Prolog
Programming in Logic

Lecture #8

Ian Lewis, Andrew Rice

Today's discussion

Videos

- Sudoku
- Constraints

Q: What are the extra-logical equalities?

A: So you know to avoid them:

```
x(A) == x(A)    -- Equivalence. x(A) == x(B) fails.
  D \== 1       -- Not equivalent
Exp1 =:= Exp2   -- E1 is Exp1, E1 is Exp2
Exp1 =\= Exp2   -- \+ E1 is Exp1, E1 is Exp2
   A =@= B      -- variants (<Equivalence >Unification).
     :=         -- ? defined operator, no function
     <=         -- ? undefined (use =<)
unify_with_occurs_check(A,B) -- should be an infix op.
```

```
foo :- ..., \+ Term, ...
```

Q: Is [a,b,c|A]-B a difference list?

A: If it's a quiz: answer is NO.

A: If it's a CS conversation, you could reason if:
A = [d,e,f|B], X = [a,b,c|A]-B
means that [a,b,c|A]-B *is* a difference list.

Although, similar is X an integer? yes if X=7 ...

Q: When is Prolog not ML-like?

ML factorial:

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)

f = fact 5
```

Prolog factorial:

```
fact(1,1).
fact(N,Fact) :- N > 1, M is N-1, fact(M,Mfact), Fact is N * Mfact.

-? fact(5,F).
```

For a procedure intended to be used deterministically, with ground arguments, there is very little difference apart from syntax.

Type inference is a very clever bit of ML, while Prolog is typeless.

Q: When is Prolog not ML-like?

```
ML reverse list:
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]

l = reverse([1,2,3,4])

Prolog reverse list:
reverse([],[]).
reverse([X|XS],L) :- reverse(XS,XSrev), append(XSrev,[X],L).

?- reverse([1,2,3,4],L).
?- reverse(X, [4,3,2,1]). -- almost.
?- reverse([1,2,X,4],L).
?- reverse([1,2,X,Y],[4,3,2,A]).
```

Q: When is Prolog not ML-like?

Ultimately you can write a unification function in any language, define a data structure representing relations, and create a backtracking algorithm.

At that point you have 'Prolog' embedded in your language of choice.

Or you can modify Prolog to support higher-order functions...

I.e. implement the relational calculus in ML, or the functional calculus in Prolog.

Q: Last Call Optimisation

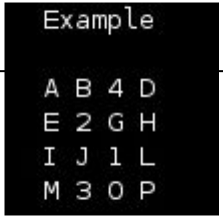
...a space optimization technique, which applies when a predicate is determinate at the point where it is about to call the last goal in the body of a clause.[1]

```
    last([L],L).
    last([_|T],L) :- last(T,L).
```

```
What about:
    foo(_,hello).
    foo(I, W) :- I > 10, J is I - 1, foo(J, W).
```

[1]Sicstus user manual

Sudoku 4x4 (video)



```
Example

A B 4 D
E 2 G H
I J 1 L
M 3 O P
```

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
range([]).
range([H|T]) :- range(1,5,H), range(T).

range(X,_,X).
range(X,Y,Next) :- N is X+1, N < Y, range(N,Y,Next).

diff(A,B,C,D) :- A=\=B, A=\=C, A=\=D, B=\=C, B=\=D, C=\=D. -- arithmetic not equals

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- range(L), rows(L), cols(L), boxes(L).
```

```
[trace]  ?- sudoku([A,B,4,D,E,2,G,H,I,J,1,L,M,3,O,P]).
   Call: (8) sudoku([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? creep
   Call: (9) range([_1472, _1478, 4, _1490, _1496, 2, _1508, _1514|...]) ? skip
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Call: (10) diff(1, 1, 4, 1) ? creep
   Call: (11) 1=\=1 ? creep
   Fail: (11) 1=\=1 ? creep
   Fail: (10) diff(1, 1, 4, 1) ? creep
   Fail: (9) rows([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 1]) ? creep
   Redo: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, _1562]) ? creep
   Exit: (9) range([1, 1, 4, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 2]) ?
```

Sudoku 4x4 (video)

```
Video efficiency improvement

diff(A,B,C,D) :- perm([1,2,3,4],[A,B,C,D]).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- rows(L), cols(L), boxes(L).
```

```
?- sudoku([A,B,4,D,
           E,2,G,H,
           I,J,1,L,
           M,3,O,P]).
```

Sudoku 4x4 (video)

```
Video efficiency improvement

diff(A,B,C,D) :- perm([1,2,3,4],[A,B,C,D]).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- rows(L), cols(L), boxes(L).
```

```
?- sudoku([A,B,4,D,
           E,2,G,H,
           I,J,1,L,
           M,3,O,P]).
```

Prolog vs ML... Imperative program ?:

```
function sudoku(board) {
    answer = make_board(board)
    if rows(answer) and cols(answer) and boxes(answer)
    then
        return answer
    else
        return fail
```

Sudoku 4x4 (video)

```
Video efficiency improvement

diff(A,B,C,D) :- perm([1,2,3,4],[A,B,C,D]).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,C,D), diff(E,F,G,H), diff(I,J,K,L), diff(M,N,O,P).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,E,I,M), diff(B,F,J,N), diff(C,G,K,O), diff(D,H,L,P).

boxes([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff(A,B,E,F), diff(C,D,G,H), diff(I,J,M,N), diff(K,L,O,P).

sudoku(L) :- rows(L), cols(L), boxes(L).
```

```
?- sudoku([A,B,4,D,
           E,2,G,H,
           I,J,1,L,
           M,3,O,P]).
```

Prolog vs ML... Imperative program ?:

```
function sudoku(board) {
        answer = make_board(board)
        if rows(answer) and cols(answer) and boxes(answer)
        then
                return answer
        else
                return fail
                            -- unification and backtracking ???
```

Sudoku 9x9 (Lecture #4)

```
% take(L1,X,L2) succeeds if output list L2 is input list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(L1,L2) succeeds if output list L2 is a permutation of input list L1
perm([],[]).
perm(A,[R|S]) :- take(A,R,P), perm(P,S).

% gen_digits(L) succeeds if L is a permutation of [1,2,3,4,5,6,7,8,9]
gen_digits(L) :- perm([1,2,3,4,5,6,7,8,9],L).
```

SUDOKU 9x9

```
puzzle1 :- RowA = [5,3,_,_,7,_,_,_,_],
           RowB = [6,_,_,1,9,5,_,_,_],
           RowC = [_,9,8,_,_,_,_,6,_],
           RowD = [8,_,_,_,6,_,_,_,3],
           RowE = [4,_,_,8,_,3,_,_,1],
           RowF = [7,_,_,_,2,_,_,_,6],
           RowG = [_,6,_,_,_,_,2,8,_],
           RowH = [_,_,_,4,1,9,_,_,5],
           RowI = [_,_,_,_,8,_,_,7,9],
           gen_digits(RowA),
           gen_digits(RowB),
           gen_digits(RowC),
           gen_digits(RowD),
           gen_digits(RowE),
           gen_digits(RowF),
           gen_digits(RowG),
           gen_digits(RowH),
           gen_digits(RowI),
           Test fixed numbers,
           Test the ‘boxes’,
           Test the ‘columns’,
           Print Solution.
```

```
solve([5,3,_,_,7,_,_,_,_],
      [6,_,_,1,9,5,_,_,_],
      [_,9,8,_,_,_,_,6,_],
      [8,_,_,_,6,_,_,_,3],
      [4,_,_,8,_,3,_,_,1],
      [7,_,_,_,2,_,_,_,6],
      [_,6,_,_,_,_,2,8,_],
      [_,_,_,4,1,9,_,_,5],
      [_,_,_,_,8,_,_,7,9]
     ).
```

```
solve(A,B,C,D,E,F,G,H,I) :- gen_digits(A),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(B),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(C),
                            test_boxes(A,B,C),
                            gen_digits(D),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(E),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(F),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            test_boxes(D,E,F),
                            gen_digits(G),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(H),
                            test_cols([A,B,C,D,E,F,G,H,I]),
                            gen_digits(I),
                            test_boxes(G,H,I),
                            test_cols([A,B,C,D,E,F,G,H,I]).
```

Q: Is our countdown mod iterative deepening?

A: ...

Q: Accumulating the Path to a Solution

```
choose(0, L, [], L).
choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Q: Accumulating the Path to a Solution

```
1. choose(0, L, [], L).
2. choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
3. choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Step 1. Assign 'number' to each of the clauses in the procedure.

Q: Accumulating the Path to a Solution

```
1. choose(0, L, [], L).
2. choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
3. choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Step 1. Assign 'number' to each of the clauses in the procedure.

Step 2. Add 2 arguments (+PathIn, -PathOut) to every (non-deterministic) relation, for the accumulated path so far and the path when this clause succeeds, e.g.:

```
choose(0,L,[],L).
```

becomes:

```
choose(0,L,[],L, PathIn, [1|PathIn]).
```

When we initially call 'choose' we will set 'PathIn' to [], and expect the completed path in PathOut:

```
?- choose(2,[a,b,c,d,e],Chosen,Remaining,[],Path).
```

Q: Accumulating the Path to a Solution

```
1. choose(0,L,[],L, PathIn, [1|PathIn]).
2. choose(N, [H|T], [H|R], S) :- N > 0, N2 is N-1, choose(N2, T, R, S).
3. choose(N, [H|T], R, [H|S]) :- N > 0, choose(N, T, R, S).
```

Step 1. Assign 'number' to each of the clauses in the procedure.

Step 2. Add 2 arguments (+PathIn, -PathOut) to every (non-deterministic) relation, for the accumulated path so far and the path when this clause succeeds.

Step 3.

```
choose(N, [H|T], [H|R], S) :-
     N > 0, N2 is N-1,
     choose(N2, T, R, S).
```

becomes:

```
choose(N, [H|T], [H|R], S, PathIn, PathOut) :-
     N > 0, N2 is N-1,
     choose(N2, T, R, S,[2|PathIn],PathOut).
```

Q: Accumulating the Path to a Solution

```
choose(0,L,[],PathIn, [1|PathIn]).

choose(N, [H|T], [H|R], S, PathIn, PathOut) :-
     N > 0, N2 is N-1,
     choose(N2, T, R, S,[2|PathIn],PathOut).

choose(N, [H|T], R, [H|S], PathIn, PathOut) :-
     N > 0,
     choose(N, T, R, S,[3|PathIn],PathOut).
```

```
?- choose(2,[a,b,c,d,e],Chosen,Remaining,[],Path).

     Chosen = [a, b],                  Chosen = [a, e],
     Remaining = [c, d, e],   . . .    Remaining = [b, c, d, f, g],
     Path = [1, 2, 2]                  Path = [1, 2, 3, 3, 3, 2]
```

Path should be read 'backwards'

Q: Accumulating the Path to a Solution

```
choose(0,L,[],PathIn, [1|PathIn]).

choose(N, [H|T], [H|R], S, PathIn, PathOut) :-
     N > 0, N2 is N-1,
     choose(N2, T, R, S,[2|PathIn],PathOut).

choose(N, [H|T], R, [H|S], PathIn, PathOut) :-
     N > 0,
     choose(N, T, R, S,[3|PathIn],PathOut).
```

```
?- choose(2,[a,b,c,d,e],Chosen,Remaining,[],[1, 2, 3, 3, 3, 2]).
                                Chosen = [a, e],
                                Remaining = [b, c, d, f, g]
```

...DEMO ...

Q: Is our countdown mod iterative deepening?

```prolog
% solve(+N, +SolnAcc, -Soln) succeeds if Soln is a solution
%  to the NxN queens problem and is an extension of
%  the accumulated solution in SolnAcc.
% e.g. :- solve(10, [], Soln), draw(Soln).
solve(N, SolnAcc, SolnAcc) :- length(SolnAcc,N).
solve(N, SolnAcc, Soln) :-
    move(N, SolnAcc, Square),
    solve(N, [Square | SolnAcc], Soln).

% move(+N, +Soln, -Square) acts as a generator for
%  'safe' squares in the next row to those already
%  allocated in the partial solution in Soln.
move(N, [], sq(1, Col)) :- drange(N, Col).
move(N, [sq(I,J) | Rest], sq(Row, Col)) :-
    I < N, Row is I + 1, drange(N, Col), safe(sq(Row, Col), [sq(I,J)|Rest]).

% drange(+N,-X) acts as generator for X = N down to 1.
drange(N, N).
drange(N, X) :- M is N - 1, M > 0, drange(M, X).

% safe_square(+QueenSquare,+Square) succeeds if QueenSquare
%  does not attack Square (or vice-versa).
safe_square(sq(I, J), sq(X, Y)) :- I \== X, J \== Y,
    U1 is I - J, V1 is X - Y, U1 \== V1,
    U2 is I + J, V2 is X + Y, U2 \== V2.

% safe(+Square,+Soln) succeeds if Square does not attack
% any queens in the partial solution Soln.
safe(_, []).
safe(Square, [QueenSquare | L]) :-
    safe_square(QueenSquare, Square), safe(Square, L).
```

N-Queens

?- solve(10,[],Soln).

Q: Is our countdown mod iterative deepening?

```
% solve(+N, +SolnAcc, -Soln) succeeds if Soln is a solution
%  to the NxN queens problem and is an extension of
%  the accumulated solution in SolnAcc.
% e.g. :- solve(10, [], Soln), draw(Soln).
solve(N, SolnAcc, SolnAcc) :- length(SolnAcc,N).
solve(N, SolnAcc, Soln) :-
    move(N, SolnAcc, Square),
    solve(N, [Square | SolnAcc], Soln).

% move(+N, +Soln, -Square) acts as a generator for
%  'safe' squares in the next row to those already
%  allocated in the partial solution in Soln.
move(N, [], sq(1, Col)) :- drange(N, Col).
move(N, [sq(I,J) | Rest], sq(Row, Col)) :-
    I < N, Row is I + 1, drange(N, Col), safe(sq(Row, Col), [sq(I,J)|Rest]).

% drange(+N,-X) acts as generator for X = N down to 1.
drange(N, N).
drange(N, X) :- M is N - 1, M > 0, drange(M, X).

% safe_square(+QueenSquare,+Square) succeeds if QueenSquare
%  does not attack Square (or vice-versa).
safe_square(sq(I, J), sq(X, Y)) :- I \== X, J \== Y,
    U1 is I - J, V1 is X - Y, U1 \== V1,
    U2 is I + J, V2 is X + Y, U2 \== V2.

% safe(+Square,+Soln) succeeds if Square does not attack
% any queens in the partial solution Soln.
safe(_, []).
safe(Square, [QueenSquare | L]) :-
    safe_square(QueenSquare, Square), safe(Square, L).
```

N-Queens

?- solve(10,[],Soln).

DETERMINISTIC

N-Queens with path accumulation

```
solve(N, SolnAcc, SolnAcc, PathIn, [1|PathIn]) :- length(SolnAcc,N).
solve(N, SolnAcc, Soln, PathIn,PathOut) :-
   move(N, SolnAcc, Square,[2|PathIn],PathOut1),
   solve(N, [Square | SolnAcc], Soln, PathOut1, PathOut).

move(N, [], sq(1, Col), PathIn, PathOut) :- drange(N, Col, [1|PathIn], PathOut).
move(N, [sq(I,J) | Rest], sq(Row, Col), PathIn, PathOut) :-
   I < N, Row is I + 1,
   drange(N, Col, [2|PathIn], PathOut), safe(sq(Row, Col), [sq(I,J)|Rest]).

% drange(N,X) acts as generator for X = N down to 1.
drange(N, N, PathIn, [1|PathIn]).
drange(N, X, PathIn, PathOut) :-
      M is N - 1, M > 0,
      drange(M, X, [2|PathIn], PathOut).
```

N-Queens

?- solve(10,[],Soln,[],Path).

ANOTHER DEMO...

Hanoi with path accumulation

```
% Hanoi puzzle
% Towers A, B, C. Each a list of rings 1,2,3 (Head = top).

ok_move([A1|A],[],A,[A1]).
ok_move([A1|A],[B1|B],A,[A1,B1|B]) :- A1 < B1.

% move A -> B
move(state(A,B,C), state(AN,BN,C), PathIn, [1|PathIn]) :- ok_move(A,B,AN,BN).
% move A -> C
move(state(A,B,C), state(AN,B,CN), PathIn, [2|PathIn]) :- ok_move(A,C,AN,CN).
% move B -> A
move(state(A,B,C), state(AN,BN,C), PathIn, [3|PathIn]) :- ok_move(B,A,BN,AN).
% move B -> C
move(state(A,B,C), state(A,BN,CN), PathIn, [4|PathIn]) :- ok_move(B,C,BN,CN).
% move C -> A
move(state(A,B,C), state(AN,B,CN), PathIn, [5|PathIn]) :- ok_move(C,A,CN,AN).


hanoi(States, State_from, State_to, PathIn, PathOut) :-
       move(State_from,State_to, [1|PathIn], PathOut),
       nl,print(States).


hanoi(States, State_from, State_to, PathIn, PathOut) :-
       move(State_from, Next_state, [2|PathIn], PathOut1),
       \+ member(Next_state, States),
       hanoi([Next_state|States],Next_state,State_to, PathOut1, PathOut).


solve(Path) :- hanoi([],state([1,2,3],[],[]),state([],[],[1,2,3]), [], Path).
```

YET ANOTHER DEMO...

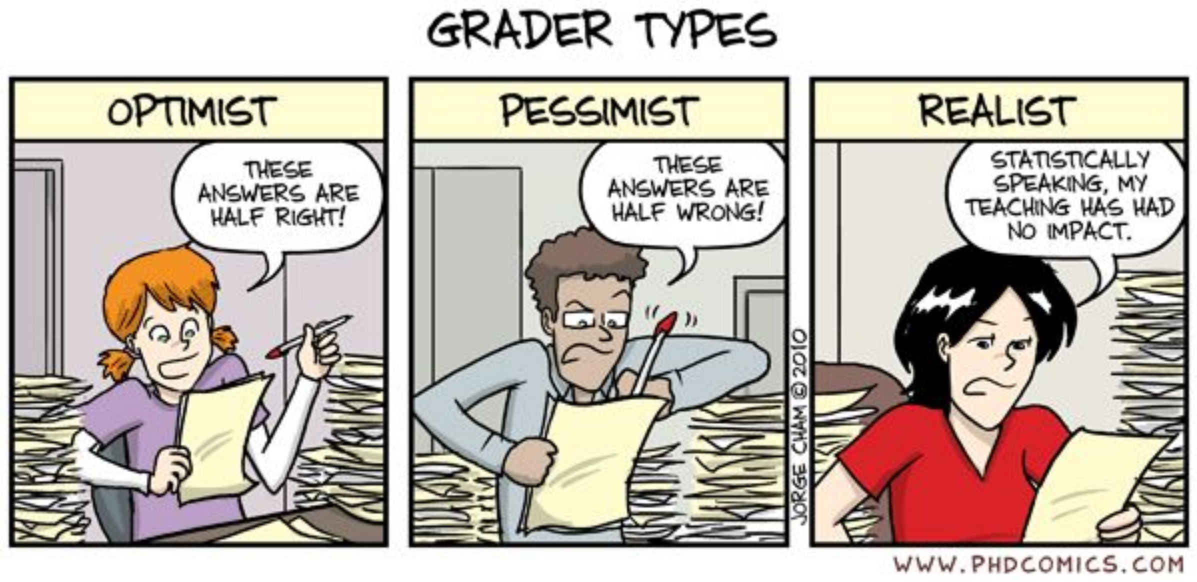
Hanoi with depth limit

```
% Hanoi puzzle
% Towers A, B, C. Each a list of rings 1,2,3 (Head = top).

ok_move([A1|A],[],A,[A1]).
ok_move([A1|A],[B1|B],A,[A1,B1|B]) :- A1 < B1.

% move A -> B
move(state(A,B,C), state(AN,BN,C), PathIn, [1|PathIn]) :- length(PathIn,N), N < Limit, ok_move(A,B,AN,BN).

 •    •    •



solve(Path, Limit) :- hanoi([],state([1,2,3],[],[]),state([],[],[1,2,3]), [], Path, Limit).
```

YET ANOTHER DEMO...

End... of... the... course...

I hope you found the format helpful - please fill out the feedback forms!


GRADER TYPES
OPTIMIST
THESE ANSWERS ARE HALF RIGHT!
PESSIMIST
THESE ANSWERS ARE HALF WRONG!
REALIST
STATISTICALLY SPEAKING, MY TEACHING HAS HAD NO IMPACT.
JORGE CHAM © 2010
WWW.PHDCOMICS.COM