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

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

  - First-order predicate logic for the specification of data and relations among data
  - Computation = logical deduction

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’
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

### Problem Description
- **Facts:** $A \leftarrow$
- **Rules:** $A \leftarrow B_1, B_2, \ldots, B_n$
- **Queries:** $\leftarrow B_1, B_2, \ldots, B_n$

### Example: Family Relations
- **Facts:** $\text{mother(juliana, beatrix)} \leftarrow$
- **Rules:**
  - $\text{parent}(X, Y) \leftarrow \text{mother}(X, Y)$
  - $\text{parent}(X, Y) \leftarrow \text{father}(X, Y)$
- **Query:** $\leftarrow \text{parent(juliana, beatrix)}$
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:

\[ A : \neg B_1, B_2, \ldots, B_n \]

head body

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

Why is Prolog so Handy?

Hotel suite design:

<table>
<thead>
<tr>
<th>front door</th>
<th>window</th>
</tr>
</thead>
<tbody>
<tr>
<td>living room</td>
<td>door</td>
</tr>
<tr>
<td>bed room</td>
<td>window</td>
</tr>
</tbody>
</table>

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 east do
  if suite(fd, lw, bd, bw) then
    print(fd, lw, bd, bw)
**In Prolog**

\[
\text{livrm(Fd, Bd, Lw) :-}
\]
\[
\quad \text{opposite(Fd, Lw), adjacent(Fd, Bd).}
\]
\[
\text{bedrm(Bd, Bw) :-}
\]
\[
\quad \text{adjacent(Bd, Bw), Bw = east.}
\]
\[
\text{suite(Fd, Lw, Bd, Be) :-}
\]
\[
\quad \text{livrm(Fd, Lw, Bd), bedrm(Bd, Bw).}
\]
\[
\text{:- suite(Fd, Lw, Bd, Bw).}
\]

**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**
- Query = **procedure call**

**More General Programs**

- Use often lists:
  \[ [a, b, c, d] = [a | [b, c, d]] \]
- Element is first element (fact):
  \[ \text{member(a, [a | [b, c, d]])}. \]
- In general:
  \[ \text{member(X, [X |_])}. \]

**Set Membership**

\[ \text{member(X, [X |_])}. \]
\[ \text{member(X, [_|Y]) :- member(X, Y)} \]

- Query:
  \[ \text{:- member(a,[b,a,c])} \]
  \[ \text{:- member(d,[b,a,c])} \]

**Example 1**

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

**Step 1**

\[ \text{:- member(a, [a, b, c])}. \]
\[ /*1*/ member(X, [X |_]). \]

**Instantiation:** \( X = \text{a match with } /*1*/ \)
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

Call:

q
r
s
p:-q,r,s

*: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`

![Diagram of a tree with paths](image)

1. `branch(a, b)`.  
2. `branch(a, c)`.  
3. `branch(c, d)`.  
4. `branch(c, e)`.  
5. `path(X, X)`.  
6. `path(X, Y) :- branch(X, Z), path(Z, Y)`.  

1. `:- path(a, d). /* query */`  
2. `path(a, d) :- branch(a, Z), path(Z, d)`.  
3. `branch(b, Z)`.  
4. `path(b, d) :- branch(b, Z), path(Z, d)`.  
5. `path(b, d) :- branch(b, Z), path(Z, d)`.  
6. `branch(b, Z)`.  
7. `call backtrack`
**Backtrack Point**

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

1. **branch(a, Z)**
   - Z = c
   - branch(a, c).

**Backtracking**

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

1. **branch(a, Z)**
   - Z = c
   - branch(a, c).

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

3. **branch(c, Z')**
   - Z' = d
   - branch(c, d).

4. **path(d, d)**

---

**Backtrack Point**

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

1. **branch(a, Z)**
   - Z = c
   - branch(a, c).

2. **branch(a, Z)**
   - Z = c
   - branch(a, c).

3. **branch(c, Z')**
   - Z' = d
   - branch(c, d).

4. **path(d, d)**

---

**Backtracking**

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

1. **branch(a, Z)**
   - Z = c
   - branch(a, c).

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

3. **branch(c, Z')**
   - Z' = d
   - branch(c, d).

4. **path(d, d)**

---