1: Append in Prolog II

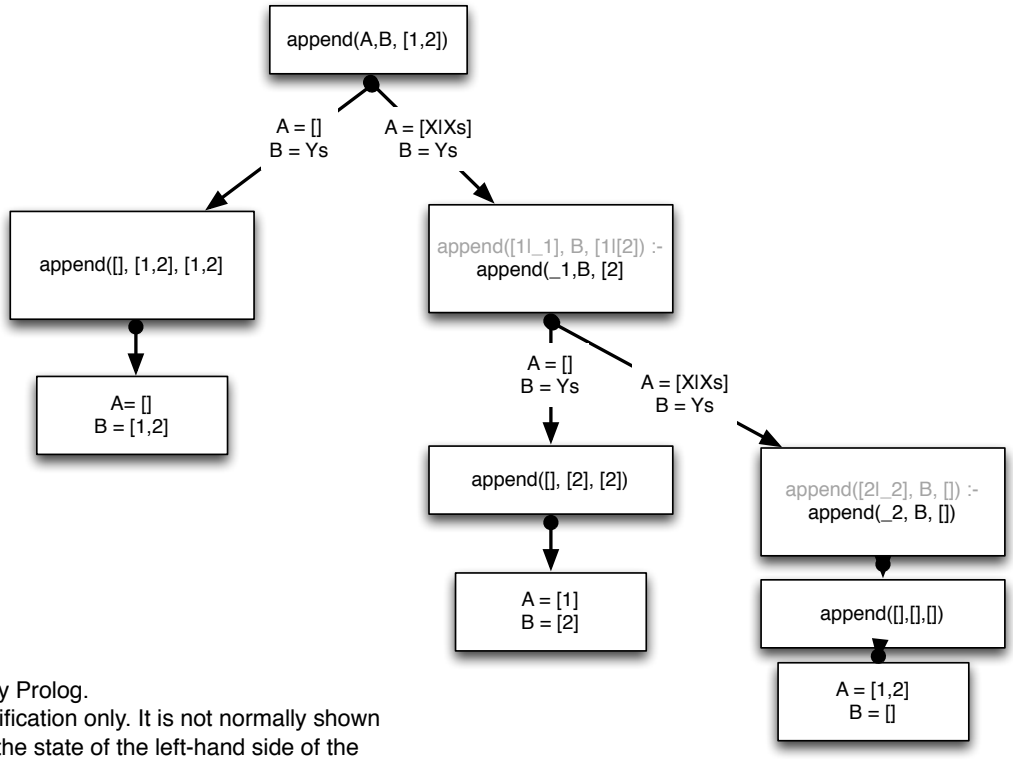
In Prolog, one list (written as [a,b,c] or [] for the empty list) is appended to another by the following code:

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

What happens if you call acc_append „in reverse“ – giving variables for the two input lists and a result list for the accumulator/result variable? Why? Trace the Prolog query and show the search tree for acc_append(X, Y, [1,2]).

Prolog searches for all possible ways of proving the query, returning all possible combinations of lists A and B given the fixed result. The search tree is shown below.

The first decision is always for the base case rule. When this is undone later in the search, the second rule is chosen and one more element of the result list is assigned to list A.



Notes:
_1 is a variable created by Prolog.
The text in grey is for clarification only. It is not normally shown in search trees. It shows the state of the left-hand side of the rule that led to the new query.

2: List-reverse in Prolog

Here is the LISP code for list reverse using an accumulator. Re-write it in Prolog.

```
(define list-reverse-aux (lambda (L A)
             (if (null? L) A
                 (list-reverse-aux (cdr L) (cons (car L) A)))))

(define list-reverse (lambda (L) (list-reverse-aux L '())))
```

Hint 1: There is no cons predicate; use the [Head|Rest] notation to create a new list.

Hint 2: If you want to see the result of the list reversal, your query will have to contain an unbound variable that takes on the value of A eventually (in addition to the actual accumulator).

```
reverse_aux([],A,A).
reverse_aux([H|T],A,Result) :- reverse_aux(T,[H|A], Result).
reverse(List,Result) :- reverse_aux(List,[],Result).
```