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]
```

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

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).
```

```
:-  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).*

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 iterativa

OPERAZIONI SULLE LISTE

- Concatenazione di due liste

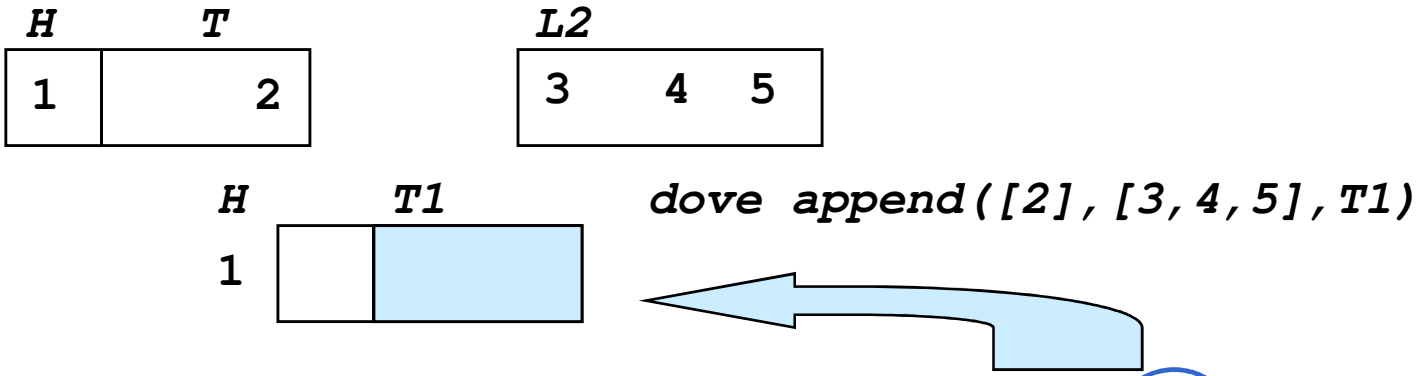
append(L1,L2,L3) "L3 e' il risultato della concatenazione di L1 e L2"

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

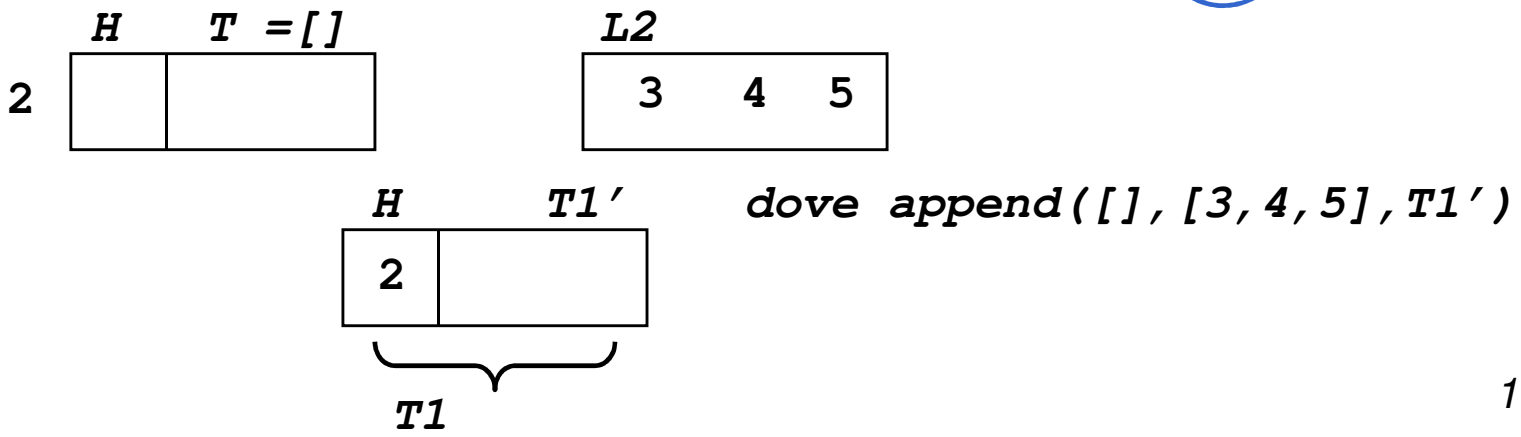
```
:- append([1,2],[3,4,5],L).
yes  L = [1,2,3,4,5]
:- append([1,2],L2,[1,2,4,5]).
yes  L2 = [4,5]
:- append([1,3],[2,4],[1,2,3,4]).
no
```

ESEMPIO

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

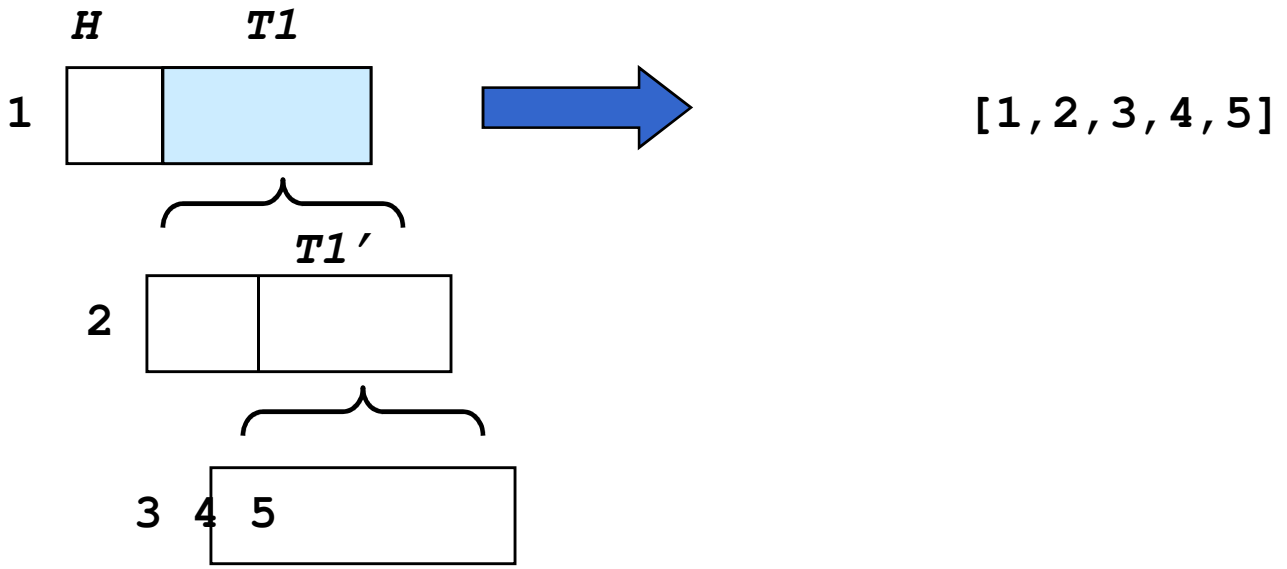


- Viene generato il sottogoal `:- append([2],[3,4,5],T1)`



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)
```

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];
     L=[b,a,c];
     L=[a,b,c];
     L=[a,b,a,c];
no
```

Unica soluzione corretta !!

- 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.

Questo problema verra' risolto dal **CUT**

OPERAZIONI SULLE LISTE

- Inversione di una lista

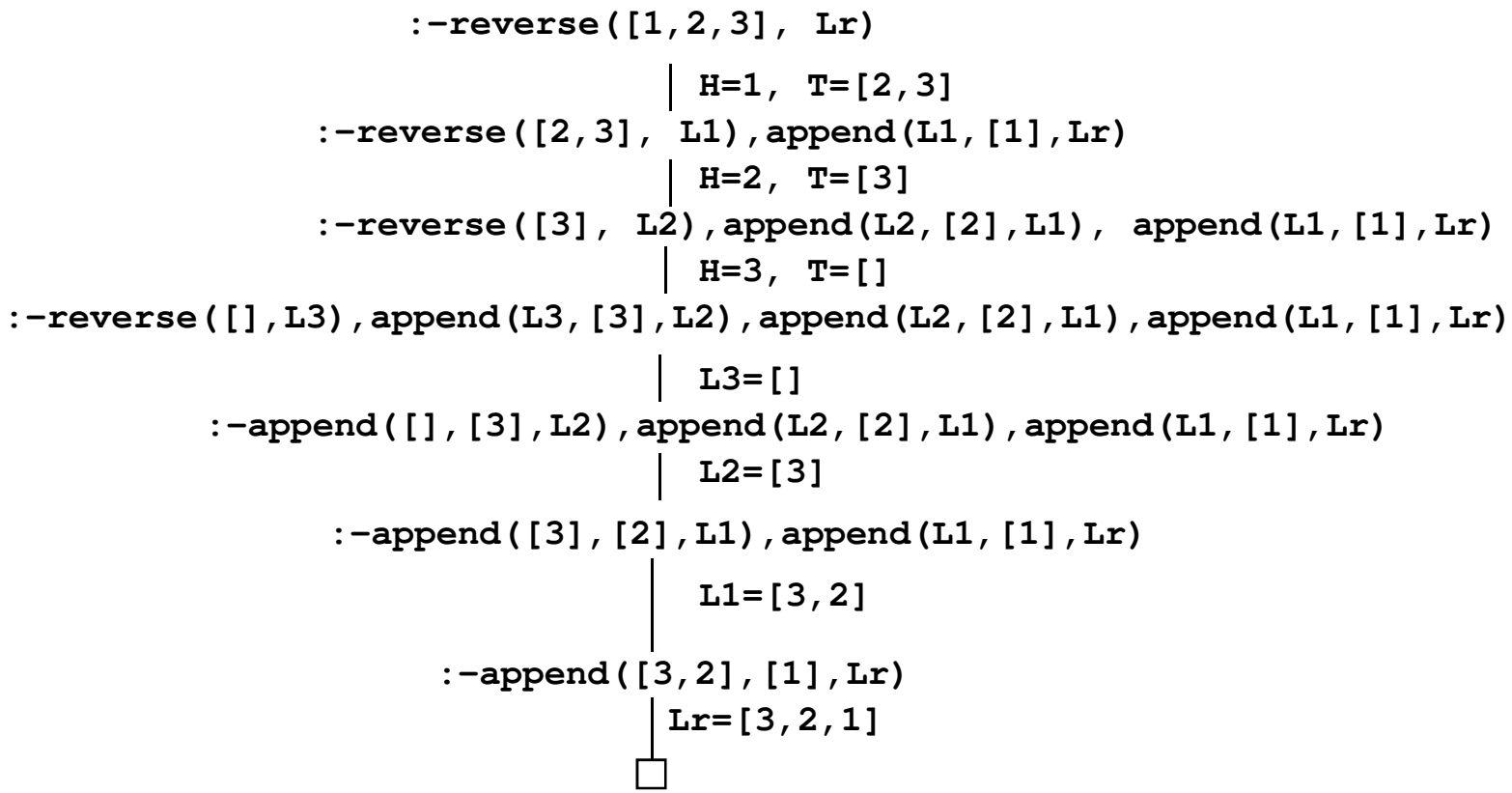
```
reverse(L,Lr) "la lista Lr contiene gli elementi di L
               in ordine inverso"
```

```
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).`

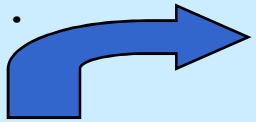


OPERAZIONI SULLE LISTE

- 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).
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*

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]
```

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

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'e' mutua esclusione tra la seconda e la terza clausola.

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".