Il Linguaggio PROLOG

- PROLOG: **PRO**gramming in **LOG**ic, nato nel 1973
- E' il più noto linguaggio di Programmazione Logica

  ALGORITMO = LOGICA + CONTROLLO

- Si fonda sulle idee di Programmazione Logica avanzate da R. Kowalski
- Basato sulla logica dei Predicati del Primo Ordine (dimostrazione automatica di teoremi - risoluzione)
- Manipolatore di SIMBOLI e non di NUMERI
- Linguaggio di alto livello, dichiarativo, utile per prototipazione rapida
- Applicazioni di IA

Caratteristiche

- Lavora su strutture ad ALBERO
  - anche i programmi sono strutture dati manipolabili
  - utilizzo della ricorsione
  - unificazione, e non assegnamento
- Metodologia di programmazione:
  - concentrarsi sulla specifica del problema rispetto alla strategia di soluzione
- Svantaggi:
  - efficienza (compilato *vs* interpretato)
  - non adatto ad applicazioni numeriche o in tempo reale
  - mancanza di ambienti di programmazione evoluti

Risoluzione – teorie a clausole

$$\frac{A_1 \vee \ldots \vee A_n \quad\_\_\_\_\quad B_1 \vee \ldots \vee B_m \quad\_\_\_\_\quad \exists\theta : [A_i]\theta = [\sim B_j]\theta}{[A_1 \vee \ldots \vee A_{i-1} \vee A_{i+1} \vee \ldots \vee A_n \vee B_1 \vee \ldots \vee B_{j-1} \vee B_{j+1} \vee \ldots \vee B_m]\theta}$$

ove θ è la sostituzione unificatrice più generale per A_i e $\sim B_j$.

- ***Esempio***

Si considerino le seguenti clausole:

p(X,m(a)) v q(S,a) v c(X) (C1)

~p(r,Z) v c(Z) v ~b(W,U) (C2)

I letterali p(X,m(a)) e p(r,Z) sono unificabili attraverso la mgu θ={X/r, Z/m(a)}.

Risolvente:

q(S,a) v c(r) v c(m(a)) v ~b(W,U) (C3)

PROGRAMMAZIONE LOGICA

- Dalla Logica dei predicati del primo ordine verso un linguaggio di programmazione;
  - requisito efficienza

- Si considerano solo clausole di Horn (al più un letterale positivo)
  - il letterale positivo corrisponde alla testa della clausola

- Si adotta una strategia risolutiva particolarmente efficiente
  - RISOLUZIONE SLD (vedremo che corrisponde al Backward chianing per clausole di Horn).
  - Non completa per la logica a clausole, ma completa per il sottoinsieme delle clausole di Horn.

Risoluzione per clausole di Horn, con strategia linear input (SLD)

- KB di clausole definite (esattamente un letterale positivo)

$$\beta \leftarrow \alpha_1 \wedge \ldots \wedge \alpha_n$$

- e Horn (goal, nessun letterale positivo)

$$\leftarrow \beta_1 \wedge \ldots \wedge \beta' \wedge \ldots \wedge \beta_m$$

- Tutte le variabili sono quantificate universalmente

$$\frac{\sim \beta_1 \vee \ldots \vee \sim \boldsymbol{\beta'} \vee \ldots \vee \sim \beta_m \qquad \sim(\alpha_1 \wedge \ldots \wedge \alpha_n) \vee \boldsymbol{\beta} \quad \text{dove } \boldsymbol{\beta'\theta = \beta\theta}}{[\sim \beta_1 \vee \ldots \vee \sim \alpha_1 \vee \ldots \vee \sim \alpha_n \vee \ldots \vee \sim \beta_m]\,\theta}$$

- **L**inear Resolution with **S**election Function for **D**efinite Clauses - Programmazione Logica e Prolog
- *β' letterale selezionato nel goal (in Prolog quello più a sinistra)*

Strategia Linear-input (*recap*)

- Strategia **linear-input** (non completa) sceglie sempre una clausola "parent" nell'insieme base C_0 (il programma) mentre la seconda clausola "parent" è il risolvente derivato al passo precedente (il goal).
- Caso particolare della risoluzione lineare:
  - vantaggio: memorizzare solo l'ultimo risolvente
  - svantaggio: non completa nel caso di clausole, ma **completa per clausole di Horn**

RISOLUZIONE SLD

- Dato un programma logico **P** (insieme di clausole definite) e una clausola goal G_0 (clausola Horn), ad ogni passo di risoluzione si ricava , se esiste, un nuovo risolvente G_{i+1} **ottenuto** dalla clausola goal del passo precedente G_i e da una *variante* di una clausola appartenente al programma **P**

- Una variante per una clausola C e' la clausola C' ottenuta da C rinominando le sue variabili (*renaming*)
  - Esempio:

```
p(X):- q(X,g(Z)).
p(X1):- q(X1,g(Z1)).
```

RISOLUZIONE SLD –backward chaining (continua)

- La Risoluzione SLD seleziona un atomo **`Am`** dal goal **`Gi`** secondo un determinato criterio (***funzione di selezione*** o regola di calcolo), e lo unifica se possibile con la testa della clausola **`Ci`** attraverso la *sostituzione* più generale (**MOST GENERAL UNIFIER, MGU**) θ_i

- Il nuovo risolvente è ottenuto da Gi riscrivendo l'atomo selezionato con la parte destra della clausola Ci ed applicando la sostituzione θ_i.

- Più in dettaglio, alla Prolog:

  $:- A_1, \ldots, A_{m-1}, A_m, A_{m+1}, \ldots, A_k.$ Risolvente

  $A :- B_1, \ldots, B_q.$ Clausola del programma P

  e $[A_m]\theta_i = [A]\ \theta_i$ allora la risoluzione SLD deriva il nuovo risolvente

  $:- [A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k\ ]\theta_i.$

RISOLUZIONE SLD –backward chaining (continua)

- La Risoluzione SLD seleziona un atomo **Am** dal goal **Gi** secondo un determinato criterio (***funzione di selezione*** o regola di calcolo), e lo unifica se possibile con la testa della clausola **Ci** attraverso la *sostituzione* più generale (**MOST GENERAL UNIFIER, MGU**) θ_i

- Il nuovo risolvente è ottenuto da Gi riscrivendo l'atomo selezionato con la parte destra della clausola Ci ed applicando la sostituzione θ_i.

- Prolog:

  :- $A_1, \ldots, A_{m-1}, A_m, A_{m+1}, \ldots, A_k$. Risolvente

  A:- $B_1, \ldots, B_q$. Clausola del programma P

  e $[A_1]\theta_i = [A]\ \theta_i$ allora la risoluzione SLD deriva il nuovo risolvente

  :- $[B_1, \ldots, B_q,\ A_2, \ldots, A_k\ ]\theta_i$.

Prolog: sintassi

- Un programma Prolog è costituito da un insieme di clausole definite della forma

`(cl1) A.` → **FATTO o ASSERZIONE**

`(cl2) A :- B1, B2,…, Bn.` → **REGOLA**

- In cui `A` e `Bi` sono formule atomiche
- `A` : testa della clausola
- `B1,B2,…,Bn` : body della clausola
- Il simbolo "`,`" indica la congiunzione; il simbolo "`:-`" l'implicazione logica in cui `A` e' il conseguente e `B1,B2,…,Bn` l'antecedente

La computazione parte da una clausola goal:

`(cl3) :- B1, B2,…, Bn.` → **GOAL**

ESEMPIO (proposizionale)

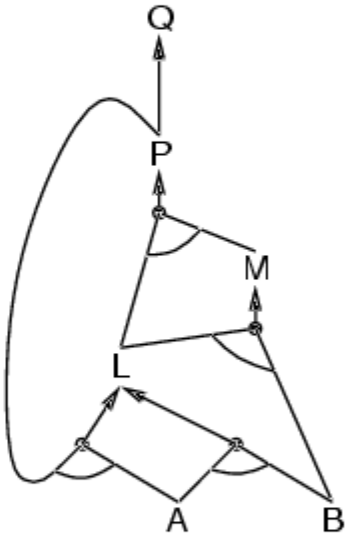
```
q:-p.
p:-l,m.
m:-b,l.
l:-a,b.
l:-a,p.
```

Regole

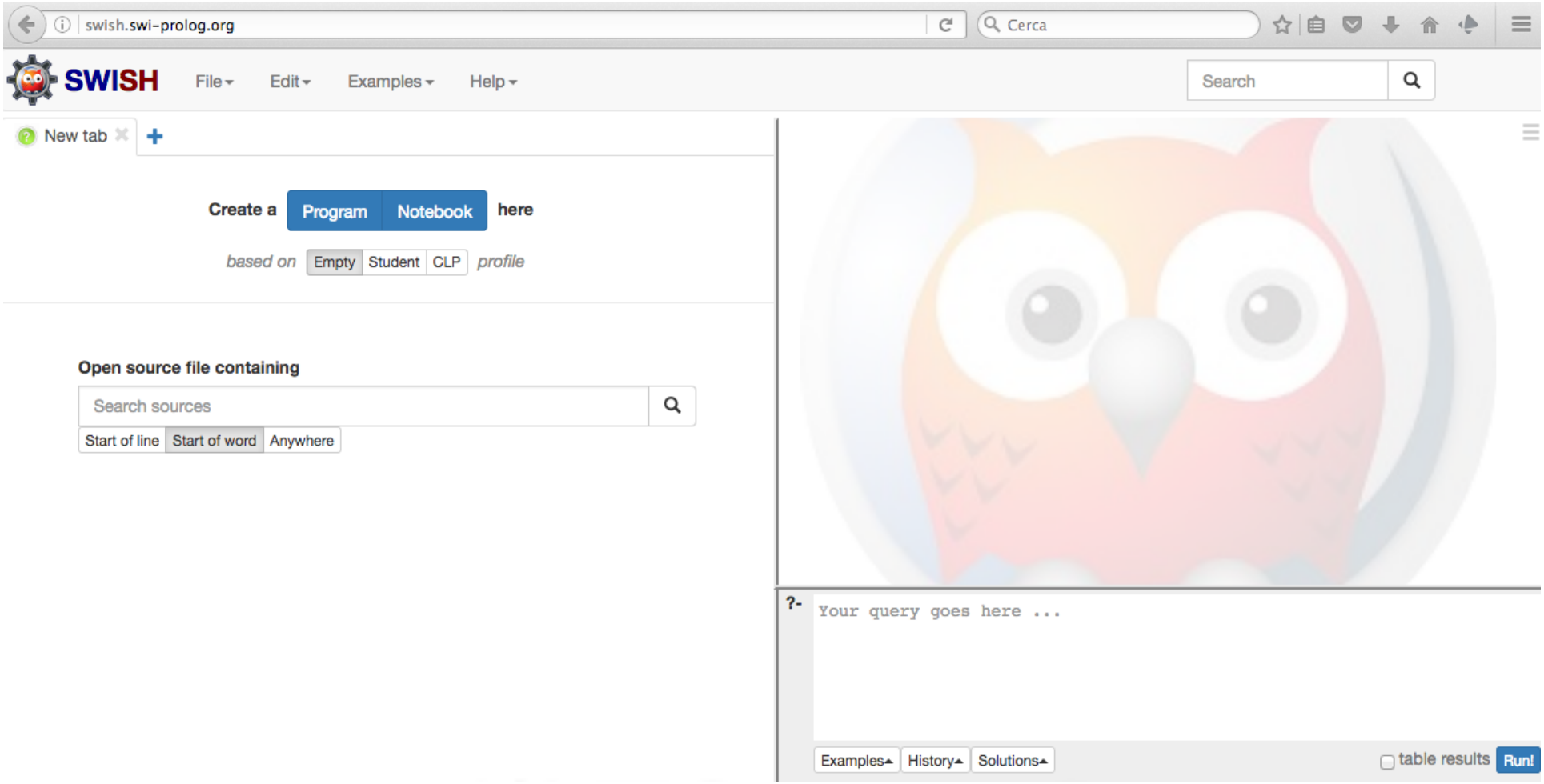
```
a.
b.
```

Fatti

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



- *Dalla query: q si ottiene il goal:*
- `:- q.` **Goal**



SWIsh Prolog, web interface di SWI Prolog (in realtà di una versione ridotta)

Oppure sui pc del laboratorio è installato SICStus Prolog

consult e listing per SICStus Prolog

```
q:-p.
p:-l,m.
m:-b,l.
l:-a,b.
l:-a,p.
a.
b.
```

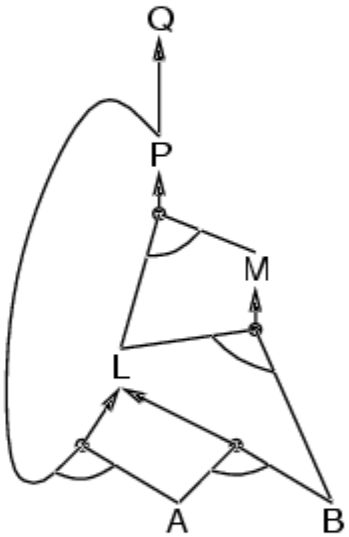
Programma (KB, Knowledge Base), file sorgente

Esempio0.pl

- `:- consult('esempio0.pl').  % ['esempio0.pl']`
- `:- listing.`

Backward chaining, per refutazione

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



$\sim Q$ *(query negata)*

$\sim P$

$\sim(L \wedge M) \equiv \sim L \vee \sim M$

$\sim(A \wedge B) \vee \sim M \equiv \sim A \vee \sim B \vee \sim M$

$\sim B \vee \sim M$

$\sim M$

$\sim(B \wedge L) \equiv \sim B \vee \sim L$

$\sim L$

$\sim(A \wedge B) \equiv \sim A \vee \sim B$

$\sim B$

clausola vuota

DIMOSTRAZIONE DI UN GOAL

- Un goal viene dimostrato dimostrando i singoli letterali da sinistra a destra

```
:- q, a, b.
:- collega(X,Y), persona(X), persona(Y).
:- sum(X,Y,s(s(s(0)))), sum(X,s(0),Y).
```

- Un goal atomico (ossia formato da un singolo letterale) viene provato confrontandolo e unificandolo con le teste delle clausole contenute nel programma

- Se esiste una sostituzione per cui il confronto ha successo
  - se la clausola con cui unifica è un fatto, la dimostrazione termina;
  - se la clausola con cui unifica è una regola, ne viene provato il Body

- Se non esiste una sostituzione il goal fallisce

DIMOSTRAZIONE: ESEMPIO 1

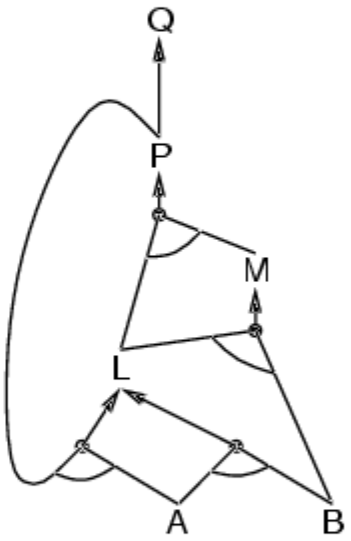
```
q:-p.
p:-l,m.
m:-b,l.
l:-a,b.
l:-a,p.
```

Regole

```
a.
b.
```

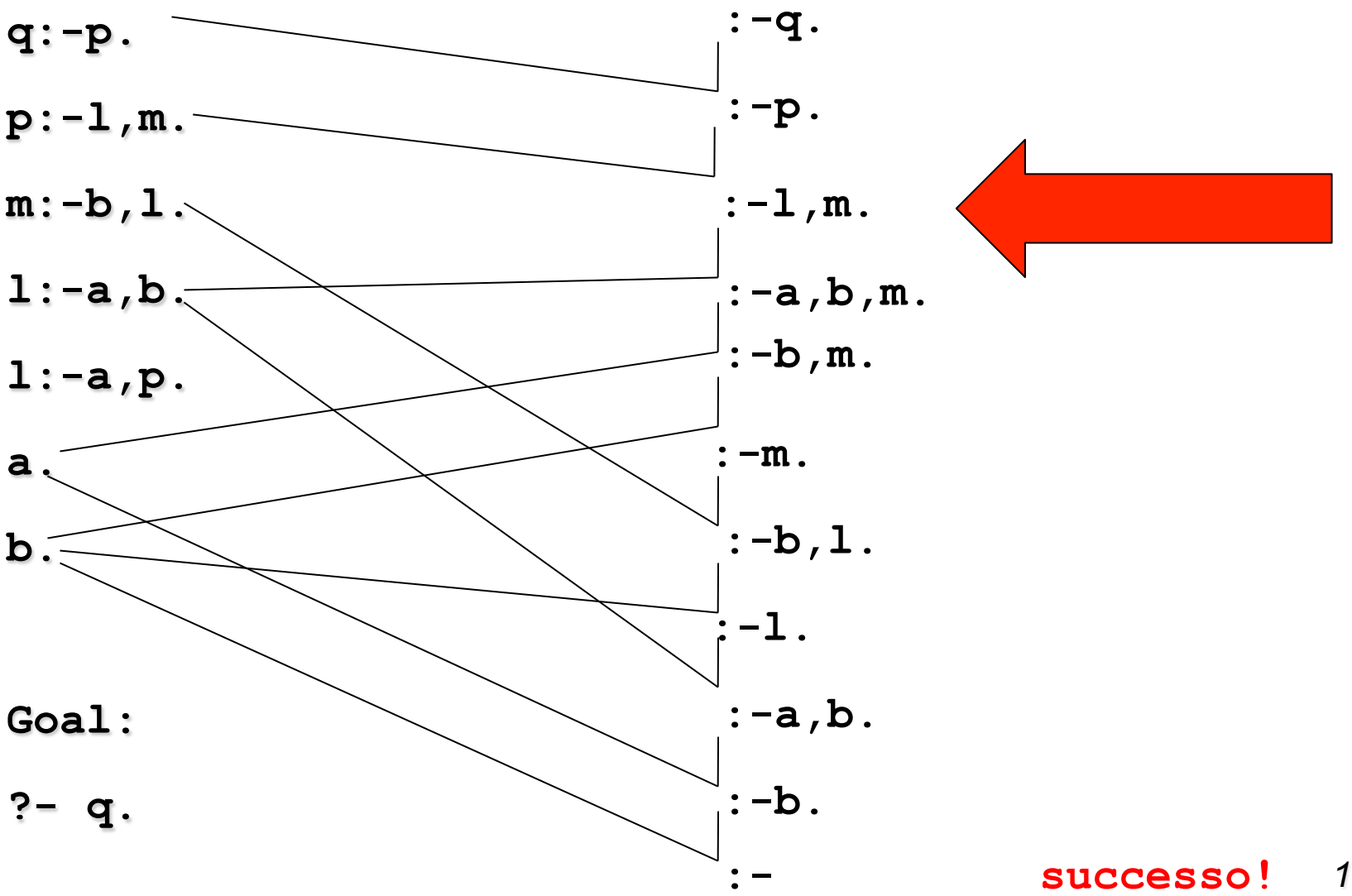
Fatti

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



- *Dalla query: q si ottiene il goal:*
- `:- q.` **Goal**

DIMOSTRAZIONE: ESEMPIO 1



successo!

Dimostrazione (query)

```
q:-p.
p:-l,m.
m:-b,l.
l:-a,b.
l:-a,p.
a.
b.
```

esempio0.pl

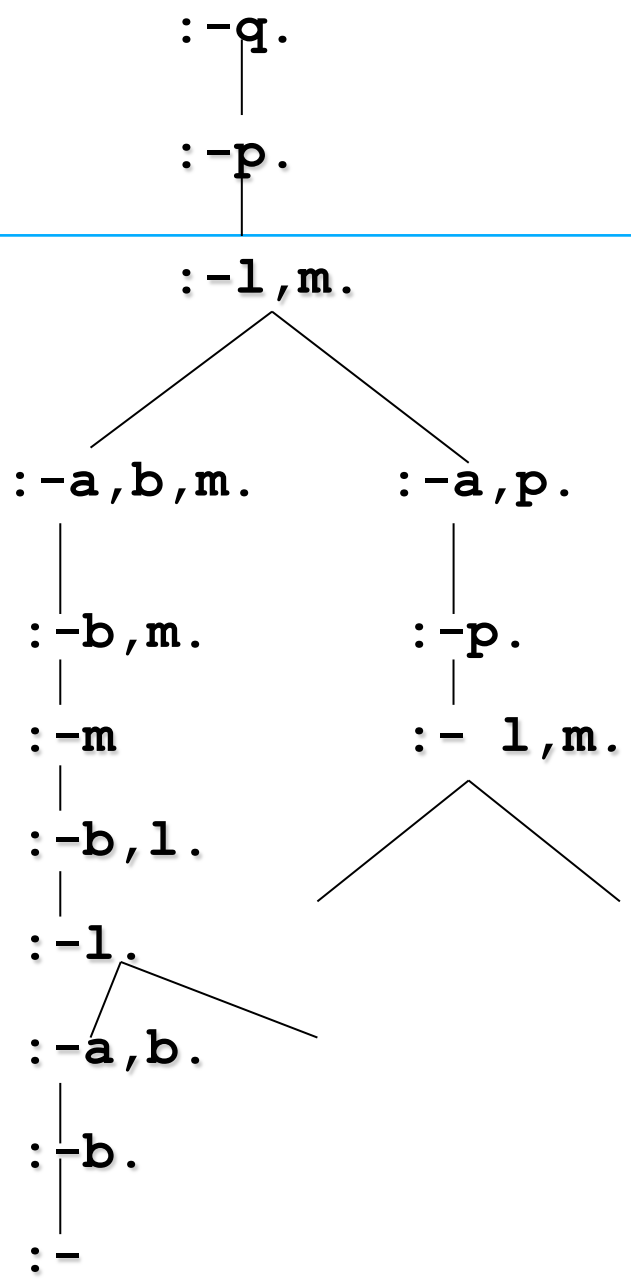
- `:- consult('esempio0.pl').  % ['esempio0.pl']`
- `:- listing.`
- `:- q.`

Goal

- `:- trace.  % :- notrace.`

ESEMPIO 1

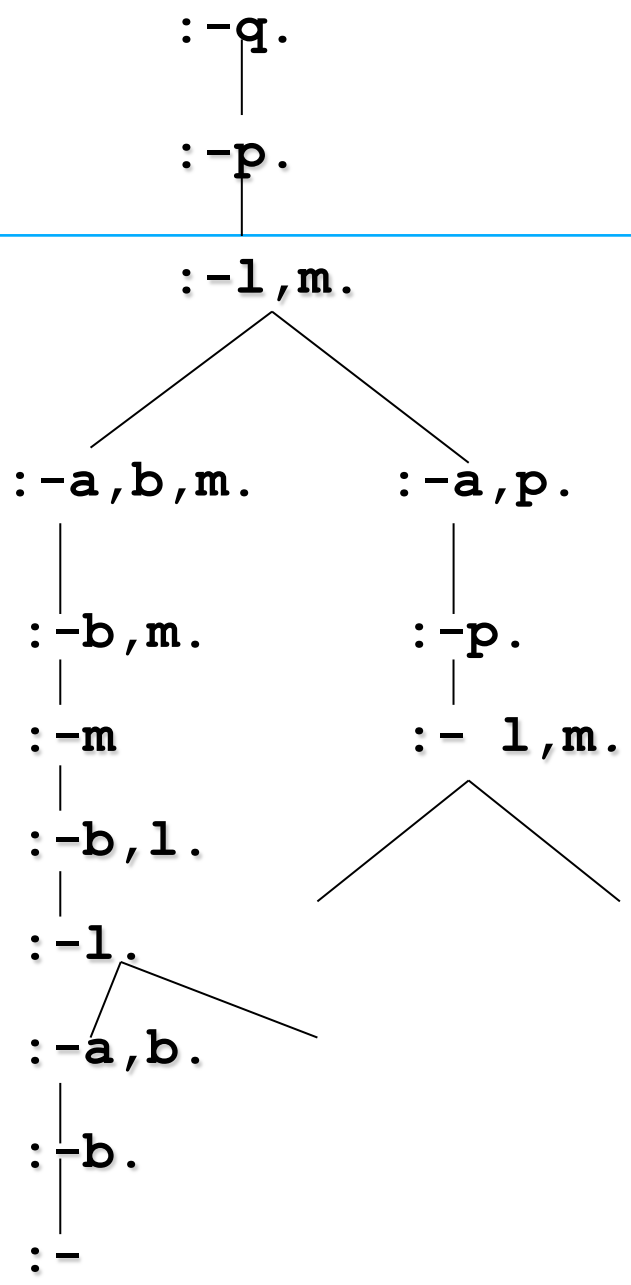
```
q:-p.

p:-l,m.

m:-b,l.

l:-a,b.

l:-a,p.

a.

b.

Goal:

?- q.

Yes    ;
```



successo

Albero SLD

- Albero OR
- Ha come radice il goal
- Rami di:
  - successo
  - fallimento
    - Finito
    - Infinito
- Prolog, lo spazio di ricerca rappresentato dall'albero SLD è esplorato con depth-first search con backtracking



PROGRAMMA PROLOG

- Un PROGRAMMA PROLOG è un insieme di clausole di Horn che rappresentano:
  - FATTI riguardanti gli oggetti in esame e le relazioni che intercorrono
  - REGOLE sugli oggetti e sulle relazioni (SE…..ALLORA)
  - GOAL (clausole senza testa), negazione della query sulla base della conoscenza definita
- ESEMPIO: due individui sono colleghi hanno un collega comune

```
collega(a,b).
collega(b,c).
```

FATTI

Testa — **Body (Corpo)**

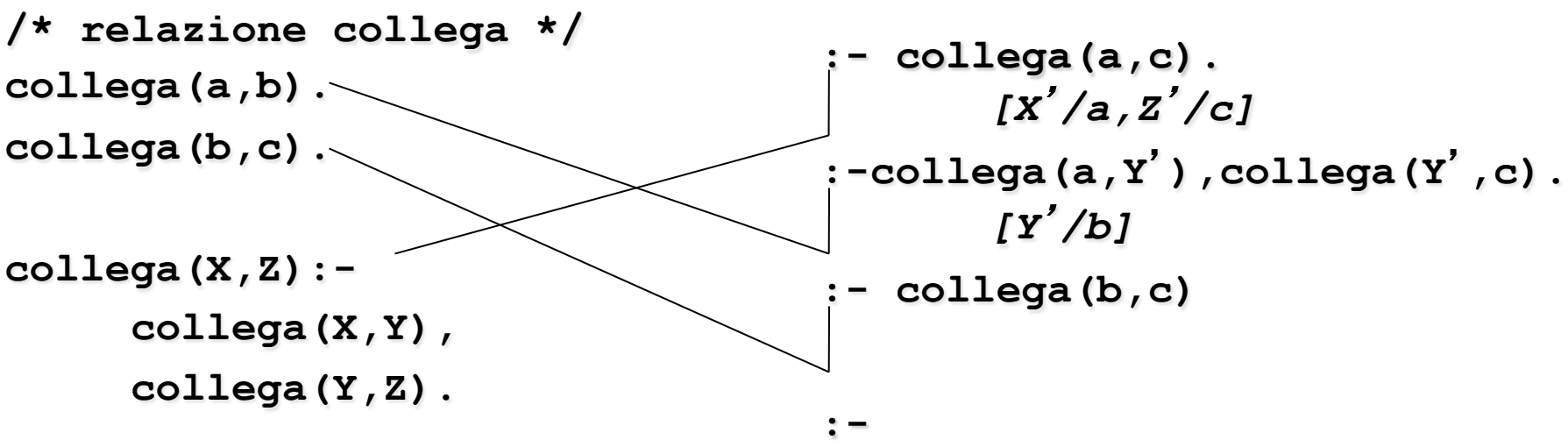
```
collega(X,Y):- collega(X,Z), collega(Z,Y).
```

→ **REGOLA**

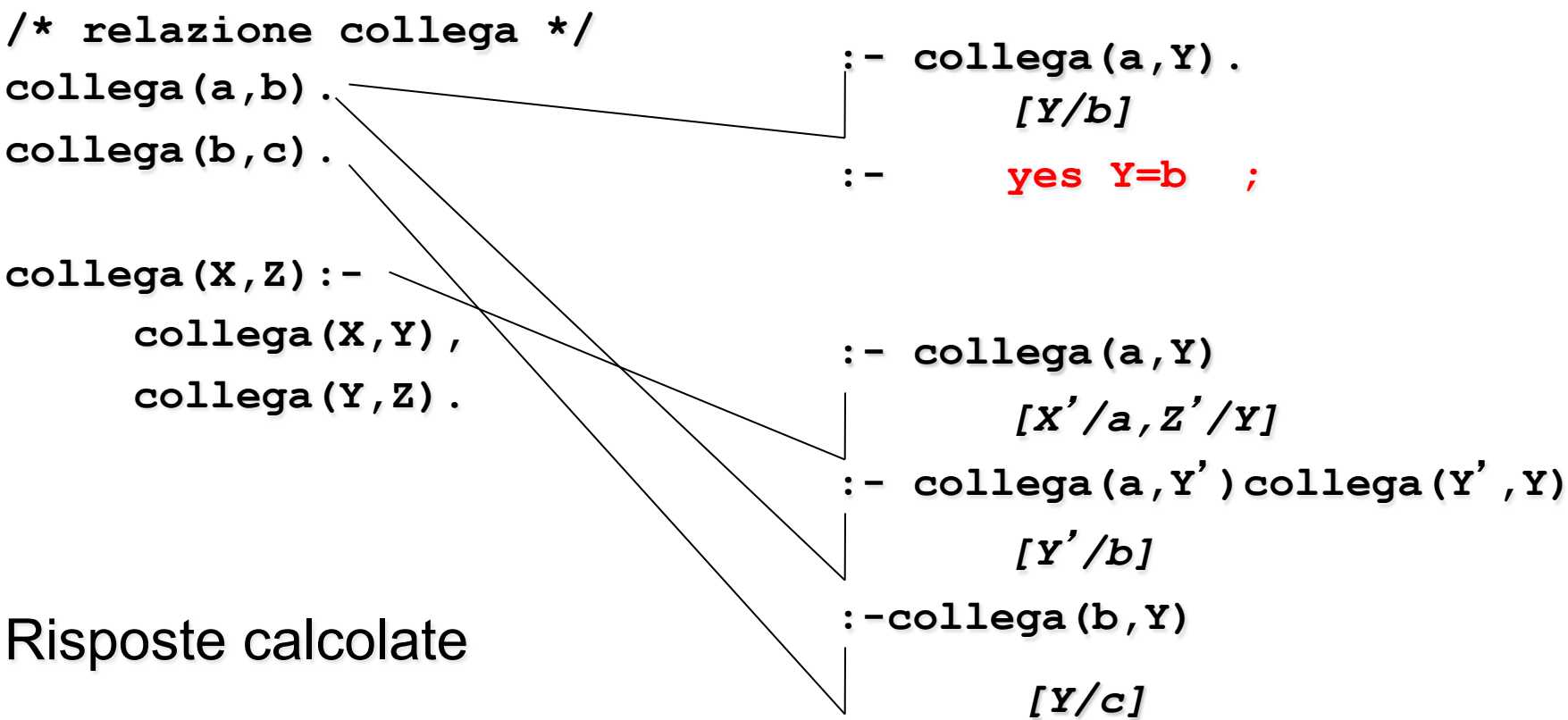
```
:- collega(X,Y).
```

→ **GOAL**

Dimostrazione: esempio 2



Dimostrazione: esempio 2 (cont.)

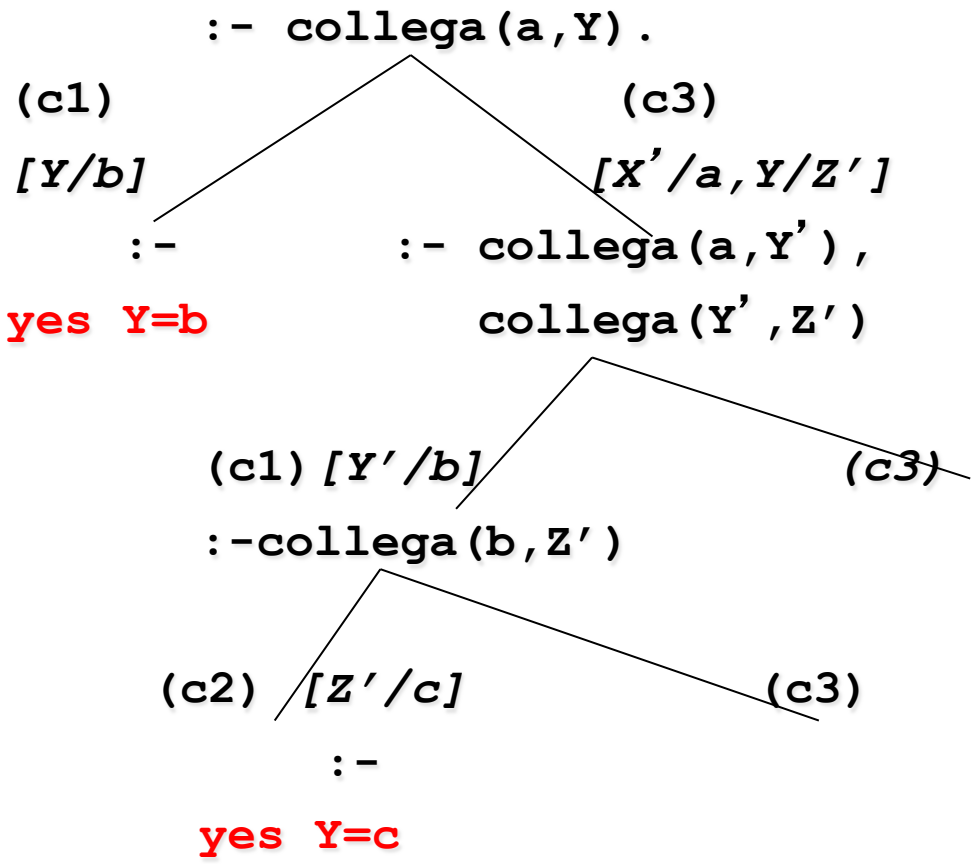


Risposte calcolate

Non determinismo (punti di scelta)

Albero SLD: esempio 2 (*cont.*)

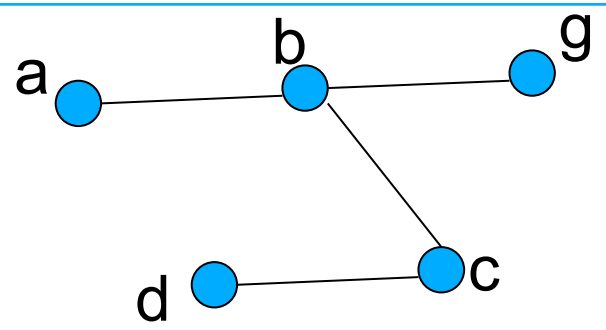
```
/* relazione collega */
collega(a,b).
collega(b,c).

collega(X,Z):-
      collega(X,Y),
      collega(Y,Z).
```



Risposte calcolate

Non determinismo (punti di scelta)

Prolog – esempio 0




```
arco(a,b).
arco(b,c).
arco(c,d).
arco (b,g).
connesso(X,Y) :- arco(X,Y).
connesso(X,Y) :- arco(X,Z), connesso(Z,Y).
```

- *Posso verificare <u>automaticamente</u> se "a è connesso a d"?*

```
?-connesso(a,d).
?-connesso(a,Y).
?-connesso(X,Y).
```

DIMOSTRAZIONE: ESEMPIO 0

```
arco(a,b).

arco(b,c).

arco(c,d).

arco (b,g).
```

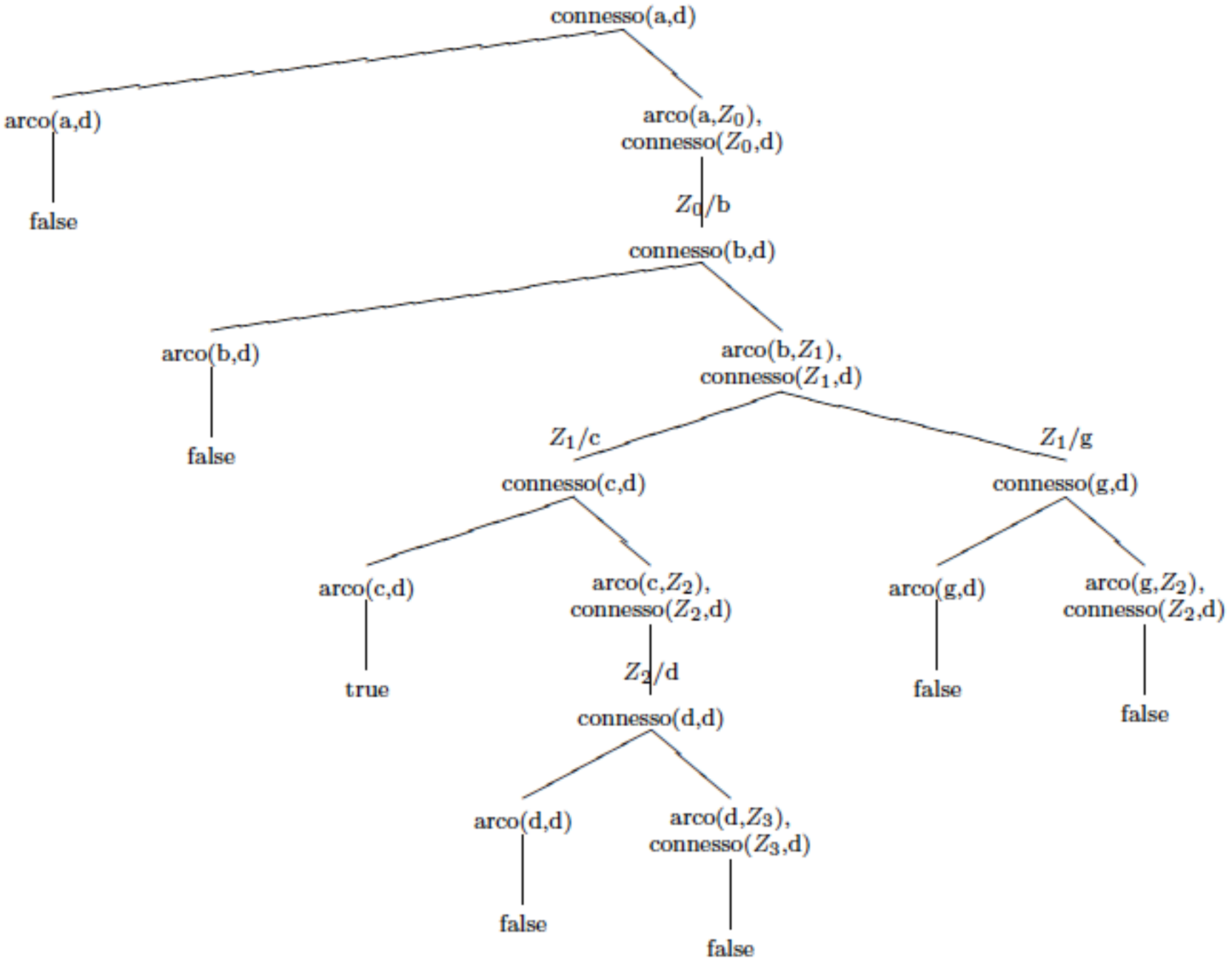
Fatti

```
connesso(X,Y):-
        arco(X,Y).

connesso(X,Y) :-
        arco(X,Z),
        connesso(Z,Y).
```

Regole

```
?-connesso(a,d).
```

Goal


connesso(a,d)
arco(a,d)
arco(a,Z_0),
connesso(Z_0,d)
false
Z_0/b
connesso(b,d)
arco(b,d)
arco(b,Z_1),
connesso(Z_1,d)
false
Z_1/c
Z_1/g
connesso(c,d)
connesso(g,d)
arco(c,d)
arco(c,Z_2),
connesso(Z_2,d)
arco(g,d)
arco(g,Z_2),
connesso(Z_2,d)
true
Z_2/d
false
false
connesso(d,d)
arco(d,d)
arco(d,Z_3),
connesso(Z_3,d)
false
false

Esercizio 1.1

- Si scriva la relazione **`collega(X,Y)`** che è vera se X e Y lavorano nella stessa ditta, e sono diversi.

Si parta dai predicati: **`lavora(X,Z)`** vero se l'impiegato X lavora nell'azienda Z, e **`X\==Y`** vero se X e Y sono diversi.

Si supponga che **`lavora(X,Y)`** sia definita dai seguenti fatti

```
lavora(emp1,ibm).
lavora(emp2,ibm).
lavora(emp3,google).
lavora(emp4,crif).
lavora(emp5,google).
```

Esercizio 1.1(cont.)

- Si consulti il file .pl e lo si interroghi con il goal:

```
?- collega(X,Y).
X=emp1 Y=emp2;
X=emp2 Y=emp1;
X=emp3 Y=emp5;
X=emp5 Y=emp3
. .
```

Soluzione Esercizio 1.1

```
lavora(emp1,ibm).
lavora(emp2,ibm).
lavora(emp3,google).
lavora(emp4,crif).
lavora(emp5,google).

collega(X,Y):-lavora(X,Z), lavora(Y,Z), X\==Y.
```

PROLOG: ELABORATORE DI SIMBOLI

- ESEMPIO: somma di due numeri interi

```
sum(0,X,X).
```
→ **FATTO**

```
sum(s(X),Y,s(Z)):- sum(X,Y,Z).
```
→ **REGOLA**

- Simbolo `sum` non interpretato.
- Numeri interi interpretati dalla struttura "successore" `s(X)`
- Si utilizza la ricorsione
- Esistono molte possibili interrogazioni (goal):

```
:- sum(s(0),s(s(0)),Y).
:- sum(s(0),Y,s(s(s(0)))).
:- sum(X,Y,s(s(s(0)))).
:- sum(X,Y,Z).
:- sum(X,Y,s(s(s(0)))), sum(X,s(0),Y).
```

Esercizio 1.2

- Si scriva un predicato **`sum(X,Y,Z)`** che è vero se Z è la somma di X e Y. Si supponga che X, Y e Z siano rappresentati utilizzando la forma s(0) per 1, s(s(0)) per 2, etc.
- Si consulti il file .pl con le definizioni e si verifichino le interrogazioni d'esempio.

- Esempi

?- sum(s(0),s(s(0)),Y).

Y=s(s(s(0)));

No

Esercizio 1.2 (cont.)

?- sum(s(0),Y,s(s(s(0)))).

Y=s(s(0));

no

?- sum(X,Y,s(s(s(0)))).

X=0, Y=s(s(s(0)));

X=s(0), Y=s(s(0));

.....

- **Tante interrogazioni possibili** (invertibilità, non esistono a priori argomenti di input e di output)

?- sum(X,Y,Z).

X=0, Z=Y;

X=s(0), Z=s(Y);

X=s(s(0)), Z=s(s(Y));

.....

.....

?- sum(X,Y,s(s(s(0)))), sum(X,s(0),Y).

.....

Esercizio 1.3

- A partire da alcuni fatti del tipo **padre(X,Y)** e **madre(X,Y)**, nei quali X è padre (madre) di Y, si definiscano le relazioni **nonno/a(X,Y)**, **bisnonno/a(X,Y)**, **nipote(X,Y)**, **pronipote(X,Y)** e **antenato(X,Y)**.
- Fatti:

```
padre(ugo,luisa).
madre(marina,luisa).
padre(mario,luigi).
madre(luisa,luigi).
padre(mario,giovanna).
madre(luisa,giovanna).
padre(marco,giorgio).
madre(maria,giorgio).
padre(giorgio,andrea).
madre(giovanna,andrea).
padre(giorgio,ilaria).
madre(giovanna,ilaria).

uomo(ugo).
uomo(mario).
...
donna(luisa).
donna(marina)
...
```

Soluzione Esercizio 1.3

```
% nonno(X,Y) X e' nonno di Y
nonno(X,Y) :- padre(X,Z),
              padre(Z,Y).
nonno(X,Y) :- padre(X,Z),
              madre(Z,Y).

% antenato(X,Y) X e' antenato di Y
antenato(X,Y):-padre(X,Y).
antenato(X,Y):-madre(X,Y).
antenato(X,Y):-padre(X,Z), antenato(Z,Y).
antenato(X,Y):-madre(X,Z), antenato(Z,Y).
```

.

Soluzione Esercizio 1.3

```
% nonno(X,Y) X e' nonno di Y
nonno(X,Y) :- padre(X,Z),
              (padre(Z,Y);
              madre(Z,Y)).

% antenato(X,Y) X e' antenato di Y
antenato(X,Y):-padre(X,Y).
antenato(X,Y):-madre(X,Y).
antenato(X,Y):-padre(X,Z), antenato(Z,Y).
antenato(X,Y):-madre(X,Z), antenato(Z,Y).
```

.

INTERPRETAZIONE DICHIARATIVA

- ESEMPI

  `?-padre(X,Y).` "esiste `X,Y` tale che `X` e' il padre di `Y`"

  `?-madre(X,Y).` "esiste `X,Y` tale che `X` e' 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 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 padre di `Z` e `Z` è la madre di `Y`"

PROLOG, PIU' FORMALMENTE:

- Linguaggio Prolog: caso particolare del paradigma di Programmazione Logica

- SINTASSI: un programma Prolog e' costituito da un insieme di clausole definite della forma

(cl1) A. ⟹ **FATTO o ASSERZIONE**

(cl2) A :- B1, B2,…, Bn. ⟹ **REGOLA**

(cl3) :- B1, B2,…, Bn. ⟹ **GOAL**

- In cui **A** e **Bi** sono formule atomiche
- **A** : testa della clausola
- **B1,B2,…,Bn** : body della clausola
- Il simbolo "," indica la congiunzione; il simbolo "**:-**" l'implicazione logica in cui **A** e' il conseguente e **B1,B2,…,Bn** l'antecedente

PIU' FORMALMENTE

- Una formula atomica è una formula del tipo

  `p(t1,t2,…,tn)`

  in cui **`p`** e' un simbolo predicativo e **`t1,t2,…,tn`** 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(t1,t2,…,tk)`** e' un termine se "**`f`**" e' un simbolo di funzione (operatore) a k argomenti e **`t1,t2,…,tk`** sono termini. **`f(t1,t2,…,tk)`** viene detta struttura

  NOTA: le costanti possono essere viste come simboli funzionali 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)`
- FORMULE ATOMICHE: `p, p(a,f(X)), p(Y), q(1)`
- CLAUSOLE DEFINITE:

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

- GOAL:

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

- Non c'e' distinzione tra costanti, simboli funzionali e predicativi.

INTERPRETAZIONE DICHIARATIVA

- Le variabili all'interno di una clausola sono quantificate universalmente
- per ogni asserzione (fatto)

  `p(t1,t2,…,tm).`

  se `X1,X2,…,Xn` sono le variabili che compaiono in `t1,t2,…,tm` il significato e': `∀X1,∀X2,…,∀Xn (p(t1,t2,…,tm))`
- per ogni regola del tipo

  `A:- B1,B2,…,Bk.`

  se `Y1,Y2,…,Yn` sono le variabili che compaiono solo nel body della regola e `X1,X2,…,Xn` sono le variabili che compaiono nella testa e nel corpo, il significato e':

  `∀X1,∀X2,…,∀Xn,∀Y1,∀Y2,…,∀Yn ((B1,B2,…,Bk) → A)`

  `∀X1,∀X2,…,∀Xn ( (∃Y1,∃Y2,…,∃Yn(B1,B2,…,Bk)) → A)`

INTERPRETAZIONE DICHIARATIVA

- ESEMPI

`?-padre(X,Y).` "esiste `X,Y` tale che `X` e' il padre di `Y`"

`?-madre(X,Y).` "esiste `X,Y` tale che `X` e' 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 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 padre di `Z` e `Z` è la madre di `Y`"

ESECUZIONE DI UN PROGRAMMA

- Una computazione corrisponde al tentativo di dimostrare, tramite la risoluzione, che una formula segue logicamente da un programma (è un teorema).
- Inoltre, si deve determinare una **sostituzione** per le variabili del goal (detto anche "query") per cui la query segue logicamente dal programma.
- Dato un programma P e la query:

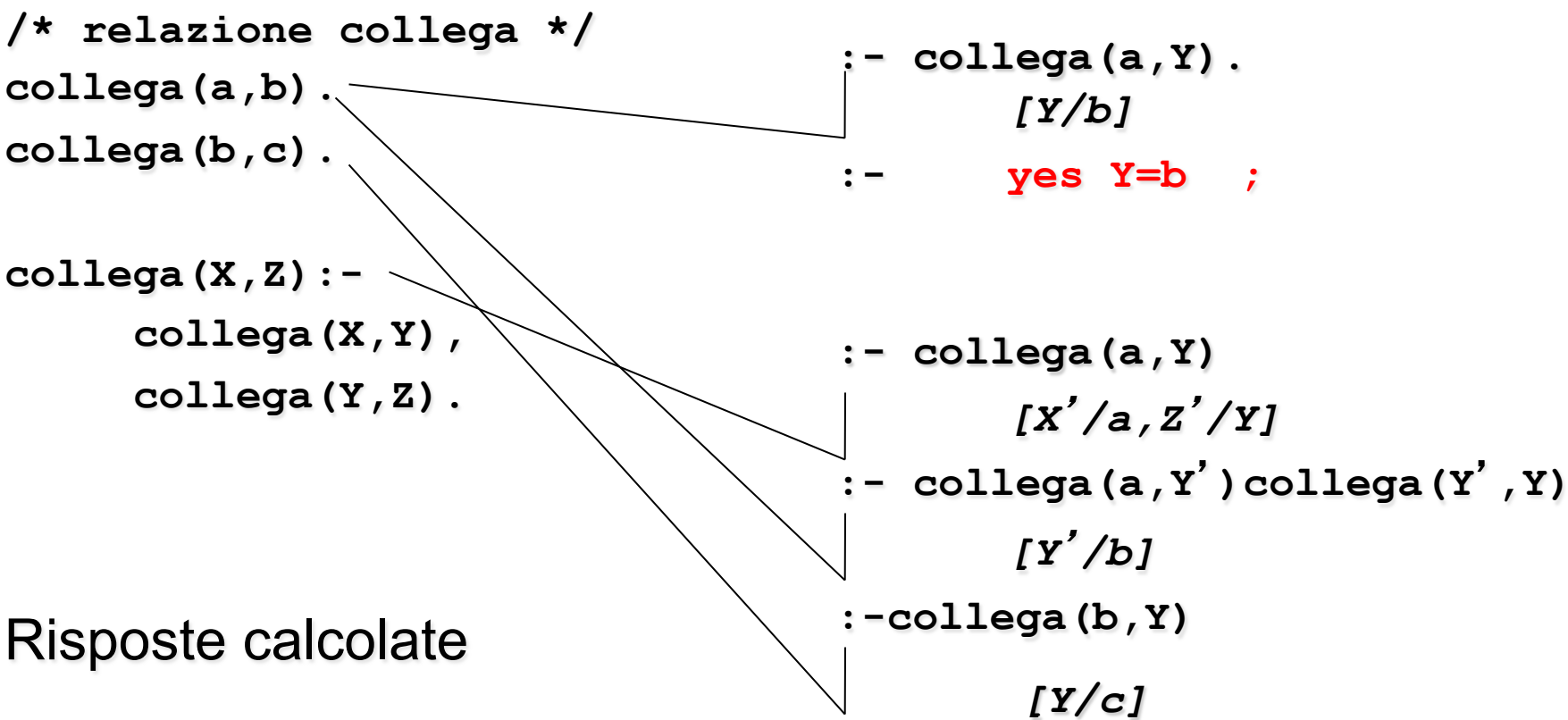
  `:- p(t1,t2,…,tm).`

  se `X1,X2,…,Xn` sono le variabili che compaiono in `t1,t2,…,tm` il significato della query e': `∃X1, ∃X2,…, ∃Xn p(t1,t2,…,tm)`
  e l'obiettivo e' quello di trovare una **sostituzione di risposta**

  σ = `{X1/s1,X2/s2,…,Xn/sn}`

  dove `si` sono termini tale per cui `P |= [p(t1,t2,…,tm)]`σ

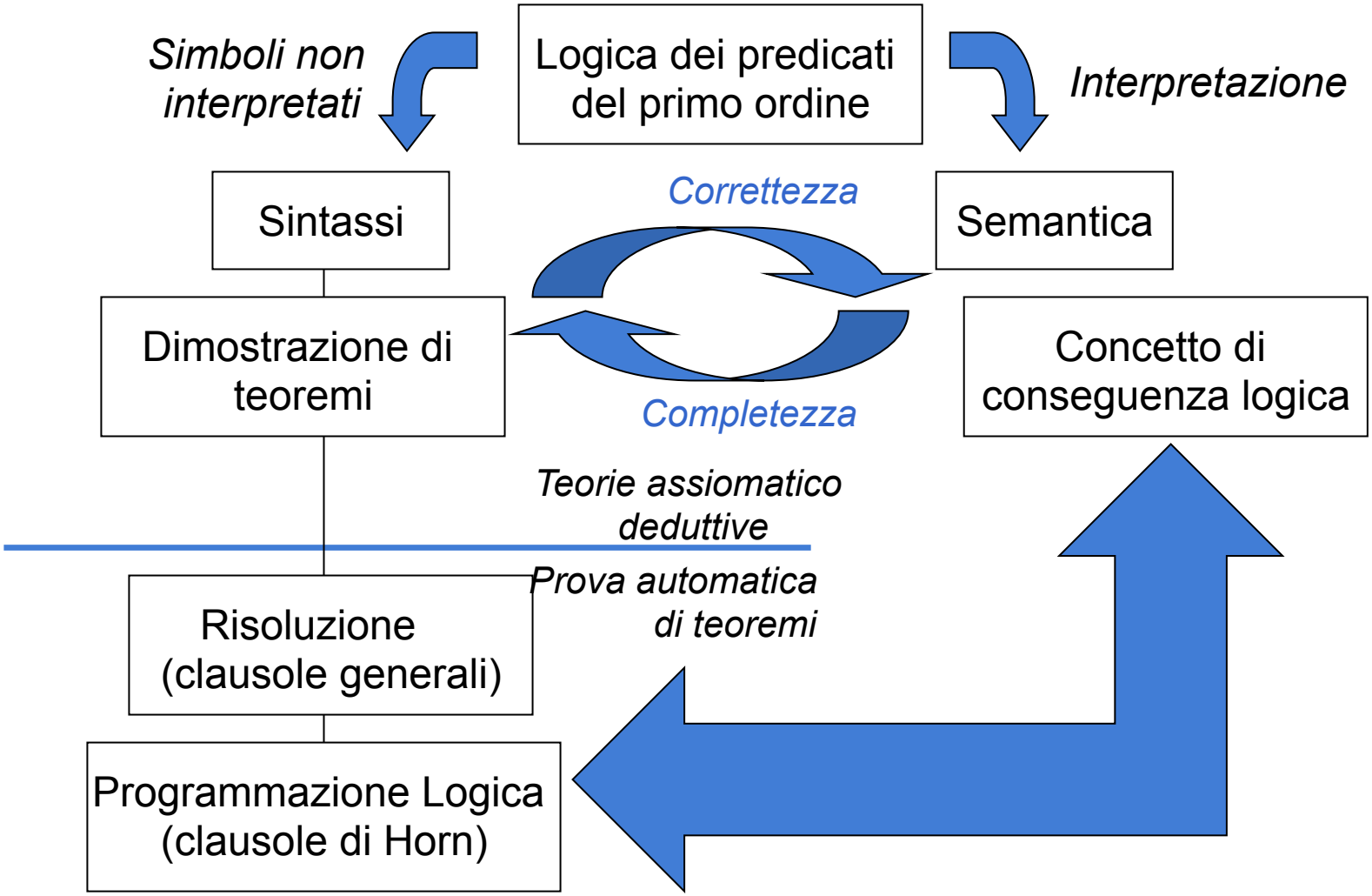
Dimostrazione: esempio 2 (*recap*)



Risposte calcolate

Non determinismo (punti di scelta)

SCHEMA RIASSUNTIVO



SLD resolution (Linear resolution with Selection function for Definite clauses

Prolog – correttezza e completezza

- Le proprietà di correttezza e completezza della risoluzione SLD per le clausole Horn ***non valgono per Prolog***
- Prolog ***non utilizza l'occur check*** nell'algoritmo di unificazione (si ***perde la correttezza***, deriva formule che non seguono logicamente)
- Usa la ***strategia di ricerca depth-first***, non completa, per esplorare lo spazio delle soluzioni rappresentato dall'albero SLD (***perde la completezza***)

Unificazione in Prolog

T1 \ T2	costante c2	variabile X2	termine composto S2
costante c1	unificano se c1==c2	unificano X2/c1	non unificano
variabile X1	unificano X1/c2	unificano X1/X2	unificano X1/S2
termine composto S1	non unificano	unificano X2/S1	Unificano se uguale funtore e parametri unificabili

ALGORITMO DI UNIFICAZIONE

- Funzione UNIFy che ha come parametri di ingresso due atomi (o termini) da unificare A e B e la sostituzione eventualmente già applicata.
- La funzione termina sempre ed è in grado di fornire o la sostituzione più generale per unificare A e B o un fallimento (FALSE).
- Un termine composto (cioè diverso da costante o variabile) è rappresentato da un operatore OP (il simbolo funzionale del termine) e come altri elementi gli argomenti del termine ARGS (lista).
- Es: f(a,g(b,c),X) rappresentato come: [f,a,[g,b,c],X].
- La funzione FIRST, applicata a una lista L, restituisce il primo elemento di L, mentre la funzione REST il resto della lista L.

L'algoritmo di unificazione

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
    y, a variable, constant, list, or compound
    θ, the substitution built up so far

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure

L'algoritmo di unificazione

function UNIFY-VAR(var, x, θ) **returns** a substitution

inputs: var, a variable
x, any expression
θ, the substitution built up so far

if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
else if OCCUR-CHECK?(var, x) **then return** failure
else return add $\{var/x\}$ to θ

OCCUR CHECK

- È il controllo che un termine variabile da unificare con un secondo termine non compaia in quest'ultimo. Necessario per assicurare la terminazione dell'algoritmo e la correttezza del procedimento di unificazione.
- I due termini "X" e "f(X)" non sono unificabili: non esiste una sostituzione per X che renda uguali i due termini.
- Se un termine x ha una struttura complessa, la verifica Occur-Check?(*var,x*) se *var* compare in *x* può essere anche molto costosa.
- Per guadagnare in efficienza, vedremo che **Prolog NON utilizza l'occur-check: per questo è non corretto !**

OCCUR CHECK: ESEMPIO

- **Esempio:** Si considerino il programma e il goal seguenti:

```
maggiore(s(X),X).

?-maggiore(Y,Y).
```

- Per Prolog (senza occur check) unificano e viene prodotta la sostituzione (contenente un termine infinito):

```
Y=s(s(s(.....)))
```

- Quindi se intendiamo rappresentare la relazione maggiore e con s la funzione successore, Prolog deriverebbe che esiste un numero Y maggiore di se stesso!

- **Dimostrazione non corretta**!!

RISOLUZIONE SLD (e PROLOG)

- Risoluzione Lineare per Clausole Definite con funzione di Selezione (backward chaining)
  - Corretta e **completa** per le clausole di Horn

- Dato un programma logico **P** e una clausola goal G_0, ad ogni passo di risoluzione si ricava , se esiste, un nuovo risolvente G_{i+1} ottenuto selezionando un goal atomico da G_i e unificandolo con la testa di (una variante di) una clausola appartenente al programma **P**

- Una variante per una clausola C e' la clausola C' ottenuta da C rinominando le sue variabili (*renaming*)

- **In Prolog, si usa la regola che seleziona l'atomo più a sinistra nel goal corrente, si applicano le clausole secondo il loro ordine testuale e nell'algoritmo di unificazione non si esegue il test di occur-check (ad esempio, X=f(X) unifica in Prolog)**

RISOLUZIONE SLD: ESEMPIO

```
sum(0,X,X).                          (C1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (C2)
```

- Goal `:-sum(s(0),0,W).`
- Al primo passo genero una variante della clausola (C2)

  `sum(s(X1),Y1,s(Z1)):- sum(X1,Y1,Z1).`

  Unificando la testa di questa variante con il goal ottengo la sostituzione MGU

  θ_1 = [`X1/0,Y1/0,W/s(Z1)`]

  Ottengo il nuovo risolvente G1: `:- [sum(X1,Y1,Z1)]` θ_1

  ossia

  `:-sum(0,0,Z1).`

RISOLUZIONE SLD: ESEMPIO

```
sum(0,X,X).                          (C1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (C2)
```

- Graficamente il singolo passo di derivazione è indicato:

- Top goal:

```
:-sum(s(0),0,W)
```

(C2) θ_1 = `[X1/0,Y1/0,W/s(Z1)]`

```
:-sum(0,0,Z1)
```

DERIVAZIONE SLD (backward chaining)

- Una derivazione SLD per un goal G_0 dall'insieme di clausole definite P è una sequenza di clausole goal G_0,… G_n, una sequenza di varianti di clausole del programma C_1, …C_n, e una sequenza di sostituzioni MGU θ_1 ,…, θ_n tali che G_{i+1} è derivato da G_i e da C_{i+1} attraverso la sostituzione θ_{i+1} . La sequenza può essere anche infinita.

- Esistono tre tipi di derivazioni;
  - *successo*, (detta anche ***refutazione***) se per n finito Gn è uguale alla clausola vuota Gn = :-
  - *fallimento finito*: se per n finito non è più possibile derivare un nuovo risolvente da Gn e Gn non è uguale a :-
  - *fallimento infinito*: se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.

DERIVAZIONE DI SUCCESSO

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

- Goal G_0 `:-sum(s(0),0,W)` ha una derivazione di successo

  C1: variante di CL2 `sum(s(X1),Y1,s(Z1)):- sum(X1,Y1,Z1).`

  θ_1 = `[X1/0,Y1/0,W/s(Z1)]`

  G_1: `:-sum(0,0,Z1).`

  C2: variante di CL1 `sum(0,X2,X2).`

  θ_2 = `[Z1/0,X2/0]`

  G_2: `:-`

DERIVAZIONE DI SUCCESSO

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

- Graficamente:

Goal G_0 `:-sum(s(0),0,W)`

(CL2) | θ_1 = `[X1/0,Y1/0,W/s(Z1)]`

`:-sum(0,0,Z1)`

(CL1) | θ_2 = `[Z1/0,X2/0]`

`:-`

LEGAMI PER LE VARIABILI IN USCITA

- Risultato della computazione:
  - successo
  - *legami* per le variabili del goal G0, ottenuti componendo le sostituzioni MGU applicate

Se il goal G_0 è del tipo:

- $\neg\ (\ A_1(t_1,...,t_k),\ A_2(t_{k+1},...,t_h),\ ...,\ A_n(t_{j+1},...,t_m)\ )$
- i termini t_i "ground" rappresentano i *valori di ingresso* al programma, mentre i termini variabili sono i destinatari dei *valori di uscita* del programma.

- Dato un programma logico P e un goal G_0, una *risposta* per $P \cup \{G_0\}$ è una sostituzione per le variabili di G_0.

Legami per le variabili: esempio

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

Top goal G_0 `:-sum(s(0),0,W)` `/* query esiste W ... */`

(CL2) θ_1 = `[X1/0,Y1/0,W/s(Z1)]`

`:-sum(0,0,Z1)`

(CL1) θ_2 = `[Z1/0,X2/0]`

`:-`

(successo)

Risposta calcolata, composizione $\theta_1\ \theta_2$ ristretta alle variabili del top goal: [W/s(0)]

`Yes W=s(0)`

LEGAMI PER LE VARIABILI IN USCITA

- Si consideri una refutazione SLD per P ∪ {G_0}. Una *risposta calcolata* q per P ∪ {G_0} è la sostituzione ottenuta restringendo la composizione delle sostituzioni mgu $q_1,...,q_n$ utilizzate nella refutazione SLD di P ∪ {G_0} alle variabili di G_0.

- La risposta calcolata o *sostituzione di risposta calcolata* è il "testimone" del fatto che esiste una dimostrazione costruttiva di una formula quantificata esistenzialmente (la formula goal iniziale).

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

G= `:-sum(s(0),0,W)` la sostituzione θ `={W/s(0)}` è la risposta calcolata, ottenuta componendo θ_1 con θ_2 e considerando solo la sostituzione per la variabile `W` di G.

DERIVAZIONE SLD (backward chaining)

- Una derivazione SLD per un goal G_0 dall'insieme di clausole definite P è una sequenza di clausole goal G_0,… G_n, una sequenza di varianti di clausole del programma C_1, …C_n, e una sequenza di sostituzioni MGU θ_1 ,…, θ_n tali che G_{i+1} è derivato da G_i e da C_{i+1} attraverso la sostituzione θ_{i+1} . La sequenza può essere anche infinita.

- Esistono tre tipi di derivazioni;
  - *successo*, (detta anche ***refutazione***) se per n finito Gn è uguale alla clausola vuota Gn = :-
  - *fallimento finito*: se per n finito non è più possibile derivare un nuovo risolvente da Gn e Gn non è uguale a :-
  - *fallimento infinito*: se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.

DERIVAZIONE DI FALLIMENTO FINITA

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

- Goal G_0 `:-sum(s(0),0,0)` ha una derivazione di fallimento finito perché l'unico atomo del goal non è unificabile con alcuna clausola del programma
- Graficamente:

Goal G_0 `:-sum(s(0),0,0)`

fail

DERIVAZIONE DI FALLIMENTO INFINITA

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

- Goal G_0 `:-sum(A,B,C)` ha una derivazione SLD infinita, ottenuta applicando ripetutamente varianti della seconda clausola di P

  C1: variante di CL2 `sum(s(X1),Y1,s(Z1)):- sum(X1,Y1,Z1).`

  θ_1 = [`A/s(X1),B/Y1,C/s(Z1)`]

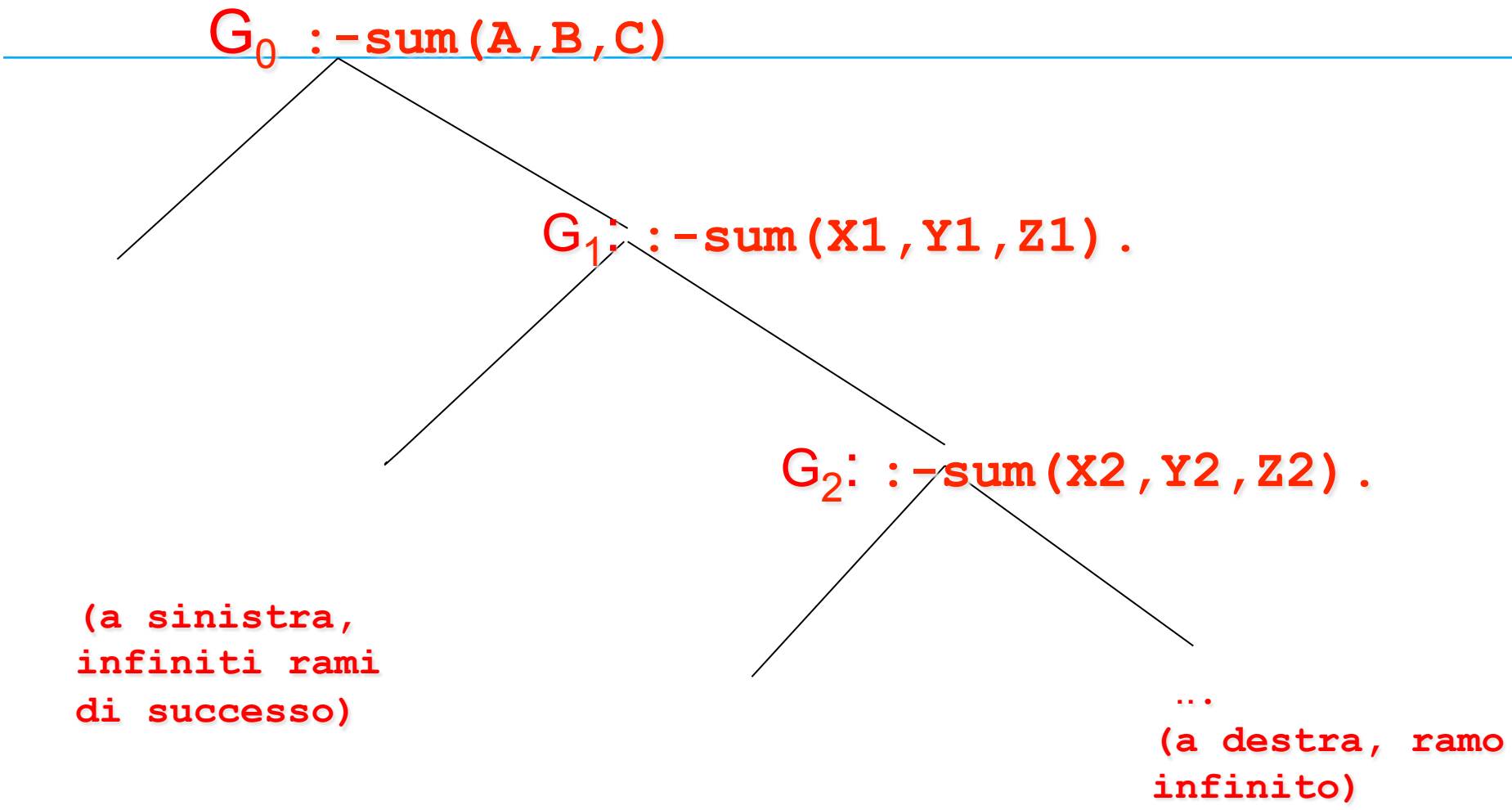
  G_1: `:-sum(X1,Y1,Z1).`

  C2: variante di CL2 `sum(s(X2),Y2,s(Z2)):- sum(X2,Y2,Z2).`

  θ_2 = [`X1/s(X2),Y1/Y2,Z1/s(Z2)`]

  G_2: `:-sum(X2,Y2,Z2).`

  ...


G0 :-sum(A,B,C)
G1: :-sum(X1,Y1,Z1).
G2: :-sum(X2,Y2,Z2).
(a sinistra, infiniti rami di successo)
...
(a destra, ramo infinito)

Albero SLD

- E' lo spazio delle derivazioni SLD per un goal G_0 dall'insieme di clausole definite P
- Albero OR
- Raccoglie i tre tipi di derivazioni;
  - *successo*, (detta anche ***refutazione***) se per n finito Gn è uguale alla clausola vuota Gn = :-
  - *fallimento finito*: se per n finito non è più possibile derivare un nuovo risolvente da Gn e Gn non è uguale a :-
  - *fallimento infinito*: se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.
- Ci interessano, se esistono, i rami di successo (le derivazioni di successo sono le soluzioni)

NON DETERMINISMO

- Nella risoluzione SLD così come è stata enunciata si hanno *due forme di non determinismo*
- La prima forma di non determinismo è legata alla selezione di un atomo A_m del goal da unificare con la testa di una clausola, e viene risolta definendo una particolare ***regola di calcolo (o funzione di selezione)***.
  - **Prolog usa la regola left-most**
  - **La regola di calcolo non ha impatto sulla completezza**
- La seconda forma di non determinismo è legata alla scelta di quale clausola del programma P utilizzare in un passo di risoluzione, e viene risolta adottando una ***strategia di ricerca***, nello spazio delle soluzioni che è l'albero SLD
  - **Prolog usa la strategia depth-first**
  - **ha impatto sulla completezza**

REGOLA DI CALCOLO

- Una *regola di calcolo* è una funzione che ha come dominio l'insieme dei goal e che seleziona un suo atomo A_m dal goal

  $:-A_1,...,A_{m-1},A_m,A_{m+1},...,A_k$, ($A_m$:*atomo selezionato*).

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

G0 = `:-sum(0,s(0),s(0)),sum(s(0),0,s(0)).`

– Se si seleziona l'atomo più a sinistra al primo passo, unificando l'atomo `sum(0,s(0),s(0))` con la testa di CL1, si otterrà:

G1 = `:- sum(s(0),0,s(0)).`

– Se si seleziona l'atomo più a destra al primo passo, unificando l'atomo `sum(s(0),0,s(0))` con la testa di CL2, si avrà:

G1 = `:- sum(0,s(0),s(0)),sum(0,0,0).`

INDIPENDENZA DALLA REGOLA DI CALCOLO

- La regola di calcolo influenza solo l'efficienza
- Non influenza né la correttezza né la completezza del dimostratore.

- **Proprietà** *(Indipendenza dalla regola di calcolo)*
  - Dato un programma logico P, l'insieme di successo di P non dipende dalla regola di calcolo utilizzata dalla risoluzione SLD.

Insieme di successo, formule atomiche ground per le quali si ha una refutazione SLD (formule ground che sono teoremi derivabili dal programma)

Prolog usa la regola di calcolo left-most (riscrive sempre il primo atomo del goal, unificandolo con la testa di una clausola del programma)

STRATEGIA DI RICERCA

- Definita una regola di calcolo, nella risoluzione SLD resta un ulteriore grado di non determinismo poiché **possono esistere più teste di clausole unificabili con l'atomo selezionato**.

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```

G0 = `:-sum(W,0,K).`

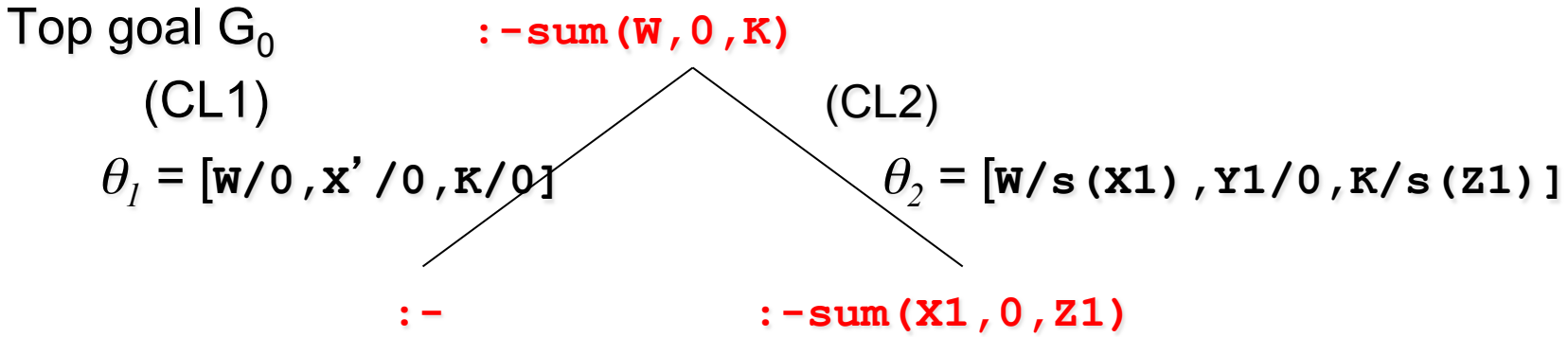
- Se si sceglie la clausola CL1 si ottiene il risolvente
  G1 = `:-`
- Se si sceglie la clausola CL2 si ottiene il risolvente
  G1 = `:- sum(X1,0,Z1)`

STRATEGIA DI RICERCA

- Questa forma di non determinismo implica che possano esistere più soluzioni alternative per uno stesso goal.
- La risoluzione SLD (completezza), deve essere in grado di generare tutte le possibili soluzioni e quindi deve considerare ad ogni passo di risoluzione tutte le possibili alternative.
- La strategia di ricerca deve garantire questa completezza
- Una forma grafica utile per rappresentare la risoluzione SLD e questa forma di non determinismo sono gli alberi SLD.

Albero SLD: esempio

```
sum(0,X,X).                          (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).       (CL2)
```



ALBERI SLD

- Dato un programma logico P, un goal G_0 e una regola di calcolo R, un albero SLD per $P \cup \{G_0\}$ via R è definito come segue:
  - ciascun nodo dell'albero è un goal (eventualmente vuoto);
  - la radice dell'albero è il goal G_0;
  - dato il nodo $:-A_1,...,A_{m-1},A_m,A_{m+1},...,A_k$ se A_m è l'atomo selezionato dalla regola di calcolo R, allora questo nodo (*genitore*) ha un nodo *figlio* **per ciascuna clausola $C_i = A:-B_1,...,B_q$ di P tale che A e A_m sono unificabili** attraverso una sostituzione unificatrice più generale θ. Il nodo figlio è etichettato con la clausola goal:

    $:-[A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k]\theta$ e il ramo dal nodo padre al figlio è etichettato dalla sostituzione θ e dalla clausola selezionata C_i;
  - il nodo vuoto (indicato con "$:-$") non ha figli.

ALBERI SLD

- A ciascun nodo dell'albero può essere associata una *profondità*.
  - La radice dell'albero ha profondità 0, mentre la profondità di ogni altro nodo è quella del suo genitore più 1.

- **Ad ogni cammino di un albero SLD corrisponde una derivazione SLD.**
  - Ogni ramo che termina con il nodo vuoto (“:-”) rappresenta una derivazione SLD di successo (refutazione SLD).

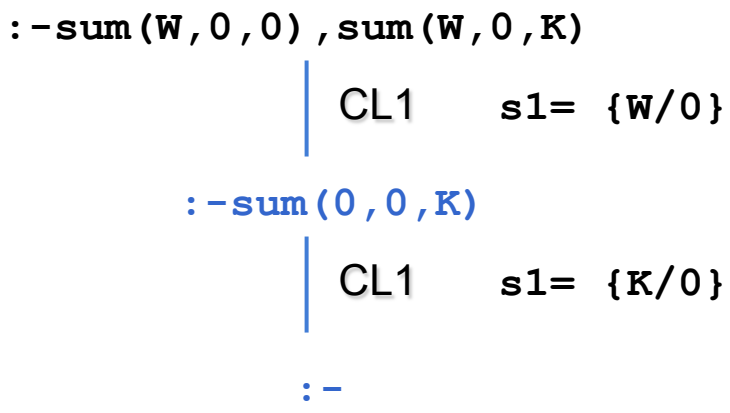
- La regola di calcolo influisce sulla struttura dell’albero per quanto riguarda sia l’ampiezza sia la profondità. Tuttavia non influisce su correttezza e completezza. Quindi, qualunque sia R, **il numero di cammini di successo (se in numero finito) è lo stesso in tutti gli alberi SLD costruibili per P ∪ {G0}** .
- R influenza solo il numero di cammini di fallimento (finiti ed infiniti).

ALBERI SLD: ESEMPIO

```
sum(0,X,X).                                  (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).               (CL2)
```

G0= `:- sum(W,0,0),sum(W,0,K).`

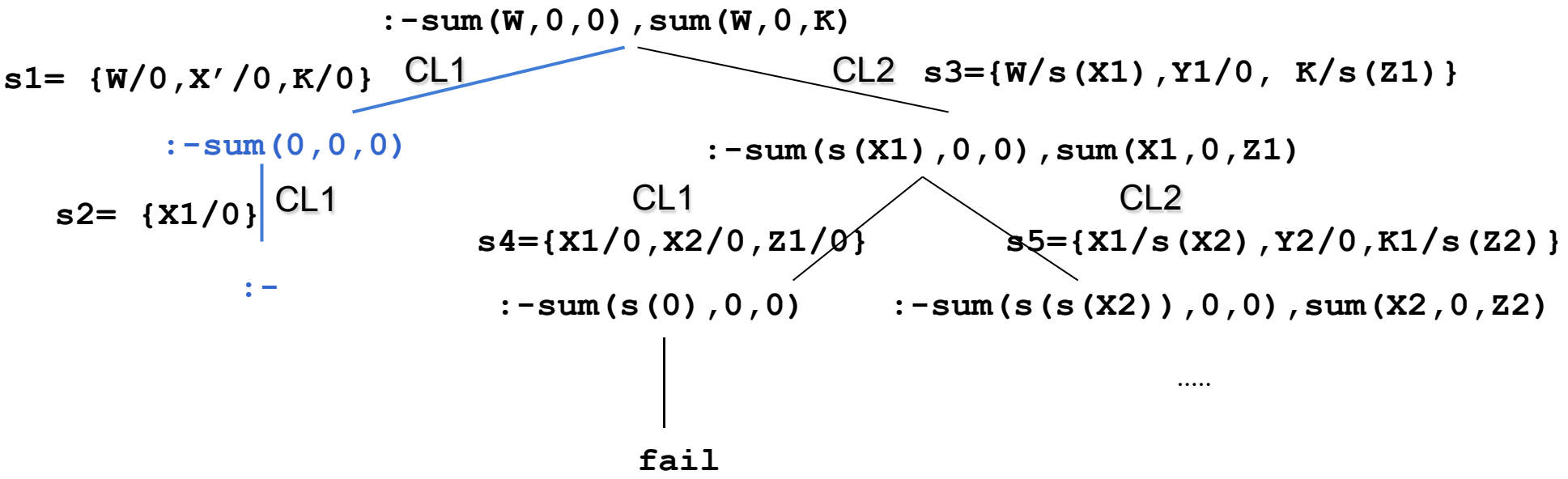
- Albero SLD con regola di calcolo "left-most" (come fa Prolog)



ALBERI SLD: ESEMPIO

```
sum(0,X,X).                                  (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).               (CL2)
G0= :- sum(W,0,0),sum(W,0,K).
```

- Albero SLD con regola di calcolo "right- most"



ALBERI SLD: ESEMPIO

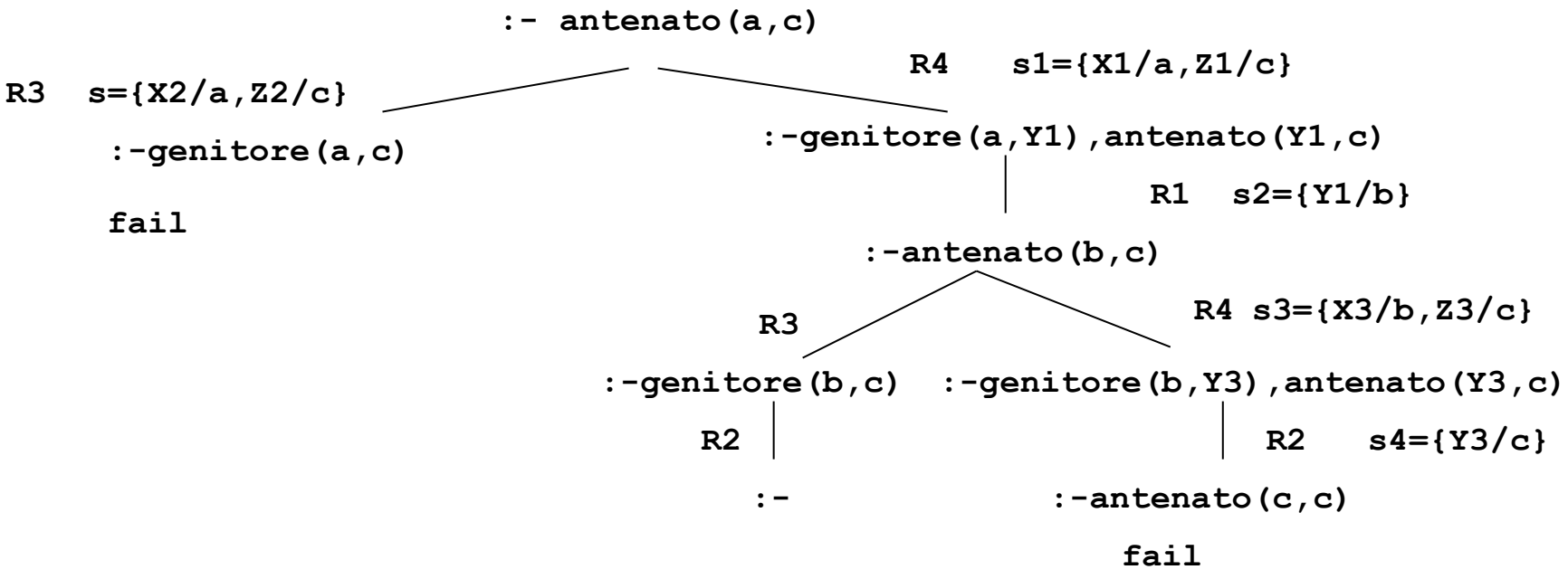
- Confronto albero SLD con regola di calcolo left most e right most:
  - In entrambi gli alberi esiste una refutazione SLD, cioè un cammino (ramo) di successo il cui nodo finale è etichettato con ”`:-`".

- La composizione delle sostituzioni applicate lungo tale cammino genera la sostituzione di risposta calcolata `{W/0,K/0}`.

- Si noti la differenza di struttura dei due alberi. In particolare cambiano i rami di fallimento (finito e infinito).

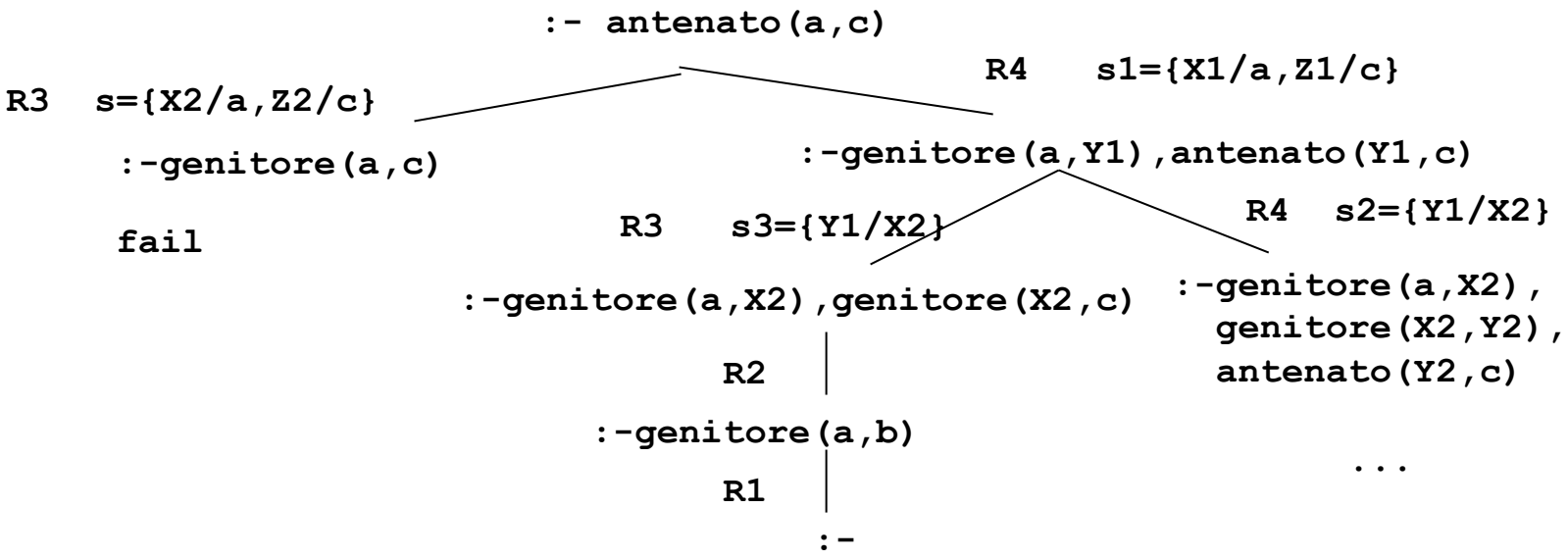
ALBERI SLD LEFT MOST: ESEMPIO (1)

```
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)
```



ALBERI SLD RIGHT MOST: ESEMPIO (1)

```
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)
```



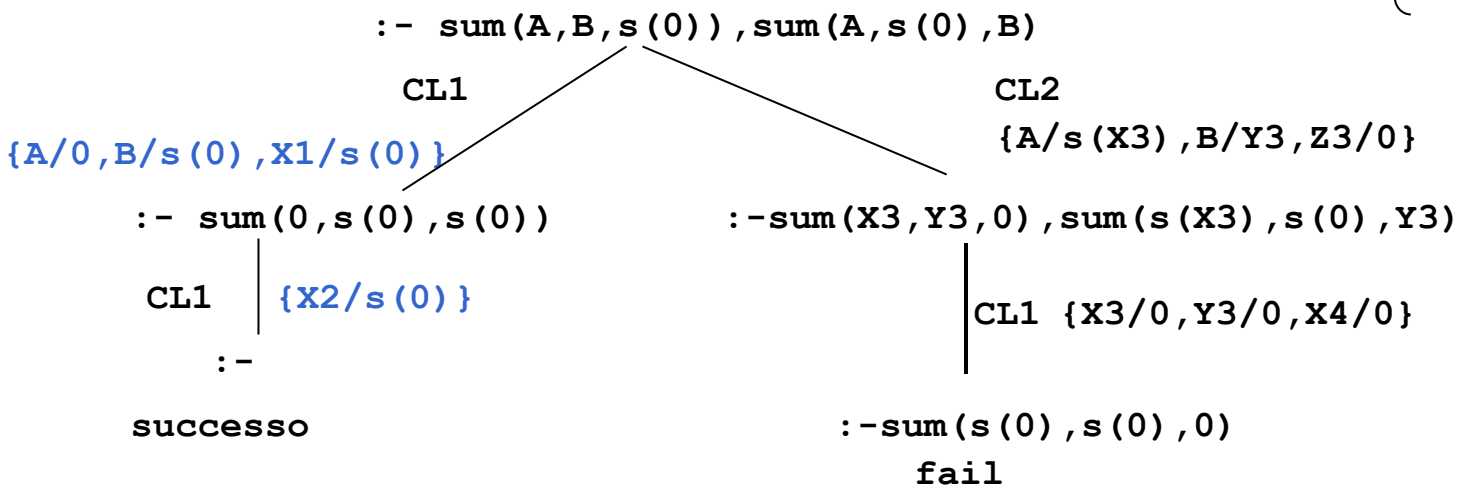
ALBERI SLD LEFT MOST: ESEMPIO (2)

```
sum(0,X,X).                                  (CL1)
sum(s(X),Y,s(Z)):- sum(X,Y,Z).               (CL2)
```

G0= `:- sum(A,B,s(0)),sum(A,s(0),B).`

– La query rappresenta il sistema di equazioni

$$\begin{cases} A+B=1 \\ B-A=1 \end{cases}$$



– Per l'unica derivazione di successo, la composizione delle sostituzioni applicate (cioè `{A/0,B/s(0),X1/s(0)}{X2/s(0)}`), ristretta alle variabili del goal G0, produce la risposta calcolata: `{A/0,B/s(0)}`

STRATEGIA DI RICERCA

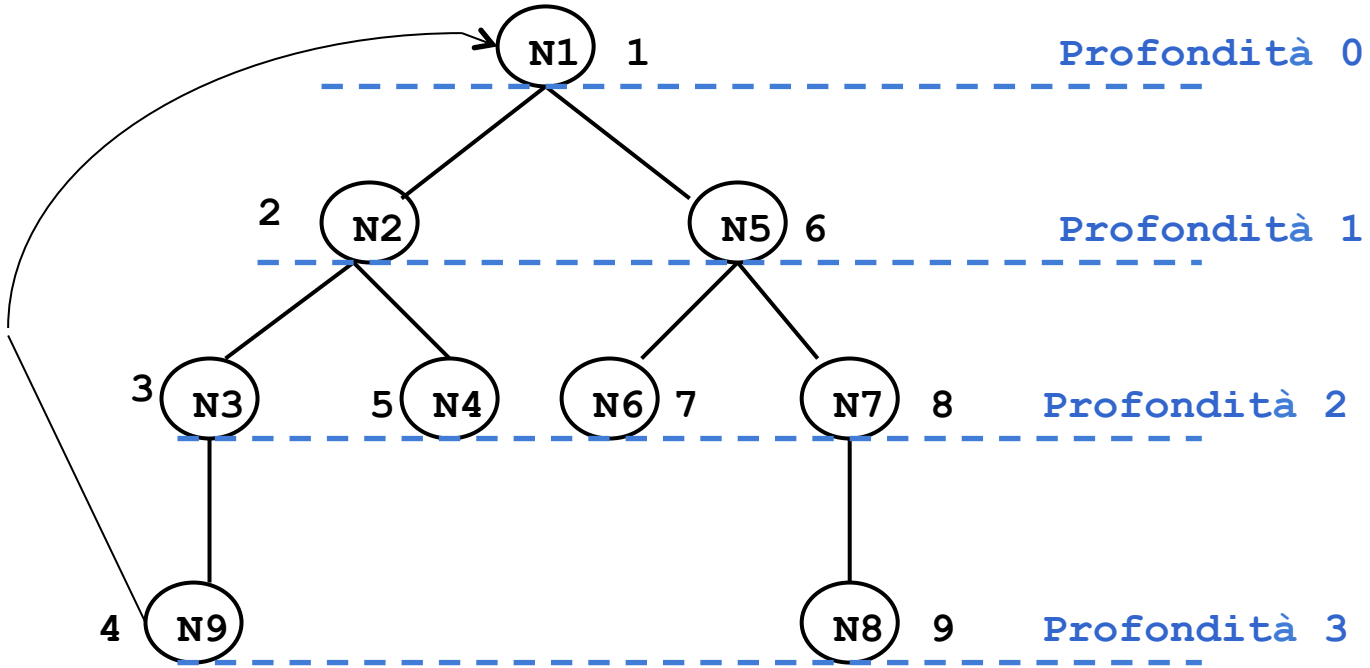
- La realizzazione effettiva di un dimostratore basato sulla risoluzione SLD richiede la definizione non solo di una regola di calcolo, ma anche di una strategia di ricerca che stabilisce una particolare *modalità di esplorazione* dell'albero SLD alla ricerca dei rami di successo.

- Le modalità di esplorazione dell'albero più comuni (non informate) sono:
  - depth first
  - breadth first

- Entrambe le modalità implicano l'esistenza di una opportuna struttura dati (frontiera, pila/stack o coda come già visto)
- Entrambe richiedono un meccanismo a supporto del backtracking, per esplorare tutte le strade alternative che corrispondono ai diversi nodi dell'albero.

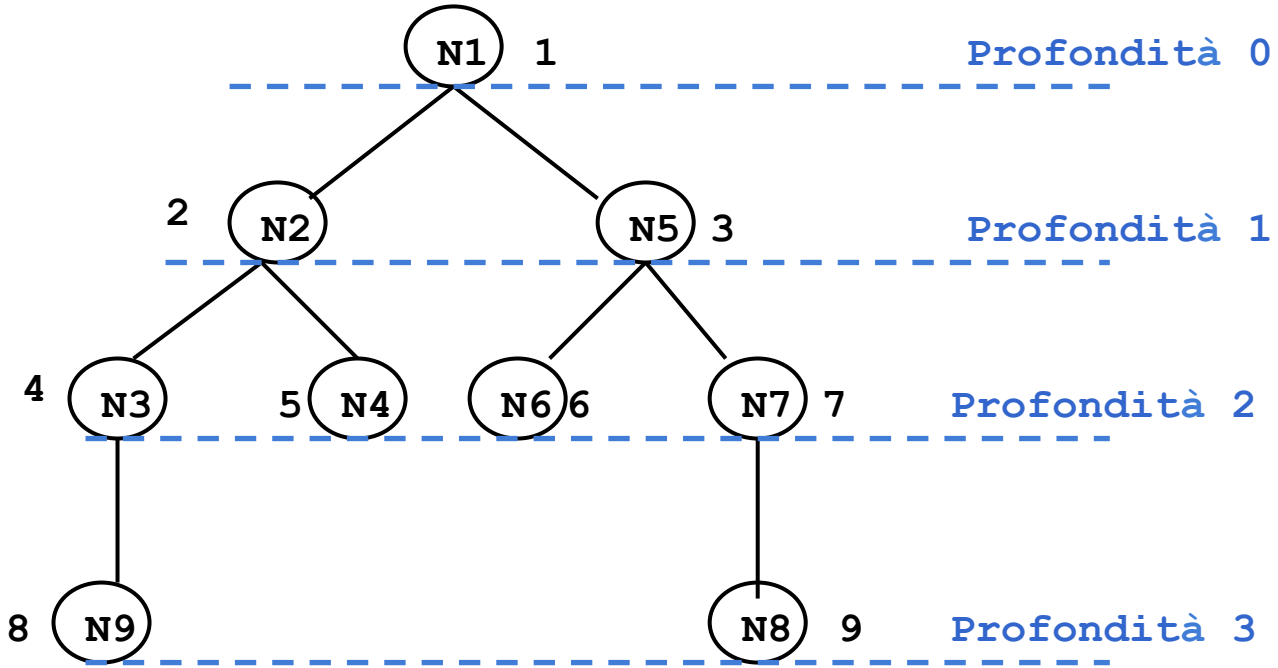
STRATEGIA DEPTH-FIRST

– Ricerca in profondità: vengono prima esplorati i nodi a profondità maggiore. NON COMPLETA, in caso di cicli



STRATEGIA BREADTH-FIRST

– Ricerca in ampiezza: vengono prima esplorati i nodi a profondità minore. COMPLETA



STRATEGIE DI RICERCA E ALBERI SLD

- Nel caso degli alberi SLD, lo spazio di ricerca non è esplicito, ma resta definito implicitamente dal programma P e dal goal G_0.
  - I nodi corrispondono ai risolventi generati durante i passi di risoluzione.
  - I figli di un risolvente G_i sono tutti i possibili risolventi ottenuti unificando un atomo A di G_i, selezionato secondo una opportuna regola di calcolo, con le clausole del programma P.
  - Il numero di figli generati corrisponde al numero di clausole alternative del programma P che possono unificare con A.

- Agli alberi SLD possono essere applicate entrambe le strategie discusse in precedenza.
  - Nel caso di alberi SLD**, attivare il "backtracking" implica che tutti i legami per le variabili determinati dal punto di "backtracking" in poi non devono essere più considerati**.

PROLOG E STRATEGIE DI RICERCA

- Il linguaggio Prolog, adotta la *strategia in profondità con "backtracking"* perché può essere realizzata in modo efficiente attraverso un unico stack di goal.
  - tale stack rappresenta il ramo che si sta esplorando e contiene opportuni riferimenti a rami alternativi da esplorare in caso di fallimento (punti di scelta, o choice point).

- Per quello che riguarda la scelta fra nodi fratelli, la strategia Prolog li ordina seguendo l'ordine testuale delle clausole che li hanno generati.

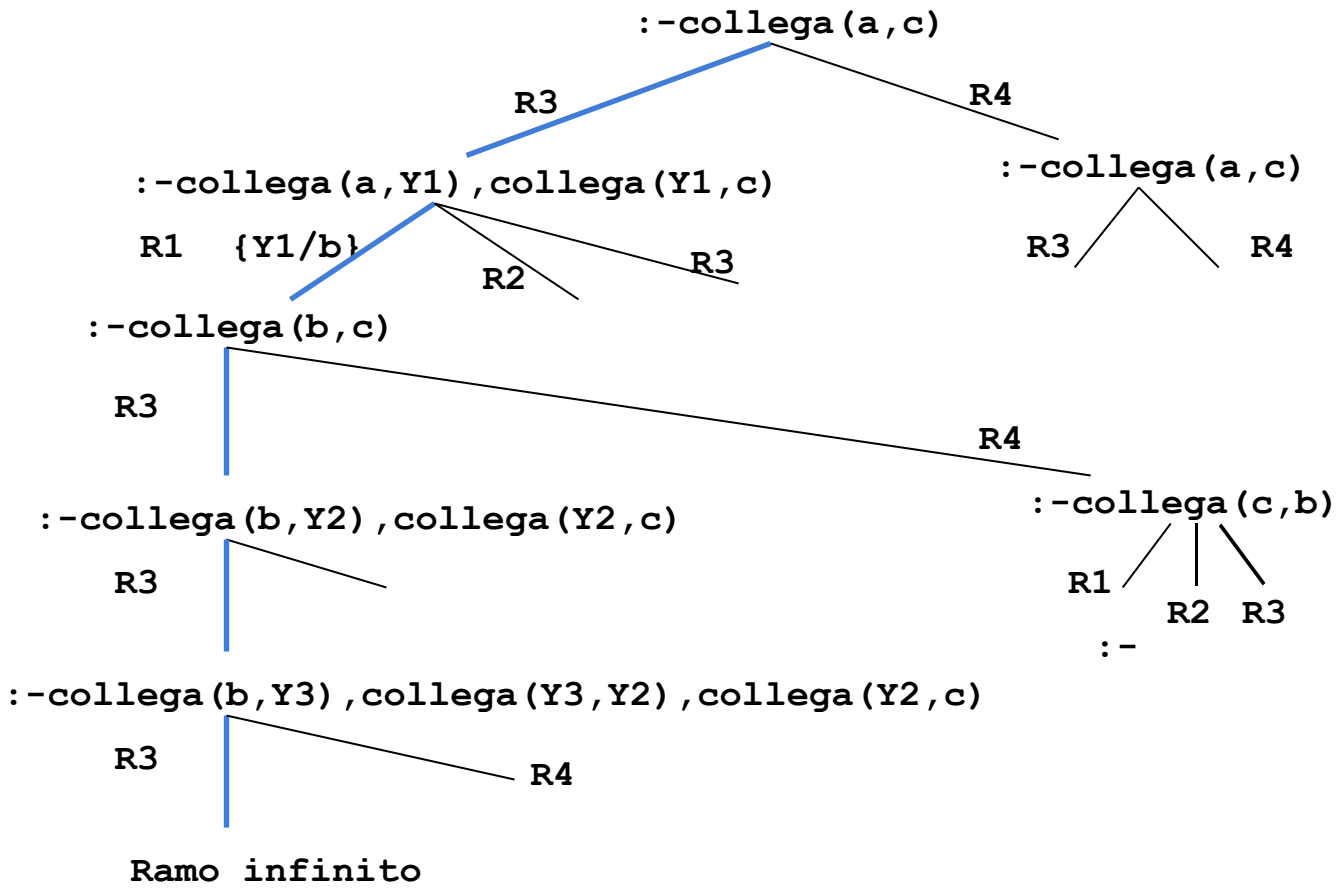
- La strategia di ricerca adottata in Prolog è dunque depth-first, e quindi non completa.

PROLOG E STRATEGIE DI RICERCA

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

- La formula **`collega(a,c)`** segue logicamente dagli assiomi, ma la procedura di dimostrazione di Prolog non è in grado di dimostrarlo.

ALBERO SLD CON RAMO INFINITO



☞RIASSUMENDO…

- La forma di risoluzione utilizzata dai linguaggi di programmazione logica è la **risoluzione SLD**, che in generale, presenta due forme di non determinismo:
  - la regola di calcolo (***selection function***)
  - la strategia di ricerca

- Il linguaggio Prolog utilizza la risoluzione SLD con le seguenti scelte

  - *Regola di calcolo*
    - Regola "**left-most**"; data una "query":

      ?- $G_1, G_2, ..., G_n$.

      viene sempre selezionato il letterale più a sinistra G_1.
  - *Strategia di ricerca (non completa, ma efficiente)*
    - **In *profondità* (*depth-first*) con *backtracking cronologico*.**

RISOLUZIONE IN PROLOG

- Dato un letterale G_1 da risolvere, viene selezionata la prima clausola (secondo l'ordine delle clausole nel programma P) la cui testa è unificabile con G_1.

- Nel caso vi siano più clausole la cui testa è unificabile con G_1, la risoluzione di G_1 viene considerata come un *punto di scelta* (*choice point*) nella dimostrazione.

- In caso di fallimento in un passo di dimostrazione, l'interprete Prolog ritorna in backtracking all'ultimo punto di scelta in senso cronologico (il più recente), e seleziona la clausola successiva utilizzabile in quel punto per la dimostrazione.

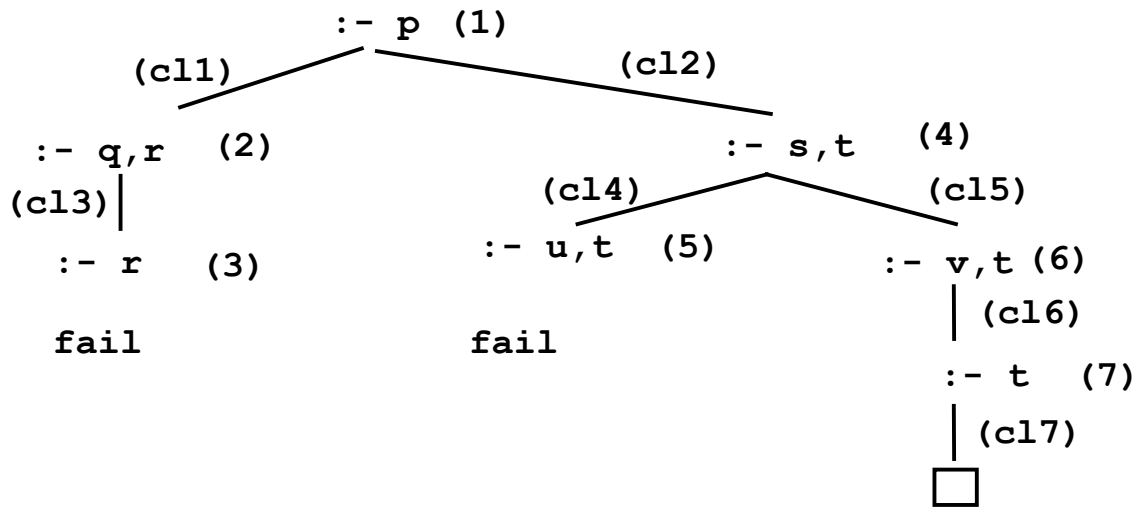
Ricerca in profondità con backtracking cronologico dell'albero di dimostrazione SLD.

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.
```

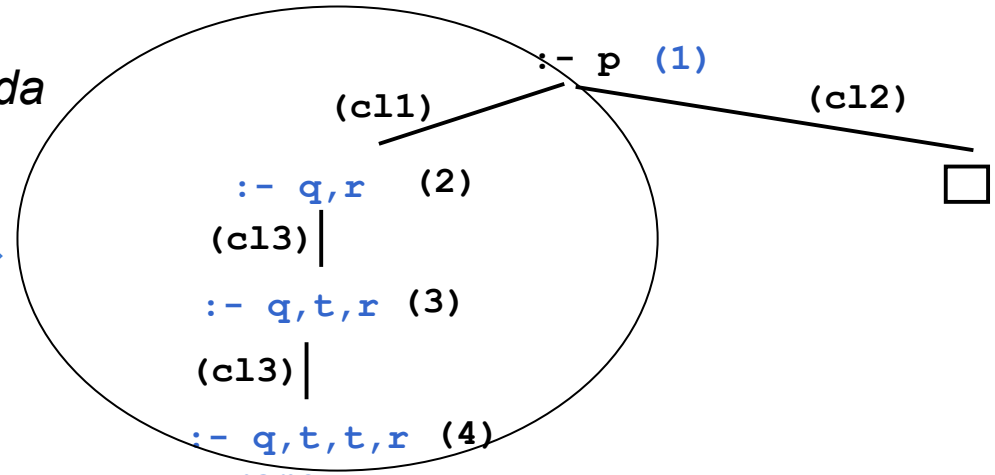


RISOLUZIONE IN PROLOG: INCOMPLETEZZA

- Un problema della strategia in profondità utilizzata da Prolog è la sua incompletezza.

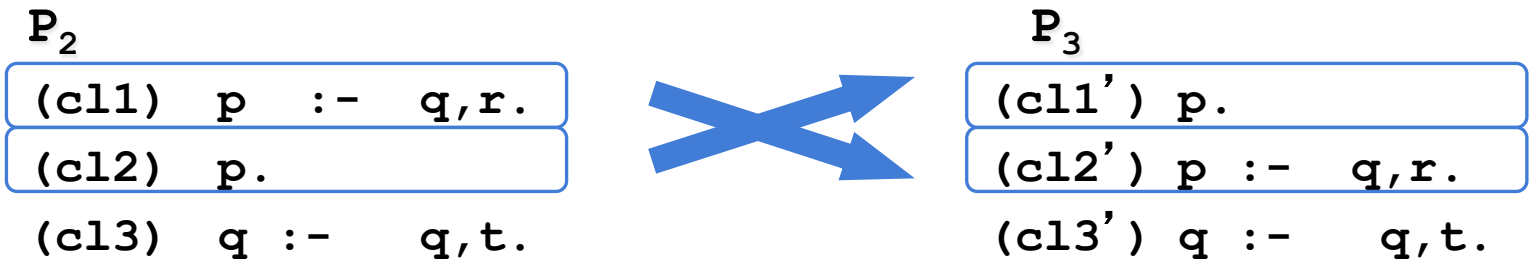
```
P2          (cl1)  p  :-  q,r.
            (cl2)  p.
            (cl3)  q :-  q,t.        /*ricorsione a sinistra*/
            :- p.
```

Cammino esplorato da Prolog



ORDINE DELLE CLAUSOLE

- **L'ordine delle clausole in un programma Prolog è rilevante.**



- I due programmi P_2 e P_3 non sono due programmi Prolog equivalenti. Infatti, data la "query": `:-p.` si ha che
  - la dimostrazione con il programma P_2 non termina;
  - la dimostrazione con il programma P_3 ha immediatamente successo.
- ***Perché una strategia in profondità che è incompleta?***
- Una strategia di ricerca in profondità può essere realizzata in modo efficiente utilizzando tecniche non troppo differenti da quelle utilizzate nella realizzazione dei linguaggi imperativi tradizionali (stack di goal, anziché stack di record di attivazione).

ORDINE DELLE CLAUSOLE: ESEMPIO

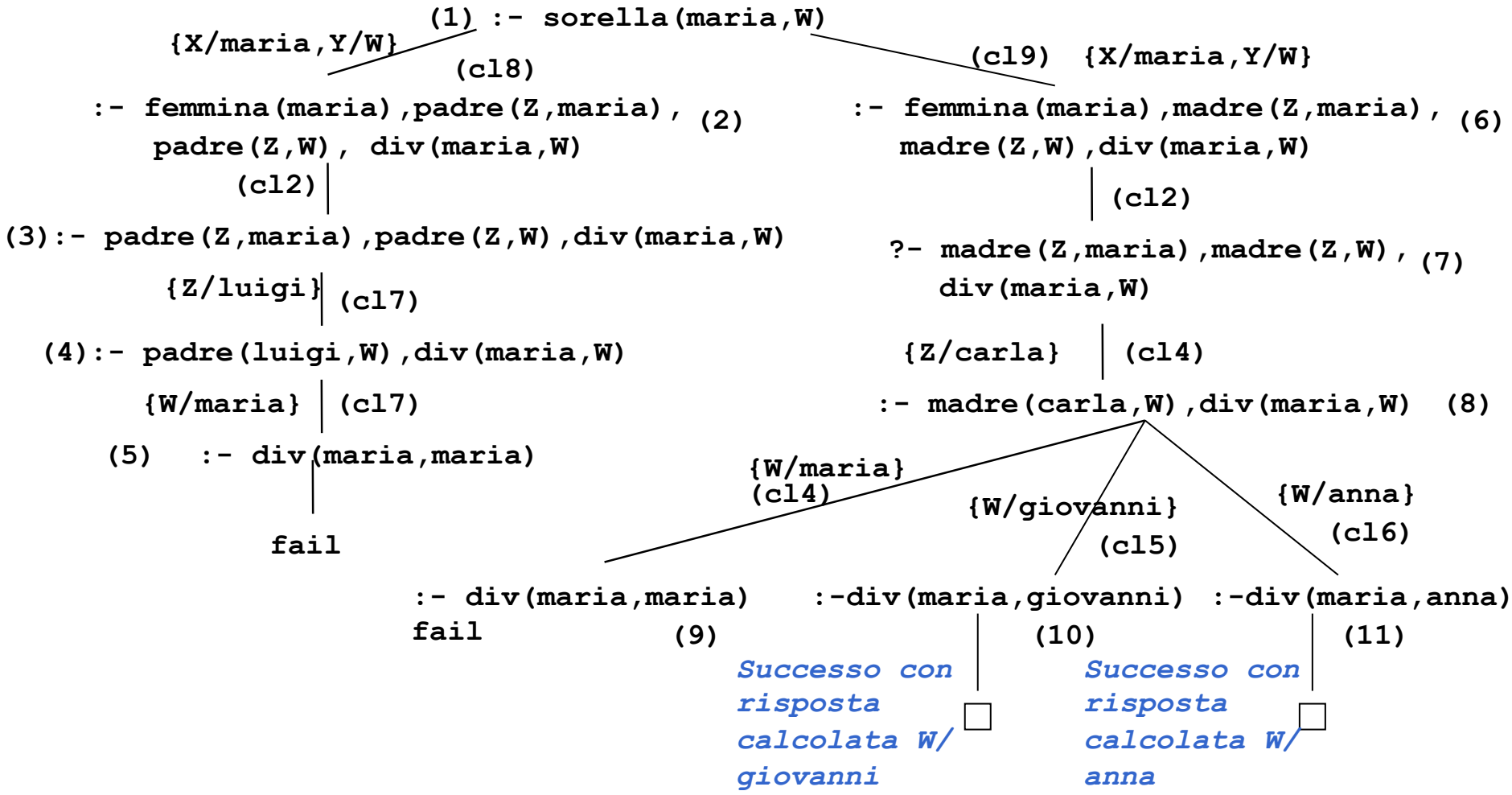
P_4

```
(cl1) femmina(carla).
(cl2) femmina(maria).
(cl3) femmina(anna).
(cl4) madre(carla,maria).
(cl5) madre(carla,giovanni).
(cl6) madre(carla,anna).
(cl7) padre(luigi,maria).
(cl8) sorella(X,Y):- femmina(X),
                     padre(Z,X),
                     padre(Z,Y),
                     div(X,Y).
(cl9) sorella(X,Y):- femmina(X),
                     madre(Z,X),
                     madre(Z,Y),
                     div(X,Y).
(cl10) div(carla,maria).
(cl11) div(maria,carla).
```

..... `div(A,B).` per tutte le coppie `(A,B)` con `A≠B`

E la "query": `:- sorella(maria,W).`

ORDINE DELLE CLAUSOLE: ESEMPIO



SOLUZIONI MULTIPLE E DISGIUNZIONE

- Possono esistere più sostituzioni di risposta per una “query”.
  - Per richiedere ulteriori soluzioni è sufficiente forzare un fallimento nel punto in cui si è determinata la soluzione che innesca il backtracking.
  - Tale meccanismo porta ad espandere ulteriormente l’albero di dimostrazione SLD alla ricerca del prossimo cammino di successo.

- In Prolog standard tali soluzioni possono essere richieste mediante l'operatore “ ; “.

```
?-  sorella(maria,W).
W=giovanni;
W=anna;
no
```

- Il carattere ";" può essere interpretato come
  - un operatore di disgiunzione che separa soluzioni alternative.
  - all'interno di un programma Prolog per esprimere la disgiunzione.

INTERPRETAZIONE DICHIARATIVA

- ESEMPI

  `?-padre(X,Y)` "esiste **X,Y tale che X** e' il padre di **Y**"

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

  "per ogni **X** e **Y**, **X** e' il nonno di **Y** se esiste **Z** tale che **X** e' padre di **Z** e **Z** e' il padre di **Y**"

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

  "per ogni **X** e **Y**, **X** e' il nonno di **Y** se esiste **Z** tale che **X** e' padre di **Z** e **Z** e' la madre di **Y**"

- Oppure:

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

"per ogni **X** e **Y**, **X** e' il nonno di **Y** se esiste **Z** tale che **X** e' padre di **Z** e **Z** e' il padre o la madre di **Y**"

INTERPRETAZIONE PROCEDURALE

- Prolog può avere un'interpretazione procedurale. Una ***procedura*** è un insieme di clausole di P le cui teste hanno lo stesso simbolo predicativo e lo stesso numero di argomenti (arità).
  - Gli argomenti che compaiono nella testa della procedura possono essere visti come i *parametri formali*.

    Una "query" del tipo: `:- p(t1,t2,...,tn).`

    è la *chiamata* della procedura `p/n` Gli argomenti di `p/n` (ossia i termini `t1,t2,...,tn`) sono i *parametri attuali*.
  - L'unificazione è il meccanismo di ***passaggio dei parametri***.

- Non vi è alcuna distinzione a priori tra i parametri di ingresso e i parametri di uscita (*reversibilità*).

INTERPRETAZIONE PROCEDURALE (2)

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

- Due clausole con le stesse teste corrispondono a due definizioni alternative del corpo di una procedura.

- Tutte le variabili sono a *singolo assegnamento*. Il loro valore è unico durante tutta la computazione (singola derivazione, cammino nell'albero SLD) e slegato solo quando si cerca una soluzione alternativa ("backtracking").

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).

:-  pratica_sport(X,calcio).
          "esiste X tale per cui X pratica il calcio?"
yes       X=mario;
          X=giovanni;
          X=alberto;
no
:-  pratica_sport(giovanni,Y).
          "esiste uno sport Y praticato da giovanni?"
yes       Y=calcio;
no
```

ESEMPIO (2)

```
:-  pratica_sport(X,Y).
 "esistono X e Y tali per cui X pratica lo sport Y"
yes      X=mario          Y=calcio;
         X=giovanni       Y=calcio;
         X=alberto        Y=calcio;
         X=marco          Y=basket;
no

:- pratica_sport(X,calcio), abita(X,genova).
 "esiste una persona X che pratica il calcio e abita a Genova?”
yes      X=giovanni;
         X=alberto;
no
```

ESEMPIO (3)

- A partire da tali relazioni, si potrebbe 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,Z)
             abita(Y,Z),
             pratica_sport(X,S),
             pratica_sport(Y,S).
```

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

"esiste Y tale per cui Giovanni e amico di Y?"

```
yes    Y=giovanni;
       Y=alberto;
no
```

- si noti che secondo tale relazione ogni persona è amica di se stessa (potrei aggiungere **X\==Y** nel corpo della clausola per **amico/2**).

ESEMPIO (4)

```
padre(X,Y)                  "X è il padre di Y"
madre(X,Y)                  "X è la madre di Y"
zia(X,Y)            "X è la zia di Y"
zia(X,Y)     :-sorella(X,Z),padre(Z,Y).
zia(X,Y)     :-sorella(X,Z),madre(Z,Y).
```

(la relazione `"sorella"` è stata definita in precedenza).

- Definizione della relazione `"antenato"` in modo ricorsivo:

  `"X è un antenato di Y se X è il padre (madre) di Y"`

  `"X è un antenato di Y se X è un antenato del padre (o della madre) di Y"`

```
antenato(X,Y)           "X è un antenato di Y"
antenato(X,Y)   :-      padre(X,Y).
antenato(X,Y)   :-      madre(X,Y).
antenato(X,Y)   :-      padre(Z,Y),antenato(X,Z).
antenato(X,Y)   :-      madre(Z,Y),antenato(X,Z).
```

VERSO UN VERO LINGUAGGIO DI PROGRAMMAZIONE

- Al Prolog puro devono, tuttavia, 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.