Linguaggi di Programmazione 2022-2023

Prolog e Programmazione Logica III

Marco Antoniotti
Gabriella Pasi
Rafael Peñaloza

Modello di esecuzione Prolog

```
p :- q, r.
```

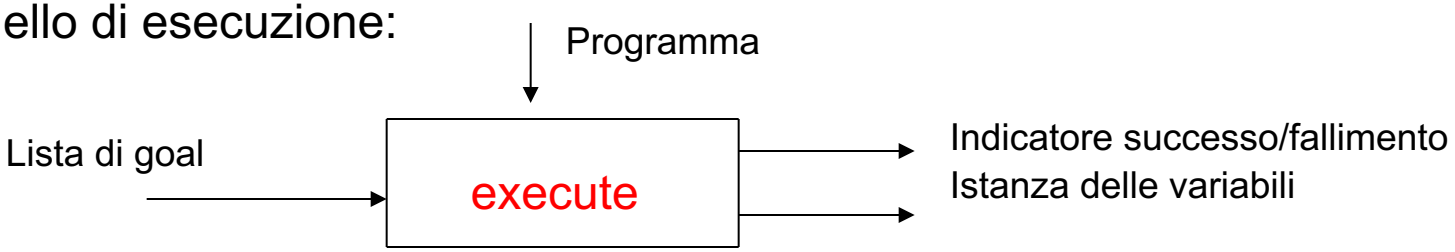
Interpretazione dichiarativa:

`p` è vero se sono veri `q` e `r`.

Interpretazione procedurale:

Il problema `p` può essere scomposto nei sottoproblemi `q` e `r`.

Modello di esecuzione:



Un goal può essere visto come una chiamata ad una procedura.

Una regola può essere vista come la definizione di una procedura in cui la testa è l'intestazione mentre la parte destra è il corpo.

Estensioni

- Per rendere il Prolog un linguaggio effettivamente utilizzabile vengono aggiunti
  - Già introdotti
    - Notazione per le liste
    - Meccanismi per il caricamento del codice Prolog
  - Da vedere
    - Meccanismi di controllo del backtracking
    - Operazioni aritmetiche
    - Trattamento della negazione
    - Possibilità di manipolare e confrontare le strutture dei termini
    - Predicati meta-logici ed extra-logici
    - Predicati di input/output
    - Meccanismi per modificare/accedere alla base di conoscenza

Il controllo di esecuzione di un programma

- Come abbiamo intuito, le clausole nel data base di un programma Prolog vengono considerate “da sinistra, verso destra” e “dall’alto al basso”

- Se un (sotto)goal fallisce, allora il dimostratore Prolog sceglie un’alternativa, scandendo “dall’alto” verso “il basso” la lista delle clausole

- Il Prolog mette a disposizione un predicato speciale, chiamato **cut** (*taglio*, scritto con il solo simbolo esclamativo `!`) per controllare questa sequenza di scelte
  - Il cut è molto complesso da interpretare (non ha un’interpretazione “logica” ma solo procedurale)
  - La sua importanza per il Prolog non può essere sottovalutata
  - Per capire come funziona è necessario avere un’idea più approfondita del funzionamento del dimostratore Prolog (ovvero della sua “macchina virtuale”)

Il predicato cut '!'

- Consideriamo la seguente clausola generica con cut

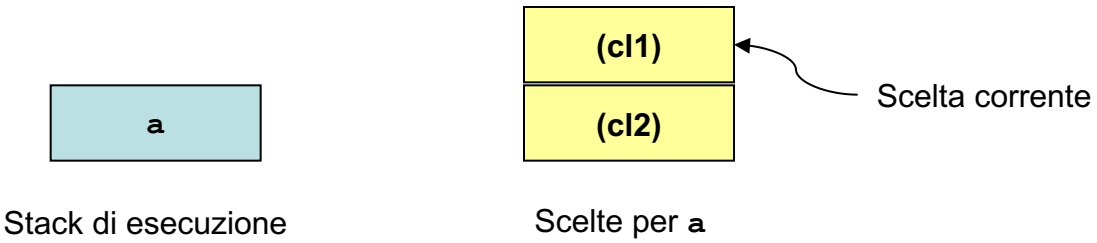
  C = **a :- b_1, b_2, …, b_k, !, b_{k+1}, …, b_n.**

- L'effetto del cut è il seguente
  - Se il goal corrente **G** unifica con **a** e **b_1**, …, **b_k** hanno successo, allora il dimostratore si impegna inderogabilmente alla scelta di C per dimostrare **G**
  - Ogni clausola alternativa (successiva, in basso) per **a** che unifica con **G** viene ignorata
  - Se un qualche **b_j** con **j > k** fallisse, il backtracking si fermerebbe al cut **!**
    - Le altre scelte per i **b_i** con **i ≤ k** sono di conseguenza rimosse dall'albero di derivazioni
  - Quando il backtracking raggiunge il cut, allora il cut fallisce e la ricerca procede dall'ultimo punto di scelta prima che **G** scegliesse C

Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

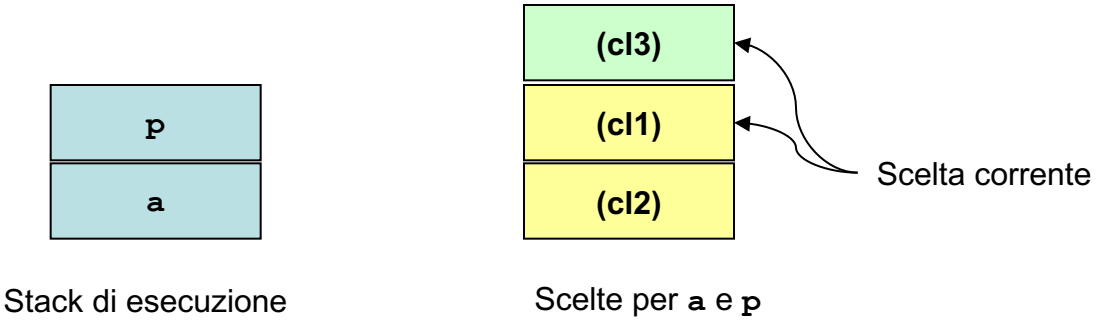
- Considerate la valutazione della query
  `?- a.`
- Lo stato interno del sistema Prolog diventa il seguente



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

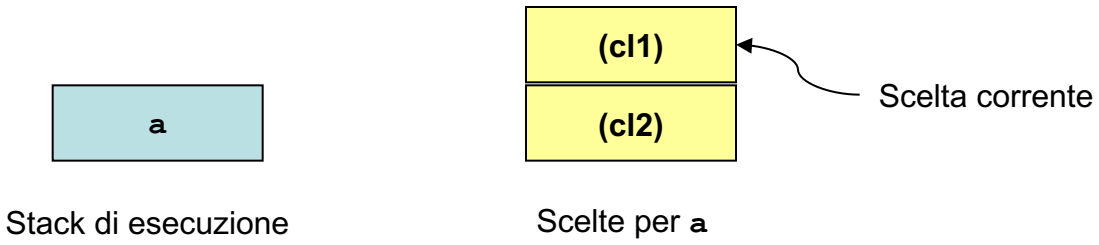
- Si mette `p` in cima allo stack



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- La valutazione di `p` ha successo

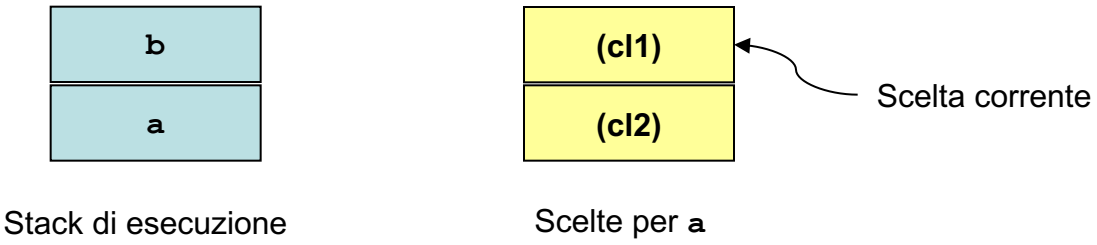


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- Quindi si inserisce `b` in cima allo stack

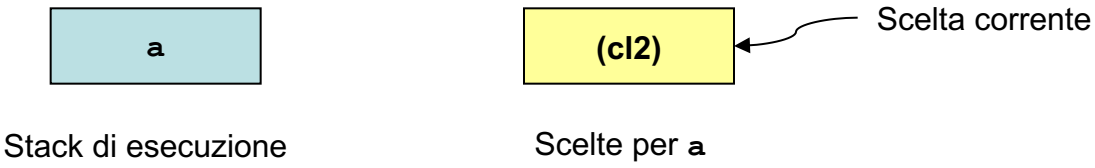
La valutazione di `b` fallisce; quindi viene attivato il meccanismo di **backtracking**



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

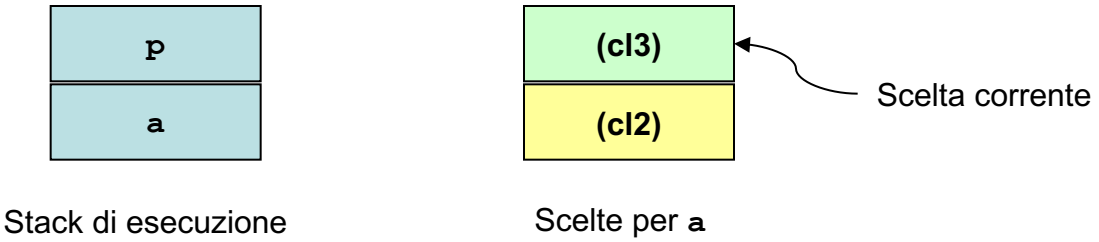
- Per proseguire con la valutazione di `?- a.` si passa a considerare la seconda clausola
- Lo stato interno del sistema Prolog diventa il seguente



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

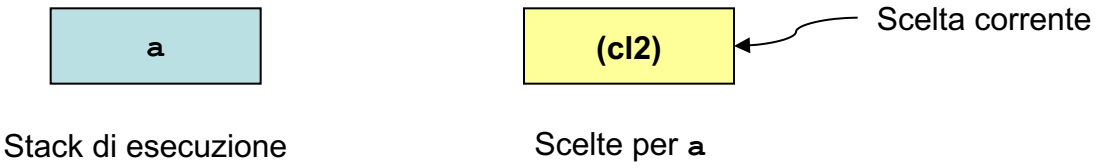
- Si mette `p` in cima allo stack



Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- La valutazione di `p` ha successo

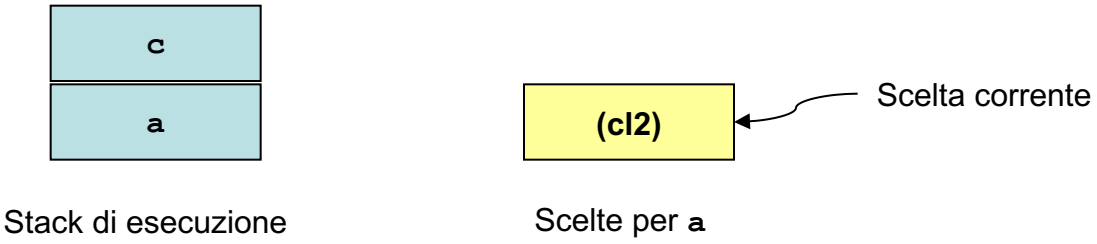


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog
  (cl1) `a :- p, b.`
  (cl2) `a :- p, c.`
  (cl3) `p.`

- Quindi si inserisce `c` in cima allo stack

La valutazione di `c` fallisce; quindi viene attivato il meccanismo di **backtracking**; ma visto che non ci sono più clausole anche `a` fallisce e quindi lo stack di esecuzione si svuota



Modello di Esecuzione di un Programma Prolog

- Due pile (stacks):

1. Pila di esecuzione che contiene i record di attivazione delle varie “procedure” (ovvero le sostituzioni per l’unificazione delle varie regole)

2. Pila di **backtracking** che contiene l’insieme dei “punti di scelta”; ad ogni fase della valutazione questa pila contiene dei “puntatori” alle scelte *aperte* nelle fasi precedenti della dimostrazione

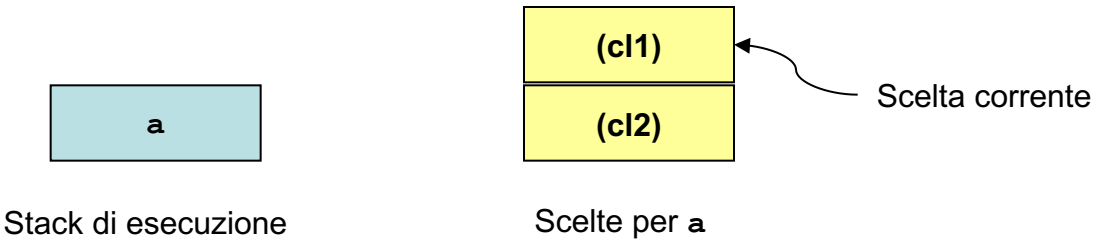
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Considerate ancora la valutazione della query

```
?- a.
```

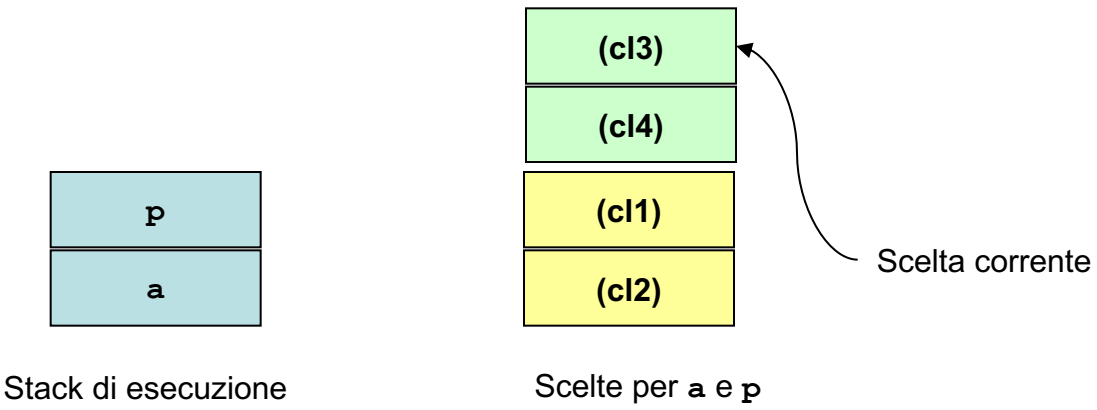


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si mette `p` in cima allo stack e lo stato interno del sistema Prolog diventa il seguente



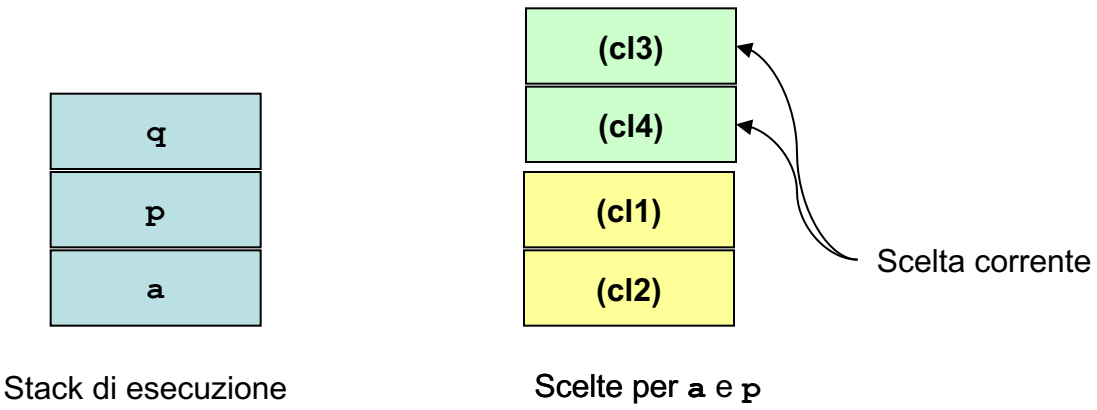
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si prosegue inserendo `q` in cima allo stack di esecuzione

La valutazione di `q` fallisce ⇒ **backtracking**

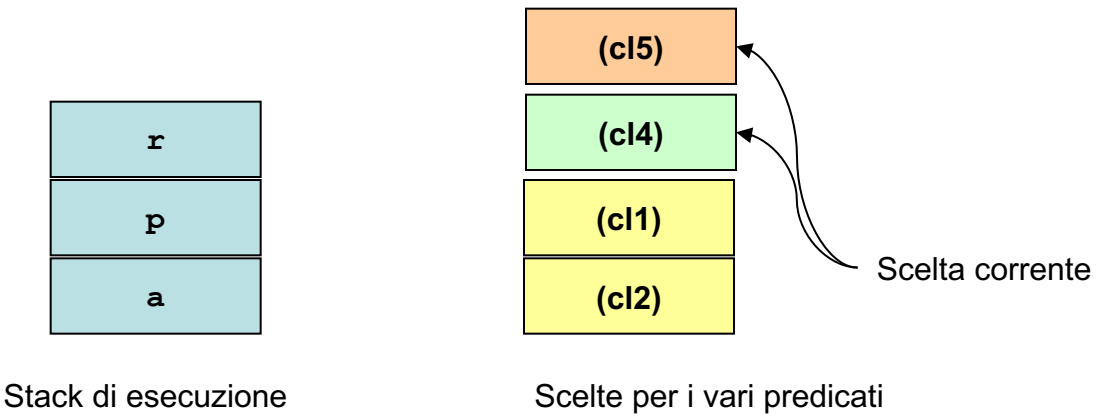


Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si prosegue inserendo `r` in cima allo stack di esecuzione



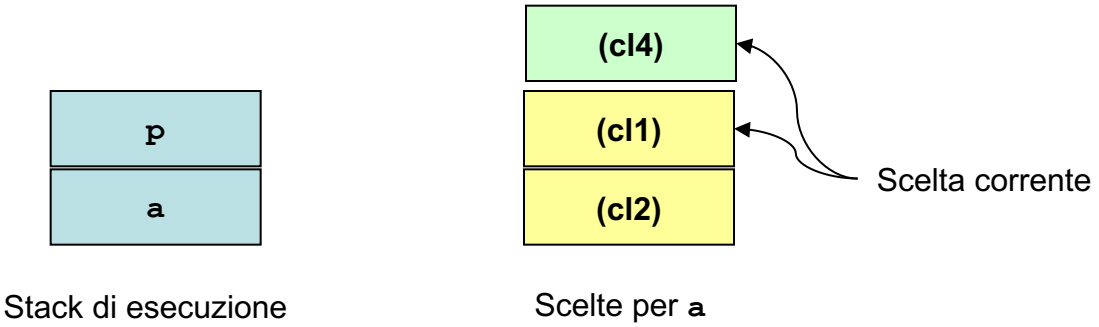
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si prosegue inserendo `r` in cima allo stack di esecuzione

La valutazione di `r` ha successo, e quindi anche `p`



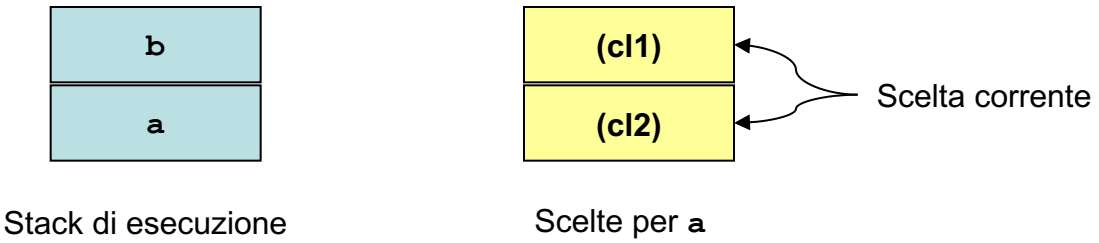
Modello di Esecuzione di un Programma Prolog

- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si inserisce `b` in cima allo stack di esecuzione

La valutazione di `b` fallisce; si cambia clausola



Modello di Esecuzione di un Programma Prolog

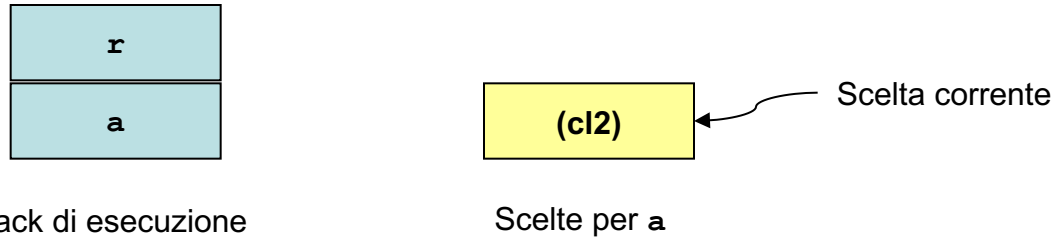
- Considerate il seguente programma Prolog

```
(cl1)  a :- p, b.
(cl2)  a :- r.
(cl3)  p :- q.
(cl4)  p :- r.
(cl5)  r.
```

- Si inserisce `r` in cima allo stack di esecuzione

`r` ha successo

Gli stack sono ora vuoti e la query iniziale **`?- a`** ha successo



Modello di Esecuzione di un Programma Prolog

Cosa succede se abbiamo delle queries con il **cut** (**!**)?

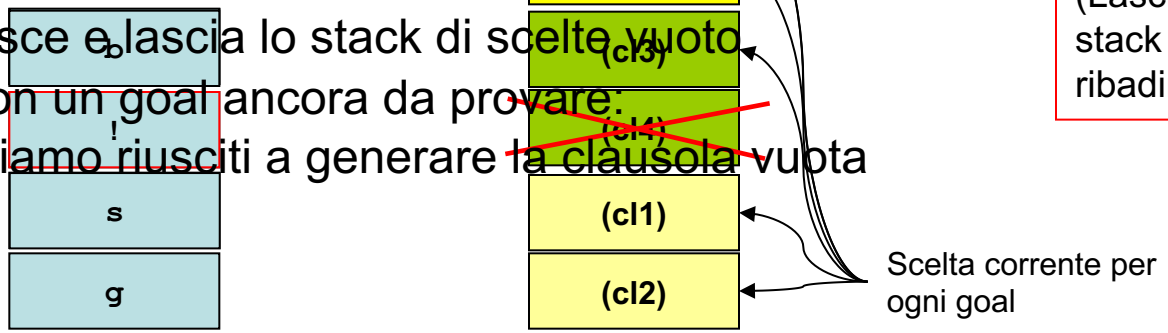
Modello di esecuzione: il trattamento del cut

- Considerate il seguente programma Prolog

```
(cl1)  g :- a.
(cl2)  g :- s.
(cl3)  a :- p, !, b.
(cl4)  a :- r.
(cl5)  p :- q.
(cl6)  p :- r.
(cl7)  r.
```

- Consideriamo il goal

```
:- g.
```



`s` fallisce e lascia lo stack di scelte vuoto ma con un goal ancora da provare: non siamo riusciti a generare la clausola vuota

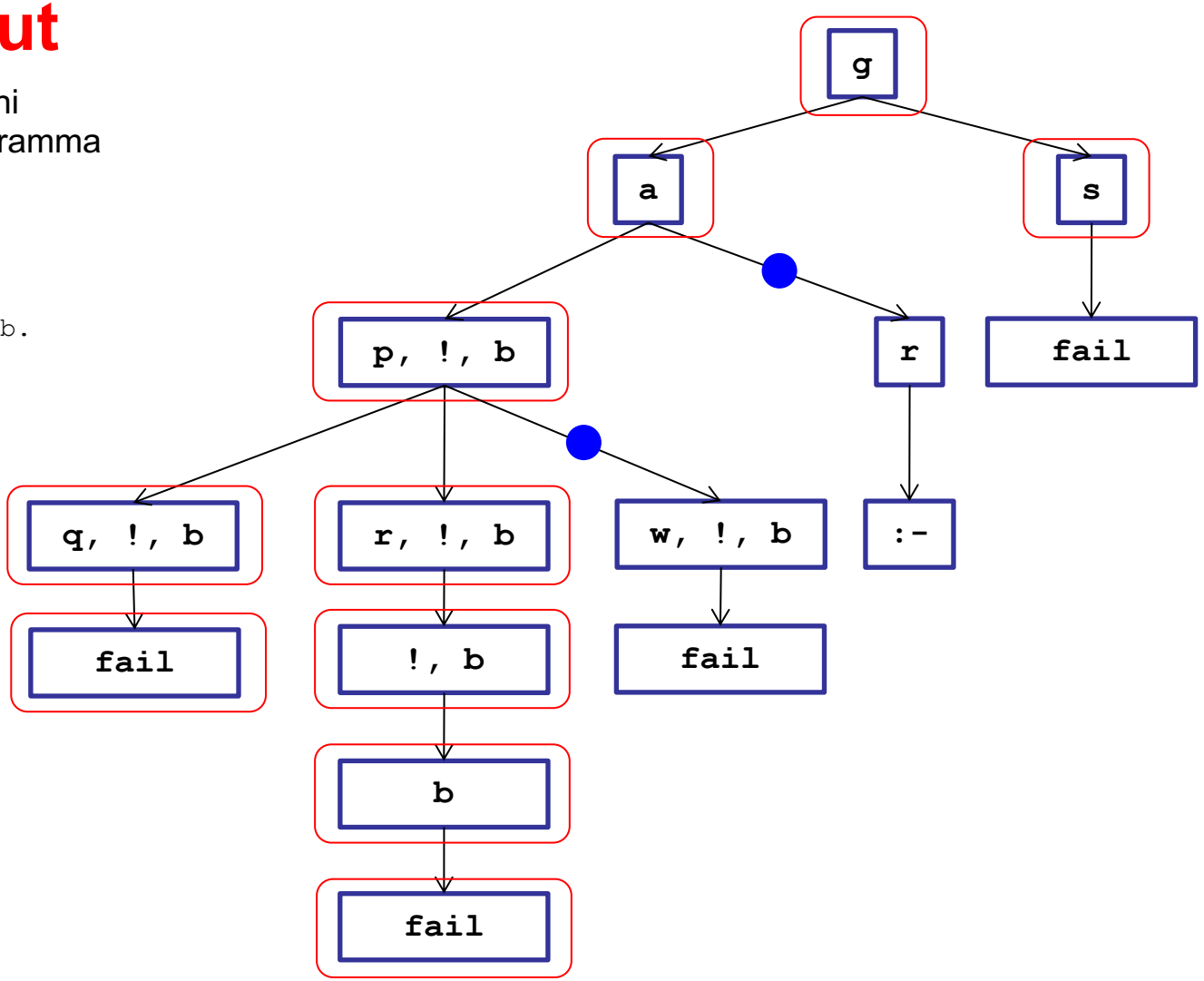
Effetto del **cut** (`!`): tutti i punti di scelta per `a` (e per `p`) sono "rimossi" dallo stack

(Lasciamo il `!` sullo stack giusto per ribadire il suo effetto).

Modello di esecuzione: il trattamento del cut

- Albero di derivazioni SLD/LM per il programma con goal `:- g.`

```
(cl1)   g :- a.
(cl2)   g :- s.
(cl3)   a :- p, !, b.
(cl4)   a :- r.
(cl5)   p :- q.
(cl6)   p :- r.
(cl7)   p :- w.
(cl8)   r.
```



Due tipi di cut

- Si possono distinguere due tipi di cut (ovvero due usi del predicato cut)

- Green Cuts
  utili per esprimere “determinismo” (e quindi per rendere più efficiente il programma)

- Red Cuts
  usati per soli scopi di efficienza, hanno per caratteristica principale quella di omettere alcune condizioni esplicite in un programma e, *soprattutto*, quella di modificare la semantica del programma equivalente senza cuts
  - Quindi sono tendenzialmente **indesiderabili** (anche se, a volte, utili)

Esempio 1

Consideriamo il seguente programma che serve a fare il "merge" di due liste ordinate

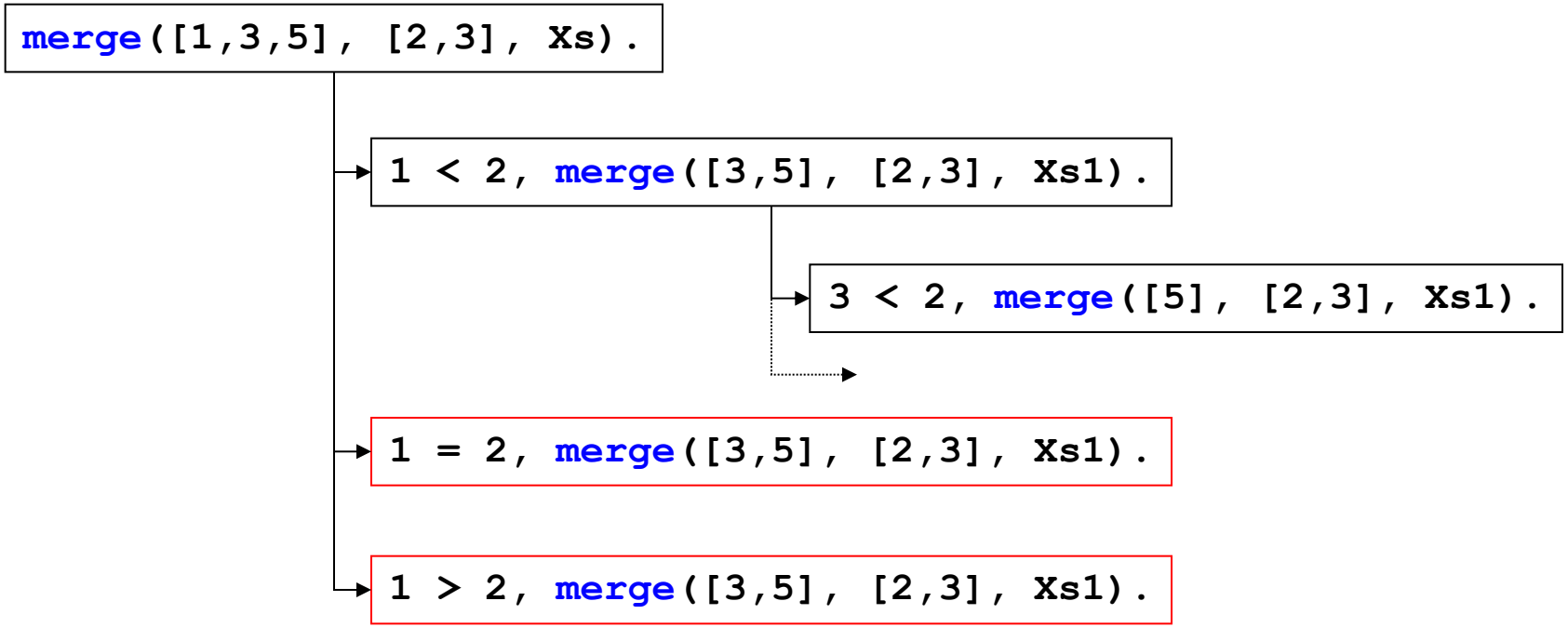
```
merge([X | Xs], [Y | Ys], [X | Zs]) :-
   X < Y,
   merge(Xs, [Y | Ys], Zs).
merge([X | Xs], [Y | Ys], [X, Y | Zs]) :-
   X = Y,
   merge(Xs, Ys, Zs).
merge([X | Xs], [Y | Ys], [Y | Zs]) :-
   X > Y,
   merge([X | Xs], Ys, Zs).
merge([], Ys, Ys).
merge(Xs, [], Xs).
```

Esempio 2

- Consideriamo anche il seguente programma che serve a controllare se quale è il minimo tra due numeri

```
minimum(X, Y, X) :- X =< Y.
minimum(X, Y, Y) :- Y < X.
```

Esecuzione di merge



Solo la prima clausola ha successo, le altre due falliscono al momento del confronto numerico; ciononostante tutte e tre le clausole vengono considerate

Esecuzione di merge

- Considerate anche la query seguente

```
?- merge([], [], Xs).
Xs = [];
Xs = [];
No
```

- Abbiamo una soluzione di troppo!

Determinismo e green cuts

- Un programma Prolog si dice deterministico quando una sola delle clausole serve (o si vorrebbe servisse) per provare un dato goal

- Come già visto i cuts che servono per esplicitare questo determinismo vengono detti **green cuts**

Esempio 1: merge con green cuts

Consideriamo il seguente programma che serve a fare il "merge" di due liste ordinate

```
merge([X | Xs], [Y | Ys], [X | Zs]) :-
   X < Y, !,
   merge(Xs, [Y | Ys], Zs).
merge([X | Xs], [Y | Ys], [X, Y | Zs]) :-
   X = Y, !,
   merge(Xs, Ys, Zs).
merge([X | Xs], [Y | Ys], [Y | Zs]) :-
   X > Y, !,
   merge([X | Xs], Ys, Zs).
merge([], Ys, Ys) :- !.
merge(Xs, [], Xs) :- !.
```

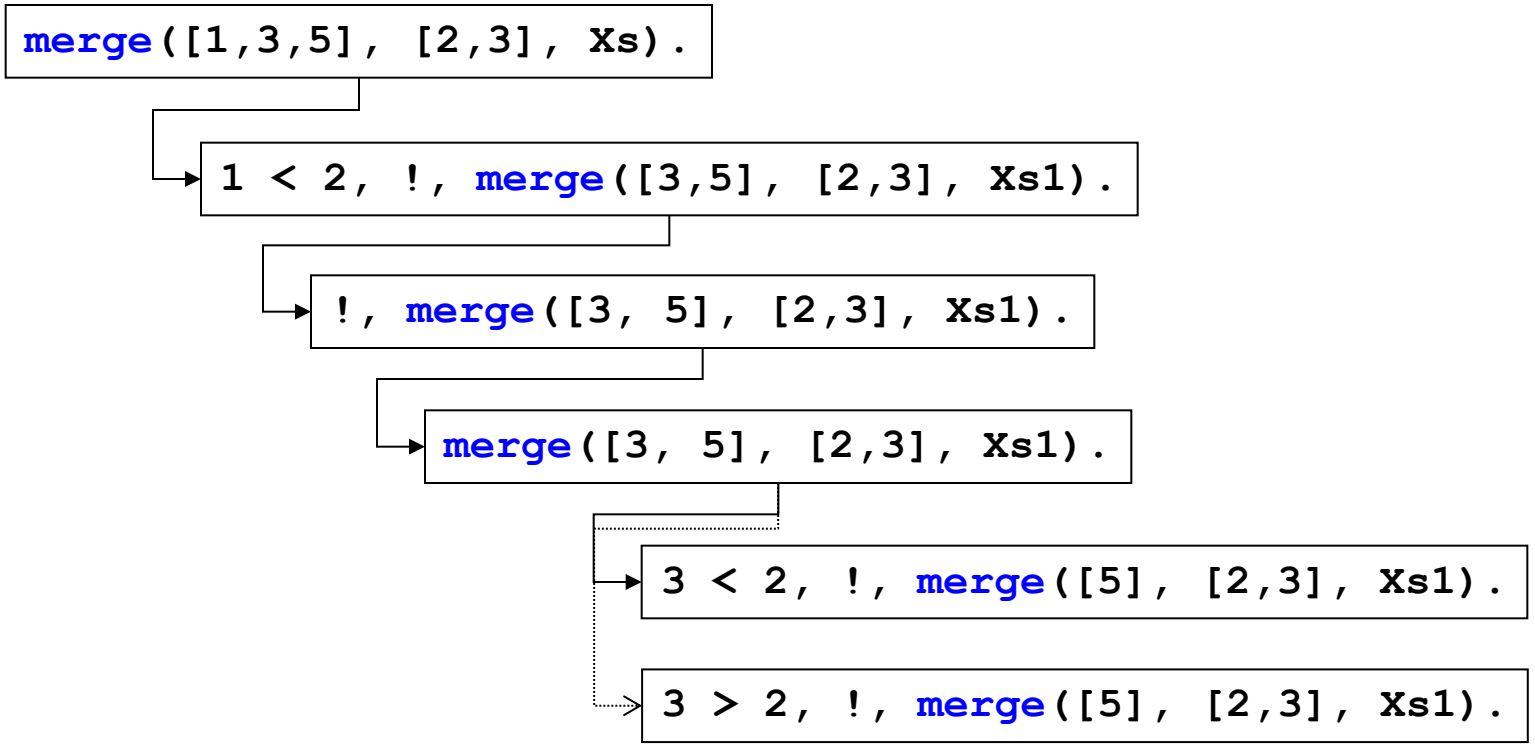
Esecuzione di merge con green cuts

- La query con troppe soluzioni ora diventa

```
?- merge([], [], Xs).
Xs = [];
No
```

- Ovvero abbiamo esattamente il numero di soluzioni che ci interessa!

Esecuzione di merge



Solo la prima clausola ha successo, ed il cut fa si che la seconda e la terza clausola **non** vengano considerate

Esempio 2: minimum con green cuts

- Il programma diventa

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

- Notate come il secondo cut sia in realtà ridondante
- Viene comunque messo nel programma per motivi di simmetria

Esempio 2: minimum con red cuts

- Riconsideriamo il programma minimum

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

- Non solo il secondo cut è ridondante
- Una volta che il programma ha fallito la prima clausola (ovvero il test `X =< Y`) al sistema Prolog non rimane che controllare la clausola seguente

- Premature optimization is the root of all evil (cit. Dijkstra)
- Some optimizations, even if not premature, are still evil

Esempio 2: minimum con red cuts

- Il programma potrebbe essere riscritto in maniera non simmetrica nel seguente modo

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

- In questo caso il cut è red, dato che serve solo (?!?) a tagliare delle soluzioni
- Non solo, il goal `minimum(2, 5, 5)` viene verificato
  - Quindi il programma è scorretto

- **I red cuts vanno usati con estrema cura**

Sommario

- Modello di esecuzione Prolog
  - Attraversamento di un albero di derivazioni **SLD/left-most**
    - In profondità (“**depth-first**”)
    - Con “**backtracking**”

- Controllo dell’attraversamento
  - “Predicato” **taglio** (“**cut**”), indicato con ‘**!**’
    - Rimozione di rami dall’albero di derivazioni