

# 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

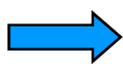
```
p(X) :- call(X).
```

```
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.
```

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)

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

# 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) .
```

## 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** ( $\mathbf{X}, \mathbf{P}, \mathbf{S}$ ) .
  - $\mathbf{S}$  e' l'insieme delle istanze  $\mathbf{X}$  che soddisfano il goal  $\mathbf{P}$
- **bagof** ( $\mathbf{X}, \mathbf{P}, \mathbf{L}$ ) .
  - $\mathbf{L}$  e' la lista delle istanze  $\mathbf{X}$  che soddisfano il goal  $\mathbf{P}$
  - In entrambi i casi, se non esistono  $\mathbf{X}$  che soddisfano  $\mathbf{P}$  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) .`

*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.*

`:- 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`

## 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`  
 $\text{padre}(p, Y) \Rightarrow \text{impiegato}(Y)$

```
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