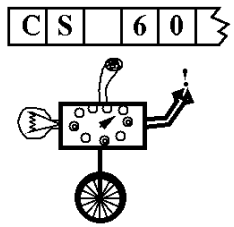


More Prolog

November 21, 2001

Rule and Sub-Goal Ordering

Suppose the goal is `knows(john, Y, R).`

This rule is tried first.

```
knows(X, Y, living) :-
    lives(X, Z),
    lives(Y, Z).
```

This sub-goal is satisfied first, which binds Z.

This sub-goal is satisfied next.

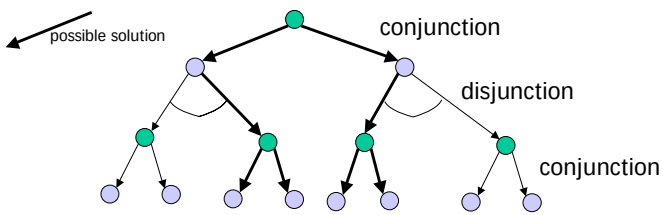
This rule is tried after the first rule is **exhausted**.

```
knows(X, Y, tutoring) :-
    canTutor(Y, X).
```

In effect, we have **disjunction** (*or*) **among** rules, and **conjunction** (*and*) **within** rules.
Remember that Prolog execution is **depth-first search**.

And-Or Trees

- In AI, problem-solving trees are typically "And-Or" trees.
- This applies to Prolog's goals.



"Logical Variables" in Prolog

- A variable in Prolog is like an **object** that can have one of two states:
  - unbound
  - bound, to some Prolog term, e.g. an individual
- Once the variable is bound, it only gets re–bound in **backtracking**, which results in removing the former binding first.

Lists in Prolog

- The rex list notation was derived from Prolog's list notation.
- A list can contain logical variables, which are already bound, or may get bound **later**.
- The process of binding is known as **unification** (which means "make the same").

Unification (1)

- Unification causes two logical variables to denote the same thing.
  - The symbol for unification in Prolog is =
  - Note: this is a *symmetric* operation!
- Examples:

`X = a`	Variable unified with constant
`X = [a,b,c]`	Variable unified with a list
`[X,b,c] = [a|Y]`	`X = a`, `Y = [b,c]`
`[X,b] = [a,c]`	NOT UNIFIABLE

Unification (2)

- Unification takes place **implicitly** when a rule is used for a goal:

```
knows(john, U, R).
      ⇕     ⇕  ⇕
knows(X,    Y, living) :-
    lives(X, Z),
    lives(Y, Z).
```

```
X = john
Y = U
R = living
```

- The rule is usable iff the goal and the rule consequent variables unify.

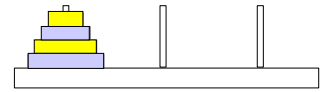
Special handling of _ in Prolog

- The variable _ is special.
- It is called a "throw-away" variable.
- _ unifies with anything, but different occurrences of _ are not identified, unlike other variables.

Other variables beginning with _

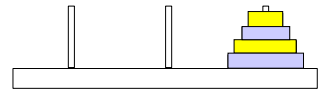
- A variable that occurs only once in a clause is called a “singleton variable”.
- Often singleton variables are the result of a typing error, and certain compilers will warn about them.
- To prevent the warning, when this is the intention, use a variable that begins with _, such as _Name rather than Name.

Example: Towers of Hanoi



Move only one disk at a time.

Never place a larger disk on a smaller one.

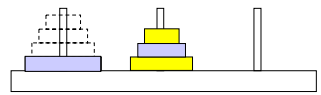


Solving Towers of Hanoi

- Some approaches:
  - Pre-programmed solution
    - Recursive solution is easy in most languages
  - Let prolog find solution using depth-first search
    - Trickier, but shows off Prolog’s capabilities
    - May not find shortest solution
  - Program breadth-first search in Prolog
    - Still trickier

Pre-Programmed Solution

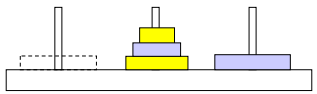
- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack Other (the stack other than ***From*** and ***To***)



A key point throughout is that the N-1 disk moves can be done without violating the constraint that a larger disk not be put atop a smaller one.

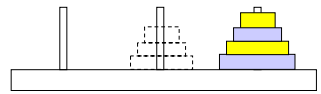
Pre-Programmed Solution

- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack ***Other*** (the stack other than ***From*** and ***To***)
  - Move 1 disk from stack ***From*** to stack ***To***



Pre-Programmed Solution

- To move N disks from stack ***From*** to stack ***To***:
  - Move N-1 disks from stack ***From*** to stack ***Other*** (the stack other than ***From*** and ***To***)
  - Move 1 disk from stack ***From*** to stack ***To***
  - Move N-1 disks from stack ***Other*** to stack ***To***



Pre-Programmed Solution

```
% towers(N, From, To, Moves) means that Moves is the list of
% moves to move N disks from stack From to stack To
towers(N, From, To, Moves) :-
    towers(N, From, To, [ ], ReversedMoves),
    reverse(ReversedMoves, Moves).

% towers(N, From, To, Acc, Moves) means that Moves is the reverse of the list
% of moves to move N disks from stack From to stack To, with Acc being
% the reverse of the accumulated moves going in (to avoid appending).

towers(0, _, _, Acc, Acc).

towers(N, From, To, Acc, Moves) :-
    other(From, To, Other),
    N1 is N - 1,
    towers(N1, From, Other, Acc, Moves1),
    towers(N1, Other, To, [ [From, To] | Moves1], Moves).

other(1,2,3).  other(2,1,3).
other(1,3,2).  other(3,1,2).
other(3,2,1).  other(2,3,1).
```

is is explained on next slide

The `is` operator in Prolog

- Unlike a functional language, expressions that look like arithmetic are not automatically evaluated.
  - Example: `X - 1` is kept as its syntax tree `-(X,1)`
- The `is` operator is used to force evaluation.

  `Y is X-1` means unify `Y` with the arithmetic **value** of `X-1`

Depth-First Towers of Hanoi (1)

Does not require a human to solve the puzzle first

First characterize the possible moves.

This is a move from stack 1 to stack 2:

from / to stack 1 before stack 2 after

```
move([1, 2], [[F1 | R1], S2, S3], [R1, [F1 | S2], S3]) :-
    ok(F1, S2).
```

provided that it is ok to move disk F1 onto stack S2

Depth-First Towers of Hanoi (2)

All the possible moves in six rules:

```
move([1, 2], [[F1 | R1], S2, S3], [R1, [F1 | S2], S3]) :- ok(F1, S2).
move([1, 3], [[F1 | R1], S2, S3], [R1, S2, [F1 | S3]]) :- ok(F1, S3).
move([2, 1], [S1, [F2 | R2], S3], [[F2 | S1], R2, S3]) :- ok(F2, S1).
move([2, 3], [S1, [F2 | R2], S3], [S1, R2, [F2 | S3]]) :- ok(F2, S3).
move([3, 1], [S1, S2, [F3 | R3]], [[F3 | S1], S2, R3]) :- ok(F3, S1).
move([3, 2], [S1, S2, [F3 | R3]], [S1, [F3 | S2], R3]) :- ok(F3, S2).
```

from / to state before state after condition

Depth-First Towers of Hanoi (3)

When is it ok to move a disk onto a stack?
Assume the disks are represented by numbers 1, 2, 3, ...
with smaller numbers representing smaller disks.

```
ok(_, [ ]).
```

empty target stack

```
ok(A, [B | _]) :- smaller(A, B).

smaller(A, B) :- A < B.
```

Depth-First Towers of Hanoi (4)

`towers([S1, S2, S3], Moves)` will mean that Moves is a valid move sequence that results in S1 and S2 being empty (so all disks are on S3).

`towers([S1, S2, S3], Seen, Moves)` means the same, except that Seen will be a list of all previous states (to prevent infinite looping).

```
towers(InitialState, Moves) :- towers(InitialState, [ ], Moves).

towers([[ ], [ ], _], _, [ ]). % final state, no more moves

towers(Before, Seen, [Move | Moves]) :-
    nonMember(Before, Seen),
    move(Move, Before, After),
    towers(After, [Before | Seen], Moves).
```

only consider if Before not already seen

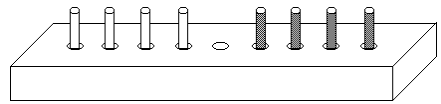
recurse

Depth-First Towers of Hanoi (5)

Auxiliary Predicates:

```
nonMember(X, L) :- \+ member(X, L).

member(X, [X | _]).

member(X, [_ | L]) :- member(X, L).
```

Exercise



Reverse the pegs by moving peg "forward" or jumping forward over a peg of either color.

Work out a depth-first solution in Prolog.

(You don't have to check for cycles, because there can't be any.)

Bi-Directional Execution (1)

Consider:

```
member(X, [X | _]).

member(X, [_ | L]) :- member(X, L).
```

This predicate can be viewed as a member **test**.

It can also be viewed as a member **generator**.

Bi-Directional Execution (2)

test	**generate**
`| ?- member(3,[1,2,3,4,5]).`	`| ?- member(X,[1,2,3,4,5]).`
`yes`	`X = 1 ;`
`| ?- member(6,[1,2,3,4,5]).`	`X = 2 ;`
`no`	`X = 3 ;`
	`X = 4 ;`
	`X = 5 ;`
	`no`

Generating with append

```
append ([],M,M).

append ([A|L],M,[A|N]) :-
      append (L, M, N).
```

functional

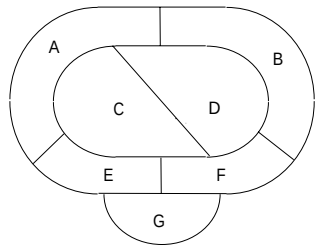
```
| ?- append([1,2,3],[4,5],Z).

Z = [1,2,3,4,5] ;

no
```

relational

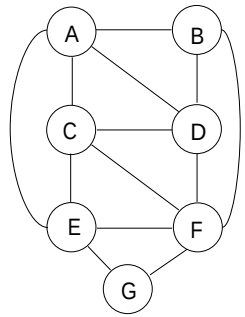
```
| ?- append(X,Y,[1,2,3,4,5]).

X = [ ],
Y = [1,2,3,4,5] ;

X = [1],
Y = [2,3,4,5] ;

...

X = [1,2,3,4,5],
Y = [ ] ;

no
```

Map Coloring

A map



Corresponding graph

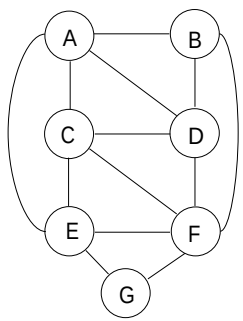


Map Coloring (2)

Prolog Clause

```
map([A,B,C,D,E,F,G]) :-
      next(A, B),
      next(A, C),
      next(A, D),
      next(A, E),
      next(B, D),
      next(B, F),
      next(C, D),
      next(C, E),
      next(C, F),
      next(D, F),
      next(E, F),
      next(E, G),
      next(F, G).
```

Graph



Map Coloring (3): Color Constraints

```
next(X,Y) :- color(X),color(Y), X \== Y.
```

```
color(red).
color(blue).
...
```

colors to be used

means individuals are **not equal**

These and the preceding clause are the entire program.

Constraint Solving with Generate & Test:

"Zebra" Problem

There are five consecutive houses, each of a different color and inhabited by men of different nationalities. They each own a different pet, have a different favorite drink and drive a different car.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Porsche driver owns snails.
7. The Masserati is driven by the man who lives in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The man who drives a Saab lives in the house next to the man with the fox.
11. The Masserati is driven by the man in the house next to the house where the horse is kept.
12. The Honda driver drinks orange juice.
13. The Japanese drives a Jaguar.
14. The Norwegian lives next to the blue house.

The problem is: Who owns the Zebra? Who drinks water?

Prolog Solution to "Zebra" (1)

```
left_right(L,R,[L,R,_,_,_]).
left_right(L,R,[_,L,R,_,_]).
left_right(L,R,[_,_,L,R,_]).
left_right(L,R,[_,_,_,L,R]).

next_to(X,Y,L) :-
      left_right(X,Y,L).
next_to(X,Y,L) :-
      left_right(Y,X,L).
```

Prolog Solution to "Zebra" (2)

```
zebra(S) :-
  S = [[_,norwegian,_,_,_],_,[_,_,_,milk,_],_,_],
   next_to([_,norwegian,_,_,_],[blue,_,_,_,_],S),
   member([green,_,_,coffee,_],S),
   left_right([ivory,_,_,_,_],[green,_,_,_,_],S),
   member([red,englishman,_,_,_],S),
   member([_,ukranian,_,tea,_],S),
   member([yellow,_,masserati,_,_],S),
   member([_,_,honda,orange_juice,_],S),
   member([_,japanese,jaguar,_,_],S),
   member([_,spaniard,_,_,dog],S),
   next_to([_,_,masserati,_,_],[_,_,_,_,horse],S),
   member([_,_,porsche,_,snails],S),
   next_to([_,_,saab,_,_],[_,_,_,_,fox],S).
```

Warning: Builtin Arithmetic is Not Reversible

- Consider a clause
  `p(X, Y) :- Y is X+1.`
- A possible use of this clause is:
  ```
  | ?-  p(5, Z).
  Z = 6
  ```
- This clause cannot be used in reverse because the is predicate is not reversible:
  ```
  | ?- p(X, 6).
  ```
  will not give `X = 5`; it will fail.