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

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

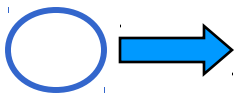
- Il predicato **`call`** può essere utilizzato all'interno di programmi

```
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

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).
```

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"

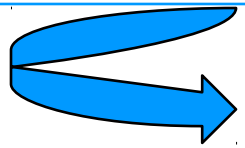
`assertz(T)`

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

ESEMPI assert

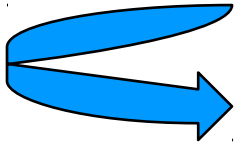
```
?-dynamic(a/1).
a(1).
b(X):-a(X).
```

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



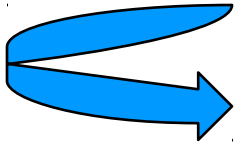
```
a(1).
a(2).
b(X):-a(X).
```

```
?- 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):

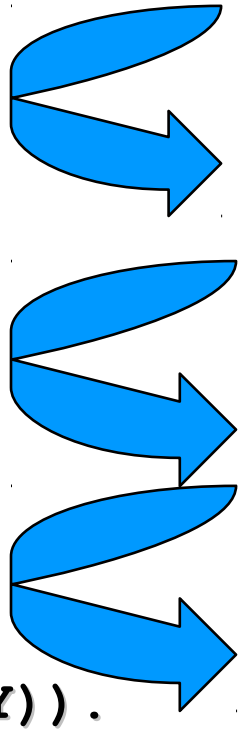
  `abolish(NAME,ARITY)`

ESEMPI retract

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

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

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



```
?-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

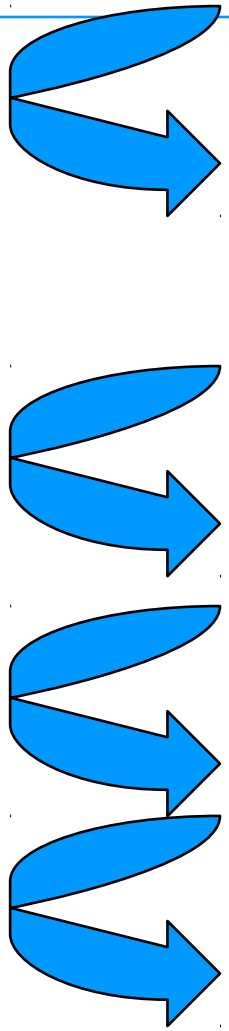
```
?-  retract(a(X)).
yes X=3;

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).
```

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.

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 rolog (almeno ciale per ognuno di essi prima dell'ultima clausola per "solve".

ESERCIZIO

Si scriva un metainterprete che riceva in ingresso, oltre al goal da dimostrare, il nome di un predicato e la sua arità e conti, nel corso della computazione (solo i rami di successo) il numero di chiamate al predicato.

Ad esempio, dato il programma:

```
p(X):- q(X), r(X,Y).
q(1).
q(2).
r(1,Y):- s(Y).
r(2,Y):- t(Y).
t(3).
```

Il metainterprete risponde alla chiamata `solve(p(Y), r,2,N)` yes legando Y a 2 e `N` a `1`.

ESERCIZIO

Si scriva un metainterprete per Prolog che all'interno del body di ogni clausola selezioni sempre prima quei sottogoal con un maggior numero di parametri diversi **prima dell'unificazione**.

Due parametri sono considerati uguali se sono due costanti uguali, oppure due variabili con nome uguale, oppure due termini composti con uguale funtore, uguale arità e argomenti uguali.

Ad esempio, nel body di

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

selezionerà prima `a(X, Y)` (numero di parametri diversi = 2) poi indifferentemente `b(Y, Y)` (numero di parametri diversi = 1) o `c(Z)` (numero di parametri diversi = 1).

Si realizzi un predicato `sort(L,L1,L2)` che fornisce in `L2` la lista `L` ordinata (in senso decrescente) secondo i valori contenuti in `L1`.

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

ESERCIZIO

Si scriva un meta-interprete che intercetti il meccanismo di unificazione del Prolog e che lo utilizzi solo se il termine da unificare (goal o sottogoal) è una generalizzazione del termine che unifica (testa della clausola).

- Due costanti non unificano mai (anche se sono uguali).
- Nel caso in cui i termini da unificare siano entrambi variabili o il termine da unificare è una variabile e il termine che unifica una costante, l'unificazione avviene normalmente. Si noti che se il termine da unificare è una costante e il termine che unifica una variabile l'unificazione fallisce.
- Nel caso in cui i termini da unificare siano entrambi composti, l'unificazione avviene se i funtori sono identici mentre per l'unificazione degli argomenti il metainterprete dovrà richiamarsi ricorsivamente sugli argomenti dei termini stessi

Esempio

```
p(X,Y):- q(X),r(2).
p(X,Y):- s(Y).
q(3).
r(2).
s(Y):- r(Y).
```

Si supponga di voler chiamare il metainterprete con il goal `p(A,B)`.

A e `B`, essendo due variabili, unificano con `X` e `Y`. Viene quindi eseguito `q(X)` che è una generalizzazione di `q(3)` e quindi unifica. Per `r(2)`, invece, non esistono termini meno generali. Quindi la computazione fallisce. Rimane quindi choice point per `p(X,Y)` che viene esplorato e porta ad un ramo di successo. Questa volta infatti `r(Y)` è più generale di `r(2)`.

Lo stesso metainterprete, con il goal `p(2,B)`, fallirebbe subito.