

# Prolog

# Lists

list

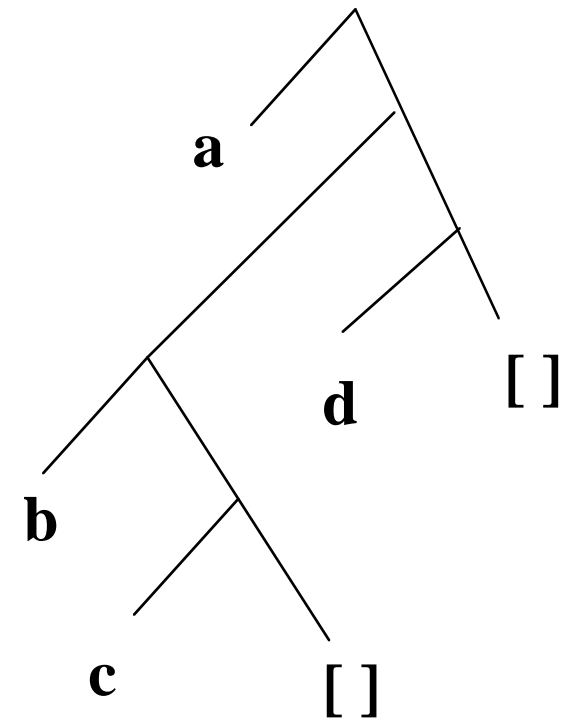
[a,[b,c],d]

head

a

tail

[[b,c],d]

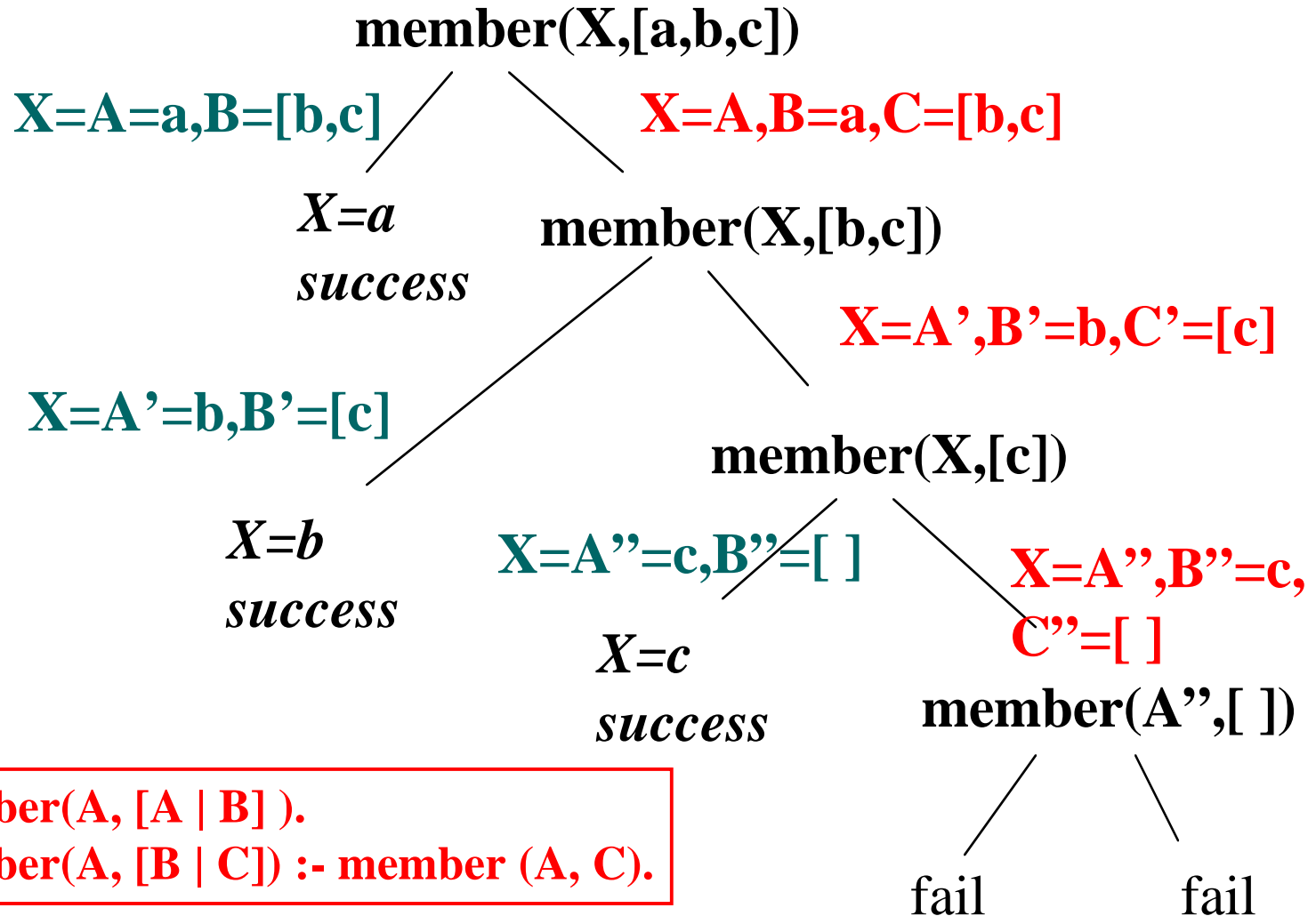


## Unifying Lists:

[ [the, Y] | Z ] = [ [X, hare] | [ is here] ]

*X = the, Y = hare, Z = [ is here]*

# Prolog Search Tree



# List Functions

```
member(A, [A|B]).
```

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

```
length([], 0).
```

```
length([A|B], N) :- length(B, M), N is M+1.
```

**Note the “accumulator” style recursion.**

```
(define (length lst res)
  (if (null? lst)
      res
      (length (cdr lst) (+ res 1))))
```

# 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**

# 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*

	red	blue	
purple	X		
		yellow	brown

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**

# Generate and Test

**generate all sets of winning positions in tic tac toe**

```
row(1).
row(2).
row(3).
col(1).
col(2).
col(3).
position((A,B)):-row(A),col(B).
threePos(t(X,Y,Z)):- position(X),position(Y),position(Z).
alldiff([]).
alldiff([H | T]):- \+(member(H,T)),alldiff(T).
slope((X1,Y1),(X2,Y2),S):- S is (Y2-Y1)/(X2-X1).
win(t((X1,Y1),(X1,Y2),(X1,Y3))):- alldiff([Y1,Y2,Y3]).
win(t((X1,Y1),(X2,Y1),(X3,Y1))):- alldiff([X1,X2,X3]).
win(t((X1,Y1),(X2,Y2),(X3,Y3))):- alldiff([X1,X2,X3]),
                                   slope((X1,Y1),(X2,Y2),S),
                                   slope((X1,Y1),(X3,Y3),S).

winPos(P):-threePos(P), win(P).
```

# Generate and Test

?- winpos(W).

W = t((1,1),(1,2),(1,3)) ;

W = t((1,1),(1,3),(1,2)) ;

W = t((1,1),(2,1),(3,1)) ;

W = t((1,1),(2,1),(3,1)) ;

W = t((1,1),(2,1),(3,2)) ;

W = t((1,1),(2,2),(3,3)) ;

W = t((1,1),(3,1),(2,1)) ;

W = t((1,1),(3,1),(2,1)) ;

# 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  $\langle \text{functor} \rangle(\langle \text{term} \rangle\{, \langle \text{term} \rangle\})$  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* ( $\sigma$ ) is a finite map from variables to terms in the language

```
append([A|B], Y, [A|Z]) :- ...
```

```
?- append([a,b], [c], W)
```

```
 $\sigma$ : A=a, B=[b], Y=[c], W=[a | z]
```

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

# Unification

- **Two terms  $S, T$  *unify* if they have a common instance  $U$  (that is,  $S \sigma_1 = T \sigma_2 = U$ )**
  - **Note: if variable  $X$  is contained in both  $S$  and  $T$ , then  $\sigma_1$  and  $\sigma_2$  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)`

$\sigma_1 : Z=4, Y=\text{plus}(3,5), W=\text{times}(\text{plus}(3,5),7)$

$\sigma_2 : Z=4, W=\text{times}(Y,7)$

**Which substitution is simpler? less restrictive on the values of the variables?**       $\sigma_2$

# Most General Unifier

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