```
%%%  Output cyclic terms as equations.                                          %%%
%%%                                                                             %%%
%%%  Written by Ronald de Haan at UT Dresden (January, April 2011).            %%%
%%%                                                                             %%%
%%%  Reformatted and extensively modified by FK.                                %%%
%%%                                                                             %%%


% :- module( output_equation,
%          [
%            cyclic/2,
%            get_term_equation/3,
%            get_printable_term_equation/3
%          ]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Output cyclic terms as equations                           %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                            %%%
%%% Example of use:                                            %%%
%%%                                                            %%%
%%% ?- X = f(Y), Y = g(Y), get_term_equation(X, Eq, InitVar).  %%%
%%% X = f(g(**)),                                              %%%
%%% Y = g(**),                                                 %%%
%%% Eq = [InitVar=f(_G805), _G805=g(_G805)] .                  %%%
%%%                                                            %%%
%%%                                                            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%---------------------------------------------------------------------------
%% cyclic( + Term, + Max ) :
%% Succeeds iff the term Term is cyclic within a depth of Max.

cyclic( Term, Max ) :-
        cyclic_term( Term ),  % if the term is not cyclic at all, don't even try
        list_subterms_up_to_depth( Term, Max, Subterms ),
        check_list_for_duplicates( Subterms ).


%% The following is the original version:
%
%% list_subterms_up_to_depth( + Term, + MaxDepth, - Subterms ) :
%% Produces a list of all subterms of Term upto the given depth.
%
% list_subterms_up_to_depth( Term, MaxDepth, Subterms ) :-
%         list_subterms_up_to_depth( [ (0 , Term) ], MaxDepth, [], Subterms ).
%
%
% %% list_subterms_up_to_depth( + Terms, + MaxDepth, + Acc, - Answer ) :
% %%   - Terms is the list of pairs of numbers/terms to be handled,
% %%     where each number is the depth of the accompanying subterm within the
% %%     original term;
% %%   - MaxDepth is the depth up to which subterms are to be listed;
% %%   - Acc is an accumulator;
% %%   - Answer returns the list of subterms.
%
% list_subterms_up_to_depth( [], _, Acc, Acc ).
%
% list_subterms_up_to_depth( [ (CurDepth , _) | Terms ], MaxDepth, Acc, Ans ) :-
%         CurDepth > MaxDepth,
%         !,
%         list_subterms_up_to_depth( Terms, MaxDepth, Acc, Ans ).
%
% list_subterms_up_to_depth( [ (_ , CurTerm) | Terms ], MaxDepth, Acc, Ans ) :-
%         var( CurTerm ),
%         !,
%         list_subterms_up_to_depth( Terms, MaxDepth, Acc, Ans ).
%
% list_subterms_up_to_depth( [ (CurDepth , CurTerm) | Terms ],
%                             MaxDepth, Acc, Ans
%                           ) :-
%         CurTerm =.. [ _ | Args ],
%         NewCurDepth is CurDepth + 1,
%         number_with( NewCurDepth, Args, NumberedArgs ),
%         append( Terms, NumberedArgs, NewTerms ),
%         list_subterms_up_to_depth( NewTerms, MaxDepth, [ CurTerm | Acc ],
%                                    Ans
%                                  ).
%
%
% % Convert the list in arg2 to arg3, by pairing each element with arg1.
%
% number_with( _, [], [] ).
%
% number_with( N, [ H | T ], [ (N , H) | Ans ] ) :-
%         number_with( N, T, Ans ).
```

```
% I have rewritten the above to the form given below, which is not strictly
% tail-recursive, but does not use append/3 to extend the list of
% (depth , term) pairs that are to be processed.  We probably have enough memory
% for the execution stack, but the use of append may slow the program down in
% extreme cases.
%
% Additionally, now that I have a truly recursive version of
% aux_subterms_up_to_depth/4, I can avoid building an explicit list of
% (depth , subterm)  pairs.  [FK]


%% list_subterms_up_to_depth( + Term, + MaxDepth, - Subterms ) :
%% Produces a list of all subterms of Term upto the given depth.

list_subterms_up_to_depth( Term, MaxDepth, Subterms ) :-
        aux_subterms_up_to_depth( [ Term ], MaxDepth, Subterms, [] ).


% aux_subterms_up_to_depth( + Terms, + MaxDepth, - Subterms, - End ) :
%    - Terms is a list of subterms, all at the same level of the original term;
%    - MaxDepth is the maximum further depth to which we should descend;
%    - Subterms is an _open_ list of subterms (upto the maximum depth) obtained
%      from Terms;
%    - End is the end of the open list.
% This is an auxiliary of list_subterms_up_to_depth/3.

aux_subterms_up_to_depth( [], _MaxDepth, End, End ).

aux_subterms_up_to_depth( [ Term | Terms ], MaxDepth, Subterms, End ) :-
        (
            ( \+ compound( Term ) ; MaxDepth =< 0 )
        ->                                 % Term has no interesting subterms
            Subterms = [ Term | RestOfSubterms ]
        ;
                                           % Term has interesting subterms
            Subterms = [ Term | ArgSubterms ],
            Term =.. [ _ | Args ],
            NewMaxDepth is MaxDepth - 1,
            aux_subterms_up_to_depth( Args,        NewMaxDepth,
                                      ArgSubterms, RestOfSubterms
                                    )
        ),
        aux_subterms_up_to_depth( Terms, MaxDepth, RestOfSubterms, End ).
```

```
%% check_list_for_duplicates( + List ) :
%% Checks whether the list contains duplicates, i.e., at least two identical
%% terms ("identical" as opposed to "unifiable").
%
% Ronald's original version was very elegant:
%
%    check_list_for_duplicates( List ) :-
%            setof( X, member( X, List ), Set ),
%            length( List, N ),
%            length( Set, M ),
%            N \= M.
%
% I have replaced it with the following in the interest of "efficiency": if
% a duplicate is found early, there is no need to go through the entire list.
% The worst-case cost should be about the same, i.e., quadratic in the length of
% the list (in the original version this is hidden within setof/3). [FK]

check_list_for_duplicates( List ) :-
        % append/3 is used to generate various splittings of List:
        append( _Prefix, [ Term | Postfix ], List ),
        identical_member( Term, Postfix ),
        !.
```

```
%%---------------------------------------------------------------------------------
%% get_printable_term_equation( + Term, - Head, - List ) :
%% Returns the equation of a term as a list of strings.
%% Head is a string containing the initial variable of the equation.
%% List is a list of strings containing parts of the equation.

get_printable_term_equation( Term, Head, List ) :-
        get_term_equation( Term, Eq, H ),
        swritef( Head, '%w\n', [ H ] ),
        get_printable_list( Eq, List ).


% Convert a list of equations (arg1) to a list of their string forms.

get_printable_list( [], [] ).

get_printable_list( [ ( A = B ) | T ], [ String | Rest ] ) :-
        swritef( String, '%w = %w\n', [ A, B ] ),
        get_printable_list( T, Rest ).


%% get_term_equation( Term, EquationList, HeadVar ) obtains a list of
%% equations EquationList corresponding to the cyclic term Term, in which
%% HeadVar is the variable corresponding to Term.
%% Added conversion to more sensible variable names. [FK]

get_term_equation( Term, EquationList, HeadVar ) :-
        get_equation( Term, Equation ),
%        clean_equation( Equation, CleanEquation ),          % [FK]
        Equation = CleanEquation,                            % [FK]
        get_equation_with_variables( CleanEquation,
                                     UnsortedEquationList, HeadVar
                                   ),
        % ADDED:
        mk_variable_dictionary( p( HeadVar, UnsortedEquationList ),
                                VarDict
                              ),
        bind_variables_to_names( VarDict ),
        sort( UnsortedEquationList, EquationList ).
```

```
%%-----------------------------------------------------------------------------
%% get_equation( Term, Equation )  gets the equation corresponding to a term in
%% the form of a list of equalities in which the cyclic points are marked with
%% x/1 markers. The argument of x/1 is the integer that is paired with the
%% replaced term.
%%
%% example:
%% ?-  X = [ a | Y ],  Y = [ b | Y ],  get_equation( X, E ).
%% X = [ a, b | ** ],
%% Y = [ b | ** ],
%% E = [ (0 , [ a | x( 1 ) ]), (1 , [ b | x( 1 ) ]) ].

get_equation( Term, Equation ) :-
        obtain_all_cyclic_subterms( Term, List ),
        number_list_starting_at( List, 1, NumberedList ),
% originally:
%         replace_loop( [ (0 , Term) ], NumberedList, [],
%                        EquationWithDuplicates
%                      ),
%         setof( X, member( X, EquationWithDuplicates ), Equation ).
%
% [FK] replace_loop/4 replaced by convert/5, which produces no duplicates.
%      The equations are reversed to obtain a more natural correspondence
%      between the subequations and the order of arguments in the original
%      term:
        convert( [ Term ], NumberedList,
                 [ (0 , NewTerm)], REquation, [ NewTerm ]
               ),
        reverse( REquation, Equation ).


%% obtain_all_cyclic_subterms( + Term, - List ) :
%% Create a list of all the cyclic subterms of this term.
%% A "cyclic term" in this context is a term whose main functor is involved in a
%% cycle, as opposed to a term that only contains cyclic subterms.  For example,
%%  ?-  X = f( X ), obtain_all_cyclic_subterms( t( a( X ), b( X ) ), L ).
%% will yield only  f( X ) and not, for example, a( X ).

obtain_all_cyclic_subterms( Term, List ) :-
        obtain_all_cyclic_subterms( [ Term ], [ Term ], root, [], List ).
```

```
%% obtain_all_cyclic_subterms( Terms, SeenBefore, N, Acc, Ans ) :
%%  - Terms are the terms that still have to be handled;
%%  - SeenBefore is the list of terms that have already been seen;
%%  - Root = root if we are at the root of the term;
%%  - Acc is an accumulator;
%%  - Ans will contain the list of different subterms.
%%
%% Additional explanation (FK):
%% When first seen, a cyclic subterm is added to SeenBefore.
%% Since it is cyclic, it will be seen again, and at that point it will be added
%% to the accumulator. This ensures that a term that satisfy cyclic_term/1 by
%% virtue of containing cyclic subterms will not be put on the list unless its
%% main functor is actually a part of the cycle.
%% MODIFIED by FK:
%%    1. Replaced counter with Root (i.e., just a flag).
%%    2. Suppressed repetitions in the resulting list.
%%    3. Replaced the call to append/3 with a recursive invocation. So the first
%%       argument is now always a list of remaining siblings.  Notice that this
%%       change makes SeenBefore shorter, but that is a good thing. There
%%       is no need to check whether a sibling has been seen before: all we care
%%       about is whether this term is identical with one of its ancestors.
%%    4. Suppressed addition of siblings to SeenBefore.

obtain_all_cyclic_subterms( [], _, _, Acc, Acc ) :- !.

obtain_all_cyclic_subterms( [ T | TS ], SeenBefore, noroot, Acc, List ) :-
        identical_member( T, SeenBefore ),
        !,                   % identical with an ancestor: should be in the list
        (
            identical_member( T, Acc )
        ->                                                    % avoid repetitions
            NAcc = Acc
        ;
            NAcc = [ T | Acc ]
        ),
        obtain_all_cyclic_subterms( TS, SeenBefore, noroot, NAcc, List ).

obtain_all_cyclic_subterms( [ T | TS ], SeenBefore, _Root, Acc, List ) :-
        (
            % No need to remember terms for which cyclic_term/1 fails, no need
            % to visit their  arguments.
            \+ cyclic_term( T )
        ->
            NAcc = Acc
        ;
            % since cyclic_term( T ) succeeded, so would compound( T ):
            T =.. [ _ | SubtermList ],
            obtain_all_cyclic_subterms( SubtermList, [ T | SeenBefore ],
                                        noroot, Acc, NAcc
                                      )
        ),
        % No need to remember that we have seen a sibling:
        obtain_all_cyclic_subterms( TS, SeenBefore, noroot, NAcc, List ).
```

```
%% number_list_starting_at( List, InitialNr, NumberedList )
%%  - List is the list to be numbered;
%%  - InitialNr is the number to start numbering with;
%%  - NumberedList is the result of numbering elements of List from InitialNr
%%    on.

number_list_starting_at( [], _, [] ).

number_list_starting_at( [ H | T ], N, [ (N , H) | A ] ) :-
        M is N + 1,
        number_list_starting_at( T, M, A ).
```

```
%%%%  The original form (reformatted, with some additional comments and minor
%%%% fixes by FK):
%
% %% replace_loop( + Agenda, + SubtermList, + DoneBefore, - Results ) :
% %% Replaces all subterms at cyclic positions with x/1 markers.
% %%  - Agenda is a list of pairs of numbers and terms that still have to be
% %%    handled (i.e., they or their terms may have to be replaced);
% %%  - SubtermList is a list of similar pairs, for subterms that have been
% %%    picked up by obtain_all_cyclic_subterms/2: it is occurrences of subterms
% %%    that are in this list (as second elements) that will be replaced with
% %%    x(N) items, where N is the first element of the pair that contains the
% %%    subterm.
% %%  - DoneBefore is a list of subterms that have already been handled;
% %%  - Results is the list of the modified subterms.
%
% replace_loop( [], _, _, [] ).
%
% replace_loop( [ (I , Term) | RestAgenda ], SubtermList, DoneBefore,
%               [ (I , Result1) | Results ]
%             ) :-
%         replace_term_proper( Term, SubtermList, NewAgenda1, Result1 ),
%         findall( (N , AgendaItem),
%                  ( member( (N , AgendaItem), NewAgenda1 ),
%                    \+ identical_member( AgendaItem, DoneBefore ) ),
%                   RealNewAgenda
%                 ),
%         append( RestAgenda, RealNewAgenda, NewAgenda ),
%         replace_loop( NewAgenda, SubtermList, [ Term | DoneBefore ],
%                       Results
%                     ).
%
%
% %% replace_term( + Term, + SubtermList, - NewAgenda, - Result ) :
% %% Replaces all subterms of a term with cycle markers x/1.
% %% Also returns all replaced subterms in NewAgenda.
%
% replace_term( Term, SubtermList, [ (N , Term) ], x( N ) ) :-
%         identical_member2( (N , Term), SubtermList ),
%         !.
%
% replace_term( Term, SubtermList, NewAgenda, Result ) :-
%         replace_term_proper( Term, SubtermList, NewAgenda, Result ).
%
%
% %% replace_term_proper( + Term, + SubtermList, - NewAgenda, - Result ) :
% %% Acts like replace_term/4 but skips any replacements in the root of the
% %% term.
%
% replace_term_proper( Term, SubtermList, NewAgenda, Result ) :-
%         compound( Term ),
%         !,
%         Term =.. [ Functor | Args ],
%         replace_term_list( Args, SubtermList, NewAgenda, Results ),
%         Result =.. [ Functor | Results ].
%
% replace_term_proper( Term, _SubtermList, _NewAgenda, Term ) :-
%         \+ compound( Term ).
%
%
% %% replace_term_list( + Terms, + SubtermList, - NewAgenda, - Results ) :
% %% Straightforwardly extends replace_term/4 to act on lists of terms instead
% %% of on single terms.
%
```

```
% replace_term_list( [], _, [], [] ).
%
% replace_term_list( [ Term | List ], SubtermList, NewAgenda, Results ) :-
%         replace_term( Term, SubtermList, NewAgenda1, Result1 ),
%         replace_term_list( List, SubtermList, NewAgenda2, Results2 ),
%         append( NewAgenda1, NewAgenda2, NewAgenda ),
%         Results = [ Result1 | Results2 ].
```

```
%% After finally having understood (?) the code above, I rewrote it from
%% scratch, as follows [FK]:

%% convert( + Terms, + CyclicSubterms, + Accumulator, - Equation, - NewTerms ) :
%%    - Terms is (the remainder of) a list containing one terms, or all the
%%      arguments of one term (sibling terms);
%%    - CyclicSubterms is a cyclic subterms (produced by
%%      obtain_all_cyclic_subterms/2), each paired with a unique number;
%%    - Accumulator is the accumulator for the sub-equations of the entire
%%      equation: each element is a pair consisting of a number and a term;
%%    - Equation is the accumulator, augmented with information produced in this
%%      instance of convert/5;
%%    - NewTerms is a list with the converted forms of the input terms.
%%
%% Conversion consists in replacing each occurrence of a (sub)term that is
%% identical to one of the terms on CyclicSubterms with x( N ), where N is the
%% number that is associated with the term on CyclicSubterms.  For each such
%% replacement a "subequation" of the form (N , Term) must appear on Equation:
%% however, care is taken not to allow repetitions on that list.  Replacement is
%% carried out also for the arguments of the cyclic subterms: to prevent
%% infinite looping, it is not carried out if an argument already has its number
%% on the Equation list.

convert( [], _, Acc, Acc, [] ).

convert( [ T | Ts ],  CyclicSubterms, Acc, Equation, [ x( N ) | NewTs ] ) :-
        identical_member2( (N , T), CyclicSubterms ),
        !,                                              % a cyclic term: replace
        (
            member( (N , _), Acc )
        ->
            % Break the loop: don't add to Equation, don't replace in arguments
            % (if any):
            NewAcc = Acc
        ;
            % Add the term to Equation, and run through its arguments, if any:
            NAcc = [ (N , NewT) | Acc ],
            (
                compound( T )
            ->
                T =.. [ F | Args ],
                convert( Args, CyclicSubterms, NAcc, NewAcc, NewArgs ),
                NewT =.. [ F | NewArgs ]
            ;
                NewT   = T,
                NewAcc = NAcc
            )
        ),
        convert( Ts, CyclicSubterms, NewAcc, Equation, NewTs ).
```

```
convert( [ T | Ts ],  CyclicSubterms, Acc, Equation, [ NewT | NewTs ] ) :-
        % \+ identical_member2( (N , T), CyclicSubterms ),
        % Don't add to equation, but convert arguments (if any):
        (
            compound( T )
        ->
            T =.. [ F | Args ],
            convert( Args, CyclicSubterms, Acc, NewAcc, NewArgs ),
            NewT =.. [ F | NewArgs ]
        ;
            NewT   = T,
            NewAcc = Acc
        ),
        convert( Ts, CyclicSubterms, NewAcc, Equation, NewTs ).
```

```
%%---------------------------------------------------------------------------------
%% get_equation_with_variables( + Equation, - EquationList, - HeadVar ) :
%% turns an equation with x/1 markers into an equation with variables for the
%% cyclic points
%%
%% Example:
%% ?- X = [ a | Y ], Y = [ b | Y ],  get_equation( X, E ),
%%    get_equation_with_variables( E, EL, HV ).
%% X = [ a, b | ** ],
%% Y = [ b | ** ],
%% E = [ (0 , [ a | x( 1 ) ]), (1 , [ b | x( 1 ) ]) ],
%% EL = [ HV=[ a | _G930 ], _G930=[ b | _G930 ] ] .

get_equation_with_variables( Equation, EquationList, HeadVar ) :-
        variable_list( Equation, VarList ),
        member( ( 0, HeadVar ), VarList ),
        convert_markers_to_vars( Equation, VarList, EquationList ).


%% variable_list( + Equation, - VariableList ) :
%% Gets a list of numbered variables for every term in the list of equations.

variable_list( [], [] ).

variable_list( [ ( N, _ ) | T ], [ ( N, _ ) | R ] ) :-
        variable_list( T, R ).


% convert_markers_to_vars( + Equation, + VarList, - NewEquation ) :
% For each pair in Equation:
%   - replace the number by the corresponding variable in VarList;
%   - convert the term by replacing each x( N ) marker with the
%     N'th variable in VarList.

convert_markers_to_vars( [], _, [] ).

convert_markers_to_vars( [ (N , T) | Rest ], VarList, [ V = NT | RestAns ] ) :-
        member( (N , V), VarList ),
        replace_markers_by_variables( T, VarList, NT ),
        convert_markers_to_vars( Rest, VarList, RestAns ).


%% replace_markers_by_variable( + Term, + NumberedVarList, - NewTerm ) :
%% Replaces cyclic positions, marked with x/1, with corresponding variables from
%% a numbered list of variables.
%%
%% The original version spuriously unified a variable term with x( N ), which
%% led to wrong results.  This is fixed below.  [FK]

replace_markers_by_variables( T, _VL, T ) :-
        \+ compound( T ),
        !.

replace_markers_by_variables( x( N ), VL, V ) :-
        !,
        member( (N , V), VL ).

replace_markers_by_variables( T, VL, NewT ) :-
        T =.. [ F | Args ],
        replace_markers_by_variables_in_list( Args, VL, NewArgs ),
        NewT =.. [ F | NewArgs ].
```

```
% replace_markers_by_variables_in_list( +Terms, +VarList, -Vars ) :
% Invokes replace_marker_by_variable for each item on the list.

replace_markers_by_variables_in_list( [], _, [] ).

replace_markers_by_variables_in_list( [ T | Ts ], VL, [ V | Vs ] ) :-
        replace_markers_by_variables( T, VL, V ),
        replace_markers_by_variables_in_list( Ts, VL, Vs ).
```

```
%%-----------------------------------------------------------------------------
%%%% [FK}: With the new version of convert/5, all this stuff seems to be
%%%%%      unnecessary:
%
% %% The following predicates are used to remove redundancies from the equation.
% %%
% %% the following exemplifies what kind of redundancies can occur:
% %% ?- X = [ a | X ],  get_equation( X, E ),
% %%    get_equation_with_variables( E, EL, HV ).
% %% X = [ a | ** ],
% %% E = [  ( 0, [ a | x( 1 ) ] ), ( 1, [ a | x( 1 ) ] ) ],
% %% EL = [ HV=[ a | _G735 ], _G735=[ a | _G735 ] ] .
% %%
% %% ?- X = [ a | X ],  get_equation( X, E ),  clean_equation( E, F ),
% %%    get_equation_with_variables( F, EL, HV ).
% %% X = [ a | ** ],
% %% E = [  (0 , [ a | x( 1 ) ]), (1 , [ a | x( 1 ) ]) ],
% %% F = [  (0 , [ a | x( 0 ) ]) ],
% %% EL = [ HV=[ a | HV ] ] .
%
% %% clean_equation( + Equation, - Result ) :
% %% Checks if there is a redundancy, and if so gets rid of it.
%
% clean_equation( Equation, Result ) :-
%         identical_member( (0 , Init), Equation ),
%         find_duplicate_eq( Equation, Init, N ),
%         !,
%         doclean_equation( Equation, N, Result ).
%
% clean_equation( Equation, Equation ).
%
%
% %% doclean_equation( Equation, SuperfluousVar, Result )
% %% Actually gets rid of the redundancy.
%
% doclean_equation( [], _, [] ).
%
% doclean_equation( [ (N , _) | Rest ], N, Ans ) :-
%         !,
%         doclean_equation( Rest, N, Ans ).
%
% doclean_equation( [ (N , Term) | Rest ], M, [ (N , NewTerm) | Ans ] ) :-
%         replace_markers_with_initial( Term, M, NewTerm ),
%         doclean_equation( Rest, M, Ans ).
%
%
% %% replace_markers_with_initial( Term, SuperfluousVar, Result )
% %% replaces markers of the superfluous variable with markers for the initial
% %% variable
%
% replace_markers_with_initial( x( N ), N, x( 0 ) ) :- !.
%
% replace_markers_with_initial( x( K ), N, x( K ) ) :- K \= N,  !.
%
% replace_markers_with_initial( Term, N, Result ) :-
%         Term =.. [ H | TS ],
%         replace_markers_with_initial_list( TS, N, RS ),
%         Result =.. [ H | RS ].
%
%
% %% replace_markers_with_initial_list( Terms, SuperfluousVar, Results )
% %% straightforwardly extends replace_markers_with_initial/3 to act on lists
%
```

```
% replace_markers_with_initial_list( [], _, [] ).
%
% replace_markers_with_initial_list( [ H | T ], N, [ R | RS ] ) :-
%         replace_markers_with_initial( H, N, R ),
%         replace_markers_with_initial_list( T, N, RS ).
%
%
% %% find_duplicate_eq( Equation, InitialTerm, SuperfluousVar ) :
% %% Produce information about a duplicate.
%
% find_duplicate_eq( [ (0 , Init) | T ], Init, M ) :- !,
%         find_duplicate_eq( T, Init, M ).
%
% find_duplicate_eq( [ (M , Init) | _ ], Init, M ) :- !.
%
% find_duplicate_eq( [ _ | T ], Init, M ) :-
%         find_duplicate_eq( T, Init, M ).
```

```
%%----------------------------------------------------------------------------
%% identical_member( + term, + list ) :
%% Succeed if the list contains this term (as opposed to something that is
%% unifiable with this term).

identical_member( X, Items ) :-
        member( T, Items ),
        X == T,
        !.


%% identical_member2( (-+ number , + term), + list of pairs ) :
%% Succeed if the list contains a pair whose second element is identical to the
%% second element of arg1, and whose first element unifies with the first
%% element of arg1.

identical_member2( (N , Term), Items ) :-
        member( (N, T), Items ),
        Term == T,
        !.

%%============================================================================
```