Topic 7. An introduction to PROLOG

- Free -PROLOG interpreters
  - SWI -PROLOG http://www.swi-prolog.org
  - GNU-PROLOG http://pauillac.inria.fr/~diaz/gnu-prolog/#download
  - Visual -PROLOG http://www.visual-prolog.com

Introduction

- A PROLOG program is a set of specifications in the first-order predicate calculus describing the objects and relations in a program domain.
  - **Specification**
- The PROLOG interpreter processes queries, searching the database to find out whether the query is a logical consequence of the database of specifications.
  - **Control**
- **The specification and control are separated!**

Syntax

Predicate calculus	PROLOG
$\wedge$	,
$\vee$	;
$\leftarrow$	:-
$\neg$	not

First PROLOG program (1)

- First PROLOG program

```
likes(george,kate).
likes(george,susie).
likes(george,wine).
likes(kate,gin).
likes(kate,susie).
```

terminator

Search order.

first.pl

- The programmer must be aware of **the order** in which PROLOG searches entries in the database.
- PROLOG assumes that all knowledge of the world is present in the database.
  - Unique name axiom.
  - Closed world axiom.
  - Domain closure axiom.

First PROLOG program (2)

question

```
?- likes(george,pizza).
```

- Does George like pizza?

```
?- likes(george,X).
```

- What is the food that George likes?

```
?- likes(george,X), likes(kate,X).
```

- What is the food that George and Kate like?

Specifications of Rules (1)

$$\boxed{\mathbf{P} \leftarrow \mathbf{Q} \wedge \mathbf{R}}$$

where P, Q, and R are terms.

- Declarative readings
  - P is true if Q and R are true.
  - From Q and R follows P.
- Procedural readings
  - To solve problem P, first solve subproblem Q and the the subproblem R.
  - To satisfy P, first satisfy Q and then satisfy R.

Specifications of Rules (2)

```
friends(X,Y) :- likes(X,Z), likes(Y,Z).
```

- This expression can be read as “X and Y are friends if there exists a Z such that X likes Z and Y likes Z.”
  - friends(X,Y) is a logical consequence of likes(X,Z) and likes(Y,Z).
  - The scopes of X, Y, and Z are limited to the rule friends.

Example: Family

```
parent( pam, bob).   % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female( pam).        % Pam is female
male( tom).          % Tom is male
male( bob).
female( liz).
female( ann).
female( pat).
male( jim).

offspring( Y, X)  :-   % Y is an offspring of X if
   parent( X, Y).      % X is a parent of Y
```

```
mother( X, Y)  :-        % X is the mother of Y if
   parent( X, Y),        % X is a parent of Y and
   female( X).           % X is female

grandparent( X, Z)  :-   % X is a grandparent of Z if
   parent( X, Y),        % X is a parent of Y and
   parent( Y, Z).        % Y is a parent of Z

sister( X, Y)  :-        % X is a sister of Y if
   parent( Z, X),
   parent( Z, Y),        % X and Y have the same parent and
   female( X),           % X is female and
   different( X, Y).     % X and Y are different

predecessor( X, Z)  :-   % Rule prl: X is a predecessor of Z
   parent( X, Z).

predecessor( X, Z)  :-% Rule pr2: X is a predecessor of Z
   parent( X, Y),
   predecessor( Y, Z).
```

Family.pl

Relation between PROLOG and logic

- PROLOG’s syntax is that of the first-order predicate calculus formula written in the clause form (the Horn clause).
  - The procedural meaning of PROLOG is based on the resolution principle for mechanical theorem proving.
    - PROLOG uses a special strategy for resolution theorem proving called SLD.
  - Matching in PROLOG corresponds to unification in logic.
    - However, for efficiency reasons, most PROLOG systems do not implement matching in a way that exactly corresponds to unifications.

Logic program

- A logic program is a set of universally quantified expressions in first-order predicate calculus of three forms (the Horn clauses)
    1. Goals
       $\leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_n$
    2. Facts:
       $a_1 \leftarrow$
       $a_2 \leftarrow$
       ...
       $a_m \leftarrow$
    3. Rule
       $\underbrace{a}_{\text{head}} \leftarrow \overbrace{b_1 \wedge b_2 \wedge \ldots \wedge b_n}^{\text{body}}$

Goal

$\leftarrow a1 \wedge a2 \wedge ... \wedge an \Rightarrow \neg(a1 \wedge a2 \wedge ... \wedge an) \vee \square \Rightarrow \neg a1 \vee \neg a2 \vee ... \vee \neg an$

1. Search for the first clause in PROLOG P whose head unifies with a1. ξ is the unification.

$a1 \leftarrow b1 \wedge b2 \wedge ... \wedge bm \Rightarrow a1 \vee \neg(b1 \wedge b2 \wedge ... \wedge bm)$ → resolution

$\neg(b1 \wedge b2 \wedge ... \wedge bm) \vee \neg a2 \vee ... \vee \neg an$

$\Rightarrow \neg(b1 \wedge b2 \wedge ... \wedge bm) \vee \neg(a2 \wedge ... \wedge an$

$\Rightarrow \leftarrow b1 \wedge b2 \wedge ... \wedge bm \wedge a2 \wedge ... \wedge an$

$(b1 \wedge b2 \wedge ... \wedge bm \wedge a2 \wedge ... \wedge an)\xi,$

2. Search for the first clause whose head unifies with b1. ϕ is the unification.

$b1 \leftarrow c1 \wedge c2 \wedge ... \wedge cp$

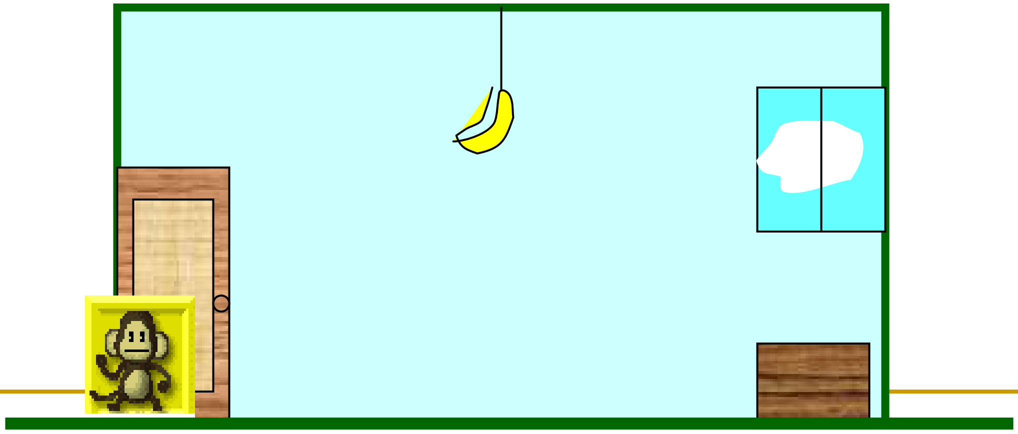
$(c1 \wedge c2 \wedge ... \wedge cp \wedge b2 \wedge ... \wedge bm \wedge a2 \wedge ... \wedge an)\xi\phi,$

If the goal is reduced to the empty clause ($\square$) then the composition of unifications that made the reductions ($\square$)$\xi\phi... \omega$, $\xi\phi... \omega$ provides an interpretation under which the goal clause was true.

PROLOG implements its left-to-right, depth-first search of the clause space.

Example: monkey and banana

Problem. There is a monkey at the door into a room. In the middle of the room a banana is hanging from the ceiling. The monkey is hungry and wants to get the banana but it cannot stretch high enough from the floor. At the window of the room there is a box the monkey may use. The monkey can perform the following actions: walk on the floor, climb the box, push the box around, and grasp the banana if standing on box directly under the banana. Can the monkey get the banana?



monkey.pl

The state of the monkey world

state(Horizontal position of monkey, Vertical position of monkey, Position of box, Monkey has or has no banana).

Horizontal position of monkey: atdoor, middle, atwindow
Vertical position of monkey: onfloor, onbox
Position of box: atdoor, middle, atwindow
Monkey has or has no banana: has, hasnot

State transition

move(state1, method, state2)

method

1.	Grasp banana	move(state(middle,onbox,middle,hasnot),grasp,state(middle,onbox,middle,has)).
2.	Climb(VPos1,VPos2)	move(state(Pos,onfloor,Pos,H),climb,state(Pos,onbox,Pos,H)).
3.	Push(HPos1,HPos2)	move(state(P1,onfloor,P1,H),push(P1,P2),state(P2,onfloor,P2,H)).
4.	Walk(HPos1,HPos2)	move(state(P1,onfloor,P,H),walk(P1,P2),state(P2,onfloor,P,H)).

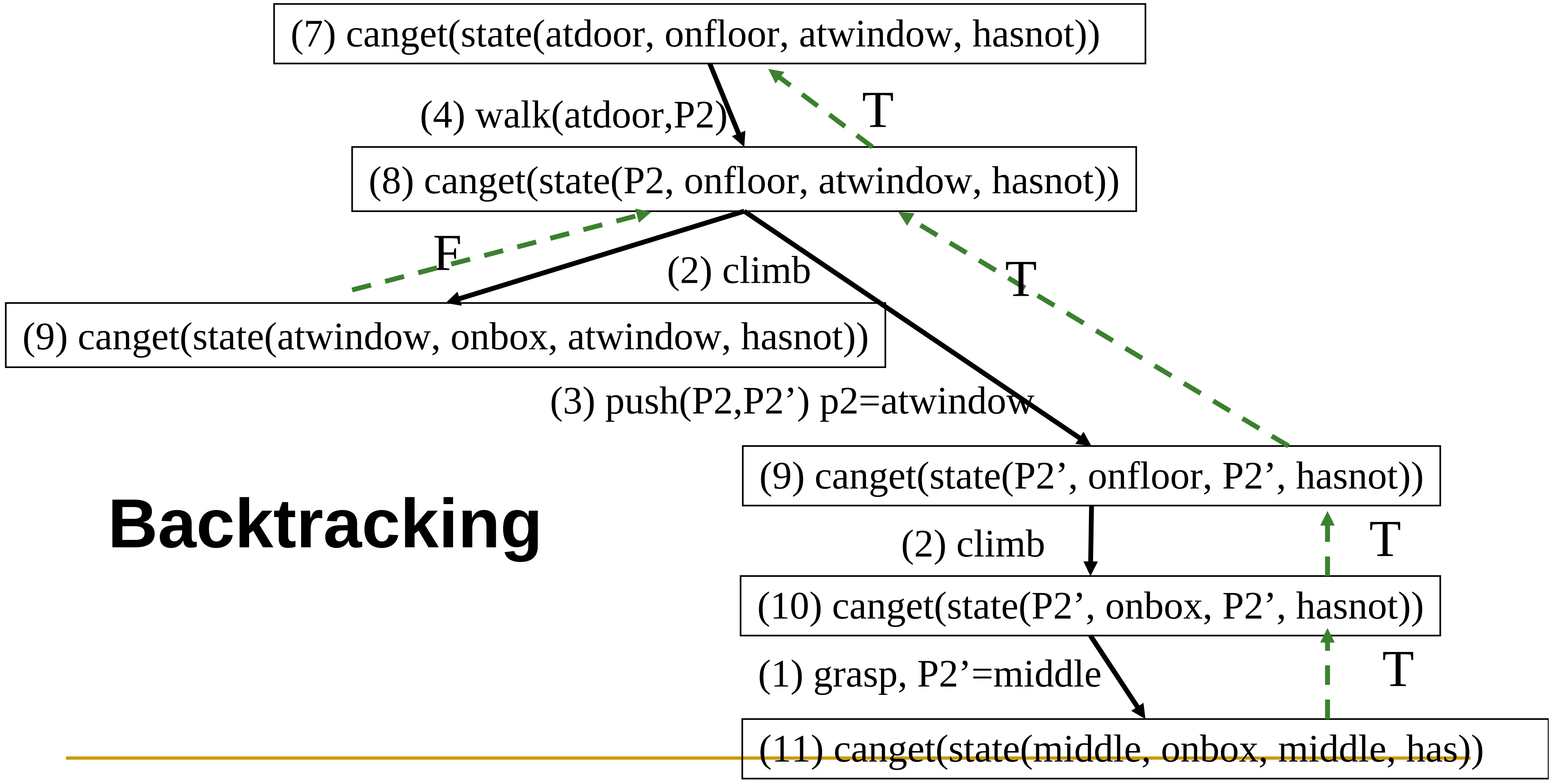
Answer

```
canget(state(_,_,_,has)).
canget(State1):-
        move(State1,Move,State2),
        canget(State2).
```

Question

```
?-canget(state(atdoor,onfloor,atwindow,hasnot)).
```

1.	Grasp banana	move(state(middle,onbox,middle,hasnot),grasp,state(middle,onbox,middle,has)).
2.	Climb(VPos1,VPos2)	move(state(Pos,onfloor,Pos,H),climb,state(Pos,onbox,Pos,H)).
3.	Push(HPos1,HPos2)	move(state(P1,onfloor,P1,H),push(P1,P2),state(P2,onfloor,P2,H)).
4.	Walk(HPos1,HPos2)	move(state(P1,onfloor,P,H),walk(P1,P2),state(P2,onfloor,P,H)).

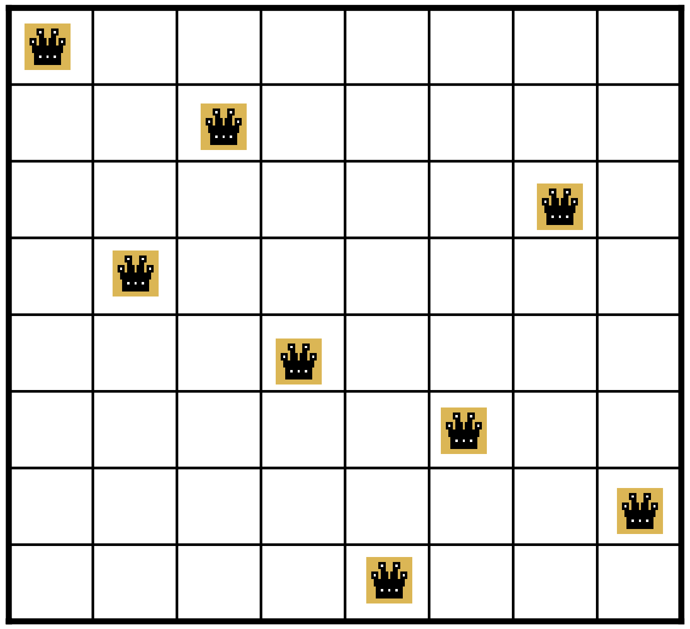

(7) canget(state(atdoor, onfloor, atwindow, hasnot))
(4) walk(atdoor,P2)
T
(8) canget(state(P2, onfloor, atwindow, hasnot))
F
(2) climb
T
(9) canget(state(atwindow, onbox, atwindow, hasnot))
(3) push(P2,P2') p2=atwindow
(9) canget(state(P2', onfloor, P2', hasnot))
(2) climb
T
(10) canget(state(P2', onbox, P2', hasnot))
(1) grasp, P2'=middle
T
(11) canget(state(middle, onbox, middle, has))

Backtracking

```
%State transition
%Grasp banana
move(state(middle,onbox,middle,hasnot),grasp,state(middle,onbox,middle,has)).

%Climb(VPos1,VPos2)
move(state(Pos,onfloor,Pos,H),climb,state(Pos,onbox,Pos,H)).

%Push(HPos1,HPos2)
move(state(P1,onfloor,P1,H),push(P1,P2),state(P2,onfloor,P2,H)).

%Walk(HPos1,HPos2)
move(state(P1,onfloor,P,H),walk(P1,P2),state(P2,onfloor,P,H)).

canget(state(_,_,_,has)).
canget(State1):-
            move(State1,Move,State2),
            canget(State2).
```

Example: Eight queens problem

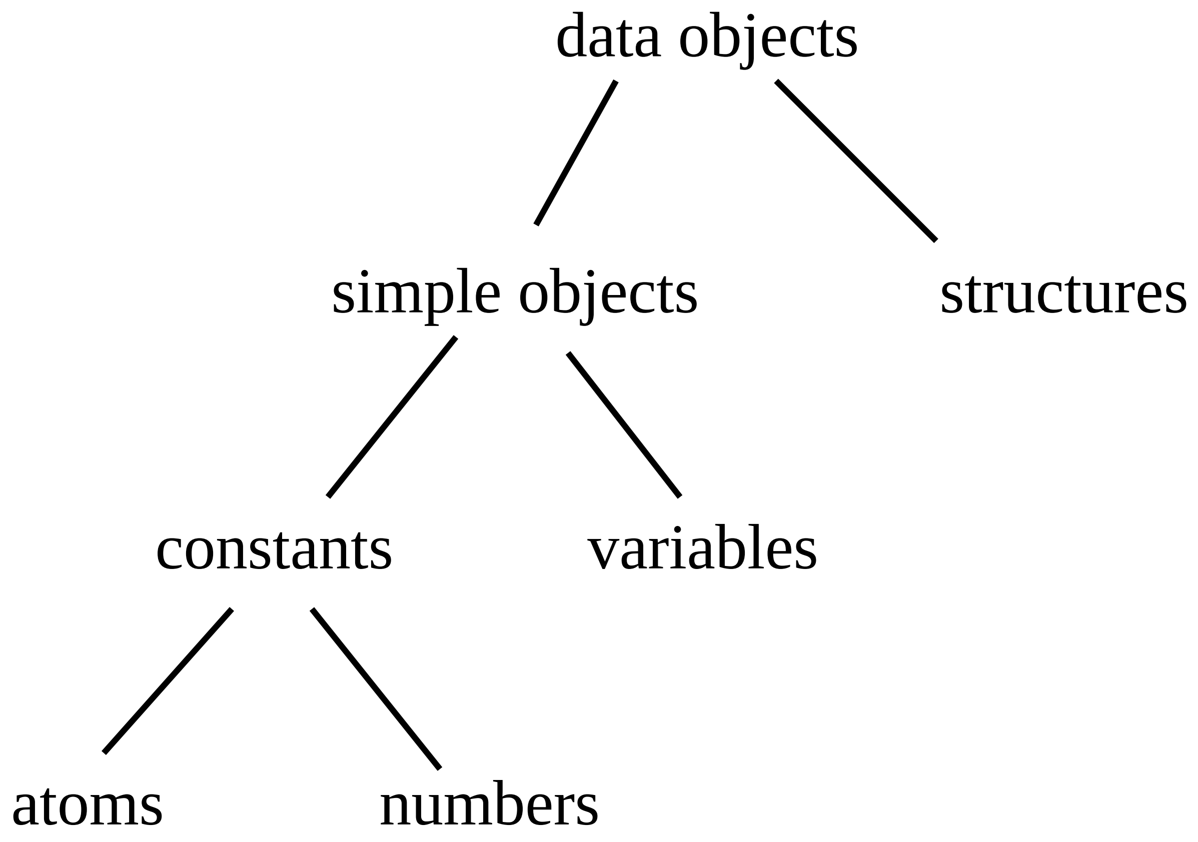
- The eight queens problem is to place eight queens on the empty chessboard in such a way that no queen attacks any other queens.



eight_queen.pl

```
valid([],0,_,_).
valid([Q|Qboard],QC,C,R) :-                    % check validity of configuration
          valid(Qboard,CC,C,R),
          QC is CC + 1,
          Dc is abs(QC-C),
          Dr is abs(R-Q),
          Dr =\= 0,
          Dc =\= Dr.

find_valid_position(Q,C,R,R) :-                % find a valid position
          valid(Q,_,C,R).
find_valid_position(Q,C,IR,RR) :-
          NR is IR + 1,
          NR =< 8,
          find_valid_position(Q,C,NR,RR).

eight_queen(Q,9,Q).
eight_queen(Q,C,RQ):-
          find_valid_position(Q,C,1,X),
          NQ = [X | Q],
          NC is C+1,
          eight_queen(NQ,NC,RQ).
solution(Pos) :-
          eight_queen([],1,Pos).
```

Data objects



Atoms

- Atoms can be constructed in three ways:
  - Strings of letters, digits, and the underscore character, starting with a lowercase letter.
    - Example abc, ab_cd, x123, _123.
  - Strings of special characters. ✗
    - Example …,.:.
  - Strings of characters enclosed in single quotes.
    - Example ‘Tom’, ‘Hello’.

Numbers

- Numbers used in PROLOG included integer numbers and real numbers.
  - Integer numbers: 1, 100, 123
  - Real numbers: 100.0, 0.001

Variables (1)

- Variables are strings of letters, digits and underscore characters. They start with an **upper-case letter** or an **underscore character**.

  Example.
  - X
  - Result
  - Tom
  - _123
  - X_y
  - **_ (the so-called anonymous variable)**

  Example haschild(X) :- parent(X,_).

  We don't care about the value of this variable.

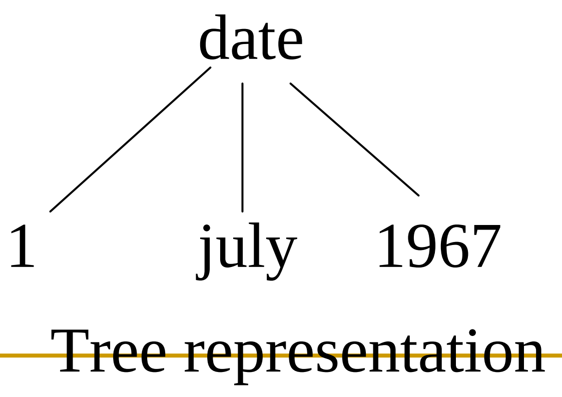
Variables (2)

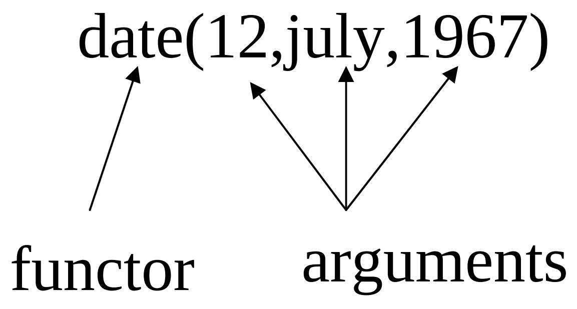
- ?-likes(george,_).
  - Is there anything that George likes?
  - We are interested in whether George likes something but not in the thing that George likes.

Structures (1)

- Structured objects are objects that have several components. The components themselves can be structures.

date(12,july,1967)





Structures (2)

- All structured objects in PROLOG are represented by trees.

List (1)

- A list is a sequence of any number of items.
  - The empty list: [].
  - A list of three items: [item1, item2, item3].
- A list in PROLOG can be viewed as consisting of two parts:
  1) the first item, called the head of the list;
     - The head can be any PROLOG object.
  2) the remaining items, called the tail of the list.
     - **The tail is a list.**
- Lists are handled in PROLOG as a special case of binary trees.

List (2)

Concatenate two lists.

```
conc([],L2,L2).
conc([X|T1],L2,[X|L3]):-
    conc(T1,L2,L3).
```

Head Tail

Determine whether or not an item is in a list.

```
member(X,[]):-!, fail.
member(X, [X|L]):-!.
member(X,[_|L]):-member(X,L).
```

List (3)

Add an item to a list.

```
add(X,L,[X|L]).
```

Delete an item from a list.

```
del(X,[X|L],L).
del(X,[Y|L1],[Y|L2]]):- del(X,L1,L2).
```

Insert an item into a list.

```
insert(X,L1,[X|L1]).
```

Insert an item into a list at any place.

```
insert(X,L1,L2):-
   del(X,L2,L1).
```

List (4)

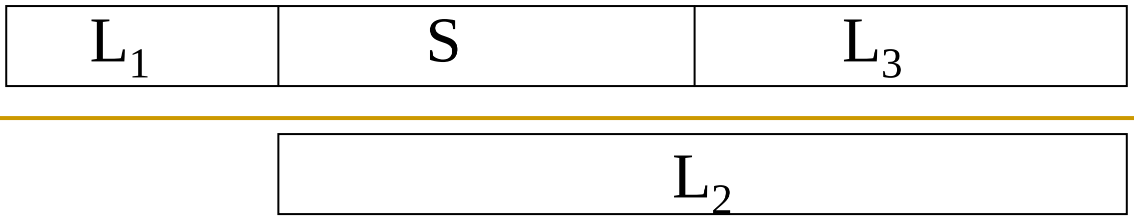
Sublist

sublist([b,c,d],[a,b,c,d,e]) is true, but
sublist([b,d],[a,b,c,d,e]) is false.

```
sublist(S,L) :-
        conc(L1,L2,L), conc(S,L3,L2).
```

List L can be decomposed into two parts L_1 and L_2.

L_2 is composed of S and L_3.



List (5)

Permutation

```
permutation([],[]).
permutation([X|L],P):-
    permutation(L,L1),
    insert(X,L1,P).
```

The ADT Stack

- empty_stack([]).
- stack(Top,Stack,[Top|Stack]).
  - push:
    stack(Element,stack before pushing,stack after pushing).
  - pop:
    stack(Element,stack after poping, stack before poping).
- member_stack(Element,Stack):- member(Element,Stack).
- add_list_to_stack(List,Stack,Result):- append(List,Stack,Result).

The ADT Queue

- `empty_queue([]).`
-
```
enqueue(E,[],[E]).
enqueue(E,[H|T],[H|Tnew]):- enqueue(E,T,Tnew).
```
- `dequeue(E,[E|T],T).`
-
```
member_queue(Element,Queue):-
        member(Element,Queue).
```
-
```
add_list_to_queue(List,Queue,Newqueue):-
        append(Queue,List,NewQueue).
```

The ADT Priority Queue

```
insert_pq(State,[],[State]).
insert_pq(State,[H|Tail],[State,H|Tail]):-
        precedes(State,H).
insert_pq(State,[H|T],[H|Tnew]):-
        insert_pq(State,T,Tnew).
insert_list_pq([],L,L).
insert_list_pq([State|Tail],L,New_L):-
        insert_pq(State,L,L2),
        insert_list_pq(Tail,L2,New_L).
```

Negation: not

Example. Mary likes all animals but snakes.

```
likes(mary, X) :- snake(X),!,fail.
likes(mary, X) :- animal(X).
```

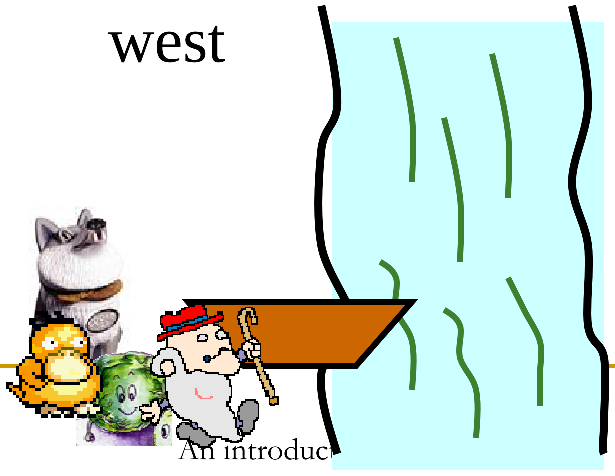
```
not(P):-
        P,!,fail
        ;
        true.
```

```
likes(mary,X) :- animals(X), not(snakes(X)).
```

A production system example in PROLOG:

The farmer, wolf, goat, and cabbage problem

Problem. A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river’s edge, but only the farmer can row. The boat also can carry only two things (including the rower) at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.



Farmer.pl

The state of this problem

state(F,W,G,C)

F: w,e

W:w,e

G:w,e

C:w,e

State transition

```
unsafe(state(F,W,G,C)):- W=G,not(F=G).  % closed world assumption
unsafe(state(F,W,G,C)):- G=C,not(F=C).
opp(e,w).
opp(w,e).
move(state(X,X,G,C),state(Y,Y,G,C)) :- opp(X,Y), not(unsafe(state(Y,Y,G,C))).
move(state(X,W,X,C),state(Y,W,Y,C)) :- opp(X,Y), not(unsafe(state(Y,W,Y,C))).
move(state(X,W,G,X),state(Y,W,G,Y)) :- opp(X,Y), not(unsafe(state(Y,W,Y,C))).
move(state(X,W,G,C),state(Y,W,G,C)) :- opp(X,Y), not(unsafe(state(w,W,G,C))).

reverse_print_stack([]).
reverse_print_stack([X|Y]) :- reverse_print_stack(Y),write(X),nl.

path(Goal, Goal, S) :- write('Solution path is'),nl,reverse_print_stack(S).

path(State,Goal,S) :-
                move(State,Next_state),
                not(member(Next_state,S)),
                NS = [ Next_state | S],
                path(Next_state,Goal,NS),!.
go(Start,Goal) :-
                S = [Start],
                path(Start,Goal,S).
```

Operator (1)

Basic arithmetic operations

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
//	Integer division
mod	modulo

Operator (2)

Comparison operators

X>Y	X is greater than Y
X<Y	X is less than Y
X>=Y	X is greater than or equal to Y
X=<Y	X is less than or equal to Y
X=:=Y	The **values** of X and Y are equal
X=\=Y	The **values** of X and Y are not equal

- X=Y is true if X and Y match.
- X is E is true if X matches the value of the arithmetic expression E.
- E1=:= E2 is true if the values of the arithmetic expressions E1 and E2 are equal.
  - Complementary relation: E1=\=E2
- T1==T2 is true if terms T1 and T2 have exactly the same structure and all corresponding components are the same. In particular, the names of the variables also have to be the same.
  - Complementary relation: T1\==T2.
  - Example

```
f(a,b)==f(a,b).
f(a,b)==f(a,X).
```

Operator (3)

- X=Y

  Operator = is the matching operator (like unification).

  ```
  ?-X=1+2.
  X=1+2
  ?-1+X=Y+2.
  X=2
  Y=1
  ```
- X is Y

  Operator ‘is’ forces evaluation of Y.

  ```
  ?-X is 1+2.
  X=3
  ```

Cut ! (1)

Control backtracking.

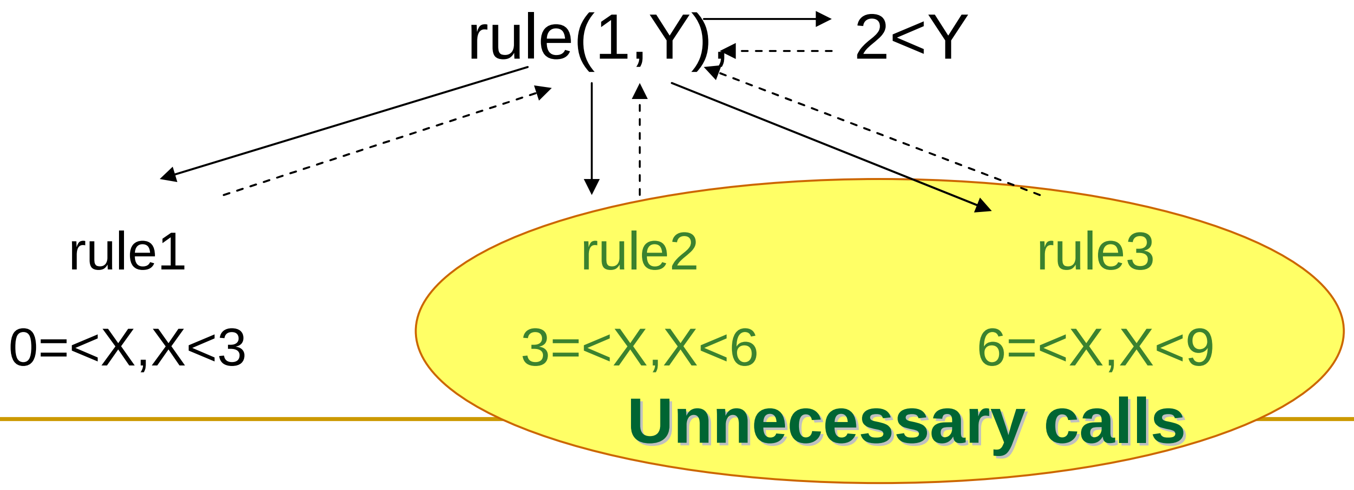
Example

Rule 1: if 0≤ X < 3, Y=0. rule(X,0):- 0=<X,X<3.

Rule 2: if 3≤X < 6, Y=8. rule(X,8):- 3=<X,X<6.

Rule 3: if 6≤ X < 9, Y=9. rule(X,9):-6=<X,X<9.

?- rule(1,Y), 2<Y.



H:- B_1, B_2, …, B_m,!,…,B_n.

When the cut is executed, the current solution of B_1, B_2, … B_m are frozen and all possible remaining alternatives are discarded. Any attempt to match the alterative clause about H is precluded.

```
rule(X,0):- 0=<X,X<3,!.
rule(X,8):- 3=<X,X<6,!.
rule(X,9):-6=<X,X<9,!.
```

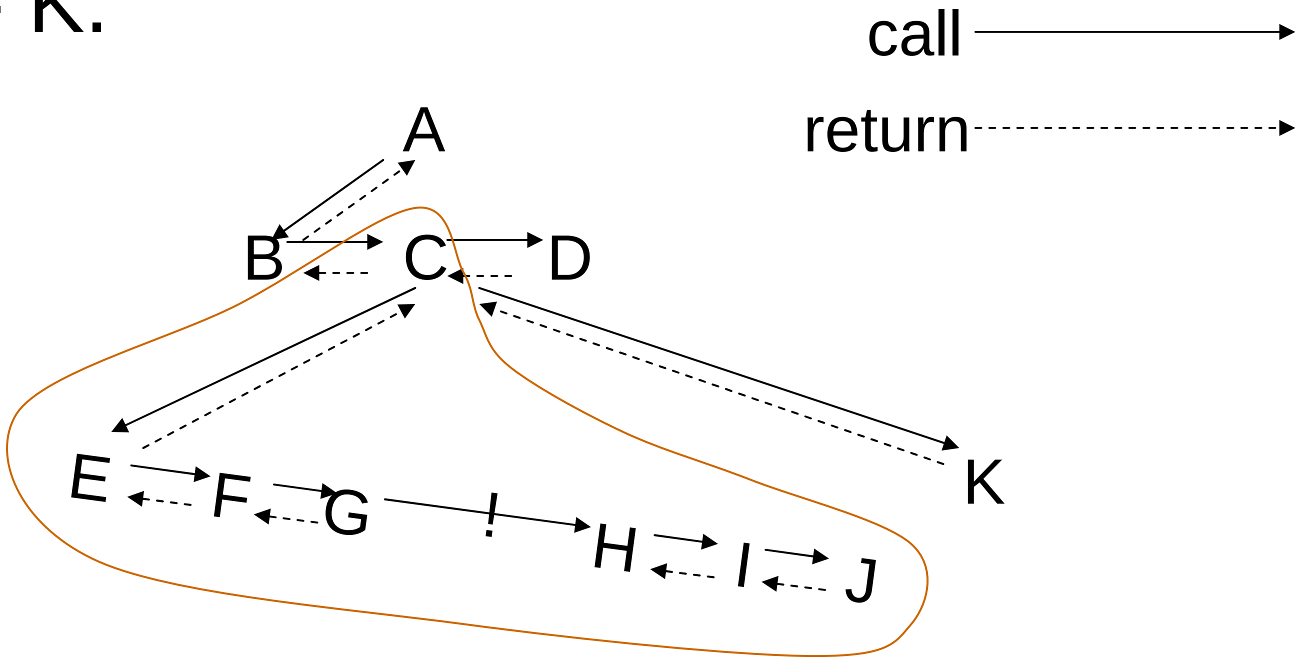
case.pl

Any remaining alternatives between C and the cut are discarded.

A:- B, C, D.

C:- E, F, G, !, H, I, J.

C:- K.



Using rules like case statements

```
action(1):-!,write(‘action 1’),nl.
action(2):-!,write(‘action 2’),nl.
action(3):-!,write(‘action 3’),nl.
action(_):- write(‘default action’),nl.
```

Cut ! (2)

- With cut we can often improve the efficiency of the program by explicitly telling PROLOG not to try other alternatives.
- Using cut we can specify mutually exclusive rules.

  If Condition P then conclusion Q; otherwise conclusion R.

- **A tail-recursion procedure** only has one recursive call, and this call appears as the last goal of the last clause in the procedure.
- Typical tail-recursive procedure

```
p(…) :- ….
p(…):- ….
…
p(…):-…,!,          % this cut ensures no backtracking.
        p(…). % tail recursive call
```

We know that these subgoals are deterministic.

- In the case of such tail-recursive procedures, a PROLOG system will typically realize tail-recursion as **iteration** to save memory.
    - This is called *tail recursion optimization*, or *last call optimization*.

```
p:- a,b,!.
p:- c.
```

p is true if a and b are both true or c is true.

```
p:- a,!,b.
p:- c.
```

p is true if a and b are both true or not(a) and c is true.

```
p:- c.
p:- a,!,b.
```

p is true if c is true or a and b are both true.

Designing alternative search strategies

- PROLOG uses depth-first search with backtracking.
- Alternative search strategies can be implemented in PROLOG.
  - Depth-first search using the close list
  - Breadth-first search in PROLOG
  - Best-first search in PROLOG

    (See Luger02.)

Input/Output

- Open/Close input/output stream.
  - **see**(Filename) succeeds with opening an input stream.
    - see(user) % user: standard input/output stream
  - **tell**(Filename) succeeds with opening an output stream.
  - **seen** succeeds with closing the current input stream.
  - **told** succeeds with closing the current output stream.
- read and write
  - read(X)
  - write(X)
  - tab(N): n spaces to output.
  - nl: start of a new line at output

Meta-logical predicates (1)

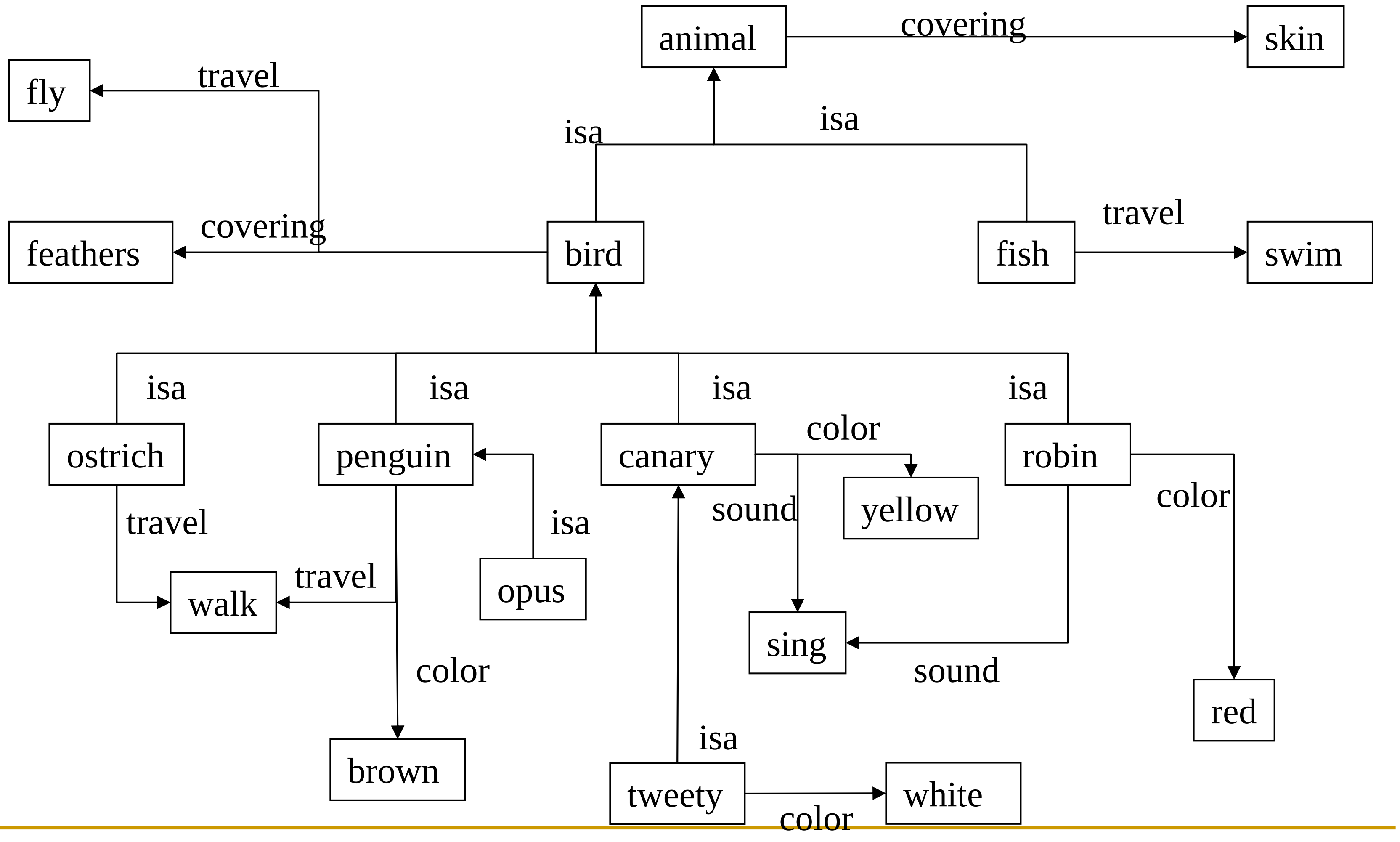
- Meta-predicates are predicates that are designed to match, query, manipulate other predicates.
- Some meta-predicates
  - **assert**(C) adds the clause C to the current set of clauses.
  - **retract**(C) removes the C from the current set of clauses.
  - **var**(X) succeeds only when X is an unbound variable.
  - **nonvar**(X) succeeds only when X is bound to a nonvariable term.
  - **=..** creates a list from a predicate term.
    - Example
      foo(a,b,c)=..X.   X has value [foo, a, b, c].
      X=..[foo, a, b, c]. X has value foo(a,b,c).

Meta-logical predicates (2)

- **functor**(A, B, C) succeeds with A, a term, whose principal functor has name B and the arity C.
- **clause**(A,B) unifies B with the body of a clause whose head unifies with A.
  - Example
    p(X) :- q(X).
    clause(p(a),Y). Y has value q(a).
- any_predicate(…,X,…) :- X.
- **call**(X), where X is a clause, also succeeds with the execution of predicate X.

Knowledge Representation in PROLOG

Semantic nets in PROLOG



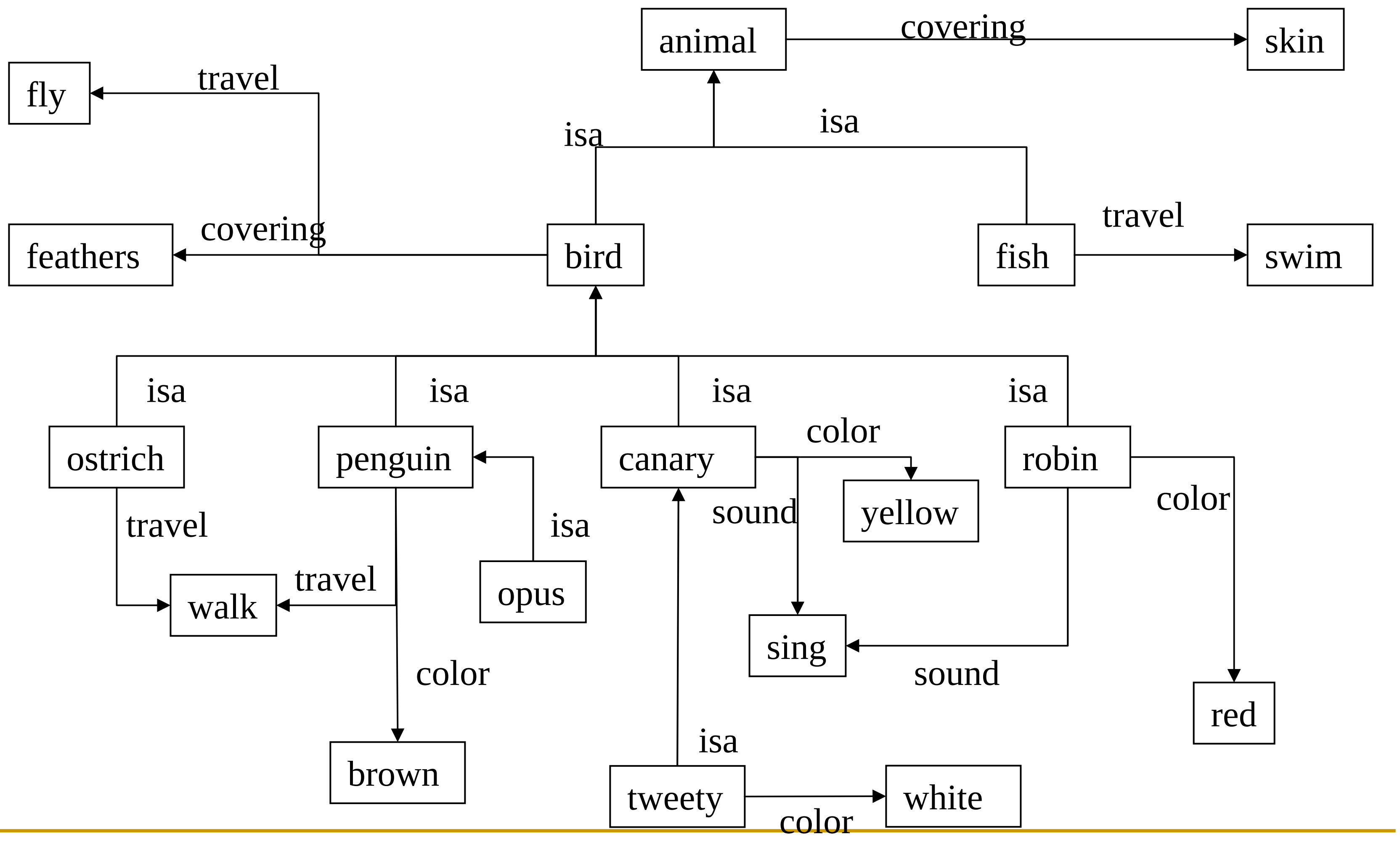
```
isa(canary,bird).        hasprop(animal,cover,skin).
isa(ostrich, bird).      hasprop(fish,travel,swim).
isa(bird,animal).        hasprop(bird,travel,fly).
isa(opus,penguin).       hasprop(bird,cover,feathers).
isa(robin,bird).         hasprop(ostrich,sound,sing).
isa(penguin,bird).       hasprop(robin,color,red).
isa(fish,animal).        hasprop(penguin,color,brown).
isa(tweety,canary).      hasprop(penguin,travel,walk).
                         hasprop(canary,color,yellow).
                         hasprop(canary,sound,sing).
                         hasprop(tweety,color,white).
```

%Implement inheritance

```
hasproperty(Object,Property,Value) :-
        hasprop(Object,Property,Value).
hasproperty(Object,Property,Value) :-
        isa(Object,Parent),
        hasproperty(Parent,Property,Value).
```

Semantic nets in PROLOG

Using meta-logical predicates



Semantic nets in Prolog (cont.)

Using meta-logical predicates.

semantics_net.pl

```
isa(canary,bird).
isa(ostrich, bird).
isa(bird,animal).
isa(opus,penguin).
isa(robin,bird).
isa(penguin,bird).
isa(fish,animal).
isa(tweety,canary).
```

```
cover(animal, skin).
travel(fish,swim).
travel(bird,fly).
cover(bird, feathers).
sound(ostrich, sing).
color(robin, red).
color(penguin, brown).
travel(penguin, walk).
color(canary, yellow).
sound(canary, sing).
color(tweety, white).
```

```
% Implement inheritance
property(Property) :-
        Property,!.
property(Property) :-
        Property=..[Functor,Object,Value],
        isa(Object,Parent),
        Parent_property=..[Functor, Parent, Value],
        property(Parent_property).
```

Frames in PROLOG

- Some knowledge about birds put into frames

FRAME: bird
A_kind_of: animal
Moving_method:fly
Active_at:daylight

FRAME: albatross
A_kind_of: bird
Color:black_and_white
Size:115

FRAME: kiwi
A_kind_of: bird
Moving_method: walk
Active_at:night
Color: brown
Size:40

FRAME: Albert

Instance_of: albatross

Size: 120

FRAME: Ross

Instance_of: albatross

Size: 40

Frame.pl

```
% Frame bird: the prototypical bird

bird( a_kind_of, animal).
bird( moving_method, fly).
bird( active_at, daylight).

% Frame albatross: albatross is a typical bird with some
% extra facts: it is black and white, and it is 115 cm long

albatross( a_kind_of, bird).
albatross( color, black_and_white).
albatross( size, 115).

% Frame kiwi: kiwi is a rather untypical bird in that it
% walks instead of flies, and it is active at night

kiwi( a_kind_of, bird).
kiwi( moving_method, walk).
kiwi( active_at, night).
kiwi( size, 40).
kiwi( color, brown).

% Frame albert: an instance of a big albatross

albert( instance_of, albatross).
albert( size, 120).

% Frame ross: an instance of a baby albatross

ross( instance_of, albatross).
ross( size, 40).

% Frame animal: slot relative_size obtains its value by
% executing procedure relative_size

animal( relative_size, execute( relative_size( Object, Value), Object, Value) ).
```

Direct retrieval or retrieval by inheritance

```
value(Frame,Slot,Value) :-
        Query=..[Frame, Slot, Value],
        call(Query), !.
value(Frame,Slot,Value):-
        parent(Frame,ParentFrame),
        value(Parentframe,Slot,Value).

parent(Frame,ParentFrame):-
        (Query=..[Frame, a_kind_of, ParentFrame];
         Query=..[Frame, instance_of,ParentFrame]),
        call(Query).
```

A procedure for computing value is in a slot.

```
animal( relative_size, execute( relative_size( Object, Value), Object, Value) ).

relative_size(Object,RelativeSize):-
            value(Object,size,ObjSize),
            value(Object,instance_of,ObjClass),
            value(ObjClass,size,ClassSize),
            RelativeSize is ObjSize/ClassSize*100.

value(Frame,Slot,Value):-
            value(Frame,Frame,Slot,Value).
value(Frame,SuperFrame,Slot,Value):-
            Query=..[SuperFrame, Slot, Information],
            call(Query),
            process(Information,Frame,Value),!.
value(Frame,SuperFrame,Slot,Value) :-
            parent(SuperFrame,ParentSuperFrame),
            value(Frame,ParentSuperFrame,Slot,Value).
%process(Information,Frame,Value)
process(execute(Goal,Frame,Value),Frame,Value):-
            call(Goal).
process(Value,_,Value).
```

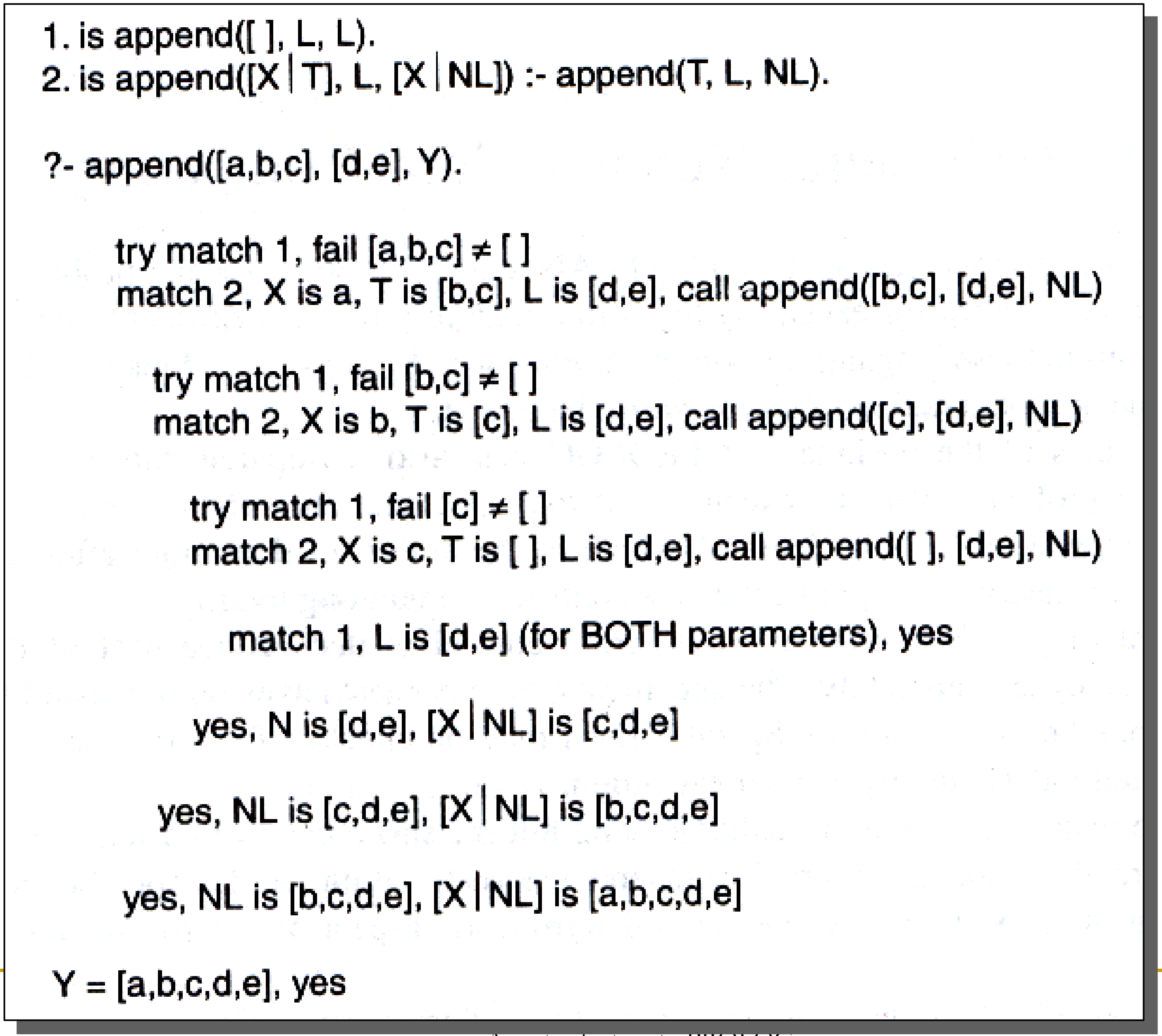
Toward nonprocedural computing

Declarative nature of PROLOG

- **Separation of specification and control.**

Example.

```
  append([],L,L).
  append([X|T],L,[X|NL]):-append(T,L,NL).
?-append([a,b,c],[d,e],Y).
Y=[a,b,c,d,e]
```



```
1. is append([ ], L, L).
2. is append([X|T], L, [X|NL]) :- append(T, L, NL).

?- append([a,b,c], [d,e], Y).

    try match 1, fail [a,b,c] ≠ [ ]
    match 2, X is a, T is [b,c], L is [d,e], call append([b,c], [d,e], NL)

      try match 1, fail [b,c] ≠ [ ]
      match 2, X is b, T is [c], L is [d,e], call append([c], [d,e], NL)

        try match 1, fail [c] ≠ [ ]
        match 2, X is c, T is [ ], L is [d,e], call append([ ], [d,e], NL)

          match 1, L is [d,e] (for BOTH parameters), yes

        yes, N is [d,e], [X|NL] is [c,d,e]

      yes, NL is [c,d,e], [X|NL] is [b,c,d,e]

    yes, NL is [b,c,d,e], [X|NL] is [a,b,c,d,e]

 Y = [a,b,c,d,e], yes
```

The parameters can be either “input” or “output”

```
?-append([a,b],[c],[a,b,c]).
yes
?-append([a],[c],[a,b,c]).
no
?- append(X,[b,c],[a,b,c]).
X=[a]
```

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

Simple Parser and Generator of Sentences

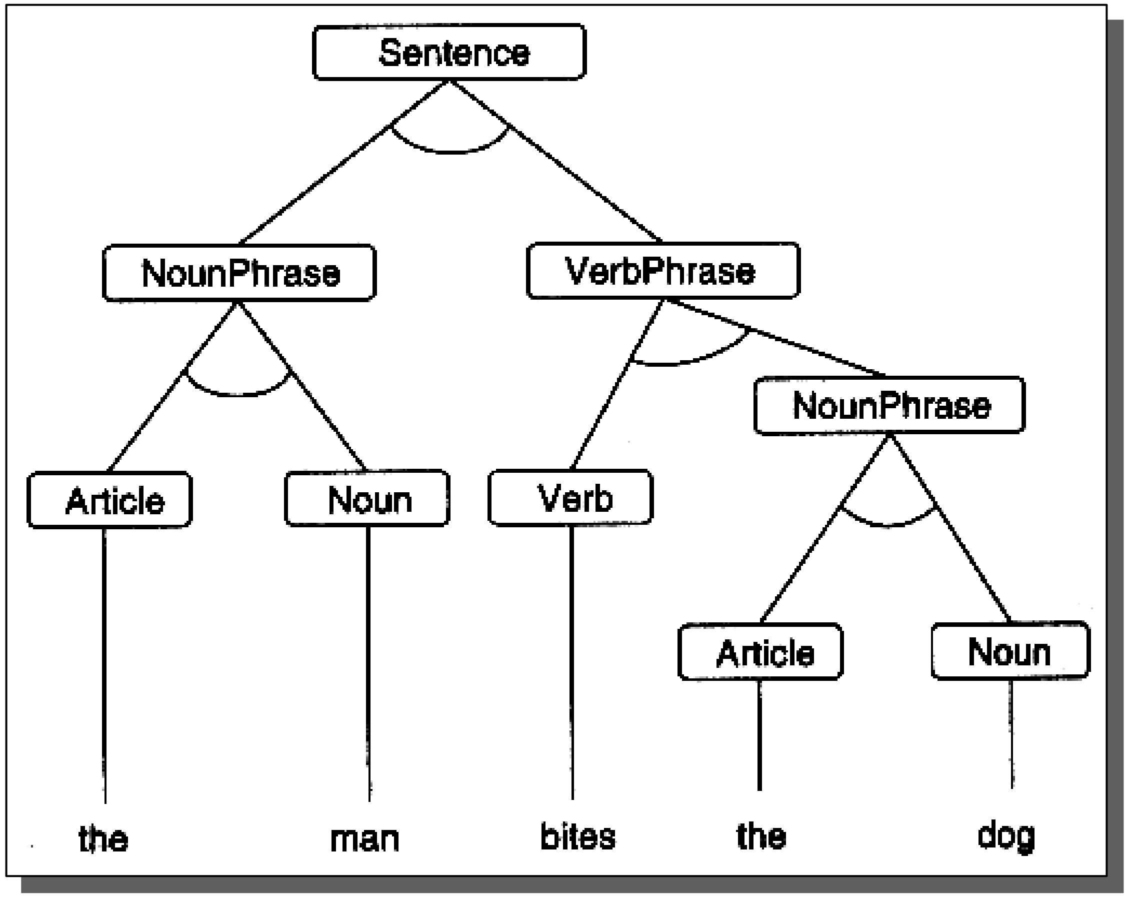
- A subset of English grammar rules
  - <sentence> ::= <nounphase> <verbphase>

    ```
    sentence(Start,End) :- nounphase(Start,Rest),
        verbphase(Rest,End).
    ```
  - <nounphase>::=<noun>| <article><noun>

    ```
    nounphase([Noun|End],End):-noun(Noun).
    nounphase([Article,Noun|End],End):-article(Article),noun(Noun).
    ```
  - <verbphase> ::= <verb>|<verb><nounphase>

    ```
    verbphase([Verb|End],End):- verb(Verb).
    verbphase([Verb|Rest],End):- verb(Verb),nounphase(Rest,End).
    utterance(X) :- sentence(X,[]).
    ```

```
article(a).

article(the).

noun(man).

noun(dog).

verb(likes).

verb(bites).
```



parse tree

```
utterance([the, man, bites, the, dog]).

yes
```