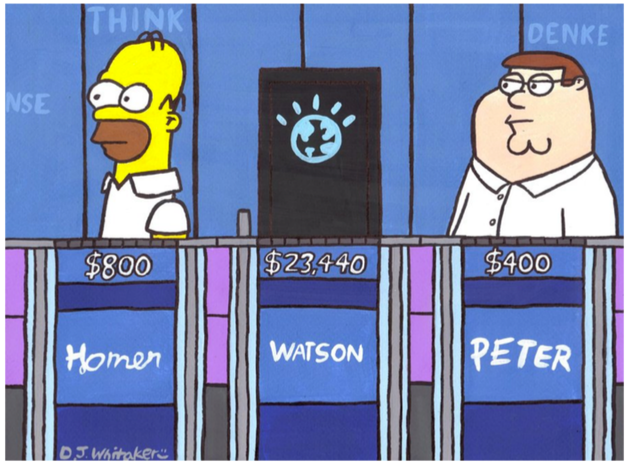
Prolog and declarative programming, pt 3



```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([])     --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

Animal identification

```
/* start with ?- go.  Answer with yes or no.      */

go :- hypothesize(Animal),
      write('I guess that the animal is: '),
      write(Animal),
      nl,
      undo.

/* hypotheses to be tested */
hypothesize(cheetah)   :- cheetah.
hypothesize(tiger)     :- tiger.
hypothesize(giraffe)   :- giraffe.
hypothesize(zebra)     :- zebra.
hypothesize(ostrich)   :- ostrich.
hypothesize(penguin)   :- penguin.
hypothesize(albatross) :- albatross.
hypothesize(unknown).              /* no diagnosis */
```

Animal identification

```
/* animal identification rules */
cheetah :- mammal,
           carnivore,
           verify(has_tawny_color),
           verify(has_dark_spots).
tiger :- mammal,
         carnivore,
         verify(has_tawny_color),
         verify(has_black_stripes).
giraffe :- ungulate,
           verify(has_long_neck),
           verify(has_long_legs).
zebra :- ungulate,
         verify(has_black_stripes).
```

```
ostrich :- bird,
           verify(does_not_fly),
           verify(has_long_neck).
penguin :- bird,
           verify(does_not_fly),
           verify(swims),
           verify(is_black_and_white).
albatross :- bird,
             verify(appears_in_story_
                    Ancient_Mariner),
             verify(flys_well).
```

Animal identification

```
/* classification rules */
mammal    :- verify(has_hair).
mammal    :- verify(gives_milk).
bird      :- verify(has_feathers).
bird      :- verify(flys),
             verify(lays_eggs).
carnivore :- verify(eats_meat).
carnivore :- verify(has_pointed_teeth),
             verify(has_claws),
             verify(has_forward_eyes).
ungulate :- mammal,
            verify(has_hooves).
ungulate :- mammal,
            verify(chews_cud).
```

Animal identification

```
/* how to ask questions */
ask(Question) :-
    write('Does the animal have the following attribute: '),
    write(Question),
    write('? '),
    read(Response),
    nl,
    ( (Response == yes ; Response == y)
      ->
       assert(yes(Question)) ;
       assert(no(Question)), fail).

:- dynamic yes/1,no/1.
```

Animal identification

```
/* How to verify something */
verify(S) :-
   (yes(S)
    ->
    true ;
    (no(S)
     ->
     fail ;
     ask(S))).

/* undo all yes/no assertions */
undo :- retract(yes(_)),fail.
undo :- retract(no(_)),fail.
undo.
```

Discussion

- What do:
  - DFA checkers
  - The chess queen problem
  - Natural language processing
  - Robotic planning
  - etc.

  - Have in common?

How does it work?

- Short answer: depth-first backtracking search
- Given a query and knowledge base, for each top-level term in the query, Prolog:
  - Tries to match the term against the head of a clause in the KB.
  - If it fails to find one it returns failure.
  - If it finds one then the body of the clause becomes the current query and this process recurses.
  - If that process succeeds then Prolog returns success along with any bindings used to succeed.
  - If it fails then Prolog tries this loop again (i.e., tries to match the term against the head of a different clause in KB).
- This process bottoms out either when a term matches a fact
  - or when a term matches certain system relations that are guaranteed to succeed

Prolog also allows complex terms

- What we’ve seen so far is called Datalog: “databases in logic.”

- Prolog is “programming in logic.” It goes a little bit further by allowing complex terms, including records, lists and trees.

- These complex terms are the source of the only hard thing about Prolog, “unification.”

Properties of Prolog

Homoiconic

```
solve(true).
solve((Subgoal1,Subgoal2)) :-
    solve(Subgoal1),
    solve(Subgoal2).
solve(Head) :-
    clause(Head, Body),
    solve(Body).
```

Turing complete

```
turing(Tape0, Tape) :-
    perform(q0, [], Ls, Tape0, Rs),
    reverse(Ls, Ls1),
    append(Ls1, Rs, Tape).

perform(qf, Ls, Ls, Rs, Rs) :- !.
perform(Q0, Ls0, Ls, Rs0, Rs) :-
    symbol(Rs0, Sym, RsRest),
    once(rule(Q0, Sym, Q1, NewSym, Action)),
    action(Action, Ls0, Ls1, [NewSym|RsRest], Rs1),
    perform(Q1, Ls1, Ls, Rs1, Rs).

symbol([], b, []).
symbol([Sym|Rs], Sym, Rs).

action(left, Ls0, Ls, Rs0, Rs) :- left(Ls0, Ls, Rs0, Rs).
action(stay, Ls, Ls, Rs, Rs).
action(right, Ls0, [Sym|Ls0], [Sym|Rs], Rs).

left([], [], Rs0, [b|Rs0]).
left([L|Ls], Ls, Rs, [L|Rs]).
```