

# **Prolog: un linguaggio di programmazione logica**

Armando Stellato

[stellato@info.uniroma2.it](mailto:stellato@info.uniroma2.it)

# Acknowledgements

- Il materiale di queste slides è un “summa” riorganizzato delle informazioni presenti nei precedenti lavori di:
  - Programming Languages
    - Adam Webber  
<http://www.webber-labs.com/mpl/lectures/19.ppt>
  - “Prolog in 90 minutes”
    - Ulf Nilsson, Linköping University
  - Fabio Massimo Zanzotto
    - Slide dello scorso anno
  - Miei appunti personali

# Indice degli argomenti

---

- 0. Le origini...
- 1. Elementi del linguaggio
- 2. L'interprete prolog
- 3. Regole
- 4. Un Database in Prolog
- 5. Regole Ricorsive
- 6. Operatori
- 7. Liste
  - 1. Il predicato append/3
  - 2. Altri predicati sulle liste
  - 3. Insertion Sort
- 8. Il predicato not/1

## 0. Le origini...

---

- La prima versione ufficiale di Prolog è stata sviluppata all'università di Marsiglia, in Francia, da Alain Colmerauer all'inizio degli anni 70, come strumento di programmazione logica.

# 1. Elementi del linguaggio

---

- Termini
- Fatti e Regole
  - I termini sono usati come strutture di dati
- Predicati
  - Composti di fatti e regole
- Il Programma Logico
  - Composto di predicati

- I termini sono dati:
  - Costanti
    - mario gino 5 3.14 [] 'Adam' ...
  - Variabili
    - Iniziano con lettera grande (o con in il simbolo '\_')
    - X Y List \_12 \_ ...
  - Termini composti
    - somma(2,3) sopra(cuboA,sopra(cuboA,cuboB)) ...
    - 2+3 // notazione infissa

- Un fatto è espresso dalla notazione  $p(t_1, \dots, t_n)$ .
  - $p$  è il nome del fatto
  - $t_1, \dots, t_n$  sono gli argomenti del fatto
    - $t_1, \dots, t_n$  sono termini

- Esempi:

`lato(a, x).`

`parent(adam, bill).`

Ogni fatto si chiude con un punto

# Esempi

```
parent(kim,holly).  
parent(margaret,kim).  
parent(margaret,kent).  
parent(esther,margaret).  
parent(herbert,margaret).  
parent(herbert,jean).
```

- Sei fatti riguardanti la relazione “genitore”:
  - Kim è il genitore di Holly (secondo una interpretazione assegnata)
- Ho implicitamente definito un predicato parent di arità 2 sul dominio
  - In breve: `parent/2`



# I Fatti *non sono* Termini

---

- Un Fatto :

`fratello(aldo, giovanni).`

Notare  
il punto

– è parte del programma.

- Un termine:

`finestra(32, 56)`

– è un frammento dei dati utilizzati all'interno di un fatto (o regola).

## Esempi di Fatti composti di più termini

---

- Predicato staff/2:
  - staff( nome(qui), room(101) ).
  - staff( nome(quo), room(403) ).
  - staff( nome(qua), room(301) ).
- Ogni predicato staff/2 utilizza due termini.

- Un *programma logico* è rappresentato da un insieme di predicati:
  - Un predicato è composto di fatti e regole
- The program is used to answer user queries.
- *Prolog è un linguaggio di programmazione logica*

## 2. L'interprete prolog

---

- Per le nostre esercitazioni, utilizzeremo

SWI-Prolog

- interprete Prolog sviluppato all'università SWI di Amsterdam e distribuito gratuitamente su internet
- Per scaricarlo  
<http://www.swi-prolog.org/>

# Prompt dei comandi

```
Welcome to SWI-Prolog
...
For help, use ?- help(Topic). or ?- apropos(Word)

1 ?-
```

- La shell Prolog richiede una query usando i simboli  
?-
- Forma di Interazione:
  - Esecuzione di una query
  - Stampa dei risultati
  - Stampa del simbolo di prompt

# Il predicato `consult`

```
?- consult('parents.pl').
```

```
Yes
```

```
?-
```

← Notare il punto '.'

- `consult/1` legge un programma logico dal file specificato
- In questo caso, *parents.pl* contiene i fatti circa il predicato `parent/2` presentato precedentemente

# Esempi di Query

note the '.'

```
?- parent(margaret,kent).
```

**Yes**

```
?- parent(fred,pebbles).
```

**No**

```
?-
```

# Query contenenti variabili

```
?- parent(P, jean) .
```

```
P = herbert ←
```

```
Yes
```

```
?- parent(P, esther) .
```

```
No
```

*Dopo aver prodotto il risultato della query, il Prolog attende nuovamente un input. Se premiamo “Enter” chiudiamo la query.*

- L'interprete Prolog mostra i “bindings” che provano la query.



- Le variabili possono apparire ovunque in una query:
  - ?- parent (Parent, jean) .
  - ?- parent (esther, Child) .
  - ?- parent (Parent, Child) .
  - ?- parent (Person, Person) .

# Congiunzioni

osservate la ','

```
?- parent(margaret,X), parent(X,holly).
```

```
X = kim
```

```
Yes
```

- Una congiunzione equivale ad una serie di query
- Il sistema Prolog prova a dimostrare tutte le query attraverso opportune sostituzioni di variabili
  - e.g. sostituisci X con kim

# Soluzioni Multiple

```
?- parent(margaret,Child).
```

```
Child = kim ;
```

```
Child = kent ;
```

```
No
```

L'utente, invece di premere "Enter", digita ';' per far cercare nuove possibili soluzioni al sistema

- Potrebbero esistere più soluzioni che provano la query

### 3. Regole

*“Head”, o “Testa”, della regola*

`greatgrandparent (GGP, GGC) :-`

`parent (GGP, GP),`

`parent (GP, P),`

`parent (P, GGC) .`

*condizioni*

- Per dimostrare la testa, occorre dimostrare le condizioni.
- Per dimostrare `greatgrandparent (GGP, GGC)`, il sistema cerca delle sostituzioni per `GP` e `P` tali che sia possibile provare, nel seguente ordine, `parent (GGP, GP)`, poi `parent (GP, P)`, ed infine `parent (P, GGC)`.

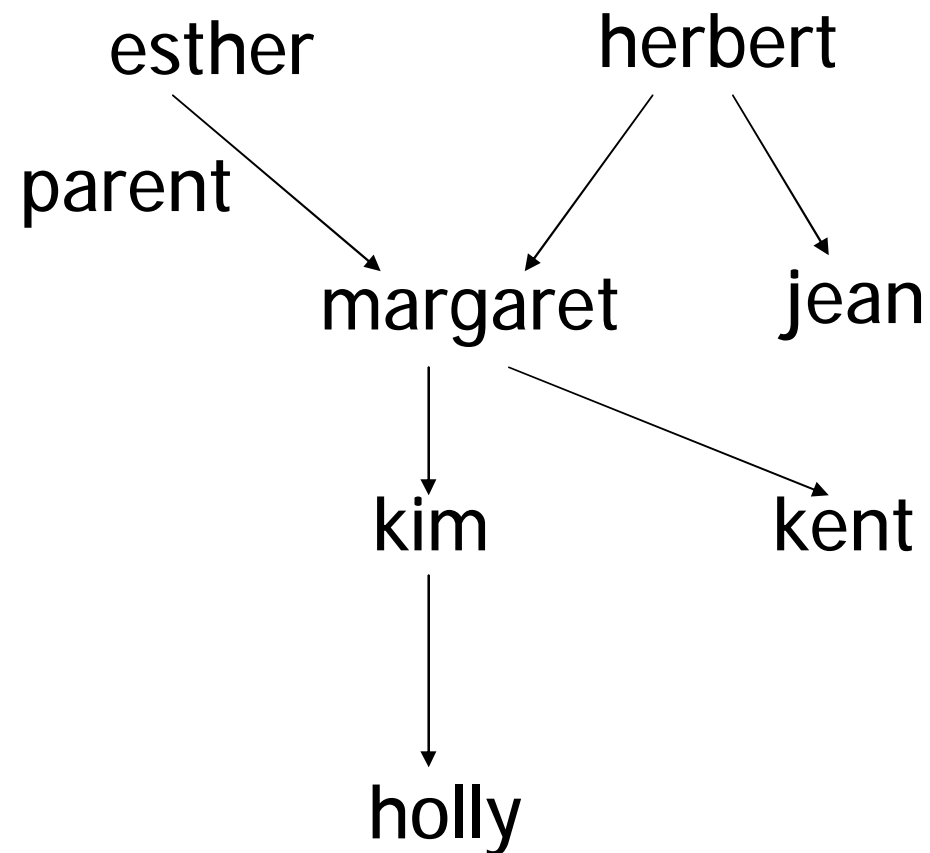
# Un programma con una regola

```
parent(kim,holly).  
parent(margaret,kim).  
parent(margaret,kent).  
parent(esther,margaret).  
parent(herbert,margaret).  
parent(herbert,jean).
```

```
greatgrandparent(GGP,GGC) :-  
    parent(GGP,GP), parent(GP,P), parent(P,GGC).
```

- All'interno del programma sono definiti i predicati `parent/2` e `greatgrandparent/2`
- Per dimostrare `greatgrandparent/2` è necessario dimostrare le sue condizioni, per sostituzioni ammissibili delle sue variabili

# Grafo di parent/2



# Esempio

```
?- greatgrandparent(esther, GreatGrandchild).
```

```
GreatGrandchild = holly
```

```
Yes
```

- La query viene proposta alla shell
- Il sistema ci informa (con la risposta **yes**) che esiste una soluzione per la query sottoposta
- veniamo inoltre informati per quale sostituzione della variabile GreatGrandChild la query è stata soddisfatta
- Come è possibile intuire dalla dichiarazione del predicato `greatgrandparent`, vi sono inoltre dei *goal*/intermedi che devono essere soddisfatti per verificare la soluzione della query. Il sistema riporta comunque solo le sostituzioni delle variabili che sono presenti nella query.

1. `parent(kim,holly).`
2. `parent(margaret,kim).`
3. `parent(margaret,kent).`
4. `parent(esther,margaret).`
5. `parent(herbert,margaret).`
6. `parent(herbert,jean).`
7. `greatgrandparent(GGP,GGC) :-`  
    `parent(GGP,GP), parent(GP,P), parent(P,GGC).`

`greatgrandparent(esther,GreatGrandchild)`

↓ Clausola 7, sostituisce **GGP** con **esther** e **GGC** con **GreatGrandChild**

`parent(esther,GP), parent(GP,P), parent(P,GreatGrandchild)`

↓ Clausola 4, sostituisce **GP** con **margaret**

`parent(margaret,P), parent(P,GreatGrandchild)`

↓ Clausola 2, sostituisce **P** con **kim**

`parent(kim,GreatGrandchild)`

↓ Clausola 1, sostituisce **GreatGrandchild** con **holly**



# Regole basate su regole

```
greatgrandparent (GGP,GGC) :-  
    grandparent (GGP,P), parent (P,GGC).
```

```
grandparent (GP,GC) :-  
    parent (GP,P), parent (P,GC).
```

- Entrambe le regole usano una variabile **P**.
- L'ambito di una variabile è tuttavia è ristretto al fatto/regola che la contiene.

## 4. Un esempio di Database in Prolog

```
lecturer(Lecturer, Course) :-  
    course(Course, _, Lecturer, _).
```

```
teaches(Lect, Day) :-  
    course(_, time(Day, _, _), Lect, _).
```

```
duration(Course, Length) :-  
    course(Course, time(_, S, F), _, _),  
    Length is F-S.
```

```
occupied(Room, Day, Time) :-  
    course(_, time(Day, S, F), _, Room),  
    S =< Time,  
    Time =< F.
```

% Database

```
course(logic, time(monday, 8, 10), dave, a12).  
course(java, time(tuesday, 9, 11), ad, r204).
```

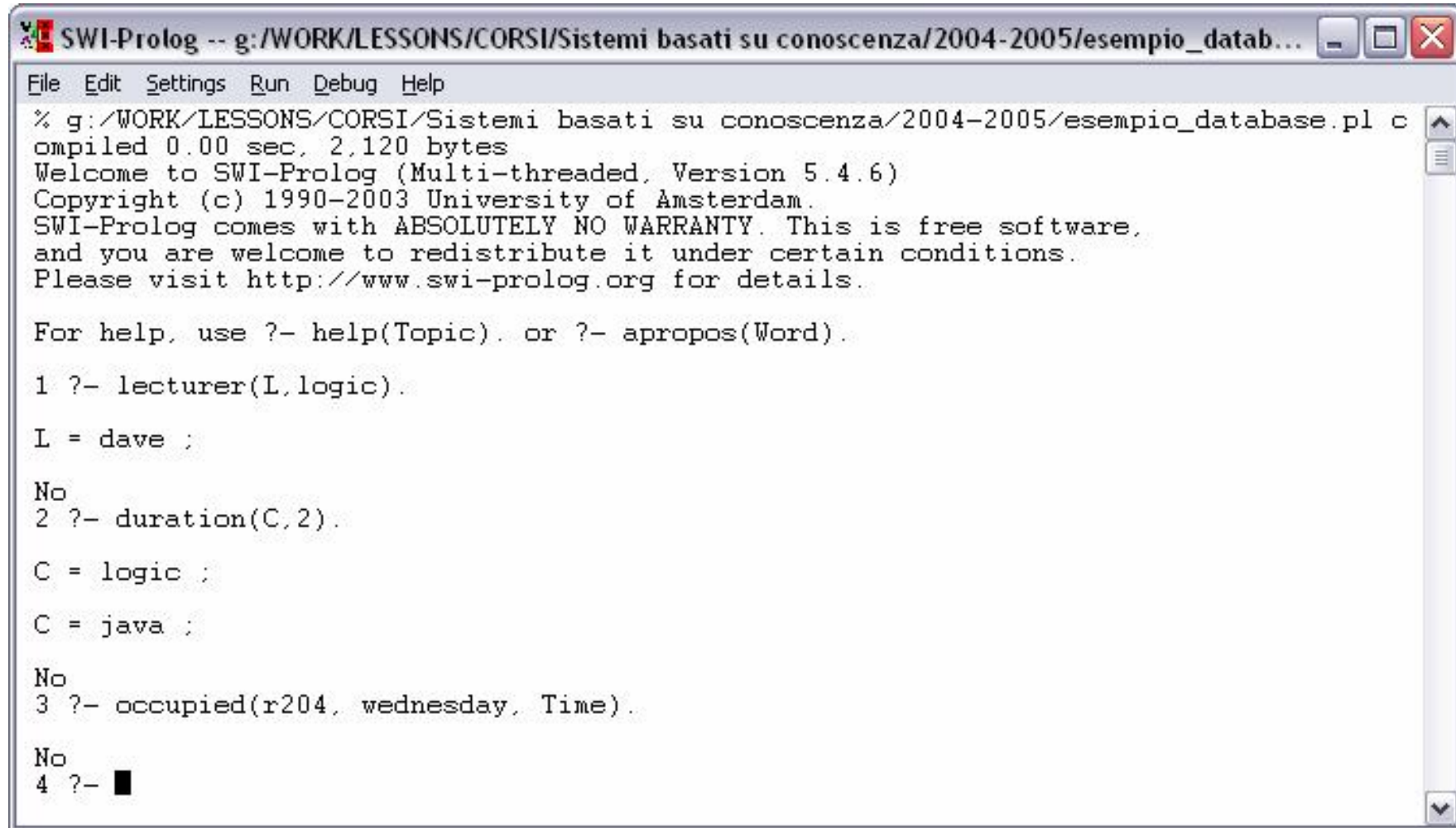
:

## ...e delle query su di esso

---

- ?- lecturer(L, logic).
  - Chi insegna logica?
- ?- duration(C, 2).
  - Quale corso dura due ore?
- ?- occupied(r204, wednesday, Time).
  - A che ore è occupato r204 di mercoledì?

# Esempio di esecuzione



```
SWI-Prolog -- g:/WORK/LESSONS/CORSI/Sistemi basati su conoscenza/2004-2005/esempio_datab...
File Edit Settings Run Debug Help
% g:/WORK/LESSONS/CORSI/Sistemi basati su conoscenza/2004-2005/esempio_database.pl c
Compiled 0.00 sec, 2,120 bytes
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.6)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- lecturer(L,logic).
L = dave ;

No
2 ?- duration(C,2).
C = logic ;
C = java ;

No
3 ?- occupied(r204, wednesday, Time).
No
4 ?- █
```

## 5. Regole Ricorsive

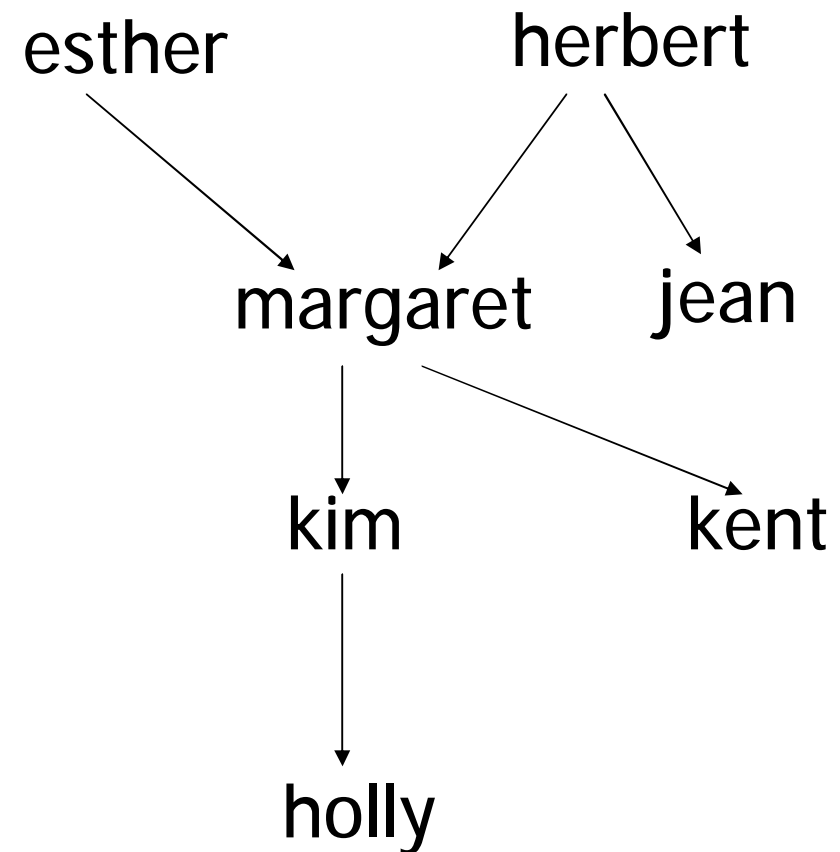
---

```
ancestor(X,Y) :- parent(X,Y).  
ancestor(X,Y) :-  
    parent(Z,Y),  
    ancestor(X,Z).
```

- **x** è un antenato di **y** se:
  - Caso base: **x** è genitore di **y**
  - Caso ricorsivo: esiste un **z** tale che **z** è genitore di **y**, e **x** è antenato di **z**
- Esplorazione delle regole
  - Prolog esplora le regole nell'ordine in cui gli sono presentate, per questo motivo è importante inserire le regole base e i fatti per primi.

# riprendiamo il grafo di parent

```
parent(kim,holly).  
parent(margaret,kim).  
parent(margaret,kent).  
parent(ester,margaret).  
parent(herbert,margaret).  
parent(herbert,jean).
```



# Query su ancestor/2

?- *ancestor(kim,holly)*.

**Yes**

?- *ancestor(A,holly)*.

**A = kim ;**

**A = margaret ;**

**A = esther ;**

**A = herbert ;**

**No**

# Le trappole dell'algoritmo di risoluzione

- Riscriviamo le regole per inferire la relazione ancestor

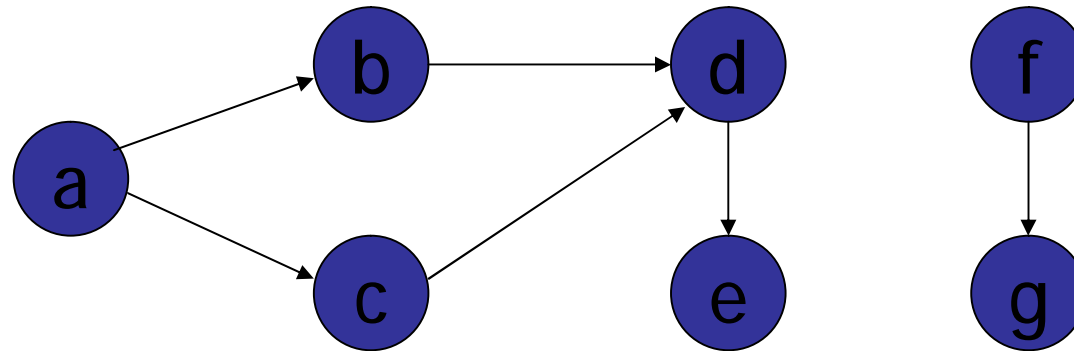
```
ancestor(X,Y):-  
    ancestor(X,Z),  
    parent(Z,Y).
```

```
ancestor(X,Y):-  
    parent(X,Y).
```

- cosa succede?



# Path Searching

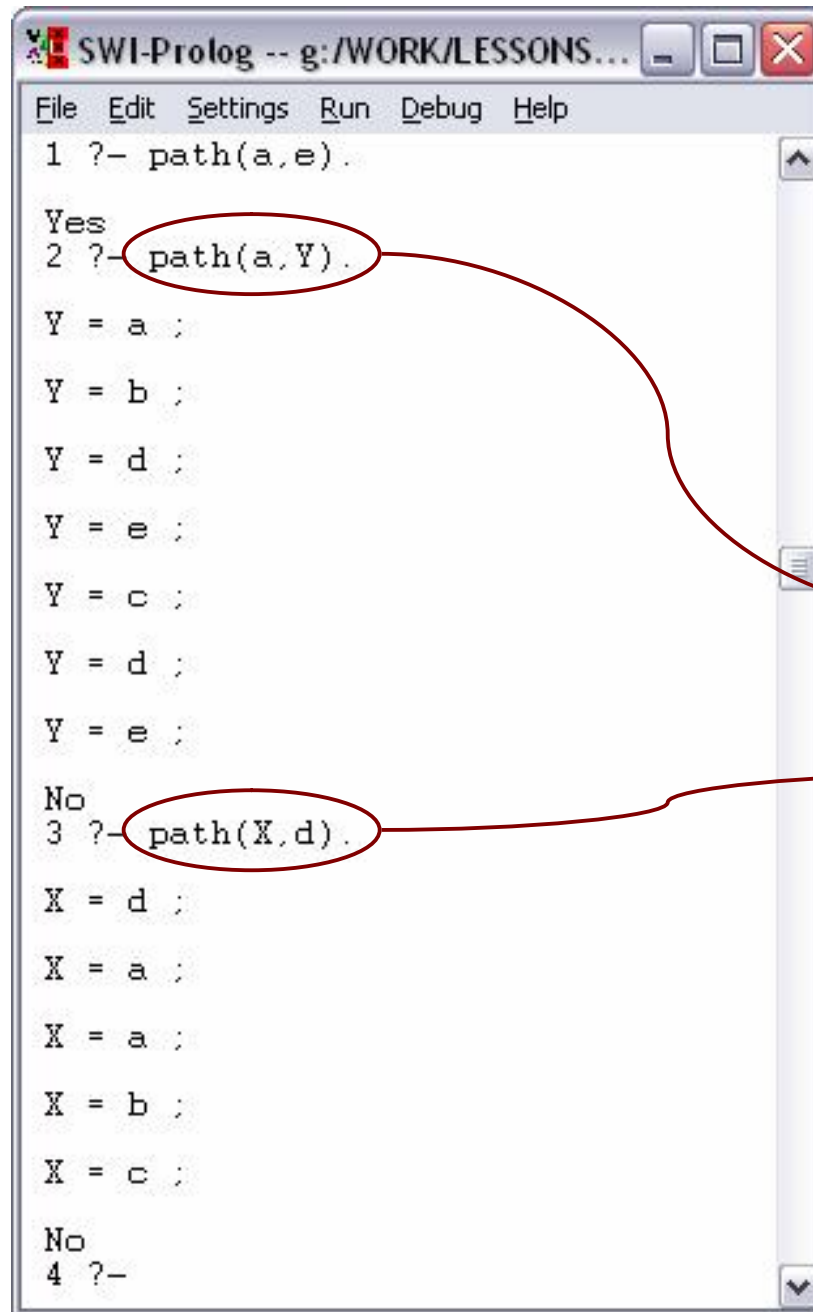


edge(a,b).  
edge(a,c).  
edge(b,d).  
edge(c,d).  
edge(d,e).  
edge(f,g).

path(Node,Node).  
path(Node1,Node3) :-  
 edge(Node1,Node2),  
 path(Node2,Node3).

# Alcune Query

- **?- path(a, e) .**
  - Esiste un path da a a e?
- **?- path(a, Y) .**
  - Quale nodo può essere raggiunto da a?
- **?- path(X, d) .**
  - Quale nodo ha un path verso d?



```
SWI-Prolog -- g:/WORK/LESSONS...
File Edit Settings Run Debug Help
1 ?- path(a,e).
Yes
2 ?- path(a,Y).
Y = a ;
Y = b ;
Y = d ;
Y = e ;
Y = c ;
Y = d ;
Y = e ;
No
3 ?- path(X,d).
X = d ;
X = a ;
X = a ;
X = b ;
X = c ;
No
4 ?-
```

vengono fornite  
risposte multiple  
quando uno dei due  
argomenti è lasciato  
non istanziato

# Come prevenire il backtracking: la Cut

- La Cut è utilizzato per eseguire un “taglio” sull’albero di esplorazione delle possibili soluzioni, prevenendo così il backtracking
  - È rappresentata dal predicato `! / 0`
- L’effetto di una cut può essere sintetizzato in:
  - Viene tagliata la regola entro la quale è espressa la cut
  - Vengono tagliati tutti i predicati precedenti la cut.
- Esempi dal manuale SWI-Prolog

```
t1 :- (a, !, fail ; b).           % cuts a/0 and t1/0
t2 :- (a -> b, ! ; c).           % cuts b/0 and t2/0
t3 :- call((a, !, fail ; b)).    % cuts a/0
t4 :- \+(a, !, fail ; b).       % cuts a/0
```

# Esempio di prevenzione del backtracking

```
buy_car(Model,Color):-  
    car(Model,Color,Price),  
    check_color(Color,sexy),  
    Price < 25000,  
    !,  
    go_buy( ).
```

A questo punto siamo  
soddisfatti della soluzione  
ottenuta

# Alcuni consigli per disporre opportunamente le Cut

---

- Disporle il più “a sinistra” possibile
- Disporre solo quelle strettamente necessarie
- Se siete in dubbio, eseguite delle prove per trovare la locazione giusta!
- Una cut alla fine di una clausola rende inutile la last-call optimization\*
- Ancora una volta...attenzione a dove mettete le cut!!!

## 6. Operatori

---

- I sistemi Prolog contengono diversi operatori built-in, come:

`=/2`

`is/2`

`+/2`

- Notazione
  - Un operatore è equivalente ad un predicato, del quale si fornisce una sintassi a notazione infissa:

`=(2,3)` è equivalente a `2 = 3`.

`is(A,3)` è equivalente a `A is 3`.

## Il Predicato = / 2

- Il goal  $= (x, y)$  ha successo se  $x$  e  $y$  possono essere *unificati*
  - “unificare” significa creare dei legami (binding) tra le variabili
- È tipicamente rappresentato come  $X = Y$

```
?- name(adam, seth) = name(adam, X).  
X = seth  
Yes
```



# Operatori Aritmetici

- I termini  $+$ ,  $-$ ,  $*$  e  $/$  sono operatori, con la precedenza e associatività universalmente adottate in algebra

?-  $X = +(1, *(2, 3))$ .

**X = 1+2\*3**

**Yes**

?-  $X = 1+2*3$ .

**X = 1+2\*3**

**Yes**

Stranamente la formula non è stata computata come una qualsiasi operazione matematica

```
?- X is 1+2*3.
```

```
X = 7
```

```
yes
```

- Sintassi:  
Variabile **is** termine\_che\_rappresenta\_l'operazione

## 7. Liste

Notazione a lista	Termine effettivo
[ ]	[ ]
[ 1 ]	. ( 1 , [ ] )
[ 1 , 2 , 3 ]	. ( 1 , . ( 2 , . ( 3 , [ ] ) ) )
[ 1 , name ( X , Y ) ]	. ( 1 , . ( name ( X , Y ) , [ ] ) )

- L'atomo [ ] rappresenta la lista vuota.

# Esempi

```
?- X = .(1,.(2,.(3,[ ]))) .  
X = [1, 2, 3]  
Yes
```

```
?- [X, Y, Z] = [1, 2, 3] .  
X = 1  
Y = 2  
Z = 3  
Yes
```

# Coda di una lista

```
?- [1,2|X] = [1,2,3,4,5].  
X = [3, 4, 5]  
Yes
```

- $[1, 2 | x]$  unifica con una lista che inizia con 1, 2 , legando  $x$  alla coda 3, 4, 5.
- Attenzione, la coda è anch'essa una lista, non un semplice elenco di variabili!

## 8. Il predicato `append` / 3

```
append([], B, B).  
append([Head|TailA], B, [Head|TailC]) :-  
    append(TailA, B, TailC).
```

```
?- append([1,2],[3,4],Z).  
Z = [1, 2, 3, 4]  
Yes
```

- `append(X,Y,Z)` ha successo quando `Z` unifica con la lista `Y` “appesa” alla fine della lista `X`.

# Altri usi di `append/3`

```
?- append(X,[3,4],[1,2,3,4]).  
X = [1, 2]  
Yes
```

- `append/3` può essere chiamato con delle variabili al posto di ognuno dei suoi argomenti.

# Risposte multiple

```
?- append(X,Y,[1,2,3]).
```

```
X = []
```

```
Y = [1, 2, 3] ;
```

```
X = [1]
```

```
Y = [2, 3] ;
```

```
X = [1, 2]
```

```
Y = [3] ;
```

```
X = [1, 2, 3]
```

```
Y = [] ;
```

```
No
```



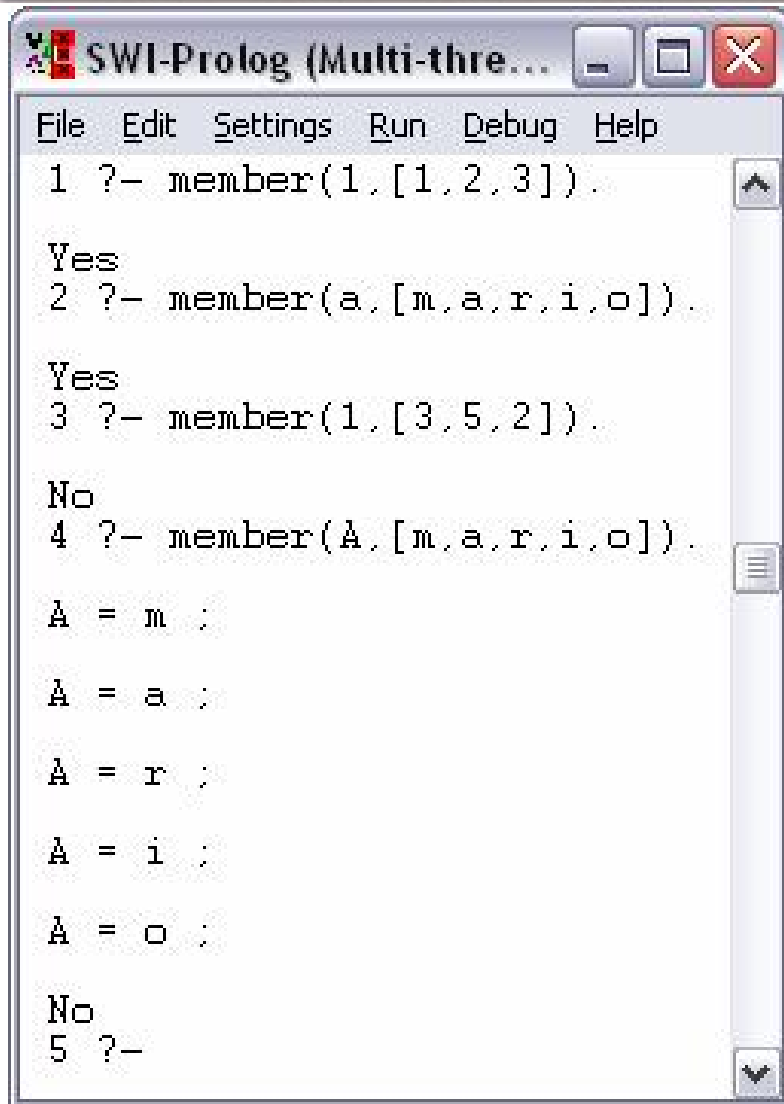
## 9. Altri predicati sulle liste

Predicato	Descrizione
<code>member(X, Y)</code>	Ha successo se <b>X</b> è un elemento della lista <b>Y</b> .
<code>length(X, Y)</code>	Ha successo se la lista <b>X</b> ha lunghezza <b>Y</b> .

- Ovviamente i due predicati sono flessibili, allo stesso modo di **append/3**
  - Le query possono contenere variabili in ognuno dei loro argomenti

## member / 2

```
member(X, [X|_]).  
member(X, [Y|Rest]) :-  
    member(X, Rest).
```



```
SWI-Prolog (Multi-thre...  
File Edit Settings Run Debug Help  
1 ?- member(1,[1,2,3]).  
Yes  
2 ?- member(a,[m,a,r,i,o]).  
Yes  
3 ?- member(1,[3,5,2]).  
No  
4 ?- member(A,[m,a,r,i,o]).  
A = m ;  
A = a ;  
A = r ;  
A = i ;  
A = o ;  
No  
5 ?-
```

- `member(X,L)` ha successo se `X` è un elemento della lista `L`

## length/2

```
length([], 0).  
length([X|Rest], Len) :-  
    length(Rest, LenRest),  
    Len is LenRest + 1.
```

```
?- length([a,b,c,d], L).  
  
L = 4  
  
yes  
?- length([1,2,3], 4).  
no  
?-
```

- `length(X,Y)` ha successo se `Y` è la lunghezza della lista `X`.

## 10. Insertion Sort

```
?- isort([4, 3, 1, 5], S).  
S = [1, 3, 4, 5]  
yes
```

```
% isort(A,B): B è una versione ordinata di A  
isort([X|Xs],Ys) :- isort(Xs,Zs), insert(X,Zs,Ys).  
isort([ ],[ ]).
```

```
% insert(A,B,C)  
% se B è una lista ordinata, allora C è ordinata  
% e contiene tutti gli elementi in B più A  
insert(X,[ ],[X]).  
insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).  
insert(X,[Y|Ys],[X,Y|Ys]) :- X <= Y.
```

# 11. Il Predicato `not/1`

```
?- not( member( 4, [ 1, 2, 3 ] ) ).
```

**Yes**

```
?- not( member( 1, [ 1, 2, 3 ] ) ).
```

**No**

- Il predicato built-in `not(x)` ha successo solo se `x` fallisce.
- Usate `not/1` solo quando il goal non contiene variabili. (la sua esecuzione è pesante, in quanto per avere successo, deve fallire esaustivamente in tutti i punti di scelta dell'albero di dimostrazione del predicato presente al suo interno)

# Esempio sul db della famiglia

```
sibling(X,Y) :-  
    not(X=Y),  
    parent(P,X),  
    parent(P,Y).
```

```
?- sibling(kim,kent).  
Yes
```

```
?- sibling(kim,kim).  
No
```

- `findall/3`, `bagof/3`, e `setof/3` sono considerati *meta-predicati*

- prendono un altro predicato per argomento, restituendone diverse possibili soluzioni.

<code>findall(X,P,L)</code> <code>bagof(X,P,L)</code> <code>setof(X,P,L)</code>	}	producono una lista L di oggetti X tali che il goal P è soddisfatto
---	---	---

- Tutti e tre i predicati chiamano ripetutamente il goal P, istanziando la variabile X che è presente in P and adding it to the list L.
- Chiudono con successo quando non ci sono più soluzioni per P.
- Si comportano di fatto come una ripetuta pressione del tasto ';' da shell, a seguito di una risposta.

# findall/3

- **findall/3** è il più immediato dei tre, e quello usato più comunemente:

```
| ?- forall(X, member(X, [1,2,3,4]), Results).  
    Results = [1,2,3,4]  
yes
```

- Esplicitamente: `trova tutte le X tali che X è un membro della lista [1, 2, 3, 4] e inserisci la lista dei risultati in Results'.
- Le soluzioni sono inserite nella lista nello stesso ordine in cui sono trovate dal sistema Prolog
- Se esistono delle soluzioni coincidenti (duplicati), queste vengono incluse. Se esiste un numero infinito di soluzioni, il predicato non avrà mai termine!



## findall/3 (2)

- Possiamo utilizzare `findall/3` in modi più sofisticati.
- Il secondo argomento, cioè il goal, può essere rappresentato da un termine composto:

```
| ?- findall(X, (member(X, [1,2,3,4]), X > 2), Results).  
    Results = [3,4]  
    yes
```

- Anche il primo argomento può essere un termine composto:

```
| ?- findall(X/Y, (member(X, [1,2,3,4]), Y is X * X),  
    Results).  
    Results = [1/1, 2/4, 3/9, 4/16]  
    yes
```

- **bagof/3** è molto simile a **findall/3**, con una differenza:
  - **bagof/3** restituirà un risultato separato per ogni possibile istanziazione di ogni variabile utilizzata nel goal che non appare al contempo nel primo argomento; es. basato sul database della famiglia:

```
| ?- bagof(Son, parent(Par,Son),Sons).  
    Par = kim, Sons = [holly]           ;  
    Par = margaret, Sons = [kim,kent]   ;  
    Par = esther, Sons = [margaret]     ;  
    Par = herbert, Sons = [margaret,jean] ;  
    no
```

- Risultato che avrebbe fornito **findall/3**:

```
| ?- findall(Son, parent(Par,Son),Sons).  
    Sons = [holly,kim,kent,margaret,margaret,jean] ;  
    no
```

## bagof/3 (2)

- Possiamo effettuare delle chiamate *annidate* a `bagof/3` per ricollezionare tutti i risultati in un'unica lista:

```
| ?- bagof(Par/Sons, bagof(Son,parent(Par,Son),Sons),Families).  
    Families = [kim/[holly], margaret/[kim, kent],  
               esther/[margaret], herbert/[margaret, jean]]  
yes
```

- Se non vogliamo fattorizzare il risultato rispetto ad una variabile che non compare come primo argomento, possiamo “ignorarla” frapponendo il simbolo ‘^’ tra essa e il predicato a secondo argomento:

```
| ?- bagof(Son,Par^parent(Par,Son),Sons).  
    Sons = [holly, kim, kent, margaret, margaret, jean] ;  
no
```

- Come possiamo osservare, il risultato in questo caso coincide con la findall. `findall/3` è in effetti considerata una `bagof/3` con tutte le sue variabili libere ignorate.

- **setof/3** è del tutto simile a **bagof/3**, tranne che:
  - Produce un *insieme ordinato* dei risultati.
- In SWI-Prolog la sua implementazione utilizza in effetti i due predicati **bagof/3** e **sort/2**.
  - **bagof/3** produce la “bag”
  - **sort/2** elimina i duplicati e ordina la lista dei risultati
- Esempio di uso di **setof/3**, ancora sul db della famiglia:

```
| ?- setof(Son, Par^parent(Par, Son), Sons).  
    Sons = [holly, jean, kent, kim, margaret] ;  
    no
```

# Problema delle N-Regine

---

- Vedere il file prolog 'n\_queens.pl' pubblicato assieme agli altri esercizi J

- Sito Web di SWI Prolog:
  - <http://www.swi-prolog.org/>
- Libri sul Prolog (in formato PDF):
  - <http://www.ida.liu.se/~ulfni/lpp/>
- Tutorial sul Prolog:
  - [http://www.csupomona.edu/~jrffisher/www/prolog\\_tutorial/pt\\_framer.html](http://www.csupomona.edu/~jrffisher/www/prolog_tutorial/pt_framer.html)