MBASE -(Mecho Support System)

Loading and running

- MBASE -
- KNOW
- Directory listing
- MECHO.OPS -
- MECHO.INI -

PLCODE:

Meta level Database (static)

PLCODE.SUB	PREDS -	META -	MUST -	ERR -
PLCODE	MLPRP1	KSTYPE -	LOAD -	
PLCODE_CNG	MLPRP2 -	T1 -	KS -	
PLIB.OPS -	MLFACE -	T2 ·	RULEF -	
		T3 ·	TYLOAD -	
		T4 ·		

XREF, cross reference listing

utilities, etc.

- POLICE -
- MECHOU
- WDSW -
- OK -
- (BITS) HACKS -
- (LOOK).

Problem Solver

- TOP -
- MARPLE -
- CHOOSE -
- APPLIC -
- FMS -

Object Level Database (fluid)

- INPUT -
- KNOWN -
- INDEX -

Inference engine

CC -	HIST -
PROVEØ -	BOUND -
PRFØ	META2 -
PSØ -	RUN -
	TYPES -

```
/* MBASE : Load Mecho support code
                                                                    [illegible]
                                                                    Lawrence
                                                                    Updated: 1 December 81
*/

 %% Consult this file after running MUTIL or UTIL

  % This file loads all the Mecho support machinery. Mecho's actual knowledge
  % of the domain is not included - this is loaded by the file KNOW.


   :- [

%------------------
% Initialisations
%------------------

        'mecho.ops',                % Mecho specific operator declarations
                                    %  (NB util.ops & plib.ops will also be loaded)
        'mecho.ini',                % Initialisations

%----------------------------------
% Basic predicate library support
%----------------------------------

        'plcode:plcode',            % Compile Predicate library support

%---------------------------
% The Mecho Problem Solver
%---------------------------

         top,                       % Top level
         marple,                    % Marples - goal directed problem solver
         choose,                    % Applying strategies
         applic,                    % Applicable formulae selection        gounls

        'fms.pl',                   % (Re)loading formulae definitions

%---------------------
% Database management
%---------------------

        'input.pl',                 % Input facts
        'db.pl',                    % Asserting into database
        'known.pl',                 % Database management
        'index.pl',                 % Database indexing

%-----------------------------
% Underlying Inference Engine
%-----------------------------

        'cc.pl',                    % Interface to inference system
        'prove0.pl',                % Mecho Theorem Prover
        'prf0.pl',                  % Proof execution
        'ps0.pl',                   % Proof structure operations

        'hist.pl',                  % Proof history operations
        'bound.pl',                 % Instantiation states
        'meta2.pl',                 % Miscellaneous meta predicates
        'run.pl',                   % Terminal top level for inference engine

        'types.pl',                 % Type hierarchy operations
%       'thload.pl',                % Theory loading loop (belongs elsewhere)
%       'thcrun.pl',                % Theory transformation (belongs elsewhere)

%--------------------
% Utilities and Junk
%--------------------

        'police.pl',                % Invariant enforcement

        'mechou.pl',                % Mecho specific utilities
        'wdsin.pl',                 % wordsin utility
%       'vector.pl',                % Vector arithmetic
         ok,                        % Setting up runnable images (convenience)
         ~~bits,~~                  % odd bits (to be removed)
         hacks,                     % Various, possibly temporary, things
        'util:feach.pl'             % (General) foreach utility

    ].

:-       db_init.                   % Set up database
```

```
/* KNOW : Load Mecho's "knowledge"

                                                Lawrence
                                                Updated: 30 November 81
*/

%% Consult from a Mecho Base (MBASE)

:- [

%----------------------
% The predicate library
%----------------------

        'plib:pldef',           % Load Predicate library

%---------------------------------------------------------
% Coded inference rules (ie hacks/stuff to be done better)
%---------------------------------------------------------

        'k:force',              % (coded) Inference about forces etc
        'k:infer',              % (coded) Inference rules - various
        'k:schema'              % (coded) Schemata
   ].

%------------------------------
% Physical formulae definitions
%------------------------------

                           % NB This is an assertion used by 'fms'
formulae([
        'fm:resolv.fm',         % Resolution of forces
        'fm:mom.fm',            % Turning moments
        'fm:sum.fm',            % Time and length sums
        'fm:const.fm',          % Motion under constant acceleration
        'fm:rel.fm',            % Relative motion
        'fm:hooke.fm'           % Springs
 ]).

  :- fms.                       % Load the formulae list

%-------------------------------
% Already known particular facts
%-------------------------------

  :- input( 'k:facts' ).        % a few facts
```

```
;; MBASE.SUB : Mecho Base - Problem solver and Inference System
;;
;;                                                          Updated: 15 January 82
;;
mbase.sub                ;; This file
mecho.ops                ; Mecho specific operator declarations
mecho.ini                ;; Initialisations
top                      ; Top level
marple                   ; Marples - goal directed problem solver
quants                   ; analyse_result machinery
choose                   ; Applying strategies
applic                   ; Applicable formulae selection
fms.pl                   ; (Re)loading formulae definitions
input.pl                 ; Input facts
db.pl                    ; Asserting into database
known.pl                 ; Database management
index.pl                 ; Database indexing
cc.pl                    ; Interface to inference system
prove0.pl                ; Mecho Theorem Prover
method.pl                ; Proof methods
  -0.pl                  ; Proof structure operations
    st.pl                ; Proof history operations
bound.pl                 ; Instantiation states
meta2.pl                 ; Miscellaneous meta predicates
run.pl                   ; Terminal top level for inference engine
types.pl                 ; Type hierarchy operations
police.pl                ; Invariant enforcement
mechou.pl                ; Mecho specific utilities
wdsin.pl                 ; wordsin utility
vector.pl                ;; Vector arithmetic
ok                       ; Setting up runnable images (convenience)
hacks                    ; odd bits (to be removed)
```

```
mecho.ops

top.
marple.
quants.
choose.
applic.
fms.pl
input.pl
db.pl
known.pl
index.pl
cc.pl
prove0.pl
method.pl
ps0.pl
hist.pl
bound.pl
meta2.pl
run.pl
types.pl
police.pl
mechou.pl
wdsin.pl
ok.
hacks.
```

```
plcode:plib.ops
plcode:preds.pl
plcode:mlprp2.pl
plcode:mlface.pl
plcode:meta.pl
plcode:kstype.pl
plcode:t1.pl
plcode:t2.pl
plcode:t3.pl
plcode:t4.pl
plcode:must.pl
plcode:load.pl
plcode:ks.pl
plcode:rulef.pl
plcode:tyload.pl
plcode:err.pl
mecho:mecho.ops
mecho:top.
mecho:marple.
mecho:quants.
mecho:choose.
 cho:applic.
mecho:fms.pl
mecho:input.pl
mecho:db.pl
mecho:known.pl
mecho:index.pl
mecho:cc.pl
mecho:prove0.pl
mecho:method.pl
mecho:ps0.pl
mecho:hist.pl
mecho:bound.pl
mecho:meta2.pl
mecho:run.pl
mecho:types.pl
mecho:police.pl
mecho:mechou.pl
mecho:wdsin.pl
  cho:ok.
  cho:hacks.
```

```
       Directory listing        1-Dec-81        4:45:31         Page  1
Name Extension Len  Prot      Access       ---Creation---   Mode      Version

DSKB:   [400,441,MECTOP]
PLIB    SFD       5  <775>   30-Nov-81      1:35   12-Dec-80  17
PLCODE  SFD       5  <775>   22-Nov-81      2:52   11-Apr-81  17
MECHO   SFD       5  <775>    1-Dec-81     21:25   28-Apr-81  17
MECHO   MIC       5  <005>   30-Nov-81     18:56    1-Jun-81   0
EQENTR            5  <005>    4-Nov-81     23:59    1-Jun-81   0
FM      SFD       5  <775>   11-Nov-81      5:05    2-Jun-81  17
TH      SFD       5  <775>   11-Nov-81      7:29    6-Sep-81  17
K       SFD       5  <775>   11-Nov-81      7:33    6-Sep-81  17
COAST   SFD       5  <775>    1-Dec-81      1:43   23-Nov-81  17
MOFI    SFD       5  <775>    1-Dec-81      1:43   23-Nov-81  17
MBASE   BAK      10  <005>   30-Nov-81     23:36   30-Nov-81   0
KNOW              5  <005>   30-Nov-81     23:44   30-Nov-81   0
MBASE            10  <005>    1-Dec-81      4:09    1-Dec-81   0
  Total of 75 blocks in 13 files on DSKB: [400,441,MECTOP]

[400,441,MECTOP,PLIB]
PLIB    HLP      15  <005>   23-Nov-81      4:26   11-Apr-81   0
 TA     DOC       5  <005>   23-Nov-81      0:20   14-Jun-81   0
LATEG             5  <005>   22-Nov-81     10:49   13-Jul-81   0
MOTION  DEF      10  <005>   27-Nov-81     18:56    5-Aug-81   0
AGES    DEF       5  <005>   26-Nov-81     18:58    5-Aug-81   0
NOTREC  OLD       5  <005>   22-Nov-81     19:03    5-Aug-81   1
TIME    DEF       5  <005>   27-Nov-81     21:46   25-Aug-81   0
PLIB    FL        5  <005>   26-Nov-81      1:28    6-Sep-81   0
PLDEF             5  <005>   27-Nov-81      1:38    6-Sep-81   0
PLIB    SUB       5  <005>   26-Nov-81      1:44    6-Sep-81   0
UNITS   DEF       5  <005>   27-Nov-81     12:51   10-Sep-81   0
NLPRED  DEF       5  <005>   26-Nov-81      9:53   14-Sep-81   0
TYPES   HI       10  <005>   27-Nov-81      9:54   14-Sep-81   0
OBJP    DEF      10  <005>   27-Nov-81     10:00   23-Sep-81   0
OBJR    DEF      15  <005>   27-Nov-81      9:16   28-Sep-81   0
CONTAC  DEF      15  <005>   27-Nov-81      9:28   28-Sep-81   0
SOLLIN  DEF      10  <005>   27-Nov-81     14:27    2-Nov-81   0
SPACE   DEF      10  <005>   27-Nov-81     14:28    2-Nov-81   0
PLIB    CNG      15  <005>   22-Nov-81     14:29    2-Nov-81   0
PLIB             10  <005>   26-Nov-81     20:31   22-Nov-81   0
  Total of 170 blocks in 20 files on DSKB: [400,441,MECTOP,PLIB]

[400,441,MECTOP,PLCODE]
PLIB    OPS       5  <005>    1-Dec-81     16:52   15-Jun-81   0
RULEF   PL        5  <005>    1-Dec-81     16:53   15-Jun-81   0
KSTYPE  PL       10  <005>    1-Dec-81     16:53   15-Jun-81   0
ERR     PL        5  <005>    1-Dec-81     16:53   15-Jun-81   0
LOAD    PL       15  <005>    1-Dec-81      9:04    6-Jul-81   0
PREDS   PL        5  <005>    1-Dec-81     15:17    9-Jul-81   0
T3      PL        5  <005>    1-Dec-81     17:22    9-Jul-81   0
T2      PL        5  <005>    1-Dec-81     17:22    9-Jul-81   0
T4      PL        5  <005>    1-Dec-81     17:23    9-Jul-81   0
MUST    PL        5  <005>    1-Dec-81     17:24    9-Jul-81   0
T1      PL       10  <005>    1-Dec-81      2:09   10-Jul-81   0
WELL              5  <005>   22-Nov-81      2:25   10-Jul-81   0
MEDIN             5  <005>   22-Nov-81     15:37   13-Jul-81   0
KS      PL       15  <005>    1-Dec-81     17:04    4-Aug-81   0
TYLOAD  PL       10  <005>    1-Dec-81     14:22   27-Aug-81   0
```

```
        Directory listing        1-Dec-81        4:45:34        Page  2
Name Extension Len  Prot      Access      ---Creation---   Mode     Version

META     PL      5  <005>    1-Dec-81     3:26    6-Sep-81   0
MLPRP1   PL     10  <005>    1-Dec-81     3:27    6-Sep-81   0
PLCODE   CNG    10  <005>   22-Nov-81     4:10    6-Sep-81   0
PLCODE   SUB     5  <005>    1-Dec-81    22:49   22-Nov-81   0
PLCODE           5  <005>   30-Nov-81    22:52   22-Nov-81   0
MLFACE   PL     10  <005>    1-Dec-81     1:43    1-Dec-81   0
  Total of 155 blocks in 21 files on DSKB: [400,441,MECTOP,PLCODE]

[400,441,MECTOP,MECHO]
POLICE   PL      5  <005>    1-Dec-81    21:54    5-Jul-81   0
SHELF    SFD     5  <775>    1-Dec-81     2:34    6-Jul-81  17
INDEX    PL      5  <005>    1-Dec-81     7:53    6-Jul-81   0
TYPES    PL     10  <005>    1-Dec-81    14:09   13-Jul-81   0
FMS      PL      5  <005>    1-Dec-81     6:50    6-Sep-81   0
NORMAL          10  <005>    1-Dec-81    13:44   10-Apr-81  14
OK               5  <005>    1-Dec-81     5:31    6-Jul-81   0
APPLIC          10  <005>    1-Dec-81     3:33    2-Jun-81   0
VECTOR   PL      5  <005>   30-Nov-81    14:03    1-Apr-81  14
[illegible]IND   PL     10  <005>    1-Dec-81    23:03   30-Nov-81   0
[illegible]IN    PL      5  <005>    1-Dec-81    23:38   30-Nov-81   0
BITS             5  <005>    1-Dec-81    23:39   30-Nov-81   0
HIST     PL      5  <005>    1-Dec-81     0:21    1-Dec-81   0
CC       PL      5  <005>    1-Dec-81     0:34    1-Dec-81   0
MECHO    INI     5  <005>    1-Dec-81     0:36    1-Dec-81   0
MECHO    OPS     5  <005>    1-Dec-81     0:36    1-Dec-81   1
PSO      PL      5  <005>    1-Dec-81     1:50    1-Dec-81   0
PRFO     PL      5  <005>    1-Dec-81     2:30    1-Dec-81   0
META2    PL      5  <005>    1-Dec-81     2:41    1-Dec-81   1
MECHOU   PL      5  <005>    1-Dec-81     2:45    1-Dec-81   0
PROVEO   PL     15  <005>    1-Dec-81     3:29    1-Dec-81   0
INPUT           10  <005>    1-Dec-81     3:36    1-Dec-81   0
MARPLE          10  <005>    1-Dec-81     3:50    1-Dec-81   0
KNOWN    PL     15  <005>    1-Dec-81     3:57    1-Dec-81   0
CHOOSE          10  <005>    1-Dec-81     3:58    1-Dec-81   0
MBASE    SUB     5  <005>    1-Dec-81     4:17    1-Dec-81   1
FLS              5  <005>    1-Dec-81     4:20    1-Dec-81   0
FL               5  <005>    1-Dec-81     4:20    1-Dec-81   0
[illegible]LL              5  <005>    1-Dec-81     4:29    1-Dec-81   0
[illegible]             10  <005>    1-Dec-81     4:32    1-Dec-81   0
RUN      PL     10  <005>    1-Dec-81     4:35    1-Dec-81   0
  Total of 220 blocks in 31 files on DSKB: [400,441,MECTOP,MECHO]

[400,441,MECTOP,MECHO,SHELF]
T1       OLD     5  <005>    4-Nov-81    16:53   15-Jun-81   0
TYLOAD   OLD    10  <005>    4-Nov-81    10:02   27-Jul-81   0
ACCESS   OLD    10  <005>    4-Nov-81     0:28   27-Feb-81   0
INPUT    OLD     5  <005>    1-Dec-81     2:16    1-Dec-81   1
  Total of 30 blocks in 4 files on DSKB: [400,441,MECTOP,MECHO,SHFLF]

[400,441,MECTOP,FM]
RESOLV   FM     10  <005>   23-Nov-81     4:15    2-Jun-81   0
MOM      FM      5  <005>   23-Nov-81     4:25    2-Jun-81   0
HOOKE    FM      5  <005>    4-Nov-81     4:36    2-Jun-81   0
REL      FM      5  <005>    4-Nov-81     4:51    2-Jun-81   0
CONST    FM      5  <005>   23-Nov-81     5:00    2-Jun-81   0
```

```
        Directory listing        1-Dec-81        4:45:36        Page  3
Name Extension Len  Prot     Access      ---Creation---   Mode     Version

SUM     FM       5  <005>   23-Nov-81     5:05    2-Jun-81   0
  Total of 35 blocks in 6 files on DSKB: [400,441,MECTOP,FM]

[400,441,MECTOP,TH]
R       TH       5  <005>   23-Nov-81     7:39   29-May-81   0
S       TH       5  <005>    4-Nov-81     7:40   29-May-81   0
FOO              5  <005>   23-Nov-81     9:03   29-May-81   0
THLOAD  PL      10  <005>    4-Nov-81    22:41   13-Jun-81   0
RELS    DEF      5  <005>    4-Nov-81    10:38   29-May-81   0
  Total of 30 blocks in 5 files on DSKB: [400,441,MECTOP,TH]

[400,441,MECTOP,K]
FORCE           10  <005>   23-Nov-81    16:34    8-Apr-81  14
INFER           15  <005>    4-Nov-81    16:21   20-Apr-81  14
FACTS            5  <005>   30-Nov-81    23:54    1-Jun-81   0
SCHEMA          10  <005>    4-Nov-81     0:17    2-Jun-81   0
  Total of 40 blocks in 4 files on DSKB: [400,441,MECTOP,K]

   RA:   [400,441,MECTOP]
    HO  SFD     10  <775>    1-Dec-81     4:39    1-Dec-81  17
  Total of 10 blocks in 1 file on SCRA: [400,441,MECTOP]

[400,441,MECTOP,MECHO]
MBASE   RND     40  <005>    1-Dec-81     4:36    1-Dec-81   0
MBASE   MEM     40  <005>    1-Dec-81     4:37    1-Dec-81   0
  Total of 80 blocks in 2 files on SCRA: [400,441,MECTOP,MECHO]
  Grand total of 845 blocks in 107 files
```

```
/* MECHO.OPS : Operator declarations for Mecho (ie MBASE)

                                                    Lawrence
                                                    Updated: 1 December 81
*/


  :- op(710,fx,[dc,ncc,cc,pc,cue]).
```

```
/* MECHO.INI : Various initialisations for MECHO.

                                                                  Lawrence
                                                                  Updated: 1 December 81
*/


/* Global Flags */

  :- flag(ccflag,_,on),          % cc create flag
     flag(tflag,_,4),            % trace level
     flag(eqlabel,_,0),          % equation label counter
     flag(filter,_,in),          % applicable formulae filtering
     flag(accept,_,on).          % "Do you accept..." interruptions
```

MBASE

TODO 1 Dec 81

number(_) ✓ (MUTIL)

check INDEX. <2> PLCODE: META. ✓

Type expansion. ✓

(Theories) - esp equiv/simil

("Picking up ancestors/history")

remember

History weak

No negrules

default rules ~ dependencies

~~1) Marking~~ ~~Copies~~ Alan Chris

~~2) EBCC (Now) (Test)~~ ✓

~~3) [illegible]~~

~~4) Richard/Luis (Reasons)~~

~~(5) Charles & terminals.)~~

```
                    **********************************
                    * PROLOG CROSS REFERENCE LISTING *
                    **********************************

                MBASE - Problem Solver and Inference System


  PREDICATE                FILE           CALLED BY



<-> /2                  undefined ✓    <user> or_rule/2 and_rule/2

? /1                    bits

[illegible] /2          utility        and_rule/2 run_chk_conti/i run_chk_cont2/4

accept/1                marple         marples/7

accmess/2               marple         accept/1

add_entry/5             plcode:ks.pl   chkadd/6

add_hidden_ands/2       plcode:tyload.pl rewrite_a_type/0 add_hidden_ands/2

add_rule/2              plcode:tyload.pl     ty_process/1      rewrite_a_type/0
                                       add_hidden_ands/2

add_type_info/2         types.pl       db_assert/1

addtoslot/2             plcode:ks.pl   add_entry/5 addtoslot/2

addword/3               wdsin.pl       wordsin/3

[illegible]iorelative/2 plcode:mlprp2.pl

aliorelative_pattern/3  plcode:mlface.pl aliorelative/2

all_ground/2            bound.pl       ground/1 all_ground/2

allbound/2              bound.pl       exists/1  unique/1   function/2   exists/2
                                       unique/2 allbound/2

allunbound/2            bound.pl       exists/1 function/2 exists/2 allunhound/2

already_applied/3       undefined ✓    apply_strategy/2

and_rule/2              plcode:tyload.pl       rewrite_a_type/0         basify/3
                                       do_derived_types/0

append/3                utility        marples/7
```

```
appl/2                  applic       <user> applicable_formulae/3

applicable_formulae/3   applic       choose_and_apply_strategy/5

apply_strategy/2        choose       choose_and_apply_strategy/5

argument_names/1        plcode:mlface.pl

argument_names/2        plcode:mlface.pl proof_method/3

argument_types/1        plcode:mlface.pl

argument_types/2        plcode:mlface.pl handle_definitions/2

backthrough/2           plcode:ks.pl  ks_flush/2 set/3 backthrough/2

basify/3                plcode:tyload.pl        rewrite_a_type/0        basify/3
                                      do_derived_types/0

before/2                applic        partition/4

blank_ks/2              plcode:ks.pl  new_ks/3  ks_flush/2   unknown_predicate/1
                                      fetch/3

bound/1                 bound.pl

cassertz/1              utility       apply_strategy/2

cc/1                    cc.pl         db_normal_form/1

cgensym/2               utility       gensym_args/3

change_i_pattern/2      types.pl      add_type_info/2

check_and_add/6         plcode:load.pl loading/5

check_pred/2            plcode:load.pl loading/5

check_type/2            plcode:load.pl loading/5

chk_names/2             plcode:load.pl check_pred/2 chk_names/2

chkadd/6                plcode:load.pl check_and_add/6

choose_and_apply_strategy/5           choose marples/7

close/2                 utility       load/1

combine_plans/4         prove0.pl     filter_plan/3

commutative/2           plcode:mlprp2.pl

commutative_pattern/3   plcode:mlface.pl commutative/2

compatible/2            types.pl      not_type/2
```

```
complete_ks/2           plcode:ks.pl  finalise/2

compound_plan/4         prove0.pl     filter_plan/3 combine_plans/4

concat/3                utility       input/1

cons_fact_name/2        known.pl      next_fact_name/1 db_scrub/2

consider_history/3      hist.pl       prove/3

consider_pruning/2      prove0.pl     plan_proof/6

const/1                 undefined ✓   constant/1

constant/1              wdsin.pl      wordsin/3

continue/0              utility       <user> set_type/2

copy_args/3             plcode:preds.pl proof_method/3

copy_args_1/3           plcode:preds.pl copy_args/3 copy_args_1/3

db_add/1                known.pl      h_defs/4

db_add/2                known.pl      db_assert/1 db_add/1

db_add_keys/2           known.pl      db_add/2

db_assert/1             db.pl         intprt2/1   db_assert/1   db_normal_form/1
                                      proof_method/3

db_forget/1             known.pl

db_forget/2             known.pl      db_forget/1

db_init/0               known.pl

db_normal_form/1        db.pl         db_assert/1

db_restore/1            known.pl

db_scrub/2              known.pl      db_restore/1 db_scrub/2

db_state/1              known.pl      go/0

dc/1                    cc.pl

default_rule/2          plcode:mlface.pl proof_method/3

definition/3            db.pl         choose_and_apply_strategy/5

derived/1               meta2.pl      make_plan/3

do_basic_types/0        plcode:tyload.pl finish_types/0
```

do_derived_types/0	plcode:tyload.pl	finish_types/0
easy_inference/1	plcode:mlface.pl	size_up_task/3
edit/1	utility	must_change/1
eliminable/1	undefined	marples/7
errmess/1	plcode:err.pl	load/1 read_next/1
errmess/2	plcode:err.pl	t2_checkdo/2 check_pred/2 check_type/2 pred_ok/2 type_ok/2 chkadd/6 ks_slot/3 stillf/2 ks_key/2 ty_process/1 type_pattern/2
error/3	utility	must_know_predicate/1 set_type/2 db_assert/1 db_restore/1 must_be_term/2 must_be_ground/2
establishes_falsity/1	prove0.pl	postmortem/3
eval/1	utility	prove_single/3
eval/2	utility	prove_single/3
exists/1	plcode:mlprp2.pl	
exists/2	plcode:mlprp2.pl	method_applicable/2
exists_pattern/3	plcode:mlface.pl	exists/1 exists/2
fetch/3	plcode:ks.pl	set/3
file_exists/1	utility	input/1
fillall/2	plcode:ks.pl	complete_ks/2 fillall/2
fillslot/1	plcode:ks.pl	fillall/2 fillslot/1
filter/3	applic	applicable_formulae/3
filter/6	applic	filter/3 filter/6
filter_plan/3	prove0.pl	make_plan/3 filter_plan/3
finalise/2	plcode:ks.pl	loading/5
findall/3	utility	treep/3 soughts/1 givens/1 applicable_formulae/3
finish_types/0	plcode:tyload.pl	ty_process/1
flag/3	utility	marples/7 tell1/2 accept/1 filter/3 method_applicable/2

```
fms/0                   fms.pl

forget_fact/1           known.pl       db_forget/2 db_scrub/2

formul_sort/2           applic         applicable_formulae/3

formula_predicate/1     fms.pl         zap_fms/0

formulae/1              undefined ✓    fms/0

function/2              plcode:mlprp2.pl

function_pattern/3      plcode:mlface.pl function/2

gak/6                   known.pl       get_a_key/4 gak/6

gensym_args/3           prf0.pl        proof_method/3 gensym_args/3

  t/3                   plcode:ks.pl   commutative_pattern/3
                                       aliorelative_pattern/3  function_pattern/3
                                       exists_pattern/3          unique_pattern/3
                                       normal_form/2          object_level_rule/2
                                       object_level_neg_rule/2     default_rule/2
                                       argument_names/1          argument_names/2
                                       argument_types/1          argument_types/2
                                       easy_inference/1 index/2 not_derived/1

get_a_key/4             known.pl       known2/2 db_add_keys/2 forget_fact/1

get_i_pattern/2         types.pl       type/2    compatible/2     add_type_info/2
                                       print_types/1

get_indiv/1             types.pl       get_i_pattern/2

get_rule/3              plcode:rulef.pl    normal_form/2     object_level_rule/2
                                       object_level_neg_rule/2 default_rule/2

  t_type/2              top            get_types/3

get_types/3             top            solve_problem/2 get_types/3 marples/7

givens/1                top            solve_problem/2

go/0                    top

go/1                    top

ground/1                bound.pl       bound/1   pure/1    allbound/2    ground/1
                                       all_ground/2 must_be_ground/2

h_defs/4                db.pl          handle_definitions/2 h_defs/4

handle_definitions/2    db.pl          db_assert/1

hidden_ors/2            plcode:tyload.pl rewrite_a_type/0 hidden_ors/2
```

his_pattern/2	types.pl	set_i_pattern/2
how_dest/3	prf0.pl	proof_method/3
i_pattern/2	undefined ✓*	<user> his_pattern/2 set_indiv/1
indep/2	choose	choose_and_apply_strategy/5
index/2	index.pl	set_a_key/4
input/1	input.pl	
intprt/1	input.pl	input/1
intprt2/1	input.pl	intprt/1 intprt2/1
isform/3	undefined	apply_strategy/2
own/1	known.pl	<user> soughts/1 givens/1 set_type/2 db_assert/1 no_defn/1 definition/3 proof_method/3 problem/0
known/2	known.pl	definition/3 db_forget/2
known2/2	known.pl	known/1 known/2
known_predicate/1	plcode:ks.pl	must_know_predicate/1
ks_flush/2	plcode:ks.pl	finalise/2
ks_init/0	plcode:ks.pl	
ks_key/2	plcode:ks.pl	finalise/2 unknown_predicate/1 fetch/3
ks_max/1	plcode:kstype.pl	blank_ks/2
ks_recforms/3	plcode:ks.pl	ks_record_ruleforms/1 ks_recforms/3
_record_ruleforms/1	plcode:ks.pl	finalise/2
ks_slot/3	plcode:ks.pl	add_entry/5 ks_flush/2 complete_ks/2 fetch/3
ks_style/5	plcode:ks.pl	add_entry/5
ks_translate/4	plcode:kstype.pl	chkadd/6
ks_type/1	plcode:kstype.pl	check_type/2
ks_type/3	plcode:kstype.pl	ks_type/1 ks_slot/3 add_entry/5 ks_flush/2
load/1	plcode:load.pl	must_change/1 load/1
load_finish/1	plcode:load.pl	load_sortout/2 ty_process/1

```
load_fms/1                fms.pl        fms/0

load_resync/0             plcode:load.pl

load_sortout/2            plcode:load.pl load_resync/0

load_start/1              plcode:load.pl load/1 load_resync/0 ty_process/1

loading/5                 plcode:load.pl load_start/1 loading/5

make_plan/3               prove0.pl     plan_proof/6

make_ruleform/3           plcode:rulef.pl t3_trans/3 t4_trans/3

maketype/2                plcode:tyload.pl treep/3 do_derived_types/0

marples/7                 marple        solve/5 marples/7

  ber/2                   utility       choose_and_apply_strategy/5        indep/2
                                        useless/2

memberchk/2               utility       new_quantities/5
                                        choose_and_apply_strategy/5 twoin/3 appl/2
                                        consider_history/3 addword/3

meta_predicate/2          plcode:meta.pl ti_trans/3

meta_predicate_index/2    plcode:meta.pl index/2

method_applicable/2       prove0.pl     filter_plan/3

must_be_ground/2          police.pl     db_add/2

must_be_term/2            police.pl     known2/2 db_add/2

must_change/1             plcode:must.pl must_know_predicate/1

  st_know_predicate/1     plcode:must.pl must_change/1 fetch/3

ncc/1                     cc.pl

new_ks/3                  plcode:ks.pl  loading/5

new_quantities/5          marple        marples/7

next_fact_name/1          known.pl      db_add/2

no_defn/1                 db.pl         h_defs/4

nonvar_same_predicate/2 plcode:preds.pl   t1_trans2/4   t2_trans/3    t3_trans/3
                                        t4_trans/3

normal_form/2             plcode:mlface.pl db_assert/1 proof_method/3

not_derived/1             meta2.pl      derived/1
```

```
not_member/2               mechou.pl      set_types/3 not_member/2

not_subsume/2              types.pl       pattern_subsume/2

not_type/2                 types.pl

not_unusual/1              plcode:preds.pl type_predicate/1 type_predicate/3

number/1                   utility        constant/1

object_level_neg_rule/2    plcode:mlface.pl

object_level_rule/2        plcode:mlface.pl proof_method/3

ok/0                       ok

ok/1                       ok

or/2                       applic         filter/6

open/2                     utility        load/1

or_rule/2                  plcode:tyload.pl rewrite_a_type/0 <user> treep/3

p_type_arg/2               types.pl       print_types/1 p_types/3

p_types/3                  types.pl       p_types/3 p_type_arg/2

partition/4                applic         qsort/3 partition/4

pattern_subsume/2          types.pl       type/2 super_type/2

pc/1                       cc.pl

perm2/4                    utility        prove_single/3

plan/5                     choose         choose_and_apply_strategy/5

plan_proof/6               prove0.pl      prove_single/3

postmortem/3               prove0.pl      run_proof/4

pred_ok/2                  plcode:load.pl check_and_add/6

pref/2                     applic         before/2

preference/2               undefined ✓    pref/2

prepare/5                  undefined ✓    plan/5

print_types/1              types.pl

problem/0                  mechou.pl      save_answer/3

proof_exec/3               prf0.pl        proof_start/1 proof_exec/3
```

proof_method/3	prf0.pl	proof_exec/3
proof_start/1	prf0.pl	run_proof/4
prove/2	prove0.pl	dc/1 ncc/1 cc/1 pc/1
prove/3	prove0.pl	prove/2 prove_subgoals/2 prove/3 proof_method/3
prove_single/3	prove0.pl	prove/3
prove_subgoals/2	prove0.pl	proof_method/3
ps_cons/7	ps0.pl	plan_proof/6
ps_dest/3	ps0.pl	proof_start/1
ps_effort/2	ps0.pl	prove_subgoals/2
ps_goal/2	ps0.pl	
ps_history/2	ps0.pl	prove_subgoals/2
ps_plan/2	ps0.pl	
ps_result/2	ps0.pl	proof_start/1 proof_exec/3
pure/1	bound.pl	method_applicable/2
qsort/3	applic	formul_sort/2 qsort/3
read_next/1	plcode:load.pl	load/1 load_resync/0 loading/5 ty_start/0
recorda/1	bits	
reinitialise/0	undefined ?	ok/0 ok/1
lates/2	undefined ✓	appl/2 okfor/2
remove_rule/2	plcode:tyload.pl	rewrite_a_type/0
remove_rules/0	plcode:tyload.pl	finish_types/0
repeat/1	bits	
rewrite_a_type/0	plcode:tyload.pl	rewrite_types/0
rewrite_types/0	plcode:tyload.pl	finish_types/0
ruleform/2	plcode:rulef.pl	ks_flush/2 ks_recforms/3
rulename/2	plcode:rulef.pl	ks_style/5
run/0	run.pl	

run/1	run.pl	run/0 run_cont/1
run_chk_cont1/1	run.pl	run_eval/3
run_chk_cont2/4	run.pl	run_eval/3
run_cont/1	run.pl	run/1
run_eval/2	undefined	run/1
run_eval/3	run.pl	
run_eval2/4	run.pl	run_eval/3 run_eval2/4
run_format/2	run.pl	run_report/3
run_mode/2	run.pl	run/1 run_eval2/4
[illegible]_proof/4	prove0.pl	prove_single/3
run_report/2	run.pl	
run_report/3	run.pl	run_eval/3
same_predicate/2	plcode:preds.pl	
same_predicate/3	plcode:preds.pl	same_predicate/2 same_predicate/3 copy_args/3
save_answer/3	top	go/1
seteq/2	utility	useless/2
size_up_task/3	prove0.pl	plan_proof/6
solve/5	top	solve_problem/2
[illegible]ve_problem/2	top	go/0 go/1
soughts/1	top	solve_problem/2
specific_equation/2	undefined ✓	choose_and_apply_strategy/5
specific_relates/2	undefined ✓	choose_and_apply_strategy/5
standard_plan/2	prove0.pl	make_plan/3
still_fresh/1	plcode:ks.pl	complete_ks/2
stillf/2	plcode:ks.pl	still_fresh/1 stillf/2
subset/2	utility	okfor/2
subtract/3	utility	new_quantities/5

```
subtype/2               plcode:tyload.pl basify/3 treep/3 subtype/2

succ/2                  mechou.pl     prove_single/3

super_type/2            types.pl      h_defs/4

t1_argnorm/2            plcode:t1.pl  t1_trans2/4

t1_collect/2            plcode:t1.pl  ti_trans2/4

t1_copy_args/3          plcode:t1.pl  ti_trans2/4 t1_copy_args/3

t1_sweep/3              plcode:t1.pl  ti_collect/2 t1_sweep/3

t1_sweep_one/4          plcode:t1.pl  ti_sweep/3

t1_trans/3              plcode:t1.pl  ks_translate/4

   _trans2/4            plcode:t1.pl  t1_trans/3

t1_twiddle/2            plcode:t1.pl  t1_trans2/4 t1_twiddle/2

t2_check/2              plcode:t2.pl  t2_trans/3 t2_check/2

t2_checkdo/2            plcode:t2.pl  t2_check/2

t2_flatten/2            plcode:t2.pl  t2_trans/3 t2_flatten/2

t2_trans/3              plcode:t2.pl  ks_translate/4

t3_trans/3              plcode:t3.pl  ks_translate/4

t4_cases/3              plcode:t4.pl  t4_trans/3

t4_norm/2               plcode:t4.pl  t4_trans/3

t4_trans/3              plcode:t4.pl  ks_translate/4

 ell1/2                 marple        marples/7

tell2/2                 marple        marples/7

th_start/1              undefined     loading/5

tidy/2                  utility       prove_single/3

trace/2                 utility       save_answer/3 marples/7 consider_history/3

trace/3                 utility       solve/5    marples/7    telli/2    tell2/2
                                      choose_and_apply_strategy/5
                                      applicable_formulae/3 db_assert/1  prove/2
                                      prove/3 postmortem/3

treep/3                 plcode:tyload.pl do_basic_types/0 treep/3
```

ttyprint/1	utility	errmess/2
twoin/3	choose	useless/2 twoin/3
ty_intersect/2	plcode:tyload.pl	do_derived_types/0 ty_intersect/2
ty_nmember/3	plcode:tyload.pl	treep/3 ty_nmember/3
ty_process/1	plcode:tyload.pl	ty_start/0 ty_process/1
ty_start/0	plcode:tyload.pl	loading/5
type/2	types.pl	prove_single/3
type_name/2	plcode:tyload.pl	treep/3
type_ok/2	plcode:load.pl	check_and_add/6
[illegible]e_pattern/2	plcode:tyload.pl	ty_intersect/2 type/2 super_type/2 compatible/2 add_type_info/2 his_pattern/2
type_predicate/1	plcode:preds.pl	unknown_predicate/1 db_assert/1
type_predicate/3	plcode:preds.pl	t2_flatten/2 prove_single/3
unbound/1	bound.pl	
union/3	utility	new_quantities/5
unique/1	plcode:mlprp2.pl	consider_pruning/2
unique/2	plcode:mlprp2.pl	method_applicable/2
unique_pattern/3	plcode:mlface.pl	unique/1 unique/2
unit/1	bits	
[illegible]nown_predicate/1	plcode:ks.pl	known_predicate/1
unusual/1	plcode:preds.pl	not_unusual/1
useless/2	choose	indep/2
wordsin/2	wdsin.pl	new_quantities/5
wordsin/3	wdsin.pl	wordsin/2 wordsin_term/4
wordsin_term/4	wdsin.pl	wordsin/3 wordsin_term/4
writef/2	utility	save_answer/3 accept/1 input/1 run_report/3 problem/0
zap_fms/0	fms.pl	load_fms/1

```
/* TOP : Top level of the problem solver

                                                    Lawrence
                                                    Updated: 18 December 81
*/

 % This assumes that the problem has been loaded into the database.
% The connection should be made more explicit.


                           % Go go go

go :-   db_state(State),
        asserta( last_state(State) ),           % remember for convenience
        solve_problem(_,_).


                           % Redo from last state
  rego
  go :-                                   bug
        retract( LastState ),
        !,
        db_restore(LastState),
        solve_problem(_,_).


                           % Solve problem and put result into file

go(OutFile)
    :- solve_problem(Equations,Quantities),
       save_answer(OutFile,Equations,Quantities).


                           % Actually solve the problem

solve_problem(Equations,Quantities)
    :- soughts(Soughts),
       givens(Givens),
       get_types(Soughts,[],Xtypes),
       get_types(Givens,Xtypes,Types),
       solve(Soughts,Givens,Types,Equations,Quantities).


                           % (Also used by QA stuff - elsewhere)

solve(Soughts,Givens,Types,Equations,Quantities) :-
            trace('\nAttempting to solve for %t in terms of %t\n\n',
                                                          [Soughts,Givens],1),
        marples(Soughts,Givens,Types,[],Equations,Quantities,Strategies),

            rev(Strategies,Strategies2),                 % Get right way up!
            trace('\n\nStrategies Used : %l\n',[Strategies2],1),
            trace('\nEquations extracted : %c\n',[Equations],1).
```

✓

```
                              % Find all the sought quantities

soughts(Slist)
     :- findall(X, known(sought(X)) ,Slist).



                              % Find all the given quantities

givens(Glist)
     :- findall(X, known(given(X)) ,Glist).



                              % Find the types of a set of quantities
                              %  Algorithm acculculates on the 2nd arg, checking
                              %  for membership so that the final result will be
                              %  a set.

get_types([],Types,Types) :- !.

get_types([X|Rest],Types,TFinal)
     :- get_type(X,T),
        not_member(T,Types),
        !,
        get_types(Rest,[T|Types],TFinal).

get_types([X|Rest],Types,TFinal)
     :- get_types(Rest,Types,TFinal),
        !.



                              % Get type of quantity (from definition info)

get_type(X,T) :- known( defn(X,T,_) ), !.

get_type(X,T) :- error('Type unknown: %t',[X],continue), fail.



                              % Write out answer to a file
                              %  There used to be some problems with the fact that
                              %  Prolog can write out certain terms which then get
                              %  read in wrongly (by PRESS say). This mainly involved
                              %  negative numbers and the use of unary minus in terms.
                              %  Things have been improved a bit since then but they
                              %  are not perfect yet, so watch out.

save_answer(OutFile,Equations,Quantities)
     :- telling(Old),
        tell(OutFile),
                writef('\n/* %t : Mecho output\n',[OutFile]),
                problem,
                writef('\n*/\n\ngo :- simsolve(\n\n%t,\n\n%t\n).\n',
                                                [Equations,Quantities]),
        told,
        tell(Old),
        trace('\nAnswer written to: %t\n\n',[OutFile]).
```

```
/* MARPLE : Mecho Problem solver - The Marples "algorithm"

                                                         Lawrence
                                                         Updated: 11 December 81
*/

  % ( 1 December 81 )
 %
  %       Updated to return final list of strategies.
 %       I must give all these variables nice names some time...
 %
  % ( 10 December 81 )
 %
  %       New analyse_result mechanism added (with elimination checking).
 %       A whole new piece of code for this module is now under construction!



                             % The main marples loop

marples([],Gs,Types,Us,true,[],Us).

marples([X|Xs],Gs,Types,Us,( E & Es ),[X|Xs1],FinalUs)
        :- flag(ccflag,_,off),
                trace('\nI am now trying to solve for %t ',[X],2),
                trace('without introducing any unknowns.\n',2),
           choose_and_apply_strategy(X,Types,E,U,Us),
                tell1(E,U),
           analyse_result(X,E,Xs,Gs,NewQs,NewXs,NewGs),
                check_new(NewQs),
                tell2(X,NewXs,NewGs),
                accept(1),
           !,
           marples(NewXs,NewGs,Types,[U|Us],Es,Xs1,FinalUs).


marples([X|Xs],Gs,Types,Us,(E & Es),[X|Xs1],FinalUs)
        :- flag(ccflag,_,on),
                trace('\nNo luck - I will now accept unknowns ',2),
                trace('in solving for %t.\n',[X],2),
           choose_and_apply_strategy(X,Types,E,U,Us),
                tell1(E,U),
           analyse_result(X,E,Xs,Gs,NewQs,NewXs,NewGs),
                show_new(NewQs),
                tell2(X,NewXs,NewGs),
                accept(2),
           set_types(NewQs,Types,Ntypes),

           marples(NewXs,NewGs,Ntypes,[U|Us],Es,Xs1,FinalUs).


marples([X|Xs],Gs,Types,Us,Es,Xs1,FinalUs)
        :- trace('\nI am unable to solve for %t.\n',[X],2),
           fail.



                             % Check that no new quantities were introduced

check_new([]) :-
```

```
        !,
        trace('  This introduces no new unknowns.\n',3).

check_new(NewQs) :-
        trace('  This introduces unknowns %t which is not allowed.\n',
                                                                  [NewQs],3),
        fail.


                          % Just show the new things

show_new(NewQs) :-
        trace('  This introduces %t as new unknowns.\n',[NewQs],3).



                          % Various messages
                          %  Equation labels are for messages only at the moment
                          %  they should be first class entities!

tell1(E,U)
      :- flag(eqlabel,N,N),
         N1 is N+1,
         flag(eqlabel,_,N1),
         ( trace('\n Equation-%t : %t\n formed by applying : %t\n',
                                                          [N1,E,U],2)
         ; trace('\n Equation-%t rejected.\n\n',[N1],2), fail
         ).



tell2(X,Xs,Gs)
      :- ( trace('\n  New state:     Sought: %t',[Xs],3),
           trace('\n                 Givens:  %t\n',[Gs],3)

           ; trace('\nI will go back to solve for %t again\n',[X],2), fail
         ).



                          % Talk to user for a while

accept(N)
      :- flag(accept,on,on),
         !,
         accmess(N,M),
         writef('\n  %t  Do you accept this equation (yes/no)?\n\n',[M]),
         do_accept.

accept(_) :- !.


accmess(1,'[ No unknowns ]     ') :- !.

accmess(2,'[ Unknowns allowed ]') :- !.


                          % A slightly better accept interface
                          %  Needs improving and interfacing with RUN.PL etc.
```

```
do_accept :-
        prompt(Old,'       (accept) >> '),
        repeat,
          read(X),
          do_acc(X,Cont),
        !,
        prompt(_,Old),
        Cont = yes.


do_acc(V,_) :- var(V), !, fail.

do_acc(yes,yes) :- !.

do_acc(no,no) :- !.

do_acc(Goal,_) :- call(Goal), !, fail.
```

```
/* QUANTS. : Handle soughts, givens, intermediates etc.

                                                    Lawrence
                                                    Updated: 11 December 81
*/


analyse_result(Sought,Eqn,Soughts,Givens,NewQs,NewSoughts,NewGivens) :-
        wordsin(Eqn,Quantities),
        check_solvesfor(Sought,Quantities,Termsof),
            trace('\n  Prior state:   Soughts: %t',[[Sought|Soughts]],3),
            trace('\n                 Givens:  %t\n',[Givens],3),
            trace('\n  This equation solves for %t in terms of %t\n',
                                  [Sought,Termsof],3),
        intersect(Termsof,Givens,AlreadyDone),
        tak_givens(Termsof,AlreadyDone,InterMeds),
        intersect(InterMeds,Soughts,PossEliminated),
        elim_filter(PossEliminated,Eliminated),
        tak_elims(InterMeds,Eliminated,NewQs;Soughts,NewSoughts),
        append([Sought|Eliminated],Givens,NewGivens),
        !.



                          % Check that Sought occurs in Eqn's Qauntities

check_solvesfor(Sought,Quantities,Termsof) :-
        select(Sought,Quantities,Termsof),
        !.

check_solvesfor(Sought,_,_) :-
        trace('   Very strange - %t does not occur in Equation\n',[Sought],2),
        fail.



                          % Take account of of those already given

tak_givens(Termsof,[],Termsof) :- !.

tak_givens(Termsof,AlreadyDone,InterMeds) :-
        subtract(Termsof,AlreadyDone,InterMeds),
            trace('  %t are already solved-for or given.\n',[AlreadyDone],3).



                          % We cannot eliminate Qs that are sought
                          %  so filter them out.

elim_filter([],[]).

elim_filter([X|Rest],Result) :-
        known( sought(X) ),
        !,
        elim_filter(Rest,Result).

elim_filter([X|Rest],[X|Result]) :-
        elim_filter(Rest,Result).
```

```
                          % Take account of those eliminated

tak_elims(InterMeds,[],InterMeds,Soughts,NewSoughts) :-
        !,
        union(Soughts,InterMeds,NewSoughts).

tak_elims(InterMeds,Eliminated,NewQs,Soughts,NewSoughts) :-
        subtract(InterMeds,Eliminated,NewQs),
        subtract(Soughts,Eliminated,X),
        union(X,NewQs,NewSoughts),
          trace('  %t can be eliminated and doesn''t need to be solved for.\n',
                                  [Eliminated],3).
```

```
/* CHOOSE : Simple problem solving steps - solving for single quantities

                                                  Lawrence
                                                  Updated: 18 December 81
*/


 % To solve for a quantity it is necessary to relate it to other quantities.
 % Various general strategies may exist for trying to do this.
 % Each strategy will involve some general rule - a formula.
 % Applying a strategy (formula) relates the quantity to specific
 %       other quantities.
 % This specific relation can be expressed (mathematically) as an equation.
 %
 % The important things are, of course, the strategies, the general rules
 %       and the specific relations produced. The fact that we produce
 %       "equations" is not of major significance in the problem solving.



                          % Choose a strategy and apply it
                          %  We are given:
                          %       Q        - Quantity to solve for
                          %       Types    - Set of types of all known quantities
                          %       Used     - Set of already applied strategies
                          %  We must return:
                          %       Strategy - A successful strategy
                          %       Eqn       - Set of quantities related by
                          %                   applying the strategy (expressed
                          %                   as an equation).
                          % Currently this code relies on Prolog backtracking
                          %  to search through all possible strategies (if
                          %  required to).
                          % Note that there is effectively an extra argument -
                          %  the ccflag value which decides whether or not
                          %  we can create during inference. This should be
                          %  made explicit.

                                  % We know a specific relation

oose_and_apply_strategy(Q,_,Eqn,strategy(specific,Eqname),Used)
     :- know( specific_relates(Eqname,Symbols) ),
        memberchk(Q,Symbols),
        indep(strategy(specific,Eqname),Used),
        know( specific_equation(Eqname,Eqn) ),
            trace(' Using specific equation %t\n',[Eqname],3).

                                  % We must use a general strategy

choose_and_apply_strategy(Q,Types,Eqn,Strategy,Used)
     :- definition(Q,Qtype,Defn),
        applicable_formulae(Qtype,Types,Formulae_list),

        member(Formula,Formulae_list),
            trace(' (try  %t)\n',[Formula],3),

        plan(Formula,Q,Qtype,Defn,Strategy),
        indep(Strategy,Used),
            trace(' Trying to apply %t\n',[Strategy],3),
          apply_strategy(Strategy,Eqn).



                          % Plan - given some general strategy, plan how
                          %  to actualise it, ie produce a complete strategy
                          %  by deciding on a situation in which it can be
                          %  applied. (Use 'prepare' rules)

plan(Formula,Q,Qtype,Defn,strategy(Formula,Situation))
       :- prepare(Formula,Q,Qtype,Defn,Situation).



                          % Strategy is independent of previously applied
                          %  strategies.
                          %  (The useless info should be distributed - it
                          %   belongs with the formulae).

indep(Strategy,Used)
       :- not member(Strategy,Used),
          not useless(Strategy,Used),
          !.


useless(strategy(moments,situation(Point,Rod,Set,Dir,Rtdir,Time)),Used)
       :- twoin(strategy(moments,situation(Pt1,Rod,_,_,_,Time)),
                strategy(moments,situation(Pt2,Rod,_,_,_,Time)), Used).

useless(strategy(resolve,situation(Type,P,Set,Dir,Time)),Used)
       :- twoin(strategy(resolve,situation(_,P,_,X,Period)),
                strategy(resolve,situation(_,P,_,Y,Period)), Used).

useless(strategy(relvel,situation(Objs,Time)),Used)
       :- member(strategy(relvel,situation(X,Time)),Used),
          seteq(Objs,X).

useless(strategy(relaccel,situation(Objs,Time)),Used)
       :- member(strategy(relaccel,situation(X,Time)),Used),
          seteq(Objs,X).

useless(strategy(constaccel-N,Situation),Used)
       :- twoin(strategy(constaccel-X,Situation),
                strategy(constaccel-Y,Situation), Used).



                          % Apply a strategy

apply_strategy(strategy(Formula,Situation),Eqn)
       :- already_applied(Formula,Situation,Eqn),
          !.

apply_strategy(strategy(Formula,Situation),Eqn)
       :- isform(Formula,Situation,Eqn),
          cassertz(already_applied(Formula,Situation,Eqn)),
          !.
```

```
/* APPLIC : Generation of applicable formulae

                                                  Lawrence
                                                  Updated: 4 December 81
*/

   :- public      applicable_formulae/3.


   :- mode        applicable_formulae(+,+,?),
                  appl(+,-),
                  formul_sort(+,-),
                  pref(+,-),
                  filter(+,+,?),
                  filter(+,+,?,?,?,?),
                  okfor(+,+),
                  qsort(+,?,?),
                  partition(+,+,-,-),
                  before(+,+).


                          % Flist is a list of formulae that relate quantities
                          %  of type Qtype to other quantities.
                          %  This list is sorted; given information about
                          %  general preferences and quantity types known in
                          %  this problem (Types).

applicable_formulae(Qtype,Types,Flist)
     :- findall(X, appl(Qtype,X), List1),

        formul_sort(List1,List2),                 % sort on general preferences
        filter(List2,Types,Flist),                % split on known types

            trace('\n Applicable formulae : %t\n',[Flist],3),
        !.



                          % An applicable formula is one which relates Qtype
                          %  and others.

appl(Qtype,Formula)
     :- relates(Formula,Types),
        memberchk(Qtype,Types).



                          % Sort using value of pref

formul_sort(L1,L2) :- qsort(L1,[],L2), !.


        pref(Formula,N) :- preference(Formula,N), !.

        pref(Formula,1).                  % Don't cares are best (hohum?)


                          % Filter list, flag permitting
                          %  This splits the formula list into two:
                              %    1) Those formulae which only relate known
                              %       quantity types.
                              %    2) Those formulae which relate types which
                              %       are not currently known.
                              %  The new list is formed by appending these two
                              %  lists (i then 2). Within the sub-lists the original
                              %  ordering is preserved.
                              % The implementation uses difference-pairs.

filter(L,_,L) :- flag(filter,out,out), !.

filter(L1,Types,L2) :- filter(L1,Types,L2,Z,Z,[]), !.


filter([],_,Z1,Z1,Z2,Z2).

filter([First|Rest],Types,[First|Bests],Z1,Worsts,Z2)
      :- okfor(First,Types),
         !,
         filter(Rest,Types,Bests,Z1,Worsts,Z2).

  lter([First|Rest],Types,Bests,Z1,[First|Worsts],Z2)
      :- filter(Rest,Types,Bests,Z1,Worsts,Z2).


okfor(Formula,Types)
      :- relates(Formula,Ftypes),
         subset(Ftypes,Types),
         !.



/*----------------------------------------------------------------------*/


                              % Quick sort a list.
                              %  The ordering criteria (before) uses pref/2
                              %  defined above.

  ort([X|L],R0,R)
      :- partition(L,X,L1,L2),
         qsort(L2,R0,R1),
         qsort(L1,[X|R1],R).

qsort([],R,R).


partition([F|L],X,[F|L1],L2)
      :- before(F,X),
         !,
         partition(L,X,L1,L2).

partition([F|L],X,L1,[F|L2])
      :- partition(L,X,L1,L2).

partition([],_,[],[]).



before(X,Y)
```

```
:- pref(X,N1),
   pref(Y,N2),
   N1 < N2,
   !.
```

```
/* FMS.PL : Gizmo for loading required formulae

                                                            Lawrence
                                                            Updated: 6 September 81
*/

 % This code expects formulae(List) to return a list of formulae FILES.
 % All code for the predicates involving formulae is abolished and
 % all the files are then consulted. To work properly these files must
 % only contain formula predicates. This method of reloading things is
 % not particularly elegant! The problem is having the formulae read in
 % as Prolog clauses while having them spread across several files. They
 % should be handled the way the predicate library handles predicates
 % (indeed it should all be integrated).



                                % Reload all the formulae definitions

fms   :- formulae(FileList),
         load_fms(FileList).



                                % Load formulae from FileList

load_fms(FileList)
      :- zap_fms,
         ttynl, display('Loading formulae:'), ttynl,
         call( FileList ),                              % ie consult them!
         ttynl, display('Formulae loaded'), ttynl,
         fail.

load_fms(_).



                                % Abolish ALL old formulae

zap_fms
      :- formula_predicate( Name/Arity ),
         abolish(Name,Arity),
         fail.

zap_fms.



                                % These are the predicates which count
                                %  All clauses for these will be abolished before
                                %  the new files are consulted.

formula_predicate( relates/2    ).
formula_predicate( preference/2 ).
formula_predicate( prepare/5    ).
formula_predicate( isform/3     ).
```

```
/* INPUT.PL : Loading facts

                                                      Lawrence
                                                      Updated: 1 December 81
*/

   :- public     input/1.


   :- mode       input(?),
                 intprt(?),
                 intprt2(+).



                          % Fact loading loop

input(F) :-
        ( file_exists(F), File = F   ;  concat(F,'.prb',File) ),
        !,
        seeing(Old),
        see(File),
        repeat,
            read(X),
            intprt(X),
        !,
        seen,
        see(Old),
        writef('\nFacts read into data base from %t\n\n',[F]).


                          % Interpret an entry

intprt( end_of_file ).

intprt(X) :- intprt2(X), !, fail.


intprt2( :-(X) ) :- ( call(X)   ;   true ), !.

intprt2( (A,B) ) :- !, intprt2(A), intprt2(B).

intprt2( Fact ) :- db_assert(Fact).
```

```
/* DB.PL : Asserting into (object level) database

                                                    Lawrence
                                                    Updated: 18 December 81
*/

   :- public      db_assert/1,
                  definition/3.


  :- mode         db_assert(+),
                  handle_type_pred(+),
                  handle_types(+),
                  h_typs(+,+,+),
                  h_typ(+,+,+),
                  handle_definitions(+,+),
                  not_a_definition(+),
                  h_defs(+,+,+,+),
                  no_defn(+),
                  definition(?,?,?).


                          % Assert a fact into database
                          %  Also willing to accept conjunctions.

db_assert(V) :-
        var(V),
        !,
        error('Attempt to db_assert a variable: %t',[V],continue).

db_assert(A&B) :-
        !,
        db_assert(A),
        db_assert(B).

db_assert(Fact) :-
        type_predicate(Fact),
        !,
        handle_type_pred(Fact).

db_assert(Fact) :-
        known(Fact),
        !,
        error('db_assert of duplicate fact: %t',[Fact],continue).

db_assert(Fact) :-
        normal_form(Fact,NForm),
        !,
            trace(')) Normal forming: %t\n',[Fact],db),
        db_normal_form(NForm).

db_assert(Fact) :-
        db_add(Fact,Fname),
        handle_types(Fact),
        handle_definitions(Fact,Fname),
            trace(')) DB assert: %t\n',[Fact],db).
```

```
                          % Special handling for type predicates

handle_type_pred(Fact) :-
        functor(Fact,Type,1),
        arg(1,Fact,Obj),
        nonvar(Obj),
        add_type_info(Type,Obj),
        !.

handle_type_pred(Fact) :-
        error('Invalid db_assert of %t',[Fact],continue).



                          % Types.
                          %  Add the new type information implied by this fact.

handle_types(Fact) :-
        argument_types(Fact,Types),
        !,
        functor(Fact,_,Arity),
        h_typs(Arity,Fact,Types).

handle_types(Fact) :- unusual(Fact), !.         % Special case for sought etc.

handle_types(Fact) :-
        error('No type rule when db_asserting: %t',[Fact],continue).


h_typs(0,_,_) :- !.

h_typs(N,Fact,Types) :-
        arg(N,Fact,Obj),
        arg(N,Types,Type),
        h_typ(Type,Obj,Fact),
        N1 is N-1,
        h_typs(N1,Fact,Types).


  typ(Type,Obj,Fact) :- add_type_info(Type,Obj), !.

h_typ(Type,Obj,Fact) :-
        error('Type failure (%w) of %w in %t',[Type,Obj,Fact],continue).



                          % Definitions
                          %  Each quantity occuring in the problem has a
                          %  definition: this is the first assertion that
                          %  introduced it.
                          % There ought to be a condition here that the predicate
                          %  is appropriate. Ie: it should be a quantity defining
                          %  predicate, not some rubbish like 'measure'.
                          %  This requires better meta info about classes of
                          %  predicates. Currently there is a special case hack.

handle_definitions(Fact,_) :-
        not_a_definition(Fact),
        !.
```

```
                not_a_definition( measure(_,_,_) ).      % hack

handle_definitions(Fact,Fname) :-
        argument_types(Fact,Types),
        !,
        functor(Fact,_,Arity),
        h_defs(Arity,Types,Fact,Fname).

handle_definitions(_,_).


h_defs(0,_,_,_) :- !.

h_defs(N,Types,Fact,Fname) :-
        arg(N,Types,Type),
        super_type(quantity,Type),
        arg(N,Fact,Q),
        no_defn(Q),
        !,
        db_add( defn(Q,Type,Fname) ),
        N1 is N-1,
        h_defs(N1,Types,Fact,Fname).

h_defs(N,Types,Fact,Fname) :-
        N1 is N-1,
        h_defs(N1,Types,Fact,Fname).



no_defn(Q) :- known(defn(Q,_,_)), !, fail.

no_defn(Q).



                        % How to retrieve a definition

definition(Q,Qtype,Defn) :-
        known( defn(Q,Qtype,Fname) ),
        known( Defn, Fname ).



                        % Normal forms

db_normal_form(context(Context,Conseq)) :-
        !,
        cc Context,
        db_assert(Conseq).

db_normal_form(Conseq) :- db_assert(Conseq).
```

```
/* KNOWN.PL : Indexed database

                                                    Lawrence
                                                    Updated: 18 December 81
*/

 /* EXPORT */

  :- public      db_init/0,
                 generate_fact_names/1,
                 db_dump/1,
                 db_dump/0,
                 db_show/0,
                 known/1,
                 known/2,
                 db_add/1,
                 db_add/2,
                 db_forget/1,
                 db_forget/2,
                 db_state/1,
                 db_restore/1,
                 forget_fact/1.


/* IMPORT */
/*
                 must_be_term/2                     from  POLICE
                 must_be_ground/2                   from  POLICE

                 index/2                            from  INDEX
*/


 /* MODES */

  :- mode        db_init,
                 next_fact_name(-),
                 cons_fact_name(+,-),
                 generate_fact_names(?),
                 gen_fnames(+,+,?),
                 db_dump(+),
                 db_dump,
                 db_show,
                 known(+),
                 known(+,?),
                 known2(+,?),
                 db_add(+),
                 db_add(+,?),
                 db_forget(+),
                 db_forget(+,?),
                 db_state(?),
                 db_restore(+),
                 db_scrub(+,+),
                 forget_fact(+),
                 get_a_key(+,?,?,?),
                 gak(+,+,+,?,?,?).



                           % Initialise database
```

```
                        %  This must be called to set things up
                        % Currently involves:
                        %       Set Fact counter to 0
                        % This has now been extended so that it can be used
                        %  to reinitialise the database at any time.

db_init :-
        recorded('$fact','$fact'(_),_),          % already under way
        !,
        db_restore(db_state(0)).                 % reinitialise

db_init
     :- recorda('$fact','$fact'(0),_).           % Startup initialisation



                        % Return the next (new) FactName
                        %  This is currently a new gensymed atom

next_fact_name(FactName)
     :- recorded('$fact','$fact'(N),Ref),
        erase(Ref),
        NewN is N+1,
        recorda('$fact','$fact'(NewN),_),
        cons_fact_name(NewN,FactName).



                        % Cons a facts name given a number
                        %  Currently produces an atom composed from "fact"
                        %  and the number.

cons_fact_name(N,FactName)
     :- name(N,Number),
        name(FactName,[102,97,99,116|Number]).          % "fact"



                        % Generate all current facts names
                        %  Ie from 0 to current. Note that facts may have been
                        %  removed.

generate_fact_names(FactName) :-
        recorded('$fact','$fact'(Current),_),
        gen_fnames(0,Current,FactName).


gen_fnames(N,N,_) :- !, fail.

gen_fnames(N,Max,FactName) :-
        cons_fact_name(N,FactName).

gen_fnames(N,Max,FactName) :-
        N1 is N+1,
        gen_fnames(N1,Max,FactName).


                        % Dump the database (onto terminal or file)
                        %  The format allows the dump to be read back into
```

```
                              %  an image using input(File)

db_dump(File) :-
        open(Old,File),
        db_dump,
        close(File,Old).


db_dump :-
        generate_fact_names(FactName),
        recorded(FactName,Fact,_),
        writef('%t.\n',[Fact]),
        fail.

db_dump.



                              % Variant of dump for looking at (rather than something
                              %  that can be input).

 _show :-
        generate_fact_names(FactName),
        recorded(FactName,Fact,_),
        writef('\t%w\t%t\n',[FactName,Fact]),
        fail.

db_show.



                              % Retrieve a fact
                              %  We do this by using the first valid key
                              %  This will be some instantiated argument, failing
                              %  this we the final key will always be the functor
                              %  of the fact.

known(Fact) :- known2(Fact,_).


 own(Fact,Fname) :-
        atom(Fname),
        !,
        recorded(Fname,Fact,_).

known(Fact,Fname) :- known2(Fact,Fname).


known2(Fact,Fname)
      :- must_be_term(Fact,known),
        get_a_key(Fact,Key,Ktag,FactName),
        !,
        recorded(Key,Ktag,_),
        recorded(FactName,Fact,_),
        Fname = FactName.


                              % Add a fact to the database
```

```
db_add(Fact) :- db_add(Fact,_).


db_add(Fact,Fname)
      :- must_be_term(Fact,db_add),
         must_be_ground(Fact,db_add),
         next_fact_name(FactName),
         recorda(FactName,Fact,_),
         db_add_keys(FactName,Fact),
         Fname = FactName.



                              % Add all the links from keys to the fact
                              %  (backtrack through all keys)
                              % Note that all the arguments must be ground so
                              %  ALL possible keys will be used (this is
                              %  important given known/2's use of only one key)

db_add_keys(FactName,Fact)
      :- get_a_key(Fact,Key,Ktag,FactName),
         recorda(Key,Ktag,_),
         fail.

db_add_keys(_,_).



                              % Remove a fact from the database

db_forget(Fact) :- db_forget(Fact,_).


db_forget(Fact,Fname)
      :- known(Fact,FactName),
         forget_fact(FactName),
         Fname = FactName.



                              % Return the current state of the database

db_state(db_state(N)) :- recorded('$fact','$fact'(N),_).



                              % Restore the database to some previous state
                              %  Wipe out all new facts since that state and
                              %  reset the fact counter. (Note that this may leave
                              %  dangling names held elsewhere in the program.
                              %  BE CAREFUL (one could leave the fact counter alone
                              %  and just wipe out facts?))

db_restore(V)
      :- var(V),
         !,
         error('db_restore given variable : %w',[V],break),
         fail.

db_restore(db_state(N))
```

```
    :- integer(N),
       N >= 0,
       recorded('$fact','$fact'(Current),Ref),
       N =< Current,
       !,
       db_scrub(Current,N),
       erase(Ref),
       recorda('$fact','$fact'(N),_).

db_restore(X)
    :- error('Attempt to restore bad db state: %t',[X],break),
       fail.



                              % Actually throw away the facts (for a restore)

db_scrub(N,N) :- !.

db_scrub(N,Final)
    :- cons_fact_name(N,FactName),
       forget_fact(FactName),
       Next is N-1,
       db_scrub(Next,Final).



                              % Forget a fact and remove all key links

forget_fact(FactName)
    :- recorded(FactName,Fact,Ref),
           erase(Ref),
       get_a_key(Fact,Key,Ktag,FactName),
       recorded(Key,Ktag,Kref),
           erase(Kref),
       fail.

forget_fact(_).


   Keys %%


                              % Fact can be keyed under Key with link
                              %  Ktag involving FactName.
                              %       Fact     - the fact (or incoming goal)
                              %       Key      - currently an atom (some argument)
                              %       Ktag     - what hangs off Key
                              %       FactName - Some subpart of Ktag which will be
                              %                  what Fact hangs off
                              %  This is non-determinate. If backtracked through it
                              %  will produce all possible keys.
                              % NB it is intended that the order of generation will
                              %  be roughly the order of utility. (This will depend
                              %  on index/2). Keys are only valid (returned) if they
                              %  are instantiated. When used for adding this will be
                              %  true of all arguments to Fact. When used for
                              %  retrieving (partial) facts this will restrict the
                              %  set of keys returned.
                              % The functor of the Fact is itself returned as a
```

```
                        %  final last ditch key. (NB This is done by returning
                        %  the whole fact - its functor will thus be the key).
                        % For argument keys the Ktag includes the Fact's
                        %  predicate. This will filter out links to other
                        %  predicates early on in the retrieval process (see
                        %  known/2).

set_a_key(Fact,Key,Ktag,FactName)
     :- index(Fact,KeyList),
        functor(Fact,Pred,_),
        sak(KeyList,Fact,Pred,Key,Ktag,FactName).


sak([],Fact,_,Fact,fact(FactName),FactName).

sak([Key|_],_,Pred,Key,fact(Pred,FactName),FactName) :- nonvar(Key).

sak([_|Rest],Fact,Pred,Key,Ktag,FactName)
     :- sak(Rest,Fact,Pred,Key,Ktag,FactName).
```

```
/* CC.PL : Interface into Inference engine

                                                          Lawrence
                                                          Updated: 18 December 81
*/

   %%% This file should be interpreted %%%

 % The names of these procedures are historical (see Alan Bundy's "Will it
 % reach to top", AI Journal).


                           % We satisfy the Goal(s) by trying to prove them
                           %  using the Mecho inference engine. The different
                           %  interfaces request varying degrees of effort.
                           %  For what the effort entails see PROVEO.

dc Goals  :- prove(Goals,easy).

ncc Goals :- prove(Goals,general).

[illegible] Goals  :- prove(Goals,hard).

pc Goals  :- prove(Goals,general).      % Was once supposed to be more powerful
                                        %  Ie defaults/prediction.
```

```
/* PROVE0.PL :  [ Stage 0 ]  Mecho Theorem Prover

                                                    Lawrence
                                                    Updated: 18 December 81
*/

 % This file requires the file METHOD.PL to define all the proof methods
 % used here. The code here can be seen as a meta level axiomatisation
 % over goals, properties of goals and predicates, proof plans and methods.

  :- public     prove/2,
                prove_subgoals/2,
                prove/3.


  :- mode       prove(+,+),
                prove_subgoals(+,+),
                prove(+,+,+),
                prove_single(+,+,+),

                plan_proof(+,+,+,-,-,-),
                size_up_task(+,+,-),
                make_plan(+,+,-),
                    filter_plan(+,+,-),
                        compound_plan(?,?,?,?),
                        combine_plans(+,+,+,?),
                consider_pruning(+,-),

                run_proof(+,+,?,+),
                postmortem(+,+,-),
                    proof_start(+),
                    proof_exec(+,+,+).


                            %% Theorem prover - Interfaces

                            % Top level - from cc, ncc, pc, dc etc.

prove(Goals,Effort) :-
        prove(Goals,Effort,[]),
            trace('>> YES : %t\n',[Goals],infer).


                            % Internal, used to continue proof (recursively)

prove_subgoals(Goals,INFO) :-
        ps_effort(INFO, Effort),
        ps_history(INFO, History),
        prove(Goals, Effort, History).


                            % Prove some goals with a certain amount of effort

prove(V,_,_) :-
        var(V),
        !,
        error('Goal to prove is a variable: %w',[V],fail).
```

```
prove(A & B, Effort, History)
      :- !,
         prove(A,Effort,History),
         prove(B,Effort,History).

prove(context(Context,Subgoals),Effort,History)    % Only occurs in normal forms
      :- !,
         prove(Context,hard,History),              % ho hum?
         prove(Subgoals,Effort,History).

prove(SingleGoal,Effort,History) :-
             trace('>> Trying to prove (%w):  %t\n',[Effort,SingleGoal],prove),
         consider_history(SingleGoal,History,Future),
         prove_single(SingleGoal,Effort,Future).



                           % Proving non-compound goals
                           %  Includes various "escapes"

  ove_single(true,_,_) :- !.
  ove_single({X},_,_) :- !, call(X).
prove_single(X < Y,_,_) :- !, X < Y.
prove_single(X =< Y,_,_) :- !, X =< Y.
prove_single(X > Y,_,_) :- !, X > Y.
prove_single(X >= Y,_,_) :- !, X >= Y.
prove_single(either(W,X,Y,Z),_,_) :- !, perm2(W,X,Y,Z).
prove_single(eval(X),_,_) :- !, eval(X).
prove_single(eval(X,Y),_,_) :- !, eval(X,Y).
prove_single(tidy(X,Y),_,_) :- !, tidy(X,Y).
prove_single(succ(X,Y),_,_) :- !, succ(X,Y).
prove_single(pred(X,Y),_,_) :- !, succ(Y,X).
prove_single(type(X,Y),_,_) :- !, type(X,Y).          % lax meta-level use

prove_single(TypePred,_,_)                              % normal object-level
      :- type_predicate(TypePred,Type,Arg),
         !,
         type(Type,Arg).

  ove_single(Goal,Effort,History)
      :- plan_proof(Goal,Effort,History, Result,Prune,INFO),
         run_proof(Prune,Goal,Result,INFO).
```

```
%% Plan the proof  (prove0.pl) %%

                              % Plan a proof strategy

plan_proof(Goal,Effort,History, Result,Prune,INFO)
     :- size_up_task(Goal,Effort,NewEffort),
        make_plan(Goal,NewEffort,ProofPlan),
        consider_pruning(Goal,Prune),
        ps_cons(ProofPlan,NewEffort,History,Goal,Result,Prune,INFO).



                              % Decide if we can apply harder methods than usual
                              %  because Goal is easier than usual. This depends
                              %  on some meta_knowledge about the predicate involved

size_up_task(Goal,easy,general)
     :- easy_inference(Goal),
        !.

  e_up_task(_,Effort,Effort).



                              % Make a plan
                              %  We zip through normal form rewrites without doing
                              %  anything else, regardless of the Effort.
                              %  The usual case involves trying some standard plan.

make_plan(Goal,_,nform)
     :- derived(Goal),
        !.

make_plan(Goal,Effort,Plan)
     :- standard_plan(Effort,Standard),
        filter_plan(Standard,Goal,Plan).



                              % Filter a plan by
                              %       Checking each step for applicability. This
                              %       may involve turning general methods into
                              %       specific methods (ie instantiating them).

filter_plan(Plan,Goal,NewPlan)
     :- compound_plan(Plan,X,Y,PlanOp),
        !,
        filter_plan(X,Goal,NewX),
        filter_plan(Y,Goal,NewY),
        combine_plans(NewX,NewY,PlanOp,NewPlan).

filter_plan(Method,Goal,Method)
     :- method_applicable(Method,Goal),
        !.

filter_plan(_,_,empty).



                              % Types of compound plan
```

```
                            %  Note that they are are all built with binary
                            %  PlanOps, this is important in the code above.

compound_plan( \\(X,Y), X,Y,\\).

compound_plan( +(X,Y),  X,Y,+).



                            % Combine two parts of a plan - simplify out
                            %  occurances of 'empty', which is assumed to
                            %  be the identity element for all PlanOps.

combine_plans(empty,Y,_,Y) :- !.

combine_plans(X,empty,_,X) :- !.

combine_plans(X,Y,PlanOp,Plan) :- compound_plan(Plan,X,Y,PlanOp), !.


                            % Decide if the proof can be pruned or not.
                            %  Currently either all solutions are thrown away
                            %  after the first one (because of uniqueness) or
                            %  they are all let through.

consider_pruning(Goal,one) :- unique(Goal), !.

consider_pruning(Goal,all). % not unique(Goal)
```

```
%% Run the the proof  (prove0.pl) %%

                           % Attempt the planned proof
                           %  and decide how to react afterwards.
                           %  We convert a succeed/fail result into a Prolog
                           %  success/failure action. This is because the
                           %  current meta-level is Prolog code and thus expects
                           %  these Prolog level responses.

run_proof(one,Goal,Result,INFO)
     :- proof_start(INFO),
        !,                              % Prune choices here!
        postmortem(Result,Goal,Action),
        Action = succeed.

run_proof(all,Goal,Result,INFO)
     :- proof_start(INFO),
        postmortem(Result,Goal,Action),
        ( Action = fail, !, fail  ;  true ).



                           % See what happened
                           %  We are given the name of the method, which worked,
                           %  or 'stop' if we ran out of methods.

postmortem(stop,Goal,fail)
     :- !,
        trace('>> unknown : %t\n',[Goal],prove).

postmortem(Method,Goal,fail)
     :- establishes_falsity(Method),
        !,
        trace('>> false : %t\n',[Goal],prove).

postmortem(Method,Goal,succeed)
     :- % establishes_truth(Method),
        trace('>> true  : %t\n',[Goal],prove).



                           % Start up

proof_start(INFO) :-
        ps_dest(INFO,Plan,Goal),
        proof_exec(Plan,Goal,INFO).

proof_start(INFO) :-
        ps_result(INFO,stop).           % ran out of methods



                           % Execute a proof plan

proof_exec( (PlanA\\PlanB), Goal,INFO) :-
        proof_exec(PlanA,Goal,INFO),                                  ?
        !.

proof_exec( (PlanA\\PlanB), Goal,INFO) :-
        proof_exec(PlanB,Goal,INFO).
proof_exec( (PlanA+PlanB), Goal,INFO) :-
        proof_exec(PlanA,Goal,INFO).

proof_exec( (PlanA+PlanB), Goal,INFO) :-
        proof_exec(PlanB,Goal,INFO).

proof_exec(Method,Goal,INFO) :-
        proof_method(Method,Goal,INFO),
        ps_result(INFO,Method).
```

```
/* METHOD.PL : Particular proof methods

                                                  Lawrence
                                                  Updated: 18 December 81
*/

 % The file PROVEO axiomatises an inference system that uses particular
 % proof methods which are defined here. This file can be added to to
 % increase the number of methods available to the inference system.


  :- public     standard_plan/2,
                method_applicable/2,
                establishes_truth/1,
                establishes_falsity/1,
                proof_method/3.


  :- mode       standard_plan(?,?),
                method_applicable(+,+),
                establishes_truth(?),
                establishes_falsity(?),
                proof_method(+,+,+).


                          % The general form of a standard plan
                          %  This shows for certain amounts of effort, what proof
                          %  methods are appropriate.
                          % Plan Operators as follows:
                          %       +    Inclusive OR
                          %       \\   Exclusive OR

standard_plan(easy, known ).

standard_plan(general, known+silly(_)+inference ).

standard_plan(hard, (known+silly(_)+inference \\ default+create(_)) ).


                          % Check to see if a method is applicable to the
                          %  given goal.

method_applicable(known,_).

method_applicable(silly(How),Goal) :- unique(Goal,How), pure(Goal).

method_applicable(inference,_).

method_applicable(default,_).

method_applicable(create(How),Goal) :- flag(ccflag,on,on), exists(Goal,How).


                          % What the various methods establish

establishes_truth(known).
establishes_truth(inference).
establishes_truth(default).
establishes_truth(create(_)).

establishes_falsity(silly(_)).



                              % How to perform particular proof methods
                              %  We are given the method name, the Goal to try it
                              %  on, and the current proof INFO structure.

proof_method(known,Goal,_) :- known(Goal).


proof_method(nform,Goal,INFO) :-
        normal_form(Goal,Subgoals),
        prove_subgoals(Subgoals,INFO).


proof_method(silly(How),Goal,INFO) :-
        how_dest(How,Args,_),
        copy_args(Args,Goal,TestGoal),
        prove(TestGoal,easy,[]),                  % Don't try too hard
        Goal \== TestGoal.


proof_method(inference,Goal,INFO) :-
        object_level_rule(Goal,Subgoals),
        prove_subgoals(Subgoals,INFO).


proof_method(default,Goal,INFO) :-
        default_rule(Goal,Subgoals),
        prove_subgoals(Subgoals,INFO).


proof_method(create(How),Goal,INFO) :-
        how_dest(How,_,Vals),
        argument_names(Goal,Names),
        gensym_args(Vals,Names,Goal),
        db_assert(Goal).
```

```
/* PSO.PL :  [ Stage 0 ]  Proof structure operations

                                                  Lawrence
                                                  Updated: 1 December 81
*/

   :- public      ps_cons/7,
                  ps_dest/3,
                  ps_plan/2,
                  ps_effort/2,
                  ps_history/2,
                  ps_goal/2,
                  ps_result/2.


   :- mode        ps_cons(+,+,+,+,+,+,?),
                  ps_dest(+,?,?),
                  ps_plan(+,?),
                  ps_effort(+,?),
                  ps_history(+,?),
                  ps_goal(+,?),
                  ps_result(+,?).



                          % Cons up a proof structure
                          %  (May not use all of the available info)


ps_cons(ProofPlan,Effort,History,Goal,Result,Prune,
                ps(ProofPlan,Effort,History,Goal,Result) ).



                          % Specialised destructor

ps_dest( ps(ProofPlan,Effort,History,Goal,Result), ProofPlan,Goal).



                          % Selection operations

ps_plan( PS, X ) :- arg(1,PS,X).

ps_effort( PS, X ) :- arg(2,PS,X).

ps_history( PS, X ) :- arg(3,PS,X).

ps_goal( PS, X ) :- arg(4,PS,X).

ps_result( PS, X ) :- arg(5,PS,X).
```

```
/* HIST.PL : Operations on histories

                                                                Lawrence
                                                                Updated: 30 November 81
*/


   :- public      consider_history/3.


  :- mode        consider_history(+,+,?).


                                  % Decide whether to prune search.
                                  %  Currently a loop check although the instancing
                                  %  implications are not worked out properly yet
                                  %  (see Lawrence for details).

consider_history(Goal,History,_) :-
          numbervars(Goal,1,N),                         % Force match to be one way
          memberchk(Goal,History),
          !,
              trace('>> Looping ... so fail\n',prove),
          fail.

consider_history(Goal,History,[Goal|History]).
```

```
/* BOUND.PL : Investigating instantiation states etc.

                                                  Lawrence
                                                  Updated: 30 November 81
*/

 /* EXPORT */

   :- public      bound/1,
                  unbound/1,
                  pure/1,
                  allbound/2,
                  allunbound/2,
                  ground/1.


/* MODES */

   :- mode        bound(?),
                  unbound(?),
                  pure(?),
                  allbound(+,?),
                  allunbound(+,?),
                  ground(+),
                  all_ground(+,+).



                          % Object level variable is bound
                          %  Currently means completely ground as well

bound(OLVar) :- ground(OLVar).



                          % Object level variable is unbound
                          %  OL-variable represented as Prolog variable.
                          %  BE CAREFUL

   ound(OLVar) :- var(OLVar).



                          % Goal is pure
                          %  For Mecho pure means completely ground
                          %  (in Chris' Semantic Interpreter it is a bit
                          %   more hairy).

pure(Goal) :- ground(Goal).



                          % All specified arguments are bound
                          %  For this all arguments must be completely
                          %  ground (in Mecho). We are given a "pattern",
                          %  ie a list of argument numbers, to check.

allbound([],_).
allbound([ArgN|Rest],Goal) :-
        arg(ArgN,Goal,Argument),
        ground(Argument),                % unfolded bound(Argument)
        allbound(Rest,Goal).


                        % All arguments are unbound

allunbound([],_).

allunbound([ArgN|Rest],Goal) :-
        arg(ArgN,Goal,Argument),
        var(Argument),                   % unfolded unbound(Argument)
        allunbound(Rest,Goal).
```

```
%% General routine for testing groundness  (bound.pl) %%

                          % A goal is completely ground
                          %  This routine will work for any Prolog structure

ground(V) :- var(V), !, fail.

ground(A) :- atomic(A), !.

ground([HD|TL])                   % speed up + tail recurse to right (tail)
     :- !,
        ground(HD),
        ground(TL).

ground(Term)
     :- functor(Term,F,Arity),
        all_ground(Arity,Term).



all_ground(1,Term)                % Note Arity always >= 1
     :- !,                        % We work right to left across Term and
        arg(1,Term,Arg),          %  tail recurse on the first argument.
        ground(Arg).

all_ground(N,Term)
     :- arg(N,Term,Arg),
        ground(Arg),
        N1 is N-1,
        all_ground(N1,Term).
```

```
/* META2.PL : More meta-level usefuls

                                                    Lawrence
                                                    Updated: 18 December 81
*/

  % PLCODE:PREDS.PL corresponds to META1 and should probably be renamed.


   :- public     derived/1,
                 not_derived/1,
                 gensym_args/3,
                 how_dest/3.


   :- mode       derived(+),
                 not_derived(+),
                 gensym_args(+,+,+),
                 how_dest(+,?,?).


                         % All information concerning a predicate is derived
                         %  Ie: it has a normal form.
                         %  Implementation currently uses double negation hack
                         %  for an undoubtably minimal reason (discard of space
                         %  used by the set).

derived(Pred) :-
        not_derived(Pred),
        !,
        fail.

derived(Pred).


not_derived(Pred) :-
        set(normal_form,Pred,_),
        !,
        fail.

not_derived(Pred).



                         % Gensym up some names for specified argument posns

gensym_args([],_,_).

gensym_args([N|Ns],Names,Goal) :-
        arg(N,Names,Name),
        arg(N,Goal,Arg),
        cgensym(Name,Arg),
        gensym_args(Ns,Names,Goal).


                         % Split up a How structure (as in proof plans etc)

how_dest(how(Args,Vals),Args,Vals).
```

```
/* RUN.PL : Run the inference engine from terminal

                                                     Lawrence
                                                     Updated: 1 December 81
*/

  % WORK IN PROGRESS
 %
  % I Can't remember when I originally wrote this, tis a strange piece of
 % code which I now hardly understand! It would be nice to see it working
 % sometime...

                            % Inference engine terminal top level

run :-
        prompt(Old,Old),
        run(cc),
        prompt(_,Old).


run(Mode) :-
        run_mode(Mode,Prompt),
        prompt(_,Prompt),
        repeat,
            read(Input),
            run_eval(Input,Mode,Cont),
        !,
        run_cont(Cont).


                            % What modes there are (with prompts)

run_mode(dc,  '(dc)  '  ).
run_mode(ncc, '(ncc) ' ).
run_mode(cc,  '(cc)  '  ).
run_mode(pc,  '(pc)  '  ).
run_mode(add, '(add) ' ).


                            % Evaluate input

run_eval(Input,Mode,Cont) :-
        run_eval2(Input,Mode,RMode,Result),
        run_report(Result,Input,Cont0),
        run_chk_cont1(Cont0),
        !,
        run_chk_cont2(Cont0,Mode,RMode,Cont).



                            % Handling continuations
                            %  (Choices of continuing or failing back/round at
                            %   various points)

run_chk_cont1(C) :- C \== retry.
```

```
run_chk_cont2(stop,_,_,stop) :- !.

run_chk_cont2(_,Mode,RMode,RMode) :- Mode \== RMode.


run_cont(stop) :- !.

run_cont(Mode) :- run(Mode).



                         % Perform the actual evaluation

run_eval2(Var,Mode,Mode,error) :-
        var(Var),
        !.

run_eval2(end_of_file,Mode,Mode,stop) :- !.

run_eval2(Mode,_,Mode,nil) :-
        atom(Mode),
        run_mode(Mode,_),
        !.

run_eval2(Atomic,Mode,Mode,error) :-
        atomic(Atomic),
        !.

run_eval2(ModeChange,_,FinalMode,Result) :-
        functor(ModeChange,Mode,1),
        run_mode(Mode,_),
        !,
        arg(1,ModeChange,Goals),
        run_eval2(Goals,Mode,FinalMode,Result).

run_eval2(Goals,Mode,Mode,succeed) :-
        G =.. [Mode,Goals],
        call(G).

    n_eval2(_,Mode,Mode,fail).



                         % What to do at the end

run_report(stop,_,stop).

run_report(nil,_,nil).

run_report(error,Input,nil) :-
        writef('Bad input: %t',[Input]).

run_report(succeed,Input,Choice) :-
        run_format(Input,Format),
        writef(Format,[Input]),
        write('redo? '), ttyflush,
        run_reply(Choice).

run_report(fail,_,nil) :-
        write('Failed'), nl.
        run_format(A&B,'Proved:%c') :- !.
        run_format(_,  'Proved:  %t\n').



                        % Read a response from the user
                        %  "y" for yes, anything else for no.

run_reply(Result) :-
        repeat,
            set0(C),
            run_reply2(C,Result),
        !.


run_reply2(121,retry) :- repeat, set0(C), C =\= 31, !.

run_reply2(31,nil).

[illegible]n_reply2(C,nil) :- C >= 32, repeat, set0(C), C =\= 31, !.



                        % Another (short) name for db_assert

add(Fact) :- db_assert(Fact).
```

```
/* TYPES.PL : Inference mechanism for type predicates

                                                        Chris (now Lawrence)
                                                        Updated: 18 December 81
*/

/*
   Imports:

        type_pattern/2.
*/

:- public
        type/2,
        not_type/2,
        print_types/1,
        super_type/2,
        compatible/2,
        add_type_info/2.

   mode type(+,?),
        not_type(+,+),
        print_types(?),
        super_type(+,+),
        compatible(+,?),
        add_type_info(+,+).

/* Does an individual definitely have a particular type? */

type(Type,Indiv) :-
   get_i_pattern(Indiv,Patt1),
   type_pattern(Type,Patt2),
   pattern_subsume(Patt2,Patt1).

/* Is one type a super type of another? */

super_type(Big,Small) :-
   type_pattern(Big,Patt1),
   type_pattern(Small,Patt2),
   pattern_subsume(Patt1,Patt2).

pattern_subsume(Big,Small) :-
   not_subsume(Big,Small), !, fail.
pattern_subsume(_,_).

not_subsume(Big,Small) :-
   numbervars(Small,1,_),
   Small=Big, !, fail.
not_subsume(_,_).

/* Could an individual have a given type? */

compatible(Type,Indiv) :-
   get_i_pattern(Indiv,Patt1),
   type_pattern(Type,Patt2),
   Patt1=Patt2.

/* Does an individual definitely not have a given type? */

not_type(Type,Indiv) :-
```

```
   compatible(Type,Indiv), !, fail.
not_type(_,_).

/* Add new type information about an individual */

add_type_info(Type,Indiv) :-
   get_i_pattern(Indiv,Patt1),
   type_pattern(Type,Patt2),
   Patt1 = Patt2,
   change_i_pattern(Indiv,Patt2).

/* See what we know about an individual */

print_types(Indiv) :-
   get_i_pattern(Indiv,Patt),
   p_type_arg(Patt,Indiv).

p_types(0,_,_) :- !.
p_types(N,Patt,Indiv) :-
   arg(N,Patt,Arg),
   p_type_arg(Arg,Indiv),
   N1 is N-1, p_types(N1,Patt,Indiv).

p_type_arg(A,_) :- var(A), !.
p_type_arg(Arg,Indiv) :-
   functor(Arg,Ty,N),
   write(Ty), write('('), write(Indiv), write(').'), nl,
   p_types(N,Arg,Indiv).

/* Associating type patterns with individuals */

get_i_pattern(Indiv,Patt) :-
   get_indiv(Indiv),
   his_pattern(Indiv,Patt).

his_pattern(Indiv,Patt) :-
   call(i_pattern(Indiv,Patt)), !.
his_pattern(Indiv,Patt) :-
   type_pattern(entity,Patt).

[illegible]t_indiv(Indiv) :- nonvar(Indiv), !.
get_indiv(Indiv) :- call(i_pattern(Indiv,_)).

change_i_pattern(Indiv,Patt1) :-
   retract(i_pattern(Indiv,_)),
   fail.
change_i_pattern(Indiv,Patt1) :-
   assertz(i_pattern(Indiv,Patt1)).
```

```
%% PLCODE.SUB  -  Predicate library support code
%%
plcode.sub       %% This file
plcode.          %% File to compile modules
plib.ops         %  Operator declarations
preds.pl         %  Simple meta-level properties
mlprp1.pl        %  Various derived meta-level properties
mlface.pl        %  meta-level properties (PLIB interface)
meta.pl          %  meta-level predicate declarations
kstype.pl        %  KS type definitions
t1.pl
t2.pl
t3.pl
t4.pl
must.pl          %  meta-level Police
load.pl          %  Load predicate library files
ks.pl            %  Low level KS structure manipulation
rulef.pl         %  Rule Forms
tyload.pl        %  Type hierarchy loader
err.pl           %  Error messages
```

```
/* PLCODE : Compile support modules for predicate library facilities

                                                  Lawrence
                                                  Updated: 22 November 81
*/

 % Assumed Utility modules from UTIL:
 %
 %              FILES
 %              EDIT
 %              IOROUT

 % This is currently for Mecho  (Interp options commented out)
 % Interp uses its own load file, see interp[400,434,pros]



  :- [
        'plcode:plib.ops'        % Operator declarations
     ].


  :- compile([

        'plcode:preds.pl',       % Simple meta-level properties
%%      'plcode:mlprp1.pl',      % Derived meta-level properties [Interp]
        'plcode:mlprp2.pl',      % Derived meta-level properties [Mecho]
        'plcode:mlface.pl',      % meta-level properties (PLIB interface)
        'plcode:meta.pl',        % meta-level declarations

        'plcode:kstype.pl',      % KS type definitions
        'plcode:t1.pl',
        'plcode:t2.pl',
        'plcode:t3.pl',
        'plcode:t4.pl',

        'plcode:must.pl',        % meta-level Police
        'plcode:load.pl',        % Load predicate library files
        'plcode:ks.pl',          % Low level KS structure manipulation
        'plcode:rulef.pl',       % Rule forms

        'plcode:tyload.pl',      % Type hierarchy loading loop

        'plcode:err.pl'          % Error messages

     ]).


  :- ks_init.           % Initialise KS system
```

```
PLCODE:PLCODE.CNG -- History of changes to PLCODE: modules
==========================================================

----------------------------------------------------------------------------
Lawrence  6 September 81
Purged remaining traces of not_nec_... (mlprop,mlprp1,meta).
Fixed slight (unnoticable) naming bug in mlprop.
Built file MLPRP2 which is the Mecho version of mlprp1.
        NB my one_to_n is different (it goes 1->n ! Perhaps mlprp1 should
           be changed?)
----------------------------------------------------------------------------
Chris 27/8/81
TYLOAD changed to be slightly slower, but so as not to care what
order the type information is given in.
----------------------------------------------------------------------------
Chris 27/8/81
Last vestiges of 'not_nec...' removed from MLPRP1. New versions of
'exists', 'unique', etc with extra arguments
----------------------------------------------------------------------------
Chris 5/8/81
MLPRP1 changed so that one gets a list of argument numbers when a goal
   unique by virtue of being "ground"
----------------------------------------------------------------------------
Chris+Lawrence 4/8/81
Special checks for type predicates removed from MLFACE.
The (multiple) 'must_know_predicate' checks removed from MLPRP1
KS.PL changed so that 'must_know_predicate' is used for every get or fetch
from the PLIB database.
----------------------------------------------------------------------------
Chris a/8/81 (Using duff terminal)
New version of TYLOAD introduced, allowing the specification of A rules
for types defined by & rules
----------------------------------------------------------------------------
Chris
 27 July 81
Finished changes to TYLOAD.PL, including error message for
undefined type. Changed mode of 'type_pattern' from (+,?) to (+,-)
(Otherwise the error message will come out at wrong times)
--------------------------------------------------------------------------
Lawrence
  July 81
        Generalised T1.PL so that it is easier to add meta-predicates.
        It is now necessary to have definitions for 'meta_predicate/2',
        see META.PL.

        LOAD.PL has also undergone odd changes to allow dispatches to
        mechanisms for loading theories and type hierarchies. This won't
        affect files not using these things.
--------------------------------------------------------------------------
<baseline>
```

```
/* PLIB.OPS : Operator declarations for the predicate library
                and supporting modules.

                                                  Lawrence
                                                  Updated: 14 June 81
*/

% General operators

  :- op(1160,xfx,[ <==, <-->, <--, --> ]).
  :- op(850,xfy,&).
  :- op(700,xfy,=<).
  :- op(400,fx,[define,theory]).
  :- op(350,xfy,:).
  :- op(300,fy,^).
```

```
/* PREDS.PL : Some simple meta-level facts about predicates

                                                    Lawrence
                                                    Updated: 1 December 81
*/

 /* EXPORT */

  :- public     same_predicate/2,
                same_predicate/3,
                nonvar_same_predicate/2,
                type_predicate/1,
                type_predicate/3,
                copy_args/3.


/* MODES */

  :- mode       same_predicate(?,?,?),
                same_predicate(?,?),
                nonvar_same_predicate(+,+),
                type_predicate(+),
                type_predicate(+,?,?),
                not_unusual(+),
                unusual(+),
                copy_args(+,+,?),
                    copy_args_1(+,+,+).



                           % Two terms have the same predicate
                           %  This predicate allows for either of the
                           %  arguments to be uninstantiated, in which case
                           %  it will become instantiated to a (fresh) most
                           %  general instance of the predicate.
                           % There are two versions - one which returns the arity
                           %  as well.

same_predicate(Pred1,Pred2) :- same_predicate(Pred1,Pred2,_).



same_predicate(Pred1,Pred2,Arity)
     :- var(Pred1),
        !,
        nonvar(Pred2),
        same_predicate(Pred2,Pred1,Arity).

same_predicate(Pred1,Pred2,Arity)
     :- functor(Pred1,F,Arity),
        functor(Pred2,F,Arity).



                           % Version of same_predicate which demands that both
                           %  arguments be bound.

nonvar_same_predicate(Pred1,Pred2)
     :- nonvar(Pred1),
        nonvar(Pred2),
        functor(Pred1,F,Arity),
        functor(Pred2,F,Arity).



                          % A predicate is a type predicate
                          %  This is here assumed to be true about ALL
                          %  single argument predicates except for a few
                          %  hack cases that ought not to be here.

type_predicate(Pred) :-
        nonvar(Pred),
        functor(Pred,_,1),
        not_unusual(Pred).


type_predicate(Pred,Type,Arg)
     :- nonvar(Pred),
        functor(Pred,Type,1),
        not_unusual(Pred),
        arg(1,Pred,Arg).


        not_unusual(Pred) :- unusual(Pred), !, fail.
        not_unusual(Pred).

        unusual(sought(_)).
        unusual(given(_)).


                          % Create a new assertion by copying arguments

copy_args(Numbers,Ass1,Ass2)
     :- same_predicate(Ass1,Ass2,_),
        copy_args_1(Numbers,Ass1,Ass2).


copy_args_1([],_,_)
     :- !.
  py_args_1([N|Ns],Ass,Ass1)
     :- arg(N,Ass,Arg), arg(N,Ass1,Arg),
        copy_args_1(Ns,Ass,Ass1).
```

```
/* MLPRP1.PL : Meta level properties for the Semantic Interpreter

                                                  Chris + Lawrence
                                                  Updated: 6 September 81
*/

/* EXPORT */

  :- public     exists/3,
                exists/5,
                unique/3,
                unique/5,
                function/5,
                commutative/3,
                aliorelative/3.


/* IMPORT */
/*

    MLFACE          exists_pattern/3
                    unique_pattern/3
                    function_pattern/3
                    commutative_pattern/3
                    aliorelative_pattern/3
*/


/* MODES */

  :- mode       exists(+,+,+),
                notexists(+,+,+),
                notunique(+,+,+),
                unique(+,+,+),
                exists(+,+,+,-,-),
                unique(+,+,+,-,-),
                function(+,+,+,-,-),
                commutative(+,-,-),
                aliorelative(+,-,-).


                           % It is guaranteed that a solution to Goal exists
                           %  This is proved using the meta-level properties
                           %  of the predicate of the goal with the current
                           %  instantiation state of the goal.

exists(Goal,S,Env)
     :- notexists(Goal,S,Env),
        !,
        fail.
exists(_,_,_).

notexists(Goal,S,Env) :- exists(Goal,S,Env,_,_), !, fail.
notexists(_,_,_).


                           % If there is a solution to Goal the this is
                           %  guaranteed to be a unique solution. Again
                           %  proved on meta-level grounds.
```

```
unique(Goal,S,Env)
      :- notunique(Goal,S,Env),
         !,
         fail.
unique(_,_,_).

notunique(Goal,S,Env) :- unique(Goal,S,Env,_,_), !, fail.
notunique(_,S,Env).

/* Predicates with extra "how" arguments */

function(Ass,S,Env,Argnos,Valnos) :-
    function_pattern(Ass,Argnos,Valnos),
    allbound(Argnos,Ass,S,Env),
    allunbound(Valnos,Ass,S,Env).

exists(Ass,S,Env,Argnos,Valnos) :-
    exists_pattern(Ass,Argnos,Valnos),
    allbound(Argnos,Ass,S,Env),
    allunbound(Valnos,Ass,S,Env).

unique(Ass,S,Env,Argnos,[]) :-
    pure(Ass,S,Env), !,
    functor(Ass,_,N),
    one_to_n(N,Argnos).
unique(Ass,S,Env,Argnos,Valnos) :-
    unique_pattern(Ass,Argnos,Valnos),
    allbound(Argnos,Ass,S,Env).

    one_to_n(0,[]) :- !.
    one_to_n(N,[N|Ns]) :-
        N1 is N-1, one_to_n(N1,Ns).

commutative(Ass,TwoArgnos,Rest) :-
    commutative_pattern(Ass,TwoArgnos,Rest).

aliorelative(Ass,TwoArgnos,Rest) :-
    aliorelative_pattern(Ass,TwoArgnos,Rest).
```

```
/* MLPRP2.PL : Meta level properties for Mecho (Problem Solver)

                                                Lawrence
                                                Updated: 1 December 81
*/

 % This is the Mecho version of Interp's MLPRP1.PL  (should be kept in step
 % with significant changes in that file).


/* EXPORT */

  :- public     exists/1,
                unique/1,
                exists/2,
                unique/2,
                function/2,
                commutative/2,
                aliorelative/2.


   IMPORT */

     MLFACE           exists_pattern/3
                      unique_pattern/3
                      function_pattern/3
                      commutative_pattern/3
                      aliorelative_pattern/3
*/


/* MODES */

  :- mode       exists(+),
                unique(+),
                exists(+,?),
                unique(+,?),
                one_to_n(+,+),
                function(+,?),
                commutative(+,?),
                aliorelative(+,?).


                          % Cheapo versions of below for certain important cases

exists(Goal) :-
   exists_pattern(Goal,Argnos,Valnos),
   allbound(Argnos,Goal),
   allunbound(Valnos,Goal).


unique(Goal) :-
   unique_pattern(Goal,Argnos,Valnos),
   allbound(Argnos,Goal).


                          % Goal has properties depending on the information
                          %  known abouts its predicate (from ..._pattern calls)
                          %  which describes the instantiation state that the
                          %  goal must satisfy for the property to hold.

function(Goal,how(Argnos,Valnos)) :-
   function_pattern(Goal,Argnos,Valnos),
   allbound(Argnos,Goal),
   allunbound(Valnos,Goal).


exists(Goal,how(Argnos,Valnos)) :-
   exists_pattern(Goal,Argnos,Valnos),
   allbound(Argnos,Goal),
   allunbound(Valnos,Goal).


 %      This doesn't fit with my use of unique with silly check
 %      (What does Chris do about this?)
 %
 % unique(Goal,how(Argnos,[])) :-
 %    pure(Goal),
      !,
      functor(Goal,_,N),
 %    one_to_n(1,N,Argnos).
 %
 %
 %    one_to_n(N,Max,[]) :- N > Max, !.
 %
 %    one_to_n(N,Max,[N|NRest]) :-
 %       N1 is N+1, one_to_n(N1,Max,NRest).


unique(Goal,how(Argnos,Valnos)) :-
   unique_pattern(Goal,Argnos,Valnos),
   allbound(Argnos,Goal).


commutative(Goal,how(TwoArgnos,Rest)) :-
   commutative_pattern(Goal,TwoArgnos,Rest).


 iorelative(Goal,how(TwoArgnos,Rest)) :-
   aliorelative_pattern(Goal,TwoArgnos,Rest).
```

```
/* MLFACE.PL : Meta level interface to the predicate library

                                                        Lawrence
                                                        Updated: 1 December 81

*/

 % This module defines Prolog procedures that model meta-level predicates.
 % The ideal abstraction is that these are elements in a meta-level database.
 % This is currently implemented by accessing KS structures built by the
 % predicate library management routines.

 % Meta-level predicates can receive one of two sorts of treatment when loaded,
 % either they are turned into "patterns" (lists of numbers are used to show
 % which arguments count and which ones don't - for whatever purpose (being a
 % function, being aliorelative etc)); or they are left "simple" (as they were
 % entered). See the file META.PL which defines the type of treatment for each
 % meta-level predicate. In this file, every predicate that has undergone
 % "pattern" treatment is interfaced with the name ...._pattern. "simple"
 % predicates are interfaced with their original names.


 /* EXPORT */

   :- public     function_pattern/3,
                 exists_pattern/3,
                 commutative_pattern/3,
                 aliorelative_pattern/3,
                 unique_pattern/3,
                 normal_form/2,
                 object_level_rule/2,
                 object_level_neg_rule/2,
                 default_rule/2,
                 argument_names/1,
                 argument_names/2,
                 argument_types/1,
                 argument_types/2,
                 easy_inference/1.


    IMPORT */
 /*
     KS          set/3

     RULEF       set_rule/3
*/


/* MODES */

   :- mode       function_pattern(+,?,?),
                 exists_pattern(+,?,?),
                 commutative_pattern(+,?,?),
                 aliorelative_pattern(+,?,?),
                 unique_pattern(+,?,?),
                 normal_form(+,?),
                 object_level_rule(+,?),
                 object_level_neg_rule(+,?),
                 default_rule(+,?),
                 argument_names(+),
                  argument_names(+,?),
                  argument_types(+),
                  argument_types(+,?),
                  easy_inference(+).



                          % Commutativity

commutative_pattern(Pred,Args,Others)
        :- set(meta_knowledge,Pred,commutative(Args,Others)).


                          % Aliorelativity

aliorelative_pattern(Pred,Args,Others)
        :- set(meta_knowledge,Pred,aliorelative(Args,Others)).


                          % Function properties template

  nction_pattern(Pred,Args,Others)
     :- set(meta_knowledge,Pred,function(Args,Others)).



                          % Existence properties template

exists_pattern(Pred,Args,Others)
     :- set(meta_knowledge,Pred,function(Args,Others)).

exists_pattern(Pred,Args,Others)
     :- set(meta_knowledge,Pred,exists(Args,Others)).



                          % Uniqueness properties template

unique_pattern(Pred,Args,Others)
     :- set(meta_knowledge,Pred,function(Args,Others)).

unique_pattern(Pred,Args,Others)
     :- set(meta_knowledge,Pred,unique(Args,Others)).



                          % Normal form rules

normal_form(Assertion,NewAssertion)
     :- set(normal_form,Assertion,RuleName),
        set_rule(RuleName,Assertion,NewAssertion).



                          % Object level inference rules

object_level_rule(Goal,Subgoals)
     :- set(inference_rules,Goal,RuleName),
        set_rule(RuleName,Goal,Subgoals).
```

```
                          % Object level negative inference rules

object_level_neg_rule(Goal,Subgoals)
      :- get(inference_rules,Goal,RuleName),
         get_rule(RuleName,~Goal,Subgoals).


                          % Object level default rules

default_rule(Goal,Subgoals)
      :- get(default_rules,Goal,Rulename),
         get_rule(Rulename,Goal,Subgoals).


                          % Argument names (for gensyming etc).

argument_names(Pred)
      :- get(names,Pred,Pred),
         !.


argument_names(Pred,Names)
      :- get(names,Pred,Names),
         !.



                          % Template for the object level types of the
                          %  arguments of the predicates.

argument_types(Pred)
      :- get(types,Pred,Pred),
         !.


argument_types(Pred,Types)
      :- get(types,Pred,Types),
         !.



                          % Easy Inference
                          %  This is basically to allow special mechanisms
                          %  to be plugged in so that they always get used,
                          %  even when not trying very hard.

easy_inference(Pred) :-
         get(meta_knowledge,Pred,easy_inference),
         !.
```

```
/* META.PL : Meta level predicates (somewhat loose list)

                                                                Lawrence
                                                                Updated: 1 December 81
*/

/* EXPORT */

   :- public     meta_predicate/2,
                 meta_predicate_index/2.


                                  % Meta predicates allowed in { meta_knowledge }
                                  %  sections in the predicate library.
                                  % The second arg specifies the T1 transformation
                                  %  to be used, and can be one of {simple,pattern}.
                                  %  See the module T1 for details.

meta_predicate( function(_,_),            pattern ).
 eta_predicate( exists(_,_),              pattern ).
   ta_predicate( unique(_,_),              pattern ).
 eta_predicate( commutative(_,_),         pattern ).
meta_predicate( reflexive(_,_),           pattern ).
meta_predicate( aliorelative(_,_),        pattern ).
meta_predicate( easy_inference(_),        simple ).
meta_predicate( derived(_),               simple ).
meta_predicate( index(_,_),               simple ).


                                  % Meta-level predicates which can be dynamically
                                  %  added/forgotten from the database

meta_predicate_index( defn(Q,_,_), [Q] ).
```

```
/* KSTYPE.PL : Types of knowledge sources applicable to predicates

                                                   Lawrence
                                                   Updated: 14 June 81
*/

% This module defines the types of entry that can be found within
% predicate definitions. These types have atoms as names and these
% names in curly brackets introduce entries specific to that type
% in a predicate definition. See the predicate library for examples
% of what this looks like.
%
% To introduce a new type, specify the following information:
%
%       ks_info(Type,N,Style)
%
%               Type is the name of the KS type (Prolog atom)
%               N is an integer, ordering this type with respect to others
%               Style is one of {ruleform,other} and specifies whether or
%                 not the entries of a type are to be stored separately as
%                 rule forms, with rule names in the KS, or whether the
%                 entries just go straight into the KS. There are procedures
%                 in the module RULEF which maintain the ruleform abstraction
%                 and these things should only be manipulated with these
%                 (ie in the translation modules).
%
%       ks_max(Max)
%
%               Max is an integer giving the total number of types. Note
%               that the ordering given in ks_info (ie N) should be the
%               set of integers between 1 and Max.
%
%       ks_translate(Type,Pred,X,NewX)
%
%               This relates the input form of the entry to its internal
%               form, for each Type of entry. X is thus the input form
%               read and NewX should be returned as the desired internal form.
%               Pred is a general instance of the predicate being defined
%               which is shared across all of the KS types. (This allows
%               some sharing of variables. It is currently assumed that
%               this will not be instantiated by the definitions). If the
%               KS type has Style ruleform, then a RuleForm should be
%               produced as the internal form. There are construction
%               procedures for this in the module RULEF.
%
%               Note that the recommended way of expressing disgust at
%               some input term is just to fail. The rest of the loading
%               mechanism will handle this (and produce a message).



/* EXPORT */

  :- public     ks_type/1,
                ks_type/3,
                ks_max/1,
                ks_translate/4.


/* IMPORT */
```

```
/*
   There is a separate module defining the translation for each Type.
   Currently these are:

                    KSTYPE
                      |
                      |--------- T1                    { meta_knowledge }
                      |
                      |--------- T2                    { types }
                      |
                      |--------- T3                    { normal_form }
                      |
                      |--------- T4                    { inference_rules }
                                                       { default_rules }
*/


/* MODES */

   :- mode        ks_type(?),
                  ks_type(?,?,?),
                  ks_max(?),
                  ks_translate(+,+,+,-).


                           % KS types

ks_type(Type) :- ks_type(Type,_,_).


ks_type( meta_knowledge,  1, other ).
ks_type( types,           2, other ).
ks_type( normal_form,     3, ruleform ).
ks_type( inference_rules, 4, ruleform ).
ks_type( default_rules,   5, ruleform ).


ks_max( 5 ).



ks_translate(meta_knowledge,Pred,X,NewX)
     :- t1_trans(X,Pred,NewX).                         % module T1

ks_translate(types,Pred,X,NewX)
     :- t2_trans(X,Pred,NewX).                         % module T2

ks_translate(normal_form,Pred,X,NewX)
     :- t3_trans(X,Pred,NewX).                         % module T3

ks_translate(inference_rules,Pred,X,NewX)
     :- t4_trans(X,Pred,NewX).                         % module T4

ks_translate(default_rules,Pred,X,NewX)
     :- t4_trans(X,Pred,NewX).                         % module T4
```

```
/* T1.PL : Translate 'meta_knowledge' forms

                                                    Lawrence
                                                    Updated: 9 July 81
*/


/* EXPORT */

  :- public     t1_trans/3.


/* IMPORT */
/*
                meta_predicate/2        from  META
*/


/* MODES */

  :- mode       t1_trans(+,+,-),
                    t1_trans2(+,+,+,-),
                    t1_twiddle(+,-),
                    t1_collect(+,-),
                    t1_sweep(+,+,-),
                    t1_sweep_one(+,+,-,-),
                    t1_copy_args(+,+,+),
                    t1_argnorm(?,?).



                          % Translate meta_knowledge forms

t1_trans( MetaPred, Pred, KSForm )
     :- meta_predicate(MetaPred,HackType),
        t1_trans2(HackType,MetaPred,Pred, KSForm).



                          % Decide what kind of transformation to do

t1_trans2(simple,MetaPred,Pred, KSForm)
     :- functor(MetaPred,MetaP,Arity),
        Arity >= 1,
        arg(1,MetaPred,Pred),
        NewArity is Arity-1,
        functor(KSForm,MetaP,NewArity),
        t1_copy_args(Arity,Pred,KSForm).

t1_trans2(pattern,MetaPred,Pred, KSForm)
     :- functor(MetaPred,MetaP,2),
        arg(1,MetaPred,PredX),
        arg(2,MetaPred,In),
        nonvar_same_predicate(Pred,PredX),
        t1_argnorm(In,InList),
        t1_twiddle(InList,InNumList),
        t1_collect(PredX,OutNumList),
        functor(KSForm,MetaP,2),
        arg(1,KSForm,InNumList),
        arg(2,KSForm,OutNumList).
```

```
                                % Go through InList and mark all the variables
                                %  - these will share with the ones in PredX.
                                %  Set up the InNumList with parts of the mark,
                                %  these will get instantiated by t1_collect.

t1_twiddle([],[]).

t1_twiddle([mark(Number)|Rest],[Number|NumRest])
      :- t1_twiddle(Rest,NumRest).



                                % Sweep across PredX, building an OutNumList of all
                                %  the unmarked arguments, and also fill in the
                                %  numbers for the marked variables.

t1_collect(PredX,OutNumList)
      :- functor(PredX,_,Arity),
         t1_sweep(Arity,PredX,OutNumList).



t1_sweep(0,_,[]) :- !.

t1_sweep(N,PredX,OutNumList)
      :- arg(N,PredX,Arg),
         t1_sweep_one(Arg,N,OutNumList,OutNumRest),
         N1 is N-1,
         t1_sweep(N1,PredX,OutNumRest).


t1_sweep_one(V,N,[N|Rest],Rest) :- var(V), !.

t1_sweep_one(mark(N),N,Rest,Rest).



                                % Copy args across  (Arg n ==> Arg n-1)
                                %  NB Arity >= 1 (first arg)

t1_copy_args(1,_,_) :- !.

t1_copy_args(N,MetaPred,KSForm)
      :- N1 is N-1,
         arg(N,MetaPred,Arg),
         arg(N1,KSForm,Arg),
         t1_copy_args(N1,MetaPred,KSForm).


                                % Normalise arguments to list form

t1_argnorm(V,[V]) :- var(V), !.

t1_argnorm(X,X).
```

```
/* T2.PL : Translate 'types' forms

                                                    Lawrence
                                                    Updated: 9 July 81
*/


/* EXPORT */

  :- public      t2_trans/3.


/* IMPORT */
/*
                nonvar_same_predicate/2         from  PREDS
                type_predicate/3
*/

/* MODES */

  :- mode       t2_trans(+,+,-),
                t2_flatten(+,+),
                t2_check(+,+),
                t2_checkdo(?,+).



                           % Translate 'types' forms
                           %  This currently involves turning a rule
                           %  like structure into a flat record of the
                           %  type atoms (using the predicates functor).
                           %  Why do I bother with this sweat I ask myself?
                           %  Somehow I think it's important to emphasise the
                           %  simple object-level nature of something that can
                           %  be used in various ways at the meta-level.

t2_trans((Tpred-->Conj),Pred,Tpred)
      :- nonvar_same_predicate(Tpred,Pred),
         t2_flatten(Conj,Tpred),
         functor(Tpred,_,Arity),
         t2_check(Arity,Tpred).



                           % Flatten conjunction. I assume that the types rule
                           %  has been expressed with Prolog variables which
                           %  will link Tpred and the individual type predicates.
                           %  Thus unifying the type into the type predicates
                           %  argument will instantiate the right argument of
                           %  Tpred!

t2_flatten(A&B,Tpred)
      :- !,
         t2_flatten(A,Tpred),
         t2_flatten(B,Tpred).

t2_flatten(X,Tpred)
      :- type_predicate(X,Type,Type).
```

```
                                        % Check that a types form was complete

t2_check(0,_) :- !.

t2_check(N,Tpred)
        :- arg(N,Tpred,Arg),
           t2_checkdo(Arg,Tpred),
           N1 is N-1,
           t2_check(N1,Tpred).



t2_checkdo(V,Tpred)
        :- var(V),
           !,
           V = entity,
           errmess('Types incomplete ("entity" assumed)',Tpred).

t2_checkdo(_,_).
```

```
/* T3.PL : Translate 'normal_form'  forms

                                                  Lawrence
                                                  Updated: 9 July 81
*/


/* EXPORT */

  :- public     t3_trans/3.


/* IMPORT */
/*
                nonvar_same_predicate/2           from  PREDS

                make_ruleform/3                   from  RULEF
*/


/* MODES */

  :- mode       t3_trans(+,+,-).



                          % Translate 'normal_form' forms
                          %  The left side of the equivalence is supposed
                          %  to just be the predicate, the other side amy be
                          %  some conjunction. An enclosing implication may
                          %  be present to provide a context.

t3_trans( (Context --> (Npred <--> Conj)), Pred, RuleForm)
     :- nonvar_same_predicate(Npred,Pred),
        make_ruleform(Npred,context(Context,Conj),RuleForm).

t3_trans( (Npred <--> Conj), Pred, RuleForm )
     :- nonvar_same_predicate(Npred,Pred),
        make_ruleform(Npred,Conj,RuleForm).
```

```
/* T4.PL : Translate various object level rule forms

                                                Lawrence
                                                Updated: 9 July 81
*/


/* EXPORT */

  :- public     t4_trans/3.


/* IMPORT */
/*
                nonvar_same_predicate/2         from  PREDS

                make_ruleform/3                 from  RULEF
*/


/* MODES */

  :- mode       t4_trans(+,+,-).



                        % Translate 'inference_rules' forms
                        %       or  'default_rules' forms
                        %  This code also allows for negative rules
                        %  whose heads are marked with ~(_). This is
                        %  left on the head so that negative rules can
                        %  be distinguished on retrieval.

t4_trans(X,Pred,RuleForm)
     :- t4_cases(X,Head,Body),
        t4_norm(Head,Norm),
        nonvar_same_predicate(Norm,Pred),
        make_ruleform(Head,Body,RuleForm).



 ,_cases( (Head <-- Body), Head, Body ) :- !.

t4_cases( (Body --> Head), Head, Body ) :- !.

t4_cases( Fact, Fact, true ).



t4_norm(~(Head),Head) :- !.

t4_norm(Head,Head).
```

```
/* MUST.PL : Meta-level police force which beats the shit out
                 of you if you don't play according to the rules.

                                                  Lawrence
                                                  Updated: 14 June 81
*/

/* EXPORT */

  :- public     must_know_predicate/1.


/* IMPORT */
/*
    UTIL:EDIT        edit/1
    UTIL:IOROUT      error/3

    LOAD             load/1

    KS               known_predicate/1


/* MODES */

  :- mode        must_know_predicate(?).



must_know_predicate(Goal)
     :- var(Goal),
        !,
        error('Must know this?  %t - Its a variable!',[Goal],break).      fail  *

must_know_predicate(Goal)
     :- known_predicate(Goal),
        !.

must_know_predicate(Goal)
     :- functor(Goal,Fn,A),
        ttynl,
        display('HEY - You have told me nothing about the predicate:  '),
        display(Fn), ttyput("/"), display(A),
        ttynl, ttynl,
        must_chance(Goal).



must_chance(Goal)
     :- display('Do you want to define it ("yes." or I fail)? '),
        ttyflush,
        read(yes),
        edit('new.def'),
        load('new.def'),
        must_know_predicate(Goal).
```

```
/* LOAD.PL : Load predicate definitions

                                                 Lawrence
                                                 Updated: 6 July 81
*/

% This module provides a procedure 'load' which loads predicate library
% files into the database. It relies on KS manipulations provided by
% the modules:
%               KSTYPE  -  Definition of KS types. This module will
%                          refer to various other modules which define
%                          the particular input transformations for each
%                          KS type.
%               KS      -  Underlying manipulations on KS structures
%                          This includes low level storage and retrieval
%                          mechanisms.
%
% And also various sub-mechanisms provided by:
%
%               THLOAD  -  Theory loading loop

%               TYLOAD  -  Type hierarchy loading loop


/* EXPORT */

  :- public     load/1,
                load_start/1,
                load_finish/1,
                load_resync/0,
                read_next/1.


/* IMPORT */
/*
                open/2                  from  UTIL:FILES
                close/2

                errmess/1               from  ERR
                errmess/2

                ks_type/1               from  KSTYPE
                ks_translate/4

                new_ks/3                from  KS
                add_entry/5
                finalise/2

                th_start/1              from  THLOAD

                ty_start/0              from  TYLOAD
*/


/* MODES */

  :- mode       load(+),
                load_start(+),
                load_finish(+),
                load_resync,
                load_sortout(+,-),
                loading(+,+,+,+,+),
                check_pred(+,-),
                chk_names(+,+),
                check_type(+,-),
                check_and_add(+,+,+,+,+,-),
                pred_ok(+,+),
                type_ok(+,+),
                chkadd(+,+,+,+,+,-),
                read_next(?).



                        % Load from a list of files

load(V)
      :- var(V),
         !,
         errmess('Variable as file name').

load([]) :- !.

load([HD|TL])
      :- !,
         load(HD),
         load(TL).

load(File)
      :- open(Old,File),
         !,
         read_next(Next),
         load_start(Next),
         ttynl, display('Definitions loaded from '),
         display(File), ttynl,
         close(File,Old).

load(_).


                        % Entry to loading cycle
                        %  Expects the next term to have been read and this
                        %  is the first argument. This facilitates using it
                        %  as a return point from other loading mechanisms
                        %  who have read too far.

load_start(Next)
      :- loading(Next,null_pred,null_type,null_ks,[]).



                        % This is a list of terms which can terminate
                        %  particular loading 'blocks'. It is intended
                        %  for use by other specialised loops to which
                        %  the main loop may dispatch (expecting a return
                        %  when one of these is read).

load_finish(end_of_file).
load_finish(define(_)).
load_finish(theory(_)).
load_finish(type_hierarchy).


                           % How to resyncronise so as to continue loading with
                           %  the next 'block' in case of serious error (again
                           %  mainly for external use).

load_resync
        :- repeat,
           read_next(Next),
           load_sortout(Next,RealNext),
           !,
           load_start(RealNext).


load_sortout(Fin,Fin) :- load_finish(Fin).

load_sortout(_,_) :- fail.


                           % Main loop.
                           %  For each definition block build a KS structure
                           %  and accumulate rule forms. When a definition
                           %  block is complete we finalise the KS and the
                           %  rule forms. We also do some error checking and
                           %  allow the loading to continue despite errors. The
                           %  current error stategy is:
                           %       Bad define   -  ignore whole block
                           %       Bad KS type  -  ignore all entries in that part
                           %       Bad entry    -  ignore that entry
                           %  If defines or KS types are missing then entries
                           %  are ignored (until the next define or KS type is
                           %  entered).
                           % There is also a dispatch to the theory definition
                           %  modules for the loading of theories. This may then
                           %  return to load_start, whereupon normal loading will
                           %  continue.
                           % And another dispatch for type hierarchies ... the
                           %  define bit should be pulled out as well..

loading(end_of_file,_,_,KS,RuleForms)
        :- !,
           finalise(KS,RuleForms).

loading(theory(Theory),_,_,KS,RuleForms)
        :- !,
           finalise(KS,RuleForms),
           th_start(Theory).

loading(type_hierarchy,_,_,KS,RuleForms)
        :- !,
           finalise(KS,RuleForms),
           ty_start.

loading(define(P),_,_,KS,RuleForms)
        :- !,
           finalise(KS,RuleForms),
           check_pred(P,Pred),
           new_ks(Pred,P,NewKS),
           read_next(Next),
           loading(Next,Pred,null_type,NewKS,[]).

loading({T},Pred,_,KS,RuleForms)
       :- !,
           check_type(T,Type),
           read_next(Next),
           loading(Next,Pred,Type,KS,RuleForms).

loading(nothing_to_say,Pred,Type,KS,RuleForms)
       :- !,
           read_next(Next),
           loading(Next,Pred,Type,KS,RuleForms).

loading(X,Pred,Type,KS,RuleForms)
       :- check_and_add(X,Pred,Type,KS,RuleForms,NewRuleForms),
           read_next(Next),
           loading(Next,Pred,Type,KS,NewRuleForms).


                              % Check the predicate in a define.

check_pred(Pn,Pred)
       :- functor(Pn,F,Arity),
           chk_names(Arity,Pn),
           functor(Pred,F,Arity),
           !.

check_pred(Pn,err_pred)
       :- errmess('Invalid define (section ignored)',Pn).



                              % Check that predicate only has atoms in it
                              %  (These are gensym names)

chk_names(0,_) :- !.

chk_names(N,Pred)
       :- arg(N,Pred,Arg),
           atom(Arg),
           N1 is N-1,
           chk_names(N1,Pred).



                              % Check that a KS type is ok

check_type(Type,Type)
       :- ks_type(Type),
           !.

check_type(T,err_type)
       :- errmess('Invalid KS type (data ignored)',T).


                              % Check and add a data entry to current KS
```

```
check_and_add(X,Pred,Type,KS,RuleForms,NewRuleForms)
     :- pred_ok(Pred,X),
        type_ok(Type,X),
        chkadd(X,Pred,Type,KS,RuleForms,NewRuleForms),
        !.

check_and_add(_,_,_,_,RuleForms,RuleForms).


                        % Pred is valid

pred_ok(err_pred,_) :- !, fail.

pred_ok(null_pred,X)
     :- !,
        errmess('Missing define - ignoring',X),
        fail.

  ed_ok(_,_).


                        % Type is valid

type_ok(err_type,_) :- !, fail.

type_ok(null_type,X)
     :- !,
        errmess('Missing KS type - ignoring',X),
        fail.

type_ok(_,_).


                        % Add an entry to current KS according to Type
                        %  this may accumulate more Rule Forms as well

 nkadd(X,Pred,Type,KS,RuleForms,NewRuleForms)
     :- ks_translate(Type,Pred,X,NewX),
        add_entry(Type,NewX,KS,RuleForms,NewRuleForms),
        !.

chkadd(X,_,_,_,_,_)
     :- errmess('Bad entry ignored',X),
        fail.


                        % Read next entry - discard variables

read_next(X)
     :- repeat,
        read(Y),
        ( nonvar(Y)  ;  errmess('Variable ignored'), fail ),
        !,
        X = Y.
```

```
/* KS.PL  :  Manipulating KS structures

                                                  Lawrence
                                                  Updated: 4 August 81
*/


/* EXPORT */

  :- public     ks_init/0,
                add_entry/5,
                blank_ks/2,
                fetch/3,
                finalise/2,
                set/3,
                ks_key/2,
                new_ks/3,
                known_predicate/1,
                unknown_predicate/1.


   IMPORT */
/*
    KSTYPE      ks_type/3

    RULEF       ruleform/2
                rulename/2

    PREDS       type_predicate/1

    ERR         errmess/2
*/


/* MODES */

  :- mode       ks_init,
                new_ks(+,+,-),
                blank_ks(+,-),
                ks_slot(+,+,?),
                add_entry(+,+,+,+,-),
                addtoslot(+,+),
                finalise(+,+),
                ks_flush(+,+),
                ks_record_ruleforms(+),
                ks_recforms(+,+,-),
                complete_ks(+,-),
                still_fresh(+),
                stillf(+,+),
                fillall(+,+),
                fillslot(+),
                known_predicate(+),
                unknown_predicate(+),
                fetch(+,+,?),
                set(+,+,?),
                backthrough(+,?),
                ks_key(+,?).
```

```
                                    % Initialise KS system
                                    %  This should be called once somewhere to set
                                    %  things set up. The best place is undoubtably
                                    %  as a part of loading the system.
                                    % Currently involves:
                                    %    Set RuleForm counter to 0

ks_init
       :- recorda(ruleform,ruleform(0),_).



                                    % Build a new KS

new_ks(null_pred,_,null_ks) :- !.

new_ks(err_pred,_,null_ks) :- !.

new_ks(Pred,Pn,KS)
       :- blank_ks(Pred,KS),
          arg(2,KS,Pn).



                                    % Build an empty KS

blank_ks(Pred,KS)
       :- ks_max(Max),
          N is Max+2,
          functor(KS,ks,N),
          arg(1,KS,Pred).



                                    % Access various slots. The first two slots are
                                    %  special. Notice that the other slots get displaced
                                    %  by this but this is invisible to the outside world.

ks_slot(predicate,KS,Slot) :- !, arg(1,KS,Slot).

ks_slot(names,KS,Slot) :- !, arg(2,KS,Slot).

ks_slot(Name,KS,Slot) :- ks_type(Name,N,_), Sn is N+2, arg(Sn,KS,Slot), !.

ks_slot(Name,_,_)
       :- errmess('Unknown KS slot',Name),
          fail.



                                    % Adding new entries to slots
                                    %  There are two styles of slot entry which depend
                                    %  on the KS type involved:
                                    %    for ruleforms we add a name to the slot and
                                    %                      keep the rule separate. In fact
                                    %                      we add it the the RuleForms list
                                    %                      being accumulated.
                                    %    for others we add add the entry itself.
                                    % Adding to a slot is done by extending a list which
```

```
                              %  ends with a variable.
                              %  N^2 performance I'm afraid, but there we are.

add_entry(Type,X,KS,RuleForms,NewRuleForms)
     :- ks_type(Type,_,Style),
        ks_style(Style,X,Add,RuleForms,NewRuleForms),
        ks_slot(Type,KS,Slot),
        addtoslot(Slot,Add).



ks_style(ruleform,RuleForm,RuleName,Forms,[RuleForm|Forms])
     :- !,
        rulename(RuleForm,RuleName).

ks_style(_,Entry,Entry,Forms,Forms).



addtoslot(V,X) :- var(V), !, V = [X|_].

 ddtoslot([_|Rest],X) :- addtoslot(Rest,X).



                              % Finalise a KS and a list of RuleForms
                              %  This involves:
                              %   Completing all the slots in the KS
                              %   Flushing any previous KS structures
                              %   Recording all the RuleForms
                              %   Recording the KS structure itself
                              %  Note that it is important that the RuleForms
                              %   are recorded first as the linkage between
                              %   ruleforms in the list and their names in the
                              %   KS structure is through shared variables which
                              %   must first be "hardened" into actual names
                              %   before the KS structure itself is recorded.

finalise(null_ks,_) :- !.

 inalise(KS,RuleForms)
     :- complete_ks(KS,Pred),
        ks_key(Pred,Key),
        ks_flush(Key,Pred),
        ks_record_ruleforms(RuleForms),
        recorda(Key,KS,_),
        !.

finalise(_,_).



                              % Flush any old KS structures for a predicate
                              %  I also flush all rule forms referred to by
                              %  any KS flushed. This seems OK at the moment
                              %  but may need rethinking in the long term
                              %  (10 April 81).

ks_flush(Key,Pred)
     :- blank_ks(Pred,KS),
          recorded(Key,KS,Ref),
          erase(Ref),
          ks_type(Type,_,ruleform),
          ks_slot(Type,KS,Slot),
          backthrough(Slot,rulename(N)),
          recorded(N,RuleForm,RuleRef),
          ruleform(RuleForm,N),
          erase(RuleRef),
          fail.

ks_flush(_,_).



                          % Record a list of ruleforms under their names
                          %  (ie the integer part of their names).
                          %  This is the bit where the actual number key
                          %  gets instantiated into position in the rule form
                          %  Since this variable shares with the one in the
                          %  rule name somewhere in a KS structure slot we
                          %  achieve the right linkage between names and rules!

ks_record_ruleforms([]) :- !.

ks_record_ruleforms(List)
      :- recorded(ruleform,ruleform(Counter),Ref),
         !,
         ks_recforms(List,Counter,NewCounter),
         erase(Ref),                             % Defensive, wait to see if OK
         recorda(ruleform,ruleform(NewCounter),_).



ks_recforms([],N,N).

ks_recforms([RuleForm|Rest],N,FinalN)
      :- N1 is N+1,
         ruleform(RuleForm,N1),
         recorda(N1,RuleForm,_),
         ks_recforms(Rest,N1,FinalN).



                          % Complete all the slots by filling in the holes
                          %  Also checks that the shared predicate has not
                          %  been instantiated in any way.
complete_ks(KS,Pred)
      :- ks_slot(predicate,KS,Pred),
         still_fresh(Pred),
         functor(KS,_,N),
         fillall(N,KS).



                          % Check that Pred part has not got instantiated
                          %  in any way

still_fresh(Pred)
      :- functor(Pred,F,Arity),
         stillf(Arity,Pred).
stillf(0,_) :- !.

stillf(N,Pred)
      :- arg(N,Pred,Arg),
         var(Arg),
         !,
         N1 is N-1,
         stillf(N1,Pred).

stillf(_,Pred)
      :- errmess('KS Predicate instantiated',Pred),
         fail.



                              % Finalising slots

fillall(2,_) :- !.                     % First two slots are special (not lists)

fillall(N,KS)
      :- arg(N,KS,Slot),
         fillslot(Slot),
         N1 is N-1,
         fillall(N1,KS).


fillslot(V) :- var(V), !, V=[].

fillslot([_|Rest]) :- fillslot(Rest).



%% Low level access functions %%



                              % Predicate is known to the system
                              %  Ie it has some sort of definition

known_predicate(Pred)
      :- unknown_predicate(Pred),
         !,
         fail.

known_predicate(_).



                              % Predicate is unknown

unknown_predicate(Pred)
      :- type_predicate(Pred),
         !,
         fail.
```

```
unknown_predicate(Pred)
      :- ks_key(Pred,Key),
         blank_ks(Pred,KS),
         recorded(Key,KS,_),
         !,
         fail.

unknown_predicate(_).


                              % Fetch a slot given Name and Predicate

fetch(Name,Pred,Slot)
      :- ks_key(Pred,Key),
         blank_ks(Pred,KS),
         recorded(Key,KS,_),
         !,
         ks_slot(Name,KS,Slot).

                              % Error catch now calls general 'must_know_predicate'
                              %  (Which will blow since the above failed)

fetch(_,Pred,_)
      :- % errmess('Unknown predicate (KS access)',Pred),  (*Old code*)
         must_know_predicate(Pred),
         fail.



                              % Get an entry, this involves fetching the slot
                              %  and returning list elements one at a time
                              %  through backtracking.

get(Name,Pred,Entry)
      :- fetch(Name,Pred,Slot),
         backthrough(Slot,Entry).


backthrough([X],Y) :- !, X = Y.

backthrough([X|Rest],X).

backthrough([_|Rest],X) :- !, backthrough(Rest,X).

backthrough([],_) :- !, fail.

backthrough(Else,Else).          % Not all slots are lists
                                 % (This could be cleaner)


                              % Relation between Predicate and database Key
                              %  My decision here is to use the atom of the
                              %  predicate rather than the functor as there
                              %  is likely to less hanging off this (facts
                              %  may be hung off the functor).

ks_key(X,Key)
      :- ( var(X) ; integer(X) ),
        !,
        errmess('Invalid KS key',X),
        fail.

ks_key(Pred,Key)
      :- functor(Pred,Key,_).
```

```
/* RULEF.PL : Rule form manipulation

                                                Lawrence
                                                Updated: 14 June 81
*/

% KS structures can have slots which contain ruleform's. In this case
% the slot holds the names of rules and the rules themselves are stored
% separately but can be accessed through the name.


/* EXPORT */

  :- public     ruleform/2,
                rulename/2,
                make_ruleform/3,
                set_rule/3.


/* MODES */

  :- mode       ruleform(+,?),
                rulename(+,?),
                make_ruleform(?,?,?),
                set_rule(+,?,?).


                         % What a rule form looks like. The name of the
                         %  rule is in fact an integer but it gets bottled
                         %  for the users purposes
                         %  <low level>

ruleform(ruleform(N,_,_),N).


                         % Making a (bottled) name for a rule form
                         %  <low level>

rulename(ruleform(N,_,_),rulename(N)).


                         % Make a rule form from a Head and a Body
                         %  This is intended for use by the KSTYPE
                         %  translation modules who turn input forms
                         %  into internal forms.

make_ruleform(Head,Body,ruleform(_,Head,Body)).



                         % Given a rule name find the rule form itself
                         %  in the database.

get_rule(rulename(N),Head,Body)
     :- recorded(N,ruleform(N,Head,Body),_),
        !.
```

```
/* TYLOAD.PL :  Read in type hierarchy
                Represent types by Prolog terms

                                                          Chris
                                                          Updated: 27/8/81

*/

/*
   Imports:

         errmess/2,
         load_start/1,
         load_finish/1,
         read_next.

*/

:- public
         ty_start/0,
         type_pattern/2.
         or_rule/2.                                       % CALLed in findall

:    ode ty_process(+),
         basify(+,+,-),
         hidden_ors(+,-),
         add_hidden_ancs(+,+),
         type_name(+,-),
         ty_nmember(+,-,-),
         subtype(+,?),
         ty_intersect(+,-),
         maketype(+,+),
         type_pattern(+,-).

:- op(100,xfx,<->).
:- op(50,xfy,#).
:- op(50,xfy,&).

y_start :-
   repeat, read_next(Next),
   ty_process(Next), !.

    rocess(X <-> Y) :- !,
   add_rule(X,Y), fail.
y_process(include(F)) :-
   seeing(Old),
   see(F),
   repeat, read(type_hierarchy), !,
   repeat, read(T),
   (load_finish(T);(ty_process(T),fail)),
   !, seen, see(Old), fail.
y_process(Fin) :-
   load_finish(Fin), !,
   finish_types,
   load_start(Fin).
y_process(Garb) :-
   errmess('Invalid type specification',Garb), fail.

inish_types :-
   rewrite_types,
   do_basic_types,
   do_derivec_types,
   remove_rules,
   fail.
finish_types.

/* Rewrite decomposition of derived types */

rewrite_types :-
   repeat, rewrite_a_type, !.

rewrite_a_type :-
   or_rule(X,RHS1),
   and_rule(X,RHS2), !,
   basify(RHS2,entity,BasRHS),
   BasRHS = Bas&RHS21,
   remove_rule(X,RHS1),
   hidden_ors(RHS1,RHS11),
   add_rule(Bas,RHS11),
   add_hidden_ands(RHS11,RHS21), !,
   fail.
rewrite_a_type.

    asify(A&B,Sofar,New) :- !,
      basify(A,Sofar,Sofar1),
      basify(B,Sofar1,New).
   basify(A,Sofar,New) :-
      and_rule(A,RHS), !,
      basify(RHS,Sofar,New).
   basify(A,Sofar,Sofar) :-
      subtype(Sofar,A), !.
   basify(A,Sofar,A&Sofar).

   hidden_ors(A#B,hidden(A)#C) :- !, hidden_ors(B,C).
   hidden_ors(A,hidden(A)).

   add_hidden_ands(hidden(A)#B,Ands) :- !,
      add_rule(A,hidden(A)&Ands),
      add_hidden_ands(B,Ands).
   add_hidden_ands(hidden(A),Ands) :-
      add_rule(A,hidden(A)&Ands).

/   .eal with and/or tree of basic types */

o_basic_types :-
   treep(entity,Patt,Patt).

reep(Type,Patt,Pattarg) :-
   findall(RHS,or_rule(Type,RHS),List),
   length(List,Arity),
   type_name(Type,Name),
   functor(Pattarg,Name,Arity),
   maketype(Type,Patt),
   ty_nmember(List,N,RHS1),
   arg(N,Pattarg,NewPattarg),
   subtype(RHS1,Type1),
   treep(Type1,Patt,NewPattarg),
   fail.
reep(_,_,_).

ype_name(hidden(A),A) :- !.
```

```
type_name(A,A).

ty_nmember([A|_],1,A).
ty_nmember([_|L],N,A) :- ty_nmember(L,N1,A), N is N1+1.

subtype(A#B,C) :- subtype(A,C).
subtype(A#B,C) :- !, subtype(B,C).
subtype(A,A).

/* Dealing with derived types */

do_derived_types :-
   and_rule(X,RHS),
   basify(RHS,entity,RHS1),
   ty_intersect(RHS1,Res),
   maketype(X,Res),
   fail.
do_derived_types.

ty_intersect(A&B,Res) :- !, ty_intersect(A,Res), ty_intersect(B,Res).
ty_intersect(A,Res) :- type_pattern(A,Res), !.

/*   etrieving rules */

or_rule(LHS,RHS) :- !, call(LHS <-> RHS), RHS = _#_.
and_rule(LHS,RHS) :- !, call(LHS <-> RHS), RHS \= _#_.

add_rule(LHS,RHS) :- assertz(LHS <-> RHS), !.
remove_rule(LHS,RHS) :- retract(LHS <-> RHS), !.
remove_rules :- abolish(<->,2).

/* Information about types */

maketype(hidden(T),Patt) :-
   recorded(T,hi_patt(_),P), erase(P), fail.
maketype(hidden(T),Patt) :- !,
   recorda(T,hi_patt(Patt),_).
maketype(Type,Patt) :-
   recorded(Type,ty_patt(_),P), erase(P), fail.
maketype(Type,Patt) :-
    recorda(Type,ty_patt(Patt),_).

type_pattern(hidden(T),Patt) :- recorded(T,hi_patt(Patt),_), !.
type_pattern(Type,Patt) :- recorded(Type,ty_patt(Patt),_), !.
type_pattern(Type,_) :- errmess('Undefined type',Type), fail.
```

```
/* ERR.PL  :  Error messages etc.

                                                   Lawrence
                                                   Updated: 14 June 81
*/

/* EXPORT */

  :- public      errmess/1,
                 errmess/2.


/* MODES */

  :- mode        errmess(+),
                 errmess(+,+).



                          % Give error messages

errmess(Mess)
     :- ttynl, display('** '),
        display(Mess), ttynl.



errmess(Mess,X)
     :- ttynl, display('** '),
        display(Mess), display(' : '),
        ttyprint(X),
        ttynl.
```

```
/* POLICE.PL : Invarient enforcement agency

                                                                   Lawrence
                                                                   Updated: 5 July 81
*/

 /* EXPORT */

   :- public      must_be_term/2,
                  must_be_ground/2.


/* IMPORT */
/*
                  ground/1                from  BOUND
*/


 /* MODES */

   :- mode        must_be_term(?,?),
                  must_be_ground(?,?).


                          % Check for term

must_be_term(X,Where)
     :- ( var(X) ; atomic(X) ),
        !,
        error('Must be term (in %w): %t',[Where,X],break),
        fail.

must_be_term(_,_).



                          % Check for being ground

must_be_ground(X,_)
     :- ground(X),
        !.

must_be_ground(X,Where)
     :- error('Must be ground (in %w): %t',[Where,X],break),
        fail.
```

```
/* MECHOU.PL : Odd utilities currently specific to Mecho

                                                    Lawrence
                                                    Updated: 8 December 81

*/

 /* EXPORT */

  :- public      not_member/2,
                 two_in/3,
                 problem/0,
                 succ/2.


/* MODES */

  :- mode        not_member(+,+),
                 two_in(?,?,+).


                           % X is not a member of a Set (list)

not_member(X,[]) :- !.

not_member(X,[X|_]) :- !, fail.

not_member(X,[_|Rest]) :- not_member(X,Rest).


                           % Check for two occurances in a list

twoin(One,Two,[One|Rest]) :- !, memberchk(Two,Rest).

twoin(One,Two,[_|Rest]) :- twoin(One,Two,Rest).


                           % Piece of Junk to show current problem

problem :-
        known( problem(File,Format,List) ),
        writef('\nProblem from file : %w\n\n',[File]),
        writef(Format,List).


                           % Arithmetic
                           %  Note that at least one arg must be bound.

succ(N,N1) :- integer(N), !, N1 is N+1.

succ(N,N1) :- integer(N1), !, N is N1-1.
```

```
/* WDSIN : Produce a set of all the non constant symbols in a term

                                                    Lawrence
                                                    Updated: 1 June 81
*/

/* EXPORT */

  :- public     wordsin/2.


/* IMPORT */
/*
                memberchk/2             Utility

                number/1                from LONG (or dummied elswhere)

                const/1                 <database>
*/

/* MODES */

  :- mode       wordsin(+,?),
                wordsin(+,+,-),
                wordsin_term(+,+,+,-),
                addword(+,+,-).


                        % Wordsin in a term - entry point

wordsin(Term,Set)
     :- wordsin(Term,[],Ans),
        Ans = Set.                      % safety


                        % Implementation using accumulator

wordsin(C,Set,Set) :- constant(C), !.

wordsin(A,Sofar,Set)
     :- atomic(A),
        !,
        addword(A,Sofar,Set).

wordsin(Term,Sofar,Set)
     :- % not atomic(Term),
        functor(Term,_,Arity),
        wordsin_term(Arity,Term,Sofar,Set).



                        % Add a word to set
                        %  Use unification for equality (ie assume all ground)

addword(Word,Set,Set) :- memberchk(Word,Set), !.

addword(Word,Set,[Word|Set]).
```

```
                                % Traverse a term (right to left)

wordsin_term(1,Term,Sofar,Set)          % NB Arity always >= 1 to start with
     :- !,
        arg(1,Term,Arg),
        wordsin(Arg,Sofar,Set).

wordsin_term(N,Term,Sofar,Set)
     :- arg(N,Term,Arg),
        wordsin(Arg,Sofar,More),
        N1 is N-1,
        wordsin_term(N1,Term,More,Set).


                                % What it means to be constant  (indeed)
                                %  We don't expect variables but they are included
                                %  for safety. number/1 will include both integers
                                %  and rationals (if present - otherwise define
                                %  numbers to just be integers).

constant(C)
     :- ( var~~iable~~(C) ; number(C) ; const(C) ),
        !.
```

```
/* OK : Produce core images - with banners

                                                                Lawrence
                                                                Updated: 6 July 81
*/


ok    :- core_image,
         display('Mecho Problem Solver'), ttynl,
         reinitialise.


ok(Str)
       :- core_image,
          display('Mecho Problem Solver'), ttynl,
          display(Str), ttynl,
          reinitialise.
```

```
/* HACKS. : Various things

                                                Lawrence
                                                Updated: 11 December 81
*/



                        % Const things don't get picked up by wordsin and thus
                        %  don't get solved for by marples.

const(X) :- ncc constant(X), !.
```

```
/* LOOK : Peer at things recorded in the database

                                                Lawrence
                                                Updated: 30 November 81
*/
                              310
  :- op(300,fx,look).

                         % Look under a specific key
                         % Only print key if there is something there
                         %  Key is either Pred/Arity
                         %   or the Key itself (eg atom or general term).
                         %   NB: /(_,_) will be a difficult key to use!

look(Pred/Arity) :-
        !,
        functor(Key,Pred,Arity),
        look(Key).

   k(Key)
     :- recorded(Key,Thing,_),
        !,
        nl,
        put("{"), tab(1), write(Key), tab(1), put("}"), nl,
        look2(Key).

look(_).


look2(Key)
     :- recorded(Key,Thing,_),
        nl,
        look_show(Thing),
        fail.

look2(_).

                         % Look at all things in database

lookall
     :- current_functor(_,Key),
        look(Key),
        fail.

lookall :- lookrules.

                         % lookall into a file

lookall(File)
     :- telling(Old),
        tell(File),
        lookall,
        told,
        tell(Old).
```

```
                         % Look for ruleform counter and if there show that
                         %  many rules.

lookrules
     :- recorded(ruleform,ruleform(Counter),_),     % low level - may change
        !,
        lookrules(1,Counter).

lookrules.



lookrules(N,Max) :- N > Max, !.

lookrules(N,Max)
     :- look(N),
        N1 is N+1,
        lookrules(N1,Max).


                         % How to display a recorded thing

look_show(Thing)
     :- ( var(Thing)  ;  atomic(Thing) ),
        !,
        tab(2), print(Thing), nl.

look_show(Thing)
     :- functor(Thing,Fn,Arity),
        tab(2), write(Fn), put("("), nl,
        look_show_args(1,Arity,Thing),
        tab(2), put(")"), nl.



look_show_args(N,N,Thing)
     :- !,
        arg(N,Thing,LastArg),
        tab(8), print(LastArg), nl.

look_show_args(N,Arity,Thing)
     :- arg(N,Thing,Arg),
        tab(8), print(Arg), put(","), nl,
        N1 is N+1,
        look_show_args(N1,Arity,Thing).
```