Logic Programming Problems

Part I: problem collection:

The purpose of this problem collection is to give you the opportunity to practice your skills in logic programming. Your goal should be to find the most elegant solution of the given problems. Efficiency is important, but logical clarity is even more crucial. Some of the (easy) problems can be trivially solved using built-in predicates. However, in these cases, you learn more if you try to find your own solution.

Every predicate that you write should begin with a comment that describes the predicate in a *declarative* statement. Do *not* describe procedurally, *what* the predicate does, but write down a *logical statement* which includes the arguments of the predicate. You should also indicate the intended data types of the arguments and the allowed flow patterns.

The problems have different levels of difficulty. Those marked with a single asterisk (*) are easy. If you have successfully solved the preceding problems you should be able to solve them within a few (say 15) minutes. Problems marked with two asterisks (**) are of intermediate difficulty. If you are a skilled Prolog programmer it shouldn't take you more than 30-60 minutes to solve them. Problems marked with three asterisks (***) are more difficult. You may need more time (i.e. a few hours) to find a good solution.

Working with Prolog lists

A list is either empty or it is composed of a first element (head) and a tail, which is a list itself. In Prolog we represent the empty list by the atom [] and a non-empty list by a term [H|T] where H denotes the head and T denotes the tail.

P01 (*) Find the last element of a list.

Example:
?- my_last(X,[a,b,c,d]).
X = d

```
% P01 (*): Find the last element of a list

% my_last(X,L) :- X is the last element of the list L
%    (element,list) (?,?)

% Note: last(?Elem, ?List) is predefined

my_last(X,[X]).
my_last(X,[_|L]) :- my_last(X,L).
```

P02 (*) Find the last but one element of a list.
(zweitletztes Element, l'avant-dernier élément)

```
% P02 (*): Find the last but one element of a list

% last_but_one(X,L) :- X is the last but one element of the list L
%    (element,list) (?,?)

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

P03 (*) Find the K'th element of a list.
The first element in the list is number 1.
Example:
?- element_at(X,[a,b,c,d,e],3).
X = c

```
% P03 (*): Find the K'th element of a list.
% The first element in the list is number 1.

% element_at(X,L,K) :- X is the K'th element of the list L
%    (element,list,integer) (?,?,+)

% Note: nth1(?Index, ?List, ?Elem) is predefined

element_at(X,[X|_],1).
element_at(X,[_|L],K) :- K > 1, K1 is K - 1, element_at(X,L,K1).
```

P04 (*) Find the number of elements of a list.

```
% P04 (*): Find the number of elements of a list.

% my_length(L,N) :- the list L contains N elements
%    (list,integer) (+,?)

% Note: length(?List, ?Int) is predefined

my_length([],0).
my_length([_|L],N) :- my_length(L,N1), N is N1 + 1.
```

P05 (*) Reverse a list.

```
% P05 (*): Reverse a list.

% my_reverse(L1,L2) :- L2 is the list obtained from L1 by reversing
%    the order of the elements.
%    (list,list) (?,?)

% Note: reverse(+List1, -List2) is predefined
```

```
my_reverse(L1,L2) :- my_rev(L1,L2,[]).

my_rev([],L2,L2) :- !.
my_rev([X|Xs],L2,Acc) :- my_rev(Xs,L2,[X|Acc]).
```

P06 (*) Find out whether a list is a palindrome.

A palindrome can be read forward or backward; e.g. [x,a,m,a,x].

```
% P06 (*): Find out whether a list is a palindrome
% A palindrome can be read forward or backward; e.g. [x,a,m,a,x]

% is_palindrome(L) :- L is a palindrome list
%    (list) (?)

is_palindrome(L) :- reverse(L,L).
```

P07 (**) Flatten a nested list structure.

Transform a list, possibly holding lists as elements into a `flat' list by replacing each list with its elements (recursively).

Example:
?- my_flatten([a, [b, [c, d], e]], X).
X = [a, b, c, d, e]
Hint: Use the predefined predicates is_list/1 and append/3

```
% P07 (**): Flatten a nested list structure.

% my_flatten(L1,L2) :- the list L2 is obtained from the list L1 by
%    flattening; i.e. if an element of L1 is a list then it is replaced
%    by its elements, recursively.
%    (list,list) (+,?)

% Note: flatten(+List1, -List2) is a predefined predicate

my_flatten(X,[X]) :- \+ is_list(X).
my_flatten([],[]).
my_flatten([X|Xs],Zs) :- my_flatten(X,Y), my_flatten(Xs,Ys), append(Y,Ys,Zs).
```

P08 (**) Eliminate consecutive duplicates of list elements.

If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.
Example:
?- compress([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [a,b,c,a,d,e]

```
% P08 (**): Eliminate consecutive duplicates of list elements.

% compress(L1,L2) :- the list L2 is obtained from the list L1 by
%    compressing repeated occurrences of elements into a single copy
%    of the element.
%    (list,list) (+,?)

compress([],[]).
compress([X],[X]).
compress([X,X|Xs],Zs) :- compress([X|Xs],Zs).
compress([X,Y|Ys],[X|Zs]) :- X \= Y, compress([Y|Ys],Zs).
```

P09 (**) Pack consecutive duplicates of list elements into sublists.

If a list contains repeated elements they should be placed in separate sublists.

Example:
?- pack([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[a,a,a,a],[b],[c,c],[a,a],[d],[e,e,e,e]]

```
% P09 (**):  Pack consecutive duplicates of list elements into sublists.

% pack(L1,L2) :- the list L2 is obtained from the list L1 by packing
%    repeated occurrences of elements into separate sublists.
%    (list,list) (+,?)

pack([],[]).
pack([X|Xs],[Z|Zs]) :- transfer(X,Xs,Ys,Z), pack(Ys,Zs).

% transfer(X,Xs,Ys,Z) Ys is the list that remains from the list Xs
%    when all leading copies of X are removed and transfered to Z

transfer(X,[],[],[X]).
transfer(X,[Y|Ys],[Y|Ys],[X]) :- X \= Y.
transfer(X,[X|Xs],Ys,[X|Zs]) :- transfer(X,Xs,Ys,Zs).
```

P10 (*) Run-length encoding of a list.

Use the result of problem P09 to implement the so-called run-length encoding data compression method. Consecutive duplicates of elements are encoded as terms [N,E] where N is the number of duplicates of the element E.

Example:
?- encode([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4,a],[1,b],[2,c],[2,a],[1,d][4,e]]

```
% P10 (*):  Run-length encoding of a list

% encode(L1,L2) :- the list L2 is obtained from the list L1 by run-length
%    encoding. Consecutive duplicates of elements are encoded as terms [N,E],
%    where N is the number of duplicates of the element E.
%    (list,list) (+,?)

:- ensure_loaded(p09).

encode(L1,L2) :- pack(L1,L), transform(L,L2).

transform([],[]).
transform([[X|Xs]|Ys],[[N,X]|Zs]) :- length([X|Xs],N), transform(Ys,Zs).
```

P11 (*) Modified run-length encoding.

Modify the result of problem P10 in such a way that if an element has no duplicates it is simply copied into the result list. Only elements with duplicates are transferred as [N,E] terms.

Example:
?- encode_modified([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4,a],b,[2,c],[2,a],d,[4,e]]

```
% P11 (*):  Modified run-length encoding

% encode_modified(L1,L2) :- the list L2 is obtained from the list L1 by
%    run-length encoding. Consecutive duplicates of elements are encoded
%    as terms [N,E], where N is the number of duplicates of the element E.
%    However, if N equals 1 then the element is simply copied into the
%    output list.
%    (list,list) (+,?)

:- ensure_loaded(p10).

encode_modified(L1,L2) :- encode(L1,L), strip(L,L2).

strip([],[]).
strip([[1,X]|Ys],[X|Zs]) :- strip(Ys,Zs).
strip([[N,X]|Ys],[[N,X]|Zs]) :- N > 1, strip(Ys,Zs).
```

P12 (**) Decode a run-length encoded list.

Given a run-length code list generated as specified in problem P11. Construct its uncompressed version.

```
% P12 (**): Decode a run-length compressed list.

% decode(L1,L2) :- L2 is the uncompressed version of the run-length
%    encoded list L1.
%    (list,list) (+,?)

decode([],[]).
decode([X|Ys],[X|Zs]) :- \+ is_list(X), decode(Ys,Zs).
decode([[1,X]|Ys],[X|Zs]) :- decode(Ys,Zs).
decode([[N,X]|Ys],[X|Zs]) :- N > 1, N1 is N - 1, decode([[N1,X]|Ys],Zs).
```

P13 (**) Run-length encoding of a list (direct solution).

Implement the so-called run-length encoding data compression method directly. I.e. don't explicitly create the sublists containing the duplicates, as in problem P09, but only count them. As in problem P11, simplify the result list by replacing the singleton terms [1,X] by X.

Example:
?- encode_direct([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4,a],b,[2,c],[2,a],d,[4,e]]

```
% P13 (**): Run-length encoding of a list (direct solution)

% encode_direct(L1,L2) :- the list L2 is obtained from the list L1 by
%    run-length encoding. Consecutive duplicates of elements are encoded
%    as terms [N,E], where N is the number of duplicates of the element E.
%    However, if N equals 1 then the element is simply copied into the
%    output list.
%    (list,list) (+,?)

encode_direct([],[]).
encode_direct([X|Xs],[Z|Zs]) :- count(X,Xs,Ys,1,Z), encode_direct(Ys,Zs).

% count(X,Xs,Ys,K,T) Ys is the list that remains from the list Xs
%    when all leading copies of X are removed. T is the term [N,X],
%    where N is K plus the number of X's that can be removed from Xs.
%    In the case of N=1, T is X, instead of the term [1,X].

count(X,[],[],1,X).
count(X,[],[],N,[N,X]) :- N > 1.
count(X,[Y|Ys],[Y|Ys],1,X) :- X \= Y.
count(X,[Y|Ys],[Y|Ys],N,[N,X]) :- N > 1, X \= Y.
count(X,[X|Xs],Ys,K,T) :- K1 is K + 1, count(X,Xs,Ys,K1,T).
```

P14 (*) Duplicate the elements of a list.

Example:
?- dupli([a,b,c,c,d],X).
X = [a,a,b,b,c,c,c,c,d,d]

```
% P14 (*): Duplicate the elements of a list

% dupli(L1,L2) :- L2 is obtained from L1 by duplicating all elements.
%    (list,list) (?,?)

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

P15 (**) Duplicate the elements of a list a given number of times.

Example:
?- dupli([a,b,c],3,X).
X = [a,a,a,b,b,b,c,c,c]

What are the results of the goal:
?- dupli(X,3,Y).

```
% P15 (**): Duplicate the elements of a list agiven number of times

% dupli(L1,N,L2) :- L2 is obtained from L1 by duplicating all elements
%    N times.
%    (list,integer,list) (?,+,?)

dupli(L1,N,L2) :- dupli(L1,N,L2,N).

% dupli(L1,N,L2,K) :- L2 is obtained from L1 by duplicating its leading
%    element K times, all other elements N times.
%    (list,integer,list,integer) (?,+,?,+)

dupli([],_,[],_).
dupli([_|Xs],N,Ys,0) :- dupli(Xs,N,Ys,N).
dupli([X|Xs],N,[X|Ys],K) :- K > 0, K1 is K - 1, dupli([X|Xs],N,Ys,K1).
```

P16 (**) Drop every N'th element from a list.

Example:
?- drop([a,b,c,d,e,f,g,h,i,k],3,X).
X = [a,b,d,e,g,h,k]

```
% P16 (**):  Drop every N'th element from a list

% drop(L1,N,L2) :- L2 is obtained from L1 by dropping every N'th element.
%    (list,integer,list) (?,+,?)

drop(L1,N,L2) :- drop(L1,N,L2,N).

% drop(L1,N,L2,K) :- L2 is obtained from L1 by first copying K-1 elements
%    and then dropping an element and, from then on, dropping every
%    N'th element.
%    (list,integer,list,integer) (?,+,?,+)

drop([],_,[],_).
drop([_|Xs],N,Ys,1) :- drop(Xs,N,Ys,N).
drop([X|Xs],N,[X|Ys],K) :- K > 1, K1 is K - 1, drop(Xs,N,Ys,K1).
```

P17 (*) Split a list into two parts; the length of the first part is given.

Do not use any predefined predicates.

Example:
?- split([a,b,c,d,e,f,g,h,i,k],3,L1,L2).
L1 = [a,b,c]
L2 = [d,e,f,g,h,i,k]

```
% P17 (*): Split a list into two parts

% split(L,N,L1,L2) :- the list L1 contains the first N elements
%    of the list L, the list L2 contains the remaining elements.
%    (list,integer,list,list) (?,+,?,?)

split(L,0,[],L).
split([X|Xs],N,[X|Ys],Zs) :- N > 0, N1 is N - 1, split(Xs,N1,Ys,Zs).
```

P18 (**) Extract a slice from a list.

Given two indices, I and K, the slice is the list containing the elements between the I'th and K'th element of the original list (both limits included). Start counting the elements with 1.

Example:

?- slice([a,b,c,d,e,f,g,h,i,k],3,7,L).
X = [c,d,e,f,g]

```
% P18 (**):  Extract a slice from a list

% slice(L1,I,K,L2) :- L2 is the list of the elements of L1 between
%    index I and index K (both included).
%    (list,integer,integer,list) (?,+,+,?)

slice([X|_],1,1,[X]).
slice([X|Xs],1,K,[X|Ys]) :- K > 1, 
   K1 is K - 1, slice(Xs,1,K1,Ys).
slice([_|Xs],I,K,Ys) :- I > 1, 
   I1 is I - 1, K1 is K - 1, slice(Xs,I1,K1,Ys).
```

P19 (**) Rotate a list N places to the left.

Examples:
?- rotate([a,b,c,d,e,f,g,h],3,X).
X = [d,e,f,g,h,a,b,c]

?- rotate([a,b,c,d,e,f,g,h],-2,X).
X = [g,h,a,b,c,d,e,f]

Hint: Use the predefined predicates length/2 and append/3, as well as the result of problem P17.

```
% P19 (**): Rotate a list N places to the left 

% rotate(L1,N,L2) :- the list L2 is obtained from the list L1 by 
%    rotating the elements of L1 N places to the left.
%    Examples: 
%    rotate([a,b,c,d,e,f,g,h],3,[d,e,f,g,h,a,b,c])
%    rotate([a,b,c,d,e,f,g,h],-2,[g,h,a,b,c,d,e,f])
%    (list,integer,list) (+,+,?)

:- ensure_loaded(p17).

rotate(L1,N,L2) :- N >= 0, 
   length(L1,NL1), N1 is N mod NL1, rotate_left(L1,N1,L2).
rotate(L1,N,L2) :- N < 0,
   length(L1,NL1), N1 is NL1 + (N mod NL1), rotate_left(L1,N1,L2).

rotate_left(L,0,L).
rotate_left(L1,N,L2) :- N > 0, split(L1,N,S1,S2), append(S2,S1,L2).
```

P20 (*) Remove the K'th element from a list.

Example:
?- remove_at(X,[a,b,c,d],2,R).
X = b
R = [a,c,d]

```
% P20 (*): Remove the K'th element from a list.
% The first element in the list is number 1.

% remove_at(X,L,K,R) :- X is the K'th element of the list L; R is the
%    list that remains when the K'th element is removed from L.
%    (element,list,integer,list) (?,?,+,?)

remove_at(X,[X|Xs],1,Xs).
remove_at(X,[Y|Xs],K,[Y|Ys]) :- K > 1,
   K1 is K - 1, remove_at(X,Xs,K1,Ys).
```

P21 (*) Insert an element at a given position into a list.

Example:
?- insert_at(alfa,[a,b,c,d],2,L).
L = [a,alfa,b,c,d]

```
% P22 (*):  Create a list containing all integers within a given range.

% range(I,K,L) :- I <= K, and L is the list containing all
%    consecutive integers from I to K.
%    (integer,integer,list) (+,+,?)

range(I,I,[I]).
range(I,K,[I|L]) :- I < K, I1 is I + 1, range(I1,K,L).
```

P22 (*) Create a list containing all integers within a given range.

Example:
?- range(4,9,L).
L = [4,5,6,7,8,9]

```
% P22 (*):  Create a list containing all integers within a given range.

% range(I,K,L) :- I <= K, and L is the list containing all
%    consecutive integers from I to K.
%    (integer,integer,list) (+,+,?)

range(I,I,[I]).
range(I,K,[I|L]) :- I < K, I1 is I + 1, range(I1,K,L).
```

P23 (**) Extract a given number of randomly selected elements from a list.

The selected items shall be put into a result list.
Example:
?- rnd_select([a,b,c,d,e,f,g,h],3,L).
L = [e,d,a]

Hint: Use the built-in random number generator random/2 and the result of problem P20.

```
% P23 (**): Extract a given number of randomly selected elements
%    from a list.

% rnd_select(L,N,R) :- the list R contains N randomly selected
%    items taken from the list L.
%    (list,integer,list) (+,+,-)
```

```
:- ensure_loaded(p20).

rnd_select(_,0,[]).
rnd_select(Xs,N,[X|Zs]) :- N > 0,
    length(Xs,L),
    I is random(L) + 1,
    remove_at(X,Xs,I,Ys),
    N1 is N - 1,
    rnd_select(Ys,N1,Zs).
```

P24 (*) Lotto: Draw N different random numbers from the set 1..M.

The selected numbers shall be put into a result list.
Example:
?- rnd_select(6,49,L).
L = [23,1,17,33,21,37]

Hint: Combine the solutions of problems P22 and P23.

```
% P24 (*): Lotto: Draw N different random numbers from the set 1..M

% lotto(N,M,L) :- the list L contains N randomly selected distinct
%    integer numbers from the interval 1..M
%    (integer,integer,number-list) (+,+,-)

:- ensure_loaded(p22).
:- ensure_loaded(p23).

lotto(N,M,L) :- range(1,M,R), rnd_select(R,N,L).
```

P25 (*) Generate a random permutation of the elements of a list.

Example:
?- rnd_permu([a,b,c,d,e,f],L).
L = [b,a,d,c,e,f]

Hint: Use the solution of problem P23.

```
% P25 (*):  Generate a random permutation of the elements of a list

% rnd_permu(L1,L2) :- the list L2 is a random permutation of the
%    elements of the list L1.
%    (list,list) (+,-)

:- ensure_loaded(p23).

rnd_permu(L1,L2) :- length(L1,N), rnd_select(L1,N,L2).
```

P26 (**) Generate the combinations of K distinct objects chosen from the N elements of a list

In how many ways can a committee of 3 be chosen from a group of 12 people? We all know that there are C(12,3) = 220 possibilities (C(N,K) denotes the well-known binomial coefficients). For pure mathematicians, this result may be great. But *we* want to really generate all the possibilities (via backtracking).

Example:
?- combination(3,[a,b,c,d,e,f],L).
L = [a,b,c] ;
L = [a,b,d] ;
L = [a,b,e] ;
...

```
% P26 (**):  Generate the combinations of k distinct objects
%            chosen from the n elements of a list.

% combination(K,L,C) :- C is a list of K distinct elements
%    chosen from the list L

combination(0,_,[]).
combination(K,L,[X|Xs]) :- K > 0,
   el(X,L,R), K1 is K-1, combination(K1,R,Xs).

% Find out what the following predicate el/3 exactly does.

el(X,[X|L],L).
el(X,[_|L],R) :- el(X,L,R).
```

P27 (**) Group the elements of a set into disjoint subsets.

a) In how many ways can a group of 9 people work in 3 disjoint subgroups of 2, 3 and 4 persons? Write a predicate that generates all the possibilities via backtracking.

Example:
?- group3([aldo,beat,carla,david,evi,flip,gary,hugo,ida],G1,G2,G3).
G1 = [aldo,beat], G2 = [carla,david,evi], G3 = [flip,gary,hugo,ida]
...

b) Generalize the above predicate in a way that we can specify a list of group sizes and the predicate will return a list of groups.

Example:
?- group([aldo,beat,carla,david,evi,flip,gary,hugo,ida],[2,2,5],Gs].
Gs = [[aldo,beat],[carla,david],[evi,flip,gary,hugo,ida]]
...

Note that we do not want permutations of the group members; i.e. [[aldo,beat],...] is the same solution as [[beat,aldo],...]. However, we make a difference between [[aldo,beat],[carla,david],...] and [[carla,david],[aldo,beat],...].

You may find more about this combinatorial problem in a good book on discrete mathematics under the term "multinomial coefficients".

```
% P27 (**) Group the elements of a set into disjoint subsets.
% Problem a)

% group3(G,G1,G2,G3) :- distribute the 9 elements of G into G1, G2, and G3,
%    such that G1, G2 and G3 contain 2,3 and 4 elements respectively

group3(G,G1,G2,G3) :-
   selectN(2,G,G1),
   subtract(G,G1,R1),
   selectN(3,R1,G2),
   subtract(R1,G2,R2),
   selectN(4,R2,G3),
   subtract(R2,G3,[]).

% selectN(N,L,S) :- select N elements of the list L and put them in
%    the set S. Via backtracking return all posssible selections, but
%    avoid permutations; i.e. after generating S = [a,b,c] do not return
%    S = [b,a,c], etc.

selectN(0,_,[]) :- !.
selectN(N,L,[X|S]) :- N > 0,
   el(X,L,R),
   N1 is N-1,
   selectN(N1,R,S).

el(X,[X|L],L).
el(X,[_|L],R) :- el(X,L,R).

% subtract/3 is predefined

% Problem b): Generalization

% group(G,Ns,Gs) :- distribute the elements of G into the groups Gs.
%    The group sizes are given in the list Ns.

group([],[],[]).
group(G,[N1|Ns],[G1|Gs]) :-
   selectN(N1,G,G1),
   subtract(G,G1,R),
   group(R,Ns,Gs).
```

P28 (**) Sorting a list of lists according to length of sublists

a) We suppose that a list (InList) contains elements that are lists themselves. The objective is to sort the elements of InList according to their **length**. E.g. short lists first, longer lists later, or vice versa.

Example:
?- lsort([[a,b,c],[d,e],[f,g,h],[d,e],[i,j,k,l],[m,n],[o]],L).
L = [[o], [d, e], [d, e], [m, n], [a, b, c], [f, g, h], [i, j, k, l]]

b) Again, we suppose that a list (InList) contains elements that are lists themselves. But this time the objective is to sort the elements of InList according to their **length frequency**; i.e. in the default, where sorting is done ascendingly, lists with rare lengths are placed first, others with a more frequent length come later.

Example:
?- lfsort([[a,b,c],[d,e],[f,g,h],[d,e],[i,j,k,l],[m,n],[o]],L).
L = [[i, j, k, l], [o], [a, b, c], [f, g, h], [d, e], [d, e], [m, n]]

Note that in the above example, the first two lists in the result L have length 4 and 1, both lengths appear just once. The third and forth list have length 3 which appears, there are two list of this length. And finally, the last three lists have length 2. This is the most frequent length.

```
% P28 (**) Sorting a list of lists according to length
%
% a) length sort
%
% lsort(InList,OutList) :- it is supposed that the elements of InList
% are lists themselves. Then OutList is obtained from InList by sorting
% its elements according to their length. lsort/2 sorts ascendingly,
% lsort/3 allows for ascending or descending sorts.
% (list_of_lists,list_of_lists), (+,?)

lsort(InList,OutList) :- lsort(InList,OutList,asc).

% sorting direction Dir is either asc or desc

lsort(InList,OutList,Dir) :-
   add_key(InList,KList,Dir),
   keysort(KList,SKList),
   rem_key(SKList,OutList).

add_key([],[],_).
add_key([X|Xs],[L-p(X)|Ys],asc) :- !,
        length(X,L), add_key(Xs,Ys,asc).
add_key([X|Xs],[L-p(X)|Ys],desc) :-
        length(X,L1), L is -L1, add_key(Xs,Ys,desc).

rem_key([],[]).
rem_key([_-p(X)|Xs],[X|Ys]) :- rem_key(Xs,Ys).

% b) length frequency sort
%
% lfsort (InList,OutList) :- it is supposed that the elements of InList
% are lists themselves. Then OutList is obtained from InList by sorting
% its elements according to their length frequency; i.e. in the default,
% where sorting is done ascendingly, lists with rare lengths are placed
% first, other with more frequent lengths come later.
%
% Example:
% ?- lfsort([[a,b,c],[d,e],[f,g,h],[d,e],[i,j,k,l],[m,n],[o]],L).
% L = [[i, j, k, l], [o], [a, b, c], [f, g, h], [d, e], [d, e], [m, n]]
%
% Note that the first two lists in the Result have length 4 and 1, both
% length appear just once. The third and forth list have length 3 which
% appears, there are two list of this length. And finally, the last
% three lists have length 2. This is the most frequent length.

lfsort(InList,OutList) :- lfsort(InList,OutList,asc).
```

```
% sorting direction Dir is either asc or desc

lfsort(InList,OutList,Dir) :-
       add_key(InList,KList,desc),
   keysort(KList,SKList),
   pack(SKList,PKList),
   lsort(PKList,SPKList,Dir),
   flatten(SPKList,FKList),
   rem_key(FKList,OutList).

pack([],[]).
pack([L-X|Xs],[[L-X|Z]|Zs]) :- transf(L-X,Xs,Ys,Z), pack(Ys,Zs).

% transf(L-X,Xs,Ys,Z) Ys is the list that remains from the list Xs
%    when all leading copies of length L are removed and transfed to Z

transf(_,[],[],[]).
transf(L-_,[K-Y|Ys],[K-Y|Ys],[]) :- L \= K.
transf(L-_,[L-X|Xs],Ys,[L-X|Zs]) :- transf(L-X,Xs,Ys,Zs).

test :-
   L = [[a,b,c],[d,e],[f,g,h],[d,e],[i,j,k,l],[m,n],[o]],
   write('L = '), write(L), nl,
   lsort(L,LS),
   write('LS = '), write(LS), nl,
   lsort(L,LSD,desc),
   write('LSD = '), write(LSD), nl,
   lfsort(L,LFS),
   write('LFS = '), write(LFS), nl.
```

Arithmetic

P31 (**) Determine whether a given integer number is prime.

Example:
?- is_prime(7).
Yes

```
% P31 (**) Determine whether a given integer number is prime.

% is_prime(P) :- P is a prime number
%    (integer) (+)

is_prime(2).
is_prime(3).
is_prime(P) :- integer(P), P > 3, P mod 2 =\= 0, \+ has_factor(P,3).

% has_factor(N,L) :- N has an odd factor F >= L.
%    (integer, integer) (+,+)

has_factor(N,L) :- N mod L =:= 0.
has_factor(N,L) :- L * L < N, L2 is L + 2, has_factor(N,L2).
```

P32 (**) Determine the greatest common divisor of two positive integer numbers.

Use Euclid's algorithm.
Example:
?- gcd(36, 63, G).
G = 9

Define gcd as an arithmetic function; so you can use it like this:
?- G is gcd(36,63).
G = 9

```
% P32 (**) Determine the greatest common divisor of two positive integers.

% gcd(X,Y,G) :- G is the greatest common divisor of X and Y
%    (integer, integer, integer) (+,+,?)


gcd(X,0,X) :- X > 0.
gcd(X,Y,G) :- Y > 0, Z is X mod Y, gcd(Y,Z,G).


% Declare gcd as an arithmetic function; so you can use it
% like this:  ?- G is gcd(36,63).

:- arithmetic_function(gcd/2).
```

P33 (*) Determine whether two positive integer numbers are coprime.

Two numbers are coprime if their greatest common divisor equals 1.
Example:
?- coprime(35, 64).
Yes

```
% P33 (*) Determine whether two positive integer numbers are coprime.
%     Two numbers are coprime if their greatest common divisor equals 1.

% coprime(X,Y) :- X and Y are coprime.
%    (integer, integer) (+,+)

:- ensure_loaded(p32).

coprime(X,Y) :- gcd(X,Y,1).
```

P34 (**) Calculate Euler's totient function phi(m).

Euler's so-called totient function phi(m) is defined as the number of positive integers r (1 <= r < m) that are coprime to m.

Example: m = 10: r = 1,3,7,9; thus phi(m) = 4. Note the special case: phi(1) = 1.

?- Phi is totient_phi(10).
Phi = 4

Find out what the value of phi(m) is if m is a prime number. Euler's totient function plays an important role in one of the most widely used public key cryptography methods (RSA). In this exercise you should use the most primitive method to calculate this function (there are smarter ways that we shall discuss later).

```
% P34 (**) Calculate Euler's totient function phi(m).
%    Euler's so-called totient function phi(m) is defined as the number
%    of positive integers r (1 <= r < m) that are coprime to m.
%    Example: m = 10: r = 1,3,7,9; thus phi(m) = 4. Note: phi(1) = 1.

% totient_phi(M,Phi) :- Phi is the value of the Euler's totient function
%    phi for the argument M.
%    (integer, integer) (+,-)

:- arithmetic_function(totient_phi/1).

totient_phi(1,1) :- !.
totient_phi(M,Phi) :- t_phi(M,Phi,1,0).

% t_phi(M,Phi,K,C) :- Phi = C + N, where N is the number of integers R
%    such that K <= R < M and R is coprime to M.
%    (integer,integer,integer,integer) (+,-,+,+)

t_phi(M,Phi,M,Phi) :- !.
t_phi(M,Phi,K,C) :-
   K < M, coprime(K,M), !,
   C1 is C + 1, K1 is K + 1,
   t_phi(M,Phi,K1,C1).
t_phi(M,Phi,K,C) :-
   K < M, K1 is K + 1,
   t_phi(M,Phi,K1,C).
```

P35 (**) Determine the prime factors of a given positive integer.

Construct a flat list containing the prime factors in ascending order.
Example:
?- prime_factors(315, L).
L = [3,3,5,7]

```
% P35 (**) Determine the prime factors of a given positive integer.

% prime_factors(N, L) :- N is the list of prime factors of N.
%    (integer,list) (+,?)

prime_factors(N,L) :- N > 0,  prime_factors(N,L,2).

% prime_factors(N,L,K) :- L is the list of prime factors of N. It is
% known that N does not have any prime factors less than K.

prime_factors(1,[],_) :- !.
prime_factors(N,[F|L],F) :-                           % N is multiple of F
   R is N // F, N =:= R * F, !, prime_factors(R,L,F).
prime_factors(N,L,F) :-
   next_factor(N,F,NF), prime_factors(N,L,NF).        % N is not multiple of F
```

```
% next_factor(N,F,NF) :- when calculating the prime factors of N
%    and if F does not divide N then NF is the next larger candidate to
%    be a factor of N.

next_factor(_,2,3) :- !.
next_factor(N,F,NF) :- F * F < N, !, NF is F + 2.
next_factor(N,_,N).                                 % F > sqrt(N)
```

P36 (**) Determine the prime factors of a given positive integer (2).

Construct a list containing the prime factors and their multiplicity.
Example:
?- prime_factors_mult(315, L).
L = [[3,2],[5,1],[7,1]]

Hint: The problem is similar to problem P13.

```
% P36 (**) Determine the prime factors of a given positive integer (2).
% Construct a list containing the prime factors and their multiplicity.
% Example:
% ?- prime_factors_mult(315, L).
% L = [[3,2],[5,1],[7,1]]

:- ensure_loaded(p35).  % make sure next_factor/3 is loaded

% prime_factors_mult(N, L) :- L is the list of prime factors of N. It is
%    composed of terms [F,M] where F is a prime factor and M its multiplicity.
%    (integer,list) (+,?)

prime_factors_mult(N,L) :- N > 0, prime_factors_mult(N,L,2).

% prime_factors_mult(N,L,K) :- L is the list of prime factors of N. It is
% known that N does not have any prime factors less than K.

prime_factors_mult(1,[],_) :- !.
prime_factors_mult(N,[[F,M]|L],F) :- divide(N,F,M,R), !, % F divides N
   next_factor(R,F,NF), prime_factors_mult(R,L,NF).
prime_factors_mult(N,L,F) :- !,                          % F does not divide N
   next_factor(N,F,NF), prime_factors_mult(N,L,NF).

% divide(N,F,M,R) :- N = R * F**M, M >= 1, and F is not a factor of R.
%    (integer,integer,integer,integer) (+,+,-,-)

divide(N,F,M,R) :- divi(N,F,M,R,0), M > 0.

divi(N,F,M,R,K) :- S is N // F, N =:= S * F, !,          % F divides N
   K1 is K + 1, divi(S,F,M,R,K1).
divi(N,_,M,N,M).
```

P37 (**) Calculate Euler's totient function phi(m) (improved).

See problem P34 for the definition of Euler's totient function. If the list of the prime factors of a number m is known in the form of problem P36 then the function phi(m) can be efficiently calculated as follows: Let [[p1,m1],[p2,m2],[p3,m3],...] be the list of prime factors (and their multiplicities) of a given number m. Then phi(m) can be calculated with the following formula:

phi(m) = (p1 - 1) * p1 ** (m1 - 1) + (p2 - 1) * p2 ** (m2 - 1) + (p3 - 1) * p3 ** (m3 - 1) + ...

Note that a ** b stands for the b'th power of a.

```
% P37 (**) Calculate Euler's totient function phi(m) (improved).
% See problem P34 for the definition of Euler's totient function.
% If the list of the prime factors of a number m is known in the
% form of problem P36 then the function phi(m) can be efficiently
% calculated as follows:
%
% Let [[p1,m1],[p2,m2],[p3,m3],...] be the list of prime factors (and their
% multiplicities) of a given number m. Then phi(m) can be calculated
% with the following formula:
%
% phi(m) = (p1 - 1) * p1 ** (m1 - 1) + (p2 - 1) * p2 ** (m2 - 1) +
%          (p3 - 1) * p3 ** (m3 - 1) + ...
%
% Note that a ** b stands for the b'th power of a.

:- ensure_loaded(p36).

% totient_phi_2(N,Phi) :- Phi is the value of Euler's totient function
%    for the argument N.
%    (integer,integer) (+,?)

totient_phi_2(N,Phi) :- prime_factors_mult(N,L), to_phi(L,Phi).

to_phi([],1).
to_phi([[F,1]|L],Phi) :- !,
   to_phi(L,Phi1), Phi is Phi1 * (F - 1).
to_phi([[F,M]|L],Phi) :- M > 1,
   M1 is M - 1, to_phi([[F,M1]|L],Phi1), Phi is Phi1 * F.
```

P38 (*) Compare the two methods of calculating Euler's totient function.

Use the solutions of problems P34 and P37 to compare the algorithms. Take the number of logical inferences as a measure for efficiency. Try to calculate phi(10090) as an example.

```
% Use the solutions of problems P34 and P37 to compare the algorithms.
% Take the number of logical inferences as a measure for efficiency.

:- ensure_loaded(p34).
:- ensure_loaded(p37).

totient_test(N) :-
   write('totient_phi (P34):'),
   time(totient_phi(N,Phi1)),
   write('result = '), write(Phi1), nl,
   write('totient_phi_2 (P37):'),
   time(totient_phi_2(N,Phi2)),
   write('result = '), write(Phi2), nl.
```

P39 (*) A list of prime numbers.

Given a range of integers by its lower and upper limit, construct a list of all prime numbers in that range.

```
% P39 (*) A list of prime numbers.
% Given a range of integers by its lower and upper limit, construct a
% list of all prime numbers in that range.

:- ensure_loaded(p31).   % make sure is_prime/1 is loaded

% prime_list(A,B,L) :- L is the list of prime number P with A <= P <= B

prime_list(A,B,L) :- A =< 2, !, p_list(2,B,L).
prime_list(A,B,L) :- A1 is (A // 2) * 2 + 1, p_list(A1,B,L).

p_list(A,B,[]) :- A > B, !.
p_list(A,B,[A|L]) :- is_prime(A), !,
   next(A,A1), p_list(A1,B,L).
p_list(A,B,L) :-
   next(A,A1), p_list(A1,B,L).

next(2,3) :- !.
next(A,A1) :- A1 is A + 2.
```

P40 (**) Goldbach's conjecture.

Goldbach's conjecture says that every positive even number greater than 2 is the sum of two prime numbers. Example: 28 = 5 + 23. It is one of the most famous facts in number theory that has not been proved to be correct in the general case. It has been *numerically* confirmed up to very large numbers (much larger than we can go with our Prolog system). Write a predicate to find the two prime numbers that sum up to a given even integer.

Example:
?- goldbach(28, L).
L = [5,23]

```
% P40 (**) Goldbach's conjecture.
% Goldbach's conjecture says that every positive even number greater
% than 2 is the sum of two prime numbers. Example: 28 = 5 + 23.

:- ensure_loaded(p31).

% goldbach(N,L) :- L is the list of the two prime numbers that
%    sum up to the given N (which must be even).
%    (integer,integer) (+,-)

goldbach(4,[2,2]) :- !.
goldbach(N,L) :- N mod 2 =:= 0, N > 4, goldbach(N,L,3).

goldbach(N,[P,Q],P) :- Q is N - P, is_prime(Q), !.
goldbach(N,L,P) :- P < N, next_prime(P,P1), goldbach(N,L,P1).

next_prime(P,P1) :- P1 is P + 2, is_prime(P1), !.
next_prime(P,P1) :- P2 is P + 2, next_prime(P2,P1).
```

P41 (**) A list of Goldbach compositions.

Given a range of integers by its lower and upper limit, print a list of all even numbers and their Goldbach composition.

Example:
?- goldbach_list(9,20).
10 = 3 + 7
12 = 5 + 7
14 = 3 + 11
16 = 3 + 13
18 = 5 + 13
20 = 3 + 17

In most cases, if an even number is written as the sum of two prime numbers, one of them is very small. Very rarely, the primes are both bigger than say 50. Try to find out how many such cases there are in the range 2..3000.

Example (for a print limit of 50):
?- goldbach_list(1,2000,50).
992 = 73 + 919
1382 = 61 + 1321
1856 = 67 + 1789
1928 = 61 + 1867

```
% P41 (*) A list of Goldbach compositions.
% Given a range of integers by its lower and upper limit,
% print a list of all even numbers and their Goldbach composition.

:- ensure_loaded(p40).

% goldbach_list(A,B) :- print a list of the Goldbach composition
%    of all even numbers N in the range A <= N <= B
%    (integer,integer) (+,+)

goldbach_list(A,B) :- goldbach_list(A,B,2).

% goldbach_list(A,B,L) :- perform goldbach_list(A,B), but suppress
% all output when the first prime number is less than the limit L.

goldbach_list(A,B,L) :- A =< 4, !, g_list(4,B,L).
goldbach_list(A,B,L) :- A1 is ((A+1) // 2) * 2, g_list(A1,B,L).

g_list(A,B,_) :- A > B, !.
g_list(A,B,L) :-
   goldbach(A,[P,Q]),
   print_goldbach(A,P,Q,L),
   A2 is A + 2,
   g_list(A2,B,L).

print_goldbach(A,P,Q,L) :- P >= L, !,
   writef('%t = %t + %t',[A,P,Q]), nl.
print_goldbach(_,_,_,_).
```

Logic and Codes

P46 (**) Truth tables for logical expressions.

Define predicates and/2, or/2, nand/2, nor/2, xor/2, impl/2 and equ/2 (for logical equivalence) which succeed or fail according to the result of their respective operations; e.g. and(A,B) will succeed, if and only if both A and B succeed. Note that A and B can be Prolog goals (not only the constants true and fail).

A logical expression in two variables can then be written in prefix notation, as in the following example: and(or(A,B),nand(A,B)).

Now, write a predicate table/3 which prints the truth table of a given logical expression in two variables.

Example:
?- table(A,B,and(A,or(A,B))).
```
true true true
true fail true
fail true fail
fail fail fail
```

```
% P46 (**) Truth tables for logical expressions.
% Define predicates and/2, or/2, nand/2, nor/2, xor/2, impl/2
% and equ/2 (for logical equivalence) which succeed or
% fail according to the result of their respective operations; e.g.
% and(A,B) will succeed, if and only if both A and B succeed.
% Note that A and B can be Prolog goals (not only the constants
% true and fail).
% A logical expression in two variables can then be written in
% prefix notation, as in the following example: and(or(A,B),nand(A,B)).
%
% Now, write a predicate table/3 which prints the truth table of a
% given logical expression in two variables.
%
% Example:
% ?- table(A,B,and(A,or(A,B))).
% true  true  true
% true  fail  true
% fail  true  fail
% fail  fail  fail

and(A,B) :- A, B.

or(A,_) :- A.
or(_,B) :- B.

equ(A,B) :- or(and(A,B), and(not(A),not(B))).

xor(A,B) :- not(equ(A,B)).

nor(A,B) :- not(or(A,B)).

nand(A,B) :- not(and(A,B)).
```

```
impl(A,B) :- or(not(A),B).

% bind(X) :- instantiate X to be true and false successively

bind(true).
bind(fail).

table(A,B,Expr) :- bind(A), bind(B), do(A,B,Expr), fail.

do(A,B,_) :- write(A), write('  '), write(B), write('  '), fail.
do(_,_,Expr) :- Expr, !, write(true), nl.
do(_,_,_) :- write(fail), nl.
```

P47 (*) Truth tables for logical expressions (2).

Continue problem P46 by defining and/2, or/2, etc as being operators. This allows to write the logical expression in the more natural way, as in the example: A and (A or not B). Define operator precedence as usual; i.e. as in Java.

Example:
?- table(A,B, A and (A or not B)).

```
true true true
true fail true
fail true fail
fail fail fail
```

```
% P47 (*) Truth tables for logical expressions (2).
% Continue problem P46 by defining and/2, or/2, etc as being
% operators. This allows to write the logical expression in the
% more natural way, as in the example: A and (A or not B).
% Define operator precedence as usual; i.e. as in Java.
%
% Example:
% ?- table(A,B, A and (A or not B)).
% true  true  true
% true  fail  true
% fail  true  fail
% fail  fail  fail

:- ensure_loaded(p46).

:- op(900, fy,not).
:- op(910, yfx, and).
:- op(910, yfx, nand).
:- op(920, yfx, or).
:- op(920, yfx, nor).
:- op(930, yfx, impl).
:- op(930, yfx, equ).
:- op(930, yfx, xor).

% I.e. not binds stronger than (and, nand), which bind stronger than
% (or,nor) which in turn bind stronger than implication, equivalence
% and xor.
```

P48 (**) Truth tables for logical expressions (3).

Generalize problem P47 in such a way that the logical expression may contain any number of logical variables. Define table/2 in a way that table(List,Expr) prints the truth table for the expression Expr, which contains the logical variables enumerated in List.

Example:
?- table([A,B,C], A and (B or C) equ A and B or A and C).

```
true true true true
true true fail true
true fail true true
true fail fail true
fail true true true
fail true fail true
fail fail true true
fail fail fail true
```

```
% P48 (**) Truth tables for logical expressions (3).
% Generalize problem P47 in such a way that the logical
% expression may contain any number of logical variables.
%
% Example:
% ?- table([A,B,C], A and (B or C) equ A and B or A and C).
% true  true  true  true
% true  true  fail  true
% true  fail  true  true
% true  fail  fail  true
% fail  true  true  true
% fail  true  fail  true
% fail  fail  true  true
% fail  fail  fail  true

:- ensure_loaded(p47).

% table(List,Expr) :- print the truth table for the expression Expr,
%   which contains the logical variables enumerated in List.

table(VarList,Expr) :- bindList(VarList), do(VarList,Expr), fail.

bindList([]).
bindList([V|Vs]) :- bind(V), bindList(Vs).

do(VarList,Expr) :- writeVarList(VarList), writeExpr(Expr), nl.

writeVarList([]).
writeVarList([V|Vs]) :- write(V), write('  '), writeVarList(Vs).

writeExpr(Expr) :- Expr, !, write(true).
writeExpr(_) :- write(fail).
```

P49 (**) Gray code.

An n-bit Gray code is a sequence of n-bit strings constructed according to certain rules. For example,
n = 1: C(1) = ['0','1'].
n = 2: C(2) = ['00','01','11','10'].

n = 3: C(3) = ['000','001','011','010',´110´,´111´,´101´,´100´].

Find out the construction rules and write a predicate with the following specification:

% gray(N,C) :- C is the N-bit Gray code

Can you apply the method of "result caching" in order to make the predicate more efficient, when it is to be used repeatedly?

```
% (**) P49 Gray codes

% gray(N,C) :- C is the N-bit Gray code

gray(1,['0','1']).
gray(N,C) :- N > 1, N1 is N-1,
   gray(N1,C1), reverse(C1,C2),
   prepend('0',C1,C1P),
   prepend('1',C2,C2P),
   append(C1P,C2P,C).

prepend(_,[],[]) :- !.
prepend(X,[C|Cs],[CP|CPs]) :- atom_concat(X,C,CP), prepend(X,Cs,CPs).

% This gives a nice example for the result caching technique:

:- dynamic gray_c/2.

gray_c(1,['0','1']) :- !.
gray_c(N,C) :- N > 1, N1 is N-1,
   gray_c(N1,C1), reverse(C1,C2),
   prepend('0',C1,C1P),
   prepend('1',C2,C2P),
   append(C1P,C2P,C),
   asserta((gray_c(N,C) :- !)).

% Try the following goal sequence and see what happens:

% ?- [p49].
% ?- listing(gray_c/2).
% ?- gray_c(5,C).
% ?- listing(gray_c/2).


% There is an alternative definition for the gray code construction:

gray_alt(1,['0','1']).
gray_alt(N,C) :- N > 1, N1 is N-1,
   gray_alt(N1,C1),
   postpend(['0','1'],C1,C).

postpend(_,[],[]).
postpend(P,[C|Cs],[C1P,C2P|CsP]) :- P = [P1,P2],
   atom_concat(C,P1,C1P),
   atom_concat(C,P2,C2P),
   reverse(P,PR),
   postpend(PR,Cs,CsP).
```

P50 (***) Huffman code.

First of all, consult a good book on discrete mathematics or algorithms for a detailed description of Huffman codes!

We suppose a set of symbols with their frequencies, given as a list of fr(S,F) terms. Example: [fr(a,45),fr(b,13),fr(c,12),fr(d,16),fr(e,9),fr(f,5)]. Our objective is to construct a list hc(S,C) terms, where C is the Huffman code word for the symbol S. In our example, the result could be Hs = [hc(a,'0'), hc(b,'101'), hc(c,'100'), hc(d,'111'), hc(e,'1101'), hc(f,'1100')] [hc(a,'01'),...etc.]. The task shall be performed by the predicate huffman/2 defined as follows:

% huffman(Fs,Hs) :- Hs is the Huffman code table for the frequency table Fs

```
% (***) P50 Huffman code

% We suppose a set of symbols with their frequencies, given as a list
% of fr(S,F) terms.
% Example: [fr(a,45),fr(b,13),fr(c,12),fr(d,16),fr(e,9),fr(f,5)].
% Our objective is to construct a list hc(S,C) terms, where C is the Huffman
% code word for the symbol S. In our example the result could be
% [hc(a, '0'), hc(b, '101'), hc(c, '100'), hc(d, '111'), hc(e, '1101'),
% hc(f, '1100')]

% The task shall be performed by the predicate huffman/2 defined as follows:

% huffman(Fs,Hs) :- Hs is the Huffman code table for the frequency table Fs
% (list-of-fr/2-terms, list-of-hc/2-terms)  (+,-).

% During the construction process, we need nodes n(F,S) where, at the
% beginning, F is a frequency and S a symbol. During the process, as n(F,S)
% becomes an internal node, S becomes a term s(L,R) with L and R being
% again n(F,S) terms. A list of n(F,S) terms, called Ns, is maintained
% as a sort of priority queue.

huffman(Fs,Cs) :-
   initialize(Fs,Ns),
   make_tree(Ns,T),
   traverse_tree(T,Cs).

initialize(Fs,Ns) :- init(Fs,NsU), sort(NsU,Ns).

init([],[]).
init([fr(S,F)|Fs],[n(F,S)|Ns]) :- init(Fs,Ns).

make_tree([T],T).
make_tree([n(F1,X1),n(F2,X2)|Ns],T) :-
   F is F1+F2,
   insert(n(F,s(n(F1,X1),n(F2,X2))),Ns,NsR),
   make_tree(NsR,T).

% insert(n(F,X),Ns,NsR) :- insert the node n(F,X) into Ns such that the
%    resulting list NsR is again sorted with respect to the frequency F.

insert(N,[],[N]) :- !.
insert(n(F,X),[n(F0,Y)|Ns],[n(F,X),n(F0,Y)|Ns]) :- F < F0, !.
insert(n(F,X),[n(F0,Y)|Ns],[n(F0,Y)|Ns1]) :- F >= F0, insert(n(F,X),Ns,Ns1).

% traverse_tree(T,Cs) :- traverse the tree T and construct the Huffman
%    code table Cs,

traverse_tree(T,Cs) :- traverse_tree(T,'',Cs1-[]), sort(Cs1,Cs).

traverse_tree(n(_,A),Code,[hc(A,Code)|Cs]-Cs) :- atom(A). % leaf node
traverse_tree(n(_,s(Left,Right)),Code,Cs1-Cs3) :-         % internal node
   atom_concat(Code,'0',CodeLeft),
   atom_concat(Code,'1',CodeRight),
   traverse_tree(Left,CodeLeft,Cs1-Cs2),
   traverse_tree(Right,CodeRight,Cs2-Cs3).


% The following predicate gives some statistical information.

huffman(Fs) :- huffman(Fs,Hs) , nl, report(Hs,5), stats(Fs,Hs).

report([],_) :- !, nl, nl.
report(Hs,0) :- !, nl, report(Hs,5).
report([hc(S,C)|Hs],N) :- N > 0, N1 is N-1,
   writef('%w %8l  ',[S,C]), report(Hs,N1).

stats(Fs,Cs) :- sort(Fs,FsS), sort(Cs,CsS), stats(FsS,CsS,0,0).

stats([],[],FreqCodeSum,FreqSum) :- Avg is FreqCodeSum/FreqSum,
   writef('Average code length (weighted) = %w\n',[Avg]).
stats([fr(S,F)|Fs],[hc(S,C)|Hs],FCS,FS) :-
   atom_chars(C,CharList), length(CharList,N),
   FCS1 is FCS + F*N, FS1 is FS + F,
   stats(Fs,Hs,FCS1,FS1).
```

Binary Trees

A binary tree is either empty or it is composed of a root element and two successors which are binary trees themselves. In Prolog we represent the empty tree by the atom 'nil' and the non-empty tree by the term t(X,L,R), where X denotes the root node and L and R denote the left and right subtree, respectively.

P55 (**) Construct completely balanced binary trees

In a completely balanced binary tree, the following property holds for every node: The number of nodes in its left subtree and the number of nodes in its right subtree are almost equal, which means their difference is not greater than one.

Write a predicate cbal_tree/2 to construct completely balanced binary trees for a given number of nodes. The predicate should generate all solutions via backtracking. Put the letter 'x' as information into all nodes of the tree.
Example:
?- cbal_tree(4,T).

T = t(x, t(x, nil, nil), t(x, nil, t(x, nil, nil))) ;
T = t(x, t(x, nil, nil), t(x, t(x, nil, nil), nil)) ;
etc......No

```
% P55 (**) Construct completely balanced binary trees for a given
% number of nodes.

% cbal_tree(N,T) :- T is a completely balanced binary tree with N nodes.
% (integer, tree)  (+,?)

cbal_tree(0,nil) :- !.
cbal_tree(N,t(x,L,R)) :- N > 0,
        N0 is N - 1,
        N1 is N0//2, N2 is N0 - N1,
        distrib(N1,N2,NL,NR),
        cbal_tree(NL,L), cbal_tree(NR,R).

distrib(N,N,N,N) :- !.
distrib(N1,N2,N1,N2).
distrib(N1,N2,N2,N1).
```

P56 (**) Symmetric binary trees

Let us call a binary tree symmetric if you can draw a vertical line through the root node and then the right subtree is the mirror image of the left subtree. Write a predicate symmetric/1 to check whether a given binary tree is symmetric. **Hint:** Write a predicate mirror/2 first to check whether one tree is the mirror image of another. We are only interested in the structure, not in the contents of the nodes.

```
% P56 (**) Symmetric binary trees
% Let us call a binary tree symmetric if you can draw a vertical
% line through the root node and then the right subtree is the mirror
% image of the left subtree.
% Write a predicate symmetric/1 to check whether a given binary
% tree is symmetric. Hint: Write a predicate mirror/2 first to check
% whether one tree is the mirror image of another.

% symmetric(T) :- the binary tree T is symmetric.

symmetric(nil).
symmetric(t(_,L,R)) :- mirror(L,R).

mirror(nil,nil).
mirror(t(_,L1,R1),t(_,L2,R2)) :- mirror(L1,R2), mirror(R1,L2).
```

P57 (**) Binary search trees (dictionaries)

Use the predicate add/3, developed in chapter 4 of the course, to write a predicate to construct a binary search tree from a list of integer numbers.
Example:
?- construct([3,2,5,7,1],T).
T = t(3, t(2, t(1, nil, nil), nil), t(5, nil, t(7, nil, nil)))

Then use this predicate to test the solution of the problem P56.
Example:

?- test_symmetric([5,3,18,1,4,12,21]).
Yes
?- test_symmetric([3,2,5,7,1]).
No

```
% P57 (**) Binary search trees (dictionaries)

% Use the predicate add/3, developed in chapter 4 of the course,
% to write a predicate to construct a binary search tree
% from a list of integer numbers. Then use this predicate to test
% the solution of the problem P56

:- ensure_loaded(p56).

% add(X,T1,T2) :- the binary dictionary T2 is obtained by
% adding the item X to the binary dictionary T1
% (element,binary-dictionary,binary-dictionary) (i,i,o)

add(X,nil,t(X,nil,nil)).
add(X,t(Root,L,R),t(Root,L1,R)) :- X @< Root, add(X,L,L1).
add(X,t(Root,L,R),t(Root,L,R1)) :- X @> Root, add(X,R,R1).

construct(L,T) :- construct(L,T,nil).

construct([],T,T).
construct([N|Ns],T,T0) :- add(N,T0,T1), construct(Ns,T,T1).

test_symmetric(L) :- construct(L,T), symmetric(T).
```

P58 (**) Generate-and-test paradigm

Apply the generate-and-test paradigm to construct all symmetric, completely balanced binary trees with a given number of nodes. Example:
?- sym_cbal_trees(5,Ts).
Ts = [t(x, t(x, nil, t(x, nil, nil)), t(x, t(x, nil, nil), nil)), t(x, t(x, t(x, nil, nil), nil), t(x, nil, t(x, nil, nil)))]

How many such trees are there with 57 nodes? Investigate about how many solutions there are for a given number of nodes? What if the number is even? Write an appropriate predicate.

```
% P58 (**) Generate-and-test paradigm

% Apply the generate-and-test paradigm to construct all symmetric,
% completely balanced binary trees with a given number of nodes.

:- ensure_loaded(p55).
:- ensure_loaded(p56).


sym_cbal_tree(N,T) :- cbal_tree(N,T), symmetric(T).

sym_cbal_trees(N,Ts) :- setof(T,sym_cbal_tree(N,T),Ts).

investigate(A,B) :-
```

```
        between(A,B,N),
        sym_cbal_trees(N,Ts), length(Ts,L),
        writef('%w   %w',[N,L]), nl,
    fail.
investigate(_,_).
```

P59 (**) Construct height-balanced binary trees

In a height-balanced binary tree, the following property holds for every node: The height of its left subtree and the height of its right subtree are almost equal, which means their difference is not greater than one.

Write a predicate hbal_tree/2 to construct height-balanced binary trees for a given height. The predicate should generate all solutions via backtracking. Put the letter 'x' as information into all nodes of the tree.
Example:
?- hbal_tree(3,T).
T = t(x, t(x, t(x, nil, nil), t(x, nil, nil)), t(x, t(x, nil, nil), t(x, nil, nil))) ;
T = t(x, t(x, t(x, nil, nil), t(x, nil, nil)), t(x, t(x, nil, nil), nil)) ;
etc......No

```
% P59 (**) Construct height-balanced binary trees
% In a height-balanced binary tree, the following property holds for
% every node: The height of its left subtree and the height of
% its right subtree are almost equal, which means their
% difference is not greater than one.
% Write a predicate hbal_tree/2 to construct height-balanced
% binary trees for a given height. The predicate should
% generate all solutions via backtracking. Put the letter 'x'
% as information into all nodes of the tree.

% hbal_tree(D,T) :- T is a height-balanced binary tree with depth T

hbal_tree(0,nil) :- !.
hbal_tree(1,t(x,nil,nil)) :- !.
hbal_tree(D,t(x,L,R)) :- D > 1,
        D1 is D - 1, D2 is D - 2,
        distr(D1,D2,DL,DR),
        hbal_tree(DL,L), hbal_tree(DR,R).

distr(D1,_,D1,D1).
distr(D1,D2,D1,D2).
distr(D1,D2,D2,D1).
```

P60 (**) Construct height-balanced binary trees with a given number of nodes

Consider a height-balanced binary tree of height H. What is the maximum number of nodes it can contain?
Clearly, MaxN = 2**H - 1. However, what is the minimum number MinN? This question is more difficult. Try to find a recursive statement and turn it into a predicate minNodes/2 defined as follwos:

% minNodes(H,N) :- N is the minimum number of nodes in a height-balanced binary tree of height H.

(integer,integer), (+,?)

On the other hand, we might ask: what is the maximum height H a height-balanced binary tree with N nodes can have?

% maxHeight(N,H) :- H is the maximum height of a height-balanced binary tree with N nodes
(integer,integer), (+,?)

Now, we can attack the main problem: construct all the height-balanced binary trees with a given nuber of nodes.

% hbal_tree_nodes(N,T) :- T is a height-balanced binary tree with N nodes.

Find out how many height-balanced trees exist for N = 15.

```
% P60 (**) Construct height-balanced binary trees with a given number of nodes

:- ensure_loaded(p59).

% minNodes(H,N) :- N is the minimum number of nodes in a height-balanced
% binary tree of height H
% (integer,integer) (+,?)

minNodes(0,0) :- !.
minNodes(1,1) :- !.
minNodes(H,N) :- H > 1,
        H1 is H - 1, H2 is H - 2,
        minNodes(H1,N1), minNodes(H2,N2),
        N is 1 + N1 + N2.

% maxNodes(H,N) :- N is the maximum number of nodes in a height-balanced
% binary tree of height H
% (integer,integer) (+,?)

maxNodes(H,N) :- N is 2**H - 1.

% minHeight(N,H) :- H is the minimum height of a height-balanced
% binary tree with N nodes
% (integer,integer) (+,?)

minHeight(0,0) :- !.
minHeight(N,H) :- N > 0, N1 is N//2, minHeight(N1,H1), H is H1 + 1.

% maxHeight(N,H) :- H is the maximum height of a height-balanced
% binary tree with N nodes
% (integer,integer) (+,?)

maxHeight(N,H) :- maxHeight(N,H,1,1).

maxHeight(N,H,H1,N1) :- N1 > N, !, H is H1 - 1.
maxHeight(N,H,H1,N1) :- N1 =< N,
        H2 is H1 + 1, minNodes(H2,N2), maxHeight(N,H,H2,N2).
```

```
% hbal_tree_nodes(N,T) :- T is a height-balanced binary tree with N nodes.

hbal_tree_nodes(N,T) :-
        minHeight(N,Hmin), maxHeight(N,Hmax),
        between(Hmin,Hmax,H),
        hbal_tree(H,T), nodes(T,N).

% the following predicate is from the course (chapter 4)

%  nodes(T,N) :- the binary tree T has N nodes
% (tree,integer);  (i,*)

nodes(nil,0).
nodes(t(_,Left,Right),N) :-
   nodes(Left,NLeft),
   nodes(Right,NRight),
   N is NLeft + NRight + 1.

% count_hbal_trees(N,C) :- there are C different height-balanced binary
% trees with N nodes.

count_hbal_trees(N,C) :- setof(T,hbal_tree_nodes(N,T),Ts), length(Ts,C).
```

P61 (*) Count the internal nodes of a binary tree

An internal node of a binary tree has either one or two non-empty successors. Write a predicate count_internal_nodes/2 to count them.

% count_internal_nodes(T,N) :- the binary tree T has N internal nodes

Use the solution of problem P57 to check your predicate.

```
% P61 (*) Count the internal nodes of a binary tree

:- ensure_loaded(p57).

% internal_node(V) :- V is an internal node

internal_node(t(_,L,R)) :- \+ (L = nil, R = nil).

% count_internal_nodes(T,N) :- the binary tree T has N internal nodes

count_internal_nodes(t(_,L,R),N) :-
  internal_node(t(_,L,R)), !,
  count_internal_nodes(L,NL),
  count_internal_nodes(R,NR),
  N is NL + NR + 1.
count_internal_nodes(_,0).
```

P62 (*) Count the leaves of a binary tree

A leaf is a node with no successors. Write a predicate count_leaves/2 to count them.

% count_leaves(T,N) :- the binary tree T has N leaves

Use the solution of problem P57 to check your predicate.

```
% P62 (*) Count the leaves of a binary tree

:- ensure_loaded(p57).

% count_leaves(T,N) :- the binary tree T has N internal nodes

count_leaves(nil,0).
count_leaves(t(_,nil,nil),1) :- !.  % sorry for the red cut!
count_leaves(t(_,L,R),N) :-
        count_leaves(L,NL),
        count_leaves(R,NR),
        N is NL + NR.
```

P63 (**) Construct a complete binary tree

A *complete* binary tree with height H is defined as follows: The levels 1,2,3,...,H-1 contain the maximum number of nodes (i.e 2**(i-1) at the level i, note that we start counting the levels from 1 at the root). In level H, which may contain less than the maximum possible number of nodes, all the nodes are "left-adjusted". This means that in a levelorder tree traversal all internal nodes come first, the leaves come second, and empty successors (the nil's which are not really nodes!) come last.

Particularly, complete binary trees are used as data structures (or addressing schemes) for heaps.

We can assign an address number to each node in a complete binary tree by enumerating the nodes in levelorder, starting at the root with number 1. In doing so, we realize that for every node X with address A the following property holds: The address of X's left and right successors are 2*A and 2*A+1, respectively, supposed the successors do exist. This fact can be used to elegantly construct a complete binary tree structure. Write a predicate complete_binary_tree/2 with the following specification:

% complete_binary_tree(N,T) :- T is a complete binary tree with N nodes. (+,?)

Test your predicate in an appropriate way.

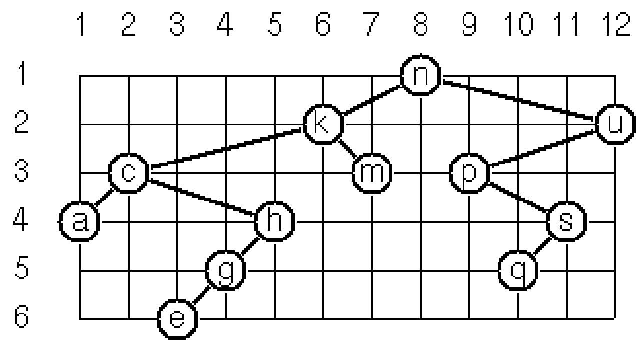
```
% P63 (**) Construct a complete binary tree
%
% A complete binary tree with height H is defined as follows:
% The levels 1,2,3,...,H-1 contain the maximum number of nodes
% (i.e 2**(i-1) at the level i, note that we start counting the
% levels from 1 at the root). In level H, which may contain less
% than the maximum number possible of nodes, all the nodes are
% "left-adjusted". This means that in a levelorder tree traversal
% all internal nodes come first, the leaves come second, and
% empty successors (the nils which are not really nodes!)
% come last. Complete binary trees are used for heaps.

:- ensure_loaded(p57).

% complete_binary_tree(N,T) :- T is a complete binary tree with
% N nodes. (+,?)
```

```
complete_binary_tree(N,T) :- complete_binary_tree(N,T,1).

complete_binary_tree(N,nil,A) :- A > N, !.
complete_binary_tree(N,t(_,L,R),A) :- A =< N,
        AL is 2 * A, AR is AL + 1,
        complete_binary_tree(N,L,AL),
        complete_binary_tree(N,R,AR).

% ----------------------------------------------------------------

% This was the solution of the exercise. What follows is an application
% of this result.

% We define a heap as a term heap(N,T) where N is the number of elements
% and T a complete binary tree (in the sense used above).

% The conservative usage of a heap is first to declare it with a predicate
% declare_heap/2 and then use it with a predicate element_at/3.

% declare_heap(H,N) :-
%    declare H to be a heap with a fixed number N  of elements

declare_heap(heap(N,T),N) :- complete_binary_tree(N,T).

% element_at(H,K,X) :- X is the element at address K in the heap H.
%  The first element has address 1.
%  (+,+,?)

element_at(heap(_,T),K,X) :-
   binary_path(K,[],BP), element_at_path(T,BP,X).

binary_path(1,Bs,Bs) :- !.
binary_path(K,Acc,Bs) :- K > 1,
   B is K /\ 1, K1 is K >> 1, binary_path(K1,[B|Acc],Bs).

element_at_path(t(X,_,_),[],X) :- !.
element_at_path(t(_,L,_),[0|Bs],X) :- !, element_at_path(L,Bs,X).
element_at_path(t(_,_,R),[1|Bs],X) :- element_at_path(R,Bs,X).


% We can transform lists into heaps and vice versa with the following
% useful predicate:

% list_heap(L,H) :- transform a list into a (limited) heap and vice versa.

list_heap(L,H) :- is_list(L), list_to_heap(L,H).
list_heap(L,heap(N,T)) :- integer(N), fill_list(heap(N,T),N,1,L).

list_to_heap(L,H) :-
   length(L,N), declare_heap(H,N), fill_heap(H,L,1).

fill_heap(_,[],_).
fill_heap(H,[X|Xs],K) :- element_at(H,K,X), K1 is K+1, fill_heap(H,Xs,K1).

fill_list(_,N,K,[]) :- K > N.
fill_list(H,N,K,[X|Xs]) :- K =< N,
   element_at(H,K,X), K1 is K+1, fill_list(H,N,K1,Xs).
```

```
% However, a more aggressive usage is *not* to define the heap in the
% beginning, but to use it as a partially instantiated data structure.
% Used in this way, the number of elements in the heap is unlimited.
% This is Power-Prolog!

% Try the following and find out exactly what happens.

% ?- element_at(H,5,alfa), element_at(H,2,beta), element(H,5,A).

% ---------------------------------------------------------------

% Test section. Suppose you have N elements in a list which must be looked
% up M times in a random order.

test1(N,M) :-
   length(List,N), lookup_list(List,N,M).

lookup_list(_,_,0) :- !.
lookup_list(List,N,M) :-
   K is random(N)+1,       % determine a random address
   nth1(K,List,_),         % look up and throw away
   M1 is M-1,
   lookup_list(List,N,M1).

% ?- time(test1(100,100000)).
% 1,384,597 inferences in 3.98 seconds (347889 Lips)
% ?- time(test1(500,100000)).
% 4,721,902 inferences in 13.82 seconds (341672 Lips)
% ?- time(test1(10000,100000)).
% 84,016,719 inferences in 277.51 seconds (302752 Lips)

test2(N,M) :-
   declare_heap(Heap,N),
   lookup_heap(Heap,N,M).

lookup_heap(_,_,0) :- !.
lookup_heap(Heap,N,M) :-
   K is random(N)+1,       % determine a random address
   element_at(Heap,K,_),   % look up and throw away
   M1 is M-1,
   lookup_heap(Heap,N,M1).

% ?- time(test2(100,100000)).
% 3,002,061 inferences in 7.81 seconds (384387 Lips)
% ?- time(test2(500,100000)).
% 4,097,961 inferences in 10.75 seconds (381206 Lips)
% ?- time(test2(10000,100000)).
% 6,366,206 inferences in 19.16 seconds (332265 Lips)

% Conclusion: In this scenario, for lists longer than 500 elements
% it is more efficient to use a heap.
```

P64 (**) Layout a binary tree (1)

Given a binary tree as the usual Prolog term t(X,L,R) (or nil). As a preparation for drawing the tree, a layout algorithm is required to determine the position of each node in a rectangular grid. Several layout methods are conceivable, one of them is shown in the illustration below.



In this layout strategy, the position of a node *v* is obtained by the following two rules:

- *x(v)* is equal to the position of the node *v* in the **inorder** sequence
- *y(v)* is equal to the depth of the node *v* in the tree

In order to store the position of the nodes, we extend the Prolog term representing a node (and its successors) as follows:

% nil represents the empty tree (as usual)
% t(W,X,Y,L,R) represents a (non-empty) binary tree with root W "positioned" at (X,Y), and subtrees L and R

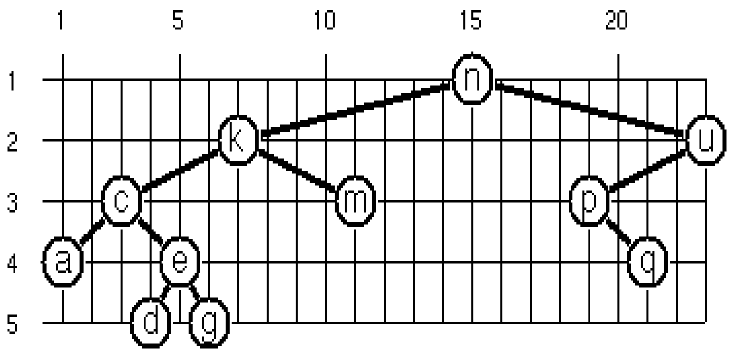
Write a predicate layout_binary_tree/2 with the following specification:

% layout_binary_tree(T,PT) :- PT is the "positioned" binary tree obtained from the binary tree T. (+,?)

Test your predicate in an appropriate way.

```
% P64 (**) Layout a binary tree (1)
%
% Given a binary tree as the usual Prolog term t(X,L,R) (or nil).
% As a preparation for drawing the tree, a layout algorithm is
% required to determine the position of each node in a rectangular
% grid. Several layout methods are conceivable, one of them is
% the following:
%
% The position of a node v is obtained by the following two rules:
%   x(v) is equal to the position of the node v in the inorder sequence
%   y(v) is equal to the depth of the node v in the tree
%
% In order to store the position of the nodes, we extend the Prolog
% term representing a node (and its successors) as follows:
%    nil represents the empty tree (as usual)
%    t(W,X,Y,L,R) represents a (non-empty) binary tree with root
%        W positionned at (X,Y), and subtrees L and R
%
% Write a predicate layout_binary_tree/2:

% layout_binary_tree(T,PT) :- PT is the "positionned" binary
%    tree obtained from the binary tree T. (+,?) or (?,+)

:- ensure_loaded(p57). % for test

layout_binary_tree(T,PT) :- layout_binary_tree(T,PT,1,_,1).

% layout_binary_tree(T,PT,In,Out,D) :- T and PT as in layout_binary_tree/2;
%    In is the position in the inorder sequence where the tree T (or PT)
%    begins, Out is the position after the last node of T (or PT) in the
%    inorder sequence. D is the depth of the root of T (or PT).
%    (+,?,+,?,+) or (?,+,+,?,+)

layout_binary_tree(nil,nil,I,I,_).
layout_binary_tree(t(W,L,R),t(W,X,Y,PL,PR),Iin,Iout,Y) :-
   Y1 is Y + 1,
   layout_binary_tree(L,PL,Iin,X,Y1),
   X1 is X + 1,
   layout_binary_tree(R,PR,X1,Iout,Y1).

% Test (see example given in the problem description):
% ?-  construct([n,k,m,c,a,h,g,e,u,p,s,q],T),layout_binary_tree(T,PT).
```

P65 (**) Layout a binary tree (2)



An alternative layout method is depicted in the illustration opposite. Find out the rules and write the corresponding Prolog predicate. Hint: On a given level, the horizontal distance between neighboring nodes is constant.

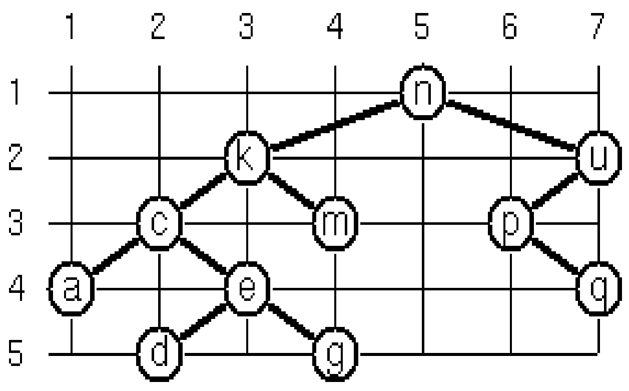
Use the same conventions as in problem P64 and test your predicate in an appropriate way.

```
% P65 (**) Layout a binary tree (2)
%
% See problem P64 for the conventions.
%
% The position of a node v is obtained by the following rules:
%   (1) y(v) is equal to the depth of the node v in the tree
%   (2) if D denotes the depth of the tree (i.e. the number of
%       populated levels) then the horizontal distance between
%       nodes at level i (counted from the root, beginning with 1)
%       is equal to 2**(D-i+1). The leftmost node of the tree
%       is at position 1.

% layout_binary_tree2(T,PT) :- PT is the "positionned" binary
%    tree obtained from the binary tree T. (+,?)

:- ensure_loaded(p57). % for test
```

```
layout_binary_tree2(nil,nil) :- !.
layout_binary_tree2(T,PT) :-
   hor_dist(T,D4), D is D4//4, x_pos(T,X,D),
   layout_binary_tree2(T,PT,X,1,D).

% hor_dist(T,D4) :- D4 is four times the horizontal distance between the
%    root node of T and its successor(s) (if any).
%    (+,-)

hor_dist(nil,1).
hor_dist(t(_,L,R),D4) :-
   hor_dist(L,D4L),
   hor_dist(R,D4R),
   D4 is 2 * max(D4L,D4R).

% x_pos(T,X,D) :- X is the horizontal position of the root node of T
%    with respect to the picture co-ordinate system. D is the horizontal
%    distance between the root node of T and its successor(s) (if any).
%    (+,-,+)

x_pos(t(_,nil,_),1,_) :- !.
x_pos(t(_,L,_),X,D) :- D2 is D//2, x_pos(L,XL,D2), X is XL+D.

% layout_binary_tree2(T,PT,X,Y,D) :- T and PT as in layout_binary_tree/2;
%    D is the the horizontal distance between the root node of T and
%    its successor(s) (if any). X, Y are the co-ordinates of the root node.
%    (+,-,+,+,+)

layout_binary_tree2(nil,nil,_,_,_).
layout_binary_tree2(t(W,L,R),t(W,X,Y,PL,PR),X,Y,D) :-
   Y1 is Y + 1,
   Xleft is X - D,
   D2 is D//2,
   layout_binary_tree2(L,PL,Xleft,Y1,D2),
   Xright is X + D,
   layout_binary_tree2(R,PR,Xright,Y1,D2).

% Test (see example given in the problem description):
% ?- construct([n,k,m,c,a,e,d,g,u,p,q],T),layout_binary_tree2(T,PT).
```

P66 (***) Layout a binary tree (3)

Yet another layout strategy is shown in the illustration opposite. The method yields a very compact layout while maintaining a certain symmetry in every node. Find out the rules and write the corresponding Prolog predicate. Hint: Consider the horizontal distance between a node and its successor nodes. How tight can you pack together two subtrees to construct the combined binary tree?



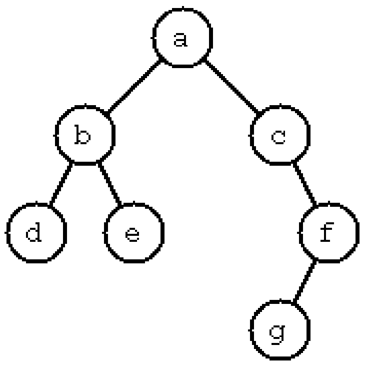
Use the same conventions as in problem P64 and P65 and test your predicate in an appropriate way. Note: This is a difficult problem. Don't give up too early!

Which layout do you like most?

```
% P66 (***) Layout a binary tree (3)
%
% See problem P64 for the conventions.
%
% The position of a node v is obtained by the following rules:
%   (1) y(v) is equal to the depth of the node v in the tree
%   (2) in order to determine the horizontal positions of the nodes we
%       construct "contours" for each subtree and shift them together
%       horizontally as close as possible. However, we maintain the
%       symmetry in each node; i.e. the horizontal distance between
%       a node and the root of its left subtree is the same as between
%       it and the root of its right subtree.
%
%       The "contour" of a tree is a list of terms c(Xleft,Xright) which
%       give the horizontal position of the outermost nodes of the tree
%       on each level, relative to the root position. In the example
%       given in the problem description, the "contour" of the tree with
%       root k would be [c(-1,1),c(-2,0),c(-1,1)]. Note that the first
%       element in the "contour" list is derived from the position of
%       the nodes c and m.

% layout_binary_tree3(T,PT) :- PT is the "positionned" binary
%    tree obtained from the binary tree T. (+,?)

:- ensure_loaded(p57). % for test

layout_binary_tree3(nil,nil) :- !.
layout_binary_tree3(T,PT) :-
   contour_tree(T,CT),      % construct the "contour" tree CT
   CT = t(_,_,_,Contour),
   mincont(Contour,MC,0),   % find the position of the leftmost node
   Xroot is 1-MC,
   layout_binary_tree3(CT,PT,Xroot,1).

contour_tree(nil,nil).
contour_tree(t(X,L,R),t(X,CL,CR,Contour)) :-
   contour_tree(L,CL),
   contour_tree(R,CR),
   combine(CL,CR,Contour).

combine(nil,nil,[]).
combine(t(_,_,_,CL),nil,[c(-1,-1)|Cs]) :- shift(CL,-1,Cs).
combine(nil,t(_,_,_,CR),[c(1,1)|Cs]) :- shift(CR,1,Cs).
combine(t(_,_,_,CL),t(_,_,_,CR),[c(DL,DR)|Cs]) :-
   maxdiff(CL,CR,MD,0),
   DR is (MD+2)//2, DL is -DR,
   merge(CL,CR,DL,DR,Cs).

shift([],_,[]).
shift([c(L,R)|Cs],S,[c(LS,RS)|CsS]) :-
   LS is L+S, RS is R+S, shift(Cs,S,CsS).
```

```
maxdiff([],_,MD,MD) :- !.
maxdiff(_,[],MD,MD) :- !.
maxdiff([c(_,R1)|Cs1],[c(L2,_)|Cs2],MD,A) :-
   A1 is max(A,R1-L2),
   maxdiff(Cs1,Cs2,MD,A1).

merge([],CR,_,DR,Cs) :- !, shift(CR,DR,Cs).
merge(CL,[],DL,_,Cs) :- !, shift(CL,DL,Cs).
merge([c(L1,_)|Cs1],[c(_,R2)|Cs2],DL,DR,[c(L,R)|Cs]) :-
   L is L1+DL, R is R2+DR,
   merge(Cs1,Cs2,DL,DR,Cs).

mincont([],MC,MC).
mincont([c(L,_)|Cs],MC,A) :-
   A1 is min(A,L), mincont(Cs,MC,A1).

layout_binary_tree3(nil,nil,_,_).
layout_binary_tree3(t(W,nil,nil,_),t(W,X,Y,nil,nil),X,Y) :- !.
layout_binary_tree3(t(W,L,R,[c(DL,DR)|_]),t(W,X,Y,PL,PR),X,Y) :-
   Y1 is Y + 1,
   Xleft is X + DL,
   layout_binary_tree3(L,PL,Xleft,Y1),
   Xright is X + DR,
   layout_binary_tree3(R,PR,Xright,Y1).

% Test (see example given in the problem description):
% ?- construct([n,k,m,c,a,e,d,g,u,p,q],T),layout_binary_tree3(T,PT).
```

P67 (**) A string representation of binary trees



Somebody represents binary trees as strings of the following type (see example opposite):
a(b(d,e),c(,f(g,)))

a) Write a Prolog predicate which generates this string representation, if the tree is given as usual (as nil or t(X,L,R) term). Then write a predicate which does this inverse; i.e. given the string representation, construct the tree in the usual form. Finally, combine the two predicates in a single predicate tree_string/2 which can be used in both directions.

b) Write the same predicate tree_string/2 using difference lists and a single predicate tree_dlist/2 which does the conversion between a tree and a difference list in both directions.

For simplicity, suppose the information in the nodes is a single letter and there are no spaces in the string.

```
% P67 (**)  A string representation of binary trees

% The string representation has the following syntax:
%
```

```
% <tree> ::=  | <letter><subtrees>
%
% <subtrees> ::=  | '(' <tree> ',' <tree> ')'
%
% According to this syntax, a leaf node (with letter x) could
% be represented by x(,) and not only by the single character x.
% However, we will avoid this when generating the string
% representation.

tree_string(T,S) :- nonvar(T), !, tree_to_string(T,S).
tree_string(T,S) :- nonvar(S), string_to_tree(S,T).

tree_to_string(T,S) :- tree_to_list(T,L), atom_chars(S,L).

tree_to_list(nil,[]).
tree_to_list(t(X,nil,nil),[X]) :- !.
tree_to_list(t(X,L,R),[X,'('|List]) :-
   tree_to_list(L,LsL),
   tree_to_list(R,LsR),
   append(LsL,[','],List1),
   append(List1,LsR,List2),
   append(List2,[')'],List).

string_to_tree(S,T) :- atom_chars(S,L), list_to_tree(L,T).

list_to_tree([],nil).
list_to_tree([X],t(X,nil,nil)) :- char_type(X,alpha).
list_to_tree([X,'('|List],t(X,Left,Right)) :- char_type(X,alpha),
   append(List1,[')'],List),
   append(LeftList,[','|RightList],List1),
   list_to_tree(LeftList,Left),
   list_to_tree(RightList,Right).
```

P68 (**) Preorder and inorder sequences of binary trees

We consider binary trees with nodes that are identified by single lower-case letters, as in the example of problem P67.

a) Write predicates preorder/2 and inorder/2 that construct the preorder and inorder sequence of a given binary tree, respectively. The results should be atoms, e.g. 'abdecfg' for the preorder sequence of the example in problem P67.

b) Can you use preorder/2 from problem part a) in the reverse direction; i.e. given a preorder sequence, construct a corresponding tree? If not, make the necessary arrangements.

c) If both the preorder sequence and the inorder sequence of the nodes of a binary tree are given, then the tree is determined unambiguously. Write a predicate pre_in_tree/3 that does the job.

d) Solve problems a) to c) using difference lists. Cool! Use the predefined predicate time/1 to compare the solutions.

What happens if the same character appears in more than one node. Try for instance pre_in_tree(aba,baa,T).

```
% P68 (**) Preorder and inorder sequences of binary trees

% We consider binary trees with nodes that are identified by
% single lower-case letters.

% a) Given a binary tree, construct its preorder sequence

preorder(T,S) :- preorder_tl(T,L), atom_chars(S,L).

preorder_tl(nil,[]).
preorder_tl(t(X,Left,Right),[X|List]) :-
   preorder_tl(Left,ListLeft),
   preorder_tl(Right,ListRight),
   append(ListLeft,ListRight,List).

inorder(T,S) :- inorder_tl(T,L), atom_chars(S,L).

inorder_tl(nil,[]).
inorder_tl(t(X,Left,Right),List) :-
   inorder_tl(Left,ListLeft),
   inorder_tl(Right,ListRight),
   append(ListLeft,[X|ListRight],List).
```

P69 (**) Dotstring representation of binary trees

We consider again binary trees with nodes that are identified by single lower-case letters, as in the example of problem P67. Such a tree can be represented by the preorder sequence of its nodes in which dots (.) are inserted where an empty subtree (nil) is encountered during the tree traversal. For example, the tree shown in problem P67 is represented as `'abd..e..c.fg...'`. First, try to establish a syntax (BNF or syntax diagrams) and then write a predicate tree_dotstring/2 which does the conversion in both directions. Use difference lists.

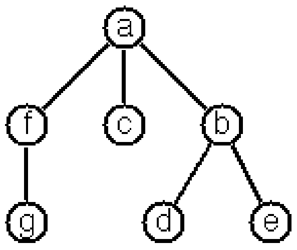
```
%  P69 (**) Dotstring representation of binary trees</B>

% The syntax of the dotstring representation is super simple:
%
% <tree> ::= . | <letter> <tree> <tree>

tree_dotstring(T,S) :- nonvar(T), !, tree_dots_dl(T,L-[]), atom_chars(S,L).
tree_dotstring(T,S) :- atom(S), atom_chars(S,L), tree_dots_dl(T,L-[]).

tree_dots_dl(nil,L1-L2) :- symbol('.',L1-L2).
tree_dots_dl(t(X,Left,Right),L1-L4) :-
   letter(X,L1-L2),
   tree_dots_dl(Left,L2-L3),
   tree_dots_dl(Right,L3-L4).

symbol(X,[X|Xs]-Xs).

letter(X,L1-L2) :- symbol(X,L1-L2), char_type(X,alpha).
```

Multiway Trees

A multiway tree is composed of a root element and a (possibly empty) set of successors which are multiway trees themselves. A multiway tree is never empty. The set of successor trees is sometimes called a forest. In Prolog we represent a multiway tree by a term t(X,F), where X denotes the root node and F denotes the forest of successor trees (a Prolog list).

P70 (**) Tree construction from a node string

We suppose that the nodes of a multiway tree contain single characters. In the depth-first order sequence of its nodes, a special character ^ has been inserted whenever, during the tree traversal, the move is a backtrack to the previous level.



By this rule, the tree in the figure opposite is represented as:
`afg^^c^bd^e^^^`

Define the syntax of the string and write a predicate tree(String,Tree) to construct the Tree when the String is given. Work with atoms (instead of strings). Make your predicate work in both directions.

```
% P70 (**) Multiway tree construction from a node string

% We suppose that the nodes of a multiway tree contain single
% characters. In the depth-first order sequence of its nodes, a
% special character ^ has been inserted whenever, during the
% tree traversal, the move is a backtrack to the previous level.

% Define the syntax of the string and write a predicate tree(String,Tree)
% to construct the Tree when the String is given. Work with atoms (instead
% of strings). Make your predicate work in both directions.
%

% Syntax in BNF:

% <tree> ::= <letter> <forest> '^'

% <forest> ::= | <tree> <forest>


% First a nice solution using difference lists

tree(TS,T) :- atom(TS), !, atom_chars(TS,TL), tree_d(TL-[],T). % (+,?)
tree(TS,T) :- nonvar(T), tree_d(TL-[],T), atom_chars(TS,TL).   % (?,+)

tree_d([X|F1]-T, t(X,F)) :- forest_d(F1-['^'|T],F).

forest_d(F-F,[]).
forest_d(F1-F3,[T|F]) :- tree_d(F1-F2,T), forest_d(F2-F3,F).


% Another solution, not as elegant as the previous one.

tree_2(TS,T) :- atom(TS), !, atom_chars(TS,TL), tree_a(TL,T). % (+,?)
```

```
tree_2(TS,T) :- nonvar(T), tree_a(TL,T), atom_chars(TS,TL).   % (?,+)

tree_a(TL,t(X,F)) :-
   append([X],FL,L1), append(L1,['^'],TL), forest_a(FL,F).

forest_a([],[]).
forest_a(FL,[T|Ts]) :- append(TL,TsL,FL),
   tree_a(TL,T), forest_a(TsL,Ts).
```

P71 (*) Determine the internal path length of a tree

We define the internal path length of a multiway tree as the total sum of the path lengths from the root to all nodes of the tree. By this definition, the tree in the figure of problem P70 has an internal path length of 9. Write a predicate ipl(Tree,IPL) for the flow pattern (+,-).

```
% P71 (*) Determine the internal path length of a tree

% We define the internal path length of a multiway tree as the
% total sum of the path lengths from the root to all nodes of the tree.

% ipl(Tree,L) :- L is the internal path length of the tree Tree
%    (multiway-tree, integer) (+,?)

ipl(T,L) :- ipl(T,0,L).

ipl(t(_,F),D,L) :- D1 is D+1, ipl(F,D1,LF), L is LF+D.

ipl([],_,0).
ipl([T1|Ts],D,L) :- ipl(T1,D,L1), ipl(Ts,D,Ls), L is L1+Ls.

% Notice the polymorphism: ipl is called with trees and with forests
% as first argument.
```

P72 (*) Construct the bottom-up order sequence of the tree nodes

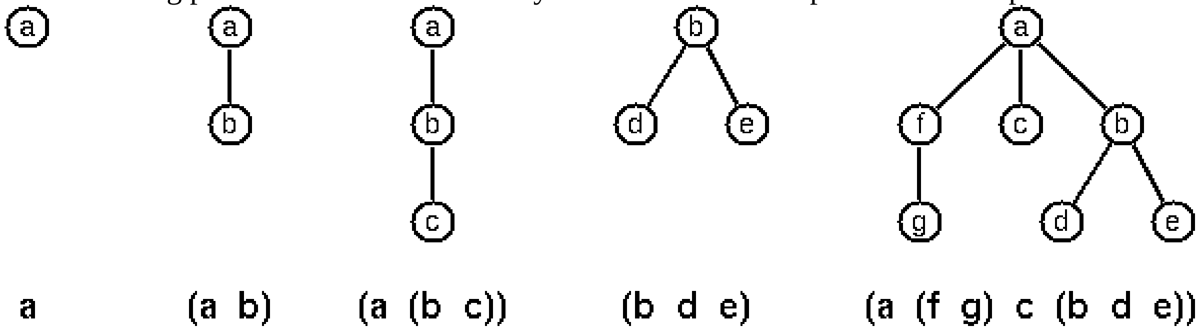
Write a predicate bottom_up(Tree,Seq) which constructs the bottom-up sequence of the nodes of the multiway tree Tree. Seq should be a Prolog list. What happens if you run your predicate backwords?

```
% P72 (*) Construct the bottom-up order sequence of the tree nodes

% bottom_up(Tree,Seq) :- Seq is the bottom-up sequence of the nodes of
%    the multiway tree Tree. (+,?)

bottom_up_f(t(X,F),Seq) :-
       bottom_up_f(F,SeqF), append(SeqF,[X],Seq).

bottom_up_f([],[]).
bottom_up_f([T|Ts],Seq):-
       bottom_up_f(T,SeqT), bottom_up_f(Ts,SeqTs), append(SeqT,SeqTs,Seq).

% The predicate bottom_up/2 produces a stack overflow when called
% in the (-,+) flow pattern. There are two problems with that.
% First, the polymorphism does not work properly, because during
% decomposing the string, the program cannot guess whether it should
% construct a tree or a forest next. We can fix this using two
% separate predicates bottom_up_tree/2 and bottom_up_forset/2.
% Secondly, if we maintain the order of the subgoals, then
% the interpreter falls into an endless loop after finding the
% first solution. We can fix this by changing the order of the
% goals as follows:

bottom_up_tree(t(X,F),Seq) :-                        % (?,+)
        append(SeqF,[X],Seq), bottom_up_forest(F,SeqF).

bottom_up_forest([],[]).
bottom_up_forest([T|Ts],Seq):-
        append(SeqT,SeqTs,Seq),
        bottom_up_tree(T,SeqT), bottom_up_forest(Ts,SeqTs).

% Unfortunately, this version doesn't run in both directions either.

% In order to have a predicate which runs forward and backward, we
% have to determine the flow pattern and then call one of the above
% predicates, as follows:

bottom_up(T,Seq) :- nonvar(T), !, bottom_up_f(T,Seq).
bottom_up(T,Seq) :- nonvar(Seq), bottom_up_tree(T,Seq).

% This is not very elegant, I agree.
```

P73 (**) Lisp-like tree representation

There is a particular notation for multiway trees in **Lisp**. Lisp is a prominent functional programming language, which is used primarily for artificial intelligence problems. As such it is one of the main competitors of Prolog. In Lisp almost everything is a list, just as in Prolog everything is a term.

The following pictures show how multiway tree structures are represented in Lisp.



Note that in the "lispy" notation a node with successors (children) in the tree is always the first element in a list, followed by its children. The "lispy" representation of a multiway tree is a sequence of atoms and parentheses '(' and ')', which we shall collectively call "tokens". We can represent this sequence of tokens as a Prolog list; e.g. the lispy expression (a (b c)) could be represented as the Prolog list ['(', a, '(', b, c, ')', ')']. Write a predicate tree_ltl(T,LTL) which constructs the "lispy token list" LTL if the tree is given as term T in the usual Prolog notation.

Example:

?- tree_ltl(t(a,[t(b,[]),t(c,[])]),LTL).
LTL = ['(', a, '(', b, c, ')', ')']

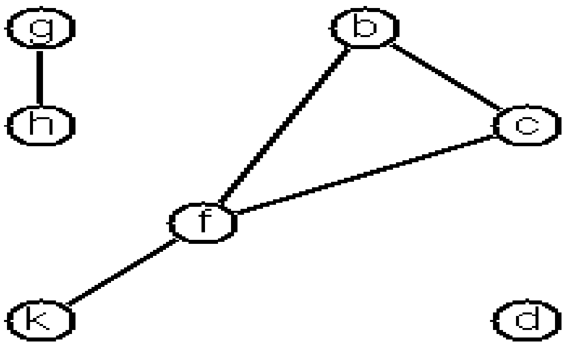
As a second, even more interesting exercise try to rewrite tree_ltl/2 in a way that the inverse conversion is also possible: Given the list LTL, construct the Prolog tree T. Use difference lists.

```
% P73 (**)  Lisp-like tree representation

% Here is my most elegant solution: a single predicate for both flow
% patterns (i,o) and (o,i)

% tree_ltl(T,L) :- L is the "lispy token list" of the multiway tree T

tree_ltl(T,L) :- tree_ltl_d(T,L-[]).

% using difference lists

tree_ltl_d(t(X,[]),[X|L]-L) :- X \= '('.
tree_ltl_d(t(X,[T|Ts]),['(',X|L]-R) :- forest_ltl_d([T|Ts],L-[')'|R]).

forest_ltl_d([],L-L).
forest_ltl_d([T|Ts],L-R) :- tree_ltl_d(T,L-M), forest_ltl_d(Ts,M-R).

% some auxiliary predicates

write_ltl([]) :- nl.
write_ltl([X|Xs]) :- write(X), write(' '), write_ltl(Xs).

dotest(T) :- write(T), nl, tree_ltl(T,L),
   write_ltl(L), tree_ltl(T1,L), write(T1), nl.

test(1) :- T = t(a,[t(b,[]),t(c,[])]), dotest(T).
test(2) :- T = t(a,[t(b,[t(c,[])])]), dotest(T).
test(3) :- T = t(a,[t(f,[t(g,[])]),t(c,[]),t(b,[t(d,[]),t(e,[])])]),
   dotest(T).
```

Graphs

A graph is defined as a set of *nodes* and a set of *edges*, where each edge is a pair of nodes.

There are several ways to represent graphs in Prolog. One method is to represent each edge separately as one clause (fact). In this form, the graph depicted below is represented as the following predicate:



```
edge(h,g).
edge(k,f).
edge(f,b).
...
```

We call this *edge-clause form*. Obviously, isolated nodes cannot be represented. Another method is to represent the whole graph as one data object. According to the definition of the graph as a pair of two sets (nodes and edges), we may use the following Prolog term to represent the example graph:

```
graph([b,c,d,f,g,h,k],[e(b,c),e(b,f),e(c,f),e(f,k),e(g,h)])
```

We call this *graph-term form*. Note, that the lists are kept sorted, they are really *sets*, without duplicated elements. Each edge appears only once in the edge list; i.e. an edge from a node x to another node y is represented as e(x,y), the term e(y,x) is not present. **The graph-term form is our default representation.** In SWI-Prolog there are predefined predicates to work with sets.

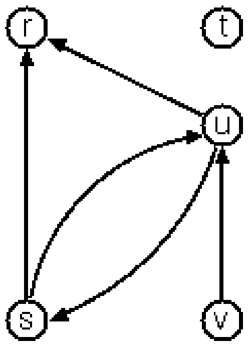
A third representation method is to associate with each node the set of nodes that are adjacent to that node. We call this the *adjacency-list form*. In our example:

```
[n(b,[c,f]), n(c,[b,f]), n(d,[]), n(f,[b,c,k]), ...]
```

The representations we introduced so far are Prolog terms and therefore well suited for automated processing, but their syntax is not very user-friendly. Typing the terms by hand is cumbersome and error-prone. We can define a more compact and "human-friendly" notation as follows: A graph is represented by a list of atoms and terms of the type X-Y (i.e. functor '-' and arity 2). The atoms stand for isolated nodes, the X-Y terms describe edges. If an X appears as an endpoint of an edge, it is automatically defined as a node. Our example could be written as:

```
[b-c, f-c, g-h, d, f-b, k-f, h-g]
```

We call this the *human-friendly form*. As the example shows, the list does not have to be sorted and may even contain the same edge multiple times. Notice the isolated node d. (Actually, isolated nodes do not even have to be atoms in the Prolog sense, they can be compound terms, as in `d(3.75,blue)` instead of d in the example).

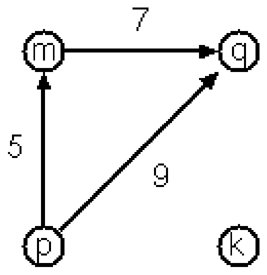


When the edges are *directed* we call them *arcs*. These are represented by *ordered* pairs. Such a graph is called **directed graph**. To represent a directed graph, the forms discussed above are slightly modified. The example graph opposite is represented as follows:

Arc-clause form
```
arc(s,u).
arc(u,r).
...
```
Graph-term form
```
digraph([r,s,t,u,v],[a(s,r),a(s,u),a(u,r),a(u,s),a(v,u)])
```
Adjacency-list form
```
[n(r,[]),n(s,[r,u]),n(t,[]),n(u,[r]),n(v,[u])]
```
Note that the adjacency-list does not have the information on whether it is a graph or a digraph.

Human-friendly form
```
[s > r, t, u > r, s > u, u > s, v > u]
```

Finally, graphs and digraphs may have additional information attached to nodes and edges (arcs). For the nodes, this is no problem, as we can easily replace the single character identifiers with arbitrary compound terms, such as `city('London',4711)`. On the other hand, for edges we have to extend our notation. Graphs with additional information attached to edges are called **labelled graphs**.



Arc-clause form
```
arc(m,q,7).
arc(p,q,9).
arc(p,m,5).
```
Graph-term form
```
digraph([k,m,p,q],[a(m,p,7),a(p,m,5),a(p,q,9)])
```
Adjacency-list form
```
[n(k,[]),n(m,[q/7]),n(p,[m/5,q/9]),n(q,[])]
```
Notice how the edge information has been packed into a term with functor '/' and arity 2, together with the corresponding node.

Human-friendly form
```
[p>q/9, m>q/7, k, p>m/5]
```

The notation for labelled graphs can also be used for so-called **multi-graphs**, where more than one edge (or arc) are allowed between two given nodes.

P80 (***) Conversions

Write predicates to convert between the different graph representations. With these predicates, all representations are equivalent; i.e. for the following problems you can always pick freely the most convenient form. The reason this problem is rated (***) is not because it's particularly difficult, but because it's a lot of work to deal with all the special cases.

```
% (**) P80 Conversions between graph representations

% We use the following notation:
%
% adjacency-list (alist): [n(b,[c,g,h]), n(c,[b,d,f,h]), n(d,[c,f]), ...]
%
% graph-term (gterm)  graph([b,c,d,f,g,h,k],[e(b,c),e(b,g),e(b,h), ...]) or
%                     digraph([r,s,t,u],[a(r,s),a(r,t),a(s,t), ...])
%
% edge-clause (ecl):  edge(b,g).  (in program database)
% arc-clause (acl):   arc(r,s).   (in program database)
%
% human-friendly (hf): [a-b,c,g-h,d-e]  or [a>b,h>g,c,b>a]
%
% The main conversion predicates are: alist_gterm/3 and human_gterm/2 which
% both (hopefully) work in either direction and for graphs as well as
% for digraphs, labelled or not.

% alist_gterm(Type,AL,GT) :- convert between adjacency-list and graph-term
%    representation. Type is either 'graph' or 'digraph'.
%    (atom,alist,gterm)  (+,+,?) or (?,?,+)

alist_gterm(Type,AL,GT):- nonvar(GT), !, gterm_to_alist(GT,Type,AL).
alist_gterm(Type,AL,GT):- atom(Type), nonvar(AL), alist_to_gterm(Type,AL,GT).

gterm_to_alist(graph(Ns,Es),graph,AL) :- memberchk(e(_,_,_),Es), ! ,
   lgt_al(Ns,Es,AL).
gterm_to_alist(graph(Ns,Es),graph,AL) :- !,
   gt_al(Ns,Es,AL).
gterm_to_alist(digraph(Ns,As),digraph,AL) :- memberchk(a(_,_,_),As), !,
   ldt_al(Ns,As,AL).
gterm_to_alist(digraph(Ns,As),digraph,AL) :-
   dt_al(Ns,As,AL).

% labelled graph
lgt_al([],_,[]).
lgt_al([V|Vs],Es,[n(V,L)|Ns]) :-
   findall(T,((member(e(X,V,I),Es) ; member(e(V,X,I),Es)),T = X/I),L),
   lgt_al(Vs,Es,Ns).

% unlabelled graph
gt_al([],_,[]).
gt_al([V|Vs],Es,[n(V,L)|Ns]) :-
   findall(X,(member(e(X,V),Es) ; member(e(V,X),Es)),L), gt_al(Vs,Es,Ns).

% labelled digraph
ldt_al([],_,[]).
ldt_al([V|Vs],As,[n(V,L)|Ns]) :-
   findall(T,(member(a(V,X,I),As), T=X/I),L), ldt_al(Vs,As,Ns).

% unlabelled digraph
dt_al([],_,[]).
dt_al([V|Vs],As,[n(V,L)|Ns]) :-
   findall(X,member(a(V,X),As),L), dt_al(Vs,As,Ns).
```

```
alist_to_gterm(graph,AL,graph(Ns,Es)) :- !, al_gt(AL,Ns,EsU,[]), sort(EsU,Es).
alist_to_gterm(digraph,AL,digraph(Ns,As)) :- al_dt(AL,Ns,AsU,[]),
sort(AsU,As).

al_gt([],[],Es,Es).
al_gt([n(V,Xs)|Ns],[V|Vs],Es,Acc) :-
   add_edges(V,Xs,Acc1,Acc), al_gt(Ns,Vs,Es,Acc1).

add_edges(_,[],Es,Es).
add_edges(V,[X/_|Xs],Es,Acc) :- V @> X, !, add_edges(V,Xs,Es,Acc).
add_edges(V,[X|Xs],Es,Acc) :- V @> X, !, add_edges(V,Xs,Es,Acc).
add_edges(V,[X/I|Xs],Es,Acc) :- V @=< X, !, add_edges(V,Xs,Es,[e(V,X,I)|Acc]).
add_edges(V,[X|Xs],Es,Acc) :- V @=< X, add_edges(V,Xs,Es,[e(V,X)|Acc]).

al_dt([],[],As,As).
al_dt([n(V,Xs)|Ns],[V|Vs],As,Acc) :-
   add_arcs(V,Xs,Acc1,Acc), al_dt(Ns,Vs,As,Acc1).

add_arcs(_,[],As,As).
add_arcs(V,[X/I|Xs],As,Acc) :- !, add_arcs(V,Xs,As,[a(V,X,I)|Acc]).
add_arcs(V,[X|Xs],As,Acc) :- add_arcs(V,Xs,As,[a(V,X)|Acc]).

% -------------------------------------------------------------------

% ecl_to_gterm(GT) :- construct a graph-term from edge/2 facts in the
%    program database.

ecl_to_gterm(GT) :-
   findall(E,(edge(X,Y),E=X-Y),Es), human_gterm(Es,GT).

% acl_to_gterm(GT) :- construct a graph-term from arc/2 facts in the
%    program database.

acl_to_gterm(GT) :-
   findall(A,(arc(X,Y),A= >(X,Y)),As), human_gterm(As,GT).

% -------------------------------------------------------------------

% human_gterm(HF,GT) :- convert between human-friendly and graph-term
%    representation.
%    (list,gterm) (+,?) or (?,+)

human_gterm(HF,GT):- nonvar(GT), !, gterm_to_human(GT,HF).
human_gterm(HF,GT):- nonvar(HF), human_to_gterm(HF,GT).

gterm_to_human(graph(Ns,Es),HF) :-  memberchk(e(_,_,_),Es), !,
   lgt_hf(Ns,Es,HF).
gterm_to_human(graph(Ns,Es),HF) :-  !,
   gt_hf(Ns,Es,HF).
gterm_to_human(digraph(Ns,As),HF) :- memberchk(a(_,_,_),As), !,
   ldt_hf(Ns,As,HF).
gterm_to_human(digraph(Ns,As),HF) :-
   dt_hf(Ns,As,HF).

% labelled graph
lgt_hf(Ns,[],Ns).
lgt_hf(Ns,[e(X,Y,I)|Es],[X-Y/I|Hs]) :-
```

```
   delete(Ns,X,Ns1),
   delete(Ns1,Y,Ns2),
   lgt_hf(Ns2,Es,Hs).

% unlabelled graph
gt_hf(Ns,[],Ns).
gt_hf(Ns,[e(X,Y)|Es],[X-Y|Hs]) :-
   delete(Ns,X,Ns1),
   delete(Ns1,Y,Ns2),
   gt_hf(Ns2,Es,Hs).

% labelled digraph
ldt_hf(Ns,[],Ns).
ldt_hf(Ns,[a(X,Y,I)|As],[X>Y/I|Hs]) :-
   delete(Ns,X,Ns1),
   delete(Ns1,Y,Ns2),
   ldt_hf(Ns2,As,Hs).

% unlabelled digraph
dt_hf(Ns,[],Ns).
dt_hf(Ns,[a(X,Y)|As],[X>Y|Hs]) :-
   delete(Ns,X,Ns1),
   delete(Ns1,Y,Ns2),
   dt_hf(Ns2,As,Hs).

% we guess that if there is a '>' term then it's a digraph, else a graph
human_to_gterm(HF,digraph(Ns,As)) :- memberchk(_>_,HF), !,
   hf_dt(HF,Ns1,As1), sort(Ns1,Ns), sort(As1,As).
human_to_gterm(HF,graph(Ns,Es)) :-
   hf_gt(HF,Ns1,Es1), sort(Ns1,Ns), sort(Es1,Es).
% remember: sort/2 removes duplicates!

hf_gt([],[],[]).
hf_gt([X-Y/I|Hs],[X,Y|Ns],[e(U,V,I)|Es]) :- !,
   sort0([X,Y],[U,V]), hf_gt(Hs,Ns,Es).
hf_gt([X-Y|Hs],[X,Y|Ns],[e(U,V)|Es]) :- !,
   sort0([X,Y],[U,V]), hf_gt(Hs,Ns,Es).
hf_gt([H|Hs],[H|Ns],Es) :- hf_gt(Hs,Ns,Es).

hf_dt([],[],[]).
hf_dt([X>Y/I|Hs],[X,Y|Ns],[a(X,Y,I)|As]) :- !,
   hf_dt(Hs,Ns,As).
hf_dt([X>Y|Hs],[X,Y|Ns],[a(X,Y)|As]) :- !,
   hf_dt(Hs,Ns,As).
hf_dt([H|Hs],[H|Ns],As) :-  hf_dt(Hs,Ns,As).

sort0([X,Y],[X,Y]) :- X @=< Y, !.
sort0([X,Y],[Y,X]) :- X @> Y.

% tests ------------------------------------------------------------

testdata([b-c, f-c, g-h, d, f-b, k-f, h-g]).
testdata([s > r, t, u > r, s > u, u > s, v > u]).
testdata([b-c/5, f-c/9, g-h/12, d, f-b/13, k-f/3, h-g/7]).
testdata([p>q/9, m>q/7, k, p>m/5]).
testdata([a,b(4711),c]).
testdata([a-b]).
testdata([]).

test :-
   testdata(H1),
   write(H1), nl,
   human_gterm(H1,G1),
   alist_gterm(Type,AL,G1),
   alist_gterm(Type,AL,G2),
   human_gterm(H2,G2),
   human_gterm(H2,G1),
   write(G1), nl, nl,
   fail.
test.
```

P81 (**) Path from one node to another one

Write a predicate path(G,A,B,P) to find an acyclic path P from node A to node b in the graph G. The predicate should return all paths via backtracking.

```
% P81 (**) Path from one node to another one

% path(G,A,B,P) :- P is a (acyclic) path from node A to node B in the graph G.
%   G is given in graph-term form.
%   (+,+,+,?)

:- ensure_loaded(p80).  % conversions

path(G,A,B,P) :- path1(G,A,[B],P).

path1(_,A,[A|P1],[A|P1]).
path1(G,A,[Y|P1],P) :-
   adjacent(X,Y,G), \+ memberchk(X,[Y|P1]), path1(G,A,[X,Y|P1],P).

% A useful predicate: adjacent/3

adjacent(X,Y,graph(_,Es)) :- member(e(X,Y),Es).
adjacent(X,Y,graph(_,Es)) :- member(e(Y,X),Es).
adjacent(X,Y,graph(_,Es)) :- member(e(X,Y,_),Es).
adjacent(X,Y,graph(_,Es)) :- member(e(Y,X,_),Es).
adjacent(X,Y,digraph(_,As)) :- member(a(X,Y),As).
adjacent(X,Y,digraph(_,As)) :- member(a(X,Y,_),As).
```

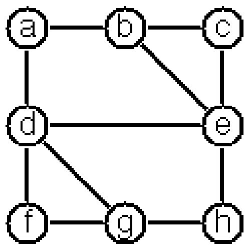
P82 (*) Cycle from a given node

Write a predicate cycle(G,A,P) to find a closed path (cycle) P starting at a given node A in the graph G. The predicate should return all cycles via backtracking.

```
% P82 (*) Cycle from a given node

% cycle(G,A,P) :- P is a closed path starting at node A in the graph G.
%    G is given in graph-term form.
%    (+,+,?)

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p81).  % adjacent/3 and path/4

cycle(G,A,P) :-
   adjacent(B,A,G), path(G,A,B,P1), length(P1,L), L > 2, append(P1,[A],P).
```

P83 (**) Construct all spanning trees

Write a predicate s_tree(Graph,Tree) to construct (by backtracking) all spanning trees of a given graph. With this predicate, find out how many spanning trees there are for the graph depicted to the left. The data of this example graph can be found in the file p83.dat. When you have a correct solution for the s_tree/2 predicate, use it to define two other useful predicates: is_tree(Graph) and is_connected(Graph). Both are five-minutes tasks!

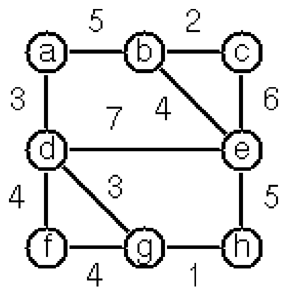


```
% P83 (**) Construct all spanning trees

% s_tree(G,T) :- T is a spanning tree of the graph G
%    (graph-term graph-term) (+,?)

:- ensure_loaded(p80).  % conversions

s_tree(graph([N|Ns],GraphEdges),graph([N|Ns],TreeEdges)) :-
   transfer(Ns,GraphEdges,TreeEdgesUnsorted),
   sort(TreeEdgesUnsorted,TreeEdges).

% transfer(Ns,GEs,TEs) :- transfer edges from GEs (graph edges)
%    to TEs (tree edges) until the list NS of still unconnected tree nodes
%    becomes empty. An edge is accepted if and only if one end-point is
%    already connected to the tree and the other is not.

transfer([],_,[]).
transfer(Ns,GEs,[GE|TEs]) :-
   select(GE,GEs,GEs1),        % modified 15-May-2001
   incident(GE,X,Y),
   acceptable(X,Y,Ns),
   delete(Ns,X,Ns1),
   delete(Ns1,Y,Ns2),
   transfer(Ns2,GEs1,TEs).

incident(e(X,Y),X,Y).
incident(e(X,Y,_),X,Y).

acceptable(X,Y,Ns) :- memberchk(X,Ns), \+ memberchk(Y,Ns), !.
acceptable(X,Y,Ns) :- memberchk(Y,Ns), \+ memberchk(X,Ns).

% An almost trivial use of the predicate s_tree/2 is the following
% tree tester predicate:

% is_tree(G) :- the graph G is a tree
is_tree(G) :- s_tree(G,G), !.


% Another use is the following connectivity tester:
```

```
% is_connected(G) :- the graph G is connected
is_connected(G) :- s_tree(G,_), !.

% Example graph p83.dat

test :-
   see('p83.dat'), read(G), seen,
   human_gterm(H,G),
   write(H), nl,
   setof(T,s_tree(G,T),Ts), length(Ts,N),
   write(N).
```

P84 (**) Construct the minimal spanning tree

Write a predicate ms_tree(Graph,Tree,Sum) to construct the minimal spanning tree of a given labelled graph. Hint: Use the algorithm of Prim. A small modification of the solution of P83 does the trick. The data of the example graph to the right can be found in the file p84.dat.



```
% P84 (**) Construct the minimal spanning tree of a
labelled graph

% ms_tree(G,T,S) :- T is a minimal spanning tree of the graph G.
%    S is the sum of the edge values. Prim's algorithm.
%    (graph-term graph-term) (+,?)

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p83).  % transfer/3, incident/3, and accept/3

ms_tree(graph([N|Ns],GraphEdges),graph([N|Ns],TreeEdges),Sum) :-
   predsort(compare_edge_values,GraphEdges,GraphEdgesSorted),
   transfer(Ns,GraphEdgesSorted,TreeEdgesUnsorted),
   sort(TreeEdgesUnsorted,TreeEdges),
   edge_sum(TreeEdges,Sum).

compare_edge_values(Order,e(X1,Y1,V1),e(X2,Y2,V2)) :-
        compare(Order,V1+X1+Y1,V2+X2+Y2).

edge_sum([],0).
edge_sum([e(_,_,V)|Es],S) :- edge_sum(Es,S1), S is S1 + V.

% Example graph p84.dat

test :-
   see('p84.dat'), read(G), seen,
   human_gterm(H,G),
   write(H), nl,
   ms_tree(G,T,S),
        human_gterm(TH,T),
   write(S), nl,
        write(TH).
```

P85 (**) Graph isomorphism

Two graphs G1(N1,E1) and G2(N2,E2) are isomorphic if there is a bijection f: N1 -> N2 such that for any nodes X,Y of N1, X and Y are adjacent if and only if f(X) and f(Y) are adjacent.

Write a predicate that determines whether two graphs are isomorphic. Hint: Use an open-ended list to represent the function f.

```
% P85 (**) Graph isomorphism

:- ensure_loaded(p80).  % conversions

% This is a solution for graphs only. It is not difficult to write the
% corresponding predicates for digraphs.

% isomorphic(G1,G2) :- the graphs G1 and G2 are isomorphic.

isomorphic(G1,G2) :- isomorphic(G1,G2,_).

% isomorphic(G1,G2,Iso) :- the graphs G1 and G2 are isomorphic.
%    Iso is a list representing the bijection between the node
%    sets of the graphs. It is an open-ended list and contains
%    a term i(X,Y) for each pair of corresponding nodes

isomorphic(graph(Ns1,Es1),graph(Ns2,Es2),Iso) :-
   append(Es1,Ns1,List1),
   append(Es2,Ns2,List2),
   isomo(List1,List2,Iso).

% isomo(List1,List2,Iso) :- the graphs represented by List1 and
%    List2 are isomorphic.

isomo([],[],_) :- !.
isomo([X|Xrest],Ys,Iso) :-
   select(Ys,Y,Yrest),
   iso(X,Y,Iso),
   isomo(Xrest,Yrest,Iso).

% iso(E1,E2,Iso) :- the edge E1 in one graph corresponds
%    to the edge E2 in the other. Note that edges are undirected.
% iso(N1,N2,Iso) :- matches isolated vertices.

iso(E1,E2,Iso) :-
   edge(E1,X1,Y1), edge(E2,X2,Y2),
   bind(X1,X2,Iso), bind(Y1,Y2,Iso).
iso(E1,E2,Iso) :-
   edge(E1,X1,Y1), edge(E2,X2,Y2),
   bind(X1,Y2,Iso), bind(Y1,X2,Iso).
iso(N1,N2,Iso) :-
   \+ edge(N1,_,_),\+ edge(N2,_,_),     % isolated vertices
   bind(N1,N2,Iso).

edge(e(X,Y),X,Y).
edge(e(X,Y,_),X,Y).

% bind(X,Y,Iso) :- it is possible to "bind X to Y" as part of the
%    bijection Iso; i.e. a term i(X,Y) is already in the list Iso,
%    or it can be added to it without violating the rules. Note that
%    bind(X,Y,Iso) makes sure that both X and Y are really "new"
%    if i(X,Y) is added to Iso.

bind(X,Y,Iso) :- memberchk(i(X,Y0),Iso), nonvar(Y0), !, Y = Y0.
bind(X,Y,Iso) :- memberchk(i(X0,Y),Iso), X = X0.

% -----------------------------------------------------------------

test(1) :-
   human_gterm([f-e,e-d,e-g,c-e,c-b,a-b,c-d,beta],G1),
   human_gterm([6-3,6-4,3-4,alfa,4-5,7-4,6-2,1-2],G2),
   isomorphic(G1,G2,Iso), write(Iso).
test(2) :-
   human_gterm([f-e,e-d,e-g,c-e,c-b,a-b,c-d,beta],G1),
   human_gterm([6-3,6-4,3-4,4-5,7-4,6-2,1],G2),
   isomorphic(G1,G2,Iso), write(Iso).
test(3) :-
   human_gterm([a-b,c-d,e,d-f],G1),
   human_gterm([1-2,1-3,5,4-6],G2),
   isomorphic(G1,G2,Iso), write(Iso).
```

P86 (**) Node degree and graph coloration

a) Write a predicate degree(Graph,Node,Deg) that determines the degree of a given node.

b) Write a predicate that generates a list of all nodes of a graph sorted according to decreasing degree.

c) Use Welch-Powell's algorithm to paint the nodes of a graph in such a way that adjacent nodes have different colors.

```
% P86 (**) Node degree and graph coloration

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p81).  % adjacent/3

% a) Write a predicate degree(Graph,Node,Deg) that determines the degree
% of a given node.

% degree(Graph,Node,Deg) :- Deg is the degree of the node Node in the
%    graph Graph.
%    (graph-term, node, integer), (+,+,?).

degree(graph(Ns,Es),Node,Deg) :-
   alist_gterm(graph,AList,graph(Ns,Es)),
   member(n(Node,AdjList),AList), !,
   length(AdjList,Deg).

% ----------------------------------------------------------------

% b) Write a predicate that generates a list of all nodes of a graph
% sorted according to decreasing degree.

% degree_sorted_nodes(Graph,Nodes) :- Nodes is the list of the nodes
%    of the graph Graph, sorted according to decreasing degree.

degree_sorted_nodes(graph(Ns,Es),DSNodes) :-
   alist_gterm(graph,AList,graph(Ns,Es)),
   predsort(compare_degree,AList,AListDegreeSorted),
   reduce(AListDegreeSorted,DSNodes).

compare_degree(Order,n(N1,AL1),n(N2,AL2)) :-
   length(AL1,D1), length(AL2,D2),
   compare(Order,D2+N1,D1+N2).

% Note: compare(Order,D2+N1,D1+N2) sorts the nodes according to
% decreasing degree, but alphabetically if the degrees are equal. Cool!

reduce([],[]).
reduce([n(N,_)|Ns],[N|NsR]) :- reduce(Ns,NsR).

% ------------------------------------------------------------------

% c) Use Welch-Powell's algorithm to paint the nodes of a graph in such
% a way that adjacent nodes have different colors.

% Use Welch-Powell's algorithm to paint the nodes of a graph
% in such a way that adjacent nodes have different colors.

paint(Graph,ColoredNodes) :-
   degree_sorted_nodes(Graph,DSNs),
   paint_nodes(Graph,DSNs,[],1,ColoredNodes).

% paint_nodes(Graph,Ns,AccNodes,Color,ColoNodes) :- paint the remaining
%    nodes Ns with a color number Color or higher. AccNodes is the set
%    of nodes already colored. Return the result in ColoNodes.
%    (graph-term,node-list,c-node-list,integer,c-node-list)
%    (+,+,+,+,-)
paint_nodes(_,[],ColoNodes,_,ColoNodes) :- !.
paint_nodes(Graph,Ns,AccNodes,Color,ColoNodes) :-
   paint_nodes(Graph,Ns,Ns,AccNodes,Color,ColoNodes).

% paint_nodes(Graph,DSNs,Ns,AccNodes,Color,ColoNodes) :- paint the
%    nodes in Ns with a fixed color number Color, if possible.
%    If Ns is empty, continue with the next color number.
%    AccNodes is the set of nodes already colored.
%    Return the result in ColoNodes.
%    (graph-term,node-list,c-node-list,c-node-list,integer,c-node-list)
%    (+,+,+,+,+,-)
paint_nodes(Graph,Ns,[],AccNodes,Color,ColoNodes) :- !,
   Color1 is Color+1,
   paint_nodes(Graph,Ns,AccNodes,Color1,ColoNodes).
paint_nodes(Graph,DSNs,[N|Ns],AccNodes,Color,ColoNodes) :-
   \+ has_neighbor(Graph,N,Color,AccNodes), !,
   delete(DSNs,N,DSNs1),
   paint_nodes(Graph,DSNs1,Ns,[c(N,Color)|AccNodes],Color,ColoNodes).
paint_nodes(Graph,DSNs,[_|Ns],AccNodes,Color,ColoNodes) :-
   paint_nodes(Graph,DSNs,Ns,AccNodes,Color,ColoNodes).

has_neighbor(Graph,N,Color,AccNodes) :-
   adjacent(N,X,Graph),
   memberchk(c(X,Color),AccNodes).
```

P87 (**) Depth-first order graph traversal

Write a predicate that generates a depth-first order graph traversal sequence. The starting point should be specified, and the output should be a list of nodes that are reachable from this starting point (in depth-first order).

```
% (**) P87 Depth-first order graph traversal

% Write a predicate that generates a depth-first order graph
% traversal sequence. The starting point should be specified,
% and the output should be a list of nodes that are reachable from
% this starting point (in depth-first order).

% The main problem is that if we traverse the graph recursively,
% we must store the encountered nodes in such a way that they
% do not disappear during the backtrack step.

% In this solution we use the "recorded database" which is a
% more efficient alternative to the well-known assert/retract
% mechanism. See the SWI-Prolog manuals for details.

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p81).  % adjacent/3

depth_first_order(Graph,Start,Seq) :-
   (Graph = graph(Ns,_), !; Graph = digraph(Ns,_)),
   memberchk(Start,Ns),
   clear_rdb(dfo),
   recorda(dfo,Start),
   (dfo(Graph,Start); true),
   bagof(X,recorded(dfo,X),Seq).

dfo(Graph,X) :-
   adjacent(X,Y,Graph),
   \+ recorded(dfo,Y),
   recordz(dfo,Y),
   dfo(Graph,Y).

clear_rdb(Key) :-
   recorded(Key,_,Ref), erase(Ref), fail.
clear_rdb(_).
```

P88 (**) Connected components

Write a predicate that splits a graph into its connected components.

```
% P88 (**) Connected components

%  Write a predicate that splits a graph into its connected components.

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p81).  % path/4

% connected_components(G,Gs) :- Gs is the list of the connected components
%    of the graph G (only for graphs, not for digraphs!)
%    (gterm, list-of-gterms), (+,-)

connected_components(graph([],[]),[]) :- !.
connected_components(graph(Ns,Es),[graph(Ns1,Es1)|Gs]) :-
   Ns = [N|_],
   component(graph(Ns,Es),N,graph(Ns1,Es1)),
   subtract(Ns,Ns1,NsR),
   subgraph(graph(Ns,Es),graph(NsR,EsR)),
   connected_components(graph(NsR,EsR),Gs).

component(graph(Ns,Es),N,graph(Ns1,Es1)) :-
   Pred =..[is_path,graph(Ns,Es),N],
   sublist(Pred,Ns,Ns1),
   subgraph(graph(Ns,Es),graph(Ns1,Es1)).

is_path(Graph,A,B) :- path(Graph,A,B,_).

% subgraph(G,G1) :- G1 is a subgraph of G
subgraph(graph(Ns,Es),graph(Ns1,Es1)) :-
   subset(Ns1,Ns),
   Pred =.. [edge_is_compatible,Ns1],
   sublist(Pred,Es,Es1).

edge_is_compatible(Ns1,Z) :-
   (Z = e(X,Y),!; Z = e(X,Y,_)),
   memberchk(X,Ns1),
   memberchk(Y,Ns1).
```

P89 (**) Bipartite graphs

Write a predicate that finds out whether a given graph is bipartite.

```
% P89 (**) Bipartite graphs

%  Write a predicate that finds out whether a given graph is bipartite.

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p88).  % connected_components/2

% is_bipartite(G) :- the graph G is bipartite

is_bipartite(G) :-
   connected_components(G,Gs),
   checklist(is_bi,Gs).

is_bi(graph(Ns,Es)) :- Ns = [N|_],
   alist_gterm(_,Alist,graph(Ns,Es)),
   paint(Alist,[],red,N).

% paint(Alist,ColoredNodes,Color,ActualNode)
% (+,+,+,+)

paint(_,CNs,Color,N) :-
   memberchk(c(N,Color),CNs), !.
paint(Alist,CNs,Color,N) :-
   \+ memberchk(c(N,_),CNs),
   other_color(Color,OtherColor),
   memberchk(n(N,AdjNodes),Alist),
   Pred =.. [paint,Alist,[c(N,Color)|CNs],OtherColor],
   checklist(Pred,AdjNodes).

other_color(red,blue).
other_color(blue,red).
```

Miscellaneous Problems

P90 (**) Eight queens problem

This is a classical problem in computer science. The objective is to place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal.

Hint: Represent the positions of the queens as a list of numbers 1..N. Example: [4,2,7,3,6,8,5,1] means that the queen in the first column is in row 4, the queen in the second column is in row 2, etc. Use the generate-and-test paradigm.

```
% P90 (**) Eight queens problem

% This is a classical problem in computer science. The objective is to
% place eight queens on a chessboard so that no two queens are attacking
% each other; i.e., no two queens are in the same row, the same column,
% or on the same diagonal. We generalize this original problem by
% allowing for an arbitrary dimension N of the chessboard.

% We represent the positions of the queens as a list of numbers 1..N.
% Example: [4,2,7,3,6,8,5,1] means that the queen in the first column
% is in row 4, the queen in the second column is in row 2, etc.
% By using the permutations of the numbers 1..N we guarantee that
% no two queens are in the same row. The only test that remains
% to be made is the diagonal test. A queen placed at column X and
% row Y occupies two diagonals: one of them, with number C = X-Y, goes
% from bottom-left to top-right, the other one, numbered D = X+Y, goes
% from top-left to bottom-right. In the test predicate we keep track
% of the already occupied diagonals in Cs and Ds.

% The first version is a simple generate-and-test solution.

% queens_1(N,Qs) :- Qs is a solution of the N-queens problem

queens_1(N,Qs) :- range(1,N,Rs), permu(Rs,Qs), test(Qs).

% range(A,B,L) :- L is the list of numbers A..B

range(A,A,[A]).
range(A,B,[A|L]) :- A < B, A1 is A+1, range(A1,B,L).

% permu(Xs,Zs) :- the list Zs is a permutation of the list Xs

permu([],[]).
permu(Qs,[Y|Ys]) :- del(Y,Qs,Rs), permu(Rs,Ys).

del(X,[X|Xs],Xs).
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).

% test(Qs) :- the list Qs represents a non-attacking queens solution

test(Qs) :- test(Qs,1,[],[]).

% test(Qs,X,Cs,Ds) :- the queens in Qs, representing columns X to N,
% are not in conflict with the diagonals Cs and Ds

test([],_,_,_).
test([Y|Ys],X,Cs,Ds) :-
        C is X-Y, \+ memberchk(C,Cs),
        D is X+Y, \+ memberchk(D,Ds),
        X1 is X + 1,
        test(Ys,X1,[C|Cs],[D|Ds]).

%---------------------------------------------------------------

% Now, in version 2, the tester is pushed completely inside the
% generator permu.

queens_2(N,Qs) :- range(1,N,Rs), permu_test(Rs,Qs,1,[],[]).

permu_test([],[],_,_,_).
permu_test(Qs,[Y|Ys],X,Cs,Ds) :-
        del(Y,Qs,Rs),
        C is X-Y, \+ memberchk(C,Cs),
        D is X+Y, \+ memberchk(D,Ds),
        X1 is X+1,
        permu_test(Rs,Ys,X1,[C|Cs],[D|Ds]).
```

P91 (**) Knight's tour

Another famous problem is this one: How can a knight jump on an NxN chessboard in such a way that it visits every square exactly once?

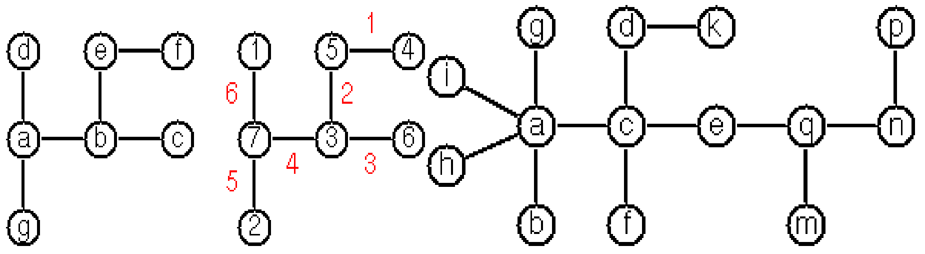
Hints: Represent the squares by pairs of their coordinates of the form X/Y, where both X and Y are integers between 1 and N. (Note that '/' is just a convenient functor, not division!) Define the relation jump(N,X/Y,U/V) to express the fact that a knight can jump from X/Y to U/V on a NxN chessboard. And finally, represent the solution of our problem as a list of N*N knight positions (the knight's tour).

```
% P91 (**) Knight's tour
% Another famous problem is this one: How can a knight jump on an
% NxN chessboard in such a way that it visits every square exactly once?

% knights(N,Knights) :- Knights is a knight's tour on a NxN chessboard

knights(N,Knights) :- M is N*N-1,  knights(N,M,[1/1],Knights).

% closed_knights(N,Knights) :- Knights is a knight's tour on a NxN
% chessboard which ends at the same square where it begun.

closed_knights(N,Knights) :-
   knights(N,Knights), Knights = [X/Y|_], jump(N,X/Y,1/1).
```

```
% knights(N,M,Visited,Knights) :- the list of squares Visited must be
% extended by M further squares to give the solution Knights of the
% NxN chessboard knight's tour problem.

knights(_,0,Knights,Knights).
knights(N,M,Visited,Knights) :-
   Visited = [X/Y|_],
   jump(N,X/Y,U/V),
   \+ memberchk(U/V,Visited),
   M1 is M-1,
   knights(N,M1,[U/V|Visited],Knights).

% jumps on an NxN chessboard from square A/B to C/D
jump(N,A/B,C/D) :-
   jump_dist(X,Y),
   C is A+X, C > 0, C =< N,
   D is B+Y, D > 0, D =< N.

% jump distances
jump_dist(1,2).
jump_dist(2,1).
jump_dist(2,-1).
jump_dist(1,-2).
jump_dist(-1,-2).
jump_dist(-2,-1).
jump_dist(-2,1).
jump_dist(-1,2).


% A more user-friendly presentation of the results -----------------------

show_knights(N) :-
   get_time(Time), convert_time(Time,Tstr),
   write('Start: '), write(Tstr), nl, nl,
   knights(N,Knights), nl, show(N,Knights).

show(N,Knights) :-
   get_time(Time), convert_time(Time,Tstr),
   write(Tstr), nl, nl,
   length(Chessboard,N),
   Pred =.. [invlength,N],
   checklist(Pred,Chessboard),
   fill_chessboard(Knights,Chessboard,1),
   checklist(show_row,Chessboard),
   nl, fail.

invlength(N,L) :- length(L,N).

show_row([]) :- nl.
show_row([S|Ss]) :- writef('%3r',[S]), show_row(Ss).

fill_chessboard([],_,_).
fill_chessboard([X/Y|Ks],Chessboard,K) :-
   nth1(Y,Chessboard,Row),
   nth1(X,Row,K),
   K1 is K+1,
   fill_chessboard(Ks,Chessboard,K1).
```

P92 (***) Von Koch's conjecture

Several years ago I met a mathematician who was intrigued by a problem for which he didn't know a solution. His name was Von Koch, and I don't know whether the problem has been solved since.



Anyway the puzzle goes like this: Given a tree with N nodes (and hence N-1 edges). Find a way to enumerate the nodes from 1 to N and, accordingly, the edges from 1 to N-1 in such a way, that for each edge K the difference of its node numbers (labels) equals to K. The conjecture is that this is always possible.

For small trees the problem is easy to solve by hand. However, for larger trees, and 14 is already very large, it is difficult to find a solution. And remember, we don't know for sure whether there is always a solution!

Write a predicate that calculates a labelling scheme for a given tree. What is the solution for the larger tree pictured above?

Meanwhile I have been told by mathematicians that the problem is known as the **Graceful Labeling Problem** or **Ringel-Kotzig Conjecture**. (A graph with a labelling described above is called graceful.)

Jim Nastos (nastos@cs.ualberta.ca>) writes: *The long-standing conjecture that all trees are graceful is still open. The conjectures has been verified for all trees with 16 or less vertices. It is known, however, that all "caterpillars" are graceful. (A caterpillar is a tree which is one long path and any vertex not on this path is at a distance at most 1 away from the path.) A "lobster" is like a caterpillar, but vertices can be a distance of at most 2 away from the main path. A colleague of mine proved in his MSc thesis that all lobsters with a perfect matching are graceful, and I believe the proof is constructive (meaning it gives a method to gracefully label such graphs.) I don't know much about algorithms to gracefully label trees; considering the conjecture is open for trees with 17 vertices, I don't think any efficient method is known.*

You can find much more on this problem in an article by Josph A. Gallion (jgallian@d.umn.edu).

```
% P92 (***) Von Koch's conjecture

% Von Koch's Conjecture: Given a tree with N nodes (and hence N-1 edges).
% Find a way to enumerate the nodes from 1 to n and, accordingly, the
% edges from 1 to N-1 in such a way, that for each edge K the difference
% of its node numbers equals to K. The conjecture is that this is always
% possible.

% Example:      *      Solution:     4    Note that the node number
%              /                    /     differences of adjacent nodes
%        * -- *              3 -- 1       are just the numbers 1,2,3,4
%        |     \             |     \      which can be used to enumerate
%        *      *            2      5     the edges.

:- ensure_loaded(p80).  % conversions
:- ensure_loaded(p83).  % is_tree

% vonkoch(G,Enum) :- the nodes of the graph G can be enumerated
%    as described in Enum. Enum is a list of pairs X/K, where X
%    is a node and K the corresponding number.

vonkoch(Graph,Enum) :-
   is_tree(Graph),            % check before doing too much work!
   Graph = graph(Ns,_),
   length(Ns,N),
   human_gterm(Hs,Graph),
   vonkoch(Hs,N,Enum).

vonkoch([IsolatedNode],1,[IsolatedNode/1|_]).  % special case
vonkoch(EdgeList,N,Enum) :-
   range(1,N,NodeNumberList),
   N1 is N-1,range(1,N1,EdgeNumberList),
   bind(EdgeList,NodeNumberList,EdgeNumberList,Enum).

% The tree is given as an edge list; e.g. [d-a,a-g,b-c,e-f,b-e,a-b].
% Our problem is to find a bijection between the nodes (a,b,c,...) and
% the integer numbers 1..N which is compatible with the condition
% cited above. In order to construct this bijection, we use an open-
% ended list Enum; and we scan the given edge list.

bind([],_,_,_) :- !.
bind([V1-V2|Es],NodeNumbers,EdgeNumbers,Enum) :-
   bind_node(V1,K1,NodeNumbers,NodeNumbers1,Enum),
   bind_node(V2,K2,NodeNumbers1,NodeNumbers2,Enum),
   D is abs(K1-K2), select(D,EdgeNumbers,EdgeNumbers1), % modif 15-May-2001
   bind(Es,NodeNumbers2,EdgeNumbers1,Enum).

% bind_node(V,K,NodeNumsIn,NodeNumsOut,Enum) :-
% V/K is an element of the list Enum, and there is no V1 \= V
% such that V1/K is in Enum, and there is no K1 \= K such that
% V =:= K1 is in Enum. In the case V gets a new number, it is
% selected from the set NodeNumsIn; what remains is NodeNumsOut.
% (node,integer,integer-list,integer-list,enumeration)  (+,?,+,-,?)

bind_node(V,K,NodeNumbers,NodeNumbers,Enum) :-
   memberchk(V/K1,Enum), number(K1), !, K = K1.
bind_node(V,K,NodeNumbers,NodeNumbers1,Enum) :-
   select(K,NodeNumbers,NodeNumbers1), memberchk(V/K,Enum).

% range(A,B,L) :- L is the list of numbers A..B

range(B,B,[B]) :- !.
range(A,B,[A|L]) :- A < B, A1 is A+1, range(A1,B,L).

% test suite ----------------------------------------------------------

test(K) :-
   test_tree(K,TH),
   write(TH), nl,
   human_gterm(TH,T),
   vonkoch(T,Enum),
   write(Enum).

test_tree(1,[a-b,b-c,c-d,c-e]).
test_tree(2,[d-a,a-g,b-c,e-f,b-e,a-b]).
test_tree(3,[g-a,i-a,a-h,b-a,k-d,c-d,m-q,p-n,q-n,e-q,e-c,f-c,c-a]).
test_tree(4,[a]).

% Solution for the tree given in the problem statement (picture p92b.gif):
%
% ?- test(3).
% [g-a, i-a, a-h, b-a, k-d, c-d, m-q, p-n, q-n, e-q, e-c, f-c, c-a]
% [a/1, b/2, c/12, g/11, h/13, i/14, d/3, e/4, f/5, k/8, q/10, m/6, n/7,
p/9|_]
%
% Remark: In most cases, there are many different solutions.
```

P93 (***) An arithmetic puzzle

Given a list of integer numbers, find a correct way of inserting arithmetic signs (operators) such that the result is a correct equation. Example: With the list of numbers [2,3,5,7,11] we can form the equations 2-3+5+7 = 11 or 2 = (3*5+7)/11 (and ten others!).

```
% P93 (***)  Arithmetic puzzle: Given a list of integer numbers,
% find a correct way of inserting arithmetic signs such that
% the result is a correct equation. The idea to the problem
% is from Roland Beuret. Thanx.

% Example: With the list of  numbers [2,3,5,7,11] we can form the
% equations  2-3+5+7 = 11  or  2 = (3*5+7)/11 (and ten others!).

% equation(L,LT,RT) :- L is the list of numbers which are the leaves
%    in the arithmetic terms LT and RT - from left to right. The
%    arithmetic evaluation yields the same result for LT and RT.

equation(L,LT,RT) :-
   split(L,LL,RL),              % decompose the list L
   term(LL,LT),                 % construct the left term
   term(RL,RT),                 % construct the right term
   LT =:= RT.                   % evaluate and compare the terms

% term(L,T) :- L is the list of numbers which are the leaves in
%    the arithmetic term T - from left to right.

term([X],X).                    % a number is a term in itself
% term([X],-X).                   % unary minus
term(L,T) :-                    % general case: binary term
   split(L,LL,RL),              % decompose the list L
   term(LL,LT),                 % construct the left term
   term(RL,RT),                 % construct the right term
   binterm(LT,RT,T).            % construct combined binary term

% binterm(LT,RT,T) :- T is a combined binary term constructed from
%    left-hand term LT and right-hand term RT

binterm(LT,RT,LT+RT).
binterm(LT,RT,LT-RT).
binterm(LT,RT,LT*RT).
binterm(LT,RT,LT/RT) :- RT =\= 0.   % avoid division by zero

% split(L,L1,L2) :- split the list L into non-empty parts L1 and L2
%    such that their concatenation is L

split(L,L1,L2) :- append(L1,L2,L), L1 = [_|_], L2 = [_|_].

% do(L) :- find all solutions to the problem as given by the list of
%    numbers L, and print them out, one solution per line.

do(L) :-
   equation(L,LT,RT),
      writef('%w = %w\n',[LT,RT]),
   fail.
do(_).


% Try the following goal:   ?- do([2,3,5,7,11]).
```

P94 (***) Generate K-regular simple graphs with N nodes

In a K-regular graph all nodes have a degree of K; i.e. the number of edges incident in each node is K. How many (non-isomorphic!) 3-regular graphs with 6 nodes are there? See also a table of results and a Java applet that can represent graphs geometrically.

```
% P94 (**) Generate K-regular simple graphs with N nodes.
%
% In a K-regular graph all nodes have a degree of K.

% k_regular(K,N,Graph) :- Graph is a K-regular simple graph with N nodes.
% The graph is in graph-term form. The nodes are identified by numbers 1..N.
% All solutions can be generated via backtracking.
% (+,+,?)  (int,int,graph(nodes,edges))
%
% Note: The predicate generates the Nodes list and a list of terms u(V,F)
% which indicates, for each node V, the number F of unused (or free) edges.
% For example: with N=5, K=3 the algorithm starts with Nodes=[1,2,3,4,5]
% and UList=[u(1,3),u(2,3),u(3,3),u(4,3),u(5,3)].

k_regular(K,N,graph(Nodes,Edges)) :-
   range(1,N,Nodes),                         % generate Nodes list
   maplist(mku(K),Nodes,UList),              % generate initial UList
   k_reg(UList,0,Edges).

mku(K,V,u(V,K)).

% k_reg(UList,MinY,Edges) :- Edges is a list of e(X,Y) terms where u(X,UX)
% is the first element in UList and u(Y,UY) is another element of UList,
% with Y > MinY. Both UX and UY, which indicate the number of free edges
% of X and Y, respectively, must be greater than 0. They are both reduced
% by 1 for the recursion if the edge e(X,Y) is chosen.
% (+,+,-) (ulist,int,elist)

k_reg([],_,[]).
k_reg([u(_,0)|Us],_,Edges) :- !, k_reg(Us,0,Edges).   % no more unused edges
k_reg([u(1,UX)|Us],MinY,[e(1,Y)|Edges]) :- UX > 0,    % special case X = 1
   pick(Us,Y,MinY,Us1), !,                          % pick a Y
   UX1 is UX - 1,                                   % reduce number of unused edges
   k_reg([u(1,UX1)|Us1],Y,Edges).
k_reg([u(X,UX)|Us],MinY,[e(X,Y)|Edges]) :- X > 1, UX > 0,
   pick(Us,Y,MinY,Us1),                             % pick a Y
   UX1 is UX - 1,                                   % reduce number of unused edges
   k_reg([u(X,UX1)|Us1],Y,Edges).

% pick(UList_in,Y,MinY,UList_out) :- there is an element u(Y,UY) in UList_in,
% Y is greater than MinY, and UY > 0. UList_out is obtained from UList_in
% by reducing UY by 1 in the term u(Y,_). This predicate delivers all
% possible values of Y via backtracking.
% (+,-,+,-) (ulist,int,int,ulist)

pick([u(Y,UY)|Us],Y,MinY,[u(Y,UY1)|Us]) :- Y > MinY, UY > 0, UY1 is UY - 1.
pick([U|Us],Y,MinY,[U|Us1]) :- pick(Us,Y,MinY,Us1).

% range(X,Y,Ls) :- Ls is the list of the integer numbers from X to Y.
% (+,+,?) (int,int,int_list)

range(B,B,[B]).
range(A,B,[A|L]) :- A < B, A1 is A + 1, range(A1,B,L).

:- dynamic solution/1.

% all_k_regular(K,N,Gs) :- Gs is the list of all (non-isomorphic)
% K-regular graphs with N nodes.
% (+,+,-) (int,int,list_of_graphs)
% Note: The predicate prints each new solution as a progress report.
% Use tell('/dev/null') to switch off the printing if you don't like it.

all_k_regular(K,N,_) :-
   retractall(solution(_)),
   k_regular(K,N,Graph),
   no_iso_solution(Graph),
   write(Graph), nl,
   assert(solution(Graph)),
   fail.
all_k_regular(_,_,Graphs) :- findall(G,solution(G),Graphs).

:- ensure_loaded(p85).  % load isomorphic/2

% no_iso_solution(Graph) :- there is no graph G in the solution/1 data base
% predicate which is isomorphic to Graph

no_iso_solution(Graph) :-
   solution(G), isomorphic(Graph,G), !, fail.
no_iso_solution(_).
```

```
% The rest of this program constructs a table of K-regular simple graphs
% with N nodes for N up to a maximum N and sensible values of K.
% Example:  ?- table(6).

table(Max) :-
   nl, write('K-regular simple graphs with N nodes'), nl,
   table(3,Max).

table(N,Max) :- N =< Max, !,
   table(2,N,Max),
   N1 is N + 1,
   table(N1,Max).
table(_,_) :- nl.

table(K,N,Max) :- K < N, !,
   tell('/dev/null'),
   statistics(inferences,I1),
   all_k_regular(K,N,Gs),
   length(Gs,NSol),
   statistics(inferences,I2),
   NInf is I2 - I1,
   told,
   plural(NSol,Pl),
   writef('\nN = %w  K = %w   %w solution%w  (%w
inferences)\n',[N,K,NSol,Pl,NInf]),
   checklist(print_graph,Gs),
   K1 is K + 1,
   table(K1,N,Max).
table(_,_,_) :- nl.

plural(X,' ') :- X < 2, !.
plural(_,'s').

:- ensure_loaded(p80).  % conversion human_gterm/2

print_graph(G) :- human_gterm(HF,G), write(HF), nl.
```

P95 (**) English number words

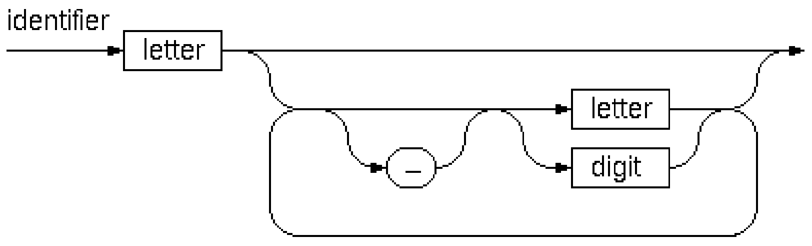
On financial documents, like cheques, numbers must sometimes be written in full words. Example: 175 must be written as one-seven-five. Write a predicate full_words/1 to print (non-negative) integer numbers in full words.

```
% P95 (**) English number words
% On financial documents, like cheques, numbers must sometimes be
% written in full words. Example: 175 must be written as one-seven-five.
% Write a predicate full_words/1 to print (non-negative) integer numbers
% in full words.

% full_words(N) :- print the number N in full words (English)
% (non-negative integer) (+)

full_words(0) :- !, write(zero), nl.
full_words(N) :- integer(N), N > 0, full_words1(N), nl.
```

```
full_words1(0) :- !.
full_words1(N) :- N > 0,
   Q is N // 10, R is N mod 10,
   full_words1(Q), numberword(R,RW), hyphen(Q), write(RW).

hyphen(0) :- !.
hyphen(Q) :- Q > 0, write('-').

numberword(0,zero).
numberword(1,one).
numberword(2,two).
numberword(3,three).
numberword(4,four).
numberword(5,five).
numberword(6,six).
numberword(7,seven).
numberword(8,eight).
numberword(9,nine).
```

P96 (**) Syntax checker (alternative solution with difference lists)

In a certain programming language (Ada) identifiers are defined by the syntax diagram (railroad chart) opposite. Transform the syntax diagram into a system of syntax diagrams which do not contain loops; i.e. which are purely recursive. Using these modified diagrams, write a predicate identifier/1 that can check whether or not a given string is a legal identifier.



% identifier(Str) :- Str is a legal identifier

```
% P96 (**) Syntax checker for Ada identifiers

% A purely recursive syntax is:
%
% <identifier> ::= <letter> <rest>
%
% <rest> ::=  | <optional_underscore> <letter_or_digit> <rest>
%
% <optional_underscore> ::=  | '_'
%
% <letter_or_digit> ::= <letter> | <digit>

% identifier(Str) :- Str is a legal Ada identifier
%    (atom) (+)

identifier(S) :- atom(S), atom_chars(S,L), identifier(L).
identifier([X|L]) :- char_type(X,alpha), rest(L).

rest([]) :- !.
rest(['_',X|L]) :- !, letter_or_digit(X), rest(L).
rest([X|L]) :- letter_or_digit(X), rest(L).

letter_or_digit(X) :- char_type(X,alpha), !.
letter_or_digit(X) :- char_type(X,digit).

% Try also a solution with difference lists!
% See p96a.pl
```

P98 (***) Nonograms

Around 1994, a certain kind of puzzles was very popular in England. The "Sunday Telegraph" newspaper wrote: "Nonograms are puzzles from Japan and are currently published each week only in The Sunday Telegraph. Simply use your logic and skill to complete the grid and reveal a picture or diagram." As a Prolog programmer, you are in a better situation: you can have your computer do the work! Just write a little program ;-).

The puzzle goes like this: Essentially, each row and column of a rectangular bitmap is annotated with the respective lengths of its distinct strings of occupied cells. The person who solves the puzzle must complete the bitmap given only these lengths.

```
Problem statement:          Solution:

|_|_|_|_|_|_|_|_| 3         |_|X|X|X|_|_|_|_| 3
|_|_|_|_|_|_|_|_| 2 1       |X|X|_|X|_|_|_|_| 2 1
|_|_|_|_|_|_|_|_| 3 2       |_|X|X|X|_|_|X|X| 3 2
|_|_|_|_|_|_|_|_| 2 2       |_|_|X|X|_|_|X|X| 2 2
|_|_|_|_|_|_|_|_| 6         |_|_|X|X|X|X|X|X| 6
|_|_|_|_|_|_|_|_| 1 5       |X|_|X|X|X|X|X|_| 1 5
|_|_|_|_|_|_|_|_| 6         |X|X|X|X|X|X|_|_| 6
|_|_|_|_|_|_|_|_| 1         |_|_|_|_|X|_|_|_| 1
|_|_|_|_|_|_|_|_| 2         |_|_|_|X|X|_|_|_| 2
 1 3 1 7 5 3 4 3             1 3 1 7 5 3 4 3
 2 1 5 1                     2 1 5 1
```

For the example above, the problem can be stated as the two lists [[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]] and [[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]] which give the "solid" lengths of the rows and columns, top-to-bottom and left-to-right, respectively. Published puzzles are larger than this example, e.g. 25 x 20, and apparently always have unique solutions.

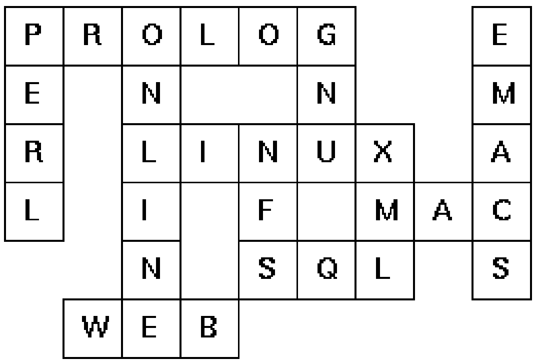
```
%  P98 (***) Nonograms

%   Around 1994, a certain kind of puzzles was very popular in England.
%   The "Sunday Telegraph" newspaper wrote: "Nonograms are puzzles from
%   Japan and are currently published each week only in The Sunday
%   Telegraph.  Simply use your logic and skill to complete the grid
%   and reveal a picture or diagram." As a Prolog programmer, you are in
%   a better situation: you can have your computer do the work! Just write
%   a little program ;-).
%   The puzzle goes like this: Essentially, each row and column of a
%   rectangular bitmap is annotated with the respective lengths of
%   its distinct strings of occupied cells. The person who solves the puzzle
%   must complete the bitmap given only these lengths.

%          Problem statement:          Solution:

%          |_|_|_|_|_|_|_|_| 3         |_|X|X|X|_|_|_|_| 3
%          |_|_|_|_|_|_|_|_| 2 1       |X|X|_|X|_|_|_|_| 2 1
%          |_|_|_|_|_|_|_|_| 3 2       |_|X|X|X|_|_|X|X| 3 2
%          |_|_|_|_|_|_|_|_| 2 2       |_|_|X|X|_|_|X|X| 2 2
%          |_|_|_|_|_|_|_|_| 6         |_|_|X|X|X|X|X|X| 6
%          |_|_|_|_|_|_|_|_| 1 5       |X|_|X|X|X|X|X|_| 1 5
%          |_|_|_|_|_|_|_|_| 6         |X|X|X|X|X|X|_|_| 6
%          |_|_|_|_|_|_|_|_| 1         |_|_|_|_|X|_|_|_| 1
%          |_|_|_|_|_|_|_|_| 2         |_|_|_|X|X|_|_|_| 2
%           1 3 1 7 5 3 4 3             1 3 1 7 5 3 4 3
%           2 1 5 1                     2 1 5 1

%   For the example above, the problem can be stated as the two lists
%   [[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]] and
%   [[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]] which give the
%   "solid" lengths of the rows and columns, top-to-bottom and
%   left-to-right, respectively. (Published puzzles are larger than this
%   example, e.g. 25 x 20, and apparently always have unique solutions.)

% Basic ideas  -----------------------------------------------------

% (1) Every square belongs to a (horizontal) row and a (vertical) column.
%     We are going to treat each square as a variable that can be accessed
%     via its row or via its column. The objective is to instantiate each
%     square with either an 'x' or a space character.

% (2) Rows and columns should be processed in a similar way. We are going
%     to collectively call them "lines", and we call the strings of
%     successive 'x's "runs". For every given line, there are, in
%     general, several possibilities to put 'x's into the squares.
%     For example, if we have to put a run of length 3 into a line
%     of length 8 then there are 6 ways to do so.

% (3) In principle, all these possibilities have to be explored for all
%     lines. However, because we are only interested in a single solution,
%     not in all of them, it may be advantageous to first try the lines
%     with few possibilities.

% -------------------------------------------------------------------

% nonogram(RowNums,ColNums,Solution,Opt) :- given the specifications for
%    the rows and columns in RowNums and ColNums, respectively, the puzzle
%    is solved by Solution, which is a row-by-row representation of
%    the filled puzzle grid. Opt = 0 is without optimization, Opt = 1
%    optimizes the order of the line tasks (see below).
%    (list-of-int-lists,list-of-int-lists,list-char-lists)    (+,+,-)

nonogram(RowNums,ColNums,Solution,Opt) :-
   length(RowNums,NRows),
   length(ColNums,NCols),
   make_rectangle(NRows,NCols,Rows,Cols),
   append(Rows,Cols,Lines),
   append(RowNums,ColNums,LineNums),
   maplist(make_runs,LineNums,LineRuns),
   combine(Lines,LineRuns,LineTasks),
   optimize(Opt,LineTasks,OptimizedLineTasks),
   solve(OptimizedLineTasks),
   Solution = Rows.

combine([],[],[]).
combine([L1|Ls],[N1|Ns],[task(L1,N1)|Ts]) :- combine(Ls,Ns,Ts).

solve([]).
solve([task(Line,LineRuns)|Tasks]) :-
   place_runs(LineRuns,Line),
   solve(Tasks).


% (1) The first basic idea is implemented as follows. ----------------

% make_rectangle(NRows,NCols,Rows,Cols) :- a rectangular array of variables
%    with NRows rows and NCols columns is generated. The variables can
%    be accessed via the Rows or via the Cols list. I.e the variable in
%    row 1 and column 2 can be addressed in the Rows list as [[_,X|_]|_]
%    or in the Cols list as [_,[X|_]|_]. Cool!
%    (integer,integer,list-of-char-list,list-of-char-list)    (+,+,_,_)

make_rectangle(NRows,NCols,Rows,Cols) :-
   NRows > 0, NCols > 0,
   length(Rows,NRows),
   Pred1 =.. [inv_length, NCols],
   checklist(Pred1,Rows),
   length(Cols,NCols),
   Pred2 =.. [inv_length, NRows],
   checklist(Pred2,Cols),
   unify_rectangle(Rows,Cols).

inv_length(Len,List) :- length(List,Len).

% unify_rectangle([[]|_],[]).
unify_rectangle(_,[]).
unify_rectangle([],_).
unify_rectangle([[X|Row1]|Rows],[[X|Col1]|Cols]) :-
   unify_row(Row1,Cols,ColsR),
   unify_rectangle(Rows,[Col1|ColsR]).

unify_row([],[],[]).
unify_row([X|Row],[[X|Col1]|Cols],[Col1|ColsR]) :- unify_row(Row,Cols,ColsR).


% (2) The second basic idea is implemented as follows ---------------

% make_runs(RunLens,Runs) :- Runs is a list of character-lists that
%    correspond to the given run lengths RunLens. Actually, each run
%    is a difference list; e.g ['x','x'|T]-T.
%    (integer-list,list-of-runs) (+,-)

make_runs([],[]) :- !.
make_runs([Len1|Lens],[Run1-T|Runs]) :-
   put_x(Len1,Run1,T),
   make_runs2(Lens,Runs).

% make_runs2(RunLens,Runs) :- same as make_runs, except that the runs
%    start with a space character.
make_runs2([],[]).
make_runs2([Len1|Lens],[[' '|Run1]-T|Runs]) :-
   put_x(Len1,Run1,T),
   make_runs2(Lens,Runs).

put_x(0,T,T) :- !.
put_x(N,['x'|Xs],T) :- N > 0, N1 is N-1, put_x(N1,Xs,T).

% place_runs(Runs,Line) :- Runs is a list of runs, each of them being
%    a difference list of characters. Line is a list of characters.
%    The runs are placed into the Line, optionally separated by
%    additional space characters. Via backtracking, all possibilities
%    are generated.
%   (run-list,square-list)  (+,?)

place_runs([],[]).
place_runs([Line-Rest|Runs],Line) :- place_runs(Runs,Rest).
place_runs(Runs,[' '|Rest]) :- place_runs(Runs,Rest).

% In order to understand what the predicates make_runs/2 make_runs2/2
% put_x/3, and place_runs/2, try the following goal:

% ?-  make_runs([3,1],Runs), Line = [_,_,_,_,_,_,_], place_runs(Runs,Line).

% (3) The third idea is an optimization. It is performed by ordering
%     the line tasks in an advantageous way. This is done by the
%     predicate optimize.

% optimize(LineTasks,LineTasksOpt)

optimize(0,LineTasks,LineTasks).

optimize(1,LineTasks,OptimizedLineTasks) :-
   label(LineTasks,LabelledLineTasks),
   sort(LabelledLineTasks,SortedLineTasks),
        unlabel(SortedLineTasks,OptimizedLineTasks).

label([],[]).
label([task(Line,LineRuns)|Tasks],[task(Count,Line,LineRuns)|LTasks]) :-
   length(Line,N),
   findall(L,(length(L,N), place_runs(LineRuns,L)),Ls),
   length(Ls,Count),
   label(Tasks,LTasks).

unlabel([],[]).
unlabel([task(_,Line,LineRuns)|LTasks],[task(Line,LineRuns)|Tasks]) :-
   unlabel(LTasks,Tasks).

% Printing the solution ------------------------------------------

% print_nonogram(RowNums,ColNums,Solution) :-
```

```
print_nonogram([],ColNums,[]) :- print_colnums(ColNums).
print_nonogram([RowNums1|RowNums],ColNums,[Row1|Rows]) :-
   print_row(Row1),
   print_rownums(RowNums1),
   print_nonogram(RowNums,ColNums,Rows).

print_row([]) :- write('  ').
print_row([X|Xs]) :- print_replace(X,Y), write(' '), write(Y), print_row(Xs).

print_replace(' ',' ') :- !.
print_replace(x,'*').

print_rownums([]) :- nl.
print_rownums([N|Ns]) :- write(N), write(' '), print_rownums(Ns).

print_colnums(ColNums) :-
   maxlength(ColNums,M,0),
        print_colnums(ColNums,ColNums,1,M).

maxlength([],M,M).
maxlength([L|Ls],M,A) :- length(L,N), B is max(A,N), maxlength(Ls,M,B).

print_colnums(_,[],M,M) :- !, nl.
print_colnums(ColNums,[],K,M) :- K < M, !, nl,
   K1 is K+1, print_colnums(ColNums,ColNums,K1,M).
print_colnums(ColNums,[Col1|Cols],K,M) :- K =< M,
   write_kth(K,Col1), print_colnums(ColNums,Cols,K,M).

write_kth(K,List) :- nth1(K,List,X), !, writef('%2r',[X]).
write_kth(_,_) :- write('  ').

% ---------------------------------------------------------------

% Test with some "real" puzzles from the Sunday Telegraph:

test(Name,Opt) :-
   specimen_nonogram(Name,Rs,Cs),
   nonogram(Rs,Cs,Solution,Opt), nl,
   print_nonogram(Rs,Cs,Solution).

% Results for the nonogram 'Hen':

% ?- time(test('Hen',0)).     - without optimization
% 16,803,498 inferences in 39.30 seconds (427570 Lips)

% ?- time(test('Hen',1)).     - with optimization
% 5,428 inferences in 0.02 seconds (271400 Lips)

% specimen_nonogram( Title, Rows, Cols) :-
%       NB  Rows, Cols and the "solid" lengths are enlisted
%       top-to-bottom or left-to-right as appropriate

specimen_nonogram(
        'Hen',
        [[3], [2,1], [3,2], [2,2], [6], [1,5], [6], [1], [2]],
        [[1,2], [3,1], [1,5], [7,1], [5], [3], [4], [3]]
        ).
```

```
specimen_nonogram(
        'Jack & The Beanstalk',
        [[3,1], [2,4,1], [1,3,3], [2,4], [3,3,1,3], [3,2,2,1,3],
         [2,2,2,2,2], [2,1,1,2,1,1], [1,2,1,4], [1,1,2,2], [2,2,8],
         [2,2,2,4], [1,2,2,1,1,1], [3,3,5,1], [1,1,3,1,1,2],
         [2,3,1,3,3], [1,3,2,8], [4,3,8], [1,4,2,5], [1,4,2,2],
         [4,2,5], [5,3,5], [4,1,1], [4,2], [3,3]],
        [[2,3], [3,1,3], [3,2,1,2], [2,4,4], [3,4,2,4,5], [2,5,2,4,6],
         [1,4,3,4,6,1], [4,3,3,6,2], [4,2,3,6,3], [1,2,4,2,1], [2,2,6],
         [1,1,6], [2,1,4,2], [4,2,6], [1,1,1,1,4], [2,4,7], [3,5,6],
         [3,2,4,2], [2,2,2], [6,3]]
        ).

specimen_nonogram(
        'WATER BUFFALO',
        [[5], [2,3,2], [2,5,1], [2,8], [2,5,11], [1,1,2,1,6], [1,2,1,3],
         [2,1,1], [2,6,2], [15,4], [10,8], [2,1,4,3,6], [17], [17],
         [18], [1,14], [1,1,14], [5,9], [8], [7]],
        [[5], [3,2], [2,1,2], [1,1,1], [1,1,1], [1,3], [2,2], [1,3,3],
         [1,3,3,1], [1,7,2], [1,9,1], [1,10], [1,10], [1,3,5], [1,8],
         [2,1,6], [3,1,7], [4,1,7], [6,1,8], [6,10], [7,10], [1,4,11],
         [1,2,11], [2,12], [3,13]]
        ).

% Thanks to ----------------------------------------------------------
%  __   __    Paul Singleton (Dr)           JANET: paul@uk.ac.keele.cs
% |__) (__    Computer Science Dept.        other: paul@cs.keele.ac.uk
% |  .  __).  Keele University, Newcastle,    tel: +44 (0)782 583477
%             Staffs ST5 5BG, ENGLAND         fax: +44 (0)782 713082
% for the idea and the examples ----------------------------------------
```

P99 (***) Crossword puzzle

Given an empty (or almost empty) framework of a crossword puzzle and a set of words. The problem is to place the words into the framework.



The particular crossword puzzle is specified in a text file which first lists the words (one word per line) in an arbitrary order. Then, after an empty line, the crossword framework is defined. In this framework specification, an empty character location is represented by a dot (.). In order to make the solution easier, character locations can also contain predefined character values. The puzzle opposite is defined in the file p99a.dat, other examples are p99b.dat and p99d.dat. There is also an example of a puzzle (p99c.dat) which does not have a solution.

Words are strings (character lists) of at least two characters. A horizontal or vertical sequence of character places in the crossword puzzle framework is called a *site*. Our problem is to find a compatible way of placing words onto sites.

Hints: (1) The problem is not easy. You will need some time to thoroughly understand it. So, don't give up too early! And remember that the objective is a clean solution, not just a quick-and-dirty hack!
(2) Reading the data file is a tricky problem for which a solution is provided in the file p99-readfile.pl. Use the predicate read_lines/2.
(3) For efficiency reasons it is important, at least for larger puzzles, to sort the words and the sites in a particular order. For this part of the problem, the solution of P28 may be very helpful.

```
% Crossword puzzle
%
% Given an empty (or almost empty) framework of a crossword puzzle and
% a set of words. The problem is to place the words into the framework.
%
% werner.hett@hta-bi.bfh.ch     Time-stamp: <8-Oct-2000 14:46 hew>
% modified argument order in select/3 predicate (SWI 3.3 -> 3.4)
% 15-May-2001 hew
%
% The particular crossword puzzle is specified in a text file which
% first lists the words (one word per line) in an arbitrary order. Then,
% after an empty line, the crossword framework is defined. In this
% framework specification, an empty character location is represented
% by a dot (.). In order to make the solution easier, character locations
% can also contain predefined character values. (See example files p99*.dat;
% note that p99c.dat does not have a solution).
%
% Words are strings (character lists) of at least two characters.
% A horizontal or vertical sequence of character places in the
% crossword framework is called a site. Our problem is to find a
% compatible way of placing words onto sites.

:- ensure_loaded('p99-readfile.pl').  % used to read the data file

% main program section -----------------------------------------------------

crossword :-
        write('usage: crossword(File)'), nl,
   write('or     crossword(File,Opt)        with Opt one of 0,1, or 2'), nl,
   write('or     crossword(File,Opt,debug)   for extra output'), nl.

:- crossword.

% crossword/1 runs without optimization (not recommended for large files)
crossword(FileName) :- crossword(FileName,0).

% crossword/2 runs with a given optimization and no debug output
crossword(FileName,Opt) :- crossword(FileName,Opt,nodebug).

% crossword/3 runs with a given optimization and a given debugging modus
crossword(FileName,Opt,Debug) :-
   read_lines(FileName,Lines),  % from file p99-readfile.pl
                                % read_lines returns a list of character-lists
   separate(Lines,Words,FrameLines),
   length(Words,NWords),
   construct_squares(FrameLines,Squares,MaxRow,MaxCol),
   debug_write(Debug,Squares),
   construct_sites(Squares,MaxRow,MaxCol,Sites),
        length(Sites,NSites),
   check_lengths(NWords,NSites),
   solve(Words,Sites,Opt,Debug), % do the real work
   show_result(Squares,MaxRow,MaxCol).

debug_write(debug,X) :- !, write(X), nl, nl.
debug_write(_,_).

check_lengths(N,N) :- !.
check_lengths(NW,NS) :- NW \= NS,
        write('Number of words does not correspond to number of sites.'), nl,
   fail.

% input preparation ---------------------------------------------------

% parse the data file and separate the word list from the framework
% description
separate(Lines,Words,FrameLines) :-
   trim_lines(Lines,LinesT),
   parse_non_empty_lines(LinesT-L1,Words),  % difference lists!
   parse_empty_lines(L1-L2),
        parse_non_empty_lines(L2-L3,FrameLines),
   parse_empty_lines(L3-[]).

% remove white space at the end of the lines
trim_lines([],[]).
trim_lines([L|Ls],[LT|LTs]) :- trim_line(L,LT), trim_lines(Ls,LTs).

trim_line(L,LT) :- reverse(L,RL), rm_white_space(RL,RLT), reverse(RLT,LT).

rm_white_space([X|Xs],L) :- char_type(X,white), !, rm_white_space(Xs,L).
rm_white_space(L,L).

% separate the word lines from the frame lines
parse_non_empty_lines([L|L1]-L2,[L|Ls]) :- L \= [], !,
   parse_non_empty_lines(L1-L2,Ls).
parse_non_empty_lines(L-L,[]).

parse_empty_lines([[]|L1]-L2) :- !, parse_empty_lines(L1-L2).
parse_empty_lines(L-L).

% A square is a position for a single character. As Prolog term a square
% has the form sq(Row,Col,X), where X denotes the character and Row and
% Col define the position within the puzzle frame. Squares is simply
% the list of all sq/3 terms.

construct_squares(FrameLines,Squares,MaxRow,MaxCol) :-   % (+,-,+,+)
   construct_squares(FrameLines,SquaresList,1),
   flatten(SquaresList,Squares),
   maxima(Squares,0,0,MaxRow,MaxCol).

construct_squares([],[],_).                              % (+,-,+)
construct_squares([FL|FLs],[SL|SLs],Row) :-
   construct_squares_row(FL,SL,Row,1),
   Row1 is Row+1,
   construct_squares(FLs,SLs,Row1).

construct_squares_row([],[],_,_).                        % (+,-,+,+)
construct_squares_row(['.'|Ps],[sq(Row,Col,_)|Sqs],Row,Col) :- !,
   Col1 is Col+1, construct_squares_row(Ps,Sqs,Row,Col1).
construct_squares_row([X|Ps],[sq(Row,Col,X)|Sqs],Row,Col) :-
   char_type(X,alpha), !,
   Col1 is Col+1, construct_squares_row(Ps,Sqs,Row,Col1).
construct_squares_row([_|Ps],Sqs,Row,Col) :-
   Col1 is Col+1, construct_squares_row(Ps,Sqs,Row,Col1).

% maxima(Squares,0,0,MaxRow,MaxCol) :- determine maximum dimensions

maxima([],MaxRow,MaxCol,MaxRow,MaxCol).
maxima([sq(Row,Col,_)|Sqs],AccRow,AccCol,MaxRow,MaxCol) :-
   AccRow1 is max(AccRow,Row),
   AccCol1 is max(AccCol,Col),
   maxima(Sqs,AccRow1,AccCol1,MaxRow,MaxCol).

% construction of sites -------------------------------------------

% construct_sites/4 traverses the framework twice in order to
% collect all the sites in the list Sites

construct_sites(Squares,MaxRow,MaxCol,Sites) :-             % (+,+,+,-)
        construct_sites_h(Squares,MaxRow,MaxCol,1,SitesH,[]),    % horizontal
        construct_sites_v(Squares,MaxRow,MaxCol,1,Sites,SitesH). % vertical

% horizontal sites

construct_sites_h(_,MaxRow,_,Row,Sites,Sites) :- Row > MaxRow, !.
construct_sites_h(Squares,MaxRow,MaxCol,Row,Sites,AccSites) :-
   construct_sites_h(Squares,MaxRow,MaxCol,Row,1,AccSites1,AccSites),
   Row1 is Row+1,
        construct_sites_h(Squares,MaxRow,MaxCol,Row1,Sites,AccSites1).

construct_sites_h(_,_,MaxCol,_,Col,Sites,Sites) :- Col > MaxCol, !.
construct_sites_h(Squares,MaxRow,MaxCol,Row,Col,Sites,AccSites) :-
   construct_site_h(Squares,Row,Col,Site,Incr), !,
   Col1 is Col+Incr,
   AccSites1 = [Site|AccSites],
   construct_sites_h(Squares,MaxRow,MaxCol,Row,Col1,Sites,AccSites1).
construct_sites_h(Squares,MaxRow,MaxCol,Row,Col,Sites,AccSites) :-
   Col1 is Col+1,
   construct_sites_h(Squares,MaxRow,MaxCol,Row,Col1,Sites,AccSites).

construct_site_h(Squares,Row,Col,[X,Y|Cs],Incr) :-
   memberchk(sq(Row,Col,X),Squares),
   Col1 is Col+1,
   memberchk(sq(Row,Col1,Y),Squares),
   Col2 is Col1+1,
   continue_site_h(Squares,Row,Col2,Cs,3,Incr).

continue_site_h(Squares,Row,Col,[X|Cs],Acc,Incr) :-
```

```
   memberchk(sq(Row,Col,X),Squares), !,
   Acc1 is Acc+1,
   Col1 is Col+1,
   continue_site_h(Squares,Row,Col1,Cs,Acc1,Incr).
continue_site_h(_,_,_,[],Incr,Incr).

% vertical sites

construct_sites_v(_,_,MaxCol,Col,Sites,Sites) :- Col > MaxCol, !.
construct_sites_v(Squares,MaxRow,MaxCol,Col,Sites,AccSites) :-
   construct_sites_v(Squares,MaxRow,MaxCol,1,Col,AccSites1,AccSites),
   Col1 is Col+1,
        construct_sites_v(Squares,MaxRow,MaxCol,Col1,Sites,AccSites1).

construct_sites_v(_,MaxRow,_,Row,_,Sites,Sites) :- Row > MaxRow, !.
construct_sites_v(Squares,MaxRow,MaxCol,Row,Col,Sites,AccSites) :-
   construct_site_v(Squares,Row,Col,Site,Incr), !,
   Row1 is Row+Incr,
   AccSites1 = [Site|AccSites],
   construct_sites_v(Squares,MaxRow,MaxCol,Row1,Col,Sites,AccSites1).
construct_sites_v(Squares,MaxRow,MaxCol,Row,Col,Sites,AccSites) :-
   Row1 is Row+1,
   construct_sites_v(Squares,MaxRow,MaxCol,Row1,Col,Sites,AccSites).

construct_site_v(Squares,Row,Col,[X,Y|Cs],Incr) :-
   memberchk(sq(Row,Col,X),Squares),
   Row1 is Row+1,
   memberchk(sq(Row1,Col,Y),Squares),
   Row2 is Row1+1,
   continue_site_v(Squares,Row2,Col,Cs,3,Incr).

continue_site_v(Squares,Row,Col,[X|Cs],Acc,Incr) :-
   memberchk(sq(Row,Col,X),Squares), !,
   Acc1 is Acc+1,
   Row1 is Row+1,
   continue_site_v(Squares,Row1,Col,Cs,Acc1,Incr).
continue_site_v(_,_,_,[],Incr,Incr).

% --------------------------------------------------------------------------

:- ensure_loaded('p28.pl').  % lsort and lfsort

% solve/4 does the optimization of the word and site lists

solve(Words,Sites,0,Debug) :- !,   % unsorted
        solve(Words,Sites,Debug).
solve(Words,Sites,1,Debug) :- !,   % length sorted
        lsort(Words,Words1,desc),
   lsort(Sites,Sites1,desc),
        solve(Words1,Sites1,Debug).
solve(Words,Sites,2,Debug) :-      % length frequency sorted
        lfsort(Words,Words1),
   lfsort(Sites,Sites1),
        solve(Words1,Sites1,Debug).

% solve/3 does the debug_write of the prepared Words and Sites
% and then calls solve/2 to do the real work
```

```
solve(Words,Sites,Debug) :-
   debug_write(Debug,Words),
   debug_write(Debug,Sites),
   solve(Words,Sites).

% solve/2 does the real work: find the right site for every word

solve([],[]).
solve([W|Words],Sites) :-
        select(W,Sites,SitesR),
        solve(Words,SitesR).

% ---------------------------------------------------------------------------

show_result(Squares,MaxRow,MaxCol) :-
   show_result(Squares,MaxRow,MaxCol,1), nl.

show_result(_,MaxRow,_,Row) :- Row > MaxRow, !.
show_result(Squares,MaxRow,MaxCol,Row) :-
   show_result(Squares,MaxRow,MaxCol,Row,1), nl,
   Row1 is Row+1, show_result(Squares,MaxRow,MaxCol,Row1).

show_result(_,_,MaxCol,_,Col) :- Col > MaxCol, !.
show_result(Squares,MaxRow,MaxCol,Row,Col) :-
   (memberchk(sq(Row,Col,X),Squares), !, write(X); write(' ')),
   Col1 is Col+1, show_result(Squares,MaxRow,MaxCol,Row,Col1).

% --------------------------------------------------------------------------

% Benchmark results <8-Oct-2000 14:45 hew>

% On a 330 MHz Pentium II the following results have been measured
% with SWI-Prolog version 3.3.10 for i686-linux under SuSE Linux 6.3

% ?- time(crossword('p99b.dat',0)).
% 439,743,691 inferences in 1975.34 seconds (222617 Lips)

% ?- time(crossword('p99b.dat',1)).
% 19,644,100 inferences in 76.37 seconds (257223 Lips)

% ?- time(crossword('p99b.dat',2)).
% 152,880 inferences in 0.94 seconds (162638 Lips)
```

Part II: Exam collection (MidTerm)

% Ideal Solution for the E1 Mid-term Examination: 2003

```
%  (1) a) Powerset(Set, Subset).
powerset(Set, Subset) :-
        findall(Sub, subset(Set, Sub), Subset) .

subset([], []).
subset([H|T], [H|T1]) :-
        subset(T, T1).
subset([_|T], S) :-
        subset(T, S).

% (1)     last(Item, List).
%         Using append(L1, L2, L3)

last(X, List) :-
        append(_, [X], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2) :-
        append(Tail, List, Tail2).

%         Without append
last1(X, [X]).
last1(X, [_|T]) :-
        last1(X, T).

% (1)     c)  deleteall(X, L1, L2).

deleteall(_,[],[]).
deleteall(X, [X|T], T1) :-
        !, deleteall(X, T, T1).
deleteall(X,[Y|T], [Y|T1]) :-
        deleteall(X, T, T1).

%  (1) d) palindrom(List)
palindrom(List) :-
        reverse(List, List).

reverse([], []).
reverse([H|T], List):-
        reverse(T, T1),
        append(T1, [H], List).
%  (2)  a) subsum(Set, Sum, Subset)
```

```
subsum([], 0, []).
subsum([N|List], Sum, [N|Sub]) :-
        Sum1 is Sum - N,
        subsum(List, Sum1, Sub).
subsum([N|List], Sum, Sub) :-
        subsum(List, Sum, Sub).

%   (2)  b) substitute(S, T, S1, T1)

substitute(_, [], _, []).
substitute(X, [X|T], A, [A|T1]) :-
        !, substitute(X, T, A, T1).
substitute(X, [Y|T], A, [Y|T1]) :-
        substitute(X, T, A, T1).

%  (2)  c)  split(Numbers, Postivesw, Negatives)

split([], [], []).
split([H|T], [H|T1], List) :-
        H >= 0, !,
        split(T, T1, List).
split([H|T], List, [H|T1]) :-
        split(T, List, T1).

%  (2)  d) Logic program

p(1).
p(2) :-!.
p(3).

% ?- p(X).  X=1; X=2.
% ?p(x), !, p(Y).  X=1, Y=1; X=1, Y=2.
% ?- p(X), p(Y).  X=1, Y=1; X=1, Y=2; X=2, Y=1; X=2, Y=2.

%  (3) A logic program to find who visits whom and where?
find(X) :-
        number(X, N).
find(_).

number(X, N) :-
        at(X, Y),
        phone(Y, N).
number(X, N) :-
        phone(X, N).
at(X, Z) :-
        visits(X, Y),
        at(Y, Z).
```

```
phone(ali, 3883350).
phone(sayed, 77665544).
phone(jhone, 4601213).
phone(smith, 4036655).

visits(ali, smith).
visits(ali, jhon).
visits(smith, sayed).

%  (4)  a)  fib(N, F)

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

%  (4) b) real_average(List, A)

real_average(List, A) :-
        length_One(List, N),
        sum_list(List, Sum),
        A is Sum / N.

length_One([], 0).
length_One([H|T], N) :-
        length_One(T, M1),
        N is N1 + 1.

sum_list([], 0).
sum_list([H|T], S) :-
        sum_list(T, S1),
        S is S1 + H.
```

% Ideal Solution for the E2 Mid-term Examination: 2003

```
% (1) a) Powerset(Set, Subset).

powerset(Set, Subset) :-
        findall(Sub, subset(Set, Sub), Subset).

subset([], []).
subset([H|T], [H|T1]) :-
        subset(T, T1).
subset([_|T], S) :-
        subset(T, S).

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

powerset2(Set, Subset) :-
        findall(X, append(X, _, Set), Subset).

% (1)  b) last(Item, List).
%       Using append(L1, L2, L3)

last(X, List) :-
        append(_, [X], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
        append(Tail, List, Tail2).

%       Without append
last1(X, [X]).
last1(X, [_|T]) :-
        last1(X, T).

%  (1) c)  double_lis(List, ListList)

double_list([], []).
double_list([H|T], [H,H|T1]) :-
        double_list(T, T1).

%  (1) d) reverse(List, Reverse)

reverse([], []).
reverse([H|T], List):-
        reverse(T, T1),
        append(T1, [H], List).
%  (2)  a) subset(Set, Sum, Subset)
```

```
subset2([H|T], List) :-
        member(H, List),
        subset2(T, List).
subset2([], _).

subset1([H|T], [H|T1]) :-
        prefix(T, T1).
subset1(List, [_|List1]) :-
        subset1(List, List1).

prefix([], _).
prefix([H|T], [H|T1]) :-
        prefix(T, T1).

%   (2)  b) substitute(S, T, S1, T1)
substitute(_, [], _, []).
substitute(X, [X|T], A, [A|T1]) :-
        substitute(X, T, A, T1).
substitute(X, [Y|T], A, [Y|T1]) :-
        substitute(X, T, A, T1).

%  (2)  c)  split(Numbers, Postivesw, Negatives)
split([], [], []).
split([H|T], [H|T1], List) :-
        H >= 0, !,
        split(T, T1, List).
split([H|T], List, [H|T1]) :-
        split(T, List, T1).

%  (2)  d) Logic program

p(1):- !.
p(2).
p(3).

% ?- p(X).  X=1.
% ?p(x), !, p(Y).  X=1, Y=1.
% ?- p(X), p(Y).  X=1, Y=1.

%  (3) A logic program to find who visits whom and where?
solution(Nqueens) :-
        permutation([1, 2, 3, 4, 5, 6, 7, 8], Nqueens),
        safe(Nqueens).
permutation([], []).
permutation([Head|Tail], Permlist) :-
        permutation(Tail, Permtail),
        delete(Head, Permlist, Permtail).  % insert Head in permuted Tail
delete(X, [X|T], T).
delete(X, [Y|T], [Y|T1]) :-
        delete(X, T, T1).

% safe(quuens) if Queens is a list of Y-cordinates of non-attacking quuens
safe([]).
safe([Queen|Others]) :-
        safe(Others),
        noattack(Queen, Others, 1).

noattack(_, [], _).
noattack(Y, [Y1|Ylist], Xdist) :-
        Y1 - Y =\= Xdist,
        Y - Y1 =\= Xdist,
        Dist1 is Xdist + 1,
        noattack(Y, Ylist, Dist1).

%  (4)  a)  gcd(N1, N2, F)

gcd(N, 0, N).
gcd(X, Y, N) :-
        X > Y,
        N1 is X mod Y,
        gcd(Y, N1, N).
gcd(X, Y, N) :-
        N1 is Y mod X,
        gcd(X, N1, N).

%  (4) b) real_average(List, A)
real_average(List, A) :-
        length_1(List, N),
        sum_list(List, Sum),
        A is Sum / N.
length_1([], 0).
length_1([H|T], N) :-
        length_1(T, N1),
        N is N1 + 1.
sum_list([], 0).
sum_list([H|T], S) :-
        sum_list(T, S1),
        S is S1 + H.
```

% Ideal Solution for the Mid-term Exam model (A): 2004

```
% (1) a) sum(X, Y, Z).
sum(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X + Y.
sum(X, Y, Z) :- var(X), number(Y), number(Z), X is Z - Y.
sum(X, Y, Z) :- var(Y), number(X), number(Z), Y is Z - X.
sum(X, Y, Z) :- var(X), var(Y), number(Z), findlist(Z, List),!,
                member(X, List), member(Y, List), Z is X + Y.
sum(X, Y, Z) :- !, fail.
findlist(0,[0]).
findlist(X, [X|T]):-
        Z is X - 1,
        findlist(Z, T).
% (1) a) sum(X, Y, Z). Using VP

DOMAINS
        list = integer*
        x, y, z = integer
PREDICATES
        nondeterm sum(x, y, z)
        nondeterm findlist(integer, list)
        nondeterm member(integer, list)

CLAUSES
sum(X, Y, Z) :-  bound(X), bound(Y), Z = X + Y.
sum(X, Y, Z) :-  bound(Y), bound(Z), X = Z - Y.
sum(X, Y, Z) :-  bound(X), bound(Z), Y = Z - X.
sum(X, Y, Z) :-  bound(X), free(Y), free(Z), findlist(X, List),!,
                 member(Y, List),   member(Z, List), X = Z - Y.
sum(X, Y, Z) :-  bound(Z), free(X), free(Y),  findlist(Z, List),!,
                 member(X, List),   member(Y, List), Z = X + Y.
sum(X, Y, Z) :-  bound(Y), free(X), free(Z),  findlist(Y, List), !,
                 member(X, List),   member(Z, List), Y = Z - X.
sum(_, _, _) :-  !, fail.

findlist(0,[0]).
findlist(X, [X|T]):-
        Z = X - 1,
        findlist(Z, T).



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

GOAL
        sum(Y, X, 10).
% (1)  b) last(Item, List).
%       i) Using append(L1, L2, L3)

last(X, List) :-
        append(_, [X], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :- !,
        append(Tail, List, Tail2).

%       ii) Without append
last1(X, [X]).
last1(X, [_|T]) :-
        last1(X, T).

% (1)    c)  deleteall(X, L1, L2).

deleteall(_,[],[]).
deleteall(X, [X|T], T1) :-
        !, deleteall(X, T, T1).
deleteall(X,[Y|T], [Y|T1]) :-
        deleteall(X, T, T1).

%  (1) d) palindrom(Atom)

begin :-
                read(X),
                (X == stop, !;
                test_palindrome(X), begin).
test_palindrome(X) :-
                name(X, Nx),
                palindrom(Nx),
                write(X), write(' is a palindrome'), nl, !.
test_palindrome(X) :-
                write(X), write(' is not a palindrome'), nl.

palindrom(List) :-
        reverse(List, List).
reverse([], []).
reverse([H|T], List):-
        reverse(T, T1),
        append(T1, [H], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
        append(Tail, List, Tail2).
 %  (1)  e) Logic program
```

```
p(1).
p(2):- !.
p(3).
% ?- p(X).  X=1; X=2.
% ?- p(x), !, p(Y).  X=1, Y=1; X=1, Y=2.
% ?- p(X), p(Y).  X=1, Y=1; X=1, Y=2; X=2, Y=1; X=2, Y=2.

%   (2)  a) wo-place predicate termtype(Term,Type) that takes a term and gives
%        back the type(s) of that term (atom, number, constant, variable etc.).
termtype(Term,atom)             :- atom(Term).
termtype(Term,number)                   :- number(Term).
termtype(Term,constant)         :- atomic(Term). % atom(Term); number(Term).
termtype(Term,variable)         :- var(Term).
termtype(Term,complex_term)     :- Term =.. [H|T], T \== [].
termtype(Term, term).

%   (2)  b) b) Predict the results of these unification queries.
% ?- a(b,c) = a(X,Y).
% X = a, Y = b
% ?- a(X,c(d,X)) = a(2,c(d,Y)).
%  X = 2, Y = 2
% ?- [a,b,c,d] = [H|T].
% H = a, T = [b, c, d]
% ?- [a,[b,c,d]] = [H|T].
% H = a, T = [[b, c, d]]
% ?- [] = [H|T].
% No match exist
% ?- [[a,b,c],[d,e,f],[g,h,i]] = [H|T].
% H = [a, b, c], T = [[d, e, f], [g, h,i]]

%  (2) c) For each of the following pairs of terms, state whether or not they unify.
% If they do not unify, give an explanation. If they do unify, give the most general
% unifier for the two terms and the most general common instance defined by the
% substitution. You do not need to show the derivation, just the final solutions.
% ?- japan(X,sushi(Y,white,Y,W)) = japan(wasabi,sushi(green,Z,green,W)).
%  X = wassabi
%  Y = green
%  W = _00AC
%  Z = white
% The output from lp compiler
% X = wasabi
% Y = green
% W = _G570
% Z = white
% ?- italy(pizza(A,C),lasagne(C,pasta),bake(B,C)) =
%        italy(pizza(cheese,size(X)),B,bake(lasagne(size(X),M),Y)).
%  A = cheese
%  C = size(_00C0)
%  B = lasagne(size(_00C0), pasta)
%  X = _00C0
%  M = pasta
%  Y = size(_00C0)
% The output from lp compiler
% A = cheese
% C = size(_G808)
% B = lasagne(size(_G808), pasta)
% X = _G808
% M = pasta
% Y = size(_G808)

%  (2)  d)  split(Numbers, Postivesw, Negatives)
%        i) With CUT (!)

split([], [], []).
split([H|T], [H|T1], List) :-
        H >= 0, !,
        split(T, T1, List).
split([H|T], List, [H|T1]) :-
        split(T, List, T1).

%        ii) Without CUT
splitw([], [], []).
splitw([H|T], [H|T1], List) :-
        H >= 0,
        splitw(T, T1, List).
splitw([H|T], List, [H|T1]) :-
        splitw(T, List, T1).
%  (3)  logic program to answer " What course would Steve like?
like(steve, X) :-
                 course(X, easy).
course(science, hard).
course(X, easy) :-
                course1(computer, X).
course1(computer, 'Logic programming CSC 353').

run(ZZ)  :-
        like(steve, ZZ).

%  (4)  a) a) exp(X, Y, Z).
exp(X, 0, 1) :- !.
exp(X, Y, Z) :-
                even(Y),
                R is Y / 2,
                P is X * X, !,   exp(P, R, Z).
```

```
exp(X, Y, Z) :-
                T is Y - 1, exp(X, T, Z1),
                Z is Z1 * X.
even(X) :-
                R is X mod 2, R == 0.

%  (4)  b) factorial(N, F).

fact(0, 1) :- !.
fact(N, F) :-
                M is N - 1,
                fact(M, F1),
                F is  N * F1.
```

% Ideal Solution for the E1 Mid-term Exam. Model (B): 2004

```
%  (1) a) palindrom(List)
palindrom(List) :-
        reverse(List, List).

reverse([], []).
reverse([H|T], List):-
        reverse(T, T1),
        append(T1, [H], List).

% (1)   b)   last(Item, List).
%       i) Using append(L1, L2, L3)
last(X, List) :-
        append(_, [X], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :- !,
        append(Tail, List, Tail2).

%       ii) Without append
last1(X, [X]).
last1(X, [_|T]) :-
        last1(X, T).

%  (1) c)  double_lis(List, ListList)
double_list([], []).
double_list([H|T], [H,H|T1]) :-
        double_list(T, T1).

%  (1) d) increment(X,Y)
increment(X, Y) :- number(X), number(Y), Y is X + 1.
increment(X, Y) :- number(X), Y is X + 1.
increment(X, Y) :- number(Y), X is Y - 1.
increment(X, Y) :- !, fail.

%  (1) e) two-place predicate termtype(Term,Type) that takes a term and gives
%        back the type(s) of that term (atom, number, constant, variable etc.).
termtype(Term,atom)           :- atom(Term).
termtype(Term,number)         :- number(Term).
termtype(Term,constant)       :- atomic(Term). % atom(Term); number(Term).
termtype(Term,variable)       :- var(Term).
termtype(Term,complex_term)     :- Term =.. [H|T], T \== [].
termtype(Term, term).

% (2) a) predict the result of these unification queries.
% ?- a(X,Y) = a(b(c,Y),Z).
%  X = b(c, _0098)
%  Y = _0098
%  Z = _0098
% ?- tree(left, root, Right) = tree(left, root, tree(a, b, tree(c, d, e))).
%  Right = t(a,b,t(c,d,e))
% ?- [a,b,c,d] = [H|T].
%  H = a, T = [b,c,d]
% ?- [a] = [H|T].
%  H = a, T = []
% ?- [a(X,c(d,Y)), b(2,3), c(d,Y)] = [H|T].
%  X = _0084
%  Y = _0098
%  H = a(_0084, c(d, _0098))
%  T = [b(2,3), c(d,_0098)]

%  (2) b) For each of the following pairs of terms, state whether or not
% they unify.  If they do not unify, give an explanation. If they do unify,
% give the most general unifier for the two terms, and the most general
% common instance defined by the substitution.
% ?- apan(X,sushi(Y,white,Y,W)) = japan(wasabi,sushi(green,Z,green,W)).
%  X = wassabi
%  Y = green
%  W = _00AC
%  Z = white
% ?- italy(pizza(A,C),lasagne(X,pasta),bake(B,C))=italy(pizza(cheese,
%    size(X)),B,bake(lasagne(shape(X),M),Y)).
%  A = cheese
%  C = size(_00C0)
%  B = lasagne(size(_00C0), pasta)
%  X = _00C0
%  M = pasta
%  Y = size(_00C0)

%  (2)  c)  split(Numbers, Postivesw, Negatives)
split([], [], []).
split([H|T], [H|T1], List) :-
          H >= 0, !,
          split(T, T1, List).
split([H|T], List, [H|T1]) :-
          split(T, List, T1).
%  (2)  d) Logic program
p(1).
p(2).
p(3):- !.

% ?- p(X).  X=1; X=2.
% ?- p(x), !, p(Y).  X=1, Y=1; X=1, Y=2; X = 1, Y = 3.
% ?- p(X), p(Y).  X=1, Y=1; X=1, Y=2; X= 1, Y = 3; X=2, Y=1; X=2, Y=2;
%    X = 2, Y = 3; X = 3, Y = 1, X = 3, Y = 2; X = 3, Y = 3.

%  (3) Represent the following in Prolog
% Butch is a killer.
% Mia and Marcellus are married.
% Zed is dead.
% Jules eats anything that is nutritious or tasty.
% Marcellus kills everyone who gives Mia a footmassage.
% Mia loves everyone who is a good dancer.

killer("Butch").
marries("Mia", "Marcellus").
dead("Zed").
eats("Jules", X) :-
                tasty(X), nutrious(X).
kills("Marcellus", X):-
                gives(X, "Mia", footmassage).
loves("Mia", X) :-
                good_dancer(X).

% (4)   a) abs(X, Y)

abs(X,X) :-
                X >=0, !.
abs(X, Y) :-
                Y is -X.

%  (4)  b)  fib(N, F)
fib(0, 1) :- !.
fib(1,1) :- !.
fib(N, F) :-
        N1 is N -1,
        fib(N1, F1),
        N2 is N - 2,
        fib(N2, F2),
        F is F1 + F2.
```

Part III: Exam collection (Final)

% Ideal Solution for the Final Examination 2003

```
% (1) a) Quick Sort
quicksort([], []).
quicksort([X|Tail], Sorted) :-
                split(X, Tail, Small, Big),
                quicksort(Small, SortedSmall),
                quicksort(Big, SortedBig),
                append(SortedSmall, [X|SortedBig], Sorted).

split(X, [], [], []).
split(X, [H|T], [H|T1], List) :-
        X > H, !,
        split(X, T, T1, List).
split(X, [H|T], List, [H|T1]) :-
        split(X, T, List, T1).


% (1) b) Powerset(Set, Subset).

powerset(Set, Subset) :-
        findall(Sub, subset(Set, Sub), Subset).

subset([], []).
subset([H|T], [H|T1]) :-
        subset(T, T1).
subset([_|T], S) :-
        subset(T, S).

%  (1)  c)  substitute(S, T, S1, T1)

substit(Term, Term, Term1, Term1) :- !.
substit(_, Term, _, Term) :-
                atomic(Term), !.
substit(Sub, Term, Sub1, Term1) :-
                Term =.. [F|Args],
                sublist(Sub, Args, Sub1, Args1),
                Tem1 =..[F|Args1].
sublist(_, [], _, []).
sublist(Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
                substit(Sub, Term, Sub1, Term1),
                sublist(Sub, Terms, Sub1, Terms1).

substitute1(_, [], _, []).
substitute1(X, [X|T], A, [A|T1]) :-     substitute1(X, T, A, T1).
```

```
substitute1(X, [Y|T], A, [Y|T1]) :-
        substitute1(X, T, A, T1).

%       (1)     d) palindrom(Atom)

begin :-
                read(X),
                (X == stop, !;
                test_palindrome(X), begin).
test_palindrome(X) :-
                name(X, Nx),
                palindrom(Nx),
                write(X), write(' is a palindrome'), nl, !.
test_palindrome(X) :-
                write(X), write(' is not a palindrome'), nl.

palindrom(List) :-
        reverse(List, List).

reverse([], []).
reverse([H|T], List):-
        reverse(T, T1),
        append(T1, [H], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
        append(Tail, List, Tail2).

%=================================================================
%       (2)     a) height(BinaryTree, Height).

height(nil, 0).
height(t(Left, Root, Right), H) :-
                height(Left, Hl),
                height(Right, Hr),
                max(Hl, Hr, Mh),
                H is Mh + 1.
max(A, B, A) :-
                A >= B, !.
max(A, B, B).

%       (2)     b) maxlist(List, Max).

maxlist([X], X).
maxlist([X, Y|Rest], Max) :-
                maxlist([Y|Rest], MaxRest),
                max(X, MaxRest, Max).
```

```
%       (2)     c) last(Item, List).
%       Using append(L1, L2, L3)
last(X, List) :-
        append(_, [X], List).

%       Without append
last1(X, [X]).
last1(X, [_|T]) :-
        last1(X, T).

%       (2)     d)  copy (Term, Copy).

copy_term(Term, Copy) :-
                asserta(term_to_copy(Term)),
                retract(term_to_copy(Copy)).

copy_term1(Term, Copy) :-

                findall(X, X = Term, [Copy]).

%==================================================================
%       (3)     a) findallterms(Term).
findterm(Term) :-
                see(f),
                read(Term),
                write(Term) ,
                seen;
                findterm(Term).

findallterm(Term) :-
                see(f),   read(CTerm),
                process(CTerm, Term), seen.
process(end-of-file, _) :- !.
process(CTerm, Term) :-
                ( not (CTerm = Term), !;
                write(CTerm), nl),
                findallterm(Term).

%       (3)     b) show(Tree).
show(Tree) :-
                show2(Tree, 0).
show2(nil, _).
show2(t(Left, X, Right), Ind) :-
                Ind2 is Ind + 2,
                show2(Right, Ind2),
                tab(Ind), write(X), nl,
                show2(Left, Ind2).
```

```
%       (3)     c)difference(Set1, Set2, SetDifference).
difference([], _, []).
difference([X|T], L1, L) :-
                member(X, L1), !,
                difference(T, L1, L).
difference([X|T], L2, [X|L]) :-
                difference(T, L2, L).

% ==================================================================
%       (4)     a) Tower of Hanoi Puzzle

:- module towers.
:- public towers/0.

towers :-
        repeat,
        write('Number of rings (or ctl-c to end): '),
        read(X),
        hanoi(X),
        fail.

hanoi(N) :- move(N,left,center,right), !.

move(0,_,_,_) :- !.
move(N,A,B,C) :-
        M is N-1,
        move(M,A,C,B),
        inform(A,B),
        move(M,C,B,A).

inform(A,B) :- write([move,disk,from,A,to,B]), nl.

%       (4)     b)  depthfirst(Path, Node, Solution).

solve(Node, Solution)  :-
                depthfirst([], Node, Solution).

depthfirst(Path, Node, [Node|Path]) :-
                        goal(Node).
depthfirst(Path, Node, Sol) :-
                        s(Node, Node1),
                        not member(Node1, Path),
                        depthfirst([Node|Path], Node1, Sol).
goal(z).
s(a, b).
s(b, c).
s(c, z).
```

```
% ============================================================
%       (5)     a)      getsentence(WodList).
go(W) :- getsentence(W).

getsentence(Word) :-
                get0(Char),
                getrest(Char, Word).
getrest(46, []) :- !.               % End of sentence
getrest(32, Word) :- !, % ASCII for blank
                getsentence(Word).
getrest(Letter, [Word|Wordlist]) :-
                getletters(Letter, Letters, Nextchar),
                name(Word, Letters),
                getrest(Nextchar, Wordlist).
getletters(46, [], 46) :- !.        % End of word : 46 = full stop
getletters(32, [], 32) :- !.        % End of word : 32 = blank
getletters(Let, [Let|Letters], Nextchar) :-
                get0(Char),
                getletters(Char, Letters, Nextchar).

%       (5)     b) How to represent graphs in Prolog
% a)  connect(a, b).  connect(b, c).  ....  arc(s, t, 3).arc(t, u, 2). ...
% b) G1 = graph( [a, b, c, d], [e(a,b), e(b,d), e(b, c), e(c,d)]).
%    G2 = graph( [s, t, u, v], (a(s, t,3), a(t,v,1), a(u, t, 2)...]).
% c) G1 = [a ->[b], b->[a, c,d], c -> [b, d], d -> [b, c]]
%    G2 = [ s - > [t/3], t -> [u/5, v/1], u -> [t/2], v -> [u/2]]

%       (5)     c) path(A, Goal, Graph, Path).
path(A, Z, Graph, Path) :-
                        path1(A, [Z], Graph, Path).
path1(A, [A|Path1], _, [A|Path1]).
path1(A, [Y|Path1], Graph, Path) :-
                        adjacent(X, Y, Graph),
                        not member(X, Path1),
                        path1(A, [X, Y|Path1], Graph, Path).
adjacent(X, Y, graph(Node, Edges)) :-
                        member(e(X, Y), Edges) ;
                        memeber(e(Y, X), Edges).
e(X, Y).

%       (5)     d)  addleaf(Tree, X, NewTree).
addleaf(nil, X, t(nil, X, nil)).
addleaf(t(Left, X, Right), X, t(Left, X, Right)).
addleaf(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
                        Root > X,  addleaf(Left, X, Left1).
addleaf(t(Left, Root, Right), X, t(Left, Root, Right1)) :-
                        X > Root, addleaf(Right, X, Right1).
```

```
%=====================================================
%       (6)     a) exp(X, Y, Z).
exp(X, 0, 1) :- !.
exp(X, Y, Z) :-
                even(Y),
                R is Y / 2,
                P is X * X,
                exp(P, R, Z).
exp(X, Y, Z) :-
                T is Y - 1, exp(X, T, Z1),
                Z is Z1 * X.
even(X) :-
                R is Y mod 2, R == 0.

%       (6)     b) factorial(N, F).
fact(0, 1) :- !.
fact(N, F) :-
                M is N - 1,
                fact(M, F1),
                F is  N * F1.
%       (6)     c) abs(X, Y)
abs(X,X) :-
                X >=0, !.
abs(X, Y) :-
                Y is -X.

%       (6)     d) A logic program to answer " What course would Steve like?
like(steve, X) :-
                 course(X, easy).
course(science, hard).
course(X, easy) :-
                course1(computer, X).
course1(computer, 'Logic programming CSC 353').

run(ZZ)  :-
        like(steve, ZZ).
%=======================================================
%               More Utilites
%=======================================================
%  real_average(List, A)
real_average(List, A) :-
        length_1(List, N),
        sum_list(List, Sum),
        A is Sum / N.
length_1([], 0).
length_1([H|T], N) :-
        length_1(T, N1),  N is N1 + 1.
```

```
sum_list([], 0).
sum_list([H|T], S) :-
        sum_list(T, S1),
        S is S1 + H.

%   deleteall(X, L1, L2).
deleteall(_,[],[]).
deleteall(X, [X|T], T1) :-
        !, deleteall(X, T, T1).
deleteall(X,[Y|T], [Y|T1]) :-
        deleteall(X, T, T1).

%  subsum(Set, Sum, Subset)
subsum([], 0, []).
subsum([N|List], Sum, [N|Sub]) :-
        Sum1 is Sum - N,
        subsum(List, Sum1, Sub).
subsum([N|List], Sum, Sub) :-
        subsum(List, Sum, Sub).

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

%  gcd(N1, N2, F)
gcd(N, 0, N).
gcd(X, Y, N) :-
        X > Y,
        N1 is X mod Y,
        gcd(Y, N1, N).
gcd(X, Y, N) :-
        N1 is Y mod X,
        gcd(X, N1, N).

member(X,[X|_]).
member(X,{_|T]) :-
        member(X,T).

powerset2(Set, Subset) :-
        findall(X, append(X, _, Set), Subset).
```

```
%  reverse(List, Reverse)
reverse([], []).
reverse([H|T], List):-
        reverse(T, T1),
        append(T1, [H], List).
%  A logic program for 8 - Queens problem
solution(Nqueens) :-
        permutation([1, 2, 3, 4, 5, 6, 7, 8], Nqueens),
        safe(Nqueens).
permutation([], []).
permutation([Head|Tail], Permlist) :-
        permutation(Tail, Permtail),
        delete(Head, Permlist, Permtail).  % insert Head in permuted Tail
delete(X, [X|T], T).
delete(X, [Y|T], [Y|T1]) :-
        delete(X, T, T1).
% safe(quuens) if Queens is a list of Y-cordinates of non-attacking quuens
safe([]).
safe([Queen|Others]) :-
        safe(Others),
        noattack(Queen, Others, 1).
noattack(_, [], _).
noattack(Y, [Y1|Ylist], Xdist) :-
        Y1 - Y =\= Xdist,
        Y - Y1 =\= Xdist,
        Dist1 is Xdist + 1,
        noattack(Y, Ylist, Dist1).

%  A logic program to find who visits whom and where?
find(X) :-
        number(X, N).
find(_).
number(X, N) :-
        at(X, Y),
        phone(Y, N).
number(X, N) :-
        phone(X, N).
at(X, Z) :-
        visits(X, Y),
        at(Y, Z).
phone(ali, 3883350).
phone(sayed, 77665544).
phone(jhone, 4601213).
phone(smith, 4036655).
visits(ali, smith).
visits(ali, jhon).
visits(smith, sayed).
```

```
%=================================================================
%       Ideal Sol;ution for the Corrective Examination 2003-2004
%=================================================================
% (1) a)   Design and test  prune(A,B) which is intended to remove multiple occurrences
%               of elements from A to produce result B. For example,
%                       ?- prune([a,1,b,2,a,3,a,4,b],B).
%                       B = [a,1,b,2,3,4]
%                Try to make it so that B has remaining elements in the order that they
%                occurred in A.
%                               VERSION 1
%                       ?- prune([a,1,b,2,a,3,a,4,b],B).
proune([], L, L).
proune([H|T], List, L):-
                member(H, List),        !,
                proune(T, List, L).
proune([H|T], L1, L):-
                append(L1, [H], L2), !,
                proune(T, L2, L).

member(_,[]):- fail.
member(X,[X|_]).
member(X, [_|T]):-
                member(X, T).

append([], L, L).
append([H|T], L, [H|T1]) :-
                append(T, L, T1).

%                               VERSION 2
%                       ?- prune([a,1,b,2,a,3,a,4,b],B).
proune2([], []).
proune2([H|T], List):-
        proune2(T, List1),
      not member(H, List1), !,
        append([H], List1,  List).
proune2([H|T], List):-
        proune2(T, List).
%=================================================================
%       b)   Suppose we represent sets (no duplicated elements) ranging over integers as
%            lists of integers. Write Prolog predicates for performing the following set
%            operations: union (S1,S2): returns the set-union of sets S1 and S2.
union([X|Y],Z,W) :-
                        member(X,Z),
                        union(Y,Z,W).
union([X|Y],Z,[X|W]) :-
                        not member(X,Z),  union(Y,Z,W).
union([],Z,Z).
```

```
%==================================================================
%       c)  Write a procedure that will sort a list using bubble sort algorithm.

bubble_sort(List, Sorted) :-
                        swap(List, List1),  !,
                        bubble_sort(List1, Sorted).
bubble_sort(Sorted, Sorted). %List is already sorted

swap([X,Y|Rest],[Y,X|Rest]):-% swap first two elements
                        gt( X, Y).
swap([Z|Rest], [Z|Rest1]) :- % swap elements in tail
                        swap(Rest, Rest1).
gt(X, Y) :-
                        X > Y.
%============================================================
%d)  Define the procedure
%               substitute (Subterm, Term, Subterm1, Term1).
%   To substitute all occurrences of Subterm in Term with Subterm1, then we get Term1.
%                substitute(S, T, S1, T1)

substit(Term, Term, Term1, Term1) :- !.
substit(_, Term, _, Term) :-
                        atomic(Term), !.
substit(Sub, Term, Sub1, Term1) :-
                        Term =.. [F|Args],
                        sublist(Sub, Args, Sub1, Args1),
                        Tem1 =..[F|Args1].
sublist(_, [], _, []).
sublist(Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
                        substit(Sub, Term, Sub1, Term1),
                        sublist(Sub, Terms, Sub1, Terms1).

substitute1(_, [], _, []).
substitute1(X, [X|T], A, [A|T1]) :-
                        substitute1(X, T, A, T1).
substitute1(X, [Y|T], A, [Y|T1]) :-
                        substitute1(X, T, A, T1).

%==================================================================
%   (2) a)  A palindrome is a word or phrase that spells the same forwards and backwards.
%          For example, `rotator', `eve', and `nurses run' are all palindromes. Write a
%          predicate palindrome (List), which checks whether List is a palindrome.
palindrom(List) :-
                        reverse(List, List), !.
reverse([], []).
reverse([H|T], List):-
                        reverse(T, T1),  append(T1, [H], List).
```

```
append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
                        append(Tail, List, Tail2).

%=====================================================================
%       b)  Write a two-place predicate termtype(Term,Type) that takes a term and gives
%          back the type(s) of that term (atom, number, constant, variable etc.). The types
%          should be given back in the order of their generality. The predicate should, e.g.,
%          behave in the following way.
%               ?- termtype(Vincent,variable).
%               Yes
%               ?- termtype(mia,X).
%               X = atom ;
%               X = constant ;
%               X = term ;
%               No
%               ?- termtype(dead(zed),X).
%               X = complex_term ;
%               X = term ;
%               no

termtype(Term,atom)             :- atom(Term).
termtype(Term,number)                   :- number(Term).
termtype(Term,constant)         :- atomic(Term). % atom(Term); number(Term).
termtype(Term,variable)         :- var(Term).
termtype(Term,complex_term)     :- Term =.. [H|T], T \== [].
termtype(Term, term).

%=====================================================================
%       c)  Define the procedure
%               height(BinaryTree, Height).
%          to find the height of a binary tree. Assume that the height of empty tree is 0,
%          and that of one element is 1.
height(nil, 0).
height(t(Left, Root, Right), H) :-
                        height(Left, Hl),
                        height(Right, Hr),
                        max(Hl, Hr, Mh),
                        H is Mh + 1.
max(A, B, A) :-
                        A >= B, !.
max(A, B, B).

%=====================================================================
%       d)  Let f be a file of terms. Write a procedure
%                       findallterms(Term).
%          That displays on the terminal all terms in f that matches Term.
```

```
 findterm(Term) :-
                        see(f),
                        read(Term),
                        write(Term) ,
                        seen;
                        findterm(Term).

findallterm(Term) :-
                        see(f),
                        read(CTerm),
                        process(CTerm, Term), seen.

process(end-of-file, _) :- !.
process(CTerm, Term) :-
                        ( not (CTerm = Term), !;
                        write(CTerm), nl),
                        findallterm(Term).
%================================================================
% (3) a)  Write a logic program for solving the following puzzle.
%       Four people each use a different editor, UNIX variant and programming language.
%               • People:  Daisy (female),   Steve (male), Llewellyn (male) and
%                 Gertrude (female).
%               • Editors:  emacs,  vi,  pico, and joe
%               • Unix variants:  Solaris,  AIX, Linux and  BSD
%               • Languages:  C++ (OO),  Java (OO),  Perl and  Lisp
%               Using the clues that follow, find out who uses what.
%               • The emacs user is a man who likes Lisp.
%               • The woman who uses pico likes BSD, but does not use object-oriented
%                 (OO) languages.
%               • The C++ programmer uses AIX and joe.
%               • Daisy loves Java, but thinks Solaris is a joke of an operating system.
%               • Llewellyn cannot stand AIX.

male('Steve').
male('Llewellyn').
female('Daisy').
female('Gertude').
editor(emac).
editor(vi).
editor(pico).
editor(joe).

unix_v('Solaries').
unix_v('AIX').
unix_v('Linux').
unix_v('BSD').
```

```
language('Java', oo).
language('C++', oo).
language('Perl', net).
language('Lisp', ai).

likes('Steve', 'Lisp').
likes('Daisy', 'Lisp').
likes('Daisy', 'BSD').

editor_user(X, Y) :-
                        male(X),
                        editor(Y) , Y == emac,
                        language('Lisp', _),
                        likes(X, 'Lisp').
editor_user(X, Y):-
                        female(X),
                        editor(Y), Y == pico, unix_v('BSD'),
                        likes(X, L),        !,
                        not language(L, oo),
                        likes(X, 'BSD').
editor_user(_, _).
cpp_programmer(X, Y, Z) :-
                        (male(X), X \== 'Llewellyn'; female(X)),
                        editor(Y), Y ==joe,
                        unix_v(Z), Z == 'AIX'.
love(X, L) :-
                        female(X), X = 'Daisy',
                        language(L, oo), L == 'Java',
                        unix_v('Solaries').
stand(X, Y) :-
                        male(X), ((X == 'Llewellyn', !,
                        unix_v(Y), Y \= 'AIX'); unix_v(Y)).

who_use_what(X, Y):-
                        editor_user(X, Y).
who_use_what(X, Y):
                        love(X, Y), X \= Y.
who_use_what(X, Y):-
                        stand(X, Y).
who_use_what(X, Y):-
                        cpp_programmer(X, Y, _).
who_use_what(X, Z):-
                        cpp_programmer(X, _, Z).
%===============================================================
%       b)  Write a breadth-first search procedure
%               breadthfirst([[Start]], Solution).
%           Let Solution be represented as a list of nodes in inverse order from Start to a
%           goal, so that the goal node is the head of Solution.
%                  solve(Start, Solution):
%           Solution is a path (in reverse order) from start to a goal

solve(Start, Solution) :-
                              breadthfirst([[Start]], Solution).
%           breadthfirst([Path1, Path2, …], Solution):
%           Solution is an extension to a goal of one of paths
breadthfirst([[Node|Path]|_], [Node|Path]):-
                              goal(Node).
breadthfirst([Path|Paths], Solution):-
                              extend(Path, NewPaths),
                              append(Paths, NewPaths,Paths1),
                              breadthfirst(Paths1, Solution).

extend([Node|Path], NewPaths) :-
                              bagof([NewNode, Node|Path], (s(Node, NewNode),
not member(NewNode, [Node|Path])), NewPaths), !.
extend(Path, [ ]). % bagof failed: Node has no successor

%=====================================================================
%(4)  a)  Write a procedure to transform a natural language sentence, taken as a string
%         of characters with some spaces between them, into a list whose elements are its
%          words. For example: Wordlist = [tom, is, ironing, the, shirt]
%          If the input sentence is "tom is ironing the shirt."
go(W) :- getsentence(W).
getsentence(Word) :-
                              get0(Char),
                              getrest(Char, Word).
getrest(46, []) :- !.         % End of sentence
getrest(32, Word) :- !, % ASCII for blank
                              getsentence(Word).
getrest(Letter, [Word|Wordlist]) :-
                              getletters(Letter, Letters, Nextchar),
                              name(Word, Letters),
                              getrest(Nextchar, Wordlist).
getletters(46, [], 46) :- !.  % End of word : 46 = full stop
getletters(32, [], 32) :- !.  % End of word : 32 = blank
getletters(Let, [Let|Letters], Nextchar) :-
                              get0(Char),
                              getletters(Char, Letters, Nextchar).

%====================================================================
%       b)  Explain how we can represent graphs in logic programming.
%       i-  connect(a, b).  connect(b, c).  ....  arc(s, t, 3).arc(t, u, 2). ...
%     ii-  G1 = graph( [a, b, c, d], [e(a,b), e(b,d), e(b, c), e(c,d)]).
%          G2 = graph( [s, t, u, v], (a(s, t,3), a(t,v,1), a(u, t, 2)...]).
%    iii-  G1 = [a ->[b], b->[a, c,d], c -> [b, d], d -> [b, c]]
%          G2 = [ s - > [t/3], t -> [u/5, v/1], u -> [t/2], v -> [u/2]]

%=================================================================
%       c)  Define the procedure
%                       path(A, Z, Graph, Path, Cost).
%           Finding an acyclic path, Path with Cost from A to Z in a Graph..
%                 path(A, Goal, Graph, Path).

path(A, Z, Graph, Path) :-
                                path1(A, [Z], Graph, Path).
path1(A, [A|Path1], _, [A|Path1]).
path1(A, [Y|Path1], Graph, Path) :-
                                adjacent(X, Y, Graph),
                                not member(X, Path1),
                                path1(A, [X, Y|Path1], Graph, Path).
adjacent(X, Y, graph(Node, Edges)) :-
                                member(e(X, Y), Edges) ;
                                memeber(e(Y, X), Edges).
e(X, Y).

%                 path(A, Goal, Graph, Path, Cost).

%       Path is an acyclic path with cost Cost in Graph

path (A, Z, Graph, Path, Cost) :-
                        path1(A, [Z], 0, Graph, Path, Cost).
path1(A, [A|Path1], Cost1, _, [A|Path1], Cost1).

path1(A, [Y|Path1], Cost1, Graph, Path, Cost) :-
                        adjacent (X, Y, CostXY, Graph),
                        not member (X, Path1),  % No cycle condition
                        Cost2 is Cost1 + CostXY,
                        path1(A, [X, Y|Path1], Cost2, Graph, Path, Cost).

adjacent (X, Y, CostXY, graph (Nodes, Edges)):-
                        member (e (X, Y, CostXY), Edges) ; % Or
                        member (e (Y, X, CostXY), Edges).
member (X, [X|_]).
member (X,[_|T]):-
                        member( X, T).

%=================================================================
%       d) Write a procedure that is used to insert an item into a binary dictionary
%          (i.e., inserting an item at any level into the binary dictionary).
%                add(Tree, X, NewTree).
%          inserting X at any level of binary dictionary Tree gives NewTree
```

```
add(Tree, X, NewTree) :-
                        addroot(Tree, X, NewTree). % add X as new root
add(t(L, Y, R), X, t(L1, Y, R)):- % insert X into left subtree
                        gt(Y, X),
                        add(L, X, L1).
add(t(L,Y, R), X, t(L, Y, R1)):-%insert X into right subtree
                        gt(X, Y),
                        add(R, X, R1).

%               addroot(Tree, X, NewTree)
%               inserting X as the root into Tree gives NewTree
addroot(nil, X, t(nil, X, nil)).  % insert into empty tree
addroot(t(L,Y, R), X, t(L1, X, t(L2, Y, R))) :-
                        gt(Y, X),
                        addroot(L, X, t(L1, X, L2)).
addroot(t(L,Y, R), X, t(t(L, Y, R1), X, R2)) :-
                        gt(X, Y),
                        addroot(R, X, t(R1, X, R2)).

%===================================================================
%       (5) a) Write a logic program defining the relationship
%                       gcd(N1,N2,F).
%            To find the greatest common divisor of two integers N1 and N2.

gcd(N, 0, N).
gcd(X, Y, N) :-
                        X > Y,
                        N1 is X mod Y,
                        gcd(Y, N1, N), !.
gcd(X, Y, N) :-
                        N1 is Y mod X,
                        gcd(X, N1, N), !.

%===================================================================
%          b) Write a logic program to define the relationship
%                       real_average(List, A).
%               That calculates the average value of all elements in a list of integers

real_average(List, A) :-
                        length_1(List, N),
                        sum_list(List, Sum),
                        A is Sum / N.

length_1([], 0).
length_1([H|T], N) :-
                        length_1(T, N1),
                        N is N1 + 1.
```

```
sum_list([], 0).
sum_list([H|T], S) :-
                        sum_list(T, S1),
                        S is S1 + H.
%=================================================================
%          c)  Write a logic program considering the following sentences:
%                       John likes all kinds of food
%                       Apples are food
%                       Chicken is food
%                       Anything anyone eats and isn't killed by is food
%                       Bill eats peanuts and is still alive
%                       Sue eats every thing Bill eats.
%               and try to answer the question "what does Sue like?"

food (apple).
food (chicken).
eat (Sue, X) :-
                eat (Bill,X).
eat (Bill, peanut):-
                 not died( Bill).

eat (X, Y) :-
                food(Y),
                not killed (X, Y),

like (john, X):-
                food (X).
%==================================================================
%         d)  Consider the following facts:
%                       p( a,  b ).
%                       p(X,  d ).
%                       p(e,  Y ).
%             List all answers to the following queries:
%                       ?- p(X, Y), !.
%                       X = a, Y = b

%                       ?- p(c, Y), !.
%                       Y = d

%                       ?- p(X,  f ), !.
%                       X = e
```

```
%==============================================================
%       Ideal Sol;ution for the Corrective Examination 2003-2004
%==============================================================
% (1) a)   Design and test  prune(A,B) which is intended to remove multiple occurrences
%          of elements from A to produce result B. For example,
%                      ?- prune([a,1,b,2,a,3,a,4,b],B).
%                      B = [a,1,b,2,3,4]
%                Try to make it so that B has remaining elements in the order that they
%                occurred in A.
%                                  VERSION 1
%                      ?- prune([a,1,b,2,a,3,a,4,b],B).
proune([], L, L).
proune([H|T], List, L):-
                member(H, List),        !,
                proune(T, List, L).
proune([H|T], L1, L):-
                append(L1, [H], L2), !,
                proune(T, L2, L).
member(_,[]):- fail.
member(X,[X|_]).
member(X, [_|T]):-
                member(X, T).
append([], L, L).
append([H|T], L, [H|T1]) :-
                append(T, L, T1).

%                                  VERSION 2
%                      ?- prune([a,1,b,2,a,3,a,4,b],B).
proune2([], []).
proune2([H|T], List):-
        proune2(T, List1),
     not member(H, List1), !,
        append([H], List1,  List).
proune2([H|T], List):-
        proune2(T, List).
%=============================================================
%       b)   Suppose we represent sets (no duplicated elements) ranging over integers as
%            lists of integers. Write Prolog predicates for performing the following set
%            operations: union (S1,S2): returns the set-union of sets S1 and S2.
union([X|Y],Z,W) :-
                        member(X,Z),
                        union(Y,Z,W).
union([X|Y],Z,[X|W]) :-
                        not member(X,Z),
                        union(Y,Z,W).
union([],Z,Z).
```

```
%=====================================================================
%       c)  Write a procedure that will sort a list using bubble sort algorithm.

bubble_sort(List, Sorted) :-
                        swap(List, List1),  !,
                        bubble_sort(List1, Sorted).
bubble_sort(Sorted, Sorted). %List is already sorted

swap([X,Y|Rest],[Y,X|Rest]):-% swap first two elements
                        gt( X, Y).
swap([Z|Rest], [Z|Rest1]) :- % swap elements in tail
                        swap(Rest, Rest1).
gt(X, Y) :-
                        X > Y.
%=================================================================
%       d)  Define the procedure
%                       substitute (Subterm, Term, Subterm1, Term1).
%           To substitute all occurrences of Subterm in Term with Subterm1, then we get Term1.
%                 substitute(S, T, S1, T1)
substit(Term, Term, Term1, Term1) :- !.
substit(_, Term, _, Term) :-
                        atomic(Term), !.
substit(Sub, Term, Sub1, Term1) :-
                        Term =.. [F|Args],
                        sublist(Sub, Args, Sub1, Args1),
                        Tem1 =..[F|Args1].
sublist(_, [], _, []).
sublist(Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
                        substit(Sub, Term, Sub1, Term1),
                        sublist(Sub, Terms, Sub1, Terms1).

substitute1(_, [], _, []).
substitute1(X, [X|T], A, [A|T1]) :-
                        substitute1(X, T, A, T1).
substitute1(X, [Y|T], A, [Y|T1]) :-
                        substitute1(X, T, A, T1).
%=================================================================
%   (2) a)  A palindrome is a word or phrase that spells the same forwards and backwards.
%           For example, `rotator', `eve', and `nurses run' are all palindromes. Write a
%           predicate palindrome (List), which checks whether List is a palindrome.
palindrom(List) :-
                        reverse(List, List), !.
reverse([], []).
reverse([H|T], List):-
                        reverse(T, T1),
                        append(T1, [H], List).
append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
                        append(Tail, List, Tail2).
%=====================================================
%       b)  Write a two-place predicate termtype(Term,Type) that takes a term and gives
%           back the type(s) of that term (atom, number, constant, variable etc.). The types
%           should be given back in the order of their generality. The predicate should, e.g.,
%           behave in the following way.
%                ?- termtype(Vincent,variable).
%                Yes
%                ?- termtype(mia,X).
%                X = atom ;
%                X = constant ;
%                X = term ;
%                No
%                ?- termtype(dead(zed),X).
%                X = complex_term ;
%                X = term ;
%                no

termtype(Term,atom)            :- atom(Term).
termtype(Term,number)                  :- number(Term).
termtype(Term,constant)        :- atomic(Term). % atom(Term); number(Term).
termtype(Term,variable)        :- var(Term).
termtype(Term,complex_term)     :- Term =.. [H|T], T \== [].
termtype(Term, term).
%=================================================
%       c)  Define the procedure
%                height(BinaryTree, Height).
%           to find the height of a binary tree. Assume that the height of empty tree is 0,
%           and that of one element is 1.
height(nil, 0).
height(t(Left, Root, Right), H) :-
                        height(Left, Hl),
                        height(Right, Hr),
                        max(Hl, Hr, Mh),
                        H is Mh + 1.
max(A, B, A) :-
                        A >= B, !.
max(A, B, B).
%====================================================
%       d)  Let f be a file of terms. Write a procedure
%                       findallterms(Term).
%           That displays on the terminal all terms in f that matches Term.
 findterm(Term) :-
                        see(f),  read(Term),
                        write(Term) ,   seen;
                        findterm(Term).
```

```
findallterm(Term) :-
                    see(f),
                    read(CTerm),
                    process(CTerm, Term), seen.

process(end-of-file, _) :- !.
process(CTerm, Term) :-
                    ( not (CTerm = Term), !;
                    write(CTerm), nl),
                    findallterm(Term).

%================================================================
%(3) a)  Write a logic program for solving the following puzzle.
%       Four people each use a different editor, UNIX variant and programming language.
%            • People:  Daisy (female),   Steve (male), Llewellyn (male) and
%              Gertrude (female).
%            • Editors:  emacs,  vi,  pico, and joe
%            • Unix variants:  Solaris,  AIX, Linux and  BSD
%            • Languages:  C++ (OO),  Java (OO),  Perl and  Lisp
%            Using the clues that follow, find out who uses what.
%            • The emacs user is a man who likes Lisp.
%            • The woman who uses pico likes BSD, but does not use object-oriented
%             (OO)  languages.
%            • The C++ programmer uses AIX and joe.
%            • Daisy loves Java, but thinks Solaris is a joke of an operating system.
%            • Llewellyn cannot stand AIX.
male('Steve').
male('Llewellyn').

female('Daisy').
female('Gertude').

editor(emac).
editor(vi).
editor(pico).
editor(joe).

unix_v('Solaries').
unix_v('AIX').
unix_v('Linux').
unix_v('BSD').

language('Java', oo).
language('C++', oo).
language('Perl', net).
language('Lisp', ai).
```

```
likes('Steve', 'Lisp').
likes('Daisy', 'Lisp').
likes('Daisy', 'BSD').
editor_user(X, Y) :-
                    male(X),
                    editor(Y) , Y == emac,
                    language('Lisp', _),
                    likes(X, 'Lisp').
editor_user(X, Y):-
                    female(X),
                    editor(Y), Y == pico, unix_v('BSD'),
                    likes(X, L),       !,
                    not language(L, oo),
                    likes(X, 'BSD').
editor_user(_, _).
cpp_programmer(X, Y, Z) :-
                    (male(X), X \== 'Llewellyn'; female(X)),
                    editor(Y), Y ==joe,
                    unix_v(Z), Z == 'AIX'.
love(X, L) :-
                    female(X), X = 'Daisy',
                    language(L, oo), L == 'Java',
                    unix_v('Solaries').

stand(X, Y) :-
                    male(X), ((X == 'Llewellyn', !,
                    unix_v(Y), Y \= 'AIX'); unix_v(Y)).

who_use_what(X, Y):-
                    editor_user(X, Y).
who_use_what(X, Y):
                    love(X, Y), X \= Y.
who_use_what(X, Y):-
                    stand(X, Y).
who_use_what(X, Y):-
                    cpp_programmer(X, Y, _).
who_use_what(X, Z):-
                    cpp_programmer(X, _, Z).
%===========================================================
%   b)  Write a breadth-first search procedure
%              breadthfirst([[Start]], Solution).
%          Let Solution be represented as a list of nodes in inverse order from Start to a
%          goal, so that the goal node is the head of Solution.
%               solve(Start, Solution):
%         Solution is a path (in reverse order) from start to a goal
solve(Start, Solution) :-
                              breadthfirst([[Start]], Solution).
```

```
%           breadthfirst([Path1, Path2, …], Solution):
%           Solution is an extension to a goal of one of paths
breadthfirst([[Node|Path]|_], [Node|Path]):-
                                    goal(Node).
breadthfirst([Path|Paths], Solution):-
                                    extend(Path, NewPaths),
                                    append(Paths, NewPaths,Paths1),
                                    breadthfirst(Paths1, Solution).

extend([Node|Path], NewPaths) :-
                                    bagof([NewNode, Node|Path], (s(Node, NewNode),
not member(NewNode, [Node|Path])), NewPaths), !.
extend(Path, [ ]). % bagof failed: Node has no successor

%==================================================================
%   (4)  a)  Write a procedure to transform a natural language sentence, taken as a string
%            of characters with some spaces between them, into a list whose elements are its
%            words. For example: Wordlist = [tom, is, ironing, the, shirt]
%            If the input sentence is “tom is ironing the shirt.”

go(W) :- getsentence(W).

getsentence(Word) :-
                                    get0(Char),
                                    getrest(Char, Word).
getrest(46, []) :- !.               % End of sentence
getrest(32, Word) :- !, % ASCII for blank
                                    getsentence(Word).
getrest(Letter, [Word|Wordlist]) :-
                                    getletters(Letter, Letters, Nextchar),
                                    name(Word, Letters),
                                    getrest(Nextchar, Wordlist).
getletters(46, [], 46) :- !.        % End of word : 46 = full stop
getletters(32, [], 32) :- !.        % End of word : 32 = blank
getletters(Let, [Let|Letters], Nextchar) :-
                                    get0(Char),
                                    getletters(Char, Letters, Nextchar).

%================================================================
%        b)  Explain how we can represent graphs in logic programming.
%        i-  connect(a, b).  connect(b, c).  ....  arc(s, t, 3).arc(t, u, 2). ...
%      ii-  G1 = graph( [a, b, c, d], [e(a,b), e(b,d), e(b, c), e(c,d)]).
%           G2 = graph( [s, t, u, v], (a(s, t,3), a(t,v,1), a(u, t, 2)...]).
%     iii-  G1 = [a ->[b], b->[a, c,d], c -> [b, d], d -> [b, c]]
%           G2 = [ s - > [t/3], t -> [u/5, v/1], u -> [t/2], v -> [u/2]]
```

```
%=================================================================
%       c)  Define the procedure
%                      path(A, Z, Graph, Path, Cost).
%          Finding an acyclic path, Path with Cost from A to Z in a Graph..
%                 path(A, Goal, Graph, Path).

path(A, Z, Graph, Path) :-
                              path1(A, [Z], Graph, Path).
path1(A, [A|Path1], _, [A|Path1]).
path1(A, [Y|Path1], Graph, Path) :-
                              adjacent(X, Y, Graph),
                              not member(X, Path1),
                              path1(A, [X, Y|Path1], Graph, Path).
adjacent(X, Y, graph(Node, Edges)) :-
                              member(e(X, Y), Edges) ;
                              memeber(e(Y, X), Edges).
e(X, Y).

%                 path(A, Goal, Graph, Path, Cost).
%       Path is an acyclic path with cost Cost in Graph

path (A, Z, Graph, Path, Cost) :-
                      path1(A, [Z], 0, Graph, Path, Cost).
path1(A, [A|Path1], Cost1, _, [A|Path1], Cost1).

path1(A, [Y|Path1], Cost1, Graph, Path, Cost) :-
                      adjacent (X, Y, CostXY, Graph),
                      not member (X, Path1),  % No cycle condition
                      Cost2 is Cost1 + CostXY,
                      path1(A, [X, Y|Path1], Cost2, Graph, Path, Cost).

adjacent (X, Y, CostXY, graph (Nodes, Edges)):-
                      member (e (X, Y, CostXY), Edges) ; % Or
                      member (e (Y, X, CostXY), Edges).
member (X, [X|_]).
member (X,[_|T]):-
                      member( X, T).
%=================================================================
%       d) Write a procedure that is used to insert an item into a binary dictionary
%          (i.e., inserting an item at any level into the binary dictionary).
%                add(Tree, X, NewTree).
%         inserting X at any level of binary dictionary Tree gives NewTree
add(Tree, X, NewTree) :-
                      addroot(Tree, X, NewTree). % add X as new root
add(t(L, Y, R), X, t(L1, Y, R)):- % insert X into left subtree
                      gt(Y, X),
                      add(L, X, L1).
add(t(L,Y, R), X, t(L, Y, R1)):-%insert X into right subtree
                        gt(X, Y),
                        add(R, X, R1).

%               addroot(Tree, X, NewTree)
%               inserting X as the root into Tree gives NewTree
addroot(nil, X, t(nil, X, nil)).  % insert into empty tree
addroot(t(L,Y, R), X, t(L1, X, t(L2, Y, R))) :-
                        gt(Y, X),
                        addroot(L, X, t(L1, X, L2)).
addroot(t(L,Y, R), X, t(t(L, Y, R1), X, R2)) :-
                        gt(X, Y),
                        addroot(R, X, t(R1, X, R2)).

%===================================================================
%       (5) a) Write a logic program defining the relationship
%                       gcd(N1,N2,F).
%            To find the greatest common divisor of two integers N1 and N2.

gcd(N, 0, N).
gcd(X, Y, N) :-
                        X > Y,
                        N1 is X mod Y,
                        gcd(Y, N1, N), !.
gcd(X, Y, N) :-
                        N1 is Y mod X,
                        gcd(X, N1, N), !.

%===================================================================
%          b) Write a logic program to define the relationship
%                       real_average(List, A).
%               That calculates the average value of all elements in a list of integers

real_average(List, A) :-
                        length_1(List, N),
                        sum_list(List, Sum),
                        A is Sum / N.

length_1([], 0).
length_1([H|T], N) :-
                        length_1(T, N1),
                        N is N1 + 1.

sum_list([], 0).
sum_list([H|T], S) :-
                        sum_list(T, S1),
                        S is S1 + H.
```

```
%=====================================================================
%          c)  Write a logic program considering the following sentences:
%                         John likes all kinds of food
%                         Apples are food
%                         Chicken is food
%                         Anything anyone eats and isn’t killed by is food
%                         Bill eats peanuts and is still alive
%                         Sue eats every thing Bill eats.
%                and try to answer the question "what does Sue like?"
food (apple).
food (chicken).
eat (Sue, X) :-
                eat (Bill,X).
eat (Bill, peanut):-
                 not died( Bill).

eat (X, Y) :-
                food(Y),
                not killed (X, Y),

like (john, X):-
                food (X).

%====================================================================
%          d)  Consider the following facts:
%                         p( a,  b ).
%                         p(X,  d ).
%                         p(e,  Y ).
%             List all answers to the following queries:
%                         ?- p(X, Y), !.
%                         X = a, Y = b

%                         ?- p(c, Y), !.
%                         Y = d

%                         ?- p(X,  f ), !.
%                         X = e
```

```
%==============================================================%
%    Ideal Solution for the Final Examination 2003-2004 %
%==============================================================%
%  (1) a) Define the relation
%                    subsume(Set, Sum, Subset).
%     So that Set is a list of numbers, Subset is a subset of these numbers and the sum
%     of the numbers in Subset is Sum. For example:
%     ?- subsume ([1,2,5,3,2], 5, Sub).
%      Sub = [1,2,2] ;
%      Sub = [2,3] ;
%      Sub = [5] ;
%          …..
%===============================================================
subsume ([], 0, []).
subsume ([N|List], Sum, [N|Sub]) :-
          Sum1 is Sum - N,
          subsume (List, Sum1, Sub).
subsume ([N|List], Sum, Sub) :-
          subsume (List, Sum, Sub).

%===============================================================
%b) Define the procedure
%         substitute (Subterm, Term, Subterm1, Term1).
%to substitute all occurrences of Subterm in Term with Subterm1, then we get Term1.
%===============================================================
substit(Term, Term, Term1, Term1) :- !.
substit(_, Term, _, Term) :-
          atomic(Term), !.
substit(Sub, Term, Sub1, Term1) :-
          Term =.. [F|Args],
          sublist(Sub, Args, Sub1, Args1),
          Tem1 =..[F|Args1].
sublist(_, [], _, []).
sublist(Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
          substit(Sub, Term, Sub1, Term1),
          sublist(Sub, Terms, Sub1, Terms1).

substitute1(_, [], _, []).
substitute1(X, [X|T], A, [A|T1]) :-
          substitute1(X, T, A, T1).
substitute1(X, [Y|T], A, [Y|T1]) :-
          substitute1(X, T, A, T1).

%================================================================
% c) Check whether a word is palindrome, i.e., the same if read backwards, as for example
%     MADAM". Write a logic program able to check a palindrome for each word you give.
%================================================================
begin :-
                read(X),
                (X == stop, !;  test_palindrome(X), begin).
test_palindrome(X) :-
                name(X, Nx),
                palindrom(Nx),
                write(X), write(' is a palindrome'), nl, !.
test_palindrome(X) :-
                write(X), write(' is not a palindrome'), nl.

palindrom(List) :-
                reverse(List, List).
reverse([], []).
reverse([H|T], List):-
                reverse(T, T1),
                append(T1, [H], List).

append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
                append(Tail, List, Tail2).

%==============================================================
%       d) Check whether H is a member of a list L; search the first member of a list L.
%==============================================================
member1(X,[X|_]) :- !.
member1(X,{_|T]) :-
        member1(X,T).

%==============================================================
%  (2)  a) Define the procedure
%                               height (BinaryTree, Height).
%         to compute the height of a binary tree. Assume that the height of empty tree is 0,
%         and that of one element is 1.
%==============================================================
height (nil, 0).
height (t(Left, Root, Right), H) :-
                        height (Left, Hl),
                        height (Right, Hr),
                        max (Hl, Hr, Mh),
                        H is Mh + 1.
max(A, B, A) :-
                        A >= B, !.
max(A, B, B).

%==============================================================
%       b) Define a predicate eq_length(Xs, Ys) that, given two lists Xs and Ys, succeeds if
%         and only if the lengths of Xs and Ys are equal.
%                                    VERSION 1
%                            eq_length(List1, List2).
%==============================================================
eq_length([], []).
eq_length([H|T], [H1|T1]) :-          eq_length(T, T1).
eq_length(_,[]):- fail.
eq_length([], _):- fail.
%==============================================================
%                                VERSION 2
%                            equal_l(L1, L2)
%==============================================================
equal_l(L1, L2) :-
                      length1(L1, N),
                      length1(L2, M),
                      N = M.
length1([], 0).
length1([H|T], N) :-
                      length1(T, M),
                      N is M + 1.
%==============================================================
%      c) Suppose we represent sets (no duplicated elements) ranging over integers as lists
%         of integers. Write Prolog predicates for performing the following set operations:
%              • intersection (S1, S2): returns set-intersection of sets S1 and S2.
%==============================================================
intersect([H|T], L, [H|U]) :-
                      member(H, L), !,
                      intersect(T, L, U).
intersect([_|T], L, U) :-
                      !, intersect(T, L, U).
intersect(_, _, []).

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

%==============================================================
%         d)   Define the quick sort relation
%                            quick_sort(List, Sorted, X).
%              that will sort List without append (make X = [] in the input)..
%==============================================================
%              QuickSort(List, Sorted).  Using append
%==============================================================
quicksort([], []).
quicksort([X|Tail], Sorted) :-
                      split(X, Tail, Small, Big),
                      quicksort(Small, SortedSmall),
                      quicksort(Big, SortedBig),
                        append(SortedSmall, [X|SortedBig], Sorted).

split(X, [], [], []).
split(X, [H|T], [H|T1], List) :-
                        X > H, !,
                        split(X, T, T1, List).
split(X, [H|T], List, [H|T1]) :-
                        split(X, T, List, T1).

%==========================================================
%               quick_sort(List, Sorted, X).  Without using, append
%==========================================================
quick_sort([H|T], S, X) :-
                        split1(H, T, U1, U2),
                        quick_sort(U1, S, [H|Y]),
                        quick_sort(U2, Y, X).
quick_sort([], Sorted, Sorted).

split1(_, [], [], []).
split1(H, [H1|T], [H1|T1], List) :-
                        H1 < H, !,
                        split1(H, T, T1, List).
split1(X, [H|T], List, [H|T1]) :-
                        split1(X, T, List, T1).

%==========================================================
%  (3) a) The projects in a company can be described as a database with entries being
%          facts of the following form:
%               project (ProjectName, Manager, Participants, StartTime, StopTime).
%          StartTime and StopTime are integer numbers, and you can assume that StartTime
%          is strictly less than StopTime.  ProjectName and Manager are constants and
%          Participants is a list of unique constants all representing persons participating
%          in the project.  A manager does not necessarily participate in the project he/she
%          manages.
%          Note that a project can be represented by several entries, if for instance
%          different individuals participate during different time-slots of the project,
%          or if the project has more than one manager or there is a break in the project,
%          i.e., a time period when there is no activity in the project.
%          Define queries for the following questions:
%            •    Find the names P1, P2 of two different persons participating in the
%                 same project.
%            •    Find a person P participating in exactly one project, possibly with
%                 different managers.
%            •    Find the names M1, M2 of two different managers that share responsibility
%                 for the same project, possibly overlapping in time.
%            •    Identify a project ProjectName, such that there is a break in the
%                 duration of the project (no activity for some time period in the total
%               length of the project).
%==================================================================
%        Example of database
%               project(ProjectName,Manager, Participants,StartTime,StopTime).
project(a, lisa, [], 10, 30).
project(b, lisa, [benny, sanna],  23, 41).
project(b, erik, [benny], 45, 50).
project(b, lisa, [sanna],  55, 60).
project(c, Johnny, [], 71, 85).
project(d, johnny, [rut, lisa, anna], 32, 68).
project(e, Johnny, [rune], 40, 77).
project(f, erik, [sven, mats], 44, 57).
project(g, erik, [jenny, lars, bo], 30, 47).
project(h, bengt, [jenny], 10, 49).
project(i, bengt, [lars], 20, 40).
project(j, bengt, [bo], 30, 40).
project(k, bengt, [erik, jenny], 40, 44).

%       i) Find the names P1, P2 of two different persons participating in the same
%         project
%               project(ProjectName, Manager, Participants, StartTime, StopTime)

insameproject(P1, P2) :-
                project(Name, _, Participants1, _, _),
                project(Name, _, Participants2, _, _),
                append(Participants1,  Participants2, X),
                member(P1, X),
                member(P2, X),
                P1 \= P2.
%       Note that you need to look up the project twice to cover the possibility that the
%        two  persons participate in different time slots.

%       ii) Find a person P participating in exactly one project (possibly with different
%          managers)
%               project(ProjectName, Manager, Participants, StartTime, StopTime)

personinexactlyoneproject(P) :-
                project(Name1, _, Participants1, _, _),
                member(P, Participants1),
                not (    project(Name2, _, Participants2, _, _),
                         Name1 \= Name2,
                         member(P, Participants2)).

%       iii) Find the names M1, M2 of two different managers that share responsibility for
%          the same project (possibly overlapping in time)
%               project(ProjectName, Manager, Participants, StartTime, StopTime)
```

```
commonproject(M1, M2) :-
                project(Name,M1, _, _, _),
                project(Name,M2, _, _, _),
                M1 \= M2.

%       vi) Identify a project ProjectName, such that there is a break in the duration of the
%           project  (no activity for some time period in the total length of the project)
%               project(ProjectName, Manager, Participants, StartTime, StopTime)

break(ProjectName) :-
                project(ProjectName, _, _, Start1, Stop1),
                project(ProjectName, _, _, Start2, Stop2),
                Start1 < Start2, Stop1 < Start2 - 1.


%==================================================================
%       b) Write a depth-first search procedure (With cycle detection and depth limiting
%          mechanisms)
%                               depthfirst(Node, Solution, MaxDepth).
%          to find a solution path Solution from Node to the goal node. Let Solution be
%          represented as a list of nodes in inverse order, so that the goal node is the
%          head of Solution.
%==================================================================
solve(Node, Solution)  :-
                        depthfirst([], Node, Solution).

depthfirst(Path, Node, [Node|Path]) :-
                        goal(Node).
depthfirst(Path, Node, Sol) :-
                        s(Node, Node1),
                        not member(Node1, Path),
                        depthfirst([Node|Path], Node1, Sol).
goal(z).
s(a, b).
s(b, c).
s(c, z).
%==================================================================
%                           depthfirst(Node, Solution, MaxDepth).
%               Solution is path not longer than Maxdepth from Node to a goal
%==================================================================
depthfirst2(Node, [Node],_):-
                                goal(Node).
depthfirst2(Node, [Node|Sol], Max) :-
                                Max > 0,
                                s(Node, Node1),
                                not member(Node1, Path), % prevent a cycle
                                Max1 is Max - 1,  depthfirst2(Node1, Sol, Max1).
```

```
%=============================================================
% (4) a) Write a procedure to transform a natural language sentence, taken as a string of
%        characters with some spaces between them, into a list whose elements are its words.
%        for example:   Wordlist = [tom, is, ironing, the, shirt]
%        If the input sentence is:   "tom is ironing the shirt."
%=============================================================
go(W) :-
                    getsentence(W).
getsentence(Word) :-
                    get0(Char),      getrest(Char, Word).
getrest(46, []) :- !.          % End of sentence
getrest(32, Word) :- !, % ASCII for blank
                    getsentence(Word).
getrest(Letter, [Word|Wordlist]) :-
                    getletters(Letter, Letters, Nextchar),
                    name(Word, Letters),
                    getrest(Nextchar, Wordlist).
getletters(46, [], 46) :- !.   % End of word : 46 = full stop
getletters(32, [], 32) :- !.   % End of word : 32 = blank
getletters(Let, [Let|Letters], Nextchar) :-
                    get0(Char),
                    getletters(Char, Letters, Nextchar).
%=============================================================
%   b) Explain how we can represent trees in logic programming.
%      1.It is to make the root of a binary tree the principle functor of the term, and
%        the subtrees its arguments.
%      2.Use a special symbol to represent empty tree, and we need a functor to construct
%        a non-empty tree from its components:
%             • Let the atom nil represent the empty tree
%             • Let the functor be t so the tree that has a root X, left subtree L
%               and a right subtree R is represented by the term: t(L, X, R)
%                    t(t(nil, b, nil), a, t(t(nil, d, nil), c, nil)).

%=============================================================
%   c) Write a procedure "pathfinding" that is used to find an acyclic
%      path from node A to node Z in a graph.
%                          ath(A, Goal, Graph, Path).
%=============================================================
path(A, Z, Graph, Path) :-
                    path1(A, [Z], Graph, Path).
path1(A, [A|Path1], _, [A|Path1]).
path1(A, [Y|Path1], Graph, Path) :-
                    adjacent(X, Y, Graph),
                    not member(X, Path1),
                    path1(A, [X, Y|Path1], Graph, Path).
adjacent(X, Y, graph(Node, Edges)) :-
                    member(e(X, Y), Edges) ;memeber(e(Y, X), Edges).
```

```
e(X, Y).
%==============================================================
%   d) Write a procedure that is used to find an item X in a binary dictionary.
%                               in(X, Tree): X in binary dictionary Tree
%==============================================================
in(X, t(_, X, _)).
in(X, t(Left, Root, Right)) :-
                        gt(Root, X),                % Root greater than X
                        in(X, Left).                % Search Left subtree
in(X, t(Left, Root, Right)) :-
                        gt(X, Root),                % X greater than Root
                        in(X, Right).               % Search right subtree
%       Modify the defined procedure by adding the third argument Path, so that Path
%       is the path between the root of the dictionary and X.
in1(Item, t(_,Item,_),[Item]).
in1(Item, t(Left, Root, _), [Root|Path]) :-
                        gt(Root, Item),  in1(Item, Left, Path).
in1(Item, t(_, Root, Right), [Root|Path]) :-
                        gt(Item, Root), in1(Item, Right, Path).
%==============================================================
%  (5) a) Write a logic program to define the exponentiation function:
%                       exp (x, y, z) if and only if    x ^y = z
%        The following properties of integer exponentiation are taken as the defining ones:
%               • x^0 = 1
%               • x^y = x . x ^y - 1
%                exp(X, Y, Z).
%==============================================================
exp(X, 0, 1) :- !.
exp(X, Y, Z) :-
                        even(Y),  R is Y / 2,
                        P is X * X,      exp(P, R, Z).
exp(X, Y, Z) :-
                        T is Y - 1,       exp(X, T, Z1),
                        Z is Z1 * X.
even(X) :-
                R is Y mod 2, R == 0.
%==============================================================
%       b) Generate primes in first N integers and call them primes (N, CP).
%==============================================================
primes(N, [1|LP]) :-
                        M is N - 1,
                        integers (2, M, LI),
                        shift (LI, LP), !.
integers(N, 1, [N]) :- !.
integers(N, M, [N|LI]) :-
                        R is M - 1,
                        Q is N + 1,      integers(Q, R, LI).
```

```
shift([], []) :- !.
shift([P|LI], [P|LP]) :-
                        filter(P, LI, NLI),
                        shift(NLI, LP).
filter(P, [], []) :- !.
filter(P, [N|LI], NLI) :-
                        divide(P, N, true), !,
                        filter(P, LI, NLI).
filter(P, [N|LI], [N|NLI]) :-
                        divide(P, N, false), !,
                        filter(P, LI, NLI).
divide(P, N, true):-    %  / real division, // integer division
                        0 is N - P * (N // P).
divide(P, N, false).

%=============================================================
%      c) Write a logic program considering the following facts:
%         • number(X, N) means that person X can be reached by calling phone number N.
%         • visits(X, Y) means that person X is visiting person Y.
%         • at (X, Y) means that person X is at the residence of person Y.
%         • phone(X, N) mean that person X has phone number N.
%      and the following premises:
%            i) (∀X) (∀Y) (∀Z) [visits (X, Y) & at (Y, Z)          ⇒ at (X, Z)]
%            ii) (∀U) (∀V) (∀N) [at (U, V) & Phone (V, N)         ⇒  number  (U, N)]
%               able to find the way of reaching a person, having a particular phone number
%      and knowing who visits whom and where is the visited person.
%=============================================================
find(X) :-
                        number(X, N), write('Phone:  '),  write(N), nl.
find(_) :-
                        write('Don''t know.') nl.
number(X, N) :-
                        at(X, Y),
                        phone(Y, N).
number(X, N) :-
                        phone(X, N).
at(X, Z) :-
                        visits(X, Y),
                        at(Y, Z).
% Phone numbers data base
phone(ali, 3883350).
phone(sayed, 77665544).
phone(jhone, 4601213).
phone(smith, 4036655).
visits(ali, smith).
visits(ali, jhon).
visits(smith, sayed).
```

```
%=============================================================
%       d) Consider the following logic program:
                       p( 1,  2 ).
                       p(X,  4 ).
                       p(5,  Y ).
%=============================================================
%         List all answers to the following queries:
                       ?- p(X, Y), p(X, 7).
%                      X = 5, Y = 4;
%                      X = 5, Y= _0098
                       ?- p(3, Y), !, p(X, Y).
%                      Y = 4, X = _0098;
%                      y = 4, X = 5.
```

```
%=============================================================%
%  Ideal Solution for the Corrective Examination 2003-2004  %
%=============================================================%
%       (1) a)  Design and test prune (A,B) which is intended to remove multiple occurrences
%             of elements from A to produce result B. For example,
%                       ?- prune([a,1,b,2,a,3,a,4,b],B).
%                       B = [a,1,b,2,3,4]
%                Try to make it so that B has remaining elements in the order that they
%                occurred in A.
%                                VERSION 1
%                       ?- prune([a,1,b,2,a,3,a,4,b],B).
%==============================================================
proune([], L, L).
proune([H|T], List, L):-
                member(H, List),        !,
                proune(T, List, L).
proune([H|T], L1, L):-
                append(L1, [H], L2), !,
                proune(T, L2, L).

member(_,[]):- fail.
member(X,[X|_]).
member(X, [_|T]):-
                member(X, T).
append([], L, L).
append([H|T], L, [H|T1]) :-
                append(T, L, T1).

%                                VERSION 2
%                       ?- prune([a,1,b,2,a,3,a,4,b],B).
proune2([], []).
proune2([H|T], List):-
        proune2(T, List1),      not member(H, List1), !,
        append([H], List1,  List).
proune2([H|T], List):-
        proune2(T, List).
%==============================================================
%       b)  Suppose we represent sets (no duplicated elements) ranging over integers as
%           lists of integers. Write Prolog predicates for performing the following set
%           operations: union (S1, S2): returns the set-union of sets S1 and S2.
%==============================================================
union([X|Y],Z,W) :-
                        member(X,Z),   union(Y,Z,W).
union([X|Y],Z,[X|W]) :-
                        not member(X,Z), union(Y,Z,W).
union([],Z,Z).
```

```
%==============================================================
%        c)   Write a procedure that will sort a list using bubble sort algorithm.
 ==============================================================
bubble_sort(List, Sorted) :-
                        swap(List, List1),  !,
                        bubble_sort(List1, Sorted).
bubble_sort(Sorted, Sorted). %List is already sorted

swap([X,Y|Rest],[Y,X|Rest]):-% swap first two elements
                        gt( X, Y).
swap([Z|Rest], [Z|Rest1]) :- % swap elements in tail
                        swap(Rest, Rest1).
gt(X, Y) :-
                        X > Y.
%==============================================================
%        d)   Define the procedure
%                  substitute (Subterm, Term, Subterm1, Term1).
%          To substitute all occurrences of Subterm in Term with Subterm1, then we get Term1.
%==============================================================
substit(Term, Term, Term1, Term1) :- !.
substit(_, Term, _, Term) :-
                        atomic(Term), !.
substit(Sub, Term, Sub1, Term1) :-
                        Term =.. [F|Args],
                        sublist(Sub, Args, Sub1, Args1),
                        Tem1 =..[F|Args1].
sublist(_, [], _, []).
sublist(Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
                        substit(Sub, Term, Sub1, Term1),
                        sublist(Sub, Terms, Sub1, Terms1).

substitute1(_, [], _, []).
substitute1(X, [X|T], A, [A|T1]) :-
                        substitute1(X, T, A, T1).
substitute1(X, [Y|T], A, [Y|T1]) :-
                        substitute1(X, T, A, T1).
%==============================================================
%   (2) a) A palindrome is a word or phrase that spells the same forwards and backwards.
%           For example, `rotator', `eve', and `nurses run' are all palindromes. Write a
%           predicate palindrome (List), which checks whether List is a palindrome.
%==============================================================
palindrom(List) :-
                        reverse(List, List), !.

reverse([], []).
reverse([H|T], List):-
                        reverse(T, T1), append(T1, [H], List).
```

```
append([], List1, List1).
append([Head|Tail], List, [Head|Tail2]) :-
                        append(Tail, List, Tail2).
%=================================================================
%       b) Write a two-place predicate termtype(Term,Type) that takes a term and gives
%          back the type(s) of that term (atom, number, constant, variable etc.). The types
%          should be given back in the order of their generality. The predicate should, e.g.,
%          behave in the following way.
%               ?- termtype(Vincent,variable).
%               Yes
%               ?- termtype(mia,X).
%               X = atom;
%               X = constant;
%               X = term;
%               No
%               ?- termtype(dead(zed),X).
%               X = complex_term ;
%               X = term;
%               no
%=================================================================
termtype(Term,atom)             :- atom(Term).
termtype(Term,number)           :- number(Term).
termtype(Term,constant)         :- atomic(Term). % atom (Term); number (Term).
termtype(Term,variable)         :- var(Term).
termtype(Term,complex_term)      :- Term =.. [H|T], T \== [].
termtype(Term, term).
%=================================================================
%       c) Define the procedure
%                       height (BinaryTree, Height).
%          to find the height of a binary tree. Assume that the height of empty tree is zero,
%          and that of one element is 1.
%=================================================================
height(nil, 0).
height(t(Left, Root, Right), H) :-
                        height(Left, Hl),
                        height(Right, Hr),
                        max(Hl, Hr, Mh),
                        H is Mh + 1.
max(A, B, A) :-
                        A >= B, !.
max(A, B, B).

%=================================================================
%       d)  Let f be a file of terms. Write a procedure
%                               findallterms(Term).
%          That displays on the terminal all terms in f that matches Term.
%=================================================================
findterm(Term) :-
                see(f),  read(Term), write(Term) , seen;  findterm(Term).

findallterm(Term) :-
                see(f),
                read(CTerm),
                process(CTerm, Term), seen.

process(end-of-file, _) :- !.
process(CTerm, Term) :-
                ( not (CTerm = Term), !;
                write(CTerm), nl),
                findallterm(Term).
%=============================================================
% (3) a) Write a logic program for solving the following puzzle.  Four people each use a
%         different editor, UNIX variant and programming language.
%       • People:  Daisy (female),   Steve (male), Llewellyn (male) and   Gertrude (female).
%       • Editors:  emacs,  vi,  pico, and joe
%       • Unix variants:  Solaris,  AIX, Linux and  BSD
%       • Languages:  C++ (OO),  Java (OO),  Perl and  Lisp
%       Using the clues that follow, find out who uses what.
%       • The emacs user is a man who likes Lisp.
%       • the woman who uses pico likes BSD, but does not use object-oriented (oo)
%         languages.
%       • The C++ programmer uses AIX and joe.
%       • Daisy loves Java, but thinks Solaris is a joke of an operating system.
%       • Llewellyn cannot stand AIX.
%=============================================================
male('Steve').
male('Llewellyn').

female('Daisy').
female('Gertude').

editor(emac).
editor(vi).
editor(pico).
editor(joe).

unix_v('Solaries').
unix_v('AIX').
unix_v('Linux').
unix_v('BSD').

language('Java', oo).
language('C++', oo).
language('Perl', net).
language('Lisp', ai).

likes('Steve', 'Lisp').
likes('Daisy', 'Lisp').
likes('Daisy', 'BSD').

editor_user(X, Y) :-
                        male(X),
                        editor(Y) , Y == emac,
                        language('Lisp', _),
                        likes(X, 'Lisp').

editor_user(X, Y):-
                        female(X),
                        editor(Y), Y == pico, unix_v('BSD'),
                        likes(X, L),        !,
                        not language(L, oo),
                        likes(X, 'BSD').
editor_user(_, _).

cpp_programmer(X, Y, Z) :-
                        (male(X), X \== 'Llewellyn'; female(X)),
                        editor(Y), Y ==joe,
                        unix_v(Z), Z == 'AIX'.
love(X, L) :-
                        female(X), X = 'Daisy',
                        language(L, oo), L == 'Java',
                        unix_v('Solaries').

stand(X, Y) :-
                        male(X), ((X == 'Llewellyn', !,
                        unix_v(Y), Y \= 'AIX'); unix_v(Y)).

who_use_what(X, Y):-
                        editor_user(X, Y).
who_use_what(X, Y):
                        love(X, Y), X \= Y.
who_use_what(X, Y):-
                        stand(X, Y).
who_use_what(X, Y):-
                        cpp_programmer(X, Y, _).
who_use_what(X, Z):-
                        cpp_programmer(X, _, Z).

%=============================================================
%       b) Write a breadth-first search procedure
%                               breadthfirst([[Start]], Solution).
%          Let Solution be represented as a list of nodes in inverse order from Start to a
%          goal, so that the goal node is the head of Solution.
%                solve (Start, Solution):
%          Solution is a path (in reverse order) from start to a goal
%=============================================================
solve(Start, Solution) :-
                              breadthfirst([[Start]], Solution).
%          breadthfirst([Path1, Path2, …], Solution):
%          Solution is an extension to a goal of one of paths
breadthfirst([[Node|Path]|_], [Node|Path]):-
                              goal(Node).
breadthfirst([Path|Paths], Solution):-
                              extend(Path, NewPaths),
                              append(Paths, NewPaths,Paths1),
                              breadthfirst(Paths1, Solution).

extend([Node|Path], NewPaths) :-
                              bagof([NewNode, Node|Path], (s(Node, NewNode),
not member(NewNode, [Node|Path])), NewPaths), !.
extend(Path, [ ]). % bagof failed: Node has no successor

%=============================================================
%       (4) a) Write a procedure to transform a natural language sentence, taken as a string
%              of characters with some spaces between them, into a list whose elements are its
%              words. For example: Wordlist = [tom, is, ironing, the, shirt]
%               If the input sentence is “tom is ironing the shirt.”
%=============================================================
go(W) :- getsentence(W).

getsentence(Word) :-
                              get0(Char),
                              getrest(Char, Word).
getrest(46, []) :- !.         % End of sentence
getrest(32, Word) :- !, % ASCII for blank
                              getsentence(Word).
getrest(Letter, [Word|Wordlist]) :-
                              getletters(Letter, Letters, Nextchar),
                              name(Word, Letters),
                              getrest(Nextchar, Wordlist).
getletters(46, [], 46) :- !.  % End of word : 46 = full stop
getletters(32, [], 32) :- !.  % End of word : 32 = blank
getletters(Let, [Let|Letters], Nextchar) :-
                              get0(Char),
                              getletters(Char, Letters, Nextchar).
%=============================================================
%       b) Explain how we can represent graphs in logic programming.
%       i- connect (a, b).  connect (b, c).  ....  arc(s, t, 3).arc(t, u, 2). ...
%       ii- G1 = graph( [a, b, c, d], [e(a,b), e(b,d), e(b, c), e(c,d)]).
%           G2 = graph( [s, t, u, v], (a(s, t,3), a(t,v,1), a(u, t, 2)...]).
%      iii- G1 = [a ->[b], b->[a, c,d], c -> [b, d], d -> [b, c]]
%           G2 = [ s - > [t/3], t -> [u/5, v/1], u -> [t/2], v -> [u/2]]
%==============================================================
%       c) Define the procedure
%                               path(A, Z, Graph, Path, Cost).
%           Finding an acyclic path, Path with Cost from A to Z in a Graph..
%                               path(A, Goal, Graph, Path).
%==============================================================
path(A, Z, Graph, Path) :-
                                path1(A, [Z], Graph, Path).
path1(A, [A|Path1], _, [A|Path1]).
path1(A, [Y|Path1], Graph, Path) :-
                                adjacent(X, Y, Graph),
                                not member(X, Path1),
                                path1(A, [X, Y|Path1], Graph, Path).
adjacent(X, Y, graph(Node, Edges)) :-
                                member(e(X, Y), Edges) ;
                                memeber(e(Y, X), Edges).
e(X, Y).
%                   path(A, Goal, Graph, Path, Cost).
%       Path is an acyclic path with cost Cost in Graph

path (A, Z, Graph, Path, Cost) :-
                        path1(A, [Z], 0, Graph, Path, Cost).
path1(A, [A|Path1], Cost1, _, [A|Path1], Cost1).

path1(A, [Y|Path1], Cost1, Graph, Path, Cost) :-
                        adjacent (X, Y, CostXY, Graph),
                        not member (X, Path1),  % No cycle condition
                        Cost2 is Cost1 + CostXY,
                        path1(A, [X, Y|Path1], Cost2, Graph, Path, Cost).

adjacent (X, Y, CostXY, graph (Nodes, Edges)):-
                        member (e (X, Y, CostXY), Edges) ; % Or
                        member (e (Y, X, CostXY), Edges).
%==============================================================
%       d) Write a procedure that is used to insert an item into a binary dictionary
%          (i.e., inserting an item at any level into the binary dictionary).
%                       add (Tree, X, NewTree).
%          inserting X at any level of binary dictionary Tree gives NewTree
%==============================================================
add(Tree, X, NewTree) :-
                        addroot(Tree, X, NewTree). % add X as new root
add(t(L, Y, R), X, t(L1, Y, R)):- % insert X into left subtree
                        gt(Y, X),        add(L, X, L1).
```

```
add(t(L,Y, R), X, t(L, Y, R1)):-%insert X into right subtree
                                gt(X, Y),
                                add(R, X, R1).

%               addroot(Tree, X, NewTree)
%               inserting X as the root into Tree gives NewTree
addroot(nil, X, t(nil, X, nil)).  % insert into empty tree
addroot(t(L,Y, R), X, t(L1, X, t(L2, Y, R))) :-
                                gt(Y, X),
                                addroot(L, X, t(L1, X, L2)).
addroot(t(L,Y, R), X, t(t(L, Y, R1), X, R2)) :-
                                gt(X, Y),
                                addroot(R, X, t(R1, X, R2)).
%==================================================================
%       (5) a) Write a logic program defining the relationship
%                               gcd(N1,N2,F).
%           to find the greatest common divisor of two integers N1 and N2.
%==================================================================
gcd(N, 0, N).
gcd(X, Y, N) :-
                                X > Y,
                                N1 is X mod Y,
                                gcd(Y, N1, N), !.
gcd(X, Y, N) :-
                                N1 is Y mod X,
                                gcd(X, N1, N), !.
%==================================================================
%           b) Write a logic program to define the relationship
%                               real_average(List, A).
%               That calculates the average value of all elements in a list of integers
%==================================================================
real_average(List, A) :-
                                length_1(List, N),
                                sum_list(List, Sum),
                                A is Sum / N.
length_1([], 0).
length_1([H|T], N) :-
                                length_1(T, N1),
                                N is N1 + 1.
sum_list([], 0).
sum_list([H|T], S) :-
                                sum_list(T, S1),
                                S is S1 + H.
%==================================================================
%           c) Write a logic program considering the following sentences:
%                               John likes all kinds of food.
%                               Apples are food.
%                       Chicken is food.
%                       anything anyone eats and isn't killed by is food.
%                       Bill eats peanuts and is still alive.
%                       Sue eats every thing Bill eats.
%               and try to answer the question "what does Sue like?"
%=================================================================
food (apple).
food (chicken).
eat (Sue, X) :-
                eat (Bill,X).
eat (Bill, peanut):-
                 not died( Bill).
eat (X, Y) :-
                food(Y),
                not killed (X, Y),
like (john, X):-
                food (X).

%=================================================================
%          d) Consider the following facts:
                        p( a,  b ).
                        p(X,  d ).
                        p(e,  Y ).
%=================================================================
%            List all answers to the following queries:
                        ?- p(X, Y), !.
%                       X = a, Y = b

                        ?- p(c, Y), !.
%                       Y = d

                        ?- p(X,  f ), !.
%                       X = e
```