Finding Maximum of a List

predicates

maximum (list, integer, integer)

clauses

```
maximum ([], Max, Max).

maximum ([Head|Tail], Max, X) :-
        Head > Max, maximum (Tail, Head, X), !.

maximum ([_|Tail], Max, X)
        :- maximum (Tail, Max, X).
```

goal: maximum ([1,2,6,9,5], 0, Y).

Y = 9, 1 solution

goal: maximum ([1,1,1], 0, Y)

Y = 1 1 solution

goal: maximum ([-1,-2,-3], 0, Y)

Y = 0 1 solution

Instead of this, we may write clause as:

largest (H|T, X) :- maximum (T, H, X).

This is preferred.

Factorial of a Number

goal: factorial(3, N).
N = 6 1 soln

```
factorial(N, NFact)
    :- fact(N, NFact, 1, 1).

/*CL1*/ fact(N, NFact, N, NFact) :- !.

/*CL2*/ fact(N, NFact, I, J) :-
        NextI = I+1, NextJ = NextI * J,
        fact(N, NFact, NextI, NextJ).
```

Execution Trace / Tree

goal: factorial(3, X) calls fact(3, NFact, 1, 1)

fact(N: 3, NFact, I: 1, J: 1)

Matches with CL1, but fails as 1st & 3rd arguments are different. Triggers CL2

↓

NextI = 2, NextJ = 2 (2 * 1)

↓

fact(N: 3, NFact, I: 2, J: 2) CL1 matching fails. CL2 is invoked

↓

NextI = 3, NextJ = 3*2 = 6 /* NextJ is actually (NextI)! */

↓

fact(3, NFact, 3, 6) CL1 succeeds this time and binds 6 to NFact and thus X = 6 is the result

Here, finally when NextI is N, NextJ is already N!, our answer

Rotating Right Elements of a list

goal, rr ([1,2,3,4], R).
R = [4,1,2,3]
1 soln

```
/* Version 1 */

domains
        list = integer*

predicates
        findlast (list, integer)
        findstart (list, list)
        append (list, list, list)
        rr (list, list)

clauses
        findstart ([_], []).
        findstart ([Head | List1]), [Head | List2])
                :- findstart (List1, List2).
        findlast ([Lastelement], Lastelement).
        findlast ([_ | List1], Answer) :-
                findlast (List1, Answer).
        append ([], L, L).
        append ([Head | List1], List2, [Head | List3])
                :- append (List1, List2, List3).
```

```
rr(Z, X) :- findlast(Z, Last),
            findstart(Z, Start),
            append([Last], Start, X).
```

Version 2 — Using 2 "Appends"

```
rr(List, Res) :- last_element(List, Z),
                 append(Prefix, [Z], List),
                 append([Z], Prefix, Res).
```

Version 3 — Using 2 "Reverse"

```
rr(List, Res) :- rev(List, [H|T]),
                 rev(T, T1),
                 append([H], T1, Res).
```

List: 1 2 3 4 — [H|T]: H = 4, T = 3 2 1 — T: 3 2 1 — T1: 1 2 3 — Res: 4 1 2 3

Version 4 — Using "Append" w/o last_element

```
rr(List, Res) :- append(Prefix, [Z], List),
                 append([Z], Prefix, Res).
```

Removes Duplicates from a list

goal: rm_dup ([1,2,3,1,2], X)

X = [3,1,2] 1 soln

goal: rm_dup ([1,1,2,2,3,3], X).

X = [1,2,3]

1 soln

```
rm_dup ([], []) :- !.

rm_dup ([H|T], R) :- member (H, T),
                     rm_dup (T, R), !.

rm_dup ([H|T], [H|Rest]) :-
                     rm_dup (T, Rest).

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

To determine whether a string is a prefix in the list or not

goal: prefix([1, 2], [1, 2, 3, 4]).

Yes

goal: prefix ([], [1, 2, 3]).

Yes

Version 1 — W/o Append

```
prefix([], _).
prefix([H| List1], [H| List2])
        :- prefix(List1, List2).
```

Version 2 — Using Append

```
prefix(R, L) :- append(R, _, L).
```

- To determine whether a string is a suffix of the other list

```
domains
        list = integer*
predicates
        suffix (list, list)
        append (list, list, list)

clauses
        append ([], L, L).
        append ([X|L1], L2, [X|L3]) :-
                append (L1, L2, L3).
        suffix (S, L) :- append (_, S, L).
```

goal: suffix([3,4], [1,2,3,4])

Yes

goal: suffix([], [1,2,3,4])

Yes

Assignment

goal: infix (bb, happy)

Yes

- To find out all the suffixes of a list.

goal: allsuffix ([1,2,3], X).

X = [1,2,3]
X = [2,3]
X = [3]
X = []
4 sol^n

```
allsuffix (Z, Z).
allsuffix ([_|Tail], Z)
        :- allsuffix(Tail, Z).
```

- Deletes all the occurances of an element in a list

goal: delete(1, [1,2,3,1], X).

X = [2,3] 1 soln

predicate

delete (integer, list, list)

clauses

delete(_, [], []) :- !.

This could have been delete(X, [], []) for clarity, but a warning

delete(X, [X|T], R) :-

delete(X, T, R), !.

delete(X, [Y|T], [Y|R]) :-

delete(X, T, R).

Delete first occurance of an element

goal: del([1,2,2,3,1], 1, X).

X = [2,2,3,1] 1 soln

goal: del([2,2,3,1], 1, X).

X = [2,2,3] 1 soln

```
del([], _, []).

del([X|Tail], X, Tail) :- !.

del([Y|Tail], X, [Y|Res]) :-
        del(Tail, X, Res).
```

- Finding last element of a list

```
last_element(L, R)
        :- append(_, [R], L).
```

- Finding Last 3 elements of a list

```
last3element(L, [X, Y, Z]) :-
        append(_, [X, Y, Z], L).
```

Finding last N elements of a list

goal: lastnele ([1, 2, 3, 4], 2, T)

T = [3, 4] 1 soln

goal; lastnele ([1, 2, 3, 4], 0, T)

T = [] 1 soln

```
lastnele (L, N, R) :-
        append(_, R, L),
        length (R, N).
```

Determining whether one set is a subset of other set

goal: subset ([4, 3], [2, 3, 5, 4])

Yes

goal: subset ([], [1, 2])

Yes

Clauses

```
subset ([X | Tail], List) :-
        member (X, List),
        subset (Tail, List).

subset ([], _).
```

Union of two list2

goal: union([1,2], [3,4], X).

X = [1,2,3,4] 1 soln

goal: union(X, [3,4], [1,2,3,4])

gives stack overflow.

clauses

```
/* C1 - Head is in the second list. Ignore it */
union([X|List1], List2, Res) :-
        member(X, List2),
        union(List1, List2, Res), !.

/* C2 - Head is not in second list, add it to 3rd list */
union([X|List1], List2, [X|Res]):-
        union(List1, List2, Res).

/* C3 - Terminating condition */
union([], Z, Z).
```

Count Vowels in a list

goal: nr_vowel([], X).

X = 0

goal: nr_vowel([a, r, e, d, i], X)

X = 3 1 soln

goal: nr_vowel([s, e, e, d], X).

X = 2 1 soln

```
vowel(X) :- member(X, [a, e, i, o, u]).

nr_vowel([], 0).

nr_vowel([X|T], N) :- vowel(X),
                      nr_vowel(T, N1),
                      N = N1 + 1, !.

nr_vowel([X|T], N) :- nr_vowel(T, N).
```

Subset Using Append — Ver 2

```
subset([], _).
subset([H|T], List) :- append(_, [H|_], List),
                       subset(T, List).
```

Counting Number of occurances in a list

goal : num_occ([2, 3, 2, 3], 3, Z).

Z = 2, 1 soln

goal : num_occ([], 2, Z).

Z = 0, 1 soln

predicates

num_occ (list, integer, integer)

clauses

```
c1 num_occ([], _, 0).
c2 num_occ([X|Tail], X, N) :-
                num_occ(Tail, X, NN),
                N = NN+1, !.
c3 num_occ([_|Tail], X, N)
                :- num_occ(Tail, X, N).
```

cut is mandatory in c2, place may be any

To add an element at the beginning

goal: addbeg (3, [4,1,2,6], x).

x = [3,4,1,2,6]

1 soln

clauses

addbeg (X, List, [X | List]).

To check whether a list is ordered or not. in ascending order — Using Append

goal: ordered ([1,2,3,4]).

Yes

goal: ordered ([5,4,1,2]).

No

goal: ordered ([]).

No

```
ordered ([_]).   /* can't write ordered [] */

ordered ([Head, Head1 | Tail])
        :- Head1 > Head, append ([Head1], Tail,
                                 List),
           ordered (List).
```

Using Cut

- Consider the following predicate max/3 that succeeds if the third argument is the maximum of the first two

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

```
?- max(2,3,2).
no

?- max(2,3,5).
no
```

2

The max/3 predicate

- What is the problem?
- There is a potential inefficiency
  - Suppose it is called with ?- max(3,4,Y).
  - It will correctly unify Y with 4
  - But when asked for more solutions, it will try to satisfy the second clause. This is completely pointless!

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

max/3 with cut

- With the help of cut this is easy to fix

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

- Note how this works:
  - If the X =< Y succeeds, the cut commits us to this choice, and the second clause of max/3 is not considered
  - If the X =< Y fails, Prolog goes on to the second clause

Green Cuts

- Cuts that do not change the meaning of a predicate are called **green cuts**
- The cut in max/3 is an example of a green cut:
  - the new code gives exactly the same answers as the old version,
  - but it is more efficient

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - okay

```
?- max(200,300,X).
X=300
yes
```

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - okay

```
?- max(400,300,X).
X=400
yes
```

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - oops....

```
?- max(200,300,200).
yes
```

Problem is of "Output- Unification",
i.e. data is changed before the conditions are tested (Unification made before cut)

Revised max/3 with cut

- Unification after crossing the cut

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

- This does work

```
?- max(200,300,200).
no
```

Red Cuts

- Cuts that change the meaning of a predicate are called **red cuts**
- The cut in the revised max/3 is an example of a red cut:
  - If we take out the cut, we don't get an equivalent program
- Programs containing red cuts
  - Are not fully declarative
  - Can be hard to read
  - Can lead to subtle programming mistakes

https://en.wikibooks.org/wiki/Prolog/Cuts_and_Negation

Negation

not(X) is the way to implement negation in Prolog; however **not(X)** does **not** mean that **X** is false, it means that **X** can't be proven true.

For example, with the database:

```
man('Adam').
woman('Eve').
```

asking *not(man('Abel')).* will return yes.

Cut-Fail negation

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