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.

- Vedremo:

  - 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.

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"

ESEMPI assert

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

```
?- assert(a(2)).
```

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

```
?- asserta(a(3)).
```

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

```
?- assertz(a(4)).
```

```
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):

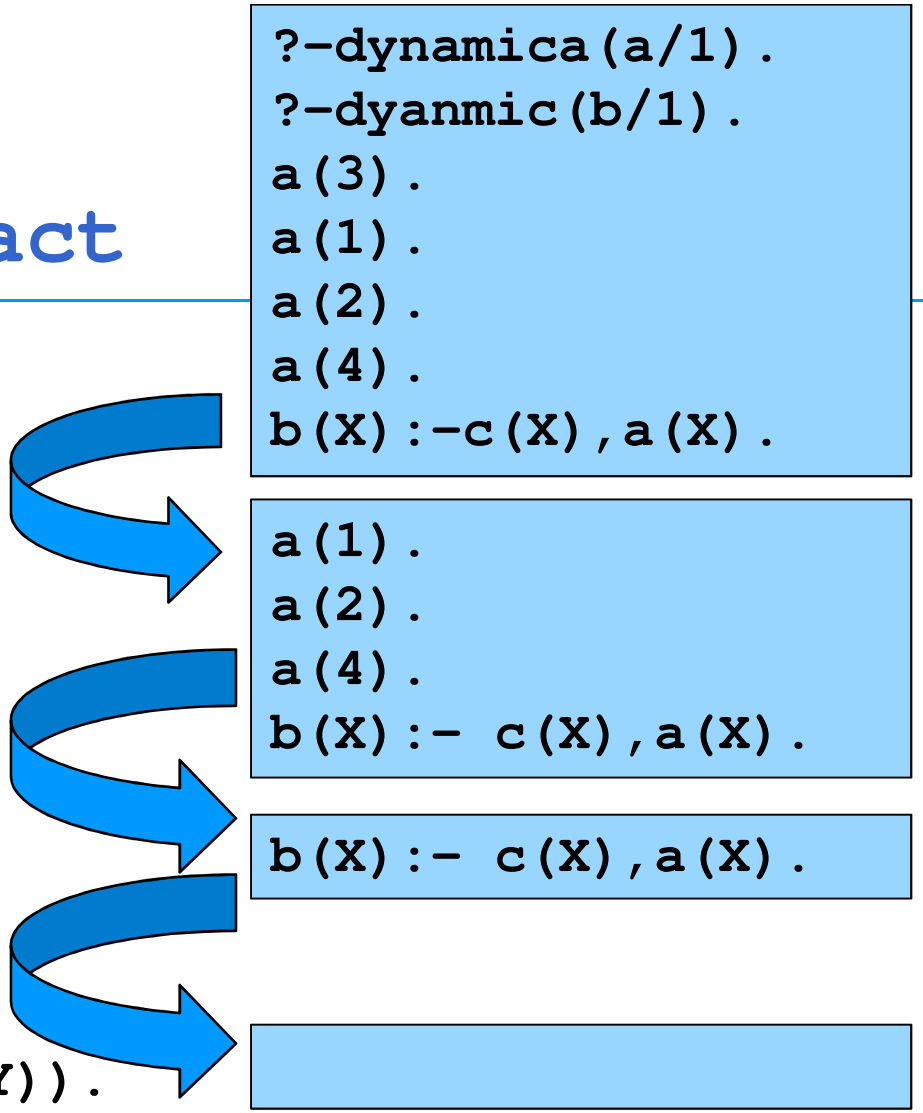
  `abolish(NAME,ARITY)`

ESEMPI retract

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

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

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



ESEMPI retract

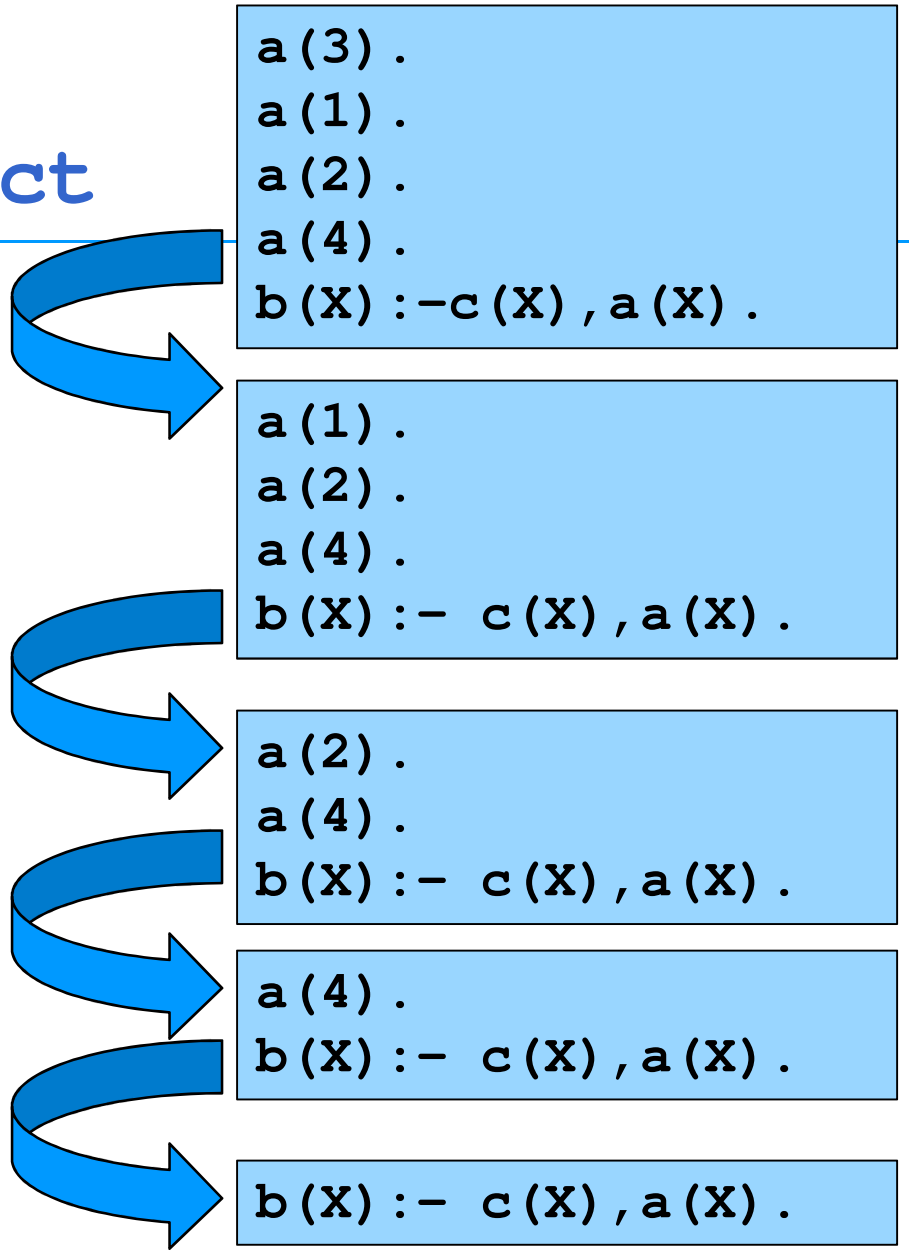
```
?-  retract(a(X)).
yes X=3;

yes X=1;

yes X=2;

yes X=4;

no
```



Problemi di **`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 di `assert` e `retract`

- Un altro esempio è dato dai due programmi:

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

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

- La valutazione della query `:- p(X).` produce la stessa risposta, ma due modifiche differenti del database:
  - in P1 viene aggiunto `b(X).` nel database, ossia ∀`X p(X)`
  - in P2 viene aggiunto `b(1).`

Problemi di `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: `:- 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)).
```

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

- Realizzazione di meta-programmi, ossia di programmi che operano su altri programmi.
- 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.
- Discuteremo come possa essere realizzato un semplice meta-interprete per il Prolog (in Prolog).

METAINTERPRETE PER PROLOG PURO

`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).
```

- 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".

PROLOG MA CON REGOLA DI CALCOLO RIGHT-MOST

- Il meta-interprete per Prolog puro può essere modificato per adottare una regola di calcolo diversa (ad esempio right-most):

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

ESERCIZIO 7.3: METAINTERPRETE

- 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 7.3 METAINTERPRETE

- Per esempio, per il programma:

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

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.

SOLUZIONE ESERCIZIO 7.3 METAINT.

```
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.
```

ESERCIZIO 7.4: FATTORI 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).
- **`rule(a, (b,c), 10).`**
- **`rule(b,true, 100).`**
- **`rule(c,true, 50).`**

ESERCIZIO 7.4: METAINTERPRETE

- 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 7.4: Esempio

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

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

SOLUZIONE ESERCIZIO 7.4 METAINT.

```
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).
```

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.*

Meta-Interprete forward

- *Meta-interprete che utilizza 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).
```

REGOLE APPLICABILI

- *Affinché una regola sia applicabile devono essere soddisfatte due condizioni:*
  - *L'antecedente della regola deve essere soddisfatto.*
  - *l 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

```
interpreta1  :- <verifica se si è raggiunto un obiettivo>
interpreta1  :-
        match(REG_APPLICABILI),
        conflict_res(REG_APPLICABILI,REGOLA),
        applica(REGOLA),
        interpreta1.
interpreta1 :- <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

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).
```

COSTRUZIONE DI SISTEMI ESPERTI CON META-INTERPRETAZIONE

- ***Vantaggi***
  - *elevata flessibilità;*
  - *facilità di realizzazione dei meta-interpreti;*
  - *leggibilità e modificabilità (almeno per i meta-interpreti semplici);*
  - *portabilità;*
  - *possibilità di definire meta-interpreti per diversi linguaggi di rappresentazione della conoscenza e diverse strategie di controllo.*
- ***Svantaggi***
  - *i meta-interpreti possono diventare difficili da mantenere se il linguaggio di rappresentazione e le strategie di controllo diventano molto 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****.*