Monday, February 5, 2018

SD 206

Logic and Knowledge representation

Jean-Louis Dessalles

Dep. InfRes

History of Prolog

- **1965: resolution (Robinson)**
- **1972 : Prolog created by A. Colmerauer and P. Roussel in Luminy.**
- **1980 : Prolog acknowledged as a major A.I. language**
- **Now various versions, some of them used in Constraint Programming**

prolog

Bibliography

- Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge, MA.
- Blackburn, P., Bos, J. & Striegnitz, K. (2006). *Learn prolog now!* Londres: College Publications.
  www.academia.edu/download/31015322/pro.pdf
- Cervoni L., Ed-Dbali A., Deransart P. (1996). *Prolog*. Springer.
- Bellot Patrick (1994). *Objectif Prolog*. Masson, Paris
- Bratko Ivan (1990). PROLOG - *Programming for Artificial Intelligence*. Addison-Wesley
- Clocksin W.F., Mellish C.S. (2003/1981). *Programming in Prolog*. Springer Verlag, Berlin
- Harrison, J. (2009). *Handbook of practical logic and automated reasoning*. Cambridge, MA.
- Niederlinski A. (2014). A Gentle Guide to Constraint Logic Programming via ECLiPSe.
  www.anclp.pl/download/AN_CLP.pdf
- Scott, Peter & Nicolson, Rod (1991). *Cognitive Science Projects in Prolog*. Lawrence Erlbaum Assoc., Hove.

Some links

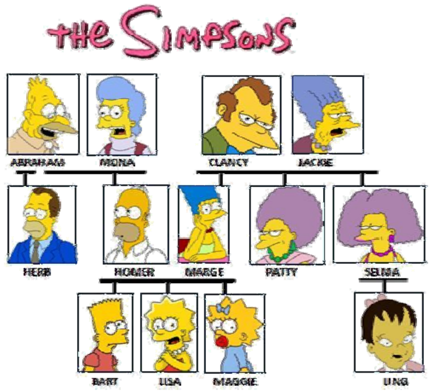
Lecture notes

‘First steps in Prolog’ on the web
Brief Introduction to Prolog (by Ken Been)
Adventure in Prolog (by Amzi)

SWI-Prolog
SWI-Prolog web site
reference manual

First programme

```
parent(marge, lisa).
parent(marge, bart).
parent(marge, maggie).
parent(homer, lisa).
parent(homer, bart).
parent(homer, maggie).
parent(abraham, homer).
parent(abraham, herb).
parent(mona, homer).
parent(jackie, marge).
parent(clancy, marge).
parent(jackie, patty).
parent(clancy, patty).
parent(jackie, selma).
parent(clancy, selma).
parent(selma, ling).
```



famille.pl

prolog

Prolog clauses

* **fact :**

  `female(marge).`

* **clauses:**

```
child(X,Y) :-
              parent(Y,X).
```

* **Exercises:**
  `mother(X,Y), grandparent(X,Y), ancestor(X,Y), cousin(X,Y)`

prolog

Data representation

- Predicates :
  - Name
  - Argument number (arity)

```
coulour(car, red)
```

- Constants :
  - Atoms : character strings (start with lower case)
  - Numbers : integers or floats
- Variables :
  - Character string beginning with a capital

prolog

Unification

- **Example :**

brother(patrick,Who)

Unification with brother(X,Y)

X=patrick and Y=Who

```
brother(X,Y) :-
      male(X),
      parent(X,Z),
      parent(Y,Z),
      X \== Y.
```

male(patrick)
parent(patrick,Z)
parent(Y,Z)
patrick \== Y

prolog

Unification

- **Unification predicate : =**
  - `a(B,C) = a(2,3).`
    `YES {B=2, C=3}`
  - `a(X,Y,L) = a(Y,2,carole).`
    `YES {X=2, Y=2, L=carole}`
  - `a(X,X,Y) = a(Y,u,v).`
    `NO`
- **Exercise**
  - **Unify** `p(X,b(Z,a),X)` **with** `p(Y,Y,b(V,a))`

prolog

Unification

- **Unification results in a** unifier **(or** substitution**), which instantiates some variables.**
- **The unifier is often not unique, but it is maximal for generality.**
- **Unification may fail.**
  `e(X,X)` and `e(2,3)` can't be unified.

prolog

Unification algorithm

A free variable can be seen as a pointer to NIL. If a variable isn't free, it is said bound. Dereferencing a variable means going over the unification chain until reaching its value.
The algorithm (in pseudo-code) is:

```
procedure unify(t1,t2);
(* t1 and t2 are two terms *)
start
        t3:=dereference(t1);
        t4:=dereference(t2);
        (* dereference is a fonction that acts on bound variables and does nothing otherwise *)
        if t3 is a variable then t3 points to t4 (*success*)
        else    if t4 is a variable then t4 points to t3 (*success*)
                else    if t3 is an atom and t4 is an atom then
                                if t3=t4 then success
                                else fail
                        else    let t3=f(t31,..,t3n) and t4=g(t41,..,t4m)
                                if f=g then (* n=m *)
                                        for i:=1 to n do unify(t3i,t4i)
                                        (* if unify(t3i,t4i) fails for some i
                                                then unify(t1,t2) fails *)
                                else (* f<>g *) fail
end (*unify*)
```

prolog

Operators

arithmetic operators : `+, -, *, /, div, mod`

affectation : `X is 6 - 2`. The variable `X` takes value 4. The member of right-hand side must be completely instantiated at the time of the call.

These operators are in fact Prolog functors (`3+2` can be rewritten `+(3,2)`), but these functors enjoy the particular property of being able to be evaluated.

Arithmetic comparison: `<, >, > =, = <, =: =, = \ =`. The terms entering these comparisons must be instantiated at the time of the call. They are evaluated, then the values are compared.

Comparison of terms:

`X = Y`	succeeds if X unifies with Y
`X == Y`	succeeds if X and Y are syntactically identical
`X \== Y`	succeeds if X and Y have different syntactic structures

```
Comparison operators are predicates.
3 > 4+1 can be rewritten
  > (3,4+1), or  > (3, +(4, 1)).
```

prolog

Representing lists

[a, b, c, d] = [a | [b, c, d]]   a : **head**; [b, c, d] : **queue**

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

[_ | _]   has at least one element.

extract(Elt, List, ListWithoutElt)

prolog

Exercises

- **Write** `extract(X,L,Rest)`
- **Write** `attach(L1,L2,L3)` **that concatenates two lists**

prolog

Horn clauses

- **Facts and rules.**
- **General form :** `F :- F1, F2,…, Fn.`
  - To prove `F`, one must successively prove `F1, F2,…, and Fn`.

- `F` **is the clause's head**
- **The goals** `F1, F2,…, Fn` **constitute the clause's tail**
- **A fact is a clause with an empty tail**

prolog

Functionning of Prolog

- **Declarativity - Reversibility**
- **Depth-first strategy**
- **Backtracking**
- **Recursivity**
- **Unification**

prolog

Prolog’s strategy

- To answer a question, Prolog builds a search tree

- When several clauses match the question:
  * Successive trials, in the order of declaration
  * Set of possibilities represented as a tree
  * Each choice is a node of the search tree

- Depth-first strategy
  * Success node : that’s a solution. Prolog displays it and stops
  * Failure node : Prolog backtracks up in the tree, until it finds a choice point with unexplored branches

prolog

Prolog’s strategy

- If backtracking encounters no choice point left, Prolog stops. No further solution.

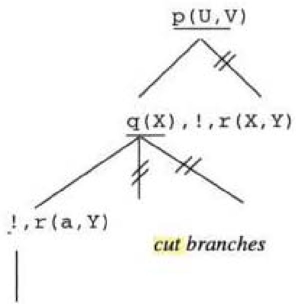
- Some branches are infinite! So search may not stop… Take care of:
  * The order of goals in clause tails
  * The order of clauses

prolog

The *cut*

```
p(X, Y) :-
        q(X),
        !,
        r(X,Y).
p(X, Y) :-
        s(X).

q(X) :-
        !,
        r(a,Y).
```



- **Preventing backtracking**
  - **Example: « add in a list »**

prolog

The *cut*

- **Negation by failure (closed world)**

```
fancy(belle_d_argent).
fancy(maximus).

expensive(belle_d_argent).

affordable(Restaurant) :-
        not(expensive(Restaurant)).

?- fancy(X), affordable(X).
   X=maximus
?- affordable(X), fancy(X).
```

prolog

Declarativity

- **Prolog and logic**
- **Limits of déclarativity**
  - **Clauses order**
  - **Order within clauses**

prolog

Prolog and logic

- **Prolog clause = generalised disjunction**

  p :- q , r . $q \wedge r \Rightarrow p$ $\neg q \vee \neg r \vee p$

- **In first order logic**
  - **Correct and complete proof methods**
  - **Semi-decidable algorithms**
- **In first order logic restricted to Horn clauses**
  - **Decidable algorithmes**

prolog

Exercises

- **Exercises:**
  - **Deny a sentence**
  - **Duplicate each element of a list**
  - **Intertwine two lists**
  - **Palindrome test**
  - **Palindrome building**
  - **Remove redundant elements**
  - **Test prime numbers**
  - **To repeated patterns in a list**
  - **To interlace an unspecified number of lists**
  - **Generate lists containing the terms A, B, C, D without identical consecutive terms**

prolog

Exercises

- **Write a programme** `extract(X,L,Remainder)` **that takes an element from a list**
- **Write a programme that permutes a list**
- **Write** `attach(L1,L2,L3)`**, a programme that appends two lists**

prolog