Prolog

-Recursive Rules

-List Data structure

Prolog Features

- Prolog has the power of designing knowledge based systems at very ease.
- Prolog can be interfaced with almost all high level languages. E.g. with Python, C Java etc.; https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages#Prolog
- Prolog is very much suited for parsing data or parsing other languages (natural and computer).
- Prolog is also very popular as a Game Description language with very powerful libraries.

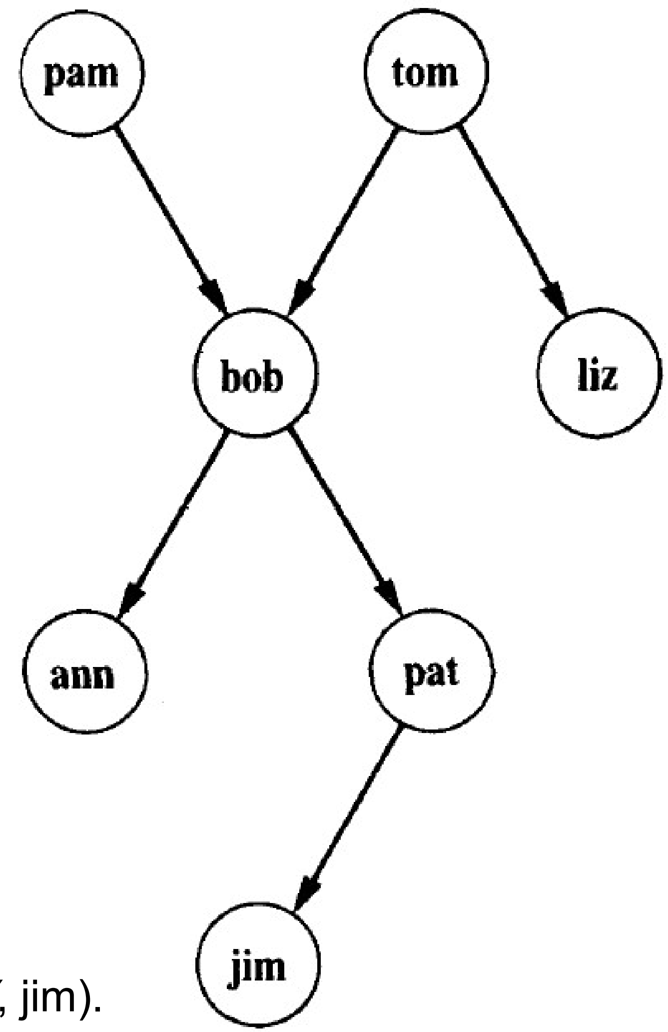
For installation of Prolog editor in Eclipse for SWI prolog, see instructions at:
https://sewiki.iai.uni-bonn.de/research/pdt/docs/download
https://sewiki.iai.uni-bonn.de/research/pdt/docs/start

SWI tutorials

- CLP(FD) Constraint Logic Programming over Finite Domains (http://www.pathwayslms.com/swipltuts/clpfd/clpfd.html)
- Using Definite Clause Grammar in Prolog (http://www.pathwayslms.com/swipltuts/dcg/)
- https://www.swi-prolog.org/features.html

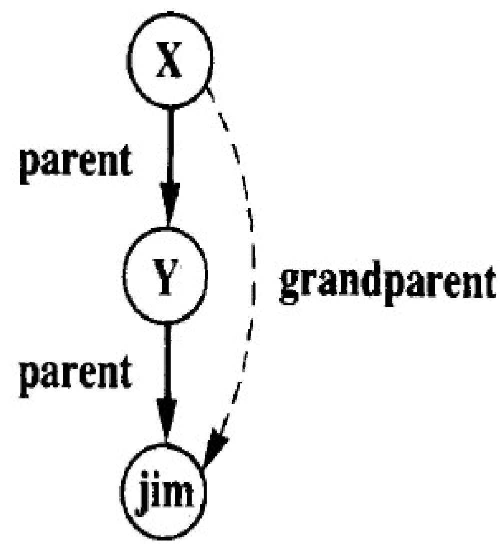
Example 2: A Family Tree

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
female(pam).
male(tom).
male(bob).
female(liz).
female(pat).
female(ann).
male(jim).

?- parent(Y, jim), parent(X,Y).
?- parent(tom, X), parent(X,Y).
?- parent(pam, X), parent(X,Y), parent(Y, jim).
```



Recursive Rule

Figure 1.2 The **grandparent** relation expressed as a composition of two **parent** relations.



Recursive Rules

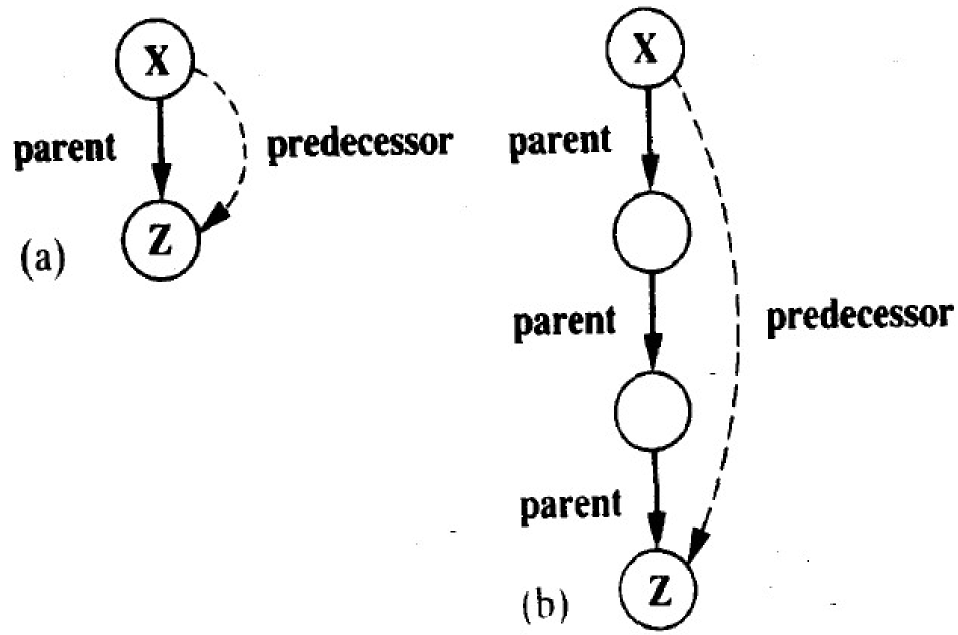
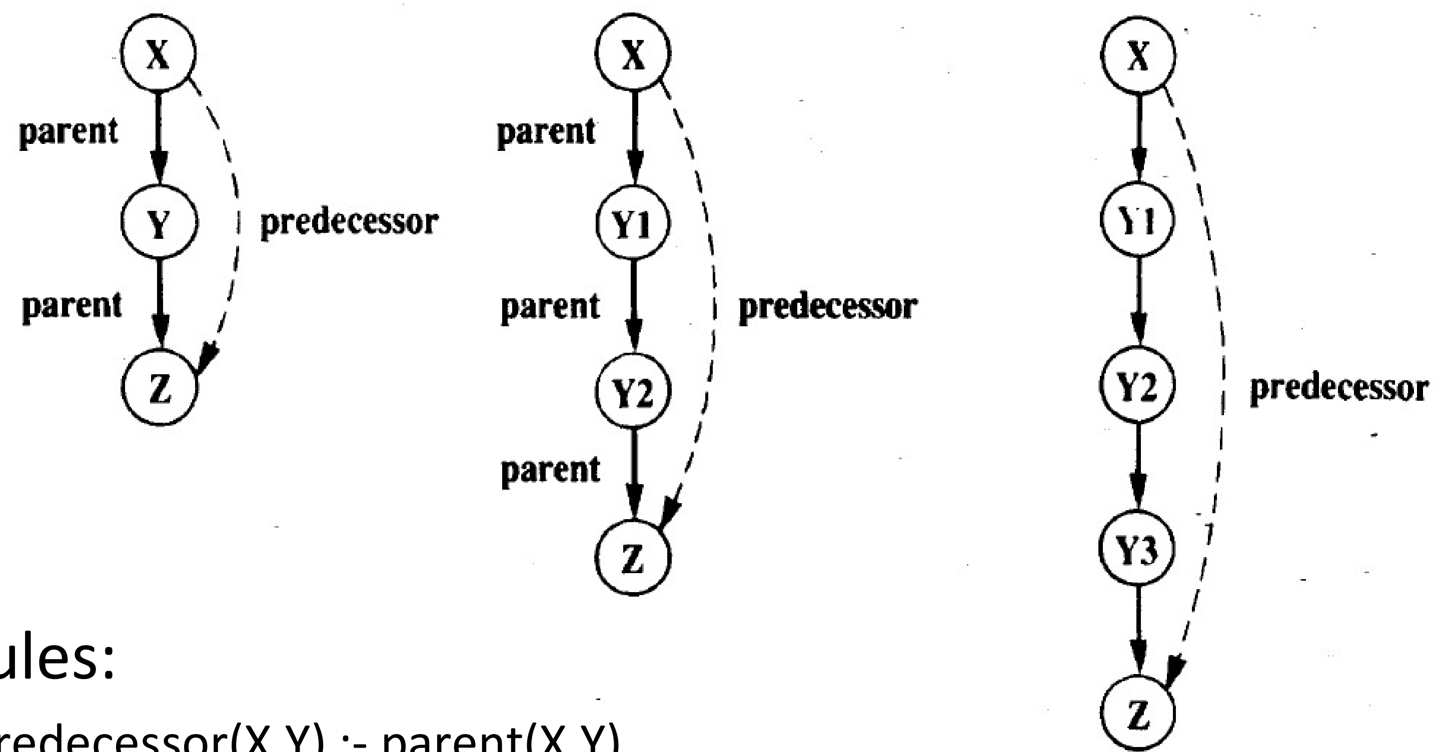


Figure 1.5 Examples of the **predecessor** relation: (a) **X** is a *direct* predecessor of **Z**; (b) **X** is an indirect predecessor of **Z**.

Recursive Rules



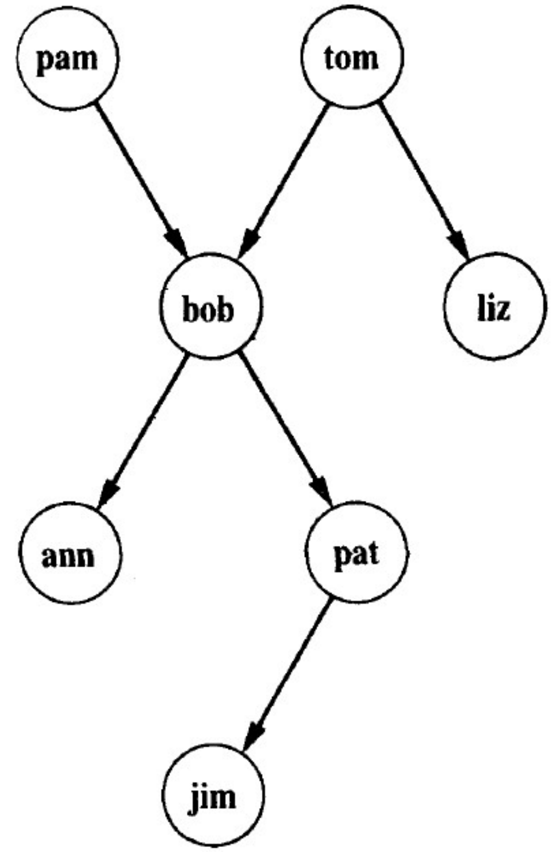
- Rules:

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

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

Recursive Definition

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

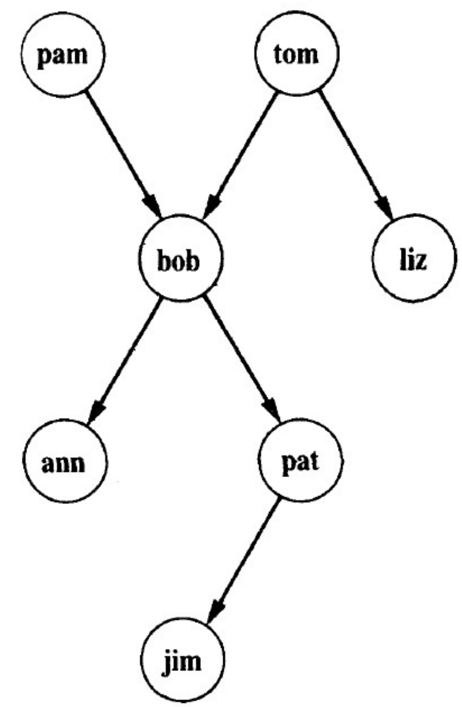


- Rules:

```
predecessor(X,Y) :- parent(X,Y).
predecessor(X,Y) :- parent(X, Z), predecessor (Z, Y).

?-predecessor(pam, Y), write(Y).
X=bob;
ann;
pat;
jim.
no.
```

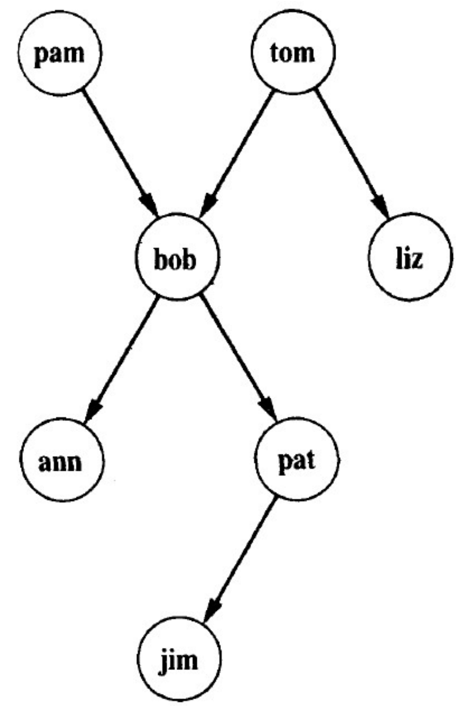
```
pred (X, Y) :- parent (X, Y).
pred (X, Y) :- parent (X, Z), pred (Z, Y).

?- pred ( pam, jim).
```

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



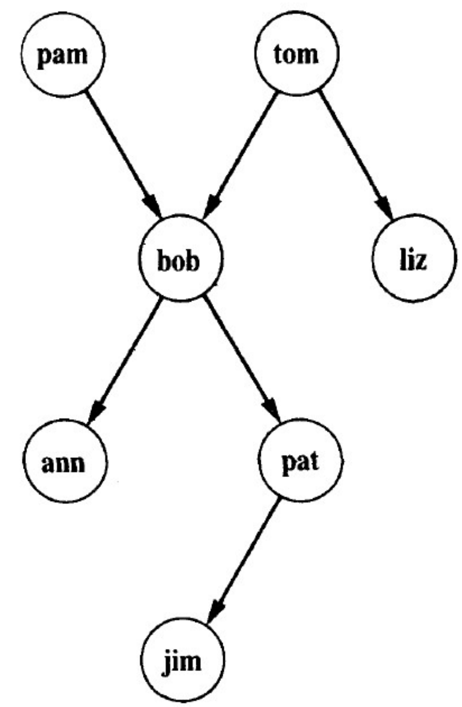
```
pred(X, Y) :- parent(X, Y).
pred(X, Y) :- parent(X, Z), pred(Z, Y).

?- pred(pam, jim).
        X = pam
        Y = jim
parent(pam, Z), pred(Z, jim).
```

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).



```
pred(X, Y) :- parent(X, Y).
pred(X, Y) :- parent(X, Z), pred(Z, Y).

?- pred(pam, jim).
        X = pam
        Y = jim
parent(pam, Z), pred(Z, jim).
        Z = bob
parent(pam, bob), pred(bob, jim).
```

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```



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

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

?- pred(pam, jim).

X = pam
Y = jim

parent(pam, Z), pred(Z, jim).

Z = bob

parent(pam, bob), pred(bob, jim).

true

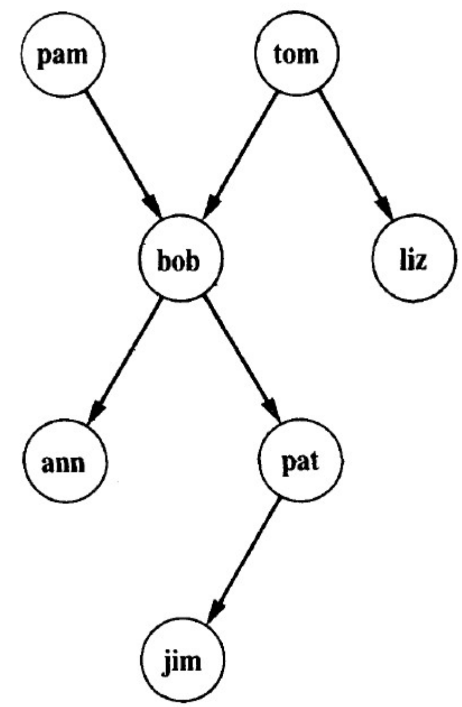
X = bob
Y = jim

parent(bob, Z1), pred(Z1, jim).

Z1 = ann

Z1 = pat

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).



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

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

?- pred (pam, jim).

X = pam
Y = jim

parent (pam, Z), pred (Z, jim).

Z = bob

parent (pam, bob), pred (bob, jim).

true

X = bob
Y = jim

parent (bob, Z1), pred (Z1, jim).

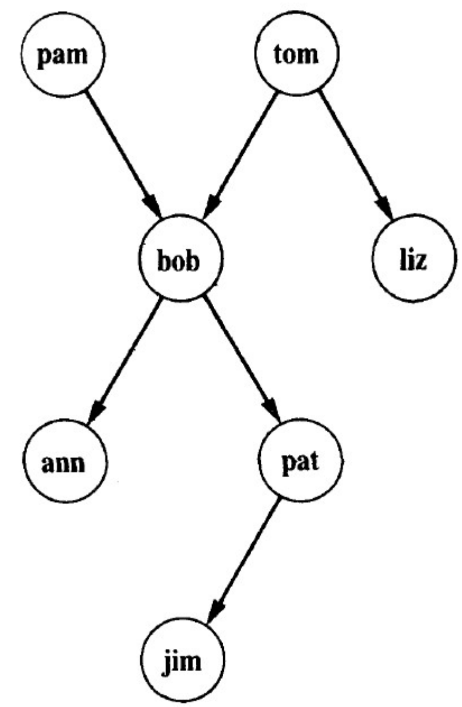
Z1 = ann

Z1 = pat

true parent (bob, ann),
pred (ann, jim)
false

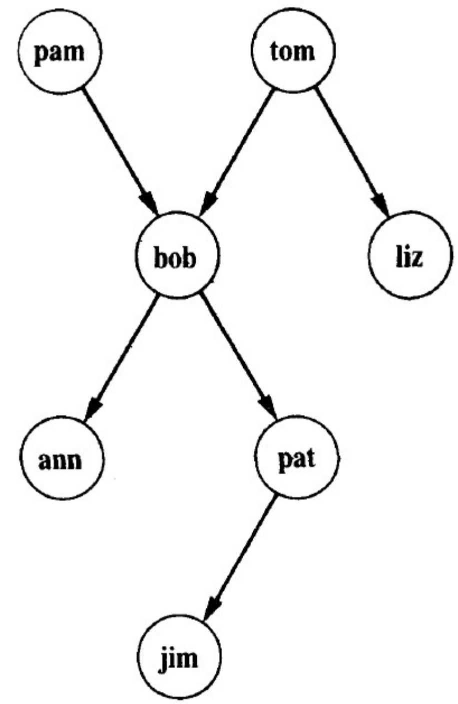
parent (bob, pat), pred (pat, jim).
true

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).



```
pred(X, Y) :- parent(X, Y).
pred(X, Y) :- parent(X, Z), pred(Z, Y).

?- pred(pam, jim).
        X = pam
        Y = jim
parent(pam, Z), pred(Z, jim).
        Z = bob
parent(pam, bob), pred(bob, jim).
 true
        X = bob
        Y = jim
parent(bob, Z1), pred(Z1, jim).
   Z1 = ann                          Z1 = pat
true parent(bob, ann),               parent(bob, pat), pred(pat, jim).
  pred(ann, jim)                        true
    false                                  parent(pat, jim)
                                             true
```

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).



Lists

- A collection of ordered data.
- Has *zero* or more elements enclosed by square brackets ('[ ]') and separated by commas (',').

`[a]` ← a list with one element

`[]` ← an empty list

```
 1      2      3
             1   2
[34,tom,[2,3]]
```

`[34,tom,[2,3]]` ← a list with 3 elements where the 3rd element is a list of 2 elements.

- Like any object, a list can be unified with a variable

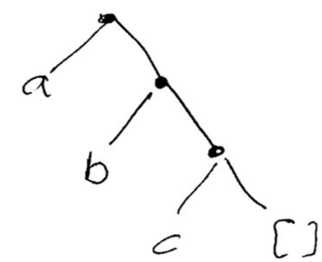
```
|?- [Any, list, 'of elements'] = X.
X = [Any, list, 'of elements']?
yes
```

Recursive nature of List

(1) A list is a data structure that is either empty or consists of 2 parts →

- first element (called head)
- Remaining list

e.g. $L = [a, b, c]$



can be represented as →

$L = [a \mid X]$

or $\cdot(a, X)$

special functor

OR $\cdot(a, \cdot(b, \cdot(c, [\,])))$.

OR

$[a, b, c] = [a \mid [b, c]] = [a, b \mid [c]]$

$= [a, b, c \mid [\,]]$

List Unification

- Two lists unify if they are the same length and all their elements unify.

```
|?-[a,B,c,D]=[A,b,C,d].
A = a,
B = b,
C = c,
D = d ?
yes
```

```
|?-[(a+X),(Y+b)]=[(W+c),(d+b)].
W = a,
X = c,
Y = d?
yes
```

```
|?- [[X,a]]=[b,Y].
no
```

Length 1 Length 2

```
|?-[[a],[B,c],[]]=[X,[b,c],Y].
B = b,
X = [a],
Y = [] ?
yes
```

Definition of a List

- Lists are *recursively defined* structures.

"An empty list, [], is a list.

A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty."

- This recursiveness is made explicit by the bar notation
  - `[Head|Rem_list]` ('|' = bottom left PC keyboard character)
- Head must unify with a single term.
- Rem_list unifies with a list of any length, including an empty list, [ ].
  - the bar notation turns everything after the Head into a list and unifies it with Tail.

Head and rest of list List (in terms of Head and rest of the list)

```
?-[a,b,c,d]=[X|Y].        ?-[a,b,c,d]=[X|[Y|Z]].
X = a,                     X = a,
Y = [b,c,d]?               Y = b,
yes                        Z = [c,d];
                           yes

?-[a] = [H|T].                  ?-[a,b,c]=[W|[X|[Y|Z]]].
H = a,                     W = a,
T = [ ];                        X = b,
yes                             Y = c,
                                  Z = [ ]? yes

?-[ ] = [H|T].           ?-[a | [ b | [c|[ ] ] ] ]= List.
no                         List = [a,b,c]?
                           yes
```

Identifying a list

- lists: `[a,[],green(bob)]`
- lists are *recursively defined* structures:

"An empty list, [ ], is a list.

A structure of the form [X, …] is a list if X is a term and […] is a list, possibly empty."

- This can be tested using the Head and Rem_list notation, [H|T], in a recursive rule.

```
is_a_list([]).                ← A term is a list if it is an empty list.
is_a_list([_|Rem_list]):-     ← A term is a list if it has two
    is_a_list(Rem_list).        elements and the second is a list.
```

Base and Recursive Cases

- A recursive definition, whether in prolog or some other language (including English!) needs two things.

- A definition of when the recursion *terminates*.
  - Without this the recursion would never stop!
  - This is called the *base case:* `is_a_list([]).`
  - *Almost always comes before recursive clause*

- A definition of how we can define the problem in terms of a similar, smaller problem.
  - This is called the *recursive case*:
```
is_a_list([_|T]):-
    is_a_list(T).
```

- There might be more than one base or recursive case.

Focussed Recursion

- To ensure that the predicate terminates, the recursive case must move the problem closer to a solution.
  - If it doesn't it will loop infinitely.

- With list processing this means stripping away the Head of a list and recursing on the Tail.

Head is replaced with an underscore as we don't want to use it.

```
        is_a_list([_|T]):-
    is_a_list(T).
```

- The same focussing has to occur when recursing to find a property or fact.

```
is_older(Ancestor,Person):-
      is_older(Someone,Person),
      is_older(Ancestor,Someone).
```

Doesn't focus

Printing items in a List

```
print([ ]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
```

Printing items in a List

```
print([]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
        X=1
        Y=[2,3]
print([1|2,3])
```

Printing items in a List

```
print([ ]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
        X=1
        Y=[2,3]
print([1|2,3])
  write(1), print([2,3]).
```

Printing items in a List

```
print([ ]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
        X = 1
        Y = [2,3]
print([1|2,3])
  write(1), print([2,3]).
                X = 2
                Y = [3]
        print([2|3])
```

Printing items in a List

```
print([]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
        X = 1
        Y = [2,3]
print([1|2,3])
  write(1), print([2,3]).
            X = 2
            Y = [3]
        print([2|3])
   write(2), print([3])
```

Printing items in a List

```
print([ ]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
        X = 1
        Y = [2,3]
print([1|2,3])
  write(1), print([2,3]).
              X = 2
              Y = [3]
            print([2|3])
      write(2), print([3])
                  X = 3
                  Y = [ ]
             print([3|[ ]])
```

Printing items in a List

```
print([ ]).
print([X|Y]) :- write(X), print(Y).

?- print([1,2,3]).
        X = 1
        Y = [2,3]
print([1|2,3])
  write(1), print([2,3]).
              X = 2
              Y = [3]
        print([2|3])
     write(2), print([3])
                 X = 3
                 Y = [ ]
          print([3|[ ]])
        write(3), print([ ])
                   true.
Output: 1 2 3
```

List Processing Predicates: Member/2

- `Member/2` is possibly the most used user-defined predicate (i.e. you have to define it every time you want to use it!)
- It checks to see if a term is an element of a list.
  - it returns `yes` if it is
  - and `fails` if it isn't.

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

```
member(H,[H|_]).

member(H,[_|T]):-
        member(H,T).
```

- It 1st checks if the Head of the list unifies with the first argument.
  - If yes then succeed.
  - If no then fail first clause.
- The 2nd clause ignores the head of the list (which we know doesn't match) and recurses on the Tail.

member/2

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

?-member(3,[1, 2, 3,4]).
```

List Processing Predicates: Member/2

|?- member(ringo,[john,paul,ringo,george]).

Fail(1): member(ringo,[john|_]).

(2): member(ringo,[_|paul,ringo,george]):-

Call: member(ringo,[paul,ringo,george]).

Fail(1): member(ringo,[paul|_]).

(2): member(ringo,[_|ringo,george]):-

Call: member(ringo,[ringo,george]).

Succeed(1): member(ringo,[ringo|_]]).

```
1) member(H , [ H | _ ]).
2) member(H , [ _ | T]) :- member(H , T).
```

Concatenation of 2 Lists

```
conc( [ ], L, L).
conc( [ X | L1], L2, [X | L3] ) :- conc( L1, L2,L3).
```

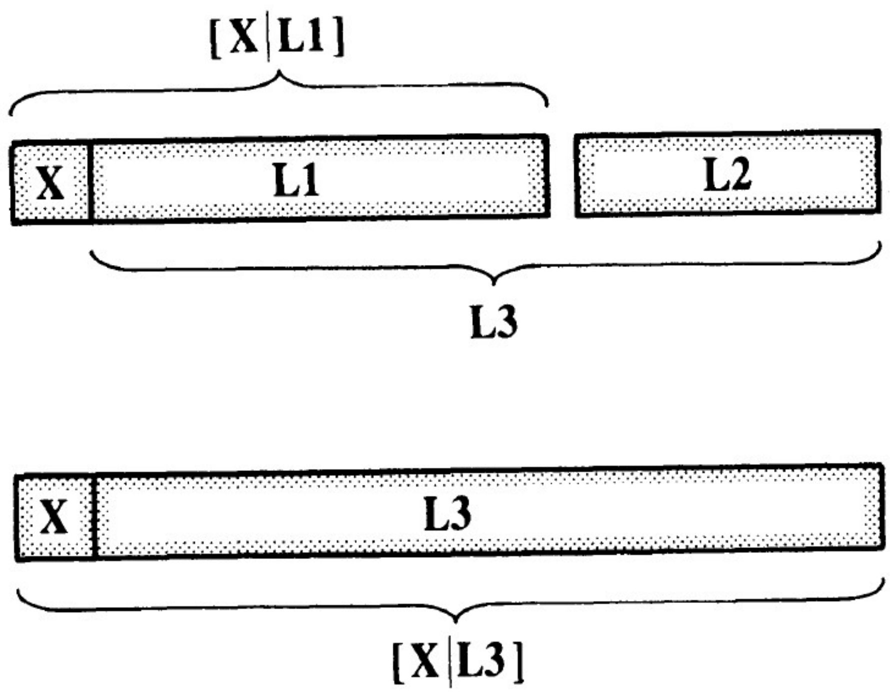


Figure 3.2 Concatenation of lists.

Concatenation of 2 Lists

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :-
                conc(L1, L2, L3).
?- conc([a,b], [1,2,3], L).
```

Concatenation of 2 Lists

```
conc([], L, L).

conc([X|L1], L2, [X|L3]) :-
        conc(L1, L2, L3).

?- conc([a, b], [1, 2, 3], L).

conc([a|b], [1, 2, 3], [a|L3])
```

Concatenation of 2 Lists

```
conc([], L, L).

conc([X|L1], L2, [X|L3]) :-
        conc(L1, L2, L3).

?- conc([a, b], [1, 2, 3], L).

conc([a|b], [1, 2, 3], [a|L3])

conc([b], [1, 2, 3], L3)
```

Concatenation of 2 Lists

```
conc([], L, L).

conc([X|L1], L2, [X|L3]) :-
        conc(L1, L2, L3).

?- conc([a, b], [1,2,3], L).

  conc([a|b], [1,2,3], [a|L3])

conc([b], [1,2,3], L3)

 conc([b|[]], [1,2,3], [b|L3'])
```

Concatenation of 2 Lists

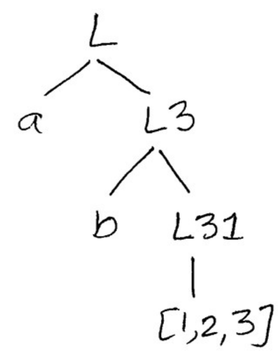
```
conc([], L, L).

conc([X|L1], L2, [X|L3]) :-
        conc(L1, L2, L3).

?- conc([a, b], [1, 2, 3], L).

conc([a|b], [1, 2, 3], [a|L3])

conc([b], [1, 2, 3], L3)

conc([b|[]], [1, 2, 3], [b|L31])

conc([], [1, 2, 3], L31)
```

Concatenation of 2 Lists

```
conc([], L, L).

conc([X|L1], L2, [X|L3]) :-
        conc(L1, L2, L3).

?- conc([a, b], [1,2,3], L).
   /
 conc([a|b], [1,2,3], [a|L3])
  /
conc([b], [1,2,3], L3)
  /
 conc([b|[]], [1,2,3], [b|L31])
 /
conc([], [1,2,3], L31)
  /     L31 = [1,2,3]
conc([], [1,2,3], [1,2,3]).
 /
true
```

Concatenation of 2 Lists

```
conc([], L, L).

conc([X|L1], L2, [X|L3]) :-
        conc(L1, L2, L3).

?- conc([a, b], [1,2,3], L).

conc([a|b], [1,2,3], [a|L3])

conc([b], [1,2,3], L3)

conc([b|[]], [1,2,3], [b|L31])

conc([], [1,2,3], L31)
        L31 = [1,2,3]
conc([], [1,2,3], [1,2,3]).

true
```



L= [a, b, 1, 2, 3]

Concatenating 2 Lists

```
conc( [ ], L, L).
conc( [ X | L1], L2, [X | L3] )
  :- conc( L1, L2,L3).
```

- Although the conc program looks rather simple it can be used flexibly in many other ways.
- For example, we can use conc in the inverse direction for decomposing a given list into two lists, as follows:

```
?-  conc( L1, L2, [a,b,c] ).

L1 = []
L2 = [a,b,c];

L1 = [a]
L2 = [b,c];

L1 = [a,b]
L2 = [c];

L1 = [a,b,c]
L2 = [];

no
```

Concatenating 2 Lists

```
conc( [ ], L, L).
conc( [ X | L1], L2, [X | L3] ) :- conc( L1, L2,L3).
```

- We can also use this program to look for a certain pattern in a list.
- example, we can find the months that precede and the months that follow a given month say may, as in the following goal:

```
?-  conc( Before, [may | After],
          [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Before = [jan,feb,mar,apr]
After = [jun,jul,aug,sep,oct,nov,dec].
```

Concatenating 2 Lists

```
conc( [ ], L, L).
conc( [ X | L1], L2, [X | L3] ) :- conc( L1, L2,L3).
```

- Further we can find the immediate predecessor and the immediate successor of may by asking:

```
?-  conc( _, [Month1,may,Month2 | _],
          [jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] ).

Month1 = apr
Month2 = jun
```

Concatenating 2 Lists

```
conc( [ ], L, L).
conc( [ X | L1], L2, [X | L3] ) :- conc( L1, L2,L3).
```

- we can, for example, delete from some list, L1, everything that follows three successive occurrences of z in L1 together with the three z's. For example:

```
?-  L1 = [a,b,z,z,c,z,z,z,d,e],
    conc( L2, [z,z,z | _], L1).

L1 = [a,b,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

List processing in Prolog

```
english_spanish("One", "Uno").
english_spanish("Two", "Dos").
english_spanish("Three", "Tres").
english_spanish("Four", "Cuatro").
english_spanish("Five", "Cinco").
english_spanish("Six", "Seis").
english_spanish("Seven", "Siete").
english_spanish("Eight", "Ocho").
english_spanish("Nine", "Nueve").
english_spanish("Ten", "Diez").
translate_td([], []).

translate_td([English|EnglishList], [Spanish|SpanishList]) :-
  english_spanish(English, Spanish), translate_td(EnglishList, SpanishList).

?- translate_td(["One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight",
"Nine", "Ten"], S).
?- translate_td(E, ["Uno", "Dos", "Tres", "Cuatro", "Cinco", "Seis", "Siete", "Ocho",
"Nueve", "Diez"])
```

```
%Traditional top-down iteration
%% translate_td(?EnglishList,
?SpanishList) is det
```

- One of the advantages of this is its bidirectionality.
- We can either put in a list of English numbers and get Spanish numbers, or vice versa.