

Programmazione Logica

- PROLOG: PROgramming in LOGic, nato nel 1973
- È il più noto linguaggio di Programmazione Logica
- Basato sulla logica dei Predicati del Primo Ordine (prova automatica di teoremi)
- Linguaggio dichiarativo
- Consente di concentrarsi sulla specifica del problema piuttosto che sulla strategia di soluzione
- Particolarmente adatto per applicazioni di Intelligenza Artificiale

INTERPRETE PROLOG: SWI PROLOG

- Interprete consigliato: SWI Prolog
- Scaricabile gratuitamente per
 - Linux
 - macOS
 - Windows
- Comprende un debugger grafico
- <http://www.swi-prolog.org>
- Possibili alternative:
 - GNU Prolog, SICStus Prolog (a pagamento), ...
 - SWISH (online): <https://swish.swi-prolog.org/>



SWI Prolog

LIBRO

- *Libro Prolog*
 - L. Console, E. Lamma, P. Mello, M. Milano: *Programmazione Logica e Prolog*. UTET, 1997



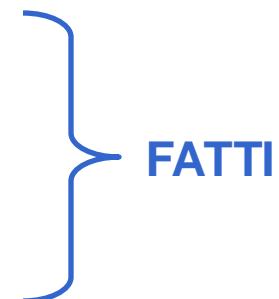
PROGRAMMA PROLOG

- Un PROGRAMMA PROLOG è costituito da:
 - FATTI asserzioni riguardanti gli oggetti in esame e le loro relazioni
 - REGOLE sugli oggetti e sulle relazioni (SE... ALLORA)
 - GOAL da raggiungere sulla base della conoscenza definita

PROGRAMMA PROLOG

- Un PROGRAMMA PROLOG è costituito da:
 - **FATTI** asserzioni riguardanti gli oggetti in esame e le loro relazioni
 - **REGOLE** sugli oggetti e sulle relazioni (SE... ALLORA)
 - **GOAL** da raggiungere sulla base della conoscenza definita
- ESEMPIO:

```
lavora(marco, ibm).  
lavora(giulia, ibm).  
lavora(erica, apple).  
lavora(dario, amazon).  
lavora(francesca, apple).
```



PROGRAMMA PROLOG

- Un PROGRAMMA PROLOG è costituito da:
 - FATTI asserzioni riguardanti gli oggetti in esame e le loro relazioni
 - REGOLE sugli oggetti e sulle relazioni (SE... ALLORA)
 - GOAL da raggiungere sulla base della conoscenza definita
 - ESEMPIO:

```
lavora(marco, ibm) .  
lavora(giulia, ibm) .  
lavora(erica, apple) .  
lavora(dario, amazon) .  
lavora(francesca, apple) .
```

FATTI

Head

Body

`collega(X,Y) :- lavora(X,Z), lavora(Y,Z), diverso(X,Y).`  **REGOLA**

PROGRAMMA PROLOG

- Un PROGRAMMA PROLOG è costituito da:
 - FATTI asserzioni riguardanti gli oggetti in esame e le loro relazioni
 - REGOLE sugli oggetti e sulle relazioni (SE... ALLORA)
 - GOAL da raggiungere sulla base della conoscenza definita
- ESEMPIO:

```
lavora(marco, ibm).  
lavora(giulia, ibm).  
lavora(erica, apple).  
lavora(dario, amazon).  
lavora(francesca, apple).
```

} **FATTI**

{ **Head** } { **Body** } **REGOLA**
`collega(X,Y) :- lavora(X,Z), lavora(Y,Z), diverso(X,Y).`

`:- collega(marco,giulia).` **GOAL**

ESEMPI

```
possiede(alessandro, negozio) .
```

```
possiede(giovanni, casa) .
```

```
fratelli(alessandro, giovanni) .
```

```
in_italia(roma) .
```

```
in_italia(palermo) .
```

```
in_francia(parigi) .
```

Possiamo porre delle domande (query) all'interprete.

```
: - possiede(alessandro, negozio) .
```

```
: - possiede(alessandro, casa) .
```

```
: - possiede(alessandro, X) .
```

ESEMPI

```
possiede(alessandro, negozio) .
```

```
possiede(giovanni, casa) .
```

```
fratelli(alessandro, giovanni) .
```

```
in_italia(roma) .
```

```
in_italia(palermo) .
```

```
in_francia(parigi) .
```

Possiamo porre delle domande (query) all'interprete.

```
: - possiede(alessandro, negozio) .      -> true
```

```
: - possiede(alessandro, casa) .
```

```
: - possiede(alessandro, X) .
```

ESEMPI

```
possiede(alessandro, negozio) .
```

```
possiede(giovanni, casa) .
```

```
fratelli(alessandro, giovanni) .
```

```
in_italia(roma) .
```

```
in_italia(palermo) .
```

```
in_francia(parigi) .
```

Possiamo porre delle domande (query) all'interprete.

```
: - possiede(alessandro, negozio) .      -> true
```

```
: - possiede(alessandro, casa) .           -> false
```

```
: - possiede(alessandro, X) .
```

ESEMPI

```
possiede(alessandro, negozio) .
```

```
possiede(giovanni, casa) .
```

```
fratelli(alessandro, giovanni) .
```

```
in_italia(roma) .
```

```
in_italia(palermo) .
```

```
in_francia(parigi) .
```

Possiamo porre delle domande (query) all'interprete.

```
: - possiede(alessandro, negozio) .      -> true
```

```
: - possiede(alessandro, casa) .          -> false
```

```
: - possiede(alessandro, X) . -> true, X = negozio
```

ESEMPI

```
genitore(aldo, chiara).  
genitore(maria, franco).  
genitore(aldo, giulia).
```

% *X è figlio di Y se Y è genitore di X*

```
figlio(X, Y) :- genitore(Y, X).
```

```
:- figlio(franco, Genitore).  
:- figlio(aldo, Genitore).
```

ESEMPI

```
genitore(aldo, chiara).  
genitore(maria, franco).  
genitore(aldo, giulia).
```

% *X è figlio di Y se Y è genitore di X*

```
figlio(X, Y) :- genitore(Y, X).
```

```
:- figlio(franco, Genitore) . -> true, G = maria  
:- figlio(aldo, Genitore) .     -> false
```

ESEMPI

```
genitore(aldo, chiara).  
genitore(maria, franco).  
genitore(aldo, giulia).
```

% *X è figlio di Y se Y è genitore di X*

```
figlio(X, Y) :- genitore(Y, X).
```

```
:- figlio(franco, Genitore) . -> true, G = maria  
:- figlio(aldo, Genitore) .     -> false
```

```
:- genitore(X, chiara), genitore(X, giulia).
```

```
:- genitore(X, chiara), genitore(X, franco).
```

ESEMPI

```
genitore(aldo, chiara).  
genitore(maria, franco).  
genitore(aldo, giulia).
```

% *X è figlio di Y se Y è genitore di X*

```
figlio(X, Y) :- genitore(Y, X).
```

```
:- figlio(franco, Genitore) . -> true, G = maria  
:- figlio(aldo, Genitore) .     -> false
```

```
:- genitore(X, chiara), genitore(X, giulia) .  
    -> true, X = aldo  
:- genitore(X, chiara), genitore(X, franco) .  
    -> false
```

PROVA DI UN GOAL

- Un goal viene considerato “vero” se sono veri tutti i letterali che lo compongono, considerati **da sinistra a destra**
`: - collega(X,Y) , persona(X) , persona(Y) .`
- Un goal formato da un singolo letterale (atomico) viene valutato confrontandolo con le teste delle clausole contenute nel programma
- Se esiste una sostituzione per cui il confronto ha successo
 - se la clausola scelta è un fatto, la prova termina con successo;
 - se la clausola scelta è una regola, si deve verificare se il suo Body è vero.
- Se non esiste una sostituzione possibile, il goal fallisce e l’interprete restituisce **false**.

PIÙ FORMALMENTE

- Un programma Prolog è costituito da un insieme di **clausole**:

(c11) **A.**  **FATTO o ASSERZIONE**

(c12) **A :- B₁, B₂, ..., B_n.**  **REGOLA**

(c13) **:- B₁, B₂, ..., B_n.**  **GOAL**

- **A** e **B_i** sono formule atomiche
- **A** : **testa** della clausola
- **B₁, B₂, ..., B_n** : **body** della clausola
- Il simbolo “,” indica la congiunzione;
- il simbolo “:-” indica l’implicazione logica in cui **A** è il conseguente e **B₁, B₂, ..., B_n** l’antecedente
- le relazioni tra entità, definite da fatti e regole, vengono anche dette **predicati**

PIÙ FORMALMENTE

- Una formula atomica è una formula del tipo

$$p(t_1, t_2, \dots, t_n)$$

in cui p è chiamato simbolo di funzione e t_1, t_2, \dots, t_n sono termini

PIÙ FORMALMENTE

- Una **formula atomica** è una formula del tipo

$$p(t_1, t_2, \dots, t_n)$$

in cui **p** è chiamato **simbolo di funzione** e t_1, t_2, \dots, t_n sono **termini**

- Un **termine** è definito ricorsivamente come segue:
 - le costanti (numeri interi/floating point, stringhe alfanumeriche aventi come primo carattere una lettera minuscola) sono termini;
 - le variabili (stringhe alfanumeriche aventi come primo carattere una lettera maiuscola oppure il carattere “_”) sono termini;
 - $f(t_1, t_2, \dots, t_k)$ è un termine se “**f**” è un simbolo di funzione a k argomenti e t_1, t_2, \dots, t_k sono termini.

Le costanti possono essere considerate come simboli di funzione a zero argomenti.

ESEMPI

- COSTANTI: `a`, `pippo`, `aB`, `9`, `135`, `a92`
- VARIABILI: `x`, `x1`, `Pippo`, `_Pippo`, `_x`, `_`
 - la variabile `_` prende il nome di variabile anonima
- TERMINI COMPOSTI: `f(a)`, `f(g(1))`, `f(g(1), b(a))`, `27`
- CLAUSOLE:

```
q.  
p :- q, r.  
r(Z).  
p(X) :- q(X, g(a)).
```
- GOAL:

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

ESEMPIO

```
mortale(X) :- uomo(X).
```

```
uomo(socrate).
```

: - mortale(socrate). -> true

- Scansionando le clausole nel programma, Prolog tenta il matching di **mortale(socrate)** con il primo fatto o testa di regola compatibile
- Trova **mortale(x)** che ha un match con il goal tramite la sostituzione **x=socrate**.
- La sostituzione si estende al corpo della regola: **uomo(socrate)** che diventa il nuovo goal
- Prolog trova quindi il fatto **uomo(socrate)** identico al goal, e la prova si conclude con successo **-> true**

INTERPRETAZIONE DICHIARATIVA

padre (X, Y) “**X** è il padre di **Y**”

madre (X, Y) “**X** è la madre di **Y**”

nonno (X, Y) :- padre (X, Z), padre (Z, Y).

“Per ogni **x** e **y**, **x** è il nonno di **y** se esiste **z** tale che **x** è il padre di **z** e **z** è il padre di **y**”

nonno (X, Y) :- padre (X, Z), madre (Z, Y).

“Per ogni **x** e **y**, **x** è il nonno di **y** se esiste **z** tale che **x** è il padre di **z** e **z** è la madre di **y**”

ESEMPIO

```
padre(aldo, bruna) .  
padre(aldo,mario) .  
padre(mario,francesco) .  
padre(alfredo, carlo) .  
padre(carlo, giulia) .
```

```
madre(marta, alessio) .  
madre(lia, giorgio) .  
madre(giulia, francesco) .  
madre(giulia, dario) .
```

```
nonno(X,Y) :- padre(X,Z), padre(Z,Y) .  
nonno(X,Y) :- padre(X,Z), madre(Z,Y) .
```

```
:- nonno(aldo, Nipote) .  
:- nonno(Nonno, dario) .
```

SOLUZIONI MULTIPLE E DISGIUNZIONE

- Possono esistere più soluzioni per una query.
- In Prolog, tali soluzioni possono essere richieste con l'operatore “;”.

```
:– nonno(carlo, Nipote).  
true   Nipote = francesco;  
true   Nipote = dario;  
false
```

- Se non ci sono più soluzioni, l'interprete restituisce **false**.

INTERPRETAZIONE PROCEDURALE

- Una *procedura* è un insieme di clausole le cui teste hanno lo stesso simbolo di funzione e lo stesso numero di argomenti.
- Una query del tipo:

:- **p(t₁, t₂, ..., t_n)**.

è la *chiamata* della procedura **p**. Gli argomenti di **p** sono i *parametri*.

:- **nonno(alberto, vincenzo)**.

- La sostituzione (pattern matching) è il meccanismo di *passaggio dei parametri* ($X = alberto$, $Y = vincenzo$)
- Non vi è alcuna distinzione a priori tra i parametri di ingresso e i parametri di uscita (*reversibilità*).

INTERPRETAZIONE PROCEDURALE

- Il corpo di una regola può a sua volta essere visto come una sequenza di chiamate di procedure.

nonno (X, Y) :- padre (X, Z) , padre (Z, Y) .

- La procedura ha esito positivo se tutte le chiamate alle sotto-procedure hanno esito positivo.
- Due regole con la stessa testa corrispondono a due definizioni alternative del corpo di una procedura.

nonno (X, Y) :- padre (X, Z) , padre (Z, Y) .

nonno (X, Y) :- padre (X, Z) , madre (Z, Y) .
- Tutte le variabili sono a *singolo assegnamento*. Il loro valore è unico durante la computazione e può cambiare solo quando si cerca una soluzione alternativa.

ESEMPIO

```
pratica_sport(mario, calcio).
pratica_sport(giovanni, calcio).
pratica_sport(alberto, calcio).
pratica_sport(marco, basket).
abita(mario, torino).
abita(giovanni, genova).
abita(alberto, genova).
abita(marco, torino).

:- practica_sport(X, calcio).
"Esiste una persona X che pratica il calcio?"
```

ESEMPIO

```
pratica_sport(mario, calcio).
pratica_sport(giovanni, calcio).
pratica_sport(alberto, calcio).
pratica_sport(marco, basket).
abita(mario, torino).
abita(giovanni, genova).
abita(alberto, genova).
abita(marco, torino).

:- practica_sport(X, calcio).
    "Esiste una persona X che pratica il calcio?"
true      X = mario;
            X = giovanni;
            X = alberto;
false
```

ESEMPIO

```
pratica_sport(mario, calcio).
pratica_sport(giovanni, calcio).
pratica_sport(alberto, calcio).
pratica_sport(marco, basket).
abita(mario, torino).
abita(giovanni, genova).
abita(alberto, genova).
abita(marco, torino).

:- practica_sport(X, calcio).
    "Esiste una persona X che pratica il calcio?"
true      X = mario;
            X = giovanni;
            X = alberto;
false

:- practica_sport(giovanni, Y).
    "Esiste uno sport Y praticato da giovanni?"
```

ESEMPIO

```
pratica_sport(mario, calcio).
pratica_sport(giovanni, calcio).
pratica_sport(alberto, calcio).
pratica_sport(marco, basket).
abita(mario, torino).
abita(giovanni, genova).
abita(alberto, genova).
abita(marco, torino).

:- practica_sport(X, calcio).
    "Esiste una persona X che pratica il calcio?"
true      X = mario;
            X = giovanni;
            X = alberto;
false

:- practica_sport(giovanni, Y).
    "Esiste uno sport Y praticato da giovanni?"
true      Y = calcio;
false
```

ESEMPIO (2)

```
: - pratica_sport(X, Y).
```

"Esistono X e Y tali per cui X pratica lo sport Y?"

ESEMPIO (2)

```
: - pratica_sport(X, Y).  
    "Esistono X e Y tali per cui X pratica lo sport Y?"  
true      X = mario        Y = calcio;  
            X = giovanni     Y = calcio;  
            X = alberto       Y = calcio;  
            X = marco         Y = basket;  
false
```

ESEMPIO (2)

```
: - pratica_sport(X, Y).  
    "Esistono X e Y tali per cui X pratica lo sport Y?"  
true      X = mario        Y = calcio;  
            X = giovanni     Y = calcio;  
            X = alberto       Y = calcio;  
            X = marco         Y = basket;  
false
```

```
: - pratica_sport(X, calcio), abita(X, genova).  
    "Esiste una persona X che pratica il calcio e abita a Genova?"
```

ESEMPIO (2)

```
: - pratica_sport(X, Y).  
    "Esistono X e Y tali per cui X pratica lo sport Y?"  
true      X = mario        Y = calcio;  
            X = giovanni     Y = calcio;  
            X = alberto       Y = calcio;  
            X = marco         Y = basket;  
false
```

```
: - pratica_sport(X, calcio), abita(X, genova).  
    "Esiste una persona X che pratica il calcio e abita a Genova?"  
true      X = giovanni;  
            X = alberto;  
false
```

ESERCIZIO

```
pratica_sport(mario, calcio).  
pratica_sport(giovanni, calcio).  
pratica_sport(alberto, calcio).  
pratica_sport(marco, basket).  
abita(mario, torino).  
abita(giovanni, genova).  
abita(alberto, genova).  
abita(marco, torino).
```

- A partire da queste relazioni, definire una relazione
amico(X, Y) “X è amico di Y”
con la seguente specifica:
“X è amico di Y se X e Y praticano lo stesso sport e abitano nella stessa città”.

SOLUZIONE

- Definire una relazione `amico(X, Y)` “`X` è amico di `Y`” a partire dalla seguente specifica: “`X` è amico di `Y` se `X` e `Y` praticano lo stesso sport e abitano nella stessa città”.

```
amico(X, Y) :-  
    abita(X, Citta),  
    abita(Y, Citta),  
    pratica_sport(X, Sport),  
    pratica_sport(Y, Sport).
```

```
:- amico(giovanni, Y).
```

“esiste `Y` tale per cui Giovanni è amico di `Y`?”

true `Y = giovanni;`

`Y = alberto;`

false

Problema: `X` è amico di se stesso? Ci serve un modo per considerare solo le soluzioni in cui `X` è diverso da `Y`

STRATEGIA DI RICERCA

- Possono esistere più teste di regole unificabili con il goal selezionato.

```
nonno(X,Y) :- padre(X,Z), padre(Z,Y).  
nonno(X,Y) :- padre(X,Z), madre(Z,Y).  
:- nonno(marco, N).
```

Cosa succede in questo caso?

- Prolog deve essere in grado di generare tutte le possibili soluzioni e quindi deve considerare ad ogni passo di esecuzione tutte le possibili alternative.

Come viene scelta la regola da applicare per prima?

STRATEGIA DI RICERCA

- Dato un goal G_1 , viene selezionata la prima clausola (secondo l'ordine delle clausole nel programma) che è possibile applicare.
- Nel caso vi siano più clausole applicabili, la risoluzione di G_1 viene considerata come un *punto di scelta* nell'esecuzione.
- In caso di fallimento in un passo dell'esecuzione, Prolog ritorna al punto di scelta più recente e seleziona la clausola successiva applicabile per continuare l'esecuzione.

DEBUGGER GRAFICO

- Per attivare il debugger grafico, usare il comando:

```
: - gtrace .
```

- Consente di seguire tutto il processo di prova di un goal attraverso la rappresentazione dell'albero di prova.
- Il riquadro bindings mostra le sostituzioni di costanti a variabili.
- I punti di scelta sono rappresentati da un bivio nello stack delle chiamate.
- Per disattivare il debugger grafico, usare il comando:

```
: - nodebug .
```

ESEMPIO

```
genitore (a,b) .          (R1)
genitore (b,c) .          (R2)
antenato (X,Z) :- genitore (X,Z) .      (R3)
antenato (X,Z) :- genitore (X,Y) , antenato (Y,Z) .    (R4)
G0 :- antenato (a,c) .
```

ESEMPIO

```
genitore (a,b) . (R1)
genitore (b,c) . (R2)
antenato (X,Z) :- genitore (X,Z) . (R3)
antenato (X,Z) :- genitore (X,Y) , antenato (Y,Z) . (R4)
G0 :- antenato (a,c) .
```

:- antenato (a,c)

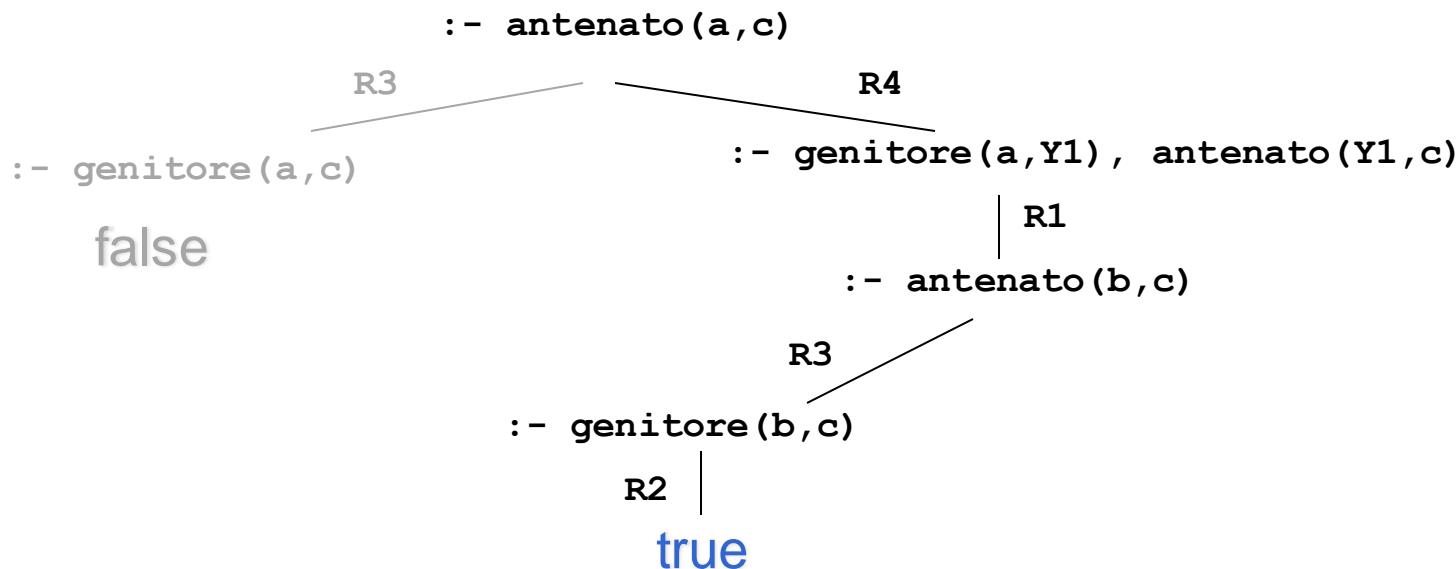
R3

:- genitore (a,c)

false

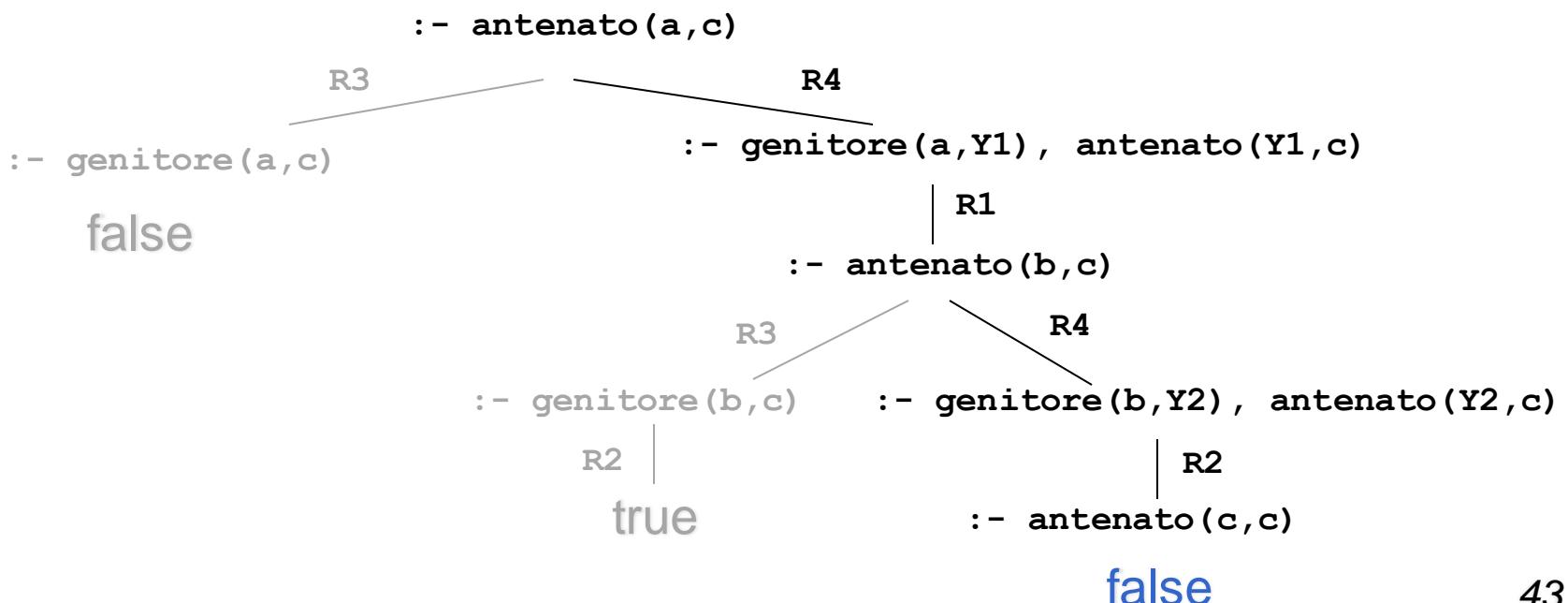
ESEMPIO

```
genitore (a,b) . (R1)
genitore (b,c) . (R2)
antenato (X,Z) :- genitore (X,Z) . (R3)
antenato (X,Z) :- genitore (X,Y) , antenato (Y,Z) . (R4)
G0 :- antenato (a,c) .
```



ESEMPIO

```
genitore (a,b) . (R1)
genitore (b,c) . (R2)
antenato (X,Z) :- genitore (X,Z) . (R3)
antenato (X,Z) :- genitore (X,Y) , antenato (Y,Z) . (R4)
G0 :- antenato (a,c) .
```



RISOLUZIONE IN PROLOG: ESEMPIO

P₁

```
(c11) p :- q, r.  
(c12) p :- s, t.  
(c13) q.  
(c14) s :- u.  
(c15) s :- v.  
(c16) t.  
(c17) v.
```

: - p.

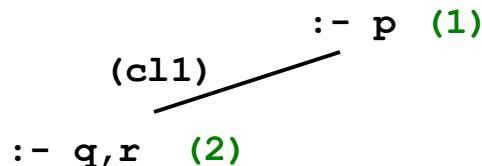
: - p (1)

RISOLUZIONE IN PROLOG: ESEMPIO

P_1

```
(c11) p :- q, r.  
(c12) p :- s, t.  
(c13) q.  
(c14) s :- u.  
(c15) s :- v.  
(c16) t.  
(c17) v.
```

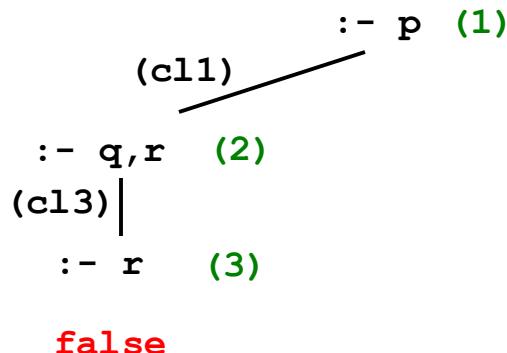
`: - p.`



RISOLUZIONE IN PROLOG: ESEMPIO

P_1

```
(cl1) p :- q, r.  
(cl2) p :- s, t.  
(cl3) q.  
(cl4) s :- u.  
(cl5) s :- v.  
(cl6) t.  
(cl7) v.  
  
:- p.
```

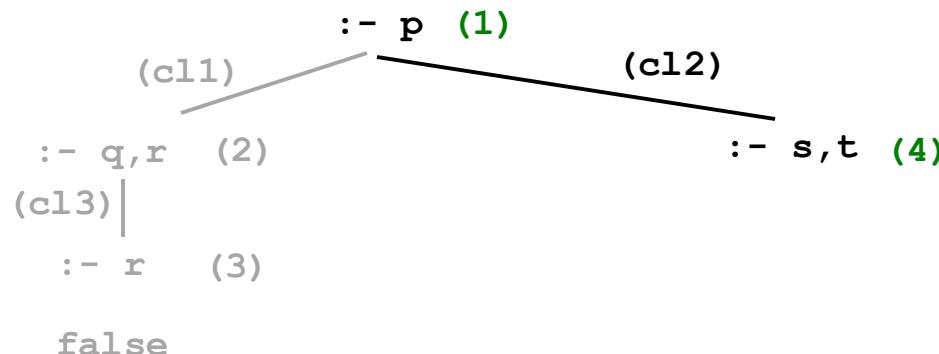


RISOLUZIONE IN PROLOG: ESEMPIO

P_1

```
(cl1) p :- q, r.  
(cl2) p :- s, t.  
(cl3) q.  
(cl4) s :- u.  
(cl5) s :- v.  
(cl6) t.  
(cl7) v.
```

`: - p.`

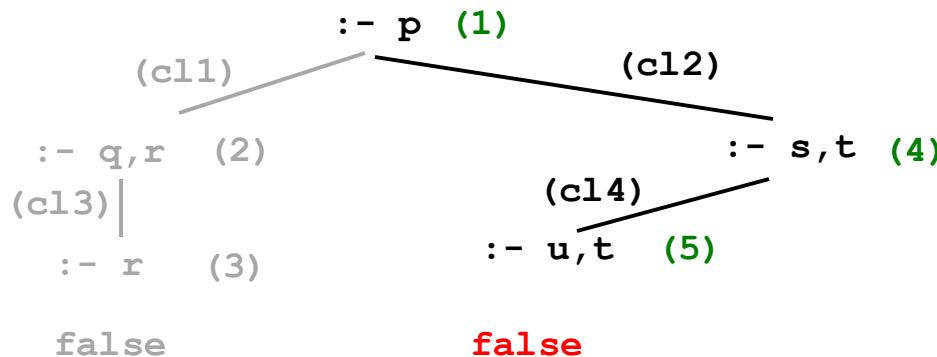


RISOLUZIONE IN PROLOG: ESEMPIO

P_1

```
(c11) p :- q, r.  
(c12) p :- s, t.  
(c13) q.  
(c14) s :- u.  
(c15) s :- v.  
(c16) t.  
(c17) v.
```

`: - p.`

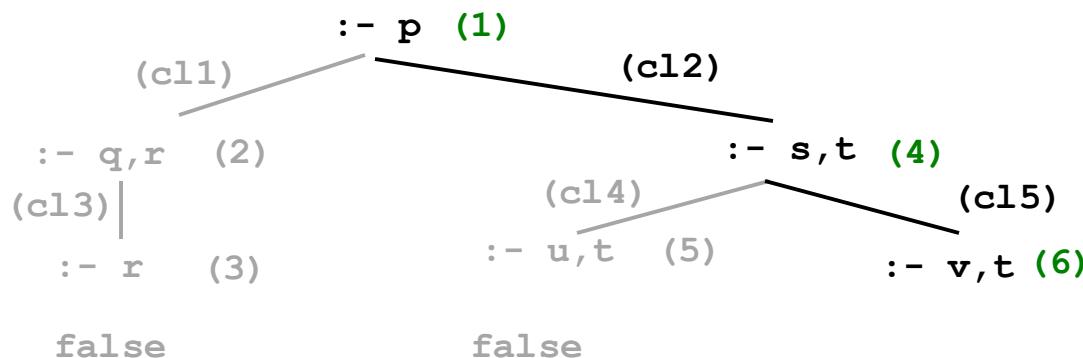


RISOLUZIONE IN PROLOG: ESEMPIO

P_1

```
(c11) p :- q, r.  
(c12) p :- s, t.  
(c13) q.  
(c14) s :- u.  
(c15) s :- v.  
(c16) t.  
(c17) v.
```

`: - p.`

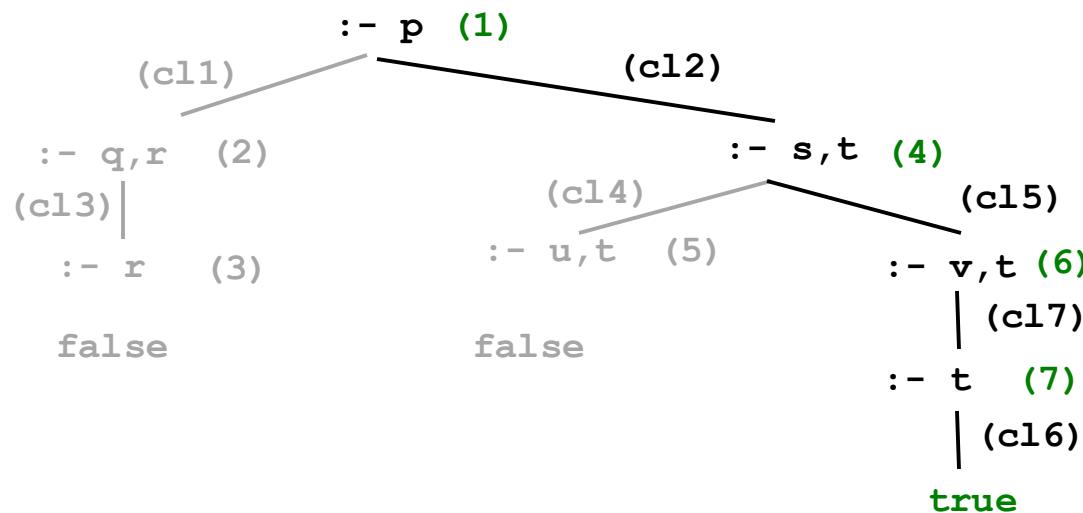


RISOLUZIONE IN PROLOG: ESEMPIO

P_1

```
(c11) p :- q, r.  
(c12) p :- s, t.  
(c13) q.  
(c14) s :- u.  
(c15) s :- v.  
(c16) t.  
(c17) v.
```

`: - p.`



ESERCIZIO

```
(c11) femmina(carla).  
(c12) femmina(maria).  
(c13) femmina(anna).  
(c14) madre(carla,maria).  
(c15) madre(carla,giovanni).  
(c16) madre(carla,anna).  
(c17) padre(luigi,maria).
```

Si ricorda che per indicare che X è diverso da Y si usa l'operatore \==

X \== Y

- A partire da queste relazioni, definire una relazione

sorella(X,Y) “X è sorella di Y”

con la seguente specifica:

“X è sorella di Y se X e Y hanno lo stesso padre o la stessa madre ed X è femmina”.

SOLUZIONE

```
(c11) femmina(carla).  
(c12) femmina(maria).  
(c13) femmina(anna).  
(c14) madre(carla,maria).  
(c15) madre(carla,giovanni).  
(c16) madre(carla,anna).  
(c17) padre(luigi,maria).
```

```
sorella(X,Y) :-  
    femmina(X),  
    padre(Z,X),  
    padre(Z,Y),  
    X \== Y.
```

```
sorella(X,Y) :-  
    femmina(X),  
    madre(Z,X),  
    madre(Z,Y),  
    X \== Y.
```

SOLUZIONE

```
(c11) femmina(carla).  
(c12) femmina(maria).  
(c13) femmina(anna).  
(c14) madre(carla,maria).  
(c15) madre(carla,giovanni).  
(c16) madre(carla,anna).  
(c17) padre(luigi,maria).
```

```
sorella(X,Y) :-  
    femmina(X),  
    padre(Z,X),  
    padre(Z,Y),  
    X \== Y.
```

```
sorella(X,Y) :-  
    femmina(X),  
    madre(Z,X),  
    madre(Z,Y),  
    X \== Y.
```

Esempio di esecuzione

```
: - sorella(maria, W).
```

VERSO UN VERO LINGUAGGIO DI PROGRAMMAZIONE

- Al Prolog puro devono essere aggiunte alcune caratteristiche per poter ottenere un linguaggio di programmazione utilizzabile nella pratica.
- In particolare:
 - Strutture dati e operazioni per la loro manipolazione.
 - Meccanismi per la definizione e valutazione di espressioni e funzioni.
 - Meccanismi di input/output.
 - Meccanismi di controllo della ricorsione e del backtracking.
 - Negazione.
- Tali caratteristiche sono state aggiunte al Prolog puro attraverso la definizione di alcuni predicati speciali (*predicati built-in*) predefiniti nel linguaggio e trattati in modo speciale dall'interprete.

ARITMETICA E RICORSIONE

- Non esiste, in logica, un meccanismo per la **valutazione** di espressioni, operazione fondamentale in un linguaggio di programmazione.
- I numeri interi possono essere rappresentati come termini in Prolog puro.
 - Il numero intero N è rappresentato dal termine:

$$\underbrace{s(s(s(\dots s(0)\dots)))}_{N \text{ volte}}$$

PREDICATI PER LA VALUTAZIONE DI ESPRESSIONI

- *L'insieme degli atomi Prolog contiene sia i numeri interi sia i numeri floating point.*
- *I principali operatori aritmetici sono simboli funzionali (operatori) predefiniti del linguaggio.*
- *Per gli operatori aritmetici binari, Prolog consente sia una notazione prefissa, sia la più tradizionale notazione infissa*

TABELLA OPERATORI ARITMETICI

Operatori Unari	<i>-, exp, log, log10, sin, cos, tan</i>
Operatori Binari	<i>+, -, *, /, div, mod, **</i>

- $+(2, 3)$ e $2 + 3$ sono due rappresentazioni equivalenti.
- $2 + 3 * 5$ viene interpretata correttamente come $2 + (3 * 5)$

PREDICATI PER LA VALUTAZIONE DI ESPRESSONI

- Per valutare un'espressione, possiamo utilizzare il predicato predefinito *is*.

T is Expr

- *T* può essere un numero o una variabile
 - *Expr* deve essere un'espressione aritmetica.
-
- L'espressione *Expr* viene valutata e il risultato della valutazione viene sostituito a *T*, se possibile.
 - Se non è possibile effettuare la sostituzione, si ha un fallimento.

*Le variabili in **Expr** devono essere già istanziate al momento della valutazione*

ESEMPI

?- *X is 2 + 3.*

?- *X1 is 2 + 3, X2 is exp(X1), X3 is X1 * X2.*

?- *0 is 3 - 3.*

?- *X is Y - 1.*

?- *X is 2 + 3, X is 4 + 5.*

ESEMPI

```
?- X is 2 + 3.
```

```
true      X = 5
```

```
?- X1 is 2 + 3, X2 is exp(X1), X3 is X1 * X2.
```

```
true      X1 = 5           X2 = 148.413          X3 = 742.065
```

```
?- 0 is 3 - 3.
```

```
true
```

```
?- X is Y - 1.
```

ERROR: is/2: Arguments are not sufficiently instantiated

```
?- X is 2 + 3, X is 4 + 5.
```

false

ESEMPI

?- *5 + 3 is 8.*

?- *X is 2 + 3, X is X + 1.*

?- *X is 2 + 3, X is 4 + 1.*

ESEMPI

?- *5 + 3 is 8.*

false

?- *X is 2 + 3, X is X + 1.*

false

?- *X is 2 + 3, X is 4 + 1.*

true *X = 5*

- Dopo aver valutato il primo goal, il secondo goal diventa

?- *5 is 4 + 1.* (**true**)

ESEMPI

- Il predicato predefinito *is* non è reversibile; le procedure che fanno uso di tale predicato non sono (in generale) reversibili.

```
sum(X, Y, Result) :-  
    Result is X + Y.
```

```
?- sum(3, 7, Sum).
```

```
?- sum(X, 7, Sum).
```

ESEMPI

- Il predicato predefinito *is* non è reversibile; le procedure che fanno uso di tale predicato non sono (in generale) reversibili.

```
sum(X, Y, Result) :-  
    Result is X + Y.
```

```
?- sum(3, 7, Sum).  
true      Sum = 10
```

```
?- sum(X, 7, Sum).
```

ERROR: *is/2: Arguments are not sufficiently instantiated*

OPERATORI RELAZIONALI

- *Prolog fornisce operatori relazionali per confrontare i valori di espressioni.*
- *Tali operatori sono utilizzabili come goal all'interno di una clausola Prolog ed hanno notazione infissa*

OPERATORI RELAZIONALI

>, <, >=, =<, =:=, =\=



→ ***Disuguaglianza
aritmetica***

***Uguaglianza
aritmetica***

CONFRONTO TRA ESPRESSIONI

- Passi effettuati nella valutazione di

Expr1 REL Expr2

dove *REL* è un operatore relazionale ed *Expr1* e *Expr2* sono espressioni:

1. Vengono valutate *Expr1* ed *Expr2*
2. I risultati della valutazione delle due espressioni vengono confrontati tramite l'operatore *REL*

`: - 3 + 2 < 20 / 2.`

`true`

`: - 5 + X =:= 8.`

`false`

SIMBOLI DI UGUAGLIANZA E DISUGUAGLIANZA

1. $\text{Term1} =:= \text{Term2}$

$\text{Term1} =\backslash= \text{Term2}$

- $\text{Term1} =:= \text{Term2}$ se Term1 e Term2 sono aritmeticamente uguali dopo essere stati valutati.

1. $\text{Term1} == \text{Term2}$

$\text{Term1} \backslash== \text{Term2}$

- $\text{Term1} == \text{Term2}$ se Term1 e Term2 sono uguali senza applicare sostituzioni.
- $?- 3 + 2 =\backslash= 5.$ $?- 3 + 2 \backslash== 5.$

false

true

$\text{Term1} = \text{Term2}$

$\text{Term1} \backslash= \text{Term2}$

- $\text{Term1} = \text{Term2}$ se esistono delle sostituzioni che rendono Term1 e Term2 uguali.

ESERCIZIO

- *Definire una regola `special_number(N)` che abbia successo se il parametro N soddisfa la relazione:*

$$N + N = N * N$$

ESERCIZIO - SOLUZIONE

- Definire una regola *special_number(N)* che abbia successo se il parametro N soddisfa la relazione:

$$N + N = N * N$$

1) *special_number(N) :-*

Somma is N + N,

*Prod is N * N,*

Somma =:= Prod.

1) *special_number(N) :-*

X is N + N,

*X is N * N.*

1) *special_number(N) :-*

*N + N =:= N * N.*

CALCOLO DI FUNZIONI

- Una funzione può essere realizzata attraverso relazioni Prolog.
- Data una funzione f ad n argomenti, essa può essere realizzata mediante un predicato ad $n+1$ argomenti nel modo seguente

— $f : x_1, x_2, \dots, x_n \rightarrow y$ diventa
 $f(X1, X2, \dots, Xn, Y) :- <\text{calcolo di } Y>$

ESEMPIO

- Funzione valore assoluto, $\text{abs}(x) = |x|$
- Pseudocodice:

```
def abs(x):  
    if x >= 0:  
        return x  
  
    else:  
        return -x
```

ESEMPIO

- Funzione valore assoluto, $\text{abs}(x) = |x|$

% $\text{abs}(X, Y)$ - "Y è il valore assoluto di X"

$\text{abs}(X, Y) :- X \geq 0, Y \text{ is } X.$

$\text{abs}(X, Y) :- X < 0, Y \text{ is } -X.$

ESEMPIO

- *Funzione valore assoluto, $\text{abs}(x) = |x|$*

% $\text{abs}(X, Y)$ - "Y è il valore assoluto di X"

$\text{abs}(X, Y) :- X \geq 0, Y \text{ is } X.$

$\text{abs}(X, Y) :- X < 0, Y \text{ is } -X.$

- *Alternativa più sintetica:*

$\text{abs}(X, X) :- X \geq 0.$

$\text{abs}(X, Y) :- X < 0, Y \text{ is } -X.$

ESERCIZIO

- Calcolare il massimo comun divisore tra due numeri interi positivi, $mcd(x, y)$

% $mcd(X, Y, Result)$

% "Result è il massimo comun divisore di X e Y"

- Si ricorda che il MCD può essere calcolato come segue:
 - Il MCD tra X e 0 è X
 - Il MCD tra X e Y è uguale al MCD tra Y e $X \bmod Y$

```
def mcd(x, y):  
    if y == 0:  
        return x  
    else:  
        return mcd(y, x % y)
```

SOLUZIONE

- Calcolare il massimo comun divisore tra due numeri interi positivi, $mcd(x, y)$

% $mcd(X, Y, Result)$

% "Result è il massimo comun divisore di X e Y"

- Si ricorda che il MCD può essere calcolato come segue:
 - Il MCD tra X e 0 è X
 - Il MCD tra X e Y è uguale al MCD tra Y e $X \bmod Y$

$mcd(X, 0, X)$.

$mcd(X, Y, Result) :-$

$Y > 0$,

New_X is $X \bmod Y$,

$mcd(Y, New_X, Result)$.

RICORSIONE E ITERAZIONE

Pseudo-codice

iterativo

```
def print_n(string, n):  
    for i = 0; i < n; i++:  
        print(string)
```

Pseudo-codice

ricorsivo

```
def print_n(string, n):  
    if n > 0:  
        print(string)  
        print_n(string, n - 1)
```

RICORSIONE E ITERAZIONE

Prolog

```
print_n(String, N) :-  
    N > 0,  
    writeln(String),  
    N1 is N - 1,  
    print_n(String, N1).
```

**Pseudo-codice
ricorsivo**

```
def print_n(string, n):  
    if n > 0:  
        print(string)  
        print_n(string, n - 1)
```

FATTORIALE DI UN NUMERO

Pseudo-codice iterativo

```
def fact(n):
    result = 1

    while n > 1:
        result *= n
        n -= 1

    return result
```

Pseudo-codice ricorsivo

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)
```

FATTORIALE DI UN NUMERO

Prolog

```
fact(0, 1).  
  
fact(N, Result) :-  
    N > 0,  
    N1 is N - 1,  
    fact(N1, Result1),  
    Result is N * Result1.
```

Pseudo-codice ricorsivo

```
def fact(n) :  
    if n <= 1:  
        return 1  
    return n * fact(n - 1)
```

ESERCIZIO - FIBONACCI

- Calcolare l' N -esimo numero della successione di Fibonacci.
 - ***fib(N, Y)***
 - "Y è l' N -esimo numero di Fibonacci"
- Si ricorda che l' N -esimo numero di Fibonacci può essere calcolato come segue:
 - $fib(0) = 0$
 - $fib(1) = 1$
 - $fib(n) = fib(n - 2) + fib(n - 1)$ $\forall n > 1$

SOLUZIONE - FIBONACCI

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(n) = fib(n - 2) + fib(n - 1)$ $\forall n > 1$

% **fib(N, Y)** - "Y è l'**N**-esimo numero di Fibonacci"

fib(0, 0).

fib(1, 1).

fib(N, Y) :-

N > 1,

N1 is N - 1,

N2 is N - 2,

fib(N1, Fib1),

fib(N2, Fib2),

Y is Fib1 + Fib2.

RICORSIONE TAIL

- Una funzione f è definita per *ricorsione tail* se f è la funzione “più esterna” nella definizione ricorsiva o, in altri termini, se sul risultato della chiamata ricorsiva di f non vengono effettuate altre operazioni.
- La definizione di funzioni che usano ricorsione tail può essere ottimizzata dall’interprete Prolog, ottenendo programmi più efficienti.
 - Dato che la ricorsione è l’ultima chiamata della funzione, non è necessario memorizzare i risultati intermedi.

ESEMPI

Ricorsione non tail

```
fact(0, 1).  
fact(N, Result) :-  
    N > 0,  
    N1 is N - 1,  
    fact(N1, Result1),  
    Result is N * Result1.
```

Ricorsione tail

```
mcd(X, 0, X).  
mcd(X, Y, Z) :-  
    Y > 0,  
    X1 is X mod Y,  
    mcd(Y, X1, Z).
```

RICORSIONE NON TAIL → TAIL

- *In alcuni casi una funzione ricorsiva non tail può essere trasformata in una funzione ricorsiva tail*
- *Ad esempio, il fattoriale di N può essere calcolato utilizzando un accumulatore, inizializzato a 1, e un contatore incrementato ad ogni passo.*
- *Quando il contatore sarà maggiore di N, l'accumulatore sarà uguale al risultato finale.*

$$Acc_0 = 1$$

$$Acc_1 = 1 * Acc_0 = 1 * 1$$

$$Acc_2 = 2 * Acc_1 = 2 * (1 * 1)$$

$$Acc_3 = 3 * Acc_2 = 3 * (2 * (1 * 1))$$

...

$$Acc_N = N * Acc_{N-1} = N * ((N-1) * ((N-2) * (\dots * (2 * (1 * 1)) \dots)))$$

RICORSIONE NON TAIL → TAIL

Pseudo-codice iterativo

```
def fact(n):
    acc = 1
    counter = 1

    while counter <= n:
        acc *= counter
        counter += 1

    return acc
```

RICORSIONE NON TAIL → TAIL

Pseudo-codice iterativo

```
def fact(n):
    acc = 1
    counter = 1

    while counter <= n:
        acc *= counter
        counter += 1

    return acc
```

Pseudo-codice ricorsivo

```
def fact(n):
    return fact_aux(n, 1, 1)

def fact_aux(n, counter, acc):
    if counter > n:
        return acc
    acc *= counter
    return fact_aux(n, counter + 1, acc)
```

RICORSIONE NON TAIL → TAIL

% Inizializziamo Counter e Acc a 1

```
fact(N, Result) :-
```

```
    N >= 0,
```

```
    fact(N, 1, 1, Result).
```

% Se Counter > N, il risultato finale è Acc

```
fact(N, Counter, Acc, Acc) :-
```

```
    Counter > N.
```

% Se Counter <= N, aggiorniamo Acc, incrementiamo Counter

% ed effettuiamo la chiamata ricorsiva

```
fact(N, Counter, Acc, Result) :-
```

```
    Counter <= N,
```

```
    New_Acc is Acc * Counter,
```

```
    Counter1 is Counter + 1,
```

```
    fact(N, Counter1, New_Acc, Result).
```

RICORSIONE NON TAIL → TAIL

- Versione alternativa per il fattoriale con ricorsione tail:

```
fact2(N, Result) :-
```

```
    fact2(N, 1, Result).
```

```
fact2(0, Acc, Acc).
```

```
fact2(Counter, Acc, Result) :-
```

```
    Counter > 0,
```

```
    New_Acc is Acc * Counter,
```

```
    Counter1 is Counter - 1,
```

```
    fact2(Counter1, New_Acc, Result).
```

LISTE

- Le liste sono una delle strutture dati primitive più diffuse nei linguaggi di programmazione per l'elaborazione simbolica.
- Le liste in Prolog sono termini, che è possibile definire ricorsivamente:
 - Il termine `[]` rappresenta la lista vuota
 - Il termine `[Testa | Coda]` è una lista se **Testa** è un termine qualsiasi e **Coda** è una Lista.

LISTE

- Le liste sono una delle strutture dati primitive più diffuse nei linguaggi di programmazione per l'elaborazione simbolica
- Le liste in Prolog sono termini, che è possibile definire ricorsivamente:
 - Il termine `[]` rappresenta la lista vuota
 - Il termine `[Testa | Coda]` è una lista se **Testa** è un termine qualsiasi e **Coda** è una Lista.
- Esempi:

(1) <code>[]</code>	(1) Lista vuota
(2) <code>[a []]</code>	(2) Lista che contiene a
(3) <code>[a [b []]]</code>	(3) Lista che contiene a e b
(4) <code>[[[] []]]</code>	(4) Lista che contiene la lista vuota
(5) <code>[[a []] [b []]]</code>	(5) Lista che contiene due elementi: <ul style="list-style-type: none">- una lista contenente a- l'elemento b

LISTE: SINTASSI PIÙ SEMPLICE

- La lista `[a | b | c | []]` può essere rappresentata nel modo seguente: `[a, b, c]`
- Le liste nell'esempio precedente diventano:

(1)	<code>[]</code>	
(2)	<code>[a]</code>	$= [a []]$
(3)	<code>[a, b]</code>	$= [a [b []]]$
(4)	<code>[[[]]]</code>	$= [[[]] []]$
(5)	<code>[[a], b]</code>	$= [[a []] [b []]]$

LISTE: SINTASSI PIÙ SEMPLICE

- La lista `[a | [b | [c | []]]]` può essere rappresentata nel modo seguente: `[a, b, c]`
- Le liste nell'esempio precedente diventano:

(1)	<code>[]</code>	
(2)	<code>[a]</code>	$= [a \mid []]$
(3)	<code>[a, b]</code>	$= [a \mid [b \mid []]]$
(4)	<code>[[[]]]</code>	$= [[[]] \mid []]$
(5)	<code>[[a], b]</code>	$= [[a \mid []] \mid [b \mid []]]$

- Unendo le due notazioni per le liste, possiamo isolare Testa e Coda da una qualunque lista:

```
: - L = [1, 2, 3, 4], L = [Testa | Coda].
```

```
L = [1, 2, 3, 4],
```

```
Testa = 1,
```

```
Coda = [2, 3, 4].
```

ACCESSO AGLI ELEMENTI DI UNA LISTA

- La sostituzione (combinata con le varie notazioni per le liste) è un potente meccanismo per l'accesso agli elementi delle liste

```
p([1,2,3,4,5,6,7,8,9]).
```

```
:- p(X).
```

```
:- p([Testa | Coda]).
```

```
:- p([X, Y | Coda]).
```

```
:- p([_ | Coda]).
```

ACCESSO AGLI ELEMENTI DI UNA LISTA

- La sostituzione (combinata con le varie notazioni per le liste) è un potente meccanismo per l'accesso agli elementi delle liste

```
p([1,2,3,4,5,6,7,8,9]).
```

```
:- p(X).
```

```
true X = [1,2,3,4,5,6,7,8,9]
```

```
:- p([Testa | Coda]).
```

```
true Testa = 1,           Coda = [2,3,4,5,6,7,8,9]
```

```
:- p([X, Y | Coda]).
```

```
true X = 1,   Y = 2,   Coda = [3,4,5,6,7,8,9]
```

```
:- p([_ | Coda]).
```

```
true Coda = [2,3,4,5,6,7,8,9]
```

OPERAZIONI SULLE LISTE

- Le procedure che operano su liste sono definite come procedure ricorsive basate sulla definizione di lista

- Verificare se un termine è una lista:

```
:‐ is_list([1,2,3]). → true  
:‐ is_list([a|b]). → false
```

- Definizione ricorsiva di lista:

- Il termine [] rappresenta la lista vuota
- Il termine [Testa | Coda] è una lista se Testa è un termine qualsiasi e Coda è una Lista.

OPERAZIONI SULLE LISTE

- Le procedure che operano su liste sono definite come procedure ricorsive basate sulla definizione di lista
- Verificare se un termine è una lista:

```
:‐ is_list([1,2,3]). → true  
:‐ is_list([a|b]). → false
```

```
% is_list(Term) – true se Term è una lista  
  
is_list([]).  
  
is_list([Testa | Coda]) :-  
    is_list(Coda).
```

OPERAZIONI SULLE LISTE - MEMBER

- Verificare se un termine appartiene ad una lista

```
% member(Element, List) - "Element è un elemento della lista List"
```

Suggerimenti:

- Element è contenuto in List se è la Testa di List
- Element è contenuto in List se appartiene alla coda di List

```
: - member(2, [1,2,3]).
```

true

```
: - member(1, [2,3]).
```

false

OPERAZIONI SULLE LISTE - MEMBER

- Verificare se un termine appartiene ad una lista

```
% member(Element, List) - "Element è un elemento della lista List"
```

```
member(Element, [Element | _Rest]).
```

```
member(Element, [_Head | Rest]) :- member(Element, Rest).
```

```
:- member(2, [1,2,3]).
```

```
true
```

```
:- member(1, [2,3]).
```

```
false
```

```
:- member(Element, [1,2,3]).
```

```
true Element = 1;
```

```
true Element = 2;
```

```
true Element = 3;
```

```
false
```

OPERAZIONI SULLE LISTE - LENGTH

- Determinare la lunghezza di una lista

% `length(List, Len)` – “La lista List ha Len elementi”

OPERAZIONI SULLE LISTE - LENGTH

- Determinare la lunghezza di una lista

```
% length(List, Len) :- "La lista List ha Len elementi"
```

```
length([], 0).
```

```
length([_Head | Rest], Len) :-
```

```
    length(Rest, Len_Rest),
```

```
    Len is Len_Rest + 1.
```

OPERAZIONI SULLE LISTE - LENGTH

- Determinare la lunghezza di una lista

```
% length(List, Len) - "La lista List ha Len elementi"
```

```
length([], 0).
```

```
length([_Head | Rest], Len) :-
```

```
    length(Rest, Len_Rest),
```

```
    Len is Len_Rest + 1.
```

```
% Versione che utilizza la ricorsione tail
```

```
length1(List, Len) :- length1(List, 0, Len).
```

```
length1([], Acc, Acc).
```

```
length1([_ | Rest], Acc, Len) :-
```

```
    Acc1 is Acc + 1,
```

```
    length1(Rest, Acc1, Len).
```

CONCATENAZIONE DI DUE LISTE

- Concatenazione di due liste

% **append(L1, L2, L3)** – “L3 è la concatenazione di L1 e L2”

```
:‐ append([1,2,3], [4,5], Concat).  
true  Concat = [1,2,3,4,5]
```

- La concatenazione di una lista vuota ed L2 ha come risultato L2
- La concatenazione di [Head1 | Rest1] ed L2 ha come risultato una lista [Head1 | Rest_Concat], in cui Rest_Concat è la concatenazione tra Rest1 ed L2:
 - L1 = [1, 2, 3] L2 = [4, 5] L3 = ?

CONCATENAZIONE DI DUE LISTE

- Concatenazione di due liste

% `append(L1, L2, L3)` – “`L3` è la concatenazione di `L1` e `L2`”

```
:‐ append([1,2,3], [4,5], Concat).  
true  Concat = [1,2,3,4,5]
```

- La concatenazione di una lista vuota ed `L2` ha come risultato `L2`
- La concatenazione di `[Head1 | Rest1]` ed `L2` ha come risultato una lista `[Head1 | Rest_Concat]`, in cui `Rest_Concat` è la concatenazione tra `Rest1` ed `L2`:

- | | | |
|---|--------------------------|---|
| – <code>L1 = [1, 2, 3]</code> | <code>L2 = [4, 5]</code> | <code>L3 = ?</code> |
| – <code>L3 = [1 Rest_Concat]</code> | | <code>Rest_Concat = append([2, 3], [4, 5])</code> |
| – <code>L3 = [1, 2 Rest_Concat]</code> | | <code>Rest_Concat = append([3], [4, 5])</code> |
| – <code>L3 = [1, 2, 3 Rest_Concat]</code> | | <code>Rest_Concat = append([], [4, 5])</code> |
| – <code>L3 = [1, 2, 3, 4, 5]</code> | | |

CONCATENAZIONE DI DUE LISTE

- Concatenazione di due liste

```
% append(L1, L2, L3) - "L3 è la concatenazione di L1 e L2"  
append([], L2, L2).  
append([Head1 | Rest1], L2, [Head1 | Rest_Concat]) :-  
    append(Rest1, L2, Rest_Concat).
```

- La concatenazione di una lista vuota ed **L2** ha come risultato **L2**
- La concatenazione di **[Head1 | Rest1]** ed **L2** ha come risultato una lista **[Head1 | Rest_Concat]**, in cui **Rest_Concat** è la concatenazione tra **Rest1** ed **L2**:

- $L1 = [1, 2, 3] \quad L2 = [4, 5] \quad L3 = ?$
 $Rest_Concat = append([2, 3], [4, 5])$
- $L3 = [1 | Rest_Concat]$
 $Rest_Concat = append([3], [4, 5])$
- $L3 = [1, 2 | Rest_Concat]$
 $Rest_Concat = append([], [4, 5])$
- $L3 = [1, 2, 3 | Rest_Concat]$
- $L3 = [1, 2, 3, 4, 5]$

CONCATENAZIONE DI DUE LISTE - ESEMPI

```
% append(L1, L2, L3) - "L3 è la concatenazione di L1 e L2"
:- append([1,2], [3,4,5], Concat).
true  Concat = [1,2,3,4,5]
:- append([1,2], L2, [1,2,4,5]).
true  L2 = [4,5]
:- append([1,3], [2,4], [1,2,3,4]).
false
:- append(L1, L2, [1, 2, 3]).
L1 = [],
L2 = [1, 2, 3];
L1 = [1],
L2 = [2, 3];
L1 = [1, 2],
L2 = [3];
L1 = [1, 2, 3],
L2 = [].
```

ESERCIZIO - CANCELLAZIONE DI UN ELEMENTO

- Cancellazione di un elemento dalla lista

% `delete1(Element, List, New_List)`

% “La lista `New_List` contiene tutti gli `elementi` di `List` tranne il primo termine uguale ad `Element`”

ESERCIZIO - CANCELLAZIONE DI UN ELEMENTO

- Cancellazione di un elemento dalla lista

```
% delete1(Element, List, New_List)  
% "La lista New_List contiene tutti gli elementi di List tranne il  
% primo termine uguale ad Element"
```

Suggerimenti:

- Cancellare qualsiasi elemento dalla lista vuota restituisce la lista vuota
 - `delete1(22, [], Result) -> []`

ESERCIZIO - CANCELLAZIONE DI UN ELEMENTO

- Cancellazione di un elemento dalla lista

% `delete1(Element, List, New_List)`

% “La lista `New_List` contiene tutti gli `elementi` di `List` tranne il primo termine uguale ad `Element`”

Suggerimenti:

- Cancellare qualsiasi elemento dalla lista vuota restituisce la lista vuota
 - `delete1(22, [], Result) -> []`
- Cancellare un elemento che è uguale alla testa della lista restituisce la coda della lista
 - `delete1(22, [22 | [1, 2, 3]], Result) -> [1, 2, 3]`

ESERCIZIO - CANCELLAZIONE DI UN ELEMENTO

- Cancellazione di un elemento dalla lista

% `delete1(Element, List, New_List)`

% “La lista `New_List` contiene tutti gli `elementi` di `List` tranne il primo termine uguale ad `Element`”

Suggerimenti:

- Cancellare qualsiasi elemento dalla lista vuota restituisce la lista vuota
 - `delete1(22, [], Result) -> []`
- Cancellare un elemento che è uguale alla testa della lista restituisce la coda della lista
 - `delete1(22, [22 | [1, 2, 3]], Result) -> [1, 2, 3]`
- Cancellare un elemento che è diverso dalla testa della lista restituisce una lista che ha come testa la testa della lista originaria, e come coda il risultato della cancellazione dell'elemento dalla coda della lista originaria
 - `delete1(22, [5 | [22, 1, 2, 3]], Result) -> [5 | ?]` con ? risultato di `delete1(22, [22, 1, 2, 3], Result)`

SOLUZIONE - CANCELLAZIONE DI UN ELEMENTO

- Cancellazione di un elemento dalla lista

```
% delete1(Element, List, New_List)
% "La lista New_List contiene tutti gli elementi di List tranne il
  primo termine uguale ad Element"
```

```
delete1(_Element, [], []).
```

```
delete1(Element, [Element | Rest], Rest) .
```

```
delete1(Element, [Head | Rest], [Head | New_Rest]) :-
    Element \== Head,
    delete1(Element, Rest, New_Rest) .
```

CANCELLAZIONE DI PIÙ ELEMENTI

- Cancellazione di più elementi dalla lista

```
% delete_all(Element, List, New_List)
% "La lista New_List contiene gli elementi di List tranne tutti i
  termini uguali ad Element"
```

CANCELLAZIONE DI PIÙ ELEMENTI

- Cancellazione di più elementi dalla lista

```
% delete_all(Element, List, New_List)
% "La lista New_List contiene gli elementi di List tranne tutti i
termini uguali ad Element"
```

```
delete_all(_Element, [], []).
```

```
delete_all(Element, [Element | Rest], New_List) :-
    delete_all(Element, Rest, New_List).
```

```
delete_all(Element, [Head | Rest], [Head | New_Rest]) :-
    Element \== Head,
    delete_all(Element, Rest, New_Rest).
```

CONTROLLO DI UN PROGRAMMA

- L'interprete fa uso di due stack:

Lo stack di esecuzione contiene i record di attivazione delle varie procedure

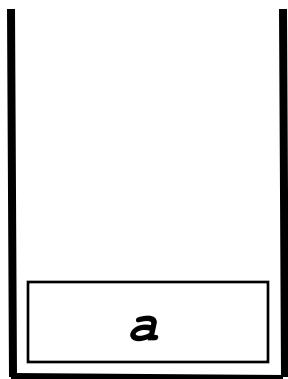
Lo stack di backtracking contiene l'insieme dei punti di scelta.

- Ad ogni fase della valutazione, tale stack contiene puntatori alle scelte ancora aperte.

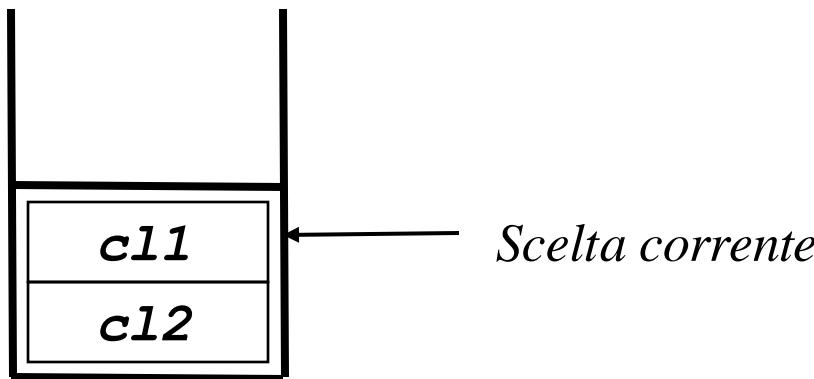
CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



Stack di esecuzione

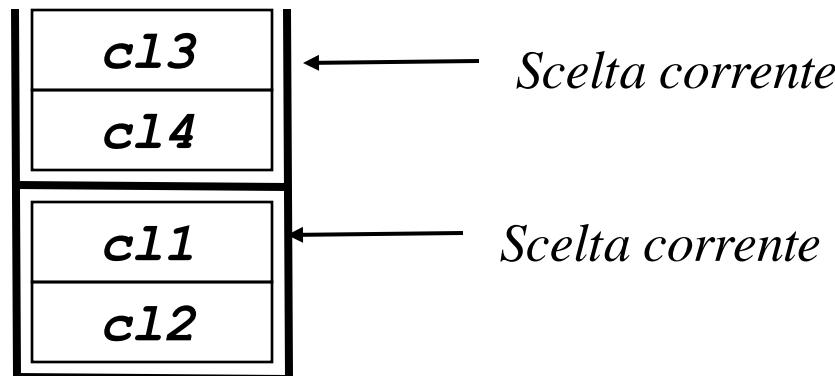
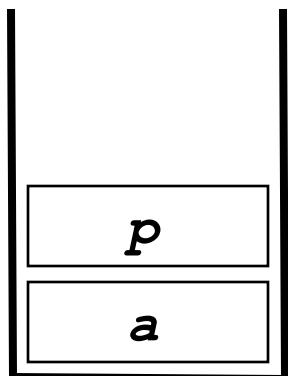


Stack di backtracking

CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



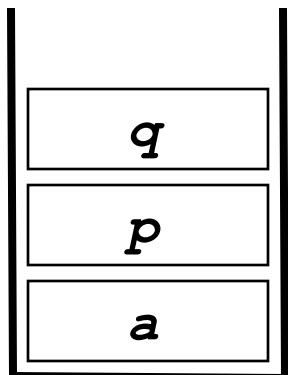
Stack di esecuzione

Stack di backtracking

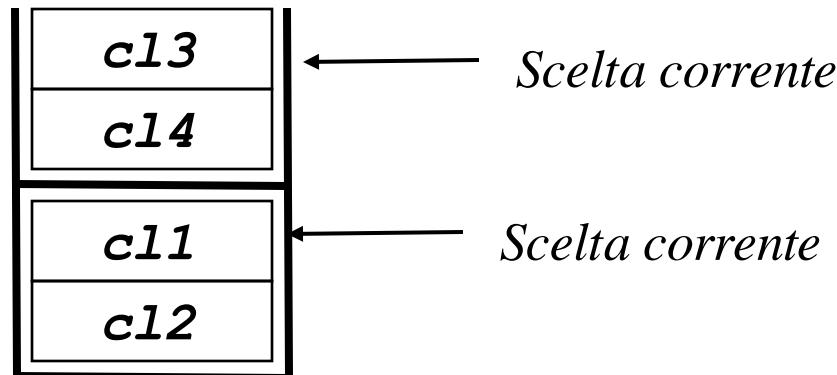
CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



Stack di esecuzione



Stack di backtracking

Scelta corrente

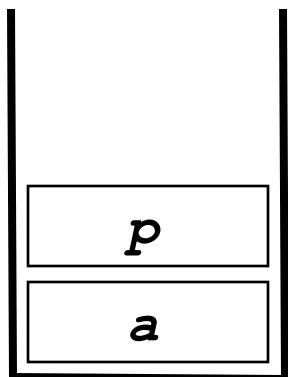
Scelta corrente

Fallimento

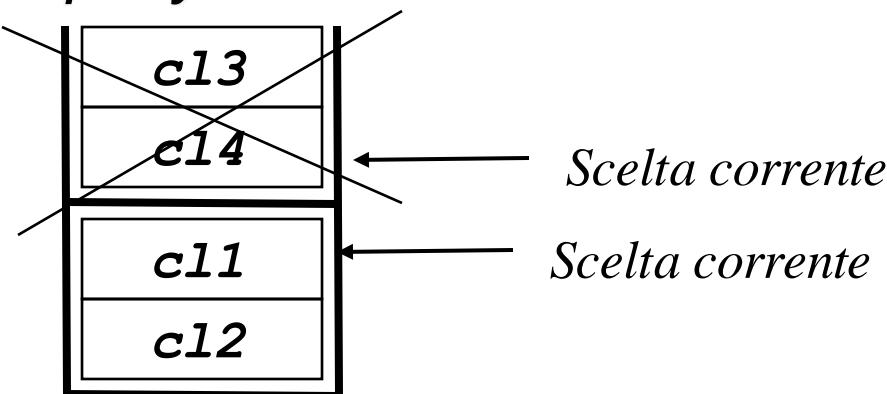
CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



Stack di esecuzione

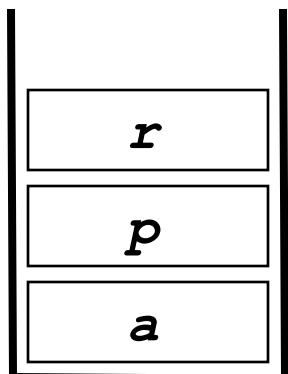


Stack di backtracking

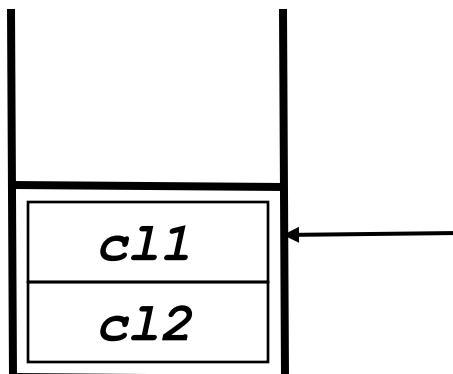
CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



Stack di esecuzione



Stack di backtracking

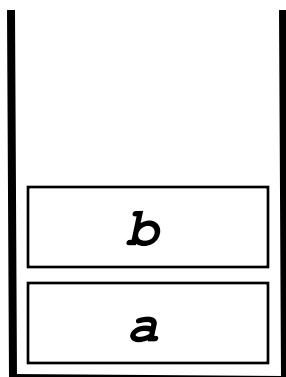
Scelta corrente

r ha successo

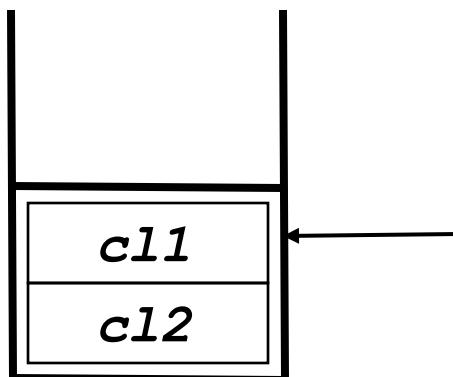
CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



Stack di esecuzione



Stack di backtracking

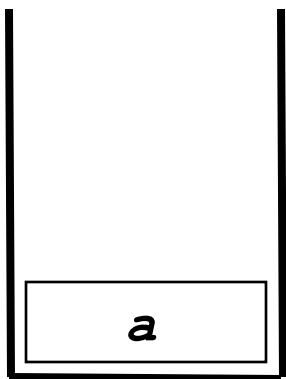
Scelta corrente

b fallisce

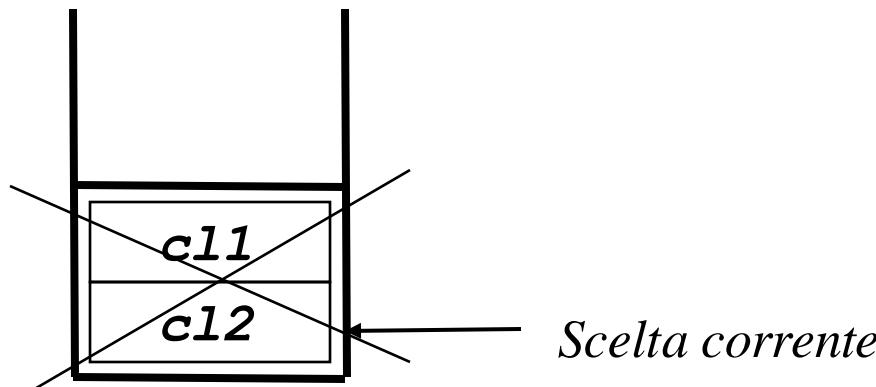
CONTROLLO DI UN PROGRAMMA

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

- Valutazione della query $: - a.$



Stack di esecuzione



Stack di backtracking

CONTROLLO DI UN PROGRAMMA

(c11) $a :- p, b.$

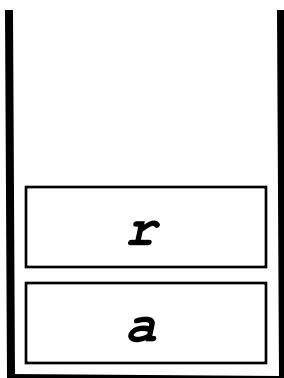
(c12) $a :- r.$

(c13) $p :- q.$

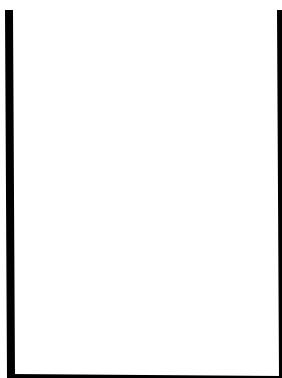
(c14) $p :- r.$

(c15) $r.$

- Valutazione della query $: - a.$



Stack di esecuzione



Stack di backtracking

Successo

IL PREDICATO CUT !

- *L'effetto del cut ! è quello di rendere definitive alcune scelte fatte nel corso della valutazione dall'interprete Prolog*
- *Il cut altera il controllo del programma*
- *Effetto collaterale più importante: perdita di dichiaratività*

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) è il seguente:

- *La valutazione di ! ha sempre successo 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. Tutti i punti di scelta per tali goal vengono rimossi dallo stack di backtracking.*
- *Le alternative riguardanti i goal seguenti al cut non vengono modificate*
- *Neanche le alternative riguardanti i goal precedenti il goal p vengono modificate*

EFFETTO DEL CUT

- Si consideri la clausola:

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

- Se la valutazione di $q_{i+1}, q_{i+2}, \dots, q_n$ fallisce, fallisce tutta la valutazione di p .

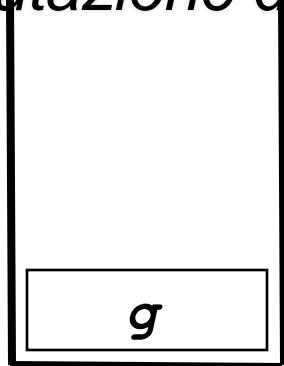
Infatti, anche se p o q_1, q_2, \dots, q_i avessero punti di scelta, questi sarebbero eliminati dal cut.

Il cut non può essere definito in modo dichiarativo

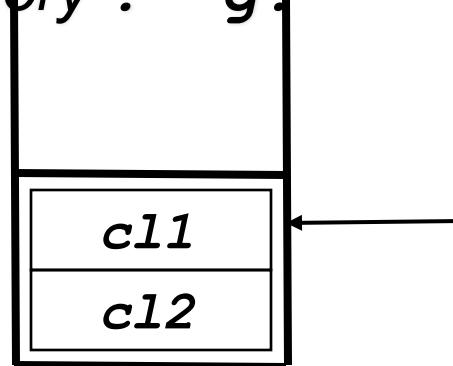
EFFETTO DEL CUT

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

- Valutazione della query $: - g.$



Stack di esecuzione

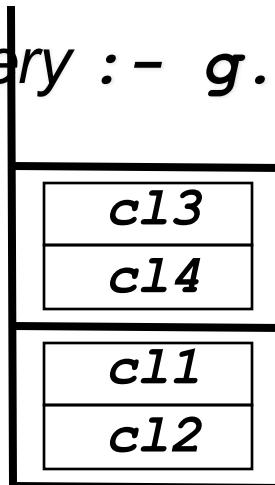
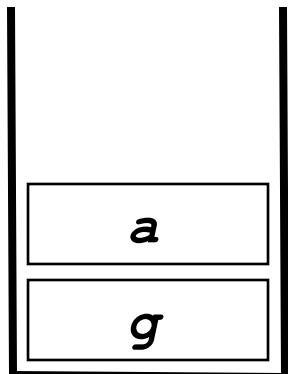


Stack di backtracking

EFFETTO DEL CUT

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

- Valutazione della query $: - g.$



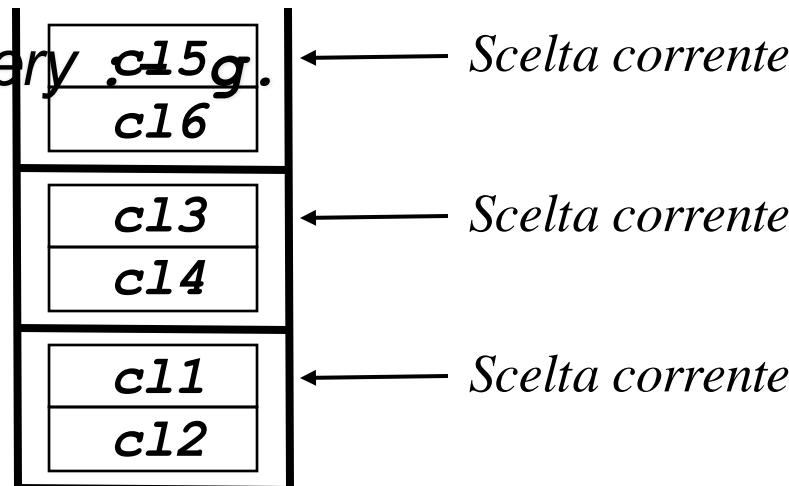
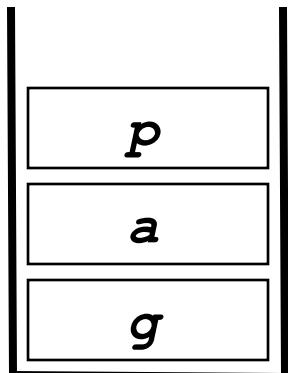
Stack di esecuzione

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) $r.$

- Valutazione della query $.c15g.$



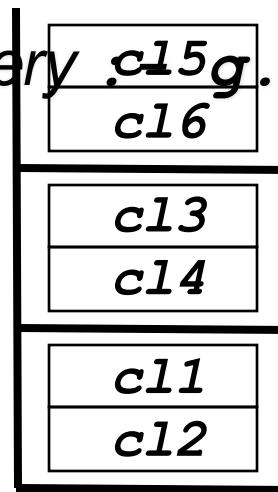
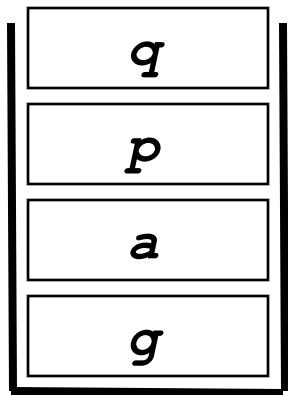
Stack di esecuzione

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) $r.$

- Valutazione della query $.c15g.$



Stack di esecuzione

Stack di backtracking

← Scelta corrente

← Scelta corrente

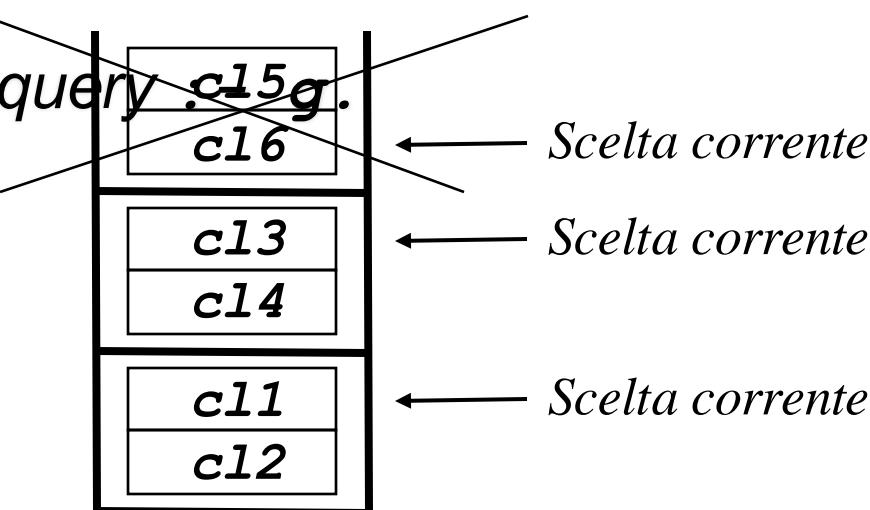
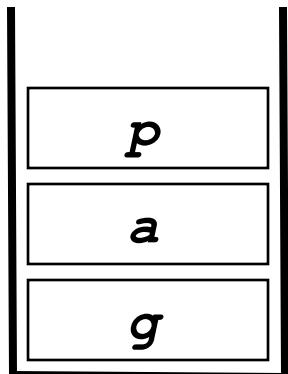
← Scelta corrente

Fallimento per q

EFFETTO DEL CUT

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

- Valutazione della query $.c15g.$



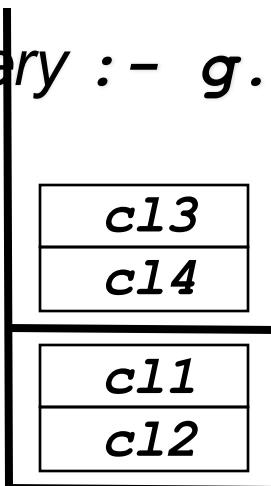
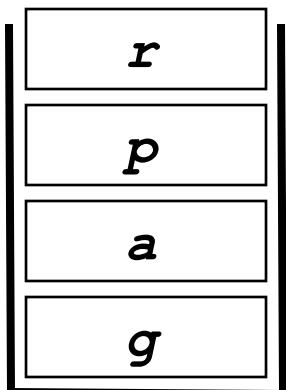
Stack di esecuzione

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) $r.$

- Valutazione della query $: - g.$



Stack di esecuzione

Stack di backtracking

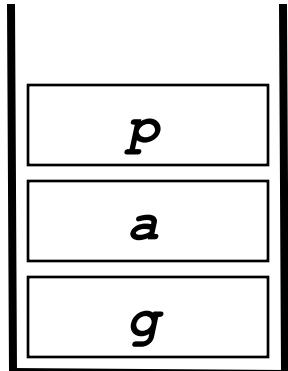
← Scelta corrente

← Scelta corrente

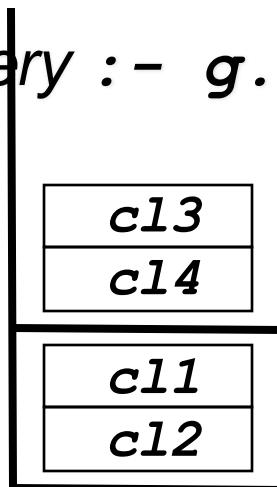
EFFETTO DEL CUT

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

- Valutazione della query $: - g.$



Stack di esecuzione



Stack di backtracking

← Scelta corrente

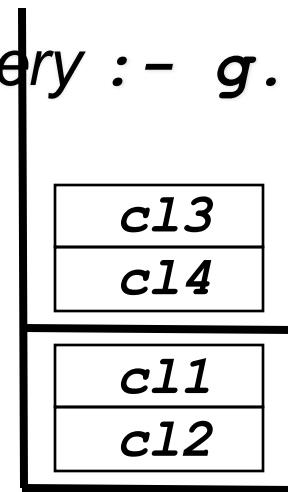
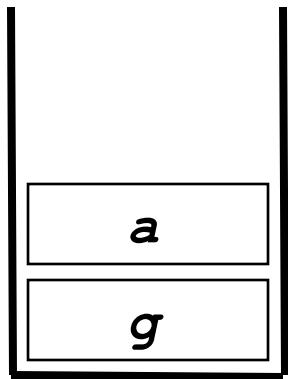
← Scelta corrente

r ha successo

EFFETTO DEL CUT

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

- Valutazione della query $: - g.$



Stack di esecuzione

Stack di backtracking

p ha successo

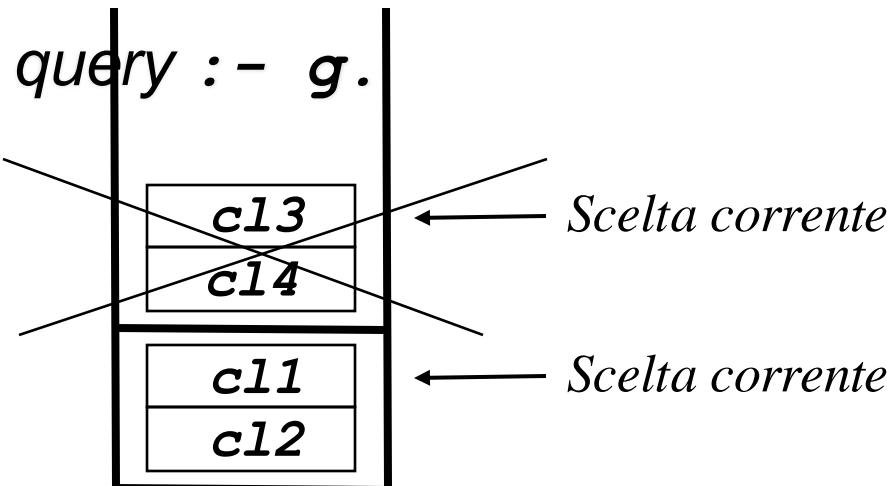
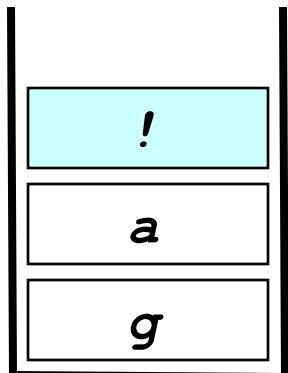
EFFETTO DEL CUT

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

Effetto del !

Tutti i punti di scelta per p e per a
sono rimossi dallo stack

- Valutazione della query $: - g.$



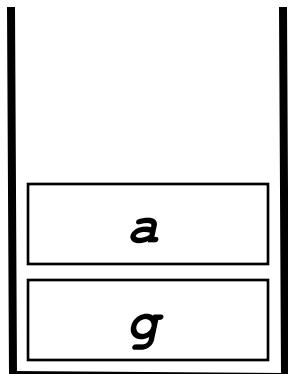
Stack di esecuzione

Stack di backtracking

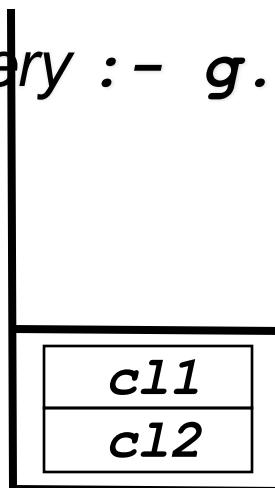
EFFETTO DEL CUT

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

- Valutazione della query $: - g.$



Stack di esecuzione



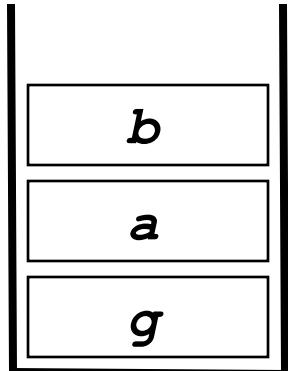
Stack di backtracking

← Scelta corrente

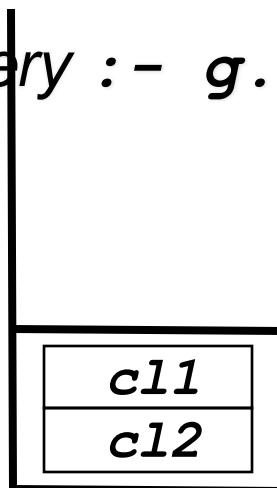
EFFETTO DEL CUT

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

- Valutazione della query $: - g.$



Stack di esecuzione



Stack di backtracking

b fallisce

EFFETTO DEL CUT

(c11) $g :- a.$

(c12) $g :- s.$

(c13) $a :- p, !, b.$

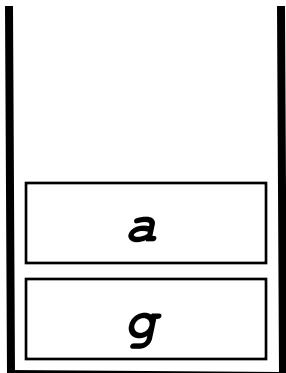
(c14) $a :- r.$

(c15) $p :- q.$

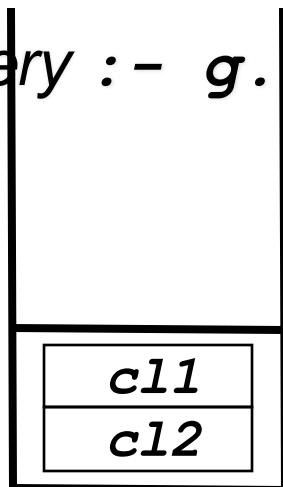
(c16) $p :- r.$

(c17) $r.$

- Valutazione della query $: - g.$



Stack di esecuzione



Stack di backtracking

e fallisce
anche a

Scelta corrente

EFFETTO DEL CUT

(c11) $g :- a.$

(c12) $g :- s.$

(c13) $a :- p, !, b.$

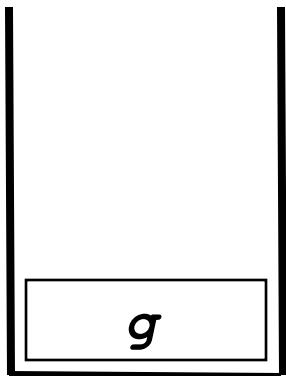
(c14) $a :- r.$

(c15) $p :- q.$

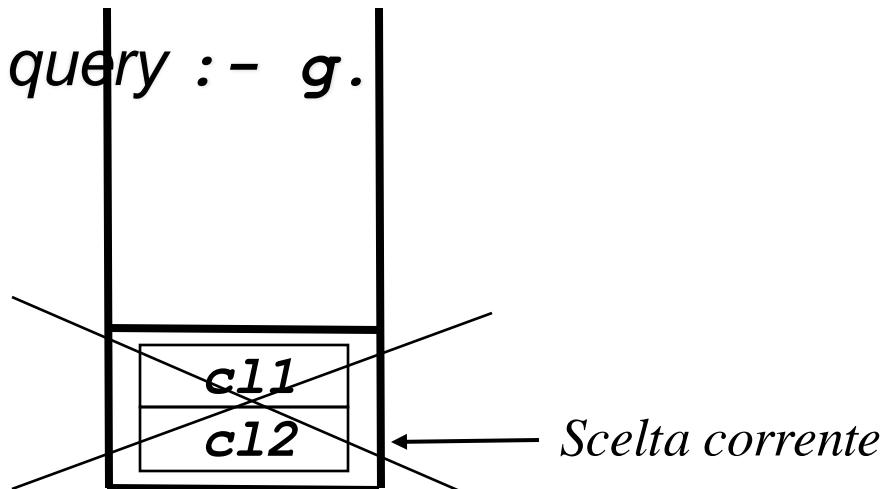
(c16) $p :- r.$

(c17) $r.$

- Valutazione della query $: - g.$



Stack di esecuzione



Stack di backtracking

EFFETTO DEL CUT

(c11) $g :- a.$

(c12) $g :- s.$

(c13) $a :- p, !, b.$

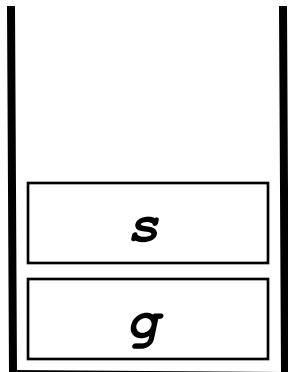
(c14) $a :- r.$

(c15) $p :- q.$

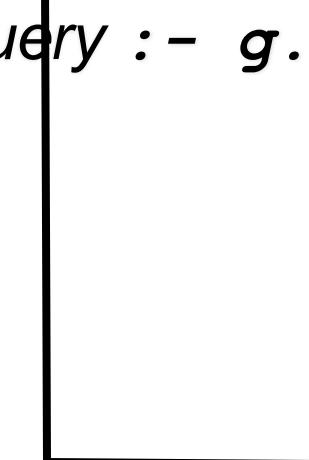
(c16) $p :- r.$

(c17) $r.$

- Valutazione della query $: - g.$



Stack di esecuzione



Stack di backtracking

s fallisce

Senza il cut la query
avrebbe avuto

ESEMPIO

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

```
a(0, 0).
```

```
b(1).
```

```
b(2).
```

```
c(1).
```

```
c(2).
```

```
:- a(X, Y).
```

```
X = 1  Y = 1;
```

```
X = 1  Y = 2.
```

ESEMPIO – SENZA CUT

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

```
a(0, 0).
```

```
b(1).
```

```
b(2).
```

```
c(1).
```

```
c(2).
```

```
:- a(X, Y).
```

```
X = 1  Y = 1;
```

```
X = 1  Y = 2;
```

```
X = 2  Y = 1;
```

```
X = 2  Y = 2;
```

```
X = 0  Y = 0.
```

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 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(X) then b else c

MUTUA ESCLUSIONE TRA CLAUSOLE

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

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

$p(X) :- c.$

n

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

if a(X) then b else c

- Utilizzando il cut:

$p(X) :- a(X), !, b.$

$p(X) :- c.$

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).*

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 è `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).  
S=[2,3].
```

CUT PER RAGIONI DI EFFICIENZA

- Anche quando le regole sono state già rese mutuamente esclusive grazie al controllo di tutte le condizioni opportune, l'utilizzo del cut è utile per ragioni di efficienza
- L'interprete infatti non ha alcun modo di sapere in anticipo che le regole sono mutuamente esclusive
- Ciò comporta che i punti di scelta restano comunque nello stack di backtracking, anche se porteranno a fallimento
- Inoltre, per poter realizzare l'ottimizzazione delle funzioni con ricorsione tail, la chiamata ricorsiva deve essere l'ultima istruzione possibile, senza alcun punto di scelta

CUT PER RAGIONI DI EFFICIENZA

- Utilizzare il cut nei casi base: *in questo modo le possibilità alternative non verranno esplorate in fase di backtracking*
- Utilizzare il cut nelle regole, quando si è sicuri che dopo il controllo di una condizione, non è utile procedere alla valutazione di soluzioni alternative
- **Attenzione:** domandarsi sempre se le soluzioni alternative possono essere ignorate o meno
- Esempio: per la relazione `zio(Zio, Nipote)` siamo interessati a conoscere tutte le soluzioni alternative

ESEMPIO (1)

```
my_numlist(High, High, [High]).  
my_numlist(Low, High, [Low | Rest]) :-  
    Low < High,  
    Low1 is Low + 1,  
    my_numlist(Low1, High, Rest).
```

- La condizione di mutua esclusione è l'unificazione dell'estremo inferiore *Low* con l'estremo superiore *High*

```
my_numlist(High, High, [High]) :- !.  
my_numlist(Low, High, [Low | Rest]) :-  
    Low < High,  
    Low1 is Low + 1,  
    my_numlist(Low1, High, Rest).
```

ESERCIZIO - MAKE_PAIRS

Definire un predicato:

make_pairs(Element, List, List_of_Pairs)

che restituisca una lista composta dalle coppie ordinate [Element, X] con X in List, per tutti gli elementi di List.

- *Esempio 1: make_pairs(a, [b, c, d], Pairs)
deve restituire Pairs = [[a, b], [a, c], [a, d]].*
- *Esempio 2: make_pairs(a, [], Pairs)
deve restituire Pairs = [] .*

SOLUZIONE – MAKE_PAIRS

```
% make_pairs(Element, List, List_of_Pairs)

% Caso base: la lista di coppie formate da un qualsiasi
% elemento e dalla lista vuota è la lista vuota
make_pairs(_Element, [], []):- !.

% Il risultato è una lista di liste che ha come testa la
% lista formata dai due elementi Element e Head e come
% coda il risultato della chiamata ricorsiva su Rest
make_pairs(Element, [Head | Rest], [[Element, Head] |
Rest_Pairs]) :-
    make_pairs(Element, Rest, Rest_Pairs).
```

ESERCIZIO - PRODOTTO CARTESIANO

Definire un predicato:

cartesian_prod(List1, List2, Result)

per creare una lista formata da tutte le coppie ordinate [X, Y] con X in List1 e Y in List2.

- *Esempio:* *cartesian_prod([a, b], [c, d], Result)*
dove restituire Result = [[a, c], [a, d], [b, c], [b, d]].
- *Suggerimento:* *utilizzare il predicato make_pairs per ogni elemento di List1.*

SOLUZIONE - PRODOTTO CARTESIANO

```
% cartesian_prod(List1, List2, Result)

% Caso base: se la prima lista è vuota, restituiamo la
lista vuota

cartesian_prod([], _List2, []):- !.

% Costruiamo la lista delle coppie con primo elemento
Head1 e secondo elemento preso da List2 e vi appendiamo
il risultato della chiamata ricorsiva su Rest1

cartesian_prod([Head1 | Rest1], List2, Result) :-
    make_pairs(Head1, List2, List_Of_Pairs),
    cartesian_prod(Rest1, List2, Rest_Product),
    append(List_Of_Pairs, Rest_Product, Result).
```

ESEMPIO (2)

```
except_last([_Last], []).

except_last([Head | Rest], [Head | Rest_Except_Last]) :-  
    except_last(Rest, Rest_Except_Last).
```

- La condizione di mutua esclusione è l'aver raggiunto una lista con un solo elemento

```
except_last([_Last], []) :- !.

except_last([Head | Rest], [Head | Rest_Except_Last]) :-  
    except_last(Rest, Rest_Except_Last).
```

ESEMPIO (3)

```
delete1(_Element, [], []).  
  
delete1(Element, [Element | Rest], Rest).  
  
delete1(Element, [Head | Rest], [Head | New_Rest]) :-  
    Element \== Head,  
    delete1(Element, Rest, New_Rest).
```

- Le condizioni di mutua esclusione sono:
 1. l'aver raggiunto la lista vuota
 2. l'unificazione dell'elemento da cancellare con la testa della lista

```
delete1(_Element, [], []) :- !.  
  
delete1(Element, [Element | Rest], Rest) :- !.  
  
delete1(Element, [Head | Rest], [Head | New_Rest]) :-  
    delete1(Element, Rest, New_Rest).
```

ESEMPIO (4)

```
delete_all(_Element, [], []).  
  
delete_all(Element, [Element | Rest], New_Rest) :-  
    delete_all(Element, Rest, New_Rest).  
  
delete_all(Element, [Head | Rest], [Head | New_Rest]) :-  
    Element \== Head,  
    delete_all(Element, Rest, New_Rest).
```

- Le condizioni di mutua esclusione sono:
 1. l'aver raggiunto la lista vuota
 2. l'unificazione dell'elemento da cancellare con la testa della lista

```
delete_all(_Element, [], []) :- !.  
  
delete_all(Element, [Element | Rest], New_Rest) :-  
    !,  
    delete_all(Element, Rest, New_Rest).  
  
delete_all(Element, [Head | Rest], [Head | New_Rest]) :-  
    delete_all(Element, Rest, New_Rest).
```

ESEMPIO (5)

- 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|_]) :- !.
```

FARE ATTENZIONE QUANDO SI USA IL CUT!

- Programma per aggiungere un elemento ad una lista solo se l'elemento non è già presente

```
add(Element, List, List) :-
```

```
    member(Element, List),
```

```
!.
```

```
add(Element, List, [Element | List]).
```

```
: - add(4, [1, 2, 3], Result).
```

```
true  Result = [4, 1, 2, 3]
```

```
: - add(2, [1, 2, 3], Result).
```

```
true  Result = [1, 2, 3]
```

FARE ATTENZIONE QUANDO SI USA IL CUT!

- Programma per aggiungere un elemento ad una lista solo se l'elemento non è già presente

```
add(Element, List, List) :-  
    member(Element, List),  
    !.  
  
add(Element, List, [Element | List]).  
  
:- add(4, [1, 2, 3], Result).  
true  Result = [4, 1, 2, 3]  
  
:- add(2, [1, 2, 3], Result).  
true  Result = [1, 2, 3]  
  
:- add(1, [1, 2, 3], [1, 1, 2, 3]).  
true  (soluzione non corretta!)
```

FARE ATTENZIONE QUANDO SI USA IL CUT! – SOLUZIONE

- Effettuare l'unificazione tra *List* e *Result* dopo il cut

```
add(Element, List, Result) :-
```

```
member(Element, List),
```

```
!,
```

```
Result = List.
```

```
add(Element, List, [Element | List]).
```

```
: - add(1, [1, 2, 3], [1, 1, 2, 3]).
```

false

NEGAZIONE IN PROLOG

- La forma di negazione introdotta in Prolog è la negazione per fallimento
- Può essere realizzata facilmente scambiando tra loro la nozione di successo e di fallimento
- L'operatore utilizzato in SWI-Prolog è `\+`
- Il meccanismo di valutazione di una query negativa `\+ Q` è definito in Prolog nel modo seguente: si valuta la controparte positiva della query `Q`. Se `Q` fallisce, si ha successo, mentre se `Q` ha successo la negazione fallisce
- È possibile implementare la negazione utilizzando il cut:

```
not(X) :- X, !, false.
```

```
not(X) .
```

ESEMPIO

- Consideriamo il seguente programma:

disoccupato(X) :-

\+ occupato(X),

adulto(X).

occupato(luigi).

adulto(mario).

adulto(luigi).

- E la query:

: - disoccupato(mario).

true

ESEMPIO

- Consideriamo il seguente programma:

disoccupato(X) :-

\+ occupato(X),

adulto(X).

occupato(luigi).

adulto(mario).

adulto(luigi).

- Se invece consideriamo la query:

: - disoccupato(X).

false ... perché?

ESEMPIO

- Consideriamo il seguente programma:

```
disoccupato(X) :-  
    \+ occupato(X),  
    adulto(X).
```

```
occupato(luigi).  
adulto(mario).  
adulto(luigi).
```

- Se invece consideriamo la query:

```
: - disoccupato(X).
```

false ... perché?

Il sottogoal

\+ occupato(X)

fallisce, perché tramite la sostituzione **X=luigi**,

il sottogoal

occupato(X)

ha successo!

SOLUZIONE

- Nell'utilizzo della negazione per fallimento, prestare attenzione a non avere mai variabili non istanziate al momento della valutazione dell'operatore \+:

disoccupato (X) :-

adulto (X) ,

% invertendo l'ordine

\+ occupato (X) . *% adesso X è già stata istanziata da adulto*

occupato (luigi) .

adulto (mario) .

adulto (luigi) .

- Adesso la query:

: - disoccupato (X) .