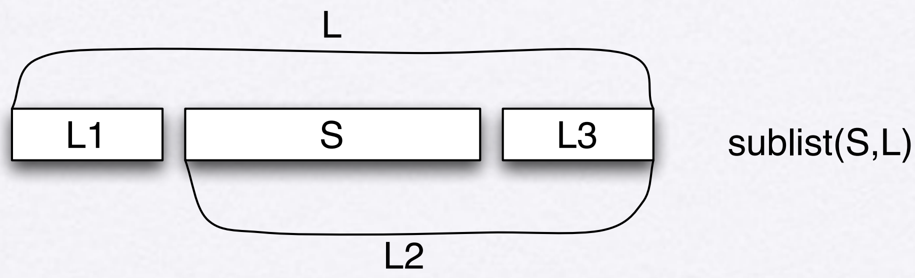
sublists, subsets

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




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

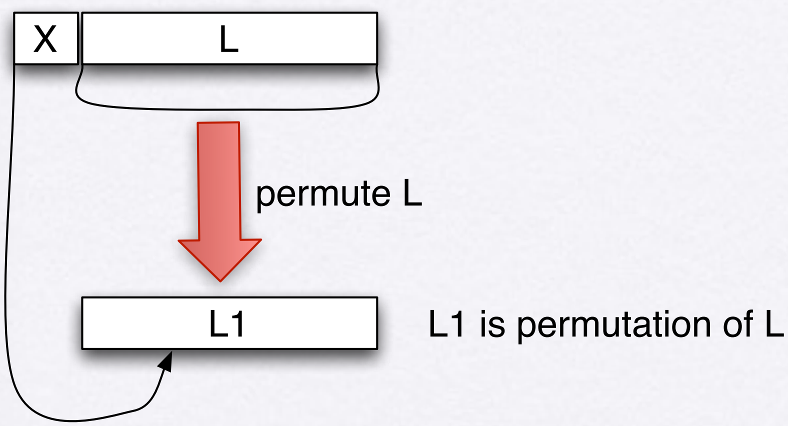
```
now: sublist(S,[a,b,c]).
S = [];
S = [a];
S = [a,b];
S = [a,b,c];
S = [b]; etc...
```

subset(Set,Subset)…. so that subset([a,b,c],S) produces all subsets of [a,b,c]:

```
subset([],[]).

subset([First|Rest],[First|Sub]):-
  subset(Rest,Sub). %keep First

subset([First|Rest],Sub) :-
  subset(Rest,Sub).
```

permutations

```
permutation([a,b,c],P)  =>
P = [a,b,c]; P = [a,c,b]; P = [b,a,c]… etc
```




```
permutation([],[]).

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

```
permutation1([],[]).

permutation1(L,[X|P]) :-
 del(X,L,L1),
 permutation1(L1,P).
```

controlling backtracking with cut -- !

use cut

- to execute only 1 clause of a predicate (now the predicate functions as if-then-else)
- to limit search through facts
- to handle exceptions

if-then-else

alternate paths are defined by multiple clauses with same name and arity

once a path is chosen (then-part or else-part), prevent backtracking from taking a different path… so

each clause in predicate contains the cut among its goals!

cut !

• when a cut is encountered as a goal, the system is committed to all choices made since the parent goal was invoked. Any attempt to resatisfy a goal between parent goal and cut fails -- choices made are now frozen

• foo :- a,b,c, !, d,e,f.
prolog may backtrack among a,b,c, until c succeeds; now it crosses the *fence*, then it may backtrack among d,e,f but if d fails, the entire conjunction and the goal foo fail…

• cut: if you get to here, you found the right rule.

• cut with fail: if you get to here, stop trying to satisfy this goal.

• cut: if you get to here, you found the only solution, don't try alternatives, i.e. don't backtrack.

cut.....

```
p(X):-  a(X).

p(X):-  b(X),  c(X),  d(X),  e(X).

p(X):-  f(X).

a(1). b(1). c(1). d(2). e(2).
f(3). b(2). c(2).

?- p(X).

  X  =  1  ;

  X  =  2  ;

  X  =  3  ;
  no
```

```
p(X):-  a(X).

p(X):-  b(X),  c(X), !, d(X), e(X).

p(X):-  f(X).

a(1). b(1). c(1). d(2). e(2).
f(3). b(2). c(2).

?- p(X).

  X  =  1  ;

  no
```

what happens? X = 1 in 1. rule.
2. rule: goal: b(1),c(1),!,d(1),e(1)…. b(1),c(1) succeeds so !, goals d(1),e(1). Now we are committed to X=1 and to using 2.rule!!!!
d(1) fails. We can't try X=2 and we can't try 3. rule...

F. Oppacher, 3007

cut...

```
s(X,Y):-  q(X,Y).
s(0,0).
q(X,Y):-  i(X),  j(Y).
i(1). i(2). j(1). j(2). j(3).
```

```
?-  s(X,Y).
X  =  1
Y  =  1  ;
X  =  1
Y  =  2  ;
X  =  1
Y  =  3  ;
X  =  2
Y  =  1  ;
X  =  2
Y  =  2  ;
X  =  2
Y  =  3  ;
X  =  0
Y  =  0;
no
```

```
s(X,Y):-  q(X,Y).
s(0,0).
q(X,Y):-  i(X), !, j(Y).
i(1). i(2). j(1). j(2). j(3).
```

```
?-  s(X,Y).
X  =  1   committed to this
Y  =  1  ;
X  =  1
Y  =  2  ; ok to backtrack on j
X  =  1
Y  =  3  ; ok to backtrack on j
X  =  0   ok to use s(0,0)...
Y  =  0;
no
```

F. Oppacher, 3007

cut and **not**

```
sum from 1 to N so that ?- sum_to(5,X). ==> X=15

sum_to(1,1) :- !.  %easy to specify patterns with lists but...

sum_to(N, Res) :-
       N1 is N-1,
       sum_to(N1,Res1),
       Res is Res1+N.
```

2. rule should only be used for numbers ≠ 1. As far as Prolog knows both rules are alternatives for sum_to(1,X)…
Cut says: never try 2. rule if the number is 1.

```
sum_to(1,1).

sum_to(N, Res) :-
       not(N=1),  % not instead of cut    not(N=1) is also N\=1
       N1 is N-1,
       sum_to(N1,Res1),
       Res is N1+Res1.
```

not preferable but sometimes inefficient

```
A :- B, C.

A :- not(B), D.

Here Prolog may try to satisfy B twice!!

more efficient:

A :- B, !, C.

A :- D.
```

suppose we need *append* only when we have 2 known lists and want to know what the longer list is, i.e. append([a,b],[c,d],L):

```
append([],X,X) :- !.  % instead of append([],X,X).

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

F. Oppacher, 3007

controlling backtracking with cut -- !

- coming from left, predicate ! always succeeds
- coming from right,
  ! fails,
  whole rule fails,
  other rules with same predicate fail

suppose function f(X,Y) such that Y = 0 if X ≤ 0, else Y = 1

```
f(X, 0) :- X =< 0, !.
f(X, 1) :- X > 0.
```

without !, if 1. rule succeeds and subsequent backtracking happens, 2. rule will be tried *uselessly*…

===> green cut:

program still works without it but inefficient

controlling backtracking with cut -- !

suppose function f(X,Y) such that Y = 0 if X ≤ 0, else Y = 1

```
f(X, 0) :- X =< 0, !.
f(X, 1). % this is only done when 1. rule has failed
```

without !, if 1. rule succeeds and subsequent backtracking happens, 2. rule gives *wrong* result…

===> red cut:

program gives wrong result without !

```
max(X,Y,X) :- X >= Y, !.      max(X,Y,Y).

mem1(X, [X|T]) :- !.
mem1(X, [_|T]) :- mem1(X,T).
% if X occurs several times, mem1 only answers once…
```

red and green cut

```
 max(X,Y,Y):-  X  =<  Y.
 max(X,Y,X):-  X>Y.
e.g. max(1,2,Y) ==> Y=2  ok but when later backtracking
happens it tries to resatisfy max(1,2,Y) with 2. rule, silly!
```

```
better:
max(X,Y,Y)  :-  X  =<  Y,!.   green cut doesn't change
max(X,Y,X)  :-  X>Y.          meaning of program!!!!
```

```
better ?:
max(X,Y,Y)  :-  X  =<  Y,!.
max(X,Y,X).  max(1,2,X) => X=2 but
?- max(1,2,1) succeeds grrrrrr maybe don't unify vbl before
traversing the cut  hmmmm
```

```
red cut:
max(X,Y,Z)  :-  X  =<  Y,!,  Y  =  Z.
max(X,Y,X).
```

F. Oppacher, 3007

cut and fail to state exceptions

- built-in **fail** fails immediately, i.e. causes backtracking

```
?- mortal(X), write(X), fail.  % writes all mortals...
```

- **cut** prevents backtracking

so??? **exceptions to general rules**

```
i like all chips:   like(i,X)  :-  chips(X).

but I don't like Doritos:

like(i,X)  :-  dorito(X),!,fail.
like(i,X)  :-  chips(X).

chips(X)  :-  chip_kind_x(X).
chips(X)  :-  dorito(X).
chips(X)  :-  chip_kind_y(X).

chip_kind_x(a).                 i like a,c,d but not b!!!
dorito(b).
chip_kind_x(c).
chip_kind_y(d).
```

cut and fail to state exceptions negation as failure

not so good:

- order of rules is crucial
- doesn't work without cut -- red cut!

better:

```
neg(Goal)  :-  Goal,!,fail.
neg(Goal).
```

```
like(i,X) :- chips(X), neg(dorito(X)).
```

```
or +\ dorito(X).
```

unfortunately order of clauses is crucial again:

```
like(i,X) :- +\ dorito(X), chips(X).
?- like(i,X) ==> no  :((
```

cut and fail

built-in **fail** fails immediately, i.e. causes backtracking

how much tax to pay, excluding foreigners, wealthy people, poor pensioners etc?

```
average_taxpayer(X) :- foreigner(X), fail.
average_taxpayer(X) :- .........
```

now ask ?- average_taxpayer(foo), foo is foreigner; so 1. rule matches, fail;
Prolog now tries next rule and treats foo like an ordinary taxpayer :(hmmm?

```
average_taxpayer(X) :- foreigner(X), !, fail.  % don't try alternative ways of
                                               % satisfying the goal!!!!

average_taxpayer(X) :-                         % spouse too rich
            spouse(X,Y), income(Y,Inc), Inc > 5000, !, fail.

average_taxpayer(X) :-
            income(X,Inc), 3000 < Inc, 80000 > Inc. % average guy

income(X,Y) :-
            receives_pension(X,P),
            P < 5000, !, fail.
etc.........
```

F. Oppacher, 3007

controlling backtracking with cut -- !

```
thermostat: if temperature too high
            then turn heat off
            else if temperature too low
                 then turn heat on
                 else do nothing

suppose we read from some device: temp(37). temp(14)…..


thermostat(Action) :-
       temp(X),
       action(X, Action),
       write('Action': Action),nl,
       fail.


action(X, 'turn off heat') :-
    X > 24, !.
action(X, 'turn on heat') :-
    X < 16, !.
action(_, 'do nothing').
```

```
?- thermostat(Temp).
Action: turn off heat
Action: turn on heat

without cut:
Action: turn off heat
Action: do nothing   etc...
```

controlling backtracking with cut -- !

limiting search through facts:

e.g. look for only 1 temperature

```
thermostat(Action) :-
        temp(X), !,
        action(X, Action),
        write('Action': Action),nl,
        fail.
```

```
A :- B, !, fail.
A :- D.
```

equivalent:

```
A:- not(B), D.
```

confirming failure:

built-in not, i.e \+ can be defined in terms of cut and fail

```
not(Goal) :-
  call(Goal), !, fail.
not(Goal).
```

don't turn on heat if it's already on;
add fact: heat(on). (to be updated...)

```
action(X, 'turn on heat') :-
     X < 16,
     not(heat(on)), !.
```

btw, keep in mind...

- write facts **before** rules!
- always write a correct program **before** using cuts!!
- think: what has to be **true** for the goal to succeed, not what has to happen...

is X an element in an ordered binary tree represented as node(Left,Value,Right)?

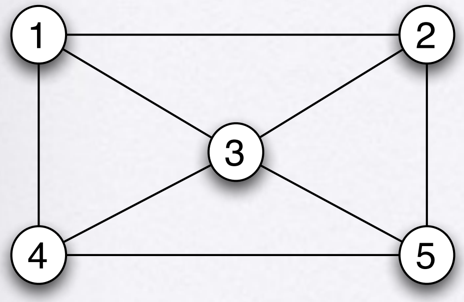
```
element(X, node(Left, X, Right)).

element(X, node(Left, Y, Right)) :-
        X < Y,
        element(X, Left).

element(X, node(Left, Y, Right)) :-
        X > Y,
        element(X, Right).
```

graph

```
given a graph described with edges: edge(a,b) etc.

path(X,X).
path(X,Z) :- edge(X,Y), path(Y,Z).

What if the graph is cyclic?

path(X,X,Visited).

path(X,Z,Visited) :-
      edge(X,Y),
      not member(Y,Visited),      %%  +\
      path(Y,Z,[Y|Visited]).

path(X,Y,Path) :- path(X,Y,[X],Path).   %% keeping the path found
path(X,X,Visited,Visited).
path(X,Z,Visited,Path) :-
      edge(X,Y),
      not member(Y,Visited),      %%  +\
      path(Y,Z,[Y|Visited],Path).
```

F. Oppacher, 3007

graph




```
edge(1,2).
edge(1,4).
edge(1,3).
edge(2,3).
edge(2,5).
edge(3,4).
edge(3,5).
edge(4,5).
```

Express the fact that the edges are bi-directional without adding more facts like edge(2,1). etc.:

```
connected(X,Y) :- edge(X,Y) ; edge(Y,X).
```

```
?- path(1,5,P).
P = [1,2,5] ;
P = [1,2,3,5] ;
P = [1,2,3,4,5] ;
P = [1,4,5] ;
P = [1,4,3,5] ;
P = [1,4,3,2,5] ;
P = [1,3,5] ;
P = [1,3,4,5] ;
P = [1,3,2,5] ;
no
```

```
path(A,B,Path) :-
        travel(A,B,[A],Q),
        reverse(Q,Path).

travel(A,B,P,[B|P]) :-
        connected(A,B).
travel(A,B,Visited,Path) :-
        connected(A,C),
        C \== B,
        \+member(C,Visited),
        travel(C,B,[C|Visited],Path).
```

map

```
(define (map f l)
 (if (null? l)
     '()
      (cons (f (car l)) (map f (cdr l)))))
```

```
prolog map......

map_pl([],[]).     %% which function??

map_pl([X|T1],[Y|T2]) :-
     f(X,Y),
     map_pl(T1,T2).
```

ok but we have to write the same pattern for **each** function :(grrrrr

map

```
use   =..
=.. converts between a prolog term S and a list L
S =.. L
?- S=..[somefun, bla1,bla2]
S = somefun(bla1, bla2)


map_pl(F,[],[]).
map_pl(F,[X|T1],[Y|T2]) :-
     Goal =.. [F,X,Y],
     call(Goal),
     map_pl(F,T1,T2).


e.g.
     fun(X,Y) :- Y is X + 10.

     ?- map_pl(fun, [1,2,3,4,5], Res).
        Res = [11,12,13,14,15]
```

quicksort

how many prolog
programmers does it
take to change a
light bulb?
Yes.

```
quicksort ....
1. partition, split  around a pivot
2. quicksort left side, smaller than pivot
3. quicksort right side, larger than pivot
4. combine results


quicksort([], []).        % easy….

quicksort([X | Tail], Sorted) :- % head is pivot
1. partition  around a pivot
2. quicksort left side, smaller than pivot
3. quicksort right side, larger than pivot
4. combine results


quicksort([], []).        % easy...

quicksort([X | Tail], Sorted) :- % head is pivot
    partition(X, Tail, Small, Big), %assume we have it...
    quicksort(Small, SortedSmall),  % easy….
    quicksort(Big, SortedBig),      % easy….
4. combine results
```

quicksort

```
quicksort([], []).          % easy...

quicksort([X | Tail], Sorted) :- % head is pivot
    partition(X, Tail, Small, Big), %assume we have it...
    quicksort(Small, SortedSmall),  % easy….
    quicksort(Big, SortedBig),      % easy….
    append(SortedSmall, [X|SortedBig], Sorted).
                                 % don't forget pivot!!
```

```
partition(X, [], [], []).

partition(X, [Y|Tail], [Y|Small], Big) :-
    X > Y, !,
    partition(X, Tail, Small, Big).

partition(X, [Y|Tail], Small, [Y|Big]) :-
    partition(X, Tail, Small, Big).
```

insertion sort

```
to insertion-sort non-empty list L = [X | T]:

  (i)  sort tail T of L
  (ii) insert head X of L into sorted tail so that resulting
  list is sorted => result is the whole sorted list

insert_sort([],[]).

insert_sort([X|T],Sorted) :-
  insert_sort(T,SortedTail),
  insert(X, SortedTail, Sorted).

insert(X,[],[X]).

insert(X,[Y|Sorted],[Y| Sorted1]) :-
   X > Y, !,
   insert(X,Sorted, Sorted1).

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

```
sort([],[]).
sort([X|Xs],Ys) :-
  sort(Xs,Zs),
  insert(X,Zs,Ys).

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

other version of sublist

• find a matching first element, and then
• make sure rest of 1. argument matches rest of 2. argument element for element…

so that, e.g.

```
sublist([b,c,d],[a,x,y,b,c,d,e,f,g]) ==> success
```

```
sublist([X|L],[X|M]) :- prefix(L,M), !.

sublist(L,[_|M]) :- sublist(L,M).


prefix([],_).

prefix([X|L],[X|M]) :- prefix(L,M).
```

depth-first, like Prolog itself

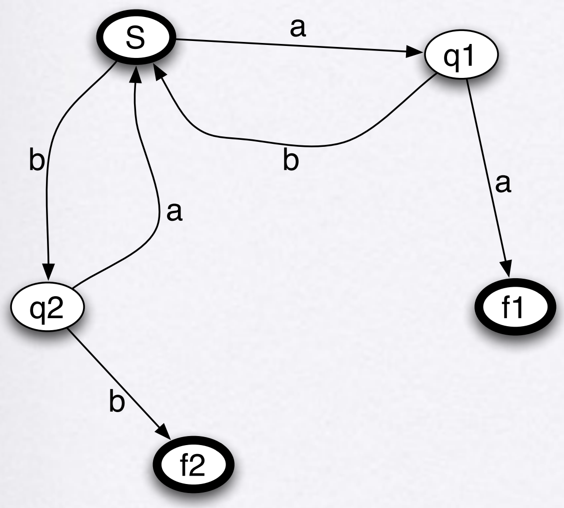
to find a solution path Sol from a node N to a goal node:

- if N is a goal node then Sol = [N], or

- if there is a successor node N1 of N, such that there is a path Sol1 from N1 to a goal node, then
  Sol = [N | Sol1].

```
solve(N, [N]) :- goal(N).

solve(N, [N | Sol1]) :-

     s(N, N1),     %% there is a legal move from N to N1,
                   %% i.e. N1 is successor of N

     solve(N1, Sol1).
```

FSM



start state s, final states f1, f2.

which strings will be accepted?

```
start(s).
final(f1).
final(f2).
delta(s,a,q1).  %% state transitions...
delta(s,b,q2).
delta(q1,b,s).
delta(q1,a,f1).
delta(q2,a,s).
delta(q2,b,f2).
```

a program to run **any** FSM:

```
accept(S) :-
       start(s), accept(Q,S).

accept(Q, [X|Xs]) :-
         delta(Q,X,Q1),
         accept(Q1,Xs).

accept(F, []) :-  final(F).
```

monkey, banana

```
monkey is atdoor, onfloor, box is atwindow, monkey hasnot banana.

To get banana, monkey must push box to middle of room under
banana (which hangs from ceiling), stand on box and grasp banana.

state(horiz pos of monkey, vertical pos of monkey, position of
box, has_or_has_not)

initial state:

          state(atdoor,onfloor,atwindow,hasnot)

goal: any state such that

          state(_,_,_,has)

monkey can: grasp banana, climb box, push box, walk around.

monkey's moves: move(State1, M, State2), i.e. State1 obtains
before move M, State2 is the resulting state.
```

monkey, banana

```
grasp:
                move(?, grasp, ?)   hmmmm...
                move(state(middle,onbox,middle,hasnot),
                     grasp,
                     state(middle,onbox,middle,has)).

climb box:
                move(state(P,onfloor,P,H),
                     climb,
                     state(P,onbox,P,H)).

push box:
                move(state(P1,onfloor,P1,H),
                     push(P1,P2),
                     state(P2,onfloor,P2,H)).


walk around: as a move schema
                move(state(P1,onfloor,B,H),
                     walk(P1,P2),
                     state(P2,onfloor,B,H)).
```

monkey, banana

can the monkey in some initial state S get the banana?

```
canget(S) ???

canget(state(_,_,_,has)).

canget(S1) :-
        move(S1,M,S2),
        canget(S2).        that's all…
```

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

unfortunately, the order of clauses and subgoals is important. Procedurally, the monkey prefers grasping to climbing etc.

Suppose walk clause is first:
move(state(atdoor,onfloor,atwindow,hasnot),M',S2'), canget(S2')
with walk(atdoor,P2') we get canget(state(P2',onfloor,atwindow,hasnot)) and this becomes move(state(P2',onfloor,atwindow,hasnot),M'',S2''), canget(S2'');
walk matches again and the goal list becomes:
move(state(P2'',onfloor,atwindow,hasnot),M''',S2'''), canget(S2''') etc etc etc

This program is declaratively correct but could be procedurally incorrect, may never find a solution if it takes a wrong path, based on ordering of clauses….