



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Prolog – Metainterpreti

Federico Chesani

DISI

Department of Informatics – Science and Engineering

Disclaimer & Further Reading

- These slides are largely based on previous work by Prof. Paola Mello



Predicati di Meta-Livello

- In Prolog non vi è alcuna differenza sintattica tra programmi e dati e che essi possono essere usati in modo intercambiabile.
- Questa caratteristica apre ad alcune possibilità:
 - la possibilità di accedere alle clausole che costituiscono un programma e trattare tali clausole come termini;
 - la possibilità di modificare dinamicamente un programma (il data-base);
 - **la meta-interpretazione.**



Predicati dinamici in Prolog

- In Prolog, fatti e regole possono essere **statici** se sono già stati definiti e non cambiano durante l'esecuzione o **dinamici** se possono essere modificati, aggiunti o cancellati.
- Per poter utilizzare predicati dinamici si deve utilizzare un predicato **dynamic**.
- La differenza sta meramente a livello dell'interprete Prolog, e di strutture dati (più o meno) efficienti per la risoluzione. Tuttavia l'uso di **dynamic** in alcuni interpreti è obbligatorio.

Esempio: si supponga di avere un insieme di fatti a due argomenti che descrivono i nomi e cognomi dei miei amici del tipo **amico(Nome, Cognome)** e che dinamicamente si voglia arricchire questa base di conoscenza, aggiungendo nuovi amici.

Dovrò quindi scrivere:

```
:- dynamic(amico/2) .
```



Predicati di meta-livello – Accesso alle clausole

- Una clausola (o una query) è rappresentata come un termine.
- Le seguenti clausole:

h.

h :- b1, b2, ..., bn.

e la loro forma equivalente:

h :- true.

h :- b1, b2, ..., bn.

corrispondono ai termini:

:- (h, true)

:- (h, ',' (b1, ',' (b2, ',' (... ',' (bn-1, bn) ...))))



Accesso alle clausole: clause

`clause (HEAD , BODY)`

- “vero se `:- (HEAD , BODY)` è (unificato con) una clausola all'interno del data base“
- Quando valutata:
 - `HEAD` deve essere istanziata ad un termine non numerico
 - `BODY` può essere o una variabile o un termine che denota il corpo di una clausola.
- Apre un punto di scelta per procedure non-deterministiche (più clausole con testa unificabile con `HEAD`)



Esempio clause (Head, Body)

```
?-dynamic(p/1).
```

```
?-dynamic(q/2).
```

```
p(1).
```

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

```
q(2,Y) :- d(Y).
```

```
?- clause(p(1),BODY).
```

```
yes    BODY=true
```

```
?- clause(p(X),true).
```

```
yes    X=1
```

```
?- clause(q(X,Y), BODY).
```

```
yes    X=_1      Y=a      BODY=p(_1),r(a);
```

```
        X=2      Y=_2      BODY=d(_2);
```

```
no
```

```
?- clause(HEAD,true).
```

```
Error - invalid key to data-base
```



Modifiche al database: assert

`assert(T)`

"la clausola **T** viene aggiunta al data-base"

- Alla valutazione, **T** deve essere istanziato ad un termine che denota una clausola (un atomo o una regola). **T** viene aggiunto nel data-base in una posizione **non specificata**.
- Ignorato in backtracking (non dichiarativo)
- Due varianti del predicato "assert":
 - `asserta(T)` : "la clausola T viene aggiunta all'inizio data-base"
 - `assertz(T)` : "la clausola T viene aggiunta al fondo del data-base"



Esempio assert

?- assert(a(2)).

?-dynamic(a/1).
a(1).
b(X):-a(X).

?- asserta(a(3)).

a(1).
a(2).
b(X):-a(X).

?- assertz(a(4)).

a(3).
a(1).
a(2).
b(X):-a(X).

a(3).
a(1).
a(2).
a(4).
b(X):-a(X).



Modifiche al database: `retract`

`retract(T)`

"la prima clausola nel data-base unificabile con `T` viene rimossa"

- Alla valutazione, `T` deve essere istanziato ad un termine che denota una clausola;
- se più clausole sono unificabili con `T` è rimossa la prima clausola (con punto di scelta a cui tornare in backtracking *in alcune versioni del Prolog*).
- Alcune versioni del Prolog forniscono un secondo predicato predefinito: il predicato "`abolish`" (o "`retract_all`", a seconda delle implementazioni):

`abolish(NAME, ARITY)`



Esempio retract

```
?- retract(a(X)) .  
yes X=3
```

```
?-dynamic(a/1) .  
?-dynamic(b/1) .  
a(3) .  
a(1) .  
a(2) .  
a(4) .  
b(X):-c(X),a(X) .
```

```
?- abolish(a,1) .
```

```
a(1) .  
a(2) .  
a(4) .  
b(X):- c(X),a(X) .
```

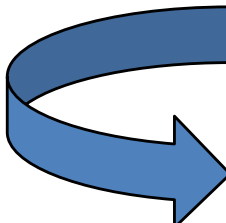
```
?- retract((b(X):-BODY)) .  
yes BODY=c(X),a(X)
```

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



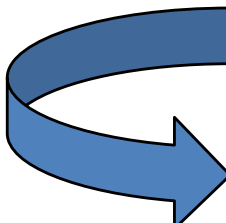
Esempio retract

```
?- retract(a(X)).  
yes X=3;
```



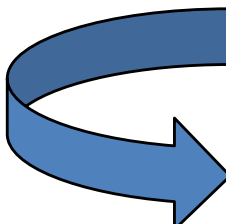
```
a(3).  
a(1).  
a(2).  
a(4).  
b(X):-c(X),a(X).
```

```
yes X=1;
```



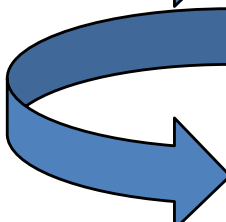
```
a(1).  
a(2).  
a(4).  
b(X):-c(X),a(X).
```

```
yes X=2;
```



```
a(2).  
a(4).  
b(X):-c(X),a(X).
```

```
yes X=4;
```



```
a(4).  
b(X):-c(X),a(X).
```

```
no
```

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



Problemi legati ad **assert** e **retract**

Usando **assert** e **retract** si perde la semantica dichiarativa dei programmi Prolog.

- Si considerino le seguenti query, in un database vuoto.:
?- **assert**(**p(a)**) , **p(a)** .
?- **p(a)** , **assert**(**p(a)**) .
- La prima valutazione ha successo, la seconda genera un fallimento.
- L'ordine dei letterali è rilevante nel caso in cui uno dei due letterali sia il predicato predefinito **assert**.



Problemi legati ad assert e retract

Un altro esempio è dato dai due programmi:

```
% (P1) :  
a(1) .  
p(X) :- assert(b(X)), a(X) .  
  
% (P2) :  
a(1) .  
p(X) :- a(X), assert(b(X)) .
```

- La valutazione della query `:- p(X) .` produce la stessa risposta, ma due modifiche differenti del data-base:
 - in P1 viene aggiunto `b(X) .` nel database, ossia $\forall X \text{ } p(X)$
 - in P2 viene aggiunto `b(1) .`



Problemi legati ad assert e retract

- Un ulteriore problema riguarda la quantificazione delle variabili.
 - Le variabili in una clausola nel data-base sono quantificate universalmente mentre le variabili in una query sono quantificate esistenzialmente.
- Si consideri la query: $:- \text{assert} (p(X)) .$
- Sebbene x sia quantificata esistenzialmente, l'effetto della valutazione della query è l'aggiunta al data-base della clausola $p(X) .$
ossia della formula $\forall x \ p(x)$



Esempio: generazione di Lemmi

- Il calcolo dei numeri di Fibonacci risulta estremamente inefficiente.

`fib(N,Y)` "Y è il numero di Fibonacci N-esimo"

```
fib(0,0) :- !.
```

```
fib(1,1) :- !.
```

```
fib(N,Y) :-      N1 is N-1, fib(N1,Y1),  
                  N2 is N-2, fib(N2,Y2),  
                  Y is Y1+Y2,  
                  genera_lemma(fib(N,Y))
```



Esempio: generazione di Lemmi

```
genera_lemma (T) :- asserta(T) .
```

- Oppure:

```
genera_lemma (T) :- clause(T,true) , !.
```

```
genera_lemma (T) :- asserta(T) .
```

- In questo secondo modo, la stessa soluzione (lo stesso fatto/lemma) non è asserita più volte all'interno del database.



Metainterpreti

I metainterpreti sono un caso specifico di meta-programmi, ossia di programmi che operano su altri programmi.

- Molto usati nella rapida prototipazione di interpreti per linguaggi simbolici (meta-interpreti)
- In Prolog, un meta-interprete per un linguaggio L è, per definizione, un **interprete** per L scritto nel linguaggio **Prolog**.
- E' possibile costruire in Prolog un interprete per il Prolog?



Metainterprete per il Prolog puro

Metainterprete "vanilla"

Realizzare un programma Prolog tale che:

`solve(GOAL)`

“il goal `GOAL` è deducibile dal programma Prolog puro definito da `clause` (ossia contenuto nel data-base)”

```
solve(true)    :- !.  
solve(A,B)    :- !, solve(A), solve(B).  
solve(A)      :- clause(A,B), solve(B).
```

- Nota: può facilmente essere esteso per trattare i predicati predefiniti del Prolog (almeno alcuni di essi). E' necessario aggiungere una clausola speciale per ognuno di essi prima dell'ultima clausola per "solve".



Esercizio: interprete con regola right-most

Realizzare un interprete Prolog che, invece della regola di selezione left-most, implementi la regola right-most.

Soluzione: il meta-interprete per Prolog puro può essere modificato per adottare una regola di calcolo diversa:

```
solve(true)      :- !.
solve( (A,B) )   :- !, solve(B), solve(A) .
solve(A)         :- clause(A,B), solve(B) .
```



Esercizio

- In un linguaggio simbolico Prolog-like la base di conoscenza è costituita da fatti e regole del tipo:

rule (Testa , Body) .

- Si scriva un metainterprete **solve (Goal , Step)** per tale linguaggio, in grado verificare se **Goal** è dimostrato e, in questo caso, in grado di calcolare in quanti passi di risoluzione (**Step**) tale goal viene dimostrato.
- Per le congiunzioni, il numero di passi è dato dalla somma del numero di passi necessari per ciascun singolo congiunto atomico.



Esercizio

- Per esempio, per il programma:

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

```
rule(b,d) .
```

```
rule(c,true) .
```

```
rule(d,true) .
```

```
a :- b,c.
```

```
b :- d.
```

```
c.
```

```
d.
```

Il metainterprete deve dare la seguente risposta:

```
?- solve(a,Step) .
```

```
yes    Step=4
```

- poiché a è dimostrato applicando 1 regola (1 passo) e la congiunzione (b,c) è dimostrata in 3 passi (2 per b e 1 per c).
- Non si vari la regola di calcolo e la strategia di ricerca di Prolog.



Esercizio – Soluzione

`solve(true,0) :-!.`

`solve((A,B),S) :- !,solve(A,SA),
solve(B,SB),
S is SA+SB.`

`solve(A,S) :- rule(A,B),
solve(B,SB),
S is 1+SB.`



Prolog e Sistemi Esperti

Vantaggi:

- possibilità di sfruttare a fondo le caratteristiche del Prolog;
- efficienza;
- facilità di realizzazione.

Svantaggi:

- approccio limitato all'uso di regole di produzione con strategia di inferenza backward;
- scarsa leggibilità e modificabilità dei programmi.



Esercizio – conoscenza e fattore di certezza

In un linguaggio simbolico Prolog-like la base di conoscenza è costituita da fatti e regole del tipo:

rule (Testa , Body , CF) .

dove **CF** rappresenta il fattore di certezza della regola (quanto è vera in termini probabilistici, espressa come intero percentuale – tra 0 e 100).

Esempio:

rule (a , (b , c) , 10) .

rule (b , true , 100) .

rule (c , true , 50) .

a :- b , c .

b .

c .



Esercizio – conoscenza e fattore di certezza

- Si scriva un metainterprete **solve (Goal, CF)** per tale linguaggio, in grado verificare se **Goal** è dimostrato e con quale probabilità.
- Per le congiunzioni, la probabilità sia calcolata come il minimo delle probabilità con cui sono dimostrati i singoli congiunti.
- Per le regole, è il prodotto della probabilità con cui è dimostrato il corpo per il CF della regola, diviso 100.



Esercizio – conoscenza e fattore di certezza

```
rule(a, (b,c), 10) .  
rule(a, d, 90) .  
rule(b,true, 100) .  
rule(c,true, 50) .  
rule(d,true, 100) .
```

```
a :- b,c.  
a :- d.  
b.  
c.  
d.
```

```
?-solve(a,CF) .  
yes CF=5;  
yes CF=90
```



Esercizio – conoscenza e fattore di certezza – Soluzione

```
solve(true,100) :- !.  
solve((A,B),CF) :- !,solve(A,CFA),  
                        solve(B,CFB),  
                        min(CFA,CFB,CF).  
  
solve(A,CFA) :- rule(A,B,CF),  
                solve(B,CFB),  
                CFA is ((CFB*CF)/100).  
  
min(A,B,A) :- A<B,!.  
min(A,B,B).
```



Un esempio di semplice Expert System shell (da SWI-Prolog manuale) (1)

```
% A meta-interpreter implementing a tiny expert-system
% -----
prove(true) :- !.
prove((B, Bs)) :- !,
    prove(B),
    prove(Bs).
prove(H) :-
    clause(H, B),
    prove(B).
prove(H) :-
    askable(H),
    writeln(H),
    read(Answer),
    Answer == yes.
```



Un esempio di semplice Expert System shell (da SWI-Prolog) (2)

```
good_pet(X) :- bird(X), small(X).  
good_pet(X) :- cuddly(X), yellow(X).
```

```
bird(X) :- has_feathers(X), tweets(X).
```

```
yellow(tweety).
```

```
askable(tweets(_)).
```

```
askable(small(_)).
```

```
askable(cuddly(_)).
```

```
askable(has_feathers(_)).
```

```
?- prove(good_pet(tweety)).
```

?- prove(good_pet(tweety)).

has_feathers(tweety)

yes

tweets(tweety)

yes

small(tweety)

yes

true



Esercizio – tracciamento dei goal

Si scriva un meta-interprete Prolog che stampi a video, prima e dopo l'invocazione di ogni sottogoal, tale sotto-goal. Per stampare a video si può usare il predicato built-in **write/1** che stampa l'argomento a video, e **nl/0** che stampa un **newline**.

Esempio:

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

```
q(1) .
```

```
q(2) .
```

```
?- solve(p(X)) .
```

```
yes    X/1
```

```
'Solving: 'p(X_e0)
```

```
'Selected Rule: 'p(X_e0) ':-'q(X_e0)
```

```
'Solving: 'q(X_e0)
```

```
'Selected Rule: 'q(1) ':-'true
```

```
'Solved: 'true
```

```
'Solved: 'q(1)
```



Esercizio – tracciamento dei goal – Soluzione

Il punto di partenza (quasi sempre) è il meta-interprete vanilla:

```
solve(true) :- ! .
```

```
solve( (A,B) ) :- ! , solve(B) , solve(A) .
```

```
solve(A) :- clause(A,B) , solve(B) .
```

La soluzione si può ottenere modificando l'interprete sopra:

```
solve(true) :- ! .
```

```
solve( (A,B) ) :- ! , solve(A) , solve(B) .
```

```
solve(A) :-
```

```
    write('Solving: '), write(A), nl,
```

```
    clause(A,B) ,
```

```
    write('Selected Rule: '), write(A) , write(":-"), write(B) , nl,
```

```
    solve(B) ,
```

```
    write('Solved: '), write(B) , nl.
```



Esercizio – ragionamento forward

- Realizzare un interprete che utilizzi una strategia di inferenza **forward**.
- Consideriamo, in primo luogo, il caso di un interprete di base per regole rappresentate come asserzioni Prolog del tipo:

rule (CONSEG, ANTEC)

- Un interprete forward per le regole è definito dal seguente programma:

```
interpreta:- <verifica se si è raggiunto un obiettivo>
interpreta:-      rule(CONSEG, ANTEC) ,
                  verifica_antec(ANTEC) ,
                  not(CONSEG) ,
                  assert(CONSEG) ,
                  interpreta.

interpreta  :- <riporta fallimento della dimostrazione>.
verifica_antec(ANTEC) "la congiunzione ANTEC è soddisfatta"
verifica_antec((A,B)):- !, call(A), verifica_antec(B).
verifica_antec(A)  :- call(A).
```



Ragionamento forward – regole applicabili

- Affinché una regola sia applicabile devono essere soddisfatte due condizioni:
 - L'antecedente della regola deve essere soddisfatto.
 - Il conseguente della regola non deve essere già vero. Ciò permette di evitare che una regola venga applicata più volte sugli stessi dati e che l'interprete vada in ciclo.
- L'interprete non ha una vera e propria fase di risoluzione di conflitti: viene semplicemente attivata la prima regola applicabile.
- Non è difficile realizzare un meta-interprete in cui le fasi di MATCH e CONFLICT-RESOLUTION sono separate.
- Supponiamo che le regole siano rappresentate da asserzioni del tipo:

`regola(NOME, CONSEG, ANTEC)`

in cui **NOME** è un nome che identifica univocamente ogni regola.



INTERPRETE FORWARD: risoluzione di conflitti esplicita

```
interpretal :- <verifica se si è raggiunto un obiettivo>
```

```
interpretal :-
```

```
    match(REG_APPLICABILI),  
    conflict_res(REG_APPLICABILI,REGOLA),  
    applica(REGOLA),  
    interpretal.
```

```
interpretal :- <riporta fallimento della dimostrazione>.
```

```
match(REG_APPLICABILI)
```

```
"REG_APPLICABILI: l'insieme di regole applicabili dato il  
    contenuto della memoria di lavoro (data base)"
```

```
match(REG_APPLICABILI) :-
```

```
    setof([REG,CONSEG],applic(REG,CONSEG),REG_APPLICABILI).
```



INTERPRETE FORWARD: risoluzione di conflitti esplicita (cont.)

```
% applic (REG,CONSEG)
```

```
% "la regola REG con conseguente CONSEG è applicabile"
```

```
applic (REG,CONSEG) :-
```

```
    regola (REG, CONSEG,ANTEC) ,
```

```
    verifica_antec (ANTEC) ,
```

```
    not (CONSEG) .
```

```
% conflict_res (REG_APPLIC, [REG,CONSEG])
```

```
% "la regola REG con conseguente CONSEG è la regola selezionata all'interno  
della lista REG_APPLIC di regole"
```

```
conflict_res (REG_APPLIC, [REG,CONSEG]) :-
```

```
    <selezione della regola da applicare>.
```

```
% applica ([REG,CONSEG])
```

```
% "applicazione regola REG con conseguente CONSEG"
```

```
applica ([REG,CONSEG]) :- assert (CONSEG) .
```



Sistemi esperti come meta-interpreti

- **Vantaggi**

- flessibilità;
- facilità di realizzazione;
- leggibilità e modificabilità (almeno per i meta-interpreti semplici);
- possibilità di definire meta-interpreti per diversi linguaggi di rappresentazione della conoscenza e diverse strategie di controllo.

- **Svantaggi**

- difficili da mantenere se il linguaggio di rappresentazione e le strategie di controllo diventano complesse;
- elevata inefficienza dovuta alla sovrapposizione di uno o più livelli di interpretazione al di sopra di quello del Prolog;
- Per ovviare al problema di inefficienza è stato proposto di utilizzare tecniche di valutazione parziale (trasformazione del programma in una versione ottimizzata più specializzata).

