# Introduction to Prolog

Jean G. Vaucher
Incognito
Université de Montréal
January 1987

# PROLOG

**A programming language based on the formalism and the concepts of formal logic.**

- PROgrammation LOGique
- Robinson (resolution)
- Kowalski (Logic Programming)
- Colmerauer & Roussel ( PROLOG )
- Warren DEC-10 (Quintus)
- I.C.O.T. PSI Engine
- 5th Generation

# Declarative programming

- ☞Concise Notation
- ☞Programming by specification
- ☞Constraints
- ☞DB Programming
  Facts and Rules

# LOGIC PROGRAMMING BASICS

## Terms

- relationship between objects

```
loves (peter,jane)
```

## Variables & functional expressions

```
loves (X, mother(jane))
```

## STATEMENTS

```
<head> :- <body> .
```

- RULES

```
bird(B) :- flies(B), lays_eggs(B).
```

- FACTS (*no body* )

```
flies (sparrow).
```

- QUERY (*no head )*

```
:- loves(X,peter) , girl(X).
```

1) Prove that there is someone who loves Peter and is a girl
2) Find values of X so that the **terms** in the query match **facts** in the database

# Facts

```
parent (john, robert) .
```

- relationship (predicate) : parent
- objects: John, Robert

*John is a parent of Robert*

- interpretation

*A parent of John is Robert. ? ? ?*

- Arity

```
2    parent (john, robert) .
1    man ( john ) .
3    loves ( graham , simula , madly ) .
0    hot .
```

- Relational programming

# Program

Set of Facts
Database

```
likes ( fred , susan) .
likes ( mary , fred) .
likes ( fred , beer ) .
likes ( susan , fred ) .
likes ( jack , mary) .

man (jack) .
man (fred) .

woman  (susan).
woman  (mary).

drink (beer) .
```

# Queries

- Is fred a man ?

```
:- man (fred) .
     => ok
```

==> Is **man(fred)** TRUE ?
==> Is the *fact* **man(fred)** in the database ?

## Goal to be proved

- Does Fred like Mary ?

```
:- likes (fred , mary ) .
     => no
```

- Conjunction of goals

```
:- likes ( fred , beer )  ,  man ( fred ) .
     => ok
```

# Variables

- An unspecified individual
- Denoted by initial Capital letter

  - Who likes Mary ?

  ==> Is there an X such **likes( X ,mary)** can be found in the DB ?

```
:- likes ( X , mary )  .
      => X = jack
```

- Multiple answers

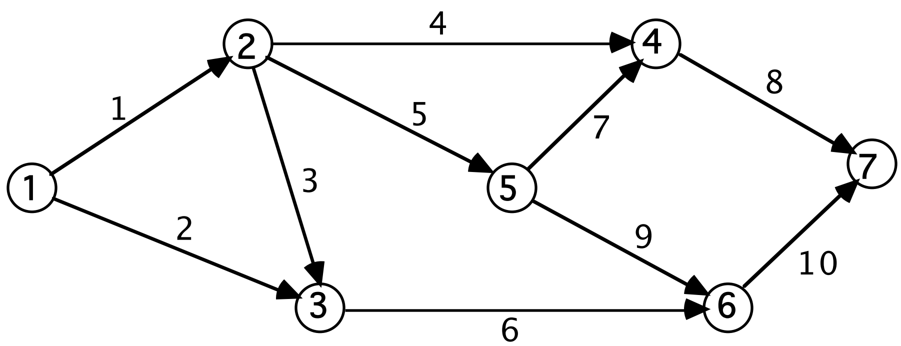
  - What does Fred like ?

```
:- likes ( fred , T ) .
  => T = susan ;
      T = beer
```

  - What girls does Fred like ?

```
:- likes ( fred , G ) , woman (G) .
  => G = susan
```

# PERT Example




**Prolog description**

arc ( Id, start-node, end-node, duration, resource)

```
arc (1,n1,n2,1,r1).
arc (2,n1,n3,4,r2).
arc (3,n2,n3,2,r3).
arc (4,n2,n4,3,r1).
arc (5,n2,n5,1,r2).
arc (6,n3,n6,4,r3).
arc (7,n5,n4,2,r1).
arc (8,n4,n7,3,r2).
arc (9,n5,n6,1,r3).
arc(10,n6,n7,4,r1).
```

# DATABASE PROGRAMMING

```
arc (1,n1,n2,1,r1).
arc (2,n1,n3,4,r2).
arc (3,n2,n3,2,r3).
arc (4,n2,n4,3,r1).
arc (5,n2,n5,1,r2).
arc (6,n3,n6,4,r3).
arc (7,n5,n4,2,r1).
arc (8,n4,n7,3,r2).
arc (9,n5,n6,1,r3).
arc(10,n6,n7,4,r1).
```

```
arc ( Id, start-node, end-node, duration, resource)
```

```
:- arc(4,n2,n4,_,_).              % Does arc4 link nodes 2 and 4 ?
    => OK

:- arc(6,_,_,D,R).                % What is the duration of activity 6
    => D=4, R=r3          %     and what resource does it use ?
```

Does activity 7 follows directly activity 5.

```
:- arc(5,_,N,_,_), arc(7,N,_,_,_).
    => N=5                  %  Yes, they meet at node 5.

:- arc(A,n5,_,_,_).               % What activities start at node 5 ?
    => A=7
       A=9
```

# PROLOG's Algorithm

$:-\ term_1, term_2, \ldots, term_n .$

equivalent to

```
for_all DB matches for term1 do
    for_all DB matches for term2 do
        . . . .
        for_all DB matches for termn do
            PRINT VALUES OF VARIABLES
            ( and optionally stop );
```

## BACKTRACKING

- When a term **fails**, Prolog uses another match for the previous term and tries again.

# Example

```
likes ( mary , fred) .
likes ( fred , susan) .
likes ( fred , beer ) .
likes ( susan , fred ) .
likes ( jack , mary) .

woman (susan).
woman (mary).
```

```
:- likes ( fred,G) , woman (G) .

likes ( mary , fred)
likes ( fred , susan) => woman(susan)
                         woman (susan)   ==> G=susan
                         woman (mary).

likes ( fred , beer ) => woman(beer)

                         woman (susan).
                         woman (mary).
likes ( susan , fred ) .
likes ( jack , mary) .
```

# Rules

Example:

```
bird (B) :- flies (B) , lays_eggs (B) .
```

<head> :- <tail>

1) *Head* is true IF *tail* is true
2) To prove *Head* , try to prove *tail*

Examples:

```
friends (A,B) :- likes (A,B) , likes (B,A).

person (P) :- woman (P) .
person (P) :- man (P) .

likes (M, mary) :- man (M) .
```

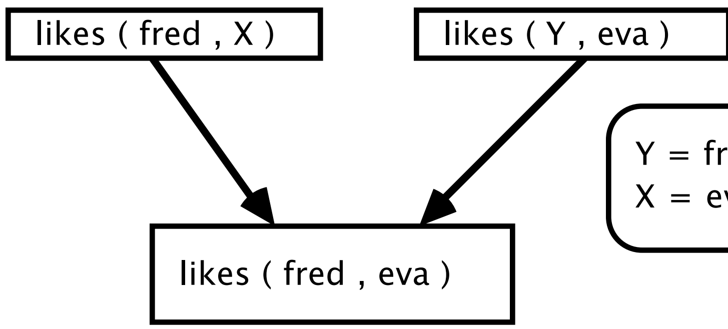
```
likes  (  mary  ,  fred)  .
likes  (  fred  ,  susan)  .
likes  (  fred  ,  beer  )  .
likes  (  susan  ,  fred  )  .
likes  (  jack  ,  mary)  .

woman  (susan).
woman  (mary).
```

```
:- friends (susan , F) .
 => F = fred

:- likes (F , mary) .
 => F = jack ;
    F = fred ;
    F = jack

:- person (X) .
 => X = susan ;
    X = mary ;
    X = jack;
    X = fred
```

# Unification

- Find constraints on variables so that 2 terms become identical

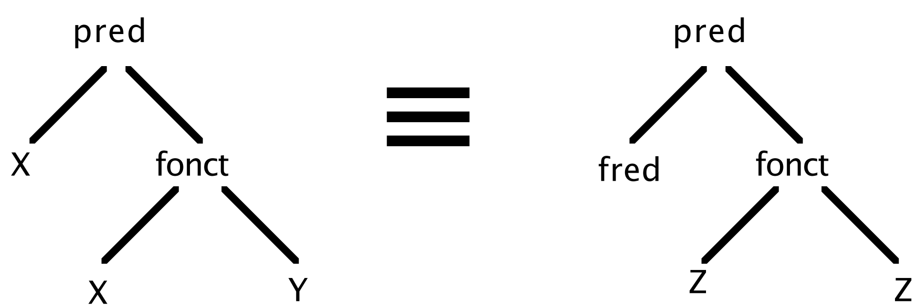
1




Y = fred
X = eva

2

pred (X, fonct (X,Y)) <==> pred (fred, fonct (Z,Z))




pred ( fred , fonct ( fred , fred) )

X = fred
X = Z
Z = Y

# Operators

## Infix Notation

- **1 + 2**
- **'+'(1,2)**

## Unary & Binary operators

- **op( 500 , yfx , + ) .**

  - Priority , arity, associativity

## Useful Operators

| | | |
|---|---|---|
| arithmetic: | **+ - * /** | |
| | **is :=** | *evaluable predicates* |
| comparison: | **> < =:=** | |

## Arithmetic

```
:-  X  is  4 + 5 ,  Y  is  ( X - 1 ) * 2 , 2 is 3-1.
      X = 9
      Y = 16
```

# "=" Equality ?

Unification operator

```
op ( 700 , xfx ,  =  ) .
= ( X , X) .


:-  f(1) = f(X) ,  =(Var,123) .
       X = 1
       Var = 123

:- X = 1 + 3 , Y is 1 + 3 .
       X = 1 + 3
       Y = 4
       A = 1

:- 4 = 1 + 3 .
       no.

:-  f (X , g( X , Y)) =  f (123 , g( Z , Z )) .
       X = 123
       Y = 123
       Z = 123

:-  A + 1  =  red + 1 .
       A = red
```

==

=

is

**:- X = 1 + 2 .**
X = 1 + 2

**:- X == 1 + 2 .**
no

**:- X  is  1 + 2 .**
X = 3

**:-  X = 1 ,  f ( 1 ) == f ( X ) .**
X = 1

**:-  X is  f ( 1 , 2 ) .**
no *or error*

# *Inequality*

| | |
|---|---|
| Not_unifiable | **\=** |
| Not Equal | **=:=** |
| Not Identical | **\==** |

```
:-  joe  \=  fred.
      ok

:-  X  \=  123  .
      no


:-  1 + 2  =:=  3  .
      no

:-  1 + 2  \==  3  .
      ok

:-  1 + 2  \=  3  .
      ok

:-  1 + 2  \=  X  .
      no
```

# Complex Unification

*(Zaniolo,,C. 1984  Object-oriented programming in Prolog)*

```
area( rect( H , W )  ,A)  :-  A is  H * W .
area( square (Side)  , A)  :-  area( rect(Side,Side), A)
.
area( triang (Side)  ,   A)  :-  A is  (H * W) / 2 .
```

**:- X = square(10) , area(X,Z) , write(area=Z) .**

```
area=100
ok
```

- selection of rule
- parameter passing
  - values in / out
- Note use of **=** to combine terms

# Input / Output

Prolog Terms

```
read (X)
write (X)
nl
```

Prolog syntax delimited by period

Characters

```
get  (C)
get0 (C)
put  (C)
```

Primitive but can be extended

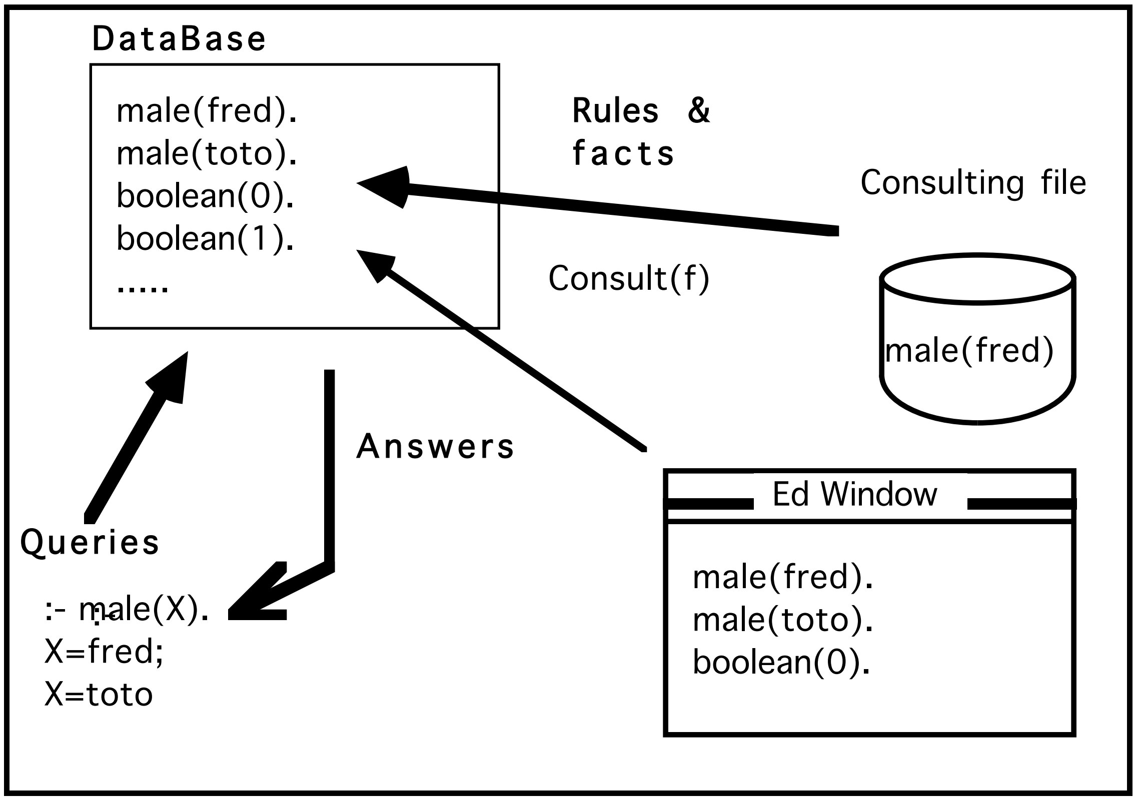
M-Prolog, Prolog II primitives

```
read_record(Str)
read_token(Token)
in-sentence(Sent,_)
```

```
:-    write('Input:  ') ,
      read(I),
      I2 is I+1,
      write(I2), nl .
```

```
Input:   123 .
124
ok
```

# Consulting and querying



# Assert & Retract

- To update the database during execution

```
:- male(X).
  X = fred ;
  X = toto

:- assert( male(john) ).

:- male(X).
  X = fred ;
  X = toto ;
  X = john

:- retract( male(toto) ).

:- male(X).
  X = fred ;
  X = john
```

- Asserta & assertz

# *Assignment*

## Pascal

```
Var x;
    x := 5;
    writeln (x);
    x := x+1;
```

## Prolog

```
assert(  value_of( x,5 ) ),
value_of( x,X ), writeln( X ),
retract( value_of( x,OldX )) ,
NewX  is OldX+1,
assert(  value_of( x,NewX   ) ).
```

# List Processing

# List Processing

Lists:

- Dynamic data structures
- Trees
- Graphs

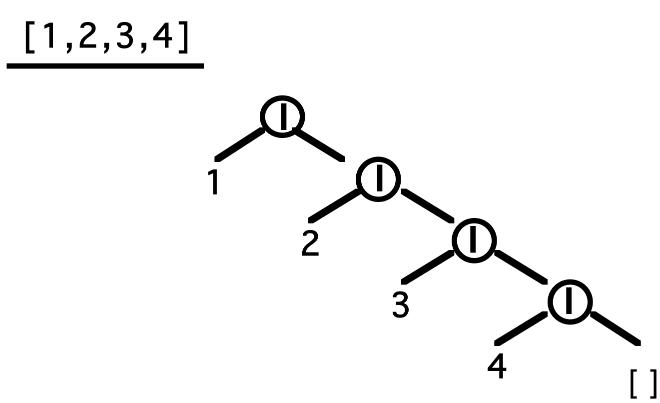
LISP: ( 1 2 3 4 )
nil
( 1 ((2) 3) 4 )

1) special symbol for empty list:
**nil or []**

2) binary nodes: **"|"** or **"."**

- Left = *first*
- Right = *rest of list*

PROLOG:

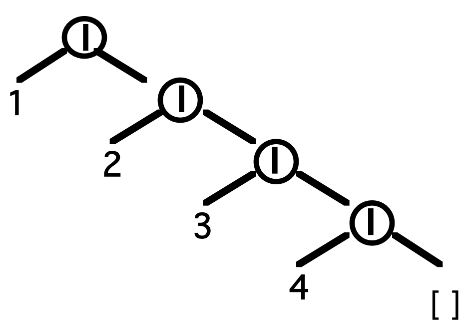


# *List Processing*

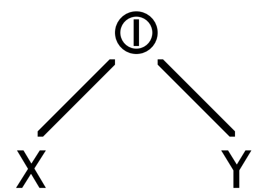
- Accessing elements
- Building lists

**[ 1,2,3,4 ] = [ X|Y ]**

[1,2,3,4]




[X|Y]




**=> X =1 , Y = [ 2,3,4 ]**

**:- [1,2,3 ] = [_,A| B] .**
**A = 2 , B =[3]**

# *Procedures for Lists*

## first (E,L)

E is the first element of the list L

```
first ( X , [X|_] ) .

:- first ( C , [red,white,blue] ) .
   C = red
```

## last (E,L)

E is the last element of the list L

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

:- last ( C , [red,white,blue] ) .
   C = blue
```

## member( E,L )

E is a member of the list L

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

:- member( red , [red,white,blue] ).
     ok
```

# *Using the Rules*

$\boxed{\text{member( E,L )}}$

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

- Generator

```
:- member( X , [red,white,blue] ).
      X = red ;
                member( X , [white,blue])
      X = white ;
                member( X , [blue])
      X = blue
```

- Constraints

```
:- L = [_,_,_] , member( fred,L ), member (3,L),
      last(zzz,L) .

      L = [fred,3,zzz] ;
      L = [3,fred,zzz]
```

# Logic Black Holes

- infinite loops after a few good answers
- generators of elements of infinite set

    -i.e. lists

Example

```
:- member( 1 ,L ) , L = [X,X] .
   L = [1,1] ;
   L = [1,1] ;
   ..... infinite loop ...
```

The first term generates all lists of which "1" is a member:

[1], **[1,..], [–,1],** [–,1,..], [–,–,1], [–,–,–,1]... etc ...
  ^          ^

Only two are acceptable, but the generator keeps providing candidates for the second term to reject.

# APPEND

## Prototypical Example of DECLARATIVE PROGRAMMING

- Procedure with multiple uses

# *APPEND*

append (L1, L2, L3)

L3 is the result of concatenating L1 and L2

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

## Checking

```
append( [1,2] , [3] , [1,2,3] ).
      |                              by rule 2
append( [2] , [3] , [2,3] ).
      |                              by rule 2
append( [] , [3] , [3] ).
      |
  Proved by rule 1
```

# *APPEND*

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

- Concatenating lists

```
:- append( [a,b] , [c,d] , L) .
      L = [a,b,c,d]
```

- Spliting lists

```
:- append(L1,L2 , [a,b,c] ).

      L1 = [] , L2 = [a,b,c]  ;
      L1 = [a] , L2 = [b,c]  ;
      L1 = [a,b] , L2 = [c]  ;
      L1 = [a,b,c] , L2 = []
```

## *APPEND*

- prefix( P,L ): P is a list of elements which prefix list L.

```
prefix( P,L ) :- append( P, _ , L).
```

- Sublist( S,L): S is a sublist of L

```
sublist( S,L ) :- prefix( P, L), append(_, S , P) .
```

- Naïve Reverse

```
reverse( [],[] ).
reverse( [X|Xs], Zs ) :-
    reverse( Xs,Ys), append( Ys, [X], Zs) .
```

# Permutations

- insert( E, L, L2 ): The list L2 is obtained by inserting E into the list L.

```
insert( E , [] , [E] ).
insert( E , [F|R] , [F|Rx] ) :- insert( E,R,Rx ).
```

- permute (L, Lp): The list Lp is a permutation of L

```
permute( [],[] ).
permute( [E|R], Lp) :-
     permute( R,Rp) , insert( E,Rp ).


:- permute( [ i, die, broke] , P ), writeln(P),fail.

     [i, die, broke]
     [die, i, broke]
     [die, broke, i]
     [i, broke, die]
     [broke, i, die]
     [broke, die, i]
```

# *CUT*

- Stops the combinatorial backtracking

```
:- term1, term2,     . . .  termn.

for  all  term1  do
     for  all  term2  do  ....   etc...
```

Examples:

```
max( A,B, M) :- A > B , M=A.
max( A,B, M) :- B>=A , M=B.
```

... , max(10,1,M), print(M), fail ...

```
M=10
M=1
```

or

```
digit(0).  digit(1).  ...   digit(9).
```

..., read(X), digit(X), ... fail ....

# *CUT (2)*

p ! or /

- within a clause prevents backtrack to goals to its left
- prevents use alternative clauses

CUT

```
f(...) :- .... ... .
f(...) :- .... , ... , ! , ... .
f(...) :- .... ... .
f(...) :- .... ... .
```

```
max(M, B, M) :- M<B, ! .
max(_, M, M) .          % else

one_of( P ) :- P , ! .

:- ... read(D) , one_of(digit(D)) , ...
```

# *Prolog and Logic Programming*

- Prolog is not Logic (programming)
    - Limited use of rules
    - Problems with
        - negative facts
        - disjunctive conclusions

**Modus ponens**

```
( A -> B )
& A        =>   B
```

**Modus tolens**

```
( A -> B )
& ¬ B           =>   ¬ A
```

Example:

1) Professors are poor
2) Paul is a professor
3) Peter is rich (not poor) (!!)

==>> Paul is poor
Peter is not a professor (!!)

4) Professors are devoted or crazy (!!)

# Implementations

## Edinburgh Prologs

DEC-10, C-Prolog
Quintus (speed champ ?)
BIM
Arity / AI Systems / Chalcedony (MAC / PC)
M-Prolog
Industrial strength

## Others

Prolog II (Apple II, VAX, MAC)
- different syntax - dif - freeze

Micro Prolog (LPA) early robust & fast Z80 -> IBM -> MAC
Borland's Turbo-Prolog (Type declarations)

## Parallelism & Object-Orientation

- Concurrent Prolog *parallelism*
- IC-Prolog, ParLog
- T-Prolog

- LOGLISP *Multiple languages*
- Smalltalk V

- POOPS, SIMPOOPS *Objects*

SIM ulation
P rolog
O bject
O riented
P rogramming
S ystem

- Natural integration
- Process Oriented Structure of SIMULA
- Knowledge manipulation of PROLOG
- Parallelism and Time

# Implementation

- 450 lines of Prolog
- built-in compiler

# *Single Server Queue*

```
class client (id);
methods
        seize(R)     :- wait(R,1).
        release(R)  :- send(R,1).
begin
        uniform(0,10,Ta),          hold(Ta),
        id(N), Nx is N+1, new_process (client, [id(Nx)] ),
                        write(N), writeln(" Waiting"),
        seize (res),
                        write(N), writeln(" Entering resource"),
        uniform(0,8,Ts),
        hold(Ts),
        release (res),
                        write(N), writeln (" Leaving system").
end

begin
        send(res,1),
        new_process( client, [id(1)] ),
        hold (20),
        writeln("Closing down system"),
        terminate.
end.
```

# Compiler Output

**CLASSES**

```
class(client)
class(main)
```

**CLASS PREDICATES**

```
class_predicate(client,seize)
class_predicate(client,release)
```

**OBJECT PREDICATES**

```
obj_pred(client,[id])
obj_pred(main,[])
```

**CLAUSES**

```
1: clauses(client,seize(_89),[wait(_89,1)],_90)
2: clauses(client,release(_81),[send(_81,1)],_82)
3: clauses(client, begin, [uniform(0,10,_70), hold(_70), id(_71),  _72 is   _71+1,
    new_process(client, [id(_72)]), write(_71), writeln( Waiting), seize(res),
    write(_71), writeln( Entering resource), uniform(0, 8,_73), hold(_73), release(res),
    write(_71), writeln( Leaving system)], _74)
4: clauses(main, begin, [send(res,1), new_process(client, [id(1)]), hold(20),
    writeln(Closing down system), terminate], _45)
```

# *Simple Queue Output*

```
1 Waiting
1 Entering resource
2 Waiting
3 Waiting
1 Leaving system
2 Entering resource
4 Waiting
2 Leaving system
3 Entering resource
3 Leaving system
4 Entering resource
Closing down system

*** THE END ***
CPU  : 0.95 sec
EVALS: 3779
FAILS: 453
```

# Conclusions

, Prolog - 5th Generation Language

, Declarative programming

, Power of Unification
    - pattern matching
    - rule selection

, No longer a research curiosity
    . Industrial language

, (SIM) POOPS : structured Prolog

, Knowledge and Processes

## Modelling for the future

# *References*

Clocksin, W.F. and Mellish, C.S., *Programming in Prolog,* 2nd Edition, Springer Verlag, New York, 1984.

Kowalski, R. "*Logic for problem solving",* North-Holland, 1979.

Sterling, L. and Shapiro, E. "The Art of Prolog", The MIT Press, 1986.

Giannesini F., Kanoui H., Pasero R., Van Caneghem M., "Prolog", InterÉditions, Paris, 1985.

Bratko, Ivan, "PROLOG, Programming for artificial intelligence", Addison-Wesley, 1986.