

La programmazione in PROLOG

LINGUAGGIO PROLOG

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

INTERPRETE PROLOG: SWI PROLOG

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




PROGRAMMA PROLOG

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

PROGRAMMA PROLOG

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

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



FATTI

PROGRAMMA PROLOG

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

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

FATTI

Head **Body**

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

PROGRAMMA PROLOG

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

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

FATTI

Head **Body**

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

:- collega(marco,giulia). ➡ **GOAL**

ESEMPI

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

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

```
in_italia(roma) .  
in_italia(palermo) .  
in_francia(parigi) .
```

Possiamo porre delle domande (query) all'interprete.

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

ESEMPI

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

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

```
in_italia(roma).  
in_italia(palermo).  
in_francia(parigi).
```

Possiamo porre delle domande (query) all'interprete.

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

ESEMPI

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

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

```
in_italia(roma).  
in_italia(palermo).  
in_francia(parigi).
```

Possiamo porre delle domande (query) all'interprete.

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

ESEMPI

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

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

```
in_italia(roma).  
in_italia(palermo).  
in_francia(parigi).
```

Possiamo porre delle domande (query) all'interprete.

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

ESEMPI

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

```
% X è figlio di Y se Y è genitore di X  
figlio(X, Y) :- genitore(Y, X).
```

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

ESEMPI

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

```
% X è figlio di Y se Y è genitore di X  
figlio(X, Y) :- genitore(Y, X).
```

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

ESEMPI

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

```
% X è figlio di Y se Y è genitore di X  
figlio(X, Y) :- genitore(Y, X).
```

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

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

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

ESEMPI

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

```
% X è figlio di Y se Y è genitore di X  
figlio(X, Y) :- genitore(Y, X).
```

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

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

PROVA DI UN GOAL

- Un goal viene considerato “vero” se sono veri tutti i letterali che lo compongono, considerati **da sinistra a destra**

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

- Un goal formato da un singolo letterale (atomico) viene valutato confrontandolo con le teste delle clausole contenute nel programma
- Se esiste una sostituzione per cui il confronto ha successo
 - se la clausola scelta è un fatto, la prova termina con successo;
 - se la clausola scelta è una regola, si deve verificare se il suo Body è vero.
- Se non esiste una sostituzione possibile, il goal fallisce e l'interprete restituisce **false**.

PIÙ FORMALMENTE

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

(c11) $A.$

➡ **FATTO o ASSERTZIONE**

(c12) $A :- B_1, B_2, \dots, B_n.$

➡ **REGOLA**

(c13) $:- B_1, B_2, \dots, B_n.$

➡ **GOAL**

- A e B_i sono formule atomiche
- A : **testa** della clausola
- B_1, B_2, \dots, B_n : **body** della clausola
- Il simbolo “,” indica la congiunzione;
- il simbolo “:-” indica l’implicazione logica in cui A è il conseguente e B_1, B_2, \dots, B_n l’antecedente
- le relazioni tra entità, definite da fatti e regole, vengono anche dette predicati

PIÙ FORMALMENTE

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

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

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

PIÙ FORMALMENTE

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

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

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

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

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

ESEMPI

- COSTANTI: `a`, `pippo`, `aB`, `9`, `135`, `a92`
- VARIABILI: `x`, `x1`, `Pippo`, `_Pippo`, `_X`, `_`
 - la variabile `_` prende il nome di variabile anonima
- TERMINI COMPOSTI: `f(a)`, `f(g(1))`, `f(g(1), b(a), 27)`
- CLAUSOLE:
 - `q.`
 - `p :- q, r.`
 - `r(Z).`
 - `p(X) :- q(X, g(a)).`
- GOAL:
 - `:- q, r.`

ESEMPIO

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

```
:- mortal(e(socrate)) . -> true
```

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

INTERPRETAZIONE DICHIARATIVA

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

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

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

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

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

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

ESEMPIO

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

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

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

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

SOLUZIONI MULTIPLE E DISGIUNZIONE

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

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

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

INTERPRETAZIONE PROCEDURALE

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

- Una query del tipo:

$$:- p(t_1, t_2, \dots, t_n) .$$

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

$$:- \text{nonno}(\text{alberto}, \text{vincenzo}) .$$

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

INTERPRETAZIONE PROCEDURALE

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

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

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

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

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

- Tutte le variabili sono a **singolo assegnamento**. Il loro valore è unico durante la computazione e può cambiare solo quando si cerca una soluzione alternativa.

ESEMPIO

```
pratica_sport(mario, calcio).  
pratica_sport(giovanni, calcio).  
pratica_sport(alberto, calcio).  
pratica_sport(marco, basket).  
abita(mario, torino).  
abita(giovanni, genova).  
abita(alberto, genova).  
abita(marco, torino).  
  
:- pratica_sport(X, calcio).  
    "Esiste una persona X che pratica il calcio?"
```

ESEMPIO

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

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

ESEMPIO

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

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

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

ESEMPIO

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

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

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

ESEMPIO (2)

```
:- pratica_sport(X, Y).  
    "Esistono X e Y tali per cui X pratica lo sport Y?"
```

ESEMPIO (2)

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

ESEMPIO (2)

```
:- pratica_sport(X, Y).  
    "Esistono X e Y tali per cui X pratica lo sport Y?"  
true      X = mario      Y = calcio;  
          X = giovanni   Y = calcio;  
          X = alberto    Y = calcio;  
          X = marco      Y = basket;  
false  
  
:- pratica_sport(X, calcio), abita(X, genova).  
    "Esiste una persona X che pratica il calcio e abita a Genova?"
```

ESEMPIO (2)

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

false

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

false

ESERCIZIO

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

- A partire da queste relazioni, definire una relazione

amico (X,Y) “X è amico di Y”

con la seguente specifica:

“x è amico di y se x e y praticano lo stesso sport e abitano nella stessa città”.

SOLUZIONE

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

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

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

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

```
true    Y = giovanni;  
        Y = alberto;
```

```
false
```

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

STRATEGIA DI RICERCA

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

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

Cosa succede in questo caso?

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

Come viene scelta la regola da applicare per prima?

STRATEGIA DI RICERCA

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

DEBUGGER GRAFICO

- Per attivare il debugger grafico, usare il comando:

`:- gtrace.`

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

`:- nodebug.`

ESEMPIO

<code>genitore(a,b) .</code>	<code>(R1)</code>
<code>genitore(b,c) .</code>	<code>(R2)</code>
<code>antenato(X,Z) :- genitore(X,Z) .</code>	<code>(R3)</code>
<code>antenato(X,Z) :- genitore(X,Y) , antenato(Y,Z) .</code>	<code>(R4)</code>
<code>G0 :- antenato(a,c) .</code>	

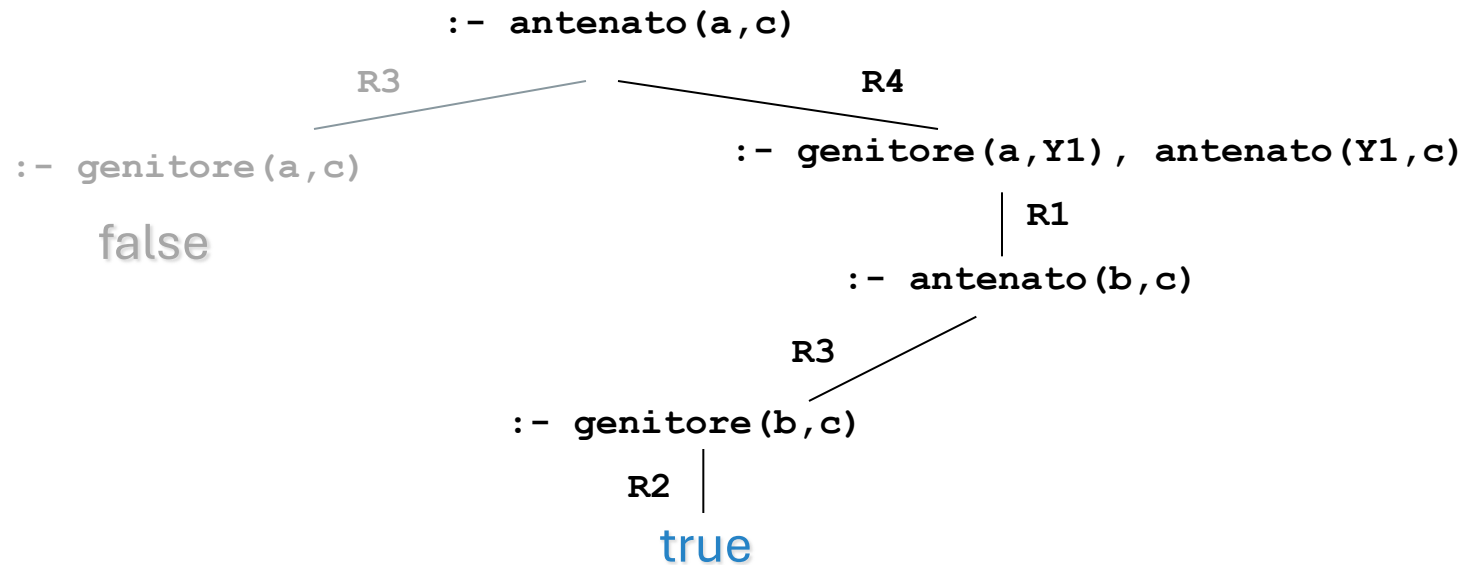
ESEMPIO

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

```
                :- antenato(a,c)
            R3
    /
:- genitore(a,c)
false
```

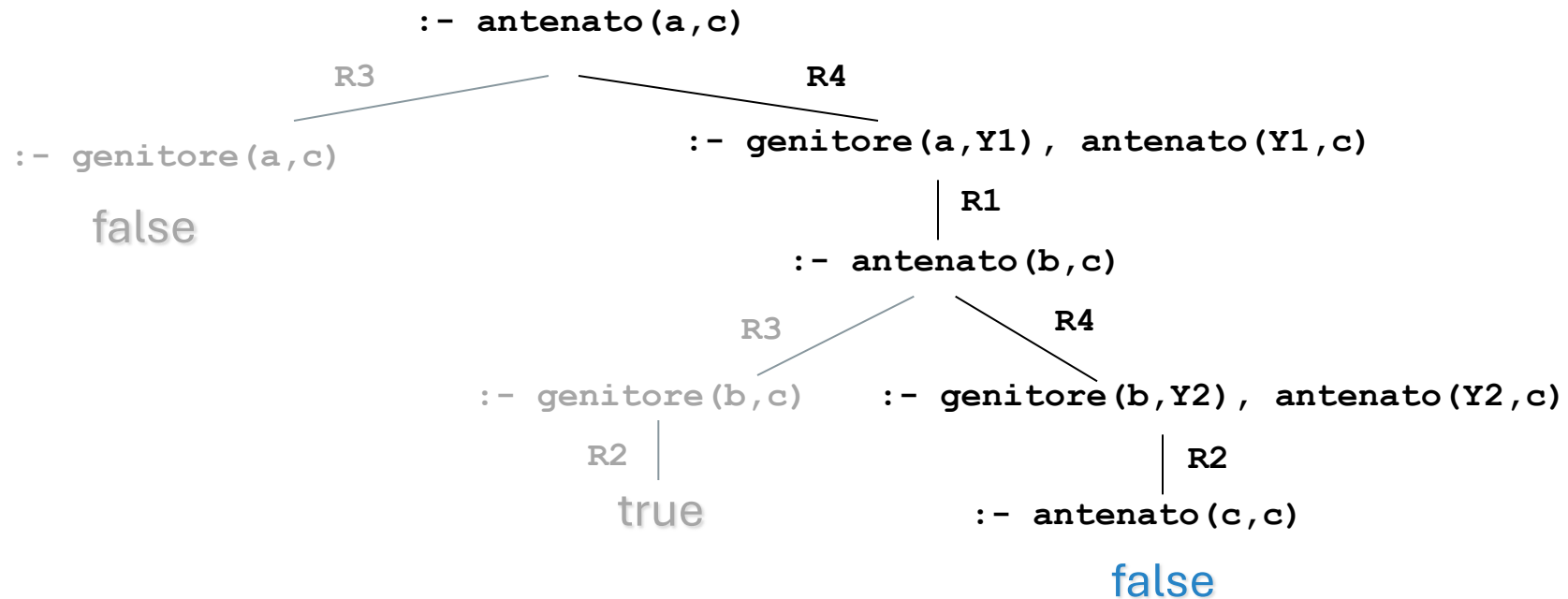
ESEMPIO

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



ESEMPIO

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



RISOLUZIONE IN PROLOG: ESEMPIO

P₁

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

:- p.

:- p (1)

RISOLUZIONE IN PROLOG: ESEMPIO

P₁

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

:- p.

		:- p (1)
(c11)		
:- q,r (2)		

RISOLUZIONE IN PROLOG: ESEMPIO

P₁

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

`:- p.`

```

              :- p (1)
            (c11) /
              :- q,r (2)
            (c13) |
              :- r (3)
```

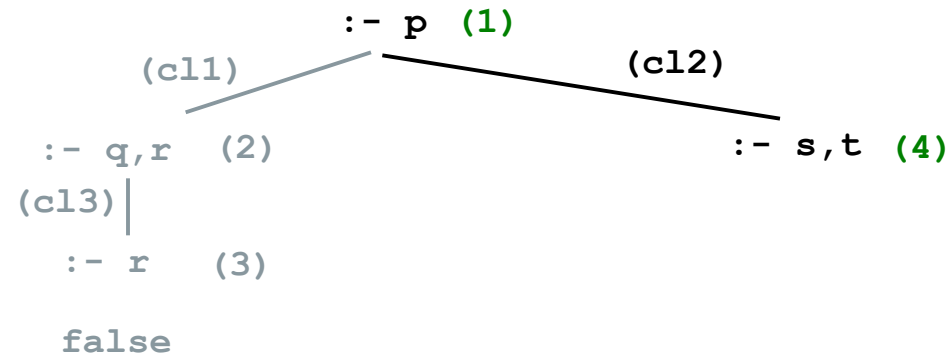
false

RISOLUZIONE IN PROLOG: ESEMPIO

P₁

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

:- p.

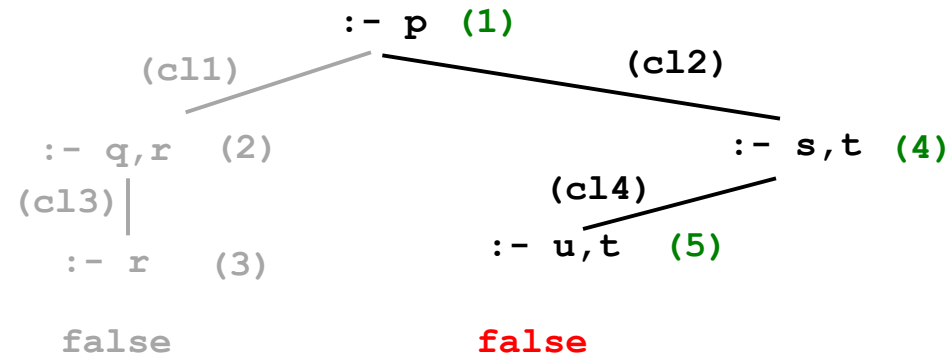


RISOLUZIONE IN PROLOG: ESEMPIO

P₁

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

:- p.

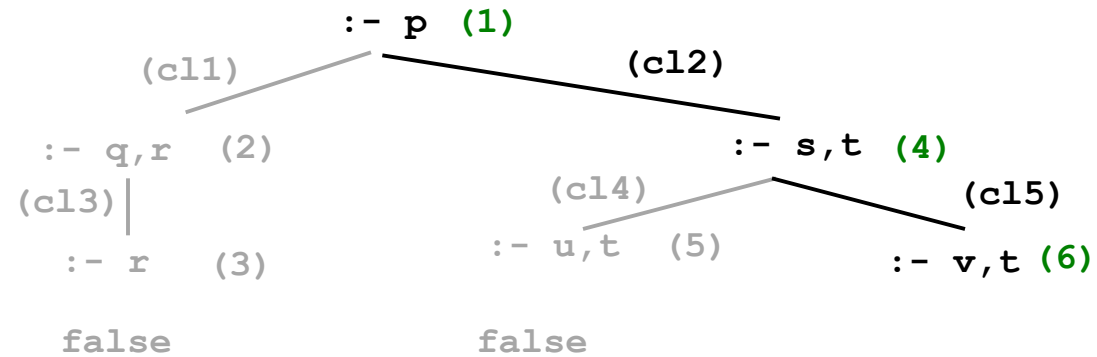


RISOLUZIONE IN PROLOG: ESEMPIO

P₁

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

:- p.

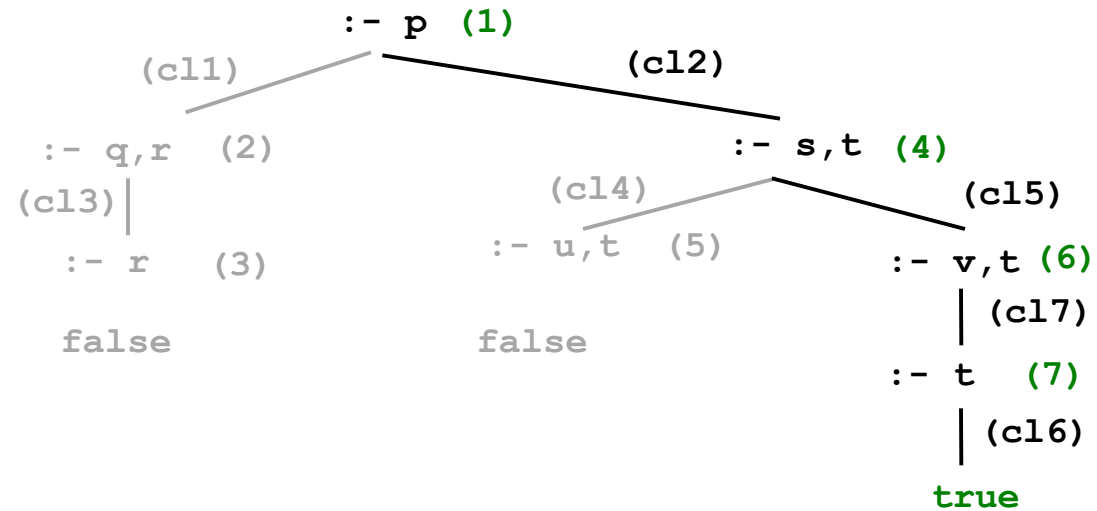


RISOLUZIONE IN PROLOG: ESEMPIO

P_1

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

:- p.



ESERCIZIO

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

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

$X \backslash== Y$

- A partire da queste relazioni, definire una relazione

sorella(X, Y) “ X è sorella di Y ”

con la seguente specifica:

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

SOLUZIONE

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

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

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

SOLUZIONE

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

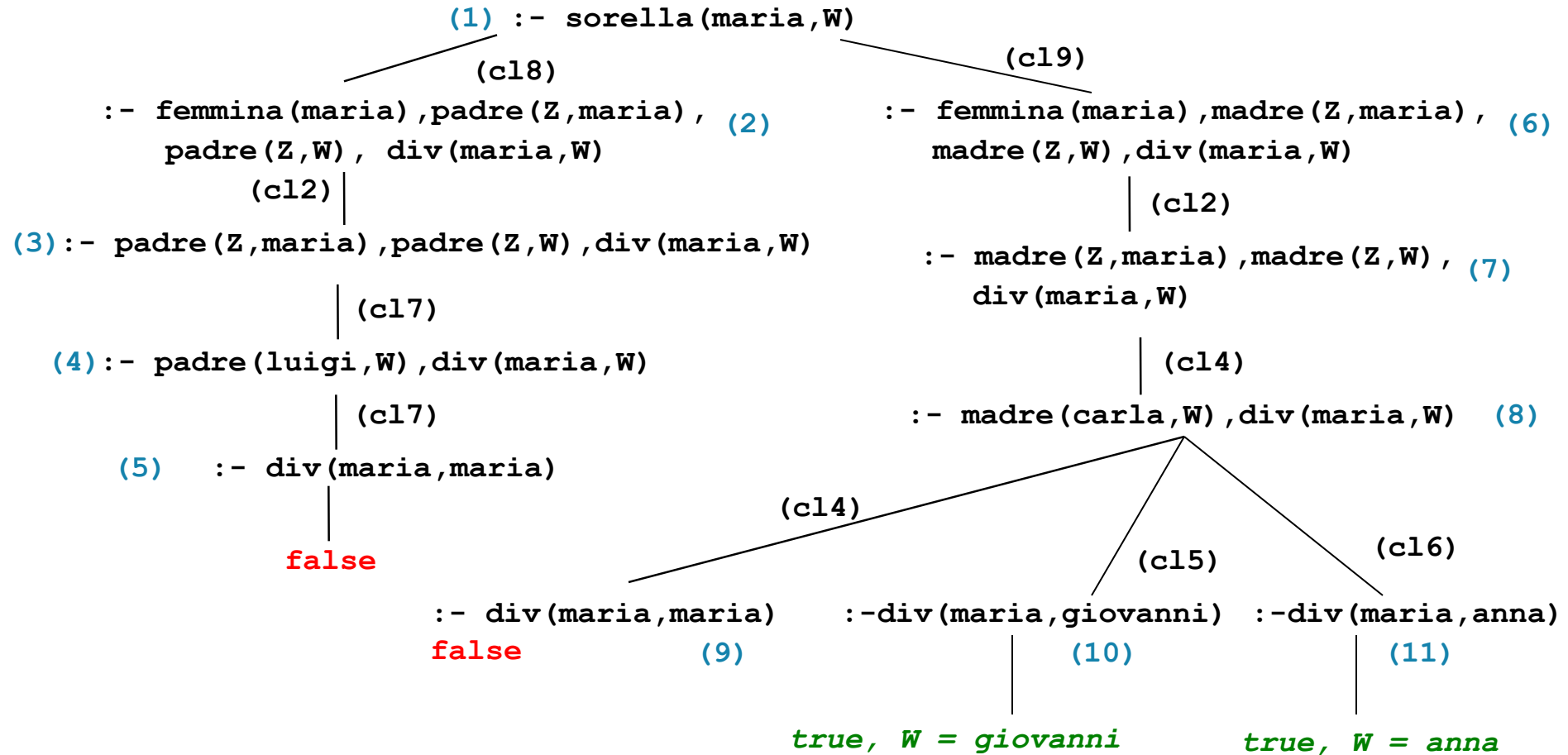
```
sorella(X,Y):-  
    femmina(X),  
    padre(Z,X),  
    padre(Z,Y),  
    X \== Y.
```

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

Esempio di esecuzione

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

ESEMPIO DI QUERY :- sorella(maria,W)



VERSO UN VERO LINGUAGGIO DI PROGRAMMAZIONE

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

ARITMETICA E RICORSIONE

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

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



N volte

PREDICATI PER LA VALUTAZIONE DI ESPRESSIONI

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

TABELLA OPERATORI ARITMETICI

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

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

PREDICATI PER LA VALUTAZIONE DI ESPRESSIONI

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

T is Expr

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

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

ESEMPI

?- X is 2 + 3.

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

?- 0 is 3 - 3.

?- X is Y - 1.

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

ESEMPI

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

```
true      X = 5
```

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

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

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

```
true
```

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

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

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

```
false
```

ESEMPI

?- 5 + 3 is 8.

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

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

ESEMPI

?- 5 + 3 is 8.

false

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

false

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

true X = 5

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

?- 5 is 4 + 1. (true)

ESEMPI

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

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

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

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

ESEMPI

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

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

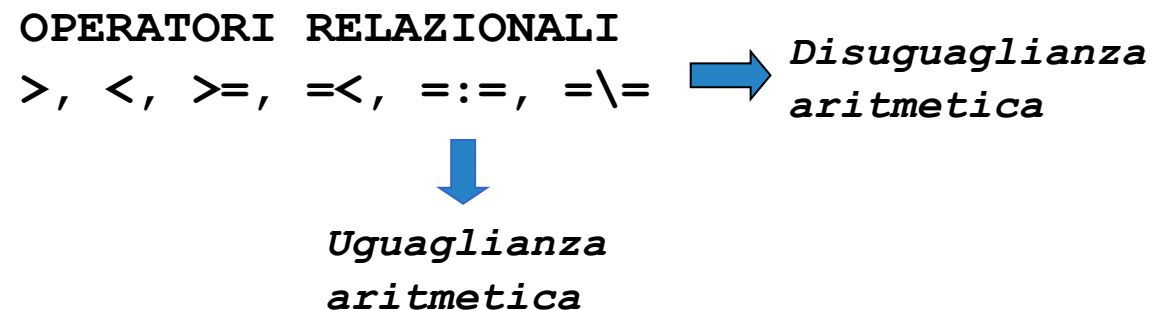
```
?- sum(3, 7, Sum).  
true      Sum = 10
```

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

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

OPERATORI RELAZIONALI

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



CONFRONTO TRA ESPRESSIONI

- Passi effettuati nella valutazione di

Expr1 REL Expr2

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

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

:- 3 + 2 < 20 / 2.

true

:- 5 + x == 8.

false

SIMBOLI DI UGUAGLIANZA E DISUGUAGLIANZA

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

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

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

ESERCIZIO

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

$$N + N = N * N$$

ESERCIZIO - SOLUZIONE

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

$$N + N = N * N$$

1) `special_number(N) :-`

`Somma is N + N,`

`Prod is N * N,`

`Somma == Prod.`

1) `special_number(N) :-`

`X is N + N,`

`X is N * N.`

1) `special_number(N) :-`

`N + N == N * N.`

CALCOLO DI FUNZIONI

- Una funzione può essere realizzata attraverso relazioni Prolog.
- Data una funzione f ad n argomenti, essa può essere realizzata mediante un predicato ad $n+1$ argomenti nel modo seguente
 - $f : x_1, x_2, \dots, x_n \rightarrow y$ diventa
 $f(x_1, x_2, \dots, x_n, Y) \text{ :- } \langle \text{calcolo di } Y \rangle$

ESEMPIO

- Funzione valore assoluto, **abs (x) = |x|**
- Pseudocodice:

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

ESEMPIO

- Funzione valore assoluto, **abs**(**x**) = |**x**|

```
% abs(X, Y) - "Y è il valore assoluto di X"
```

```
abs(X, Y) :- X >= 0, Y is X.
```

```
abs(X, Y) :- X < 0, Y is -X.
```

ESEMPIO

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

```
% abs(X, Y) - "Y è il valore assoluto di X"
```

```
abs(X, Y) :- X >= 0, Y is X.
```

```
abs(X, Y) :- X < 0, Y is -X.
```

- Alternativa più sintetica:

```
abs(X, X) :- X >= 0.
```

```
abs(X, Y) :- X < 0, Y is -X.
```

ESERCIZIO

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

```
% mcd(X, Y, Result)
```

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

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

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

SOLUZIONE

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

```
% mcd(X, Y, Result)
```

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

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

```
mcd(X, 0, X) .
```

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

```
Y > 0,
```

```
New_X is X mod Y,
```

```
mcd(Y, New_X, Result) .
```

RICORSIONE E ITERAZIONE

Pseudo-codice

iterativo

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

Pseudo-codice

ricorsivo

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

RICORSIONE E ITERAZIONE

Prolog

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

Pseudo-codice

ricorsivo

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

FATTORIALE DI UN NUMERO

Pseudo-codice iterativo

```
def fact(n):  
    result = 1  
  
    while n > 1:  
        result *= n  
        n -= 1  
  
    return result
```

Pseudo-codice ricorsivo

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

FATTORIALE DI UN NUMERO

Prolog

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

Pseudo-codice ricorsivo

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

ESERCIZIO - FIBONACCI

- Calcolare l'N-esimo numero della successione di Fibonacci.

```
% fib(N, Y)
```

```
% "Y è l'N-esimo numero di Fibonacci"
```

- Si ricorda che l'N-esimo numero di Fibonacci può essere calcolato come segue:
 - $\text{fib}(0) = 0$
 - $\text{fib}(1) = 1$
 - $\text{fib}(n) = \text{fib}(n - 2) + \text{fib}(n - 1) \quad \forall n > 1$

SOLUZIONE - FIBONACCI

- $\text{fib}(0) = 0$
- $\text{fib}(1) = 1$
- $\text{fib}(n) = \text{fib}(n - 2) + \text{fib}(n - 1) \quad \forall n > 1$

```
% fib(N, Y) - "Y è l'N-esimo numero di Fibonacci"
fib(0, 0).
fib(1, 1).
fib(N, Y) :-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, Fib1),
    fib(N2, Fib2),
    Y is Fib1 + Fib2.
```

RICORSIONE TAIL

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

ESEMPI

Ricorsione non tail

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

Ricorsione tail

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

RICORSIONE NON TAIL ➡ TAIL

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

$$\text{Acc}_0 = 1$$

$$\text{Acc}_1 = 1 * \text{Acc}_0 = 1 * 1$$

$$\text{Acc}_2 = 2 * \text{Acc}_1 = 2 * (1 * 1)$$

$$\text{Acc}_3 = 3 * \text{Acc}_2 = 3 * (2 * (1 * 1))$$

...

$$\text{Acc}_N = N * \text{Acc}_{N-1} = N * ((N-1) * ((N-2) * (\dots * (2 * (1 * 1)) \dots)))$$

RICORSIONE NON TAIL ➡ TAIL

Pseudo-codice iterativo

```
def fact(n):  
    acc = 1  
    counter = 1  
  
    while counter <= n:  
        acc *= counter  
        counter += 1  
  
    return acc
```

RICORSIONE NON TAIL ➡ TAIL

Pseudo-codice iterativo

```
def fact(n):  
    acc = 1  
    counter = 1  
  
    while counter <= n:  
        acc *= counter  
        counter += 1  
  
    return acc
```

Pseudo-codice ricorsivo

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

RICORSIONE NON TAIL ➡ TAIL

```
% Inizializziamo Counter e Acc a 1
```

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

```
    N >= 0,
```

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

```
% Se Counter > N, il risultato finale è Acc
```

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

```
    Counter > N.
```

```
% Se Counter <= N, aggiorniamo Acc, incrementiamo Counter ed effettuiamo la chiamata ricorsiva
```

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

```
    Counter <= N,
```

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

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

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

RICORSIONE NON TAIL ➡ TAIL

- Versione alternativa per il fattoriale con ricorsione tail:

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

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

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

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

```
    Counter > 0,
```

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

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

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