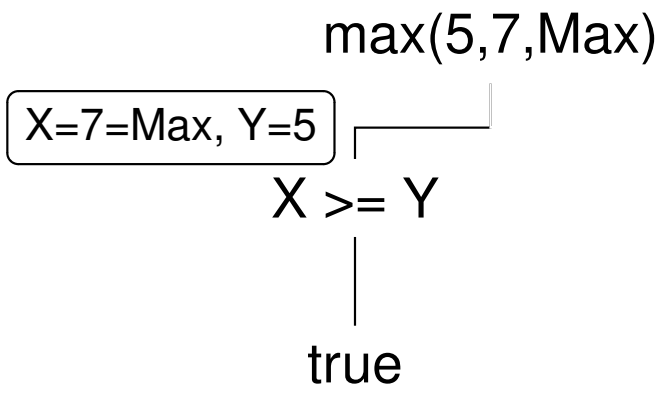
# Il controllo del backtracking

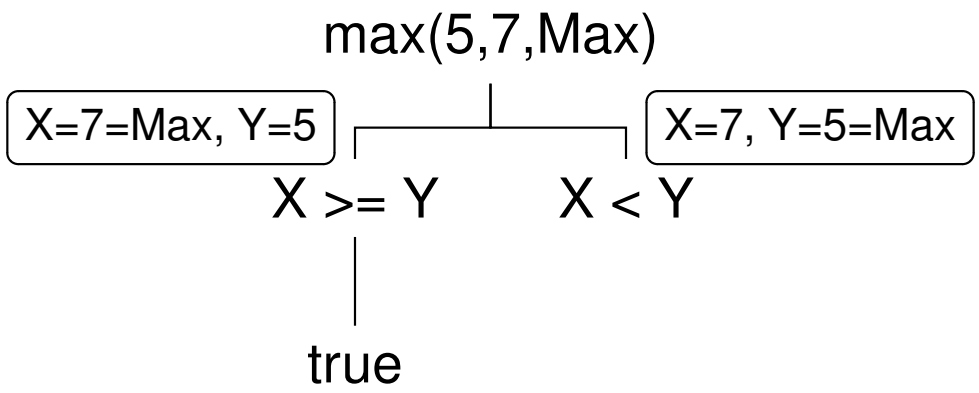
```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.

?- max(7,5,Max).
Max = 7
```




Le due clausole di max sono mutuamente esclusive: se X>=Y ha successo, è inutile cercare un'altra soluzione mediante la seconda clausola.

# Il controllo del backtracking

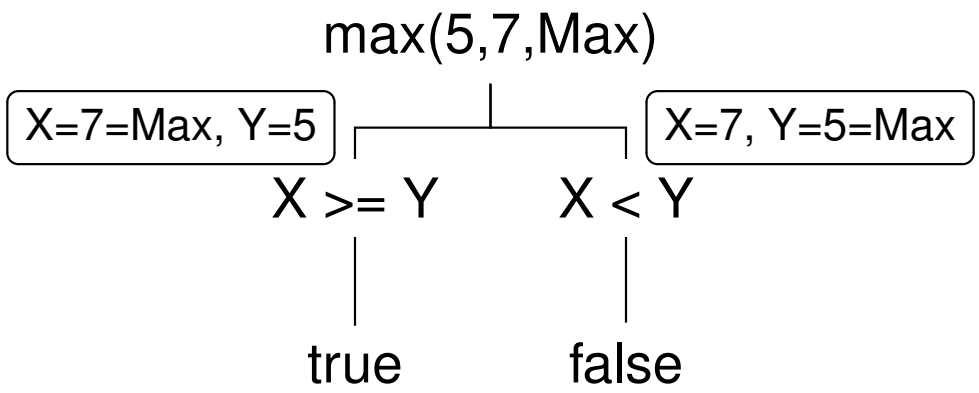
```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.

?- max(7,5,Max).
Max = 7 ;
```




Le due clausole di max sono mutuamente esclusive: se X>=Y ha successo, è inutile cercare un'altra soluzione mediante la seconda clausola.

# Il controllo del backtracking

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.

?- max(7,5,Max).
Max = 7 ;
false.
```




Le due clausole di max sono mutuamente esclusive: se X>=Y ha successo, è inutile cercare un'altra soluzione mediante la seconda clausola.

# Il controllo del backtracking

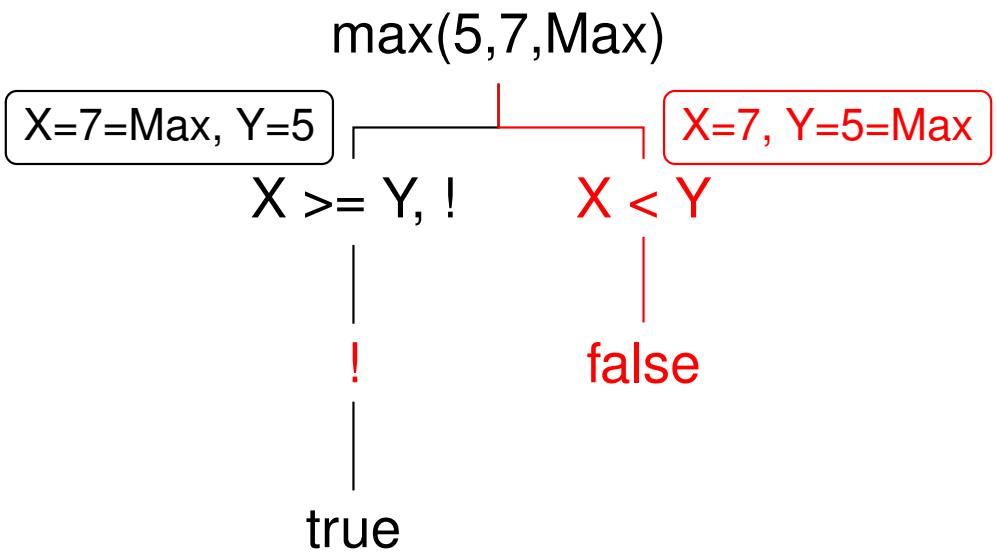
```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.

?- max(7,5,Max).
Max = 7.
```

Il Prolog non consente di chiedere un'altra soluzione

# Il controllo del backtracking

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.

?- max(7,5,Max).
Max = 7.
```

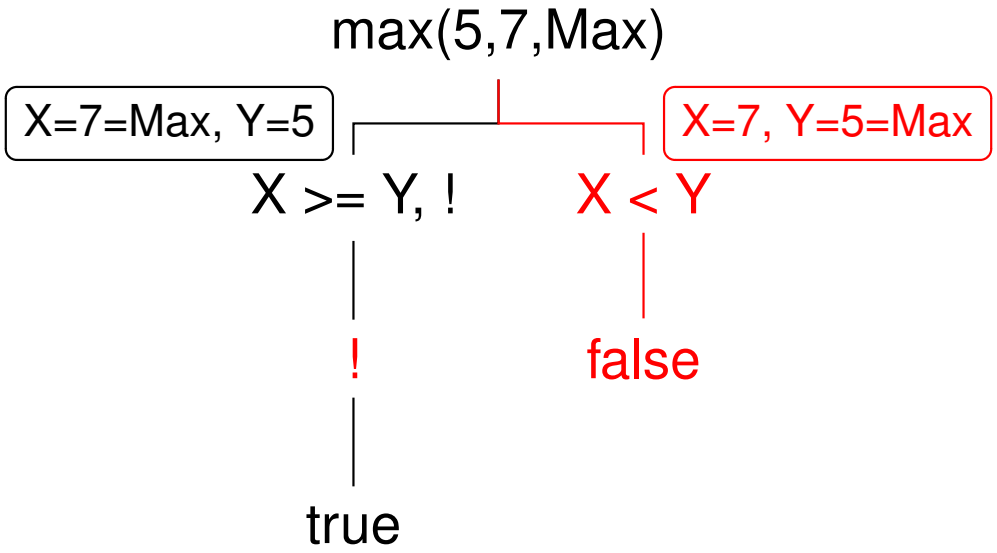
Il Prolog non consente di chiedere un'altra soluzione




Quando viene invocato il "cut", Il sottoalbero rosso viene potato

# La potatura

**max(5,7,Max)** è il “parent goal” (il goal che ha attivato la clausola contentente il cut).




Quando viene “eseguito” il cut, tutte le alternative aperte al di sotto del parent goal vengono potate (i punti di backtracking sono cancellati).

# Il cut

- Il cut è un predicato, senza argomenti, che consente di controllare il backtracking
- È un goal che ha sempre successo (dal punto di vista dichiarativo, equivale a **true**)
- Vincola il Prolog alle scelte fatte da quando è stato chiamato il parent goal.

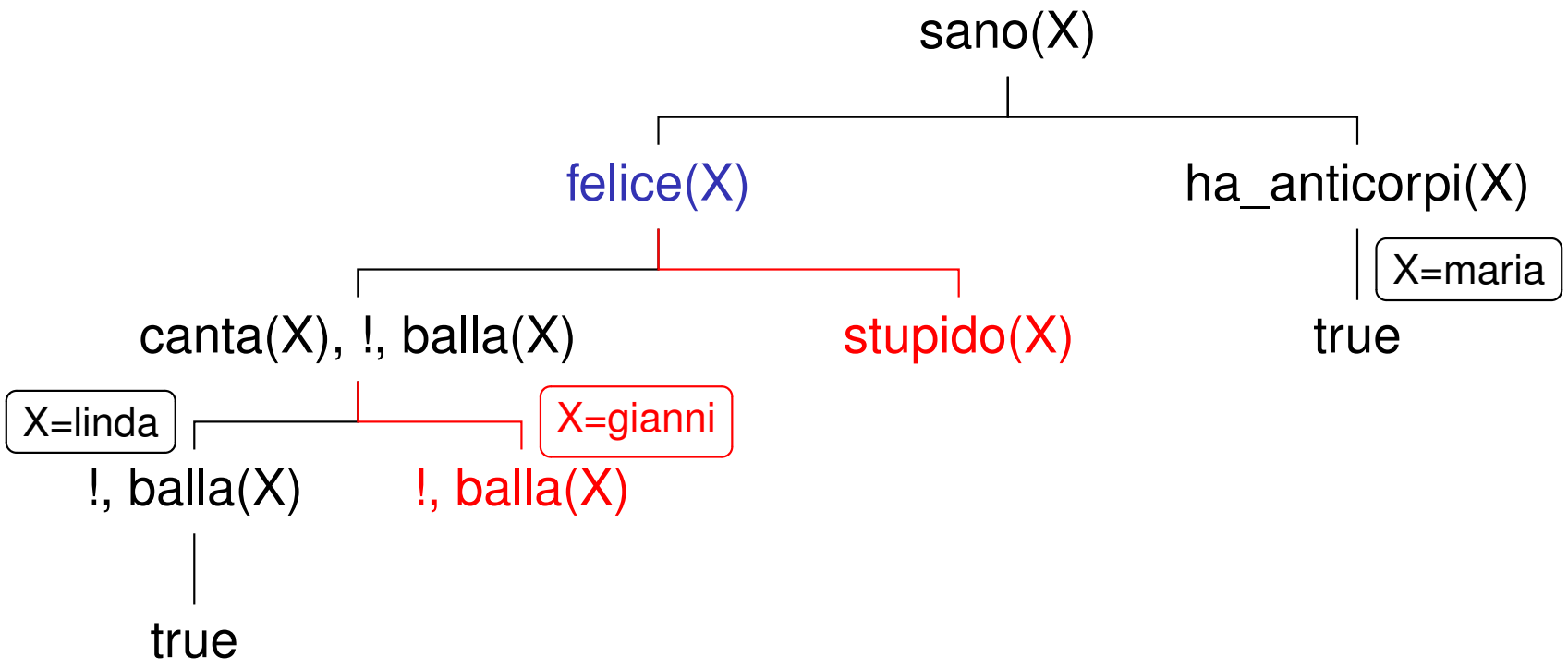
# Un altro esempio

Consideriamo inizialmente un programma senza cut

```
sano(X) :- felice(X).
sano(X) :- ha_anticorpi(X).

felice(X) :- canta(X), balla(X).
felice(X) :- stupido(X).

ha_anticorpi(maria).
balla(linda).
canta(linda).
canta(gianni).
stupido(gianni).

?- sano(X).
X = linda ;
X = gianni ;
X = maria.
```

# Un altro esempio

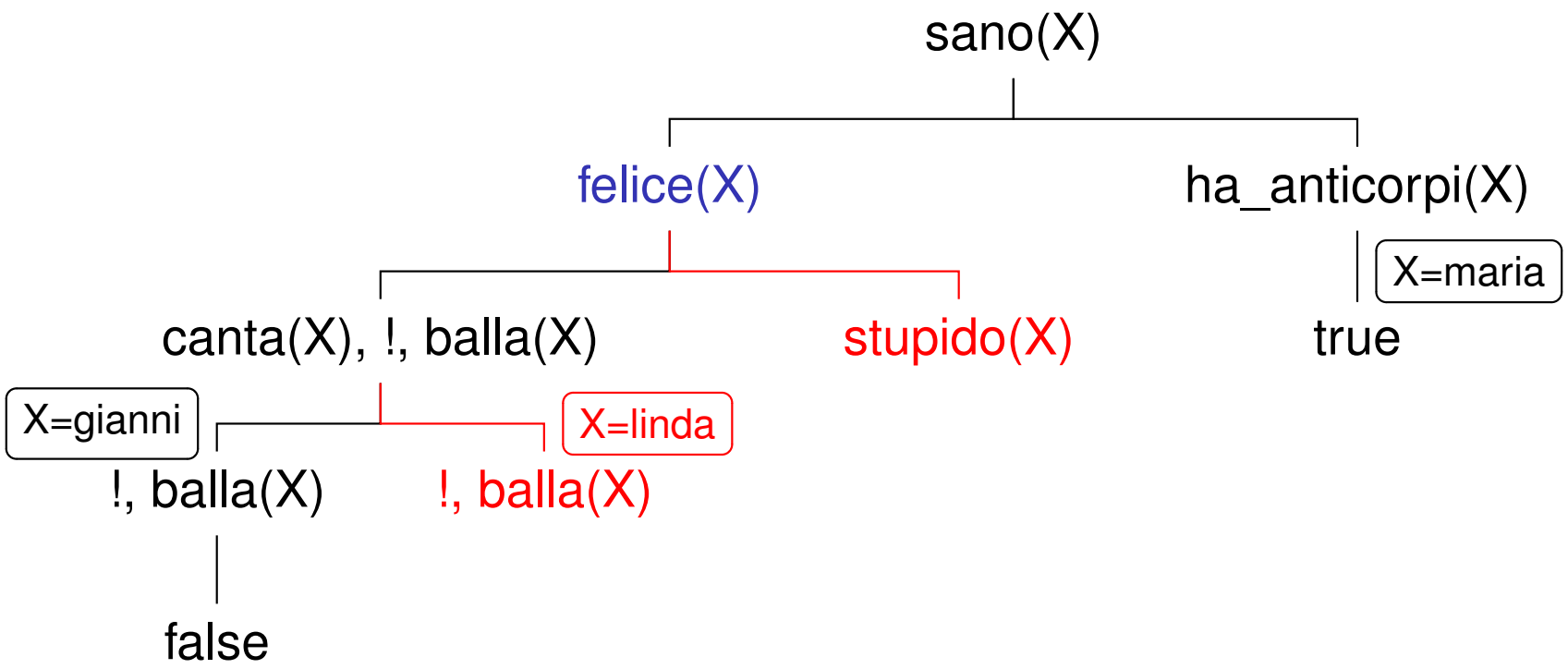
```
sano(X) :- felice(X).
sano(X) :- ha_anticorpi(X).
felice(X) :- canta(X), !, balla(X).
felice(X) :- stupido(X).
ha_anticorpi(maria).   balla(linda).
canta(linda).          canta(gianni).   stupido(gianni).
```




**felice(X)** è il parent goal.

# Cambiamo l'ordine di due fatti

```
sano(X) :- felice(X).
sano(X) :- ha_anticorpi(X).
felice(X) :- canta(X), !, balla(X).
felice(X) :- stupido(X).
ha_anticorpi(maria).    balla(linda).
canta(gianni).          canta(linda).      stupido(gianni).
```




**X=maria** è l'unica soluzione

# Cut verdi e cut rossi

- **Cut verde:** influenza l'efficienza ma non cambia il significato del programma (con o senza cut, le soluzioni sono le stesse)

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```

- **Cut rosso:** cambia il significato del programma

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

# Il cut e la logica

```
p :- a, b.
p :- c.
```

$$p \leftrightarrow (a \wedge b) \vee c$$

---

```
p :- a, !, b.
p :- c.
```

$$p \leftrightarrow (a \wedge b) \vee (NOT\ a \wedge c)$$

---

```
p :- c.
p :- a, !, b.
```

$$p \leftrightarrow (a \wedge b) \vee c$$

# Il cut e la logica

```
p :- a, b.
p :- c.
```

$$p \leftrightarrow (a \wedge b) \vee c$$

---

```
p :- a, !, b.
p :- c.
```

$$p \leftrightarrow (a \wedge b) \vee (NOT\ a \wedge c)$$

---

```
p :- c.
p :- a, !, b.
```

$$p \leftrightarrow (a \wedge b) \vee c$$

Ma attenzione: NOT non è la negazione logica.

**NOT a = il goal a non è dimostrabile dal programma**

# La combinazione cut & fail

**fail**: goal che fallisce immediatamente (inutile? Ma quando il Prolog fallisce, tenta il backtracking...)

Tutti i francesi, eccetto i parigini, sono gentili.

```
gentile(X) :- parigino(X), !, fail.
gentile(X) :- francese(X).

francese(antoine).
francese(charles).
parigino(antoine).

?- gentile(antoine).
false.

?- gentile(charles).
true.
```

Ma attenzione:

```
?- gentile(X).
false.
```

# La negazione come fallimento

La combinazione cut & fail sembra fornire qualche forma di negazione: la **negazione come fallimento**, che è definita così:

```
not(Goal) :- Goal, !, fail.
not(Goal).
```

Usando il **not**, possiamo modificare la definizione di gentile:

```
gentile(X) :- francese(X), not(parigino(X)).
```

```
?- gentile(X).
X = charles.
```

La negazione come fallimento si comporta come la negazione logica, a patto che, nel momento in cui il goal **not(G)** viene invocato, tutte le variabili in **G** siano legate.

```
gentile(X) :- not(parigino(X)), francese(X).
```

```
?- gentile(X).
false.
```

# Diversi usi del cut: cut verde

L'uso del "cut" consente di specificare situazioni deterministiche e di migliorare l'efficienza dei programmi: non si perde tempo a cercare di soddisfare goal dei quali si sa già che non contribuiranno alla soluzione

Inoltre: risparmio di memoria (non si conservano punti di backtracking)

Quando il cut non cambia il significato dichiarativo del programma: **cut verde**.

```
/* merge di liste ordinate */
merge(X,[],X).
merge([],X,X).
merge([X|R1],[Y|R2],[X,Y|R]) :-
        X=Y, !,
        merge(R1,R2,R).
merge([X|R1],[Y|R2],[X|R]) :-
        X<Y, !,
        merge(R1,[Y|R2],R).
merge([X|R1],[Y|R2],[Y|R]) :-
        X>Y,
        merge([X|R1],R2,R).
```

# Diversi usi del cut: cut rosso

In alcuni casi l'uso del "cut" è necessario per ottenere il funzionamento corretto dei programmi.

Il cut cambia il significato dichiarativo del programma: **cut rosso**.

Esempio: non è possibile definire l'intersezione insiemistica senza il cut o un costrutto derivato dal cut (not).

```
/* intersezione insiemistica */
intersect(_,[],[]).
intersect([],_,[]).
intersect([X|Rest],Y,[X|R]) :-
        member(X,Y), !,
        intersect(Rest,Y,R).
intersect([_|Rest],Y,R) :-
        intersect(Rest,Y,R).
```

Cut rosso: il cut è necessario per definire la relazione correttamente (dal punto di vista procedurale), e non solo per migliorare l'efficienza

# Cut verdi e cut rossi

**cut verdi:** migliorano l'efficienza; in caso di fallimento non si tentano altre alternative, che sarebbero comunque destinate a fallire

Un cut verde dice al sistema: "se sei arrivato fin qui, hai trovato e scelto l'unica regola corretta per soddisfare questo goal ed è inutile cercare alternative"

**cut rossi:** il significato procedurale non corrisponde a quello dichiarativo.

L'uso di cut rossi aumenta il potere espressivo del linguaggio, consentendo di specificare regole mutuamente esclusive.

Un cut rosso dice al sistema: "se sei arrivato fin qui, hai trovato e scelto la regola corretta per soddisfare questo goal" (conferma la scelta di una regola)

**La combinazione cut & fail:** "se sei arrivato fin qui, non devi più cercare di soddisfare questo goal"

```
/* different */
different(X,X) :- !, fail.
different(_,_).
```

# Il cut e la reversibilità dei predicati

Esempio: assumiamo di voler definire **concat** sapendo che verrà sempre utilizzato con i primi due argomenti come input:

concat(+L1,+L2,?Result)

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

?- concat([a,b],[c,d,e],X).
X = [a, b, c, d, e]
```

Se la prima lista è vuota, la prima regola è l'unica corretta

Ma il cut ha rovinato la reversibilità del predicato:

```
?- concat(X,Y,[a,b,c]), member(a,X).
false.

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

# Utilizzando il cut si deve aver presente come si intende utilizzare i predicati

number_of_parents(+X,-Y)

```
number_of_parents(adam,0) :- !.
number_of_parents(eve,0) :- !.
number_of_parents(_,2).

?- number_of_parents(adam,X).
X  = 0.

?- number_of_parents(adam,2).
true.
```

# Ancora l'intersezione

intersect(+Set1,+Set2,-Intersection)

```
/* intersezione insiemistica */
intersect(_,[],[]).
intersect([],_,[]).
intersect([X|Rest],Y,[X|R]) :-
        member(X,Y), !,
        intersect(Rest,Y,R).
intersect([_|Rest],Y,R) :-
        intersect(Rest,Y,R).

?- intersect([2],[2],[]).
true.
```

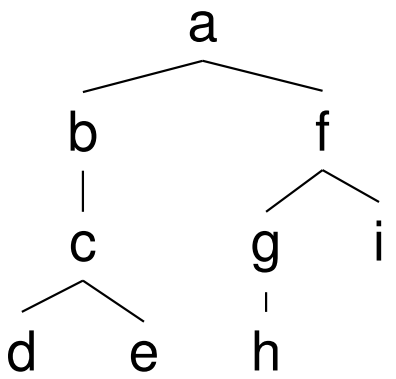
Per avere un comportamento (quasi) corretto anche quando il terzo argomento non è variabile, si dovrebbe modificare l'ultima clausola:

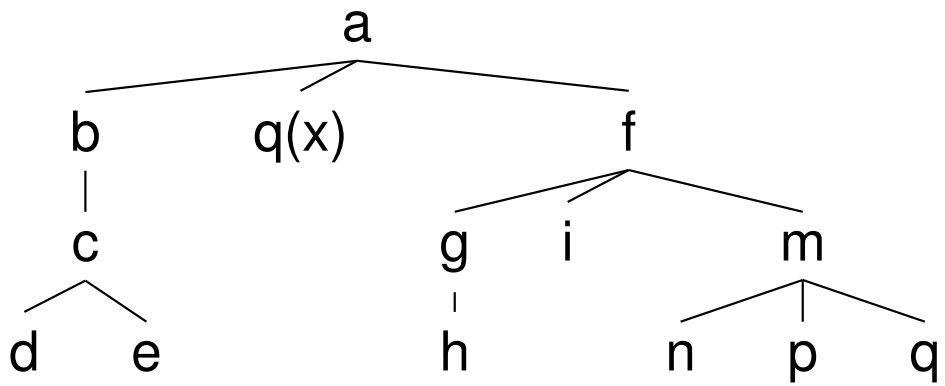
```
intersect([X|Rest],Y,R) :-
        not(member(X,Y)),
        intersect(Rest,Y,R).
```

# 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 => [b=>[c => [d,e]],
      f => [g => [h],
            i]]
```

```
a => [b=> [c => [d,e]],
      q(x),
      f => [g => [h],
            i,
            m=>[n,p,q]]]
```

# Test

- Test: un termine rappresenta un albero n-ario?

  **is_ntree(+Tree)**: vero se Tree rappresenta un albero
  **is_ntreelist(+Tlist)**: vero se Tlist e‘ una lista di termini rappresentanti alberi

```
is_ntree(_ => Tlist) :-
        !, length(Tlist,N),
        N>0, %% almeno un sottoalbero
        is_ntreelist(Tlist).
is_ntree(_).  % caso foglia

is_ntreelist([]).
is_ntreelist([T|Tlist]) :-
        is_ntree(T),
        is_ntreelist(Tlist).
```

- Test: un albero e’ una foglia?

  **is_leaf(+Tree)**: vero se Tree è la rappresentazione di una foglia

```
is_leaf(Leaf) :- Leaf \= _ => _.
```

# Selettori

- **root(+Tree,?Label)** true se Label è l'etichetta della radice di Tree

```
root(X => _, X) :- !.
root(X,X).
```

- **left(+Tree,?Left)** true se Left è il primo sottoalbero di Tree. Fallisce se Tree è una foglia.

```
left(_ => [Left|_], Left).
```

- **treerest(+Tree,?Rest)** true se Rest è l'albero che si ottiene da Tree eliminandone il primo sottoalbero. Fallisce se Tree è una foglia.

```
treerest(X=> [_], X).    % se c'è un unico sottoalbero
                         % si  ottiene una foglia
treerest(X =>[_|Rest], X=>Rest).
```

# Fattore di ramificazione: versione 1

Utilizzando la mutua ricorsione.

**branching_factor(+T,?N)** = N e' il fattore di ramificazione di T
**max_bf(+Tlist,?N)** = N e' il massimo tra i fattori
di ramificazione degli alberi in Tlist

```
branching_factor(_ => Tlist,B) :- !,
        length(Tlist,N),
        max_bf(Tlist,Blist),
        B is max(N,Blist).
branching_factor(_,0).

max_bf([T],B) :- !,
        branching_factor(T,B).
max_bf([T|Rest],B) :-
        branching_factor(T,BT),
        max_bf(Rest,Brest),
        B is max(BT,Brest).
```

# Fattore di ramificazione: versione 2

Utilizzando i selettori **left** e **treerest**

**branching_factor2(+T,?N)** = N e' il fattore di ramificazione di T

```
branching_factor2(T,B) :-
        T = _ => Tlist, !,
        left(T,Left),
        length(Tlist,N),
        branching_factor2(Left,BLeft),
        treerest(T,Rest),
        branching_factor2(Rest,BRest),
        max_list([N,BLeft,BRest],B).
branching_factor2(_,0).
```

**max_list(+List:list(number),-Max:number):**
True if Max is the largest number in List.
Fails if List is empty.

# Ricerca di un ramo mediante backtracking: versione 1

Con la mutua ricorsione

**ramo(+T,?Leaf,?Ramo)** = Ramo e' una lista che rappresenta un ramo di T dalla radice a una foglia etichettata da Leaf
**ramo_in_lista(+Tlist,?Leaf,?Ramo)** = Ramo e' un ramo di uno degli alberi in Tlist dalla radice a una foglia etichettata da Leaf

```
ramo(Root => Tlist,Leaf,[Root|Rest]) :- !,
        ramo_in_lista(Tlist,Leaf,Rest).
ramo(Leaf,Leaf,[Leaf]).

ramo_in_lista([T|_],Leaf,Ramo) :-
        ramo(T,Leaf,Ramo).
ramo_in_lista([_|Rest],Leaf,Ramo) :-
        ramo_in_lista(Rest,Leaf,Ramo).
```

Osservazione: se si escludesse la possibilità di avere nodi etichettati da liste, il predicato **ramo_in_lista** potrebbe anche chiamarsi semplicemente **ramo**.
Il Prolog non è a tipizzazione statica

# Ricerca di un ramo mediante backtracking: versione 2

Con l'uso dei selettori

```
ramo2(T,Leaf,[Root|Ramo]) :-
        left(T,Left),
        root(T,Root),
        ramo(Left,Leaf,Ramo).
ramo2(T,Leaf,Ramo) :-
        treerest(T,Rest), !,
        ramo(Rest,Leaf,Ramo).
ramo2(Leaf,Leaf,[Leaf]).
```

Il cut si potrebbe evitare sostituendo l'ultima clausola con

```
ramo2(Leaf,Leaf,[Leaf]) :- is_leaf(Leaf).
```