Introduction to Prolog

Properties of Prolog as a Programming language:

• no explicit types or classes
• rule-based, founded on first-order logic
• high expressibility: functionality per program line
• interactive, experimental programming

NB: A few examples in these ppt slides differ from note, sorry 'bout that, but I had some nice animations prepared... :)}
Background for Prolog

**PROgramming in LOGic**

**Syntax:** subset of 1.-order logic

**Declarative semantics:** Logical consequence

**Procedural semantics:**

Resolution, proof rule with unification; Robinson, 1965

A.Colmerauer & co. (Marseille), ca. 1970: "Prolog"


**Language made known by** R.Kowalski "Logic for Problem solving", 1979, ....
Prolog and AI

• First major AI language was LISP, McCarthy & al., 1960
  – symbolic computation
  – programs ≈ data
• Prolog, intended for computational linguistics, has become a successor of LISP for AI applications
• A Prolog program is representation of knowledge ≈ a database (relational DB + a lot more)
• Prolog applies backward-chaining (cf. MN, chap 2).
• Prolog includes strong metaprogramming facilities (programs ≈ data; easy to defining interpreters)
Later in the course, extensions to Prolog

- Constraint Handling Rules
- allows to mix forward and backward chaining...
- Abductive logic programming

But now, let's jump into basic Prolog
Program is a *description* of data

```
parent( pam, bob). % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```
Basic notions:

• predicates: **parent**
  – describes a relation
  – defined by facts, **rules**, collectively called clauses

• constant (symbol)s: **tom**, **bob**, **x**, **y**

• variables: **X**, **Y**, **Tom**

• atoms (simple goals): **parent**(**A**, **a**)  

• Queries....

In Prolog literature, constants are called atoms :)
Queries

Atomic queries

?- parent(X,Y).
... give me values of X and Y so parent(X,Y) logically follows from program

Compound query

?- parent(pam, X), parent(X, Y).
... give me x and y, so that...
Procedural semantics

parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

?- parent(pam, X), parent(X, Y).
X=bob
?- parent( bob, Y).
Y=ann
Success!

Other solutions?
Y=pat
Success!

No more possible solutions here :(

Unification term=term?
• from left to right
• from start to end
• backtracking
≈ undo and try new choices

Other Solutions?
Rules

female(pam).
male(tom).
male(bob).
female(liz).
female(pat).
female(ann).
male(jim).

mother(X, Y):-
    parent(X, Y),
    female(X).

Procedural semantics
as before + rewrite subgoal using rules

Declarative semantics \approx logical consequence

with rules read as, e.g.

\( \forall x, y, x: p(x, y) \land f(x) \rightarrow m(x, y) \)

The nice property:
procedural \approx declarative
(unless procedural semantics loops)
A recursive rule

\[
\text{ancestor}(X, Z) :\text{-}
\]
\[
\text{parent}(X, Z).
\]

\[
\text{ancestor}(X, Z) :\text{-}
\]
\[
\text{parent}(X, Y),
\]
\[
\text{ancestor}(Y, Z).
\]

?- ancestor(tom, pat).

Works fine but may loop if ordering of things changed
Range-restricted programs (RR)

≈ those that can be understood as databases
≈ guaranteed finite relations

Counter examples:

\begin{align*}
equal(X,X). \\
big_number(X) :- X>4.
\end{align*}

Definition:

A clause is RR if any variable in its head occurs in its body and any variable in a predefined test occurs also in an atom with program-defined predicate in that body [to the left of it].

A program is RR if all its clauses are RR.
Negation-as-failure

**Closed-world assumption:** Anything not known by database considered false.

Example:

\[
\text{orphan}(X) :\neg \text{person}(X), \neg \text{father}(\_, X), \neg \text{mother}(\_, X).
\]

\[
\text{person}(\text{adam}).
\]

\[
\text{person}(\text{abel}).
\]

\[
\text{father}(\text{adam}, \text{abel}).
\]

Extend definition of range restriction:

... and any variable in negated atom not covered by \( \exists \), must occurs also in an atom with program-defined predicate [to the left of it].

Counter example:
Problems with Prolog's approximation to NaF

\[ p(a). \]

Test negation
\[
? - \neg p(a).
no
\]
\[
? - \neg p(b).
yes
\]

Looks fine but sem'cs problematic in case of variables:
\[
? - X = b, \neg p(X).
X = b ?
yes
\]
\[
? - \neg p(X), X = b.
no
\]
Consider

• How many lines of Java code is needed for implementing the little family database?
• Another example suited to illustrate
  – Prolog's semantics
  – "Simple, yet powerful"
Logical circuits

(Abstraction over) simple, electrical circuits
often app. 0V \approx 0, app. 5V \approx 1

In Prolog:
\[
\text{not}(0,1).
\]
\[
\text{not}(1,0).
\]
More simple gates

\[
\begin{align*}
\text{and}(0, 0, 0). \\
\text{and}(0, 1, 0). \\
\text{and}(1, 0, 0). \\
\text{and}(1, 1, 1). \\
\text{xor}(0, 0, 0). \\
\text{xor}(0, 1, 1). \\
\text{xor}(1, 0, 1). \\
\text{xor}(1, 1, 0). \\
\text{or}(0, 0, 0). \\
\text{or}(0, 1, 1). \\
\text{or}(1, 0, 1). \\
\text{or}(1, 1, 1). \\
\end{align*}
\]
Building circuits from gates

Example: A half-adder
Adding two bits, A and B:

```
halfadder(A, B, Carry, Sum):-
```
A full-adder, now with old carry

\[
\text{fulladder}(A, B, \text{Carryin}, \text{Sum}, \text{Carryout}):- \\
\]

\[
\text{xor}(A, B, X), \\
\text{and}(A, B, Y), \\
\text{and}(X, \text{Carryin}, Z), \\
\text{xor}(\text{Carryin}, X, \text{Sum}), \\
\text{or}(Y, Z, \text{Carryout}) .
\]
Predicates in Prolog *(often) reversible*

What do we get of output when inputting 0,1,1?
```prolog
?- fulladder(0,1,1,S,C).
C = 1, S = 0 ?
```

What input gives output = 0, 1?
```prolog
?- fulladder(X,Y,Z,0,1).
X = 0, Y = 1, Z = 1 ? ;
X = 1, Y = 0, Z = 1 ? ;
X = 1, Y = 1, Z = 0 ? ;
no
```

**Reversible:** no distinction between input- og output-variable!

Another word for reversible: Relationel