

# **Meta-Programming in Prolog**

**Jørgen Villadsen**

**Simple Deductive Database in 100 Lines**

water contains salt.  
salt compound.

X substance if X element.  
X substance if X compound.  
X substance if X mixture.

salt contains sodium. salt contains chlorine.  
sodium element. chlorine element.

X component Y if Y contains X.  
X component Y if Y contains Z and X component Z.

what substance?

what component water?



- 1: sodium chlorine salt
- 2: salt sodium chlorine

```
io :- io(user).

io(I) :- io(I,user).

io(I,O) :- scan(I,L), phrase(program(Ss,Qs),L), result(O,Ss,Qs).

scan(F,L) :- see(F), get0(C), list(C,L), seen.

list(C,L) :-
    C < 0 -> L = [] ; get0(D),
    ( C =< " " -> list(D,L) ; L = [A|R],
      ( char(C) -> word(D,V,E), name(A,[C|V]), list(E,R) ; name(A,[C]), list(D,R)
        ) ).

word(C,W,E) :- char(C) -> W = [C|V], get0(D), word(D,V,E) ; E = C, W = [].

char(C) :- C >= "a", C =< "z".

chars([]).
chars([C|L]) :- char(C), chars(L).


scan(user, ['X', cheats, '.', who, cheats, ?])

"abc" = [97,98,99]

chars([97,98,99])

name(abc, [a,b,c])

name('X?', ['X','?'])
```



```

program(Ss,Qs) --> rules(Ss), queries(Qs).

rules([]) --> [].
rules([[A|As]|Ss]) --> atom(A), condition(As), [], rules(Ss).

condition([]) --> [].
condition([A|As]) --> [if], atom(A), coordination(As).

coordination([]) --> [].
coordination([A|As]) --> [and], atom(A), coordination(As).

atom(unary(T,I)) --> ident(I), token(T).
atom(binary(T,I1,I2)) --> ident(I1), token(T), ident(I2).

queries([]) --> [].
queries([Q|Qs]) --> wh, rest(Q), [?], queries(Qs).

wh --> [who] ; [what].

rest(unary(P,var(0))) --> token(P).
rest(binary(P,var(0),const(T))) --> token(P), token(T).

token(T) --> [T], {name(T,I), chars(I)}.

vars(3).

ident(const(T)) --> token(T).
ident(var(1)) --> ['X'].
ident(var(2)) --> ['Y'].
ident(var(3)) --> ['Z'].

phrase(program([[unary(cheats, var(1))]],
                [unary(cheats, var(0))]), ['X', cheats, '.', who, cheats, ?])

```



```

result(F,Ss,Qs) :- tell(F), nl, lines(1,Ss,Qs), told.

lines(_,_, []).
lines(L,Ss,[Q|Qs]) :- write(L), write(':'),
    answer(Ss,Q), nl, K is L+1, lines(K,Ss,Qs).

answer(Ss,Q) :- \+ ( refute(Ss,[Q],[],B,0), solution(B), fail ).

solution(B) :- eval(var(0),B,E),
    write(' '), ( E = const(T) -> write(T) ; write('_') ).

eval(const(T),_,const(T)).
eval(var(N),B,I2) :- member([N,I1],B) -> eval(I1,B,I2) ; I2 = var(N).

refute(.,.,B,B,_).
refute(Ss,[Goal|Goals],B1,B3,V) :-
    member(S,Ss), copy(S,[Head|Body],V), vars(T), W is V+T,
    unify(Goal,Head,B1,B2), append(Body,Goals,NewGoals),
    refute(Ss,NewGoals,B2,B3,W).

```

---

```

result(user, [[unary(cheats, var(1))]], [unary(cheats, var(0))])
refute([[unary(cheats, var(1))]], [unary(cheats, var(0))], [], [[0, var(1)]]), 0)
solution([[0, var(1)]])
eval(var(1), [[0, var(1)]], var(1))
answer([[unary(cheats, var(1))]], unary(cheats, var(0)))

```

```

unify(unary(P, I1), unary(P, I2), B1, B2) :-
  unify_term(I1, I2, B1, B2).
unify(binary(P, I11, I12), binary(P, I21, I22), B1, B3) :-
  unify_term(I11, I21, B1, B2), unify_term(I12, I22, B2, B3).

unify_term(I11, I21, B1, B2) :-
  eval(I11, B1, I12), eval(I21, B1, I22), unify_ident(I12, I22, B1, B2).

unify_ident(var(N), var(N), B, B).
unify_ident(const(T), const(T), B, B).
unify_ident(var(N), const(T), B, [[N, const(T)] | B]).
unify_ident(const(T), var(N), B, [[N, const(T)] | B]).
unify_ident(var(N), var(M), B, [[N, var(M)] | B]) :- N =\= M.

copy([], [], _).
copy([A1|As1], [A2|As2], V) :- copy_atom(A1, A2, V), copy(As1, As2, V).

copy_atom(unary(P, I1), unary(P, I2), V) :-
  copy_ident(I1, I2, V).
copy_atom(binary(P, I11, I12), binary(P, I21, I22), V) :-
  copy_ident(I11, I21, V), copy_ident(I12, I22, V).

copy_ident(const(T), const(T), _).
copy_ident(var(N), var(M), V) :- M is N+V.

```

---

```

copy([unary(cheats, var(1))], [unary(cheats, var(1))], 0)
unify(unary(cheats, var(0)), unary(cheats, var(1)), [], [[0, var(1)]])

```