Il controllo del backtracking

```
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.
```

Il controllo del backtracking: il “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).
```

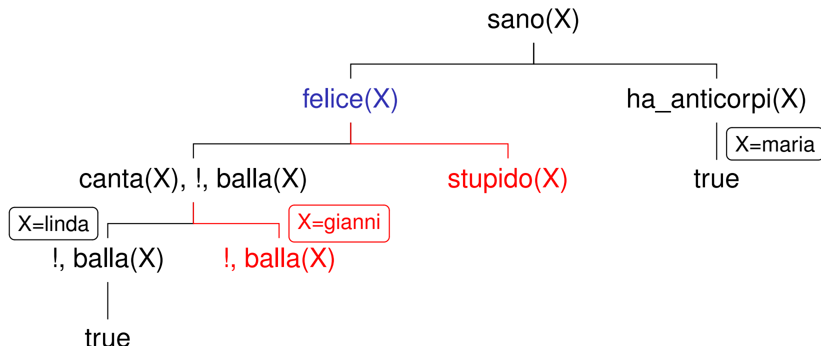
Il controllo del backtracking: il “cut”

```
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 = maria.
```

Non viene più data la soluzione `X = gianni`.

Perché Gianni non è sano

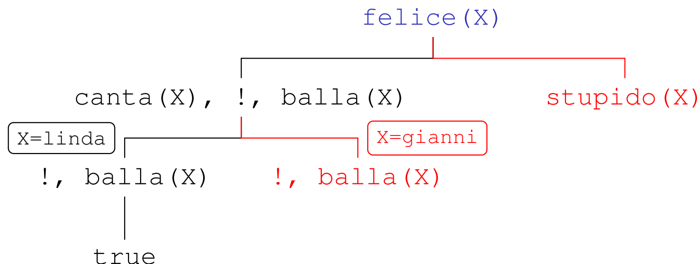
```
sano(X)  :- felice(X).
sano(X)  :- ha_anticorpi(X).
felice(X)  :- canta(X), !, balla(X).
felice(X)  :- stupido(X).
ha_anticorpi(maria).   balla(linda).
canta(linda).          canta(gianni).    stupido(gianni).
```



La potatura

felice(X) è il **parent goal**: quello che ha attivato la clausola con il cut.

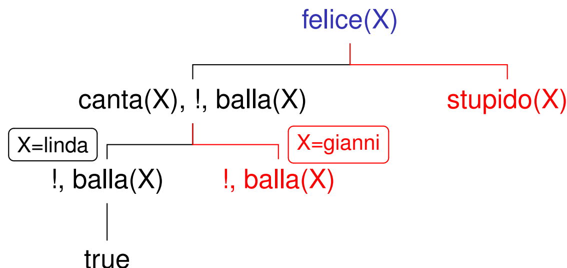
```
felice(X) :- canta(X), !, balla(X).
felice(X) :- stupido(X).
```



La potatura

felice(X) è il **parent goal**: quello che ha attivato la clausola con il cut.

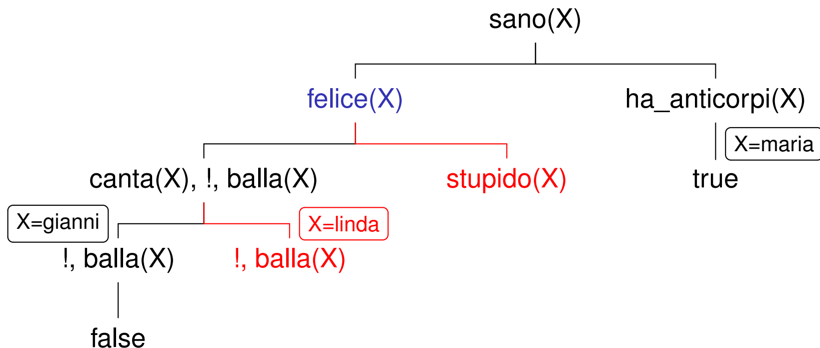
```
felice(X) :- canta(X), !, balla(X).
felice(X) :- stupido(X).
```



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

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

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**)

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** (il goal che ha utilizzato la clausola contenente il cut).

  Taglia le alternative lasciate aperte per il backtracking da quando il parent goal è stato unificato con la testa della regola utilizzata (inclusa la scelta di utilizzare quella clausola).

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** (il goal che ha utilizzato la clausola contenente il cut).

  Taglia le alternative lasciate aperte per il backtracking da quando il parent goal è stato unificato con la testa della regola utilizzata (inclusa la scelta di utilizzare quella clausola).
- Il cut può essere utilizzato per
  - evitare di perdere tempo ad esplorare strade “inutili”
  - evitare di esplorare strade che porterebbero a soluzioni non corrette

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** (il goal che ha utilizzato la clausola contenente il cut).

  Taglia le alternative lasciate aperte per il backtracking da quando il parent goal è stato unificato con la testa della regola utilizzata (inclusa la scelta di utilizzare quella clausola).
- Il cut può essere utilizzato per
  - evitare di perdere tempo ad esplorare strade “inutili”
  - evitare di esplorare strade che porterebbero a soluzioni non corrette
- Il cut può “rovinare” la reversibilità dei programmi.

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.
```

$p \leftrightarrow (a \land b) \lor c$

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

$p \leftrightarrow (a \land b) \lor (NOT\ a \land c)$

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

$p \leftrightarrow (a \land b) \lor c$

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.
```

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).
```