

# CONTROLLO DI UN PROGRAMMA

---

- Predicati predefiniti che consentono di influenzare e controllare il processo di esecuzione (dimostrazione) di un goal.
- PREDICATO CUT (!)
  - E' denotato dal simbolo !
  - E' uno dei più importanti e complessi predicati di controllo forniti da Prolog
- Per capire come funziona il predicato cut e' necessario vedere il modello run time di Prolog

# CONTROLLO DI UN PROGRAMMA

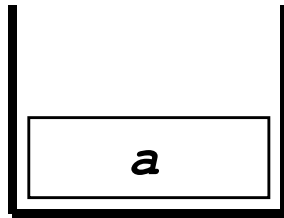
---

(c11)       $a :- p, b.$

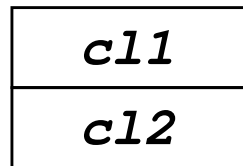
(c12)       $a :- p, c.$

(c13)       $p.$

- E la valutazione della query  $:-a.$



*Stack di esecuzione*



*Scelte per a*

*Scelta corrente*

# CONTROLLO DI UN PROGRAMMA

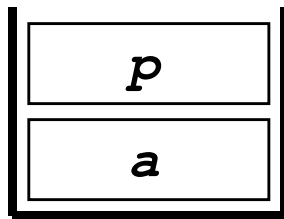
---

(c11)       $a :- p, b.$

(c12)       $a :- p, c.$

(c13)       $p.$

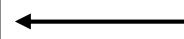
- E la valutazione della query  $:-a.$



*Stack di esecuzione*



*Scelte per a*



*Scelta corrente*

La valutazione di  $p$  ha successo

# CONTROLLO DI UN PROGRAMMA

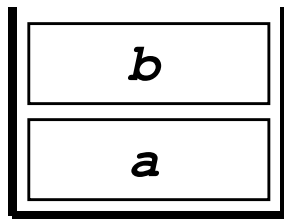
---

(c11)      a :- p, b.

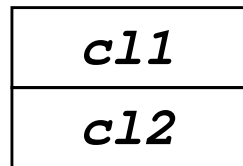
(c12)      a :- p, c.

(c13)      p.

- E la valutazione della query :-a.



*Stack di esecuzione*



*Scelte per a*

← *Scelta corrente*

La valutazione di b fallisce → viene attivato il meccanismo di backtracking

# CONTROLLO DI UN PROGRAMMA

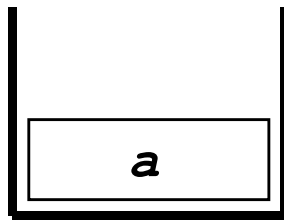
---

(c11)       $a :- p, b.$

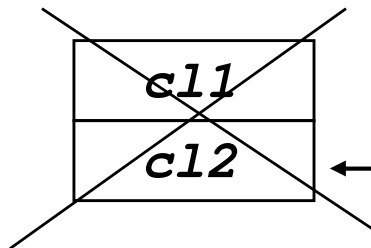
(c12)       $a :- p, c.$

(c13)       $p.$

- E la valutazione della query  $:-a.$



*Stack di esecuzione*



*Scelte per a*

*Scelta corrente*

# CONTROLLO DI UN PROGRAMMA

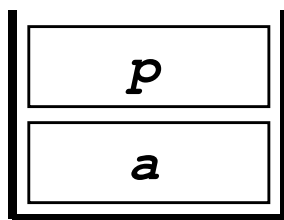
---

(c11)       $a :- p, b.$

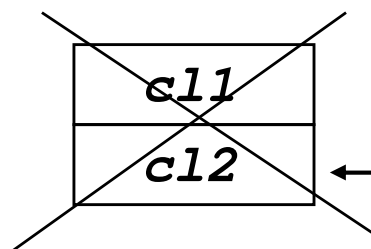
(c12)       $a :- p, c.$

(c13)       $p.$

- E la valutazione della query  $:-a.$



*Stack di esecuzione*



*Scelte per a*

← *Scelta corrente*

La valutazione di  $p$  ha successo

# CONTROLLO DI UN PROGRAMMA

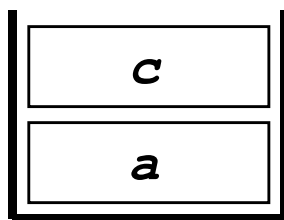
---

(c11)       $a :- p, b.$

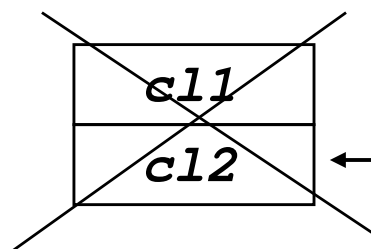
(c12)       $a :- p, c.$

(c13)       $p.$

- E la valutazione della query  $:-a.$



*Stack di esecuzione*



*Scelte per a*

← *Scelta corrente*

La valutazione di  $c$  fallisce → viene attivato il meccanismo di backtracking ma non ci sono più punti di scelta. Quindi si ha il fallimento di  $a$

# CONTROLLO DI UN PROGRAMMA

---

- Due stack:
  - Stack di esecuzione che contiene i record di attivazione delle varie procedure
  - Stack di backtracking che contiene l'insieme dei punti di scelta. Ad ogni fase della valutazione tale stack contiene puntatori alle scelte aperte nelle fasi precedenti della dimostrazione.

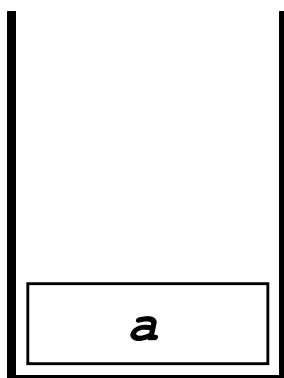


# CONTROLLO DI UN PROGRAMMA

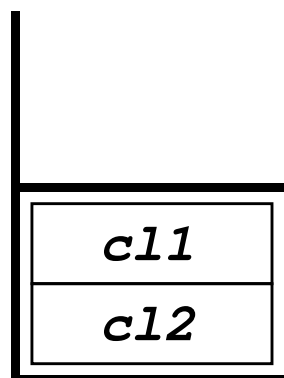
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a.



*Stack di esecuzione*



*Scelta corrente*

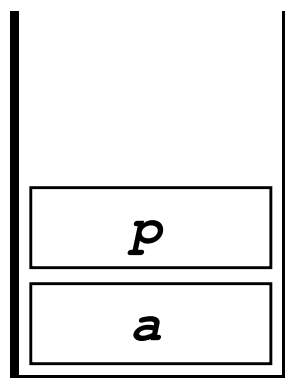
*Stack di backtracking*

# CONTROLLO DI UN PROGRAMMA

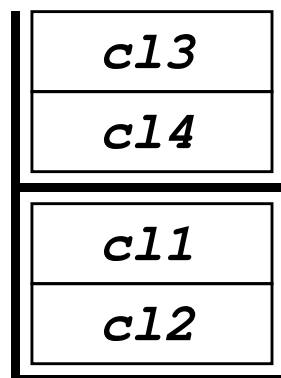
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a.



*Stack di esecuzione*



*Stack di backtracking*

← *Scelta corrente*

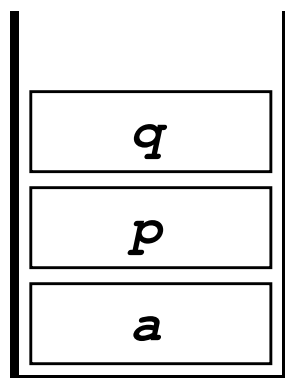
← *Scelta corrente*

# CONTROLLO DI UN PROGRAMMA

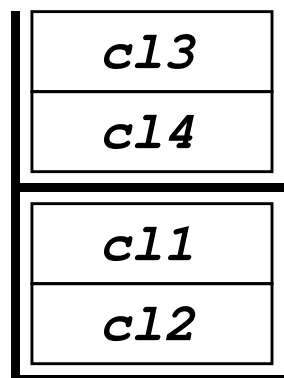
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a.



*Stack di esecuzione*



*Stack di backtracking*

← *Scelta corrente*

← *Scelta corrente*

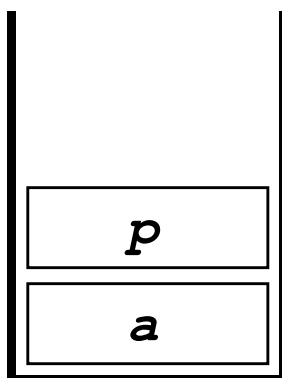
Fallimento

# CONTROLLO DI UN PROGRAMMA

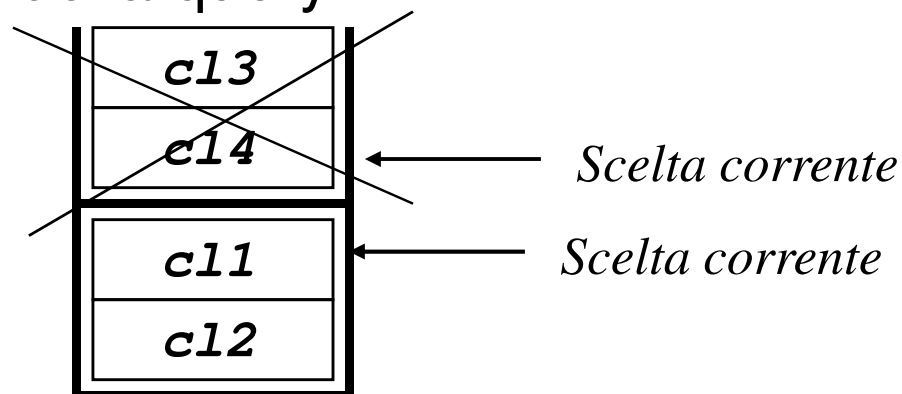
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a.



*Stack di esecuzione*



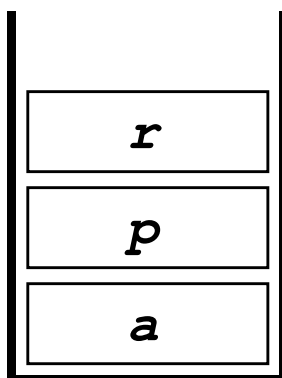
*Stack di backtracking*

# CONTROLLO DI UN PROGRAMMA

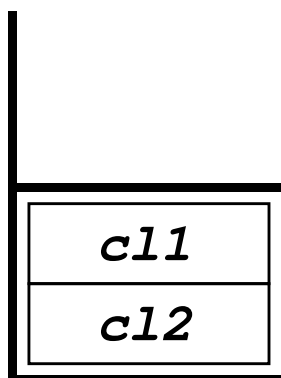
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a .



*Stack di esecuzione*



*Stack di backtracking*

← *Scelta corrente*

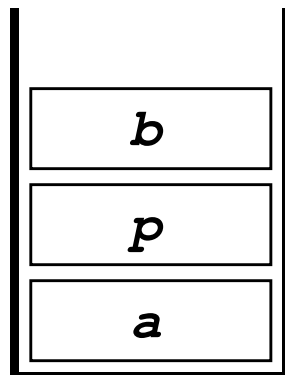
**r** ha successo

# CONTROLLO DI UN PROGRAMMA

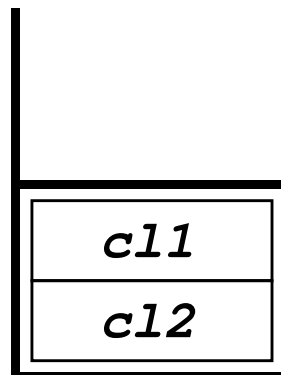
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a .



*Stack di esecuzione*



*Stack di backtracking*

← *Scelta corrente*

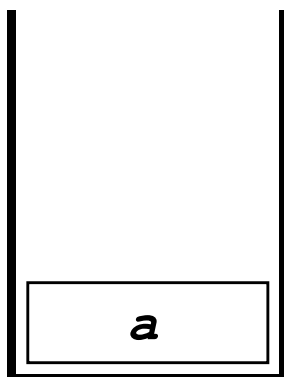
**b** fallisce

# CONTROLLO DI UN PROGRAMMA

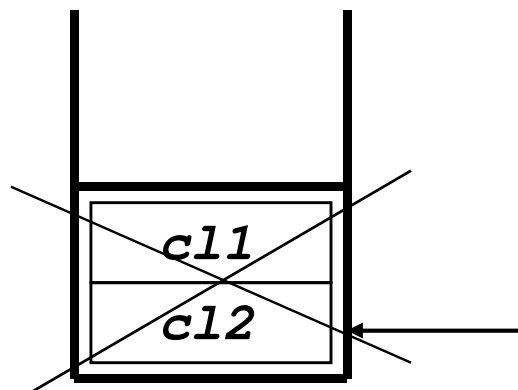
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a.



*Stack di esecuzione*



*Stack di backtracking*

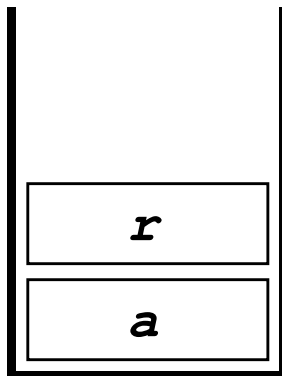
*Scelta corrente*

# CONTROLLO DI UN PROGRAMMA

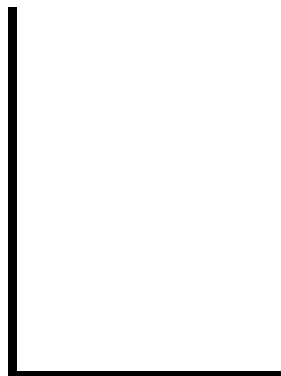
---

(c11)      a :- p, b.  
(c12)      a :- r.  
(c13)      p :- q.  
(c14)      p :- r.  
(c15)      r.

- E la valutazione della query :-a.



*Stack di esecuzione*



*Stack di backtracking*

**Successo**



# EFFETTO DEL CUT

---

- L'effetto del cut e' quello di rendere definitive alcune scelte fatte nel corso della valutazione dall'interprete Prolog ossia quello di eliminare alcuni blocchi dallo stack di backtracking
- Il cut altera quindi il controllo del programma
- Effetto collaterale piu' importante: perdita di dichiarativita'

## EFFETTO DEL CUT

---

- Si consideri la clausola:

$p :- q_1, q_2, \dots, q_i, !, q_{i+1}, q_{i+2}, \dots, q_n.$

l'effetto della valutazione del goal ! (cut) durante la dimostrazione del goal "p" è il seguente:

- la valutazione di ! ha successo (come quasi tutti i predicati predefiniti) e ! viene ignorato in fase di backtracking;
- tutte le scelte fatte nella valutazione dei goal  $q_1, q_2, \dots, q_i$  e in quella del goal  $p$  vengono rese definitive; in altri termini, tutti i punti di scelta per tali goal (per le istanze di tali goal utilizzate) vengono rimossi dallo stack di backtracking.
- Le alternative riguardanti i goal seguenti al cut non vengono modificate

## EFFETTO DEL CUT

---

- Si consideri la clausola:

$p \text{ :- } q_1, q_2, \dots, q_i, !, q_{i+1}, q_{i+2}, \dots, q_n.$

- Se la valutazione di  $q_{i+1}, q_{i+2}, \dots, q_n$  fallisce, fallisce tutta la valutazione di  $p$ . Infatti, anche se  $p$  o  $q_1, q_2, \dots, q_i$  avessero punti di scelta questi sarebbero eliminati dal cut;
- Il cut taglia rami dell'albero SLD

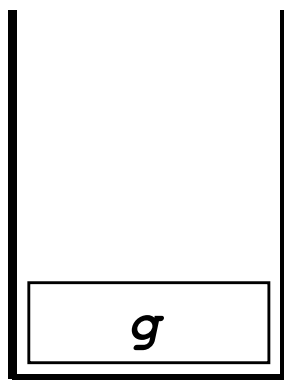
Pertanto il cut non può essere definito in modo dichiarativo

# EFFETTO DEL CUT

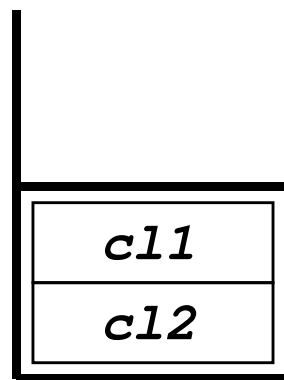
---

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



*Stack di esecuzione*



*Stack di backtracking*

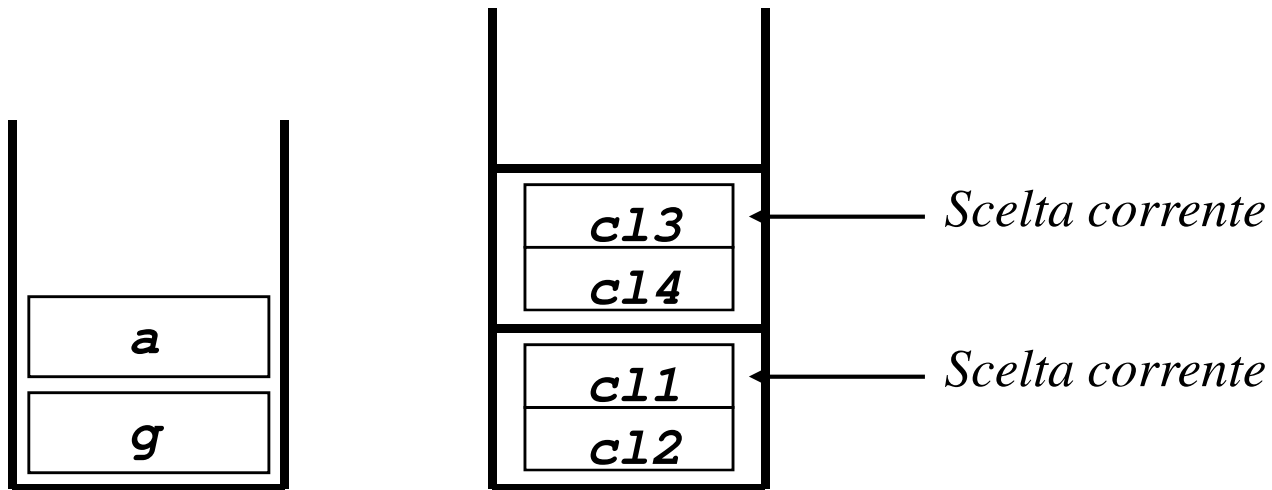
*Scelta corrente*

# EFFETTO DEL CUT

---

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



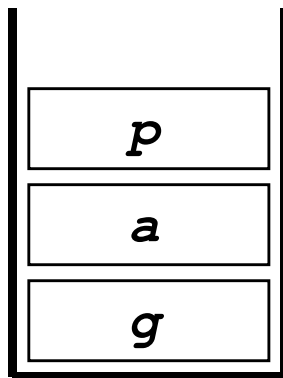
*Stack di esecuzione*

*Stack di backtracking*

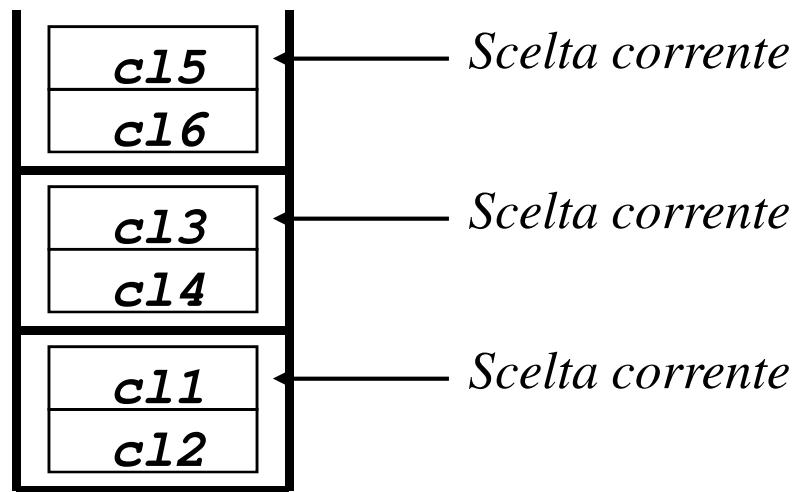
# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



*Stack di esecuzione*

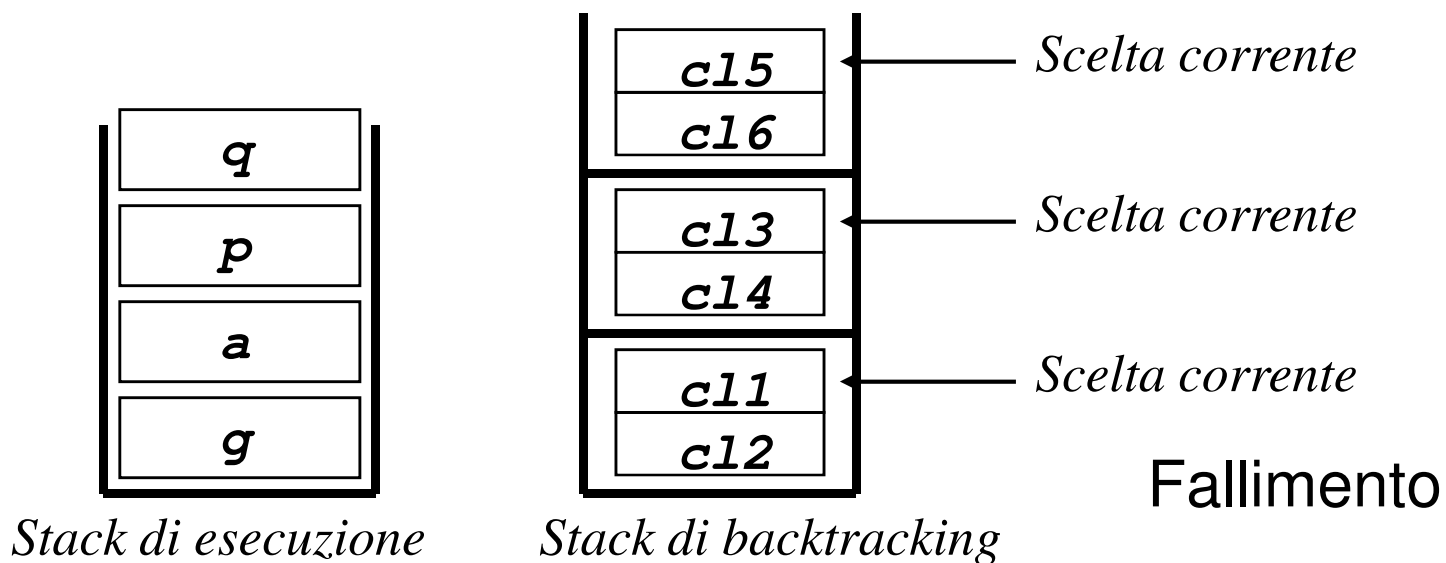


*Stack di backtracking*

# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

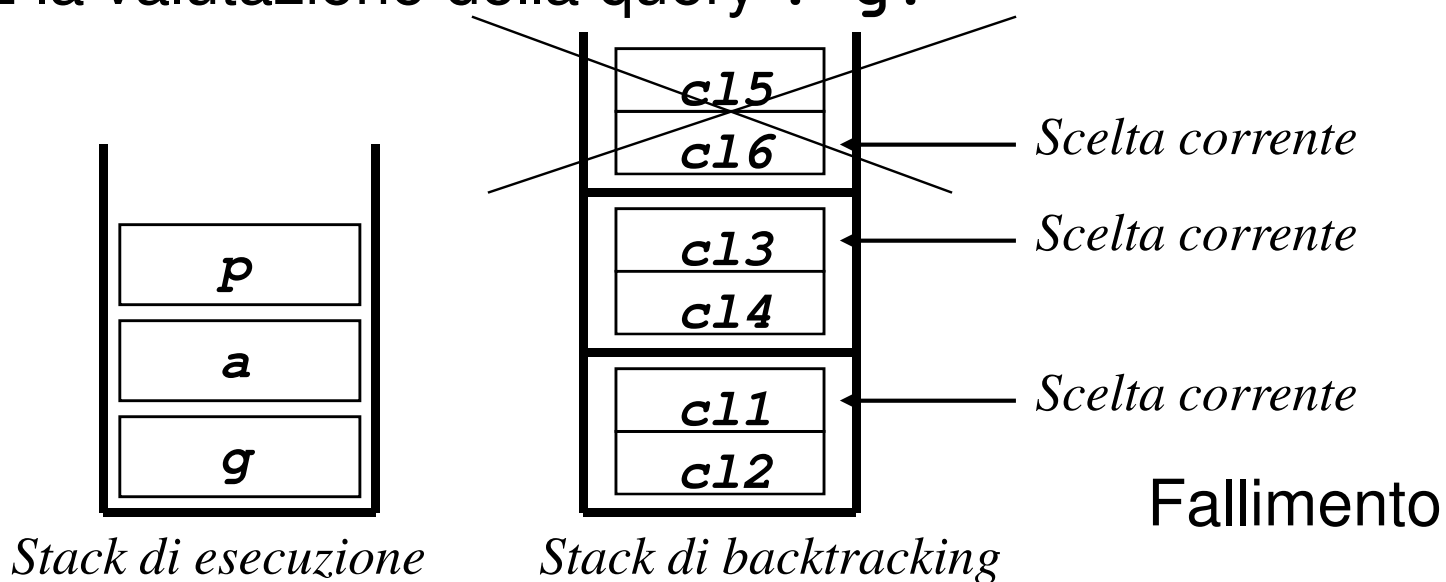
- E la valutazione della query  $:-g$ .



# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query `:-g.`

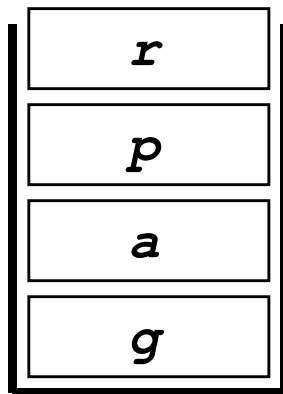




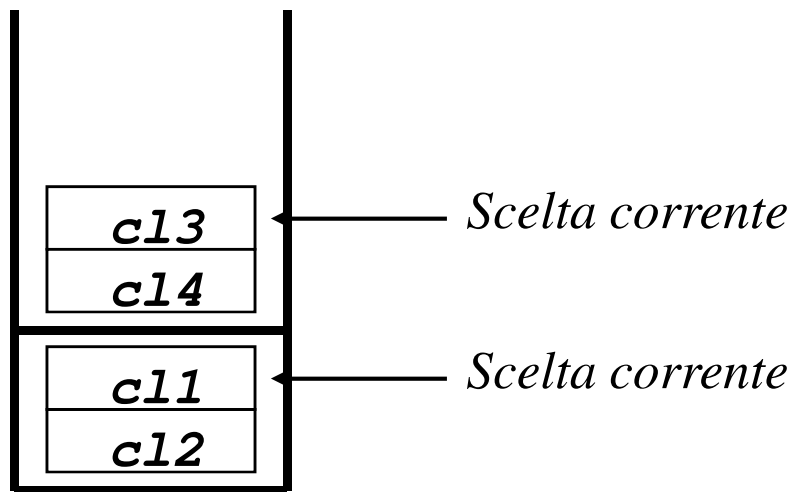
# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



*Stack di esecuzione*

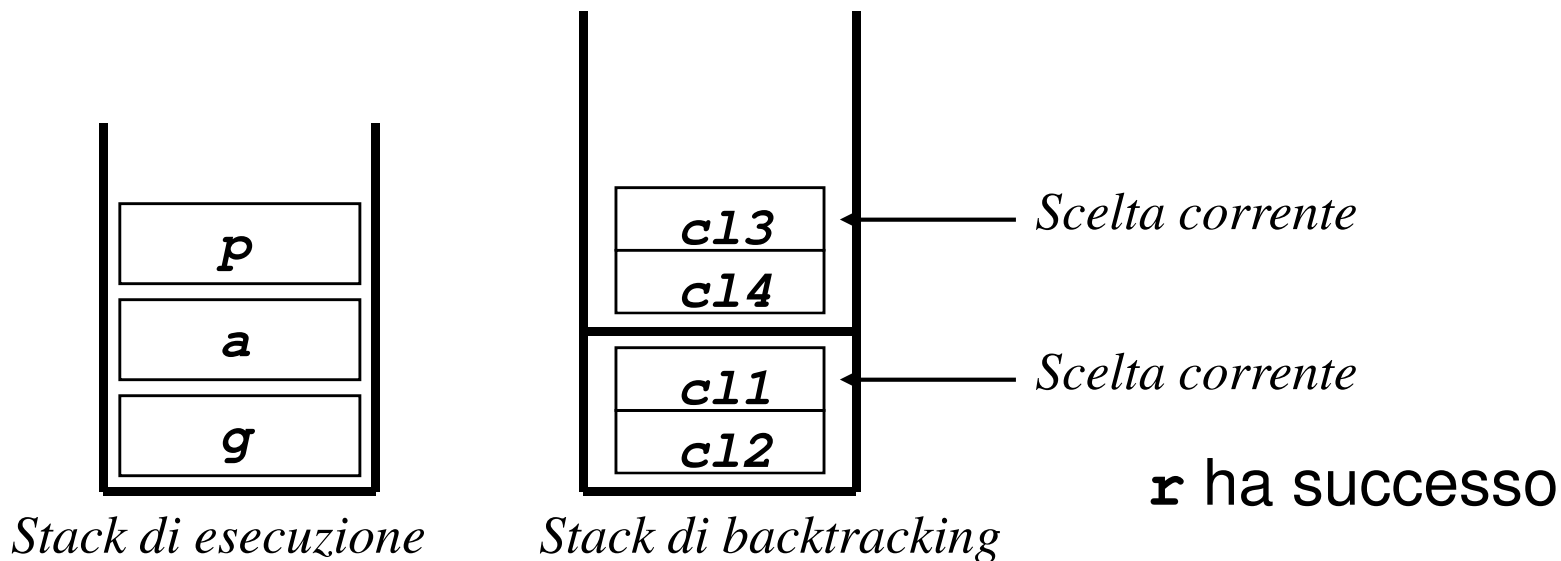


*Stack di backtracking*

# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

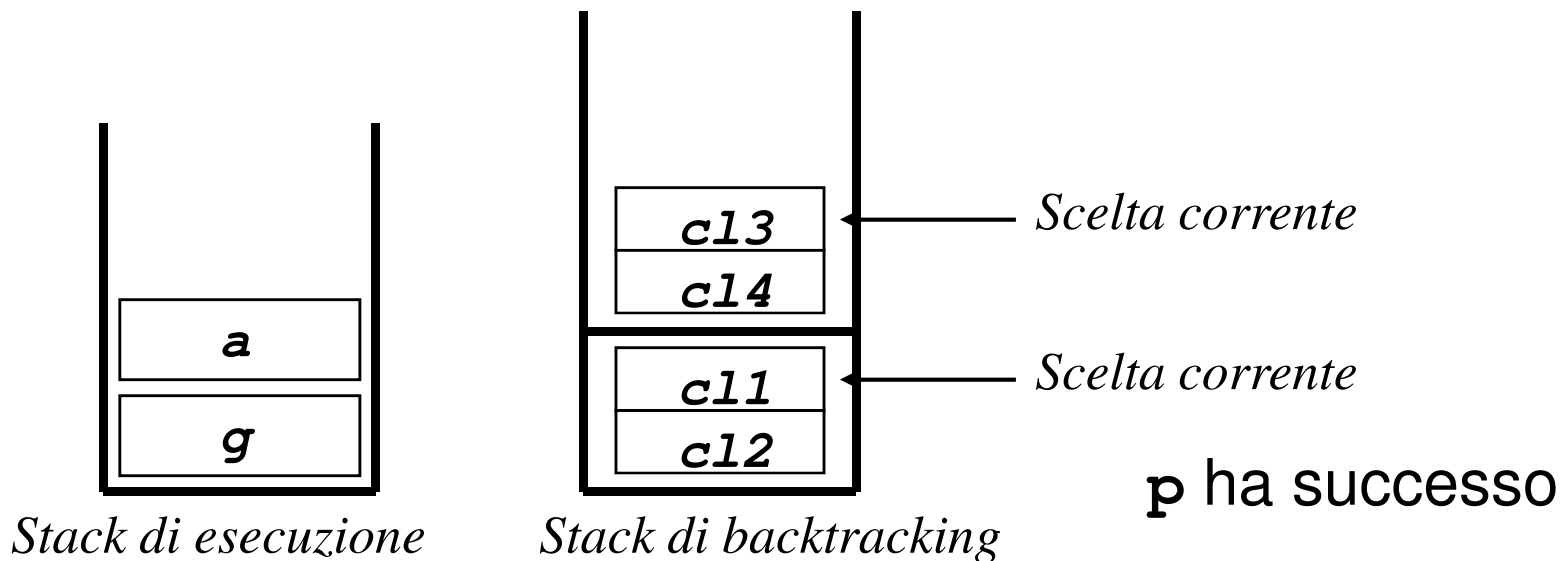
- E la valutazione della query `:-g`.



# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.

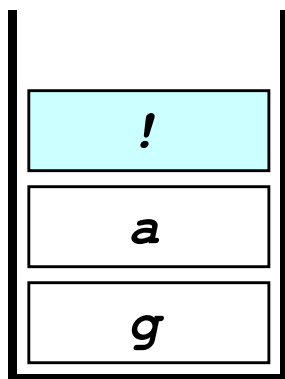


# EFFETTO DEL CUT

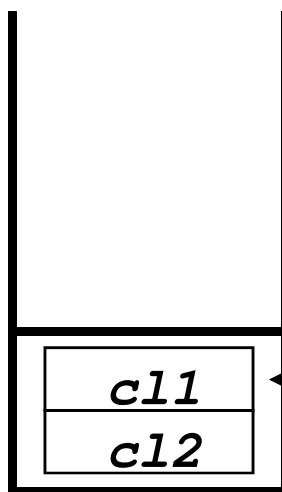
```
(c11)    g :- a.  
(c12)    g :- s.  
(c13)    a :- p,!,b.  
(c14)    a :- r.  
(c15)    p :- q.  
(c16)    p :- r.  
(c17)    r.
```

- E la valutazione della query :-g.

**Effetto del !**  
Tutti i punti di scelta per **p** e per **a** sono rimossi dallo stack  
Il cut ha successo



*Stack di esecuzione*



*Stack di backtracking*

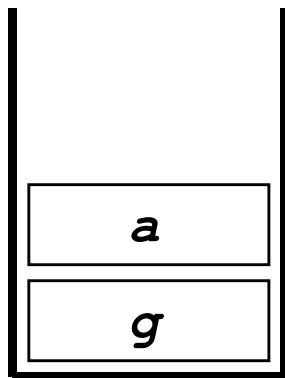
*Scelta corrente*

# EFFETTO DEL CUT

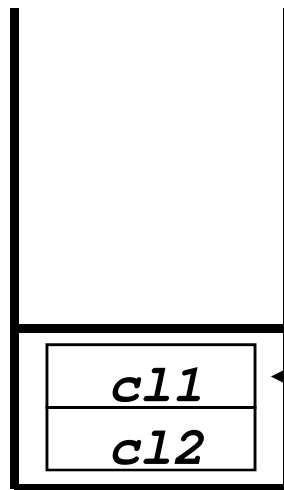
---

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



*Stack di esecuzione*



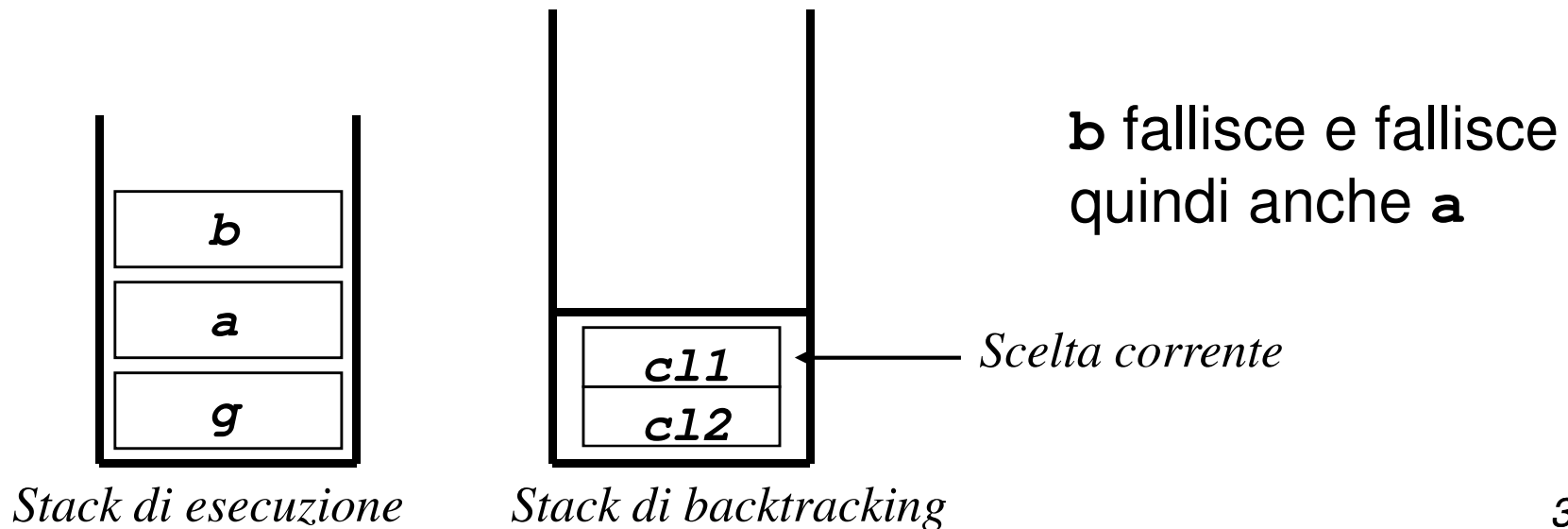
*Stack di backtracking*

*Scelta corrente*

# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

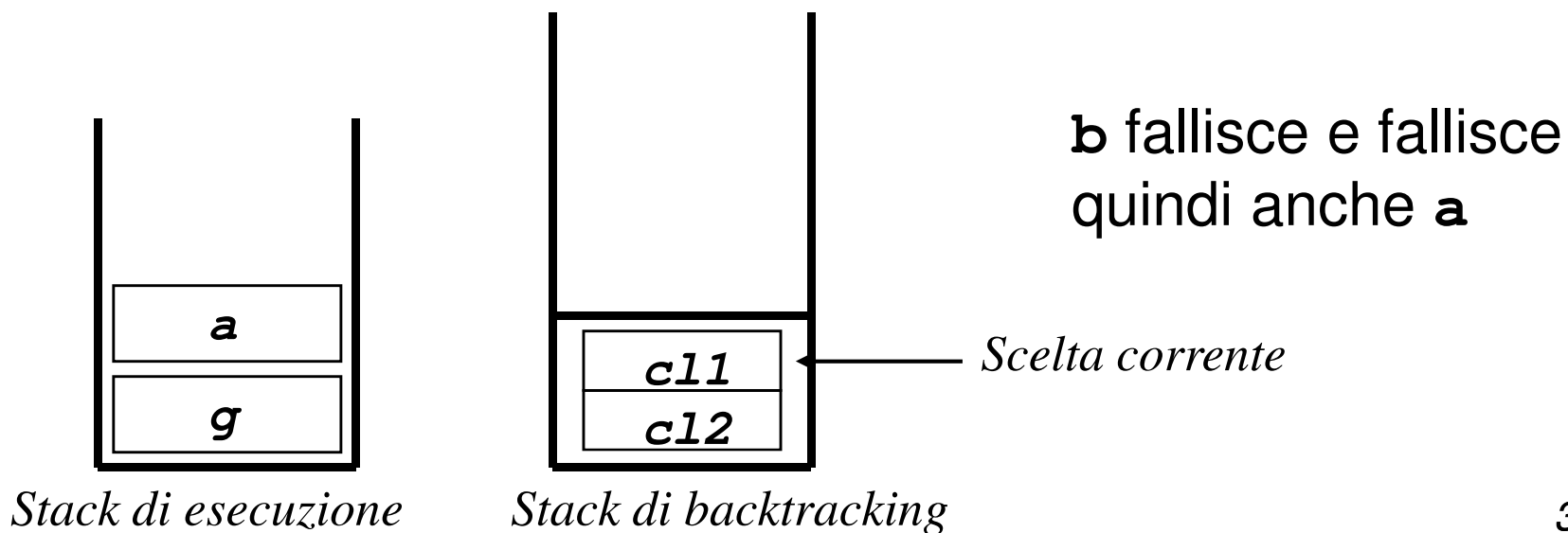
- E la valutazione della query `:-g`.



# EFFETTO DEL CUT

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query `:-g`.

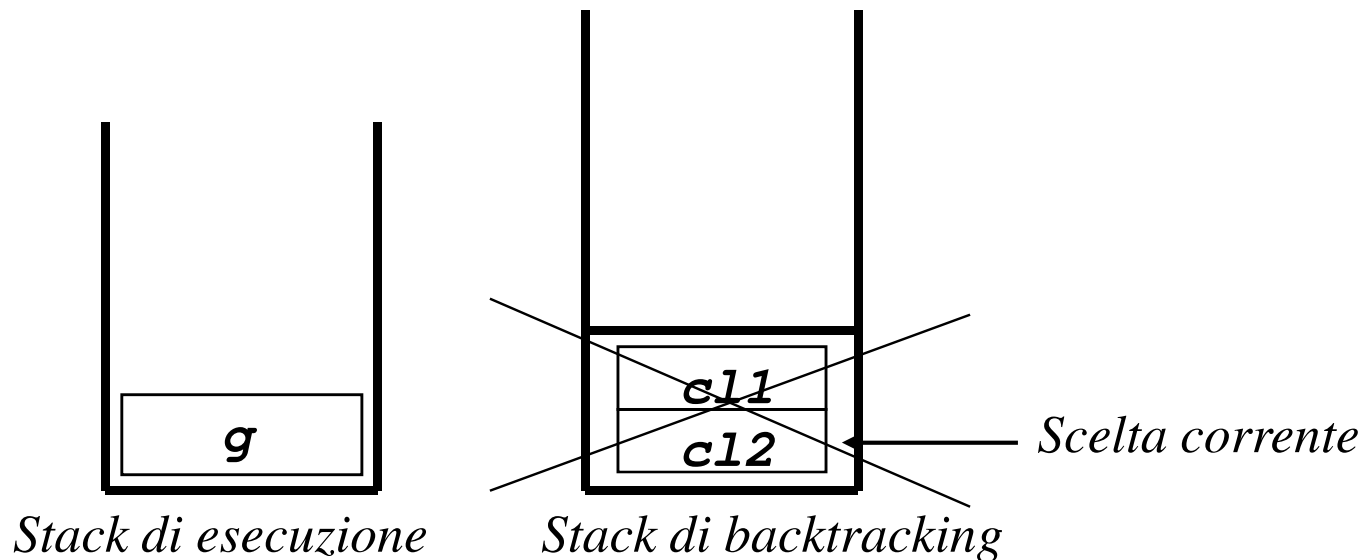


# EFFETTO DEL CUT

---

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



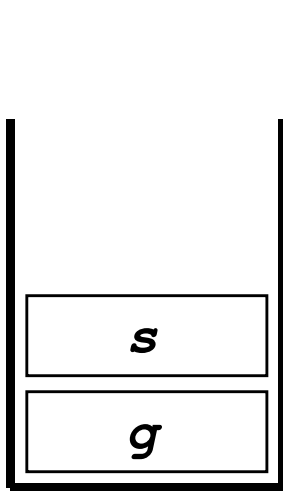


# EFFETTO DEL CUT

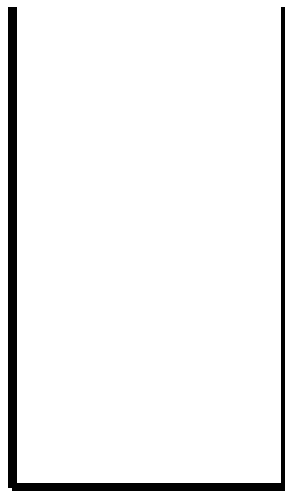
---

```
(c11)      g :- a.  
(c12)      g :- s.  
(c13)      a :- p,!,b.  
(c14)      a :- r.  
(c15)      p :- q.  
(c16)      p :- r.  
(c17)      r.
```

- E la valutazione della query :-g.



*Stack di esecuzione*



*Stack di backtracking*

Fallimento !!!

Senza il cut la query  
avrebbe avuto  
successo

# ESEMPIO

---

`a(X, Y) :- b(X), !, c(Y) .`

`a(0, 0) .`

`b(1) .`

`b(2) .`

`c(1) .`

`c(2) .`

`:- a(X, Y) .`

`yes X=1 Y=1;`

`X=1 Y=2;`

`no`

## ESEMPIO

---

`p(X) :- q(X), r(X).`

`q(1).`

`q(2).`

`r(2).`

`:- p(X).`

`yes X=2`

`p(X) :- q(X), !, r(X).`

`q(1).`

`q(2).`

`r(2).`

`:- p(X).`

`no`

# CUT

---

- La perdita della dichiaratività è il maggiore svantaggio derivante dall'uso del "cut".
- Tuttavia l'uso del "cut" è necessario per la correttezza di alcune classi di programmi ed è utile per l'efficienza di altre classi di programmi.

# MUTUA ESCLUSIONE TRA CLAUSOLE

---

- Il cut può essere utilizzato molto semplicemente per rendere deterministica la scelta tra due o più clausole alternative

`p(X) :- a(X), b.`

`p(X) :- c.`

- Si supponga che la condizione `a(X)` debba rendere le due clausole mutuamente esclusive per realizzare uno schema del tipo:

`if a(.) then b else c`

# MUTUA ESCLUSIONE TRA CLAUSOLE

---

- Si supponga che la condizione  $a(x)$  debba rendere le due clausole mutuamente esclusive per realizzare uno schema del tipo:

`if a(.) then b else c`

- Utilizzando il predicato predefinito "cut":

`p(x) :- a(x), !, b.`

`p(x) :- c.`

**ATTENZIONE:** la mancanza del cut rende il programma **SCORRETTO**

*Se  $a(x)$  e' vera, viene valutato il cut che toglie il punto di scelta per  $p(x)$ . Se invece  $a(x)$  fallisce, si innesca il backtracking prima che il cut venga eseguito.*

## ESEMPIO: INTERSEZIONE DI INSIEMI

---

- Riprendiamo l'esempio dell'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).
```

- La seconda e la terza clausola **devono essere mutuamente esclusive**

```
:- intersection([1,2,3], [2,3,4], S).
```

```
yes    S=[2,3];
```

```
       S=[2];
```

```
       S=[3];
```

```
       S=[]
```

*Risposte **scorrette** a causa delle non mutua esclusione tra la seconda e la terza clausola*

## ESEMPIO: INTERSEZIONE DI INSIEMI

---

- La condizione che determina la mutua esclusione e' `member(H, S2)` quindi il cut va inserito dopo tale condizione

```
intersection([], S2, []).  
intersection([H|T], S2, [H|T3]) :- member(H, S2), !,  
                                   intersection(T, S2, T3).  
intersection([H|T], S2, S3) :- intersection(T, S2, S3).
```

- La seconda e la terza clausola sono **mutuamente esclusive**

```
:- intersection([1,2,3], [2,3,4], S).  
   yes      S=[2,3];
```



## RIMOZIONE DI ELEMENTI DA LISTE

---

- Cancellazione di un elemento uguale a T dalla lista  
(c11) `delete1(T, [], []).`  
(c12) `delete1(T, [T|TAIL], TAIL).`  
(c13) `delete1(T, [HEAD|TAIL], [HEAD|L]) :-`  
`delete1(T, TAIL, L).`
- Cancellazione di tutti gli elementi uguale a T dalla 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).`

## RIMOZIONE DI ELEMENTI DA LISTE

---

- Cancellazione di un elemento uguale a T dalla lista: le clausole (c12) e (c13) devono essere mutuamente esclusive.
- La condizione di mutua esclusione e' l'unificazione dell'elemento da cancellare con la testa della lista

```
(c11) delete1(T, [], []).
```

```
(c12') delete1(T, [T|TAIL], TAIL) :- !.
```

```
(c13) delete1(T, [HEAD|TAIL], [HEAD|L]) :-  
      delete1(T, TAIL, L).
```

## RIMOZIONE DI ELEMENTI DA LISTE

---

- Cancellazione di tutti gli elementi uguali a T dalla lista: le clausole (c15) e (c16) devono essere mutuamente esclusive.
- La condizione di mutua esclusione e' 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).
```

## EFFICIENZA

---

- La presenza del "cut" rende in molti casi un programma ricorsivo deterministico e consente l'applicazione dell'ottimizzazione della ricorsione tail.
- Merge di due liste ordinate di numeri interi in una nuova lista ordinata.

```
merge([], L2, L2) .
```

```
merge(L1, [], L1) .
```

```
merge([X|REST1], [X|REST2], [X,X|REST]) :-
```

```
    merge(REST1, REST2, REST) .
```

```
merge([X|REST1], [Y|REST2], [X|REST]) :- X < Y,
```

```
    merge(REST1, [Y|REST2], REST) .
```

```
merge([X|REST1], [Y|REST2], [Y|REST]) :- X > Y,
```

```
    merge([X|REST1], REST2, REST) .
```

## EFFICIENZA

---

- Sebbene merge sia definita in modo tail ricorsivo, la presenza dei punti di scelta rende l'ottimizzazione impossibile
- Inserendo il cut il programma cambia così.

```
merge([], L2, L2) .
merge(L1, [], L1) .
merge([X|REST1], [X|REST2], [X,X|REST]) :- !,
    merge(REST1, REST2, REST) .
merge([X|REST1], [Y|REST2], [X|REST]) :- X < Y,
    !,
    merge(REST1, [Y|REST2], REST) .
merge([X|REST1], [Y|REST2], [Y|REST]) :- X > Y,
    merge([X|REST1], REST2, REST) .
```

# EFFICIENZA

---

- Abbiamo visto il predicato **member**

```
member(E1, [E1|_]).
```

```
member(E1, [_|Tail):- member(E1, Tail) .
```

- Se abbiamo bisogno di interpretare tale predicato solo per la verifica di appartenenza di un elemento a una lista, possiamo inserire un cut per migliorare l'efficienza

```
member(E1, [E1|_]) :- !.
```

```
member(E1, [_|Tail):- member(E1, Tail) .
```

- In questo caso pero' non e' possibile usare il predicato **member** per
  - individuare tutti gli elementi di una lista
  - verificare l'appartenenza multipla di un elemento alla lista