

## More on Append

What if we type in the query

```
app(X,Y,Z).
```

The answer given by prolog:

```
1 ?- [app].
app compiled, 0.00 sec, 1,108 bytes.
```

```
Yes
2 ?- app(X,Y,Z).
```

```
X = []
Y = _G143
Z = _G143 ;
```

```
X = [_G260]
Y = _G143
Z = [_G260|_G143] ;
```

```
X = [_G260, _G266]
Y = _G143
Z = [_G260, _G266|_G143] ;
```

```
X = [_G260, _G266, _G272]
Y = _G143
Z = [_G260, _G266, _G272|_G143] ;
```

```
Yes
3 ?-
```

**Note the order of the answers depends on the order of the clauses in the database**

If the rules were in the other order then prolog would reverse the order of the output and never generate even one solution

## Removing an Item from a List

How do we remove a single item from a list

```
skip(item, list, list_with_item_removed)
```

If the first element of the list equals the element to remove then the answer is the rest of the list

```
skip(X, [X|Y], Y).
```

Or the first element followed by the rest of the list with the specified element removed

```
skip(X, [Y|Ys], [Y|Zs]) :- skip(X, Ys, Zs).
```

Testing

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

```
X = [1, 3] ;
```

```
No
5 ?- skip(2, X, [a,b,c]).
```

```
X = [2, a, b, c] ;
```

```
X = [a, 2, b, c] ;
```

```
X = [a, b, 2, c] ;
```

```
X = [a, b, c, 2] ;
```

```
No
6 ?- skip(1, [2,3,4], X).
```

```
No
```

## Permutations

How do we generate permutations of a list?

Then the following will check for permutations

```
perm([], []).
perm([X|Xs], Y) :- skip(X, Y, R), perm(Xs, R).
```

Empty lists are permutations of each other

Removing the first element from a list and the same element from the second list and the remainders are permutations

Try it out

```
9 ?- perm(X, [1,2]).
```

```
X = [1, 2] ;
```

```
X = [2, 1] ;
```

```
No
10 ?-
```

## But it goes into an infinite loop the other way

```
12 ?- trace, perm([1,2], X).
Call: ( 8) perm([1,2], _G209)
Call: ( 9) skip(1, _G209, _L143)
Exit: ( 9) skip(1, [1|_G322], _G322)
Call: ( 9) perm([2], _G322)
Call: (10) skip(2, _G322, _L170)
Exit: (10) skip(2, [2|_G325], _G325)
Call: (10) perm([], _G325)
Exit: (10) perm([], [])
Exit: ( 9) perm([2], [2])
Exit: ( 8) perm([1,2], [1,2])
```

```
X = [1,2] ;
Redo: (10) skip(2, _G322, _L170)
Call: (11) skip(2, _G325, _G328)
Exit: (11) skip(2, [2|_G328], _G328)
Exit: (10) skip(2, [_G324, 2|_G328], [_G324|_G328])
Call: (10) perm([], [_G324|_G328])
Fail: (10) perm([], [_G324|_G328])
Redo: (11) skip(2, _G325, _G328)
Call: (12) skip(2, _G331, _G334)
Exit: (12) skip(2, [2|_G334], _G334)
Exit: (11) skip(2, [_G330, 2|_G334], [_G330|_G334])
Exit: (10) skip(2, [_G324, _G330, 2|_G334], [_G324, _G330|_G334])
Call: (10) perm([], [_G324, _G330|_G334])
Fail: (10) perm([], [_G324, _G330|_G334])
Redo: (12) skip(2, _G331, _G334)
Call: (13) skip(2, _G337, _G340)
Exit: (13) skip(2, [2|_G340], _G340)
Exit: (12) skip(2, [_G336, 2|_G340], [_G336|_G340])
Exit: (11) skip(2, [_G330, _G336, 2|_G340], [_G330, _G336|_G340])
```

## The Letter Problem

What consistent assignments to letters make the following sum correct?

```
  S E N D
+ M O R E
-----
M O N E Y
```

We need to be able to add

We can type in all of the basic digit addition facts

We need to make sure that the letters have distinct values

We could do this by using permutations of all the digits and insisting that the letters be part of this permutation

## Addition problem

```
addlist([], [], 0, []).
addlist([X|Xs], [Y|Ys], CO, [Z|Zs]) :-
    add(X, Y, CI, CO, Z), addlist(Xs, Ys, CI, Zs).

% addlist([0, S, E, N, D], [0, M, O, R, E], 0, [M, O, N, E, Y]).

% addlist([1, 9, 2], [3, 4, 5], 0, X).

%
addlist([0, S, E, N, D], [0, M, O, R, E], 0, [M, O, N, E, Y]), per
m([S, E, N, D, M, O, R, Y|_], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).

%
addlist([W, R, O, N, G], [W, R, O, N, G], 0, [R, I, G, H, T]), per
m([W, R, O, N, G, I, H, T|_], [1, 2, 3, 4, 5, 6, 7, 8, 9]).

%add(A, B, C, T, U)
add(0, 0, 0, 0, 0).
add(1, 0, 0, 0, 1).
add(2, 0, 0, 0, 2).
add(3, 0, 0, 0, 3).
add(4, 0, 0, 0, 4).
add(5, 0, 0, 0, 5).
add(6, 0, 0, 0, 6).
add(7, 0, 0, 0, 7).
add(8, 0, 0, 0, 8).
add(9, 0, 0, 0, 9).
add(0, 0, 1, 0, 1).
add(1, 0, 1, 0, 2).
add(2, 0, 1, 0, 3).
add(3, 0, 1, 0, 4).
add(4, 0, 1, 0, 5).
add(5, 0, 1, 0, 6).
add(6, 0, 1, 0, 7).
add(7, 0, 1, 0, 8).
add(8, 0, 1, 0, 9).
add(9, 1, 0, 1, 0).
add(0, 1, 0, 0, 1).
add(1, 1, 0, 0, 2).
add(2, 1, 0, 0, 3).
add(3, 1, 0, 0, 4).
add(4, 1, 0, 0, 5).
add(5, 1, 0, 0, 6).
add(6, 1, 0, 0, 7).
add(7, 1, 0, 0, 8).
...
add(0, 9, 0, 0, 9).
add(1, 9, 0, 0, 0).
add(2, 9, 0, 1, 1).
add(3, 9, 0, 1, 2).
add(4, 9, 0, 1, 3).
add(5, 9, 0, 1, 4).
add(6, 9, 0, 1, 5).
add(7, 9, 0, 1, 6).
add(8, 9, 0, 1, 7).
add(9, 9, 0, 1, 8).
add(0, 9, 1, 1, 0).
add(1, 9, 1, 0, 1).
add(2, 9, 1, 1, 2).
add(3, 9, 1, 1, 3).
add(4, 9, 1, 1, 4).
add(5, 9, 1, 1, 5).
add(6, 9, 1, 1, 6).
add(7, 9, 1, 1, 7).
add(8, 9, 1, 1, 8).
add(9, 9, 1, 1, 9).
```

## Output

```
3 ?- addlist([1, 9, 2], [3, 4, 5], 0, X).
```

```
X = [5, 3, 7] ;
```

```
No
```

```
4 ?-
```

```
addlist([0, S, E, N, D], [0, M, O, R, E], 0, [M, O, N, E, Y]).
```

```
S = 0
```

```
E = 0
```

```
N = 0
```

```
D = 0
```

```
M = 0
```

```
O = 0
```

```
R = 0
```

```
Y = 0 ;
```

```
S = 0
```

```
E = 0
```

```
N = 0
```

```
D = 1
```

```
M = 0
```

```
O = 0
```

```
R = 0
```

```
Y = 1 ;
```

```
S = 0
```

```
E = 0
```

```
N = 0
```

```
D = 2
```

```
M = 0
```

```
O = 0
```

```
R = 0
```

```
Y = 2
```

```
Yes
```

## Trying and Avoiding Duplicates

```
7 ?-
```

```
addlist([0, S, E, N, D], [0, M, O, R, E], 0, [M, O, N, E, Y]), per
m([S, E, N, D, M, O, R, Y|_], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
```

```
S = 2
```

```
E = 8
```

```
N = 1
```

```
D = 7
```

```
M = 0
```

```
O = 3
```

```
R = 6
```

```
Y = 5 ;
```

```
S = 2
```

```
E = 8
```

```
N = 1
```

```
D = 7
```

```
M = 0
```

```
O = 3
```

```
R = 6
```

```
Y = 5 ;
```

```
S = 2
```

```
E = 8
```

```
N = 1
```

```
D = 9
```

```
M = 0
```

```
O = 3
```

```
R = 6
```

```
Y = 7 ;
```

## Cuts

Sometimes a programmer would like to abandon a search or prune the depth-first search that Prolog uses

A solution may have been found and it is a waste of time to continue looking

The cut operator is `!` and causes the search never to back up past the `!` operator

In particular, no further clauses with the same head will be tried

Neither will other possibilities in terms before the cut in the current clause be tried

Goals to the right of the cut behave normally and can backtrack

Cut is a "hack" and does not have a good logical explanation

But it is useful tool

It can cause illogical behavior if it is used incorrectly

They have global effects

## Cut Example

Given the database

```
s(a).
s(b).
r(a).
r(b).
p(X,Y):-l(X).
p(X,Y):-r(X),!,... << note the cut
p(X,Y):-m(X),...
```

and the query

```
?- s(A),p(B,C).
```

The system will

unify A with a

unify B with X and C with Y in the first p clause  
search for l(X) and fail

unify B with X and C with Y in the second p clause  
unify r(X) with r(a)  
do the ... stuff

Because of the cut

r(X) will not be unified with r(b)  
the third p(X,Y) clause will not be tried  
if the ... fails then the entire clause fails

## Kinds of cuts - White cuts

White cuts are those which do not discard solutions

They improve performance because they avoid backtracking (which should fail, anyway), and they, in some Prolog implementations, avoid creating choicepoints at all. An example of white cut is:

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

The two tests are mutually exclusive: since (because of the way arithmetic works in Prolog) both X and Y must be instantiated to numbers, if the first clause succeeds (which will happen if the cut is reached), then the second will not; conversely, if the second clause is to succeed, then the first one could not have succeeded, and the cut in it would not have been reached.

## Kinds of cuts - Green cuts

Green cuts are those which discard correct solutions which are not needed

Sometimes a predicate yields several solutions, but one is enough for the purposes of the program--or one is preferred over the others

Green cuts discard solutions not wanted, but all solutions returned are correct

For example, if we had a database of addresses of people and their workplaces, and we wanted to know the address of a person, we might prefer his/her home address, and if not found, we should resort to the business address. This predicate implements this query:

```
address(X,Add):-home_address(X,Add),!.
address(X,Add):-business_address(X,Add).
```

Another useful example is checking if an element is member of a list, without either enumerating (on backtracking) all the elements of a list or instantiating on backtracking possible variables in the list

The membercheck/2 predicate does precisely this: when the element sought for is found, the alternative clause which searches in the rest of the list is not taken into account:

```
membercheck(X,[X|Xs]):-!.
membercheck(X,[Y|Xs]):-membercheck(X,Xs).
```

Again, it might be useful in some situations, mainly because of the savings in memory and time it helps to achieve

But it should be used with caution, ensuring that it does not remove solutions which are needed.

## Kinds of cuts - Red cuts

Red cuts both discard correct solutions not needed, and can introduce wrong solutions, depending on the call mode. This causes predicates to be wrong according to almost any sensible meaning.

For example, if we wanted to know how many days there are in a year, taking into account leap years, we might use the following predicate:

```
days_in_year(X,366):-number(X),leap_year(X),!.
days_in_year(X,365).
```

The idea behind is: "if X is a number and a leap year, then we succeed, and do not need to go to the second clause. Otherwise, it is not a leap year"

But the query

```
?- leap_year(z,D)
```

succeeds (with D = 365), because the predicate does not take into account that, in any case, a year must be a number

It is arguable that this predicate would behave correctly if it is always called with X instantiated to a number, but the check number(X) would not be needed, and correctness of the predicate will then be completely dependent on the way it is called--which is not a good way of writing predicates.

## More Red Cut Examples

Look at the following implementation of the max/3 predicate which works out the maximum of two numbers:

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

The idea is: if  $X > Y$ , then there is no need to check whether  $X \leq Y$  or not, hence the cut. And, if the first clause failed, then clearly the case is that  $X \leq Y$

But there are two serious counterexamples to this: the first is the query `?- max(5, X, X).`, which succeeds binding nothing (instead of failing or giving an error, which would in any case be a better behavior, at least indicating that there has been a call with a wrong instantiation mode).

In any case, the second counterexample does not violate any sensible assumption: the call `?- max(5, 2, 2).` succeeds instead of failing, because the first head unification fails and the second succeeds!

What happens here is a case of the so-called "output unification": there are unifications made before the cut, which means that data is changed prior to the tests which determine if the (first, in this case) clause is the right one or not. Changing the program to

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

will make the predicate behave correctly in both counterexamples (giving an error in the first, failing in the second).

## Negation as Failure

Negation in Prolog is implemented based on the use of cut

Actually, negation in Prolog is the so-called negation as failure, which means that to negate  $p$  one tries to prove  $p$  (just executing it), and if  $p$  is proved, then its negation,  $\text{not}(p)$ , fails

Conversely, if  $p$  fails during execution, then  $\text{not}(p)$  will succeed. The implementation of `not/1` is as follows:

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

(`fail/0` is a builtin predicate which always fails. It can be trivially defined as `fail:- a = b.`)

`not/1` is usually available as the (prefix) predicate `+ / 1` in most Prolog systems. I.e., `not(p)` would be written `+ p`.

Since `not(p)` will try to execute  $p$ , if the execution of  $p$  does not terminate, the execution of `not(p)` will not terminate, either

Also, since `not(p)` succeeds if and only if  $p$  failed, `not(p)` will not instantiate any variable which could appear in  $p$

This is not a logically sound behavior, since, from a formal point of view, `not(p)` should succeed and instantiate variables for each term for which  $p$  is false

The problem is that this will very likely lead to an infinite number of solutions.

But using negation with ground goals (or, at least with calls to goals which do not further instantiate free variables which are passed to them) is safe, and the programmer should ensure this to hold. Otherwise, unwanted results may show up:

## Examples of Negation as Failure

```
unmarried_student(X) :-
    not(married(X)), student(X) .

student(joe) .
married(john) .
```

This program seems to suggest that joe is an unmarried student, and that joe is not an unmarried student, and indeed:

```
?- unmarried_student(joe) .

yes
?- unmarried_student(john) .

no
```

But, for logical consistence, asking for unmarried students should return joe as answer, and this is not what happens:

```
?- unmarried_student(X) .

no
```

The reason for this is that the call to `not(married(X))` is not returning the students which are not married: it is just failing because there is at least a married student.

The use of cut and a fail in a clause forces the failure of the whole predicate, and is a technique termed cut-fail. It is useful to make a predicate fail when a condition (which may be a call to an arbitrary predicate) succeeds

An example of cut-fail combinations is implementing in Prolog the predicate `ground/1`, which succeeds if no variables are found in a term, and fails otherwise

The technique is recursively traversing the whole term, and forcing a failure as soon as a variable is found:

```
ground(Term) :- var(Term), !, fail.
ground(Term) :-
    nonvar(Term),
    functor(Term, F, N),
    ground(N, Term) .

ground(0, T) .
ground(N, T) :-
    arg(N, T, Arg),
    ground(Arg),
    N1 is N-1,
    ground(N1, T) .
```