Semantic Interpreter

1) INTERP ~ load file

various odd convenience files

---

2) Prolog code for INTERP

(layed out in the order given in 'INTERP' (sect 1))

---

3) *.SIT files

---

4) Various additional code files

```
/* INTERP
   Loading the Semantic Interpreter
   C.M., 27/7/81
*/

   % ----------------------------------------
   % First, deal with operator declarations
   % ----------------------------------------

:- [
        'ops.pl'                    % For semantic interpreter
   ].

:- compile( [

   % ------------------------------------
   % Basic datastructures and operations
   % ------------------------------------

        'env.cpl',                  % Operations on the "environment" parameter
        'task.cpl',                 % Task construction and accessing
        'obj.cpl',                  % Basic operations on objects
        'bind.cpl',                 % Keeping track of variable bindings
        'index.cpl',                % Indexed database facility
        'undo.cpl',                 % Database "undo" facility

   % ------------------------------------
   % Top level of the inference system
   % ------------------------------------

        'topdb.cpl',                % Top level interface
        'infer.cpl',                % Underlying backwards inference
        'forwar.cpl',               % Top level of forwards inference
        'demon.cpl',                % Forwards inference facility
        'create.cpl',               % Creation of object tokens
        'silly.cpl',                % Rejection of silly assertions
        'predic.cpl',               % Prediction
        'dbase.cpl',                % Top level of database
        'assum.cpl',                % Default assumptions
        'meta.cpl',                 % Picking strategies for tasks
        'nudge.cpl',                % Forcing held constraints

   % ------------------
   % Compiled utilities
   % ------------------

        'map.cpl',                  % Mapping over object-level assertions
        'invoca.cpl',               % Invocation routines
        'portra.cpl',               % Special version of 'portray'
        'listro.cpl',               % List manipulation utilities
        'util:writef.pl',           % 'writef' utility
        'util:trace.pl',            % Tracing utilities
        'util:flagro.pl',           % Flag operations (only for 'tlim'?)
        'util:files.pl',            % File existence checking routines
        'util:cmisce.pl',           % Misc compiled utilities
        'util.cpl',                 % Yet more miscellaneous utilities

   % ------------------------------------
   % Expanding meanings applied to sets
   % ------------------------------------
```

```
        'distri.cpl',           % Expanding operations on sets
        'expand.cpl',           % Expand an assertion with lists as arguments

    % ----------------------------------
    % Meta level database and properties
    % ----------------------------------

        'metami.cpl',           % Misc meta-level properties
        'types.cpl',            % Inference in type hierarchies
        'plcode:preds.pl',      % Simple meta-level properties
        'plcode:mlprp1.pl',     % Interface to predicate library
        'plcode:mlface.pl',     % meta-level properties (PLIB interface)
        'plcode:meta.pl',       % meta-level declarations
        '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:kstype.pl',     % KS type definitions
        'plcode:t1.pl',
        'plcode:t2.pl',
        'plcode:t3.pl',
        'plcode:t4.pl',
        'plcode:tyload.pl',     % How to load type hierarchies

        'plcode:err.pl'         % Error messages
        ]).

    % ---------------------
    % Interpreted Utilities
    % ---------------------

:- [    'util:applic.pl',       %
        'util:imisce.pl',       % Misc interpreted utilities
        'util:edit.pl',         % Editing facilities
        'util.pl'               % Misc utilities
    ].

 - ks_init.                     % Initialise KS system

    % ----------------------------------------
    % Now load in the object-level predicates
    % ----------------------------------------

:- load( [
        'plib:time.def',        % Time
        'plib:space.def',       % Space
        'plib:sollin.def',      % Solid lines
        'plib:contac.def',      % Contact
        'plib:objp.def',        % Properties of objects
        'plib:objr.def',        % Relations between objects
        'plib:motion.def',      % Motion of objects
        'plib:units.def',       % Measures and Units
        'plib:ages.def',        % Odd things for AGE problems
        'plib:nlpred.def',      % Odd hacks for NLU
        'nltype.hi'             % Types (includes plib:types.hi)
      ]).
```

```
:- [    'operat.pl',            % Prediction operators
        'schema.pl',            % Forwards inference rules
        'alt.pl'                % Alternative axiomatisations
   ].

:- do_schemas.                  % Set up forwards rules

   % ---------------------------------
   % Semantic Interpretation Routines
   % ---------------------------------

:- [    'init.pl',              % Top level of interpreter
        'ters.pl',              % Interpretation of sentences
        'ternp.pl',             % Interpretation of noun phrases
        'terpp.pl',             % Interpretation of prepositional phrases
        'meas.pl',              % Interpretation of various measure phrases
        'teradv.pl',            % Interpretation of adverbs
        'tmods.pl',             % Interpretation of time modifiers
        'dict.pl',              % Dictionary
        'interf.pl'             % Interface with syntax
   ].

   % -------------
   % Verb routines
   % -------------

:- [    'verbs.pl',             % The routines themselves
        'versup.pl'             % Support routines
   ].

   % ---------------------------------
   % Expansion of underlying meanings
   % ---------------------------------

:- [    'mean1.pl',             % General meanings
        'mean2.pl',             % Simple word meanings
        'time.pl'               % Basic operations on times
   ].

   % -------------------------
   % Actions at the meta-level
   % -------------------------

:- [    'metact.pl',            % Certain meta-level actions
        'mlxpro.pl',            % Extra uniqueness properties
        'metadb.pl',            % Meta level database
        'filter.pl'             % Filtering output
   ].

:- clean_db.                    % Initialise database
:- setfile.                     % Make up save file name
:- tt.                          % Set medium level of tracing
```

FILES

ops.pl
env.cpl
task.cpl
obj.cpl
bind.cpl
index.cpl
undo.cpl
topdb.cpl
infer.cpl
forwar.cpl
demon.cpl
create.cpl
silly.cpl
predic.cpl
dbase.cpl
assum.cpl
meta.cpl
nudge.cpl
map.cpl
invoca.cpl
portra.cpl
 istro.cpl
util.cpl
distri.cpl
expand.cpl
metami.cpl
types.cpl
util.pl ← alpha.h
operat.pl
schema.pl
alt.pl
init.pl
ters.pl
ternp.pl
terpp.pl
meas.pl
teradv.pl
tmods.pl
dict.pl
interf.pl
 erbs.pl
 ersup.pl
mean1.pl
mean2.pl
time.pl
metact.pl
mlxpro.pl
metadb.pl
filter.pl

```
; INTERP.MIC  -  Load Interp     '<silence>
;
;       This junk allows for automatic loadins believe it or not
;
;       Call as:        /interp          - to load interp (normal use)
;                       /interp auto     - used by MAKSYS
;
.on error:backto death
.error ?
.on operator:backto death
.operator !
.goto cont
death::
*^C
*^C
.if ($a = "auto") .let e1 = "error"
! INTERP.MIC HALTED
.mic return
cont::
;
;                                   Use latest version of Prolog
 cd [400,434,pros] <005>
.run prolog[400,444]  '<revive>
* :- [interp].
* :- init.
.save interp.exe[400,444]
.cd -
```

PATHS.CCL [400,434]

```
sys: = [400,444], usr:, dske:[1,5], dske:[1,4], pop:
sen: = [400,434,sen]
syn: = [400,434,syn]
prb: = [400,445,prb]
winsyn: = [400,434,syn,winsyn]
parse: = [400,434,parse]
pros: = [400,434,pros]
papers: = [400,434,papers]
home: = [400,434]
@mecho.pth[400,444]
```

```
/* INTERP.DEF
   Extras for the semantic interpreter
*/

known(writef(_,_),'util:writef.pl').
known(error(_,_,_),'util:trace.pl').
known(tlim(_),'util:trace.pl').
known(ton(_),'util:trace.pl').
known(toff(_),'util:trace.pl').
known(toff,'util:trace.pl').
known(trace(_,_,_),'util:trace.pl').

known(file_exists(_),'util:files.pl').

known(gensym(_,_),'util:cmisce.pl').

known(checklist(_,_),'util:applic.pl').

known(continue,'util:imisce.pl').
known(\=(_,_),'util:imisce.pl').


known(load(_),'plcode:*.pl').
known(type_pattern(_,_),'plcode:*.pl').

known(same_predicate(_,_,_),'plcode:preds.pl').
known(type_predicate(_),'plcode:preds.pl').
known(type_predicate(_,_,_),'plcode:preds.pl').
known(copy_args(_,_,_),'plcode:preds.pl').

known(exists(_,_,_,_,_),'plcode:mlprp1.pl').
known(exists(_,_,_),'plcode:mlprp1.pl').
known(unique(_,_,_,_,_),'plcode:mlprp1.pl').
known(unique(_,_,_),'plcode:mlprp1.pl').
known(function(_,_,_,_,_),'plcode:mlprp1.pl').
known(commutative(_,_,_),'plcode:mlprp1.pl').
known(aliorelative(_,_,_),'plcode:mlprp1.pl').

known(function_pattern(_,_,_),'plcode:mlface.pl').
known(exists_pattern(_,_,_),'plcode:mlface.pl').
known(unique_pattern(_,_,_),'plcode:mlface.pl').
known(normal_form(_,_),'plcode:mlface.pl').
known(object_level_rule(_,_),'plcode:mlface.pl').
known(object_level_neg_rule(_,_),'plcode:mlface.pl').
known(default_rule(_,_),'plcode:mlface.pl').
known(argument_names(_),'plcode:mlface.pl').
known(argument_types(_),'plcode:mlface.pl').

applies(error(A,B,C),C).
applies(checklist(A,B),A+1).
applies(findall(A,B,C),B).
applies(not(A),A).
applies(appars(A,B),A+1).
applies(mapars(A,B,C),A+2).
applies(appconj(A,B),A+1).
applies(mapconj(A,B,C),A+2).
applies(special_test(_,_,_,A),A).

called(do_schemas).
called(help).
```

```
called(hit(_)).
called(init).
called(portray(_)).
called(rules).
called(run(_)).
called(t).
called(tt).
called(ttt).
called(unhit(_)).
called(-(_)).
called(o(_)).
```

WW.MIC

```
.cd syn
.copy winsyn:'A.syn = 'A.syn
.delete 'A.syn
.cd pros
```

```
/* OPS

   Operator declarations for the semantic interpreter
   C.M., 1/9/81
*/

/* For printing out constrained variables */

:- op(45,fx,$).


/* For predicate library entries */

:- op(400,fx,define).

/* For indicating assertions to be got from the parser */

:- op(45,fx,'@@').
:- op(45,fx,'@').

/* For certain utilities */

 - op(700,xfx,\=).
:- op(34,fx,o).

/* For inference rules of various sorts */

% Main connectives:

:- op(1160,xfx,'==>').                  % Word meaning rewrite
:- op(1160,xfx,[-->,<-->,<--]).         % Implications
:- op(1160,xfx,<-).                     % Implication for forwards rules
:- op(1160,xfx,<=).                     % Implication for operators
:- op(1160,xfx,<->).                    % Type definition

% Rule names:

:- op(905,xfx,<<).                      % To separate off rule name

% Lambda expressions:

 - op(900,xfx,:).
:- op(850,fx,'\').

% Logical connectives

:- op(800,xfx,where).

:- op(750,xfy,#).
:- op(750,xfy,&).

:- op(300,fx,'~').
```

```
/* ENV
   Basic accessing of things in the environment parameter
   C.M., 6/8/81
   This is the only file that knows what 'Env' looks like
*/

%declarations%

:- public
                current_time/2,
                cvars/2,
                definite/1,
                firstenv/1,
                indefinite/1,
                isenv/1,
                newenv/3,
                nextenv/2,
                objects/2,
                recordedenv/1,
                recordenv/1,
                remove_envs/0,
                t_focus/3,
                tense/2,
                updef/3.

% imports:
%               new_evar/2              (from obj.cpl)
%               rewrite_cvars/3         (from obj.cpl)

%end%

 :- mode isenv(+).
 :- mode firstenv(-).
 :- mode nextenv(+,-).
 :- mode newenv(+,+,-).
 :- mode updef(+,+,-).
 :- mode objects(+,-).
 :- mode definite(+).
 :- mode indefinite(+).
 :- mode top_of(+,-).
 :- mode re_top(+,+,-).
 :- mode e_main(+,-).
 :- mode cvars(+,-).
 :- mode current_time(+,?).
 :- mode recordenv(+).
 :- mode recordedenv(-).
 :- mode tense(+,?).
 :- mode t_focus(+,+,-).
 :- mode sc_env(+,-).

/* Structure of the 'environment' parameter.
   The structure has main functor '/', with two parameters:

      1. The 'top'. This represents the transient features of
         the environment, and is a structure with functor 'top'
         and 2 arguemts:

         1. An atom specifying 'definite' or 'indefinite'

         2. Something representing the current time moment/period
```

2. The main part of the environment. This has functor 'env' and arity 4, the arguments being:

   1. A functor of big arity, effectively an array, where each argument is an entry recording the values and suspended actions associated with a constrained variable.

   2. A list of the objects that have been encountered so far with their types as ascertained so far.

   3. An atom giving the tense of the current clause.

   4. A structure giving the time foci (past, present and future).

*/

```
/* Recognising an environment */

isenv(top(_,_)/env(_,_,_,_)).

/* Creating an environment for a S */

firstenv(Env) :-
   Env=top(indef,_)/env(Cv,_,_,tfoc(_,_,_)),
   new_evar(C1,Env),
   current_time(Env,C1).

nextenv(Env,top(indef,M)/env(V,R,_,F)) :-
   Env = top(_,C)/env(V,R,_,F),
   new_evar(M,Env), !.

/* Creating environments with differences in def/indef */

updef(top(_,M)/E,Def,top(Def,M)/E) :- !.

newenv(E,def,E1) :- !, updef(E,def,E1).
newenv(E,indef,E).

/* Definite and indefinite */

definite(top(def,_)/_) :- !.
indefinite(top(indef,_)/_) :- !.

/* Things with the "top" */

current_time(T/_,C) :- !, arg(2,T,C).

top_of(T/_,T) :- !.
re_top(T,_/E,T/E) :- !.

/* Accessing the main fields */

e_main(_/E,E) :- !.

cvars(Env,C) :- !, e_main(Env,M), arg(1,M,C).
objects(Env,R) :- !, e_main(Env,M), arg(2,M,R).
tense(Env,T) :- !, e_main(Env,M), arg(3,M,T).
t_foci(Env,F) :- !, e_main(Env,M), arg(4,M,F).

/* Time foci */
```

```
t_focus(past,Env,List) :- !, t_foci(Env,F), args(1,F,List).
t_focus(pres,Env,List) :- !, t_foci(Env,F), args(2,F,List).
t_focus(future,Env,List) :- !, t_foci(Env,F), args(3,F,List).

/* Recording environments */

recordenv(Env) :- !,
   sc_env(Env,Env1),
   recorda(env,Env1,_), !.

recordedenv(Env) :- !,
   recorded(env,Env,P), !.

remove_envs :-
   recorded(env,X,P), erase(P), fail.
remove_envs.

/* Garbage collect an environment */

sc_env(Top/env(Cv,M,T,TF),Top/env(Cv1,M,T,TF)) :-
   rewrite_cvars(0,Cv,Cv1).
```

```
/* TASK
   Basic operations on tasks to be performed
   C.M.,  24/7/81
*/

%declarations%

:- public
                action/2,
                hold_task/3,
                mark_done/1,
                newtask/3,
                proposition/2,
                undone/1.

% imports:
%               demons/3                (from obj.cpl)
%               sappend/2               (from bind.cpl)
%               satisfy/3               (from topdb.cpl)
%               trace/3                 (from util:trace.pl)
%               unbound/3               (from bind.cpl)

%end%

:- mode
        newtask(+,+,-),
        mark_done(+),
        undone(+),
        mark_held(+),
        held(+),
        action(+,?),
        proposition(+,?),
        hold_task(+,+,+),
        holdall(+,+,+,+,+),
        wait(?,+,+,+).

/* Creating a task */

newtask(Act,Ass,task(Act,Ass,Undone,Notheld)).

 * Accessing the components */

action(task(Act,_,_,_),Act).
proposition(task(_,Ass,_,_),Ass).

/* Affecting the 'done' flag */

undone(task(_,_,F,_)) :- var(F).

mark_done(task(_,_,done,_)).

/* Affecting the 'held' flag */

held(task(_,_,_,F)) :- nonvar(F).

mark_held(task(_,_,_,held)).

/* Suspending a task */

hold_task(Task,Ans,Env) :-
```

```
    held(Task), !.
hold_task(Task,Ans,Env) :-
    mark_held(Task),
    proposition(Task,Ass),
    action(Task,Act),
    trace('Holding action %p on %p\n',[Act,Ass],hold),
    functor(Ass,F,N),
    holdall(N,Ass,Task,Ans,Env).

    holdall(0,_,_,_,_) :- !.
    holdall(N,Ass,W,Ans,Env) :-
        ars(N,Ass,A), unbound(A,any,Env), !,
        wait(A,W,Ans,Env),
        N1 is N-1, holdall(N1,Ass,W,Ans,Env).
    holdall(N,Ass,W,Ans,Env) :-
        N1 is N-1, holdall(N1,Ass,W,Ans,Env).

/* Hang constraints off a single variable */

wait(W,P,Ans,Env) :-
    unbound(W,any,Env), !,
    demons(W,Env,D),
    sappend(P,D).
wait(W,P,Ans,Env) :-
    satisfy(P,Ans,Env).
```

```
/* OBJ
   Objects and their components
   C.M., 3/9/81
*/

%declarations%

:- public
                a_constant/2,
                demons/3,
                fix/2,
                iscvar/1,
                known/2,
                mark_known/2,
                mark_ref/2,
                new_evar/2,
                obj_bind_info/3,
                obj_type/3,
                prevent_bind/3,
                ref/2,
                rewrite_cvars/3,
                skolemise/2,
                universal/2,
                universalise/2.

% imports:
%               '$diff' /1                  (from undefined)
%               bound/2                   (from bind.cpl)
%               cvars/2                   (from env.cpl)
%               error/3                   (from util:trace.pl)
%               lastof/2                  (from listro.cpl)
%               ndsin/2                   (from listro.cpl)
%               objects/2                 (from env.cpl)

%end%

:- mode new_evar(-,+),
        new_uvar(-,+),
        iscvar(+),
        demons(+,+,-),
        a_constant(+,-),
        obj_entry(+,+,-),
        obj_bind_info(+,+,?),
        obj_type(+,+,?),
        obj_diffs(+,+,?),
        prevent_bind(+,+,+),
        universal(+,+).

/* Information kept with an object:
   An object is either a constant or a constrained variable.
   A constrained variable is represented by an entry in the
   environment, which consists of a "type" indication and
   a variable-terminated list of actions
   to be carried out when the variable becomes instantiated.
   The type indication has 5 components:

   1. instantiated or not, according to whether the variable is
      "existential" or "universal"
   2. instantiated or not, according to whether the variable is
      "known" or not
```

```
   3. instantiated or not, according to whether the variable is
      a "referent" or not
   4. a type pattern (maintained by 'types.cpl' etc)
   5. a pattern that will not match with the corresponding pattern
      of objects supposed to be different

The variable is represented by a term $(N), where N is the integer
specifying where the entry appears in the environment.

A constant only has the last two components
*/

/* Creation functions */

new_evar($N,Env) :- setwaits(N), mark_existential($N,Env).

new_uvar($N,Env) :- setwaits(N), mark_universal($N,Env).

   setwaits(N) :-
      retract(waits(N1)), !,
      N is N1+1,
      assert(waits(N)).
   setwaits(1) :-
      assert(waits(1)).

/* Enumerating constants */

a_constant(Env,I) :-
   objects(Env,Objs), ndsin(obj(I,_),Objs).

/* Getting the info kept about an object */

obj_entry($N,Env,Ent) :- !,
   cvars(Env,Cv), functor(Cv,cv,101), varentry1(N,Cv,Ent).
obj_entry(Obj,Env,Ent) :-
   objects(Env,Objs),
   ndsin(obj(Obj,Ent),Objs), !.
obj_entry(Obj,Env,Ent) :-
   objects(Env,Objs),
   lastof(Objs,obj(Obj,Ent)).

   varentry1(N,Cv,Ent) :- N<101, !,
      arg(N,Cv,Ent), !.
   varentry1(N,Cv,Ent) :-
      N1 is N-100,
      arg(101,Cv,Cv1),
      functor(Cv1,cv,101),
      varentry1(N1,Cv1,Ent).

/* Looking inside the object entry */

% Info to be unified on bindings

obj_bind_info(I,Env,Bind) :-
   obj_entry(I,Env,Ent),
   functor(Ent,o,2), arg(1,Ent,Bind).

% Term representing the type info

obj_type(I,Env,Patt) :-
```

```
   obj_bind_info(I,Env,Bind), functor(Bind,b,3), arg(2,Bind,Patt).

% Term distinguishing it from different objects

obj_diffs(I,Env,Diffs) :-
   obj_bind_info(I,Env,Bind), functor(Bind,b,3), arg(3,Bind,Diffs).

% Actions to be performed when it gets bound (cvar only)

demons(I,Env,Dem) :-
   obj_entry(I,Env,Ent), functor(Ent,o,2), arg(2,Ent,Dem).

% Extra bits for cvars

vartypes(Cv,Env,Ty) :-
   iscvar(Cv), obj_bind_info(Cv,Env,Bind),
   functor(Bind,b,3), arg(1,Bind,Ty), functor(Ty,v,3).

mark_existential(Cv,Env) :- vartypes(Cv,Env,Ty), arg(1,Ty,exists).

mark_universal(Cv,Env) :- universal(Cv,Env).
universal(Cv,Env) :- vartypes(Cv,Env,Ty), arg(1,Ty,V), var(V).

mark_known(Cv,Env) :- vartypes(Cv,Env,Ty), arg(2,Ty,known).
known(Cv,Env) :- vartypes(Cv,Env,Ty), arg(2,Ty,K), nonvar(K).

mark_ref(Cv,Env) :- vartypes(Cv,Env,Ty), arg(3,Ty,ref).
ref(Cv,Env) :- vartypes(Cv,Env,Ty), arg(3,Ty,K), nonvar(K).

/* Prevent two objects from being bound together */

prevent_bind(A,B,Env) :-
   obj_diffs(A,Env,Diffs1),
   obj_diffs(B,Env,Diffs2),
   new_diff_arg(N),
   diff_patts(N,Diffs1,Diffs2,D11,D22),
   arg(1,D11,1), arg(1,D22,2).

diff_patts(0,D1,D2,D1,D2) :- !,
   functor(D1,diff,3),
   functor(D2,diff,3).
diff_patts(N,D1,D2,D111,D222) :-
   N1 is N mod 2, N2 is N/2,
   diff_patts(N2,D1,D2,D11,D22),
   discrim(N1,D11,D111),
   discrim(N1,D22,D222).

discrim(0,Diff,Diff1) :- !, arg(2,Diff,Diff1), functor(Diff1,diff,3).
discrim(1,Diff,Diff1) :- !, arg(3,Diff,Diff1), functor(Diff1,diff,3).

new_diff_arg(N) :-
   retract('$diff'(N)), !, N1 is N+1, asserta('$diff'(N1)).
new_diff_arg(1) :-
   asserta('$diff'(2)).

/* Recognising function  for constrained vars */

iscvar($_).

/* Skolemise an entity, changing variables into evars */
```

```
skolemise(S,Env) :- var(S), !,
   new_evar(S,Env).
skolemise(_,_) :- !.

/* Change variables into uvars */

universalise(S,Env) :- var(S), !,
   new_uvar(S,Env).
universalise(_,_) :- !.

/* Fix any universal variables */

fix(A,Env) :- nonvar(A), mark_existential(A,Env), !.
fix(_,_).

/* Garbage collect information about cvars */

rewrite_cvars(_,Old,Old) :- var(Old), !.
rewrite_cvars(N,Old,New) :-
   functor(Old,cv,101), functor(New,cv,101),
   r_c1(100,N,Old,New),
   ars(101,Old,Old1), ars(101,New,New1),
   N1 is N+101,
   rewrite_cvars(N1,Old1,New1).

r_c1(0,_,_,_) :- !.
r_c1(N,Sofar,Old,New) :-
   ars(N,Old,Oars), ars(N,New,Nars),
   Realno is N+Sofar,
   bound($Realno,Z),
   r_c2(Z,Oars,Nars),
   N1 is N-1,
   r_c1(N1,Sofar,Old,New).

r_c2(Z,Old,New) :- iscvar(Z), functor(Old,o,2), ars(1,Old,Dem), var(Dem), !.
r_c2(Z,Old,Old) :- iscvar(Z), !.
r_c2(_,Old,New).
```

```
/* BIND
   Keeping track of variable bindings
   C.M., 27/7/81
*/

%declarations%

:- public
                bind/5,
                bound/2,
                bound/3,
                gappend/2,
                matchable/4,
                possinstance/4,
                pure/3,
                unbound/3,
                unify/5.

% imports:
%               ase/1                   (from undo.cpl)
%               appars/2                (from map.cpl)
                assumption/1            (from undo.cpl)
%               bind_check/3            (from metami.cpl)
%               demons/3                (from obj.cpl)
%               index_add/2             (from index.cpl)
%               index_find/3            (from index.cpl)
%               iscvar/1                (from obj.cpl)
%               list/3                  (from listro.cpl)
%               makein/2                (from listro.cpl)
%               mapars/3                (from map.cpl)
%               obj_bind_info/3         (from obj.cpl)
%               satisfy/3               (from topdb.cpl)
%               universal/2             (from obj.cpl)

%end%

:- mode bound(?,?),
        unbound(?,+,+),
        bound(?,+,+),
        satisfy_all(+,+),
        bindins(+,+,+,+,-),
        bindinss(+,+,+,+,+,-),
        allunbound(+,+,+,+),
        allbound(+,+,+,+),
        unify(+,+,+,+),
        key(+,-),
        bind(+,?,?,+).

/* What is the root of the equivalence class containing A? */

bound(A,B) :- var(A), !, A=B.
bound(A,C) :- b_recorded(A,B), !, bound(B,C).
bound(A,A).

unbound(V,_,_) :- var(V), !.
unbound(V,S,Env) :- bound(V,R), matchable(R,S,Env).

   matchable(Cv,any,_) :- !, iscvar(Cv).
   matchable(Cv,strict,Env) :- iscvar(Cv), universal(Cv,Env).
```

```
allunbound([],_,_,_) :- !.
allunbound([N|Ns],Ass,S,Env) :- !,
   ars(N,Ass,A), unbound(A,S,Env), allunbound(Ns,Ass,S,Env).

bound(V,S,Env) :- unbound(V,S,Env), !, fail.
bound(_,_,_).

allbound([],_,_,_) :- !.
allbound([N|Ns],Ass,S,Env) :-
   ars(N,Ass,A), bound(A,S,Env), allbound(Ns,Ass,S,Env).

pure(Ass,S,Env) :- appars(bound(S,Env),Ass).

   b_recorded(R1,R2) :- key(R1,K),
      index_find(K,_,bound(K,R2)), !.

   key($N,N) :- !.
   key(V,V).

/* Cause two things to become  bound together */

 nd(S,V1,V2,Ans,Env) :-
   bindings(S,V1,V2,Env,Dems),
   satisfy(Dems,Ans,Env).

/* Carry out all binds to make two assertions match */

unify(S,Ass1,Ass2,Ans,Env) :-
   mapars(matchable(S,Env),Ass1,Ass2),
   functor(Ass1,F,N),
   bindingss(N,S,Ass1,Ass2,Env,Sats),
   satisfy_all(Sats,Ans,Env).

   bindingss(0,_,_,_,_,[]) :- !.
   bindingss(N,S,A,B,Env,[Sa|Ss]) :-
      ars(N,A,A1), ars(N,B,B1),
      binding(S,A1,B1,Env,Sa),
      N1 is N-1, bindingss(N1,S,A,B,Env,Ss).

   satisfy_all([],_,_) :- !.
   satisfy_all([Sa|L],Ans,Env) :- var(Sa), !, satisfy_all(L,Ans,Env).
   satisfy_all([Sa|Ss],Ans,Env) :-
      satisfy(Sa,Ans,Env), satisfy_all(Ss,Ans,Env).

/* Basic binding operations */

binding(S,V1,V2,Env,Dems) :-
   bound(V1,R1), bound(V2,R2),
   bind1(S,R1,R2,Env,Dems).

   bind1(S,R,R,_,_) :- !.
   bind1(S,Cv1,Cv2,Env,_) :-
      matchable(Cv1,S,Env), matchable(Cv2,S,Env), !,
      obj_bind_info(Cv1,Env,Bind),
      obj_bind_info(Cv2,Env,Bind),
      assumption(Cv1=Cv2),
      b_record(Cv1,Cv2),
      combine_demons(Cv1,Cv2,Env).
   bind1(S,V,Cv,Env,Dems) :-
      matchable(Cv,S,Env), !,
```

```
      bind1(S,Cv,V,Env,Dems).
   bind1(S,Cv,V,Env,Dems) :-
      matchable(Cv,S,Env),
      bind_check(Cv,V,Env),
      obj_bind_info(Cv,Env,Bind),
      obj_bind_info(V,Env,Bind),
      assumption(Cv=V),
      b_record(Cv,V),
      demons(Cv,Env,Dems).

   b_record(R1,R2) :- key(R1,K), ase(N),
      index_add(keys(N,K),bound(K,R2)).

/* Combine the demons for two cvars */

combine_demons(Cv1,Cv2,Env) :-
   demons(Cv1,Env,D1), demons(Cv2,Env,D2),
   sappend(D1,D2), sappend(D2,D1),
   hole(D1,H), hole(D2,H).

sappend(New,Old) :- var(New), !.
 ppend(New,Old) :-
   list(New,First,Rest), !,
   makein(First,Old),
   sappend(Rest,Old).
sappend(New,Old) :-
   makein(New,Old).

hole(Var,Var) :- var(Var), !.
hole(List,Var) :-
   list(List,_,Rest),
   hole(Rest,Var).

/* Cound one entity be an instance of another? */

possinstance(A,B,S,Env) :- bound(A,A1), bound(B,B1), pi(A1,B1,S,Env).

   pi(X,X,_,_) :- !.
   pi(_,X,S,Env) :- matchable(X,S,Env).

 * See if two entities may be matchable
   (quick test that lets too much through) */

matchable(A,B,S,Env) :- bound(A,A1), bound(B,B1), mtch(A1,B1,S,Env).

   mtch(A,A,_,_) :- !.
   mtch(A,B,S,Env) :- matchable(A,S,Env), !.
   mtch(A,B,S,Env) :- matchable(B,S,Env), !.
```

```
/* INDEX
   Basic indexing facility for database
   C.M., 21/4/81
*/

%declarations%

:- public
                index_add/2,
                index_find/3.

%end%

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

index_add(Keys,Item) :- !,
   functor(Keys,keys,N),
   add_keys(N,Keys,Item).

   add_keys(0,_,_) :- !.
   add_keys(N,K,I) :- !,
      ars(N,K,Key),
      recordz(Key,db(K,I),_),
      N1 is N-1,
      add_keys(N1,K,I).

index_find(Key,Keys,Item) :- !,
   recorded(Key,db(Keys,Item),_).
```

```
/* UNDO
   Maintaining a database where additions can be undone
   C.M., 21/7/81
*/

%declarations%

:- public
                ase/1,
                assumption/1,
                clean_all/0,
                clean_db/0,
                db_state/1,
                restore_dbase/1.

% imports:
%               '$ase' /1                   (from undefined)
%               index_find/3            (from index.cpl)
%               remove_envs/0           (from env.cpl)
%               trace/3                 (from util:trace.pl)

 nd%

:- mode
        ase(-),
        rule_zero(-),
        dbase_zero(-),
        change_ase(+),
        db_state(-),
        restore_dbase(+),
        remove_all(+,+),
        rem_keys(+,+,+),
        assumption(+).

/* Backtrackable additions -

   'db_state(N)' says "remember the current state,
        and make N the number given to it"
   'restore_dbase(N)' says "restore the database
        to where it was at state N"
   Assertions are stored under the number of the
   last database state.
*/

ase(X) :- call('$ase'(X)), !.
ase(X) :- dbase_zero(X).

rule_zero(50000).
dbase_zero(50001).

change_ase(N) :- retract('$ase'(_)), fail.
change_ase(N) :- asserta('$ase'(N)), !.

db_state(N1) :-
   ase(N),
   N1 is N+1,
   change_ase(N1), !.

clean_db :- !,
   dbase_zero(X), restore_dbase(X),
```

```
    abolish('$diff',1),
    remove_envs.

clean_all :- !,
    rule_zero(X), restore_dbase(X),
    abolish('$diff',1),
    remove_envs.

restore_dbase(N) :-
    ase(N1),
    remove_all(N1,N),
    chanse_ase(N), !.

    remove_all(N1,N) :- N1<N, !.
    remove_all(N1,N) :-
        index_find(N1,Keys,Others),
        functor(Keys,keys,Ks),
        rem_keys(Ks,Keys,db(Keys,Others)),
        fail.
    remove_all(N1,N) :-
        N2 is N1-1,
        remove_all(N2,N).

    rem_keys(0,_,_) :- !.
    rem_keys(N,K,I) :- !,
        ars(N,K,Key), recorded(Key,I,P),
        erase(P),
        N1 is N-1, rem_keys(N1,K,I).

/* Make an assumption */

assumption(A) :- call(A), !.
assumption(A) :- !,
    trace('\nAssumins that %p\n',[A],assum),
    db_state(N),
    (true;
     (restore_dbase(N),trace('\nMaybe %p is not true\n',[A],assum),fail)).
```

```
/* TOPDB
   High level interface to database and inference
   C.M., 6/8/81
*/

%declarations%

:- public
                add/2,
                declare/2,
                satisfy/3,
                test/2,
                test/6.

% imports:
%               action/2                (from task.cpl)
%               add/6                   (from meta.cpl)
%               appars/2                (from map.cpl)
%               definite/1              (from env.cpl)
%               enter/5                 (from meta.cpl)
'               indefinite/1            (from env.cpl)
                list/3                  (from listro.cpl)
%               mapars/3                (from map.cpl)
%               meta_act/2              (from metact.pl)
%               ndsin/2                 (from listro.cpl)
%               newtask/3               (from task.cpl)
%               notsilly1/2             (from silly.cpl)
%               nudge/3                 (from nudge.cpl)
%               possinstance/4          (from bind.cpl)
%               proposition/2           (from task.cpl)
%               skolemise/2             (from obj.cpl)
%               test/8                  (from meta.cpl)
%               trace/3                 (from util:trace.pl)
%               undone/1                (from task.cpl)

%end%

:- mode
        declare(+,+),
        add(+,+),
        test(+,+),
        test(+,+,+,+,+,+),
        new_perform(+,+,+,?,?,+),
        perform(+,+,?,?,+),
        proceed(+,+,+,+,?,?,+,+),
        proceed1(+,+,+,+,?,?,+),
        loopcheck(+,+,+,-,-,+).

declare(Ass,Env) :-
   indefinite(Env), !,
   new_perform(add,Ass,Env,_,_,[]).
declare(Ass,Env) :-
   definite(Env), !,
   test(Ass,Env,deep,any,any,[]).

add(Ass,Env) :- !, new_perform(add,Ass,Env,_,_,[]).

test(Ass,Env) :- !, test(Ass,Env,deep,any,any,[]).

/* Underlying calls to inference */
```

```
test(Ass,Env,Dep,S,immed,Ans) :- !,
   new_perform(test(Dep,S),Ass,Env,H,[],Ans),
   nudge(H,[(test(Dep,S),Ass)|Ans],Env).
test(Ass,Env,Dep,S,any,Ans) :- !,
   new_perform(test(Dep,S),Ass,Env,_,_,Ans).

/* Create and perform new tasks */

new_perform(_,true,_,H,H,_) :- !.
new_perform(Act,A1&A2,Env,H3,H1,Ans) :- !,
   new_perform(Act,A1,Env,H2,H1,Ans),
   new_perform(Act,A2,Env,H3,H2,Ans).
new_perform(_,{Asss},Env,H,H,_) :- !,
   meta_act(Env,Asss).
new_perform(Act,Ass,Env,Hs,H1s,Ans) :- !,
   appars(skolemise(Env),Ass),
   notsilly1(Ass,Env),
   newtask(Act,Ass,Task),
   perform(Task,Env,Hs,H1s,Ans).

/* Satisfy a set of pending adds/tests */

satisfy(Z,Ans,Env) :-
   var(Z), !.
satisfy(List,Ans,Env) :-
   list(List,Task,L), undone(Task), !,
   perform(Task,Env,H,[],Ans),
   satisfy(L,Ans,Env).
satisfy(List,Ans,Env) :- !,
   list(List,_,L),
   satisfy(L,Ans,Env).

/* Perform an action on some assertions */

perform(Task,Env,H1,H2,Ans) :- !,
   action(Task,Act),
   proposition(Task,Ass),
   loopcheck(Act,Ass,Ans,Ans1,Env,Response),
   proceed(Response,Act,Ass,Task,H1,H2,Ans1,Env).

proceed(yes,test(_,_),_,_,_,_,_,_) :- !, fail.
proceed(yes,add,_,_,_,_,_,_) :- !.
proceed(no,Act,Ass,Task,H1,H2,Ans,Env) :- !,
   length(Ans,L),
   trace('%r%t %p\n',['.',L,Act,Ass],act),
   proceed1(Act,Ass,Env,Task,H1,H2,Ans).

proceed1(add,Ass,Env,Task,H1,H2,Ans) :- !,
   add(Ass,Env,Task,H1,H2,Ans).
proceed1(test(S,D),Ass,Env,Task,H1,H2,Ans) :- !,
   test(S,D,Ass,Env,Task,H1,H2,Ans).
proceed1(enter,Ass,Env,Task,H1,H2,Ans) :- !,
   enter(Ass,Env,Task,H1,H2).

/* Fail if we are in a loop */

loopcheck(Act,Ass,Ans,Ans,Env,yes) :-
   functor(Ass,F,N), functor(Sk,F,N),
   strictness(Act,S),
```

```
    ndsin((Act,Sk),Ans),
    mapars(possinstance(S,Env),Sk,Ass), !.
loopcheck(Act,Ass,Ans,[(Act,Ass)|Ans],_,no) :- !.

    strictness(test(_,S),S) :- !.
    strictness(_,any).
```

```
/* INFER
   Basic inference
   C.M., 21/7/81
*/

%declarations%

:- public
                infer/7,
                uniinfer/4.

% imports:
%               appars/2                (from map.cpl)
%               appconj/2               (from map.cpl)
%               bound/2                 (from bind.cpl)
%               commute_ass/2           (from metami.cpl)
%               copy_args/3             (from plcode:preds.pl)
%               findall/3               (from invoca.cpl)
%               in_dbase/3              (from dbase.cpl)
%               mapars/3                (from map.cpl)
%               normal_form/2           (from plcode:mlface.pl)
                object_level_rule/2     (from plcode:mlface.pl)
%               predict/4               (from predic.cpl)
%               test/6                  (from topdb.cpl)
%               toff/2                  (from util.pl)
%               ton/2                   (from util.pl)
%               trace/3                 (from util:trace.pl)
%               unify/5                 (from bind.cpl)
%               unique/5                (from plcode:mlprp1.pl)
%               universalise/2          (from obj.cpl)

%end%

:- mode
        infer(+,?,?,+,+,+,+),
        infer1(+,?,?,+,+,+,+),
        uniinfer(+,+,+,+).

/* Using database, prediction and object-level inference
   rules */

infer(G&Gs,H1,H3,S,D,Ans,Env) :- !,
   infer(G,H1,H2,S,D,Ans,Env),
   infer(Gs,H2,H3,S,D,Ans,Env).
infer(Goal,H1,H2,S,D,Ans,Env) :-
   commute_ass(Goal,Goal1),
   infer1(Goal1,H1,H2,S,D,Ans,Env).
infer(Goal,H1,H2,S,D,Ans,Env) :- !,
   infer1(Goal,H1,H2,S,D,Ans,Env).

infer1(Goal,H1,H2,S,D,Ans,Env) :-
   normal_form(Goal,Goals), !,
   appconj(universalise(Env),Goals),
   test(Goals,Env,D,S,immed,Ans).
infer1(Goal,H,H,S,deep,Ans,Env) :-
   predict(Goal,S,Ans,Env).
infer1(Goal,H,H,S,D,Ans,Env) :-
   in_dbase(Env,S,Goal).
infer1(Goal,H1,H1,S,D,Ans,Env) :- !,
   functor(Goal,F,N),
```

```
    functor(Patt,F,N),
    call(object_level_rule(Patt,Gs)),
    unify(S,Goal,Patt,Ans,Env),
    appconj(universalise(Env),Gs),
    trace('trying subgoals %p for %p\n',[Gs,Goal],inference),
    test(Gs,Env,D,S,immed,Ans).

/* Special inference for unique goals */

uniinfer(Ass,D,Env,Args) :-
    same_predicate(Ass,Ass1,N),
    copy_args(Args,Ass,Ass1),
    appars(universalise(Env),Ass1),
    uniin1(Ass1,D,Env,Args), !,
    unify(any,Ass1,Ass,[],Env).

uniin1(Ass,D,Env,Args) :-
    same_predicate(Ass,Ass1,N),
    copy_args(Args,Ass,Ass1),
    appars(universalise(Env),Ass1),
    toff(assum,Y_N),
    (true;(ton(assum,Y_N),fail)),
    infer(Ass1,H1,H2,strict,D,[],Env),
    nident(Ass,Ass1), !,
    ton(assum,Y_N),
    unify(any,Ass1,Ass,[],Env),
    uniin1(Ass,D,Env,Args).
uniin1(_,_,_,_).

nident(A,B) :- same_predicate(A,B,N), nid(N,A,B).

nid(0,_,_) :- !, fail.
nid(N,A,B) :- arg(N,A,A1), arg(N,B,B1),
    bound(A1,A2), bound(B1,B2), A2=B2, !,
    N1 is N-1, nid(N1,A,B).
nid(_,_,_).
```

```
/* FORWAR
   Forwards inference when we add a new assertion
   C.M., 8/7/81
*/

%declarations%

:- public
                forwards_infer/3.

% imports:
%               add/2                   (from topdb.cpl)
%               commute_ass/2           (from metami.cpl)
%               normal_form/2           (from plcode:mlface.pl)
%               tryschema/3             (from demon.cpl)

%end%

:- mode
        forwards_infer(+,+,+),
        fw1(+,+,+).

forwards_infer(P,Ans,Env) :- commute_ass(P,P1), !,
   fw1(P,Ans,Env), fw1(P1,Ans,Env).
forwards_infer(P,Ans,Env) :- !, fw1(P,Ans,Env).

fw1(P,Ans,Env) :- normal_form(P,Asss), !, add(Asss,Env), tryschema(P,Ans,Env).
fw1(P,Ans,Env) :- tryschema(P,Ans,Env).
```

```
/* DEMON
   Implementation of forwards inference rules
   C.M., 25/8/81
*/

%declarations%

:- public
                do_schemas/0,
                forwards_rule/6,
                rules/0,
                tryschema/3.

% imports:
%               <- /2                          (from schema.pl)
%               add/2                         (from topdb.cpl)
%               ase/1                         (from undo.cpl)
%               assumption/1                  (from undo.cpl)
%               bind/5                        (from bind.cpl)
%               bound/2                       (from bind.cpl)
~               clean_all/0                   (from undo.cpl)
                db_state/1                    (from undo.cpl)
%               findall/3                     (from invoca.cpl)
%               firstenv/1                    (from env.cpl)
%               index_add/2                   (from index.cpl)
%               index_find/3                  (from index.cpl)
%               mapconj/3                     (from map.cpl)
%               restore_dbase/1               (from undo.cpl)
%               test/6                        (from topdb.cpl)
%               toff/2                        (from util.pl)
%               ton/2                         (from util.pl)
%               trace/3                       (from util:trace.pl)
%               writef/2                      (from util:writef.pl)

%end%

:- mode proc1(+,+,+),
        selectc(+,?,?),
        proc2(+,+,+,+,+).

 * Convert schemas from a readable into a usable form */

do_schemas :-
   clean_all,
   firstenv(Env),
   call((A <- B where C <<Name)),
   add_rules(A,B,C,Name,Env), fail.
do_schemas.

/* See what demons a given assertion will plug into */

tryschema(L,Ans,Env) :- !,
   findall(rule(H,L,B,F,Name),forwards_rule(L,H,L,F,B,Name),List),
   proc1(List,Ans,Env).

   proc1([],Ans,Env) :- !.
   proc1([rule(H,G,B,F,Name)|List],Ans,Env) :-
      length(Ans,Len),
      trace('%rTrying demon %p\n',['>',Len,Name],demon),
      db_state(N),
```

```
      toff(assum,Y_N),
      findall(Other,
         (all_in_dbase(F,any,Env),
          test(B,Env,shallow,any,immed,Ans),
          mapconj(bound,B&F,Other)),
       Others),
      ton(assum,Y_N),
      restore_dbase(N),
      proc2(Others,B&F,H,Name,Env),
      proc1(List,Ans,Env).

   proc2([],_,_,_,_) :- !.
   proc2([O|Os],B,H,Name,Env) :-
      proc3(O,B,H,Name,Env),
      proc2(Os,B,H,Name,Env).

   proc3(O,B,H,Name,Env) :-
      mapconj(bind(strict,[],Env),O,B), !,
      trace('>Running demon %p\n',[Name],4),
      add(H,Env).
   proc3(O,B,H,Name,Env) :-
      mapconj(bind(any,[],Env),O,B),
      assumption(applicable(Name)),
      add(H,Env).
   proc3(_,_,_,_,_).

/* Make additions for a rule with given
   consequences, forwards and backwards conditions */

add_rules(H,F,B,Na,Env) :-
   selectc(F,G,F1),
   Rule = fwd(G,H,F1,B,Na),
   ase(N),
   functor(G,Fn,Ar),
   index_add(keys(rule,N,G),Rule),
   fail.
add_rules(_,_,_,_,_).

   selectc(A&B,A,B).
   selectc(A&B&C,D,A&E) :- !, selectc(B&C,D,E).
   selectc(A&B,B,A) :- !.
   selectc(A,A,true) :- !.

/* Find a rule with a given assertion as a condition */

forwards_rule(Key,Head,Goal,Fwd,Bwd,Na) :-
   index_find(Key,_,fwd(Goal,Head,Fwd,Bwd,Na)).

/* Print out current rules */

rules :-
   forwards_rule(rule,Head,Goal,Fwd,Bwd,Na),
   writef('\nRule %t\n',[Na]),
   writef('     waiting for %t\n',[Goal]),
   writef('     conclusion %t\n',[Head]),
   writef('     also needs %t\n',[Fwd]),
   writef('     constraints %t\n',[Bwd]),
   fail.
rules.
```

```
/* CREATE
   Create object tokens
   C.M., 4/9/81
*/

%declarations%

:- public
                initref/4,
                stronscreate/2.

% imports:
%               add_type_info/3         (from types.cpl)
%               argument_names/1        (from plcode:mlface.pl)
%               argument_types/1        (from plcode:mlface.pl)
%               definite/1              (from env.cpl)
%               gensym/2                (from util:cmisce.pl)
%               gensymlist/3            (from util.cpl)
%               ground/1                (from util.cpl)
%               indefinite/1            (from env.cpl)
%               meaning/3               (from mean1.pl)
                meta_act/2              (from metact.pl)
%               meta_add/2              (from metadb.pl)
%               new_evar/2              (from obj.cpl)
%               trace/3                 (from util:trace.pl)
%               unbound/3               (from bind.cpl)
%               unify/5                 (from bind.cpl)

%end%

:- mode
        initref(+,+,+,?),
        stronscreate(+,+),
        createvals(+,+,+,+).

/* Initial referents */

initref(_,_,_,R) :- ground(R), !.
initref(Env,1,Noun,[Ref]) :-
   definite(Env), !,
   new_evar(Ref,Env),
   meta_add(known(Ref),Env).
initref(Env,N,Noun,Ref) :-
   indefinite(Env), !,
   gensymlist(N,Noun,Ref).
initref(Env,_,Noun,[Ref1,Ref2]) :-
   trace('Assuming there are two %ps\n',[Noun],assum),
   new_evar(Ref1,Env), new_evar(Ref2,Env),
   meta_add(known(Ref1),Env),
   meta_add(known(Ref2),Env),
   meaning(Env,different,(Ref1,Ref2)).

/* Create tokens to make an assertion true */

   stronscreate(L,Env) :- !,
      functor(L,Pred,N),
      functor(New,Pred,N),
      createvals(N,L,Env,New),
      unify(any,New,L,[],Env), !.
```

```
createvals(0,_,Env,_) :- !.
createvals(N,L,Env,New) :-
   arg(N,L,A),
   unbound(A,any,Env),
   not(meta_act(Env,test(known(A)))), !,
   functor(L,F,Ar),
   functor(Names,F,Ar), functor(Types,F,Ar),
   argument_names(Names), arg(N,Names,Name),
   argument_types(Types), arg(N,Types,Type),
   gensym(Name,A1), add_type_info(Type,A1,Env),
   arg(N,New,A1),
   N1 is N-1, createvals(N1,L,Env,New), !.
createvals(N,L,Env,New) :- !,
   N1 is N-1, createvals(N1,L,Env,New), !.
```

```
/* SILLY  - Characterisation of 'silly' goals, ie
   goals where the answer is 'no' rather than 'I dont know'
   C.M., 30/7/81
*/

%declarations%

:- public
                notsilly1/2,
                notsilly2/3.

% imports:
%               add_type_info/3          (from types.pl)
%               aliorelative/3           (from plcode:mlprp1.pl)
%               appars/2                 (from map.pl)
%               appconj/2                (from map.pl)
%               argument_types/1         (from plcode:mlface.pl)
%               bound/2                  (from bind.pl)
%               fix/2                    (from obj.pl)
%               iscvar/1                 (from obj.pl)
%               object_level_neg_rule/2  (from plcode:mlface.pl)
                prevent_bind/3           (from obj.pl)
                test/6                   (from topdb.pl)
%               trace/3                  (from util:trace.pl)
%               type_predicate/1         (from plcode:preds.pl)
%               universalise/2           (from obj.pl)

%end%

:- mode
        notsilly1(+,+),
        not_silly1(+,+),
        type_check(+,+,+,+),
        t_check(+,+,+),
        notsilly2(+,+,+),
        silly2(+,+,+).

/* Version 1 - tried on all goals */

notsilly1(Ass,Env) :-
   type_predicate(Ass), !.
 notsilly1(Ass,Env) :-
   functor(Ass,F,N),
   functor(Patt,F,N),
   argument_types(Patt),
   type_check(N,Ass,Patt,Env),
   not_silly1(Ass,Env), !.
notsilly1(Ass,Env) :-
   trace('%p is silly!\n',[Ass],silly), fail.

not_silly1(Ass,Env) :-
   aliorelative(Ass,Args,_), Args=[N1,N2],
   arg(N1,Ass,Arg1), arg(N2,Ass,Arg2),
   bound(Arg1,A), bound(Arg2,B),
   makediff(A,B,Env), !.
not_silly1(_,_).

   makediff(A,A,_) :- !, fail.
   makediff(A,B,Env) :- (iscvar(A);iscvar(B)), !, prevent_bind(A,B,Env).
   makediff(_,_,_).
```

```
    type_check(0,_,_,_) :- !.
    type_check(N,Ass,Patt,Env) :-
        arg(N,Ass,A), arg(N,Patt,Ty), t_check(A,Ty,Env),
        N1 is N-1, type_check(N1,Ass,Patt,Env).

    t_check(_,number,_) :- !.
    t_check(_,unit,_) :- !.
    t_check(A,Ty,Env) :-  add_type_info(Ty,A,Env).


/* Version 2 - tried only on unique goals */

notsilly2(Ass,Ans,Env) :-
    silly2(Ass,Ans,Env), !,
    trace('%p is silly!\n',[Ass],silly), fail.
notsilly2(_,_,_).

silly2(Ass,Ans,Env) :-
    object_level_neg_rule(Ass,Asss),
    appars(fix(Env),Ass),
    appconj(universalise(Env),Asss),
    test(Asss,Env,shallow,strict,immed,Ans).
```

```
/* PREDIC
   Simple prediction capability
   C.M., 30/6/81
*/

%declarations%

:- public
                operator/4,
                predict/4.

% imports:
%               <= /2                          (from operat.pl)
%               add/2                         (from topdb.cpl)
%               appconj/2                     (from map.cpl)
%               assumption/1                  (from undo.cpl)
%               meta_act/2                    (from metact.pl)
%               ndconjin/2                    (from util.cpl)
%               test/6                        (from topdb.cpl)
%               trace/3                       (from util:trace.pl)
%               universalise/2                (from obj.cpl)

%end%

:- mode
        predict(+,+,+,+),
        operator(-,+,-,-).

predict(Ass,S,Ans,Env) :- !,
   operator(Prec,Ass,Res,Name),
   trace('trying to use operator %p\n',[Name],predict),
   appconj(universalise(Env),Prec),
   test(Prec,Env,deep,S,immed,Ans),
   assumption(Name),
   meta_act(Env,use(Name)),
   add(Res,Env).

operator(Prec,Ass,Res,Name) :- !,
   call((Res <= Prec   <<Name)),
   ndconjin(Ass,Res).
```

```
/* DBASE
   Object level database facility
   C.M., 24/7/81
*/

%declarations%

:- public
                add_to_dbase/2,
                any_in/2,
                in_dbase/3,
                quick_retrieve/4,
                all_in_dbase/3.

% imports:
%               ase/1                   (from undo.cpl)
%               hold_task/3             (from task.cpl)
%               index_add/2             (from index.cpl)
%               index_find/3            (from index.cpl)
%               isenv/1                 (from env.cpl)
%               mapars/3                (from map.cpl)
                matchable/4             (from bind.cpl)
%               newtask/3               (from task.cpl)
%               pure/3                  (from bind.cpl)
%               same_predicate/3        (from plcode:preds.pl)
%               special_add/3           (from undefined)
%               special_test/4          (from alt.pl)
%               unify/5                 (from bind.cpl)

%end%

:- mode
        add_to_dbase(+,+),
        add_main(+,+,+),
        perhaps_enter(+,+,+),
        in_dbase(+,+,+),
        retrieve_main(+,+,+),
        any_in(+,-),
        quick_retrieve(+,+,-),
        q_r(+,+,-),
        all_in_dbase(+,+,+).

/* Adding to a database */

add_to_dbase(P,Env) :- isenv(Env), !,
   ase(N), !,
   add_main(P,Env,N).

   add_main(P,Env,N) :-
      special_add(P,Env,Z), !,
      perhaps_enter(P,Z,Env).
   add_main(P,Env,N) :- !,
      index_add(keys(N,object,P),[]).

   perhaps_enter(P,Z,Env) :-
      pure(P,any,Env), !, call(Z).
   perhaps_enter(P,Z,Env) :-
      newtask(enter,P,Task), hold_task(Task,[],Env).

/* Retrieving from a database */
```

```
in_dbase(Env,S,P) :- isenv(Env), !,
   retrieve_main(Env,S,P).

   retrieve_main(Env,S,P) :-
      special_test(P,Env,S,Z), !,
      call(Z).
   retrieve_main(Env,S,P) :- !,
      index_find(P,keys(_,object,Y),[]),
      unify(S,P,Y,[],Env).

/* Retrieval of any fact */

any_in(Env,P) :- !,
   index_find(object,keys(_,object,P),[]).

/* Quick retrieval without proper matching */

quick_retrieve(Ass,S,Env,Ass1) :- !,
   q_r(Ass,Env,Ass1), mapars(matchable(S,Env),Ass,Ass1).

   q_r(Ass,Env,Ass1) :-
      same_predicate(Ass,Ass1,N),
      special_test(Ass1,Env,any,Z), !, call(Z).
   q_r(Ass,Env,Ass1) :- !,
      index_find(Ass,keys(_,object,Ass1),[]).

/* Finding multiple assertions */

   all_in_dbase(true,_,_) :- !.
   all_in_dbase(A&B,S,Env) :- !,
      all_in_dbase(A,S,Env), all_in_dbase(B,S,Env).
   all_in_dbase(Ass,S,Env) :- in_dbase(Env,S,Ass).
```

```
/* ASSUM
   What relations hold in the default case?
   C.M., 30/6/81
*/

%declarations%

:- public
                assume/2.

% imports:
%               assumption/1            (from undo.cpl)
%               default_rule/2          (from plcode:mlface.pl)
%               test/6                  (from topdb.cpl)
%               unify/5                 (from bind.cpl)

%end%

:- mode
        assume(+,+).

  sume(Ass,Env) :- !,
   functor(Ass,F,N),
   functor(Ass1,F,N),
   default_rule(Ass1,Goals),
   unify(strict,Ass,Ass1,[],Env),
   test(Goals,Env,deep,any,immed,[]), !,
   assumption(Ass).
```

```
/* META
   Using meta-level information to decide what to do
   C.M., 1/8/81
*/

%declarations%

:- public
                add/6,
                enter/5,
                test/8.

% imports:
%               add_to_dbase/2          (from dbase.cpl)
%               add_type_info/3         (from types.cpl)
%               assume/2                (from assum.cpl)
%               bound/3                 (from bind.cpl)
%               test_constrained/3      (from metami.cpl)
%               add_constrained/3       (from metami.cpl)
%               exists/3                (from plcode:mlprp1.pl)
%               find_type/3             (from types.cpl)
                forwards_infer/3        (from forwar.cpl)
%               hold_task/3             (from task.cpl)
%               in_dbase/3              (from dbase.cpl)
%               infer/7                 (from infer.cpl)
%               mark_done/1             (from task.cpl)
%               notsilly2/3             (from silly.cpl)
%               predictable/1           (from metami.cpl)
%               pure/3                  (from bind.cpl)
%               quick_retrieve/4        (from dbase.cpl)
%               stronscreate/2          (from create.cpl)
%               type_predicate/3        (from plcode:preds.pl)
%               unbound/3               (from bind.cpl)
%               unify/5                 (from bind.cpl)
%               uniinfer/3              (from infer.cpl)
%               unique/3                (from plcode:mlprp1.pl)
%               unique_type/2           (from types.cpl)

%end%

 - mode
        add(+,+,+,-,?,+),
        test(+,+,+,+,+,?,?,+).

/* Strategy for new information */

add(Ass,Env,Task,[Task|H],H,Ans) :-
   type_predicate(Ass,T,X),
   unbound(X,any,Env), !, hold_task(Task,Ans,Env).
add(Ass,Env,Task,H,H,Ans) :-
   type_predicate(Ass,T,X), !,
   mark_done(Task),
   add_type_info(T,X,Env),
   forwards_infer(Ass,Ans,Env).
add(Ass,Env,Task,H,H,Ans) :-
   unique(Ass,any,Env,Args,_), !,
   mark_done(Task),
   notsilly2(Ass,Ans,Env),
   uniinfer_create(Ass,Ans,shallow,Env,Args).
add(point_of(A,B),Env,Task,H,H,Ans) :-
```

```
    bound(A,any,Env), !,
    mark_done(Task),
    notsilly2(point_of(A,B),Ans,Env),
    infer_or_create(point_of(A,B),any,shallow,Ans,Env).
add(Ass,Env,Task,H1,H2,Ans) :-
    add_constrained(Ass,any,Env), !,
    mark_done(Task),
    notsilly2(Ass,Ans,Env),
    dbfind(Ass,Ans,Env,Task,H1,H2).
add(Ass,Env,Task,[Task|H],H,Ans) :-
    hold_task(Task,Ans,Env), to_dbase(Ass,Ans,Env).

/* Strategy for postponed database additions */

enter(Ass,Env,Task,H,H) :- pure(Ass,any,Env), !,
    mark_done(Task),
    to_dbase(Ass,[],Env).
enter(_,Env,Task,[Task|H],H) :-
    hold_task(Task,[],Env).

/* Strategy for tests */

test(_,S,Ass,Env,Task,H,H,Ans) :-
    type_predicate(Ass,T,X), bound(X,S,Env), !,
    mark_done(Task),
    add_type_info(T,X,Env).
test(_,S,Ass,Env,Task,H,H,Ans) :-
    type_predicate(Ass,T,X), unbound(X,S,Env),
    unique_type(T,Env), !,
    mark_done(Task),
    find_type(T,X,Env), !.
test(_,_,Ass,Env,Task,[Task|H],H,Ans) :-
    type_predicate(Ass,_,_), !,
    hold_task(Task,Ans,Env).
test(D,S,Ass,Env,Task,H,H,Ans) :-
    unique(Ass,S,Env,Args,_), exists(Ass,S,Env,Args,_), !,
    mark_done(Task),
    notsilly2(Ass,Ans,Env),
    uniinfer_create(Ass,Ans,D,Env,Args).
test(D,S,Ass,Env,Task,H,H,Ans) :-
    unique(Ass,S,Env), !,
    mark_done(Task),
    notsilly2(Ass,Ans,Env),
    not_infer_then_assume(Ass,S,D,Ans,Env).
test(deep,S,Ass,Env,Task,H1,H2,Ans) :-
    predictable(Ass), !,
    notsilly2(Ass,Ans,Env),
    infer_or_hold(Ass,S,deep,H1,H2,Task,Ans,Env).
test(D,S,Ass,Env,Task,H1,H2,Ans) :-
    exists(Ass,S,Env), !,
    notsilly2(Ass,Ans,Env),
    infer_or_hold(Ass,S,D,H1,H2,Task,Ans,Env).
test(D,S,Ass,Env,Task,H1,H2,Ans) :-
    test_constrained(Ass,S,Env), !,
    mark_done(Task),
    notsilly2(Ass,Ans,Env),
    infer(Ass,H1,H2,S,shallow,Ans,Env), !.
test(_,_,Ass,Env,Task,[Task|H],H,Ans) :-
    hold_task(Task,Ans,Env).
```

```
/* Alternatives to be tried */

uniinfer_create(Ass,Ans,D,Env,Args) :-
   uniinfer(Ass,D,Env,Args), to_dbase(Ass,Ans,Env), strongcreate(Ass,Env).

to_dbase(Ass,Ans,Env) :- in_dbase(Env,strict,Ass), !.
to_dbase(Ass,Ans,Env) :- add_to_dbase(Ass,Env), forwards_infer(Ass,Ans,Env).

infer_or_create(Ass,S,D,Ans,Env) :- infer(Ass,H,H,S,D,Ans,Env).
infer_or_create(Ass,_,_,_,Env) :-
   strongcreate(Ass,Env), to_dbase(Ass,Ans,Env).

dbfind(Ass,Ans,Env,Task,H,H) :-
   mark_done(Task),
   quick_retrieve(Ass,any,Env,Ass1),
   unify(any,Ass,Ass1,[],Env), !.
dbfind(Ass,Ans,Env,Task,[Task|H],H) :-
   hold_task(Task,Ans,Env), to_dbase(Ass,Ans,Env).

not_infer_then_assume(Ass,S,D,Ans,Env) :- infer(Ass,H,H,S,D,Ans,Env), !.
not_infer_then_assume(Ass,S,D,Ans,Env) :- assume(Ass,Env).

infer_or_hold(Ass,S,D,H1,H2,Task,Ans,Env) :-
   mark_done(Task),
   infer(Ass,H1,H2,S,D,Ans,Env).
infer_or_hold(Ass,S,D,[Task|H],H,Task,Ans,Env) :-
   hold_task(Task,Ans,Env).
```

```
/* NUDGE
   Forcing held constraints to be satisfied
   C.M,  7/9/81
*/

%declarations%

:- public
                kickrefs/1,
                nudge/3.

% imports:
%               action/2                    (from task.cpl)
%               assume/2                    (from assum.cpl)
%               database_goal/1             (from metami.cpl)
%               error/3                     (from util:trace.pl)
%               find_pattern/3              (from types.cpl)
%               find_type/3                 (from types.cpl)
%               in_dbase/3                  (from dbase.cpl)
%               infer/7                     (from infer.cpl)
%               list/3                      (from listro.cpl)
                mark_done/1                 (from task.cpl)
%               meta_test/2                 (from metadb.pl)
%               obj_type/3                  (from obj.cpl)
%               plus_pred/1                 (from metami.cpl)
%               proposition/2               (from task.cpl)
%               type_predicate/3            (from plcode:preds.pl)
%               unbound/3                   (from bind.cpl)
%               undone/1                    (from task.cpl)
%               wont_ever_function/3           (from metami.cpl)

%end%

:- mode
        kickrefs(+),
        nudge(?,+,+),
        nudge1(?,+,+),
        take_one_goal(+,+,+,-),
        remove_one_task(+,-,-,-,-),
        decomp(?,-,-),
        undone(?,-).

kickrefs(Env) :-
   meta_test(mentioned(X),Env),
   unbound(X,any,Env), !,
   instantiate(X,Env),
   kickrefs(Env).
kickrefs(_) :- !.

instantiate(Ref,Env) :-
   obj_type(Ref,Env,Patt),
   find_pattern(Patt,Ref,Env).

nudge(D,Ans,Env) :- undone(D,D1), nudge1(D1,Ans,Env).

nudge1(D,_,Env) :- (var(D);D=[]), !.
nudge1(D,Ans,Env) :-
   take_one_goal(D,Ans,Env,D1),
   nudge(D1,Ans,Env).
```

```
take_one_goal(D,Ans,Env,D1) :-
   remove_one_task(D,D1,Ass,S,Dep),
   type_predicate(Ass,T,I), !,
   find_type(T,I,Env).
take_one_goal(D,Ans,Env,D1) :-
   remove_one_task(D,D1,Ass,S,Dep),
   database_goal(Ass),
   wont_ever_function(Ass,S,Env), !,
   in_dbase(Env,S,Ass).
take_one_goal(D,Ans,Env,D1) :-
   remove_one_task(D,D1,Ass,S,Dep),
   plus_pred(Ass),
   wont_ever_function(Ass,S,Env), !,
   infer(Ass,Wis,Ws,S,Dep,Ans,Env).
take_one_goal(D,Ans,Env,D1) :-
   remove_one_task(D,D1,Ass,S,Dep),
   plus_pred(Ass), !,
   infer(Ass,_,_,S,Dep,Ans,Env).
take_one_goal(D,Ans,Env,D1) :-
   remove_one_task(D,D1,Ass,_,_),
   assume(Ass,Env), !.
 ake_one_goal(D,_,Env,_) :-
   error('cant nudge %p\n',[D],fail).   % Assume a context of cwa

remove_one_task(D,D1,Ass,S,Dep) :-
   decomp(D,Task,D1),
   action(Task,test(Dep,S)), undone(Task),
   proposition(Task,Ass),
   mark_done(Task).

   decomp(Ws,_,_) :- var(Ws), !, fail.
   decomp(List,W,Ws) :- list(List,W,Ws).
   decomp(List,W1,[W|W1s]) :- !, list(List,W,Ws), decomp(Ws,Wi,Wis).

   undone(V,V) :- var(V), !.
   undone([],[]) :- !.
   undone(List,[W|W1s]) :- list(List,W,Ws),
      action(W,test(_,_)), undone(W), !,
      undone(Ws,W1s).
   undone(List,W1s) :- list(List,_,Ws), !, undone(Ws,W1s).
```

```
/* MAP
   Mapping functions over object_level assertions
   C.M., 6/8/81
*/

%declarations%

:- public
                appars/2,
                appconj/2,
                mapars/3,
                mapconj/3.

% imports:
%               bind/5                  (from bind.cpl)
%               bound/2                 (from bind.cpl)
%               bound/3                 (from bind.cpl)
%               fix/2                   (from obj.cpl)
%               matchable/4             (from bind.cpl)
%               possinstance/4          (from bind.cpl)
%               same_predicate/3        (from plcode:preds.pl)
                skolemise/2             (from obj.cpl)
%               universalise/2          (from obj.cpl)

%end%

:- mode
        appars(+,+),
        mapars(+,+,?),
        appconj(+,+),
        mapconj(+,+,?),
        app1(+,+,+),
        map1(+,+,+,+),
        apply1(+,?),
        apply2(+,?,?).

/* Over single assertions */

appars(Op,Ass) :-
   functor(Ass,F,N),
   app1(N,Op,Ass).

   app1(0,_,_) :- !.
   app1(N,Op,Ass) :-
      arg(N,Ass,Arg),
      apply1(Op,Arg),
      N1 is N-1, app1(N1,Op,Ass).

mapars(Op,Ass1,Ass2) :-
   same_predicate(Ass1,Ass2,N),
   map1(N,Op,Ass1,Ass2).

   map1(0,_,_,_).
   map1(N,Op,Ass1,Ass2) :-
      arg(N,Ass1,Arg1), arg(N,Ass2,Arg2),
      apply2(Op,Arg1,Arg2),
      N1 is N-1, map1(N1,Op,Ass1,Ass2).

/* Over conjunctions */
```

```
appconj(Op,A&B) :- !,
   appconj(Op,A), appconj(Op,B).
appconj(Op,Ass) :-
   appars(Op,Ass).

mapconj(Op,A&B,Asss) :- !,
   functor(Asss,&,2),
   ars(1,Asss,A1), ars(2,Asss,B1),
   mapconj(Op,A,A1),
   mapconj(Op,B,B1).
mapconj(Op,Ass1,Ass2) :-
   mapars(Op,Ass1,Ass2).

/* Known things to map */

apply1(bound(S,Env),X) :- bound(X,S,Env).
apply1(var,X) :- var(X).
apply1(skolemise(Env),X) :- skolemise(X,Env).
apply1(universalise(Env),X) :- universalise(X,Env).
apply1(fix(Env),X) :- fix(X,Env).

 pply2(bound,A,B) :- bound(A,B).
_pply2(bind(S,Ans,Env),A,B) :- bind(S,A,B,Ans,Env).
apply2(matchable(S,Env),A,B) :- matchable(A,B,S,Env).
apply2(possinstance(S,Env),A,B) :- possinstance(A,B,S,Env).
```

```
/* INVOCA.CPL
   Invocation routines
   C.M., 25/8/81
*/

:- public
        findall/3,
        /(not,1).

:- mode
        findall(?,+,?),
        after_found(+,+,-),
        not(+).

findall(X,P,List) :-
   asserta(found('$mark')), call(P), asserta(found(X)), fail.
findall(_,_,List) :-
   retract(found(X)), !, after_found(X,[],List).

   after_found('$mark',L,L) :- !.
   after_found(X,L,L1) :- retract(found(Y)), !, after_found(Y,[X|L],L1).

not(X) :- call(X), !, fail.
not(X) :- !.
```

```
/* PORTRA

   Converting datastructures to things that can be read, unified, etc
   C.M., 24/6/81
*/

%declarations%

:- public
                portray/1.

% imports:
%               bound/2                 (from bind.cpl)
%               isenv/1                 (from env.cpl)

%end%

/* Portray */

portray($N) :- !, bound($N,V), write(V).
portray(Z) :- isenv(Z), !, write('<env>').
```

```
/* FILE  listro.pl                    (compiled)
   Special version for semantic interpreter
   C.M., 24/7/81
*/

%declarations%

:- public
                append/3,
                lastof/2,
                list/3,
                makein/2,
                ndsin/2.

%end%

:- mode append(?,?,?),
        lastof(?,?),
        ndsin(?,?),
        makein(?,?),
        sin(?,?),
        list(+,-,-).

list([A|B],A,B).

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

lastof(L,X) :- var(L), !, L=[X|_].
lastof(List,X) :- list(List,_,L), lastof(L,X).

ndsin(X,L) :- var(L), !, fail.
ndsin(X,List) :- list(List,X,_).
ndsin(X,List) :- list(List,_,L), ndsin(X,L).

makein(X,L) :- var(L), !, L=[X|_].
makein(X,List) :- list(List,X,_), !.
makein(X,List) :- list(List,_,L), makein(X,L).

sin(X,L) :- var(L), !, fail.
 in(X,List) :- list(List,X,_), !.
sin(X,List) :- list(List,_,L), sin(X,L).
```

```
/* UTIL.CPL
   Compiled misc utilities
   C.M., 8/7/81
*/

%declarations%

:- public
                gensymlist/3,
                ground/1,
                ndconjin/2.

% imports:
%               gensym/2                    (from util:cmisce.pl)

%end%

:- mode
        gensymlist(+,+,-),
        ndconjin(?,+),
        ground(?),
        argsground(+).

/* Gensym a list of atoms */

gensymlist(0,_,[]) :- !.
gensymlist(N,Ty,[New|L]) :- !, gensym(Ty,New),
   N1 is N-1, gensymlist(N1,Ty,L).

/* Membership of a conjunction */

ndconjin(A,A&B).
ndconjin(A,_&B) :- !, ndconjin(A,B).
ndconjin(A,A) :- !.

/* Is a goal ground? */

ground(L) :- atomic(L), !.
ground(L) :- nonvar(L),
   L =.. [Pred|Args],
   argsground(Args), !.

   argsground([]) :- !.
   argsground([HD|TL]) :- !,
      ground(HD),
      argsground(TL).
```

```
/* DISTRI
   Interpret propositions about sets and references
   C.M., 29/9/81
   Use with other interpretation stuff
*/

%declarations%

:- public
                function/4,
                function2/5,
                property/3,
                relation/4,
                relation3/5.

% imports:
%               meaning/3                   (from mean1.pl)

%end%

:- mode property(+,+,+),
        function(+,+,+,-),
        tolist(+,?,?),
        relation(+,+,+,+),
        corresp(+,+,-),
        dopairs(+,+,+),
        function2(+,+,+,+,-),
        fn2(+,+,+,-),
        relation3(+,+,+,+,+).

/* Predicates with irregular distributions */

function(Env,[O1,O2],coeff,[C]) :- !,
   call(meaning(Env,coeff,((O1,O2),C))).

/* Unary predicates */

property(Env,[Ref|Refs],Key) :- !,
   call(meaning(Env,Key,Ref)),
   property(Env,Refs,Key).
 roperty(_,[],_) :- !.

/* Result bearing relations */

function(Env,[Ref|Refs],Key,Res) :- !,
   call(meaning(Env,Key,(Ref,Ref1))),
   tolist(Ref1,Res,Res1),
   function(Env,Refs,Key,Res1).
function(_,[],_,[]) :- !.

   tolist((A,B),[A|Res1],Res) :- !,
      tolist(B,Res1,Res).
   tolist(A,[A|Res],Res) :- !.

function2(Env,Ref1s,Ref2s,Key,Vals) :-
   corresp(Ref1s,Ref2s,Corr),
   fn2(Corr,Key,Env,Vals).

   fn2([],_,_,[]) :- !.
   fn2([P|Ps],Key,Env,Res) :-
```

```
      call(meaning(Env,Key,(P,Val))),
      tolist(Val,Res,Res1),
      fn2(Ps,Key,Env,Res1).

/* Other Relations */

relation(Env,Ref1,Ref2,Key) :- !,
   corresp(Ref1,Ref2,Pairs),
   dopairs(Pairs,Key,Env).

dopairs([],_,_) :- !.
dopairs([P|Ps],Key,Env) :- !,
   call(meaning(Env,Key,P)),
   dopairs(Ps,Key,Env).

relation3(Env,[Ref1],[Ref2],[Ref3],Key) :- !,
   call(meaning(Env,Key,(Ref1,Ref2,Ref3))).     % Can't do other cases

/* Finding corresponding pairs */

corresp([R1],[R2],[(R1,R2)]) :- !.
 corresp([R1],[R2|Rs],[(R1,R2)|Ps]) :- !,
   corresp([R1],Rs,Ps).
corresp([R1|Rs],[R2],[(R1,R2)|Ps]) :- !,
   corresp(Rs,[R2],Ps).
corresp([R1|R1s],[R2|R2s],[(R1,R2)|Ps]) :- !,
   corresp(R1s,R2s,Ps).
```

```
relation3(Env,[Ref1],[Ref2],[Ref3],Key) :- !,
   call(meanins(Env,Key,(Ref1,Ref2,Ref3))).     % Can't do other cases
```

```
/* EXPAND
   Expand an action on an assertion with lists as its arguments
   C.M., 7/10/80
*/

%declarations%

:- public
                expand/3.

%end%

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

expand(Ass,Action,Env) :- !,
   Ass=..[Pred|Args],
   expand1(yes,Pred,Args,Action,Env).

expand1(Flag,_,_,_,_) :- var(Flag), !.
expand1(_,Pred,Args,Action,Env) :-
   take1(Args,Args1,Args11,Flag), !,
   Ass=..[Pred|Args1],
   Goal=..[Action,Ass,Env],
   call(Goal),
   expand1(Flag,Pred,Args11,Action,Env).
expand1(_,_,_,_,_) :- !.

   take1([],[],[],_) :- !.
   take1([A|As],[A1|A1s],[A11|A11s],Flag) :- !,
      take2(A,A1,A11,Flag),
      take1(As,A1s,A11s,Flag).

   take2(A,B,C,_) :- var(A), !, A=[B|C].
   take2([A],A,[A],_) :- !.
   take2([A|As],A,As,yes) :- !.
```

```
/* METAMI
   Miscellaneous meta-level properties
   C.M., 30/7/81
*/

%declarations%

:- public
                bind_check/3,
                commute_ass/2,
                add_constrained/3,
                test_constrained/3,
                database_goal/1,
                plus_pred/1,
                predictable/1,
                wont_ever_exist/3.

% imports:
%               \= /2                       (from util:imisce.pl)
%               bind/5                      (from bind.cpl)
%               bound/3                     (from bind.cpl)
                commutative/3               (from plcode:mlprp1.pl)
%               copy_args/3                 (from plcode:preds.pl)
%               default_rule/2              (from plcode:mlface.pl)
%               exists_pattern/3            (from plcode:mlface.pl)
%               meta_test/2                 (from metadb.pl)
%               ndsin/2                     (from listro.cpl)
%               object_level_rule/2         (from plcode:mlface.pl)
%               operator/4                  (from predic.cpl)
%               quick_retrieve/4            (from dbase.cpl)
%               same_predicate/3            (from plcode:preds.pl)
%               special_test/4              (from alt.pl)
%               unbound/3                   (from bind.cpl)
%               unique_most_general/1       (from mlxpro.pl)

%end%

:- mode
        database_goal(+),
        predictable(+),
        constrained(+,+,+),
        wont_ever_exist(+,+,+),
        happens_to_be_unique(+,+),
        commute_args(+,-),
        bind_check(+,+,+),
        plus_pred(+).

database_goal(Goal) :- !,
   same_predicate(Goal,G,_),
   not(object_level_rule(G,_)),
   not(special_test(G,_,_,_)),
   not(default_rule(G,_)),                  % NB we allow predictables
 % not(time_inher_ars(G,N),allunbound([N],Goal,S,Env)),
   database_pred(G).

/* Is a goal potentially predictable? */

predictable(X) :- operator(_,X,_,_), !.

/* Will a goal actually succeed in at most one way? */
```

```
test_constrained(L,S,Env) :-
   constrained(L,S,Env), wont_ever_function(L,S,Env).

add_constrained(L,S,Env) :-
   constrained(L,S,Env), wont_ever_unique(L,S,Env).

constrained(L,S,Env) :-
   database_goal(L),
   happens_to_be_unique(L,S,Env), !.
constrained(L,S,Env) :-
   call(unique_most_general(L)), !.

   wont_ever_function(L,S,Env) :-
      function_pattern(L,Args,Vals),
      ndsin(X,Vals), arg(X,L,A),
      unbound(A,S,Env), !, fail.
   wont_ever_function(_,_,_).

   wont_ever_unique(L,S,Env) :-
      unique_pattern(L,Args,Vals),
      ndsin(X,Vals), arg(X,L,A),
      unbound(A,S,Env), !, fail.
   wont_ever_unique(_,_,_).

   happens_to_be_unique(L,S,E) :-
      quick_retrieve(L,S,E,A1),
      quick_retrieve(L,S,E,A2),
      A1 \= A2, !, fail.
   happens_to_be_unique(_,_,_) :- !.

/* Commute the arguments of a goal */

commute_ass(Goal,Goal1) :-
   commutative(Goal,Args,Rest), Args=[A1,A2],
   same_predicate(Goal,Goal1,N),
   arg(A1,Goal,Ar1), arg(A2,Goal1,Ar1),
   arg(A2,Goal,Ar2), arg(A1,Goal1,Ar2),
   copy_args(Rest,Goal,Goal1).

 * Check that a binding of an object level variable is OK */

bind_check(Cv,V,Env) :-
   meta_test(ref(Cv),Env), !,
   meta_test(refers(NP,[V1]),Env),
   bound(V1,any,Env),
   bind(strict,V1,V,[],Env).
bind_check(_,_,_).

/* What predicates appear in the database? */

database_pred(diff(_,_)) :- !, fail.
database_pred(vacant(_,_)) :- !, fail.
database_pred(G) :-
   % call((Word ==> Lambdas: Ass)),
   % ndconjin(G,Ass),
   !.
database_pred(G) :-
   exists_pattern(G,_,_), !.
```

```
/* What predicates do we get positive information about? */

plus_pred(X) :- database_pred(X), !.
plus_pred(X) :- predictable(X), !.
plus_pred(X) :- object_level_rule(X,_), !.
```

```
/* TYPES.CPL
   Chris's file for inferences about types
   C.M., 27/7/81
*/

%declarations%

:- public
                add_type_info/3,
                find_pattern/3,
                find_type/3,
                print_types/2,
                super_type/2,
                unique_type/2.

% imports:
%               \= /2                       (from util:imisce.pl)
%               a_constant/2                (from obj.cpl)
%               appars/2                    (from map.cpl)
%               bind/5                      (from bind.cpl)
%               obj_type/3                  (from obj.cpl)
                type_pattern/2              (from plcode:*.pl)

%end%

:- mode not_type(+,+,+,+),
        unique_type(+,+),
        find_type(+,?,+),
        print_types(?,+),
        super_type(+,+),
        compatible(+,?,+),
        add_type_info(+,+,+).

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

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

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

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

/* Is there at most one individual with a type? */

unique_type(Type,Env) :-
   type_pattern(Type,P),
   a_constant(Env,I1), obj_type(I1,Env,P1), comp(P,P1),
   a_constant(Env,I2), obj_type(I2,Env,P2), comp(P,P2),
   I1 \= I2, !, fail.
unique_type(_,_).

/* Find an object of a type */
```

```
find_type(Type,Ind,Env) :-
   type_pattern(Type,P),
   find_pattern(P,Ind,Env).

find_pattern(P,Ind,Env) :-
   a_constant(Env,Ind1), obj_type(Ind1,Env,P),
   bind(any,Ind,Ind1,[],Env).

/* Add new type information about an individual */

add_type_info(Type,Indiv,Env) :-
   obj_type(Indiv,Env,Patt),
   type_pattern(Type,Patt1), !, Patt=Patt1.

/* See what we know about an individual */

print_types(Indiv,Env) :-
   obj_type(Indiv,Env,Patt),
   p_type_ars(Patt,Indiv).

 _types(0,_,_) :- !.
 _types(N,Patt,Indiv) :-
   ars(N,Patt,Ars),
   p_type_ars(Ars,Indiv),
   N1 is N-1, p_types(N1,Patt,Indiv).

p_type_ars(A,_) :- var(A), !.
p_type_ars(Ars,Indiv) :-
   functor(Ars,Ty,N),
   appars(var,Ars), !,
   write(Ty), write('('), write(Indiv), write('),'), nl.
p_type_ars(Ars,Indiv) :-
   functor(Ars,_,N),
   p_types(N,Ars,Indiv).

/* Two patterns are compatible */

comp(A,B) :- A \= B, !, fail.
comp(_,_).
```

```
/* UTIL.PL
   Utilities
   C.M., 2/7/81
*/


/* Environment */

save :- (@sentence(_);recordedenv(_)), !,
   error('Saving unclean state',[],fail).
save :- tempfile(File), save(File).
restore :- tempfile(File), restore(File).

setfile :- statistics(heap,[A,B]),
   N3 is ((A-B) mod 10)+"0",
   N2 is (((A-B)/10) mod 10)+"0",
   N1 is (((A-B)/100) mod 10)+"0",
   append("scratch.",[N1,N2,N3],Name),
   name(File,Name),
   asserta((tempfile(File):-!)).

  -[A|B] :- !, o(A), -B.
-[] :- !, save.
-F :- !, o(F), save.

o(A) :- atom(A), add_ext("pl",A,A1), file_exists(A1), !, reconsult(A1).
o(A) :- atom(A), add_ext("cpl",A,A1), reconsult(A1).

/* Tracing */

t :- tlim(2), toff.
tt :- tlim(4), toff, ton(act), ton(assum), ton(silly), ton(demon).
ttt :- tlim(4), ton(all).

/* Add an extension to a file name */

add_ext(Ext,F,F1) :- name(F,Na), append(Na,[46|Ext],Na1), name(F1,Na1).

/* Putting tracing on and off */

  t(Goal) :- !, hitclause(Goal,C), asserta(C).

unhit(Goal) :- !, hitclause(Goal,C), retract(C).

   hitclause(Goal,(Goal:-unhit(Goal),trace,Goal)) :- !.

/* Ask user for some value */

user_supply(Att,Args,Val) :- !,
   seeing(I), telling(O),
   see(user), tell(user),
   write('**Please supply '), writef(Att,Args), write(': '),
   ttyflush, read(Val).


/* Temporary trace alterations */

toff(Name,yes) :- retract(tracing(Name)), !.
toff(Name,no).
```

```
ton(Name,yes) :- !, assertz(tracing(Name)).
ton(_,no).
```

```
/* NLTYPE.HI
   Types for natural language stuff
   C.M., 13/7/81
*/

 type_hierarchy.
%===============

{include('plib:types.hi')}.

person <-> particle.

man <-> person & male.
man <-> painter # boy.

woman <-> person & female.

n_particle <-> particle & neuter.
n_particle <-> ball # pulley # stone # train # blob # crane # hammer.

 _rod <-> rod & neuter.
  rod <-> bench # bridge # lever # pier # pole # scaffold # tower # bar.

n_point <-> point & neuter.
n_point <-> cliff # station.

n_surface <-> surface & neuter.
n_surface <-> plane # table.

string <-> rope # cord.
```

```
/* OPERAT
   Operators for prediction
   C.M., 7/9/81
*/

minimal_motion(Obj,M1,M2,Sys)
& body_contact(Obj,Surf,Fin,M2)
& motion(Obj,Start,Fin,M1,M2,Sys) <=
                                    body(Obj)
                                  & unsupported(Obj,M1)
                                  & at(Obj,Start,M1)
                                  & surface(Surf)
                                  & below(Surf,Start)
                                                    << fall(Obj,M1).

minimal_motion(Obj,T1,T2,M)
& motion(Obj,P1,P2,T1,T2,M)  <=
                                    one_d(P)
                                  & solid(P)
                                  & monopath(P)
                                  & inplace(Obj,P,P1,Side,T1)
                                  & getstarted(Obj,Path,P1,T1)
                                  & ifreaches(Obj,P2,T2)
                                  & period(Per,T1,T2)
                                  & nostopping(Obj,Path,P1,Per)
                                  & notakeoff(Obj,Path,P1,Side,Per)
                                  & nofalloff(Obj,Path,P1,Side,Per)
                                  & opposite(Side,Oside)
                                  & end(P,P2,Oside)
      << simple_path(Obj,P,T1).

% motion(Obj,P1,P3,T1,T3,M)   <=
%                                   minimal_motion(Obj,T1,T2,Mi)
%                                 & motion(Obj,P1,P2,T1,T2,M1)
%                                 & motion(Obj,P2,P3,T2,T3,M2)
%      << addmotion(M1,M2).
```

```
/* SCHEMA
   Forwards inference rules
   C.M., 23/9/81

*/

/* A point in contact */

contact(Obj,P,Mom)
                <- body_contact(Obj,_,P,Mom)
        where      point(Obj)
                                                    << point_at.

/* Support by a string, etc. */

/* A point stuck */

fixed(Obj,P,Mom)
                <- body_fixed(Obj,_,P,Mom)
        where      point(Obj)
                                                    << point_stuck.

/* String connecting two objects */

connects(Str,O1,O2,forever)
                <- body_fixed(O1,E1,_,forever)
        where      farend(Str,E1,E2)
                 & string(Str)
                 & body_fixed(O2,E2,_,forever)          << connects.

/* Supported string connecting two objects (pulley system) */

  pulley_sys(Str,Pull,O1,O2,Sys)
                <- connects(Str,O1,O2,forever)
                 & supports(Pull,Str,_)
        where true                                  << pulley_sys.

/* Support of a string */

  typical_point(Str,Pt)
  fixed(Pt,Par,forever)
                <- body_fixed(Obj,Str,Par,forever)
                 & supports(Obj,Str,Mom)
        where      string(Str)                      << string_support.

/* Consequences of motion */

  motion(Sys) &
  period_of(Sys,Per) &
  period(Per,Mom1,Mom2) &
  motion_of(Obj,Per,Sys) &
  object_of(Sys,Obj) &
  path_of(Sys,Path) &
  path(Path,Left,Right) &
  pathat(Obj,Left,Mom1) &
  pathat(Obj,Right,Mom2) &
  at(Obj,Start,Mom1) &
  at(Obj,Dest,Mom2)
                        <-
                           motion(Obj,Start,Dest,Mom1,Mom2,Sys)
```

```
                        where true                          <<motion.

/* Distance between two points */

separation(O1,O2,Sep,T)
                          <-
                            body_distance(O1,O2,Sep,T)
                          where zero_d(O1) & zero_d(O2)     <<point_distance.

  separation(O1,O2,Sep,T)
& perp_project(O1,Line,O2)
                          <-
                            body_distance(O1,Line,Sep,T)
                          where zero_d(O1) & one_d(Line)    <<line_distance.

  separation(O1,O2,Sep,T)
& perp_project(O2,Line,O1)
                          <-
                            body_distance(Line,O2,Sep,T)
                          where zero_d(O2) & one_d(Line)    <<line_distance.

 ; Scaffolds and painters */

true
                          <-
                            scaffold(Scaff)
                            & supports(Scaff,Painter,_)
                         where
                            painter(Painter)                <<scaffold.

/* Accelerations */

accel(P,A,T)
                          <-
                            sys_accel(P,A,T)
                         where
                            particle(P)                     <<particle_accel.

accel(P,A,T)
                          <-
                            sys_accel(Sys,A,T)
                          & pulleysys(_,_,P,_,Sys)          <<pulleysys_accel.
```

```
/* ALT
   Alternative axiomatisation for some predicates
   C.M., 21/7/81
*/

special_test(intersect(X,Y,Z),Env,_,inters(X,Y,Z,Env)).


inters(X,Y,Z,Env) :- bound(X,forever), !, bind(any,Y,Z,[],Env).
inters(X,Y,Z,Env) :- bound(Y,forever), !, bind(any,Z,X,[],Env).
inters(X,Y,Z,Env) :- bound(Z,forever), !,
   bind(any,X,forever,[],Env), bind(any,Y,forever,[],Env).
inters(X,Y,Z,Env) :- bind(any,X,Y,[],Env), bind(any,Y,Z,[],Env),
   (unbound(X,any,Env) -> error('uninstantiated intersection',[],fail); true).
```

```
/* INIT
   Top level functions of the semantic interpreter
   C.M., 13/7/81
*/

/* Initialise semantic interpretation system */

init :- setfile, core_image,
   display('Department of Artificial Intelligence'), nl,
   display('University of Edinburgh'), nl, nl,
   display('[ Semantic Interpretation Program ]'),
   ttynl, ttynl, restore_state.

restore_state :- tempfile(F), file_exists(F), !, restore.
restore_state :- !.

/* Read from file and then run */

run(File) :- input(File), go, output(File).

/* Get input from a file */

input(File) :-
   add_ext("syn",File,File1),
   nice(File1,File2),
   see(File2), repeat,
   read(T), proc(T),
   !, seen.

   nice(F,F) :- file_exists(F), !.
   nice(F1,F2) :-
      name(F1,Na1),
      append("syn:",Na1,Na2),
      name(F2,Na2),
      file_exists(F2), !.
   nice(F1,F2) :-
      name(F1,Na1),
      append("winsyn:",Na1,Na2),
      name(F2,Na2).

   proc(end_of_file) :- !.
   proc(Z) :- srecord(Z), fail.

/* Run interpreter on input given */

go :- clean_db, @sentence(S), go1(S), fail.
go.

   go1(S) :-
      recordedenv(Env), !,
      assumption(interpretable(S)),
      nextenv(Env,Env1),
      sentencetype(S,T),
      newenv(Env1,T,Env2),
      interp_s(Env2,S),
      final_sent(S,Env2),
      remove_envs,
      recordenv(Env2), !.
   go1(S) :- !,
      db_state(_),
```

```
      assumption(interpretable(S)),
      firstenv(Env),
      meta_act(Env,load(times)),
      sentencetype(S,T),
      newenv(Env,T,Env1),
      interp_s(Env1,S),
      final_sent(S,Env1),
      recordenv(Env1), !.

   sentencetype(S,def) :- @stype(S,question), !.
   sentencetype(S,def) :- @stype(S,command), !.
   sentencetype(S,indef) :- @stype(S,statement), !.
   sentencetype(S,def) :- @stype(S,wh_quest), !.
   sentencetype(S,indef) :- @conj(S,_,_,_), !.  % Hack
   sentencetype(S,T) :- user_supply('stype of %t (def/indef)',[S],T).

/* Record output on a file */

output(user) :- !,
   display.
output(File) :-
   add_ext("prb",File,File1),
   name(File1,Na), append("syn:",Na,Na1),
   name(File2,Na1), tell(File2), display, told.

   display :-
      recordedenv(E), !,
      disp1(E).
   display :-
      error('No recorded environment',[],continue),
      firstenv(E),
      disp1(E).

disp1(E) :-
      any_in(E,X), filter(X,E,Y),
      print(Y), put("."), nl, fail.
disp1(E) :-
      meta_assertion(X,E), filter(X,E,Y),
      print(Y), put("."), nl, fail.
disp1(E) :-
      a_constant(E,I),
      print_types(I,E), fail.
   disp1(_).

/* Help information */

help :- display('

Commands:

        input(FILE)     - read in the syntax trees from FILE.syn
        so              - process the current problem
        so1(S)          - process sentence S
        output(FILE)    - output the database to FILE.prb
        run(FILE)       :- input(FILE), so, output(FILE).
').
```

```
/* TERS
   Sentence Interpretation
   C.M., 14/9/81
   Use with rest of semantic interpretation stuff
*/

%here%

/* Interpret one sentence */

interp_s(Env,S) :-
   @conj(S,S1,Conj,S2), !,
   conj_s(Conj,S1,S2,Env).
interp_s(Env,S) :- !,
   trace('\n\nInterpretins sentence %t\n',[S],2),
   collectnps(Env,S,Res),
   set_tense(S,Env),
   timemods(Env,Res,Res1),
   @main_verb(S,Verb),
   trace('Invokins semantics for verb ''%t''\n',[Verb],2),
   interp_verb(Verb,Res1,Env).

/* Rules for conjunctions */

conj_s(when,S1,S2,Env) :- !,
   interp_s(Env,S1),
   interp_s(Env,S2).
conj_s(while,S1,S2,Env) :- !,
   interp_s(Env,S1),
   interp_s(Env,S2).

/* Collect and interpret top level NP's of sentence */

collectnps(Env,S,Res) :-
   findall((Role,NP),collect(Role,S,NP),Res), !.

   collect(subj,S,NP) :- logsubj(S,NP).
   collect(obj,S,NP) :- logobj(S,NP).
   collect(obj,S,NP) :- @passive(S), @syn_obj(S,NP).
   collect(by,S,NP) :- not(@passive(S)), @pp(S,by,NP).
   collect(P,S,NP) :- @pp(S,P,NP), P\=by.
   collect(adv,S,Adv) :- @adverb(S,Adv).

   logsubj(S,NP) :- @passive(S), !,
      @pp(S,by,NP).
   logsubj(S,NP) :- @syn_subj(S,NP).

   logobj(S,NP) :- @passive(S), !,
      @syn_subj(S,NP).
   logobj(S,NP) :- @syn_obj(S,NP).

/* Finalisins some of the decisions */

final_sent(S,Env) :-
   trace('\nFinalisins decisions for sentence %p\n',[S],2),
   final_time(S,Env),
   kickrefs(Env), !.
```

```
/* TERNP
   Semantic interpretation of noun phrases
   C.M., 8/9/81
   Use with other interpretation stuff
*/

%here%

/* Noun phrases */
interp_np(Env,NP,Ref) :-
   printhead(NP,N),
   trace('Interpreting noun phrase %t (%t)\n',[NP,N],2),
   i_np(Env,NP,Ref).

   printhead(NP,N) :- @headnoun(NP,N), !.
   printhead(NP,N) :- @name(NP,N), !.
   printhead(NP,'<trace>') :- @wh_trace(_,NP), !.
   printhead(_,?) :- !.

/* 0. Possessive determiner */
 np(Env,NP,Ref) :-
   @poss_det(NP,NP1), !,
   @fnof(NP,Fn),
   interp_np(Env,NP1,Ref1),
   function(Env,Ref1,Fn,Ref),
   entry(NP,Ref,Env).

/* 1. We already know the ref */
i_np(Env,NP,Ref) :- refof(NP,Env,Ref), !.
i_np(Env,NP,[Name]) :-
   @name(NP,Name), !,
   meaning(Env,Name,Name),
   entry(NP,[Name],Env).
i_np(Env,NP,Ref) :- @wh_trace(NP1,NP), !,
   i_np(Env,NP1,Ref).

/* 2. Conjoined NP */
i_np(Env,NP,Ref) :-
   @conj(NP,NP1,NP2), !,
   interp_np(Env,NP1,Ref1),
   interp_np(Env,NP2,Ref2),
   append(Ref1,Ref2,Ref).

/* 3. Initial quantifier phrase to be ignored */

i_np(Env,NP,Ref) :-
   @quant_front(NP,NP1), !,
   interp_np(Env,NP1,Ref).

/* 4. Definite measure */
i_np(Env,NP,Ref) :-
   @headnoun(NP,Dim), function(Dim),     % NB does indefinite phrases too
   @num(NP,_,Def),
   function_def(Dim,Def),
   not(@qp_det(NP,_)), !,
   ofobj(NP,Env,Ref1,Ref),               % Assumes definition of, eg "an end"
   entry(NP,Ref,Env).

   function_def(end,_) :- !.
   function_def(Fn,def).
```

```
/* 5. Various special phrases */
i_np(Env,NP,Ref) :-
   @headnoun(NP,N),
   specialref(N,NP,Env,Ref), !, entry(NP,Ref,Env).

/* 6. Ordinary NP */
i_np(Env,NP,Ref) :-
   @num(NP,Num,Det),
   newenv(Env,Det,Env1),
   @headnoun(NP,Noun),
   (@post_name(NP,Ref);true),
   initref(Env1,Num,Noun,Ref), !,
   entry(NP,Ref,Env),
   property(Env1,Ref,Noun),
   doprops(NP,Ref,Env1).

/* Special referents */

specialref(you,_,_,[you]) :- !.
 specialref(horizontal,_,Env,[Xax]) :- !,
   meaning(Env,x_axis,Xax).
specialref(Pr,_,Env,[Ref]) :-
   pron(Pr,Gen,_), !,
   testmeaning(Env,Gen,Ref),
   meaning(Env,known,Ref),
   meaning(Env,ref,Ref).
specialref(earth,_,Env,Ref) :- !, specialref(ground,_,Env,Ref).
specialref(ground,_,Env,[Gd]) :- !,
   meaning(Env,ground,Gd).
specialref(sea,_,Env,[Sea]) :- !,
   meaning(Env,sea,Sea).
specialref(level,NP,Env,Ref) :- !, @hasfeat(NP,ground),
   specialref(ground,_,Env,Ref).
specialref(london,NP,Env,[london]) :- !,
   meaning(Env,station,london).
specialref(edinburgh,NP,Env,[edinburgh]) :- !,
   meaning(Env,station,edinburgh).

'* Routines for processing modifiers */

doprops(NP,Ref,Env) :- !,
   doqps(NP,Ref,Env),
   doadjs(NP,Ref,Env),
   dopps(NP,Ref,Env),
   dorelc(NP,Ref,Env).

doqps(NP,Ref,Env) :-
   @qp_det(NP,QP), !,
   interp_qp1(QP,Ref,Env).
doqps(_,_,_) :- !.

doadjs(NP,Ref,Env) :-
   findall(Adj,@hasfeat(NP,Adj),List), !,
   checklist(interp_ap(Env,Ref),List).

dopps(NP,Ref,Env) :-
   findall(pp(P,NP1),@pp(NP,P,NP1),List), !,
   checklist(interp_pp(Ref,Env),List).
```

```
dorelc(NP,Ref,Env) :-
    findall(S,@relc(NP,S),List), !,
    checklist(interp_s(Env),List).

/* Handling measure phrases */

ofobj(NP,Env,Obj,Meas) :-
    @fnof(NP,Dim),
    prep_of(Dim,P),
    @pp(NP,P,NP1), !,
    interp_np(Env,NP1,Obj),
    function(Env,Obj,Dim,Meas).
ofobj(NP,Env,_,Meas) :-
    @relc(NP,S), !,
    interp_s(Env,S),
    refof(NP,Env,Meas).
ofobj(NP,Env,Obj,Meas) :-
    @fnof(NP,Dim),
    Obj=[O],
    new_evar(O,Env),
    meaning(Env,known,O),
    meaning(Env,ref,O),
    function(Env,Obj,Dim,Meas),
    entry(ars(NP),Obj,Env).

/* Reference list in environment */

entry(NP,Ref,Env) :-
    meta_act(Env,add(refers(NP,Ref))),
    allmentioned(Ref,Env), !.

refof(NP,Env,Ref) :-
    meta_act(Env,test(refers(NP,Ref))), !.

    allmentioned([],_) :- !.
    allmentioned([R|Rs],Env) :-
       meta_act(Env,add(mentioned(R))),
       allmentioned(Rs,Env).
```

```
/* TERPP
   Interpreting prepositional phrases
   C.M., 18/6/81
*/

interp_pp(Ref,Env,pp(P,NP)) :- !,
   trace('Interpreting prepositional phrase (%t)\n',[P],2),
   i_pp(Ref,Env,P,NP).

i_pp(Ref,Env,full_of,NP) :- !,
   @num(NP,plur,[]),
   @headnoun(NP,N),
   relation(Env,Ref,[N],full_of).
i_pp(Ref,Env,in,NP) :-
   @headnoun(NP,equilibrium), !,
   property(Env,Ref,equilibrium).
i_pp(Ref,Env,of,NP) :-
   @lone_qp(NP,QP), !,
   interp_qp1(QP,Ref,Env).
i_pp(Ref,Env,P,NP) :-
   (P=of; P=with), !,
   interp_indefmeas(NP,Ref,Env).
i_pp(Ref,Env,P,NP) :- !,
   interp_np(Env,NP,Ref1),
   relation(Env,Ref,Ref1,P).
```

```
/* MEAS
   Interpreting measure phrases
   C.M., 18/6/81
*/

%here%

/* Interpret an indefinite measure in the context of the
   referents it applies to       */

interp_indefmeas(NP1,Ref,Env) :-
   constit_base(NP1,NP2,_),
   refof(NP2,Env,_), !.
interp_indefmeas(NP1,Ref,Env) :- !,
   constit_base(NP1,NP,_),
   @fnof(NP,Dim),
   trace('Interpreting indefinite measure %t (%t)\n',[NP,Dim],2),
   function(Env,Ref,Dim,Val),
   entry(NP,Val,Env),
   interp_meas_mods(NP,Dim,Env,Val),
   optrelc(NP,Env).

interp_meas_mods(NP,Dim,Env,Val) :-
   ((@pp(NP,of,NP1),@qp_det(NP1,QP)); @qp_modify(NP,QP)), !,
   interp_qp(QP,Dim,Val,Env).
interp_meas_mods(_,_,_,_).

optrelc(NP,Env) :- @relc(NP,S), !, interp_s(Env,S).
optrelc(_,_) :- !.

/* Interpret a QP in the context of the definite measure
   referents to which it applies      */

interp_qp(how_much,_,Val,Env) :- !,
   property(Env,Val,sought).
interp_qp(QP,Dim,Val,Env) :- !,
   measures(QP,Mp1),
   trace('Interpreting measure pair %t\n',[Mp1],2),
   relation(Env,Val,Mp1,breakdown(Dim)).

  asures(QP,[[N,U]]) :-
   @measure(QP,N,U), !.
measures(QP,[[N1,U1],[N2,U2]]) :- !,
   @conj(QP,QP1,QP2),
   @measure(QP1,N1,U1),
   @measure(QP2,N2,U2).

/* Interpret a QP when no dimension given */

interp_qp1(QP,Ref,Env) :- !,
   measures(QP,Mp1),
   (Mp1=[[_,U]]; Mp1=[_,[_,U]]), !,
   unit_of(U,Dim), !,
   function(Env,Ref,Dim,Val),
   interp_qp(QP,Dim,Val,Env).

/* General adjectival phrases */

interp_ap(Env,Ref,AP) :- !,
   @head_adjective(AP,A),
```

```
    trace('Interpreting adjectival phrase (%t)\n',[A],2),
    i_ap(Env,Ref,AP).

i_ap(Env,Ref,AP) :-
    @head_adjective(AP,A),
    fnadj(A,Dim,Pol), !,
    function(Env,Ref,Dim,Val),
    intensifiers(Env,Dim,Pol,Val,AP).
i_ap(Env,Ref,AP) :- !,
    @head_adjective(AP,A),
    property(Env,Ref,A).

    intensifiers(Env,Dim,Pol,Val,AP) :-
        @intensifier(AP,how), !,
        Val=[V], meaning(Env,sought,V).

    intensifiers(Env,Dim,Pol,Val,AP) :-
        @qp_modify(AP,QP),
        @measure(QP,N,times), !,
        @comparative(_,AP,NP),
        interp_np(Env,NP,Ref),
        function(Env,Ref,Dim,Val1),
        Val=[V], Val1=[V1],
        timesop(Pol,Op),
        RHS =.. [Op,V1,N],
        meaning(Env,equation,V=RHS).

    intensifiers(Env,Dim,Pol,Val,AP) :-
        @comparative(AP,NP), !,
        interp_np(Env,NP,Ref),
        function(Env,Ref,Dim,Val1),
        Val=[V], Val1=[V1],
        sensym(Dim,Q), RHS=..[Pol,V1,Q],
        meaning(Env,equation,V=RHS),
        @qp_modify(AP,QP),
        property(Env,[Q],noun(Dim)),
        interp_qp(QP,Dim,[Q],Env).

    intensifiers(Env,Dim,_,Val,AP) :-
        @qp_modify(AP,QP), !,
        interp_qp(QP,Dim,Val,Env).

/* Which adjectives correspond to simple dimensions? */

fnadj(apart,seat_separation,+).
fnadj(old,age,+).
fnadj(young,age,-).
fnadj(long,length,+).
fnadj(short,length,-).

/* What multiplicative operators correspond to
   what polarities?                    */

timesop(+,*).
timesop(-,/).
```

```
/* TERADV
   Semantic interpretation of adverbs
   C.M., 21/7/81
*/

interp_adv(Env,Adv,Verb,Ref) :- !,
   @adv_head(Adv,A),
   trace('Interpreting adverb %t (%t)\n',[Adv,A],2),
   i_adv(A,Env,Adv,Verb,Ref).

i_adv(upward,Env,Adv,_,Ref) :-
   @hasfeat(Adv,vertical), !,
   function(Env,Ref,velocity,Vel),
   relation(Env,Vel,[_:[90,degrees]],breakdown(velocity)).
i_adv(horizontal,Env,_,Verb,Ref) :-
   motion_verb(Verb), !,
   function(Env,Ref,velocity,Vel),
   relation(Env,Vel,[_:[0,degrees]],breakdown(velocity)).
i_adv(horizontal,Env,_,_,Ref) :- !,
   property(Env,Ref,horizontal).
i_adv(vertical,Env,_,Verb,Ref) :-
   motion_verb(Verb), !,
   function(Env,Ref,velocity,Vel),
   relation(Env,Vel,[_:[90,degrees]],breakdown(velocity)).
i_adv(vertical,Env,_,_,Ref) :- !,
   property(Env,Ref,vertical).
i_adv(free,_,_,_,_) :- !.
```

```
/* TMODS
   Semantics of time modifiers
   C.M., 23/7/81

Exports:

        timemods(3)
        set_date(3)

Imports:

        lone_qp(2)
        @(1)
        time_focus(3)
        newmoment(2)
        meaning(3)
        fmeaning(3)
        function(4)
        interp_qp(4)
        set_moment(2)
        lone_number(2)

*/

/* Filter list of syntactic roles, taking out time modifiers
   and interpreting them                                          */

timemods(Env,[],[]) :- !.
timemods(Env,[(R,NP)|Rs],R1s) :-
   time_modifier(Env,R,NP), !,
   timemods(Env,Rs,R1s).
timemods(Env,[R|Rs],[R|R1s]) :- !,
   timemods(Env,Rs,R1s).

/* Deal with individual types of modifiers */

time_modifier(Env,in,NP) :-
   @lone_qp(NP,QP),
   @measure(QP,N,years), !,
   time_focus(pres,Env,F),
   newmoment(Env,M),
   function(Env,[(F,M)],period,Per),
   function(Env,Per,duration,Dur),
   interp_qp(QP,duration,Dur,Env),
   set_time(Env,M).

time_modifier(Env,in,NP) :-
   @lone_number(NP,I), !,
   set_date(I,Env,Time),
   set_time(Env,Time).

time_modifier(Env,in,NP) :-
   @num(NP,_,def),
   @hasfeat(NP,AP), @head_adjective(AP,first), !,
   time_decomp(NP,Mp,M1,M3,Env),
   newmoment(Env,M2),
   meaning(Env,period,((M1,M2),Per)),
   Env=top(_,P)/R, Env1=top(indef,P)/R,
   adddim(duration,Mp,Per,Env1),
   set_time(Env,Per).
```

```
time_modifier(Env,in,NP) :-
   @num(NP,_,def),
   @hasfeat(NP,AP), @head_adjective(AP,last), !,
   time_decomp(NP,Mp,M1,M3,Env),
   newmoment(Env,M2),
   meaning(Env,period,((M2,M3),Per)),
   adddim(duration,Mp,Per,Env),
   set_time(Env,Per).

/* Convert an integer to a date */

set_date(I,Env,Mom) :-
   newmoment(Env,Mom),
   function(Env,[Mom],dateof,D),
   relation(Env,D,[[I,years]],breakdown(duration)).

/* Get underlying moments and measure pair for
   ... first X of ..., ... last X of ... */

time_decomp(NP,Mp,M1,M2,Env) :- !,
   timeqp(NP,Mp),
   @pp(NP,of,NP1),
   interp_np(Env,NP1,[Ref]),
   testmeaning(Env,period_of,(Ref,Per)),
   testmeaning(Env,period,((M1,M2),Per)).

   timeqp(NP,[1,U]) :- @headnoun(NP,U), !.
   timeqp(NP,[N,U]) :- !,
      @qp_det(NP,QP), @measure(QP,N,U).
```

```
/* DICT
   Dinctionary
   C.M., 18/6/81
*/

/* Dimensions */

function(mass). function(length). function(son). function(daughter).
function(father). function(mother). function(velocity).
function(speed). function(acceleration). function(tension).
function(end). function(weight). function(age). function(angle).
function(distance). function(motion). function(work).
function(magnitude). function(force). function(constant).
function(top). function(edge). function(coefficient). function(height).

/* Pronouns and Names */

pron(it,neuter,noposs).
pron(its,neuter,poss).
pron(he,male,noposs).
pron(his,male,poss).
  on(him,male,noposs).
pron(she,female,noposs).
pron(her,female,_).

/* Units */

unit_of(mph,velocity).
unit_of(lbs,mass).
unit_of(lb,mass).
unit_of(kg,mass).
unit_of(U,Dim) :-
   user_supply('dimension for unit %p',[U],Dim).

/* Prepositions marking function application */

prep_of(tension,in).
prep_of(friction,between) :- !.
prep_of(coeff,between) :- !.
prep_of(Dim,of).
```

```
/* INTERF
   Interface to parser
   C.M.  14/9/81
   Use with rest of interpretation stuff
*/

%here%

/* Record an assertion from the parser */

srecord(Z) :- recordz(Z,@(Z),_), !.

/* Provide an assertion to the interpreter. The followins are
   the predicates that the parser notionally provides:
*/

@adverb(S,A)     :- !,          % An adverb modifyins a S
   @@adverb(S,A).
@adv_head(Ad,A) :- !,           % The head of an adverbial phrase
   (@@hasfeat(Ad,A);
    (atom(Ad),Ad=A)).
  ux_verb(S,Aux) :- !,          % The tense of a sentence
   @@aux_verb(S,Aux).
@comparative(AP,NP) :- !,       % A "than NP" in a comparative AP
   @@comparative(AP,NP).
@conj(Bis,Sm1,Conj,Sm2) :- !,   % A constituent is a conjunction
   @@conj(Bis,Sm1,Conj,Sm2).
@conj(P,P1,P2) :- !,            % Default conjunction (with "and")
   @@conj(P,P1,and,P2).
@fnof(NP,Dim)    :- !,          % The complete name for a function phrase
   fnof(NP,Dim).
@embedded_sent(S1,S2) :- !,     % What sentences are embedded in what
   embedded_sent(S1,S2).
@hasfeat(NP,F)   :- !,          % Adjectives, adverbs, attached to phrases
   @@hasfeat(NP,F).
@head_adjective(AP,A) :- !,     % The head of an AP
   @@headadj(AP,A).
@headnoun(NP,N) :- !,           % The head of an NP
   @@headnoun(NP,N).
@intensifier(AP,I) :- !,        % An intensifier in an AP
   @@intensifier(AP,I).
@lone_ap(NP,AP) :- !,           % An AP as a sentence constituent
   lone_ap(NP,AP).
@lone_number(NP,N) :- !,        % A number as a sentence constituent
   lone_number(NP,N).
@lone_qp(NP,QP) :- !,           % A QP as a sentence constituent
   lone_qp(NP,QP).
@main_verb(S,V) :- !,           % The main verb of a clause
   @@main_verb(S,V).
@meas(NP,Dim) :- !,             % Whether a phrase is a function phrase
   meas(NP,Dim).
@measure(QP,N,U) :- !,          % How a QP breaks down
   @@measure(QP,N,U).
@name(NP,Na)     :- !,          % A noun phrase is a proper noun
   @@name(NP,Na).
@num(NP,Nu,Def) :- !,           % Number and definiteness of a NP
   num(NP,Nu,Def).
@passive(S)      :- !,          % Whether a sentence is passive
   passive(S).
@poss_det(NP1,NP2) :- !,        % Possessive determiner of NP
```

```
   @@poss_det(NP1,NP2).
@post_name(NP,Na) :- !,          % Name appears after head noun
   post_name(NP,Na).
@pp(Bas,P,NP)   :- !,            % Prepositional phrase attached to some base
   pp(Bas,P,NP).
@qp_det(NP,QP)  :- !,            % Prenominal QP in an NP
   qp_det(NP,QP).
@qp_modify(X,QP) :- !,           % Postnominal QP in an NP
   @@qp_modify(X,QP).
@quant_front(NP,NP1) :- !,       % NP minus initial quantifier phrase
   quant_front(NP,NP1).
@relc(NP,S)     :- !,            % Relative clause attached to an NP
   @@relc(NP,S).
@sentence(S)    :- !,            % A top-level sentence
   @@sentence(S).
@special_np(NP,N) :- !,          % Head of NP (following bindings etc)
   special_np(NP,N).
@stype(S,Ty)    :- !,            % Type of a sentence (question, statement etc)
   @@stype(S,Ty).
@syn_obj(S,NP)  :- !,            % Syntactic object of a sentence
   syn_obj(S,NP).
  @syn_subj(S,NP) :- !,          % Syntactic subject of a sentence
   @@syn_subj(S,NP).
@wh_trace(NP1,NP2) :- !,         % A bound trace
   @@wh_trace(NP1,NP2,_).
@X :- writef('Warning - funny syntax goal %t\n',[X]), @@X.

/* Implementation of some of the above */

% Embedded sentences

embedded_sent(S1,S2) :- @relc(NP,S1), sent_of(NP,S2).
embedded_sent(S1,S2) :- @@np_comp_s(NP,S1), sent_of(NP,S2).

   sent_of(S,S) :- @sentence(S), !.
   sent_of(NP,S) :- pp(Bs,_,NP), !, sent_of(Bs,S).
   sent_of(NP,S) :- @syn_subj(S,NP), !.
   sent_of(NP,S) :- @syn_obj(S,NP).

% Passive

passive(S) :- @@passive_sent(S), !.
passive(S) :- @@embedded_sent(S), @aux_verb(S,past),
   @syn_subj(S,NP), @wh_trace(_,NP), !.

% Prepositional phrases

pp(Base,P1,NP) :-
   @@pp_linked(Base,PP),
   @@is_prep(PP,P,NP),
   set_prep(PP,P,P1).
pp(Base,P1,NP) :-                        % QP PP as sentence constituent
   @@syn_obj(Base,NP1),
   @@pp_linked(NP1,PP1),
   not(@headnoun(NP1,_)),
   @qp_det(NP1,QP),
   @@is_prep(PP1,P,NP),
   @measure(QP,N,U),
   P1=..[P,[N,U]].
```

```
   set_prep(PP,P,P1) :-
      @qp_modify(PP,QP), !,
      @measure(QP,N,U),
      P1=..[P,[N,U]].
   set_prep(_,P,P) :- !.

% Syntactic object

syn_obj(S,NP) :- @@syn_obj(S,NP),
   not((pp(S,_,NP1),@@pp_linked(NP,PP),@@is_prep(PP,_,NP1))).

% Function phrases

fnof(NP,Dim) :- !, @headnoun(NP,D), function(D), morefn(D,NP,Dim), !.
fnof(NP,_) :- @headnoun(NP,N), error('dont know function %p\n',[N],abort).

   morefn(force,NP,fric_force) :- adj_mod(NP,frictional), !.
   morefn(end,NP,lend) :- adj_mod(NP,left), !.
   morefn(end,NP,rend) :- adj_mod(NP,right), !.
   morefn(end,NP,other_end) :- adj_mod(NP,other), !.
   morefn(end,NP,ends) :- @num(NP,plur,_), !.
   morefn(coefficient,NP,coeff) :-
      @pp(NP,of,NP1), @headnoun(NP1,friction), !.
   morefn(Dim,_,Dim) :- !.

adj_mod(NP,A) :- !, @hasfeat(NP,AP), head_adjective(AP,A).

% What are the interesting bits of a noun phrase?
% Follow links to get the basic NP token and also
% a token that one can perform simple tests on

constit_base(NP,NP,NP2) :-
   @conj(NP,NP1,_), !,
   constit_base(NP1,_,NP2).
constit_base(NP,NP1,NP2) :-
   @wh_trace(NP0,NP), !,
   constit_base(NP0,NP1,NP2).
constit_base(NP,NP,NP) :- !.

most_name(NP,[Ref]) :-
   @qp_modify(NP,QP),
   @measure(QP,Ref,arbs), !.

num(NP,2,def) :- @@num(NP,1,indef), @@quantifier(NP,each), !.
num(NP,N,Def) :- @@qp_det(NP,QP), @@measure(QP,N,arbs), !, @@num(NP,_,Def).
num(NP,1,Def) :- @@num(NP,1,Def), !.    % NB handles variable in 2nd ars
num(NP,plur,Def) :- @@num(NP,plur,Def).

qp_det(NP,QP) :- @@qp_det(NP,QP), not(@@measure(QP,_,arbs)).

% Identify various funny lone constituents

lone_qp(NP,how_much) :-                  % "How much" does he weigh?
   constit_base(NP,_,NP1),
   not(@headnoun(NP1,_)),
   @hasfeat(NP1,AP),
   @@headadj(AP,much),
   @intensifier(AP,how), !.
lone_qp(NP,QP) :- !,                     % He weighs "150 lbs"
   constit_base(NP,_,NP1),
```

```
    not(@headnoun(NP1,_)),
    @qp_det(NP1,QP),
    @measure(QP,_,U),
    not(@pp(NP,_,_)).

lone_number(NP,N) :- !,                    % He is "26"
    constit_base(NP,_,NP1),
    not(@headnoun(NP1,_)),
    @@qp_det(NP1,QP),
    @measure(QP,N,arbs),
    not(@pp(NP,_,_)).

lone_ap(NP,AP) :- !,                       % It is "3 lb heavier than .."
    constit_base(NP,_,NP1),
    not(@headnoun(NP1,_)),
    @hasfeat(NP1,AP),
    @@headadj(AP,_).

% Tests on NP's

special_np(NP,N) :- !,
    constit_base(NP,NP1,_),
    @headnoun(NP1,N).

meas(NP,Dim) :- !,
    constit_base(NP,_,NP1),
    @headnoun(NP,Dim), function(Dim).

quant_front(NP,NP1) :-
    @@quantifier(NP,each),
    not(@headnoun(NP,_)),
    @pp(NP,of,NP1).

/* Get an assertion from the parser */

@@Z :- recorded(Z,@(Z),_).
```

```
/* VERBS
   Semantic routines for verbs (above 'distrib' level)
   C.M., 14/9/81
   Use with other interpretation stuff

   Exports:

        interp_verb/3

   Imports:

        @(1)
        defmeas(2)
        function(4)
        set_date(3)
        interp_indefmeas(3)
        interp_np(3)
        interp_pp(3)
        interp_qp(4)
        interp_ap(3)
        lone_ap(2)
        lone_number(2)
        lone_qp(2)
        makecontacts(3)
        newenv(3)
        property(3)
        relation(4)
        set_time(2)
        special_np(2)
*/

interp_verb(arrive,[(subj,NP1),(P,NP2)],Env) :-
   (P=in;P=at), !,
   interp_np(Env,NP1,Objs),
   interp_np(Env,NP2,Places),
   relation(Env,Objs,Places,motion_point).

interp_verb(attach,[(obj,NP1)|Roles],Env) :- !,
   interp_np(Env,NP1,Ref1),
   att_args(Roles,Env,Ref2,Advs),
   function2(Env,Ref1,Ref2,stuck_somewhere,Pt),
   apply_advs(Advs,Pt,attach,Env).

   att_args([(P,NP2)|Advs],Env,Ref,Advs) :-
      (P=to; P=at), !,
      interp_np(Env,NP2,Ref).
   att_args(Roles,Env,[Ref],Roles) :- !,
      constraint(solid,[Ref],Env),
      entry(null,[Ref],Env).

interp_verb(be,[(subj,NP1),(obj,NP2)],Env) :-
   @headnoun(NP1,it),
   @lone_number(NP2,N), !,
   set_date(N,Env,Time),
   set_time(Env,Time).
interp_verb(be,[(subj,NP1),(obj,NP2)],Env) :-
   not(@meas(NP1,_)),
   @lone_number(NP2,N), !,
   interp_np(Env,NP1,Ref),
   function(Env,Ref,ase,Ases),
```

```
    relation(Env,Ases,[[N,years]],breakdown(ase)).
interp_verb(be,[(subj,NP),(obj,NP1)],Env) :-
    @meas(NP,Dim), @lone_qp(NP1,QP), !,
    interp_np(Env,NP,Ref),
    interp_qp(QP,Dim,Ref,Env).
interp_verb(be,[(subj,NP2),(obj,NP1)],Env) :-
    @special_np(NP1,what), !,
    interp_np(Env,NP2,Ref),
    property(Env,Ref,sought).
interp_verb(be,[(subj,NP),(obj,NP1)],Env) :-
    @lone_ap(NP1,AP), !,
    interp_np(Env,NP,Ref),
    interp_ap(Env,Ref,AP).
interp_verb(be,[(subj,NP1),(of([N,arbs]),NP2)],Env) :-
    @meas(NP1,_),
    @meas(NP2,_), !,
    interp_np(Env,NP1,Ref1),
    interp_np(Env,NP2,Ref2),
    function2(Env,[N],Ref2,product,Ref1).
interp_verb(be,[(subj,NP)|Mods],Env) :- !,
    interp_np(Env,NP,Ref),
    applypps(Mods,Ref,Env).

interp_verb(bear,[(obj,NP1)],Env) :- !,
    interp_np(Env,NP1,People),
    property(Env,People,born).

interp_verb(calculate,Roles,Env) :- !, interp_verb(find,Roles,Env).

interp_verb(carry,Roles,Env) :- !, interp_verb(support,Roles,Env).

interp_verb(connect,[(subj,NP1),(obj,NP2)],Env) :- !,
    interp_np(Env,NP1,Str),
    interp_np(Env,NP2,Ps),
    Ps = [P1,P2],
    relation3(Env,Str,[P1],[P2],connects).
interp_verb(connect,[(subj,NP1),(obj,NP2),(to,NP3)],Env) :- !,
    interp_np(Env,NP1,Str),
    constraint(string,Str,Env),
    interp_np(Env,NP2,P1),
    interp_np(Env,NP3,P2),
    relation3(Env,Str,P1,P2,connects).

interp_verb(do,[(subj,NP1),(obj,NP2)|_],Env) :- !,
    interp_np(Env,NP1,Obj),
    interp_indefmeas(NP2,Obj,Env).

interp_verb(drop,[(obj,NP1),(from,NP2)],Env) :- !,
    interp_np(Env,NP1,Part),
    interp_np(Env,NP2,Place),
    property(Env,Part,stationary),
    property(Env,Part,unsupported),
    relation(Env,Part,Place,motion_start).

interp_verb(fall,[(subj,NP1),(through,NP2)],Env) :-
    interp_np(Env,NP1,Ref),
    function(Env,Ref,motion_of,Sys),
    interp_indefmeas(NP2,Ref,Env).

interp_verb(find,[(subj,NP1),(obj,NP2)],Env) :-
```

```
    interp_np(Env,NP1,Ref),
    Ref=[you], !,
    interp_np(Env,NP2,Meas),
    property(Env,Meas,sought).

interp_verb(hang,[(R,NP1),(from,NP2)],Env) :-
    (R=subj;R=obj),
    interp_np(Env,NP1,Strs),
    constraint(string,Strs,Env),
    interp_np(Env,NP2,Pts),
    function(Env,Strs,rend,Rends),
    relation(Env,Pts,Rends,stuck).
interp_verb(hang,[(subj,NP)|Advs],Env) :- !,
    interp_np(Env,NP,Obj),
    relation(Env,[Str],Obj,support),
    constraint(string,[Str],Env),
    apply_advs(Advs,Obj,hang,Env).

interp_verb(have,[(subj,NP1),(obj,NP2)],Env) :- !,
    interp_np(Env,NP1,Obj),
    interp_indefmeas(NP2,Obj,Env).

interp_verb(hit,[(subj,NP1),(obj,NP2)|Mods],Env) :- !,
    interp_np(Env,NP1,Part),
    interp_np(Env,NP2,Place),
    function2(Env,Part,Place,contact_somewhere,Point),
    relation(Env,Part,Point,motion_point),
    applypps(Mods,Part,Env).

interp_verb(hold,Roles,Env) :- !, interp_verb(support,Roles,Env).

interp_verb(incline,[(obj,NP1),(at,NP2),(to,NP3)],Env) :- !,
    interp_np(Env,NP1,Obj),
    interp_np(Env,NP3,Ref),
    function(Env,Obj,tangent,Tan1),
    function(Env,Ref,tangent,Tan2),
    function2(Env,Tan1,Tan2,difference,Diff),
    @lone_qp(NP2,QP),
    interp_qp(QP,angle,Diff,Env).

  interp_verb(leave,[(subj,NP1),(obj,NP2)],Env) :- !,
    interp_np(Env,NP1,Objs),
    interp_np(Env,NP2,Places),
    relation(Env,Objs,Places,motion_start).

interp_verb(make,[(subj,NP1),(obj,AnsNP),(with,NP2)],Env) :- !,
    @meas(AnsNP,angle),
    interp_np(Env,NP1,Obj),
    interp_np(Env,NP2,Place),
    function(Env,Obj,tangent,Tan1),
    function(Env,Place,tangent,Tan2),
    function2(Env,Tan1,Tan2,difference,Diff),
    interp_meas_mods(AnsNP,angle,Env,Diff).

interp_verb(pass,Roles,Env) :- !, interp_verb(run,Roles,Env).

interp_verb(pin,[(obj,NP1),(at,NP2)],Env) :- !,
    interp_np(Env,NP1,Obj),
    interp_np(Env,NP2,Place),
    pin1(Obj,Place,Env).
```

```
    pin1(Obj,Place,Env) :- assumption(separable(Obj,Place)),
        function2(Env,Obj,Place,stuck_somewhere,_).
    pin1(Obj,Place,Env) :- property(Env,Place,fixed).

interp_verb(place,[(obj,NP)|PPs],Env) :- !,
    interp_np(Env,NP,Obj),
    applypps(PPs,Obj,Env).

interp_verb(project,Roles,Env) :- !, interp_verb(throw,Roles,Env).

interp_verb(raise,[(subj,NP1),(obj,NP2),(obj,NP3)],Env) :- !,
    interp_np(Env,NP2,Obj),
    function(Env,Obj,motion_of,Motn),
    interp_np(Env,NP1,Agent),
    relation(Env,Agent,Obj,applied_force),
    @headnoun(NP3,height),
    function(Env,Obj,separation_travelled,Sep),
    function(Env,Sep,direction,Dir),
    relation(Env,Dir,[[90,degrees]],breakdown(direction)),
    function(Env,Sep,magnitude,Mag),
    interp_meas_mods(NP3,magnitude,Env,Mag).

interp_verb(rest,[(subj,NP),(on,NP1)|PPs],Env) :- !,
    interp_np(Env,NP,Obj),
    interp_np(Env,NP1,Place),
    relation(Env,Obj,Place,on),
    applypps(PPs,Obj,Env).
interp_verb(rest,[(subj,NP)|PPs],Env) :-
    interp_np(Env,NP,Obj),
    new_evar(Place,Env), meta_add(ref(Place),Env),        % slightly hacky
    relation(Env,Obj,[Place],on),
    applypps(PPs,Obj,Env).

interp_verb(run,[(subj,NP1)|PPs],Env) :- !,
    interp_np(Env,NP1,Str),
    constraint(string,Str,Env),
    applypps(PPs,Str,Env).

interp_verb(sit,[(subj,NP),(obj,NP2)],Env) :-
    @num(NP,plur,[]),
    @headnoun(NP,N), lone_ap(NP2,AP),
    interp_ap(Env,[N],AP).
interp_verb(sit,Roles,Env) :- !, interp_verb(rest,Roles,Env).

interp_verb(stand,Roles,Env) :- !, interp_verb(rest,Roles,Env).

interp_verb(stretch,[(_,NP1),(_,NP2)],Env) :-
    @lone_qp(NP2,QP), !,
    interp_np(Env,NP1,Ref),
    function(Env,Ref,stretch,Val),
    interp_qp(QP,stretch,Val,Env).

interp_verb(support,[(subj,NP1),(obj,NP2)|Advs],Env) :- !,
    interp_np(Env,NP1,Supp),
    interp_np(Env,NP2,Obj),
    function2(Env,Supp,Obj,stuck_somewhere,_),
    relation(Env,Supp,Obj,support),
    apply_advs(Advs,Obj,support,Env).
```

```
interp_verb(throw,[(obj,NP)|Mods],Env) :- !,
    interp_np(Env,NP,Ref),
    Mo = motion(Ref,Start,Dest,[M1],M2,Sys),
    current_time(Env,M1),
    opt_point(from,Mods,Start,Env,Mods1),
    opt_point(to,Mods1,Dest,Env,Mods2),
    apply_advs(Mods2,Ref,throw,Env),
    expand(Mo,declare,Env),
    property(Env,Ref,unsupported).

interp_verb(travel,[(subj,NP1),(obj,NP2)],Env) :- !,
    interp_np(Env,NP1,Ref),
    function(Env,Ref,motion_of,Sys),
    interp_indefmeas(NP2,Ref,Env).

interp_verb(weigh,[(subj,NP1),(obj,NP2)],Env) :- !,
    interp_np(Env,NP1,Obj),
    function(Env,Obj,mass,Res),
    @lone_qp(NP2,QP),
    interp_qp(QP,mass,Res,Env).

 nterp_verb(Verb,Roles,Env) :-
    error('I don''t know this use of %p\n',[Verb],fail).
```

```
/* VERSUP
   Support routines for verb semantics
   C.M., 18/6/81
*/

/* Apply all PPs to a given object */

applypps([],_,_) :- !.
applypps([(P,NP)|PPs],Obj,Env) :- !,
   interp_pp(Obj,Env,pp(P,NP)),
   applypps(PPs,Obj,Env).

/* Constrain the type of an object */

constraint(Type,Refs,Env) :-
   newenv(Env,def,Env1),
   property(Env1,Refs,Type), !.

/* Get an optional modifier */

opt_mod(P,Mods,Ref,Env,Mods1) :-
   append(M1,[(P,NP)|M2],Mods),
   append(M1,M2,Mods1), !,
   interp_np(Env,NP,Ref).
opt_mod(_,Mods,_,_,Mods) :- !.

/* Get an optional point specification */

opt_point(P,Mods,Ref,Env,Mods1) :- !,
   opt_mod(P,Mods,Ref1,Env,Mods1),
   (nonvar(Ref1) -> function(Env,Ref1,point_of,Ref); true).

/* Apply adverbs and PPs to a given object */

apply_advs([],_,_,_) :- !.
apply_advs([(adv,A)|Mods],Ref,Verb,Env) :- !,
   interp_adv(Env,A,Verb,Ref), apply_advs(Mods,Ref,Verb,Env).
apply_advs([(P,NP)|Mods],Ref,Verb,Env) :- !,
   interp_pp(Ref,Env,pp(P,NP)),
   apply_advs(Mods,Ref,Verb,Env).
```

```
/* MEAN1
   Expansion of word meanings (below 'distrib' level)
   C.M., 30/6/81
   Use with rest of semantic interpretation stuff
   This file should not mention object-level predicates

   Exports:

        meaning(3)
        testmeaning(3)

   Imports:

        updef(3)
        current_time(2)
        super_type(2)
        test(2)
        declare(2)
*/

/* Ordinary meanings */

meaning(E,above(Mp),(Above,Below)) :- !,
   meaning(E,body_distance,((Below,Above),S)),
   meaning(E,breakdown(separation),(S,Mp:[90,degrees])).

meaning(E,adddim(D),(O,Mp)) :- !,
   adddim(D,Mp,O,E).

meaning(E,at,Objs) :- !,
   adddim(separation,[0,arbs],Objs,E).

meaning(E,breakdown(_),(V,Mp)) :-
   add_type_info(vector,V,E), !,
   meaning(E,magnitude,(V,M)),
   meaning(E,direction,(V,D)),
   breakdown(M,D,Mp,E).
meaning(E,breakdown(_),(M,Mp)) :- !,
   breakdown(M,[],Mp,E).

 eaning(E,ends,(Obj,(E1,E2))) :- !,
   meaning(E,lend,(Obj,E1)),
   meaning(E,rend,(Obj,E2)).

meaning(E,from(Mp),Objs) :- !,
   adddim(separation,Mp,Objs,E).

meaning(E,equilibrium,Ref) :- !,
   adddim(acceleration,[0,arbs],Ref,E).

meaning(E,on,(Above,Below)) :-                  % "on the left end of the rod"
   add_type_info(part,Below,E), !,
   meaning(E,contact_somewhere,((Above,Below),_)).
meaning(E,on,(Above,Below)) :- !,               %  "on the table"
   meaning(E,contact_somewhere,((Above,Below),_)),
   meaning(E,support,(Below,Above)).

meaning(E,over,(Above,Below)) :- !,
   meaning(E,contact_somewhere,((Above,Below),_)),
   meaning(E,support,(Below,Above)).
```

```
meaning(E,roush,Ref) :- !, adddim(coeff,[mu,arbs],Ref,E).

meaning(Env,Key,Refs) :-                                % default case
   current_time(Env,T),
   means(Key,Refs1,T,Tests,Ass),
   match_type(Refs,Refs1), !,
   test(Tests,Env), declare(Ass,Env).
meaning(Env,PerhapsType,X) :-
   match_type(X,_), !,
   trace('>>Assuming null type for %p\n',[PerhapsType],assum),
   declare(entity(X),Env).
meaning(Env,Key,X) :-
   user_supply('the meaning of %p',[Key],Lexp),
   assertz((Key ==> Lexp)), meaning(Env,Key,X).

/* Make sure that meaning has the right functionality */

match_type(R,R1) :- individual(R), !, individual(R1), R=R1.
match_type((A,B),Patt) :- !, nonvar(Patt), Patt=(A1,B1),
   match_type(A,A1), match_type(B,B1).

   individual(X) :- var(X), !.
   individual((_,_)) :- !, fail.
   individual(_).

/* Basic manipulations on measurements */

adddim(Dim,Mp,Obj,Env) :- !,
   meaning(Env,Dim,(Obj,Res)),
   meaning(Env,breakdown(Dim),(Res,Mp)).

breakdown(M,D,Z:[N,U],E) :- var(Z), !,
   meaning(E,measure,(D,(N,U))).
breakdown(M,D,[N1,U1]:[N2,U2],E) :- !,
   meaning(E,measure,(M,(N1,U1))),
   meaning(E,measure,(D,(N2,U2))).
breakdown(M,D,[how_many,U],E) :- !,
   meaning(E,sought,D).
breakdown(M,_,[N,U],E) :- !,
   meaning(E,measure,(M,(N,U))).

/* See if a meaning can be applied */

testmeaning(E,Key,Objs) :- !,
   updef(E,def,E1),
   meaning(E1,Key,Objs).

/* Apply a meaning in a 'new' environment */

addmeaning(E,Key,Objs) :- !,
   updef(E,indef,E1),
   meaning(E1,Key,Objs).
```

```
/* MEAN2.PL
   Primitive word meanings
   C.M., 24/9/81
*/

means(Word,Objs,T,Presupp,Mean) :-
   (Word ==> Vars: Mean where Presupp),
   sort_vars(Vars,Objs,T).
means(Word,Objs,T,true,Mean) :-
   (Word ==> Vars: Mean),
   sort_vars(Vars,Objs,T).

   sort_vars(\Vars,A,B) :- nonvar(Vars), Vars=[A,B], !.
   sort_vars(\A,A,_) :- !.
   sort_vars(\(A,B),(A,B),_) :- !.
   sort_vars(\(A,B,C),(A,B,C),_) :- !.

'acceleration' ==>
   \[(Obj,Accel),T]: sys_accel(Obj,Accel,T).

'age' ==>
   \[(P,A),T]: birth(P,B)&period(Per,B,T)&duration(Per,A).

'applied_force' ==>
   \[(X,Y),T]: reaction(X,Y,R,T) & exerts(X,R,T).

'ball' ==>
   \X: ball(X).

'bar' ==>
   \X: bar(X).

'before' ==>
   \(M1,M2): period(Per,M1,M2).

'bench' ==>
   \X: bench(X).

'body_distance' ==>
   \[((B1,B2),Sep),T]: body_distance(B1,B2,Sep,T).

 born' ==>
   \[Pers,T]: birth(Pers,T).

'bridge' ==>
   \X: bridge(X).

'cliff' ==>
   \X: cliff(X).

'coeff' ==>
   \[(O,C),T]: coeff_friction(O,C,T).

'coeff' ==>
   \[((O1,S),C),T]: coeff_friction(O1,S,C,T).

'connects' ==>
   \(Str,O1,O2): connects(Str,O1,O2,forever).

'constant' ==>
```

```
   \[(O,C),T]: elasticity(O,C,T).

'contact' ==>
   \[(O1,O2),T]: contact(O1,O2,T).

'contact_somewhere' ==>
   \[((O1,O2),P),T]: body_contact(O1,O2,P,T).

'crane' ==>
   \X: crane(X).

'dateof' ==>
   \(Mom,Date): dateof(Mom,Date).

'daughter' ==>
   \(Per,Daus): daughter_of(Per,Daus).

'difference' ==>
   \((Q1,Q2),Q3): difference(Q1,Q2,Q3)&{add(equation(Q3=Q1-Q2))}.

'different' ==>
   \(O1,O2): diff(O1,O2).

'direction' ==>
   \(Vec,Dir): direction(Vec,Dir).

'distance' ==>
   \[(Obj,Dis),T]: dist_travelled(Obj,Dis,T).

'duration' ==>
   \(Per,Dur): duration(Per,Dur).

'edge' ==>
   \(O,E): solid_rightend(O,E).

'end' ==>
   \(O,E): solid_rightend(O,E).

'ends' ==>
   \(O,(E1,E2)): solid_leftend(O,E1)&solid_rightend(O,E2).

 equation' ==>
   \Eqn: {add(equation(Eqn))}.

'father' ==>
   \(Per,Fath): father_of(Per,Fath).

'female' ==>
   \X: female(X).

'fine' ==>
   \[Obj,T]: mass(Obj,Mass,T)&measure(Mass,O,arbs).

'fixed' ==>
   \Obj: fixed(Obj,forever).

'fric_force' ==>
   \[(O,F),T]: friction(S,O,F,T)
               where body_contact(O,S,_,T).
```

```
'sround' ==>
   \sround: {load(sround)}.

'height' ==>
   \[(X,H),T]: length(X,H,T).

'horizontal' ==>
   \0: tangent(0,Tan)&measure(Tan,0,degrees).

'inextensible' ==>
   \0: string(0).

'john' ==>
   \john: man(john).

'known' ==>
   \X: {add(known(X))}.

'lend' ==>
   \(0,E): solid_leftend(0,E).

 ength' ==>
   \[(0bJ,L),T]: length(0bJ,L,T).

'lever' ==>
   \X: lever(X).

'light' ==>
   \[0bJ,T]: mass(0bJ,Mass,T)&measure(Mass,0,arbs).

'magnitude' ==>
   \(Vec,Mag): magnitude(Vec,Mag).

'male' ==>
   \X: male(X).

'man' ==>
   \X: man(X).

'mary' ==>
   \mary: woman(mary).

'mass' ==>
   \X: blob(X).

'mass' ==>
   \[(0bJ,Mass),T]: mass(0bJ,Mass,T).

'measure' ==>
   \(Q,(N,U)): measure(Q,N,U).

'mother' ==>
   \(Per,Mot): mother_of(Per,Mot).

'motion' ==>
   \(0bJ,Motn): true where
                  motion(0bJ,_,_,_,_,Motn)&maximal_motion(0bJ,Motn).

'motion_point' ==>
   \[(0bJ,Place),T]: motion(0bJ,_,Place,_,T,_).
```

```
'motion_start' ==>
   \[(Obj,Place),T]: motion(Obj,Place,_,T,_,_).

'motion_of' ==>
   \[(Obj,Mot),T]: motion(Obj,_,_,Mom1,Mom2,Mot)
                   where period(T,Mom1,Mom2).

'neuter' ==>
   \X: neuter(X).

'other_end' ==>
   \(O,E2): true
              where solid_farend(O,E1,E2) & {test(bound_mentioned(E1))}.

'painter' ==>
   \X: painter(X).

'particle' ==>
   \X: particle(X).

 'eriod' ==>
   \((M1,M2),Per): period(Per,M1,M2).

'period_of' ==>
   \(Event,Per): period_of(Event,Per).

'pier' ==>
   \X: pier(X).

'plane' ==>
   \X: plane(X).

'point' ==>
   \X: point(X).

'point_of' ==>
   \(Obj,Pt): solid_point_of(Obj,Pt).

'pole' ==>
   \X: pole(X).

'product' ==>
   \((Q1,Q2),Q3): {add(equation(Q3=Q1*Q2))}.

'pulley' ==>
   \X: pulley(X).

'ref' ==>
   \X: {add(ref(X))}.

'relvel' ==>
   \[((O1,O2),V),T]: relvel(O1,O2,V,T).

'rend' ==>
   \(O,E): solid_rightend(O,E).

'rod' ==>
   \X: rod(X).
```

```
'rope' ==>
   \X: rope(X).

'scaffold' ==>
   \X: scaffold(X).

'sea' ==>
   \sea: {load(sea)}.

'second' ==>
   \X: true.

'separable' ==>
   \(O1,O2): separable(O1,O2).

'separation' ==>
   \[((ObJ1,ObJ2),Sep),T]: separation(ObJ1,ObJ2,Sep,T).

'separation_travelled' ==>
   \[(ObJ,Sep),T]: period(T,M1,M2) & pathat(ObJ,P11,M1) & pathat(ObJ,P12,M2)
                   & separation(P11,P12,Sep,T).

 small' ==>
   \O: particle(O).

'smooth' ==>
   \[ObJ,T]: coeff_friction(ObJ,Co,T)&measure(Co,O,arbs).

'son' ==>
   \(Par,Son): son_of(Par,Son).

'sought' ==>
   \O: {add(sought(O))}.

'speed' ==>
   \[(ObJ,Sp),T]: vel(ObJ,Sp,T).

'spring' ==>
   \X: spring(X).

 station' ==>
   \X: station(X).

'stationary' ==>
   \[ObJ,T]: vel(ObJ,Vel,T)&magnitude(Vel,Mag)&measure(Mag,O,arbs).

'stone' ==>
   \X: stone(X).

'stretch' ==>
   \[(O,S),T]: extension(O,S,T).

'string' ==>
   \X: string(X).

'stuck' ==>
   \(O1,O2): fixed(O1,O2,forever).

'stuck_somewhere' ==>
   \((O1,O2),Pt): body_fixed(O1,O2,Pt,forever).
```

```
'support' ==>
   \[(Supporter,Supportee),T]: supports(Supporter,Supportee,T).

'surface' ==>
   \X: surface(X).

'system' ==>
   \X: system(X).

'table' ==>
   \X: table(X).

'tangent' ==>
   \(O,Tan): tangent(O,Tan).

'tension' ==>
   \[(Obj,Tens),T]: tension(Obj,Tens,T).

'top' ==>
   \(O,T): solid_leftend(O,T).

 tower' ==>
   \X: tower(X) & tangent(X,T) & measure(T,90,degrees)
       & {load(ground)}
       & solid_rightend(X,Bot) & body_fixed(Bot,ground,_,forever).

'train' ==>
   \X: train(X).

'typical_point' ==>
   \(Obj,Pt): typical_point(Obj,Pt).

'unsupported' ==>
   \[Obj,T]: unsupported(Obj,T).

'velocity' ==>
   \[(Obj,Vel),T]: vel(Obj,Vel,T).

'vertical' ==>
   \O: tangent(O,Tan)&measure(Tan,90,degrees).

'weight' ==>
   \[(Obj,Wgt),T]: mass(Obj,Wgt,T).

'weight' ==>
   \X: blob(X).

'woman' ==>
   \X: woman(X).

'work' ==>
   \[(X,W),T]: work_done(X,W,T)
        where exerts(X,_,T).

'x_axis' ==>
   \x_axis: line(x_axis)&tangent(x_axis,zero).
```

```
/* TIME
   Basic operations on time periods and moments
   C.M., 14/9/81
*/

/* Imports:

        @(1)
        bind(4)
        addmeaning(3)
        current_time(2)
        error(3)
        gensym(2)
        lastof(2)
        tense(2)
        t_focus(3)
        unbound(1)

   Exports:

        final_time(2)
        newmoment(2)
        set_time(2)
        time_focus(3)
*/

/* Set up tense information */

set_tense(S,Env) :- @embedded_sent(S,_), !.
set_tense(S,Env) :- @aux_verb(S,T), tense(Env,T), !.

/* Set up default time at end of sentence */

final_time(S,Env) :-                    % 1. Time already known
   current_time(Env,M),
   not(unbound(M,any,Env)), !,
   set_time(Env,M).
final_time(S,Env) :-                    % 2. Time after last focus
   eventclass(S,Ec), Ec=event,
   tense(Env,T),
   exists_time_focus(T,Env,F), !,
   current_time(Env,C),
   time_after(F,Env,C,M),
   set_time(Env,M).
final_time(S,Env) :-                    % 3. Time as last focus
   tense(Env,T),
   exists_time_focus(T,Env,F), !,
   set_time(Env,F).
final_time(S,Env) :-                    % 4. Time is new moment
   current_time(Env,C),
   add_type_info(moment,C,Env), !,
   newmoment(Env,M),
   set_time(Env,M).
final_time(S,Env) :-                    % 5. Time is new period
   newperiod(Env,P),
   set_time(Env,P).

/* Create a new time after the previous time focus.
   4 cases according to whether the new time (=current time)
   and the focus are periods or moments.
```

```
   Create a moment if possible, otherwise a period
*/

time_after(F,Env,C,M) :-
   add_type_info(moment,C,Env),
   add_type_info(moment,F,Env), !,
   newmoment(Env,M),
   addmeaning(Env,before,(F,M)).
time_after(F,Env,C,M) :-
   add_type_info(moment,C,Env), !,
   addmeaning(Env,period,((_,M),F)).
time_after(F,Env,C,P) :-
   add_type_info(moment,F,Env), !,
   newperiod(Env,P),
   addmeaning(Env,period,((F,_),P)).
time_after(F,Env,C,P) :-
   newperiod(Env,P),
   addmeaning(Env,period,((_,M),F)),
   addmeaning(Env,period,((M,_),P)).

/* Create new times */

._wmoment(Env,M) :-
   gensym(moment,M), add_type_info(moment,M,Env), !.

newperiod(Env,P) :-
   gensym(period,P), add_type_info(period,P,Env), !.

/* Set current time to a specific time */

set_time(Env,Val) :-
   current_time(Env,M),
   nonvar(Val), !,
   bind(any,M,Val,[],Env),
   tense(Env,T),
   t_focus(T,Env,List),
   lastof(List,Val).

/* See if there is already a time focus */

  ists_time_focus(Tense,Env,F) :-
   t_focus(Tense,Env,List), nonvar(List), !,
   lastelt(List,F).

/* Get time focus, creating a moment if necessary */

time_focus(Tense,Env,F) :-
   t_focus(Tense,Env,List), nonvar(List), !,
   lastelt(List,F).
time_focus(Tense,Env,F) :- !,
   t_focus(Tense,Env,List),
   newmoment(Env,F),
   List = [F|_].

   lastelt([X|Y],X) :- var(Y), !.
   lastelt([_|Y],X) :- !, lastelt(Y,X).

/* Does a sentence describe a state or an event? */

eventclass(S,event) :-
```

```
   @main_verb(S,V), verbtype(V,event), !.
eventclass(S,event) :-
   @embedded_sent(S1,S),
   @main_verb(S1,V), verbtype(V,event), !.
eventclass(S,state).

verbtype(raise,event).
verbtype(bear,state).
verbtype(pin,state).
verbtype(support,state).
verbtype(fall,event).
verbtype(stand,state).
verbtype(be,state).
verbtype(connect,state).
verbtype(pass,state).
verbtype(find,state).
verbtype(hang,state).
verbtype(place,state).
verbtype(remain,state).
verbtype(weigh,state).
verbtype(carry,state).
  rbtype(attach,state).
.erbtype(have,state).
verbtype(hold,state).
verbtype(rest,state).
verbtype(suspend,state).
verbtype(contain,state).
verbtype(run,state).
verbtype(drop,event).
verbtype(hit,event).
verbtype(throw,event).
verbtype(reach,event).
verbtype(project,event).
verbtype(calculate,state).
verbtype(sit,state).

verbtype(Verb,Ty) :-
   not(clause(verbtype(Verb,_),true)), !,
   user_supply('type of %t (state/event)',[Verb],Ty).
```

```
/* METACT.PL
   Meta-level actions associated with meanings
   C.M., 24/7/81
*/


meta_act(Env,add(Ass)) :- !,
   trace('>>Entering %p into meta-level database\n',[Ass],meta),
   meta_add(Ass,Env).
meta_act(Env,test(Ass)) :- !,
   trace('>>Looking for meta-level fact %p\n',[Ass],meta),
   meta_test(Ass,Env).
meta_act(Env,use(Operator)) :-
   meta_test(used(Op),Env), unify(any,Op,Operator,[],Env), !, fail.
meta_act(Env,use(Operator)) :- !,
   meta_add(used(Operator),Env).
meta_act(Env,load(Sit)) :- meta_test(loaded(Sit),Env), !.
meta_act(Env,load(Sit)) :- !,
   trace('>>Loading situation ''%p''\n',[Sit],load),
   load_sit(Sit,Env),
   meta_add(loaded(Sit),Env).

/* Load in a standard situation */

load_sit(Sit,Env) :- !,
   add_ext('sit',Sit,File), see(File), read(R),
   process_rule(R,Env), !, seen.

   process_rule(end_of_file,_) :- !.
   process_rule((A<--B),_) :- !, assertz(object_level_rule(A,B)),
      read(R), process_rule(R,Env).
   process_rule((A ==> B)) :- !, assertz((A ==> B)),
      read(R), process_rule(R).
   process_rule(Ass,Env) :- add(Ass,Env), read(R), process_rule(R,Env).
```

```
/* MLXPRO
   Extra meta-level properties
   C.M., 25/8/81
*/

unique_most_general(intersect(X,Y,Z)) :-
   bound(X,forever); bound(Y,forever); bound(Z,forever).
```

```
/* METADB.PL
   Accessing the meta-level database
   C.M., 24/7/81
*/



meta_add(known(X),Env) :- !, mark_known(X,Env).
meta_add(ref(X),Env) :- !, mark_ref(X,Env).
meta_add(Ass,Env) :-
   ase(N), index_add(keys(meta,N,Ass),[]), !.

meta_test(known(X),Env) :- !, known(X,Env).
meta_test(ref(X),Env) :- !, ref(X,Env).
meta_test(bound_mentioned(X),Env) :- !,
   findall(Z,Y^((meta_find(mentioned(Y),Env),bound(Y,any,Env),bound(Y,Z))),L),
   rev_member(A,L), bind(any,X,A,[],Env).
meta_test(Ass,Env) :- meta_find(Ass,Env).

   meta_find(Ass,Env) :- !,
      index_find(Ass,keys(meta,_,Ass),[]).

   rev_member(X,[_|L]) :- rev_member(X,L).
   rev_member(X,[X|_]) :- !.

meta_assertion(Ass,Env) :-
   index_find(meta,keys(meta,_,Ass),[]).
```

```
/* FILTER
   Selecting out assertions for output
   C.M., 22/6/81
*/

filter(measure(A,B,C),E,given(A)) :-
   not(meta_test(sought(A),E)).
filter(known(_),_,_) :- !, fail.
filter(refers(_,_),_,_) :- !, fail.
filter(mentioned(_),_,_) :- !, fail.
filter(loaded(_),_,_) :- !, fail.
filter(used(_),_,_) :- !, fail.
filter(X,_,X).
```

```
/* SEA.SIT
   Rules for sea situations
*/

isa(surface,sea).

isa(solid,sea).

below(sea,_) <-- true.
```

```
/* SCAFF.SIT
   Knowledge about scaffold situations
*/

true <-
```

```
/* EARTH.SIT
   Knowledge about the earth
*/

point(earth).
```

```
/* GROUND.SIT
   Rules for ground situations
*/

surface(ground).

below(ground,_) <-- true.
```

```
/* TIMES.SIT
   Knowledge about times
*/

period(forever,alpha,omega).
```

PTEST

```
/* Check throush a .SYN file */

run(F) :- input(F), so.

/* Display the parser output in another form */

so :- @sentence(S), so1(S), nl, fail.
so.

/* Sentences */

so1(S) :- !, d1(S,0).

d1(S,T) :-
   @conJ(S,S1,ConJ,S2), !,
   nwritef(T,'sentence %t - %t(%t,%t)\n\n',[S,ConJ,S1,S2]),
   d1(S1,T), nl,
   d1(S2,T).

d1(S,T) :-
   nwritef(T,'sentence %t\n\n',[S]),
   T1 is T+1,
   set_sentencetype(S,Ty),
   nwritef(T1,'envir\t%t\n',[Ty]),
   passivity(S,P),
   nwritef(T1,'%t\n',[P]),
   set_tense(S,Te),
   nwritef(T1,'tense\t%t\n',[Te]),
   @main_verb(S,V),
   nwritef(T1,'verb\t%t\n',[V]),
   check_verb_rou(V),
   collectnps(Env,S,NPs),
   dnps(NPs,T1).

dnps([],_) :- !.
dnps([(R,NP)|NPs],T) :- !,
   nwritef(T,'%t\t%t\n',[R,NP]),
   T1 is T+1,
   drole(R,NP,T1),
   dnps(NPs,T).

 _t_sentencetype(S,'n/a') :- @embedded_sent(S,_), !.
set_sentencetype(S,Ty) :- sentencetype(S,Ty).

passivity(S,passive) :- @passive(S), !.
passivity(S,active) :- !.

set_tense(S,'n/a') :- @embedded_sent(S,_), !.
set_tense(S,T) :- @aux_verb(S,T), !.

check_verb_rou(V) :-
   clause(interp_verb(V1,_,_),_), V==V1, !.
check_verb_rou(V) :-
   nosem(V).

/* NPs and other constituents */

drole(adv,A,T) :- !,
   @adv_head(A,Ad), !,
   nwritef(T,'head\t%t\n',[Ad]),
```

```
    chadv(Ad),
    adv_feats(Ad,A,T).

    adv_feats(Ad,A,T) :- @hasfeat(A,F), F\==Ad,
       nwritef(T,'feature\t%t\n',[F]), fail.
    adv_feats(_,_,_).

    chadv(A) :- clause(i_adv(A,_,_,_,_),_), !.
    chadv(A) :- nosem(A).

drole(_,NP,T) :-
    @lone_qp(NP,QP), !,
    dqp(QP,T).

drole(_,NP,T) :-
    @lone_number(NP,N), !,
    nwritef(T,'number\t%t\n',[N]).

drole(_,NP,T) :-
    @lone_ap(NP,AP),
    dap(AP,T).

drole(_,NP,T) :-
    @special_np(NP,N),
    (N=you; N=what), !,
    nwritef(T,'special\t%t\n',[N]).

drole(_,NP,T) :-
    @poss_det(NP,NP1), !,
    nwritef(T,'det\t%t\n',[NP1]),
    T1 is T+1,
    drole(np,NP1,T1),
    @fnof(NP,Fn),
    nwritef(T,'function\t%t\n',[Fn]),
    chsem(Fn).

drole(_,NP,T) :-
    @name(NP,Na), !,
    nwritef(T,'name\t%t\n',[Na]),
    chname(Na).

    chname(Na) :- namesender(Na,_), !.
    chname(Na) :- nosem(Na).

drole(_,NP,T) :-
    @wh_trace(NP1,NP), !,
    nwritef(T,'trace\t%t\n',[NP1]).

drole(_,NP,T) :-
    @conj(NP,NP1,NP2), !,
    nwritef(T,'conjunction\t%t %t\n',[NP1,NP2]),
    T1 is T+1,
    drole(np,NP1,T1),
    drole(np,NP2,T1).

drole(_,NP,T) :-
    @quant_front(NP,NP1), !,
    nwritef(T,'quantifier + %t\n',[NP1]),
    drole(np,NP1,T).
```

```
drole(_,NP,T) :-
   @fnof(NP,Dim),
   not(@qp_det(NP,_)),
   @num(NP,_,def), !,
   nwritef(T,'function\t%t\n',[Dim]),
   chsem(Dim),
   dmars(NP,Dim,T).

   dmars(NP,D,T) :- prep_of(D,P),
      @pp(NP,P,NP1), !,
      nwritef(T,'ars\t%t\n',[NP1]),
      T1 is T+1,
      drole(np,NP1,T1).
   dmars(NP,_,T) :- @relc(NP,S), !,
      nwritef(T,'relc\t%t\n',[S]),
      T1 is T+1,
      d1(S,T1).
   dmars(NP,_,T) :- !,
      nwritef(T,'ars\tempty\n',[]).

drole(_,NP,T) :-
   @fnof(NP,Dim),
   not(@qp_det(NP,_)), !,
   nwritef(T,'indmeas\t%t\n',[Dim]),
   T1 is T+1,
   chsem(Dim),
   imars(NP,T).

   imars(NP,T) :-
      @pp(NP,of,NP1), @qp_det(NP1,QP), !, dqp(QP,T).
   imars(NP,T) :-
      @qp_modify(NP,QP), !, dqp(QP,T).
   imars(NP,T) :-
      @relc(NP,S),
      nwritef(T,'relc\t%t\n',[S]),
      d1(S,T).

drole(_,NP,T) :-
   @headnoun(NP,N),
   clause(specialref(N1,_,_,_),_), N1==N, !,
   nwritef(T,'special\t%t\n',[N]),
   feats(NP,T).

drole(_,NP,T) :-
   @headnoun(NP,N), pron(N,_,_), !,
   nwritef(T,'pronoun\t%t\n',[N]).

drole(_,NP,T) :-
   @num(NP,Nu,De),
   nwritef(T,'number\t%t\n',[Nu]),
   nwritef(T,'envir\t%t\n',[De]),
   @headnoun(NP,N),
   doptname(NP,T),
   nwritef(T,'noun\t%t\n',[N]),
   chsem(N),
   dmods(NP,T).

   doptname(NP,T) :- @post_name(NP,N), !,
      nwritef(T,'ref\t%t\n',[N]).
   doptname(_,_) :- !.
```

```
   dmods(NP,T) :-
      @qp_det(NP,QP),
      nwritef(T,'qp_det\t%t\n',[QP]),
      T1 is T+1,
      dqp(QP,T1), fail.
   dmods(NP,T) :-
      @hasfeat(NP,AP),
      nwritef(T,'ap\t%t\n',[AP]),
      T1 is T+1,
      dap(AP,T1), fail.
   dmods(NP,T) :-
      @pp(NP,P,NP1),
      nwritef(T,'pp\t%t\n',[P]),
      chp(P),
      T1 is T+1,
      drole(np,NP1,T1), fail.
   dmods(NP,T) :-
      @relc(NP,S),
      nwritef(T,'relc\t%t\n',[S]),
      T1 is T+1,
      d1(S,T1), fail.
   dmods(_,_).

   chp(with) :- !.
   chp(of) :- !.
   chp(P) :- clause(i_pp(_,_,P1,_),_), P==P1, !.
   chp(P) :- chsem(P).

/* Odd bits */

feats(A,T) :- @hasfeat(A,F),
   nwritef(T,'feature\t%t\n',[F]),
   fail.
feats(_,_) :- !.

dqp(QP,T) :-
   measures(QP,Mpl), !,
   nwritef(T,'qp\t%t\n',[Mpl]).
dqp(how_much,T) :- !,
   nwritef(T,'qp\thow much\n',[]).

dap(AP,T) :-
   @head_adjective(AP,A),
   nwritef(T,'head\t%t\n',[A]),
   chadj(A),
   dapmods(AP,T).

   chadj(A) :- dimadj(A,D,_), !, chsem(D).
   chadj(A) :- chsem(A).

   dapmods(AP,T) :-
      @qp_modify(AP,QP),
      dqp(QP,T), fail.
   dapmods(AP,T) :-
      @intensifier(AP,H),
      nwritef(T,'intens\t%t\n',[H]),
      fail.
   dapmods(AP,T) :-
      @comparative(AP,NP),
```

```
      nwritef(T,'compar\t%t\n',[NP]),
      T1 is T+1,
      drole(np,NP,T1), fail.
   dapmods(_,_).

/* Utilities */

chsem(W) :-
   clause(meaning(_,W1,P1),_), W1==W, !.
chsem(W) :-
   means(W,P1,_,_,_), !.
chsem(W) :- nosem(W).

nosem(V) :- !,
   writef('** Warning - no semantics for %t\n',[V]).

nwritef(T,M,As) :- !,
   ttab(T), writef(M,As).

   ttab(0) :- !.
   ttab(N) :- put(9), N1 is N-1, ttab(N1).

/* Get input from a file */

input(File) :-
   flush,
   add_ext("syn",File,File1),
   nice(File1,File2),
   see(File2), repeat,
   read(T), proc(T),
   !, seen.

   nice(F,F) :- file_exists(F), !.
   nice(F1,F2) :-
      name(F1,Na1),
      append("syn:",Na1,Na2),
      name(F2,Na2).

   proc(end_of_file) :- !.
   proc(Z) :- srecord(Z), fail.

   srecord(Z) :- recordz(Z,@Z,_),
      recorda(@(_),Z,_), !.

/* Get an assertion from the parser */

@@Z :- recorded(Z,@Z,_).

/* Flush all input */

flush :- recorded(@(_),Z,P),
   erase(P), recorded(Z,@Z,P1), erase(P1), fail.
flush.
```

```
/* REWRITE.CPL
   Rewriting to create cvars, etc
*/

/* Imports:
        newevar/2,
        newuvar/2,
        mark_existential/2
*/


:- public
        skolemise/2,
        universalise/2,
        fix/2.

/* Skolemise an entity, changing variables into evars */

skolemise(S,Env) :- var(S), !,
   newevar(S,Env).
skolemise(_,_) :- !.

/* Change variables into uvars */

universalise(S,Env) :- var(S), !,
   newuvar(S,Env).
universalise(_,_) :- !.

/* Fix any universal variables */

fix(A,Env) :- nonvar(A), mark_existential(A,Env), !.
fix(_,_).
```

```
/* 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

.- mode ty_process(+),
        basify(+,+,-),
        hidden_ors(+,-),
        add_hidden_ands(+,+),
        type_name(+,-),
        ty_nmember(+,-,-),
        subtype(+,?),
        ty_intersect(+,-),
        maketype(+,+),
        type_pattern(+,-).

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

ty_process(X <-> Y) :- !,
   add_rule(X,Y), fail.
ty_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.
ty_process(Fin) :-
   load_finish(Fin), !,
   finish_types,
   load_start(Fin).
ty_process(Garb) :-
   errmess('Invalid type specification',Garb), fail.

finish_types :-
   rewrite_types,
   do_basic_types,
   do_derived_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.

   basify(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).

/* Deal with and/or tree of basic types */

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

treep(Type,Patt,Pattars) :-
   findall(RHS,or_rule(Type,RHS),List),
   length(List,Arity),
   type_name(Type,Name),
   functor(Pattars,Name,Arity),
   maketype(Type,Patt),
   ty_nmember(List,N,RHS1),
   ars(N,Pattars,NewPattars),
   subtype(RHS1,Type1),
   treep(Type1,Patt,NewPattars),
   fail.
treep(_,_,_).

type_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), !.

/* Retrieving 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.
```

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

                                                        Chris
                                                        Updated: 1/8/81
*/

/*
   Imports:

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

*/

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

 - mode ty_process(+),
        basify(+,+,-),
        hidden_ors(+,-),
        add_hidden_ands(+,+),
        type_name(+,-),
        ty_nmember(+,-,-),
        subtype(+,?),
        ty_intersect(+,-),
        maketype(+,+),
        type_pattern(+,-).

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

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

ty_process(X <-> Y) :- !,
   add_rule(X,Y), fail.
ty_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.
ty_process(Fin) :-
   load_finish(Fin), !,
   finish_types,
   load_start(Fin).
ty_process(Garb) :-
   errmess('Invalid type specification',Garb), fail.

finish_types :-
   rewrite_types,
   do_basic_types,
```

```
    do_derived_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.

    basify(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).

/* Deal with and/or tree of basic types */

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

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

type_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),
   ty_intersect(RHS,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), !.

   Retrieving 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.
```