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]
```

UNIFICAZIONE SULLE LISTE (proviamo!)

- 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]).
no
:-  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
- In molti Prolog le trovate già definite (ad esempio in SWIsh)
- Verificare se un termine è una lista (***proviamo!***)

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

```
is_list([]).

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

```
:-  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).
```

```
:-  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 2.2 b)

b) 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 2.2 b)

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

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

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

OPERAZIONI SULLE LISTE

- Determinare la lunghezza di una lista

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

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

Versione ricorsiva

(non tail)

```
?-length([a,b,d,g],L).
L=4   ;
No
```

Scriverne una versione tail-ricorsiva

```
?-length(X,Y).
X=[]     Y=0 ;
X=[_1]   Y=1 ;
...
```

OPERAZIONI SULLE LISTE

- Determinare la lunghezza di una lista

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

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

Versione ricorsiva

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

Versione tail ricorsiva (processo computazionale iterativo)

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 2.1 b)

- Si scriva un predicato depth_member 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"

Esempio:

```
?-  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 2.1 b)

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

Oppure:

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

Esercizio 2.2 f)

f) 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
```

Esercizio 2.2 f) (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).

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

Esercizio 2.1 f) (tail ricorsiva)

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

c(_,[],N,N).
c(T,[T|R],Nin,Nout):- !,
                      N1 is Nin+1,
                      c(T,R,N1,Nout).
c(T,[_|R],Nin,Nout):- 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

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

Esempio:

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

Esercizio (sum_list) – vers. ricorsiva

- Lista di interi, restituisce la somma degli elementi

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

Esercizio (sum_list) – vers. tail ricorsiva

- Lista di interi, restituisce la somma degli elementi

```
sum_list([], R, R).
sum_list([H|T], Rin, Rout):-
                    Rtemp is Rin+H,
                    sum_list(T, Rtemp, Rout).

?-sum_list([1,2,3],0,X).
Yes X=6
```

RICORSIONE E ITERAZIONE

- Il Prolog non fornisce alcun costrutto sintattico per l'iterazione (quali, ad esempio, i costrutti *while* e *repeat*) e l'unico meccanismo per ottenere iterazione è la definizione ricorsiva.

- Una funzione f è definita per *ricorsione tail* se f è la funzione "più esterna" nella definizione ricorsiva o, in altri termini, se sul risultato della chiamata ricorsiva di f non vengono effettuate ulteriori operazioni

- La definizione di funzioni (predicati) per ricorsione tail può essere considerata come una definizione per *iterazione*
  - Potrebbe essere valutata in spazio costante mediante un processo di valutazione iterativo.

RICORSIONE E ITERAZIONE

- Si dice *ottimizzazione della ricorsione tail* valutare una funzione tail ricorsiva f mediante un processo iterativo ossia caricando un solo record di attivazione per f sullo stack di valutazione (esecuzione).

- In Prolog l'ottimizzazione della ricorsione tail è un po' più complicata che non nel caso dei linguaggi imperativi a causa del:
  - non determinismo
  - della presenza di punti di scelta nella definizione delle clausole.

RICORSIONE E ITERAZIONE

```
     p(X)   :-  c1(X), g(X).
(a)  p(X)   :-  c2(X), h1(X,Y), p(Y).
(b)  p(X)   :-  c3(X), h2(X,Y), p(Y).
```

- Due possibilità di valutazione ricorsiva del goal `:-p(Z).`
  - se viene scelta la clausola (a), si deve ricordare che (b) è un punto di scelta ancora aperto. Bisogna mantenere alcune informazioni contenute nel record di attivazione di `p(Z)` (i punti di scelta ancora aperti)
  - se viene scelta la clausola (b) (più in generale, l'ultima clausola della procedura), non è più necessario mantenere alcuna informazione contenuta nel record di attivazione di p(Z) e la rimozione di tale record di attivazione può essere effettuata

QUINDI...

- **In Prolog l'ottimizzazione della ricorsione tail è possibile solo se la scelta nella valutazione di un predicato "p" è deterministica o, meglio, se al momento della chiamata ricorsiva (n+1)-esima di "p" non vi sono alternative aperte per la chiamata al passo n-esimo (ossia alternative che potrebbero essere considerate in fase di backtracking)**

- Quasi tutti gli interpreti Prolog effettuano l'ottimizzazione della ricorsione tail ed è pertanto conveniente usare il più possibile ricorsione di tipo tail.

ALTRE OPERAZIONI SULLE LISTE

- Concatenazione di due liste

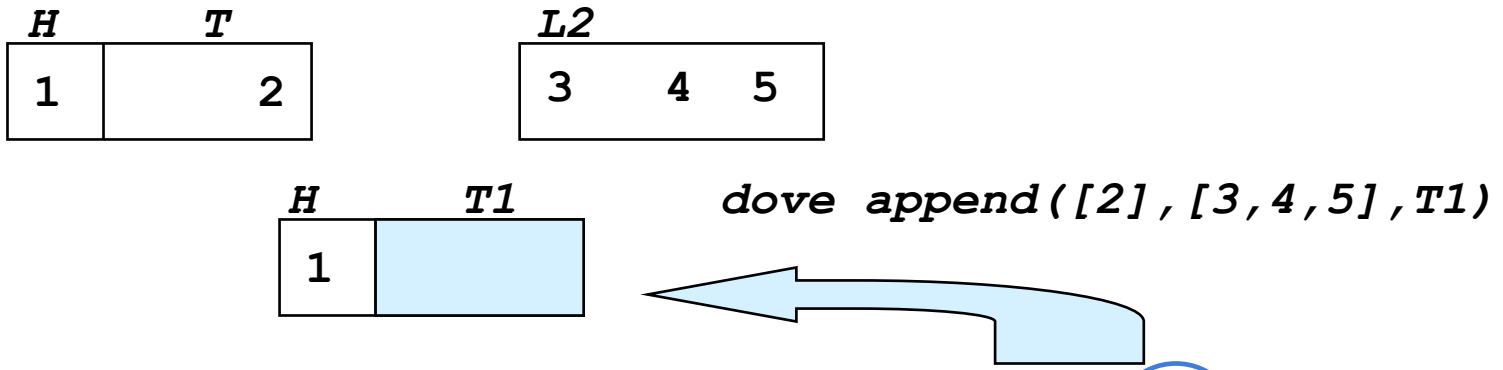
append(L1,L2,L3) "L3 e' il risultato della concatenazione di L1 e L2"

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

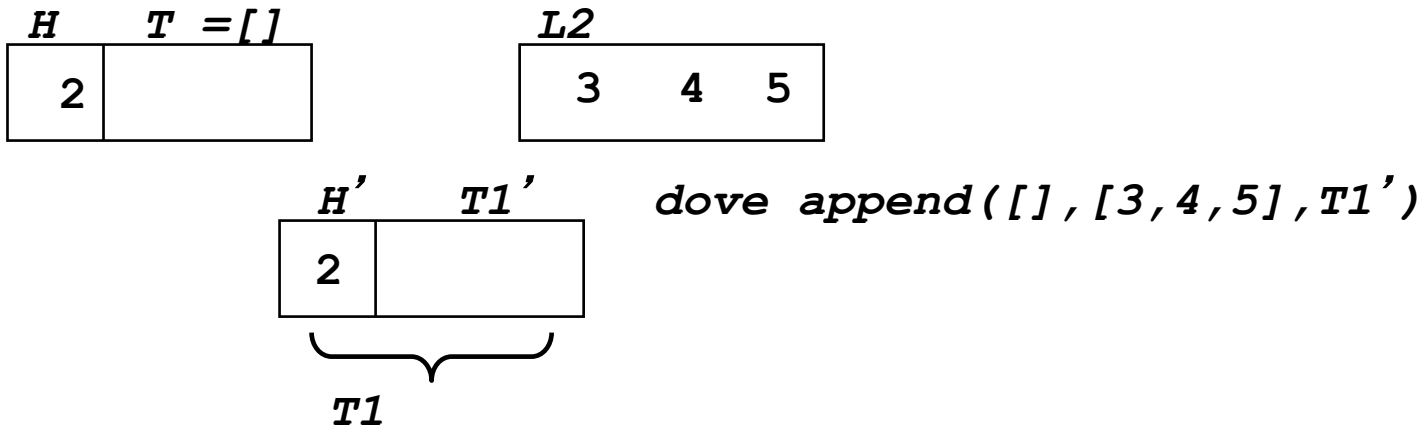
```
:- append([1,2],[3,4,5],L).
yes  L = [1,2,3,4,5]
:- append([1,2],L2,[1,2,4,5]).
yes  L2 = [4,5]
:- append([1,3],[2,4],[1,2,3,4]).
no
```

ESEMPIO

- Evoluzione della computazione in seguito alla valutazione del goal
  `:-append([1,2],[3,4,5],L).`

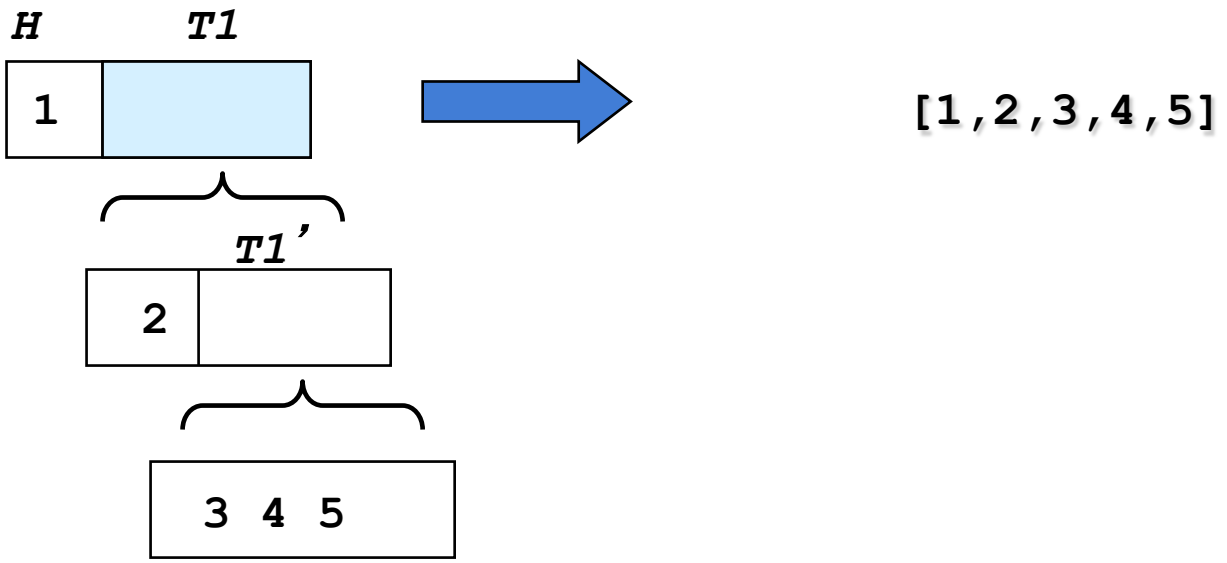


- Viene generato il sottogoal `:- append([2],[3,4,5],T1)`



ESEMPIO

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



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)
```

ESEMPI

```
:-  append([a,b,[c,d]], [f(a),[e,g,h]], L).
yes L=[a,b,[c,d],f(a),[e,g,h]]
```

Come visto, 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)

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).
```

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 soluzione 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 **`El`** appartiene alla lista (per cui la seconda clausola di **`delete`** ha successo), allora la terza clausola non deve essere considerata una alternativa valida.

⬇

Questo problema è risolto dal **CUT (!)**

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).
```

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 "!")

RIMOZIONE DI ELEMENTI DA LISTE

- Cancellazione di un elemento uguale a T dalla lista

```
(cl1)    delete1(T, [], []).
(cl2)    delete1(T, [T|TAIL], TAIL).
(cl3)    delete1(T, [HEAD|TAIL], [HEAD|L]):-
                    T\==HEAD, delete1(T, TAIL, L).
```

- Cancellazione di tutti gli elementi uguali a T dalla 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).
```

RIMOZIONE DI ELEMENTI DA LISTE – cut!

- 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]):-
                   delete1(T, TAIL, L).
```

RIMOZIONE DI ELEMENTI DA LISTE – cut!

- Cancellazione di tutti gli elementi uguali a T dalla lista: le clausole `(c15)` e `(c16)` devono essere mutuamente esclusive.

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

```
(c14)    delete(T, [], []).
(c15)    delete(T, [T | TAIL], L):- !,
                          delete(T, TAIL, L).
(c16)    delete(T,[HEAD|TAIL],[HEAD|L]):-
                         delete(T, TAIL, L).
```

Esercizio 2.2 e)

e) Si scriva un predicato che elimini da una lista tutti gli elementi ripetuti.

del_rip(L1,L2) "L2 è la lista L1 privata di tutte le ripetizioni"

Esempio:

```
?-  del_rip([a,b,b,c,a,d],L).
L = [a,b,c,d];
no
```

Esercizio 2.2 e)

```
del_rip([],[]).
del_rip([X|T],[X|T1]):- member(X,T),!,
                        delete(X,T,R),
                        del_rip(R,T1).
del_rip([X|T],[X|T1]):- del_rip(T,T1).


delete(_, [], []).
delete(X, [X | T], R):- !,delete(X, T, R).
delete(X, [H | T], [H | R]):- delete(X, T, R).
```

OPERAZIONI SULLE LISTE

- Inversione di una lista

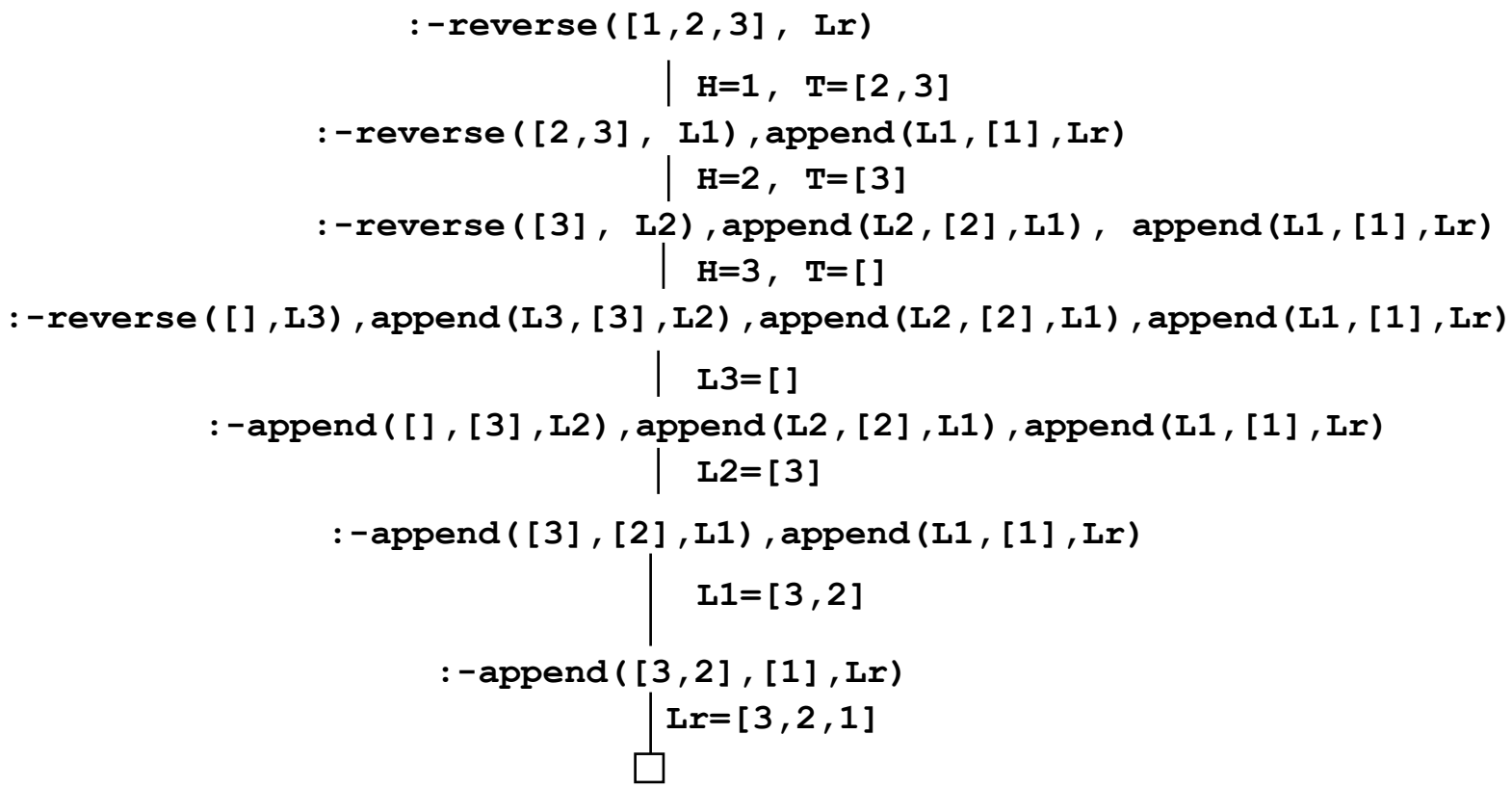
```
reverse(L,Lr) "la lista Lr contiene gli elementi di L
               in ordine inverso"
```

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

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

ESEMPIO

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



Reverse (tail ricorsiva)

- Si scriva un predicato che faccia l'inversione di una lista tramite una procedura tail-ricorsiva

- Suggerimento: procedura ausiliara con un parametro "accumulatore" che parte dalla lista vuota e nel quale le chiamate che si succedono aggiungono elementi.

`reverse(L,Lr)` "L2 contiene gli elementi della la lista L1 ma in ordine inverso"

```
reverse(L,Lr):-reverse1(L,[],Lr).
```

OPERAZIONI SULLE LISTE

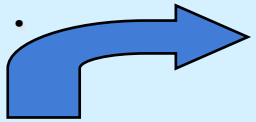
- Inversione di una lista (Versione tail ricorsiva)

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

```
reverse1(L,Lr) :- reverse1(L,[],Lr).

reverse1([], ACC, ACC).

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

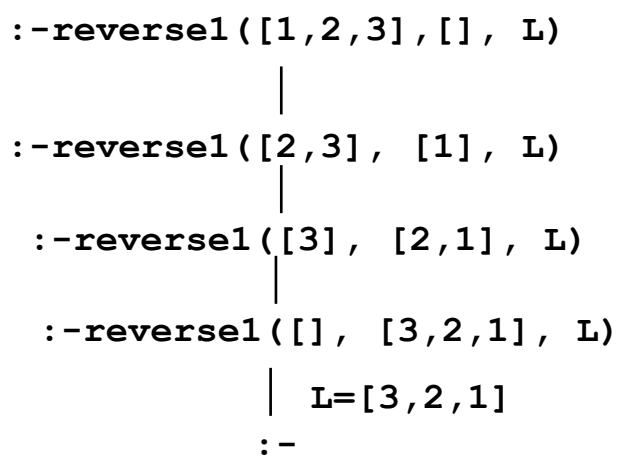


Potevamo usare la **append**

- 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*

ESEMPIO

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



Esercizio 2.2 d)

d) Si scriva un predicato che verifichi se una lista è palindroma (ossia uguale alla sua inversa).

palin(L) "L è una lista palindroma"

Esempi:

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

?-  palin([a,b,c]).
no
```

Esercizio 2.2 d)

```
palin([]):-!.
palin(L1):-reverse(L1,L1).


reverse([],[]):-!.
reverse([HEAD|TAIL],L2) :- reverse(TAIL,TAIL1),
                           append(TAIL1,[HEAD],L2).

append([],L,L):-!.
append([HEAD|TAIL],L,[HEAD|TAIL1]) :-
                  append(TAIL,L,TAIL1).
```

OPERAZIONI SUGLI INSIEMI

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

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

OPERAZIONI SUGLI INSIEMI

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

- ?-union([a,b],[c,d],[a,b,c,d]).
- ?-union([a,b],[c,b],L).

Esercizio 2.2 f)

f) 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
```

Esercizio 2.2 f) (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).

?-  conta(a,[a,a,b,c,a],N).
N = 3;
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).
```

- ?-union([a,b],[c,d],[a,b,c,d]).
- ?-union([a,b],[c,d],L).

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).
```

- ?-union([a,b],[c,d],[a,b,c,d]).
- ?-union([a,b],[c,d],L).

Esercizio 2.2 a)

a) Si scriva un predicato che fornisca la lista intersezione di due liste.

intersection(X,Y,Z) "Z è l'insieme intersezione X ∩ Y"

Esempio:

```
?-  intersection([1,2,3], [2,3,4], L).
L = [2,3];
no
```

OPERAZIONI SUGLI INSIEMI

- Insiemi 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]
```

ESEMPIO

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

Unica soluzione corretta !!
Problema della mutua esclusione tra clausole

Lo risolveremo usando il predicato cut (!)

Esercizio 2.2 c)

c) Si scriva un predicato che, date due liste L1 e L2, verifichi se L1 è una sottolista di L2 (sia come subset, sia come sottolista propria).

subset(L1,L2) "L2 contiene tutti gli elementi di L1 nello stesso ordine"

sublist(L1,L2) "L2 contiene L1 come parte di sé stessa"

Esempi:

```
?-  subset([1,2],[1,3,2,4]).
yes
?-  subset([1,2],[2,3,1,4]).
no
?-  sublist([1,2],[3,4,1,2,5]).
yes
?-  sublist([1,2],[3,4,1,5,2]).
no
```

Esercizio 2.2 c)

```
subset([],_):- !.
subset([X|T],[X|L2]):- !, subset(T,L2).
subset(L1,[_|L2]):- subset(L1,L2).


sublist([],_):- !.
sublist([X|T],L2):- member_list(X,L2,L2rest),
                    subl_aux(T,L2rest).

member_list(X,[X|T],T):- !.
member_list(X,[_|T],Trest):- member_list(X,T,Trest).

subl_aux([],_):- !.
subl_aux([X|T],[X|L2]):- subl_aux(T,L2).
```

Esercizio compito 11 Sett 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 compito 11 Sett 2008

```
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).
```

ESERCIZI PROPOSTI

- Programma Prolog per determinare l'ultimo elemento di una lista.
- Programma Prolog che, date due liste L1 e L2, verifica se L1 è una sottolista di L2.
- Programma Prolog per verificare se una lista è palindroma, ossia uguale alla sua inversa.
- Programma Prolog che elimina da una lista tutti gli elementi ripetuti.
- Programma Prolog che, dati un termine T e una lista L, conta le occorrenze di T in L.
- Programma Prolog per appiattire una lista (multipla). Ad esempio l'appiattimento della lista [1,[2,3,[4]],5,[6]] deve produrre la lista [1,2,3,4,5,6].
- Programma Prolog per effettuare l'ordinamento (sort) di una lista. Ad esempio, si realizzano algoritmi di ordinamento quali l'ordinamento per inserzione, il "bubblesort", il "quicksort".