutti i termini unificabili con El"

    delete(El,[],[]).
    delete(El,[El|T],T1)  :-  delete(El,T,T1).
    delete(El,[H|T],[H|T1]) :- delete(El,T,T1).
```

Attenzione!

Le due procedure **delete** e **delete1** forniscono **una sola** risposta corretta ma non sono corrette in fase di backtracking.

```
:-  delete(a,[a,b,a,c],L).
yes      L=[b,c]; % UNICA CORRETTA!!!
  L=[b,a,c];
  L=[a,b,c];
  L=[a,b,a,c];
no
```

- Il problema è legato alla mutua esclusione tra la seconda e la terza clausola della relazione **delete** :
  - Se **T** appartiene alla lista (per cui la seconda clausola di delete ha successo), allora la terza clausola non deve essere considerata una alternativa valida.
  - Una possibile soluzione sarà usando il **cut**; altrimenti?

Sulla Rimozione di Elementi da Liste

Cancellazione del **primo** elemento uguale a T dalla lista: le clausole (**cl2**) e (**cl3**) devono essere mutuamente esclusive.

La condizione di mutua esclusione è l'unificazione dell'elemento da cancellare con la testa della lista.

```
(cl1)  delete1(T, [], []).
(cl2') delete1(T, [T |TAIL], TAIL).
(cl3)  delete1(T, [HEAD|TAIL], [HEAD|L]):-
                    T\=HEAD, delete1(T, TAIL, L).
```

Sulla Rimozione di Elementi da Liste

Cancellazione di **tutti** gli elementi uguali a T dalla lista: le clausole **(cl5)** e **(cl6)**devono essere mutuamente esclusive.

La condizione di mutua esclusione è l'unificazione dell'elemento da cancellare con la testa della lista

```
(cl4) delete(T, [], []).
(cl5) delete(T, [T | TAIL], L):-
                        delete(T, TAIL, L).
(cl6)     delete(T,[HEAD|TAIL],[HEAD|L]):-
                T\=HEAD, delete(T, TAIL, L).
```

Vedremo un altro modo (usando un operatore detto cut "!")

Esercizio – Esame del 11 Settembre 2008

Si scriva un programma Prolog **`no_dupl(Xs,Ys)`** che è vero se **`Ys`** è la lista (senza duplicazioni) degli elementi che compaiono nella lista **`Xs`**. Nella lista **`Ys`** gli elementi compaiono nello stesso ordine di **`Xs`** ed, in caso di elementi duplicati, si manterrà l'ultima occorrenza.

Esempi:

```
?-no_dupl([a,b,a,d],[b,a,d]).
yes
```

```
?-no_dupl([a,b,a,c,d,b,e], L).
yes  L=[a,c,d,b,e]
```

Esercizio – Esame del 11 Settembre 2008 – Soluzione

```
no_dupl([], []).
no_dupl([X|Xs], Ys):-
            member(X, Xs),
            no_dupl(Xs, Ys).
no_dupl([X|Xs], [X|Ys]):-
            nonmember(X, Xs),
            no_dupl(Xs, Ys).


nonmember(_, []).
nonmember(X, [Y|Ys]) :- X \== Y,
      nonmember(X, Ys).
```

Operazioni sulle Liste

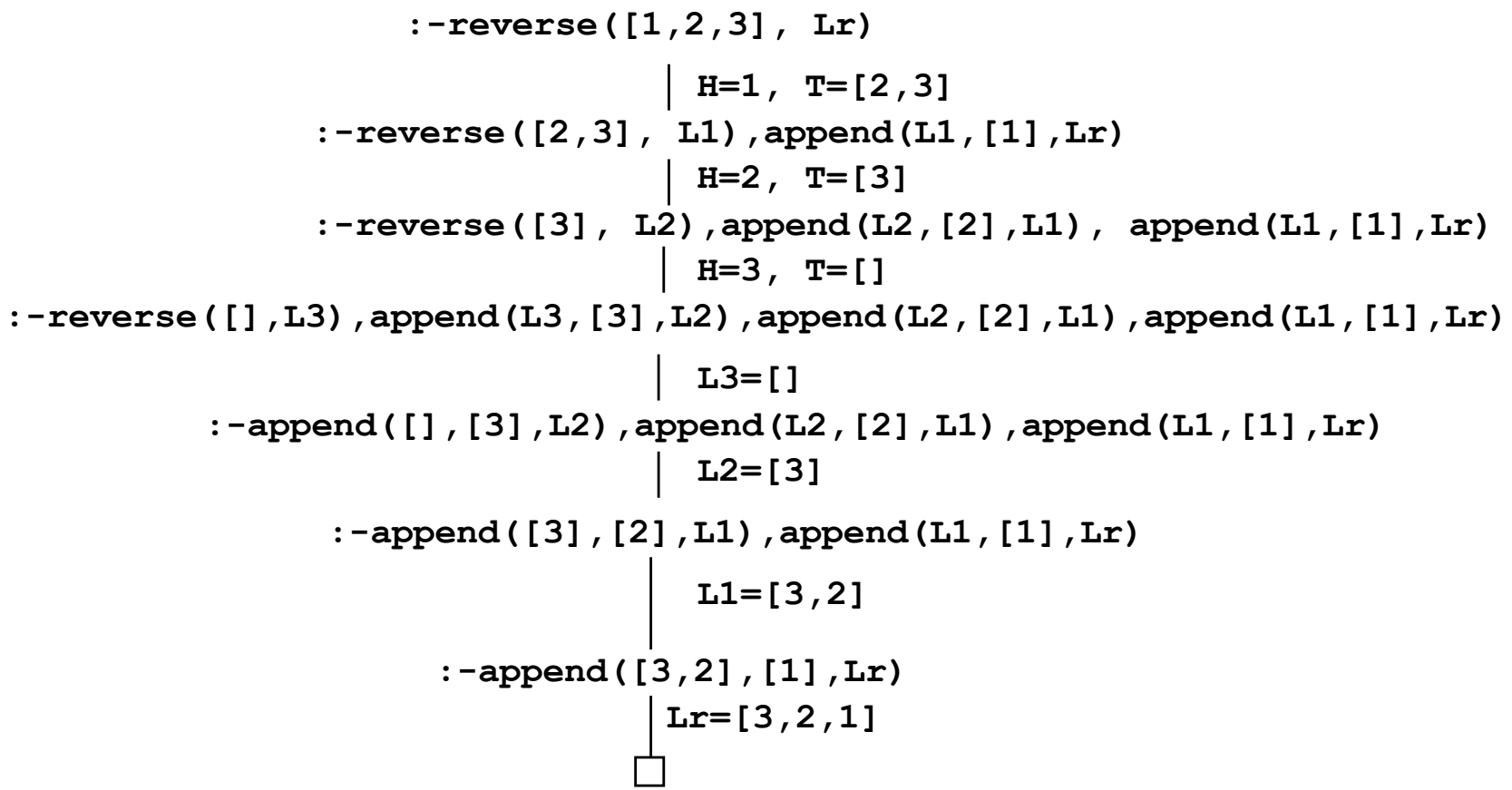
- **Inversione di una lista**

```
% reverse(L,Lr) "la lista Lr contiene gli elementi di L
                  in ordine inverso"

reverse([],[]).
reverse([H|T], Lr):- reverse(T,T2),
                     append(T2,[H],Lr).

:- reverse([],[]).
yes

:- reverse([1,2],Lr).
yes Lr = [2,1]
```

Operazioni sulle Liste – Inversione – Esempio SLD

Evoluzione della computazione in seguito alla valutazione del goal

```
:- reverse([1,2,3], Lr).
```



Operazioni sulle Liste

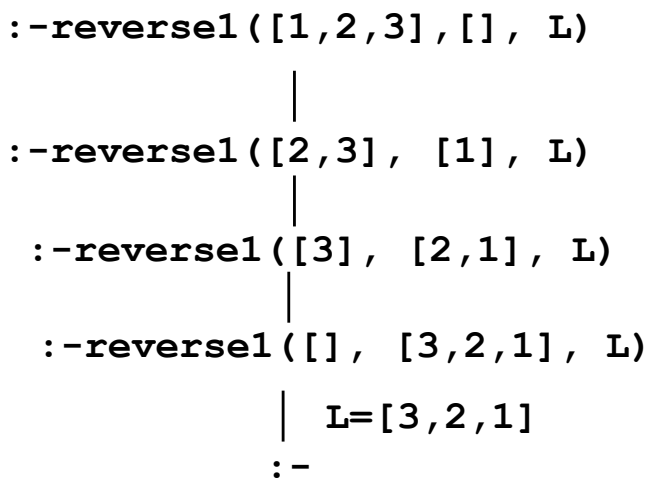
- **Inversione di una lista – versione Iterativa**

```
% reverse1(L,Lr) "la lista Lr contiene gli elementi di L
                  in ordine inverso"

reverse1(L,Lr) :- reverse1(L,[],Lr). % proxy

reverse1([], ACC, ACC).
reverse1([H|T],ACC,Y) :- reverse1(T,[H|ACC],Y).
```

- La lista L da invertire viene diminuita ad ogni passo di un elemento e tale elemento viene accumulato in una nuova lista (in testa)
  - Un elemento viene accumulato davanti all'elemento che lo precedeva nella lista originaria, ottenendo in questo modo l'inversione. Quando la prima lista è vuota, l'accumulatore contiene la lista invertita

Operazioni sulle Liste – Inversione – Esempio SLD

Evoluzione della computazione in seguito alla valutazione del goal
`:-reverse1([1,2,3],[], Lr).`



Operazioni sugli Insiemi

Gli insiemi possono essere rappresentati come liste di oggetti (senza ripetizioni).

- **Intersezione di due insiemi**

```
% intersection(S1,S2,S3) "l'insieme S3 contiene gli elementi
                         appartenenti all'intersezione di S1 e S2"

intersection([],S2,[]).
intersection([H|T],S2,[H|T3]):- member(H,S2),
                                intersection(T,S2,T3).
intersection([H|T],S2,S3):- intersection(T,S2,S3).
```

```
:-  intersection([a,b], [b,c], S).

  yes   S=[b]
```

```
:-  intersection([a,b,c,d],S2,[a,c]).

  yes   S2=[a,c | _1]
```

Intersezione tra due Insiemi – attenzione!

```
:-  intersection(S1,S2,[a,c]).
  yes   S1=[a,c]          S2=[a,c | _1];
        S1=[a,c,_2]       S2=[a,c | _1];
        S1=[a,c,_2,_3] S2=[a,c | _1];
        .....
        (infinite soluzioni)
        ...
:-  intersection([a,b,c],[b,c,d],S3).
yes     S3=[b,c]; % Unica soluzione corretta!! Problema
                  % della mutua esclusione tra clausole

        S3=[b];
        S3=[c];
        S3=[];
  no
```

Operazioni sugli Insiemi

- **Unione di due insiemi**

```
% union(S1,S2,S3) "l'insieme S3 contiene gli elementi
                   appartenenti all'unione di S1 e S2"
```

```
union([], S2, S2).
union([X|REST],S2,S):- member(X, S2),
                       union(REST, S2, S).
union([X|REST],S2,[X|S]):- union(REST, S2,
  S).
```

- Anche il predicato **`union`** in backtracking ha un comportamento scorretto. Infatti, anche in questo caso non c'è mutua esclusione tra la seconda e la terza clausola.