Logic programming

Imperative programming

Prolog

- Prolog: 'Programmation en Logique'
- 1974 - R.A. Kowalski: 'Predicate logic as a programming language', Proc IFIP 1974, pp. 569-574:
  - First-order predicate logic for the specification of data and relationships
  - Computation = logical deduction
- 1972 - A. Colmerauer, P. Roussel: first Prolog-like interpreter

Logic Programming (LP)

- R.A. Kowalski (Imperial College):

  Algorithm = Logic + Control

- Imperative languages (C++, Java):
  - data (what)
  - operations on data (how)
  - no separation between 'what' and 'how'

LP: Logic + Control

what → Logical Formulas

$\forall x\, (P(x) \rightarrow Q(x))$

how → Logical Deduction

$$\frac{\neg\psi,\ \varphi \rightarrow \psi}{\neg\varphi}$$

What: Problem Description

- *Horn clause:* $A \leftarrow B_1, B_2, \ldots, B_n$
- Equivalent to: $A \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$
- Meaning: A is true if
  - B_1 is true, and
  - B_2 is true, ..., and
  - B_n is true

What: Problem Description

$$A \leftarrow B_1, \ldots, B_n$$

- specification of **facts** concerning *objects* and *relations* between objects
- specification of **rules** concerning *objects* and *relations* between objects
- specification of **queries** concerning *objects* and *relations*

Problem Description

- Facts: $A \leftarrow$
- Rules: $A \leftarrow B_1,\ldots,B_n$
- Queries: $\leftarrow B_1,\ldots,B_n$

Meet the Royal Family

Example: Family Relations

- Facts: mother(juliana, beatrix) ←
  (constant)
- Rules:
  parent(X, Y) ← mother(X, Y)
  parent(X, Y) ← father(X, Y)
  (variable)
- Query: ← parent(juliana, beatrix)

Logic Program

mother(juliana, beatrix) ←
mother(beatrix, alexander) ←
father(claus, alexander) ←

parent(X, Y) ← mother(X, Y)
parent(X, Y) ← father(X, Y)

- Queries:
  ← parent(claus, alexander)
  ← parent(beatrix, juliana)

Prolog

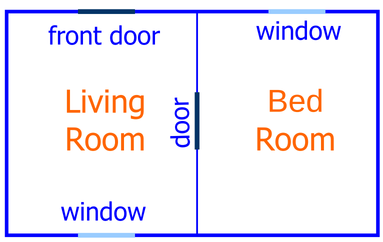
- Prolog: practical realisation of LP
- Prolog clause:

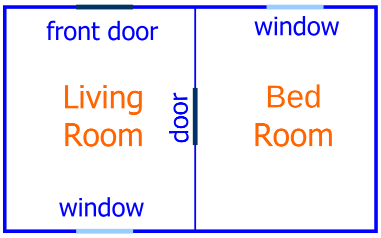
  A :- $B_1, B_2, \ldots, B_n$.
  head — body

- Example:
  mother(juliana, beatrix).
  parent(X, Y) :-
  mother(X,Y).
  :- parent(juliana, beatrix).

Why is Prolog so Handy?

Hotel suite design:





1. Living-room window opposite the front door
2. Bed-room door at right angle with front door
3. Bed-room window adjacent to wall with bed-room door
4. Bed-room window should face East

C-like

```
type dir = (north, south, east, west);
livrm(fd, lw, bd : dir) : boolean;
{
   livrm = opposite(fd, lw) and adjacent(fd, bd)
}
bedrm(bd, bw : dir) : boolean;
{
   bedrm = adjacent(bd, bw) and (bw = east)
}
suite(fd, lw, bd, bw : dir) : boolean;
{
   suite = livrm(fd, lw, bd) and bedrm(bd, bw)
}
```

Continued

```
for  fd = north to west do
     for lw = north to west do
         for bd = north to west do
             for bw = north to west do
                 if suite(fd, lw, bd, bw) then
                    print(fd, lw, bd, bw)
```

In Prolog

```
livrm(Fd, Bd, Lw) :-
   opposite(Fd, Lw), adjacent(Fd, Bd).
bedrm(Bd, Bw) :-
   adjacent(Bd, Bw), Bw = east.
suite(Fd, Lw, Bd, Be) :-
   livrm(Fd, Lw, Bd), bedrm(Bd, Bw).

:- suite(Fd, Lw, Bd, Bw).
```

Declarative Semantics

- Prolog clause: A :- $B_1, B_2, \ldots, B_n$.
- *Meaning*: A is true if
  - B_1 is true, and
  - B_2 is true, …., and
  - B_n is true

Procedural Semantics

- Prolog as a procedural language
- Prolog clause = **procedure**

  A :- B_1, B_2, …, B_n.

  procedure head — procedure body
- Query = procedure **call**

  :- B_1, B_2, …, B_n.

More General Programs

- Use often lists:

  [a, b, c , d] = [a | [b, c, d]]

  head tail

- Element is first element (fact):

  member(a, [a | [b, c, d]]).

- In general:

  member(X, [X|_]).

Set Membership

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

- Queries:

  :- member(a,[b,a,c])

  :- member(d,[b,a,c])

Example 1

```
/*1*/ member(X, [X|_]).    procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
          member(X, Y).
/*3*/ :- member(a, [a, b, c]).      call
```

Step 1

```
   :- member(a, [a, b, c]).
/*1*/   member(X, [X|_]).
```

Instantiation: X = a *match* with /*1*/

Example 2

```
/*1*/ member(X, [X|_]).    procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
          member(X, Y).
/*3*/ :- member(a, [b, a, c]).      call
```

Step 1

```
   :- member(a, [b, a, c]).
/*1*/   member(X, [X|_]).
```

Instantiation: X = a *no match* with /*1*/

Example 2 (continued)

```
/*1*/ member(X, [X|_]).    procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
          member(X, Y).
/*3*/ :- member(a, [b,a,c]).      call
```

Step 2

```
   :- member(a, [b, a, c]).
/*2*/   member(X, [_|Y]) :- member(X, Y).
```

Match: X = a; Y = [a, c]

Example 2 (continued)

```
/*1*/ member(X, [X|_]).    procedure entry
/*2*/ member(X, [_|Y]) :- procedure entry
          member(X, Y).
/*3*/ :- member(a, [b,a,c]).      call
```

Step 3

```
   :- member(a, [a, c]).          subcall
/*1*/   member(X, [X|_]).
```

Match: X = a

Matching

- A call and procedure head match if:
  - predicate symbols are equal
  - arguments in corresponding positions are equal
- Example:

```
        :- member(a, [a, c]).
/*1*/  member(a, [a|_]).
```

Variables & Atoms

```
mother(juliana, beatrix).
```

Calls:

```
:- mother(X, Y).
   X = juliana
   Y = beatrix
:- mother(_, _).    /* anonymous variable */
yes
:- mother(juliana, juliana).
no
```

Left-right Selection Rule

```
q.
r.
s.
p :- q, r, s.
```

Call:

```
:- p.
:- q, r, s.
:- r, s.
:- s.
```

Top-bottom Selection Rule

```
p(a).
p(b).
p(c).
p(X) :- q(X).
q(d).
q(e).
```

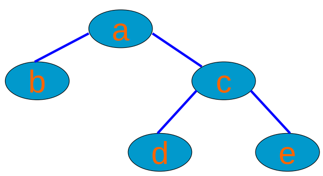
Call:

```
:- p(Y).
Y = a;
Y = b;
Y = c;
Y = d;
Y = e;
no
```

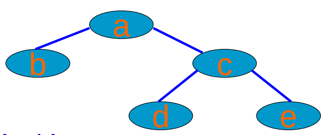
Backtracking

Backtracking: systematic search for alternatives

Example: search for paths in tree *T*

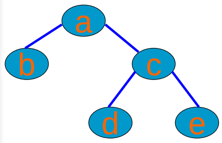


Backtracking




```
branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
    branch(X,Z), path(Z,Y).
```

Backtracking

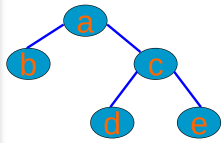


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

:- path(a, d). /* query */
path(a, d) :- <u>branch(a, Z)</u>, path (Z, d).

1 branch(a, Z) 1
Z = b
branch(a, b).

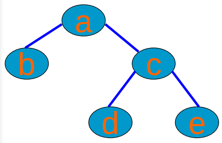
Backtracking



branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

:- path(a, d). /* query */
path(a, d) :- branch(a, Z), <u>path (Z, d)</u>.
Z = b 2
path(b, d)
2 X = b, Y = d
path(b, d) :- <u>branch(b, Z′)</u>, path(Z′, d).

Backtracking

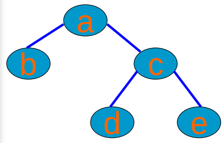


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

path(b, d) :- <u>branch(b, Z′)</u>, path(Z′, d).

3 branch(b, Z′) 3
backtrack

Backtrack Point

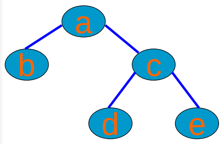


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

:- path(a, d). /* query */
path(a, d) :- <u>branch(a, Z)</u>, path (Z, d).

1′ branch(a, Z) 1′
Z = c
branch(a, c).

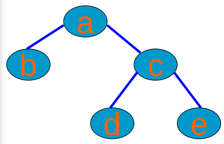
Backtracking



branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

:- path(a, d). /* query */
path(a, d) :- branch(a, Z), <u>path (Z, d)</u>.
Z = c 2′
path(c, d)
2′ X = c, Y = d
path(c, d) :- <u>branch(c, Z′)</u>, path(Z′, d).

Backtracking

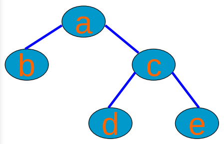


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

path(c, d) :- <u>branch(c, Z′)</u>, path(Z′, d).

3′ branch(c, Z′) 3′
Z′ = d
branch(c, d)

Backtracking



branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

path(c, d) :- branch(c, Z'), path(Z', d).
Z' = d   4
path(d, d)
4   X = d
path(d, d)

Terminology

- From programming languages (Prolog as procedural language):
  nat(0).
  nat(s(X)) :- nat(X).
  - term: nat(0), nat(s(X)), nat(X), :-(nat(s(X)), nat(X)), s(X), 0, X
  - functor: s, nat, :-
  - principal functor: nat in nat(s(X)), :- in :-(nat(s(X)), nat(X)), s in s(X)
  - number: 0
  - variable: X

Inversion of Computation (1)

- **Example:** concatenation of lists
  U = V ° W
  with U, V, W lists and ° concatenation operator
- Usage:
  - [a, b] = [a] ° **W** ⇒ **W** = [b]
  - [a, b] = **V** ° [b] ⇒ **V** = [a]
  - **U** = [a] ° [b] ⇒ **U** = [a, b]
  - [a, b] = **V** ° **W?**

Inversion of Computation (2)

- Prolog concatenation of lists:
  concat([], U, U).
  concat([X|U], V, [X|W]) :-
    concat(U, V, W).
- concat as constructor:
  ?- concat([a, b], [c, d], X).
  X = [a, b, c, d]
- concat used for decomposition:
  ?- concat(X, Y, [a, b, c, d]).
  X = []
  Y = [a, b, c, d]

Inversion of Computation (3)

- concat used for decomposition:
  ?- concat(X, Y, [a, b, c, d]).
  X = []
  Y = [a, b, c, d];
  X = [a]
  Y = [b, c, d];
  X = [a, b]
  Y = [c, d];

  ...

Order of Clauses (1)

- LP: order is irrelevant
- Prolog: order may be relevant
- Example:

  member(X,[_|Y]) :-
    member(X,Y).
  member(X,[X|_]).
  :- member(a,[b,a,c]).

Order of Clauses (2)

```
/*1*/  member(X, [_|Y]) :-
            member(X, Y).
/*2*/  member(X, [X|_]).

?- member(a, [a,b]).
   X = a, Y = [b]             match with 1
   ?- member(a, [b]).         next call
      X' = a, Y' = []         match with 1
      ?- member(a, []).       fail 1 and 2
   fail 1 and 2
   fail 1, backtracking to 2
   X = a                      match 2
yes! (but not efficient)
```

Order of Clauses (3)

```
/*1*/  member(X, [_|Y]) :-
            member(X, Y).
/*2*/  member(X, [X|_]).

?- member(X, [a, b]).
   X' = X, Y = [b]            match with 1
   ?- member(X', [b]).        next call
      X" = X', Y' = []        match with 1
      ?- member(X", []).      fail 1 and 2
   X'  = b                    fail 1, match 2
   X = b;                     backtracking
   X = a                      match 2
yes! (but not efficient)
```

Order of Clauses (4)

```
/*1*/  member(X, [_|Y]) :-
            member(X, Y).
/*2*/  member(X, [X|_]).

?- member(a, Z).
   X = a, Z = [_|Y]           match 1
   ?- member(a, Y).           next call
      X' = a, Y= [_|Y']       match 1
      ?- member(a, Y').       next call
            :
```

Stack overflow

Conclusions Order of Clauses

- LP: order clauses is irrelevant
- Prolog:
  - Order has effect on efficiency of program
  - Order may affect termination: terminating program + order change ≠ terminating program

Order of Conditions (1)

- Length of list with successor function
  s : N → N, with s(x) = x + 1
- Program:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
          length(X, M),
          N = s(M).
```

- Use:

```
?- length([a, b], N).
N = s(s(0))
```

Order of Conditions (2)

- Program:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
          length(X, M),
          N = s(M).
```

- Use:

```
?- length(L, s(0)).
L = [_A];
```

Stack overflow

Order of Conditions (3)

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
          length(X, M),
          N = s(M).
```

- Trace:

```
?- length(L, s(0)).
   L = [_A|X], N = s(0)                   match 2
   ?- length(X, M), s(0) = s(M).          subcall
      X = [], M = 0                       match 1
   ?- s(0) = s(0).                        match
   L = [_A];                              backtracks
      …                                   (1 fails)
```

Order of Conditions (4)

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
          length(X, M),
          N = s(M).
```

- Trace:

```
?- length(L, s(0)).
   L = [_A|X], N = s(0)                   match 2
   ?- length(X, M), s(0) = s(M).          subcall
      X = [_B|X'], N = M                  match 2
   ?- length(X', M'), M = s(M'),          subcall
      s(0) = s(M).
         ...
```

Order of Conditions (5)

- Program:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
          N = s(M),
          length(X, M).
```

- Use:

```
?- length(L, s(0)).
L = [_A];

no
```

Declarative vs Procedural

- Order of clauses and conditions in clauses in Prolog progams may be changed, but
- This may be at the expense of:
  - loss of termination
  - compromised efficiency
- Schema for procedural programming:
  - special case first (top, left)
  - general case (e.g. including a recursive call) last (bottom, right)

Fail & Cut

- Notation: fail and !
- Control predicates: affect backtracking
- Used for:
  - efficiency reasons
  - implementing tricks

Enforcing Backtracking: fail

- ?- fail.
  no   (no match)
- Program:

```
p(a).
p(b).
```

Query:

```
?- p(X).          (match)
X = a
yes
```

Fail - no Recursion

- Program:
  - p(a).
  - p(b).
  - p(X) :- q(X).
  - q(c).
- Query:
  - ?- p(X), write(X), nl, fail.
  - a
  - b
  - c   **backtracking**
  - no

Fail - with Recursion

- Program:
  - /*1*/ member(X, [X|_]).
  - /*2*/ member(X, [_|Y]) :- member(X,Y).
- Query/call:
  - ?- member(Z,[a,b]), write(Z), nl, fail.
  - Z = X, X = a   match 1
  - ?- write(a), nl, fail.
  - a   **backtracking**
  - ?- member(Z,[a,b]), write(Z), nl, fail.
  - Z = X, Y = [b]   match 2

Controlling Backtracking: !

- Procedural meaning of the cut !:

A :- B1, B2, !, B3, B4.

Search for alternatives (B1, B2)

Search for alternatives (B3, B4)

! **Stop searching**

Cut

Program:

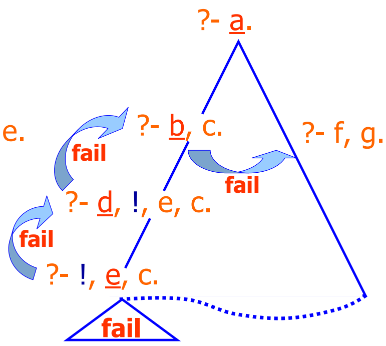
p(a).
p(b).
q(X) :- p(X), r(X).
r(Y) :- !, t(Y).
r(a).
t(c).

Execution:

?- q(Z).
Z = X
?- p(X), r(X).
X = a
?- r(a).
Y = a
?- t(a).
fail, no
! backtracking
to r(a).
Try X = b

State Space and !

a :- b, c.
a :- f, g.
...
b :- d, !, e.
b :- ... .
...
d.



Various Applications of !

- Cut as commitment operator:

if X < 3 then Y = 0
if X ≥ 3 and X < 6 then Y = 2
if X ≥ 6 then Y = 4

- Prolog:

```
t(X, 0) :- X < 3.
t(X, 2) :- X >= 3, X < 6.
t(X, 4) :- X >= 6.
```

Commitment Operator

- Cut as commitment operator:

  /*1*/ t(X, 0) :- X < 3.
  /*2*/ t(X, 2) :- X >= 3, X < 6.
  /*3*/ t(X, 4) :- X >= 6.
- Execution trace:

  ?- t(1, Y), Y > 2. match 1
  ?- 1 < 3, 0 > 2.
  ?- 0 > 2. fail 1
  ?- 1 >= 3, 1 < 6, 1 > 2. match 2
  ?- … fail 2
  ?- 1 >= 6, 4 > 2. match 3, fail 3 ☹

Commitment Operator

- Cut as commitment operator:

  /*1*/ t(X, 0) :- X < 3, !.
  /*2*/ t(X, 2) :- X >= 3, X < 6, !.
  /*3*/ t(X, 4) :- X >= 6.
- Execution trace:

  ?- t(1, Y), Y > 2. match 1
  ?- 1 < 3, !, 0 > 2.
  ?- !, 0 > 2. fail 1
  no ☺

Various Applications of !

- Cut used for removal of conditions:

  min(X, Y) is X if X ≤ Y
  min(X, Y) is Y if X > Y
- Prolog:

  min(X, Y, X) :- X =< Y.
  min(X, Y, Y) :- X > Y.
- Execution:

  ?- min(3, 5, Z).
  ?- 3 =< 5. match 1
  Z = 3 yes

Removal of Conditions

- Cut used for removal of conditions:

  min(X, Y, Z) :-
  X =< Y, !,
  Z = X.
  min(X, Y, Y).
- Execution:

  ?- min(3, 5, W).
  ?- 3 =< 5, !, W = 3. match 1
  W = 3 yes

Removal of Conditions

- Cut used for removal of conditions:

  min(X, Y, Z ⇒ X) :-
  X =< Y, !.
  Z = X. *why included?*
  min(X, Y, Y).
- Execution:

  ?- min(3, 5, 5).
  fail 1, match 2
  yes

Change in Meaning?

- Cut used for removal of conditions:

  min(X, Y, Z) :-
  X =< Y, *(! omitted)*
  Z = X.
  min(X, Y, Y).
- Execution:

  ?- min(3, 5, W), W = 5.
  ?- 3 =< 5, 5 = 3, W = 5. match 1
  ?- 5 = 3, W = 5. fail
  ?- W = 5. match 2 (with Y = 5 = W)
  W = 5
  yes

Negation by Failure

- Simulation of negation: not(p) is true if p is false (fails):

```
not(X) :- call(X), !, fail.
not(X).
```

- **Example:**

```
p(a).
q(X) :- p(X), not(r(X)).
r(c).
?- q(Y).
yes
```

Single Solution

- Circumvention of double search:

```
/*1*/  member(X, [X|_]) :- !.
/*2*/  member(X,[_|Y]) :-
           member(X,Y).
```

- **Example:**

```
?- member(a, [a, b, a]).
yes

?- member(X, [a, b]).
X = a;
no
```

Green and Red Cuts

- **Green cut:**
  - when omitted, does not change declarative (logical) meaning of program
  - used to increase efficiency
- **Red cut:**
  - when omitted, declarative meaning of program is changed
  - used for efficiency
  - used to enforce termination

Green and Red Cuts

- **Green cut:**
  - commitment operator
- **Red cut:**
  - removal of conditions
  - cut-fail combination (see notes)
  - single solution

Prolog Database

- The working environment of Prolog, containing all loaded Prolog programs is called: the 'database'
- The database can be manipulated by the programs themselves
- Compare: Pascal program that modifies itself during execution

Prolog 'Database'

add new clauses ⇨

remove clauses ⇦

Prolog Database

```
parent(jim, bob).

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

Prolog 'Database'

assertz: add to the end of a definition

assertz(parent(bob,ann)). ⇨

Prolog Database

```
parent(jim, bob).
parent(bob,ann).
pred(X,Y) :-
    parent(X,Y).
pred(X,Y) :-
    parent(X,Z),
    pred(Z,Y).
```

Asserting Clauses

Database

```
collect_data(stop).
collect_data(_) :-
    write('Next item: '),
    read(X),
    assertz(X),
    collect_data(X).

input_data :-
    collect_data(start).
```

```
name(peter).
age(35).
stop.
```

```
?- input_data.
Next item: name(peter).
Next item: age(35).
Next item: stop.
```

Database Manipulation

- Asserting (new) clauses:
  - assert(C): position C unspecified
  - asserta(C): at the beginning of the definition of the predicate
  - assertz(C): at the end of the definition of the predicate
- Deleting clauses:
  - retract(C): remove clause matching with C (top to bottom order)

Retracting Clauses

retract: remove from the beginning of the definition

```
?- retract(parent(X,Y)).
   X = jim
   Y = bob
yes
```

Prolog Database

```
?- dynamic parent/2.
parent(jim, bob).
parent(bob,ann).
parent(john,pete).
parent(pete,linda).
```

Retracting Clauses

```
?- retract_all_facts(parent(X,Y)).
yes
```

Prolog Database

```
?- dynamic parent/2.
⇨ parent(jim, bob).
⇨ parent(bob,ann).
⇨ parent(john,pete).
⇨ parent(pete,linda).

retract_all_facts(X) :-
    retract(X),
    fail.
retract_all_facts(_).
```