# Today

**Meta-Interpreters and Explanation**





# Recall

Describe

- Prolog rule base
- Prolog inference

in Prolog.

Use this for different control regimes.

We can write debuggers in the same way.

We can also get hold of derivations.





# Prolog program as inference system

We can regard a Prolog program (just Horn clauses) as an inference system.

Unit clauses are axioms, others are inference rules.

Axiom $\texttt{father(fred,jim)}$

Rule $\dfrac{\texttt{father(X,Y)}}{\texttt{ancestor(X,Y)}}$

Rule $\dfrac{\texttt{parent(X,Y)}}{\texttt{ancestor(X,Y)}}$

Rule $\dfrac{\texttt{parent(X,Y)} \quad \texttt{ancestor(Y,Z)}}{\texttt{ancestor(X,Z)}}$





Regard query

$$? \texttt{ancestor(jim,X)}$$

as asking for the derivation of some instance

$$\frac{\vdots \qquad \vdots}{\texttt{ancestor(jim,simon)}}$$

Prolog has an (incomplete) inference procedure, searching for these derivations.

We can use a meta-interpreter to construct the derivations
(see Sterling and Shapiro, "Art of Prolog").

# Meta-interpreter with Derivations

```
% solve( Goal, Proof ):
%    Proof is proof tree for Goal

?- op(500, xfy, <-).

solve_proof( true, true ) :- !.
solve_proof( (A,B), (PA,PB) ) :- !,
    solve_proof( A, PA ),
    solve_proof( B, PB ).
solve_proof( A, (A <- P) ):-
    clause( A, B ),
    solve_proof( B, P ).
```

Uses standard search, and returns a form of the derivation.





# Example

For the usual member/2:

```
?- solve( member(2,[0,1,2,3]),Proof ).

Proof = member( 2, [0,1,2,3,] )
       <- member( 2, [1,2,3] )
         <- member( 2, [2,3] )
           <- true
```





We can pretty print such derivations:

```
| ?- solve_exp(member(2,[0,1,2,3,4])).

member(2,[0,1,2,3,4]) follows from rule
IF member(2,[1,2,3,4])
THEN member(2,[0,1,2,3,4])

member(2,[1,2,3,4]) follows from rule
IF member(2,[2,3,4])
THEN member(2,[1,2,3,4])

member(2,[2,3,4]) is a fact.
```





# Pretty Printing Derivations

Note that if we just pretty print the data structure used for derivations above, we get instances (particular cases) of the rules (clauses) used, rather than the general rule. An alternative is to record the general form of the rule as well as the instance when building the derivation.

Note also that here we record only the rules that contribute to the final solution. Branches of the search space that are tried and fail are not indicated.

# Derivation = Explanation?

It is important to have an explanation for the response to a query.

1. to convince the user
   the answer may be unexpected

2. to help debugging of KB
   rule base may be incomplete/faulty.

These are different situations, and call for different supporting mechanisms.





Derivations are a good source for explanation.

Note that a proof should convince that the conclusion is correct — otherwise it is a bad proof.

However, the full derivation has too much information;
we really want to know what are the intermediate crucial steps.

For the derivations in the style above, we usually do not want to know about computation of membership of lists. This leads to "wallpaper" output where the key information is drowned by the excessive detail.





# Explanation

We can unfold the derivation to required depth:

look deeper and deeper in the derivation tree.

This is better – but what we need may be buried.

If we know which are the "interesting" rules or predicates, we can pick them out as part of the meta-interpretation.





# Different derivations

Note that usually there are several derivations possible, and so there are several corresponding explanations also.

If we only have one of them, we may not have the intuitive explanation.

We can look for multiple derivations (this is slower, of course). This is an area of active research at the moment, called answer set programming.

# Meta-interpretation for program analysis

We can use meta-interpretation for debugging and analysis.

**Example**

Loop detection — the problem is uncomputable in general.

> Does program P with input I terminate?

We can get a practical tool that will let us know if recursion exceeds some given depth, and return information about the calls leading to this (potential) loop.





# Loop detecting Meta-Interpreter

```
% solve( ?, -, - ).
% solve( Goal, Depth, Overflow )
solve( true, _, no_over ).
solve( _, 0, [] ).
solve( (A,B), D, Over ) :-
    D>0, solve( A,D,Over1 ),
    solve_conj( Over1,B,D,Over ).
solve( A,D,Over ) :-
    D>0,  clause( A,B ),
    D1 is D-1,
    solve( B, D1, Over1 ),
    return_over( Over1,A,Over ).

solve_conj( no_over,B,D,Over ) :- solve( B,D,Over ).
solve_conj( Over,_,_,Over ).
```





This will

- succeed (with `no_over`) if there is a solution within the given depth;
- return `overflow` as stack, if recursion exceeds the given depth.

**Example**

```
% buggy insert

:- dynamic insert/3.
insert( X, [H|T], [H,X|T] ) :-
    X<H.
insert( X, [H|T], [H|TT]  ) :-
    H =< X,
    insert( H, [X|T], TT ).
insert(X,[],X).
```





```
% analyse looping behaviour

?- solve_overflow(insert(2,[2],X),4,O).

O = [insert(2,[2],[2,2,2,2,2]),
     insert(2,[2],[2,2,2,2]),
     insert(2,[2],[2,2,2]),
     insert(2,[2],[2,2])],
X = [2,2,2,2,2] ?
```

In this way, we can get declarative debugging.

– if the output is wrong, then some rule or fact is wrong.

If the user understands the domain, they can look at the instances used, and track down the wrong one (algorithmic debugging).

**Other uses**

- Carry around confidence levels.
- Interactive execution
- etc





# Meta-Level Inference

See eg Bundy, "Computer Modelling of Mathematical Reasoning" The problem domain is solving algebraic equations, eg to find an $x$ such that

$$x^2 + 2 = 4 * x - x^2$$

The Object knowledge – contains rules for manipulating algebraic terms.

These rules can be applied in many ways. Use meta-level inference to decide which rule to apply.





# Object rules

Axiom $\mathtt{X} * (\mathtt{Y} + \mathtt{Z}) = \mathtt{X} * \mathtt{Y} + \mathtt{X} * \mathtt{Z}$

Rule $$\frac{\mathtt{X} = \mathtt{Y} \qquad \mathtt{P[X]}}{\mathtt{P[Y]}}$$

Rule $$\frac{\mathtt{A} * \mathtt{X}^2 + \mathtt{B} * \mathtt{X} + \mathtt{C} = 0}{\mathtt{X} = \frac{\mathtt{B} \pm \sqrt{(\mathtt{B}^2 - 4 * \mathtt{A} * \mathtt{C})}}{2 * \mathtt{A}}}$$





# Examples

To solve

$$X = Y \qquad (Y \neq 0)$$

use the rule

$$X = Y \leftrightarrow X - Y = 0.$$

To solve

$$F[X] = 0$$

where there is an axiom of the form $X = Y$ where $Y$ has fewer variables than $X$, use substitution to get

$$F[Y] = 0.$$

In this way, describe the search strategy. Note that properties like "$Y$ has fewer (object) variables than $X$" correspond to meta-level predicates.

# Summary

- Meta-level interpreters, for
  – explanation
  – debugging

- We can express control knowledge explicitly at the meta-level.

  See Sterling & Shapiro for more information
  (e.g. on two level rule interpreters).