Prolog - 3

- **Append on lists**
- **Generate and test paradigm**
  - **n Queens example**
- **Unification**
  - **Informal definition: isomorphism**
  - **Formal definition: substitution**

Prolog Nomenclature

- ***Unification:*** **(variable bindings) specializes general rules to apply to a specific proof**
- ***Backward Chaining:*** **reduces goal to one or more subgoals**
- ***Backtracking:*** **systematically searches for all possible solutions that can be obtained via unification and backward chaining.**

Append Function

```
append ([ ], A, A).
append([A|B], C, [A|D]) :- append(B,C,D).
```

- **Build a list**

```
?- append([a],[b],Y).
Y = [ a,b ]
```

- **Break a list into constituent parts**

```
?- append(X,[b],[a,b]).
X = [ a ]
?- append([a],Y,[a,b] ).
Y = [ b ]
```


More Append

```
?- append(X,Y,[a,b]).
X = [ ]
Y = [a,b]       ;
X = [a]
Y = [b]     ;
X = [a,b]
Y = [ ]     ;
no
```

Still More Append

- **Generating an unbounded number of lists**

```
?- append(X,[b],Y).
X = [ ]
Y = [b]   ;
X = [ _169]
Y =  [ _169, b]   ;
X = [ _169, _170 ]
Y = [ _169,  _170, b]  ;
etc.
```


Generate and Test

- **Paradigm in which Prolog rules generate potential solutions and then test them for the desired properties**
- **Used often in simulations with lots of alternatives**

n Queens

- **Problem is given an n by n chessboard, place each of n queens on the board so that no queen can attack another in one move**
  - **In chess, queens can move either vertically, horizontally or diagonally.**
- **This problem is a classic generate and test problem**
- **Code on remus:~ryder/314/prolog/programs/queens.pl**

n Queens

```
not(X):- X, !, fail. %same as saw in class
not(_).
in(H,[H|_]). %same as our member_of
in(H,[_|T]):- in(H,T).
%%%nums generates a list of integers between two other
   numbers, L,H by putting the first number at the
   front of the list returned by a recursive call with
   a number 1 greater than the first.  It only works
   when the first argument is bound to an integer. It
   stops when it gets to the higher number.
nums(H,H,[H]).
nums(L,H,[L|R]):- L<H, N is L+1, nums(N,H,R).
%%% The number of queens/size of board - use 4
queen_no(4).
```

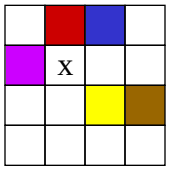
n Queens

```
%%% ranks and files generate the x and y axes of the
    chess board. Both are lists of numbers up to the
    number of queens; that is, ranks(L) binds L to the
    list [1,2,3,…,#queens].
ranks(L):- queen_no(N), nums(1,N,L).
files(L):- queen_no(N), nums(1,N,L).
%%% R is a rank on the board; selects a particular
    rank R from the list of all ranks L.
rank(R):- ranks(L), in(R,L).
%%% F is a file on the board; selects a particular
    file F from the list of all files L.
file(F):- files(L), in(F,L).
```


n Queens

```
%%% Squares on the board are (rank,file) coordinates.
    attacks decides if a queen on the square at rank R1,
    file F1 attacks the square at rank R2, file F2 or
    vice versa. A queen attacks every square on the same
    rank, the same file, or the same diagonal.
attacks((R,_),(R,_)).
attacks((_,F),(_,F)). %a Prolog tuple
attacks((R1,F1),(R2,F2)):- diagonal((R1,F1),(R2,F2)).

%%%can decompose a Prolog tuple by unification
    (X,Y)=(1,2) results in X=1,Y=2; tuples have fixed
    size and there is not head-tail type construct for
    tuples
```



same diagonal, diagonal

same rank

same file

safe placement

n Queens

```
%%% Two squares are on the same diagonal if the slope
  of the line between them is 1 or -1.  Since / is
  used, real number values for 1 and -1 are needed.
diagonal((X,Y),(X,Y)). %degenerate case, 0 length diag
diagonal((X1,Y1),(X2,Y2)):- N is Y2-Y1,D is X2-X1,
  Q is N/D, Q is 1.0E+00. %diagonal needs bound args
diagonal((X1,Y1),(X2,Y2)):- N is Y2-Y1, D is X2-X1,
  Q is N/D, Q is -1.0E+00.
%%%because of use of "is", diagonal is NOT invertible.
```


n Queens

```
%%%placement can be used as a generator.  If placement
  is called with a free variable, it will construct
  every possible list of squares on the chess board.
  The first predicate will allow it to establish the
  empty list as a list of squares on the board.  The
  second predicate will allow it to add any (R,F) pair
  onto the front of a list of squares if R is a rank
  of the board and F is a file of the board.
  placement first generates all 1 element lists, then
  all 2 element lists, etc.  Switching the order of
  predicates in the second clause will cause it to try
  varying the length of the list before it varies the
  squares added to the list
placement([]).
placement([(R,F)|P]):- placement(P), rank(R), file(F).
```

n Queens

```
%%%these two routines check the placement of the next
   queen
%%%Checks a list of squares to see that no queen on
   any of them would attack any other. does by checking
   that position j doesn't conflict with positions
   (j+1),(j+2) etc.
ok_place([]).
ok_place([(R,F)|P]):- no_attacks((R,F),P),ok_place(P).
%%% Checks that a queen at square (R,F) doesn't attack
   any square (rank,file pair) in list L; uses attacks
   predicate defined previously
no_attacks(_,[]).
no_attacks((R,F),[(R2,F2)|P]):-
   not(attacks((R,F),(R2,F2))), no_attacks((R,F),P).
```


n Queens

```
%%% This solution works by generating every list of
   squares, such that the length of the list is the
   same as the number of queens, and then checks every
   list generated to see if it represents a valid
   placement of queens to solve the N queens problem;
   assume list length function
queens(P):- queen_no(N), length(P,N), placement(P),
   ok_place(P).
```

generate code given first **test code follows**

Unification, Informally

- **Intuitively, unification between 2 Prolog terms tries to assign values to the variables so that the resulting trees, representing the terms, are isomorphic (including matching labels)**
- **To use a Prolog rule, we must unify the head of the rule with the subgoal to be proved, "matching" term by term**

Unification, Informally

- **Given a subgoal <functor>(<term>{, <term>}) how to unify it with a clause head?**
  - **Rule and subgoal have same name**
  - **Any uninstantiated variable matches any term**
    - **If term is also an uninstantiated variable, this means if either takes on a value, they both do**
  - **Integer and symbolic constants match themselves, only**
  - **A structured term matches another term iff**
    - **Has same relation name**
    - **Has same number of components and corresponding components match**

Unification

- **Unification looks for the most general (or least restrictive) value to assign**
- **A *substitution* (σ) is a finite map from variables to terms in the language**

```
append([A|B],Y,[A|Z]):- ...
?- append([a,b],[c],W)
```

σ: `A=a, B=[b], Y=[c], W=[a | Z]`

- **A term U is an *instance* of another term T, if there is a substitution σ such that U = T σ.**

Unification

- **Two terms S,T *unify* if they have a common instance U (that is, S σ_1 = T σ_2 = U)**
  - **Note: if variable X is contained in both S and T, then σ_1 and σ_2 both must have the same substitution for X.**
  - **If two terms unify, they can be made identical under some substitution**

Unification

- **There may be more than one substitution to unify two terms**

  `times(Z,times(Y,7))` and `times(4,W)`

  σ_1: **`Z=4, Y=plus(3,5), W=times(plus(3,5),7)`**

  σ_2: **`Z=4, W=times(Y,7)`**

  Which substitution is simpler? less restrictive on the values of the variables? σ_2

Most General Unifier

- **We say γ is the *most general unifier (mgu)* of two terms T, W iff for all other unifiers σ of T,W, T σ is an instance of T γ. therefore, σ can be obtained by a substitution δ applied to γ, $\sigma = \gamma \cdot \delta$**

  ?- `member(A,B)` returns A=_123, B=[A| _] when it could return A= _123, B=[A,b] or A=_123, B= [A, c, d] etc. Note, the 2nd and 3rd B values are obtainable from the mgu by additional substitutions