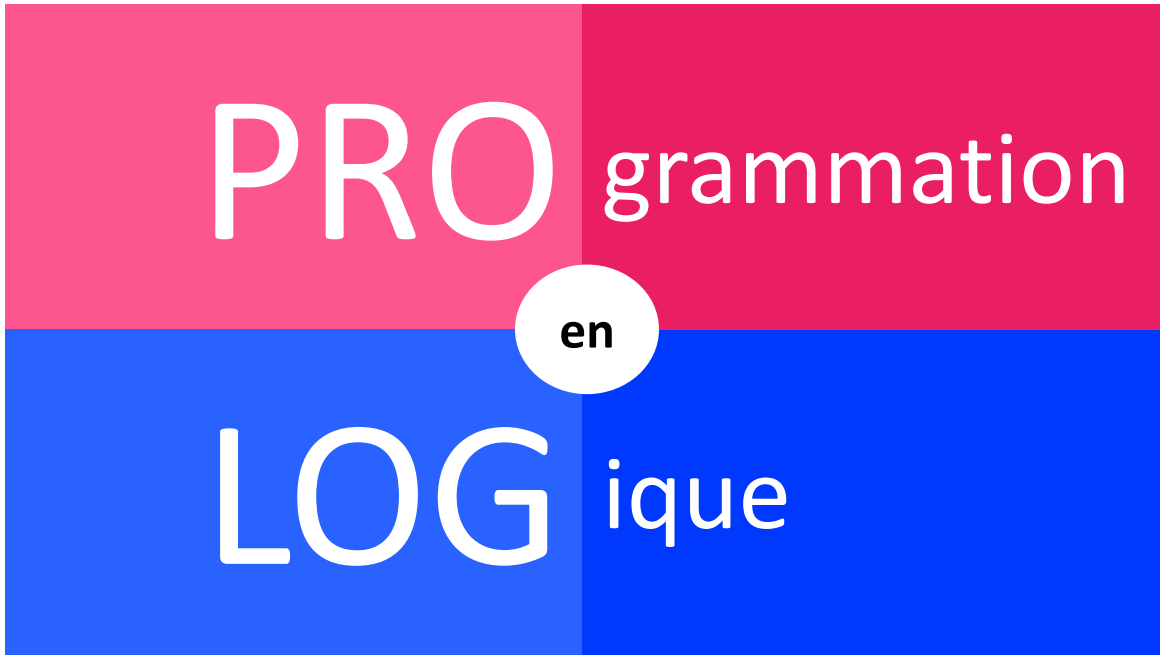
UNIVERSITÀ DEGLI STUDI DI PARMA
DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

Prolog

Prolog

- è un linguaggio orientato alla programmazione logica



Prolog

- è un linguaggio orientato alla programmazione logica

- è stato progettato ed implementato a Marsiglia da Colmerauer e Roussel nel 1972

Prolog

- è un linguaggio orientato alla programmazione logica
- è stato progettato ed implementato a Marsiglia da Colmerauer e Roussel nel 1972
- è utilizzato principalmente in ambito accademico, nel settore dell'intelligenza artificiale e nel settore della linguistica computazionale

Prolog

- è un linguaggio orientato alla programmazione logica
- è stato progettato ed implementato a Marsiglia da Colmerauer e Roussel nel 1972
- è utilizzato principalmente in ambito accademico, nel settore dell'intelligenza artificiale e nel settore della linguistica computazionale
- è un linguaggio di programmazione logica basato sulle clausole di Horn

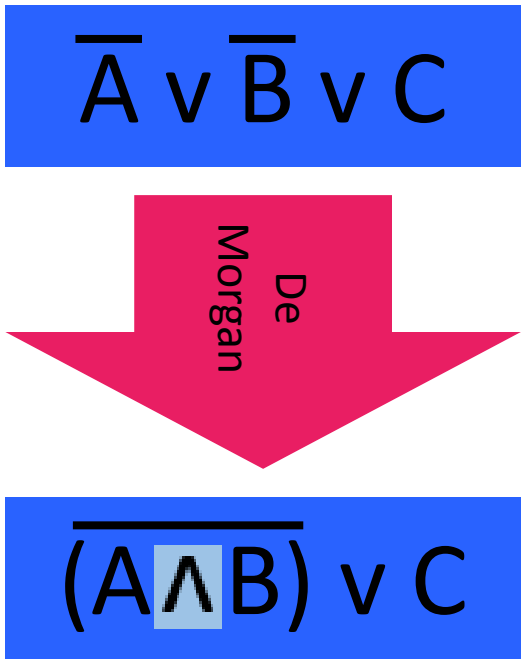
Le clausole di Horn

- è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo

$$\overline{A} \vee \overline{B} \vee C$$

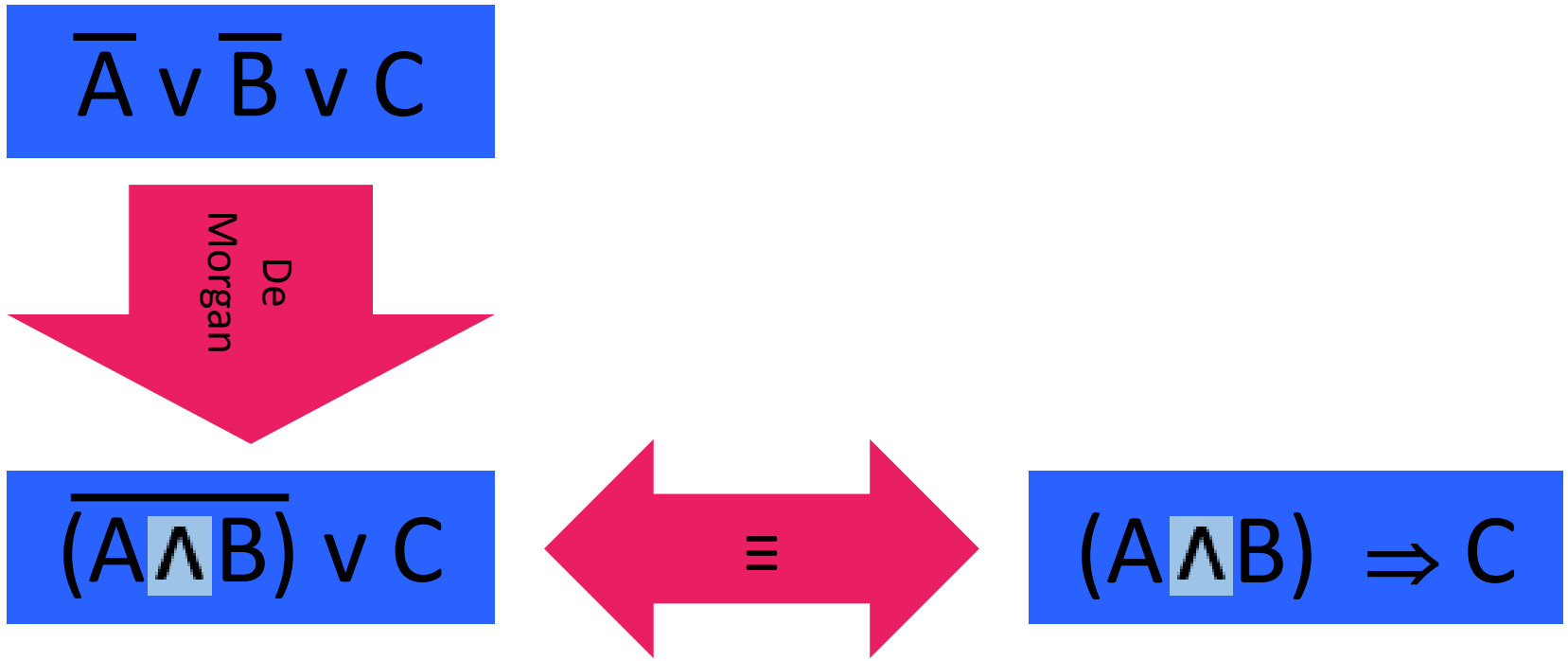
Le clausole di Horn

- è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo



Le clausole di Horn

- è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo



Le clausole di Horn

- è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo

$\overline{A} \vee \overline{B} \vee C$

De Morgan

$\overline{(A \wedge B)} \vee C$ $\equiv$ $(A \wedge B) \Rightarrow C$

$A \wedge B$	C	$!(A \wedge B)$	$!(A \wedge B) \vee C$	$(A \wedge B) \Rightarrow C$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

Le clausole di Horn

- è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo

$\overline{A} \vee \overline{B} \vee C$

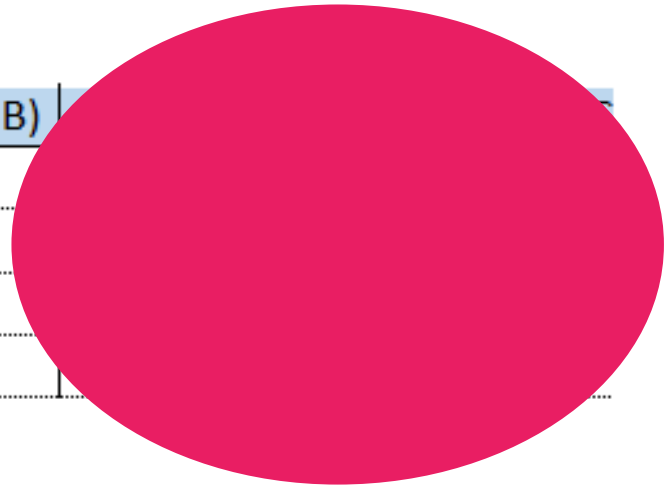
De Morgan

$\overline{(A \wedge B)} \vee C$

$\equiv$

$(A \wedge B) \Rightarrow C$

A ∧ B	C	! (A ∧ B)
0	0	1
0	1	1
1	0	0
1	1	0



Implementazioni

JIProlog

SWI-Prolog

XSB

Visual Prolog

GNU Prolog

YAP Prolog

SICStus Prolog

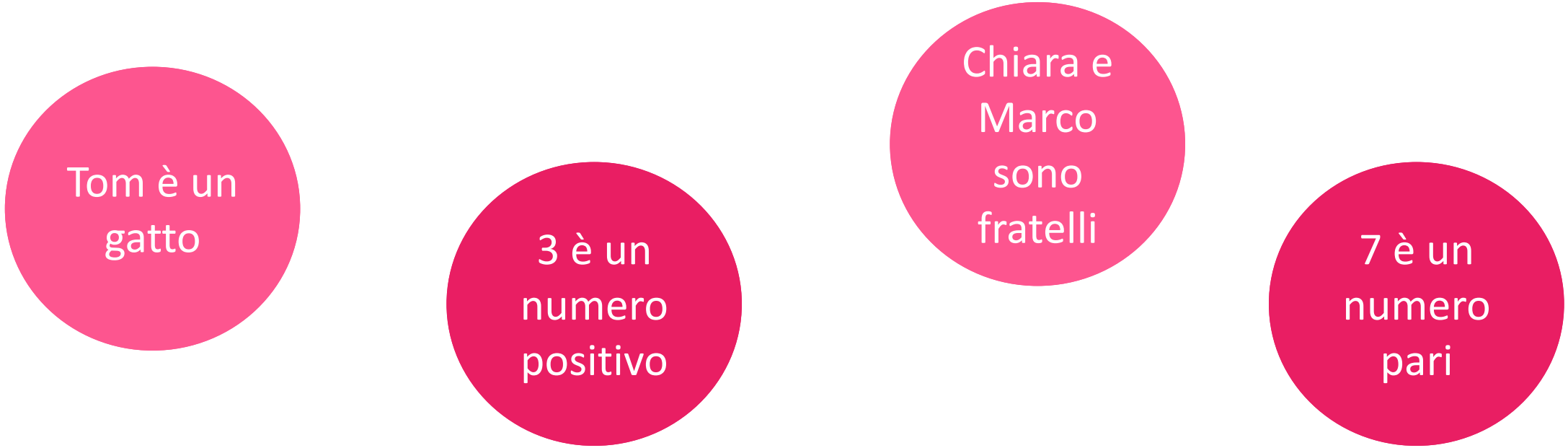
...

Caratteristiche principali

- un programma Prolog è costituito da clausole che possono essere fatti o regole

Caratteristiche principali

- un programma Prolog è costituito da clausole che possono essere fatti o regole
  - un fatto esprime qualcosa che è vero



Caratteristiche principali

- un programma Prolog è costituito da clausole che possono essere fatti o regole
  - un fatto esprime qualcosa che è vero
  - una regola esprime un modo per inferire nuovi fatti

X>0 $\Rightarrow$ X è un numero positivo

Anna e Luca sono i genitori di Chiara e Marco $\Rightarrow$ Chiara e Marco sono fratelli

Caratteristiche principali

- un programma Prolog è costituito da clausole che possono essere fatti o regole
  - un fatto esprime qualcosa che è vero
  - una regola esprime un modo per inferire nuovi fatti
- i programmi vengono interpretati attraverso un motore di inferenza che opera attraverso l'unificazione e il backtracking

Caratteristiche principali

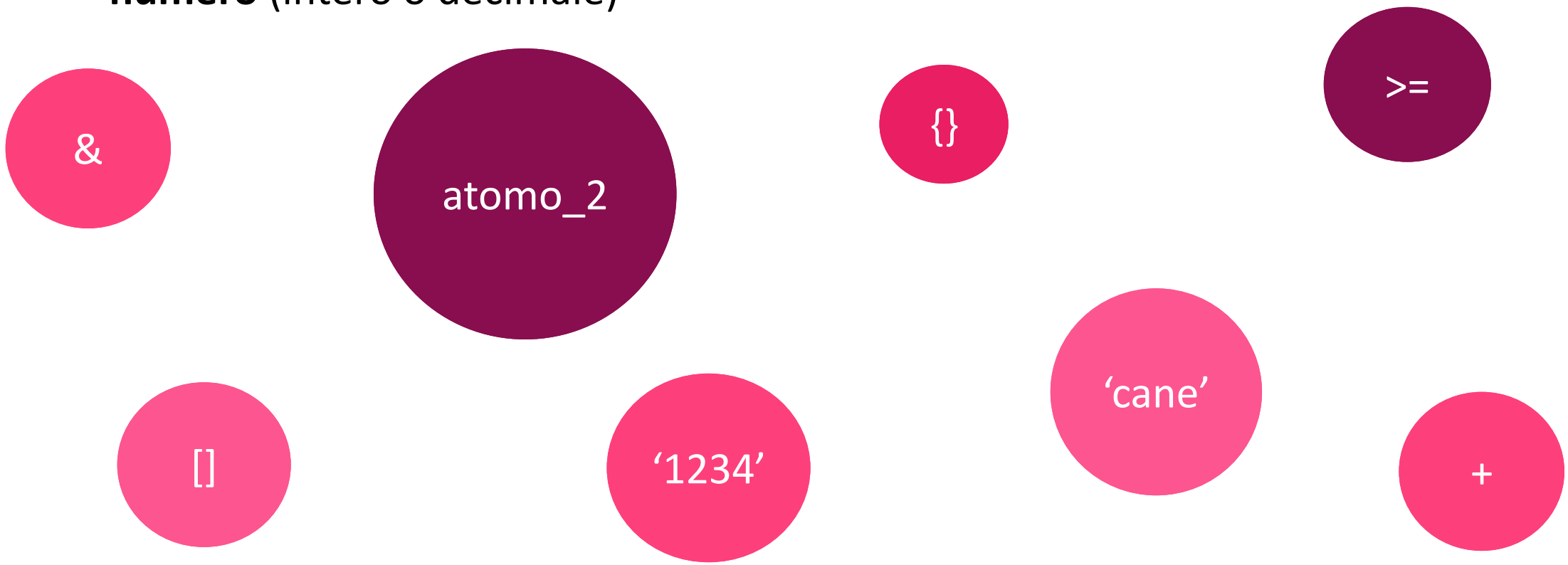
- un programma Prolog è costituito da clausole che possono essere fatti o regole
  - un fatto esprime qualcosa che è vero
  - una regola esprime un modo per inferire nuovi fatti
- i programmi vengono interpretati attraverso un motore di inferenza che opera attraverso l'unificazione e il backtracking
- lo sviluppo di un programma si basa su due attività:
  - dichiarare le clausole descriventi il problema
  - porre delle domande per risolvere il problema

Tipi di dati (termini)

- costante
  - **atomo:** è un nome generico senza significato intrinseco
  - **numero** (intero o decimale)

Tipi di dati (termini)

- costante
  - **atomo:** è un nome generico senza significato intrinseco
  - **numero** (intero o decimale)



Tipi di dati (termini)

- costante
  - **atomo:** è un nome generico senza significato intrinseco
  - **numero** (intero o decimale).

- **variabile**: è indicata per mezzo di una stringa di lettere, numeri e underscore. Il suo identificativo deve iniziare con una lettera maiuscola o con un underscore

Tipi di dati (termini)

- costante
  - **atomo:** è un nome generico senza significato intrinseco
  - **numero** (intero o decimale).

- **variabile**: è indicata per mezzo di una stringa di lettere, numeri e underscore. Il suo identificativo deve iniziare con una lettera maiuscola o con un underscore

- **termine composto**: è formato da un atomo detto "funtore" e da uno o più argomenti - anch'essi termini - scritti tra parentesi e separati da virgole

parent(tom,sara)

date('2017','06','06)

point(1.0,3.5)

Tipi di dati (termini)

- **lista**:
  - è una collezione ordinata di termini, separati da virgole;
  - viene indicata per mezzo di parentesi quadre;
  - è ammessa la lista vuota.

`[]`

`[a, b, c, d]`

`[124, 11, 2]`

Tipi di dati (termini)

- **lista**:
  - è una collezione ordinata di termini, separati da virgole;
  - viene indicata per mezzo di parentesi quadre;
  - è ammessa la lista vuota.
- **stringa**: è una sequenza di caratteri delimitata da doppi apici.

"questa è una stringa"

Fatti

- i fatti sono le parti elementari della base di conoscenza usata dal Prolog.

Fatti

- i fatti sono le parti elementari della base di conoscenza usata dal Prolog.

- la forma di un fatto è una struttura Prolog della forma:

```
fatto(argomento0,…,argomentoN).
```

dove:

- ogni termine è una costante
- gli argomenti sono separati da virgole
- il fatto è terminato da un punto

Fatti

Esempio «parents.pl» :

```
woman (giulia).
woman (emma).
man(luca).
man(davide).
man(lorenzo).
parent (emma, luca).
parent (lorenzo, luca).
parent (emma, giulia).
parent (davide, giulia).
```

Regole

- le regole permettono di inferire della nuova conoscenza dalla base di conoscenza del Prolog
- una regola Prolog è una struttura dalla forma:

- la testa è la conoscenza inferita, il corpo è la conoscenza da cui inferire. Se il corpo è vero, viene inferita la testa

Regole

Esempio «parents.pl» :

```
father(X,Y) :- man(X),parent(X,Y).
mother(X,Y):-woman(X),parent(X,Y).
sibling(X,Y):-mother(Z,X),mother(Z,Y),father(K,X),father(K,Y).
```

Domande

- una domanda permette di chiedere se l'informazione proposta (goal) può essere o non essere ricavata dal database

?- woman (giulia).
true.

?- woman (anna).
false.

?- father(lorenzo, luca).
true.

Domande

- una domanda può chiedere di completare l'informazione proposta con il database

?- woman (X).
X = giulia;
X = emma

?- father(lorenzo, X).
X = giulia;
X = luca

Domande

- una domanda permette di chiedere se l'informazione proposta (goal) può essere o non essere ricavata dal database

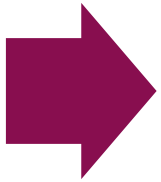
?- dog(_).
false.

?- father(lorenzo, _).
true

L’interprete

- l’interprete opera in due fasi:

Memorizza le clausole che definiscono la conoscenza sul problema (database)

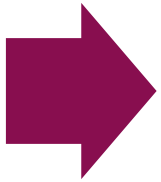


Risponde alle domande che permettono di risolvere il problema

L’interprete

- l’interprete opera in due fasi:

Memorizza le clausole che definiscono la conoscenza sul problema (database)



Risponde alle domande che permettono di risolvere il problema

UNIFICAZIONE

BACKTRACKING

Unificazione

due termini, S e T, unificano se:

- sono identici
- le variabili in essi possono essere istanziate con oggetti in modo che dopo la sostituzione delle variabili i termini diventano identici

Unificazione

due termini, S e T, unificano se:

- sono identici
- le variabili in essi possono essere istanziate con oggetti in modo che dopo la sostituzione delle variabili i termini diventano identici

in particolare:

- se S e T sono **costanti**, S e T unificano solo se sono lo stesso oggetto.
- se S è una **variabile** e T è qualsiasi cosa, allora S e T unificano e S viene istanziata a T
- se S e T sono **strutture**, allora S e T unificano solo se
  - S e T hanno lo stesso funtore principale
  - tutte le componenti corrispondenti unificano

Unificazione

Esempio:

```
g(a,b).
g(s,t).
g(f,v).
```

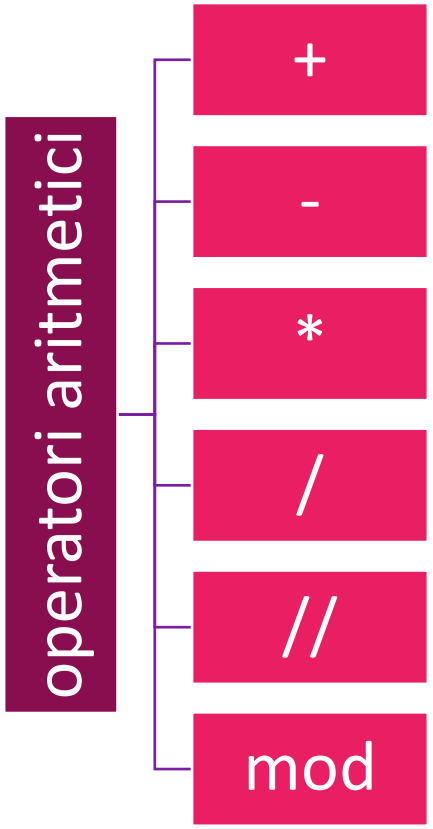
```
?- g(a,b).
true.

?- g(f,X).
X=v
true.

?- g(b,X).
false.
```

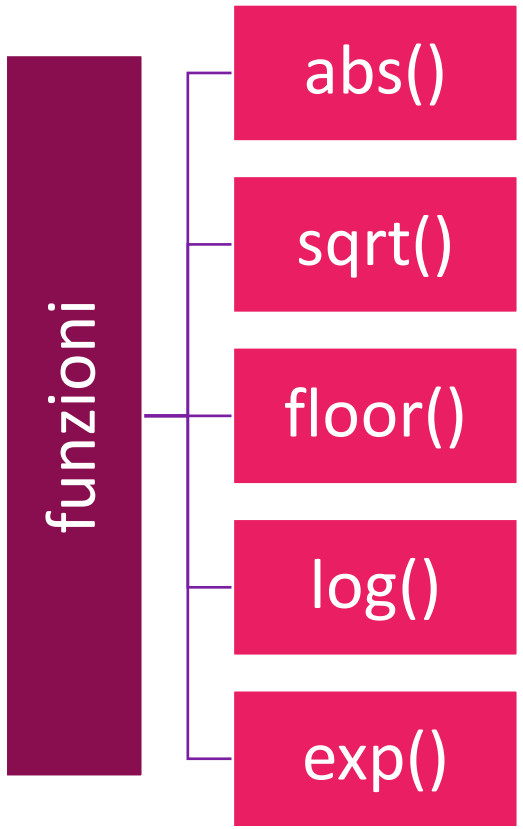
Backtracking

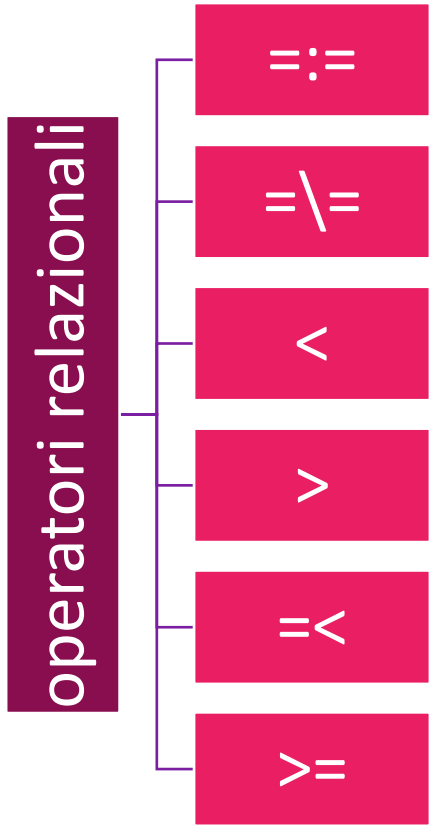
- per unificare i termini di un domanda, l'interprete Prolog dovrà scandire il database cercando di applicare regole e unificare fatti
- se più regole possono essere unificate con una domanda, l'interprete prova le regole nell'ordine in cui le trova nel database
- se una regola non ha successo, torna indietro e applica una nuova regola

Operazioni aritmetiche

- Prolog mette a disposizione:







Operazioni aritmetiche

Esempio: calcolo dei risultati di operazioni aritmetiche

? X is 3+4.
X = 7.

?- X is (2-1)*5.
X = 5.

?- X is 7/2.
X = 3.5.

?- X is 7//2.
X = 3.

?- X is 7 mod 2.
X = 1.

Operazioni aritmetiche

Esempio: relazioni tra espressioni aritmetiche.

? 3 is 3+4.
false.

? 7 is 3+4.
true.

?- 2+5 is 3+4.
false.

?- 2+5 =:= 3+4.
true.

Operazioni aritmetiche

Esempio: verifica dell'arcoseno di un numero reale

```
is_arcsin(X, nonExistent) :- X <(-1.0); X>1.0.
is_arcsin(X,Y) :- X =<(1.0), X>=(-(1.0)), Y is asin(X).
```

```
?- is_arcsin(2.0, X).
X = nonExistent .

?- is_arcsin(1.0, X).
X = 1.5707963267948966.

?- is_arcsin(1.0, 1.5707963267948966).
true.
```

```
?- is_arcsin(Y, 1.5707963267948966).
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [9] _7012=<1.0
ERROR:    [8] is_arcsin(_7038, 1.5707963267948966) at
c:/users/ids/documents/prolog/asin.pl:2
ERROR:    [7] <user>
```

Liste

- una lista è delimitata dalle parentesi quadre [ ] e gli elementi al suo interno sono separati da virgola.

```
[a,b,c,d,e,f]
```

Liste

- una lista è delimitata dalle parentesi quadre [ ] e gli elementi al suo interno sono separati da virgola.

- una lista ha una testa e una coda (o resto) e queste due parti possono essere divise dal simbolo «**|**».

[a | b,c,d,e,f]

? [Head | Tail] = [a,b,c,d,e,f].
Head = a,
Tail = [b, c, d, e, f].

?- [First, Second | Tail] = [a,b,c,d,e,f].
First = a,
Second = b,
Tail = [c, d, e, f].

Liste

Esempio: verificare che un elemento X sia contenuto in una lista

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

```
?- member(1,[a,t,5,1,g]).
true .
```

```
[trace]  ?- member(1,[3,1,4]).
   Call: (8) member(1, [3, 1, 4]) ? creep
   Call: (9) member(1, [1, 4]) ? creep
   Exit: (9) member(1, [1, 4]) ? creep
   Exit: (8) member(1, [3, 1, 4]) ? creep
true .


[trace]  ?- member(2,[3,1]).
   Call: (8) member(2, [3, 1]) ? creep
   Call: (9) member(2, [1]) ? creep
   Call: (11) member(2, []) ? creep
   Fail: (11) member(2, []) ? creep
   Fail: (9) member(2, [1]) ? creep
   Fail: (8) member(2, [3, 1]) ? creep
false.
```

Liste

Esempio: calcolo della lunghezza di una lista

```
list_length([], 0).
list_length([_ | Tail], N) :- list_length(Tail, J), N is J + 1.
```

```
?- list_length([1,2,4,5,s,3,s],L).
L=7.
```

```
?- list_length([1,2],L).

 Call: (8) list_length([1, 2], _8688) ? creep
  Call: (9) list_length([2], _9344) ? creep
  Call: (10) list_length([], _9344) ? creep
  Exit: (10) list_length([], 0) ? creep
  Call: (10) _9348 is 0+1 ? creep
  Exit: (10) 1 is 0+1 ? creep
  Exit: (9) list_length([2], 1) ? creep
  Call: (9) _8688 is 1+1 ? creep
  Exit: (9) 2 is 1+1 ? creep
  Exit: (8) list_length([1, 2], 2) ? creep
L = 2.
```

Liste

Esempio: concatenazione di due liste

```
append([], X, X).
append([[Head | Tail], Y, [Head | Z]) :- append(Tail, Y, Z).
```

```
?- append([a,b,c],[1,2,3],X) .
   Call: (8) append([a, b, c], [1, 2, 3], _19988) ? creep
   Call: (9) append([b, c], [1, 2, 3], _20234) ? creep
   Call: (10) append([c], [1, 2, 3], _20240) ? creep
   Call: (11) append([], [1, 2, 3], _20246) ? creep
   Exit: (11) append([], [1, 2, 3], [1, 2, 3]) ? creep
   Exit: (10) append([c], [1, 2, 3], [c, 1, 2, 3]) ? creep
   Exit: (9) append([b, c], [1, 2, 3], [b, c, 1, 2, 3]) ? creep
   Exit: (8) append([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3]) ? creep
X = [a, b, c, 1, 2, 3].
```

Liste

Esempio: rimozione di un elemento da una lista

```
extract(X, [X | Tail], Tail).
extract(X, [H | Tail], [H | NewTail]) :- extract(X,Tail, NewTail).
```

```
?- extract(8, [1,2,3,s,c,4,8,ds,2], L).
L = [1, 2, 3, s, c, 4, ds, 2] .
```

```
?- extract(2, [1,2], L).
 Call: (8) extract(2, [1, 2], _17774) ? creep
   Call: (9) extract(2, [2], _18432) ? creep
   Exit: (9) extract(2, [2], []) ? creep
   Exit: (8) extract(2, [1, 2], [1]) ? creep
L = [1] .
```

Liste

Esempio: inversione degli elementi di una lista

```
reverse([], []).
reverse([Head|Tail], Y)   :-  reverse(Tail, ReversedTail),
                              append(ReversedTail, [Head], Y).
```

```
?- reverse([1,2,3,4,5,6,7,8], ReversedList).
ReversedList = [8, 7, 6, 5, 4, 3, 2, 1].
```

Definizione di operatori

- Prolog consente la definizione di operatori. Per ogni operatore definito, è necessario specificare:
  - nome
  - precedenza
  - posizione e associatività

Definizione di operatori

- Prolog consente la definizione di operatori. Per ogni operatore definito, è necessario specificare:
  - nome
  - precedenza
  - posizione e associatività

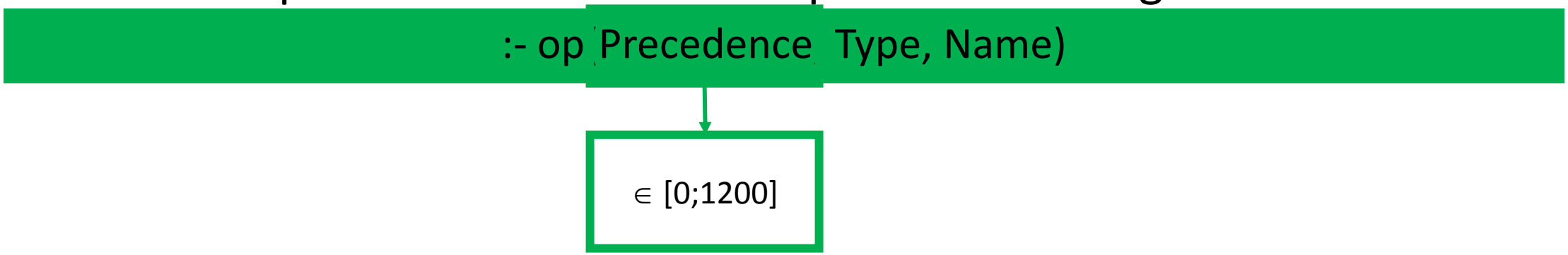
- La sintassi per la definizione di un operatore è la seguente:

```
:- op(Precedence, Type, Name)
```

Definizione di operatori

- Prolog consente la definizione di operatori. Per ogni operatore definito, è necessario specificare:
  - nome
  - precedenza
  - posizione e associatività

- La sintassi per la definizione di un operatore è la seguente:



Definizione di operatori

fx	prefisso e non associativo
fy	prefisso e associativo a destra
xf	postfisso non associativo
yf	postfisso e associativo a sinistra
xfx	infisso e non associativo
xfy	infisso e associativo a destra
yfx	infisso e associativo a sinistra

Definizione di operatori

Esempio:

```
:- op(500, xf, is_a_man).
is_a_man(luca).
is_a_man(lorenzo).
```

```
?- luca is_a_man.
true.
```

Definizione di operatori

Esempio:

```
:- op(500, xfy, is_sister_of).

woman(giulia).    woman(emma).   woman(anna).
man(luca).   man(marco).
parent(luca,emma).   parent(luca,giulia).
parent(luca,marco).   parent(anna,emma).
parent(anna,giulia).    parent(anna,marco).

father(A,B)  :-      man(A), parent(A,B).
mother(A,B)  :-     woman(A), parent(A,B).

is_sister_of(A, B)  :-          woman(A),
                                father(F,A), father(F, B),
                                mother(M,A), mother(M,B).
```

?- giulia is_sister_of emma.
true .

?- emma is_sister_of marco.
true .

?- marco is_sister_of emma.
false.

?- emma is_sister_of anna.
false.

L’operatore «cut»

- il backtracking automatico è una della caratteristiche principali del Prolog, ma, in determinate situazioni, Prolog potrebbe trovarsi ad esplorare possibilità che non portano da nessuna parte

- grazie all’operatore «cut», è possibile controllare il modo in cui Prolog esplora la base di conoscenza per la ricerca di soluzioni

- l’operatore «cut» si indica con un punto esclamativo «!»

L’operatore «cut»

- il backtracking automatico è una della caratteristiche principali del Prolog, ma, in determinate situazioni, Prolog potrebbe trovarsi ad esplorare soluzioni alternative che non portano da nessuna parte

- l’operatore cut permette di scartare delle soluzioni alternative non interessanti, che però sarebbero comunque controllate.

- l’operatore «cut» si indica con un punto esclamativo «!»

```
p(X) :-  b(X),  c(X),  !,  d(X),  e(X).
p(X) :- f(X), g(X).
```

L’operatore «cut»

Esempio: programma privo di cut

```
max_no_cut(X,Y,X) :- X >= Y.
max_no_cut(X,Y,Y) :- X<Y.
```

?- max_no_cut(1,2,Max).
Max = 2.

?- max_no_cut(2,2,Max).
Max = 2 .

?- max_no_cut(45,2,Max).
Max = 45 .

[trace] ?- max_no_cut(45,2,Max).
Call: (8) max_no_cut(45, 2, _21828) ? creep
Call: (9) 45>=2 ? creep
Exit: (9) 45>=2 ? creep
Exit: (8) max_no_cut(45, 2, 45) ? creep
Max = 45 ;
Redo: (8) max_no_cut(45, 2, _21828) ? creep
Call: (9) 45<2 ? creep
Fail: (9) 45<2 ? creep
Fail: (8) max_no_cut(45, 2, _21828) ? creep
false.

L'operatore «cut»

Esempio: programma con cut

```
max_no_cut(X,Y,X) :- X >= Y, !.
max_no_cut(X,Y,Y) :- X<Y.
```

?- max_no_cut(1,2,Max).
Max = 2.

?- max_no_cut(2,2,Max).
Max = 2 .

?- max_no_cut(45,2,Max).
Max = 45 .

[trace] ?- max_no_cut(45,2,Max).
Call: (8) max_no_cut(45, 2, _21828) ? creep
Call: (9) 45>=2 ? creep
Exit: (9) 45>=2 ? creep
Exit: (8) max_no_cut(45, 2, 45) ? creep
Max = 45.

If, Then, Else

- un modo alternativo per esprimere scelte deterministiche è il seguente:

condition -> then_clause ; else_clause

Esempio:

```
writename(X) :- (    X = 1 ->    write’(‘One’)
                ;    X = 2 ->    write’(‘Two’)
                ;    X = 3 ->    write’(‘Three’)
                ;    write’(‘Out of range’) ).
```

```
?- writename(1).
one
true.

?- writename(4).
Out of range
true.

?- writename(-4).
Out of range
true.
```

Goal sempre veri

- di seguito sono presentati alcuni modi per esprimere un goal sempre vero:

```
f(X, Y) :- X < Y, write('X is less than Y'), !.
f(_, _).
```

```
?- (f(A, B) ; true).
```

Goal sempre veri

- di seguito sono presentati alcuni modi per esprimere un goal sempre vero:

```
f(X, Y) :- X < Y, write('X is less than Y'), !.
f(_, _).
```

```
?- (f(A, B) ; true).
```

- per evitare di ottenere una doppia soluzione:

```
?- once(f(A, B) ; true).
```

Goal sempre veri

- di seguito sono presentati alcuni modi per esprimere un goal sempre vero:

```
f(X, Y) :- X < Y, write('X is less than Y'), !.
f(_, _).
```

```
?- (f(A, B) ; true).
```

- per evitare di ottenere una doppia soluzione:

```
?- once(f(A, B) ; true).
```

```
once(Goal) :- call(Goal), !.
```

Goal sempre falsi

- di seguito è mostrato un esempio di goal sempre falso:

```
f(X, Y) :- X < Y, write('X is less than Y'), !, fail.
f(X, Y) :- Y < X, write('Y is less than X'), !, fail.
```

```
?- f(2,2).
false.

?- f(7,2).
Y is less than X
false.

?- f(2,8).
X is less than Y
false
```

```
[trace]  ?- f(2,8).
   Call: (8) f(2, 8) ? creep
   Call: (9) 2<8 ? creep
   Exit: (9) 2<8 ? creep
   Call: (9) write('X is less than Y') ? creep
X is less than Y
   Exit: (9) write('X is less than Y') ? creep
   Call: (9) fail ? creep
   Fail: (9) fail ? creep
   Fail: (8) f(2, 8) ? creep
false.
```

Esempio pratico: sudoku

Livello facile:

	7		8		1	3		4
	9			5	7			6
	2	1	4			9		5
3					2	1	6	
	6						5	
	8	7	5					3
7		8			9	5	3	
2			7	8			4	
5		6	3		4		8	

```
?- sudoku_game( [_,7,_,8,_,1,3,_,4],  [_,9,_,_,5,7,_,_,6], [_,2,1,4,_,_,9,_,5],
                [3,_,_,_,_,2,1,6,_],  [_,6,_,_,_,_,_,5,_], [_,8,7,5,_,_,_,_,3],
                [7,_,8,_,_,9,5,3,_], [2,_,_,7,8,_,_,4,_], [5,_,6,3,_,4,_,8,_]).
```

6	7	5	8	9	1	3	2	4
4	9	3	2	5	7	8	1	6
8	2	1	4	6	3	9	7	5
3	5	4	9	7	2	1	6	8
9	6	2	1	3	8	4	5	7
1	8	7	5	4	6	2	9	3
7	4	8	6	1	9	5	3	2
2	3	9	7	8	5	6	4	1
5	1	6	3	2	4	7	8	9

```
[[6,7,5],[8,9,1],[3,2,4]]
[[4,9,3],[2,5,7],[8,1,6]]
[[8,2,1],[4,6,3],[9,7,5]]
-------------------------
[[3,5,4],[9,7,2],[1,6,8]]
[[9,6,2],[1,3,8],[4,5,7]]
[[1,8,7],[5,4,6],[2,9,3]]
-------------------------
[[7,4,8],[6,1,9],[5,3,2]]
[[2,3,9],[7,8,5],[6,4,1]]
[[5,1,6],[3,2,4],[7,8,9]]
true .
```

Implementazione con liste e ricorsione	113.130 s
Implementazione non ricorsiva	19.750 s

Esempio pratico: sudoku

Livello medio:

	4		7	1	5			
9					4		7	
		7				1	3	4
	1	6	4	5				
		9	1		8	5		
				7	9	4	1	
1	9	8				3		
	7		5					2
			9	3	6		8	

```
?- sudoku_game(  [_,4,_,7,1,5,_,_,_], [9,_,_,_,_,4,_,7,_], [_,_,7,_,_,_,1,3,4],
                 [_,1,6,4,5,_,_,_,_], [_,_,9,1,_,8,5,_,_], [_,_,_,_,7,9,4,1,_],
                 [1,9,8,_,_,_,3,_,_], [_,7,_,5,_,_,_,_,2], [_,_,_,9,3,6,_,8,_]).
```

3	4	2	7	1	5	6	9	8
9	8	1	3	6	4	2	7	5
5	6	7	8	9	2	1	3	4
7	1	6	4	5	3	8	2	9
4	3	9	1	2	8	5	6	7
8	2	5	6	7	9	4	1	3
1	9	8	2	4	7	3	5	6
6	7	3	5	8	1	9	4	2
2	5	4	9	3	6	7	8	1

```
[[3,4,2],[7,1,5],[6,9,8]]
[[9,8,1],[3,6,4],[2,7,5]]
[[5,6,7],[8,9,2],[1,3,4]]
-------------------------
[[7,1,6],[4,5,3],[8,2,9]]
[[4,3,9],[1,2,8],[5,6,7]]
[[8,2,5],[6,7,9],[4,1,3]]
-------------------------
[[1,9,8],[2,4,7],[3,5,6]]
[[6,7,3],[5,8,1],[9,4,2]]
[[2,5,4],[9,3,6],[7,8,1]]
true .
```

Implementazione con liste e ricorsione	1209.150 s
Implementazione non ricorsiva	177.71 s

Bibliografia

- Learn Prolog Now! - http://www.learnprolognow.org/

- Linguaggi per l'Intelligenza Artificiale – Agostino Poggi