CS3510: Artificial Intelligence

- Programming in Prolog:
  - 4 weeks
  - Lecturer: Dr Peter Lucas
- Theory and practice of Artificial Intelligence:
  - 8 weeks
  - Lecturer: Prof Jim Hunter

Structure of Course

- Lectures (Monday & Friday):
  - logic programming
  - Prolog programming principles
  - design of Prolog programs
- Practicals (Monday & Tuesday):
  - in 3 groups
  - exercises
  - 1 assignment: development of small Prolog programs for problems

Learning Outcomes

- 4 weeks Prolog:
  - Know principles of logic programming
  - Know syntax, semantics and pragmatics of Prolog programming language
  - Be able to develop small Prolog programs

Prolog

- Prolog = 'Programmation en Logique'
- Popular language in:
  - Artificial Intelligence (AI)
  - Programming language design (ASF, Goedel, Theorist)
  - Rapid prototyping

Origin

- 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 relations among data
  - 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 (Pascal, 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*: $A \leftarrow B_1, B_2, \ldots, B_n$
- 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:

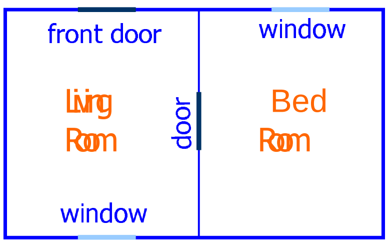
$$A \text{ :- } \underbrace{B_1, B_2, \ldots, B_n}_{\text{body}}.$$

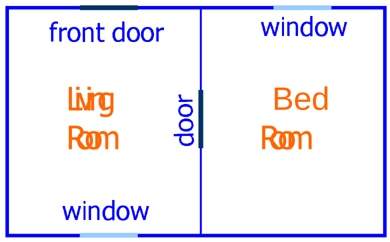
head → A

- 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

In Pascal

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

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 → A; procedure body = B_1, B_2, …, B_n
- Query = procedure **call**

  :- B_1, B_2, …, B_n.

More General Programs

- Use often lists:

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

  head = a; tail = [b, c, d]
- 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).
```

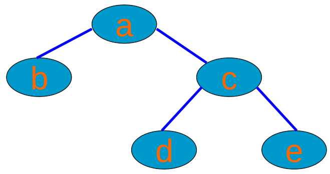
Call:

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

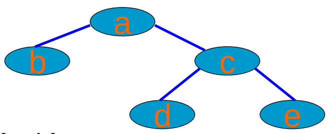
Backtracking

Backtracking: systematic search for alternatives

Example: search for paths in tree *T*

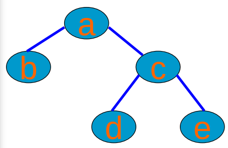


Backtracking




```
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 */

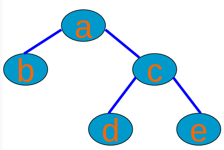
path(a, d) :- <u>branch(a, Z)</u>, path (Z, d).

branch(a, Z) 1

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>.

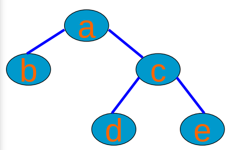
Z = b 2

path(b, d)

2 X = b, Y = d

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

Backtracking



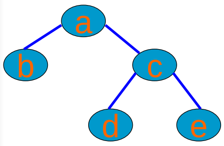

```
branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
    branch(X, Z), path(Z, Y).
```

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

3 branch(b, Z') 3

backtrack

Backtrack Point

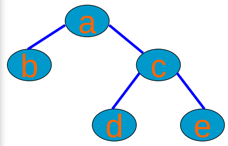


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

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

1′ branch(a, Z)
Z = c
branch(a, c).

Backtracking

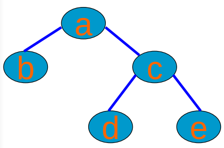


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

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

2′ Z = c
path(c, d)
X = c, Y = d
path(c, d) :- branch(c, Z′), path(Z′, d).

Backtracking

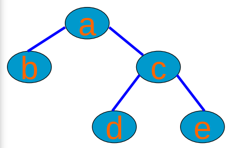


branch(a, b).
branch(a, c).
branch(c, d).
branch(c, e).
path(X, X).
path(X, Y) :-
  branch(X, Z), path(Z, Y).

path(c, d) :- branch(c, Z′), path(Z′, d).

3′ branch(c, Z′)
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).

4 Z′ = d
path(d, d)
X = d
path(d, d)