UTIL:UTIL.HLP   Updated: 22 October 81

This directory contains the Prolog utilities used by the Mecho Project.

On the ERCC DEC10 in Edinburgh all this stuff is contained in [140,143,UTIL]. The contents of this directory is now being thrown onto outgoing tapes containing the latest DEC10 Prolog system, on the offchance they might be of some use elsewhere.

The following files are present:

UTIL.MIC MUTIL.MIC	These are MIC command files for loading two standard Utilities packages (a full one and a minimal one). The EXE files produced are stored elsewhere in the Mecho library ([400,444]). These MIC files are rather hairy, their main purpose being to support an automatic reloading hack I use (they will undoubtably be useless elsewhere). The interesting work is done in the following files:
UTIL MUTIL	These are Prolog files which contain the commands to load (by compiling/consulting) the various sources which make up the above packages. Either of these files just needs consulting to do the loading.
UTIL.TXT	This file contains a list of all the predicates provided by the Utilities package. The predicates are listed first by module (source file) and then alphabetically.
WRITEF.*	A short documentary note on the formatted write utility (see WRITEF.PL).
UTIL.MSS	A rather old start to some documentation, It is horribly incomplete and uses as yet undefined SCRIBE macros so there is only the source form.
UTIL.OPS ARITH.OPS	These files declare the (syntactic) operators used by the packages.
*.PL	These are all the Prolog source files for the Utility packages.

Unfortunately there is no decent documentation for the utilities apart from the list of predicates in UTIL.TXT. This is a matter which I have been meaning to deal with for years. However most of the routines are pretty short and straightforward. I have not yet brought them all up to my current commenting standards though. Sorry about the mess.

The real goody that you may enjoy is the rational arithmetic package which can be found in LONG.PL. This provides all the standard arithmetic operations over arbitrary precision rational numbers, plus some more things like logs, square roots, and a poor mans trig function hack. There is also a symbolic simplifier/evaluator which makes use of LONG in TIDY.PL. Both these files are fairly substantial but they both contain documentation on whats going on. The rational arithmetic package is pretty fast considing this was not a deliberate intention. It needs compiling of course - try it and see!

I hope these may be of some use to you,

Lawrence Byrd
Artificial Intelligence
Hope Park Square
University of Edinburgh
Edinburgh
SCOTLAND UK

Network mail etc,

BYRD on the ERCC DEC10 (Edinburgh) (ppn = [400,441])
(If thats where you are, or you can get through)

BYRD@MIT-AI
(Regular ARPANET mailing address)

```
UTIL:UTIL.CNG  Changes in the Prolog Utilities
==============================================

------------------------------------------------------------------------------
Lawrence  13 September 81

Another fix to LONG.PL from Richard. Moved his latest version to UTIL: and
then reloaded the UTIL image in [400,444].
------------------------------------------------------------------------------
Lawrence  9 September 81.

Fix by Richard to LONG.PL involving evaluation of arcsin and arccos.
We are still waiting for someone to do this trig evaluation properly,
ie to use polynomial approximations of some flavour. (Interested?).
Old version of LONG.PL archived as LONG.001.
------------------------------------------------------------------------------
```

Utilities, by module.

```
UTIL:EDIT.PL

edit(File)
redo(File)

UTIL:FILES.PL

check_exists(File)
file_exists(File)
open(File)
open(Old,File)
close(File,Old)
delete(File)

UTIL:WRITEF.PL

 typrint(X)
prlist(List)
prconj(Conj)
prexpr(Expr)
writef(Format)
writef(Format,List)

UTIL:TRACE.PL

error(Format,List,Action)
tlim(Tlimit)
ton(Name)
toff(Name)
toff
 trace(Format,Condition)
trace(Format,List,Condition)

UTIL:READIN.PL

 ead_in(Sentence)

UTIL:LISTRO.PL

append(List1,List2,List3)
disjoint(List)
last(Element,List)
listtoset(List,Set)
nextto(X,Y,List)
numlist(N1,N2,Numberlist)
pairfrom(List,A,B,Rest)
perm(List1,List2)
perm2(X,Y,A,B)
remove_dups(List,Set)
rev(List1,List2)
select(Element,List,Rest)
sumlist(NumList,Sum)

UTIL:SETROU.PL
```

```
intersect(Set1,Set2,ISet)

member(Element,Set)
memberchk(Element,Set)
nmember(Element,Set,N)
seteq(Set1,Set2)
subset(Subset,Superset)
subtract(Set1,Set2,Subset)
union(Set1,Set2,USet)

UTIL:INVOCA.PL

&(Goal1,Goal2)
\\(Goal1,Goal2)
any(Goallist)
binding(N,Goal)
findall(Var,Goal,List)
for(N,Goal)
forall(Goal1,Goal2)
nobt(Goal)
not(Goal)
thnot(Goal)

UTIL:APPLIC.PL

apply(Pred,Args)
checkand(Pred,Conj)
checklist(Pred,List)
mapand(Pred,Conj1,Conj2)
maplist(Pred,List1,List2)
convlist(Pred,List1,List2)
some(Pred,List)
sublist(Pred,List1,List2)

UTIL:MULTIL.PL

mlmaplist(Pred,Lists)
mlmaplist(Pred,Lists,V)
mlmaplist(Pred,Lists,Vin,Vout)
mlmember(Elements,Lists)
mlselect(Elements,Lists,Rests)

UTIL:FLAGRO.PL

flag(Flag,Old,New)

UTIL:CMISCE.PL

cgensym(Prefix,PossVar)
gensym(Prefix,Var)
concat(Atom1,Atom2,Atom3)

UTIL:IMISCE.PL

continue
\=(X,Y)
casserta(X)
cassertz(X)
clean
diff(X,Y)
occ(Goal)
```

```
subgoal(exact,Goal)

UTIL:STRUCT.PL

subst(Substitution,Old,New)
occ(X,Term,N)
variables(Term,VarSet)

UTIL:TIDY.PL

tidy(Expr,TidiedExpr)

UTIL:LONG.PL

number(Rational)
eval(Command)
eval(Expr,Answer)
portray_number(Rational)
```

Utilities, alphabetically by name.

&(Goal1,Goal2)	util:invoca.pl
\=(X,Y)	util:imisce.pl
\\(Goal1,Goal2)	util:invoca.pl
any(Goallist)	util:invoca.pl
append(List1,List2,List3)	util:listro.pl
apply(Pred,Args)	util:applic.pl
binding(N,Goal)	util:invoca.pl
casserta(X)	util:imisce.pl
cassertz(X)	util:imisce.pl
cgensym(Prefix,PossVar)	util:cmisce.pl
check_exists(File)	util:files.pl
checkand(Pred,Conj)	util:applic.pl
checklist(Pred,List)	util:applic.pl
clean	util:imisce.pl
close(File,Old)	util:files.pl
concat(Atom1,Atom2,Atom3)	util:cmisce.pl
ontinue	util:imisce.pl
onvlist(Pred,List1,List2)	util:applic.pl
delete(File)	util:files.pl
diff(X,Y)	util:imisce.pl
disjoint(List)	util:listro.pl
edit(File)	util:edit.pl
error(Format,List,Action)	util:trace.pl
eval(Command)	util:long.pl
eval(Expr,Answer)	util:long.pl
file_exists(File)	util:files.pl
findall(Var,Goal,List)	util:invoca.pl
flag(Flag,Old,New)	util:flagro.pl
for(N,Goal)	util:invoca.pl
forall(Goal1,Goal2)	util:invoca.pl
gcc(Goal)	util:imisce.pl
gensym(Prefix,Var)	util:cmisce.pl
intersect(Set1,Set2,ISet)	util:setrou.pl
last(Element,List)	util:listro.pl
listtoset(List,Set)	util:listro.pl
apand(Pred,Conj1,Conj2)	util:applic.pl
aplist(Pred,List1,List2)	util:applic.pl
member(Element,Set)	util:setrou.pl
memberchk(Element,Set)	util:setrou.pl
mlmaplist(Pred,Lists)	util:multil.pl
mlmaplist(Pred,Lists,Vin,Vout)	util:multil.pl
mlmaplist(Pred,Lists,V)	util:multil.pl
mlmember(Elements,Lists)	util:multil.pl
mlselect(Elements,Lists,Rests)	util:multil.pl
nextto(X,Y,List)	util:listro.pl
nmember(Element,Set,N)	util:setrou.pl
nobt(Goal)	util:invoca.pl
not(Goal)	util:invoca.pl
number(Rational)	util:long.pl
numlist(N1,N2,Numberlist)	util:listro.pl
occ(X,Term,N)	util:struct.pl
open(File)	util:files.pl
open(Old,File)	util:files.pl
pairfrom(List,A,B,Rest)	util:listro.pl
perm(List1,List2)	util:listro.pl

perm2(X,Y,A,B)	util!listro.pl
portray_number(Rational)	util!long.pl
prconj(Conj)	util!writef.pl
prexpr(Expr)	util!writef.pl
prlist(List)	util!writef.pl
read_in(Sentence)	util!readin.pl
redo(File)	util!edit.pl
remove_dups(List,Set)	util!listro.pl
rev(List1,List2)	util!listro.pl
select(Element,List,Rest)	util!listro.pl
seteq(Set1,Set2)	util!setrou.pl
some(Pred,List)	util!applic.pl
subgoal(exact,Goal)	util!imisce.pl
sublist(Pred,List1,List2)	util!applic.pl
subset(Subset,Superset)	util!setrou.pl
subst(Substitution,Old,New)	util!struct.pl
subtract(Set1,Set2,Subset)	util!setrou.pl
sumlist(NumList,Sum)	util!listro.pl
thnot(Goal)	util!invoca.pl
tidy(Expr,TidiedExpr)	util!tidy.pl
tlim(Tlimit)	util!trace.pl
on(Name)	util!trace.pl
toff	util!trace.pl
toff(Name)	util!trace.pl
trace(Format,Condition)	util!trace.pl
trace(Format,List,Condition)	util!trace.pl
ttyprint(X)	util!writef.pl
union(Set1,Set2,USet)	util!setrou.pl
variables(Term,VarSet)	util!struct.pl
writef(Format)	util!writef.pl
writef(Format,List)	util!writef.pl

Department of Artificial Intelligence
University of Edinburgh

BAG MANIPULATION UTILITY ROUTINES

Source: R. A. O'Keefe
Program Issued: 23 September 81
Documentation: 25 September 1981

1. Description

Bags are a generalisation of sets, in which a given element may be present several times. Just as a set may be represented by its characteristic function (a mapping from some class to truth values), so may a bag be represented by its characteristic function, whose range is the non-negative integers. These routines manipulate Prolog data-structures encoding bags as tabular characteristic functions.

X is an encoded bag if

- it is the term 'bag', representing the empty bag.

- it is the term bag(Element, Count, RestOfBag) where RestOfBag is a term representing a bag, Count is a (strictly) positive integer, and Element is any Prolog term. To make these representations canonical, Element must precede all the other elements of the bag, in the sense of '@<'.

For example, the bag {a,b,c,d,c,a,d,c,a,e,d,c} would be represented by the Prolog term bag(a,3,bag(b,1,bag(c,4,bag(d,3,bag(e,1,bag))))).

2. How to Use the Program

This library may already be loaded into UTIL. To see if it is, type 'listing(is_bag)' to UTIL. If it shows you any clauses, all these predicates should be available at once. Otherwise, you may either compile or consult the file 'Util:BagUtl.Pl'. The following predicates are then available.

bag_inter(+Bag1, +Bag2, -Inter)
takes the intersection of two bags. The count of an element in the result is the minimum of its count in Bag1 and its count in Bag2.

bag_to_list(+Bag, -List)
converts a Bag to a List. Each element of the Bag will appear in the List as many times as it occurs in {the abstract value of} the Bag. E.g. [% a:2, b:3, c:1 %] => [a,a, b,b,b, c].

bag_to_set(+Bag, -SetList)
: converts a Bag to a Set, i.e. to a List in which each element of the Bag occurs exactly once.

bag_union(+Bag1, +Bag2, -Union)
: takes the union of two bags. The count of an element in the result is the sum of its count in Bag1 and its count in Bag2.

bagmax(+Bag, ?Elem)
: unifies Elem with that element of Bag which has the greatest count. NB: this is **not** an ordering on the elements themselves, but the ordinary arithmetic ordering on their frequencies. Predicates to select the alphabetically (@<) least and greatest elements could be supplied if anyone wanted them. bagmax returns the commonest one.

bagmin(+Bag, ?Elem)
: unifies Elem with that element of Bag which has the least count. In other words, with the rarest object actually in the Bag.

checkbag(+Pred, +Bag)
: is an analogue of checklist for bags. It succeeds if Pred(Elem,Count) is true for every element of the Bag and its associated Count.

is_bag(+Bag)
: succeeds if Bag is a well-formed bag representation. Not all terms which resemble bags are bags: bag(1,a,bag) is not {'a' is not a positive integer} and bag(b,1,bag(a,1,bag)) is not {'b' is not alphabetically less than 'a}.

length(+Bag, -Total, -Distinct)
: unifies Distinct with the number of distinct elements in the Bag and Total with the sum of their counts. Hence Total >= Distinct. This name was chosen to agree with the notation for lists (sets).

ist_to_bag(+List, -Bag)
: converts a List to a Bag. The elements of the list do not need to be in any special order.

make_sub_bag(+Bag, -SubBag)
: A sub_bag predicate would have two uses: testing whether one already existing bag is a sub-bag of another, and generating the sub-bags of a given bag. Since bags have so many sub-bags, this second use is likely to be rare, and has been split out as make_sub_bag. Given a Bag, make_sub_bag will backtrack through all its SubBags.

mapbag(+Pred, +BagIn, -BagOut)
: is analogous to maplist. It applies Pred(Element,Transformed) to each element of the Bag, generating a transformed bag. The counts are not given to Pred, but are preserved. If several elements are mapped to the same transformed element, their counts will be added, so the result will always be a proper bag. For the same reason, the order of results in the answer

will be alphabetic, rather than the order of the elements in the input bag.

member(?Elem, -Count, +Bag)
can be used to backtrack through all the members of a bag or to test whether some specific object is in a bag. In either case Count is set to the element's count. NB if Elem is not an element of the bag, member will **not** unify Count with 0, it will **fail**.

portray_bag(+Bag)
These predicates assume that people will never want to type bags, but will always create them using bag utilities. Hence the internal representation is meant for efficiency rather than readability. If you add the clause

```
    portray(Bag) :- portray_bag(Bag).
```

to your program you will get a prettier 'print'ed form (but not a 'write'n form, alas). A bag is printed between '[%' and '%]', with elements followed by a colon and their count, and separated by commas. For example,

```
    ?- list_to_bag([a,c,d,e,f,a,f,d,c,d,e,f,g], B).
    B = [% a:2, c:2, d:3, e:2, f:3, . g:1 %]
```

test_sub_bag(+SubBag, +Bag)
tests whether SubBag is a sub bag of Bag. This is redundant, as

```
    test_sub_bag_2(Sb, Bg) :-
            bag_inter(Sb, Bg, In),
            In = Sb.
```

It is cheaper and clearer to use test_sub_bag.

. Program Requirements

checkbag and mapbag require the utility apply/2. No other utilities are used, but BagUtl cannot be used with versions of Prolog prior to Version 3. The database is not affected.

The compiled code occupies about 2k.

```
/* UTIL : Load the (full) Utilities Package

                                                  UTILITY
                                                  Lawrence
                                                  Updated: 11 May 81
*/


%% See UTIL.MIC which calls this and then sets up a core image %%


% LONG and TIDY are now included in UTIL
%
% The logical name "util:" is assumed to point to the right area, if you
% are not using TOPS10 version 7.01, or don't understand logical names,
% then just edit them all out.


:- [
                'util:util.ops',        % General operator declarations
                'util:arith.ops'        % Arithmetic operator declarations
   ].


:- compile([
                'util:files.pl',        % Manipulate files
                'util:writef.pl',       % Formatted write (writef)
                'util:trace.pl',        % Tracing routines
                'util:readin.pl',       % Read in a sentence
                'util:listro.pl',       % List routines
                'util:setrou.pl',       % Set routines
                'util:applic.pl',       % Application routines
                'util:multil.pl',       % Multi list routines
                'util:flagro.pl',       % Flag handling
                'util:struct.pl',       % Structure crunching
                'util:cmisce.pl',       % Miscellaneous

                'util:long.pl',         % Rational arithmetic package
                'util:tidy.pl'          % Expression tidy/evalutor
              ]).

. '[
                'util:edit.pl',         % Jump to FINE and back
                'util:invoca.pl',       % Invocation routines
                'util:imisce.pl'        % Miscellaneous
   ].
```

```
/* MUTIL : Load a minimal Utilities Package

                                                UTILITY
                                                Lawrence
                                                Updated: 2 August 81
*/


%% See MUTIL.MIC which calls this and then sets up a core image %%

% The logical name "util:" is assumed to point to the right area, if you
% are not using TOPS10 version 7.01, or don't understand logical names,
% then just edit them all out.

% The following files, found in UTIL, have been ommited for MUTIL to
% make it smaller:
%                       LONG.PL
%                       TIDY.PL
%                       READIN.PL
%                       MULTIL.PL
%
;   e mainly the rational arithmetic package, plus a couple of less useful
%  / bits.


:- [
                'util:util.ops',        % General operator declarations
                'util:arith.ops'        % Arithmetic operator declarations
   ].


:- compile([
                'util:files.pl',        % Manipulate files
                'util:writef.pl',       % Formatted write (writef)
                'util:trace.pl',        % Tracing routines
                'util:listro.pl',       % List routines
                'util:setrou.pl',       % Set routines
                'util:applic.pl',       % Application routines
                'util:flagro.pl',       % Flag handling
                'util:struct.pl',       % Structure crunching
                'util:cmisce.pl'        % Miscellaneous
  /          ]).

:- [
                'util:edit.pl',         % Jump to FINE and back
                'util:invoca.pl',       % Invocation routines
                'util:imisce.pl'        % Miscellaneous
   ].
```

```
/* UTIL.OPS : Operator declarations for UTIL and other Mecho programs

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 2 August 81
*/

  :- op(1100,xfy,(\\)).                     % see INVOCA.PL
  :- op(950,xfy,#).                         % Used for disjunction
  :- op(850,xfy,&).                         % Used for conjunction
  :- op(710,fy,[not,thnot]).                % see INVOCA.PL
  :- op(700,xfx,\=).                        % see IMISCE.PL

                                    % Conveniences
  :- op(300,fx,edit).                       % see EDIT.PL
  :- op(300,fx,redo).
  :- op(300,fx,tlim).                       % see TRACE.PL
  :- op(300,fx,ton).
  :- op(300,fx,toff).
```

```
/* ARITH.OPS : Operator declarations for arithmetic expressions
                Now present in UTIL, used by PRESS and others

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 2 August 81
*/

  :- op(500,yfx,[++,--]).
  :- op(400,yfx,[div,mod]).
  :- op(300,xfy,[:,^]).
```

```
; UTIL.MIC  -  Load Util          '<silence>
;
;        This junk allows for automatic loading believe it or not
;
;        Call as:        /util           - to load util (normal use)
;                        /util auto      - used by MAKSYS
;
.on error:backto death
.error ?
.on operator:backto death
.operator !
.goto cont
death::
*^C
*^C
.if ($a = "auto") .let e1 = "error"
! UTIL.MIC HALTED
.mic return
cont::
.let y = $date.["-",20], d = $date.[1,"-"]+" "+$y.[1,"-"]+" "+$y.["-",4]
.if ($d.[1] = "0") .let d = $d.[2,20]

;                                 Use latest version of Prolog
.run prolog[400,444]  '<revive>
* :- [util].
* :- version(''Utilities Package  ('d)
*Copyright (C) 1981 Dept. Artificial Intelligence. Edinburgh'').
* :- core_image,
*    display(''Utilities Package  ('d)''), ttynl,
*    display(''[Now includes LONG and TIDY]''), ttynl,
*    reinitialise.
.save util[400,444]
```

```
; MUTIL.MIC  -  Load Mutil      '<silence>
;
;       This junk allows for automatic loading believe it or not
;
;       Call as:        /mutil          - to load mutil (normal use)
;                       /mutil auto     - used by MAKSYS
;
.on error:backto death
.error ?
.on operator:backto death
.operator !
.goto cont
death::
*^C
*^C
.if ($a = "auto") .let e1 = "error"
! MUTIL.MIC HALTED
.mic return
cont::
.let y = $date.["-",20], d = $date.[1,"-"]+" "+$y.[1,"-"]+" "+$y.["-",4]
.if ($d.[1] = "0") .let d = $d.[2,20]

;                                Use latest version of Prolog
.run prolog[400,444]  '<revive>
* :- [mutil].
* :- version(''Minimal Utilities Package  ('d)
*Copyright (C) 1981 Dept. Artificial Intelligence. Edinburgh'').
* :- core_image,
*    display(''Minimal Utilities Package  ('d)''), ttynl,
*    reinitialise.
.save mutil[400,444]
```

```
/* UTIL.DEF : XREF definitions for UTIL procedures

                                                Lawrence
                                                Updated: 2 August 81


        This file defines all the predicates found in UTIL. It is intended
        for use with the Prolog Cross Referencer XREF, which will take
        these into account when producing the Cross reference Listing
        for your programs.
*/
```

NB - Currently already loaded in the standard XREF

```
% UTIL operators
%                       (from UTIL.OPS)

op( 1100,xfy,(\\) ).
op( 950,xfy,# ).
op( 850,xfy,& ).
op( 710,fy,[not,thnot] ).
op( 700,xfx,\= ).

op( 300,fx,edit ).
op( 300,fx,redo ).
op( 300,fx,tlim ).
op( 300,fx,ton ).
op( 300,fx,toff ).

%                       (from ARITH.OPS)
op(500,yfx,[++,--]).
op(400,yfx,[div,mod]).
op(300,xfy,[:,^]).


% UTIL procedures

known(     &(Goal1,Goal2),                                      utility ).
  applies( &(Goal1,Goal2), Goal1 ).
  applies( &(Goal1,Goal2), Goal2 ).
known(     \=(X,Y),                                             utility ).
known(     \\(Goal1,Goal2),                                     utility ).
  applies( \\(Goal1,Goal2), Goal1 ).
  applies( \\(Goal1,Goal2), Goal2 ).
known(     any(Goallist),                                       utility ).
  % Hairy applies...
known(     append(List1,List2,List3),                           utility ).
known(     apply(Pred,Args),                                    utility ).
  % Hairy applies...
known(     binding(N,Goal),                                     utility ).
  applies( binding(N,Goal), Goal ).
known(     casserta(X),                                         utility ).
known(     cassertz(X),                                         utility ).
known(     cgensym(Prefix,PossVar),                             utility ).
known(     check_exists(File),                                  utility ).
known(     checkand(Pred,Conj),                                 utility ).
  applies( checkand(Pred,Conj), Pred+1 ).
known(     checklist(Pred,List),                                utility ).
  applies( checklist(Pred,List), Pred+1 ).
known(     clean,                                               utility ).
known(     close(File,Old),                                     utility ).
known(     concat(Atom1,Atom2,Atom3),                           utility ).
known(     continue,                                            utility ).
known(     convlist(Pred,List1,List2),                          utility ).
  applies( convlist(Pred,List1,List2), Pred+2 ).
known(     delete(File),                                        utility ).
known(     diff(X,Y),                                           utility ).
known(     disjoint(List),                                      utility ).
known(     edit(File),                                          utility ).
known(     error(Format,List,Action),                           utility ).
  applies( error(Format,List,Action), Action ).
known(     eval(Command),                                       utility ).
known(     eval(Expr,Ans),                                      utility ).
known(     file_exists(File),                                   utility ).
known(     findall(Var,Goal,List),                              utility ).
  applies( findall(Var,Goal,List), Goal ).
known(     flag(Flag,Old,New),                                  utility ).
known(     for(N,Goal),                                         utility ).
  applies( for(N,Goal), Goal ).
known(     forall(Goal1,Goal2),                                 utility ).
  applies( forall(Goal1,Goal2), Goal1 ).
  applies( forall(Goal1,Goal2), Goal2 ).
   wn(     gcc(Goal),                                           utility ).
  applies( gcc(Goal), Goal ).
known(     gensym(Prefix,Var),                                  utility ).
known(     intersect(Set1,Set2,ISet),                           utility ).
known(     last(Element,List),                                  utility ).
known(     listtoset(List,Set),                                 utility ).
known(     mapand(Pred,Conj1,Conj2),                            utility ).
  applies( mapand(Pred,Conj1,Conj2), Pred+2 ).
known(     maplist(Pred,List1,List2),                           utility ).
  applies( maplist(Pred,List1,List2), Pred+2 ).
known(     member(Element,Set),                                 utility ).
known(     memberchk(Element,Set),                              utility ).
known(     mlmaplist(Pred,Lists),                               utility ).
  applies( mlmaplist(Pred,Lists), Pred+1 ).
known(     mlmaplist(Pred,Lists,Vin,Vout),                      utility ).
  applies( mlmaplist(Pred,Lists,Vin,Vout), Pred+3 ).
known(     mlmaplist(Pred,Lists,V),                             utility ).
  applies( mlmaplist(Pred,Lists,V), Pred+2 ).
known(     mlmember(Elements,Lists),                            utility ).
   wn(     mlselect(Elements,Lists,Rests),                      utility ).
known(     nextto(X,Y,List),                                    utility ).
known(     nmember(Element,Set,N),                              utility ).
known(     nobt(Goal),                                          utility ).
  applies( nobt(Goal), Goal ).
known(     not(Goal),                                           utility ).
  applies( not(Goal), Goal ).
known(     number(N),                                           utility ).
known(     numlist(N1,N2,Numberlist),                           utility ).
known(     occ(X,Term,N),                                       utility ).
known(     open(File),                                          utility ).
known(     open(Old,File),                                      utility ).
known(     pairfrom(List,A,B,Rest),                             utility ).
known(     perm(List1,List2),                                   utility ).
known(     perm2(X,Y,A,B),                                      utility ).
known(     portray_number(N),                                   utility ).
known(     prconj(Conj),                                        utility ).
known(     prexpr(Expr),                                        utility ).
known(     prlist(List),                                        utility ).
known(     read_in(Sentence),                                   utility ).
known(     redo(File),                                          utility ).
known(     remove_dups(List,Set),                               utility ).
known(     rev(List1,List2),                                    utility ).
known(     select(Element,List,Rest),                           utility ).
known(     seteq(Set1,Set2),                                    utility ).
known(     some(Pred,List),                                     utility ).
  applies( some(Pred,List), Pred+1 ).
known(     subgoal(exact,Goal),                                 utility ).
known(     sublist(Pred,List1,List2),                           utility ).
  applies( sublist(Pred,List1,List2), Pred+1 ).
known(     subset(Subset,Superset),                             utility ).
known(     subst(Substitution,Old,New),                         utility ).
known(     subtract(Set1,Set2,Subset),                          utility ).
known(     sumlist(NumList,Sum),                                utility ).
known(     thnot(Goal),                                         utility ).
  applies( thnot(Goal), Goal ).
known(     tidy(Expr,TidiedExpr),                               utility ).
known(     tlim(Tlimit),                                        utility ).
known(     ton(Name),                                           utility ).
known(     toff,                                                utility ).
known(     toff(Name),                                          utility ).
   wn(     trace(Format,Condition),                             utility ).
known(     trace(Format,List,Condition),                        utility ).
known(     ttyprint(X),                                         utility ).
known(     union(Set1,Set2,USet),                               utility ).
known(     variables(Term,VarSet),                              utility ).
known(     writef(Format),                                      utility ).
known(     writef(Format,List),                                 utility ).
```

WRITEF - FORMATTED WRITE UTILITY

writef(Format)

Formatted write. Equivalent to 'writef(Format,[])'.

writef(Format,List)

Formatted write. Format is an atom whose characters will be output. Format may contain certain special character sequences which specify certain formatting actions. The following sequences result in particular (otherwise hard to use) characters being output.

```
'\n'  --  <NL> is output
'\l'  --  <LF> is output
'\r'  --  <CR> is output
'\t'  --  <TAB> is output
'\\'  --  The character "\" is output
'\%'  --  The character "%" is output
'\nnn' -  where nnn is an integer (1-3 digits)
          the character with ASCII code nnn is output
          (NB : nnn is read as DECIMAL)
```

The next set of special sequences specify that items be taken from the List and output in some way. List, then, provides all the actual terms that are required to be output, while Format specifies where and how this is to occur.

```
'%t'  --  print the next item (mnemonic: term)
'%w'  --  write the next item
'%q'  --  writeq the next item
'%d'  --  display the next item
'%p'  --  print the next item (identical to %t)
'%l'  --  output the next item using prlist
'%c'  --  output the next item using prconj
'%e'  --  output the next item using prexpr
'%n'  --  put the next item as a character (ie it is
                 an iNteger)
'%r'  --  write the next item N times where N is the
                second item (an integer)
'%f'  --  perform a ttyflush (no items used)
```

The following examples may help to explain the use of writef.

```
writef('Hello there\n\n')
writef('%t is a %t\n',[Person,Property])
writef('Input : %l \nBecomes : %e\n \7',[In,Out])
```

```
@comment[ This is some long ago started, but as yet unfinished, documentation
          for UTIL. Bits of it may not be accurate (I havn't looked).
          enter is an unwritten SCRIBE macro (so it can't be formatted yet).

          Sorry about the delay
        ]

@make(manual)
@device(lpt)
@style(spacing 1)

This manual provides a guide to the Prolog utility procedures
available when running the UTIL interpreter.
The UTIL interpreter is an extension of the NEW Prolog interpreter,
and as such contains all the features described in the NEW documentation
(which should be read before starting this manual).

This document is divided up into sections which describe related
procedures.
The index provides an alphabetical list of all the procedures available
(and points you into the text for fuller descriptions).
T   appendix provides a complete list of ALL the operators defined in UTIL.
(   s includes those also defined in NEW).

@subheading(Input/Output routines)
@begin(defns)
@entry[error(Type,List,Action)@>(+,+,+)]
Report an error. If a clause of the form 'errmess(Type,Format)' can be found,
then the elements in List will written according to this Format
(by a call to 'writef(Format,List)', op cit).
If such an error message cannot be found then the default is to
write out Type followed by List.
In all cases the message starts with "** ERROR".
A line of the form '(Action after error)' in then written before a call
to Action is made.
(Usual Action's will be continue, fail, break, abort etc.)
The effect of a call to 'error' therefore depends on the Action specified.

@entry[tlim(Level)@>(+)]
3et the level of tracing.
T'  current tracing level is set to Level (an integer).
Th  will affect the printing of messages from 'trace' (see below).

@entry[trace(Format,Level)@>(+,+)]
Output tracing message.
Equivalent to 'trace(Format,[],Level)'.

@entry[trace(Format,List,Level)@>(+,+,+)]
Output tracing message.
If the current tracing level (see 'tlim') is greater than,
or equal to, Level (an integer) then a message is output by
a call to 'writef(Format,List)'.
Calls to trace can therefore be judiciously placed throughout your program,
their output being controlled by using 'tlim' to vary the amount of tracing
aterial actually output.

entry[trdep(Action,Level)@>(+,+)]
Perform a trace level dependent action.
If the current tracing level (see 'tlim') is greater than,
or equal to, Level (an integer) then a call to Action is made.
```

```
@entry[prconj(Conjunction)@>(+)]
Output a conjunction.
Conjunction (a complex term formed from a chain of (_ & _)),
is written one conjunct per line, with an indent of four
spaces on each line.
(Only one level is considered (ie C1 & C2 & Rest etc), and any final
atom 'true' is not printed.)

@entry[prexpr(Expression)@>(+)]
Output a logical expression.
Expression is a complex term constructed with the functors
(_ & _) (conjunction) and (_ # _) (disjunction).
It is written in such a way as to bring out the logical
structure of the Expression.
(Try an example).

@entry[prlist(List)@>(+)]
Output a list.
List is written one element per line, with an indent of
four spaces on each line.
( [illegible]y one level is considered).

@entry[read_in(Sentence)@>(-)]
Input a sentence.
read_in reads characters from the current input until the next
occurence of one of the characters {".","!","?"} followed by
a (line) terminator (or cit) is found.
This character sequence is parsed in a simple fashion to
produce a list of words - Sentence.
Sentence will contain a sequence of atoms and integers.
(Groups of letters are made into atoms, groups of digits
into integers. All other characters are individually
replaced by their corresponding atoms.
The last atom in Sentence will correspond to the final
character, ie it will be one of {'.','!','?'}.)

@entry[writef(Format)@>(+)]
Formatted write.
Equivalent to 'writef(Format,[])'.

@entry[writef(Format,List)@>(+,+)]
Formatted write.
Format is an atom whose characters will be output.
Format may contain certain special character sequences
which specify certain formatting actions.
The following sequences result in particular
(otherwise hard to use) characters being output.
@begin(verse)
'\n'  --  <NL> is output
'\l'  --  <LF> is output
'\r'  --  <CR> is output
'\t'  --  <TAB> is output
'\\'  --  The character "\" is output
'\%'  --  The character "%" is output
'\nnn' -  where nnn is an integer (1-3 digits)
          the character with ASCII code nnn is output
          (NB : nnn is read as DECIMAL)
@end(verse)
The next set of special sequences specify that items be taken
```

```
from the List and output in some way.
List, then, provides all the actual terms that are required to
be output, while Format specifies when and how this is to occur.
@begin(verse)
'%t'  --  write the next item (mnemonic: term)
'%w'  --  write the next item (identical to '%t')
'%q'  --  writeq the next item
'%d'  --  display the next item
'%p'  --  print the next item
'%l'  --  output the next item using prlist
'%c'  --  output the next item using prconj
'%e'  --  output the next item using prexpr
'%n'  --  put the next item as a character (ie it is an iNteger)
'%r'  --  write the next item N times where N is the second
           item (an integer)
'%f'  --  perform a ttyflush (no items used)
@end(verse)
The following examples may help to explain the use of writef.
@begin(verse)
writef('Hello there\n\n')
writef('%t is a %t\n',[Person,Property])
w[illegible]ef('Input : %l \nBecomes : %e\n \7',[In,Out])
@e[illegible]d(verse)
@end(defns)

@subheading(List routines)
@begin(defns)
@entry[append(List1,List2,List3)@>(?,?,?)]
List3 is the list formed by appending (concatenating)
List1 and List2.

@entry[disjoint(List)@>(+)]
List is disjoint (ie it contains no duplicates.
Tested by unification).

@entry[last(Element,List)@>(+,?)]
Element is the last element in List.

@entry[listtoset(List,Set)@>(+,?)]
Set is the set of elements of the list List.
( [illegible]Set is a list containing no duplicates.)

@entry[nextto(X,Y,List)@>(?,?,?)]
The elements X and Y are next to each-other
in the list List (with X to the left of Y).

@entry[numlist(N1,N2,List)@>(+,+,?)]
List is a list of integers from N1 to N2.

@entry[perm(List1,List2)@>(+,?)]
List2 is a permutation of List1 (ie a similar list with
the elements permuted).
If List2 is a variable then backtracking will vary it across
all possible permutations of List1.

@entry[perm2(A1,A2,X1,X2)@>(?,?,?,?)]
The pair X1-X2 is a permutation of the pair A1-A2.

@entry[remove_dups(List1,List2)@>(+,?)]
List2 is a list with the same elements as List1 except
```

```
that all duplicates have been removed
(identical to 'listtoset').

@entry[rev(List1,List2)@>(+,?)]
List2 is the list List1 with the elements in reverse order.

@entry[select(Element,List1,List2)@>(?,+,?)]
Element is a member of List1 and List2 is a list identical
to List1 except that Element has been removed.
@end(defns)

@subheading(Set routines)
@begin(defns)
The following routines are set-like operations which manipulate
ordinary Prolog lists.
@entry[intersect(Set1,Set2,Set3)@>(+,+,?)]
Set3 is the set intersection of Set1 and Set2.

@entry[member(Element,List)@>(?,?)]
Element is a member of List.
(Likely to be used on non-set lists!).
m[illegible]er is nondeterminate when Element is a variable.

@entry[memberchk(Element,List)@>(+,+)]
Element is a member of List.
Like member except determinate (ie it has a cut).
Use this version when just checking membership.

@entry[seteq(Set1,Set2)@>(+,+)]
The sets Set1 and Set2 are equivalent.

@entry[subset(Set1,Set2)@>(+,+)]
Set1 is a subset of Set2.

@entry[subtract(Set1,Set2,Set3)@>(+,+,?)]
Set3 is the set formed by subtracting Set2 from Set1.

@entry[union(Set1,Set2,Set3)@>(+,+,?)]
Set3 is the union of Set1 and Set2.
@end(defns)

@s[illegible]heading(Invocation routines)
These routines provide various odd ways of calling things.
Some of them are pretty hacky!
@begin(defns)
@entry[any(Goallist)@>(+)]
A member of Goallist is true
Ie Call the members of Goallsit until any one of them succeeds
(will keep going if backtracked into).

@entry[binding(N,Goal)@>(+,+)]
Return the N'th binding when Goal is called.
Ie Goal will end up instantiated as it would after
N redo's due to backtracking (or will fail if that
wouldn't occur).

@entry[findall(X,Goal,List)@>(?,+,-)]
List is a list of all X's such that Goal.
Ie List contains all the instantiations of X that result
from the various returns of Goal during backtracking.
```

```
(Beware the non-declarative nature of this routine!!).

@entry[for(N,Goal)@>(+,+)]
Call Goal N times (recursively).

@entry[forall(Goal1,Goal2)@>(+,+)]
For all Goal1, Goal2.
Ie For every time the Goal1 succeeds (during backtracking),
Goal2 must also succeed.
Models (x)(Goal1(x) -> Goal2(x)) in some sense.

@entry[foreach(Goal1,Goal2,Op,X,Answer)@>(+,+,+,?,-)]
For each Goal1, call Goal2 and collect up all the resultant
instantiations of X, using Op as a 2-ary functor to
form the resultant right-recursive tree (of Op's) - Answer.
This is a slightly souped up 'findall', except that two
goals are involved and there is the possibility of constructing
the result in other forms than a list.
If Op = '.' then Answer will be a list,
but any other atom can also be used
(eg If Op = '+' then Answer would be of the form A + B + C etc.)
Op can also have the form [Op2,Default],
in this case Op2 is the functor used for construction and Default
is returned if Goal1 never actually succeeded
(If Op = '.' then [] is the Default).
foreach will fail if Goal2 ever fails, or if
no Default is supplied when one is required.
You may well feel that this routine is slightly special purpose!
@end(defns)

@subheading(Application routines)

<to be continued...>
```

Old (needs finishing!)

```
@make(manual)
@device(lpt)
@style(spacing 1)

This manual provides a guide to the Prolog utility procedures
available when running the UTIL interpreter.
The UTIL interpreter is an extension of the NEW Prolog interpreter,
and as such contains all the features described in the NEW documentation
(which should be read before starting this manual).

This document is divided up into sections which describe related
procedures.
The index provides an alphabetical list of all the procedures available
(and points you into the text for fuller descriptions).
The appendix provides a complete list of ALL the operators defined in UTIL.
(This includes those also defined in NEW).

@subheading(Input/Output routines)
@begin(defns)
@entry[error(Type,List,Action)@>(+,+,+)]
Report an error. If a clause of the form 'errmess(Type,Format)' can be found,
then the elements in List will written according to this Format
(by a call to 'writef(Format,List)', op cit).
If such an error message cannot be found then the default is to
write out Type followed by List.
In all cases the message starts with "** ERROR".
A line of the form '(Action after error)' in then written before a call
to Action is made.
(Usual Action's will be continue, fail, break, abort etc.)
The effect of a call to 'error' therefore depends on the Action specified.

@entry[tlim(Level)@>(+)]
Set the level of tracing.
The current tracing level is set to Level (an integer).
This will affect the printing of messages from 'trace' (see below).

@entry[trace(Format,Level)@>(+,+)]
Output tracing message.
Equivalent to 'trace(Format,[],Level)'.

@entry[trace(Format,List,Level)@>(+,+,+)]
Output tracing message.
If the current tracing level (see 'tlim') is greater than,
or equal to, Level (an integer) then a message is output by
a call to 'writef(Format,List)'.
Calls to trace can therefore be judiciously placed throughout your program,
their output being controlled by using 'tlim' to vary the amount of tracing
material actually output.

@entry[trdep(Action,Level)@>(+,+)]
Perform a trace level dependent action.
If the current tracing level (see 'tlim') is greater than,
or equal to, Level (an integer) then a call to Action is made.

@entry[prconj(Conjunction)@>(+)]
Output a conjunction.
Conjunction (a complex term formed from a chain of (_ & _)),
is written one conjunct per line, with an indent of four
spaces on each line.
(Only one level is considered (ie C1 & C2 & Rest etc), and any final
atom 'true' is not printed.)

@entry[prexpr(Expression)@>(+)]
Output a logical expression.
Expression is a complex term constructed with the functors
(_ & _) (conjunction) and (_ # _) (disjunction).
It is written in such a way as to bring out the logical
structure of the Expression.
(Try an example).

@entry[prlist(List)@>(+)]
Output a list.
List is written one element per line, with an indent of
four spaces on each line.
(Only one level is considered).

@entry[read_in(Sentence)@>(-)]
Input a sentence.
read_in reads characters from the current input until the next
occurence of one of the characters {".","!","?"} followed by
a (line) terminator (or cit) is found.
This character sequence is parsed in a simple fashion to
produce a list of words - Sentence.
Sentence will contain a sequence of atoms and integers.
(Groups of letters are made into atoms, groups of digits
into integers. All other characters are individually
replaced by their corresponding atoms.
The last atom in Sentence will correspond to the final
character, ie it will be one of {'.','!','?'}.)

@entry[writef(Format)@>(+)]
Formatted write.
Equivalent to 'writef(Format,[])'.

@entry[writef(Format,List)@>(+,+)]
Formatted write.
Format is an atom whose characters will be output.
Format may contain certain special character sequences
which specify certain formatting actions.
The following sequences result in particular
(otherwise hard to use) characters being output.
@begin(verse)
'\n'  --  <NL> is output
'\l'  --  <LF> is output
'\r'  --  <CR> is output
'\t'  --  <TAB> is output
'\\'  --  The character "\" is output
'\%'  --  The character "%" is output
'\nnn' -  where nnn is an integer (1-3 digits)
          the character with ASCII code nnn is output
          (NB : nnn is read as DECIMAL)
@end(verse)
The next set of special sequences specify that items be taken
from the List and output in some way.
List, then, provides all the actual terms that are required to
be output, while Format specifies when and how this is to occur.
@begin(verse)
'%t'  --  write the next item (mnemonic: term)
'%w'  --  write the next item (identical to '%t')
'%q'  --  writeq the next item
```

```
'%d'  --  display the next item
'%p'  --  print the next item
'%l'  --  output the next item using prlist
'%c'  --  output the next item using prconj
'%e'  --  output the next item using prexpr
'%n'  --  put the next item as a character (ie it is an iNteger)
'%r'  --  write the next item N times where N is the second
           item (an integer)
'%f'  --  perform a ttyflush (no items used)
@end(verse)
The following examples may help to explain the use of writef.
@begin(verse)
writef('Hello there\n\n')
writef('%t is a %t\n',[Person,Property])
writef('Input : %l \nBecomes : %e\n \7',[In,Out])
@end(verse)
@end(defns)

@subheading(List routines)
@begin(defns)
@entry[append(List1,List2,List3)@>(?,?,?)]
List3 is the list formed by appending (concatenating)
List1 and List2.

@entry[disjoint(List)@>(+)]
List is disjoint (ie it contains no duplicates.
Tested by unification).

@entry[last(Element,List)@>(+,?)]
Element is the last element in List.

@entry[listtoset(List,Set)@>(+,?)]
Set is the set of elements of the list List.
(ie Set is a list containing no duplicates.)

@entry[nextto(X,Y,List)@>(?,?,?)]
The elements X and Y are next to each-other
in the list List (with X to the left of Y).

@entry[numlist(N1,N2,List)@>(+,+,?)]
List is a list of integers from N1 to N2.

@entry[perm(List1,List2)@>(+,?)]
List2 is a permutation of List1 (ie a similar list with
the elements permuted).
If List2 is a variable then backtracking will vary it across
all possible permutations of List1.

@entry[perm2(A1,A2,X1,X2)@>(?,?,?,?)]
The pair X1-X2 is a permutation of the pair A1-A2.

@entry[remove_dups(List1,List2)@>(+,?)]
List2 is a list with the same elements as List1 except
that all duplicates have been removed
(identical to 'listtoset').

@entry[rev(List1,List2)@>(+,?)]
List2 is the list List1 with the elements in reverse order.

@entry[select(Element,List1,List2)@>(?,+,?)]
```

```
Element is a member of List1 and List2 is a list identical
to List1 except that Element has been removed.
@end(defns)

@subheading(Set routines)
@begin(defns)
The following routines are set-like operations which manipulate
ordinary Prolog lists.
@entry[intersect(Set1,Set2,Set3)@>(+,+,?)]
Set3 is the set intersection of Set1 and Set2.

@entry[member(Element,List)@>(?,?)]
Element is a member of List.
(Likely to be used on non-set lists!).
member is nondeterminate when Element is a variable.

@entry[memberchk(Element,List)@>(+,+)]
Element is a member of List.
Like member except determinate (ie it has a cut).
Use this version when just checking membership.

@entry[seteq(Set1,Set2)@>(+,+)]
The sets Set1 and Set2 are equivalent.

@entry[subset(Set1,Set2)@>(+,+)]
Set1 is a subset of Set2.

@entry[subtract(Set1,Set2,Set3)@>(+,+,?)]
Set3 is the set formed by subtracting Set2 from Set1.

@entry[union(Set1,Set2,Set3)@>(+,+,?)]
Set3 is the union of Set1 and Set2.
@end(defns)

@subheading(Invocation routines)
These routines provide various odd ways of calling things.
Some of them are pretty hacky!
@begin(defns)
@entry[any(Goallist)@>(+)]
A member of Goallist is true
Ie Call the members of Goallsit until any one of them succeeds
(will keep going if backtracked into).

@entry[binding(N,Goal)@>(+,+)]
Return the N'th binding when Goal is called.
Ie Goal will end up instantiated as it would after
N redo's due to backtracking (or will fail if that
wouldn't occur).

@entry[findall(X,Goal,List)@>(?,+,-)]
List is a list of all X's such that Goal.
Ie List contains all the instantiations of X that result
from the various returns of Goal during backtracking.
(Beware the non-declarative nature of this routine!!).

@entry[for(N,Goal)@>(+,+)]
Call Goal N times (recursively).

@entry[forall(Goal1,Goal2)@>(+,+)]
For all Goal1, Goal2.
```

Ie For every time the Goal1 succeeds (during backtracking),
Goal2 must also succeed.
Models (x)(Goal1(x) -> Goal2(x)) in some sense.

@entry[foreach(Goal1,Goal2,Op,X,Answer)@>(+,+,+,?,-)]
For each Goal1, call Goal2 and collect up all the resultant
instantiations of X, using Op as a 2-ary functor to
form the resultant right-recursive tree (of Op's) - Answer.
This is a slightly souped up 'findall', except that two
goals are involved and there is the possibility of constructing
the result in other forms than a list.
If Op = '.' then Answer will be a list,
but any other atom can also be used
(eg If Op = '+' then Answer would be of the form A + B + C etc.)
Op can also have the form [Op2,Default],
in this case Op2 is the functor used for construction and Default
is returned if Goal1 never actually succeeded
(If Op = '.' then [] is the Default).
foreach will fail if Goal2 ever fails, or if
no Default is supplied when one is required.
You may well feel that this routine is slightly special purpose!
@end(defns)

@subheading(Application routines)

```
/* EDIT.PL : Get to an editor and back again

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 1 June 81
*/

        %%% Run this module interpreted
        %%% EDIT requires no other modules


% This relies on the new evaluable predicates which arrived in
% recent Prolog versions. It has been updated to the latest
% (3.31 onwards) versions using save/2, tmpcor/3, run/2.
%
% edit currently runs the FINE editor and returns by running the
% version of Prolog in mec:
%


                            % Redo a file by editing it and reconsulting it

redo(File)
     :- edit(File),
        reconsult(File),
        !.



                            % Edit a file
                            %  Build the FINE CCL string
                            %  Save state into prolog.bin
                            %  Run FINE
                            %  On return - delete prolog.bin
                            % Note that the command string to Fine is very
                            %  delicate! All the newlines, "!"'s etc are
                            %  important.

edit(File)
     :- name(File,Chars),
        edit_chars(Chars,0,Command,"
mec:prolog!"),
        ( save('prolog.bin',1)
                ;  tmpcor(tell,edt,[32|Command]),
                   run('sys:fine',1)
        ),
        !,
        see('prolog.bin'),
        rename('prolog.bin',[]).


                            % Add a dot to the filename if not already there
                            %  and append on the rest of the CCL string

edit_chars([C|Cs],K0,[C|R],T)
     :- edit_dot(K0,C,K),
        edit_chars(Cs,K,R,T).

edit_chars([],0,[46|T],T) :- !.
edit_chars([],1,T,T).

edit_dot(1,_,1).                  % Already found
edit_dot(0,46,1) :- !.            % Found the dot
edit_dot(0,_,0).                  % Not found yet
```

```
/* FILES.PL : Routines for playing with files

                                                     UTILITY
                                                     Lawrence
                                                     Updated: 2 April 81
*/

        %%%  Compile this module
        %%%  FILES requires no other modules


  :- public check_exists/1,
            file_exists/1,
            open/1,
            open/2,
            close/2,
            delete/1.


  :- mode check_exists(+),
          file_exists(+),
          open(+),
          open(?,+),
          close(+,+),
          delete(+).



                        % Check to see if a file exists and provide
                        %  an error message if it doesn't

check_exists(File)
     :- file_exists(File),
        !.

check_exists(File)
     :- ttynl, display('! File: '), display(File),
        display(' does not exist.'), ttynl,
        fail.


                        % Succeed if a file exists, otherwise fail

file_exists(File)
     :- atom(File),
        seeing(Old),
        ( nofileerrors  ;  fileerrors, fail ),
        see(File),
        fileerrors,
        seen,
        see(Old),
        !.


                        % Open a file, checking that it exists

open(File)
```

```
      :- check_exists(File),
         see(File).


                            % Open a file and return current file

open(Old,File)
      :- seeing(Old),
         open(File).


                            % Close file and see old file again

close(File,Old)
      :- close(File),
         see(Old).


                            % Delete a file (note that rename requires that
                            %  the file be open)

delete(File)
      :- open(Old,File),
         rename(File,[]),
         see(Old).
```

```
/* WRITEF.PL : Formatted write routine (and support)

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 11 May 81
*/

        %%% Compile this module
        %%% WRITEF requires no other modules

% FIXES
%
%  (11 May 81)
%
%       Split the (now obsolete) module IOROUT into two: WRITEF (this one)
%       and TRACE.
%       Added cuts to writefs to make it determinate (it's tail recursive).



  /* EXPORT */

  :- public ttyprint/1,
            prlist/1,
            prconj/1,
            prexpr/1,
            writef/1,
            writef/2.


  /* MODES */

  :- mode       ttyprint(?),
                prlist(?),
                prconj(?),
                prexpr(?),
                        prlos(?,+,-,?,?),
                        doexpr(+,?,?,+,-,?,?),
                        prels(+,+),
                writef(+),
                writef(+,+),
                        nobtwritefs(+,+),
                        writefs(+,+),
                        action(+,+,-),
                        special(+,-),
                        setcode(+,-,-),
                        setdigits(+,+,-,-),
                        writelots(?,+).




                        % Print (therefore use pretty printing) onto
                        %  the terminal

ttyprint(X)
     :- seeing(Old),
        see(user),
        print(X),
        see(Old).
```

```
                           % Print a list, one element per line

  prlist([]) :- !.

  prlist([HD|TL])
        :- tab(4), print(HD), nl,
           prlist(TL).



                           % Print a conjunction, one element per line

prconj(true) :- !.

prconj(A&B)
     :- !,
        tab(4), print(A), nl,
        prconj(B).

prconj(A)
     :- tab(4), print(A), nl.



                           % Pretty print a simple logical expression
                           %  This is done by first printing the logical
                           %  structure using X1 X2 etc to name the components
                           %  and then printing the 'values' of X1 X2 etc on
                           %  separate lines.

prexpr(Expr)
     :- prlog(Expr,1,N,Elements,[]),
        nl, write('  where :'), nl,
        prels(Elements,1).


prlog(A & B,Nin,Nout,Elements,Z)
     :- !,
        doexpr(38,A,B,Nin,Nout,Elements,Z).     % Ascii 38 = "&"

prlog(A # B,Nin,Nout,Elements,Z)
     :- !,
        doexpr(35,A,B,Nin,Nout,Elements,Z).     % Ascii 35 = "#"

prlog(X,Nin,Nout,[X|Z],Z)
     :- put("X"),
        write(Nin),
        Nout is Nin+1.


doexpr(Conn,A,B,Nin,Nout,Elements,Z)
     :- put("("), put(" "),
        prlog(A,Nin,Ninter,Elements,Rest),
        put(" "), put(Conn), put(" "),
        prlog(B,Ninter,Nout,Rest,Z),
        put(" "), put(")").



prels([],_).

prels([First|Rest],N)
     :- write('    X'), write(N), write(' =  '), print(First), nl,
        N2 is N+1,
        prels(Rest,N2).



                       % Formatted write utility
                       %  This converts the format atom to a string and
                       %  uses writefs on that. Note that it fails back over
                       %  itself to recover all used space.

writef(Format) :- writef(Format,[]).


writef(Format,List)
     :- name(Format,Fstring),
        writefs(Fstring,List),
        fail.

writef(_,_).



                       % Formatted write for a string (ie a list of
                       %  character codes).

writefs([],X).

writefs([37,X|Rest],List)                    /*   "%"   */
     :- action(X,List,List2),
        !,
        writefs(Rest,List2).

writefs([92,X|Rest],L)               /*   "\" special   */
     :- special(X,Char),
        !,
        put(Char),
        writefs(Rest,L).

writefs([92|Rest],L)                         /*   "\" number    */
     :- getcode(Rest,Rest2,Char),
        !,
        put(Char),
        writefs(Rest2,L).

writefs([Char|Rest],L)               /*   character     */
     :- put(Char),
        writefs(Rest,L).



action(116,[HD|TL],TL)         /*  t  */
     :- print(HD).
```

```
action(100,[HD|TL],TL)              /*  d  */
      :- display(HD).

action(119,[HD|TL],TL)              /*  w  */
      :- write(HD).

action(113,[HD|TL],TL)              /*  q  */
      :- writeq(HD).

action(112,[HD|TL],TL)              /*  p  */
      :- print(HD).

action(108,[HD|TL],TL)              /*  l  */
      :- nl, prlist(HD).

action(99,[HD|TL],TL)               /*  c  */
      :- nl, prconj(HD).

action(101,[HD|TL],TL)              /*  e  */
      :- nl, prexpr(HD).

action(102,L,L)                     /*  f  */
      :- ttyflush.

action(110,[HD|TL],TL)              /*  n  */
      :- put(HD).

action(114,[T,N|TL],TL) /*  r  */
      :- writelots(N,T).



special(110,31).                    /*  n  */
special(108,10).                    /*  l  */
special(114,13).                    /*  r  */
special(116,9).                     /*  t  */
special(92,92).                     /*  \  */
special(37,37).                     /*  %  */


setcode(List,Rest,Char)
      :- setdigits(1,List,Rest,Digits),
         name(Char,Digits),
         Char < 128.


setdigits(N,[HD|TL1],Rest,[HD|TL2])
      :- N =< 3,
         HD >= "0",
         HD =< "9",
         !,
         N2 is N+1,
         setdigits(N2,TL1,Rest,TL2).

setdigits(_,Rest,Rest,[]).
```

```
writelots(0,_) :- !.

writelots(N,T)
      :- N > 0,
         N2 is N-1,
         write(T),
         writelots(N2,T).
```

```
/* TRACE.PL : Tracing routines

                                                        UTILITY
                                                        Lawrence
                                                        Updated: 11 May 81
*/

        %%% Compile this module
        %%% TRACE requires modules:  WRITEF, FLAG

% FIXES
%
%  (11 May 81)
%
%       Split the (now obsolete) module IOROUT into two: WRITEF and
%       TRACE (this one).
%


  /* EXPORT */

  :- public     error/3,
                tlim/1,
                ton/1,
                toff/1,
                toff/0,
                trace/2,
                trace/3.


  /* MODES */

  :- mode       error(+,+,+),
                tlim(?),
                ton(?),
                toff(?),
                toff,
                trace(+,+),
                trace(+,+,+).



                        % Error message handler
                        %  Prints a (writef style) message and then performs
                        %  the specified action.

error(Format,List,Action)
     :- nl, write('** ERROR '),
        writef(Format,List),
        writef('\n    ( %t after error )\n',[Action]),
        Action.


                        % Set tracing level for level conditional tracing

tlim(N)
     :- flag(tflag,Old,N),
        ttynl, display('Tracing level reset from '), display(Old),
          display(' to '), display(N), ttynl.


                              % Set/unset various name conditional trace messages
                              %  The Name "all" is treated specially by trace/3 to
                              %  effectively switch on ALL named tracing messages.

ton(Name)
     :- tracing(Name),
        !,
        display('You are already tracing '), display(Name), ttynl.

ton(Name)
     :- asserta( tracing(Name) ),
        display('Now tracing '), display(Name), ttynl.



toff(Name)
     :- retract( tracing(Name) ),
        !,
        display('No longer tracing '), display(Name), ttynl.

toff(Name)
     :- display('You were not tracing '), display(Name), ttynl.



toff :- abolish(tracing,1),
        display('All named tracing switched off'), ttynl.



                              % Print out a trace message
                              %  There are two styles of trace message;
                              %  Those conditional on a specific name and those
                              %  conditional on a numeric tracing level.
                              %   Name conditional trace message are switched
                              %   on and of using ton(_) and toff(_)
                              %   Number conditional trace messages are dependent
                              %   on the tlim(_) flag which specifies the current
                              %   level of tracing.

trace(Format,N) :- trace(Format,[],N).


trace(Format,List,Name)
     :- atom(Name),
        ( tracing(Name)  ;  tracing(all) ),
        !,
        writef(Format,List).

trace(Format,List,N)
     :- integer(N),
        flag(tflag,M,M),
        N =< M,
        !,
        writef(Format,List).
```

```
trace(_,_,_).
```

```
/* READIN.PL : Read in a sentence into a list of words

                                                UTILITY
                                                Lawrence
                                                Updated: 30 march 81

*/


        %%% Compile this module
        %%% READIN requires no other modules



  /* EXPORT */

  :- public read_in/1.



  /* MODES */

        :- mode read_in(?).
        :- mode initread(-).
        :- mode readrest(+,-).
        :- mode word(-,?,?).
        :- mode words(-,?,?).
        :- mode alphanum(+,-).
        :- mode alphanums(-,?,?).
        :- mode digits(-,?,?).
        :- mode digit(+).
        :- mode lc(+,-).



read_in(P):-initread(L),words(P,L,[]),!.

initread([K1,K2|U]):-get(K1),get0(K2),readrest(K2,U).

readrest(46,L) :- !, possiblythere(46,L).
readrest(63,L) :- !, possiblythere(63,L).
readrest(33,L) :- !, possiblythere(33,L).
readrest(K,[K1|U]):-K=<32,!,get(K1),readrest(K1,U).
readrest(K1,[K2|U]):-get0(K2),readrest(K2,U).


  possiblythere(C,Rest)
        :- repeat,
           get0(Next),
           Next =\= 32,
           ( Next =:= 31,
             !,
             Rest = []           ;  Rest = [Next|More],
                                    readrest(Next,More)
           ).


words([V|U]) --> word(V),!,blanks,words(U).
words([]) --> [].

word(U1) --> [K],{lc(K,K1)},!,alphanums(U2),{name(U1,[K1|U2])}.
word(N) --> [K],{digit(K)},!,digits(U),{name(N,[K|U])}.
word(V) --> [K],{name(V,[K])}.

alphanums([K1|U]) --> [K],{alphanum(K,K1)},!,alphanums(U).
alphanums([]) --> [].

alphanum(K,K1):-lc(K,K1).
alphanum(K,K):-digit(K).

digits([K|U]) --> [K],{digit(K)},!,digits(U).
digits([]) --> [].

blanks--> [K],{K=<32},!,blanks.
blanks --> [].

digit(K):-K>47,K<58.

lc(K,K1):-K>64,K<91,!,K1 is K\/8'40.
lc(K,K):-K>96,K<123.
```

```
/* LISTRO.PL : List manipulating routines

                                                  UTILITY
                                                  Lawrence
                                                  Updated: 31 March 81
*/

        %%%  Compile this module
        %%%  LISTRO requires module:  SETROU



  /* EXPORT */

  :- public append/3,
            disjoint/1,
            last/2,
            listtoset/2,
            nextto/3,
            numlist/3,
            perm/2,
            perm2/4,
            remove_dups/2,
            rev/2,
            select/3,
            sumlist/2,
            pairfrom/4.



  /* MODES */

        :- mode append(?,?,?).
        :- mode disjoint(?).
        :- mode last(?,?).
        :- mode listtoset(?,?).
        :- mode nextto(?,?,?).
        :- mode numlist(+,+,?).
        :- mode perm(?,?).
        :- mode perm2(?,?,?,?).
        :- mode remove_dups(?,?).
        :- mode rev(?,?).
        :- mode         revconc(?,+,?).
        :- mode select(?,?,?).
        :- mode sumlist(+,?).
        :- mode pairfrom(+,?,?,?).



  append([],L,L).

  append([HD|TL],L,[HD|LL]) :- append(TL,L,LL).


  disjoint([]).

  disjoint([HD|TL])
        :- memberchk(HD,TL), !, fail   ;
           disjoint(TL).
```

```
last(X,[X]) :- !.

last(X,[HD|TL]) :- last(X,TL).



listtoset([],[]).

listtoset([HD|TL],Ans)
       :- member(HD,TL),
          !,
          listtoset(TL,Ans).

listtoset([HD|TL],[HD|Ans]) :- listtoset(TL,Ans).



nextto(X,Y,[X,Y,..Rest]).

nextto(X,Y,[HD|TL]) :- nextto(X,Y,TL).



numlist(N,N,[N]) :- !.

numlist(N1,N2,[N1|Rest])
       :- N1 < N2,
          N3 is N1+1,
          numlist(N3,N2,Rest).



perm([],[]).

perm(L,[X|Xs])
       :- select(X,L,R),
          perm(R,Xs).


perm2(X,Y,X,Y).

perm2(X,Y,Y,X).



remove_dups(A,B) :- listtoset(A,B).



rev(L1,L2) :- revconc(L1,[],L2).


revconc([],L,L).

revconc([X|L1],L2,L3) :- revconc(L1,[X|L2],L3).
```

```
  select(X,[X|TL],TL).

  select(X,[Y|TL1],[Y|TL2]) :- select(X,TL1,TL2).

                              % Sum a list of integers

sumlist([],0) :- !.

sumlist([Hd|Tl],Sum) :-
        sumlist(Tl,TlSum), Sum is Hd+TlSum, !.


                              % Get a pair of elements from a list, also
                              %  return the rest. Pairs are only returned
                              %  once (not twice different ways round)

pairfrom([X|T],X,Y,R) :- select(Y,T,R).

pairfrom([H|S],X,Y,[H|T]) :- pairfrom(S,X,Y,T).
```

```
/* SETROU.PL : Set manipulating routines

                                                   UTILITY
                                                   Lawrence
                                                   Updated: 31 March 81
*/

        %%%  Compile this module
        %%%  SETROU requires no other modules


 /* EXPORT */

  :- public intersect/3,
            member/2,
            memberchk/2,
            nmember/3,
            seteq/2,
            subset/2,
            subtract/3,
            union/3.


  /* MODES */

        :- mode intersect(?,?,?).
        :- mode member(?,?).
        :- mode memberchk(?,?).
        :- mode nmember(?,+,?).
        :- mode seteq(?,?).
        :- mode subset(?,?).
        :- mode subtract(?,?,?).
        :- mode union(?,?,?).



  intersect([],Set,[]).

  intersect([HD|TL],Set,[HD|Ans])
        :- member(HD,Set),
           !,
           intersect(TL,Set,Ans).

  intersect([HD|TL],Set,Ans) :- intersect(TL,Set,Ans).


  member(X,[X|TL]).

  member(X,[Y|TL]) :- member(X,TL).


  memberchk(X,[X|TL]) :- !.

  memberchk(X,[Y|TL]) :- memberchk(X,TL).
```

```
                              % X is the N'th member of List

nmember(X,[X|_],1).

nmember(X,[_|L],N)
     :- nmember(X,L,M),
        N is M+1.



  seteq(S1,S2) :- subset(S1,S2), subset(S2,S1).



  subset([],Ys).

  subset([X|Xs],Ys)
        :- memberchk(X,Ys),
           subset(Xs,Ys).



  subtract([],Ys,[]).

  subtract([X|Xs],Ys,Zs)
        :- member(X,Ys),
           !,
           subtract(Xs,Ys,Zs).

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



  union([],Ys,Ys).

  union([X|Xs],Ys,Zs)
        :- member(X,Ys),
           !,
           union(Xs,Ys,Zs).

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

```
%   well(Functor, Arity)  asserts that Functor/Arity is no longer to be
%   checked.  So it changes the stub to a direct call, e.g.
%        dick(A,B) :- med$dick(A,B).
%   This is easily changed back by "sick".

well(Functor, Arity) :-
        atom(Functor), integer(Arity), Arity >= 0,
        'med$mode'(Functor, Arity, NewFunc, Template), !,
        abolish(Functor, Arity),   %  remove old stub
        genterms(Functor, NewFunc, Arity, [], Head, Call),
        assert((  Head :- Call  )),
        !.
well(Functor, Arity) :-
        write('! MEDIC hasn''t been consulted about '),
        write(Functor/Arity), nl,
        !.

%   genterms(F1, F2, N, [], T1, T2)
%   binds T1 to F1(A,...,Z) and T2 to F2(A,...,Z).

genterms(F1, F2, 0, Args, T1, T2) :- !,
        T1 =.. [F1|Args],
        T2 =.. [F2|Args].
genterms(F1, F2, N, Args, T1, T2) :-
        M is N-1, !,
        genterms(F1, F2, M, [Arg|Args], T1, T2).

%   modes(Modes) is given a comma-list of mode-declarations, which I call
%   "templates" here.  For each template, it checks that the new template
%   doesn't conflict with a previous template for the same procedure.  In
%   any case it creates an entry in the table med$mode and then says that
%   the procedure is "sick".  E.g. given :- mode dick(+,-) it stores
%        med$mode(dick, 2, med$dick, dick(+,-)).
%   and creates the stub
%        dick(A, B) :- med$check(dick(+,-), med$dick(A,B)).
%   MEDIC is free to create any new name in place of med$dick; only this
%   section of the package knows what that name is.  And only "sick/well"
%   know how the run-time checking is done.

modes(','(A,B)) :- !,
        modes(A),
        modes(B).
modes(Template) :-
        functor(Template, Functor, Arity),
        (   retract('med$mode'(Functor, Arity, NewFunc, OldTemp)),
                compare(OldTemp, Template)
        ;   genname(Functor, NewFunc)
        ),
        assert('med$mode'(Functor, Arity, NewFunc, Template)), !,
        sick(Functor, Arity).

%   compare(Old_template, New_template) checks that the new description
%   doesn't conflict with the old.  At the moment this is a simple = test,
%  but some more complex test might be justifiable.  Might.

compare(Same, Same).
compare(Old,  New ) :-
        write('! New mode declaration '), write(New),
        write(' conflicts with '), write(Old), nl,
        write('  New declaration accepted.'), nl.

genname(OldAtom, NewAtom) :-
        name(OldAtom, OldName),
        append("med$", OldName, NewName),
        name(NewAtom, NewName).

        append([Head|Tail], More, [Head|Rest]) :- !,
                append(Tail, More, Rest).
        append([], More, More).

%   others handles miscellaneous things like "op", "reconsult".
%   perhaps other commands should be obeyed too?

others(','(A,B)) :- !,
        others(A),  !,
        others(B).
others(op(A,B,C)) :- !,
        op(A,B,C).
others(reconsult(A)) :- !,
        medic(A).
o[illegible]rs([-A]) :- !,
        medic(A).
others(X) :-
        true.   %   is this right?
```

```
/* TIDY.PL : Lawrence's so fast, hacked up, version of Tidy

                                                  Lawrence
                                                  Updated: 2 August 81

          This code implements a simple tidy/evaluator which tries to
    maximise evaluation. It is supposed to be efficient so that there
    is not much expense involved in using it as a first step to something
    else. It traverses the input term in a single pass and does not
    build any significant intermediate structures. It performs two
    kinds of function:

     1) Simple normalisation

          All occurences of the operators / and - are removed in
          favour of * and +.

          Bags are formed for * and +. These bags are left recursive
          trees built with the functors *(_,_) and +(_,_). All numbers
          in the bag are evaluated and the result, if any,  will always
          be the top rightmost element of the bag. This will not be present
          if it would be the unit element of the bag. Nested exponentiations
          (of the form ((A^B)^C)^D etc) are collapsed into their base
          to the power of a multiplication bag (ie A^(B*C*D)).

           egs:   * bags:

                         *               *
                        / \             / \
                       *   c           *   19
                      / \             / \
                     a   b           *   c
                                    / \
                                   a   b

          (This form gives the advantages of bags, ie traversing, selecting
           elements is easy and you know when you've finished, while leaving
           the expression as a standard algebraic term. However this may not
           be optimal for all purposes)

     2) Evaluation

          Numerical evaluation is performed where possible. Advantage is
          taken of the associative properties of * and + (as the bags are
          formed). Some simplifying rules are applied concerning zero and
          unit elements of these bags. A small amount of distribution of
          one operator over another is done where this will aid evaluation.

    The total set of functions that tidy actually tidies are as follows:

          *(_,_) /(_,_) +(_,_) -(_,_) -(_) ^(_,_) &(_,_) #(_,_)

    Expressions involving other functions are handed to eval if all their
    arguments have been evaluated (to numbers). This may result in these
    simple expressions being reduced to numbers (which may then be involved
    in further evaluations).

    NB  Tidy should be used as the expression evaluator instead of eval
        (from LONG.PL) whenever the expressions are (or may be) partly
        symbolic. It will call eval for all the numeric subexpressions
```

(using its normalisation tricks to try and maximise the the amount of numeric evaluation). Over purely numeric expressions tidy will be equivalent to eval.

** TODO

There is undoubtably scope for improvement. I can think of:

a) Add knowledge of some more simple functions (NB Some odd bits have been added, see special_fun/2). The old tidy and normalise had some simplifications for equalities and inequalities as well.

b) Improve the evaluation. Only self generated numbers (such as *-1, ^-1) are distributed at the moment. tidy does not handle such things already occuring in the input. Eg (a*b^-1)^-1 is left as it is. But it is arguable how much of this should go on. (Try to produce bags with the minimum number of recipricals/negatives?)

c) Various people seem to want sin/arcsin (etc) cancelling added for good measure.

** SIGNIFICANT BUG

There is a logical bug in the current code. I have assumed that when bag sweepers bottom out, the tidied version of the bit at the bottom will not be the sort of thing that could have been incorporated into the bag, otherwise we would have spotted it and kept going. However, this is not true! For example: a*b*((c*d)+0)
When the c*d comes back it is a mulbag and should be merged with the upper a*b mulbag. The current code doesn't do this because it I didn't realise that the lower bag could be hidden as shown in the example. When I find time I intend to rewrite the affected bits so that a proper bag merge is defined and applied. The interesting bit is how to do this without rebuilding one of the bags completely - ie by using partial structures a la difference lists. But I am not sure how much I care about such efficiency/complexity hacks any more. Trying to be clever in the current code taught me what an effort it is and how it makes things much more complicated. What I really need is a practical program transformation system to Unfold/Fold a multipass specification into a single pass hack (Cf Burstall & Darlington, Feather etc). Please tell me when you have such a thing ready for use.

** RECENT FIXES

(4 March) Code for & and # added by Leon. This code leaves the structure of these functions as it was (ie no bagging is done), however simplifications are applied. Note that this includes some identical element merging (but this does not use associativity).

(10 March) Put some cuts in the code for & and #. Also renamed the predicates and flattened out the structures into separate arguments (These little things!).

```
                        Fixed problem stemming from assumption that arithmetic
                        always succeed. This was not true (power sometimes fails)
                        and resulted in tidy failing when an arithmetic operation
                        failed. The fix involved moving the cuts in the ...build
                        and ...fin routines which call arithmetic routines. No
                        assumption is now made about their success, even for
                        add and multiply which should always succeed.

                        Reordered the file a little and added some more
                        documentation.

          (18 March)    Added tidy clauses for log.
                        Also normalize clause for sort      (Leon)

          (1 April)     When given nested variables tidy produces mode errors
                        as these are not expected. Fixed this by adding checks
                        to appropriate places and a top level error message
                        if tidy fails.

          (2 August)    Added the BUG note above and a couple of other comments.


/* EXPORT */

  :- public      tidy/2,
                 simple/1.


/* IMPORT */

  % This version designed for use with rational package (LONG)
  %  In particular it uses:
  %                              number/1
  %                              eval/2
  %                              add/3
  %                              multiply/3
  %                              power/3


/* MODES */

  :- mode        tidy(+,?),
                 dotidy(+,-),
                 simple(?),
                 tidyall(+,+,+,+,-),
                 chknum(+,+,-),
                 try_eval(+,+,-),
                 special_fun(+,-),

                 mulbag(+),
                 plusbag(+),
                 expbag(+),
                 multidy(+,+,+,+,-,-),
                 m2tidy(+,+,-),
                 mulbuild(+,+,+,-,-),
                 plustidy(+,+,+,+,-,-),
                 p2tidy(+,+,-),
plusbuild(+,+,+,-,-),
exptidy(+,+,+,-,-,-),
distr_inverse(+,-),

andfin(+,+,-),
orfin(+,+,-),
mulfin(+,+,-),
plusfin(+,+,-),
expfin(+,+,+,-),
n_expfin(+,+,-),
tidy_varerr.
```

/* Implementation - some hints.

The top level is pretty straight forward, note that the invari$\overset{a}{e}$nt that all solution arguments should be initially uninstantiated is guaranteed by unifying the tidied expression with the output variable right at the end of the tidy operation (tidy/2). This is to avoid any bugs with output arg unification failures causing clauses with cuts to be missed. [You should know what this means - if not then think about it. Eg ?- foo(a,b). foo(a,c) :- !. foo(_,b).]

The bag sweeping routines sweep across the term (left to right for multiplication and division bags but right to left down exponentiation chains) with a pair of accumulators being passed across. Thus for multidy, Left and Ltag are the two incoming accumulators and Right and Rtag are the resultant accumulators after this bit of the tree has been looked at. (For exptidy the names are the other way round). One accumulator is the bag of symbolic structures (eg Left), the other is the bag of numeric structures (eg Ltag). The numeric bag will always be just a number since the arithmetic operations are done immediately whenever a number is found. The symbolic bag may be empty, which is represented by the Prolog atom 'empty'. Simultaneously with this accumulation there is a process of pushing down a distibuted term (Distr). This is one of {1,-1}. For multidy this is the power that each final element should be raised to, and for plustidy this is the multiplier that each final element should be multiplied by. This value is flipped back and forth as the top down descent passes through divisions (multidy) and subtractions (plustidy).

The bag sweepers bottom out when they hit the top of an expression which they won't be able to incorporate. m2tidy and p2tidy see that this expression gets (recursively) tidied, and then they incorporate this with the distributed term (simplifying this away when it is 1). There is a special case here when the expression bottomed out on involves the same operation as will be used with the distributed term. In this case the distibuted term can be shoved down into the bag below (by making it the initial value of the numeric bag). Mulbuild and plusbuild add whatever comes back from this to the accumulators. There is a decision here concerning which bag (symbolic or numeric) it goes in. If evaluation works on the incoming numeric bag (Ltag) and the element then this gives a new value for this bag (Rtag). Otherwise it will be put into the symbolic bag. There is a special case here for constructing with (previously) empty bags. (Not wanting explicit tail markers, like [] in lists, makes things slightly harder here).

Exptidy is simpler in that it only recursively sweeps one side. Note that it uses multidy so that all the possible multiplication bags on the right hand side of the exp chain will get packed into one. Since this a right to left sweep down the chain there will be some reordering of elements from the original. (However, they are changing from exp chains to mul bags as well - so it's not important). The bottom left term in the chain comes out as the base of course. Thus exptidy has this extra result.

The various ...fin routines take the final (accumulator) symbolic and numeric bags and produce a final term. There are various things going om here: Simplification rules get applied, empty symbolic bags disappear and so forth. Mulfin and plusfin check to see if the symbolic bag is a number, because I also want to be able to use them to glue arbitrary bits together (current example: the use of mulfin in p2tidy). Expfin combines bits of exponential stuff with bits of multiplication stuff (since the

```
    base is to be raised to a mul bag). This makes it a bit more complicated.

    In general one can make use of the zero element reduction to completely
    elininate the need to look at certain bits of the tree. In this
    implementation we would spot that the numeric bag (eg Ltag) had become
    the zero element, and we could then return a result without looking any
    further. This further optimisation is left as an exercise for the reader.

*/

% @@@   (Marker - Ie, easy to find string)

%% Top Level %%

                                % Tidy top level

tidy(X,Ans) :- dotidy(X,Y), !, Ans = Y.

tidy(X,X)
      :- ttynl, display('** TIDY error: '), ttyprint(X),
         ttynl, display('   (trace and continuing...)'), ttynl,
         trace.



                                % The general tidy routine
                                %  Dispatches on special bag types (or logical ops)
                                %  otherwise just tidies arguments recursively
                                %  and then attempts evaluation.

dotidy(V,_) :- var(V), !, tidy_varerr.

dotidy(X,X) :- simple(X), !.

dotidy(A&B,Ans)
      :- !,
         dotidy(A,Ans1),
         dotidy(B,Ans2),
         andfin(Ans1,Ans2,Ans).

    idy(A#B,Ans)
      :- !,
         dotidy(A,Ans1),
         dotidy(B,Ans2),
         orfin(Ans1,Ans2,Ans).

dotidy(X,Ans)
      :- mulbag(X),
         !,
         multidy(X,1,empty,1,Right,Rtag),
         mulfin(Rtag,Right,Ans).

dotidy(X,Ans)
      :- plusbag(X),
         !,
         plustidy(X,1,empty,0,Right,Rtag),
         plusfin(Rtag,Right,Ans).

dotidy(X,Ans)
      :- expbag(X),
          !,
          exptidy(X,empty,1,Mult,Mtag,Base),
          expfin(Mult,Mtag,Base,Ans).

dotidy(X,Newans)
      :- functor(X,Fn,Arity),
          functor(Ans,Fn,Arity),
          tidyall(Arity,X,Ans,win,Flag),
          try_eval(Flag,Ans,Newans).



                              % Simple things are always tidiest

simple(X) :- (atomic(X) ; number(X)), !.



                              % Tidy all the arguments of a term to
                              %  give some new term. Keep track of whether
                              %  or not all the tidied arguments are
                              %  numbers.

tidyall(0,_,_,Flag,Flag) :- !.

tidyall(N,X,Ans,Flag,FinalFlag)
      :- arg(N,X,Arg),
          arg(N,Ans,Narg),
          N1 is N-1,
          dotidy(Arg,Narg),
          chknum(Flag,Narg,Flag2),
          tidyall(N1,X,Ans,Flag2,FinalFlag).



                              % Maintain number checking flag

chknum(lose,_,lose).

chknum(win,X,win) :- number(X), !.

chknum(win,_,lose).



                              % Try to evaluate non bag function
                              %  Eval should just return the structure if it
                              %  can't do the arithmetic

try_eval(lose,X,Y) :- special_fun(X,Y),!.

try_eval(lose,X,X).

try_eval(win,X,Y) :- eval(X,Y).

special_fun(log(U,U^V),V).
special_fun(U^log(U,V),V).
special_fun(sqrt(U),U^number(+,[1],[2])).
```

```
%% Bag Sweeping %%

                                % Types of operation to which bag collecting
                                %  is applicable.

mulbag(A*B).
mulbag(A/B).

plusbag(A+B).
plusbag(A-B).
plusbag(-(A)).

expbag(A^B).


                                % Collecting a multiplication bag together

multidy(V,_,_,_,_,_) :- var(V), !, tidy_varerr.

multidy(A*B,Distr,Left,Ltag,Right,Rtag)
      :- !,
         multidy(A,Distr,Left,Ltag,Q,Qtag),
         multidy(B,Distr,Q,Qtag,Right,Rtag).

multidy(A/B,Distr,Left,Ltag,Right,Rtag)
      :- !,
         multidy(A,Distr,Left,Ltag,Q,Qtag),
         distr_inverse(Distr,Idistr),
         multidy(B,Idistr,Q,Qtag,Right,Rtag).

multidy(X,Distr,Left,Ltag,Right,Rtag)
      :- m2tidy(X,Distr,Q),
         mulbuild(Q,Left,Ltag,Right,Rtag).



m2tidy(E,Distr,Ans)
      :- expbag(E),
         !,
         exptidy(E,empty,Distr,Result,Tag,Base),
         expfin(Result,Tag,Base,Ans).

m2tidy(X,1,Ans)
      :- !,
         dotidy(X,Ans).

m2tidy(X,Distr,Ans)
      :- dotidy(X,Result),
         expfin(empty,Distr,Result,Ans).



                                % Build mul bags with various special cases
                                %  handled.

mulbuild(N,Left,Ltag,Left,Rtag)
      :- number(N),
         multiply(N,Ltag,Rtag),
         !.
```

```
mulbuild(X,empty,Ltag,X,Ltag) :- !.

mulbuild(X,Left,Ltag,Left*X,Ltag).


                          % Collecting a plus bag together

plustidy(V,_,_,_,_,_) :- var(V), !, tidy_varerr.

plustidy(A+B,Distr,Left,Ltag,Right,Rtag)
     :- !,
        plustidy(A,Distr,Left,Ltag,Q,Qtag),
        plustidy(B,Distr,Q,Qtag,Right,Rtag).

plustidy(A-B,Distr,Left,Ltag,Right,Rtag)
     :- !,
        plustidy(A,Distr,Left,Ltag,Q,Qtag),
        distr_inverse(Distr,Idistr),
        plustidy(B,Idistr,Q,Qtag,Right,Rtag).

   istidy(-(A),Distr,Left,Ltag,Right,Rtag)
     :- !,
        distr_inverse(Distr,Idistr),
        plustidy(A,Idistr,Left,Ltag,Right,Rtag).

plustidy(X,Distr,Left,Ltag,Right,Rtag)
     :- p2tidy(X,Distr,Q),
        plusbuild(Q,Left,Ltag,Right,Rtag).



p2tidy(M,Distr,Ans)
     :- mulbag(M),
        !,
        multidy(M,1,empty,Distr,Result,Tag),
        mulfin(Tag,Result,Ans).

p2tidy(X,1,Ans)
     :- !,
        dotidy(X,Ans).

p2tidy(X,Distr,Ans)
     :- dotidy(X,Result),
        mulfin(Distr,Result,Ans).



                          % Build plus bags with various special cases
                          %  handled.

plusbuild(N,Left,Ltag,Left,Rtag)
     :- number(N),
        add(N,Ltag,Rtag),
        !.

plusbuild(X,empty,Ltag,X,Ltag) :- !.

plusbuild(X,Left,Ltag,Left+X,Ltag).
```

```
                                  % Collecting together exp bags

exptidy(V,_,_,_,_,_) :- var(V), !, tidy_varerr.

exptidy(A^B,Right,Rtags,Left,Ltags,Base)
      :- !,
         multidy(B,1,Right,Rtags,Q,Qtags),
         exptidy(A,Q,Qtags,Left,Ltags,Base).

exptidy(X,Right,Rtags,Right,Rtags,Base)
      :- dotidy(X,Base).



                                  % Inverting factors being distributed

distr_inverse(1,-1).
distr_inverse(-1,1).



%% Finalising Structures %%

                                  % Final AND building

andfin(false,X,false) :- !.       % Zero element
andfin(true,X,X) :- !.            % Unit element
andfin(X,false,false) :- !.       % Zero element
andfin(X,true,X) :- !.            % Unit element
andfin(X,X,X) :- !.               % Merging of identical elements
andfin(X,Y,X&Y).                  % General case



                                  % Final OR building

orfin(true,X,true) :- !.          % Zero element
orfin(false,X,X) :- !.            % Unit element
[illegible]in(X,true,true) :- !.  % Zero element
[illegible]in(X,false,X) :- !.    % Unit element
orfin(X,X,X) :- !.                % Merging of identical elements
orfin(X,Y,X#Y).                   % General case



                                  % Final multiplication building

mulfin(0,_,0) :- !.               % Zero element
mulfin(N,empty,N) :- !.           % Completely evaluated
mulfin(1,X,X) :- !.               % Unit element
mulfin(N,N2,Ans)                  % Further evaluation possible
      :- number(N2),
         multiply(N,N2,Ans),
         !.
mulfin(N,Bag*N2,Ans)              % Caught a nested mult bag
      :- number(N2),
         multiply(N,N2,N3),
         mulfin(N3,Bag,Ans),
          !.

mulfin(N,X,X*N).                % General case



                          % Final plus building

plusfin(N,empty,N) :- !.        % Completely evaluated
plusfin(0,X,X) :- !.            % Unit element
plusfin(N,N2,Ans)               % Further evaluation possible
     :- number(N2),
        add(N,N2,Ans),
          !.
plusfin(N,Bag+N2,Ans)           % Caught a nested plus bag!
     :- number(N2),
        add(N,N2,N3),
        plusfin(N3,Bag,Ans),
          !.

plusfin(N,X,X+N).               % General case



                          % Final exp building

expfin(_,0,_,1) :- !.           % E^0 -> 1
expfin(_,_,0,0) :- !.           % 0^X -> 0
expfin(empty,N,E,Ans)           % special empty bag cases
     :- !,
        n_expfin(N,E,Ans).

expfin(Bag,1,E,E^Bag) :- !.     % case with bag unit element

expfin(Bag,N,E,Z^Bag)           % E^(Bag*N) -> (E^N)^Bag
     :- number(E),              %  providing E^N will evaluate
        power(E,N,Z),
          !.

expfin(Bag,N,E,E^(Bag*N)).      % General case



                          % special exp cases for when the symbolic
                          %  bag is empty

n_expfin(1,E,E) :- !.           % E^1 -> E

n_expfin(N,E,Ans)               % E^N evaluates
     :- number(E),
        power(E,N,Ans),
          !.

n_expfin(N,E,E^N).              % General case for empty bag


%% Junk %%
```

```
                                        % Produce an error message and fail (when
                                        %  variables are found).
                                        %  The failure will trip the Tidy top level
                                        %  error message (who throws a nl for us).

tidy_varerr :- ttynl, display('** Prolog variable in expression'), fail.
```