


Paradigma Logico

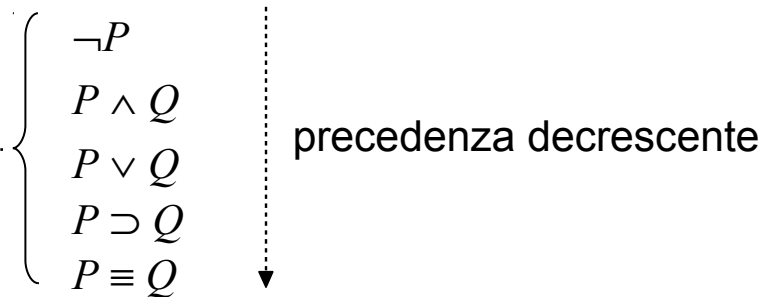
- Programmazione dichiarativa (1970s) → enfasi sul cosa invece che sul come
- Domini applicativi $\left\{ \begin{array}{l} \text{Basi di dati} \rightarrow \text{SQL} \\ \text{Intelligenza artificiale} \rightarrow \text{Prolog} \end{array} \right.$
- Paradigma logico: nasce dal bisogno di $\left\{ \begin{array}{l} \text{processing linguaggio naturale} \rightarrow \text{spec della grammatica} \\ \text{dimostrazione automatica di teoremi} \rightarrow \text{spec del teorema} \end{array} \right.$
- Prolog = linguaggio principalmente usato nella programmazione logica
 basato su scoperta di Robinson (1965) $\left\{ \begin{array}{l} \text{Risoluzione} \\ \text{Unificazione} \end{array} \right.$
- Proprietà interessanti dei programmi logici $\left\{ \begin{array}{l} \text{Nondeterminismo} \rightarrow \text{più soluzioni del pb} \\ \text{Backtracking} \rightarrow \text{incorporato nel linguaggio} \end{array} \right.$

Logica Proposizionale

- Fondamento formale alle expr booleane nei LP

Proposizione

- Costanti **true** e **false** sono proposizioni
- Variabili logiche p, q, r, \dots sono proposizioni
- Se P e Q sono proposizioni, lo sono anche



$$p \vee q \wedge r \supset \neg s \vee t \iff (p \vee (q \wedge r)) \supset ((\neg s) \vee t)$$

- Proposizioni: forniscono rappresentazione simbolica di **espressioni logiche** $\begin{cases} true \\ false \end{cases}$

p : “Maria parla russo”
 q : “Roberto parla russo”
 r : “Maria e Roberto possono comunicare fra loro”



$p \wedge q$: “Sia Maria che Roberto parlano russo”
 $p \vee q$: “Maria o Roberto parla russo”
 $p \wedge q \supset r$: “Se Maria e Roberto parlano russo, allora possono comunicare fra loro”

Logica dei Predicati

- Expr della logica dei predicati: $\left\{ \begin{array}{l} \text{proposizioni} \\ \text{variabili su svariati domini (interi, reali, ...)} \\ \text{funzioni booleane su tali variabili} \\ \text{quantificatori} \end{array} \right.$
- Predicato = proposizione in cui alcune variabili sono sostituite da $\left\{ \begin{array}{l} \text{funzioni booleane} \\ \text{espressioni quantificate} \end{array} \right.$

- Es di funzioni booleane: $\begin{array}{l} \textit{primo}(n) \\ x + y \geq 0 \\ \textit{parla}(x, y) \end{array}$

- Es di predicati: $\begin{array}{l} 0 \leq x \wedge x \leq 1 \\ \textit{parla}(x, \textit{russo}) \wedge \textit{parla}(y, \textit{russo}) \supset \textit{comunica}(x, y) \\ \forall x(\textit{parla}(x, \textit{russo})) \\ \exists x(\textit{parla}(x, \textit{russo})) \\ \forall x \exists y(\textit{parla}(x, y)) \\ \forall x(\textit{analfabeta}(x) \supset (\neg \textit{scrive}(x) \wedge \neg \exists y(\textit{legge}(x, y) \wedge \textit{libro}(y)))) \end{array}$

Logica Proposizionale e Logica dei Predicati

Notazione	Significato
$true, false$	Costanti booleane
p, q, r	Variabili booleane
$\neg p$	Negazione di p
$p \wedge q$	Congiunzione di p e q
$p \vee q$	Disgiunzione di p e q
$p \supset q$	Implicazione: p implica q
$p \equiv q$	Equivalenza logica di p e q
$\forall x(P(x))$	Quantificazione universale
$\exists x(P(x))$	Quantificazione esistenziale
p è una tautologia	Proposizione p è sempre vera (<u>es</u> : $q \vee \neg q$)
$P(x)$ è valida	Predicato $P(x)$ è vero per ogni valore di x (<u>es</u> : $pari(x) \vee dispari(x)$)

- Predicato vero per $\left\{ \begin{array}{l} \text{qualche assegnamento di valori delle sue variabili} \rightarrow \text{**soddisfacibile**} \\ \text{tutti i possibili assegnamenti di valori delle sue variabili} \rightarrow \text{**valido**} \end{array} \right.$

$parla(x, russo)$
 $y \geq 0 \wedge n \geq 0 \wedge z = x(y-n)$

} soddisfacibili ma non validi

Proprietà Algebriche dei Predicati

<i>Proprietà</i>	<i>Significato</i>	
Commutatività	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Associatività	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Distributività	$p \vee q \wedge r \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv p \wedge q \vee p \wedge r$
Idempotenza	$p \vee p \equiv p$	$p \wedge p \equiv p$
Identità	$p \vee \neg p \equiv \text{true}$	$p \wedge \neg p \equiv \text{false}$
deMorgan	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Implicazione	$p \supset q \equiv \neg p \vee q$	
Quantificazione	$\neg \forall x P(x) \equiv \exists x \neg P(x)$	$\neg \exists x P(x) \equiv \forall x \neg P(x)$

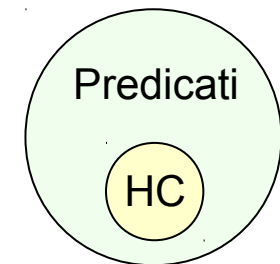
Clausole di Horn

$h \leftarrow p_1, p_2, \dots, p_n$ in cui h, p_i sono predicati: h è *true* se p_1, p_2, \dots, p_n sono *true*

- Esempio: Se nella città C c'è una precipitazione e la temperatura di C è gelida, allora nevica in C .

$nevica(C) \leftarrow precipitazione(C), gelida(C)$

- Limitata corrispondenza fra clausole di Horn e predicati



$precipitazione(C) \wedge gelida(C) \supset nevica(C)$

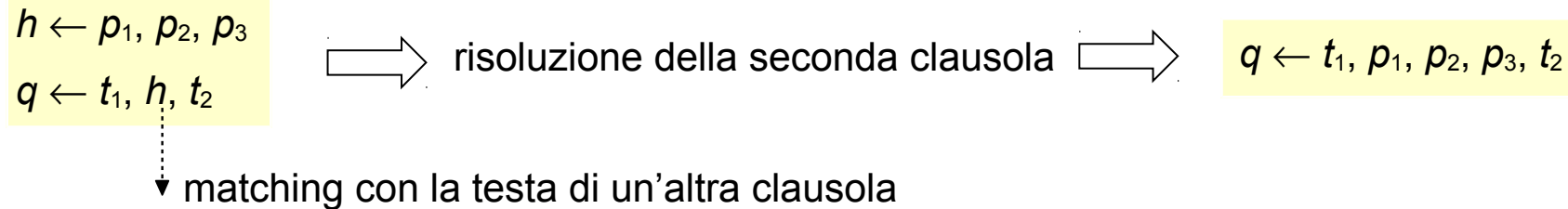
equivalente a

$\neg(precipitazione(C) \wedge gelida(C)) \vee nevica(C)$

$\neg precipitazione(C) \vee \neg gelida(C) \vee nevica(C)$

Risoluzione e Unificazione

- **Risoluzione** = singola inferenza da una coppia di clausole di Horn



parla(maria, inglese)
comunica(X, Y) ← parla(X, L), parla(Y, L), X ≠ Y



comunica(maria, Y) ← parla(maria, inglese), parla(Y, inglese), maria ≠ Y

- **Istanziamento** = assegnamento delle variabili nella risoluzione
- **Unificazione** = processo di pattern matching che determina quali istanziazioni attuare nella risoluzione

Prolog

- Programma costruito mediante **termini** $\left\{ \begin{array}{l} \text{costanti (maria, 'Luigi', inglese, 'il linguaggio', 25)} \\ \text{variabili (X, Y, Linguaggio, Persona)} \\ \text{strutture (con zero o più argomenti)} \end{array} \right.$

```
animale(tigre)
parla(Qualcuno, inglese)
comunica(X, Y)
```

- **Fatto** = termine seguito dal punto `parla(maria, inglese).`
- **Regola** \approx clausola di Horn `term :- term1, term2, ..., termn.`
- **Programma** = lista di fatti e regole

```
parla(alberto, russo).
parla(roberto, inglese).
parla(maria, russo).
parla(maria, inglese).
comunica(X, Y) :- parla(X, L), parla(Y, L), X \= Y.
```


Prolog (ii)

- Una regola Prolog *ha successo* se ci sono istanziazioni delle sue variabili tali che tutti i termini nella parte destra hanno successo simultaneamente (altrimenti *fallisce*)
- Un fatto ha sempre successo (sempre *true*)

```
parla(alberto, russo).  
parla(roberto, inglese).  
parla(maria, russo).  
parla(maria, inglese).  
comunica(X, Y) :- parla(X, L), parla(Y, L), X \= Y.
```

$$\begin{cases} X = \text{alberto} \\ Y = \text{maria} \\ L = \text{russo} \end{cases} \Rightarrow \text{successo}$$
$$\begin{cases} X = \text{alberto} \\ Y = \text{roberto} \\ L = \forall \end{cases} \Rightarrow \text{fallimento}$$

- **Query** = costruito per “esercitare” un programma

```
?- parla(Chi, russo).
```

Interazione

- Codifica di fatti e regole in un file *nomefile* e caricamento nell'interprete

```
?- consult(nomefile).
```

- Risposta ad una query \Rightarrow ricerca di fatti e regole aventi una testa che corrisponde al predicato della query; se più di uno, vengono considerati secondo l'ordine sequenziale nel programma

```
parla(alberto, russo).  
parla(roberto, inglese).  
parla(maria, russo).  
parla(maria, inglese).  
comunica(X, Y) :- parla(X, L), parla(Y, L), X \= Y.
```

```
?- parla(Chi, russo).  
Chi = alberto ;  
Chi = maria ;  
false.
```

Strutture

- Singolo oggetto composto da altri oggetti (componenti): *funtore(obj₁, obj₂, ..., obj_n)*

```
legge(anna, libro).  
legge(luigi, libro(psicoanalisi, freud)).  
legge(silvia, libro(psicoanalisi, autore(sigmund, freud))).
```

```
?- legge(X, libro).  
X = anna ;  
false.
```

```
?- legge(luigi, libro(psicoanalisi, X)).  
X = freud.
```

```
?- legge(X, L).  
X = anna,  
L = libro ;  
X = luigi,  
L = libro(psicoanalisi, freud) ;  
X = silvia,  
L = libro(psicoanalisi, autore(sigmund, freud)).
```

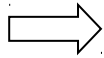
Operatori

- Operatori = funtori di strutture in forma infissa:

```
a + b ≡ +(a,b)
```

```
x + y * z ≡ +(x, *(y, z))
```

```
p(2+3).  
p(5+7).
```



```
?- p(X).  
X = 2+3 ;  
X = 5+7.
```

```
?- p(X), Y = X.  
X = 2+3,  
Y = 2+3 ;  
X = 5+7,  
Y = 5+7.
```

deve essere istanziato



```
?- p(X), Y is X.  
X = 2+3,  
Y = 5 ;  
X = 5+7,  
Y = 12.
```

- Uguaglianza e unificazione: **?- X = Y.**

- X** var non istanziata, **Y** istanziato \Rightarrow successo (**X** istanziato come **Y**)
- Costanti uguali a se stesse
- Strutture uguali se hanno lo stesso funtore, stesso n° componenti e se tutti i componenti che si corrispondono sono uguali

```
?- legge(anna, Y) = legge(X, libro).  
Y = libro,  
X = anna.
```

Operatori (ii)

- Operatori di confronto (fra numeri) che richiedono gli argomenti istanziati:

Confronto	Significato
$X == Y$	X ed Y sono lo stesso numero
$X \neq Y$	X ed Y sono numeri diversi
$X < Y$	X è minore di Y
$X > Y$	X è maggiore di Y
$X \leq Y$	X è minore o uguale a Y
$X \geq Y$	X è maggiore o uguale a Y

- Operatori aritmetici: $+$, $-$, $*$, $/$, $//$ (divisione intera), **mod**
- Uguaglianza stretta: $(X == Y) \Rightarrow (X = Y)$

$X == Y$ se e solo se **co-referenziano** (risultato dell'unificazione)

```
?- X == 3.  
false.
```

```
?- X = 2, X == Y.  
false.
```

```
?- X = 2, Y = 2, X == Y.  
X = 2,  
Y = 2.
```

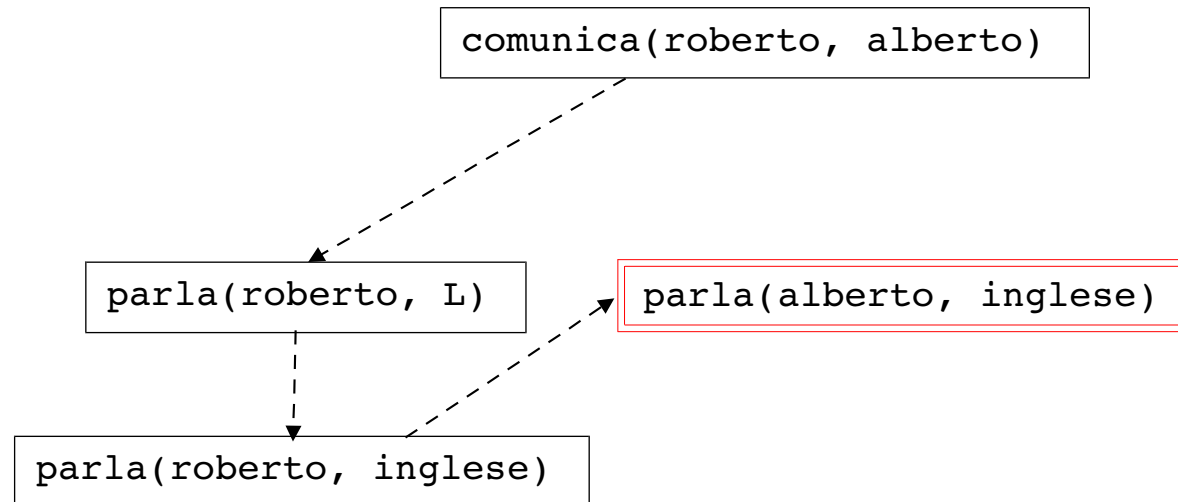
```
?- X = 2, Y = 3, X == Y.  
false.
```

```
?- X = Y, X == Y.  
X = Y.
```

Ricerca di Soluzioni

```
parla(alberto, russo).  
parla(roberto, inglese).  
parla(maria, russo).  
parla(maria, inglese).  
comunica(X, Y) :- parla(X, L), parla(Y, L), X \= Y.
```

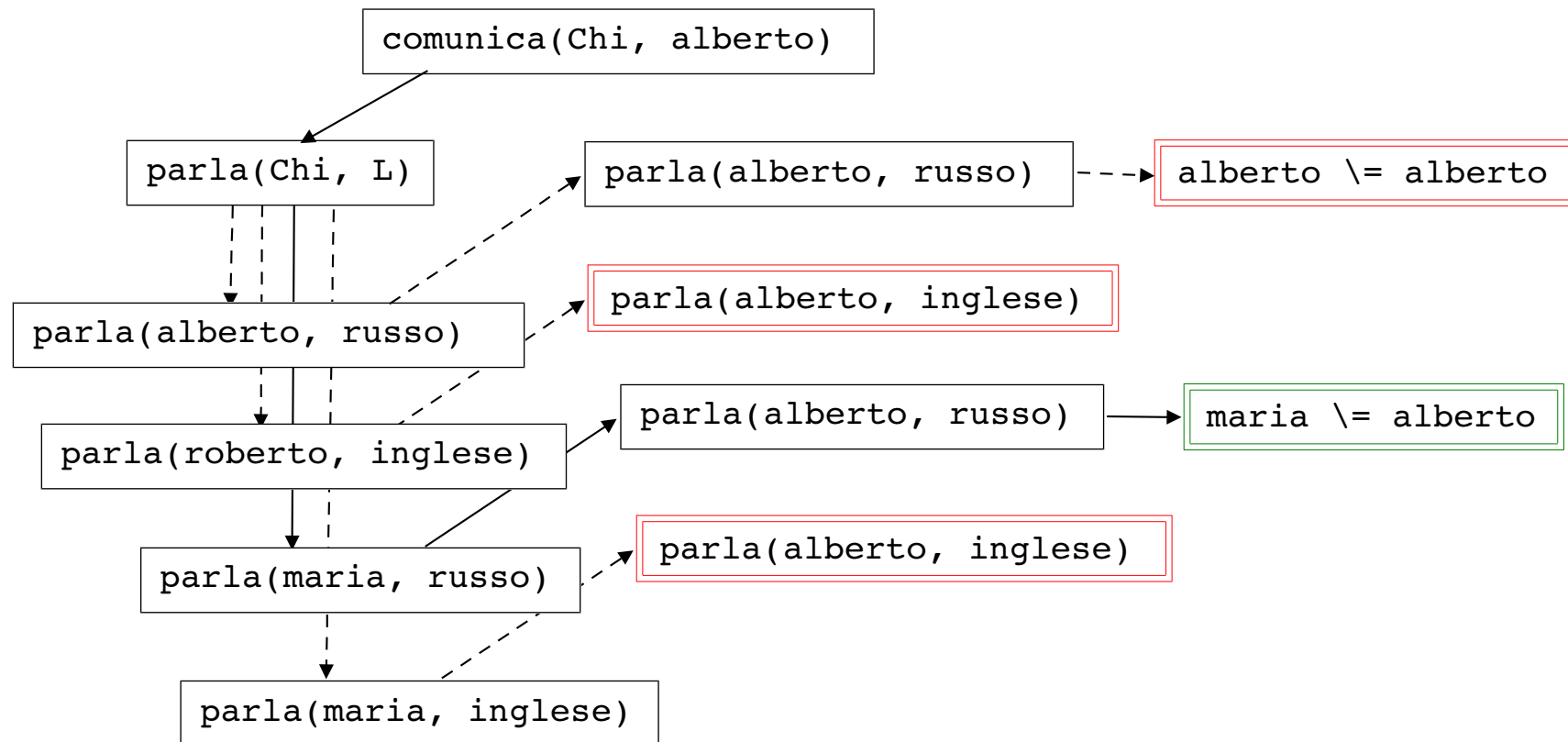
```
?- comunica(roberto, alberto).  
false.
```



Ricerca di Soluzioni (ii)

```
parla(alberto, russo).  
parla(roberto, inglese).  
parla(maria, russo).  
parla(maria, inglese).  
comunica(X, Y) :- parla(X, L), parla(Y, L), X \= Y.
```

```
?- comunica(Chi, alberto).  
Chi = maria ;  
false.
```



Interrogazione di Basi di Dati

```
madre(maria, sara).  
madre(maria, bruno).  
madre(sara, nicola).  
madre(sara, giulia).  
madre(gina, rino).
```

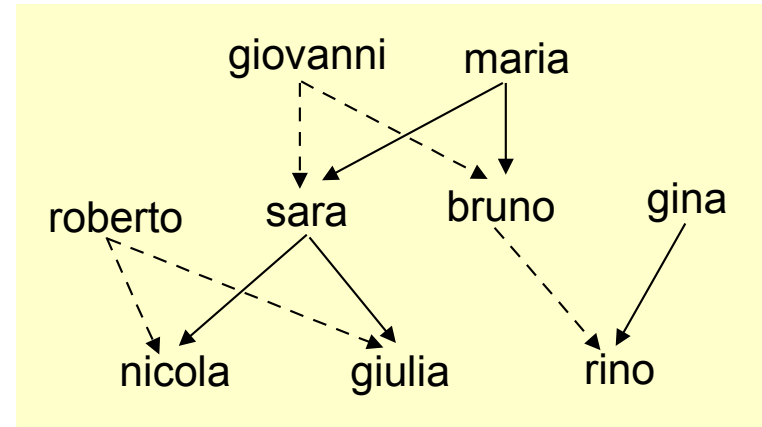
```
padre(giovanni, sara).  
padre(giovanni, bruno).  
padre(roberto, nicola).  
padre(roberto, giulia).  
padre(bruno, rino).
```

```
genitore(X, Y) :- madre(X, Y).  
genitore(X, Y) :- padre(X, Y).
```

```
nonno(X, Y) :- padre(X, Z), genitore(Z, Y).  
nonna(X, Y) :- madre(X, Z), genitore(Z, Y).
```

```
fratello(X, Y) :- genitore(Z, X),  
                  genitore(Z, Y), X \= Y.
```

```
antenato(X, Y) :- genitore(X, Y).  
antenato(X, Y) :- genitore(X, Z),  
                  antenato(Z, Y).
```



```
?- fratello(nicola, giulia).  
true
```

```
?- nonno(Chi, rino).  
Chi = giovanni ;  
false.
```

```
?- antenato(Chi, giulia).  
Chi = sara ;  
Chi = roberto ;  
Chi = maria ;  
Chi = giovanni ;  
false.
```


Liste

$[elem_1, elem_2, \dots, elem_n]$

$[il, sole, splende]$

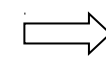
$[X]$

$[X, _, Z]$

$[X \mid Y]$

- Concatenazione:

$.(il, .(sole, .(splende, [])))$



$.(testa, coda)$



$[X \mid Y] \equiv .(X, Y)$

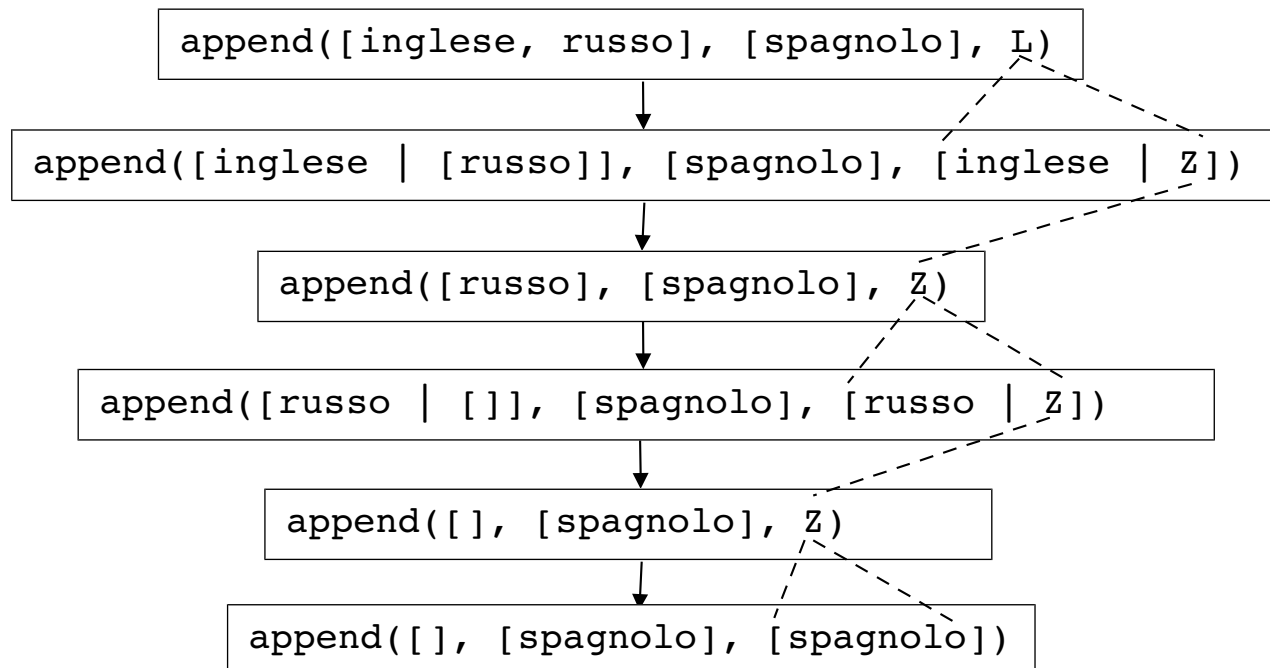
`append([], X, X).`

`append([H | T], Y, [H | Z]) :- append(T, Y, Z).`

`?- append([inglese, russo], [spagnolo], L).`

`L = [inglese, russo, spagnolo].`

$[1, 2, X \mid Y] \equiv .(1, .(2, .(X, Y)))$



Liste (ii)

- Appartenenza:

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

```
?- member(5, [4,2,5,7]).  
true ;  
false.
```

- Inversione:

```
reverse([], []).  
reverse([Testa | Coda], R) :- reverse(Coda, RC),  
                             append(RC, [Testa], R).
```

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

- Ultimo elemento:

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

```
?- last(X, [alfa,beta,gamma]).  
X = gamma ;  
false.
```

Liste (iii)

- Elementi consecutivi:

```
consecutivi(X, Y, [X,Y|_]).  
consecutivi(X, Y, [_|Z]) :- consecutivi(X, Y, Z).
```

```
?- consecutivi(2,3,[1,2,3,4]).  
true ;  
false.
```

```
?- consecutivi(2,X,[1,2,3,4]).  
X = 3 ;  
false.
```

```
?- consecutivi(X,Y,[1,2,3]).  
X = 1,  
Y = 2 ;  
X = 2,  
Y = 3 ;  
false.
```

- Prefisso:

```
prefix([],_).  
prefix([X|Coda1], [X|Coda2]) :- prefix(Coda1, Coda2).
```

```
?- prefix([1,2],[1,2,3]).  
true.
```

```
?- prefix([2,3],[1,2,3,4]).  
false.
```

```
?- prefix([1,2],X).  
X = [1, 2|_G300].
```

```
?- prefix(X,[1,2,3]).  
X = [] ;  
X = [1] ;  
X = [1, 2] ;  
X = [1, 2, 3] ;  
false.
```

Cut

- Possibile alterare il meccanismo di backtracking mediante cut “!”
- Cut: inibizione del ri-soddisfacimento di certi goal nel backtracking
- Utile per rendere il programma più efficiente (quando si sa a priori che il backtracking non contribuisce alla soluzione)
- Essenziale (in certi casi) per l'efficacia del programma
- ! = predicato senza argomenti:
 1. Ha successo immediatamente
 2. Non può essere ri-soddisfatto
 3. Congelamento delle scelte fatte dal momento della chiamata del goal genitore

Cut (ii)

- Esempio: Servizi offerti da una biblioteca (limitati per utenti non affidabili)

```
servizi(Persona, Servizio) :-  
    inaffidabile(Persona), !, servizio_limitato(Servizio).  
  
servizi(Persona, Servizio) :- servizio_generale(Servizio).  
  
servizio_limitato(consultazione).  
servizio_limitato(fotocopia).  
  
servizio_esteso(prestito).  
servizio_esteso(digitalizzazione).  
  
servizio_generale(Servizio) :- servizio_limitato(Servizio).  
servizio_generale(Servizio) :- servizio_esteso(Servizio).  
  
inaffidabile(teodoro).
```

```
?- servizi(rino,S).  
S = consultazione ;  
S = fotocopia ;  
S = prestito ;  
S = digitalizzazione.
```

```
?- servizi(teodoro,S).  
S = consultazione ;  
S = fotocopia.
```

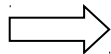
Cut (iii)

```
r :- a, b, c, !, d, e, f.
```

- Prima di “!”: backtracking fra a, b, c.
- Dopo “!”: backtracking fra d, e, f.
- Ritorno prima di “!”: fallimento di tutta la congiunzione di goal

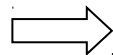
```
a(alfa).  
a(beta).  
b(alfa).  
b(beta).  
c(gamma).
```

```
r(X) :- a(X), b(X).  
r(X) :- c(X).
```



```
?- r(X).  
X = alfa ;  
X = beta ;  
X = gamma.
```

```
r(X) :- a(X), !, b(X).  
r(X) :- c(X).
```

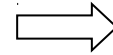


```
?- r(X).  
X = alfa.
```

Cut (iv)

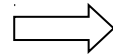
- Uso del cut per inibire ricorsione infinita:

```
sommatoria(0,0).  
sommatoria(N, S) :- N1 is N-1,  
                    sommatoria(N1,S1),  
                    S is S1+N.
```



```
?- sommatoria(10, S).  
S = 55 ;  
ERROR: Out of local stack
```

```
sommatoria(0,0) :- !.  
sommatoria(N, S) :- N1 is N-1,  
                    sommatoria(N1,S1),  
                    S is S1+N.
```

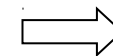


```
?- sommatoria(10, S).  
S = 55.
```

```
?- sommatoria(-10, S).  
ERROR: Out of local stack
```

- Soluzione alternativa:

```
sommatoria(0,0).  
sommatoria(N, S) :- N > 0,  
                    N1 is N-1,  
                    sommatoria(N1,S1),  
                    S is S1+N.
```

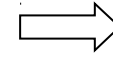


```
?- sommatoria(-10, S).  
false.
```

Cut (v)

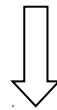
- Efficienza vs chiarezza:

```
append([ ], X, X).  
append([H|T], Y, [H|Z]) :- append(T, Y, Z).
```



```
?- append(X,Y,[a,b,c]).  
X = [ ],  
Y = [a, b, c] ;  
X = [a],  
Y = [b, c] ;  
X = [a, b],  
Y = [c] ;  
X = [a, b, c],  
Y = [ ] ;  
false.
```

Se uso di **append** con i primi due argomenti istanziati → inutile provare anche la seconda regola quando il primo argomento è []



```
append([ ], X, X) :- !.  
append([H|T], Y, [H|Z]) :- append(T, Y, Z).
```



Equivalente quando (almeno) il primo argomento è istanziato:

```
?- append([a,b,c],X,Y).  
Y = [a, b, c|X].
```

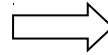
Invece:

```
?- append(X,Y,[a,b,c]).  
X = [ ],  
Y = [a, b, c].
```


Cut (vi)

- Uso di cut → necessario essere sicuri dell'uso (esercitazione) delle regole

```
numero_di_genitori(adamo, 0) :- !.  
numero_di_genitori(eva, 0) :- !.  
numero_di_genitori(X, 2).
```



```
?- numero_di_genitori(eva, N).  
N = 0.
```

```
?- numero_di_genitori(luisa, N).  
N = 2.
```

Però:

```
?- numero_di_genitori(eva, 2).  
true.
```

- Possibili modifiche:

```
numero_di_genitori(adamo, N) :- !, N = 0.  
numero_di_genitori(eva, N) :- !, N = 0.  
numero_di_genitori(X, 2).
```

oppure

```
numero_di_genitori(adamo, 0).  
numero_di_genitori(eva, 0).  
numero_di_genitori(X, 2) :- X \= adamo, X \= eva.
```

Processing di Liste

- Cancellazione di tutte le occorrenze di un elemento: `delete(X,L1,L2)`

```
delete(_, [], []).  
delete(X, [X|L], D) :- !, delete(X, L, D).  
delete(X, [Y|L1], [Y|L2]) :- !, delete(X, L1, L2).
```

```
?- delete(a, [1,2,a,3,4,a,5,a], D).  
D = [1, 2, 3, 4, 5] ;  
false.
```

- Sostituzione di tutte le occorrenze di un elemento E con S: `subst(E,L1,S,L2)`

```
subst(_, [], _, []).  
subst(E, [E|L1], S, [S|L2]) :- !, subst(E, L1, S, L2).  
subst(E, [X|L1], S, [X|L2]) :- subst(E, L1, S, L2).
```

```
?- subst(a, [1,2,a,3,4,a,5,a], b, X).  
X = [1, 2, b, 3, 4, b, 5, b] ;  
false.
```

- Rimozione di duplicati: `remdup(L,R)`

```
remdup(L,R) :- dupacc(L, [], R).  
dupacc([], A, A).  
dupacc([Testa|Coda], A, R) :- member(Testa, A), !, dupacc(Coda, A, R).  
dupacc([Testa|Coda], A, R) :- dupacc(Coda, [Testa|A], R).
```

```
?- remdup([a,b,a,c,b,a,c], X).  
X = [c, b, a].
```

Processing di Insiemi

- Sottoinsieme: **subset**(X,Y)

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

```
?- subset([1,2,3], [2,1,4,3,6,7]).  
true ;  
false.
```

- Unione: **union**(X,Y,Z)

```
union([], X, X).  
union([X|Coda], Y, Z) :- member(X, Y), !, union(Coda, Y, Z).  
union([X|Coda], Y, [X|Z]) :- union(Coda, Y, Z).
```

```
?- union([1,2,3], [2,3,4], U).  
U = [1, 2, 3, 4].
```

- Differenza: **diff**(X,Y,Z)

```
diff([],_,[]).  
diff([X|Coda], Y, Z) :- member(X, Y), !, diff(Coda, Y, Z).  
diff([X|Coda], Y, [X|Z]) :- diff(Coda, Y, Z).
```

```
?- diff([1,2,3,4], [3,4,5], D).  
D = [1, 2].
```

Processing di Insiemi (ii)

- Intersezione: `intersection(X,Y,Z)`

```
intersection([],_,[]).  
intersection([X|Cx], Y, [X|Cz]) :- member(X, Y), !, intersection(Cx, Y, Cz).  
intersection([_|Cx], Y, Z) :- intersection(Cx, Y, Z).
```

```
?- intersection([1,2,3,4], [3,4,5], I).  
I = [3, 4].
```

- Prodotto Cartesiano (generazione di strutture binarie con funtore pair):
`cartprod(X,Y,Z)`

```
cartprod([],_,[]).  
cartprod([Hx|Tx], Y, Z) :- combine(Hx, Y, Pz),  
                           cartprod(Tx, Y, Sz),  
                           append(Pz, Sz, Z).  
  
combine(_, [], []).  
combine(Hx, [Hy|Ty], [pair(Hx, Hy)|Tz]) :- combine(Hx, Ty, Tz).
```

```
?- cartprod([1,2,3], [a,b], P).  
P = [pair(1,a), pair(1,b), pair(2,a), pair(2,b), pair(3,a), pair(3,b)] ;  
false.
```

Allocazione degli Inquilini

1. Barbara, Claudio, Francesca, Marco e Silvia vivono in un palazzo di cinque piani.
2. Barbara non vive al quinto piano e Claudio non vive al primo piano.
3. Francesca non vive né al primo né all'ultimo piano e nemmeno in un piano adiacente a Silvia o Claudio.
4. Marco vive in un piano superiore a quello in cui vive Claudio.



Dove vivono Barbara, Claudio, Francesca, Marco e Silvia ?

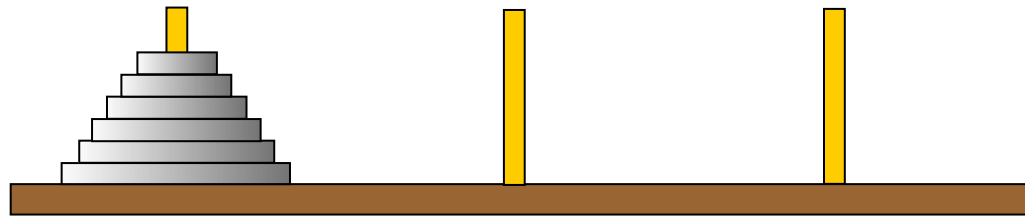
```
inquilini(Palazzo) :-  
    Palazzo = [piano(_,5),piano(_,4),piano(_,3),piano(_,2),piano(_,1)],  
    member(piano(barbara, B), Palazzo), B \= 5,  
    member(piano(claudio, C), Palazzo), C \= 1,  
    member(piano(francesca, F), Palazzo), F \= 1, F \= 5,  
    member(piano(marco, M), Palazzo), M > C,  
    member(piano(silvia, S), Palazzo),  
    \+(adiacente(S, F)), \+(adiacente(F, C)).  
  
adiacente(X, Y) :- X is Y+1.  
adiacente(X, Y) :- X is Y-1.
```

M	5
F	4
B	3
C	2
S	1

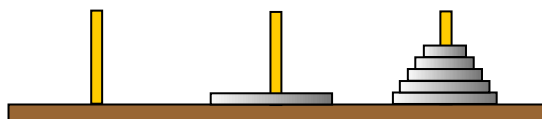
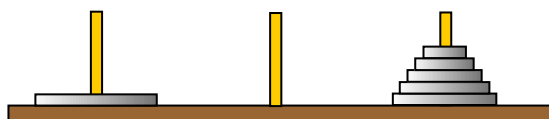
```
?- inquilini(X).  
X = [piano(marco,5), piano(francesca,4), piano(barbara,3), piano(claudio,2), piano(silvia, 1)] ;  
false.
```

Le Torri di Hanoi

- Obiettivo: spostare tutti i dischi nel piolo centrale rispettando due vincoli:
- Solo il disco superiore di una torre può essere rimosso ad ogni spostamento;
- Il disco rimosso da una torre non può essere spostato su un disco più piccolo di un'altra torre.



- Tecnica di risoluzione (per N dischi):
 - Terminazione: quando non ci sono dischi nel piolo sorgente;
 - Muovere $N-1$ dischi dal piolo sorgente al piolo di appoggio, usando il piolo di destinazione come piolo di appoggio;
 - Muovere un singolo disco dal piolo sorgente a quello di destinazione;
 - Muovere $N-1$ dischi dal piolo di appoggio al piolo di destinazione, usando il piolo sorgente come piolo di appoggio.

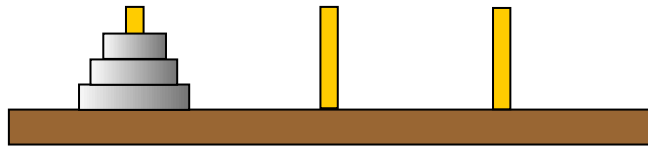


Le Torri di Hanoi (ii)

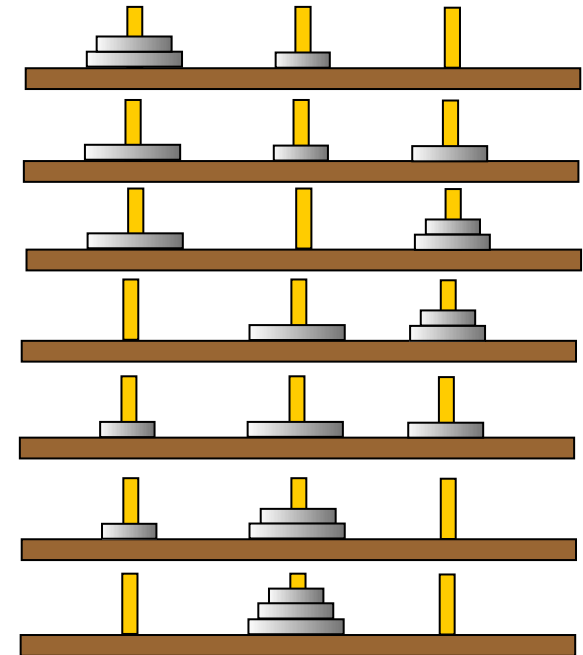
```
hanoi(N) :- muovi(N, sinistra, centro, destra).

muovi(0, _, _, _) :- !.
muovi(N, S, D, A) :- M is N-1,
    muovi(M, S, A, D),
    visualizza_mossa(S, D),
    muovi(M, A, D, S).

visualizza_mossa(X, Y) :- writeln([muovi, un, disco, dal, polo, di, X, al, polo, di, Y]).
```

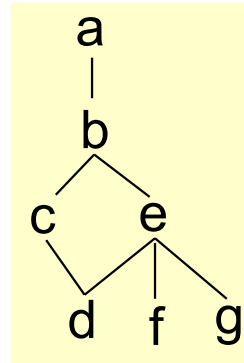
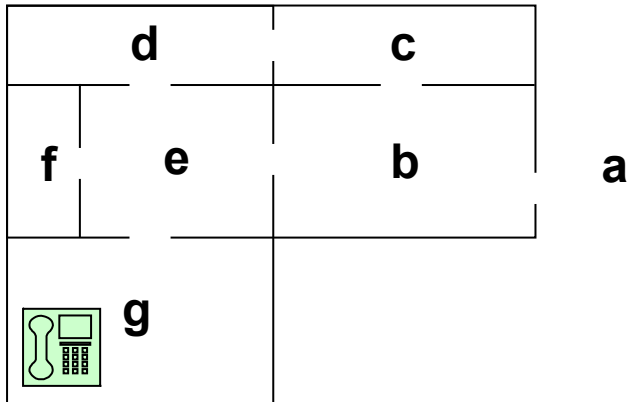


```
?- hanoi(3).
[muovi, un, disco, dal, polo, di, sinistra, al, polo, di, centro]
[muovi, un, disco, dal, polo, di, sinistra, al, polo, di, destra]
[muovi, un, disco, dal, polo, di, centro, al, polo, di, destra]
[muovi, un, disco, dal, polo, di, sinistra, al, polo, di, centro]
[muovi, un, disco, dal, polo, di, destra, al, polo, di, sinistra]
[muovi, un, disco, dal, polo, di, destra, al, polo, di, centro]
[muovi, un, disco, dal, polo, di, sinistra, al, polo, di, centro]
true.
```



Ricerca in un Labirinto

- Obiettivo: trovare una stanza con il telefono



```
porta(a, b).  
porta(b, c).  
porta(b, e).  
porta(c, d).  
porta(d, e).  
porta(e, f).  
porta(g, e).  
  
ha_telefono(g).
```

```
comunicanti(X, Y) :- porta(X, Y).  
comunicanti(X, Y) :- porta(Y, X).
```

```
vai(X, X, L) :- reverse(L, R), writeln(R).  
vai(X, Y, L) :- comunicanti(X, Z), \+(member(Z, L)), vai(Z, Y, [Z|L]).
```

```
?- ha_telefono(X), vai(a, X, []).  
[b, c, d, e, g]  
X = g ;  
[b, e, g]  
X = g ;  
false.
```

```
?- ha_telefono(X), vai(a, X, [c]).  
[c, b, e, g]  
X = g ;  
false.
```


Copertura di un grafo

- *Determinare un cammino che ricopra un grafo percorrendo ogni tratto una sola volta.*

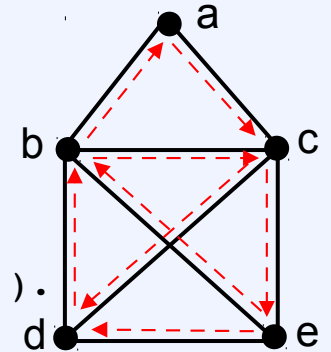
```
nodi([a,b,c,d,e]).
archi([arco(a,b),arco(a,c),arco(b,c),arco(c,e),arco(e,d),arco(d,b),arco(b,e),arco(c,d)]).

collegati(X,Y,A) :- member(arco(X,Y),A).
collegati(X,Y,A) :- member(arco(Y,X),A).

consumato(X,Y,C) :- consecutivi(X,Y,C).
consumato(X,Y,C) :- consecutivi(Y,X,C).

cammino(X) :- nodi(N), archi(A), length(A,La), member(X,N), copri(X,A,La,[X]).

copri(_,_,La,C) :- length(C,Lc), Lc is La+1, reverse(C,R), writeln(R).
copri(X,A,La,C) :- collegati(X,Y,A), \+(consumato(X,Y,C)), copri(Y,A,La,[Y|C]).
```



```
?- cammino(X).
[d, b, c, e, d, c, a, b, e]
X = d ;
...
[d, c, b, d, e, b, a, c, e]
X = d ;
[e, d, b, c, e, b, a, c, d]
X = e
...
```