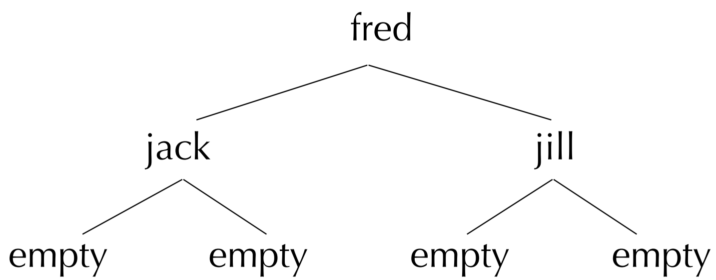
Recursive Programs

- Compound terms can contain other compound terms.
- A compound term can contain the same kind of term, i.e. it can be *recursive*.

```
tree(tree(empty, jack, empty ), fred, tree( empty, jill, empty ))
```

- "empty" is an arbitrary symbol used to represent the empty tree.
- A structure like this could be used to represent a binary tree that looks like:



Binary Trees

- A binary tree is either empty or it is a structure that contains data and left and right subtrees which are also trees.

- To test if some datum is in the tree:

```
in_tree(X, tree(_, X, _)).
in_tree(X, tree(Left, Y, _)) :-
    X \= Y,
    in_tree(X, Left).
in_tree(X, tree(_, Y, Right)) :-
    X \= Y,
    in_tree(X, Right).
```

The size of a tree

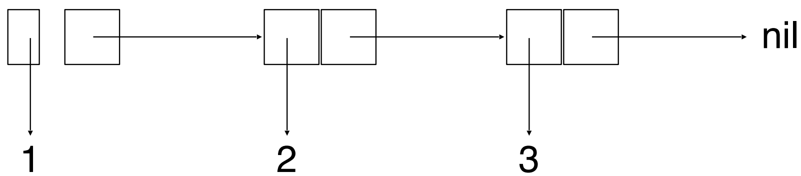
- The size of the empty tree is 0.
- The size of a non-empty tree is the size of the left subtree plus the size of the right subtree plus one for the current node.

```
tree_size(empty, 0).
tree_size(tree(Left, _, Right), N) :-
    tree_size(Left, LeftSize),
    tree_size(Right, RightSize),
    N is LeftSize + RightSize + 1.
```

Lists

- A list may be nil or it may be a term that has a head and a tail. The tail is another list.
- A list of numbers, [1, 2, 3] can be represented as:

```
list(1, list(2, list(3, nil)))
```



- Since lists are used so often, Prolog has a special notation:

```
[1, 2, 3] = list(1, list(2, list(3, nil)))
```

Examples of Lists

```
?- [X, Y, Z] = [1, 2, 3].
X = 1
Y = 2
Z = 3
```

Unify the two terms on either side of the equals sign.

Variables match terms in corresponding positions.

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

The head and tail of a list are separated by using '|' to indicate that the term following the bar should unify with the tail of the list

```
?- [X | Y] = [1].
X = 1
Y = []
```

The empty list is written as '[]'.

The end of a list is *usually* '[]'.

More list examples

```
?- [X, Y | Z] = [fred, jim, jill, mary].

X = fred
Y = jim
Z = [jill, mary]
```

There must be at least two elements in the list on the right

```
?- [X | Y] = [[a, f(e)], [n, b, [2]]].

X = [a, f(e)]
Y = [[n, b, [2]]]
```

The right hand list has two elements:

`[a, f(e)]`   `[n, b, [2]]`

Y is the tail of the list, [n, b, [2]] is just one element

List Membership

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

Rules about writing recursive programs:

- Only deal with one element at a time.
- Believe that the recursive program you are writing has already been written and works.
- Write definitions, not programs.

Concatenating Lists

```
conc([1, 2, 3], [4, 5], [1, 2, 3,4, 5])
```

Start planning by considering simplest case:

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

Clause for this case:

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

Concatenating Lists

Next case:

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

Since `conc([], [2], [2])`

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

Entire program is:

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

Reversing Lists

```
rev([1, 2, 3], [3, 2, 1])
```

Start planning by considering simplest case:

```
rev([], [])
```

Note:

```
rev([2, 3], [3, 2])
```

and

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

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

An Application of Lists

Find the total cost of a list of items:

```
cost(flange, 3).
cost(nut, 1).
cost(widget, 2).
cost(splice, 2).
```

We want to know the total cost of [flange, nut, widget, splice]

```
total_cost([], 0).
total_cost([A | B], C) :-
       total_cost(B, B_cost),
       cost(A, A_cost),
       C is A_cost +  B_cost.
```

Prolog is relational *not* functional

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

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

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


?- append(X, [], _).
X = [] ;
X = [_15986] ;
X = [_15986, _16652] ;
X = [_15986, _16652, _17318] ;
X = [_15986, _16652, _17318, _17984] ;
X = [_15986, _16652, _17318, _17984, _18650] ;
X = [_15986, _16652, _17318, _17984, _18650, _19316]
```

Prolog:

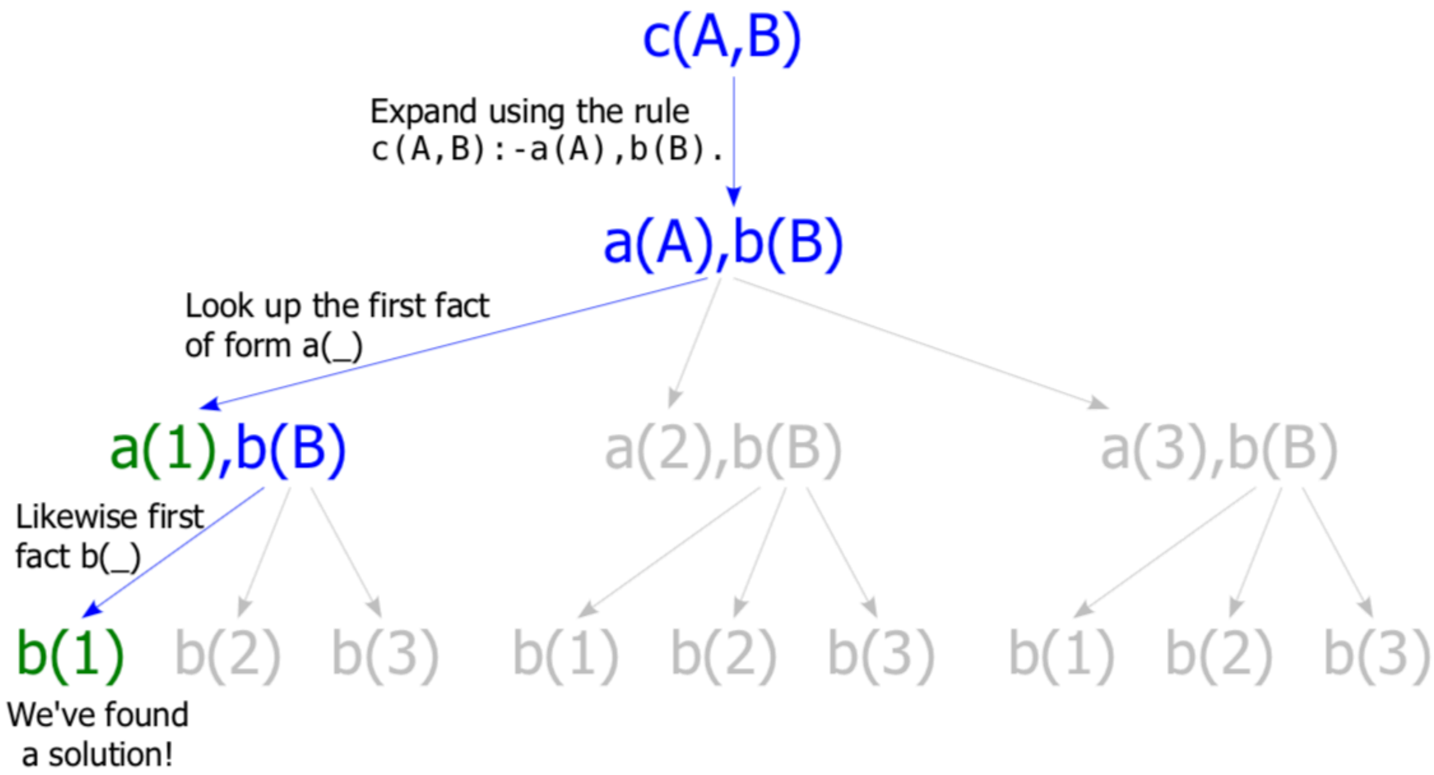
Controlling Execution

Prolog – Finding Answers

Prolog uses depth first search to find answers

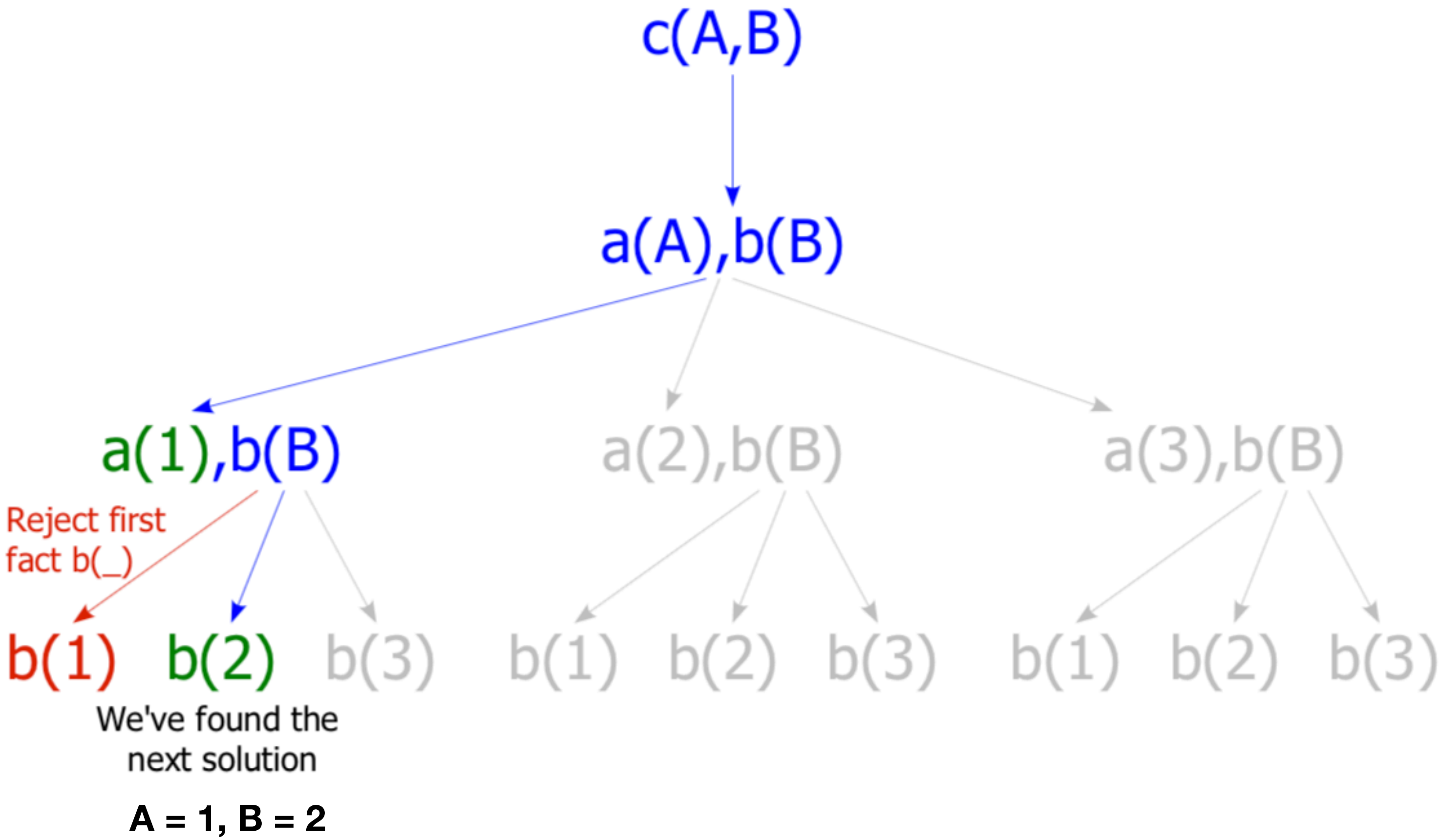
```
a(1).
a(2).
a(3).
b(1).
b(2).
b(3).

c(A, B) :- a(A), b(B).
```

Depth-first solution of query c(A,B)

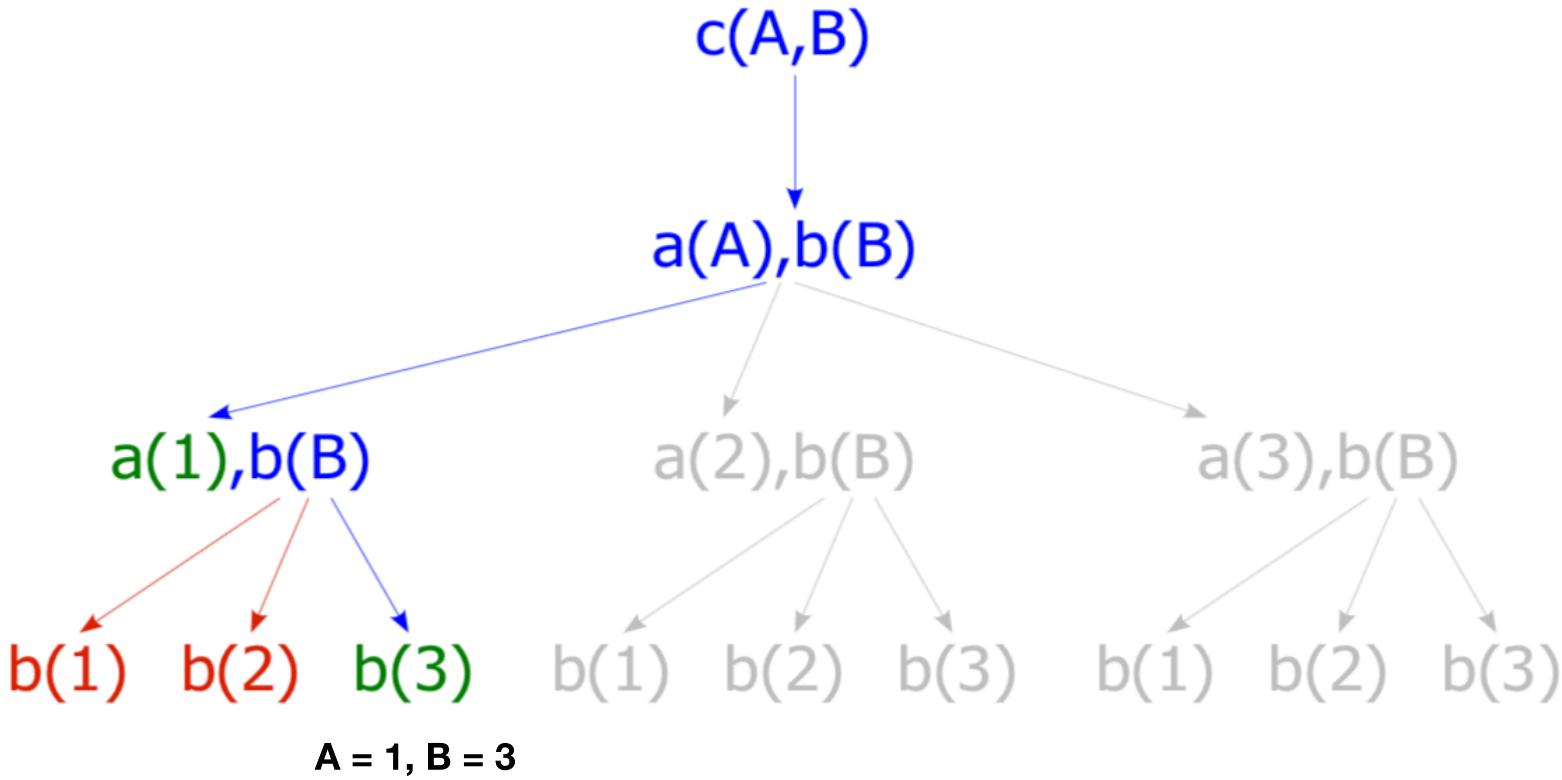


A = 1, B = 1

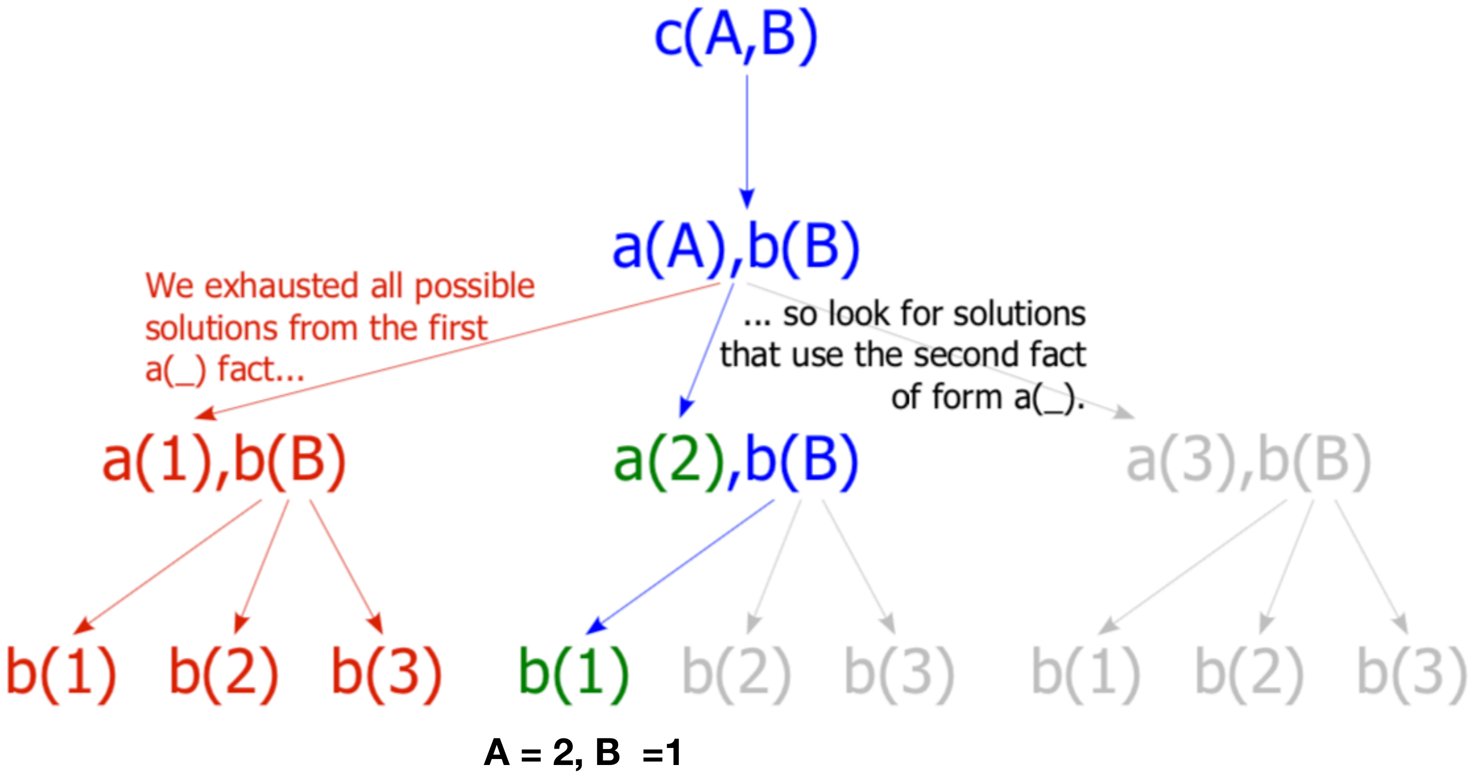
Backtrack to find another solution



Backtrack to find another solution



Backtrack to find another solution

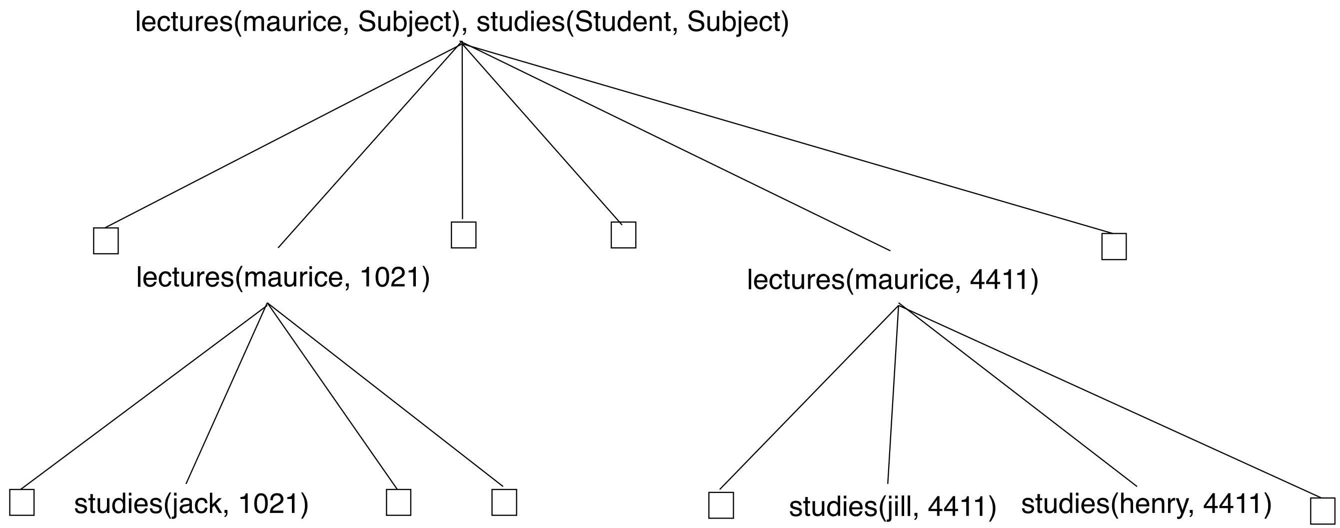


The Cut (!)

- Sometimes we need a way of preventing Prolog from finding all solutions
- The *cut* operator is a built-in predicate that prevents backtracking
- It violates the declarative reading of a Prolog programming
- Use it *VERY sparingly!!!*

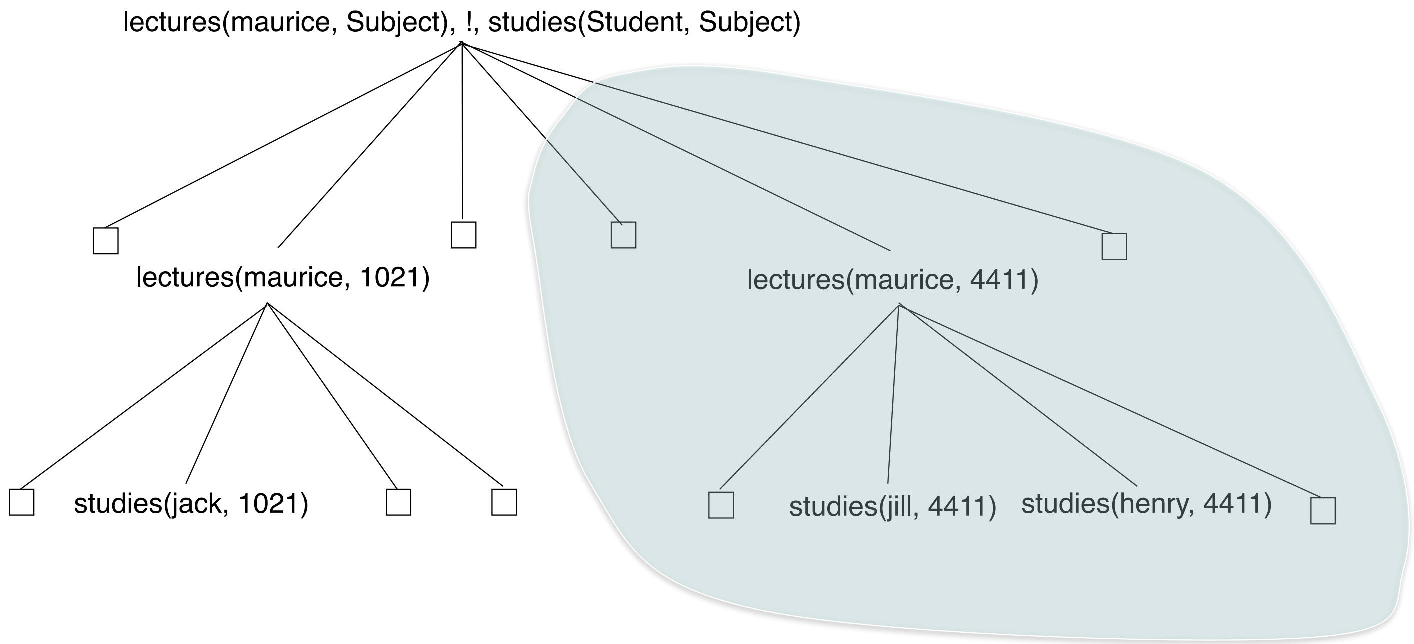
Backtracking

```
lectures(maurice, Subject), studies(Student, Subject)?
Subject = 1021
Student = jack ;

Subject = 4411
Student = Jill ;

Subject = 4411
Student = Henry
```

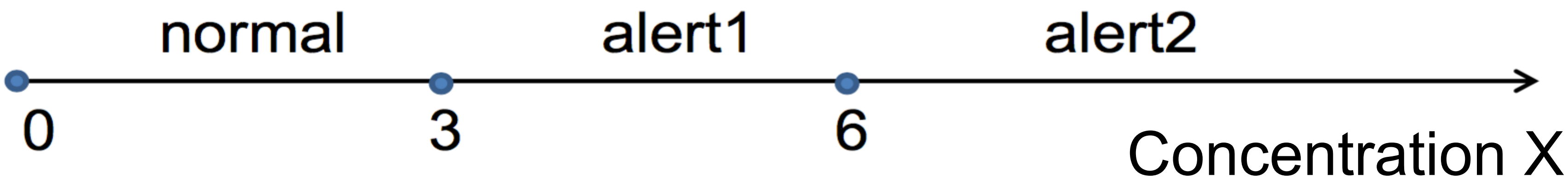


Cut prunes the search

```
lectures(maurice, Subject), !, studies(Student, Subject)?
Subject = 1021
Student = jack ;

Subject = 4411
Student = Jill ;

Subject = 4411
Student = Henry
```



Example



Rules for determining the degree of pollution

Rule 1: if $X < 3$ then Y = normal

Rule 2: if $3 \leq X$ and $X < 6$ then Y = alert1

Rule 3: if $6 \leq X$ then Y = alert2

In Prolog: **`f(Concentration, Pollution_Alert)`**

```
f(X, normal) :-  X < 3.              % Rule1
f(X, alert1) :-  3 =< X, X < 6.      % Rule2
f(X, alert2) :-  6 =< X.             % Rule3
```

Alternative Version

```
f(X, normal) :-  X < 3, !.          % Rule1
f(X, alert1) :-  X < 6, !.          % Rule2
f(X, alert2).                       % Rule3
```

Which version is easier to read?

Reference

- Ivan Bratko, *Programming in Prolog for Artificial Intelligence*, 4th Edition, Pearson, 2013.