META-PREDICATI

- In Prolog predicati (programmi) e termini (dati) hanno la stessa struttura e possono essere utilizzati in modo interscambiabile

```
sum(0,X,X).
sum(s(X),Y,s(Z)):- sum(X,Y,Z).
```

- Operatore (non associativo) `:-`

```
:-(sum(0,X,X),true).
:-(sum(s(X),Y,s(Z)), (sum(X,Y,Z))).
```

- Tutto è un termine (meta-programmazione)

IL PREDICATO CALL

- Meta-predicati, hanno come argomenti termini che rappresentano parti di programma
- 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 (termine non variabile e non numerico)

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

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

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

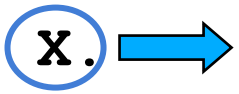
IL PREDICATO CALL

- 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

Meta-predicato `call` in SICStus Prolog

- In SICStus Prolog, per `call` si può scrivere semplicemente il termine da chiamare.

- Sono equivalenti:

```
:- read(P),call(P).
:- read(P),P.
```

- Con il programma

```
p(1).
```

- La risposta, scrivendo `p(X).` a terminale è

```
yes P=p(1)
```

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“

Esercizio

- Scrivere un database di fatti:

```
p(1).
p(2).
p(X):-q(X).
q(3).
```

- e provare il goal:

```
:- read(P), nonvar(P),\+number(P),call(P).
```

ESEMPIO

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

`if_then_else(Cond,Goal1,Goal2)`

Se **`Cond`** è 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`** è 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 è la seguente:

```
itera :- call(p(X)),
         verifica(q(X)),
         write(X),nl,
         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* (ad es. assert, retract NON VISTE)

IL PREDICATO FAIL: ITERAZIONE

```
itera :- call(p(X)),
         verifica(q(X)),
         write(X),nl, assert(pq(X)),
         fail.
itera.

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

p(1).                          pq(1).
p(2).                          pq(3).
p(3).
q(1).
q(3).
```

IL PREDICATO FAIL: NEGAZIONE

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

- **`not(P)`** **`(not è \+ in SICStusProlog)`**
  - Vero se P non è derivabile dal programma

- Una possibile realizzazione è 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`)

ESEMPIO CUT E FAIL

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

- Tipico esempio è la proprietà "volare" 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` è l'insieme delle istanze **`X`** che soddisfano il goal **`P`**

  `bagof(X,P,L).`

  `L` è 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).
```

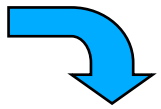
```
:- setof(X,p(X),S).
 yes S = [0,1,2]
     X = X
:- bagof(X,p(X),S).
 yes S = [1,2,0,1]
     X = X
```

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).
```



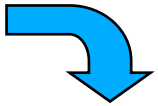
NOTA: Anche il terzo argomento può essere dato in ingresso a un goal

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

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

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

ESEMPIO

- Supponiamo di avere un data base del tipo

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



NOTA: Il secondo argomento può essere un goal più complesso (congiunzione)

```
:- 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
```

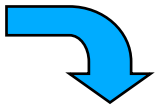
NOTA: questo e' il comportamento atteso. In realtà molti interpreti danno un errore del tipo

```
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).
```



NOTA: Il primo argomento può essere un termine complesso (non solo una variabile)

```
:- setof(f(X), p(X), S).
 yes S=[f(0),f(1),f(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, giuseppe, mario]
      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`** è la lista delle istanze **`X`** per cui la proprietà **`P`** è vera.

- Se non esiste alcun **`X`** per cui **`P`** è vera **`findall`** non fallisce, ma restituisce una lista vuota (errore in SICStusProlog se non esiste il predicato chiamato; lista vuota se esiste, ma non ci sono soluzioni)

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,giovanni,mario,mario,giuseppe]
       X=X
       Y=Y
```

- Equivale a

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

NON SOLO FATTI, MA REGOLE

- 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),c(Y).
q(0).
q(1).
r(0).
r(2).
c(1).
c(2).
```

- **Esercizio:** raccogliere le soluzioni X per la chiamata p(X,Y) (a parità di Y, e per qualsiasi Y)
- **Esercizio:** raccogliere le soluzioni (X,Y) per la chiamata p(X,Y).

NON SOLO FATTI, MA REGOLE

```
p(X,Y):- q(X),r(X),c(Y).
q(0).
q(1).
r(0).
R(1).
c(1).
c(2).

:- findall(X, p(X,Y), S).
:- bagof(X, Y^p(X,Y), S).

:- findall((X,Y), p(X,Y), S).
:- bagof((X,Y), p(X,Y), S).
```

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 tutti i figli di un certo padre *p* sono impiegati: per un dato *p*, vale per ogni **`Y`**

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

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

Esercizio 4.1

- Dato un insieme di studenti e i voti ottenuti negli esami, rappresentati come fatti del tipo:

  studente(Nome, Voto).

- si definisca il predicato **studMedia(Nome,Media)** che dato il nome di uno studente (**Nome**) ne determina la media in **Media**.

```
studente(lucia,31).
studente(carlo,18).
studente(carlo,22).
studente(lucia,30).
?-studMedia(carlo,M).
M=20
```

Esercizio 4.1 - soluzione

```
studMedia(Nome,Media):-
      findall(Voto,studente(Nome,Voto),List),
      media(List, Media).

%calcola la media su una lista di numeri
media(Lista,Media) :-
                  length(Lista,LunghezzaLista),
                  sum(Lista,Somma),
                Media is Somma / LunghezzaLista.

sum([],0).
sum([A|B],N):- sum(B,N1),N is N1+A.
```

Esercizio 3.2

- Si definisca un predicato

  `prodotto_cartesiano(L1,L2,L3)` che, date due liste qualsiasi L1 e L2 restituisca la lista L3 delle coppie ordinate che si possono formare con elementi di L1 e L2. Per esempio:

  ```
  ?-prodotto_cartesiano([a,b,c],[a,d],L3)
  L3=[(a,a),(a,d),(b,a),(b,d),(c,a),(c,d)]
  ```

- Oppure come lista di liste:

  ```
  L3=[[a,a],[a,d],[b,a,],[b,d],[c,a],[c,d]]
  ```

Es. 3.2 – sol. 1

```
prodotto_cartesiano(L1,L2,L3):-
findall([A,B],(member(A,L1),member(B,L2)),L3).
```

Es. 3.2 –sol. 2

```
prodotto_cartesiano([],_,[]).
prodotto_cartesiano([A|T],L2,L3):- prod(A,L2,L),
                        prodotto_cartesiano(T,L2,L4),
                        append(L,L4,L3).

prod(_,[],[]).
prod(A,[H|T],[[A,H]|M]):- prod(A,T,M).

append([],X,X).
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).
```

Esercizio 4.2 – costante di colonna

- Data una matrice **M** (come lista di liste) si scriva un un predicato **constant(M,C)** che è vero se **C** è un numero che compare in ciascuna riga della matrice **M**

Esempio:

```
?- constant (
  [[4, 9, 23, 55, 63, 107, 239],
   [5, 9, 31, 55, 60, 73, 82, 99, 107],
   [9, 23, 55, 107, 128, 512],
   [6, 9, 13, 17, 22, 55, 63, 107 ]], 9).
Yes
```

Esercizio 4.2 (cont.)

```
?- constant (
  [[4, 9, 23, 55, 63, 107, 239],
   [5, 9, 31, 55, 60, 73, 82, 99, 107],
   [9, 23, 55, 107, 128, 512],
   [6, 9, 13, 17, 22, 55, 63, 107 ]], 60).
No
```

Esercizio 4.2 – sol.

```
constant([], _).
constant([R|Rs], C) :-
        member(C, R),
        constant(Rs, C).

member(H,[H|_]).
member(H,[_|T]) :-
        member(H,T).
```

Esercizio 4.3

- Dato il predicato **constant/2**, definire ora un predicato **constant1(M,L)**che data la matrice **M** restituisce la lista **L** dei numeri che compaiono in ciascuna riga della matrice.
- Esempio:

?-constant1(
[[4, 9, 23, 55, 63, 107, 239],
[5, 9, 31, 55, 60, 73, 82, 99, 107],
[9, 23, 55, 107, 128, 512],
[6, 9, 13, 17, 22, 55, 63, 107]], CC).

Yes CC = [9, 55, 107]

Esercizio 4.3 – sol.

```
constant1(M,L):-
   findall(Col, constant(M,Col), L).
```

Esercizio 3.3

- Si definisca il predicato Prolog **`numat(List,Num)`** che data una lista **`List`** che può contenere quali elementi atomi o liste (a loro volta ricorsivamente contenenti atomi o liste), restituisce in uscita il numero **`Num`** di tutti gli atomi contenuti.
- Esempio:

```
?- numat([[],a,[a,b,c],[a,[g,[h]]]],Num).
Num = 7;
no
```

Es. 3.3

```
numat([],0).
numat([X|Y],Num):- is_a_list(X), !,
                   numat(X,Num1),
                   numat(Y,Num2),
                   Num is Num1 + Num2.
numat([X|Y],Num):- numat(Y,Num2),
                   Num is 1 + Num2.

is_a_list([]).
is_a_list([_|_]).
```

Esercizio 3.1: Critto-aritmetica

Si vogliono risolvere problemi di crittoaritmetica del tipo:

```
A B +
P Q =
-----
X Y
```

assegnando ad ogni lettera una cifra diversa.

A questo scopo si vuole definire la relazione:
sum(N1,N2,N) "Sommando la lista di caratteri N1 alla lista di caratteri N2 si ottiene la lista di caratteri N e ad ogni carattere incognito viene assegnata una cifra diversa“

Per semplicità si considerano N1, N2 ed N tutte della stessa lunghezza.

Esercizio 3.1: Critto-aritmetica

Tra gli operatori e le relazioni predefinite del Prolog possono sicuramente essere utili:

nonvar(X)	"X non è una variabile (oppure è una variabile già legata)"
A // B	"Divisione intera tra A e B"
A mod B	"Resto della divisione intera tra A e B"

Esempi di utilizzo di sum(N1,N2,N):

```
  A B +
  C D =
  -----
  E F
```

```
?- sum([A,B],[C,D],[E,F]).
Yes     A = 4
        B = 1
        C = 5
        D = 2
        E = 9
        F = 3
```

Esercizio 3.1: Critto-aritmetica

```
D O N A L D +
G E R A L D =
---------------
R O B E R T
```

```
?- sum([D,O,N,A,L,D],[G,E,R,A,L,D],[R,O,B,E,R,T]).
D = 5
O = 2
N = 6
A = 4
L = 8
G = 1
E = 9
R = 7
B = 3
T = 0 ;
no
```

Esercizio 3.1: Critto-aritmetica

S E N D +
M O R E =

M O N E Y

?- sum([0,S,E,N,D],[0,M,O,R,E],[M,O,N,E,Y]).
S = 7
E = 5
N = 3
D = 1
M = 0
O = 8
R = 2
Y = 6
Yes

Soluzione vedi Laboratorio Prolog Esercitazione 3.

Soluzione es. 3.1

- *sum(N1,N2,N) , sommando la lista di caratteri N1 alla lista di caratteri N2 si ottiene la lista di caratteri N e ad ogni carattere incognito viene assegnata una cifra diversa*
- *Predicato ausiliario: sum(N1,N2,N,C1,C2,Digs1,Digs2) Opera come sum(N1,N2,N), considerando un riporto iniziale C1, un riporto finale C2, un insieme di cifre disponibili Digs1, un insieme di cifre non utilizzate Digs2"*

```
sum(A, B, C) :-
        sum(A, B, C, 0, 0, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], _).
sum([], [], [], 0, 0, A, A).
sum([F|A], [G|B], [I|C], D, J, DigIn, L) :-
        sum(A, B, C, D, H, DigIn, DigOutTemp),
        digitsum(F, G, H, I, J, DigOutTemp, DigOut).
```

Soluzione es. 3.1

- *Predicati ausiliari*
- *digitsum(D1,D2,C1,D,C,Digs1,Digs2), sommando D1, D2 e C1 si ottiene D con riporto C e ad ogni carattere incognito viene assegnata una cifra diversa; l'insieme delle cifre disponibili è Digs1, quello delle cifre non utilizzate è Digs2*

```
digitsum(A, C, H, E, J, B, G) :-
        del(A, B, D),
        del(C, D, F),
        del(E, F, G),
        I is A+C+H,
        E is I mod 10,
        J is I//10.
```

Soluzione es. 3.1

- *Predicati ausiliari: l'insieme delle cifre disponibili è Digs1, quello delle cifre non utilizzate è Digs2*

- *del(D,Digs1,Digs2): se D è già istanziata l'insieme Digs2 coincide con Digs1; se D non è istanziata, l'insieme Digs2 si ottiene dall'insieme Digs1 eliminando il primo termine che unifica con D.*

```
del(B, A, A) :-
        nonvar(B), !.
del(A, [A|B], B).
del(B, [A|C], [A|D]) :-
        del(B, C, D).
```

% punto di backtracking aperto