Fibonacci numbers: (The good old) Prolog version:

fib(0, 0).  
fib(1, 1).  
fib(N, F) :-  
        N > 1,   
        N1 is N - 1,   
        N2 is N - 2,  
        fib(N1, F1),   
        fib(N2, F2),  
        F is F1 + F2.

Problem: can only be used with the first argument instantiated to a number:

?- fib(2, F).  
F = 3  
?- fib(N, 3).  
{ERROR: illegal arithmetic expression}

CLP version:

use\_module(library(clpfd)).

fib(0, 0).

fib(1, 1).

fib(N, F) :- N #> 1, N1 #= N - 1, N2 #= N - 2,

fib(N1, F1), fib(N2, F2),

F #= F1 + F2.

**2. A version of the [Zebra puzzle](https://en.wikipedia.org/wiki/Zebra_Puzzle), solution from [stackoverflow](http://stackoverflow.com/questions/11122814/solving-the-zebra-puzzle-aka-einstein-puzzle-using-the-clpfd-prolog-library?rq=1)**

There are five houses in a row, each with a different color. In each house lives one person with one pet, one favourite drink, and one favourite cigarette (sorry, old puzzle...). All pets, drinks, cigarette brands, and people's nationalities are different.

**Clues**:

The English person lives in the red house

The Swedish owns a dog

The Danish likes to drink tea

The green house is directly to the left of the white house

The owner of the green house drinks coffee

The person that smokes Pall Mall owns a bird

Milk is drunk in the middle house

The owner of the yellow house smokes Dunhill

The Norwegian lives in the first house

The marlboro smoker lives next to the cat owner

The horse owner lives next to the person who smokes dunhill

The winfield smoker likes to drink beer

The Norwegian lives next to the blue house

The German smokes rothmanns

The marlboro smoker has a neighbor who drinks water

**Question**: who has a fish as a pet?

CLP(FD) model:

Each attribute a house can have is modeled as a variable, e.g. "British", "Dog", "Green", etc. The attributes can take values from 1 to 5, depending on the house in which they occur, e.g. if the variable "Dog" takes the value 3, the dog lives in the third house.

Program:

:- use\_module(library(clpfd)).

neighbor(X, Y) :-

(X #= (Y - 1)) #\/ (X #= (Y + 1)).

solve([British, Swedish, Danish, Norwegian, German], Fish) :-

Nationalities = [British, Swedish, Danish, Norwegian, German],

Colors = [Red, Green, Blue, White, Yellow],

Beverages = [Tea, Coffee, Milk, Beer, Water],

Cigarettes = [PallMall, Marlboro, Dunhill, Winfield, Rothmanns],

Pets = [Dog, Bird, Cat, Horse, Fish],

all\_different(Nationalities),

all\_different(Colors),

all\_different(Beverages),

all\_different(Cigarettes),

all\_different(Pets),

Nationalities ins 1..5,

Colors ins 1..5,

Beverages ins 1..5,

Cigarettes ins 1..5,

Pets ins 1..5,

British #= Red, % Hint 1

Swedish #= Dog, % Hint 2

Danish #= Tea, % Hint 3

Green #= White - 1 , % Hint 4

Green #= Coffee, % Hint 5,

PallMall #= Bird, % Hint 6

Milk #= 3, % Hint 7

Yellow #= Dunhill, % Hint 8,

Norwegian #= 1, % Hint 9

neighbor(Marlboro, Cat), % Hint 10

neighbor(Horse, Dunhill), % Hint 11

Winfield #= Beer, % Hint 12

neighbor(Norwegian, Blue), % Hint 13

German #= Rothmanns, % Hint 14,

neighbor(Marlboro, Water). % Hint 15

Solve with:

?- X = [British, Swedish, Danish, Norwegian, German],  
 solve(X, Fish), label([Fish|X]).

**3. Alpha cipher**

[Solution](http://www.hakank.org/bprolog/crypto.pl) by Hakan Kjellerstrand

  The numbers from 1 to 26 are assigned to the letters of the alphabet. The number beside each word is the total of the values assigned to the letters in the word (e.g. for LYRE: L, Y, R, E might be equal to 5, 9, 20 and 13, or any other combination that adds up to 47.

Find the value of each letter with the word values:

  BALLET  45     GLEE  66     POLKA      59     SONG     61

  CELLO   43     JAZZ  58     QUARTET    50     SOPRANO  82

  CONCERT 74     LYRE  47     SAXOPHONE 134     THEME    72

  FLUTE   30     OBOE  53     SCALE      51     VIOLIN  100

  FUGUE   50     OPERA 65     SOLO       37     WALTZ    34

Also, there is exactly one letter missing from all the words. The true solution is the code of that letter ( \_D in the example below).

?- use\_module(library(clpfd)).

go:-

statistics(runtime,\_),

top,

statistics(runtime,[\_,Y]),

write('time : '), write(Y), nl.

top:-

alpha(LD),

write(LD), nl.

alpha(LD):-

LD=[A,B,C,\_D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z],

LD ins 1..26,

all\_different(LD),

B+A+L+L+E+T #= 45,

C+E+L+L+O #= 43,

C+O+N+C+E+R+T #= 74,

F+L+U+T+E #= 30,

F+U+G+U+E #= 50,

G+L+E+E #= 66,

J+A+Z+Z #= 58,

L+Y+R+E #= 47,

O+B+O+E #= 53,

O+P+E+R+A #= 65,

P+O+L+K+A #= 59,

Q+U+A+R+T+E+T #= 50,

S+A+X+O+P+H+O+N+E #= 134,

S+C+A+L+E #= 51,

S+O+L+O #= 37,

S+O+N+G #= 61,

S+O+P+R+A+N+O #= 82,

T+H+E+M+E #= 72,

V+I+O+L+I+N #= 100,

W+A+L+T+Z #= 34,

label(LD).

# More Prolog Examples for CMPUT 325

This is a page with some additional Prolog examples and explanations.

* Variations on count: how many elements are there in a list?
* Writing Some Lisp-like Functions in Prolog
* A "sorted" predicate in Prolog
* More fun with the plus(X,Y,Z) example
* DNF example
* maplist meta-predicate

## Counting Elements in a List

### A simple counting predicate

count(L,N): given a list L, return N, the number of elements in L.

#### Examples:

count([a,b,c,d],N).

N = 4

count([[a,b,c,d]],N).

N = 1

Comment: This is a list with a single element, which is again a list.

count([a,1,5],3).

yes

Comment: this predicate also works if both L and N are given.

Now try:

count(L,N).

#### Code:

count([],0).

count([F|R], N) :- count(R,N1), N is N1 + 1.

We don't need the value of the first element F, so we should write the second rule using an anonymous variable "\_":

count([],0).

count([\_|R], N) :- count(R,N1), N is N1 + 1.

Comment: this predicate does exactly the same as the built-in predicate length(L,N).

### Counting repeated elements only once

ucount(L,N) counts how many distinct elements are in list L. This "unique count" counts an element only once even if it appears several times in a list. ucount is implemented in terms of a helper predicate uc2(L,SoFar,N), which keeps track of all elements seen previously, and does not count them again.

#### Examples:

ucount([a,b,b,d,a],N).

N = 3

ucount([[a,b], a, b, [b,a]],N).

N = 4

Comment: This does not "look into" nested lists, so [a,b], a, b and [b,a] are all considered different elements of the list.

ucount([a,a,a],3).

no

ucount([a,a,a],1).

yes

Comment: this predicate also works if both L and N are given.

Also try:

ucount(L,0).

ucount(L,1).

ucount(L,2).

ucount(L,N).

Which of these work, and which fail? Why? Can you improve ucount using CLP(FD)?

#### Code:

ucount(L,N) :- uc2(L,[],N). % no elements seen so far.

uc2([],\_,0).

uc2([F|R], SoFar, N) :- member(F,SoFar), uc2(R, SoFar, N). % F already counted earlier.

uc2([F|R], SoFar, N) :- not\_member(F,SoFar), uc2(R, [F|SoFar], N1), N is N1+1. % F is new.

member(X,[X|\_]).

member(X,[\_|L]) :- member(X,L).

not\_member(X,[]).

not\_member(X, [Y|L]) :- X \== Y, not\_member(X,L).

## Writing Some Lisp-like Functions in Prolog

To illustrate the different way of thinking in Prolog, we rewrite some of our Lisp functions in Prolog.

### Null, Empty List

null([]). % just this fact, which matches only the empty list.

Here are two slightly different rules for testing nonempty lists:

nonempty([\_|\_]). % this matches any list that has a first element and a rest.

nonempty2(X) :- not(null(X)). % this matches any X that does not unify with the empty list.

#### Examples:

null([]).

yes

null(5).

no

null(X).

X = []

Comment: you can give it a variable, and it will find a solution. A big difference to Lisp...

null([X]).

no

Comment: [X] does not match an empty list. There is no substitution for X that would make this work. You cannot substitute X by "nothing".

nonempty([]).

no

nonempty([[]]).

yes

nonempty(5).

no

Comment: 5 is not a list

nonempty([5]).

yes

nonempty(X).

X = [\_3961|\_3962]

Comment: this is interesting. It returns the general answer of what a nonempty list looks like. You can substitute anything you want for the two variables \_3961 and \_3962, the result will always be a nonempty list.

nonempty2([]).

no

nonempty2([[]]).

yes

nonempty2(5).

yes

Comment: This is the case where nonempty2 is different from nonempty. null(5) fails, so nonempty2(5) succeeds.

nonempty2(X).

no

Comment: another case where they behave differently. null(X) succeeds with X=[], so nonempty2(X) fails. This is sad, because it basically says that there are no nonempty lists. Another example why we need to be careful with cut and not.

#### Remark:

Q: Why don't we simply use

nonempty([\_]).

as a definition?   
A: Because it does not work. As an explanation, see what happens when we unify these two forms:

[X] = [F|R].

X = \_4670

F = \_4670

R = []

So [\_] only matches lists with a single element. E.g. trying to unify [\_] = [a,b] fails.

### "Sorted" in Prolog

Here is a predicate sorted(L) that tests whether a simple, not nested list of numbers is sorted.

#### A "Lisp-like" implementation:

sorted(X) :- null(X).

sorted(X) :- X = [F|Rest], null(Rest).

sorted(X) :- X = [A,B|Rest], A =< B, sorted([B|Rest]).

#### The same in a more "Prolog-like" implementation:

sorted([]).

sorted([\_]).

sorted(A,B|Rest]) :- A =< B, sorted([B|Rest]).

## More fun with the "plus" predicate

Remember the definition of plus:

plus(0, X, X).

plus(s(X), Y, s(Z)) :- plus(X,Y,Z).

This predicate can be used in many ways. For example, try:

plus(X,X,Y).

This will find all solutions to the equation X+X=Y, for example:

X = 0

Y = 0 ;

X = s(0)

Y = s(s(0)) ;

X = s(s(0))

Y = s(s(s(s(0)))) ;

X = s(s(s(0)))

Y = s(s(s(s(s(s(0)))))) ;

X = s(s(s(s(0))))

Y = s(s(s(s(s(s(s(s(0))))))))

and so on.

Try to understand how this works, it is a good exercise in backtracking and unification. You can also trace the program in the debugger and see what happens.

If you try the query

plus(X,Y,Z).

something surprising happens: the answers you get contain variables, as in:

X = 0

Y = \_1299

Z = \_1299 ;

X = s(0)

Y = \_1299

Z = s(\_1299) ;

X = s(s(0))

Y = \_1299

Z = s(s(\_1299)) ;

X = s(s(s(0)))

Y = \_1299

Z = s(s(s(\_1299)))

So instead of returning fully instantiated solutions with three numbers, Prolog returns equations equivalent to:

0+X=X

1+X = (X+1)

2+X = (X+2)

and so on.

We actually have to tell Prolog that we want numbers for an answer! One way to do this is to define a predicate that generates all numbers, and demand that the sum Z is a number.

anumber(0).

anumber(s(X)) :- anumber(X).

generatesolutions(X,Y,Z) :- anumber(Z), plus(X,Y,Z).

This will find all solutions for X+Y=0 first, then X+Y=1, X+Y=2, and so on.

generatesolutions(X,Y,Z).

X = 0

Y = 0

Z = 0 ;

X = 0

Y = s(0)

Z = s(0) ;

X = s(0)

Y = 0

Z = s(0) ;

X = 0

Y = s(s(0))

Z = s(s(0)) ;

X = s(0)

Y = s(0)

Z = s(s(0)) ;

X = s(s(0))

Y = 0

Z = s(s(0)) ;

X = 0

Y = s(s(s(0)))

Z = s(s(s(0))) ;

X = s(0)

Y = s(s(0))

Z = s(s(s(0)))

and so on

## DNF

Disjunctive normal form (DNF) is a disjunction of conjunctions of literals; that is,

(l1 & l2 & ... & lk) v (m1 & ...& mj) v ... v (w1 & w2 & ... & wn)

where each conjunction is of the form

l1 & l2 & ... & lk

and each li is a literal: a proposition p, or a negated proposition -p.

Each propositional formula can be converted to DNF. Here are some of the rules for doing that:

-(-F) ==> F

-(F1 v F2) ==> -F1 & -F2

-(F1 & F2) ==> -F1 v -F2

F1 <= F2 ==> F1 v -F2

F1 & (F2 v F3) ==> (F1 & F2) v (F1 & F3)

(F1 v F2) & F3 ==> (F1 & F3) v (F2 & F3)

The same rules in Prolog look like this:

dnf(-(-F), A)

:- dnf(F, A), !.

dnf(-(F1 & F2), C1) :-

dnf(F1, A1), dnf(F2, A2),

dnf(-A1 v -A2, B1),

dnf(B1, C1), !.

dnf(-(F1 v F2), C1) :-

dnf(F1, A1), dnf(F2, A2),

dnf(-A1 & -A2, B1),

dnf(B1, C1), !.

dnf((F1 <= F2), B1) :-

dnf(F1, A1), dnf(F2, A2),

dnf(A1 v -A2, B1), !.

dnf(F1 & (F2 v F3), C1) :-

dnf(F1, A1), dnf(F2, A2), dnf(F3, A3),

dnf(A1 & A2, B1), dnf(A1 & A3, B2),

dnf(B1 v B2, C1), !.

dnf((F1 v F2) & F3, C1) :-

dnf(F1, A1), dnf(F2, A2), dnf(F3, A3),

dnf(A1 & A3, B1), dnf(A2 & A3, B2),

dnf(B1 v B2, C1), !.

These are the basic operations. A full implementation of dnf needs quite a few more rules for simplifying expressions such as a&a, a&-a, or a&c&a, and also a "catchall" rule for formulas that cannot be further simplified.

## Maplist and other Meta-predicates

A meta-predicate takes other predicates as arguments. One example we have seen is findall.

maplist is a bit similar to mapcar in Lisp.   
maplist(Goal, List) tries to apply the predicate Goal to all elements in List, and it succeeds if all these goals succeed. Example from http://www.swi-prolog.org/pldoc/man?section=apply

?- maplist(integer, [1, 2, 3]).

true.

?- maplist(integer, [1, 2.2, 3]).

false.

maplist can also be used with partially instantiated predicates, similar to how we can evaluate functions in Lisp where not all arguments are given by using higher order functions. See the example under same\_length below.

foldl is similar to the Lisp reduce with an identity element. Example:

?- foldl(plus, [1, 2, 3, 4, 5], 0, Result).

Result = 15.

same\_length(L1,L2) succeeds if L1 and L2 are (or can be made) lists of the same length. Example from sudoku.pl:

length(Rows, 9), maplist(same\_length(Rows), Rows)

This applies same\_length(Rows,X) to every element X in Rows. Compare with:

length(Rows, 9), maplist(same\_length([\_,\_,\_]),Rows).