

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
```

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 concatenazione 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.
```

```
%%% 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

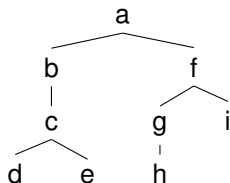
```
?- listing(ultimo).  
ultimo(B, A) :-  
    append(_, [A], B).  
true.
```

```
?- 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 (**arietà**), il Prolog non li confonde, ma li considera predicati distinti.

Rappresentazione di alberi binari



```
/* is_tree(T) :  
   T e' un albero binario */  
is_tree(empty).  
is_tree(t(_,Left,Right)) :-  
    is_tree(Left),  
    is_tree(Right).
```

```
t(a,t(b,t(c,t(d,empty,empty),t(e,empty,empty)),empty),  
t(f,t(g,t(h,empty,empty),empty),t(i,empty,empty)))
```

Notare l'uso della “variabile muta” (o “anonima”)

Occorrenza di un elemento in un albero binario

```
/* in_tree(El,T): El occorre come etichetta in T */
% caso base: El occorre in un albero che ha El come radice
in_tree(El,t(El,_,_)). %% uso dell'unificazione

%% casi ricorsivi: El occorre in un albero se occorre
%% nel suo sottoalbero sinistro o in quello destro
in_tree(El,t(_,Left,_)) :-
    in_tree(El,Left).
in_tree(El,t(_,_,Right)) :-
    in_tree(El,Right).
```

Le ultime due clausole si possono compattare usando l'OR del Prolog:

```
in_tree(El,t(_,Left,Right)) :-
    in_tree(El,Left) ; in_tree(El,Right).
```

Il programma è reversibile?

```
in_tree(El,t(El,_,_)).
in_tree(El,t(_,Left,Right)) :-
    in_tree(El,Left) ; in_tree(El,Right).

?- in_tree(X,t(b,t(c,t(d,empty,empty),
                  t(e,empty,empty)),empty)).

X = b ;
X = c ;
X = d ;
X = e ;
false.
```

Il programma è reversibile?

```
in_tree(El,t(El,_,_)).  
in_tree(El,t(_,Left,Right)) :-  
    in_tree(El,Left) ; in_tree(El,Right).
```

```
?- in_tree(X,t(b,t(c,t(d,empty,empty),  
                t(e,empty,empty)),empty)).
```

```
X = b ;
```

```
X = c ;
```

```
X = d ;
```

```
X = e ;
```

```
false.
```

```
?- in_tree(a,T).
```

```
T = t(a, _G278, _G279) ;
```

```
T = t(_G277, t(a, _G282, _G283), _G279) ;
```

```
T = t(_G277, t(_G281, t(a, _G286, _G287), _G283), _G279) ;
```

```
...
```

Procedure con input/output determinati

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.

```
/* size(+T,?N): T ha N nodi */  
size(empty,0).  
size(t(_,Left,Right),H) :-  
    size(Left,N), size(Right,M), H is N+M+1.
```

```
/* height(+T,?N): T ha altezza N */  
height(empty,0).  
height(t(_,Left,Right),H) :-  
    height(Left,N), height(Right,M),  
    (N>M, H is N+1 ; N=<M, H is M+1).
```

Algoritmi di backtracking

Implementare in Prolog algoritmi di backtracking è molto semplice.

Ad esempio: ricerca di un ramo in un albero binario fino a una foglia con una data etichetta.

branch(+T,?Leaf,?Ramo) = Ramo è un ramo di T dalla radice a una foglia etichettata da Leaf

```
branch(t(X,empty,empty),X,[X]).
branch(t(X,Left,Right),Leaf,[X|Ramo]) :-
    branch(Left,Leaf,Ramo) ;
    branch(Right,Leaf,Ramo).
```

Notare che quello che è falso non va “detto”.

Il programma si può utilizzare per generare tutte le “coppie” foglia-ramo di un albero dato:

```
?- branch(...,Leaf,Ramo).
```

Anche le sequenze di goal e le clausole sono termini complessi, i cui **operatori** principali sono, rispettivamente, la virgola e :-

Sono operatori anche i simboli usati per le funzioni aritmetiche (+, *, mod, ...), gli operatori di confronto (=, <, >, ...)

<http://www.swi-prolog.org/pldoc/man?section=operators>

Gli operatori sono come i funtori, sono utilizzati per rendere il codice più leggibile; altrimenti si dovrebbe scrivere, per esempio **+(3,2)**, anziché **3+2**

Ogni operatore è caratterizzato da: la sintassi con cui deve essere utilizzato, la precedenza e l'associatività

Sintassi e precedenza degli operatori

Un operatore può essere

- **infixo** (sempre a due argomenti): va scritto in mezzo ai suoi due argomenti (esempi: $+$ $=$ $>$, ; ...)
- **prefisso**: va scritto prima degli argomenti (ad esempio, il segno $-$ per rappresentare i numeri negativi)
- **postfisso**: va scritto dopo gli argomenti (esempio: $++$ in C)

Sintassi e precedenza degli operatori

Un operatore può essere

- **infixo** (sempre a due argomenti): va scritto in mezzo ai suoi due argomenti (esempi: $+ = > , ; \dots$)
- **prefisso**: va scritto prima degli argomenti (ad esempio, il segno $-$ per rappresentare i numeri negativi)
- **postfisso**: va scritto dopo gli argomenti (esempio: $++$ in C)

A ogni operatore è associata una **precedenza**, per disambiguare le espressioni.

Per esempio: $2+3*5$ significa $2+(3*5)$, perché la precedenza di $+$ è **maggiore** di quella di $*$:

$$?- X+Y = 3+2*5.$$

$$X = 3,$$

$$Y = 2*5.$$

$$?- X*Y = 3+2*5.$$

$$\text{false.}$$

La precedenza è un valore numerico (tra 0 e 1200) associato all'operatore: minore è la "precedenza", maggiore è la sua "forza legante"; un operatore "lega più strettamente" di altri con "precedenza" maggiore ($+$ ha precedenza 500, $*$ ha precedenza 400)

Associatività degli operatori

L'associatività degli operatori serve per disambiguare espressioni in cui occorrono operatori con la stessa precedenza.

Ad esempio: **10-3-2** significa:

- **(10-3)-2** (**associativo a sinistra**), oppure
- **10-(3-2)** (**associativo a destra**) ?

?- $X-Y = 10-3-2.$

$X = 10-3,$

$Y = 2.$

L'associatività di $-$ è rappresentata da **yfx**: alla sua sinistra possono stare (senza bisogno di mettere parentesi) termini con precedenza minore o uguale a quella di $-$ (rappresentati da **y**), alla sua destra solo termini con precedenza strettamente minore di quella di $-$ (rappresentati da **x**)

Un operatore può anche essere **non associativo**: si devono comunque usare le parentesi

Tipo di un operatore

Convenzione per indicarne sintassi e associatività

Precedenza di un termine:

- atomi, strutture costruite con funtori, espressioni tra parentesi: precedenza 0
- espressione il cui operatore principale è \otimes : precedenza di \otimes

Il simbolo **f** rappresenta l'operatore, **x** e **y** i suoi argomenti: **x** è un termine con precedenza strettamente minore di quella di **f**, **y** è un termine con precedenza minore o uguale a quella di **f**.

yfx	infixo, associativo a sinistra
xfy	infixo, associativo a destra
xfx	infixo, non associativo
fx	prefisso, non associativo
fy	prefisso, associativo a destra
xf	postfisso, non associativo
yf	postfisso, associativo a sinistra

Esempio: un modo alternativo per scrivere la negazione

not è un funtore: il suo argomento va sempre messo tra parentesi.

```
?- not member(3, [1, 2, 4]).
```

```
ERROR: Syntax error: Operator expected
```

\+ è un operatore (con precedenza 900 e associatività **fy**) che ha lo stesso significato del **not**.

```
?- \+ member(3, [1, 2, 4]).
```

```
true.
```

```
?- \+ \+ member(3, [1, 2, 4]).
```

```
false.
```

Definizione di nuovi operatori

Il programmatore può definire nuovi operatori, specificandone precedenza e associatività.

`:- op(precedenza, tipo, nome).`

`% + lega meno stretto di »`

`?- op(600, fx, ») .`

`?- X+Y = »a+b.`

`false.`

`?- »X = »a+b.`

`X = a+b.`

Definizione di nuovi operatori

Il programmatore può definire nuovi operatori, specificandone precedenza e associatività.

`:- op(precedenza, tipo, nome).`

```
% + lega meno stretto di »  
?- op(600, fx, ») .
```

```
?- X+Y = »a+b.  
false.
```

```
?- »X = »a+b.  
X = a+b.
```

```
% « lega piu' stretto di +  
?- op(200, fx, «) .
```

```
?- X+Y = «a+b.  
X = «a,  
Y = b.
```

```
?- «X = «a+b.  
false.
```

Definizione di nuovi operatori

?- op(500,xfy,&).

?- X & Y = 3 * a & b.

X = 3*a,

Y = b.

?- X & Y = 3 & a & b.

X = 3,

Y = a&b.

Definizione di nuovi operatori

```
?- op(500,xfy,&).
```

```
?- X & Y = 3 * a & b.
```

```
X = 3*a,
```

```
Y = b.
```

```
?- X & Y = 3 & a & b.
```

```
X = 3,
```

```
Y = a&b.
```

```
?- X = 3 + a & b.
```

```
ERROR: Syntax error: Operator priority clash
```

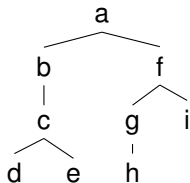
Non puo' essere **(3+a) & b**, perché **&** ha tipo **xfy**

Non puo' essere **3 + (a&b)**, perché **+** ha tipo **yfx**

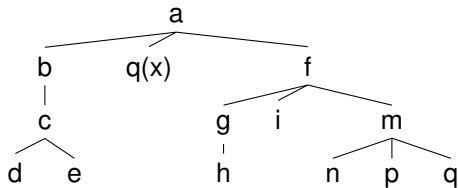
Esempio: rappresentazione di alberi n-ari

?- op(600,xfx,=>).

- **Radice => lista non vuota di sottoalberi**, oppure
- **Foglia** (qualsiasi termine che non abbia => come funtore principale)



$a \Rightarrow [b \Rightarrow [c \Rightarrow [d, e]],$
 $f \Rightarrow [g \Rightarrow [h,$
 $i]]$



$a \Rightarrow [b \Rightarrow [c \Rightarrow [d, e]],$
 $q(x),$
 $f \Rightarrow [g \Rightarrow [h,$
 $i,$
 $m \Rightarrow [n, p, q]]]$

Esempio: rappresentazione di formule proposizionali

Rappresentiamo la negazione mediante `-` (unario), che ha precedenza 200 e tipo `fy`.

```
:- op(600, yfx, &).      % Congiunzione
:- op(650, yfx, v).      % Disgiunzione
:- op(670, yfx, =>).     % Implicazione
:- op(680, yfx, <=>).    % Doppia implicazione
```

Generazione delle sottoformule di una formula

subforms(+A,-List): List è una lista con tutte le sottoformule di A.

```
subforms(A, [A]) :- atom(A).
subforms(-A, [-A|Rest]) :- subforms(A, Rest).
subforms(A, [A|Rest]) :-
    (A = F & G ; A = F v G ;
     A = F => G ; A = F <=> G),
    subforms(F, Fforms), subforms(G, Gforms),
    append(Fforms, Gforms, Rest).
```

```
?- subforms(a & b v -c => -(p v q & r), Forms),
    writeln(Forms).
```

```
[a&b v -c=> -(p v q&r), a&b v -c, a&b, a, b, -c, c,
 - (p v q&r), p v q&r, p, q&r, q, r]
```

```
Forms = [a&b v -c=> -(p v q&r), a&b v -c, a&b,
         a, b, -c, c, - (...v...), ...v...|...]
```