

LINGUAGGIO PROLOG

- PROLOG: PROgramming in LOGic, nato nel 1973
 - È 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
 - in Europa...

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 presunti
 - linguaggio relativamente giovane
 - efficienza non massima
 - non adatto ad applicazioni numeriche o in tempo reale
 - paradigma non familiare
-

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:
 - $A1 = L + C1$
 - $A2 = L + C2$
 -

ALGORITMO = LOGICA + CONTROLLO

- ESEMPIO: Decidere se i profili di due alberi sono uguali
-
- Diverse strategie di controllo che influenzano l'efficienza:
 - trovare sequenzialmente il primo profilo, poi il secondo e confrontarli
 - trovare parallelamente i due profili e confrontarli
 - trovare la prima foglia del primo profilo e del secondo, confrontarle e cercare le successive se sono uguali

`stesso_profilo(X,Y):- profilo(X,W) , profilo(Y,Z) .`

`profilo(l(X) , [X]) .`

`profilo(t(l(X) , Z) , [X|Y]) :- profilo(Z,Y) .`

`profilo(t(t(X,Y) , Z) , W) :- profilo(t(X,t(Y,Z)) , W) .`

-

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), sulla base della conoscenza definita
-
- ESEMPIO: due individui sono colleghi se lavorano per la stessa ditta
- | | | |
|------------------------------|---|-----------------------------|
| Testa | Body | |
| $\underbrace{\hspace{10em}}$ | $\underbrace{\hspace{15em}}$ | |
| <code>collega(X,Y) :-</code> | <code>lavora(X,Z) , lavora(Y,Z) , diverso(X,Y)</code> | $\Rightarrow$ REGOLA |
-
- | | |
|--------------------------------------|----------------|
| <code>lavora(emp1,ibm) .</code> | } FATTI |
| <code>lavora(emp2,ibm) .</code> | |
| <code>lavora(emp3,txt) .</code> | |
| <code>lavora(emp4,olivetti) .</code> | |
| <code>lavora(emp5,txt) .</code> | |
-
- `:- collega(X,Y) .` $\Rightarrow$ **GOAL**
-

PROLOG: ELABORATORE DI SIMBOLI

- ESEMPIO: somma di due numeri interi

-

$\text{sum}(0, X, X) \Longrightarrow$ **FATTO**

$\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) \Longrightarrow$ **REGOLA**

-

- Simbolo **sum** non interpretato.
- Numeri interi interpretati dalla struttura “successore” **s(x)**
- Si utilizza la ricorsione
- Esistono molti possibili interrogazioni

-

- $:- \text{sum}(s(0), s(s(0)), Y) .$
- $:- \text{sum}(s(0), Y, s(s(s(0)))) .$
- $:- \text{sum}(X, Y, s(s(s(0)))) .$
- $:- \text{sum}(X, Y, Z) .$
- $:- \text{sum}(X, Y, s(s(s(0)))) , \text{sum}(X, s(0), Y) .$

-

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 è un fatto, la prova termina;
 - se la clausola con cui unifica è una regola, ne viene provato il Body
-
-

PROVA DI UN GOAL: esempio

`append([ ], X, X) .`

`append([X|T1], Y, [X|T2]) :- append(T1, Y, T2) .`

`:- 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|T1], Y, [X|T2]) :- append(T1, Y, T2) .`

`:- 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 è 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 è costituito da un insieme di clausole definite della forma
-

(c11) $A. \implies$ **FATTO o ASSERTZIONE**

(c12) $A :- B1, B2, \dots, Bn. \implies$ **REGOLA**

(c13) $:- B1, B2, \dots, Bn. \implies$ **GOAL**

-
- In cui A e B_i sono formule atomiche
- A : testa della clausola
- $B1, B2, \dots, Bn$: body della clausola
- Il simbolo “,” indica la congiunzione; il simbolo “:-” l’implicazione logica in cui A è il conseguente e $B1, B2, \dots, Bn$ l’antecedente

PIU' FORMALMENTE

-
- Una formula atomica è una formula del tipo
$$p(t_1, t_2, \dots, t_n)$$
- in cui p è un simbolo predicativo 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 (operatore) a k argomenti e $t_1, t_2, \dots, t_k$ sono termini. $f(t_1, t_2, \dots, t_k)$ 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'è distinzione sintattica tra costanti, simboli funzionali e predicativi.

INTERPRETAZIONE DICHIARATIVA

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

$$p(t_1, t_2, \dots, t_m).$$

- se $x_1, x_2, \dots, x_n$ sono le variabili che compaiono in $t_1, t_2, \dots, t_m$ il significato è: $\forall x_1, \forall x_2, \dots, \forall x_n (p(t_1, t_2, \dots, t_m))$
- per ogni regola del tipo

$$A: - B_1, B_2, \dots, B_k.$$

- se $y_1, y_2, \dots, y_n$ sono le variabili che compaiono solo nel body della regola e $x_1, x_2, \dots, x_n$ sono le variabili che compaiono nella testa e nel corpo, il significato è:

$$\forall x_1, \forall x_2, \dots, \forall x_n, \forall y_1, \forall y_2, \dots, \forall y_n ((B_1, B_2, \dots, B_k) \quad A)$$

$$\forall x_1, \forall x_2, \dots, \forall x_n, (\forall y_1, \forall y_2, \dots, \forall y_n (B_1, B_2, \dots, B_k)) \quad A)$$

INTERPRETAZIONE DICHIARATIVA

-

- ESEMPI

`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 padre di z e z è il padre di y”

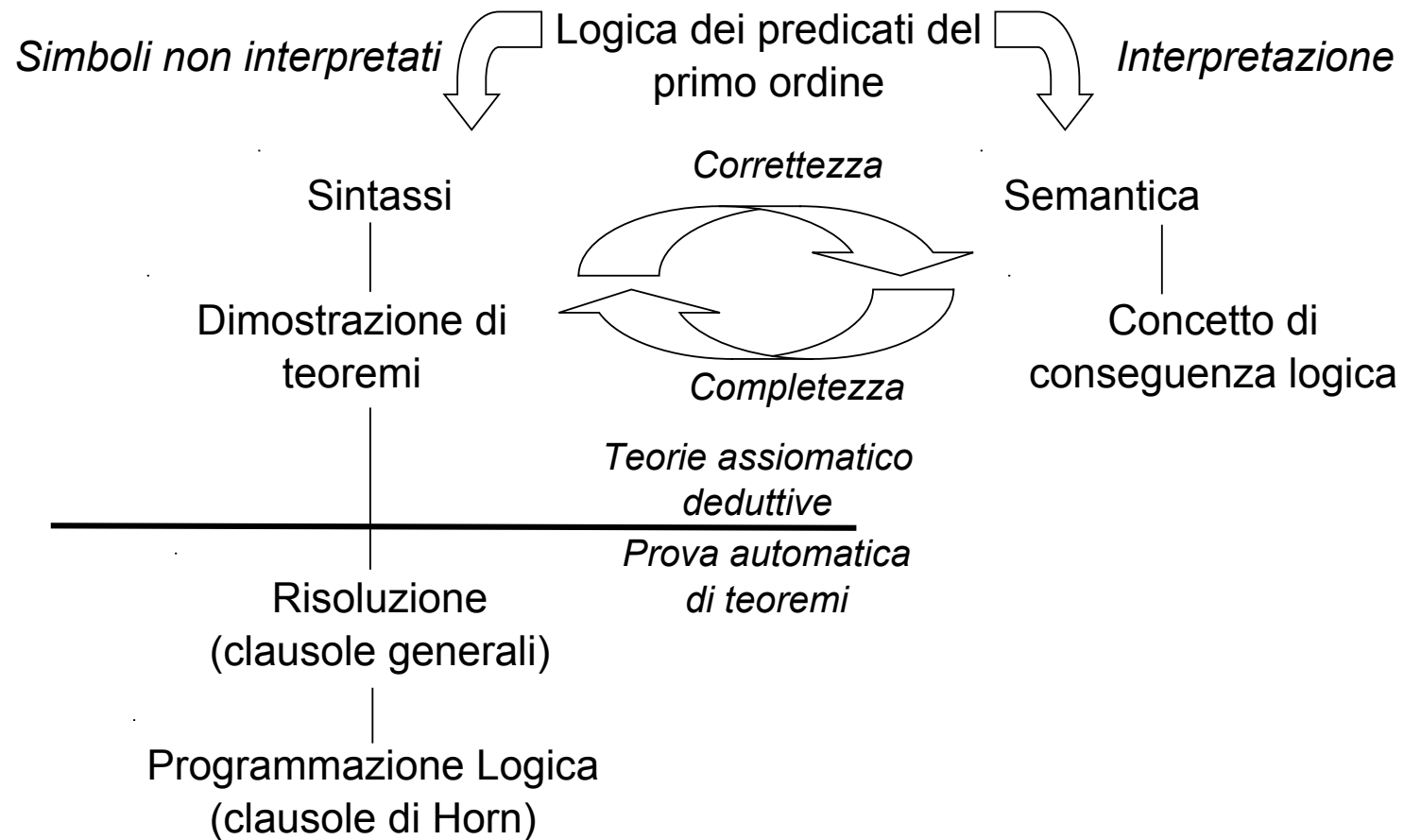
`nonno (X, Y) :- padre (X, Z) , madre (Z, Y) .`

- “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:
$$:- p(t_1, t_2, \dots, t_m) .$$
- se $x_1, x_2, \dots, x_n$ sono le variabili che compaiono in $t_1, t_2, \dots, t_m$ il significato della query è: $\exists x_1, \exists x_2, \dots, \exists x_n p(t_1, t_2, \dots, t_m)$ e l’obiettivo è quello di trovare una sostituzione
$$\theta = \{x_1/s_1, x_2/s_2, \dots, x_n/s_n\}$$
- dove s_i sono termini tale per cui $P \models [p(t_1, t_2, \dots, t_m)]\theta$

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
 - 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
 - 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
-
- Un variante per una clausola C è la clausola C' ottenuta da C rinominando le sue variabili.
 - Esempio:
 - $p(X) :- q(X, g(Z)) .$
 - $p(X1) :- q(X1, g(Z1)) .$

RISOLUZIONE SLD (continua)

-
- La Risoluzione SLD seleziona un atomo A_m dal goal G_i secondo un determinato criterio, e lo unifica se possibile con la testa della clausola C_i attraverso la sostituzione più generale: MOST GENERAL UNIFIER (MGU) θ_i
-
- Il nuovo risolvente è ottenuto da G_i riscrivendo l'atomo selezionato con la parte destra della clausola C_i ed applicando la sostituzione θ_i .
-
- Più in dettaglio, dati
 - $\vdash A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$. (risolvente)
 - $A \vdash B_1, \dots, B_q$. (clausola del programma P)
 - Se $[A_m]\theta_i = [A]\theta_i$ allora la risoluzione SLD deriva il nuovo risolvente
 - $\vdash [A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k]\theta_i$.

RISOLUZIONE SLD: ESEMPIO

- - $\text{sum}(0, X, X) .$ (C1)
 - $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (C2)
-
- Goal
 - $\text{sum}(s(0), 0, W) .$
- Al primo passo genero una variante della clausola (C2)
 - $\text{sum}(s(X1), Y1, s(Z1)) :- \text{sum}(X1, Y1, Z1) .$
 -
- Unificando la testa con il goal ottengo la sostituzione MGU
 - $\theta_1 = [X1/0, Y1/0, W/s(Z1)]$
- ottengo il nuovo risolvete
 - $G1: :- [\text{sum}(X1, Y1, Z1)] \theta_1$
- ossia
 - $:- \text{sum}(0, 0, Z1) .$

DERIVAZIONE SLD

- Una derivazione SLD per un goal G_0 dall'insieme di clausole definite P è una sequenza di clausole goal $G_0, \dots, G_n$, una sequenza di varianti di clausole del programma $C_1, \dots, C_n$, e una sequenza di sostituzioni MGU $\sigma_1, \dots, \sigma_n$ tali che G_{i+1} è derivato da G_i e da C_{i+1} attraverso la sostituzione σ_i . La sequenza può essere anche infinita.
-
- Esistono tre tipi di derivazioni;
 - *successo*, se per n finito G_n è uguale alla clausola vuota $G_n = :-$
 - *fallimento finito*: se per n finito non è più possibile derivare un nuovo risolvante da G_n e G_n non è uguale a $:-$
 - *fallimento infinito*: se è sempre possibile derivare nuovi risolventi tutti diversi dalla clausola vuota.
 -

DERIVAZIONE DI SUCCESSO

- - $\text{sum}(0, X, X) .$ (CL1)
 - $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (CL2)
-
- Goal $G_0 :- \text{sum}(s(0), 0, w)$ ha una derivazione di successo
 - C1: variante di CL2 $\text{sum}(s(X_1), Y_1, s(Z_1)) :- \text{sum}(X_1, Y_1, Z_1) .$
 - $\square_1 = [x_1/0, y_1/0, w/s(Z_1)]$
 - $G_1 :- \text{sum}(0, 0, Z_1) .$
 - C2: variante di CL1 $\text{sum}(0, X_2, X_2) .$
 - $\square_2 = [z_1/0, x_2/0]$
 - $G_2 :-$
 - $\square_1 \square_2 = [x_1/0, y_1/0, w/s(Z_1), z_1/0, x_2/0]$
 - Derivazione di successo con $w/s(0)$

DERIVAZIONE DI FALLIMENTO FINITA

- - $\text{sum}(0, X, X) .$ (CL1)
 - $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (CL2)
-
- Goal $G_0 :- \text{sum}(s(0), 0, 0)$ ha una derivazione di fallimento finito perché l'unico atomo del goal non è unificabile con alcuna clausola del programma

DERIVAZIONE DI FALLIMENTO INFINITA

- - $\text{sum}(0, X, X) .$ (CL1)
 - $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (CL2)
-
- Goal $G_0 :- \text{sum}(A, B, C)$ ha una derivazione SLD infinita, ottenuta applicando ripetutamente varianti della seconda clausola di P
 - C1: variante di CL2 $\text{sum}(s(X_1), Y_1, s(Z_1)) :- \text{sum}(X_1, Y_1, Z_1) .$
 - $\theta_1 = [A/s(X_1), B/Y_1, C/s(Z_1)]$
 - $G_1 :- \text{sum}(X_1, Y_1, Z_1) .$
 - C2: variante di CL2 $\text{sum}(s(X_2), Y_2, s(Z_2)) :- \text{sum}(X_2, Y_2, Z_2) .$
 - $\theta_2 = [X_1/s(X_2), Y_1/Y_2, Z_1/s(Z_2)]$
 - $G_2 :- \text{sum}(X_2, Y_2, Z_2) .$
 - ...

LEGAMI PER LE VARIABILI IN USCITA

- Risultato della computazione:
 - eventuale successo
 - *legami* per le variabili del goal G_0 , ottenuti componendo le sostituzioni MGU applicate
-
- Se il goal G_0 è del tipo:
 - $\neg A_1(t_1, \dots, t_k), A_2(t_{k+1}, \dots, t_h), \dots, A_n(t_{j+1}, \dots, t_m)$
 - 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 $P \sqcup \{G_0\}$ è una sostituzione per le variabili di G_0 .

LEGAMI PER LE VARIABILI IN USCITA

- Si consideri una refutazione SLD per $P \sqsubseteq \{G_0\}$. Una *risposta calcolata* q per $P \sqsubseteq \{G_0\}$ è la sostituzione ottenuta restringendo la composizione delle sostituzioni mgu $q_1, \dots, q_n$ utilizzate nella refutazione SLD di $P \sqsubseteq \{G_0\}$ 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).
- - $\text{sum}(0, X, X) .$ (CL1)
 - $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (CL2)
 - $G = :- \text{sum}(s(0), 0, w)$ la sostituzione $\sigma = \{w/s(0)\}$ è la risposta calcolata, ottenuta componendo σ_1 con σ_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

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

-

- $\text{sum}(0, X, X) . \quad (\text{CL1})$
- $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) . \quad (\text{CL2})$
- $G0 = :- \text{sum}(0, s(0), s(0)) , \text{sum}(s(0), 0, s(0)) .$
- Se si seleziona l'atomo più a sinistra al primo passo, unificando l'atomo $\text{sum}(0, s(0), s(0))$ con la testa di CL1, si otterrà:
- $G1 = :- \text{sum}(s(0), 0, s(0)) .$
- Se si seleziona l'atomo più a destra al primo passo, unificando l'atomo $\text{sum}(s(0), 0, s(0))$ con la testa di CL2, si avrà:
- $G1 = :- \text{sum}(0, s(0), s(0)) , \text{sum}(0, 0, 0)$

INDIPENDENZA DALLA REGOLA DI CALCOLO

-
- La regola di calcolo influenza solo l'efficienza
- Non influenza nè la correttezza nè 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.
 -
 - $\text{sum}(0, X, X) .$ (CL1)
 - $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (CL2)
 - $G0 = :- \text{sum}(W, 0, K) .$
 - Se si sceglie la clausola CL1 si ottiene il risolvente
 - $G1 = :-$
 - Se si sceglie la clausola CL2 si ottiene il risolvente
 - $G1 = :- \text{sum}(X1, 0, Z1) .$

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 $P \sqcup \{G_0\}$ 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 $:-A_1, \dots, A_{m-1}, A_m, A_{m+1}, \dots, A_k$ se A_m è l'atomo selezionato dalla regola di calcolo R , allora questo nodo (*genitore*) ha un nodo *figlio* per ciascuna clausola $C_i = A:-B_1, \dots, B_q$ 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:
 - $:-[A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k]\theta$ 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 $P \sqsubseteq \{G_0\}$.
- R influenza solo il numero di cammini di fallimento (finiti ed infiniti).
 -

ALBERI SLD: ESEMPIO

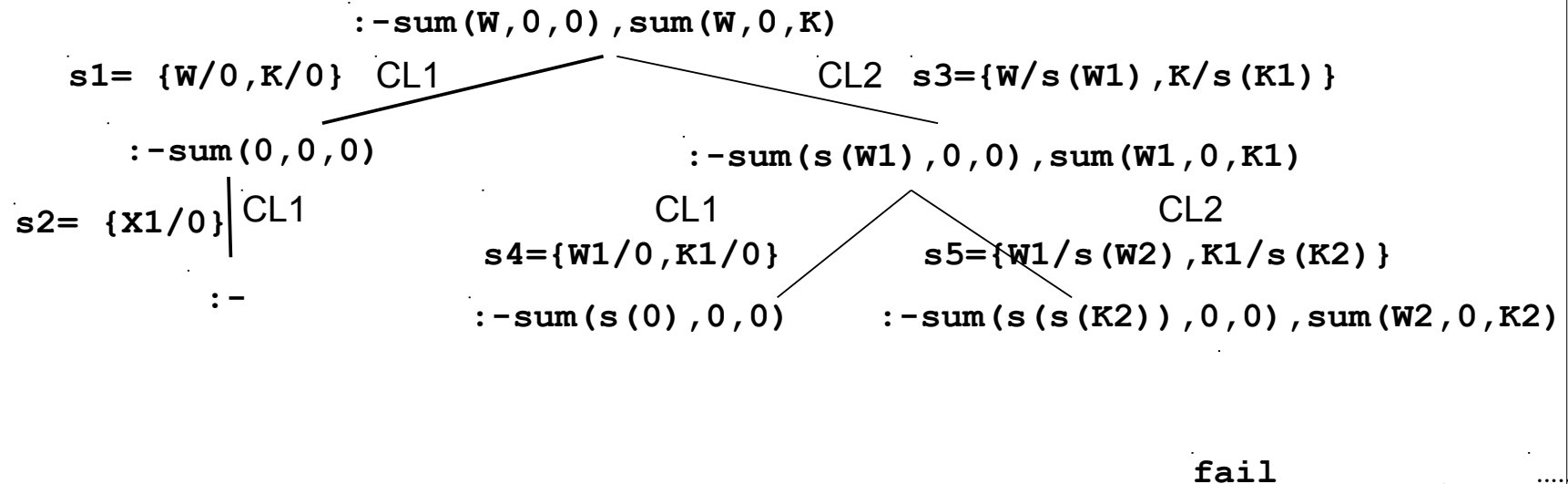
- $\text{sum}(0, X, X) .$ (CL1)
- $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) .$ (CL2)
- $G0 = :- \text{sum}(W, 0, 0), \text{sum}(W, 0, K) .$
-
- Albero SLD con regola di calcolo “left-most”

```

      :-sum(W, 0, 0), sum(W, 0, K)
          | CL1    s1= {W/0}
      :-sum(0, 0, K)
          | CL1    s1= {K/0}
      :-
  
```

ALBERI SLD: ESEMPIO

- $\text{sum}(0, X, X) . \quad (\text{CL1})$
- $\text{sum}(s(X), Y, s(Z)) :- \text{sum}(X, Y, Z) . \quad (\text{CL2})$
- $G0 = :- \text{sum}(W, 0, 0), \text{sum}(W, 0, K) .$
-
- Albero SLD con regola di calcolo “right- most”



ALBERI SLD: ESEMPIO

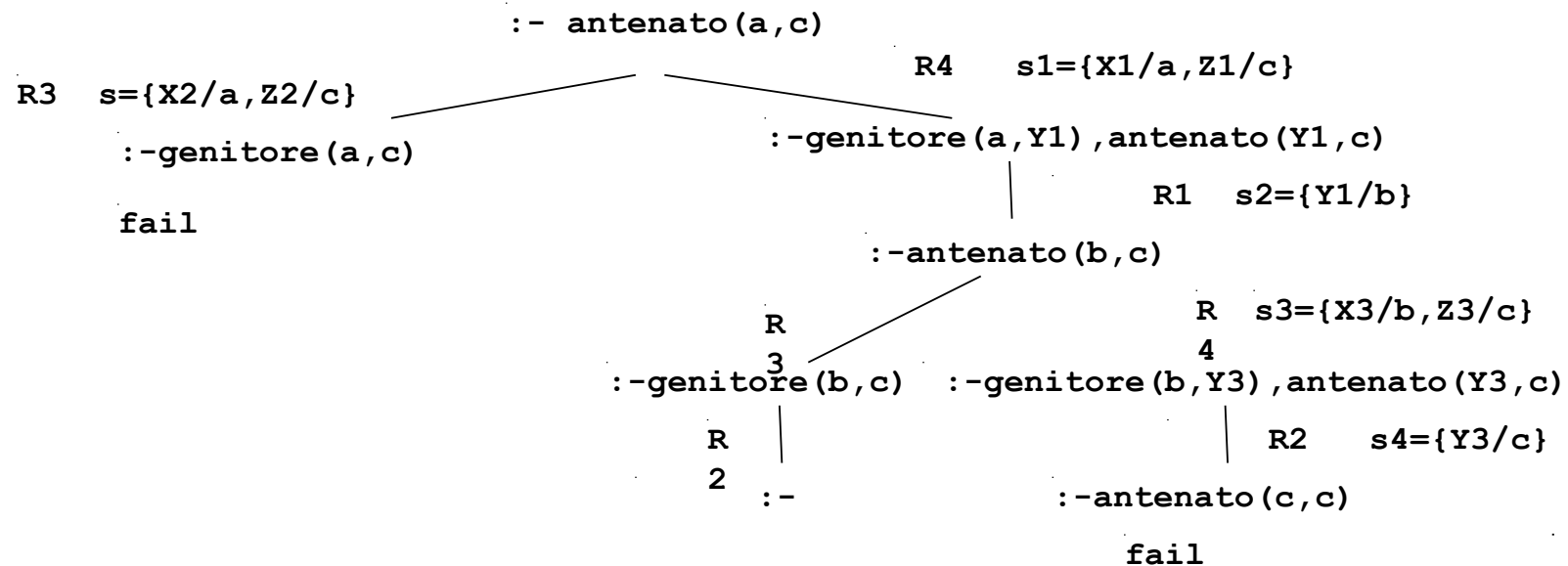
-
- Confronto albero SLD con regola di calcolo left most e right most:
 - In entrambi gli alberi esiste una refutazione SLD, cioè un cammino (ramo) di successo il cui nodo finale è etichettato con ":-".
 -
- La composizione delle sostituzioni applicate lungo tale cammino genera la sostituzione di risposta calcolata $\{\mathbf{w}/\mathbf{o}, \mathbf{\kappa}/\mathbf{o}\}$.
-
- 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)

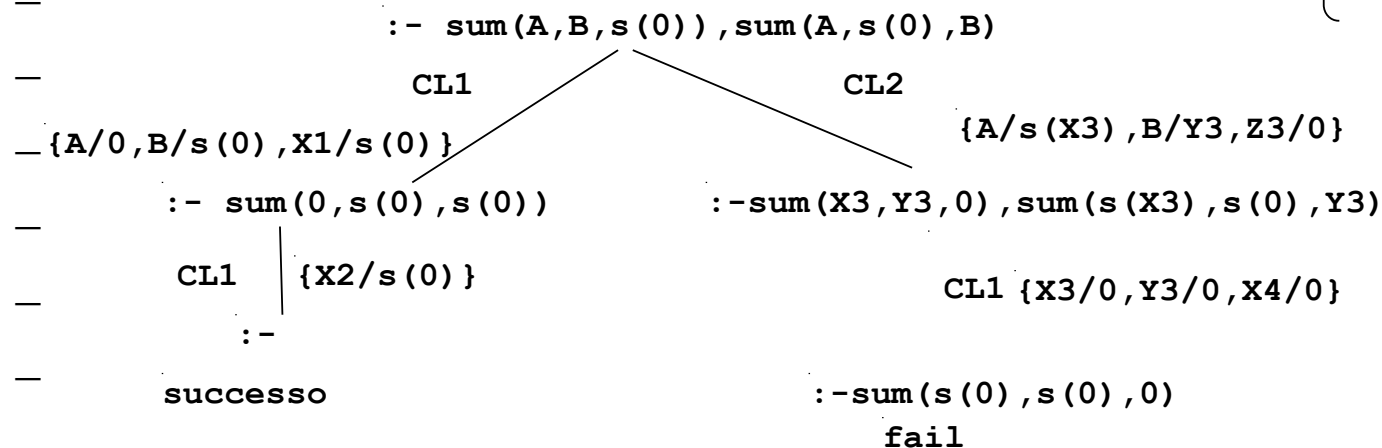
– `sum(0,X,X) .` (CL1)

– `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

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



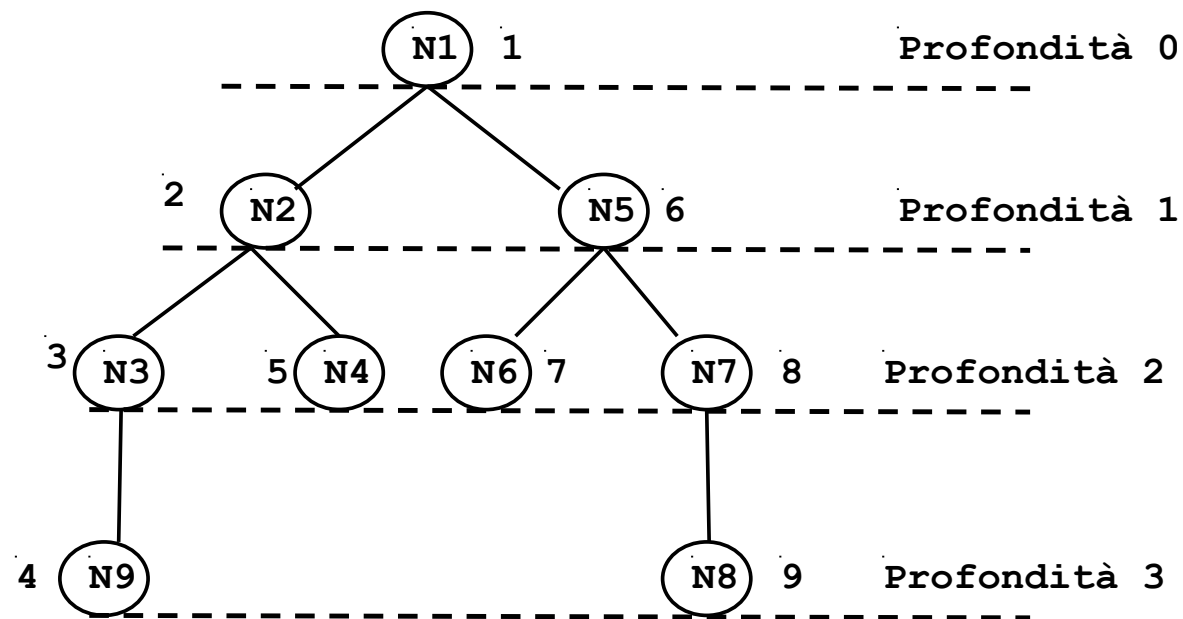
# STRATEGIA DI RICERCA

---

- 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.

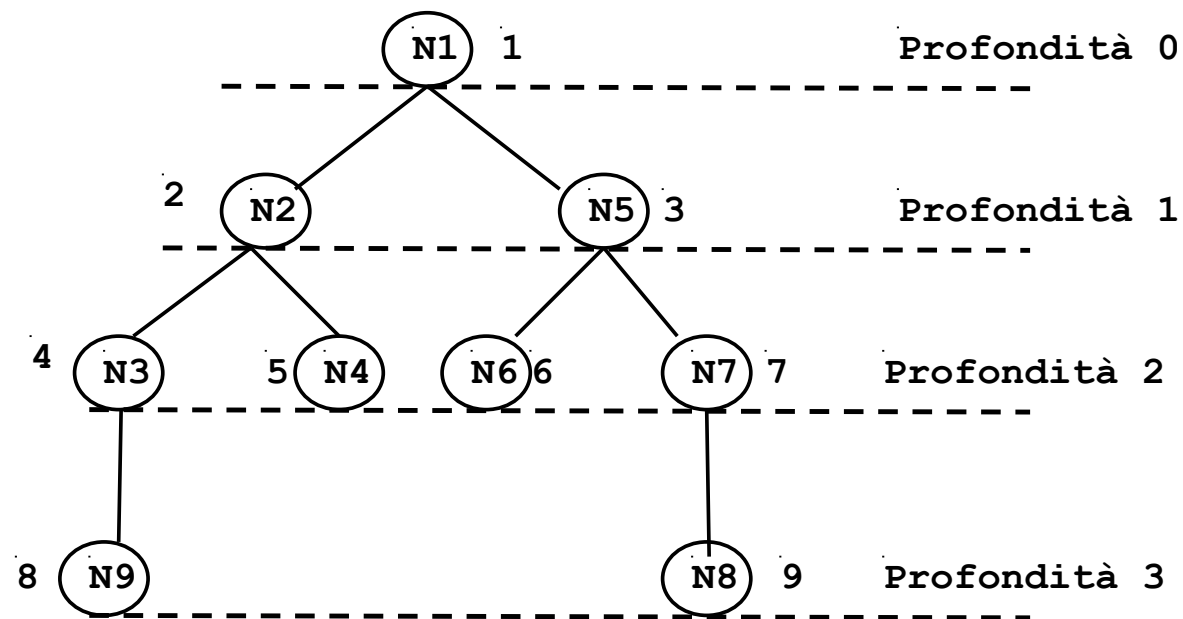
# STRATEGIA DEPTH-FIRST

- 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

---

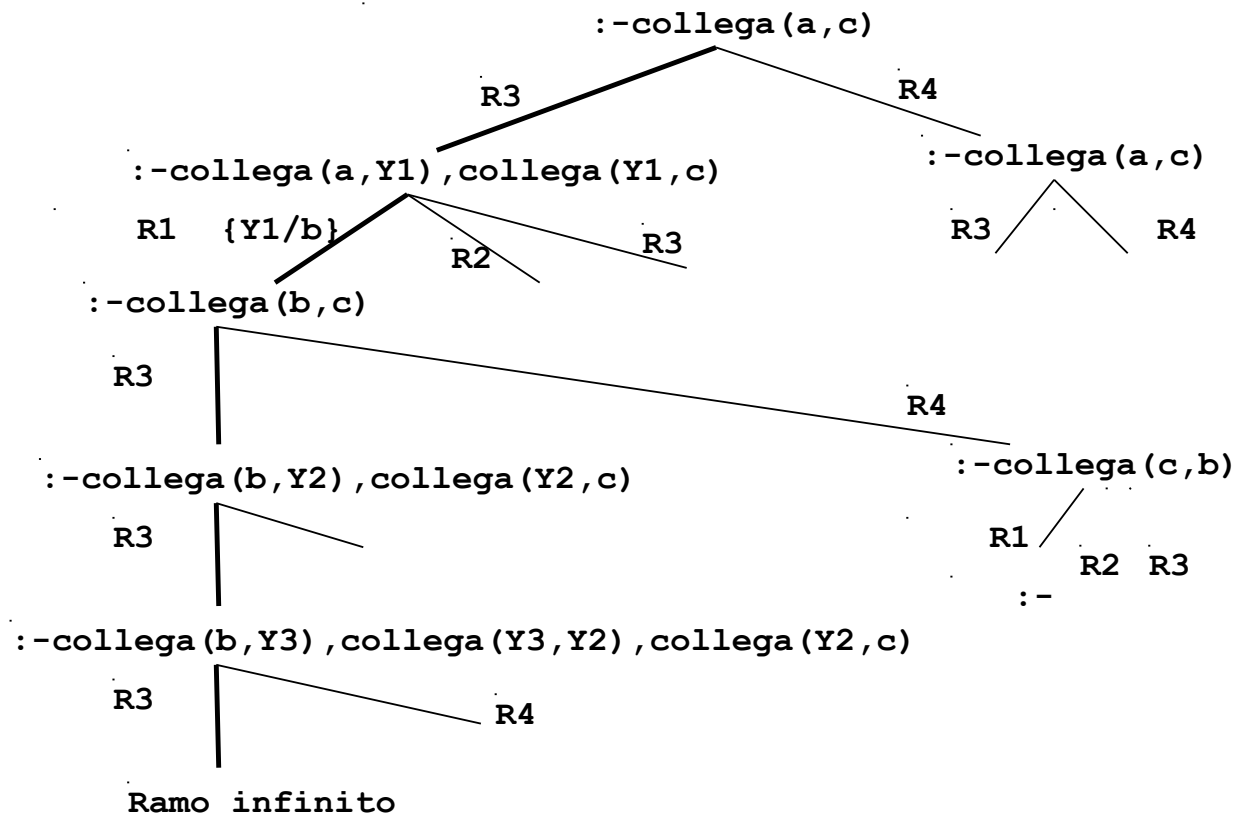
- Il linguaggio Prolog adotta la *strategia in profondità con "backtracking"* perché può essere realizzata in modo efficiente attraverso un unico stack di goal.
  - tale stack rappresenta il ramo che si sta esplorando e contiene opportuni riferimenti a rami alternativi da esplorare in caso di fallimento.
- 
- 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":  
 $?- G_1, G_2, \dots, G_n.$
  - - viene sempre selezionato il letterale più a sinistra  $G_1$ .
  - *Strategia di ricerca*
    - In *profondità (depth-first)* con *backtracking cronologico*.

# RISOLUZIONE IN PROLOG

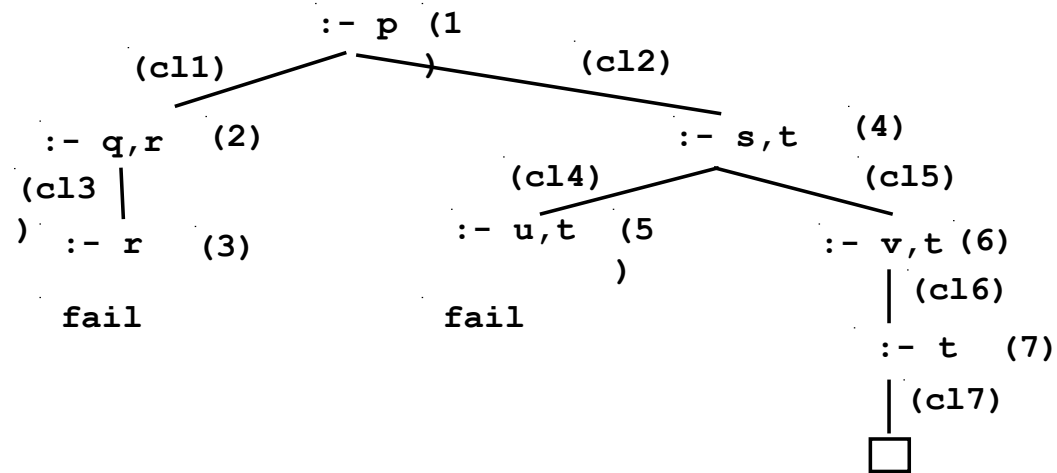
---

- 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

P<sub>1</sub>      (c11) p :- q,r.  
          (c12) p :- s,t  
              (c13) q.  
          (c14) s :- u.  
          (c15) s :- v.  
          (c16) t.  
          (c17) v.  
         :- p.



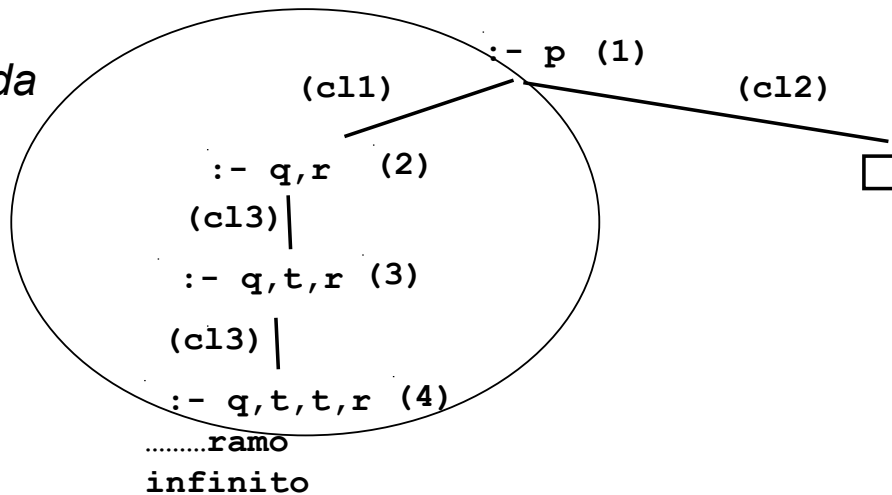
# RISOLUZIONE IN PROLOG: INCOMPLETEZZA

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

- 

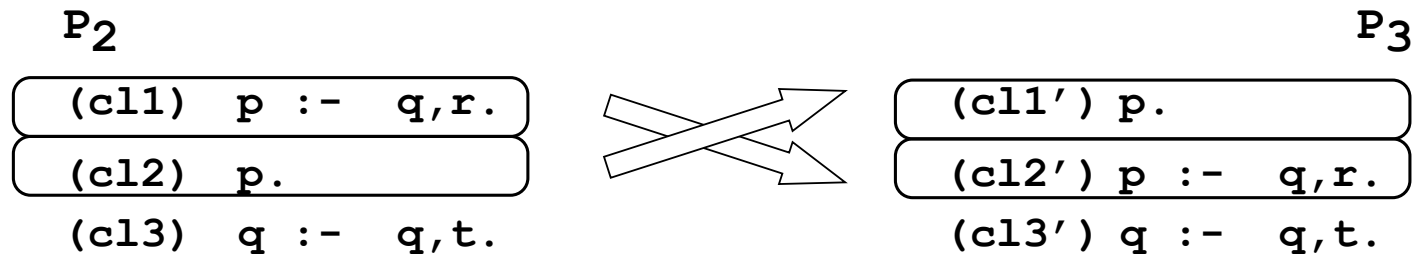
P<sub>2</sub>      (c11) p :- q,r.  
          (c12) p.  
          (c13) q :- q,t.  
          :- p.

*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.
-

# ORDINE DELLE CLAUSOLE: ESEMPIO

P<sub>4</sub>

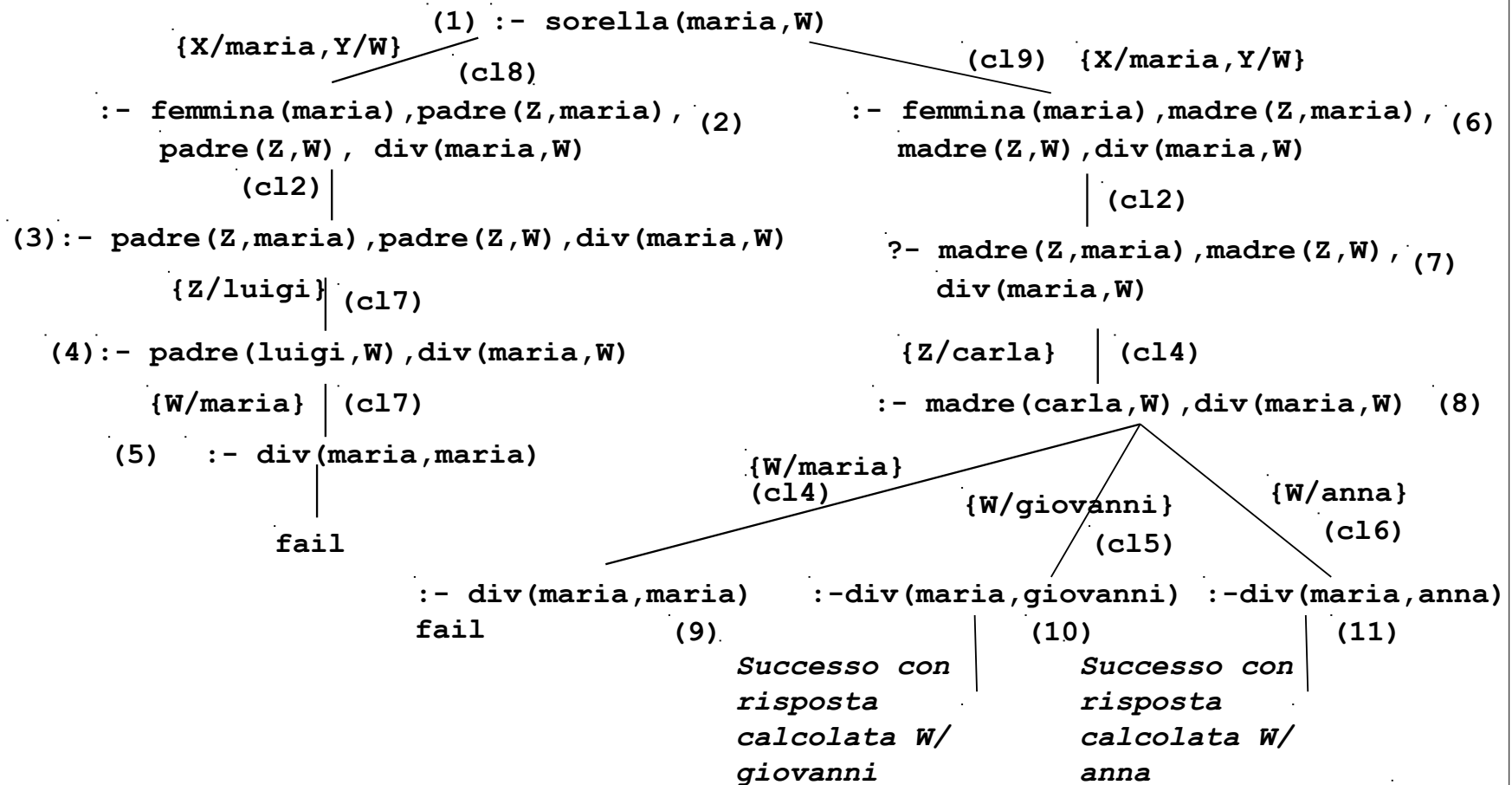
```
(c11) femmina(carla)
(c12) femmina(maria) .
(c13) femmina(anna) .
(c14) madre(carla,maria) .
(c15) madre(carla,giovanni) .
(c16) madre(carla,anna) .
(c17) padre(luigi,maria) .
(c18) sorella(X,Y) :- femmina(X) ,
 padre(Z,X) ,
 padre(Z,Y) ,
 div(X,Y) .

(c19) sorella(X,Y) :- femmina(X) ,
 madre(Z,X) ,
 madre(Z,Y) ,
 div(X,Y) .

(c110) div(carla,maria) .
(c111) 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) .
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: 
$$:- p(t_1, t_2, \dots, t_n) .$$
  - è la *chiamata* della procedura  $p$ . Gli argomenti di  $p$  (ossia i termini  $t_1, t_2, \dots, t_n$ ) 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 è 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.

—

# ARITMETICA E RICORSIONE

- Non esiste, in logica, alcun meccanismo per la *valutazione* di funzioni, operazione fondamentale in un linguaggio di programmazione

- 

- I numeri interi possono essere rappresentati come termini Prolog.

- Il numero intero N è rappresentato dal termine:

`s(s(s(...s(0)...)))`



*N volte*

----- `prodotto(X, Y, Z)` "Z è il prodotto di X e Y"

`prodotto(X, 0, 0).`

`prodotto(X, s(Y), Z) :- prodotto(X, Y, W), somma(X, W, Z).`

- 

- Non utilizzabile in pratica: predicati predefiniti per la valutazione di espressioni

-

# PREDICATI PREDEFINITI PER LA VALUTAZIONE DI ESPRESSIONI

- L'insieme degli atomi Prolog contiene tanto i numeri interi quanto i numeri floating point. I principali operatori aritmetici sono simboli funzionali (operatori) predefiniti del linguaggio. In questo modo ogni espressione può essere rappresentata come un termine Prolog.
- 
- Per gli operatori aritmetici binari il Prolog consente tanto una notazione prefissa (funzionale), quanto la più tradizionale notazione infissa
- 

**TABELLA OPERATORI ARITMETICI**

|                  |                                                                                                                                |
|------------------|--------------------------------------------------------------------------------------------------------------------------------|
| Operatori Unari  | <code>-</code> , <code>exp</code> , <code>log</code> , <code>ln</code> , <code>sin</code> , <code>cos</code> , <code>tg</code> |
| Operatori Binari | <code>+</code> , <code>-</code> , <code>*</code> , <code>\</code> , <code>div</code> , <code>mod</code>                        |

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

# PREDICATI PREDEFINITI PER LA VALUTAZIONE DI ESPRESSIONI

- Data un'espressione, è necessario un meccanismo per la valutazione
- Speciale predicato predefinito **is**.

**T is Expr ( is (T, Expr) )**

- **T** può essere un atomo numerico o una variabile
- **Expr** deve essere un'espressione.
- 
- L'espressione **Expr** viene valutata e il risultato della valutazione viene
- unificato con **T**
- 
- Le variabili in **Expr** DEVONO ESSERE ISTANZIATE al momento della valutazione

# ESEMPI

---

`:- X is 2+3.`

`yes X=5`

`:- X1 is 2+3, X2 is exp(X1), X is X1*X2.`

`yes X1=5 X2=148.413 X=742.065`

`:- 0 is 3-3.`

`yes`

`: - X is Y-1.`

`No Y non è istanziata al momento della valutazione`

(NOTA: Alcuni sistemi Prolog danno come errore Instantion Fault)

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

`no`

# ESEMPI

```
:- X is 2+3, X is 4+1.
yes X=5
```

In questo caso il secondo goal della congiunzione risulta essere:

```
:- 5 is 4+1.
```

che ha successo. *x* infatti è stata istanziata dalla valutazione del primo *is* al valore 5.

```
:- X is 2+3, (X is X+1).
no
```



NOTA: non corrisponde a un assegnamento dei linguaggi imperativi. Le variabili sono *write-once*

# ESEMPI

---

- Nel caso dell'operatore `is` l'ordine dei goal è rilevante.
  - 
  - (a) `:- X is 2+3, Y is X+1.`
  - (b) `:- Y is X+1, X is 2+3.`
- 
- Mentre il goal (a) ha successo e produce la coppia di istanziazioni `X=5, Y=6`, il goal (b) fallisce.
- 
- Il predicato predefinito "is" è un tipico esempio di un predicato predefinito non reversibile; come conseguenza le procedure che fanno uso di tale predicato non sono (in generale) reversibili.

# TERMINI ED ESPRESSIONI

---

Un termine che rappresenta un'espressione viene valutato solo se è il secondo argomento del predicato `is`

```
p(a,2+3*5).
```

```
q(X,Y) :- p(a,Y), X is Y.
```

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

```
yes (X=17) (Y=2+3*5) (Y=+(2,* (3,5)))
```

  
Valutazione di Y

  
NOTA: `Y` non viene valutato, ma unifica con una struttura che ha `+` come operatore principale, e come argomenti 2 e una struttura che ha `*` come operatore principale e argomenti 3 e 5

# OPERATORI RELAZIONALI

- Il 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

$>, <, >=, =<, ==, \neq$   $\Rightarrow$  *disuguaglianza*



*uguaglianza*

- 
- 
-

# CONFRONTO TRA ESPRESSIONI

---

- Passi effettuati nella valutazione di:  
    **Expr1 REL Expr2**
- dove **REL** e' un operatore relazionale e **Expr1** e **Expr2** sono espressioni
- - vengono valutate **Expr1** ed **Expr2**
  - NOTA: le espressioni devono essere completamente istanziate
  - I risultati della valutazione delle due espressioni vengono confrontati tramite l'operatore **REL**

# ESEMPI

---

- Calcolare la funzione  $\text{abs}(x) = |x|$   
--- `abs(X,Y)` "Y è il valore assoluto di X"  
  
`abs(X,X) :- X >= 0.`  
`abs(X,Y) :- X < 0, Y is -X.`
- Si consideri la definizione delle seguenti relazioni:  
`pari(X) = true se X è un numero pari`  
`false se X è un numero dispari`  
`dispari(X) = true se X è un numero dispari`  
`false se X è un numero pari`  
`pari(0).`  
`pari(X) :- X > 0, X1 is X-1, dispari(X1).`  
`dispari(X) :- X > 0, X1 is X-1, pari(X1).`

# 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

$f : x_1, x_2, \dots, x_n \quad y$  diventa

$f(x_1, x_2, \dots, x_n, Y) \quad :- \quad \text{<calcolo di Y>}$

- Esempio: calcolare la funzione fattoriale così definita:

$\text{fatt} : n \quad n ! \quad (n \text{ intero positivo})$

$\text{fatt}(0) = 1$

$\text{fatt}(n) = n * \text{fatt}(n-1) \quad (\text{per } n > 0)$

$\text{fatt}(0, 1) .$

$\text{fatt}(N, Y) :- N > 0, N1 \text{ is } N-1, \text{fatt}(N1, Y1), Y \text{ is } N*Y1 .$

# CALCOLO DI FUNZIONI

---

- 
- Esempio: calcolare il massimo comun divisore tra due interi positivi

—

`mcd: x,y MCD(x,y) (x,y interi positivi)`

`MCD(x,0) = x`

`MCD(x,y) = MCD(y, x mod y) (per y>0)`

`mcd(X,Y,Z)`

`"Z è il massimo comun divisore di X e Y"`

`mcd(X,0,X) .`

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

—

—

# RICORSIONE E ITERAZIONE

---

- 
- Il Prolog non fornisce alcun costrutto sintattico per l'iterazione (quali, ad esempio, i costrutti *while* e *repeat*) e l'unico meccanismo per ottenere iterazione è la definizione ricorsiva.
- 
- 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 ulteriori operazioni
- 
- La definizione di funzioni (predicati) per ricorsione tail può essere considerata come una definizione per *iterazione*
  - Potrebbe essere valutata in spazio costante mediante un processo di valutazione iterativo.

# RICORSIONE E ITERAZIONE

---

- 
- Si dice *ottimizzazione della ricorsione tail* valutare una funzione tail ricorsiva  $f$  mediante un processo iterativo ossia caricando un solo record di attivazione per  $f$  sullo stack di valutazione (esecuzione).
- 
- In Prolog l'ottimizzazione della ricorsione tail è un po' più complicata che non nel caso dei linguaggi imperativi a causa del:
  - non determinismo
  - della presenza di punti di scelta nella definizione delle clausole.

# RICORSIONE E ITERAZIONE

---

- - $p(X) \text{ :- } c1(X), g(X).$
  - (a)  $p(X) \text{ :- } c2(X), h1(X,Y), p(Y).$
  - (b)  $p(X) \text{ :- } c3(X), h2(X,Y), p(Y).$
- 
- Due possibilità di valutazione ricorsiva del goal :  $\neg p(Z)$  .
  - se viene scelta la clausola (a), si deve ricordare che (b) è un punto di scelta ancora aperto. Bisogna mantenere alcune informazioni contenute nel record di attivazione di  $p(Z)$  (i punti di scelta ancora aperti)
  - se viene scelta la clausola (b) (più in generale, l'ultima clausola della procedura), non è più necessario mantenere alcuna informazione contenuta nel record di attivazione di  $p(Z)$  e la rimozione di tale record di attivazione può essere effettuata

## QUINDI...

---

- 
- In Prolog l'ottimizzazione della ricorsione tail è possibile solo se la scelta nella valutazione di un predicato "p" è deterministica o, meglio, se al momento del richiamo ricorsivo (n+1)-esimo di "p" non vi sono alternative aperte per il richiamo al passo n-esimo (ossia alternative che potrebbero essere considerate in fase di backtracking)
- 
- Quasi tutti gli interpreti Prolog effettuano l'ottimizzazione della ricorsione tail ed è pertanto conveniente usare il più possibile ricorsione di tipo tail.

# RICORSIONE NON TAIL

- Il predicato `fatt` è definito con una forma di ricorsione semplice (non tail).
- 
- Casi in cui una relazione ricorsiva può essere trasformata in una relazione tail ricorsiva
- 

```
fatt1(N,Y) :- fatt1(N,1,1,Y) .
```

```
fatt1(N,M,ACC,ACC) :- M > N.
```

```
fatt1(N,M,ACCin,ACCout) :- ACCtemp is ACCin*M,
 M1 is M+1,
 fatt1(N,M1,ACCtemp,Accout) .
```

*Accumulatore  
in ingresso*

*Accumulatore  
in uscita*

# RICORSIONE NON TAIL

---

- Il fattoriale viene calcolato utilizzando un argomento di accumulazione, inizializzato a 1, incrementato ad ogni passo e unificato in uscita nel caso base della ricorsione.

—

—  $ACC_0 = 1$

—  $ACC_1 = 1 * ACC_0 = 1 * 1$

—  $ACC_2 = 2 * ACC_1 = 2 * (1 * 1)$

— ...

—  $ACC_{N-1} = (N-1) * ACC_{N-2} = N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots))$

—  $ACC_N = N * ACC_{N-1} = N * (N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots)))$

# RICORSIONE NON TAIL

---

- Altra struttura iterativa per la realizzazione del fattoriale
- 

```
fatt2(N,Y)
```

```
 "Y è il fattoriale di N"
```

```
fatt2(N,Y) :- fatt2(N,1,Y).
```

```
fatt2(0,ACC,ACC).
```

```
fatt2(M,ACC,Y) :- ACC1 is M*ACC,
 M1 is M-1,
 fatt2(M1,ACC1,Y).
```

# RICORSIONE NON TAIL

---

- Calcolo del numero di Fibonacci: definizione
  - `fibonacci(0) = 0`
  - `fibonacci(1) = 1`
  - `fibonacci(N) =`
    - `fibonacci(N-1) + fibonacci(N-2)`      per  $N > 1$
- Programma Prolog
  - ```
fib(0,0).
fib(1,1).
fib(N,Y) :- N>1,
            N1 is N-1,
            fib(N1,Y1),
            N2 is N-2,
            fib(N2,Y2),
            Y is Y1+Y2.
```