prolog :-in(2016).

Prolog

- Based on first-order predicate calculus
  - Allow Horn clauses (a subset of predicate calculus) only
  - Execution of a Prolog program is an application of theorem proving
- Original motivation: study of mechanical theorem proving
- Developed in 1970 by Colmerauer & Roussel (Marseilles) and Kowalski (Edinburgh) + others.
  - The name Prolog is taken from PROgramming in LOGic.
- Used in artificial intelligence, databases, expert systems.
  - Non numeric applications
- Popular in Europe
- Adopted by Japan for its 5th Generation Computer program in 80's
- Viewed as competitor for Lisp

Prolog is a declarative programming language

- Express programs in a form of symbolic logic and use a logical inferencing process to produce results
- Logic programs are declarative rather than procedural
  - only the specification of the desired results are stated rather detailed procedures for producing them

Run Prolog

- Popular Prolog implementations
  - GNU Prolog, SWI Prolog, ….
- SWI Prolog is installed on Luna machine
  - The command to run swi prolog: pl
  - The command to run gprolog: gprolog
- Edit the program

  suppose the file sample.pl consists of the following Prolog program:

  likes(john, mary).

- Run the program (red part are user inputs)

  luna:~/440/prolog>gprolog

  GNU Prolog 1.2.16

  By Daniel Diaz

  Copyright (C) 1999-2002 Daniel Diaz

  | ?- consult('sample.pl').

  compiling /fac2/jlu/440/prolog/sample.pl for byte code...

  /fac2/jlu/440/prolog/sample.pl compiled, 47 lines read - 5711 bytes written, 33 ms

  (10 ms) yes

  | ?- likes(john, mary).

  yes

  | ?- likes(john, Whom).

  Whom = mary

  yes

  | ?- likes(john, george).

  No

Prolog program and query

- In Prolog you define a set of facts and rules, then you can query based on your definitions;
- The basic unit of your program is predicate;
- A predicate consists of a name and a sequence of arguments
  - likes(john, mary).
  - "`likes`" is the predicate name, `john` and `mary` are two arguments of the predicate.
- Once you enter this fact into the system, you can ask two types of questions:
  - true/false question: `?-like(john, mary).` does john like mary?
  - what question: `?-likes(john, X).` whom does john like? The X here is a variable.

Components of Prolog programs--facts

- A fact has the form $p(t_1,...,t_n)$.
- Examples:

```
likes(john, mary).
parent(adam, bill).
```

- A fact is part of a program.
- Syntactic conventions
  - It is case sensitive:
    - Predicate and atom names must begin with a lowercase letter;
  - Every statement end with a dot ".".

Facts with variables

- Variables starts with uppercase letters.
- Facts can have variables

```
likes(john, mary).
likes(dwight, X).
                % dwight likes anybody (or anything).
                % ∀X. likes(dwight, X).
```

```
| ?- likes(dwight, mary).
yes
| ?- likes(dwight, john).
yes
| ?- likes(dwight, a).
yes
```

Rules

- The second type of statements is the rule.

  likes(john, X):-likes(mary, X).

  – The symbol ":-" means "if".
  – This rules means that if mary likes X, then john also likes X.
  – Rules make use of variables. ∀X. (`likes(mary, X)`→`likes(john, X)).`

- Running example

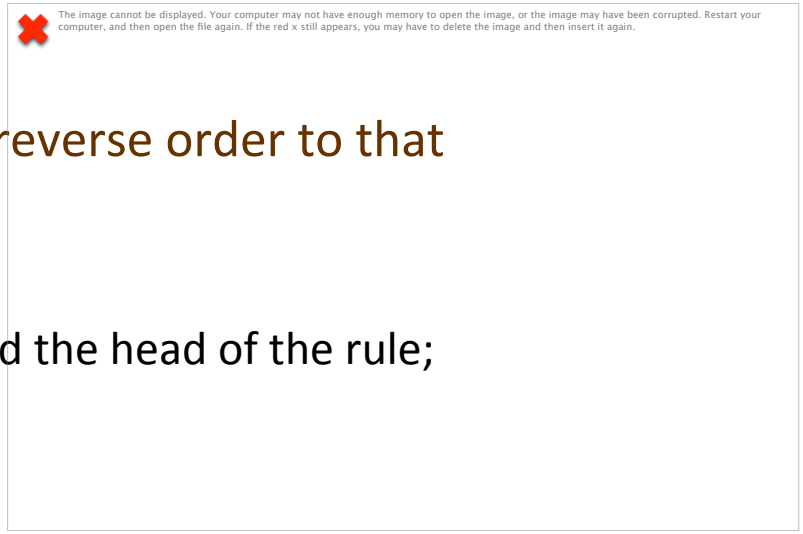
  likes(john, mary). likes(dwight, X). likes(mary, sue).

  likes(john, X):-likes(mary, X).

  – | ?- likes(john, X).
  – X = mary ? ;
  – X = sue
  – yes
  – | ?-
  – Prolog will give you one answer for X;
  – If you hit return it will stop looking for more answers;
  – If you type ";", it will look for another answer;
  – Prolog marks the database where it finds answers in order to start looking again.

Rule: clause body can have multiple predicates

father(X,Y) :- parent(X,Y), male(X).



- The antecedent and consequent are in reverse order to that normally found in logic:

  parent(X,Y) ∧ male(X) ➔father(X,Y)

  – the consequent is written first and called the head of the rule;

  – the antecedent is called the body;

  – Conjunction (and) is written as ",“.

parent(bill, dwight).

male(bill).

father(X,Y):-parent(X,Y), male(X).

| ?- father(bill, X).

X = dwight

General form of a rule (Horn Clause)

- A :- B1, B2, ..., Bn.

  meaning A is true if B1 and B2 and ... Bn are all true

- Clause: a disjunction of literals
- Horn clause: if it contains at most one positive literal.

$$
\begin{aligned}
& A :- B1, B2, \ldots, Bn \\
& \equiv B1 \wedge B2 \wedge \ldots \wedge Bn \rightarrow A \\
& \equiv \neg(B1 \wedge B2 \wedge \ldots \wedge Bn) \vee A \\
& \equiv \neg B1 \vee \neg B2 \vee \ldots \vee \neg Bn \vee A
\end{aligned}
$$

Evaluation--unification

- When the interpreter is given a query, it tries to find facts that match the query.

```
likes(john, mary).
likes(john, sue).
| ?- likes(john, X).
```

- Iterates in order through the program's "likes" clauses
  - First match is `likes(john, mary)`, so it returns `X=mary`.
  - If you hit semicolon ";", it continues and find the next match. So it returns `X=sue`.
- Unification

```
likes(john, mary).
      |     |
likes(john,  X).
```

- Unification is a two-way match

```
likes(Y,  mary).
      |     |
likes(john,  X).
```

likes(Y, mary).

| ?- likes(john, X).

X=mary

Evaluation—backward chaining

- If no outright facts are available, it attempts to satisfy all rules that have the fact as a conclusion.
- For example:

```
parent(bill, dwight).  male(bill).
father(X,Y):-parent(X,Y), male(X).

| ?- trace.
| ?- father(bill, W).
    1  1  Call: father(bill,_16) ?
    2  2  Call: parent(bill,_16) ?                X=bill
    2  2  Exit: parent(bill,dwight) ?                    Y=dwight
    3  2  Call: male(bill) ?
    3  2  Exit: male(bill) ?
    1  1  Exit: father(bill,dwight) ?
W = dwight
yes
```

Recursive rules

- How to define the ancestor predicate?

  ancestor(X,Y):-parent(X,Y).

  ancestor(X,Y):-parent(X, U), parent(U, Y).

  ancestor(X,Y):-parent(X, U), parent(U, V), parent(V, W), parent(W, Y).

  … …

- Prolog is not a pure logic programming language.

- Solution

  ancestor(X,Y):-parent(X,Y).

  ancestor(X,Z):-parent(X,Y), ancestor(Y,Z).

  parent(bill, dwight).

  parent(dwight, ryan).

  parent(dwight, samantha).

  parent(dwight, sue).

  | ?-ancestor(bill, X).

  X=dwight;

  X=ryan;

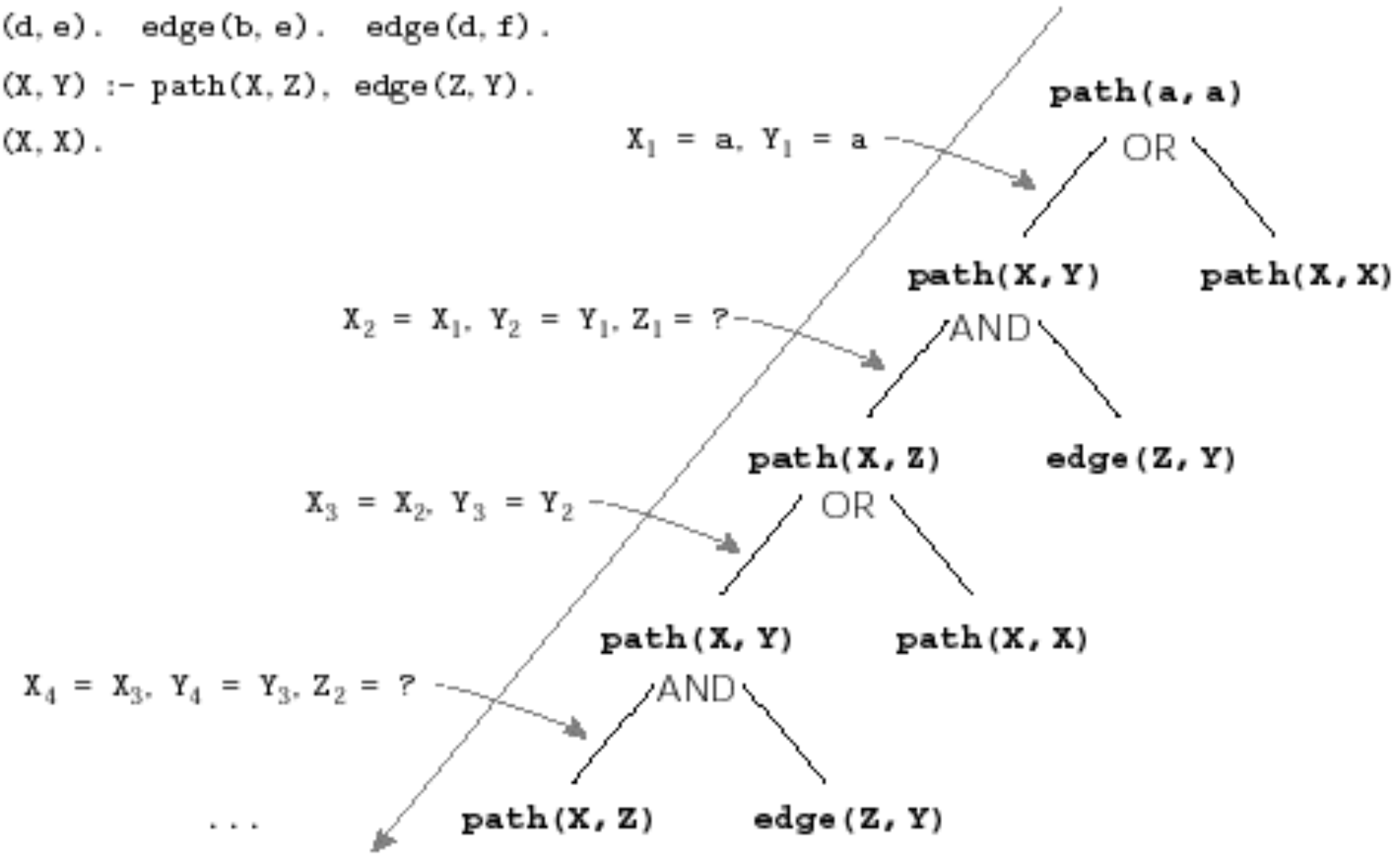
  X=samantha;

  X=sue;

- Another solution?
  - ancestor(X,Y):-parent(X,Y).
  - ancestor(X,Z):-ancestor(X, Y), parent(Y,Z).

Evaluation: infinite loop

```
edge(a, b).   edge(b, c).   edge(c, d).
edge(d, e).   edge(b, e).   edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```

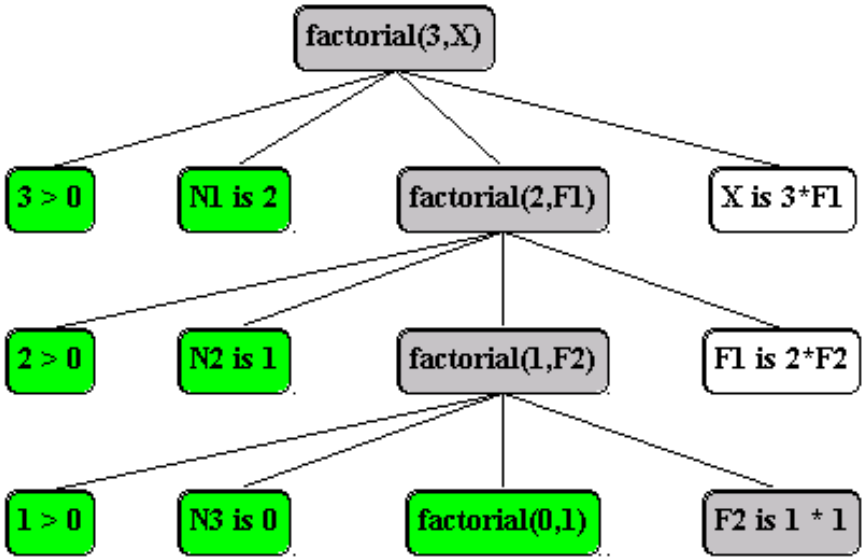


- The correct program:

```
edge(a,b).
edge(b,c).
edge(c,d).
edge(d,e).
edge(b,e).
edge(d,f).
path(X,Y):-edge(X,Z), path(Z,Y).
path(X,X).
```

Evaluation: depth first search

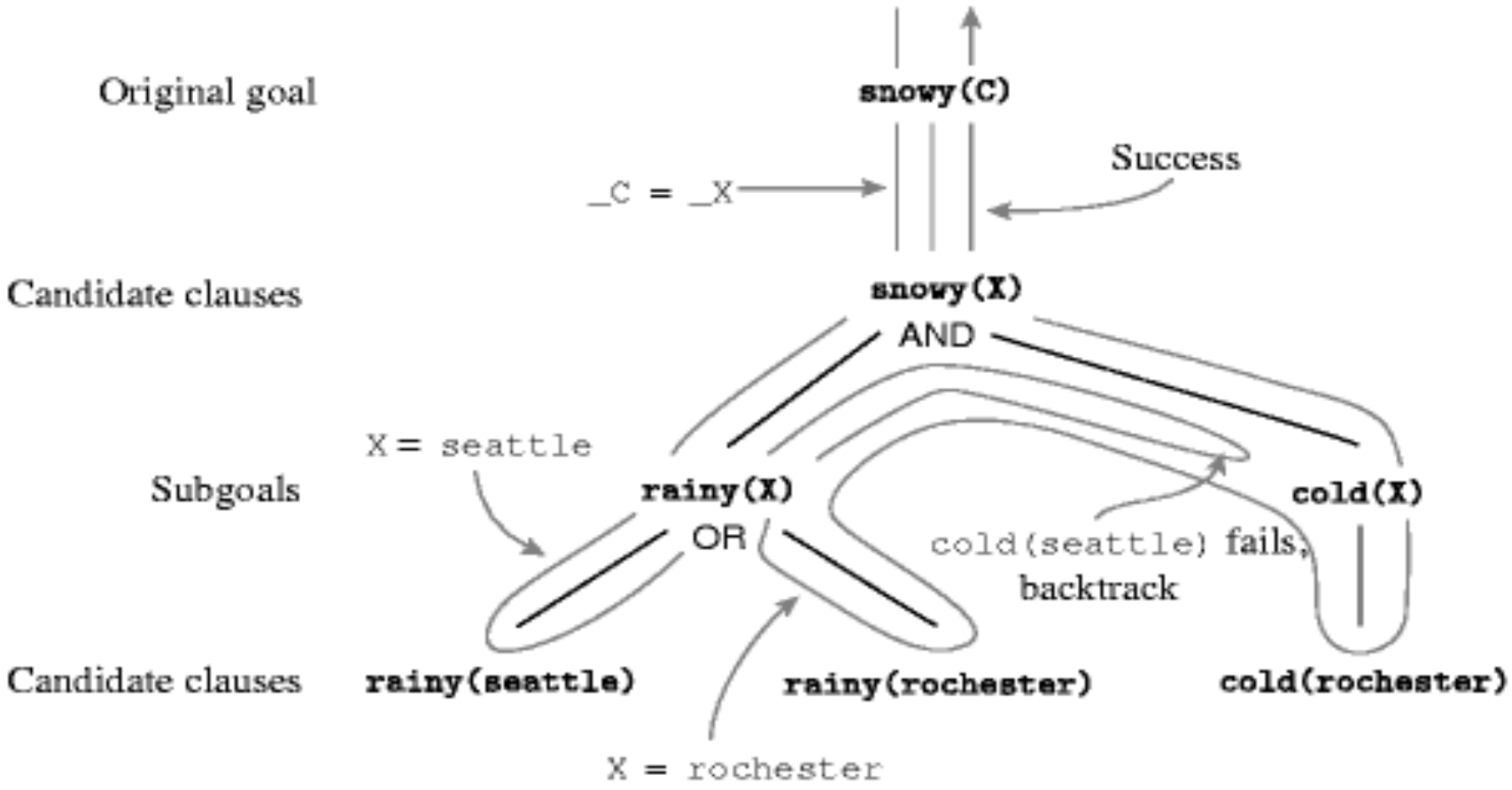
```
f(0, 1).
f(N, F):-N>0,
        N1 is N-1, f(N1, F1),
        F is N*F1.
```



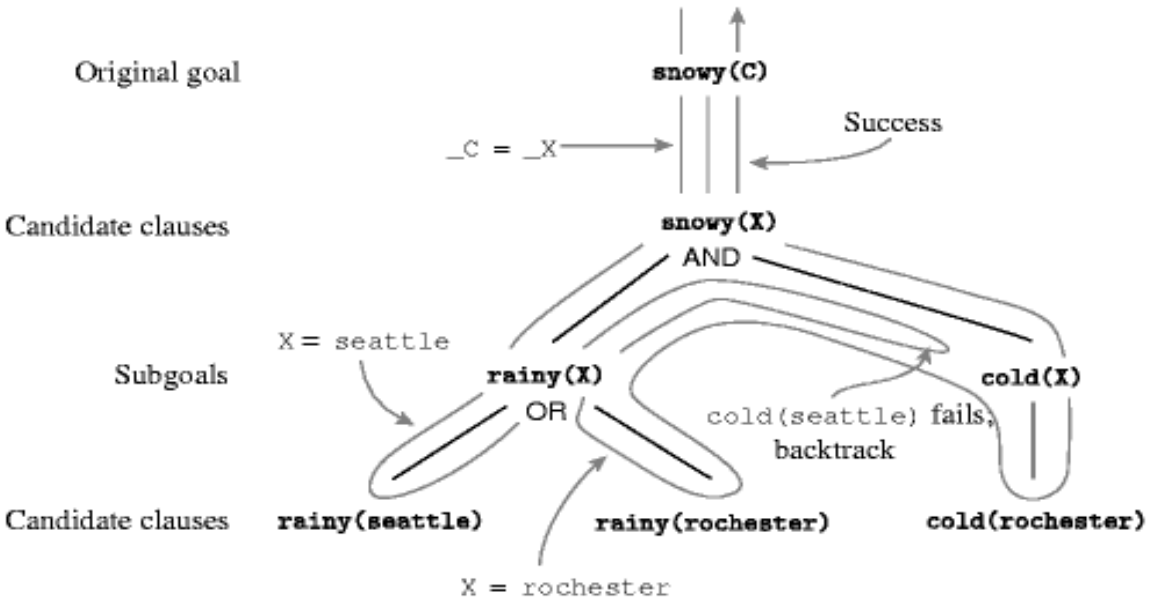

```
| ?- f(3,X).
   1  1  Call: f(3,_16) ?
   2  2  Call: 3>0 ?
   2  2  Exit: 3>0 ?
   3  2  Call: _113 is 3-1 ?
   3  2  Exit: 2 is 3-1 ?
   4  2  Call: f(2,_138) ?
   5  3  Call: 2>0 ?
   5  3  Exit: 2>0 ?
   6  3  Call: _190 is 2-1 ?
   6  3  Exit: 1 is 2-1 ?
   7  3  Call: f(1,_215) ?
   8  4  Call: 1>0 ?
   8  4  Exit: 1>0 ?
   9  4  Call: _267 is 1-1 ?
   9  4  Exit: 0 is 1-1 ?
  10  4  Call: f(0,_292) ?
  10  4  Exit: f(0,1) ?
  11  4  Call: _320 is 1*1 ?
  11  4  Exit: 1 is 1*1 ?
   7  3  Exit: f(1,1) ?
  12  3  Call: _349 is 2*1 ?
  12  3  Exit: 2 is 2*1 ?
   4  2  Exit: f(2,2) ?
  13  2  Call: _16 is 3*2 ?
  13  2  Exit: 6 is 3*2 ?
   1  1  Exit: f(3,6) ?
```

Evaluation: backtracking search

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```



```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```



| ?- snowy(C).

1 1 Call: snowy(_16) ?

2 2 Call: rainy(_16) ?

2 2 Exit: rainy(seattle) ?

3 2 Call: cold(seattle) ?

3 2 Fail: cold(seattle) ?

2 2 Redo: rainy(seattle) ?

2 2 Exit: rainy(rochester) ?

3 2 Call: cold(rochester) ?

3 2 Exit: cold(rochester) ?

1 1 Exit: snowy(rochester) ?

C = rochester

Review

- Syntax: facts, rules. Horn clause

```
f(0, 1).
f(N, F):-N>0,
        N1 is N-1, f(N1, F1),
        F is N*F1.
```

- Evaluation
  - Unification
  - Backward chaining
  - Recursion
  - Depth first search
  - Backtracking

Datalog and Prolog

- What we've seen so far is called Datalog: "databases in logic."
- Prolog is "programming in logic." It goes a little bit further by allowing complex terms, including records, lists and trees.
- These complex terms are the source of the hard thing about Prolog, "unification."

```
at_uwindsor(student(128327, 'Spammy K',date(2, may, 1986))).
at_uwindsor (student(126547, 'Blobby B', date(15, dec, 1985))).
at_uwindsor (student(456591, 'Fuzzy W', date(23, aug, 1966))).
```

- It is no longer a simple fact at_uwindsor(snammy).

at_uwindsor(student(128327, ‘Spammy K',date(2, may, 1986))).

at_uwindsor (student(126547, ‘Blobby B’, date(15, dec, 1985))).

at_uwindsor (student(456591, ‘Fuzzy W', date(23, aug, 1966))).

| ?- at_uwindsor(student(IDNum, Name, date(Day,Month,Year))), Year < 1983.
Day = 23
IDNum = 456591
Month = aug
Name = 'Fuzzy W'
Year = 1966

```
at_uwindsor(student(128327, ‘Spammy K',date(2, may, 1986))).

at_uwindsor (student(126547, ‘Blobby B’, date(15, dec, 1985))).

at_uwindsor (student(456591, ‘Fuzzy W', date(23, aug, 1966))).


| ?- at_uwindsor(student(_, Name, date(_,_,Year))), Year < 1983.
Name = 'Fuzzy W'
Year = 1966

| ?- at_uwindsor(Person),   Person=student(_,_,Birthday),   Birthday=date(_,_,Year), Year < 1983.
Birthday = date(23,aug,1966)
Person = student(456591,'Fuzzy W',date(23,aug,1966))
Year = 1966
```

List

- data structure built into prolog.
  - [ ] % the empty list
  - [1]
  - [1,2,3]
- | separate the head and tail of a list
  - `[a | b]` adds the first argument `a` to the list `b`.
- `[[1,2], 3]` % a list can be heterogeneous.
- All the following are the same as `[a, b, c]`

```
[a, b, c|[]]
[a, b| [c]]
[a | [b, c]]
```

Decompose a list

- How to get the first element of a list? (car in scheme)
  - first(X, List) :- …?
- Solution(s)

```
first(X, List) :- List=[X|Xs].
first(X, [X|Xs]).
first(X, [X|_]).
```

  - eliminate the single-use variable Xs.

Query: first(8, [7,8,9]).

Answer: no

Query: first(X, [7,8,9]).

Answer: X=7

How to define rest of a list? rest(Xs, List):-?

```
rest(Xs, [_|Xs]).
```

List processing (append)

append([],Ys): return Ys

append([X|Xs],Ys): return [X | append(Xs,Ys)]

- In Prolog there are no return values. Rather, the return value is a third argument: append(Xs,Ys,Result).
- This is a constraint saying that Result must be the append of the other lists.
- Any of the three arguments may be known (or partly known) at runtime. We look for satisfying assignments to the others.
- append(Xs,Ys,Result) is true if Xs and Ys are lists and Result is their concatenation (another list).
- Try this:

```
append([],Ys,Ys).
append([X|Xs],Ys,Result) :- … ?
```

```
append([], Ys, Ys).
append([H|T], L, [H|L2]) :- append(T, L, L2).

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

Run the program backwards

- In Prolog, once you've written it, you can also run it backwards!

```
| ?- append([1,2], Y, [1,2,3,4,5,6]).
Y = [3,4,5,6]

| ?- append(A,B,[1,2,3]).
A = [ ],
B = [1,2,3] ;

A = [1],
B = [2,3] ;

A = [1,2],
B = [3] ;

A = [1,2,3],
B = [] ;

no
```

List processing--length

```
len([],0).
len([H|T], X) :-
    len(T, Y),
    X is Y + 1.
```

```
| ?- len([1,2,3],X).
    1  1  Call: len([1,2,3],_22) ?
    2  2  Call: len([2,3],_91) ?
    3  3  Call: len([3],_115) ?
    4  4  Call: len([],_139) ?
    4  4  Exit: len([],0) ?
    5  4  Call: _167 is 0+1 ?
    5  4  Exit: 1 is 0+1 ?
    3  3  Exit: len([3],1) ?
    6  3  Call: _196 is 1+1 ?
    6  3  Exit: 2 is 1+1 ?
    2  2  Exit: len([2,3],2) ?
    7  2  Call: _22 is 2+1 ?
    7  2  Exit: 3 is 2+1 ?
    1  1  Exit: len([1,2,3],3) ?

X = 3
yes
```

List processing: member

- member(X,Y) should be true if X is any object, Y is a list, and X is a member of the list Y.

```
member(X, [X|_]).      % same as first(..).
member(X, [H|T]) :- member(X, T).

?- member(3, [1,2,3]).   %member test
yes

?- member(X, [1,2,3]).   % member generator
X = 1 ;
X = 2 ;
X = 3 ;
No
```

```
?-member(2, X).  % list generator

X = [2|_] ? ;
X = [_,2|_] ? ;
X = [_,_,2|_] ? ;
X = [_,_,_,2|_] ?
...
```

- Query: member(7, List), member(8,List), length(List, 3).
  - Answer: doesn't terminate
- Query: length(List, 3), member(7, List), member(8,List).
  - Answer: List=[7, 8, X] ;
    List=[7, X, 8] ;
    List=[8, 7, X] ;
    List=[X, 7, 8] ;
    List=[8, X, 7] ;
    List=[X, 8, 7]
- Again, only "theoretically" Prolog does not depend on the orders of predicates and clauses.

Example: test subset relationship

```
subset([ ],Y).
subset([A|X],Y) :- member(A,Y), subset(X,Y).
```

```
| ?- subset ([a, b, c], [a, c, d, b]).
```

true ?

Example: nth element

```
find_nth([H|T], 1, H).
find_nth([H|T], N, X):- N1 is N-1, find_nth(T,N1,X).

| ?- find_nth([a, b, c, d], 2, X).
```

X = b ?

Example: what are the changes for a dollar?

```
% suppose there are half-dollars, quarters, dimes,
    nickels.  P is the purchase

change ([H,Q,D,N,P]) :-

    member(H,[0,1,2]),                          // Half-dollars

    member(Q,[0,1,2,3,4]),                       // quarters

    member(D,[0,1,2,3,4,5,6,7,8,9,10]) ,    // dimes

    member(N, [0,1,2,3,4,5,6,7,8,9,10,      // nickels

            11,12,13,14,15,16,17,18,19,20]),

    S is 50*H + 25*Q +10*D + 5*N,

    S =< 100, P is 100-S.
```

```
| ?- change([H,Q,D,N,P]).

D = 0
H = 0
N = 0
P = 100
Q = 0 ? ;

D = 0
H = 0
N = 1
P = 95
Q = 0 ? ;

D = 0
H = 0
N = 2
P = 90
Q = 0 ?

...
```

```
| ?- change([H,Q,D,N,20]).

D = 0
H = 0
N = 16
Q = 0 ? ;

D = 1
H = 0
N = 14
Q = 0 ? ;

D = 2
H = 0
N = 12
Q = 0 ?
```

Insertion sort

```
isort([ ],[ ]).
isort([X|UnSorted], AllSorted) :-
    isort(UnSorted, Sorted),
    insert(X, Sorted, AllSorted).

insert(X, [ ], [X]).
insert(X, [Y|L], [X, Y|L]) :- X =< Y.
insert(X, [Y|L], [Y|IL]) :-
        X > Y, insert(X, L, IL).
```

```
| ?- isort([1, 3, 2], X).
   1  1  Call: isort([1,3,2],_22) ?
   2  2  Call: isort([3,2],_91) ?
   3  3  Call: isort([2],_115) ?
   4  4  Call: isort([],_139) ?
   4  4  Exit: isort([],[]) ?
   5  4  Call: insert(2,[],_165) ?
   5  4  Exit: insert(2,[],[2]) ?
   3  3  Exit: isort([2],[2]) ?
   6  3  Call: insert(3,[2],_194) ?
   7  4  Call: 3=<2 ?
   7  4  Fail: 3=<2 ?
   7  4  Call: 3>2 ?
   7  4  Exit: 3>2 ?
   8  4  Call: insert(3,[],_181) ?
   8  4  Exit: insert(3,[],[3]) ?
   6  3  Exit: insert(3,[2],[2,3]) ?
   2  2  Exit: isort([3,2],[2,3]) ?
   9  2  Call: insert(1,[2,3],_22) ?
  10  3  Call: 1=<2 ?
  10  3  Exit: 1=<2 ?
   9  2  Exit: insert(1,[2,3],[1,2,3]) ?
   1  1  Exit: isort([1,3,2],[1,2,3]) ?
```

A more declarative sort program

```
sort(List, Sorted):-permutation(List, Sorted), is_sorted(Sorted).

is_sorted([ ]).
is_sorted([ _ ]).
is_sorted([X,Y|T]):-X<=Y, is_sorted([Y|T]).


perm([ ], [ ]).
perm(List, [H|Perm]) :-takeout(H, List, Rest),
            perm(Rest, Perm).

takeout(X, [X|Xs], Xs) .
takeout(X, [Y|Xs], [Y|Ys]):-takeout(X,Xs,Ys).
```

Summary of Prolog Programs

- Program = a bunch of axioms
- Run your program by:
  - Enter a series of facts and declarations
  - Pose a query
  - System tries to prove your query by finding a series of inference steps
- "Philosophically" declarative
- Actual implementations are deterministic
- Many concepts not covered
  - Negation
  - Cut.

Questions?