Family Example: Some Facts

- respects(barb,dan).
- respects(barb,katie).
- respects(dan,brian).
- respects(dan,barbara).

Some Queries

?- respects(barb,katie).

yes

?- respects(barb,X).

X=dan; % you type ; and RETURN

X=katie; % you type ; and RETURN

no

% comment to end of line

Some More Queries

?- respects(X,Y).

X=barb Y=dan

?- respects(barb,katie),respects(katie,barb).

no % conjunctive query: , = AND

?- respects(barb,X),respects(X,barb).

X=dan

Some Rules

- **mutualRespect(X,Y):- respect(X,Y),respect(Y,X).**
- **selfRespect(X,X):- respect(X,X).**

  % selfRespect(X,Y):- respect(X,Y), X=Y.
- **sister(X,Y):- female(X),parents(X,Ma,Pa),**

  parents(Y,Ma,Pa),X\==Y.

Some More Rules

- **ancestor(Person,Anc) :- parent(Person,Anc).**
- **ancestor(Person,Anc):- parent(Person,X),ancestor(X,Anc).**

- **parent(a,b).**
- **parent(b,c).**
- **parent(a,e).**
- **parent(b,f).**

?- ancestor(a,X).

% returns in order b, e, c, f

Script: Running Prolog

CREATE FILE NAMED ancestor WITH RULES AND FACTS

> script scriptFileName (CCC Alpha UNIX: garden,wpi,ccc,cpu,...)

% **pl** TO ENTER PROLOG

Welcome to SWI-Prolog (Version 2.9.9)

?- **consult('ancestor').** READS IN FILE NAME ancestor

?- ancestor(a,X).

X = b ;

X = e ;

X = c ;

X = f ;

No

?- **halt.** TO EXIT PROLOG

% **exit** TO EXIT SCRIPT

Unification:
Pattern Matching with Variable Binding

- **same predicate**
- **same number of terms**
- **for each term:**
  - same constant
  - variable and constant
    - variable bound to constant
  - variable and variable
    - if both bound, must be to same
    - if not, bound to each other

The Prolog Inference Engine

- **schoolmates(X,Y) :-**

  student(X, EnterX, ExitX),

  student(Y, EnterY, ExitY),

  X\==Y,

  overlap(EnterX, ExitX, EnterY, ExitY).

- **overlap(EnterX,ExitX,EnterY,ExitY) :-**

  EnterY >= EnterX,

  EnterY < ExitX.

- **overlap(EnterX, ExitX, EnterY, ExitY) :-**

  EnterX >= EnterY,

  EnterX < ExitY.

- **student(bill, 1992,1996).**
- **student(sue,1986,1990).**
- **student(stu, 1994, 1998).**

Goals To Prove

- ?- schoolmates(bill,X).
- UNIFICATION with schoolmates rule head
- Goals to Prove:
  - student(X, EnterX, ExitX),
  - student(Y, EnterY, ExitY),
  - X \== Y,
  - overlap(EnterX, ExitX, EnterY, ExitY).
- UNIFICATION of first goal with student(bill, 1992, 1996).
- Goals to Prove:
  - student(bill, 1992, 1996),
  - student(Y, EnterY, ExitY),
  - bill\==Y,
  - overlap(1992, 1996, EnterY, ExitY).

Execution Trace.

```
?- trace.

?- schoolmates(X,Y).
      Call: ( 7) schoolmates(_G165, _G166)
      Call: ( 8) student(_G165, _L131, _L132)
      Exit: ( 8) student(bill, 1992, 1996)
      Call: ( 8) student(_G166, _L133, _L134)
      Exit: ( 8) student(bill, 1992, 1996)
      Call: ( 8) bill\==bill Fail: ( 8) bill\==bill
```

Trace Continuation 1

Redo: (8) student(_G166, _L133, _L134)

Exit: (8) student(sue, 1986, 1990)

Call: (8) bill\==sue Exit: (8) bill\==sue

Call: (8) overlap(1992, 1996, 1986, 1990)

Call: (9) 1986>=1992

Fail: (9) 1986>=1992

Trace Continuation 2

- **Redo: (8) overlap(1992, 1996, 1986, 1990)**
- **Call: (9) 1992>=1986**
- **Exit: (9) 1992>=1986**
- **Call: (9) 1992?**
- **Fail: (9) 1992?**
- **Fail: (8) overlap(1992, 1996, 1986, 1990)**

Trace Continuation 3

Redo: (8) student(_G166, _L133, _L134)

Exit: (8) student(stu, 1994, 1998)

Call: (8) bill\==stu Exit: (8) bill\==stu

Call: (8) overlap(1992, 1996, 1994, 1998)

Call: (9) 1994>=1992

Exit: (9) 1994>=1992

Call: (9) 1994? ?

Exit: (9) 1994? ?

Exit: (8) overlap(1992, 1996, 1994, 1998)

Exit: (7) schoolmates(bill, stu)

Results

?- schoolmates(X,Y).

X = bill Y = stu;

X = stu Y = bill;

No

```
?- listing.
student(bill, 1992, 1996).
student(sue, 1986, 1990).
student(stu, 1994, 1998).
schoolmates(A, B) :-
        student(A, C, D),
        student(B, E, F),
        A\==B,
        overlap(C, D, E, F).
overlap(A, B, C, D) :- C>=A, C<B.
overlap(A, B, C, D) :- A>=C, A<D.
```

Prolog: Built-in Types

- **atoms**
- **numbers**
- **record-like structure: functor(ListOfComponents)**
- **lists**

Lists

delimiters: [ ]

separator: ,

[good, bad, ugly]

empty list: [ ]

head/car, tail/cdr: [Head|Tail]

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

Example: APPEND

Invertibility

No designated input arguments and return value

```
append( [], Whole ,Whole ).
append( [HStart|TStart], End, [HStart|TWhole] ) :-
                append( TStart, End, TWhole ).

?-append([good],[bad,ugly], Movie).
?-append([good,bad],What,[good,bad,ugly]).
?-append(What,[ugly],[good,bad,ugly]).
?-append(What,WhatElse,[good,bad,ugly]).
```

Example: MYSORT

- mysort(Xs,Ys) :- permutation(Xs, Ys), ordered(Ys).
- ordered([X]).
- ordered([X, Y|Ys]) :- X =< Y, ordered([Y|Ys]).
- permutation([], []).
- permutation(Xs, [Z|Zs]) :- remove(Z, Xs, Ys), permutation(Ys, Zs) .
- remove(X, [X|Xs], Xs).
- remove(X, [Y|Ys], [Y|Zs]) :- remove(X, Ys, Zs).

More Example Rules

- count_up([], 0).
- count_up([X|R], Count):-

  count_up(R, Subcount),

  Count is 1 + Subcount.
- min([H|T],Z):- minsofar(T, H, Z).
- minsofar([], X, X).
- minsofar([H|T], X, Z) :- X =< H, minsofar(T, X, Z).
- minsofar([H|T], X, Z) :- H< X, minsofar(T,H, Z).

bagof

```
commitment(1,1,1,97,9,1).
commitment(2,1,1,98,9,1).
commitment(3,1,2,97,9,1).
commitment(4,2,1,97,9,1).
commitment(5,1,1,97,10,1).
commitment(6,1,1,97,8,2).
commitment(7,1,2,97,7,4).
in97(Id) :- commitment(Id,_,_,97,_,_).
count97commits(C) :-
        bagof(X, in97(X), L),
        count_up(L, C).
```

| ?- ['bagofadvice']. %OR consult(‘bagofadvice’).

| ?- in97(Id).

Id = 1 ;

Id = 3 ;

Id = 4 ;

Id = 5 ;

Id = 6 ;

Id = 7 ;

no

| ?- bagof(X,in97(X),L).

X = _0

L = [1,3,4,5,6,7] ;

no | ?- count97commits(C).

C = 6

Resolution Order

- **tries to unify in fixed order**

  top-down in list of facts and rules

- **tries to satisfy (sub)goals in fixed order**

  left to right in RHS of rule

- **depth-first search**

  new goals put a front of list of goals to solve

Order of Rules Matters Order of Subgoals Matters

```
ancestor(Person, Anc) :- parent(Person, Anc).
ancestor(Person, Anc):- parent(Person, X), ancestor(X, Anc).
parent(a, b).
parent(b, c).
parent(a, e).
parent(b, f).
?- ancestor(a, X).
% returns in order b, e, c, f
```

Order of Rules Matters Order of Subgoals Matters

```
ancestor(Person,Anc) :- parent(Person, X), ancestor(X, Anc).
ancestor(Person,Anc) :- parent(Person, Anc).
parent(a,b).
parent(b,c).
parent(a,e).
parent(b,f).
?- ancestor(a,X).
% returns in order c, f, b, e
```

Order of Rules Matters Order of Subgoals Matters

```
ancestor(Person, Anc):- ancestor(X,Anc), parent(Person,X).
ancestor(Person, Anc) :- parent(Person, Anc).
parent(a,b).
parent(b,c).
parent(a,e).
parent(b,f).
?- ancestor(a,X).
% returns no answer, runs out of heap
```

SOUND, but NOT COMPLETE

- **A set of inference rules or an inference procedure is SOUND if everything(theorem) that is derived using those rules or that procedure logically follows from the facts (and rules).**
- **A set of inference rules or an inference algorithm is COMPLETE if everything(theorem) that logically follows from the facts (and rules) can be derived using those rules or that procedure.**
- **Prolog is SOUND, but NOT COMPLETE**

(Extralogical) Cut !

- **! can appear in RHS of rule (clause body)**

  a :- b, c ,!, d, e.

- **! is a goal which always succeeds, exactly once**
- **no backtracking through !**
- **no possibility to use additional clauses to prove goal**

  insertIfNotThere(X, L, L) :- member(X, L) , ! .

  insertIfNotThere(X, L, [X|L]).

- **! can save computation time, BUT at a cost**

- **! can be used to imitate if-then-else**
- **max(X,Y, X) :- X >= Y.**
- **max(X,Y,Y).**

  ?-max(5, 4, M).

  M=5;

  M=4
- **max(X, Y, X) :- X >= Y, ! .**
- **max(X, Y, Y).**

  ?-max(5, 4, M).

  M=5;

  No

Cut ! CONTINUED

- **BUT max(X,Y,Y). is not a true fact.**
- **It can only be understood in context of previous statement, i.e. must mentally execute to understand, and thus loses advantage of declarative programming**
- **max(X, Y, X) :- X >= Y.**
- **max(X, Y, Y) :- X < Y.**
  - easier to understand (& parallelize), longer to execute

Negation in Prolog

- **not in Prolog is not equivalent to logical not**
- **not can only appear in RHS of rule(clause body)**
- **Negation as Failure**
  - not(X) :- X, !, fail.
  - not(_).
- **Closed World Assumption**

  All relevant knowledge (facts and rules) about domain are included in knowledge base

  – **(if so, if you can't prove a predicate X, you could conclude not X)**

Negation in Prolog

- **When you ‘prove’ something, adding additional info should not affect it.**
- **Using Negation as Failure, this is not the case.**

parent(bob, amy). % only thing in Knowledge Base

?- not(mother(bob, amy).

Yes % since no rule defining predicate mother

female(amy). % add to KB

mother(X,Y) :- parent(X,Y), female(Y). % add to KB

?- not(mother(bob, amy).

No

- **nonmonotonic reasoning**

Example: LAST

- **member(X, [X|_]).**
- **member(X, [_|Ys]) :- member(X,Ys).**
- **last([X|[ ]],X).**
- **last([X|Y], Z) :- last(Y, Z).**

Example: COUNT_UP

- count_up([ ],0).
- count_up([X|R],Count) :-

  count_up(R, Subcount),

  Count is 1 + Subcount.
- sum_up([ ],0).
- sum_up([X|R],Total) :-

  sum_up(R, Subtotal),

  Total is X + Subtotal.

Example: AVERAGE

```
average(L, Average):-
        sum_up(L, Total),
        count_up(L, Count),
        Average is Total/Count.
```

Example: PAYROLL

```
hours(emp1,10).
hours(emp2,20).
hours(emp3,30).
rank(emp1,a).
rank(emp2,a).
rank(emp3,b).
payscale(a,25).
payscale(b,50).
```

PAYROLL (CONTINUED)

- hours(emp4,40).
- rank(emp4,c).
- payscale(c,100).
- pay(X,Y) :-

  hours(X, H),

  rank(X, R),

  payscale(R, S),

  Y is H * S.
- payroll(P) :-

  bagof(Y, X^(pay(X,Y)), L),

  sum_up(L, P).

The Last Example

```
cpi(1,100).
cpi(2, 20).
cpi(3, 300).
cpi(4, 40).
cpi(5, 50).
taxable(1).
taxable(4).
taxable(5).
```

The Last Example (CONTINUED)

- **itemtotal(I,T) :-**

  cpi(I,Q),

  quantity(I,N),

  taxable(I),

  UT is Q * N,

  T is UT * 1.05.

- **itemtotal(I,T) :-**

  cpi(I,Q),

  quantity(I,N),

  not(taxable(I)), T is Q * N.

- **totalbill(G) :-**

  qreadin(0),

  bagof(T,I^(itemtotal(I,T)),L),

  sum_up(L,G).

The Last Example (CONTINUED)

- **totalbill(G) :-**

  qreadin(0),

  bagof(T,I^(itemtotal(I,T)),L),

  sum_up(L,G).

- **qreadin(Num) :-**

  write('Next item, please: '),

  read(X),

  processb(X,Num).

- **processb(stop,Num):-!.**

- **processb(X,Num) :-**

  NN is Num + 1,

  assert(quantity(NN,X)),

  qreadin(NN).

Prolog Search

Implementing Search in Prolog

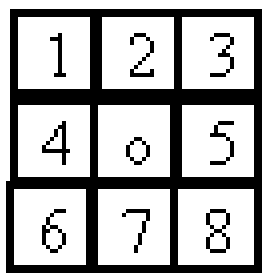
- How to represent the problem
- Uninformed Search
  - depth first
  - breadth first
  - iterative deepening search
- Informed Search
  - Hill climbing
  - Graph Search
    - which can do depth first, breadth first, best first, Algorithm A, Algorithm A*, etc.

Representing the Problem

- Represent the problem space in terms of two predicates:
  - goal/1
  - arc/3
- **goal**(S) is true iff S is a goal state.
- **arc**(**S1**,**S2**,**N**) is true iff there is an operator of cost N that will take us from state S1 to state S2.
- arc(S1,S2) :- arc(S1,S2,_)

Eight Puzzle Example

- Represent a state as a list of the eight tiles and o for blank.
- E.g., [1,2,3,4,o,5,6,7,8] for




```
goal([1,2,3,
      4,o,5,
      6,7,8]).
arc([o,B,C,
     D,E,F,
     G,H,I],
    [B,o,C,
     D,E,F,
     G,H,I]).
```

Missionaries and Cannibals

```
% Represent a state as
% [ML,CL,MR,CL,B]
start([3,3,0,0,left]).
goal([0,0,3,3,X]).

arc([ML,CL,MR,CR,left],
    [ML2,CL,MR2,CR,right]):-
  % two Ms row right
  MR2 is MR+2,
  ML2 is ML-2,
  legal(ML2,CL2,MR2,CR2).
```

```
arc([ML,CL,MR,CR,left],
    [ML2,CL,MR2,CR,right]):-
  % one M & one C row right
  MR2 is MR+1,
  ML2 is ML-1,
  CR2 is CR+1,
  CL2 is CL-1,
  legal(ML2,CL2,MR2,CR2).

Legal(ML,CL,MR,CL) :-
  % is this state a legal
  one?
  ML>0, CL>0, MR>0, CL>0,
  ML>=CL, MR>=CR.
```

Depth First Search

```
%% this is surely the simplest
%% possible DFS.

dfs(S,[S]) :- goal(S).
dfs(S,[S|Rest]) :-
     arc(S,S2),
     dfs(S2,Rest).
```

Depth First Search which avoids loops

```
%% this version of DFS keeps track of the path as
%% it explores, enabling it to avoid loops. It also
%% returns the path from Start to Goal

:- ensure_loaded(library(lists)).

dfs(S,Path) :- dfs1(S,[S],Path).

dfs1(S,Path,ReversePath) :-
  goal(S),
  reverse(Path,ReversePath).

dfs1(S,SoFar,Path) :-
  arc(S,S2),
  \+(member(S2,SoFar)),
  dfs1(S2,[S2|SoFar], Path).
```

Breadth First Search

```
:- use_module(library(queues)).

bfs(S,Path) :-
  empty_queue(Q1),
  queue_head([S],Q1,Q2),
  bfs1(Q2,Path).

bfs1(Q,[G,S|Tail]) :-
  queue_head([S|Tail],_,Q),
  arc(S,G), goal(G).

bfs1(Q1,Solution) :-
  queue_head([S|Tail],Q2,Q1),
  findall([Succ,S|Tail],
            (arc(S,Succ), \+member(Succ,Tail)),
           NewPaths),
  queue_last_list(NewPaths,Q2,Q3),
  bfs1(Q3,Solution).
```

Note on Queues

- :- use_module(library(queues))
- **empty_queue(?Q)**
  - Is true if Queue has no elements.
- **queue_head(?Head, ?Q1, ?Q2)**
  - Q1 and Q2 are the same queues except that Q2 has Head inserted in the front. Can be used to insert or delete from the head of a Queue.
- **queue_last(?Last, ?Q1, ?Q2)**
  - Q2 is like Q1 but have Last as the last element in the queue. Can be used to insert or delete from the end of a Queue.
- **list _ queue(+List, ?Q)**
  - Q is the queue representation of the elements in list List.
- Note: Queues are represented as a pair (L,Hole) where list L ends with a variable unified with Hole.

Iterative Deepening

```
id(S,Path) :-
  from(Limit,1,5),
  id1(S,0,Limit,Path).

id1(S,Depth,Limit,[S]) :-
  Depth<Limit,
  goal(S).

id1(S,Depth,Limit,[S|Rest]) :-
  Depth<Limit,
  Depth2 is Depth+1,
  arc(S,S2),
  id1(S2,Depth2,Limit,Rest).
```

```
% from(-Var,+Val,+Inc)
% instantiates Var to #s
% beginning with Val &
% incrementing by Inc.

from(X,X,Inc).
from(X,N,Inc) :-
  N2 is N+Inc,
  from(X,N2,Inc).
```

```
| ?- from(X,0,5).
X = 0 ? ;
X = 5 ? ;
X = 10 ? ;
X = 15 ? ;
X = 20 ? ;
X = 25 ? ; ...
yes
| ?-
```

Informed Search

- Hill climbing
- General graph search which can be used for
  - depth first search
  - breadth first search
  - best first search
  - Algorithm A
  - Algorithm A*

Hill Climbing

```
hc(Path) :- start(S), hc(S,Path).

hc(S,[S]) :- goal(S), !.

hc(S,[S|Path]) :-
  h(S,H),
  findall(HSS-SS,
          (arc(S,SS),h(SS,HSS)),
          L),
  keysort(L,[BestH-BestSS|_]),
  H>BestH -> hc(BestSS,Path)
           ;(dbug("Local max:~p~n", [S]), fail).
```

Graph Search

The graph is represented by a collections of facts of the form: **node(S,Parent,Arcs,G,H)** where

- **S** is a term representing a state in the graph.
- **Parent** is a term representing S's immediate parent on the best known path from an initial state to S.
- **Arcs** is either *nil* (no arcs recorded, i.e. S is in the set open) or a list of terms *C-S2* which represents an arc from S to S2 of cost C.
- **G** is the cost of the best known path from the state state to S.
- **H** is the heuristic estimate of the cost of the best path from S to the nearest goal state.

Graph Search

In order to use gs, you must define the following predicates:

- **goal(S)** true if S is a term which represents the goal state.
- **arc(S1,S2,C)** true iff there is an arc from state S1 to S2 with cost C.
- **h(S,H)** is the heuristic function as defined above.
- **f(G,H,F)** F is the meteric used to select which nodes to expand next. G and H are as defined above. Default is *"f(G,H,F) :- F is G+H."*.
- **start(S)** (optional) S isthe state to start searching from.

```
gs(Start,Solution) :-
  retractall(node(_,_,_,_,_)),
  addState(Start,Start,0,0),
  gsearch(Path),
  reverse(Path,Solution).

gsearch(Solution) :-
  select(State),
  (goal(State)
   -> collect_path(State,Solution)
    ; (expand(State),gsearch(Solution))).

select(State) :-
  % find open state with minimal F value.
  findall(F-S,
   (node(S,P,nil,G,H),f(G,H,F)),OpenList),
  keysort(OpenList,[_-State|Rest]).

expand(State) :-
  dbg("Expanding state ~p.~n",[State]),
  retract(node(State,Parent,nil,G,H)),
  findall(ArcCost-Kid,
     (arc(State,Kid,ArcCost),
      add_arc(State,Kid,G,ArcCost)),
     Arcs),
   assert(node(State,Parent,Arcs,G,H)).

add_arc(Parent,Child,ParentG,ArcCost) :-
  % Child is a new state, add to the graph.
  (\+node(Child,_,_,_,_)),
  G is ParentG+ArcCost,
  h(Child,H),
  dbg("Adding state ~p with parent ~p and
   cost ~p.~n",[Parent,Child,G]),
  assert(node(Child,Parent,nil,G,H)), !.

add_arc(Parent,Child,ParentG,ArcCost) :-
  % Child state is already in the graph.
  % update cost if the new path better.
  node(Child,_,CurrentParent,Arcs,CurrentGH),
  NewG is ParentG+ArcCost,
  CurrentG>NewG, !,
  dbg("Updating ~p 's cost thru ~p to
    ~p.~n",[State,Parent,NewG]),
  retract(node(Child,_,_,_,_)),
  assert(node(Child,Parent,Arcs,NewG,H)),
  % better way to get to any grandkids?
  foreach(member(ArcCost-Child,Arcs),
     (NewCostToChild is NewG+ArcCost,
     update(Child,State,NewCostToChild))).

add_arc(_,_,_,_).

collect_path(Start,[Start]) :-
  node(Start,Start,_Arcs,0,_H).
collect_path(S,[S|Path]) :-
  node(S,Parent,_,_,_),
  collect_path(Parent,Path).
```

Note on Sorting

- sort(+L1,?L2)
  - Elements of the list L1 are sorted into the standard order and identical elements are merged, yielding the list L2.

    | ?- sort([f,s,foo(2),3,1],L).

    L = [1,3,f,s,foo(2)] ?
- keysort(+L1,?L2)
  - List L1 must consist of items of the form *Key-Value*. These items are sorted into order w.r.t. Key, yielding the list L2. No merging takes place.

    | ?- keysort([3-bob,9-mary,4-alex,1-sue],L).

    L = [1-sue,3-bob,4-alex,9-mary] ?
  - Example:

```
youngestPerson(P) :-
    findall(Age-Person,(person(Person),age(Person,Age)),L),
    keysort(L,[_-P|_]).
```