

## Introduzione al Prolog

Lezione del Corso Interazione Uomo Macchina 2, Docente Francesco Mele

Corso di Laurea in Informatica Università di Napoli Federico II,  
Anno Accademico 2004-2005

# IL PROLOG - DUE PRINCIPALI INTERPRETAZIONI

Come linguaggio dichiarativo (in un'accezione logica)

Esempio di  
programma  
Prolog

sorella\_di(S1,S2):-

    donna(S1),

    genitori(P,M,S1),

    genitori(P,M,S2).

è vero che S1 è sorella di S2 se

è vero che S1 è donna e

è vero che P e M sono genitori di S1 e

è vero che P e M sono genitori di S2

Come linguaggio procedurale

sorella\_di(S1,S2):-

    donna(S1),

    genitori(P,M,S1),

    genitori(P,M,S2).

La procedura sorella\_di(S1,S2) ha successo (viene fornito un valore ad S1 e S2) se hanno successo le procedure donna(S1), genitori(P,M,S1), genitori(P,M,S2).

# Costanti

1- le costanti iniziano per lettera minuscola:

a, b, c, f7, u6, penna, carta, andrea, camilleri,  
numeroStudenti, prezzoLibri;

2- gli interi sono costanti:

1,3, 44, 100;

3- le stringhe sono costanti:

'penna', 'il corso delle cose', 'Andrea'

# Variabili

Le variabili iniziano con la lettera maiuscola:

X, Y, F1, F2, Testi, Titoli, MarcheProdotti, ListaLibri

# Programmi prolog

Un programma Prolog è composto da:

fatti;  
regole;  
query

## Fatti

personaggio(montalbano).  
personaggio(maigret).  
personaggio('Pepe Carvalho').

scrittore(georges,simenon).  
scrittore(andrea,camilleri).  
scrittore(manuel,montalban).

libro('Tatuaggio', montalban, 'Universale Economica Feltrinelli').  
libro('La gita a Tindari', camilleri, 'Sellerio').  
libro('Il profumo della notte', camilleri, 'Sellerio').

## Query 1

| ?- **personaggio**(X).

X = montalbano ;

X = maigret

| ?-

Unificazione fra la variabile X e le costante **montalbano**.

## Query 2

| ?- **scrittore**(X,Y).

X = georges ,

Y = simenon ;

X = andrea ,

Y = camilleri ;

X = manuel,

Y = montalban

| ?-

## Query 3

| ?- **scrittore**(georges, simenon).

yes

| ?-

## Query con variabili condivise (shared).

| ?- scrittore(H,X), libro(Titolo, X, Editore).

H = andrea ,  
X = camilleri ,  
Titolo = 'La gita a Tindari' ,  
Editore = 'Sellerio' ;

H = andrea ,  
X = camilleri ,  
Titolo = 'Il profumo della notte' ,  
Editore = 'Sellerio' ;

H = manuel ,  
X = montalban ,  
Titolo = 'Tatuaggio' ,  
Editore = 'Feltrinelli' ;

Virgola come  
congiunzione logica



Database dei fatti

scrittore(georges,simenon).  
scrittore(andrea,camilleri).  
scrittore(manuel,montalban).

libro('Tatuaggio', montalban, 'Feltrinelli').  
libro('La gita a Tindari', camilleri, 'Sellerio').  
libro('Il profumo della notte', camilleri, 'Sellerio').

# Regole

Testa della regola → `autore_di(Cognome, Titolo):-`  
Corpo della regola → `scrittore(Nome, Cognome),  
libro(Titolo, Cognome, Editore).`

Esecuzione → `| ?- autore_di(Y,U).`

`Y = camilleri ,  
U = 'La gita a Tindari' ;`

`Y = camilleri ,  
U = 'Il profumo della notte' ;`

`Y = montalban ,  
U = 'Tatuaggio' ;`

`no`

`| ?-`

# Backtracking Prolog (I)

Prima di ogni discussione - per efficienza di programmazione (e non solo) nei programmi occorre scrivere in ordine prima i fatti e poi le regole.

| ?- autore\_di(Cognome, Titolo).

## Insieme di eventi

1° l'interprete parte dall'inizio del programma a) cerca fra i fatti poi tra le regole e b) trova una regola con una testa dello stesso tipo della query.

2° l'interprete a) inserisce fra i suoi obiettivi quello di instanziare scrittore(Nome, Cognome) b) parte dall'inizio del programma c) instancia Nome= **georges** Cognome= **simenon**).

3° continua di dimostare tutte le condizioni della regola attivata - ma fallisce la ricerca per

libro(Titolo, simenon, Editor)

scrittore(Nome, Cognome).

Cognome = libro(Titolo, **simenon**, Editore).  
**simenon**

/\* Inizio programma \*/

/\* Database \*/

scrittore(georges, simenon).  
scrittore(andrea, camilleri).  
scrittore(manuel, montalban).

libro('Tatuaggio', montalban, 'Feltrinelli').  
libro('La gita a Tindari', camilleri, 'Sellerio').  
libro('Il profumo della notte', camilleri, 'Sellerio').

/\* Regole \*/

autore\_di(Cognome, Titolo): -  
scrittore(Nome, Cognome),  
libro(Titolo, Cognome, Editore).

/\* Fine programma \*/

Fallimento per Cognome = **simenon**

# Backtracking Prolog (II)

| ?- autore\_di(Cognome, Nome).

libro(Titolo, camilleri, Editore).



Cognome=simenon scrittore(Nome, simenon).  
Cognome=camilleri, scrittore(Nome, Cognome).  
Nome=andrea



Titolo= 'La gita a Tindari'  
Editore= 'Sellerio'

autore\_di(C, T).  
Cognome=camilleri, scrittore(Nome, Cognome)  
Nome=andrea libro(Titolo, camilleri, Editore).  
Titolo= 'La gita a Tindari'  
Editore= 'Sellerio'



```
/* Inizio programma */  
  
/* Database */  
  
scrittore(georges, simenon).  
scrittore(andrea, camilleri).  
scrittore(manuel, montalban).  
  
libro('Tatuaggio', montalban, 'Feltrinelli').  
libro('La gita a Tindari', camilleri, 'Sellerio').  
libro('Il profumo della notte', camilleri, 'Sellerio').  
  
/* Regole */  
  
autore_di(Cognome, Titolo): -  
    scrittore(Nome, Cognome),  
    libro(Titolo, Cognome, Editore).  
  
/* Fine programma */
```

Successo di autore\_di(camilleri, 'La gita a Tindari')

Fallimento per Cognome = simenon

# Indeterminismo del Prolog

## WIN PROLOG

c.

f.

b:-a.

a:-b,c,f.

b:-c,f.

| ?- a.

Error 1, Backtrack Stack Full, Trying  
b/0

Aborted

## XSB PROLOG

c.

f.

b:-a.

a:-b,c,f.

b:-c,f.

| ?- a.

## CONFRONTO CON DATABASE (I)

### TABELLA GIOCATORE

Id	Nome	Cognome	Ruolo	Nazione
1	Zinedine	Zidane	attaccante	Francia
2	Roberto	Carlos	terzino	Brasile
3	Francesco	Totti	attaccante	Italia
4	Gianluigi	Buffon	portiere	Italia
5	Francesco	Toldo	portiere	Italia

### FATTI PROLOG (Database Prolog)

```
giocatori(1,'Zinedine', 'Zidane', attaccante, 'Francia').  
giocatori(2,'Roberto', 'Carlos', terzino, 'Brasile').  
giocatori(3,'Francesco', 'Totti', attaccante, 'Italia').  
giocatori(4,'Gianluigi', 'Buffon', portiere, 'Italia').  
giocatori(5,'Francesco', 'Toldo', portiere, 'Italia').
```

### QUERY SQL

```
SELECT Nome,Cognome  
FROM Tabella_giocatori  
WHERE Nazione="Italia";
```

### QUERY PROLOG

```
?- giocatori(_ ,X,Y,_ , 'Italia').  
Oppure (poco usato)  
?- giocatori(_ ,X, Y,_ ,N), N = 'Italia'.
```

### FORMA DELLA RISPOSTA

Nome	Cognome
Francesco	Totti
Gianluigi	Buffon
Francesco	Toldo

### FORMA DELLA RISPOSTA

```
X = 'Zinedine' ,  
Y = 'Zidane' ;  
  
X = 'Roberto' ,  
Y = 'Carlos' ; ;  
  
X = 'Francesco' ,  
Y = 'Totti' ;  
  
X = 'Gianluigi' ,  
Y = 'Buffon'  
  
X = 'Francesco'  
Y = 'Toldo'
```

SEGUE ■

## CONFRONTO CON DATABASE (II)

### TABELLA GIOCATORE

Id	Nome	Cognome	Ruolo	Nazione	Squadra
1	Zinedine	Zidane	attaccante	Francia	1
2	Roberto	Carlos	terzino	Brasile	1
3	Francesco	Totti	attaccante	Italia	4
4	Gianluigi	Buffon	portiere	Italia	2
5	Francesco	Toldo	portiere	Italia	3

### FATTI PROLOG (Database giocatori modificato)

```
giocatori(1,'Zinedine', 'Zidane', attaccante, 'Francia',1).  
giocatori(2,'Roberto', 'Carlos', terzino, 'Brasile',1).  
giocatori(3,'Francesco', 'Totti', attaccante, 'Italia',4).  
giocatori(4,'Gianluigi', 'Buffon', portiere, 'Italia',2).  
giocatori(5,'Francesco', 'Toldo', portiere, 'Italia',3).
```

### TABELLA SQUADRE

Id	Nome	Città	Nazione
1	Real Madrid	Madrid	Spagna
2	Juventus	Torino	Italia
3	Inter	Milano	Italia
4	Roma	Roma	Italia

```
squadre(1, 'Real Madrid', 'Madrid', 'Spagna').  
squadre(2, 'Juventus', 'Torino', 'Italia').  
squadre(3, 'Inter', 'Milano', 'Italia').  
squadre(4, 'Roma', 'Roma', 'Italia').
```

### QUERY SQL

```
SELECT Giocatori.Nome,  
Giocatori.Cognome, Squadre.Nome  
FROM Giocatori INNER JOIN  
Squadre ON Giocatori.Squadra =  
Squadre.Id;
```

### QUERY PROLOG

```
?- giocatori( _,X,Y,_,_,Ids), squadre(Ids,N,_,_).
```

Domanda: se in corrispondenza di ...

QUERY  
SQL

```
SELECT Giocatori.Nome,  
       Giocatori.Cognome, Squadre.Nome  
FROM Giocatori INNER JOIN Squadre  
ON Giocatori.Squadra =  
   Squadre.Id;GioSquad.Id2=Squadre.Id;
```



QUERY  
PROLOG

```
?- giocatori(_,X,Y,_,_,I ds),  
   squadre(I ds,N,_,_).
```

allora a quale struttura dei database corrisponde una regola Prolog del tipo:

?



REGOLA  
PROLOG

```
?- squadra_di(X,Y,N):  
   giocatori(_,X,Y,_,_,I ds),  
   squadre(I ds,N,_,_).
```

## IL CUT !

### % Data base dei fatti

g(a). % Prima soluzione  
g(b). % Seconda soluzione  
g(c). % Terza soluzione

### % Prima modalità di ricerca di soluzioni: va bene la prima soluzione trovata

p1(a): -g(a), messaggio(1), !.  
p1(b): -g(a), g(b), messaggio(1), !.  
p1(c): -g(a), g(b), g(c), messaggio(1), !.  
p1(X): -messaggio(3), fail.

### % Seconda modalità di ricerca di soluzioni: esplora tutte le soluzioni

p2(a): -g(a), messaggio(2).  
p2(b): -g(a), g(b), messaggio(2).  
p2(c): -g(a), g(b), g(c), messaggio(2).  
p2(X): -messaggio(4).

### % Terzo modalità di ricerca di soluzioni: forza il fallimento per trovare tutte le soluzioni

p3(X): -g(X), write(' Soluzione: '), write(X), nl, fail.  
p3(X): -messaggio(4).

### % Messaggi

messaggio(1): -write('Questa unica soluzione va bene '), nl.  
messaggio(2): -write('Forse ci sono altre soluzioni - (digitare ;) '), nl.  
messaggio(3): -write('Nessuna soluzione per questo valore'), nl.  
messaggio(4): -write('Nessuna altra giustificazione'), nl, fail.

## RICERCARE SOLUZIONI E INSERIRLE IN LISTE

```
% Data base dei fatti
g(c,1). % Prima soluzione
g(a,4). % Seconda soluzione
g(f,5). % Terza soluzione
g(b,3). % Quarta soluzione
```

### IL PREDICATO findall

```
in_lista_disordinata1(L):-findall(X,g(X,_),L).
in_lista_disordinata2(L):-findall(J,g(_,J),L).
in_lista_disordinata3(L):-findall(g(X,J),g(X,J),L).
```

```
?- in_lista_disordinata1(L).
```

```
L = [c,a,f,b]
```

```
?- in_lista_disordinata2(L).
```

```
L = [1,4,5,3]
```

```
?- in_lista_disordinata3(L).
```

```
L = [g(c,1),g(a,4),g(f,5),g(b,3)]
```

### IL PREDICATO setof

```
in_lista_ordinata1(L):-setof(g(X,J),g(X,J),L).
in_lista_ordinata2(L):-setof(f(J,X),g(X,J),L).
in_lista_ordinata3(L):-setof(J,X^g(X,J),L).
```

```
?- in_lista_ordinata1(L).
```

```
L = [g(a,4),g(b,3),g(c,1),g(f,5)]
```

```
?- in_lista_ordinata2(L).
```

```
L = [f(1,c),f(3,b),f(4,a),f(5,f)]
```

```
?- in_lista_ordinata3(L).
```

```
L = [1,3,4,5]
```

# LE LISTE IN PROLOG

## Esempi di liste

`p([1,2,7,5]).`

`p([a,b,c,d,f]).`

`lista_x([luigi,ama,la,pesca]).`

`lista_mista([a,b,c,d],and,'Libri',[1,2,3],p(a,b),['Roma','Napoli']).`

`lista_vuota([]).`

`p1([a]).`

## Unificazione per le liste - il simbolo "|"

| ?- `p([C|R]).`      `C = 1 ,R = [2,7,5]` ; `C = a ,R = [b,c,d,f]`

| ?- `p1([C|R]).`      `C = a ,R = []`

# PROCESSAMENTO DI LISTE IN PROLOG

append/3 | ?- append([a],[b,c,f],L).

L = [a,b,c,f]

keysort/2

len/2

length/2 | ?- length([a,x1,b,f2,a,x2,b,f1],L).

L = 8

member/2 | ?- member(a, [d,g,j,a,k]).

yes

member/2 | ?- member(a, [d,g,j,[a,b],k]).

no

member/3 | ?- member(a, [d,g,j,a,b,c,k],N).

N = 4

member/3 | ?- member(a, [d,g,j,a,b,c,k,a],P).

P = 4 ; P = 8

remove/3 | ?- remove(a, [d,g,j,a,b,c,k,a],N). N = [d,g,j,b,c,k,a] ; N = [d,g,j,b,c,k]

removeall/3 | ?- removeall(a, [d,g,j,a,b,c,k,a],N).

N = [d,g,j,b,c,k]

reverse/2 | ?- reverse([f1,x2,b,x1,a],L).

L = [a,x1,b,x2,f1]

reverse/3

sort/2 | ?- sort([a,f,z,l,f],L).

L = [a,f,l,z]

sort/3

## RICORSIONE IN PROLOG

```
membro(X,[]):-messaggio(1,X),!,fail.
```

```
membro(X,[X|R]):-messaggio(2,X),!.
```

```
membro(X,[_|R]):-membro(X,R).
```

```
messaggio(1,X):-write(X - 'non è nella lista'),nl.
```

```
messaggio(2,X):-write(X - 'è nella lista'),nl.
```

```
| ?- membro(a,[s,f,g,h,a]).
```

```
a - è nella lista
```

```
yes
```

```
| ?- membro(x,[s,f,g,h,a]).
```

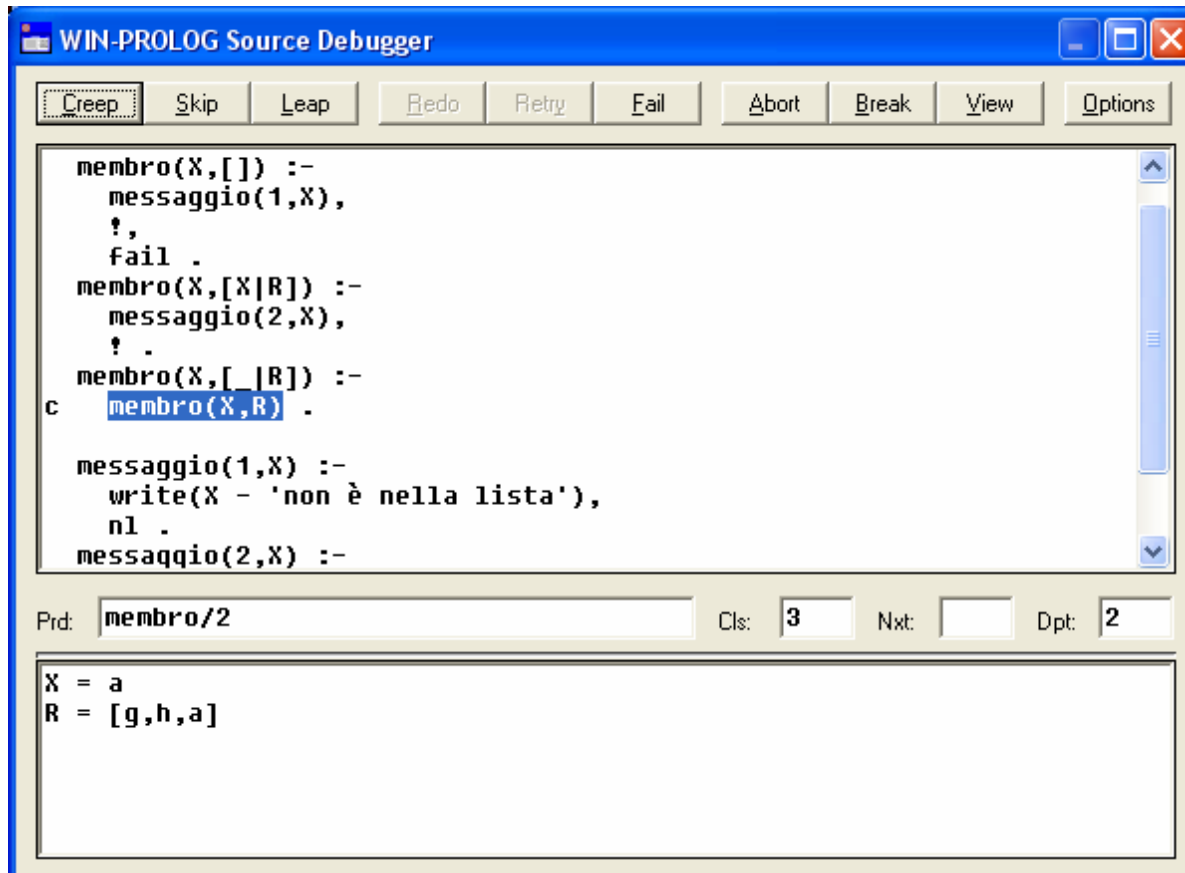
```
x - non è nella lista
```

```
no
```

## TRACE DEI PROGRAMMI

```
membro(X,[]):-messaggio(1,X),!,fail.  
membro(X,[X|R]):-messaggio(2,X),!.  
membro(X,[_|R]):-membro(X,R).
```

```
messaggio(1,X):-write(X - 'non è nella lista'),nl.  
messaggio(2,X):-write(X - 'è nella lista'),nl.
```



The screenshot shows the 'WIN-PROLOG Source Debugger' window. The title bar includes standard window controls. Below the title bar is a toolbar with buttons for 'Creep', 'Skip', 'Leap', 'Redo', 'Retry', 'Fail', 'Abort', 'Break', 'View', and 'Options'. The main text area contains the following Prolog code:

```
membro(X,[]) :-  
    messaggio(1,X),  
    !,  
    fail .  
membro(X,[X|R]) :-  
    messaggio(2,X),  
    ! .  
membro(X,[_|R]) :-  
c  membro(X,R) .  
  
messaggio(1,X) :-  
    write(X - 'non è nella lista'),  
    nl .  
messaggio(2,X) :-
```

Below the code area, there are fields for 'Prd:' (containing 'membro/2'), 'Cls:' (containing '3'), 'Nxt:' (empty), and 'Dpt:' (containing '2'). At the bottom, a variable assignment window shows:

```
X = a  
R = [g,h,a]
```

# Prolog - il parsing del linguaggio naturale, ..., ma riguarda il parsing in generale

## Una semplice grammatica per l'Inglese

sentence --> noun\_phrase, verb\_phrase.

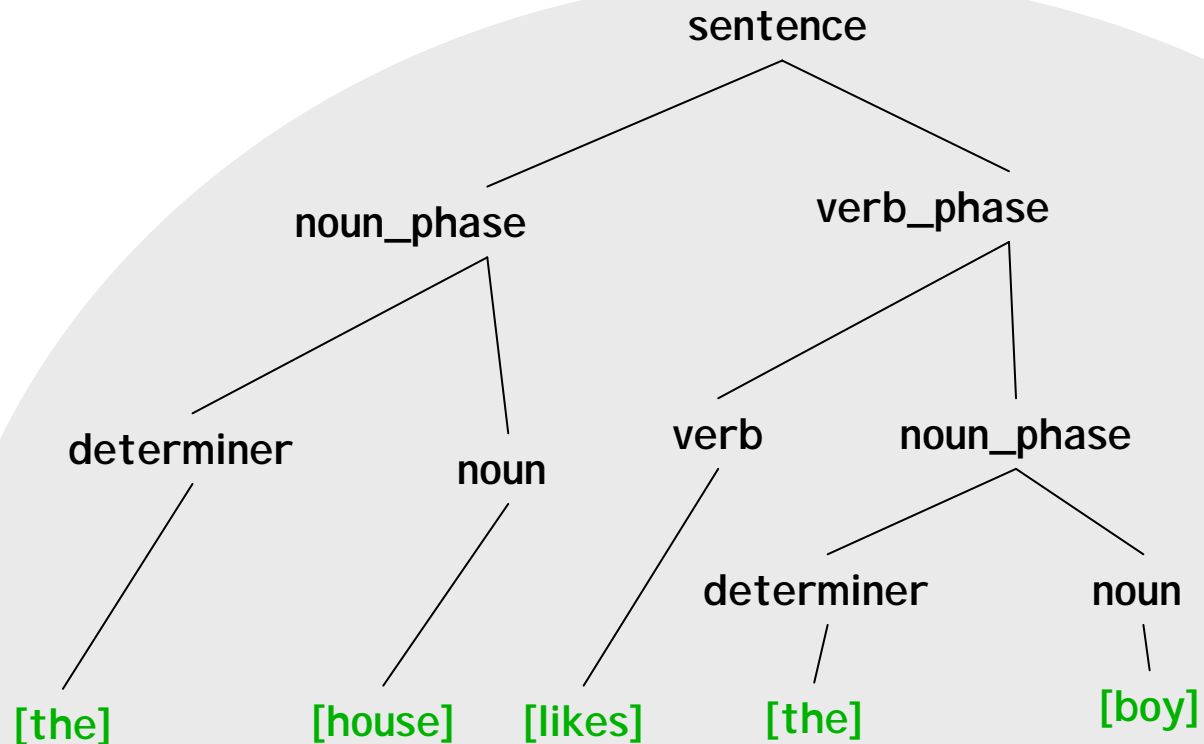
noun\_phrase --> determiner, noun.

verb\_phrase --> verb ; verb, noun\_phrase.

determiner --> [the].

noun --> [boy] ; [house].

verb --> [likes].



# Definite Clause Grammar (DCG) - il predicato built-in `phrase( Grammar, List )`

```
sentence --> noun_phrase, verb_phrase.  
noun_phrase --> determiner, noun.  
verb_phrase --> verb ; verb, noun_phrase.  
determiner --> [the].  
noun --> [boy] ; [house].  
verb --> [likes].
```

```
| ?- phrase(sentence, [the, boy, likes, the, house]).  
yes
```

Può essere usata per  
generare anche tutte le  
derivazioni della  
grammatica "sentence".

→ | ?- `phrase(sentence, S).`

S = [the, boy, likes] ;

S = [the, boy, likes, the, boy] ;

S = [the, boy, likes, the, house] ;

S = [the, house, likes] ;

S = [the, house, likes, the, boy] ;

S = [the, house, likes, the, house]

## **phrase**( Grammar, List ) per produrre la segmentazione di una frase ben formata della grammatica

sentence(sentence(N,V)) --> noun\_phrase(N), verb\_phrase(V).  
noun\_phrase(noun\_phrase(D,N)) --> determiner(D), noun(N).  
verb\_phrase(verb\_phrase(V,N)) --> verb(V), noun\_phrase(N).  
determiner(determiner(the)) --> [the].  
noun(noun(boy)) --> [boy].  
noun(noun(house)) --> [house].  
verb(verb(likes)) --> [likes].

| ?- **phrase**(sentence(X), [the, boy, likes, the, house]).

X =

sentence(noun\_phrase(determiner(the), noun(boy)), verb\_phrase(verb(likes), noun\_phrase(determiner(the), noun(house)))) ;

# SISTEMI E DOCUMENTAZIONE ASSOCIATA AL WIN-PROLOG

[DTM\\_API.PDF](#) - "*Datamining Toolkit*". This document gives an overview of datamining from LPA's perspective and describes each of the predicates making up the Datamining Toolkit.

[FLN\\_REF.PDF](#) - "*Flint Reference*". This document gives an overview of the various modes of uncertainty handling supported by Flint, namely fuzzy logic, Bayesian updating and certainty factors. It describes each of the **Flint** Toolkit predicates, the Fuzzy Editor and goes through both a *WIN*-PROLOG and a **flex** example.

[FLX\\_EGS.PDF](#) - "*flex Examples*". This document describes many of the examples supplied with the **flex** Expert System Toolkit.

[FLX\\_REF.PDF](#) - "*flex Reference*". This document describes the basic expert system constructs in **flex** - forward/backward chaining, questions and answers, frames and inheritance, rules, etc. It also describes **flex**'s Knowledge Specification Language (KSL). Note: [WIN\\_USR.PDF](#) gives a brief introduction of how to setup and use the **flex** Toolkit.

[FLX\\_TUT.PDF](#) - "*flex Tutorial*". This document is an introductory tutorial to the **flex** Expert System Toolkit. It is an excellent starting point for novices.

[INT\\_REF.PDF](#) - "*Intelligence Server*". This document tells you how to use the **Intelligence Server** Toolkit, giving examples of how to interface *WIN*-PROLOG (and **flex**) to code written in C, C++, Delphi, Java or Visual Basic.

[PDI\\_REF.PDF](#) - "*Prodata Interface*". This document describes each of the predicates of the **Prodata** Interface Toolkit.

[PPP\\_REF.PDF](#) - "*Prolog++ Reference*". This document describes the concepts of object-oriented programming (OOP) and how it benefits Prolog. Note:

[WIN\\_USR.PDF](#) gives a brief introduction of how to setup and use the **Prolog++** Toolkit.

[PWS\\_REF.PDF](#) - "*ProWeb Server User Guide*". This document is both a tutorial and a reference manual for the **ProWeb** Server Toolkit.

[TCP\\_REF.PDF](#) - "*TCP/IP Libraries*". This document describes the **TCP/IP** and **Agent** Toolkits.

[WIN\\_PRG.PDF](#) - "*Programming Guide*". This document is the *WIN*-PROLOG Prolog Programming Guide. It describes the Prolog syntax and Prolog predicates by category.

[WIN\\_REF.PDF](#) - "*Technical Reference*". This is the *WIN*-PROLOG Technical Reference Manual. It describes each of the *WIN*-PROLOG predicates alphabetically and by category. It also contains appendices on term comparison, initialisation and command line switches, window style names, window messages, the GraFiX language, programmatic manipulation of the Development Environment, error messages, memory management, time and file handling, compression, encryption, character sets and compilation.

[WIN\\_UPD.PDF](#) - "*Update Notes*". This document describes the changes between this and the last release of *WIN*-PROLOG (including the Toolkits).

[WIN\\_USR.PDF](#) - "*User Guide*". This document tells you how to install *WIN*-PROLOG. It describes the development environment and its menus. It also goes through the processes of debugging a program and creating a standalone application. It also tells you how to use the DDE, OLE, Dialog Editor, Call Graph and Cross Referencer extensions. It also gives a brief introduction of how to setup and use the **flex** and **Prolog++** Toolkits. It includes appendices on the *WIN*-PROLOG command line, memory management, compilation and file-application association.

# Sul sistema LPA Prolog (WiN Prolog)

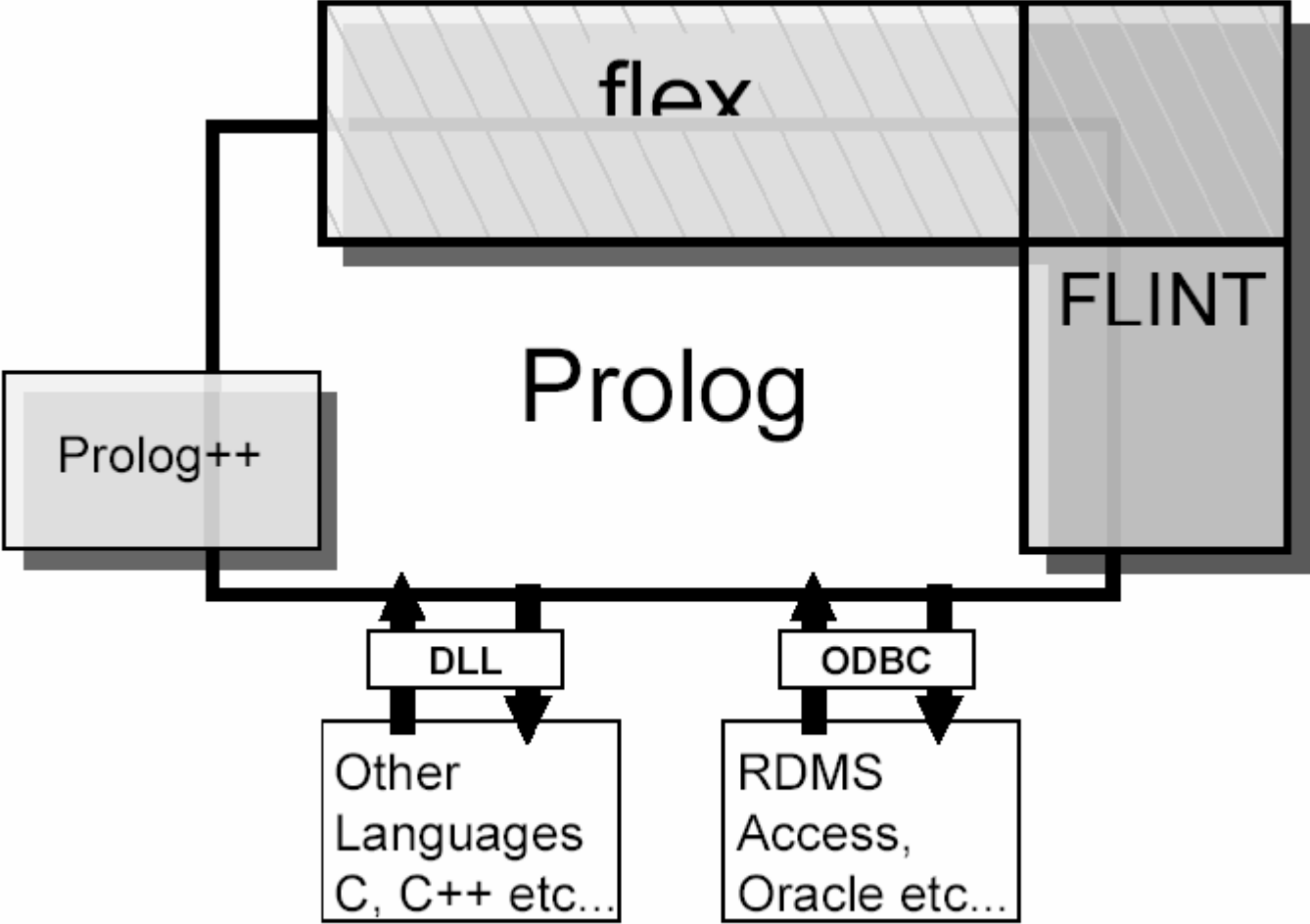


Figure 1 - The LPA suite of products

# FLEX:

- **Frame hierarchies**, where the data for an expert system can be represented as a number of linked frames, attributes and values. This enables the data to be defined in a structured way, where common attributes are inherited.
- **Forward-chaining**, this is an iterative process whereby at each stage a single rule is selected from a set of rules according to the current state of the data. The selected rule then generates a new state for the data and the whole process then repeats.
- **Backward-chaining**, this is a process whereby the truth of an overall goal is established by proving the truth of its sub-goals. This process may provide alternate solutions on request.
- **Questions and answers**, where flex programs can query the user by asking a question. The answer to the question is then stored for later reference.
- **Data-driven programs**, where programs are triggered automatically by creating, accessing or updating specified types of data.
- **Templates and synonyms**, where the KSL program for an expert system may be tailored to suit the language appropriate to that expert system's domain.

# Flint – Reasoning Under Uncertainty

Given a rule:

rule1: if A & B then C

there are 3 potential areas for uncertainty.

- Uncertainty in data (how true are A and B)
- Uncertainty in the rule (how often does A and B imply C)
- Impreciseness in general

The first two can be handled using techniques based on probability theory and the third using fuzzy logic.

Flint supports various modes of uncertainty handling, namely:

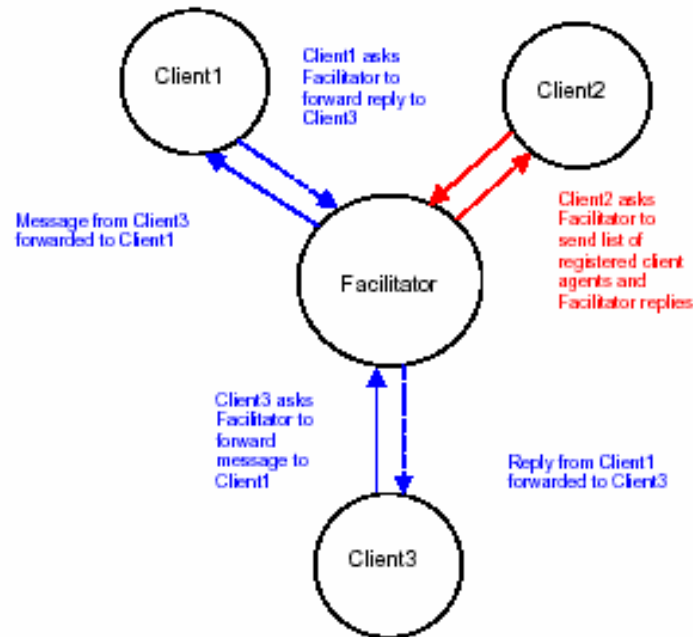
- Fuzzy Logic
- Bayesian updating
- Certainty factors

These are presented in a uniform and consistent way, using a common set of access routines.

These are made available in the context of a powerful AI programming environment supported by text editing facilities, compilers, debuggers and a variety of GUI tools.

This means you can integrate uncertainty with databases, spreadsheets, expert systems, graphics, etc.

# Agent Toolkit - KQML Agent



The agent architecture used here is that of a facilitator with multiple clients. The facilitator's role is a little like that of a telephone exchange – the Facilitator exists in order to route messages from client agents to other client agents, and also to provide information to clients as to which other agents are registered with it.