TERMINI, OPERATORI e METALIVELLO

- In Prolog tutto è un termine: variabile, costante (numerica o atomica), struttura (lista come caso particolare con funtore ".")

- Ciascun termine struttura ha un funtore e argomenti che sono termini

- L'espressione 2+3 è un termine: +(2,3)
- Ma anche ogni clausola, ad esempio:
- member(X,[X|_]):-!. è un termine

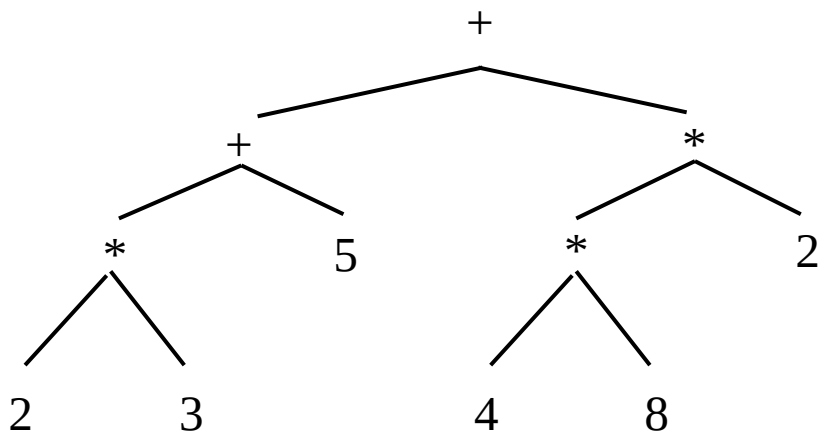
OPERATORI

- In Prolog è possibile definire "operatori" ed assegnare agli operatori regole di associatività e precedenza.

- Quando ci troviamo ad analizzare un'espressione del tipo:

  2*3+5+4*8*2

  siamo in grado di interpretare univocamente tale stringa



Associatività

- Si consideri l'espressione: 5-2-2, ci sono due possibili interpretazioni:
  - (a) (5-2)-2
  - (b) 5-(2-2)

- Per risolvere tale ambiguità è necessario specificare la regola di "associatività" dell'operatore.
-
- Nel caso degli operatori aritmetici "+", "*", "-" e "/" si assume per convenzione che gli operatori siano associativi a sinistra, ossia si privilegia la lettura (a) dell'espressione.

OPERATORE, CARATTERIZZATO DA:

- **nome**;
- **numero di argomenti**;
- **priorità** (o precedenza rispetto agli altri operatori);
- **associatività**; in particolare, un operatore può essere non associativo (come ad esempio l'operatore "=") o associativo; nel secondo caso l'operatore può essere associativo a destra o a sinistra.

DEFINIZIONEdi un OPERATORE

```
?- op(PRIORITA, TIPO, NOME)
```

- **NOME**, atomo alfanumerico con primo carattere alfabetico oppure una lista di tali atomi; nel secondo caso tutti gli operatori della lista vengono definiti con lo stesso tipo e priorità;
- **PRIORITA** è un numero (generalmente tra 0 e 1200) e specifica la priorità dell'operatore;
- **TIPO**, indica numero degli argomenti e associatività dell'operatore:
  - **fx, fy** (per operatori unari prefissi)
  - **xf, yf** (per operatori unari postfissi)
  - **xfx, yfx, xfy** (per operatori binari)

TIPO di un OPERATORE (BINARIO)

- **`xfx`** operatore non associativo
- **`yfx`** operatore associativo a sinistra

per cui, se “newop" ha tipo "yfx", l'espressione

E1 newop E2 newop E3 ... newop En

viene interpretata come

(...(E1 newop E2) newop E3) ....) newop En)

ossia come il termine

newop(newop(...newop(E1,E2), E3,...), En)

- **`xfy`** operatore associativo a destra

E1 newop (E2 newop (... newop (En-1 newop En)..))

newop(E1, newop(E2,newop(... newop (En-1,En)..))

OPERATORI PREDEFINITI

```
?-  op(1200,   xfx,   [ :-, ->]).
?-  op(1200,     fx,  [ :-, ?-]).
?-  op(1100,   xfy,   [ ;]).
?-  op(1050,   xfy,   [->]).
?-  op(1000,   xfy,   [',']).
?-  op(  900,    fy,  [not, spy, nospy]).
?-  op(  700,   xfx, [=, is, =.., ==, \==, @<, @>,
                    @=<, @>=, =:=, =\=, <, >, =<, >=]).
?-  op(  500,   yfx,  [+, -]).
?-  op(  500,    fx, [+, -]).
?-  op(  400,   yfx,  [*,  /, //]).
?-  op(  300,   xfx,  [mod]).
```

TERMINI e CLAUSOLE

- Anche i connettivi logici ",", ";" e ":-" sono definiti come operatori Prolog. In Prolog non esiste, infatti, alcuna distinzione tra dati e programmi per cui anche le congiunzioni e le clausole sono termini.

- member(X,[X|_]):-!. :-(member(X,[X|_]), !).

- p(X,Y):- q(X),r(Y). :-(p(X,Y), (q(X),r(Y)).

ESERCIZIO 7.1

- Introdurre un costrutto di iterazione di tipo "while":

  `while C do S`

- con il seguente significato informale: "fin tanto che la condizione **`C`** è vera, invoca la procedura **`S`** "
- Supponiamo che S possa essere una qualunque condizione, anche un ulteriore costrutto di iterazione; si considera quindi come legale una espressione del tipo:

  `while C1 do while C2 do S`

  imponendo l'interpretazione

  `while C1 do (while C2 do S)`

- Gli operatori **`while`** e **`do`** possono essere definiti come operatori Prolog.

SOLUZIONE ES. 7.1

- **`while`** operatore unario prefisso a priorità minore della priorità di **`do`** definito come un operatore binario associativo a destra.

```
?-  op(200, fy, while).
?-  op(300, xfy, do).
```

- Si ha allora che una espressione del tipo

  while c1 do while c2 do s

  viene interpretata secondo la seguente struttura di parentesi (while c1) do ((while c2) do s)

  ossia come il termine

  do((while c1), do((while c2), s))

SOLUZIONE ES. 7.1

- Insieme di regole Prolog per tali operatori:

```
while C do S  :-  C,
!,
S,
while C do S.
while C do S.
```

Verifica del “tipo” di un termine

- Determinare, dato un termine T, se T è un atomo, una variabile o una struttura composta.
  - atom(T) "T è un atomo non numerico"
  - number(T) "T è un numero (intero o reale)"
  - integer(T) "T è un numero intero"
  - atomic(T) "T è un'atomo oppure un numero (ossia T non è una struttura composta)"
  - var(T) "T è una variabile non istanziata"
  - nonvar(T) "T non è una variabile“
  - compound(T) “T e’ un termine composto”
- E’ anche possibile accedere alle componenti di un termine ...

Accesso alle componenti di un termine

`functor(TERM, FUNCTOR, ARITY)`

- Determina il funtore principale **`FUNCTOR`** e il numero di argomenti **`ARITY`** di un termine **`TERM`**

```
?-  functor(f(a,b,c),f,3).
yes
?-  functor(f(a,b,g(X)),F,A).
yesF=f  A=3
?-  functor(T,f,2).
yesT=f(_1,_2)
?-  functor(a,F,A).
yesF=a  A=0
```

Usi diversi possibili in base a quali argomenti sono istanziati e quali variabili

Accesso alle componenti di un termine

`arg(POS, TERM, ARG)`

- Determina (unifica) l'argomento **`ARG`** con quello in posizione **`POS`** di un termine **`TERM`**
- Il primo argomento **`POS`** deve sempre essere istanziato ad una espressione aritmetica al momento della valutazione.

```
?-  arg(1,f(a,b),A).
yes A=a
?-  arg(1+2*3,p(a,b,c,d,e,f,g,h,i,j),A).
yesA=g
?-  arg(1,f(g(X),b),A).
yesA=g(_1)
```

Accesso alle componenti di un termine

arg(POS, TERM, ARG)

```
?-  arg(2,p(a,Y),b).
yesY=b
?-  arg(1+1,p(a,g(X)),g(b)).
yesX=b
?-  arg(X,p(a,b),a).
Error in arithmetic expression
```

Lista delle componenti di un termine

```
TERM =.. [FUNCTOR, ARG1, .., ARGn]
TERM =.. [FUNCTOR |[ ARG1, .., ARGn]]
```

```
?-  f(a,b)  =..  [f,a,b].
yes
?-  a  =..  L
yes        L=[a]
?-  f(h(a),b) =.. [FUNCTOR | ARGLIST].
yes        FUNCTOR=f          ARGLIST=[h(a),b]
?- T =.. [g,1,X,h(a)].
yes        T=g(1,_1,h(a))
?-  T =.. [f | [1,2,3]].
yes        T=f(1,2,3).
```

Uso bidirezionale di `=..`
Se **`TERM`** istanziato e lista variabile, o viceversa

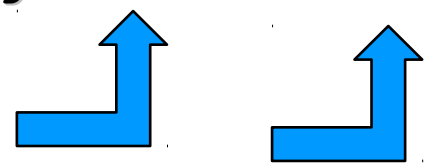
ESERCIZIO 7.2 - MSG

- Dati due termini T1 e T2, determinare la loro generalizzazione più specifica (**MSG, Most Specific Generalization**), ossia il termine più specifico di cui sia T1 sia T2 sono istanze.
- Ad esempio:

T1	T2	MSG
X	X	X
X	Y	Z
a	X	X
a	b	X
f(X)	g(Z)	Y
f(X,a)	f(b,Y)	f(X,Y)

SOLUZIONE ES 7.2 - MSG

```
msg(T1,T2,T1)   :- var(T1), var(T2), T1==T2,!.
msg(T1,T2,_)    :- var(T1),var(T2),!.
msg(T1,T2,T1)   :- var(T1),nonvar(T2),!.
msg(T1,T2,T2)   :- nonvar(T1),var(T2),!.
msg(T1,T2,_)    :- nonvar(T1),
  nonvar(T2),
  functor(T1,F1,N1),
  functor(T2,F2,N2),
  (F1 =\= F2; N1 =\= N2),!.
   or
        diverso in SICStus
                        Prolog
```



SOLUZIONE ES 7.2 – MSG (cont.)

```
msg(T1,T2,T3)   :-    nonvar(T1),
                      nonvar(T2),
                      functor(T1,F,N),
                      functor(T2,F,N),
                      T1 =.. [F | ARGS1],
                      T2 =.. [F | ARGS2],
                  msg_list(ARGS1,ARGS2,ARGS),
                      T3 =.. [F | ARGS].
```

SOLUZIONE ES 7.2 – MSG (cont.)

```
msglist(L1,L2,L)
"L è la lista delle generalizzazioni più
specifiche delle coppie di elementi delle
liste L1 e L2 in ugual posizione"
```

```
msg_list([],[],[]).
msg_list([T1|REST1], [T2|REST2], [T|REST]):-
                msg(T1,T2,T),
                msg_list(REST1,REST2,REST).
```

IL PREDICATO CALL

- In Prolog predicati (programmi) e termini (dati) hanno la stessa struttura e possono essere utilizzati in modo interscambiabile
- Un primo predicato predefinito che può essere utilizzato per trattare i dati come programmi è il predicato **`call`**
- **`call(T)`**: il termine **`T`** viene trattato come un atomo predicativo e viene richiesta la valutazione del goal **`T`** all'interprete Prolog
  - Al momento della valutazione **`T`** deve essere istanziato ad un termine non numerico (eventualmente contenente variabili)

IL PREDICATO CALL

- Il predicato **`call`** può essere considerato come un predicato di meta-livello in quanto consente l'invocazione dell'interprete Prolog all'interno dell'interprete stesso

- Il predicato **`call`** ha come argomento un predicato

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

:- call(q(Y)).
yes Y = a.
```

Il predicato **`call`** richiede all'interprete la dimostrazione di **`q(Y)`**

IL PREDICATO CALL

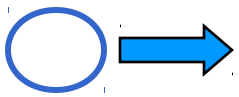
- Il predicato **`call`** può essere utilizzato all'interno di programmi

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

:- p(q(Y)).
yes Y = a.
```

- Una notazione consentita da alcuni interpreti è la seguente

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



`X` variabile meta-logica

ESEMPIO

- Si supponga di voler realizzare un costrutto condizionale del tipo **`if_then_else`**

`if_then_else(Cond,Goal1,Goal2)`

Se **`Cond`** e' vera viene valutato **`Goal1`**, altrimenti **`Goal2`**

```
if_then_else(Cond,Goal1,Goal2):-
  call(Cond), !,
  call(Goal1).
if_then_else(Cond,Goal1,Goal2):-
  call(Goal2).
```

IL PREDICATO FAIL

- **`fail`** e' un predicato predefinito senza argomenti

- La valutazione del predicato **`fail`** fallisce sempre e quindi forza l'attivazione del meccanismo di backtracking

- Vedremo alcuni esempi di uso del predicato fail:
  - Per ottenere forme di iterazione sui dati;
  - Per implementare la negazione per fallimento;
  - Per realizzare una implicazione logica.

IL PREDICATO FAIL: ITERAZIONE

- Si consideri il caso in cui la base di dati contiene una lista di fatti del tipo `p/1` e si voglia chiamare la procedura `q` su ogni elemento `x` che soddisfa il goal `p(X)`

- Una possibile realizzazione e' la seguente:

```
itera :- call(p(X)),
         verifica(q(X)),
         fail.
itera.

verifica(q(X)):- call(q(X)), !.
```

Nota: il `fail` innesca il meccanismo di backtracking quindi tutte le operazioni effettuate da q(X) vengono perse, tranne quelle che hanno effetti non *backtrackabili* (LE VEDREMO)

IL PREDICATO FAIL: NEGAZIONE

- Si supponga di voler realizzare il meccanismo della negazione per fallimento

- **`not(P)`**
  - Vero se P non e' derivabile dal programma

- Una possibile realizzazione e' la seguente:

```
not(P) :- call(P), !,
   fail.
not(P).
```

IL PREDICATO FAIL: IMPLICAZIONE

- Si supponga di voler realizzare una implicazione

  `imply(G1,G2)`

  – Vero se il goal **`G2`** è dimostrato ogni volta che è dimostrato **`G1`**

- Dalla logica sono note le seguenti equivalenze

  $p \Rightarrow q \equiv \neg p \vee q \equiv \neg (p \wedge \neg q)$

- Una possibile realizzazione e' la seguente:

```
imply(G1, G2):-  not((G1, not(G2))).
```

  – **`imply`** si comporta in modo "corretto" se la sostituzione σ per cui **`[G1]`**σ ha successo istanzia ogni variabile di **`G2`**, ossia rende **`[G2]`**σ ground

IL PREDICATO FAIL: IMPLICAZIONE

- Si consideri la seguente base di dati:

```
padre(giuseppe,aldo).
padre(giovanni,mario).
padre(giovanni,franco).
impiegato(aldo).
impiegato(mario).
impiegato(franco).
```

e il seguente goal:

- `:- imply(padre(giovanni,Y), impiegato(Y)).`

  `yes  Y = Y`

- Corrisponde alla formula

$\forall$`Y (padre(giovanni,Y)` $\Rightarrow$ `impiegato(Y))`

IL PREDICATO FAIL: IMPLICAZIONE

- Si consideri la seguente base di dati:

```
padre(giuseppe,aldo).
padre(giovanni,mario).
padre(giovanni,franco).
impiegato(aldo).
impiegato(franco).
```

e il seguente goal:

- `:- imply(padre(giovanni,Y), impiegato(Y)).`

  `no`

  Infatti per `Y/mario` `padre(giovanni,mario)` ha successo, mentre `impiegato(mario)` fallisce

IL PREDICATO FAIL: IMPLICAZIONE

- La relazione **`imply`** può avere dei comportamenti scorretti nel caso in cui la sostituzione **σ** per cui ha successo l'antecedente non istanzia completamente il conseguente
- Si consideri la seguente base di dati:

```
p(X).
q(a).
```

e il seguente goal:

-
```
:-  imply(p(Y), q(Y)).

yes
```

mentre ci si aspetterebbe un fallimento perché **`p`** e' vero per tutti gli **`X`** mentre **`q`** solo per a.

COMBINAZIONE CUT E FAIL

- La combinazione `!,fail` è interessante ogni qual volta si voglia, all'interno di una delle clausole per una relazione `p`, generare un fallimento globale per `p` (e non soltanto un backtracking verso altre clausole per `p`)

- Consideriamo il problema di voler definire una proprietà `p` che vale per tutti gli individui di una data classe tranne alcune eccezioni

COMBINAZIONE CUT E FAIL

- Tipico esempio è la proprietà **`vola`** che vale per ogni individuo della classe degli uccelli tranne alcune eccezioni (ad esempio, i pinguini o gli struzzi)

```
vola(X)   :-  pinguino(X), !, fail.
vola(X)   :-  struzzo(X), !, fail.
....
vola(X)   :-  uccello(X).
```

I PREDICATI SETOF e BAGOF

- Ogni query `:- p(X).` è interpretata dal Prolog in modo esistenziale; viene cioè proposta una istanza per le variabili di `p` che soddisfa la query

- In alcuni casi può essere interessante poter rispondere a query del secondo ordine, ossia a query del tipo quale è l'insieme S di elementi `X` che soddisfano la query `p(X)` ?

- Molte versioni del Prolog forniscono alcuni predicati predefiniti per query del secondo ordine

I PREDICATI SETOF e BAGOF

- I predicati predefiniti per questo scopo sono

  `setof(X,P,S).`

  `S` e' l'insieme delle istanze **`X`** che soddisfano il goal **`P`**

  `bagof(X,P,L).`

  `L` e' la lista delle istanze **`X`** che soddisfano il goal **`P`**

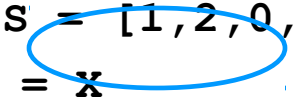
  – In entrambi i casi, se non esistono **`X`** che soddisfano **`P`** i predicati falliscono

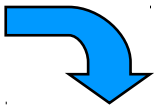
- **`bagof`** produce una lista in cui possono essere contenute ripetizioni, **`setof`** produce una lista corrispondente ad un insieme in cui eventuali ripetizioni sono eliminate.

ESEMPIO

- Supponiamo di avere un data base del tipo

```
p(1).
p(2).
p(0).
p(1).
q(2).
 r(7).
:- setof(X,p(X),S).
yes S = [0,1,2]
    X = X
:- bagof(X,p(X),S).
yes S = [1,2,0,1]
    X = X
```





NOTA: la variabile X alla fine della valutazione non e' legata a nessun valore

ESEMPIO

- Supponiamo di avere un data base del tipo

```
p(1).
p(2).
p(0).
p(1).
q(2).
 r(7).
```

```
:- setof(X,p(X),[0,1,2]).
 yes  X = X

:- bagof(X,p(X), [1,2,0,1]).
 yes X = X
```

ESEMPIO

- Supponiamo di avere un data base del tipo

```
p(1).
p(2).
p(0).
p(1).
q(2).
 r(7).
:- setof(X,(p(X),q(X)),S).
yes S = [2]
   X = X
:- bagof(X,(p(X),q(X)), S).
yes S = [2]
   X = X
```

ESEMPIO

- Supponiamo di avere un data base del tipo

```
p(1).
p(2).
p(0).
p(1).
q(2).
 r(7).
```

```
:- setof(X,(p(X),r(X)),S).
 no

:- bagof(X,(p(X),r(X)), S).
 no
```

ESEMPIO

- Supponiamo di avere un data base del tipo

```
p(1).
p(2).
p(0).
p(1).
q(2).
 r(7).
```

```
:- setof(X, s(X),S).
 no

:- bagof(X, s(X), S).
 no
```

NOTA: questo e' il comportamento atteso. In realtà molti interpreti danno un errore del tipo

```
calling an undefined procedure
s(X)
```

ESEMPIO

- Supponiamo di avere un data base del tipo

```
p(1).
p(2).
p(0).
p(1).
q(2).
 r(7).
```

```
:- setof(p(X), p(X),S).
 yes S=[p(0),p(1),p(2)]
     X=X
:- bagof(p(X), p(X), S).
 yes S=[p(1),p(2),p(0),p(1)]
     X=X
```

ESEMPIO

- Supponiamo di avere un data base del tipo

```
padre(giovanni,mario).
padre(giovanni,giuseppe).
padre(mario, paola).
padre(mario,aldo).
padre(giuseppe,maria).
```

```
:-  setof(X, padre(X,Y), S).
  yes  X=X   Y= aldo      S=[mario];
  X=X  Y= giuseppe S=[giovanni];
  X=X  Y= maria    S=[giuseppe];
  X=X  Y= mario    S=[giovanni];
  X=X  Y= paola    S=[mario];
  no
```

NOTA: non fornisce tutti gli **`X`** *per cui* **`padre(X,Y)`** *e' vera, ma tutti gli* **`X`** *per cui,* ***per lo stesso valore di*** **`Y`**, **`padre(X,Y)`** *e' vera.*

ESEMPIO

- Supponiamo di avere un data base del tipo

```
padre(giovanni,mario).
padre(giovanni,giuseppe).
padre(mario, paola).
padre(mario,aldo).
padre(giuseppe,maria).
```

NOTA: per quantificare esistenzialmente Y si puo' usare questa sintassi

```
:-  setof(X, Y^padre(X,Y), S).
  yes  [giovanni,mario,giuseppe]
       X=X
       Y=Y
```

ESEMPIO

- Supponiamo di avere un data base del tipo

```
padre(giovanni,mario).
padre(giovanni,giuseppe).
padre(mario, paola).
padre(mario,aldo).
padre(giuseppe,maria).

:-  setof((X,Y), padre(X,Y), S).
 yes  S=[(giovanni,mario),(giovanni,giuseppe),
     (mario, paola), (mario,aldo),
     (giuseppe,maria)]
 X=X
 Y=Y
```

IL PREDICATO FINDALL

- Per ottenere la stessa semantica di **`setof`** e **`bagof`** con quantificazione esistenziale per la variabile non usata nel primo argomento esiste un predicato predefinito

  `findall(X,P,S)`

  vero se **`S`** e' la lista delle istanze **`X`** (senza ripetizioni) per cui la proprietà **`P`** e' vera.

- Se non esiste alcun **`X`** per cui **`P`** e' vera **`findall`** non fallisce, ma restituisce una lista vuota.

IL PREDICATO FINDALL

- Supponiamo di avere un data base del tipo

```
padre(giovanni,mario).
padre(giovanni,giuseppe).
padre(mario, paola).
padre(mario,aldo).
padre(giuseppe,maria).

:-  findall(X, padre(X,Y), S).
  yes  S=[giovanni, mario, giuseppe]
       X=X
       Y=Y
```

- Equivale a

```
:-  setof(X, Y^padre(X,Y), S).
```

NON SOLO FATTI

- I predicati **`setof`**, **`bagof`** e **`findall`** funzionano anche se le proprietà che vanno a controllare non sono definite da fatti ma da regole.

```
p(X,Y):- q(X),r(X).
q(0).
q(1).
r(0).
r(2).
:-  findall(X, p(X,Y), S).
  yes  S=[0]
  X=X
  Y=Y
```

IMPLICAZIONE MEDIANTE SETOF

- Vediamo un esempio in cui **`setof`** viene usato per realizzare un'implicazione. Abbiamo predicati del tipo

  `padre(X,Y)` e **`impiegato(Y)`**

  vogliamo verificare se per ogni **`Y`**

  `padre(p,Y)` $\Rightarrow$ **`impiegato(Y)`**

```
implica(Y):- setof(X, padre(Y,X),L),
             verifica(L).
verifica([]).
verifica([H|T]):- impiegato(H),
                  verifica(T).
```

ITERAZIONE MEDIANTE SETOF

- Vediamo un esempio in cui **`setof`** viene usato per realizzare un'iterazione. Vogliamo chiamare la procedura q per ogni elemento per cui vale p.

```
itera:- setof(X, p(X),L),
        scorri(L).
scorri([]).
scorri([H|T]):- call(q(H)),
                scorri(T).
```

NOTA: nell'iterazione realizzata tramite il fail la procedura q doveva produrre effetti non rimovibili dal backtracking. In questo caso invece non e' necessario

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)

```
?-  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
```

```
?-dynamic(p/1).
?-dynamic(q/2).
p(1).
q(X,a)   :-  p(X),
             r(a).
q(2,Y)   :-  d(Y).
```

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"

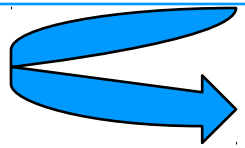
`assertz(T)`

"la clausola T viene aggiunta al fondo del data-base"

ESEMPI assert

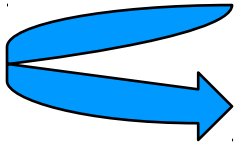
```
?-dynamic(a/1).
a(1).
b(X):-a(X).
```

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



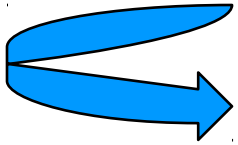
```
a(1).
a(2).
b(X):-a(X).
```

```
?- 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):

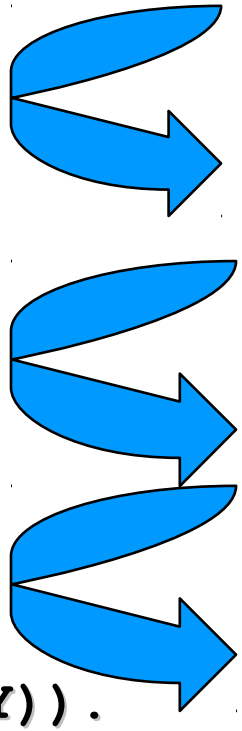
  `abolish(NAME,ARITY)`

ESEMPI retract

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

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

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



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

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

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

ESEMPI retract

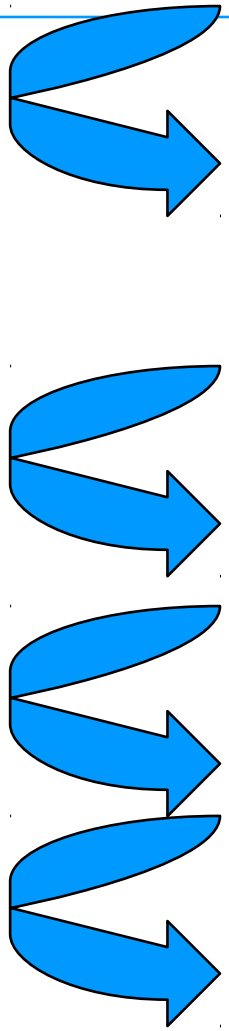
```
?-  retract(a(X)).
yes X=3;

yes X=1;

yes X=2;

yes X=4;

no
```



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

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

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

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

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

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

```
a(1).  (P2)
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 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 ∀**`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 facil… Prolog (almeno …ciale per ognuno di essi prima dell'ultima clausola per "solve".

METAINTERPRETE PER PROLOG PURO

- Oppure:

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

PROLOG MA CON REGOLA DI CALCOLO RIGHT-MOST

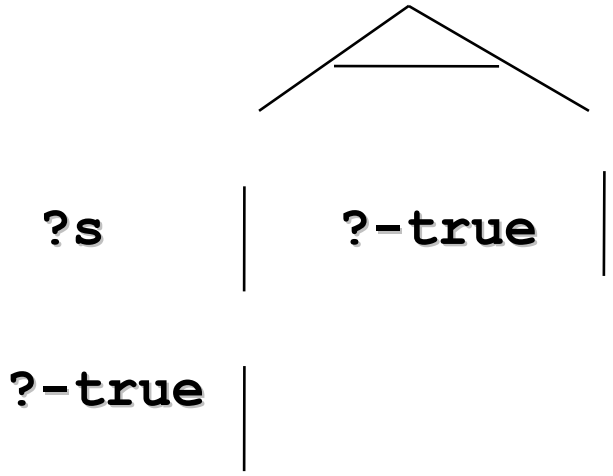
- Il meta-interprete per Prolog puro può essere modificato per adottare una regola di calcolo diversa (ad esempio right-most):

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

ESEMPIO METAINT. CON SPIEGAZIONE

- Si desidera avere, al termine della dimostrazione di un certo goal, una spiegazione della dimostrazione effettuata.
- Un semplice modo per fornire una spiegazione per un goal "g" è quello di stampare l'albero di dimostrazione per "g".
- Esempio:

```
p  :-  q,r.    ?-p
q  :-  s.
r.   ?-q,        r
s.
```



ESEMPIO METAINT. (cont.)

- Dato il programma:

```
p  :-  q,r.
q  :-  s.
r.
s.
```

- l'albero di dimostrazione per la query ?-`p.` può essere visualizzato mediante la seguente espressione:

```
p   <-  (q <- (s <- true)),
(r <- true)
```

- E` sufficiente aggiungere un argomento al predicato "solve" e utilizzare tale argomento per la costruzione della spiegazione.

ESEMPIO METAINT. (cont.)

solve(GOAL,PROVA)

"**PROVA** è un albero di dimostrazione per il goal **GOAL**"

```
?-  op(1200,   xfx, [<-]).
solve(true,true).
solve((A,B),(PROVA1,PROVA2) ) :-
solve(A, PROVA1),
solve(B, PROVA2).
solve(A,(A <- PROVA))    :-
clause(A,B),
solve(B,PROVA).
```

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).
```

SOLUZIONE ESERCIZIO 7.4 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.
```

PROLOG, MA STRATEGIA BREADTH-FIRST

- Realizzare un meta-interprete Prolog che adotti la strategia breadth-first.
- Si tenga traccia in una lista (coda) dei nodi sia dei sottogoal da dimostrare sia del top-goal con le relative istanziazioni.

- Si consideri come esempio il programma seguente:

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

- `?-a(X)` Prolog va in loop.

PROLOG, MA STRATEGIA BREADTH-FIRST

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

- l'evoluzione della coda dei sottogoal sarà:

```
[([a(X)], a(X))]
[([b],a(1)),([c],a(2))]
[([c],a(2)),([b,d],a(1))]
[([b,d],a(1)),([true],a(2))]
[([true],a(2)),([b,d,d],a(1))]
```

SOLUZIONE STRATEGIA BREADTH-FIRST

- **`bf`** costruisce una lista (coda) di sottogoal e del relativo top goal in modo da tenere traccia del ramo dell'albero che si sta percorrendo.

```
metabf(Goal):- bf([([Goal],Goal)],Goal).
```

- Se **`bf`** trova una soluzione, restituisce il top goal (con le variabili istanziate) che ha portato al successo

```
bf([([true],Istanza)|_],Istanza).
```

SOLUZIONE STRATEGIA BREADTH-FIRST

- La **bf** precedente ha trovato una soluzione. La **bf** successiva prosegue la ricerca nei restanti rami dell'albero.

```
bf([[true],_)|S],Goal):-!, bf(S,Goal).
```

- La findall espande il primo elemento della coda [A|B].

```
bf([[A|B],Goal1)|S],Goal):-
findall((BB,Goal1),trova(A,B,BB),L),
append(S,L,Newlist),
bf(Newlist,Goal).
trova(A,B,BB):-clause(A,Body),
mixed_append(Body,B,BB).
```

SOLUZIONE STRATEGIA BREADTH-FIRST

- mixed_append concatena una struttura (X,Y) a una lista Z.

```
mixed_append((X,Y),Z,T):- !,
        mixed_append(Y,Z,W),T=[X|W].
mixed_append(X,Y,[X|Y]).
```

INTERCETTARE L'UNIFICAZIONE

- Si scriva un meta-interprete che intercetti il meccanismo di unificazione del Prolog e che agisca nel modo seguente:
  - nel caso in cui i due termini da unificare siano costanti il metainterprete dovrà verificare l'uguaglianza delle costanti stesse
  - nel caso in cui siano entrambe variabili, o variabile/costante o ancora variabile/termine composto il metainterprete dovrà fornire un messaggio contenente la sostituzione effettuata senza peraltro effettuarla e senza verificare l'occur-check.
  - nel caso in cui siano entrambi termini composti il metainterprete dovrà richiamarsi ricorsivamente sugli argomenti dei termini stessi

    In ogni caso non dovrà essere effettuato un legame delle variabili
  - Si noti che il meccanismo di unificazione del Prolog non dovrà essere mai utilizzato. Inoltre l'unificazione gestita dal meta-interprete dovrà avere carattere locale al termine da unificare.

INTERCETTARE L'UNIFICAZIONE

Si supponga di avere un programma nella forma clausola(Head,Body) dove Body è una lista di sottogoal:

```
clausola(p(X,Y),[q(X),r(Y)]).
clausola(q(3),[]).
clausola(r(2),[]).
```

e di voler chiamare il metainterprete con il goal p(4,5).

Un normale programma Prolog fallirebbe. Il metainterprete deve avere successo producendo i messaggi:

```
X/4 Y/5
X/3 Y/2
```

rispettivamente per p(X,Y) e per q(X) e r(Y).

Le variabili X e Y al termine della valutazione non dovranno essere istanziate.

INTERCETTARE L'UNIFICAZIONE

Si supponga di avere un programma nella forma clausola(Head,Body) dove Body è una lista di sottogoal:

```
clausola(p(X,Y),[q(X),r(Y)]).
clausola(q(3),[]).
clausola(r(2),[]).
```

Come cambia il metainterprete vanilla?

```
solve([]):-!.
solve([A|B]):- !,solve(A),solve(B).
solve(A) :- clausola(A,B),solve(B).
```

INTERCETTARE L'UNIFICAZIONE

Per intercettare l'unificazione è necessario crearsi un template del goal da unificare in cui tutti i parametri sono variabili: in questo modo non altero la struttura dei parametri della testa della clausola

```
solve([]):-!.
solve([A|B]):- !,solve(A),solve(B).
solve(Goal) :-
      functor(Goal,F,A),
      functor(Templ,F,A), % template
      clausola_unif(Goal,Templ).
```

INTERCETTARE L'UNIFICAZIONE

clausola_unif utilizza il template per l'unificazione con la clausola nel database poi controlla se gli argomenti unificano

```
clausola_unif(Goal,Templ):-
      clausola(Templ,Body),
      Goal=..[_|Arg],
      Templ=..[_|Arg1],
      unificano(Arg,Arg1),
      check(Body).
```

INTERCETTARE L'UNIFICAZIONE

unificano controlla che liste di argomenti possano unificare

```
unificano([],[]).
```

Caso 1: due costanti unificano se sono uguali

```
unificano([H|Tail],[H1|Tail1]):-
      atomic(H),atomic(H1),
      H=H1,!,
      unificano(Tail,Tail1).
```

INTERCETTARE L'UNIFICAZIONE

Caso 2: dal momento che non si effettuano link di variabili una variabile unifica con qualunque termine (atomico o composto)

```
unificano([H|Tail],[H1|Tail1]):-
      (var(H);var(H1)),!,
      write('sostituzione '),
      write(H/H1), nl,
      unificano(Tail,Tail1).
```

INTERCETTARE L'UNIFICAZIONE

Caso 3: due termini composti unificano se unificano i funtori e gli argomenti per la verifica degli argomenti unificano/2 si richiama ricorsivamente

```
unificano([H|Tail],[H1|Tail1]):-
        compound(H),compound(H1),
        H=..[Head|Arg],
        H1=..[Head|Arg1],
        unificano(Arg,Arg1),!,
        unificano(Tail,Tail1).
```

MODIFICA ORDINE CLAUSOLE

Si scriva un metainterprete per Prolog che interpreti prima le clausole il cui body contiene un maggior numero di sottogoal diversi. Due sottogoal sono considerati uguali se hanno stesso funtore e stessa arità. Ad esempio, date le due clausole:

```
p(X,Y,Z):- a(3, 3), a(Y, Z), b(Z, Z).
p(X,Y,Z):-  b(Y, Y), a(X, Y), c(Z).
```

Il metainterprete sceglierà prima la seconda clausola: infatti, il numero di sottogoal diversi della seconda clausola è 3 mentre per la prima è 2 (a(3,3) è considerato uguale ad a(Y,Z)).

MODIFICA ORDINE CLAUSOLE

IPOTESI:

Si supponga di avere a disposizione un predicato **`sort(L,L1,L2)`** che fornisce in L2 la lista L ordinata (in senso decrescente) secondo i valori contenuti in L1.

Si supponga di avere un programma nella forma **`clausola(Head,Body)`** dove Body è una lista di sottogoal

MODIFICA ORDINE CLAUSOLE

```
solve([]).
solve([Head|Tail]):-
      solve(Head),
      solve(Tail).

solve(Goal):-
findall((Goal,Body),clausola(Goal,Body),
L),
      ordina(L,Lord),
      member((Goal,FirstBody), Lord),
      solve(FirstBody).

ordina(L,Lord):-
      conta_sottogoal(L,Nsottogoal),
```

MODIFICA ORDINE CLAUSOLE

```
conta_sottogoal([],[]).

conta_sottogoal([H|T],[NH|NT]):-
      conta_sottogoal1(H,NH),
      conta_sottogoal(T,NT).

conta_sottogoal1((_,Body),Nsottogoal):-
      conta_sottogoal1(Body,0,Nsottogoal).
```

MODIFICA ORDINE CLAUSOLE

```
conta_sottogoal1([],N,N).

conta_sottogoal1([Head|Tail],Nin,Nout):-
      functor(Head, F, A),
      functor(Temp, F, A),
      member(Temp, Tail), !,
      conta_sottogoal1(Tail,Nin,Nout).

conta_sottogoal1([Head|Tail],Nin,Nout):-
      Ntemp is Nin +1,
      conta_sottogoal1(Tail, Ntemp,Nout).
```