LINGUAGGIO PROLOG

- PROLOG: PROgramming in LOGic, 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 (prova automatica di teoremi - risoluzione)
- Manipolatore di SIMBOLI e non di NUMERI
- Linguaggio ad ALTISSIMO LIVELLO: utilizzabile anche da non programmatori
- APPLICAZIONI DI AI

LINGUAGGIO PROLOG

- Lavora su strutture ad ALBERO
  - anche i programmi sono strutture dati manipolabili
  - utilizzo della ricorsione e non assegnamento
- Metodologia di programmazione:
  - concentrarsi sulla specifica del problema rispetto alla strategia di soluzione
- Svantaggi:
  - linguaggio relativamente giovane
  - efficienza paragonabile a quella del LISP
  - non adatto ad applicazioni numeriche o in tempo reale
  - mancanza di ambienti di programmazione evoluti

ALGORITMO = LOGICA + CONTROLLO

- Conoscenza sul problema indipendente dal suo utilizzo
  - Esprimo COSA e non COME
  - Alta modularità e flessibilità
  - Schema progettuale alla base di gran parte dei SISTEMI BASATI SULLA CONOSCENZA (Sistemi Esperti)
- LOGICA: conoscenza sul problema
  - correttezza ed efficienza
- CONTROLLO: strategia risolutiva
  - efficienza
- Algoritmi equivalenti:
  - A1 = L + C1
  - A2 = L + C2

PROGRAMMA PROLOG

- Un PROGRAMMA PROLOG e' 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), sulla base della conoscenza definita
- ESEMPIO: due individui sono colleghi se lavorano per la stessa ditta

Testa — Body

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

→ REGOLA

```
lavora(emp1,ibm).
lavora(emp2,ibm).
lavora(emp3,txt).
lavora(emp4,olivetti).
lavora(emp5,txt).
```

→ FATTI

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

→ GOAL

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

```
:- 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).
```


PROVA DI UN GOAL

- Un goal viene provato provando i singoli letterali da sinistra a destra

```
:- collega(X,Y), persona(X), persona(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 e' un fatto, la prova termina;
  - se la clausola con cui unifica e' una regola, ne viene provato il Body

- Se non esiste una sostituzione il goal fallisce

PROVA DI UN GOAL: esempio

```
append([],X,X).
append([X|Z],Y,[X|T]):- append(Z,Y,T).

:- append([a,b],[c,d],[a,b,c,d]).
```

- Questo goal atomico viene provato unificandolo con la testa della seconda regola: intuitivamente `X` unifica con `a`, `Z` con la lista `[b]`, `Y` con la lista `[c,d]` `T` con la lista `[b,c,d]`

- Viene quindi provato il body dopo aver effettuato le sostituzioni

```
:- append([b],[c,d],[b,c,d]).
```

- Questo goal atomico viene provato unificandolo con la testa della seconda regola: `X` unifica con `b`, `Z` con la lista `[]`, `Y` con la lista `[c,d]` `T` con la lista `[c,d]`

PROVA DI UN GOAL: esempio

```
append([],X,X).
append([X|Z],Y,[X|T]):- append(X,Y,T).

:- append([a,b],[c,d],[a,b,c,d]).
```

- Viene quindi provato il body dopo aver effettuato le sostituzioni

```
:- append([],[c,d],[c,d]).
```

- Questo goal atomico viene provato unificandolo con la testa della prima regola che e' un fatto e quindi la prova termina con successo

PROVA DI UN GOAL: esempio

```
append([],X,X).
append([X|Z],Y,[X|T]):- append(X,Y,T).
```

Come vengono dimostrati i successivi goal ?

```
:- append([a,b],Y,[a,b,c,d]).
:- append(X,[c,d],[a,b,c,d]).
:- append(X,Y,[a,b,c,d]).
:- append(X,Y,Z).
```


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 e' una formula del tipo

  `p(t1,t2,…,tn)`

  in cui `p` e' un simbolo predicativo e `t1,t2,…,tn` sono termini

- Un termine e' 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)` "`X` e' il padre di `Y`"

  `madre(X,Y)` "`X` e' la madre 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`"

ESECUZIONE DI UN PROGRAMMA

- Una computazione corrisponde al tentativo di dimostrare, tramite la risoluzione, che una formula segue logicamente da un programma (e' 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:

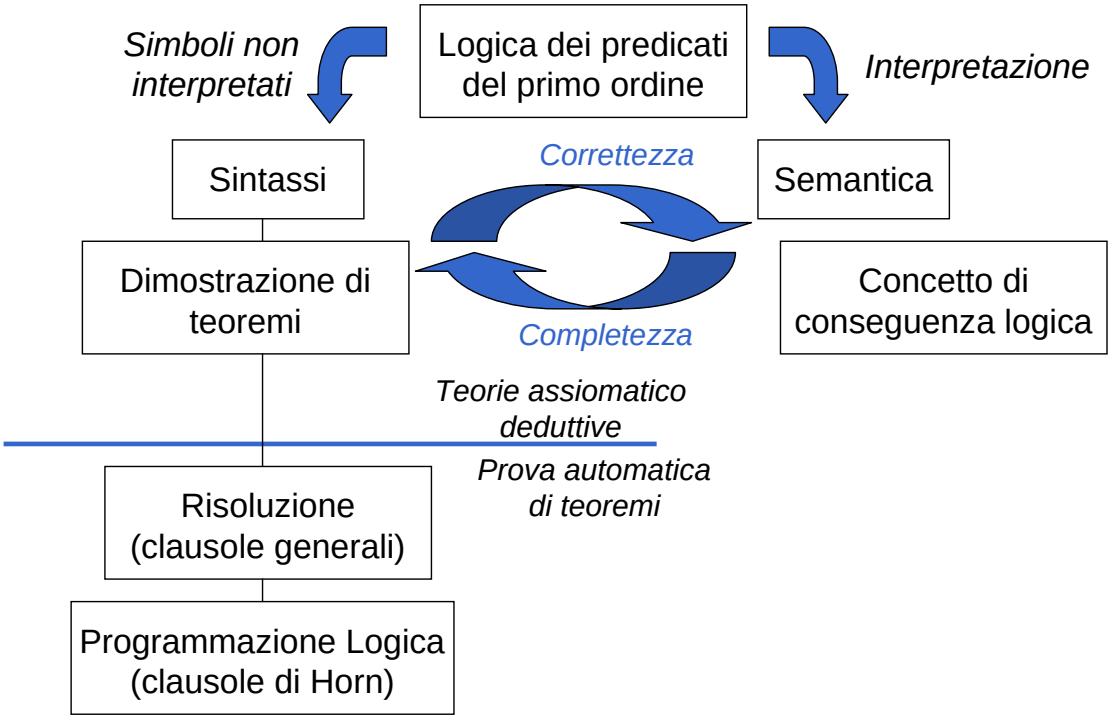
  `:- p(t1,t2,…,tm).`

  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

  `σ = {X1/s1,X2/s2,…,Xn/sn}`

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

SCHEMA RIASSUNTIVO



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 (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 SLD

- Risoluzione Lineare per Clausole Definite con funzione di Selezione (backward chaining)
  - completa per le clausole di Horn
- Dato un programma logico **P** e una clausola goal G_0, ad ogni passo di risoluzione si ricava un nuovo risolvente G_{i+1}, se esiste, dalla clausola goal ottenuta al passo precedente G_i e da una variante di una clausola appartenente a **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, 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 e' ottenuto da Gi riscrivendo l'atomo selezionato con la parte destra della clausola Ci ed applicando la sostituzione θ_i.

- Più in dettaglio:

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

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

UNIFICAZIONE

- L'unificazione è un meccanismo che permette di calcolare una sostituzione al fine di rendere uguali due espressioni. Per espressione intendiamo un termine, un letterale o una congiunzione o disgiunzione di letterali.

- SOSTITUZIONE: $\theta = [X_1/T_1,\ X_2/T_2,\ldots,\ X_n/T_n]$ insieme di legami di termini T_i a variabili X_i che rendono uguali due espressioni.
  L'applicazione di una sostituzione a un'espressione E, $[E]\theta$ produce una nuova espressione in cui vengono sostituite tutte le variabili di E con i corrispondenti termini.

- Esempio: Espressione 1: c(X,Y) Espressione 2: c(a,K)
  sostituzione unificatrice: $\theta = [X/a,\ Y/K]$

UNIFICAZIONE

- COMPOSIZIONE DI SOSTITUZIONI: $\theta_1\theta_2$
  $\theta_1 = [X_1/T_1, X_2/T_2,\ldots, X_n/T_n]$ $\theta_2 = [Y_1/Q_1, Y_2/Q_2,\ldots, Y_n/Q_n]$
  $\theta_1\theta_2 = [X_1/[T_1]\theta_2,\ldots, X_n/[T_n]\theta_2, Y_1/Q_1, Y_2/Q_2,\ldots, Y_n/Q_n]$
  equivale quindi ad applicare prima θ_1 e poi θ_2.

- Esempio: $\theta_1 = [X/f(Z), W/R, S/c]$ $\theta_2 = [Y/X, R/W, Z/b]$
  $\theta_1\theta_2 = [X/f(b), S/c, Y/X, R/W, Z/b]$

- Due atomi : A_1 e A_2 sono *unificabili* se esiste una sostituzione θ tale che $[A_1]\theta = [A_2]\theta$

UNIFICAZIONE

- Una sostituzione θ_1 è più generale di un'altra θ_2 se esiste una terza sostituzione θ_3 tale che $\theta_2 = \theta_1\theta_3$

- Esistono in generale più sostituzioni unificatrici. Noi siamo interessati nell'unificazione più generale: MOST GENERAL UNIFIER

- Esiste un algoritmo che calcola l'unificazione più generale se due atomi sono unificabili, altrimenti termina in tempo finito nel caso in cui i due atomi non sono unificabili.

UNIFICAZIONE

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

OCCUR CHECK

- L'unificazione tra una variabile **`X`** e un termine composto **`S`** è molto delicata: infatti è importante controllare che il termine composto **`S`** non contenga la variabile da unificare **`X`**.

- Questo inficerebbe sia la terminazione, sia la correttezza dell'algoritmo di unificazione.

- Esempio: si consideri l'unificazione tra `p(X,X)` e `p(Y,f(Y)).`
  La sostituzione è `[X/Y,X/f(X)]`
  Chiaramente, due termini unificati con lo stesso termine, sono uguali tra loro. Quindi, `Y/f(Y)` ma questo implica `Y=f(f(f(f(…))))` e il procedimento non termina

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 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).`

DERIVAZIONE SLD (backward chaining)

- Una derivazione SLD per un goal G_0 dall'insieme di clausole definite P e' una sequenza di clausole goal $G_0,\ldots G_n$, una sequenza di varianti di clausole del programma $C_1, \ldots C_n$, e una sequenza di sostituzioni MGU $\theta_1, \ldots, \theta_n$ tali che G_{i+1} è derivato da G_i e da C_{i+1} attraverso la sostituzione θ_n. La sequenza può essere anche infinita.

- Esistono tre tipi di derivazioni;
  - *successo*, 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 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 e' unificabile con alcuna clausola del programma

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).`

...

LEGAMI PER LE VARIABILI IN USCITA

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

Se il goal G_0 è del tipo:

- ¬ $A_1(t_1,...,t_k)$, $A_2(t_{k+1},...,t_h)$, ..., $A_n(t_{j+1},...,t_m)$
- 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∪{G_0} è una sostituzione per le variabili di G_0.

LEGAMI PER LE VARIABILI IN USCITA

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

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*.

- 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 definendo una *strategia di ricerca*.

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.

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.

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 $\texttt{:-}A_1,\ldots,A_{m-1},A_m,A_{m+1},\ldots,A_k$ se A_m è l'atomo selezionato dalla regola di calcolo R, allora questo nodo (*genitore*) ha un nodo *figlio* per ciascuna clausola $C_i = \texttt{A:-}B_1,\ldots,B_q$ 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:

    $\texttt{:-}[A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k]\theta$ 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 ramo di un albero SLD corrisponde una derivazione SLD.
  - Ogni ramo che termina con il nodo vuoto ("**:-**") rappresenta una derivazione SLD di successo.

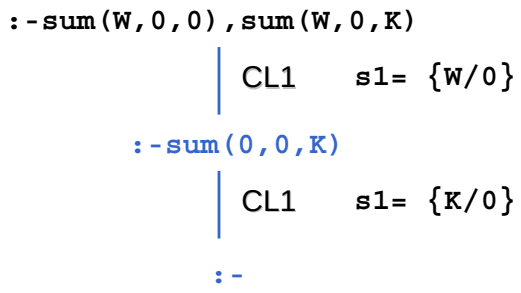
- La regola di calcolo influisce sulla struttura dell'albero per quanto riguarda sia l'ampiezza sia la profondità. Tuttavia non influisce su correttezza e completezza. Quindi, qualunque sia R, il numero di cammini di successo (se in numero finito) è lo stesso in tutti gli alberi SLD costruibili per P $\cup$ {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”

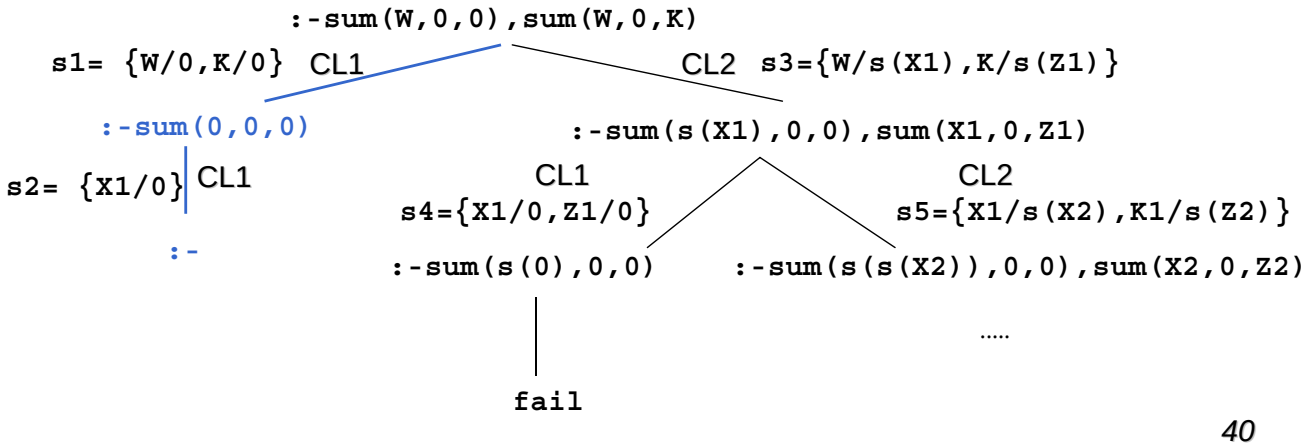


ALBERI SLD: ESEMPIO

```
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”

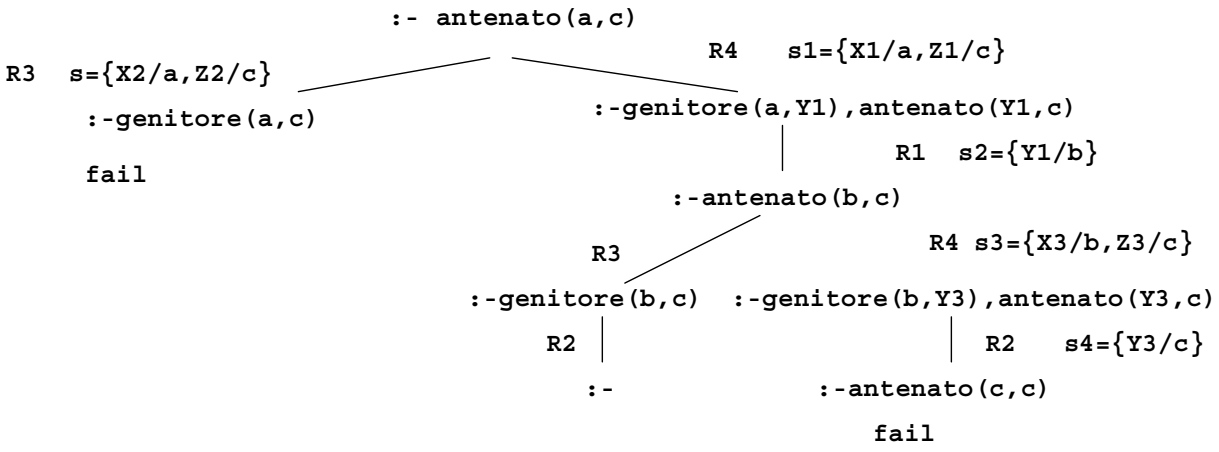


ALBERI SLD: ESEMPIO

- 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 (2)

```
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)

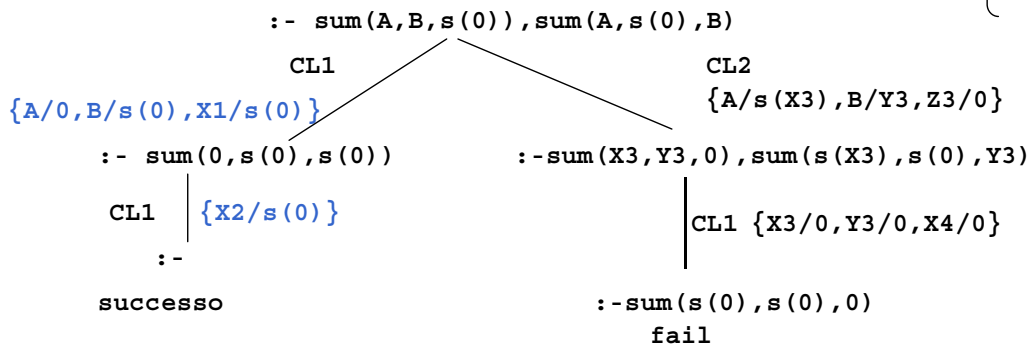
```
sum(0,X,X).                                   (CL1)

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

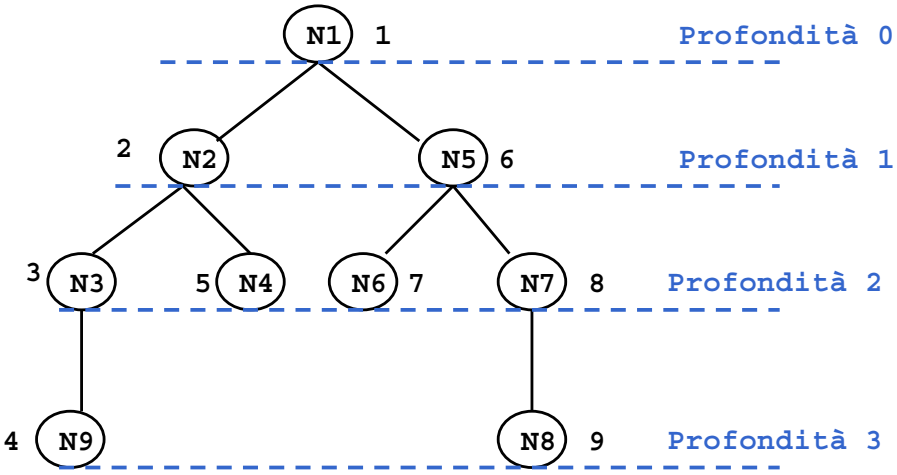
- La realizzazione effettiva di un dimostratore basato sulla risoluzione SLD richiede la definizione non solo di una regola di calcolo, ma anche di una strategia di ricerca che stabilisce una particolare *modalità di esplorazione* dell'albero SLD alla ricerca dei rami di successo.

- Le modalità di esplorazione dell'albero piu' comuni sono:
  - depth first
  - breadth first

- Entrambe le modalità implicano l'esistenza di un meccanismo di backtracking per esplorare tutte le strade alternative che corrispondono ai diversi nodi dell'albero.

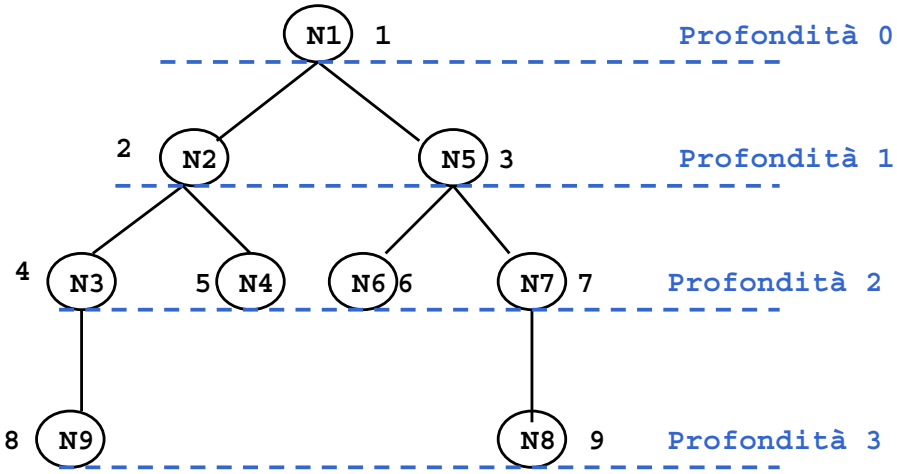
STRATEGIA DEPTH-FIRST

– Ricerca in profondità: vengono prima esplorati i nodi a profondità maggiore. NON COMPLETA



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.

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

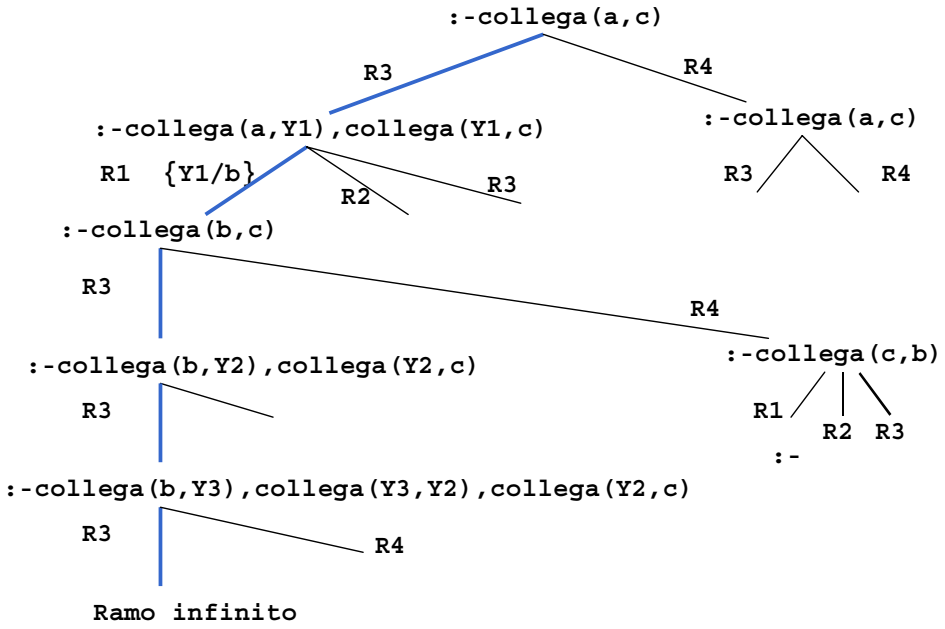
- La strategia di ricerca adottata in Prolog è dunque non completa.

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 non completa come quella che adotta la strategia in profondità 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 computazione
  - la strategia di ricerca

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

  - *Regola di computazione*
    - 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*
    - 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.

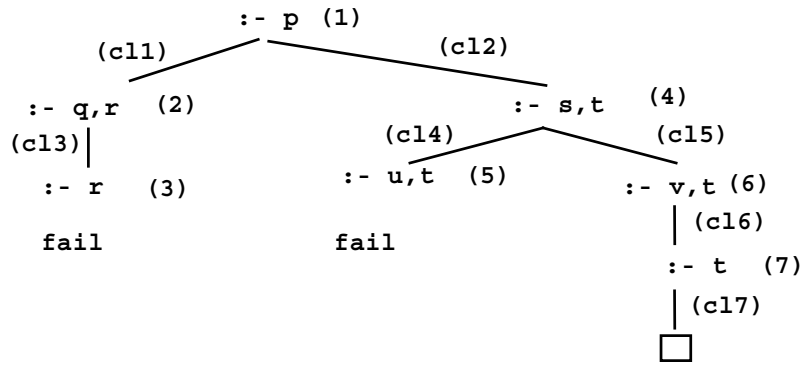
- Nel caso vi siano più clausole la cui testa è unificabile con G_1, la risoluzione di G_1 viene considerata come un *punto di scelta* (*choice point*) nella dimostrazione.

- In caso di fallimento in un passo di dimostrazione, 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

```
P1          (cl1)  p  :-  q,r.
            (cl2)  p  :-  s,t
            (cl3)  q.
            (cl4)  s  :-  u.
            (cl5)  s  :-  v.
            (cl6)  t.
            (cl7)  v.
  :- p.
```

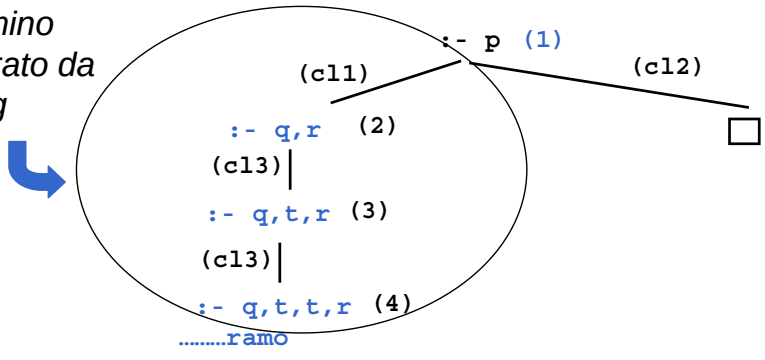


RISOLUZIONE IN PROLOG: INCOMPLETEZZA

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

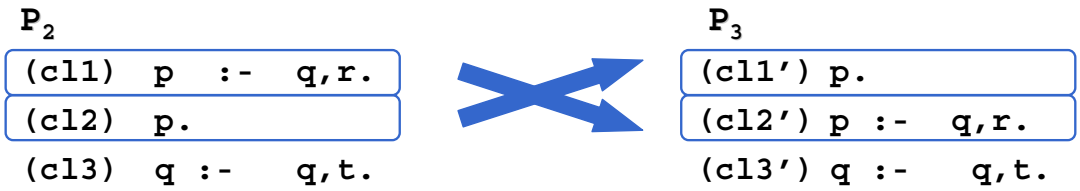
```
P2          (cl1)  p  :-  q,r.
            (cl2)  p.
            (cl3)  q  :-  q,t.
           :- p.
```

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.

- 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.

ORDINE DELLE CLAUSOLE: ESEMPIO

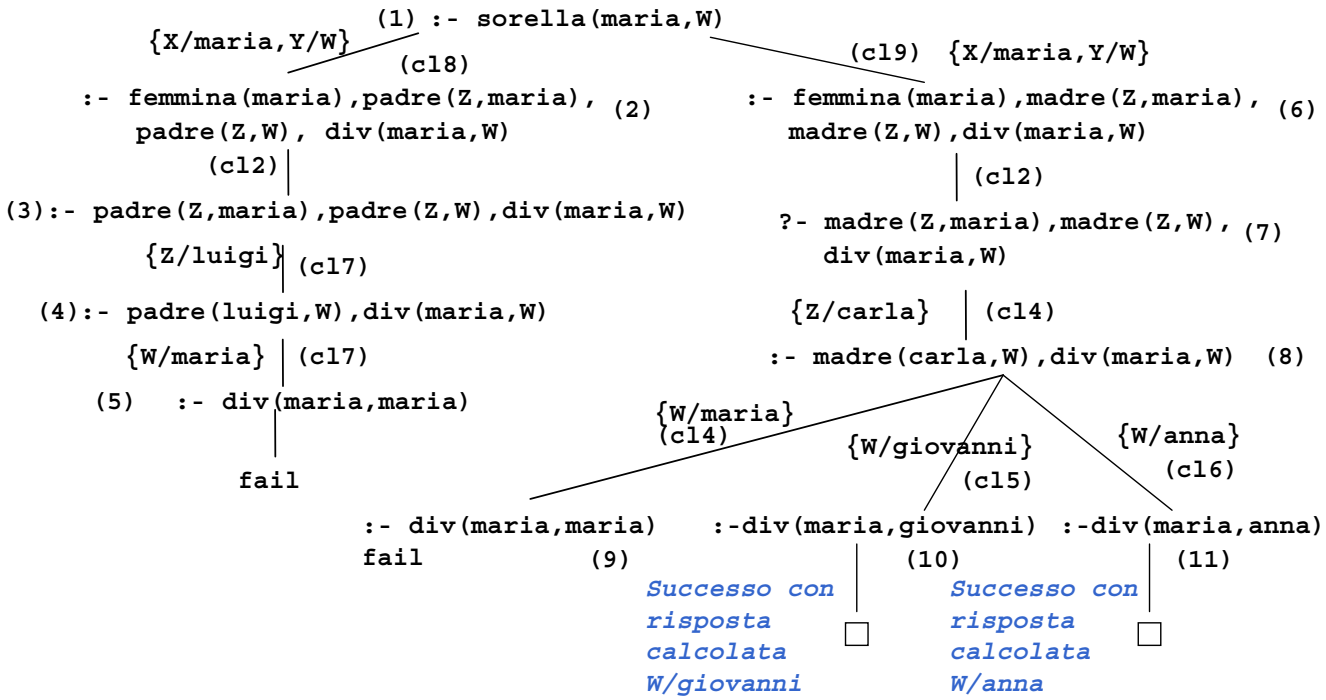
P_4

```
(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 $A \neq B$

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).
yes     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 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`. Gli argomenti di `p` (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 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.

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.

ESERCIZIO: Interprete in Pascal-like

```
function interpreta(Goal_list: list of atom, var Soluz: sostituzione):
  boolean;
  var n:integer; k,successo:boolean; Clause:clausola;
      Unif:sostituzione; Body,Goal_list1:list of atom;  Goal:atom;
begin
        Goal:=first(Goal_list); n:=1; successo:=false;
        k:=seleziona_clausola(n,Goal,Clause,Unif);
        while k=true and ¬successo do
        begin
                Body:=corpo(Clause);
                Body:=sostituisci(Unif,Body);
                Goal_list1:=rest(Goal_list);
                Goal_list1:=sostituisci(Unif,Goal_list1);
                Goal_list1:=concatena(Body,Goal_list1);
                successo:=interpreta(Goal_list1,Unif1);
                if ¬successo then
                   begin n:=n+1;
                   k:=seleziona_clausola(n,Goal,Clause,Unif);
                   end
        end;
        if successo  then  Soluz:=compose(Unif,Unif1);
        interpreta := successo
end;
```

ESERCIZIO: Interprete in Pascal-like

- **`compose(X,Y:sostituzione): sostituzione`**

  composizione delle sostituzioni X e Y.

- **`seleziona_clausola(n:integer, Goal:atom,`**
  `var Clause:clausola, Unif: sostituzione): boolean`

  se esiste una clausola "n-esima" la cui testa è unificabile con Goal, ha valore **`true`** e restituisce in **`Clause`** la clausola stessa; se la clausola non esiste, la funzione ha valore false.

- **`corpo(Clause:clausola):list of atoms`**

  è una funzione che ha come valore la lista di atomi che costituiscono il corpo della clausola Clause.

ESERCIZIO: Interprete in Pascal-like

- **`sostituisci(List:list of atom, Unif: sostituzione): list of atom`**

  è una funzione che ha come valore la lista di atomi che si ottiene effettuando la sostituzione "**`Unif`**" sulla lista di atomi "**`List`**".

- "**`first`**" e "**`rest`**" ritornano, rispettivamente il primo elemento di una lista e la lista senza il primo elemento.

- "**`concatena`**" è una operazione di concatenazione tra liste.