/*evar,s*/
/*Evan's Geometric Analogy Program - Rational Reconstruction*/
/*Alan Bundy 26.10.79*/

/*top level program*/
eyans(FisA,FisB,FisC,AnsList,Ans) :-
  findrule(FisA,FisB,Rule), Rule_is(Rule),
  applyrule(Rule,FisC,AnsObjs,AnsRels,Sims),
  ans_desc(AnsObjs,AnsRels,Sims),
  selectresult(FisC,AnsList,AnsObjs,AnsRels,Sims,Ans),
  ans_is(Ans).

/*find rule given figures*/
findrule(FisA,FisB,Rule) :-
  relations(FisA,Source), relations(FisB,Target),
  objects(FisA,Alist), objects(FisB,Blist),
  similarities(FisA,FisB,Triples),
  select_set(Triples,Matches),
  takeaway1(Alist,Matches,Removals),
  takeaway2(Blist,Matches,Adds),
  makerule(Removals,Adds,Matches,Source,Target,Rule).

/*apply rule to fis to produce answer*/
applyrule(rule(Removals,Adds,Matches,Source,Target),
           FisC, AnsObjs,Target,Matches) :-
  relations(FisC,FisDesc), objects(FisC,ObList),
  metag(FisDesc,Source),
  maplist(second,Matches,NewList),
  append(NewList,Adds,AnsObjs).

/*Select Result from those Provided */
selectresult(FisC,[FisN!Rest],AnsObjs,AnsRels,AnsSims,FisN) :-
  relations(FisN,NRel), setea(NRel,AnsRel),
  similarities(FisC,FisN,NSims), setea(NSims,AnsSim),
  objects(FisN,NObjs), setea(NObjs,AnsObjs).

selectresult(FisC,[FisN!Rest],AnsObjs,AnsRels,AnsSims,Ans) :-
  selectresult(FisC,Rest,AnsObjs,AnsRels,AnsSim,Ans).

/*select legal subset of similarity triples for matches*/
select_set(Triple,Match) :- select_set1([],[],Triple,Match).

select_set1(Aused,Bused,[],[]) :-
  select_set1(Aused,Bused,[[Aobj,Bobj,Trans]|Rest],[[Aobj,Bobj,Trans]|Rest1]) :-
    not(member(Aobj,Aused)), not(member(Bobj,Bused)),
    select_set1(Aused,Aused),[Bobj|Bused]|Rest,Rest1).

select_set1(Aused,Bused,[[Aobj,Bobj,Trans]|Rest],Rest1) :-
  select_set1(Aused,Bused,Rest,Rest1).

/*take away the triples from the list*/
takeaway1(List,Triples,Ans) :-
  maplist(first,Triples,Firsts), subtract(List,Firsts,Ans).
takeaway2(List,Triples,Ans) :-
  maplist(second,Triples,Seconds), subtract(List,Seconds,Ans).
/* First and second elements of a list */
\ filament([A,B,C], A).
\ filament([A,B,C], B).

/* make rule from descriptions inherited from figs a & b */
\ makerule(Removals, Adds, Matches, Source, Target, Rule) :-
   \ \ filament(first, Matches, SPairs), \ filament(second, Matches, TPairs),
   \ \ append(Removals, SPairs, L1), \ append(L1, TPairs, L2),
   \ \ append(L2, Adds, Consts),
   \ \ unbind(Consts, Substs),
   \ \ subst(Substs, rule(Removals, Adds, Matches, Source, Target), Rule).

/* find corresponding variable for each constant and produce substitution */
\ unbind([], true).
\ unbind([Const|Rest], Const=X & Rest1) :-
   \ unbind(Rest, Rest1).

/* Messages */
/*--*/
\ rule_is(rule(Removals, Adds, Matches, Source, Target)) :-
   \ \ writef('Rule is:
      \ removal: %t
      \ add: %t
      \ match: %t
      \ source: %t
      \ target: %t 
      \n      ', Removals, Adds, Matches, Source, Target).
\ ans_desc(ObJs, Rels, Sims) :-
   \ \ writef('Answer description is:
      \ objects: %t
      \ relations: %t
      \ similarities: %t 
      \n      ', ObJs, Rels, Sims).
\ ans_is(Ans) :-
   \ \ writef('Answer is %t
      \n      ', Ans).
/*figures*/
/*test descriptions for evans program*/
/*Alan Bundy 26.10.79*/

problem1(Ans) :- evans(fisa,fisb,fisc,[fis1,fis2,fis3,fis4,fis5],Ans).
problem2(Ans) :- evans(fisa,fisb,fisc,[fis1,fis2,fis3,fis4a,fis5],Ans).
problem3(Ans) :- evans(fisa,fisb,fisc,[fis1,fis2,fis3,fis5],Ans).

objects(fisa,[tri1,tri2]),
relations(fisa,[[inside,tri1,tri2]]).

objects(fisb,[tri3]),
relations(fisb,[]).

similarities(fisa,fisb,[[tri2,tri3,direct],[tri1,tri3,[scale,2]]]).

objects(fisc,[square,circle]),
relations(fisc,[[inside,square,circle]])

objects(fis1,[circle2,circle3]),
relations(fis1,[[inside,circle2,circle3]]),
similarities(fisc,fis1,[[circle,circle3,direct],[circle,circle2,[scale,half]]]).

objects(fis2,[square2]),
relations(fis2,[]),
similarities(fisc,fis2,[[square,square2,direct]]).

objects(fis3,[tri4,circle4]),
relations(fis3,[[inside,tri4,circle4]]),
similarities(fisc,fis3,[[circle,circle4,direct]]).

objects(fis4,[circle5]),
relations(fis4,[]),
similarities(fisc,fis4,[[circle,circle5,direct]]).

objects(fis4a,[square3]),
relations(fis4a,[]),
similarities(fisc,fis4a,[[square,square3,[scale,2]]]).

objects(fis5,[tri5]),
relations(fis5,[]),
similarities(fisc,fis5,[]).
yes
! ?- problem1(A).
Rule is:
  remove: [_527]
  add: []
  match: [[_537, _547, direct]]
  source: [[inside, _527, _537]]
  target: []

Answer description is:
  objects: [_547]
  relations: []
  similarities: [[circle, _547, direct]]

Answer is fis4

A = fis4

yes
! ?- problem2(A).
Rule is:
  remove: [_527]
  add: []
  match: [[_537, _547, direct]]
  source: [[inside, _527, _537]]
  target: []

Answer description is:
  objects: [_547]
  relations: []
  similarities: [[circle, _547, direct]]

Rule is:
  remove: [_527]
  add: []
  match: [[_537, _547, scale, 2]]
  source: [[inside, _537, _527]]
  target: []

Answer description is:
  objects: [_547]
  relations: []
  similarities: [[square, _547, scale, 2]]

Answer is fis4a

A = fis4a

yes
! ?- problem3(A).
Rule is:
  remove: [_527]
  add: []
  match: [[_537, _547, direct]]
  source: [[inside, _527, _537]]
Answer description is:
objects: [547]
relations: []
similarities: [[circle, 547, direct]]

Rule is:
remove: [527]
add: []
match: [[537, 547, [scale, 2]]]
source: [[inside, 537, 527]]
target: []

Answer description is:
objects: [547]
relations: []
similarities: [[square, 547, [scale, 2]]]

Rule is:
remove: [381, 391]
add: [401]
match: []
source: [[inside, 381, 391]]
target: []

Answer description is:
objects: [401]
relations: []
similarities: []

Answer is fig5

A = fig5

yes

<table>
<thead>
<tr>
<th>core</th>
<th>49664</th>
<th>(20480 lo-seg + 29184 hi-seg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>phys</td>
<td>15360 = 14048 in use + 1312 free</td>
<td></td>
</tr>
<tr>
<td>glob</td>
<td>1187 = 16 in use + 1171 free</td>
<td></td>
</tr>
<tr>
<td>local</td>
<td>1024 = 16 in use + 1008 free</td>
<td></td>
</tr>
<tr>
<td>trail</td>
<td>511 = 0 in use + 511 free</td>
<td></td>
</tr>
</tbody>
</table>

0.06 sec. for 1 GCs gaining 144 words
0.08 sec. for 4 local shifts and 12 trail shifts
1.77 sec. runtime
DEPARTMENT OF ARTIFICIAL INTELLIGENCE
PROLOG PROGRAM LIBRARY
PROGRAM SPECIFICATION

Program Name The Evans Geometric Analogy Program
Source Alan Bundy
Date of Issue 8 May 1981

1. Description

This is a rational reconstruction of the second part of Evan's Program to solve Geometric Analogy Problems [Evans 64, Bundy 80]. It uses the descriptions of geometric figures to form a rule; applies this rule to form the description of an answer figure and then uses this description to select an answer from those provided. The first part of Evan's Program, in which figure descriptions are formed from a cartesian representation, is not attempted.

2. Method of Use

To use the program, type

    run PLL: UTIL

and in response to the prompt type

    consult('PLL: EVANS').

The top level predicate, evans, will then be available. evans takes 5 arguments: the names of figures A, B, C; a list of possible answer figures and a variable. If it succeeds then evans binds this variable to one of the answer figures. As it runs evans prints descriptions of the rules it forms and the answer figures these suggest.

For the program to work, assertions must be present in the database describing the figures A, B, C and the possible answer figures and relating objects in figure A to objects in figure B and objects in figure C to objects in each of the answer figures. Some examples of the sort of assertions required are given in file PLL: FIGURE and a selection are reproduced in example 2-1 below.
objects(fisa,[tri1,tri2]).
relations(fisa,[inside,tri1,tri2]).

objects(fisb,[tri3]).
relations(fisb,[ ]).

similarities(fisa,fisb,[tri2,tri3,direct],
[tri1,tri3,[scale,2]]).

objects(fisc,[square,circle]).
relations(fisc,[inside,square,circle]).

Figure 2-1: Assertions to Describe Input Figures

- objects takes a figure name and a list of the objects occurring in the figure.
- relations takes a figure name and a list of symbolic descriptions describing the figure.
- similarities takes two figure names and a list of similarities between objects in the two figures.

The file, PLL: FIGURE, can be input by typing

consult('PLL: FIGURE').

The particular assertions defining fisa, fisb and fisc given above, plus similar ones for fis1-5 will then be in the database and the call

evans(fisa,fisb,fisc,[fis1,fis2,fis3,fis4,fis5],Ans).

will then succeed and bind Ans to fis4.

3. How it Works

The top level predicate, evans, is defined as follows (omitting print messages):

evans(FisA,FisB,FisC,AnsList,Ans) :-
  find_rule(FisA,FisB,Rule),
  apply_rule(Rule,FisC,AnsObjs,AnsRels,Sims),
  select_result(FisC,AnsList,AnsObj,AnsRels,Sims,Ans).
find_rule takes the names of figures A and B and forms a description of the rule relating them. apply_rule takes this rule and applies it to figure C, producing a description of the answer figure and similarities between figure C and the answer figure. select_result takes these descriptions and compares them to the available answer figures until it finds a match.

find_rule is defined as follows:

```prolog
find_rule(FigA,FigB,Rule) :-
    relations(FigA,Source), relations(FigB,Target),
    objects(FigA,Alist), objects(FigB,Blist),
    similarities(FigA,FigB,Triples),
    select_set(Triples,Matches),
    takeaway1(Alist,Matches,Removals),
    takeaway2(Blist,Matches,Adds),
    make_rule(Removals,Adds,Matches,Source,Target,Rule).
```

The first five subprocedures pick up the prestored descriptions of the two figures. The predicate select_set picks a legal subset, Matches, of the similarity descriptions between objects in figure A and B. The rule description is based on this subset. On backup a different subset will be chosen and a different rule formed. Objects occurring in figure A, but not involved in a match are then selected by takeaway1 and these become the Removals.

Similarly, takeaway2 finds those objects occurring in figure B, but not involved in a match and these become the Adds. A rule description is then formed from these components by make_rule.

select_set and the two takeaways are fairly straightforward list manipulation, but make_rule requires some explanation. It is defined as follows:

```prolog
make_rule(Removals,Adds,Matches,Source,Target,Rule) :-
    maplist(first,Matches,Spairs), maplist(second,Matches,Tpairs),
    append(Removals,Spairs,L1), append(L1,Tpairs,L2),
    append(L2,Adds,Consts),
    unbind(Consts,Substs),
    subst(Substs,rule(Removals,Adds,Matches,Source,Target),Rule).
```

A rule description consists of the predicate rule with five arguments:

- A list of objects to be removed;
- A list of objects to be added;
- A list of matches between objects;
- The relations in the source figure and
The relations in the target figure.

These are pretty much as supplied to make_rule except that the actual constants inherited from the initial figure descriptions have to be changed for Prolog variables, so that the rule can be applied to figure C.

The first five subprocedures of the make_rule definition consist of making a list, Consts, of all the object names appearing in the Removal, Adds and Matches. A substitution is then built by unbind, in which each of these object names is associated with a different Prolog variable. subst then substitutes these variables for the objects in the rule description. The definitions of unbind, first and second are straightforward and the definitions of maplist and subst are given in the Mecho utility program library.

The definition of apply_rule is given below.

```
apply_rule(rule(Removals, Adds, Matches, Source, Target),
            FisC, AnsObjs, Target, Matches) :-
    relations(FisC, FisDesc),
    objects(FisC, ObList),
    seteq(FisDesc, Source),
    maplist(second, Matches, NewList),
    append(NewList, Adds, AnsObjs).
```

The first two subprocedures recover the description of figure C. The predicate seteq then pattern matches the relations in figure C with the rule source description, which binds the first half of the similarity descriptions in Matches. seteq is set equality and is described in the Mecho utilities library. The objects in the answer figure are calculated by appending the Adds to the objects in the second half of the similarity descriptions in Matches.

The definition of select_result is:

```
select_result(FisC, [FisN!Rest], AnsObjs, AnsRel, AnsSim, FisN) :-
    relations(FisN, NRel),
    seteq(NRel, AnsRel),
    similarities(FisC, FisN, NSim),
    seteq(NSim, AnsSim),
    objects(FisN, NObjs),
    seteq(NObjs, AnsObjs).

select_result(FisC, [FisN!Rest], AnsObjs, AnsRel, AnsSim, Ans) :-
    select_result(FisC, Rest, AnsObjs, AnsRel, AnsSim, Ans).
```

select_result is defined by recursion on the list of available answers. If it recurses to the ends of this list then it fails, and evans backs up to form a new rule. The second clause is merely the recursive step; the first clause makes the interesting comparisons. FisN is the first answer figure name in the list. In turn the relations, objects and similarities of FisN are found in the database, and these are compared with FisC using seteq. FisN is returned as the answer iff the clause succeeds. Otherwise select_result recurses.
4. Program Requirements

The Prolog system takes 30K words and PLL: UTIL, PLL: EVANS, PLL: FIGURE and working space require an additional 21K words. The following predicates are used by the program:

ans_desc(3)  ans_is(1)  append(3)  apply_rule(5)
evans(5)   find_rule(3)  first(2)  make_rule(6)
maplist(3)  member(2)  objects(2)  problem1(1)
Problem2(1)  Problem3(1)  relations(2)  rule_is(1)
second(2)  select_result(6)  select_set(2)  select_set1(4)
settea(2)  similarities(3)  subst(3)  subtract(3)
takeaway1(3)  takeaway2(3)  unbind(2)  writef(2)

REFERENCES

Bundy 80]
Bundy, A.
Additional AI1 problem solving notes.

[Evans 64]
Evans, T.G.
A heuristic program to solve geometric analogy problems.
J.S.C.C., April, 1964.
Rational Reconstruction of Winston Learning Program
Alan Bundy 1.12.80
Version for functions */

/* Top Level Program - learn new concept */
/* ------------------------------------- */

/* First time only accept an example */
winston(Concept) :- !,
  writef('Please give me an example of a %t
', [Concept]),
  read(Ex), nl,
  make_rec(Concept,Ex,EObjs,ERec),
  maplist(sensym(Plato),EObjs,CObjs),
  make_subst(EObjs,CObjs,Subst),
  subst(Subst,ERec,CRec),
  maplist(add_ups,CRec,CDefn),
  assert(definition(Concept,CObjs,CDefn)),
  winstonl(Concept).

/* Is grey area in definition eliminated? */
winstonl(Concept) :-
  definition(Concept,CObjs,CDefn),
  checklist(same,CDefn), !,
  writef('I have learnt the concept of %t now. 
', [Concept]).

/* Subsequently accept either examples or near misses */
winstonl(Concept) :- !,
  writef('Please give me an example or near miss of a %t. 
', [Concept]),
  read(Ex), nl,
  writef('Is this example (yes/no)? 
', [YesNo]),
  read(YesNo), nl,
  learn(Concept,Ex,YesNo),
  winstonl(Concept).

/* Add default upper bounds in concept record */
add_ups(record(ArRs,Name,Posn), define(ArRs,Name,[],Posn)).

/* Slight modify to sensym, so it can be used in maplist */
sensym1(Prefix,_,NewConst) :- !, sensym(Prefix,NewConst).

/* Are upper and lower bound of concept definition the same? */
same(define(ArRs,Name,Posn,Posn)).

/* Learn from this example or near miss */
learn(Concept, Example, YesNo) :- !,
  definition(Concept,CObjs,CDefn),
  make_rec(Concept,Example,EObjs,ERec),
  classify(CObjs,EObjs,CDefn,ERec,Diff,Verdict),
  learn(Concept,Diff,YesNo,Verdict).

/* Make records from list of relations */
/* ------------------------------------- */
make_rec(Concept,Example,EObjs,ERec) :- !,
  example(Example,Relns),
  maplist(consts_in,Relns,CL), flatten(CL,EObjs),
maplist(convert, Relns, ERec).
/* Find all constants in terms */
consts_in([], []).
consts_in(N, []) :-
    integer(N), !.
consts_in(Const, [Const]) :-
    atom(Const), !.
consts_in(Exp, Consts) :-
    Exp =.. [Sym:Arss], maplist(consts_in, Arss, CL),
    flatten(CL, Consts).
/* Flatten List */
flatten([], []).
flatten([Hd|Tl], Ans) :-
    flatten(Tl, Rest), union(Hd, Rest, Ans).
/* Convert input relation style into internal representation as predicate tree */
convert(Reln, record(Arss, Name, ExPosn)) :-
    Reln =.. [Pred:Arss],
    length(Arss, N),
    tree(Name, N, Tree),
    position(Pred, Tree, ExPosn).
/* Find Position of Node in Tree */
position(Node, Tree, []) :-
    Tree =.. [Node:SubTrees],
position(Node, Tree, [N|Posn]) :-
    Tree =.. [Root:SubTrees],
    nth_el(N, SubTrees, SubTree),
    position(Node, SubTree, Posn).
/* find nth element of list */
nth_el(1, [Hd|Tl], Hd),
nth_el(N, [Hd|Tl], El) :-
    nth_el(PN, Tl, El), N is PN + 1.
/* Is this example, non-example or in grey area, by my definition? */
classify(CObJs, EObJs, CDefn, ERec, BestDiff, Verdict) :- !,
    findall(Diff, make_diff(CObJs, EObJs, CDefn, ERec, Diff), Diffs),
    best(Diffs, BestDiff),
    verdict(BestDiff, Verdict).
/* Find the difference between example and concept */
make_diff(CObJs, EObJs, CDefn, ERec, Diff) :- !,
    Perm(EObJs, EObJs1), make_subst(EObJs1, CObJs, Subst),
    subst(Subst, ERec, ERec1),
    pair_off(CDefn, ERec1, Diff).
/* Pair off concept definition and example record to make differences */
pair_off([], [], []) !.
pair_off([], ERec, Diff) :- !, 
    maplist(new_defn, ERec, Diff).

pair_off(CDefn, [], Diff) :- !, 
    maplist(extra_rec, CDefn, Diff).

pair_off([define(Arrix, Name, UPPosn, LowPosn) | CDefn], 
    ERec, 
    [differ(Arrix, Name, UPPosn, ExPosn, LowPosn, Verdict) | Diff]) :- !,
    select(record(Arrix, Name, ExPosn), ERec, Rest), !,
    compare(UPPosn, ExPosn, LowPosn, Verdict),
    pair_off(CDefn, Rest, Diff).

/* invent new bits of definition as necessary */
new_defn(record(Arrix, Name, ExPosn), 
    differ(Arrix, Name, [], ExPosn, DfPosn, Verdict)) :- !,
    default_posn(Name, DfPosn), 
    compare(ExPosn, DfPosn, LowPosn, Verdict).

/* invent extra bits of example record as necessary */
extra_rec(define(Arrix, Name, UPPosn, LowPosn), 
    differ(Arrix, Name, UPPosn, DfPosn, LowPosn, Verdict)) :- !,
    default_posn(Name, DfPosn), 
    compare(UPPosn, DfPosn, LowPosn, Verdict).

/* Find position of default predicate on tree */
default_posn(TreeName, Posn) :- !,
    default(TreeName, Pred), !,
    tree(TreeName, _, Tree), position(Pred, Tree, Posn).

default_posn(TreeName, []).

/* Compare positions in tree to give verdict */
compare(U, E, L, yes) :- append(L, _, E), !,
    compare(U, E, L, grey) :- append(U, _, E), !,
    compare(U, E, L, no) :- !.

/* Find best difference and return it */
est(Diffs, Diff) :- !, 
    maplist(score, Diffs, Scores), 
    lowest(Diffs, Scores, Diff, Score).

/* Return difference with lowest score */
lowest([Diff], [Score], Diff, Score) :- !,
    lowest([Diff1|Diffs], [Score1|Scores], Diff2, Score2) :- !,
    lowest(Diffs, Scores, Diff2, Score2), Score2<Score1, !.

lowest([Diff|Diffs], [Score|Scores], Diff, Score) :- !.

/* Find score of difference */
score(Diff, Score) :- !,
    maplist(score1, Diff, Scores), 
    sumlist(Scores, Score).

/* Find score of individual differ */
score1(differ(_, _, _, _, yes), 0) :- !.
score1(differ(_,_,_,_,_,grey), 1) :- !.
score1(differ(_,_,_,_,_,no), 2) :- !.

/* add up all the numbers in a list */
sumlist([], 0),
sumlist([N|Rest], Total) :- !,
    sumlist(Rest, SubTotal), Total is SubTotal + N.

/* Make a substitution for replacing all members of one list
   by corresponding members of another list */
make_subst([], [], true).
make_subst([X|XRest], [Y|YRest], X=Y & Subst) :-
    make_subst(XRest, YRest, Subst).

/* Decide whether example falls inside definition on basis of differs */
verdict(Diff, Yes) :- checklist(verdict1(Yes), Diff), !.
verdict(Diff, no) :- some(verdict1(no), Diff), !.
verdict(Diff, dgrey) :- some(verdict1(dgrey), Diff), !.

/* verdict on individual differ */
verdict1(V, differ(_,_,_,_,_,V)).

/* adjust definition appropriately */
/* -------------------------------- */
/* if new example found */
learn1(Concept, Diff, Yes, dgrey) :- !,
    writef('This is a new sort of %t. \n', [Concept]),
    maplist(lub, Diff, New),
    retract(definition(Concept, CObjs, Old)),
    assert(definition(Concept, CObjs, New)).

/* if near miss found */
learn1(Concept, Diff, no, dgrey) :- !,
    writef('This limits my idea of %t. \n', [Concept]),
    one_of(exclude, Diff, Diff1),
    maplist(diff_to_defn, Diff1, New),
    retract(definition(Concept, CObjs, Old)),
    assert(definition(Concept, CObjs, New)).

/* if nothing new is discovered */
learn1(Concept, Diff, Asree, Asree) :- !,
    writef('I have seen one of these before. \n', [Concept]),
    fail.

/* or if contradiction is discovered */
learn1(Concept, Diff, Asree, Disasree) :- !,
    writef('Uh Oh, somethings sone wrong. I will think again.\n', [Concept]),
    fail.

/* Move lower definition up a bit to include new example */
lub(differ(Ars, Name, UpPosn, ExPosn, Old, grey),
    define(Ars, Name, UpPosn, Old, New)) :- !,
    common(ExPosn, Old, New).

/* Lower definition already includes new example */
lub(differ(Ars, Name, UpPosn, ExPosn, LowPosn, yes),
    define(Ars, Name, UpPosn, LowPosn)) :- !.
/** Move upper definition down a bit to exclude near miss */
exclude(differ(Arga, Name, Old, ExPosn, LowPosn, grey),
    differ(Arga, Name, New, ExPosn, LowPosn, grey)) :- !,
    common(ExPosn, LowPosn, Comm), append(Comm, [N!], LowPosn),
    append(Comm, [N], New).

/** Take unnecessary bits out of difference */
diff_to_defn(differ(Arga, Name, UPPosn, ExPosn, LowPosn, Verdict),
    define(Arga, Name, UPPosn, LowPosn)).

/* Find common initial sublist of two lists */
common([N!Rest1], [N!Rest2], [N!Rest]) :- !,
    common(Rest1, Rest2, Rest).

common(List1, List2, []) :- !.

/* change just one member of list */
one_of(Prop, [Old:1], [New:1]) :- apply(Prop, [Old, New]),
one_of(Prop, [Head:Old], [Head:New]) :- one_of(Prop, Old, New),

/* Find out what grey areas still exist in concept */
grey(Concept) :- !,
    writeln('Grey areas in %t are:
', [Concept]),
    definition(Concept, CObjs, CDefn),
    checklist(srey1, CDefn).

srey1(define(Arga, Name, Posn, Posn)) :- !.
srey1(Defn) :- !,
    write(Defn), nl.
/* Arch */
Winston arch domain
Alan Bundy 5.12.80
use with winston */

/* space of description trees */
space(arch, [shapetree, touchtree, orienttree, directiontree, supporttree]).

/* description tree */
tree(shapetree, 1, shape(Prism(wedge, block), pyramid)).
tree(touchtree, 2, touchrel(separate, touch(marries, abuts)), default(touchtree, separate), % default predicate
tree(orienttree, 1, orientation(lying, standing)).
tree(directiontree, 2, direction(leftof, rightof)).
tree(supporttree, 2, undef(supports, unsupports)).

/* Examples and near misses */
example(arch1, [block(a), block(b), block(c),
standing(a), standing(b), lying(c),
leftof(a, b),
supports(a, c), supports(b, c),
marries(a, c), marries(c, a), marries(b, c), marries(c, b)]).
example(arch2, [block(a), block(b), wedge(c),
standing(a), standing(b), lying(c),
leftof(a, b),
supports(a, c), supports(b, c),
marries(a, c), marries(c, a), marries(b, c), marries(c, b)]).
example(arch3, [block(a), block(b), block(c),
standing(a), standing(b), lying(c),
leftof(a, b),
supports(a, c), supports(b, c),
abuts(a, c), abuts(c, a), abuts(b, c), abuts(c, b)]).
example(archn1, [block(a), block(b), block(c),
standing(a), standing(b), lying(c),
leftof(a, b),
supports(a, c), supports(b, c),
marries(a, c), marries(c, a), marries(b, c), marries(c, b),
marries(a, b), marries(b, a)]).
example(archn2, [block(a), block(b), block(c),
standing(a), standing(b), lying(c),
leftof(a, b),
marries(a, c), marries(c, a), marries(b, c), marries(c, b)]).
example(archn3, [block(a), block(b), block(c),
standing(a), standing(b), lying(c),
leftof(a, b)]).
Sample of Session with Prolog Winston

?- winston(arch).
Please give me an example of a arch
:: arch1.

Please give me an example or near miss of a arch.
:: arch2.

Is this example (yes./no.)?
:: yes.

This is a new sort of arch.
Please give me an example or near miss of a arch.
:: arch3.

Is this example (yes./no.)?
:: yes.

This is a new sort of arch.
Please give me an example or near miss of a arch.
:: archn1.

Is this example (yes./no.)?
:: no.

This limits my idea of arch.
Please give me an example or near miss of a arch.
:: archn2.

Is this example (yes./no.)?
:: no.

This limits my idea of arch.
Please give me an example or near miss of a arch.
:: archn3.

Is this example (yes./no.)?
:: no.

I have seen one of these before.
Please give me an example or near miss of a arch.

?- grey(arch).
Grey areas in arch are:
define([Plato1],[shape/tree,[],[1,2]])
define([Plato2],[shape/tree,[],[1]])
define([Plato3],[shape/tree,[],[1,2]])
define([Plato1,Plato2],[touch/tree,[],[2]])
define([Plato2,Plato1],[touch/tree,[],[2]])
define([Plato3,Plato1],[touch/tree,[],[1]])
define([Plato2,Plato3],[touch/tree,[],[2]])
define([Plato3,Plato2],[touch/tree,[],[2]])
define([Plato1,orient/tree,[],[2]])
define([Plato2,orient/tree,[],[1]])
define([Plato3,orient/tree,[],[2]])
define([Plato1,Plato3],[direction/tree,[],[1]])
define([Plato3, Plato2], supporttree, [], [1])
/*arch1
Winston arch domain
Alan Bundy 5.12.80
use with winston
version with functions */

/* description trees */
tree(shapeTree, 1, shape(prism(wedge, block), pyramid)),
tree(touchtree, 2, touchrel(separate, touch(marries, abuts))),
default(touchtree, separate), % default predicate
tree(orientTree, 1, orientation(lying, standing)),
tree(directionTree, 2, direction(leftof, rightof)),
tree(supportTree, 2, undef(supports, unsupports)),

/* Examples and near misses */
example(arch1, [block(lP(a)), block(rP(a)), block(tm(a)),
standing(lP(a)), standing(rP(a)), lying(tm(a)),
leftof(lP(a), rP(a)),
supports(lP(a), tm(a)), supports(rP(a), tm(a)),
marries(lP(a), tm(a)), marries(rP(a), tm(a))]),
example(arch2, [block(lP(a)), block(rP(a)), wedge(tm(a)),
standing(lP(a)), standing(rP(a)), lying(tm(a)),
leftof(lP(a), rP(a)),
supports(lP(a), tm(a)), supports(rP(a), tm(a)),
marries(lP(a), tm(a)), marries(rP(a), tm(a))]),
example(arch3, [block(lP(a)), block(rP(a)), block(tm(a)),
standing(lP(a)), standing(rP(a)), lying(tm(a)),
leftof(lP(a), rP(a)),
supports(lP(a), tm(a)), supports(rP(a), tm(a)),
abuts(lP(a), tm(a)), abuts(rP(a), tm(a))]),
example(archn1, [block(lP(a)), block(rP(a)), block(tm(a)),
standing(lP(a)), standing(rP(a)), lying(tm(a)),
leftof(lP(a), rP(a)),
supports(lP(a), tm(a)), supports(rP(a), tm(a)),
marries(lP(a), tm(a)), marries(rP(a), tm(a))]),
example(archn2, [block(lP(a)), block(rP(a)), block(tm(a)),
standing(lP(a)), standing(rP(a)), lying(tm(a)),
leftof(lP(a), rP(a)),
marries(lP(a), tm(a)), marries(rP(a), tm(a))]),
example(archn3, [block(lP(a)), block(rP(a)), block(tm(a)),
standing(lP(a)), standing(rP(a)), lying(tm(a)),
leftof(lP(a), rP(a))]).
Definition of Isolation space and examples for Winston Program
Alan Bundy 18.2.81 */

/* Predicate Trees */

tree(occurr_tree, 2, occur_rel(freeof, contains(simpleocc, multocc))),
default(occurr_tree, freeof).


tree(simple_tree, 2, simple_rel(different(unrelated, inverse), ident)),
default(simple_tree, unrelated).

/* Examples and near Misses */

eexample(isol1, [simpleocc(x, expr_at([1,2], before)),
  freeof(x, expr_at([1,1], before)),
  freeof(x, expr_at([2,2], before)),
  ident(expr_at([1,1], before), expr_at([2,1], after)),
  ident(expr_at([1,2], before), expr_at([1,1], after)),
  ident(expr_at([2,2], before), expr_at([1,1], after)),
  inverse(sym_at([1], before), sym([2], after)) ] ),

eexample(isol2, [simpleocc(x, expr_at([1,2], before)),
  contains(x, expr_at([1,1], before)),
  freeof(x, expr_at([2,2], before)),
  ident(expr_at([1,1], before), expr_at([2,1], after)),
  ident(expr_at([1,2], before), expr_at([1,1], after)),
  ident(expr_at([2,2], before), expr_at([1,1], after)),
  inverse(sym_at([1], before), sym([2], after)) ] ),

eexample(isol3, [simpleocc(x, expr_at([1,2], before)),
  freeof(x, expr_at([1,1], before)),
  contains(x, expr_at([2,2], before)),
  ident(expr_at([1,1], before), expr_at([2,1], after)),
  ident(expr_at([1,2], before), expr_at([1,1], after)),
  ident(expr_at([2,2], before), expr_at([2,1], after)),
  inverse(sym_at([1], before), sym([2], after)) ] ),

eexample(isol4, [simpleocc(x, expr_at([1,2], before)),
  freeof(x, expr_at([1,1], before)),
  freeof(x, expr_at([2,2], before)),
  different(expr_at([1,1], before), expr_at([2,1], after)),
  ident(expr_at([1,2], before), expr_at([2,1], after)),
  ident(expr_at([2,2], before), expr_at([1,1], after)),
  inverse(sym_at([1], before), sym([2], after)) ] ),

eexample(isol5, [simpleocc(x, expr_at([1,2], before)),
  freeof(x, expr_at([1,1], before)),
  freeof(x, expr_at([2,2], before)),
  different(expr_at([1,2], before), expr_at([2,1], after)),
  ident(expr_at([1,2], before), expr_at([1,1], after)),
  ident(expr_at([2,2], before), expr_at([2,1], after)),
  inverse(sym_at([1], before), sym([2], after)) ] ),

eexample(isol6, [simpleocc(x, expr_at([1,2], before)),
  freeof(x, expr_at([1,1], before)),
  freeof(x, expr_at([2,2], before)),
  different(expr_at([1,2], before), expr_at([2,1], after)),
  ident(expr_at([1,2], before), expr_at([1,1], after)),
  ident(expr_at([2,2], before), expr_at([2,1], after)),
  inverse(sym_at([1], before), sym([2], after)) ] )
ident(expr_at([1,1],before),expr_at([2,1],after)),
ident(expr_at([1,2],before),expr_at([1],after)),
different(expr_at([2],before),expr_at([2,2],after)),
inverse(sym_at([1],before),sym([2],after)) ] ),

example(isol7, [singleocc(x,expr_at([1,2],before)),
freeof(x,expr_at([1,1],before)),
freeof(x,expr_at([2],before)),
ident(expr_at([1,1],before),expr_at([2,1],after)),
ident(expr_at([1,2],before),expr_at([1],after)),
ident(expr_at([2],before),expr_at([2,2],after)),
unrelated(sym_at([1],before),sym([2],after)) ] ),

example(isol8, [multocc(x,expr_at([1,2],before)),
freeof(x,expr_at([1,1],before)),
freeof(x,expr_at([2],before)),
ident(expr_at([1,1],before),expr_at([2,1],after)),
ident(expr_at([1,2],before),expr_at([1],after)),
ident(expr_at([2],before),expr_at([2,2],after)),
inverse(sym_at([1],before),sym([2],after)) ] ),
Utilities package
Prolog-10 version 3

?- [winston1,isol].

winston1 consulted  2230 words  0.90 sec.

isol consulted  1610 words  0.45 sec.

yes

?- winston(isol).
Please give me an example of a isol:
:: isol1.
Please give me an example or near miss of a isol:
:: isol2.
Is this example (yes./no.)?
:: no.

This limits my idea of isol.
Please give me an example or near miss of a isol:
:: isol3.
Is this example (yes./no.)?
:: no.

This limits my idea of isol.
Please give me an example or near miss of a isol:
:: isol4.
Is this example (yes./no.)?
:: no.

This limits my idea of isol.
Please give me an example or near miss of a isol:
:: isol5.
Is this example (yes./no.)?
:: no.

This limits my idea of isol.
Please give me an example or near miss of a isol:
:: isol6.
Is this example (yes./no.)?
:: no.

[ Execution aborted. ]

! ?- grey(isol).

Grey areas in isol are:
define([Plato1,expr_at([1,2],Plato2)],occurtree,[],[2,1])
define([expr_at([2],Plato2),expr_at([2,2],Plato3)],simtree,[],[2])
define([sym_at([1],Plato2),sym([2],Plato3)],simtree,[],[1,2])
Is this example (yes./no.)?

:: no.

This limits my idea of isol.
Please give me an example or near miss of a isol.

:: isol7.

Is this example (yes./no.)?

:: no.

This limits my idea of isol.
Please give me an example or near miss of a isol.

:: isol8.

Is this example (yes./no.)?

:: no.

This limits my idea of isol.
I have learnt the concept of isol now.

yes
! ?- grey(isol).
Grey areas in isol are:

yes
! ?- core  51712  (23552  lo-seg + 28160  hi-seg)
heap  18432 = 14955 in use +  3477 free
global  1218 =  16 in use +  1202 free
local  1024 =  16 in use +  1008 free
trail  511 =  0 in use +  511 free
22.49 sec. for 25 GCs saving 46683 words
1.98 sec. for 12 local shifts and 110 trail shifts
38.49 sec. runtime
/*block
Winston block domain - simple test example
Alan Bundy 6.12.80
use with winston */

/* space of description tree(s) */
space(block,[shapetree]).

/* description tree(s) */
tree(shapetree,1,shape(Prism(wedse,block),Pyramid)).

/* Example and near miss */
example(block1,[block(a)]).
example(wedse1,[wedse(b)]).
It is difficult for the untrained fisher to follow examples of this complexity so here is a simple concept: two wedges.

/* space of description trees */
space(Pair, % each concept must have a space; is this **right**?
[shaPetree,touchtree,orienttree]).

/* description tree */
tree(shaPetree, 1, shape(wedge, block)).
tree(touchtree, 2, touchrel(separate, touch)).
tree(orienttree, 1, orientation(lying, standing)).

% Examples
example(p1, [wedge(a1), wedge(b1),
standing(a1), lying(b1), separate(b1, a1)
]).
example(p2, [wedge(a2), wedge(b2),
standing(a2), standing(b2), touch(a2, b2)
]).
example(p3, [wedge(a3), wedge(b3),
lying(a3), lying(b3)
]).

% Near misses
example(n1, [block(a4), block(b4),
standing(a4), lying(b4), separate(b4, a4)
]),  % two similar things, but not wedges
example(n2, [wedge(a5), wedge(b5), wedge(c5),
standing(a5), standing(b5), touch(a5, c5)
]),  % one wedge too many
example(n3, [wedge(a6),
standing(a6)
]),  % one wedge too few
1. Description

This is a rational reconstruction of Winston's program, [Winston 75], for learning new concepts, e.g. the arch. It takes descriptions of specimens, which can be either examples of arches or near misses to arches, and uses them to refine its definition of an arch. The rational reconstructing was done by Plotkin, Young and Linz, as reported (all too briefly) in [Young et al 77]. Their essential advance over Winston was to keep two defining descriptions around: an upper and lower bound; and use incoming evidence to try to move these descriptions closer together.

2. Method of Use

To use the program, type

```
run PLL: UTIL
```

and in response to the prompt type

```
consult('PLL: WINST').
```

The top level predicate, winston, will then be available. winston takes one argument: the name of the concept to be learnt, e.g. arch. If you call

```
```

In this program specification we will use 'arch' as the running example. Readers may safely generalize 'arch' to 'concept', wherever it appears, except when indicated to the contrary.
winston(arch)

then the program will prompt you for the name of a particular arch. It will use this to initialize its lower bound arch description: the upper one being initialized to the contentless description. The program will prompt you for the name of either an arch or a near miss. It will then ask you whether this is an example, to which you must reply either 'yes' or 'no'. All replies to prompts must be terminated with full stop, carriage return.

The program will continue to prompt you with requests for examples or near misses, until its upper and lower bound descriptions coincide at which point it will announce that it has learnt the concept and will exit winston.

If the evidence you provide is already known to the program then it will say so. If it thinks you have provided contradictory evidence then it will say so, try to back up to remake some choice, and then collapse in a heap.

To make the program work you must have compiled the following information:

- A definition of the description space of the concept you want learnt.
- Descriptions of each of the examples and near misses to be input to the program.

The information required for the concept 'arch' can be found in file PLL: ARCH:PRB. It can be input by typing

consult('PLL! ARCH:PRB').

The description of a specimen is given by the predicate, specimen. This predicate takes two arguments: the name of the specimen; and a list of defining propositions, e.g.

specimen(arch1, [block(a), block(b), block(c),
standing(a), standing(b), lying(c),
leftof(a,b),
supports(a,c), supports(b,c),
maries(a,c), marries(c,a), marries(b,c), marries(c,b)]).

The predicates used in these descriptions must be arranged into trees of related predicates, and these trees described with the predicate, tree. tree takes three arguments: the name of the tree; the common arity of all the predicates in it; and the tree itself, represented as a nested term of predicates, e.g.

tree(touchtree,2,touchrel(separate,touch(maries,abuts))).
This tree is diagrammed in figure 3-1. touchrel is a contentless predicate. Two objects may either touch or be separate. Two touching objects may either marry or abut. One of these predicates can be marked as a default with the binary predicate, default, e.g.

\[ \text{default(touchtree, separate)}. \]

Finally a list of those predicate trees, which can be used in defining the concept, must be given. This is done with the binary predicate, space, which takes the name of the concept and the list of allowed trees, e.g.

\[ \text{space(arch, [shapetree, touchtree, orienttree, directiontree, supporttree])}. \]

3. How it Works

We first describe the data-structures used by the program and then give an overview of the program.

3.1. Data-Structures

The program's description of a concept consists of a set of predicate trees, with two pointers into each tree: one representing an upper bound, and one a lower bound. Each relation, in the incoming specimen description, is translated into a predicate tree, with a pointer to the relation's predicate (see figure 3-1).

- If every specimen pointer is below the lower bound then the specimen is known to be an example.

- If one specimen pointer is above the upper bound then the specimen is known not to be an example, but to be a near-miss.

- Otherwise, one specimen pointer must lie in the grey area between the upper and lower bound and the program does not already know the status of the specimen. On being told the status, it can modify its definition, by either raising its lower bound to include the specimen pointer (e.g. to predicate 'touch' in figure 3-1), or lowering its upper bound to exclude the specimen pointer (e.g. to predicate 'marries' in figure 3-1).

Note that there will be different predicate trees for different combinations of arguments to the same predicate, e.g. for \( \text{abuts}(a,c) \), \( \text{abuts}(a,b) \) and \( \text{abuts}(c,a) \), say.
A set of predicate trees is represented by a list of terms: each term representing a predicate tree together with pointers. For instance, in the case of the definition of a concept, a typical term might be

```
define([[Plato1, Plato3], touchtree, []], [2,1])
```

**define** is a four argument function: the second argument is the name of the tree; the first gives the combination of arguments received by the predicates of this tree; and the third and fourth give the positions of the upper and lower bounds, respectively. The constants used as arguments in a concept definition are always called **Platon**, because they are ideal objects. Positions in the tree are given by lists of positive integers which specify which arcs to follow to traverse the tree from the root to the predicate being pointed to. The empty list specifies the root. The example above corresponds to the situation in figure 3-1.

```
touchrel (upper bound position)
  1  
  2
separate
  1
  2
  touches
  1
marries (lower bound position)
  2
abuts (specimen position)
```

**Figure 3-1:** A Predicate Tree with Pointers

The description of a specimen is recorded in a similar fashion. The term in this case is constructed from a ternary function, **record**, e.g.,

```
record([[a, c], touchtree, [2,2])
```

where a and c are constants mentioned in the original specimen description.

Once a match has been established between the (Platonic) constants of the definition and the particular constants of the specimen, e.g., \{a/Plato1, b/Plato3, c/Plato3\}, then a difference description is built up. This is also a list of terms, but constructed from the six argument function, **differ**, e.g.,

```
differ([[Plato1, Plato3], touchtree, [], [2,2], [2,1], srey])
```
This contains, not only, the information from the record and define terms, but also the status of the specimen, e.g., grey.

3.2. Program Overview

The program is divided into four parts.

1. Top level input/output procedures. The very top level procedure is winston, described above, but the heart of the program is the procedure learn, which links together the remaining three parts of the program.

   learn(Concept, Specimen, YesNo) :- !,
   definition(Concept,CObjs,CDefn),
   make_rec(Concept,Specimen,EObjs,ERec),
   classify(CObjs,EObjs,CDefn,ERec,Diff,Verdict),
   learn1(Concept,Diff,YesNo,Verdict).

learn takes three input arguments: the concept to be learnt; the current specimen; and its status according to the user. learn modifies the program’s concept definition appropriately.

2. Procedures to translate the original input descriptions into the internal representation as a set of predicate trees with pointers. The top level procedure of this part is make_rec, which takes the concept and the specimen and returns the internal description of the latter as a list of constants and a list of predicate tree records.

3. Procedures to match the constants in the specimen description against those in the stored definition and to classify the resulting description as example, near miss or grey. The top level procedure of this part is classify, which takes the constants and predicate trees from both the concept definition and the specimen, and forms, first a difference description and then a classification of the specimen’s status.

4. Procedures to act appropriately to this classification, in particular, to adjust the stored definition of the concept when the incoming specimen is classified as ‘grey’. The top level procedure of this part is learn1, which takes the concept, difference description and the status of the specimen according to both the user and the program.

The best match between the incoming specimen and the definition is found in a crude heuristic way. The heuristic is that even non-examples (near misses) will be almost examples. All matches of objects are tried. For each assignment all corresponding predicate trees are compared. A score for the assignment is
totted up; each pair of predicate trees contributing either 0, 2 or 1, according to whether the specimen pointer appears below the lower bound, above the upper bound or in between. The assignment with the lowest score wins.

It can happen that the concept definition contains a predicate tree which does not correspond to any tree in the specimen description, or vice versa. In such cases the program assumes that the missing tree is provided with a pointer to the default predicate in the case of a missing lower bound or specimen position, and a pointer to the tree root in the case of a missing upper bound.

4. Program Requirements

The Prolog system requires 30K words. PLL: UTIL, PLL: WINST, PLL: ARCH, PRB and working space require a further 26K words. The following predicates are used by the program:

- add_ups(2)
- checklist(2)
- consts_in(2)
- definition(3)
- findall(3)
- srev(1)
- lowest(4)
- make_subst(3)
- one_of(3)
- same(1)
- some(2)
- sumlist(2)
- verdict(2)
- append(3)
- classify(6)
- convert(2)
- diff_to_defn(2)
- flatten(2)
- srev1(1)
- lub(2)
- maPlist(3)
- pair_off(3)
- score(2)
- score1(2)
- tree(3)
- winston(1)
- apply(2)
- common(3)
- default(2)
- exclude(2)
- sensym(2)
- learn(3)
- make_diff(5)
- nth_el(3)
- new_defn(2)
- position(3)
- subst(3)
- winston1(1)
- best(2)
- compare(4)
- default_posn(2)
- extra_rec(2)
- make_rec(4)
- nth_el(3)
- position(3)
- select(3)
- verdict(2)
- writef(2)

REFERENCES

[Winston 75]

Winston, P.
Learning structural descriptions from examples.

[Youngs et al 77]

Youngs, R.M.; Plotkin, G.D. and Linz, R.F.
Analysis of an extended concept-learning task.
LANGUAGE ACQUISITION USING THE WINSTON... PROGRAM

In this note I want to explore the application of the Winston... program to the language acquisition problem studied by Pat Langley with his AMBER program. In particular, I want to address the question as to whether the Winston... technique can be used to obtain the discrimination process used by AMBER.

1. The Description Space

Using the Winston... technique, means building, as a concept definition, the condition part of a production rule like,

action & asing & present & process -> is

or

agent & aplural -> s

where the first rule means

Use 'is' as a prefix when describing an action and when the agent is singular and the action is in the present and is a process.

and the second rule means

Use 's' as a suffix, when describing an agent and when the agent is plural.

We will use a description space consisting of 7 relation trees. All the relations will be nullary, although in a pukka version of the program a nullary relation like 'asing' might be translated into agent(X) & singular(X), etc. The trees are all shallow, one level deep, with two or three arcs. They are represented in figure 1-1.

2. Specimen Induction

The first modification required to Winston... is to get it to induce its own specimens* by the following technique.

1. The current state of the concept definition will be used to generate the condition of the rule.

2. This rule will be used to generate the child's utterance.

3. The context of the utterance will be used to get the specimen description.

4. The adult utterance will be used to classify the specimen as an example or near or far miss.

We will assume a mechanism, like AMBER's, to do step 2 and concentrate on steps 1. and 4. (step 3 is trivial).

*Where a specimen is either an example, a near miss or a far miss.
Consider step 1: the use of the current concept definition to generate the rule condition. Each relation tree in the definition will give rise to a single condition. Which node of the tree should be used? In his comparison of Winston and AMBER, Pat assumed that the lower bound would be used in each case, but this is not the only possibility. One could use any node between the upper and lower bound. We will consider two cases: using all the lower bounds and using all the upper bounds. We will see that using all upper bounds leads to a discrimination type process, very like AMBER's.

3. Conditions Formed From Upper Bounds
Consider this case first: that the upper bound of each tree contributes a condition. The initial position of the upper bound in each tree is the root node, marked 'top'. The lower bounds need to be initialized by an example. Let us suppose the child hears an adult use 'is' in the context

- action & asing & adef & present & process & oplural & odef

these relations will then be adopted as the initial lower bounds. The condition of the rule generated by the current definition consists of 7 'tops'. Since 'top' is a contentless relation we will omit occurrences of it, so that the current rule is

-> is

The child will now apply the rule in some situation, listen to the adult utterance in this situation and, hence, classify the rule application as being
correct, an error of commission or an error of omission.* The regular cases are summarised in figure 3-1.

Correct - Do Nothing  
-------------------------- Upper Boundary  
Commission - Miss  
-------------------------- Target Point  
Correct - Example  
-------------------------- Lower Boundary  
Correct - Example (but no action)

Figure 3-1: Cases that Arise when Upper Bounds are Used

Suppose that an error of commission is detected and that the context is

agent & asing & adef & present & process & oplural & odef

This context becomes the description of a near-miss specimen. This is a near-miss, rather than a far-miss, because there is only one grey relation, i.e. only one relation, 'agent', lies in the grey area of the current concept definition. The concept definition can now be updated by moving the upper bound of the first relation tree down from 'top' to 'action'. The rule generated by the upper bounds of the current concept description will contain a non-trivial condition, namely 'action', i.e. it is

action \rightarrow is

Suppose that the error of commission was created in a far-miss context, that is, there is more than one grey relation, e.g.

agent & asing & aindef & present & process & oplural & odef

where both 'agent' and 'aindef' are grey relations. The present version of the Winston... program would pick one of them at random and use this as the discrimination relation, i.e. move the upper bound of its relation tree down. Here is an important difference from AMBER. If AMBER has not yet got a major relation (agent, action or object) in its rule condition then it tries to pick a major, grey relation as discrimination relation. In this case, then, AMBER would choose 'agent'. If the condition already includes a major relation or if the context contains no major grey relation, then AMBER forms a new rule for each grey relation. It might be possible to modify the Winston... program to do things the AMBER way, although it means keeping a whole set of concept definitions around.

If Winston... makes a bad choice when picking the discrimination relation in a far-miss then the resulting concept definition will generate faulty rules: rules that can cause errors of omission as well as errors of commission. We will see that an error of omission indicates that a bad choice has been made

*Actually, omission errors cannot happen this time, but they can if the upper bounds get screwed up.
and that the program should back-up and remake the choice. So let us suppose that in the example above that the program chooses the grey relation 'aindef' and moves the upper bound of the third tree down to 'adef'. This concept definition will generate the rule

\[ \text{adef} \rightarrow \text{is} \]

Suppose that this rule is used by the child and causes an error of omission, in the context

\[ \text{action & asing & aindef & present & process & oplural & odef} \]

This context constitutes an example description, but one which lies in the non-example part of the current concept definition. The Winston... program will back-up, remake the choice of discrimination relation (rejecting 'aindef' and picking 'agent') and then treat the new context as an example, moving the lower bound of the third tree up from 'adef' to 'top'. This constitutes a difference from AMBER: an erroneous rule has been erased, whereas AMBER keeps such rules around, but with a low strength.

The final case to consider is when the rule has been used correctly. There are two subcases: when the rule has correctly not fired and when it correctly has fired. We only do anything in the second case. For instance, if the rule has correctly fired and the context is

\[ \text{action & asing & aindef & present & process & oising & oindef} \]

This context constitutes an example description. Each grey relation, oising and oindef, will cause the lower bound in the corresponding tree to be lifted (to 'top' in each case). Note that, if this example had come earlier it would have merged the upper and lower bounds of the third tree and prevented the erroneous choice of 'aindef' as the discrimination relation. This is an improvement over AMBER.

4. Conditions Formed From Lower Bounds

Now we consider what happens if the lower bounds are used to generate rule conditions. This is the process implied by Pat in his contrast of the Winston generalization technique and his discrimination technique. We must consider two cases; according to whether the rule is used correctly or causes an error of omission. These cases are summarised in figure 4-1.

Suppose the initial example is as before then the rule generated from the lower bounds is

\[ \text{adef} \rightarrow \text{is} \]

*In terms of figure 3-1 the target point has been moved up above the upper boundary.

*At least, it should, but this part of the program has never been tested.

**Errors of commission cannot happen, since conditions generated from the lower bounds are always conservative and, unlike the upper bounds, can never be erroneous.
Correct - Miss (but no action)
-------------------------- Upper Boundary
Correct - Miss
-------------------------- Target Point
Omission - Example
-------------------------- Lower Boundary
Correct - Do Nothing

Figure 4-1: Cases that Arise when Lower Bounds are Used

We consider first the case when the rule is applied correctly. If the rule fired correctly then there is nothing further to do. However, if the rule was not fired, and should not have been, then the new context may still serve as a miss. Suppose the new context is

agent & asing & aindef & present & process & oplural & odef

This contains two grey relations, 'agent' and 'aindef', one of which must be chosen as the discrimination relation. Let us suppose that 'aindef' is incorrectly chosen and the upper bound of the third tree is moved down to merge with the lower bound at 'adef'. The rule condition will be unchanged, but behind the scenes the third tree has been firmed up.* The context of an error of omission serves as an example and if the upper bounds are erroneous (as above) then it may first cause back-up. We will consider only the first case (because it includes the second): i.e. the rule has not fired, but should have, and the context is

action & asing & aindef & present & process & oplural & odef

The program will back-up; remake the choice of discrimination relation (from 'aindef' to 'agent'); restore the third tree; bring down the upper bound of the first tree to merge with the lower bound at 'action'; and then use the above context as an example to raise the lower bound of the third tree to 'top'.

5. Summary

In the last two sections we have considered 6 cases: that conditions are formed solely from the upper bounds or lower bounds and that in each case the rule use is correct or leads to an error of either commission or omission. We summarise the results in table 5-1.

<table>
<thead>
<tr>
<th>Correct Bound</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do Nothing</td>
<td>Miss</td>
</tr>
<tr>
<td>Example</td>
<td>(Near or Far)</td>
</tr>
<tr>
<td>Do Nothing</td>
<td>Cannot</td>
</tr>
<tr>
<td>or Miss</td>
<td>Happen</td>
</tr>
</tbody>
</table>

Table 5-1: Summary of Action on the Six Cases

*Erroneously, of course - the target point of figure 4-1 has been moved above the upper boundary.
If rule conditions are generated from the lower bounds of the current concept then the Winston... program behaves much as Pat Langley predicted it would: starting with long, underestimating conditions and settling the rules with long conditions quicker than those with short conditions. If rule conditions are generated from the upper bounds of the current concept, however, then the Winston... program behaves much as Pat's own AMBER program: starting with short, overestimating conditions and settling the rules with short conditions quicker than those with long conditions. Thus the rationally reconstructed Winston... program unifies the old Winston generalization technique and Langley's discrimination technique. Presumably, generating rule conditions from a hybrid of upper and lower bounds would produce a hybrid generalization/discrimination technique, and this possibility might be worth exploring.

Note that a rule's conditions may not change during several episodes, even though the underlying concept definition is changing. Thus, if the conditions are being generated from the upper bounds and some examples come in causing changes to the lower bounds then the conditions will not (cannot) reflect this change. Hence, if the upper bounds are being used and the condition is short, then the condition may become settled early on, subsequent episodes appearing not to change anything, however, these episodes will firm up the conditions by moving the lower bounds up to merge with the upper bounds. If all these episodes are counted in the story of rule formation then rules with short conditions and those with long conditions will take similar lengths of time, despite appearances to the contrary. Dual remarks hold when lower bounds are used to generate rule conditions.
tree (range: range (orange, inrange (attacks, defends)))
deep (allow, [allow ([black, white])])
deep (side, deep (same, opposite))

space (fork, [range, side])

example (fork, [attack (wr, bb), attack (wr, bk), opposite (wr, bb), opposite (wr, bk), opposite (bb, wr), opposite (bk, wr), same (wr, bk), same (bk, bb)],)

white (wr), black (bb), black (bb) ]

(far miss)

error! error! [attack (wr, bb), defend (wr, bk),

black (bb), black (bb), black (bb) ]
/* Top Level Program - learn new concept */

/* First time only accept an example */

winston(Concept) :- !,
    writef('Please give me an example of a %t \n', [Concept]),
    read(Ex), nl,
    make_rec(Concept, Ex, EObjjs, ERec),
    maPlist(sensym1(plato), EObjjs, CObjjs),
    make_subst(EObjjs, CObjjs, Subst),
    subst(Subst, ERec, CRec),
    maPlist(add_ups, CRec, CDefn),
    assert(definition(Concept, CObjjs, CDefn)),
    winston1(Concept).

/* Is grey area in definition eliminated? */

winston1(Concept) :- !,
    definition(Concept, CObjjs, CDefn),
    checklist(same, CDefn), !,
    writef('I have learnt the concept of %t now, \n', [Concept]).

/* Subsequently accept either examples or near misses */

winston1(Concept) :- !,
    writef('Please give me an example or near miss of a %t, \n', [Concept]),
    read(Ex), nl,
    writef('Is this example (yes/no)? \n', [Ex]),
    read(YesNo), nl,
    learn(Concept, Ex, YesNo),
    winston1(Concept).

/* Add default upper bounds in concept record */

add_ups(record(Argss, Name, Posn), define(Argss, Name, [], Posn)).

/* Find position of default predicate on tree */

default_posn(TreeName, Posn) :-
    default(TreeName, Pred), !,
    tree(TreeName, _, Tree), position(Pred, Tree, Posn).

default_posn(TreeName, []).

/* Slight modify to sensym, so it can be used in maPlist */
sensym1(Prefix, _, NewConst) :- !, sensym(Prefix, NewConst).

/* Are upper and lower bound of concept definition the same? */
same(define(Argss, Name, Posn, Posn)).

/* Learn from this example or near miss */

learn(Concept, Example, YesNo) :- !,
    definition(Concept, CObjjs, CDefn),
    make_rec(Concept, Example, EObjjs, ERec),
    classify(CObjjs, EObjjs, CDefn, ERec, Diff, Verdict),
    learn1(Concept, Diff, YesNo, Verdict),
/* Make records from list of relations */
/* -------------------------------------------- */

make_rec(ConcePt, ExamPle, EObJs, ERec) :- !,
space(ConcePt, TreeList), example(ExamPle, Relns),
obj_in(Relns, EObjs),
form_records(TreeList, Relns, EObjs, ERec).

/* Find all objects mentioned in relations */
obj_in([], []).
obj_in([Reln | Relns], ObJs) :- !,
Reln =.. [Pred | Args],
obj_in(Relns, ObJs1), union(Args, ObJs1, ObJs).

/* Change each relation into a record */
form_records([], [], EObjs, []).

form_records([TreeName | TreeList], Relns1, ObJs, Recs) :- !,
tree(TreeName, Arity, Tree),
findall(Perm, Perm(ObJs, Arity, Perm), Perms),
form_record(TreeName, Relns1, Perms, Recs1, Relns2),
form_records(TreeList, Relns2, ObJs, Recs2),
append(Recs1, Recs2, Recs).

/* Change the relations relevant to a tree into definitions */
form_record(Name, Relns1, Perms, Rec, Relns2),
form_record(Name, Relns1, Perms, Rec, Relns2),
default_posn(Name, Default).

/* find perm of n elements of list */
perm([], 0, []).
perm([Hd | Tl], N, Perm) :- N > 0,
PN is N - 1,
perm(Tl, PN, Perm1),
append(Front, Back, Perm1),
append(Front, [Hd | Back], Perm).

default_posn(Name, Default).

/* Find Position of Node in Tree */
position(Node, Tree, []) :-
Tree =.. [Node | SubTrees],
position(Node, Tree, [N | Posn]).
/* Find nth element of list */

nth_el(1, [Hd|Tl], Hd),

nth_el(1, [Hd|Tl], El) :-
  nth_el(PN, Tl, El), PN is PN + 1.

/* Is this example, non-example or in grey area, by my definition? */

classify(CObjs, EObjs, CDefn, ERec, BestDiff, Verdict) :- !,
  findall(Diff, make_diff(CObjs, EObjs, CDefn, ERec, Diff), Diffs),
  best(Diffs, BestDiff),
  verdict(BestDiff, Verdict).

/* Find the difference between example and concept */

make_diff(CObjs, EObjs, CDefn, ERec, Diff) :- !,
  Perm(EObjs, EObjs1), make_subst(EObjs1, CObjs, Subst),
  subst(Subst, ERec, ERec1),
  Pair_off(CDefn, ERec1, Diff).

/* Pair off concept definition and example record to make differences */

Pair_off(\[define([Arss,Name,UPPosn,LowPosn]\) : CDefn],
  ERec, \[differ([Arss,Name,UPPosn,ExPosn,LowPosn,Verdict]\) : Diff]) :- !,
  select(record([Arss,Name,ExPosn], ERec, Rest), !,
    compare(UPPosn, ExPosn, LowPosn, Verdict),
    Pair_off(CDefn, Rest, Diff).

Pair_off(CDefn, ERec, Diff) :-
  writef(‘Error: defn %1 and erec %1 do not match \n’, CDefn, ERec),
  abort.

/* Compare positions in tree to give verdict */

compare(U, E, L, yes) :- append(L, _, E), !,
  compare(U, E, L, grey) :- append(U, _, E), !,
  compare(U, E, L, no) :- !.

/* Find best difference and return it */

best(Diffs, Diff) :- !,
  maplist(score, Diffs, Scores),
  lowest(Diffs, Scores, Diff, Score).

/* Return difference with lowest score */

lowest([[Diff1|Diffs]], [[Score1|Scores]], Diff2, Score2) :-
  lowest(Diffs, Scores, Diff2, Score2), Score2 < Score1, !.

lowest([[Diff|Diffs]], [[Score|Scores]], Diff, Score) :- !.

/* Find score of difference */

score(Diff, Score) :- !,
  maplist(score1, Diff, Scores),
  sumlist(Scores, Score).

/* Find score of individual differ */

score1(differ(_, _, _, _, yes), 0) :- !,
  score1(differ(_, _, _, _, grey), 1) :- !.
score1(differ(_, _, _, _, _, no), 2) :- !.

/** add up all the numbers in a list **/
sumlist([], 0),
sumlist([N|Rest], Total) :- !,
    sumlist(Rest, SubT), Total is SubT + N.

/* Make a substitution for replacing all members of one list
by corresponding members of another list */
make_subst([], [], true).
make_subst([X|XRest], [Y|YRest], X=Y & Subst) :-
    make_subst(XRest, YRest, Subst).

/** Decide whether example falls inside definition on basis of differs */
verdict(Diff, yes) :- checklist(verdict1(yes), Diff), !.
verdict(Diff, no) :- some(verdict1(no), Diff), !.
verdict(Diff, srew) :- some(verdict1(srew), Diff), !.
    /* verdict on individual differ */
verdict1(V, differ(_, _, _, _, _, V)).

/* adjust definition appropriately */
/* -------------------------------- */
/* if new example found */
learn1(Concept, Diff, yes, srew) :- !,
    writef('This is a new sort of %t. \n', [Concept]),
    maplist(lub, Diff, New),
    retract(definition(Concept, CObs, Old)),
    assert(definition(Concept, CObs, New)).

/* if near miss found */
learn1(Concept, Diff, no, srew) :- !,
    writef('This limits my idea of %t. \n', [Concept]),
    one_of(exclude, Diff, Diff1),
    maplist(diff_to_defn, Diff1, New),
    retract(definition(Concept, CObs, Old)),
    assert(definition(Concept, CObs, New)).

/* if nothing new is discovered */
learn1(Concept, Diff, Agree, Agree) :- !,
    writef('I have seen one of these before. \n', []).

/* or if contradiction is discovered */
learn1(Concept, Diff, Agree, Disagree) :- !,
    writef('Uh Oh, somethings gone wrong. I will think again.\n', [], fail.

/* Move lower definition up a bit to include new example */
lub(differ(Ars, Name, UPosn, ExPosn, Old, srew),
    define(Ars, Name, UPosn, New)) :- !,
    common(ExPosn, Old, New).

/* Lower definition already includes new example */
lub(differ(Ars, Name, UPosn, ExPosn, LowPosn, yes),
    define(Ars, Name, UPosn, LowPosn)) :- !.
/* Move upper definition down a bit to exclude near miss */
exclude(differ(Arss, Name, Old, ExPosn, LowPosn, srey),
        differ(Arss, Name, New, ExPosn, LowPosn, srey)) :- !,
        common(ExPosn, LowPosn, Comm), append(Comm, [N!_], LowPosn),
        append(Comm, [N], New).

/* Take unnecessary bits out of difference */
diff_to_defn(differ(Arss, Name, UPPosn, ExPosn, LowPosn, Verdict),
             define(Arss, Name, UPPosn, LowPosn)).

/* Find common initial sublist of two lists */
common([N!Rest1], [N!Rest2], [N!Rest]) :- !,
        common(Rest1, Rest2, Rest).

common(List1, List2, []) :- !.

/* change just one member of list */
one_of(Prop, [Old1Tl], [New1T1]) :- apply(Prop, [Old, New]),
one_of(Prop, [HdlOld], [HdlNew]) :- one_of(Prop, Old, New).

/* Find out what srey areas still exist in concept */
srey(Concept) :- !,
    writef('Grey areas in %t are:
          ', Concept),
    definition(Concept, CObjs, CDefn),
    checklist(srey1, CDefn),
    srey1(define(Arss, Name, Posn, Posn)) :- !.

srey1(Defn) :- !,
    write(Defn), nl,
<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_overlap/3</td>
<td>unify</td>
<td>combine/3</td>
</tr>
<tr>
<td>combine/3</td>
<td>unify</td>
<td>unify/4</td>
</tr>
<tr>
<td>disagree/4</td>
<td>unify</td>
<td>unify/4 find_one/4</td>
</tr>
<tr>
<td>find_one/4</td>
<td>unify</td>
<td>disagree/4 find_one/4</td>
</tr>
<tr>
<td>free_of/2</td>
<td>unify</td>
<td>make_pair/3</td>
</tr>
<tr>
<td>is_variable/1</td>
<td>unify</td>
<td>make_pair/3</td>
</tr>
<tr>
<td>make_pair/3</td>
<td>unify</td>
<td>unify/4</td>
</tr>
<tr>
<td>mapand/3</td>
<td>utility</td>
<td>combine/3</td>
</tr>
<tr>
<td>memberchk/2</td>
<td>utility</td>
<td>check_overlap/3</td>
</tr>
<tr>
<td>occ/3</td>
<td>utility</td>
<td>free_of/2</td>
</tr>
<tr>
<td>subst/3</td>
<td>utility</td>
<td>unify/4 update/3</td>
</tr>
<tr>
<td>unify/3</td>
<td>unify</td>
<td>unify/3 unify/4</td>
</tr>
<tr>
<td>unify/4</td>
<td>unify</td>
<td>&lt;user&gt; combine/3</td>
</tr>
<tr>
<td>update/3</td>
<td>unify</td>
<td></td>
</tr>
</tbody>
</table>
Unification procedure for first order logic (with occurs check)
See P80 of Artificial Mathematicians.
Alan Bundy 10.7.81

/* Top Level */

unify(Exp1, Exp2, Subst) :- % To unify two expressions
  unify(Exp1, Exp2, true, Subst). % Start with empty substitution

/* Unify with output and input substitutions */

unify(Exp, Exp, Subst, Subst), % If expressions are identical, succeed
unify(Exp1, Exp2, OldSubst, AnsSubst) :- % otherwise
  disagree(Exp1, Exp2, T1, T2), % find first disagreement pair
  make_pair(T1, T2, Pair), % make a substitution out of them, if possible
  combine(Pair, OldSubst, NewSubst), % combine this with input subst
  subst(Pair, Exp1, NewExp1), % apply it to expressions
  subst(Pair, Exp2, NewExp2),
  unify(NewExp1, NewExp2, NewSubst, AnsSubst). % and recurse

/* Find Disagreement Pair */

disagree(Exp1, Exp2, Exp1, Exp2) :-
  Exp1 =.. [Sym1: ...], Exp2 =.. [Sym2: ...], % If expressions have different
  Sym1 \== Sym2, !. % function symbol, then succeed

disagree(Exp1, Exp2, T1, T2) :-
  Exp1 =.. [Sym: Arss1], Exp2 =.. [Sym: Arss2], % otherwise
  find_one(Arss1, Arss2, T1, T2), % find their arguments
  % and recurse

/* Find first disagreement pair in argument list */

find_one([Hd: T1], [Hd: T2], T1, T2) :- !, % If heads are identical then
  find_one(T1, T1, T2), % find disagreement in rest of list
find_one([Hd1: T1], [Hd2: T2], T1, T2) :-
  disagree(Hd1, Hd2, T1, T2), !, % else find it in heads.

/* Try to make substitution out of pair of terms */

make_pair(T1, T2, T1=T2) :- % T1=T2 is a suitable substitution
  is_variable(T1), % if T1 is a variable and
  free_of(T1, T2), % T2 is free of T1
make_pair(T1, T2, T2=T1) :-
  is_variable(T2), % or if T2 is a variable and
  free_of(T2, T1), % T1 is free of T2

/* By convention: x,y,z,u,v and w are the only variables */

is_variable(u), is_variable(v), is_variable(w),
is_variable(x), is_variable(y), is_variable(z),
/* T is free of X */
free_of(X,T) :- occ(X,T,0), % if X occurs 0 times in T

/* Combine new substitution pair with old substitution */
combine(Pair, OldSubst, NewSubst) :-
  mapand(update(Pair), OldSubst, Subst1), % apply new pair to old subst
  check_overlap(Pair, Subst1, NewSubst), % and delete ambigious assigments

/* Apply new pair to old substitution */
update(Pair, Y=S, Y=S1) :- % apply new pair to rhs of old subst
  subst(Pair, S, S1),

/* If X is bound to something already then ignore it */
check_overlap(X=T, Subst, Subst) :- % Ignore X=T
  memberchk(X=S, Subst), !, % if there already is an X=S
  check_overlap(Pair, Subst, Pair & Subst), % otherwise don't

/* MINI-PROJECTS */
1. Simplify unify to a one way matcher, as per p76 of 'Artificial Mathematicians'.
2. Generalize Unify to a simultaneous unifier of a set of expressions, (gen_unify) as per p80 of 'Artificial Mathematicians'.
3. Build the associativity axiom into unify (assoc_unify), as per p82 of 'Artificial Mathematicians'.
*/
Using PROLOG as a theorem prover:
An equality axioms example.
Try goal 'equal(y,x).'.

Alan Bundy 16.6.81 */

equal(x,y). % The Hypothesis
equal(X,X). % The Reflexive Axiom

equal(U,W) :- equal(U,V), equal(W,V). % The Twisted Transitivity Axiom

/* MINI-PROJECTS

1. Try goal ?- equal(y,x).

2. Experiment by switching the order of the above axioms and trying
to same goal. What sort of bad behaviour emerges? How could it be avoided?

3. Experiment with different axioms, e.g., the group theory example
from p92 of 'Artificial Mathematicians'.

*/
/* EQUAL. 

Clauses for the SIMPLE equality example 
Symmetry can be inferred from reflexivity and twisted transitivity

Alan Bundy 22,6,81 */

clause(hypothesis, [equal(x,y)], [], input ).  % Input clauses 
clause(reflexive, [equal(x,x)], [], input ). 
clause(twisted, [equal(U,W)], [equal(U,V), equal(W,V)], input ).
clause(goal, [], [equal(y,x)], topclause).           % Top clause
<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>append/3</td>
<td>utility</td>
<td>resolve/3</td>
</tr>
<tr>
<td>breadth/1</td>
<td>breadt</td>
<td>so/0 breadth/1</td>
</tr>
<tr>
<td>clause/4</td>
<td>equal</td>
<td>find_clause/4</td>
</tr>
<tr>
<td>find_clause/4</td>
<td>breadt</td>
<td>resolve/3 record_clause/3</td>
</tr>
<tr>
<td>gensym/2</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
<tr>
<td>iso/0</td>
<td>breadt</td>
<td></td>
</tr>
<tr>
<td>record_clause/3</td>
<td>breadt</td>
<td>resolve/3</td>
</tr>
<tr>
<td>repeat/1</td>
<td>breadt</td>
<td>breadth/1</td>
</tr>
<tr>
<td>resolve/3</td>
<td>breadt</td>
<td></td>
</tr>
<tr>
<td>select/3</td>
<td>utility</td>
<td>resolve/3</td>
</tr>
<tr>
<td>write/1</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
<tr>
<td>write/2</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
</tbody>
</table>
BREADTH.

Breadth First Search Theorem Prover. 
\texttt{UAL} contains test example.

\texttt{an Bundy 16.6.81 */}

( Top Goal */

; \texttt{breadth(0)}.\)

(* Breadth First Search Strategy */

\texttt{breadth(N)} :-
    N1 is N+1, % Calculate new depth
    repeat(resolve(input,N,N1)), % Form all resolvants at that depth
    repeat(resolve(N,input,N1)), % and recurse
    breadth(N1).

(* Repeat as many times as possible */

repeat(\texttt{Goal}) :- \texttt{Goal}, fail. % Keep trying and failing
repeat(\texttt{Goal}), % and succeed only when you run out of things to do

(* Resolution Step */

\texttt{resolve(N1, N2, N) :-}
    find_clause(Parent1, Consequent1, Antecedent1, N1), % Find clauses at
    find_clause(Parent2, Consequent2, Antecedent2, N2), % appropriate depth
    select(Literal, Consequent1, RestConseq1), % find a common literal
    select(Literal, Antecedent2, RestAnte2), % return leftovers
    append(RestConseq1, Consequent2, Consequent), % cobble leftovers together
    append(Antecedent1, RestAnte2, Antecedent),
    record_clause(Consequent, Antecedent, N). % record new clause

(* Record Existence of New Clause */

record_clause([],[],N) :- % test for empty clause
    writeln('Success! Empty Clause Found\n\n'), !, % tell user
    abort. % and stop

record_clause(Consequent, Antecedent, N) :- % test for loop
    find_clause(Name, Consequent, Antecedent, M), !, % i.e., clause with same inr

record_clause(Consequent, Antecedent, N) :- !, % record new clause
    sensym(clause(Name), % make up new name
    assert(clause(Name, Consequent, Antecedent, N)), % assert clause
    writeln('%t is name of new resolvant %1 <- %1 at depth %t
\n', [Name, Consequent, Antecedent, N]). % tell user

(* Find a clause at depth N */

\texttt{find_clause(\texttt{Name}, \texttt{Consequent}, \texttt{Antecedent}, 0) :-}
    clause(Name, Consequent, Antecedent, \texttt{topclause}), !.
ineed\_clause(Name,Consequent,Antecedent,N) :-
    clause(Name,Consequent,Antecedent,N).

* MINI-PROJECTS

- Try this theorem prover with the clauses of file EQUAL.
- Experiment by making up some clauses of your own and trying them out.
- Modify the theorem prover to print out the solution when it has found it.
- Modify the theorem prover to remove the input restriction.
- Modify the theorem prover to incorporate the literal selection restriction.
- Build a depth first theorem prover along the same lines.
yes
: ?- breadth(0).
clause1 is name of new resolvant
<-
equal(y, _101)
equal(x, _101)
at depth 1

clause2 is name of new resolvant
<-equal(y, y)
at depth 2

clause3 is name of new resolvant
<-equal(x, y)
at depth 2

clause4 is name of new resolvant
<-equal(y, _128)
equal(_127, _128)
equal(x, _127)
at depth 2

clause5 is name of new resolvant
<-equal(x, _128)
equal(_127, _128)
equal(y, _127)
at depth 2

Success! Empty Clause Found

[ Execution aborted ]

: ?- core 60416 (31232 lo-ses + 29184 hi-ses)
heap 26112 = 23649 in use + 2463 free
local 1177 = 16 in use + 1161 free
local 1024 = 16 in use + 1008 free
trail 511 = 0 in use + 511 free
 0.65 sec. runtime
<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>append/3</td>
<td>utility</td>
<td>heuristic/1 resolve/3</td>
</tr>
<tr>
<td>clause/4</td>
<td>equal</td>
<td>go/0 successor/2 resolve/3 record_clause/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pick_best/3 Pick_best/5 find_clause/4</td>
</tr>
<tr>
<td>mpare/8</td>
<td>heuris</td>
<td>Pick_best/5</td>
</tr>
<tr>
<td>evaluate/3</td>
<td>heuris</td>
<td>record_clause/3 find_clause/4</td>
</tr>
<tr>
<td>find_clause/4</td>
<td>heuris</td>
<td></td>
</tr>
<tr>
<td>findall/3</td>
<td>utility</td>
<td>heuristic/1</td>
</tr>
<tr>
<td>gensym/2</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
<tr>
<td>go/0</td>
<td>heuris</td>
<td></td>
</tr>
<tr>
<td>heuristic/1</td>
<td>heuris</td>
<td>go/0 heuristic/1</td>
</tr>
<tr>
<td>Pick_best/3</td>
<td>heuris</td>
<td>heuristic/1</td>
</tr>
<tr>
<td>Pick_best/5</td>
<td>heuris</td>
<td>Pick_best/3 Pick_best/5</td>
</tr>
<tr>
<td>record_clause/3</td>
<td>heuris</td>
<td>resolve/3</td>
</tr>
<tr>
<td>resolve/3</td>
<td>heuris</td>
<td>successor/2</td>
</tr>
<tr>
<td>select/3</td>
<td>utility</td>
<td>resolve/3</td>
</tr>
<tr>
<td>successor/2</td>
<td>heuris</td>
<td>&lt;user&gt; heuristic/1</td>
</tr>
<tr>
<td>writef/1</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
<tr>
<td>writef/2</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
</tbody>
</table>
Heuristic Search Theorem Prover,
QUAL contains test example,
Alan Bundy 19.6.81 */

/* Top Goal */
so :-
  clause(Goal,_,_,topclause),
  heuristic([Goal]).

/* Heuristic Search Strategy */
heuristic(Frinse) :-
  pick_best(Frinse,Current,Rest), % Pick the clause with best score
  findall(Clause,successor(Current,Clause),NewClauses), % findall its successors
  append(Rest,NewClauses,NewFrinse), % Put them on fringe
  heuristic(NewFrinse). % and recurse

/* Clause is a resolvent of Current with an input clause */
successor(Current,Clause) :-
  clause(Input,_,_,input), % Pick an input clause
  ( resolve(Current,Input,Clause) ; resolve(Input,Current,Clause) ), % resolve it with the current clause

/* Resolution Step */
resolve( Parent1, Parent2, Resolvant) :-
  clause(Parent1, Consequent1, Antecedent1, N1), % Get the two parents
  clause(Parent2, Consequent2, Antecedent2, N2),
  select(Literal, Consequent1, RestConseq1), % Select a common literal
  select(Literal, Antecedent2, RestAnte2), % and return the rest
  append(RestConseq1, Consequent2, Consequent), % Join the odd bits together
  append(Antecedent1, RestAnte2, Antecedent),
  record_clause(Consequent,Antecedent,Resolvant). % Record the clause

/* Record Existence of New Clause */
record_clause([],[],empty) :- % test for empty clause
  writeln('Success! Empty Clause Found
1
'), !, % tell the user
  abort. % and stop
record_clause(Consequent,Antecedent,Name) :- % test for loop
  clause(Name,Consequent,Antecedent,M), !, fail.
record_clause(Consequent,Antecedent,Name) !, % record new clause
  sensum(clause,Name), % make up a name
  evaluate(Consequent,Antecedent,N), % set score of clause
  assert(clause(Name,Consequent,Antecedent,N)), % assert clause
  writeln('%t is name of new resolvant %1 <- %1 with score %t
1
', [Name,Consequent,Antecedent,N]), % tell user
**Evaluation Function on Clauses (length of clause) */

\[ \text{evaluate(Consequent, Antecedent, Score) :-} \]
\[ \quad \text{length(Consequent, C),} \quad \% \text{add length of rhs} \]
\[ \quad \text{length(Antecedent, A),} \quad \% \text{to length of lhs} \]
\[ \quad \text{Score = C + A.} \quad \% \text{to set clause length} \]

**Pick clause with best score (i.e., lowest) */

\[ \text{pick_best([Hd1:T1], Choice, Rest) :-} \]
\[ \quad \text{clause(Hd, H, N),} \quad \% \text{Get score of first clause} \]
\[ \quad \text{pick_best(T1, Hd, N, Choice, Rest).} \quad \% \text{and run down list remembering best so far} \]

\[ \text{pick_best([], Hd, N, Hd, []).} \quad \% \text{When you set to the end return running score} \]

\[ \text{pick_best([Hd1:T1], Hd, N, Choice, [Hd3:Rest]) :-} \]
\[ \quad \text{clause(Hd1, H, N1),} \quad \% \text{Get score of first clause} \]
\[ \quad \text{compare(Hd, N, Hd1, N1, Hd2, N2, Hd3, N3),} \quad \% \text{Compare with running score and or} \]
\[ \quad \text{pick_best(T1, Hd2, N2, Choice, Rest).} \quad \% \text{recurse with new best score} \]

\[ \text{compare(Hd, N, Hd1, N1, Hd, N1, Hd1, N1) :-} \]
\[ \quad \% \text{put running score first} \]
\[ \quad \text{N =< N1, !.} \quad \% \text{unless new score is best} \]

\[ \text{compare(Hd, N, Hd1, N1, Hd1, N1, Hd, N).} \quad \% \text{Otherwise put new score first} \]

**Find a clause with score N */

\[ \text{find_clause(Name, Consequent, Antecedent, N) :-} \]
\[ \quad \text{clause(Name, Consequent, Antecedent, toplevel),} \quad !, \]
\[ \quad \text{evaluate(Consequent, Antecedent, N).} \]

\[ \text{find_clause(Name, Consequent, Antecedent, N) :-} \]
\[ \quad \text{clause(Name, Consequent, Antecedent, N).} \]

**MINI-PROJECTS**

1. Try this theorem prover with the equality clauses of EQUAL.
2. Experiment with clauses of your own devising.
3. Modify the theorem prover to print out the solution when it has found it.
4. Modify the theorem prover to remove the input restriction.
5. Experiment with different versions of the evaluation function by modifying 'evaluate' (see section 6.5.3 of 'Artificial Mathematicians').

/*
es
?- heuristic([goal]),
lause1 is name of new resolvant
<- equal(y, 158)
   equal(x, 158)
with score 2

lause2 is name of new resolvant
<- equal(y, y)
with score 1

lause3 is name of new resolvant
<- equal(x, y)
with score 1

lause4 is name of new resolvant
<- equal(y, 229)
   equal(228, 229)
   equal(x, 228)
with score 3

lause5 is name of new resolvant
<- equal(x, 229)
   equal(228, 229)
   equal(y, 228)
with score 3

Success! Empty Clause Found

[ Execution aborted ]

! ?- core 60928 (31744 lo-seg + 29184 hi-seg)
heap 26624 = 23909 in use + 2715 free
global 1177 = 16 in use + 1161 free
local 1024 = 16 in use + 1008 free
trail 511 = 0 in use + 511 free
0.01 sec. for 1 trail shift
0.80 sec. runtime
### Semant, Model and Divide Xref

<table>
<thead>
<tr>
<th>Predicate</th>
<th>File</th>
<th>Called By</th>
</tr>
</thead>
<tbody>
<tr>
<td>append/3</td>
<td>utility</td>
<td>resolve/3  paramodulate/3</td>
</tr>
<tr>
<td>apply/2</td>
<td>utility</td>
<td>some/3</td>
</tr>
<tr>
<td>checklist/2</td>
<td>utility</td>
<td>model/2</td>
</tr>
<tr>
<td>clause/4</td>
<td>divide</td>
<td>so/0 successor/2  resolve/3  paramodulate/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>record_clause/3 model/2</td>
</tr>
<tr>
<td>evaluate/3</td>
<td>model</td>
<td>&lt;user&gt; evaluate/3 is_true/2 is_false/2</td>
</tr>
<tr>
<td>sensym/2</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
<tr>
<td>so/0</td>
<td>semant</td>
<td></td>
</tr>
<tr>
<td>interpret/3</td>
<td>divide</td>
<td>evaluate/3</td>
</tr>
<tr>
<td>interpretation/1</td>
<td>divide</td>
<td>satisfiable/1</td>
</tr>
<tr>
<td>is_false/2</td>
<td>model</td>
<td>&lt;user&gt; model/2</td>
</tr>
<tr>
<td>is_true/2</td>
<td>model</td>
<td>&lt;user&gt; model/2</td>
</tr>
<tr>
<td>plist/3</td>
<td>utility</td>
<td>evaluate/3</td>
</tr>
<tr>
<td>model/2</td>
<td>model</td>
<td>satisfiable/1</td>
</tr>
<tr>
<td>paramodulate/3</td>
<td>semant</td>
<td>successor/2</td>
</tr>
<tr>
<td>record_clause/3</td>
<td>semant</td>
<td>resolve/3  paramodulate/3</td>
</tr>
<tr>
<td>replace/6</td>
<td>semant</td>
<td>paramodulate/3</td>
</tr>
<tr>
<td>replace1/4</td>
<td>semant</td>
<td>replace/6  replace2/4</td>
</tr>
<tr>
<td>replace2/4</td>
<td>semant</td>
<td></td>
</tr>
<tr>
<td>resolve/3</td>
<td>semant</td>
<td>successor/2</td>
</tr>
<tr>
<td>satisfiable/1</td>
<td>model</td>
<td>vet/1</td>
</tr>
<tr>
<td>Procedure</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>select/3</td>
<td>utility</td>
<td>resolve/3 paramodulate/3</td>
</tr>
<tr>
<td>semantic/1</td>
<td>semant</td>
<td>so/0 semantic/1</td>
</tr>
<tr>
<td>some/3</td>
<td>semant</td>
<td>replace1/4 some/3</td>
</tr>
<tr>
<td>successor/2</td>
<td>semant</td>
<td>semantic/1</td>
</tr>
<tr>
<td>vet/1</td>
<td>model</td>
<td>semantic/1</td>
</tr>
<tr>
<td>writef/1</td>
<td>utility</td>
<td>record_clause/3</td>
</tr>
<tr>
<td>writef/2</td>
<td>utility</td>
<td>record_clause/3 satisfiable/1</td>
</tr>
</tbody>
</table>
% Depth First Theorem Prover
% with vetting by use of interpretations and
% incorporating input restriction.
% Use with MODEL
% DIVIDE contains test example
% Alan Bundy 22.6.81 */

/* Top Goal */
so :-
  clause(Goal,_,_,topclause),   % Find the top clause
  semantic(Goal),              % and away you go

/* Depth First theorem Prover */
semantic(Old) :-
  successor(Old,New),          % Find a successor to the current clause
  vet(New),                   % check that it is unsatisfiable
  semantic(New),              % and recurse

/* Clause is a resolvant of Current with an input clause */
successor(Current,Clause) :-
  clause(Input,_,_,input),    % Pick an input clause
  ( resolve(Current,Input,Clause) ; % resolve it with the
    resolve(Input,Current,Clause) ; % current clause
    paramodulate(Current,Input,Clause) ; % or paramodulate it
    paramodulate(Input,Current,Clause) ),

/* Resolution Step */
resolve( Parent1, Parent2, Resolvant) :-
  clause(Parent1, Consequent1, Antecedent1, _), % Get the two parents
  clause(Parent2, Consequent2, Antecedent2, _),
  select(Literal, Consequent1, RestConseq1),   % Select a common literal
  select(Literal, Antecedent2, RestAntec2),    % and return the rest
  append(RestConseq1, Consequent2, Consequent), % Join the odd bits toget
  append(Antecedent1, RestAntec2, Antecedent),% record_clause(Consequent,Antecedent,Resolvant). % Record the clause

/* Paramodulation Step */
paramodulate( Parent1, Parent2, Paramodulant ) :-
  clause(Parent1, Consequent1, Antecedent1, _), % Get the two parents
  clause(Parent2, Consequent2, Antecedent2, _),
  select(equal(T,S), Consequent1, RestConseq1), % select an equation
  replace(T,S,Consequent2,Antecedent2,NewConseq2,NewAntec2), % put it in the other
  append(RestConseq1,NewConseq2,Consequent),  % Join the odd bits toget
  append(Antecedent1,NewAntec2,Antecedent),   % record_clause(Consequent,Antecedent,Paramodulant). % Record the clause
/* Replace T by S (or S by T) in clause */

replace(T,S,OldConse,OldAnte,NewConse,OldAnte) :- % replace T by S in
    replace1(T,S,OldConse,NewConse). % the consequent

replace(T,S,OldConse,OldAnte,OldConse,NewAnte) :- % or T by S in
    replace1(T,S,OldAnte,NewAnte). % the antecedent

replace(S,T,OldConse,OldAnte,NewConse,OldAnte) :- % or S by T in
    replace1(S,T,OldConse,NewConse). % the consequent

replace(S,T,OldConse,OldAnte,OldConse,NewAnte) :- % or S by T in
    replace1(S,T,OldAnte,NewAnte). % the antecedent

/* Replace T by S in Old to set New */

replace1(T,S,Old,New) :-
    some(replace2(T,S),Old,New). % replace one of the literals

replace2(T,S,Old,New) :-
    % replace this occurrence

    replace2(T,S,Old,New) :- % replace one of the arguments
    Old =.. [Sym ! OldArgss],
    replace1(T,S,OldArgss,NewArgss),
    New =.. [Sym ! NewArgss]. % put it all together again

/* Record Existence of New Clause */

record_clause([],[],empty) :- % test for empty clause
    writeln('Success! Empty Clause Found\n\n'), !, % tell user
    abort. % and stop

record_clause(Consequent,Antecedent,Name) :- % test for loop
    clause(Name,Consequent,Antecedent,_), !. % i.e. clause with same in

    record_clause(Consequent,Antecedent,Name) :- !, % record new clause
    gensym(clause,Name), % make up new name
    assert(clause(Name,Consequent,Antecedent,New)), % assert clause
    writeln('%t is name of new resolvent %1 <- %1
\n\n', [Name,Consequent,Antecedent]), % tell user

/* Apply Pred to Just one element of list */

some(Pred, [Hd1 : T1], [Hd2 : T1]) :-
    apply(Pred, [Hd1,Hd2]), % apply it to this one

some(Pred, [Hd : T11], [Hd : T12]) :-
    some(Pred, T11, T12). % or one of the others

/* MINI-PROJECTS
1. Try out theorem prover with the arithmetic clauses and models of
   file DIVIDE.
2. Experiment with some clauses and models of your own devising (See
Chapter 10 for some ideas.

3. Modify the theorem prover so that it works by breadth first search
   (Compare with file BREADT).

4. Modify the theorem prover so that it works by heuristic search
   (Compare with file HEURIS).
* MODEL.

low to evaluate a clause in an interpretation
Provided it is variable free!!)

ilan Bundy 22.6.81 */

* Vet the clause in any interpretations */

let(Clause) :-
not satisfiable(Clause). % Clause has no model

* Clause is satisfiable in some Interpretation */
satisfiable(Clause) :-
interpretation(InterP), % If there is an interpretation
model(InterP, Clause), % in which Clause is true
writef('t rejected by %t

',Clause,Interp). % tell user

* Interpretation is a model of a Clause */
model(Interp, Clause) :-
clause(Clause, Consequent, Antecedent, _), % Get Clause definition
checklist(is_true(Interp), Consequent), % Check all lhs literals are true
checklist(is_false(Interp), Antecedent). % and all rhs ones are false

* Evaluate expression in Interpretation */
evaluate(Interp, Integer, Integer) :- integer(Integer), !, % integers represent
evaluate(Interp, Constant, Value) :- % other constants have values associated
atom(Constant), !, interpret(Interp, Constant, Value).
evaluate(Interp, Complex, Value) :-
Complex =.. [Sym : Arqs], !, % recurse on arguments of
maplist(evaluate(Interp), Arqs, Vals), % complex terms
Complex1 =.. [Sym : Vals], % then interpret topmost
interpret(Interp, Complex1, Value). % symbol

* Evaluation of clause (hacks for checklist) */
s_true(Interp, Literal) :- evaluate(Interp, Literal, true).
s_false(Interp, Literal) :- evaluate(Interp, Literal, false).
/* DIVIDE. */

clauses and Model for not divides example (see notes P124)

 Alan Bundy 22.6.81 */

/* Clauses */

clause(right, [not_div(X*Z,Y)], [not_div(X,Y)], input). % Input clauses
clause(left, [not_div(Z*X,Y)], [not_div(X,Y)], input).
clause(thirty, [equal(30,2*3*5)], [], input).
clause(hypothesis, [not_div(5,a)], [], input).
clause(conclusion, [], [not_div(30,a)], toPclause). % Top clause

/* Models */

/* arith2 */
interpretation(arith2). % arith2 is an interpretation
interpret(arith2, a, 2). % meanings of a
interpret(arith2, not_div(X,Y), false) :- 0 is Y mod X, !.
interpret(arith2, not_div(X,Y), true).
interpret(arith2, equal(X,Y), true) :- X =:= Y, !.
interpret(arith2, equal(X,Y), false).
interpret(arith2, X*Y, Z) :- Z is X*Y. % meanings of *

/* arith3 */
interpretation(arith3). % arith3 is an interpretation
interpret(arith3, a, 3). % meanings of a
interpret(arith3, not_div(X,Y), false) :- 0 is Y mod X, !.
interpret(arith3, not_div(X,Y), true).
interpret(arith3, equal(X,Y), true) :- X =:= Y, !.
interpret(arith3, equal(X,Y), false).
interpret(arith3, X*Y, Z) :- Z is X*Y. % meanings of *
yes
! ?- do.
clause1 is name of new resolvant
<-
    not_div(2*3*5,a)

clause2 is name of new resolvant
<-
    not_div(2*3,a)

clause3 is name of new resolvant
<-
    not_div(2,a)

clause3 rejected by arith2

clause4 is name of new resolvant
<-
    not_div(3,a)

clause4 rejected by arith3

clause5 is name of new resolvant
<-
    not_div(5,a)

Success! Empty Clause Found

Execution aborted ]

! ?- core 60928 (31744 lo-ses + 29184 hi-ses)
heap 26624 = 24583 in use + 2041 free
global 1177 = 16 in use + 1161 free
ocal 1024 = 16 in use + 1008 free
ail 511 = 0 in use + 511 free
0.10 sec. for 2 GCs gaining 625 words
0.04 sec. for 3 local shifts and 7 trail shifts
2.94 sec. runtime
**PROLOG CROSS REFERENCE LISTING**

<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply/2</td>
<td>utility</td>
<td>some/3</td>
</tr>
<tr>
<td>rewrite/2</td>
<td>rewrit</td>
<td>normalize/2 so/0</td>
</tr>
<tr>
<td>rewrite/2</td>
<td>rewrit</td>
<td>rewrite/2</td>
</tr>
<tr>
<td>normalize/2</td>
<td>rewrit</td>
<td>rewrite/2</td>
</tr>
<tr>
<td>replace/4</td>
<td>rewrit</td>
<td>replace/4 some/3</td>
</tr>
<tr>
<td>rule/3</td>
<td>rewrit</td>
<td>rewrite/2</td>
</tr>
<tr>
<td>some/3</td>
<td>rewrit</td>
<td>rewrite/2 so/0</td>
</tr>
<tr>
<td>writef/2</td>
<td>utility</td>
<td>rewrite/2</td>
</tr>
</tbody>
</table>
Simple depth-first search rewrite rule system
see Artificial Mathematicians p 104.
Use with UTIL
Alan Bundy 10.7.81 */

Find Normal Form of Expression */
normalize(Expression, NormalForm) :- % To put an expression in normal form:
    rewrite(Expression, Rewritings), % Rewrite it once
    normalize(Rewritings, NormalForm), % and recurse
normalize(Expression, Expression) :- % The expression is in normal form
    not rewrite(Expression, _). % if it cannot be rewritten

Rewrite Rule of Inference */
rewrite(Expression, Rewritings) :- % To rewrite Expression
    rule(Name, LHS, RHS), % set a rule LHS =\rightarrow RHS
    replace(LHS, RHS, Expression, Rewritings), % replace LHS by RHS
    writef('%t rewritten to %t by rule %t
', [Expression, Rewritings, Name]).

Replace single occurrence of T by S in Old to set New */
replace(T, S, T, S). % replace this occurrence
replace(T, S, Old, New) :- % replace one of the arguments
    Old =.. [Sym : OldArss], % set the arguments
    some(replace(T, S), OldArss, NewArss), % replace one
    New =.. [Sym : NewArss]. % put it all together again

Apply Pred to just one element of list */
some(Pred, [Hd1 \ T1], [Hd2 \ T1]) :-
    apply(Pred, [Hd1, Hd2]). % apply it to this one
some(Pred, [Hd1 \ T1], [Hd2 \ T12]) :-
    some(Pred, T11, T12), % or one of the others

Some rules */
rule(1, X*0, 0). % Algebraic Simplification rules
rule(2, 1*X, X).
rule(3, X^0, 1).
rule(4, X+0, X).

A Typical Problem */
io :- normalize( (a^2*0)*5 + b*0, NormalForm),
    writef('Normal Form is %t\n\n', [NormalForm]),
    fail.
MINI-PROJECTS

1. Try out the system with the arithmetic rules given above.

2. Experiment with some rules of your own devising (Suggestions can be found in chapter 9 and section 5.2 of 'Artificial Mathematicians').

3. Modify the system so that it works by: (a) call by value, (b) call by name (See p107 of 'Artificial Mathematicians').
\[
\begin{align*}
\text{Normal Form is } 5
\end{align*}
\]
rewritten to $5+0$ by rule 2
rewritten to $a^0*5$ by rule 4
rewritten to $1*5$ by rule 3
rewritten to $5$ by rule 2
Form is $5$
rewritten to $5$ by rule 2
rewritten to $1*5$ by rule 3
rewritten to $a^0*5$ by rule 4
rewritten to $1*5$ by rule 3
rewritten to $5$ by rule 2
Form is $5$
rewritten to $5$ by rule 2
rewritten to $1*5$ by rule 3
rewritten to $a^0*5$ by rule 4
rewritten to $1*5$ by rule 3
rewritten to $5$ by rule 2
rewritten to $5$ by rule 2
rewritten to $1*5$ by rule 3
rewritten to $a^0*5$ by rule 4
rewritten to $a^0*5+b*0$ by rule 1

?- core 60416 (31232 lo-ses + 29184 hi-ses)
heap 26112 = 23454 in use + 2658 free
local 1175 = 16 in use + 1159 free
ocal 1024 = 16 in use + 1008 free
ail 511 = 0 in use + 511 free
.00 sec. for 1 trail shift
.02 sec. runtime
<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>sensym/2</td>
<td>utility</td>
<td>skolem/4</td>
</tr>
<tr>
<td>opposite/2</td>
<td>skolem</td>
<td>skolem/4</td>
</tr>
<tr>
<td>skolem/2</td>
<td>skolem</td>
<td>skolem/4 test1/1 test2/1</td>
</tr>
<tr>
<td>skolem/4</td>
<td>skolem</td>
<td>skolem/2 skolem/4</td>
</tr>
<tr>
<td>subst/3</td>
<td>utility</td>
<td>skolem/4</td>
</tr>
<tr>
<td>test1/1</td>
<td>formul</td>
<td>skolem/4</td>
</tr>
<tr>
<td>test2/1</td>
<td>formul</td>
<td>skolem/4</td>
</tr>
<tr>
<td>univ/3</td>
<td>skolem</td>
<td>skolem/4</td>
</tr>
</tbody>
</table>
/* Skolem Normal Form Procedure
(FORMUL contains test examples)
Alan Bundy 17.3.79 */

/* Operator declarations */
:- op(400,xfy,;), % Disjunction
:- op(300,xfy,<>), % Double Implication
:-op(400,xfy,->), % Implication

/* Skolem normal form */
skolem(Sentence,NormalForm) :- !,
    skolem(Sentence,NormalForm,[J,0]), % Normal calling pattern
    % assume no free vars and being as
    skolem(P <-> Q, (P1->Q1)&(Q1->P1),Vars,Par) :- !, % Double implication
    skolem(P,P1,Vars,Par), skolem(Q,Q1,Vars,Par).

skolem(all(X,P),P1,Vars,0) :- !,
    skolem(P,P1,[X;Vars],0). % Universal quantification

skolem(all(X,P),P2,Vars,1) :- !,
    sensym(f,Fs), univ(Fs,Vars,F),
    subst(X=F,P,P1),
    skolem(P1,P2,Vars,1). % Dual case

skolem(some(X,P),P2,Vars,0) :- !,
    sensym(f,Fs), univ(Fs,Vars,F),
    subst(X=F,P,P1),
    skolem(P1,P2,Vars,0). % Existential quantification

skolem(some(X,P),P1,Vars,1) :- !,
    skolem(P,P1,[X;Vars],1). % Dual case

skolem(P->Q,P1->Q1,Vars,Par) :- !, % Implication
    opposite(Par,Par1), skolem(P,P1,Vars,Par1),
    skolem(Q,Q1,Vars,Par).

skolem(P;Q,P1;Q1,Vars,Par) :- !, % Disjunction
    skolem(P,P1,Vars,Par), skolem(Q,Q1,Vars,Par).

skolem(P&Q,P1&Q1,Vars,Par) :- !, % Conjunction
    skolem(P,P1,Vars,Par), skolem(Q,Q1,Vars,Par).

skolem(not P,not P1,Vars,Par) :- !, % Negation
    opposite(Par,Par1), skolem(P,P1).

skolem(Pred,Pred,Vars,Par) :- !. % Atomic formula

/* Opposite parities */
/* MINI-PROJECTS

1. Try out on various formulae.

2. Modify the program to deal with bounded quantification, e.g.,
   all_in(X,Set,P) — meaning, for all X in Set, P is true.

3. Build programs for putting formulae in: (a) Prenex Normal Form, (b)
   Conjunctive Normal Form, as per section 5.2 of 'Artificial
   Mathematicians’. You may wish to use file REWRIT.

//
/* FORMUL.

Test Formulae for SKOLEM program

Alan Bundy 23.6.81 */

test1(Ans) :-
    skolem( all(a, all(b, all(c, some(x, a*x^2 + b*x + c = 0)))), Ans).

test2(Ans) :-
    skolem( all(m, some(delta, all(x, (abs(x)=<delta) -> (1/x)>m))), Ans).
yes
! ?- test1(A).
A = a*f1(c,b,a)^2+b*f1(c,b,a)+c=0

yes
! ?- test2(A).
A = (abs(x)=f2(m))->1/x>m

yes
! ?- core 60416 (31232 lo-seg + 29184 hi-seg)
heap 26112 = 23809 in use + 2303 free
global 1177 = 16 in use + 1161 free
local 1024 = 16 in use + 1008 free
trail 511 = 0 in use + 511 free
1.00 sec. runtime
<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_bomb/3</td>
<td>boyer</td>
<td>usliness/3</td>
</tr>
<tr>
<td>analyse/3</td>
<td>boyer</td>
<td>find_usly/3 add_bomb/3</td>
</tr>
<tr>
<td>append/3</td>
<td>utility</td>
<td>find_usly/3 def_eval/2</td>
</tr>
<tr>
<td>def_eval/2</td>
<td>boyer</td>
<td>symbol_eval/3 open_eval/2</td>
</tr>
<tr>
<td>eval/2</td>
<td>utility</td>
<td>symbol_eval/3</td>
</tr>
<tr>
<td>eval/2</td>
<td>boyer</td>
<td>symbol_eval/3 def_eval/2</td>
</tr>
<tr>
<td>expand/4</td>
<td>boyer</td>
<td>symbol_eval/3</td>
</tr>
<tr>
<td>fertilise/3</td>
<td>boyer</td>
<td>prove_step/2</td>
</tr>
<tr>
<td>find_usly/3</td>
<td>boyer</td>
<td>usliness/3 find_usly/3</td>
</tr>
<tr>
<td>generalise/2</td>
<td>undefined</td>
<td>prove1/2</td>
</tr>
<tr>
<td>loop_threat/2</td>
<td>boyer</td>
<td>analyse/3</td>
</tr>
<tr>
<td>make_fault/2</td>
<td>boyer</td>
<td>expand/4 make_fault/2</td>
</tr>
<tr>
<td>memberchk/2</td>
<td>utility</td>
<td>nontrivial/1</td>
</tr>
<tr>
<td>merge/3</td>
<td>boyer</td>
<td>symbol_eval/3 rewrite/3</td>
</tr>
<tr>
<td>nasty_car_or_cdr/1</td>
<td>boyer</td>
<td>usliness/3</td>
</tr>
<tr>
<td>nontrivial/1</td>
<td>boyer</td>
<td>nasty_car_or_cdr/1</td>
</tr>
<tr>
<td>open_eval/2</td>
<td>boyer</td>
<td>expand/4</td>
</tr>
<tr>
<td>open_fn/2</td>
<td>boyer</td>
<td>expand/4</td>
</tr>
<tr>
<td>pickindvars/3</td>
<td>boyer</td>
<td>prove1/2</td>
</tr>
<tr>
<td>prove/1</td>
<td>boyer</td>
<td>prove1/2 prove_base/2</td>
</tr>
<tr>
<td>Command</td>
<td>Type</td>
<td>Command</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>prove/2</td>
<td>boyer</td>
<td>prove/1</td>
</tr>
<tr>
<td>prove_base/2</td>
<td>boyer</td>
<td>prove_by_induction/2</td>
</tr>
<tr>
<td>prove_by_induction/2</td>
<td>boyer</td>
<td>prove/2</td>
</tr>
<tr>
<td>prove_step/2</td>
<td>boyer</td>
<td>prove_by_induction/2</td>
</tr>
<tr>
<td>rewrite/3</td>
<td>boyer</td>
<td>symbol_eval/3 rewrite/3</td>
</tr>
<tr>
<td>select/3</td>
<td>utility</td>
<td>merge/3</td>
</tr>
<tr>
<td>subst/3</td>
<td>utility</td>
<td>prove_base/2 prove_step/2 fertilise/3</td>
</tr>
<tr>
<td>symbol_eval/3</td>
<td>boyer</td>
<td>prove/1 symbol_eval/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>prove_step/2</td>
</tr>
<tr>
<td>uselessness/3</td>
<td>boyer</td>
<td>expand/4 find_usly/3</td>
</tr>
<tr>
<td>itef/1</td>
<td>utility</td>
<td>prove/1 prove_base/2</td>
</tr>
<tr>
<td>writef/2</td>
<td>utility</td>
<td>prove/1</td>
</tr>
<tr>
<td>w/0</td>
<td>boyer</td>
<td></td>
</tr>
<tr>
<td>z/0</td>
<td>boyer</td>
<td></td>
</tr>
</tbody>
</table>
A simple Boyer-Moore theorem prover as in Chap. II The Productive Use of Failure in 'Artificial Mathematicians'.

The code written here is a simplified version of the algorithm described in a paper of J Moore 'Computational Logic: Structure Sharing and Proof of Program Properties', Xerox report CSL 75-2, which appeared also as Dept. of Computational Logic memo no. 68, and the second part of Moore's Ph.D. thesis. This is a simplified version of the theorem-prover described in the book 'A Computational Logic' by Boyer and Moore.

Variable and procedure names have been chosen to be the same as much as possible, and the mini-projects will be aided by the description in the paper.

%% PROOF STRATEGY %%

% To prove a theorem use the following algorithm % (on p. of 'Artificial Mathematicians'):
% 1.-2. Try symbolic evaluation, recording the reasons for failure in the list Analysis,
% 3.-4. If unsuccessful, try proof by induction using the previously generated failure list to suggest the induction scheme,
% 5. Finally try generalising the theorem if the induction was unsuccessful.

prove(A) :-
  writef('Tryins to Prove %t
', [A]),
  symbol_eval(A, B, Analysis),
  prove(B, Analysis).

prove(tt, _) :-
  writef('Expression evaluated to tt
').

prove(A, Analysis) :-
  pickindvars(A, Analysis, Var),
  prove_by_induction(A, Var).

prove(A, Analysis) :-
  generalise(A, New),
  !,
  prove(New=tt).

%%% SYMBOLIC EVALUATION %%
% Symbolic evaluation is performed by the procedure
% symbol_eval(A,New,Analysis) which evaluates
% expression A into expression New producing
% failure bas Analysis.
% Primitive pure LISP functions, i.e. car,cdr,cond
% and equal are handled by an evaluator if they
% can be simplified. Otherwise function arguments
% are symbolically evaluated bottom-up,
% bottomins out on atomic expressions. Function
% definitions are expanded according to the criteria
% described in the section of code
% % EXPANSION OF FUNCTION DEFINITIONS %
%

symbol_eval(tt,tt,[]) :- !. % Finished if expression evaluates to tt

symbol_eval([],[],[]) :- !.

symbol_eval(A,B,Analysis) :-
  eval(A,A1),
  !,
  symbol_eval(A1,B,Analysis).

symbol_eval(A,B,Analysis) :-
  A=..[Pred|Arss],
  rewrite(Arss,Arss1,Analysis),
  A1=..[Pred|Arss1],
  expand(Pred,A1,A2,Fault),
  merge(Fault,Analysis,Analysis),
  def_eval(A2,B).

rewrite([],[],[]) :- !,

rewrite(X,X,[]) :- atomic(X), !.

rewrite([H:T],[H1:T1],Analysis) :-
  symbol_eval(H,H1,Fault),
  rewrite(T,T1,Desc),
  merge(Fault,Desc,Analysis).

% Evaluator

eval(car([],[]),[]),!,
eval(cdr([],[]),[]),!,
eval(car(cons(H,T)),H),!,
eval(cdr(cons(H,T)),T),!,
eval(cond([],U,V),V),!,
eval(cond(cons(X,Y),U,V),U),!,
eval(equal(X,X),tt),!.

% EXPANSION OF FUNCTION DEFINITIONS %

% Functions that can be expanded according to
% their function definition are contained in an
% open_fn predicate. In rewriting expressions
% functions are expanded where possible
% using the predicate open_eval unless
% usly expressions are found. In this case the
% fault description is returned as described
% in section 3.2 of Moore's paper.

expand(Pred,Clause,Clause,Fault) :-
    open_fn(Clause,Newclause),
    usliness(Pred,Newclause,Bomb),
    Bomb\==[[]],
    !,
    make_fault(Bomb,Fault).

expand(Pred,Clause,Newclause,[]) :-
    open_eval(Clause,Newclause),
    usliness(Pred,Newclause,Bomb),
    !,
    make_fault(Bomb,Fault).

expand(Pred,Clause,Clause,[]).% original

open_fn(append(X,Y),cond(X,cons(car(X),append(cdr(X),Y)),Y)) :- !.

open_eval(append(X,Y),Clause) :-
    def_eval(car(X),C1),
    def_eval(cdr(X),C2),
    def_eval(cond(X,cons(C1,append(C2,Y)),Y)),Y),Clause).

def_eval(X,Y) :- eval(X,Y), !.

def_eval(X,Y), !.

% Make up a fault description from the faults
% returned when trying to expand a recursively
% defined function.

make_fault([],[]) :- !.

make_fault([fault(B,F);Bombs],[fault(B,F);Faults]) :-
    make_fault(Bombs,Faults).

make_fault([bomb(X)],[fault(bomb(X),fail([]))]).

make_fault([fail(X)],[fault(bomb([]),fail(X))]).

make_fault([fail(Y),bomb(X)],[fault(bomb(X),fail(Y))]).

% Is expression usly?

usliness(Pred,X,[]) :- atomic(X), !.

usliness(Pred,X,Analysis) :-
    nasty_car_or_cdr(X),
    !,
    analyse(Pred,X,Analysis).

usliness(_,X,Analysis) :-
    X=[],[Pred|Args],
    find_usly(Pred,Args,Analysis).

find_usly(_,[],[]) :- !.
% Find induction candidate
% A simple majority vote is used to decide
% which list to induct on.
% This is calculated by max.

pickindvirs(_,Bas,Var) :-
    max(Bas,[C],Term,0,N),
    Term = fault(bomb(cdr(Var)),fail(_)).

max([],A,A,N,N) :- !.

% Successively prove the base and step cases
prove_by_induction(A,Literal) :-
    % To prove the base case, substitute the nil list
    % for the induction variable, and try to prove
    % the resultant clause
    prove_base(A,Literal),
    prove_step(A,Literal),

prove_base(A,Literal) :-
    writeln('
 Base case'),
% To prove the step case, substitute the appropriate
% cons expression into the clause, symbolically
% evaluate as much as possible, then fertilise
% to prove the expression

prove_step(A,Literal) :-
    writef('\nStep case'),
    subst(Literal=cons(a1,Literal),A,New),
    symbol_eval(New,Clause,_),
    fertilise(A,Clause,Newclause),
    prove(Newclause).

fertilise(equal(X,Y),Clause,New) :- subst(X=Y,Clause,New).

%%% PROOF BY GENERALISATION %

%% Code to be written

% test cases
z :- prove(equal(append([],x),x)).
y :- prove(equal(append(a,append(b,c)),append(append(a,b),c))).

/* Mini-Projects

1. Add definitions of functions like reverse and copy to enable
   the theorem-prover to work on other examples. Explain
   how the theorem-prover might be modified to overcome
   any small problems that might arise.

2. Use more sophisticated criteria to choose the induction variable,
   or more generally the induction schema to be used.
   This will probably involve changing the way the failures
   are returned in the variable Analysis.

3. Write a more powerful version of fertilise.

4. Write code to perform generalise.

*/

% merge merges two lists
merge([],Bas,Bas) :- !.
merge(Bas,Var,Hack) :- var(Var), !, merge(Bas,[variable],Hack).
merge([pair(Usly,K):T],Bas,Newbas) :-
    select(pair(Usly,N),Bas,Rest),
    !,
    M is N + K,
    merge(T,[pair(Usly,M):Rest],Newbas),
    !,
    merge(T,Bas,Newbas).
merse([Pair(Ugly,N):T],Bas,Newbas) :-
    !,
    merge(T,[Pair(Ugly,N):Bas],Newbas).

merge([H:T],Bas,Newbas) :-
    select([Pair(H,N):Bas],Rest),
    !,
    M is N + 1,
    merge(T,[Pair(H,M):Rest],Newbas).

merge([H:T],Bas,Newbas) :-
    merge(T,[Pair(H,1):Bas],Newbas).

A simple tautology checker based on the one by Boyer & Moore. The basic idea is to take a propositional formula formed from the constants '◊'=false, '1'=true, propositional variables represented by Prolog atoms, and the functors not/1, and/2, or/2, imp/2, if/3. This is then converted to an equivalent formula using 'if' alone, where the test is a single propositional variable. Then the cases of the test variables are examined.

```prolog

A simple tautology checker based on the one by Boyer & Moore. The basic idea is to take a propositional formula formed from the constants '◊'=false, '1'=true, propositional variables represented by Prolog atoms, and the functors not/1, and/2, or/2, imp/2, if/3. This is then converted to an equivalent formula using 'if' alone, where the test is a single propositional variable. Then the cases of the test variables are examined.

\*/

:- op(100, fy, not).
:- op(200, xfy, and).
:- op(300, xfy, or).
:- op(400, xf, imp).

rewrite(if(Test, Left, Right), Answer) :-
    rewrite(Test, Test1),
    rewrite(Left, Left1),
    rewrite(Right, Right1),
    buildif(Test1, Left1, Right1, Answer).

rewrite(and(Left, Right), Answer) :-
    rewrite(Left, Left1),
    rewrite(Right, Right1),
    buildif(Left1, Right1, 0, Answer).

rewrite(or(Left, Right), Answer) :-
    rewrite(Left, Left1),
    rewrite(Right, Right1),
    buildif(Left1, Right1, 1, Answer).

rewrite(imp(Left, Right), Answer) :-
    rewrite(Left, Left1),
    rewrite(Right, Right1),
    buildif(Left1, Right1, 1, Answer).

rewrite(not(Left), Answer) :-
    rewrite(Left, Left1),
    buildif(Left1, 0, 1, Answer).

rewrite(Answer, Answer) :-
    atomic(Answer).

buildif(T, 1, 0, T).
buildif(T, L, L, L).
buildif(1, L, R, L).
buildif(0, L, R, R).
buildif(T, L, R, if(T, L, R)) :- atom(T).
buildif(if(T, TL, TR), L, R, if(T, AL, AR)) :-
    buildif(TL, L, R, AL),
    buildif(TR, L, R, AR).
```
An association list is a list of variable=value terms, where each variable is a Propositional variable, and the value is 0 or 1.

```
/*
lookup(Atom, [Atom=Value|_], Value) :- !,
lookup(Atom, [_:Rest], Value) :- !, lookup(Atom, Rest, Value),
lookup(Atom, [], 2),

check(if(Test, Left, Right), Alist, Value) :-
  lookup(Test, Alist, Selector), !,
  cases(Selector, Test, Left, Right, Alist, Value),
check(1, Alist, 1),
check(0, Alist, 0),
check(Atom, Alist, Value) :-
  atom(Atom),
  lookup(Atom, Alist, Value).

/* check(Expr, Alist, 0) => Expr can only be false under Alist, check(Expr, Alist, 1) => Expr can only be true under Alist, check(Expr, Alist, 2) => Expr may be true or false. */

cases(0, Test, Left, Right, Alist, Value) :-
  check(Right, Alist, Value),
cases(1, Test, Left, Right, Alist, Value) :-
  check(Left, Alist, Value),
cases(2, Test, Left, Right, Alist, Value) :-
  check(Left, [Test=1:Alist], VL),
  check(Right, [Test=0:Alist], VR),
  combine(VL, VR, Value).

combine(X, X, X),
combine(X, Y, 2) :- X \= Y.
/\ The top level routine */

classify(Formula) :-
  rewrite(Formula, IfTree),
  check(IfTree, [], Value),
  describe(Value, Formula),
describe(2, Formula) :- write(Formula), write(‘ is contingent.’),
describe(1, Formula) :- write(Formula), write(‘ is always true.’),
describe(0, Formula) :- write(Formula), write(‘ is always false.’),
```
/* RULES.*/

Production Rule System for All
Alan Bundy 17.9.81

Use with UTIL, Example rules etc on file SUBTRA and SUM.
*/

so :-  % Top level goal
    apply_rule,   % Apply a production rule
    so,   % and recurse

apply_rule :-  % To apply a rule
    rule(Name,Condition, Action),  % Find a production rule
    satisfied(Condition),   % Check that its condition list is satisfied
    refract(Name,Condition),  % Check that rule has not already been fir
    writeln('Rule %s fired\n',[Name]),
    Action,  % If so, run its action

satisfied(Cond1 & Cond2) :- !,  % To satisfy two conditions
    satisfied(Cond1), satisfied(Cond2),  % satisfy one after another

satisfied(Condition) :-  % To check a condition
    short_memory(B1,B2,B3,B4,B5,B6),  % Get short term memory
    memberchk(Condition,[B1,B2,B3,B4,B5,B6]),  % Match condition

add(Item) :-  % To add item to memory
    retract(short_memory(B1,B2,B3,B4,B5,B6)),  % recover & delete memory
    assert(short_memory(Item,B1,B2,B3,B4,B5)),  % add new item & drop B6
    writeln('%s remembered\n',[Item]).

refract(Name,Cond) :-  % Refractoriness
    history(HistList),  % Recall history
    memberchk(Pair(Name,Cond),HistList),  % If we have been here before
    !, fail.  % then fail

refract(Name,Cond) :-  % otherwise update history
    retract(history(HistList)),
    assert(history([Pair(Name,Cond) | HistList])).

history([]).  % Initial settings for history
Production Rules for Subtraction by Decomposition
from O’Shea and Young WP42
Alan Bundy 22.9.81

Example for RULES, use with UTIL *
/

/* The Rules */

rule(fin, next_column, shift_left & take_diff & write_answer & abort).
rule(b2a, s_str_m, add(borrow)).
rule(b2c, s_ea_m, result(0) & add(next_column)).
rule(bs2, borrow, decrement).
rule(bs3, borrow, add_ten_to_m).
rule(cm, Process_column, compare).
rule(ts, Process_column, take_diff & add(next_column)).

/* The Actions */

shift_left :-
   retract(mark(X)),
   X1 is X+1,
   assert(mark(X1)).

take_diff :-
   mark(X),
   column(X, M, S),
   pos_diff(M, S, R),
   assert(answer(X, R)).

pos_diff(M, S, R) :-
   M>=S, !, R is M-S.

pos_diff(M, S, R) :-
   R is S-M.

result(R) :-
   mark(X),
   assert(answer(X, R)).

add_ten_to_m :-
   mark(X),
   retract(column(X, M, S)),
   M1 is M+10,
   assert(column(X1, M1, S)).

decrement :-
   mark(X), X1 is X+1,
   retract(column(X1, M, S)),
   M1 is M-1,
   assert(column(X1, M1, S)).

compare :-
   mark(X),
   column(X, M, S),
   compare1(M, S, Verdict),
   column(X, M, S).
add(Verdict).

compare(M,M,s_eq_m).
compare(M,S,s_str_m) :- S>M.
compare(M,S,m_str_s) :- M>S.

write_answer :-
column(1,M1,S1), answer(1,R1),
column(2,M2,S2), answer(2,R2),
writef(\texttt{t\hfill t
\hfill t
---
\hfill t
---
\hfill n
---
\hfill n

},
[M2,M1,S2,S1,R2,R1]).

/* Short Term Memory */
short_memory(process_column,nil,nil,nil,nil,nil).
/* SUM1.
Example sum for use with RULES and SUBTRA
Alan Bundy 23.9.81
*/
column(1,8,9).
column(2,4,1).
mark(1).
/* SUM2.

Example sum for use with RULES and SUBTRA
Alan Bundy 23.9.81
*/

column(1,9,8).
column(2,4,1).
mark(1).
/* SUM3.
Example sum for use with RULES and SUBTRA
Alan Bundy 23.9.81
*/
column(1,8,8),
column(2,4,1),
nark(1).
[ Execution aborted ]

! ?- restore(foo).

[ Execution aborted ]

! ?- sum2.

sum2 consulted 16 words 0.01 sec.

[ Execution aborted ]

! ?- sum3.

sum3 consulted 16 words 0.01 sec.
Execution aborted

?- core 60416 (31232 lo-ses + 29184 hi-ses)
heap 26112 = 23975 in use + 2137 free
global 1173 = 16 in use + 1157 free
local 1024 = 16 in use + 1008 free
trail 511 = 0 in use + 511 free

0.01 sec. for 5 trail shifts
1.25 sec. runtime
<table>
<thead>
<tr>
<th>PREDICATE</th>
<th>FILE</th>
<th>CALLED BY</th>
</tr>
</thead>
<tbody>
<tr>
<td>answer_query/1</td>
<td>mycin</td>
<td>check_for_query/4</td>
</tr>
<tr>
<td>askfor/3</td>
<td>mycin</td>
<td>sethypos/3</td>
</tr>
<tr>
<td>check_again_for_query/2</td>
<td>mycin</td>
<td>check_for_query/4</td>
</tr>
<tr>
<td>check_ans/5</td>
<td>mycin</td>
<td>askfor/3 check_ans/5</td>
</tr>
<tr>
<td>check_for_more/2</td>
<td>mycin</td>
<td>setdata/2</td>
</tr>
<tr>
<td>check_for_query/4</td>
<td>mycin</td>
<td>askfor/3</td>
</tr>
<tr>
<td>compare/4</td>
<td>mycin</td>
<td>merge/2 compare/4</td>
</tr>
<tr>
<td>consider/3</td>
<td>mycin</td>
<td>check_for_more/2</td>
</tr>
<tr>
<td>consult/0</td>
<td>mycin</td>
<td>start_session/0</td>
</tr>
<tr>
<td>current/2</td>
<td>mycin</td>
<td>setdata/2 transpred/1</td>
</tr>
<tr>
<td>data_class/3</td>
<td>mycin</td>
<td>sennew/2 setdescr/3 current/2</td>
</tr>
<tr>
<td>divide/3</td>
<td>mycin</td>
<td>sennew/2</td>
</tr>
<tr>
<td>explain/1</td>
<td>mycin</td>
<td>report/2</td>
</tr>
<tr>
<td>f/1</td>
<td>mycin</td>
<td>therapy_required/0 check_again_for_query/2</td>
</tr>
<tr>
<td>senheader/2</td>
<td>mycin</td>
<td>sennew/2</td>
</tr>
<tr>
<td>senmess/2</td>
<td>mycin</td>
<td>setdescr/3</td>
</tr>
<tr>
<td>sennew/2</td>
<td>mycin</td>
<td>setdata/2</td>
</tr>
<tr>
<td>senno/2</td>
<td>mycin</td>
<td>sennew/2 setans/3</td>
</tr>
<tr>
<td>set/4</td>
<td>mycin</td>
<td>same/4</td>
</tr>
</tbody>
</table>
set_nearest/1  mycin  check_for_query/4
setans/3    mycin  setlist/3
setdata/2   mycin  consult/0 setdata/2 consider/3
setdescr/3  mycin  setdata/2
sethYPos/3  mycin  setdata/2 set/4
setlist/3   mycin  setdescr/3 setlist/3
sive_evidence/1 mycin  transpred/1
sram/4      mycin  readans/2
intersect/3 utility
\textit{intersect}/3 mycin  set/4 intersect/3 known/1
know/3      undefined sethYPos/3 known/1
known/1     mycin  divide/3
last_class/2 mycin  setdata/2
maxhYP/2    mycin  set/4 maxhYP/2 known/1
member/2    utility
member/2    mycin  check_ans/5 member/2 set_nearest/1
merge/2     mycin  deduce/3 merge/2
min/2       mycin  min/2 rule/6
number/2    undefined senno/2 current/2
\textit{member}/3 mycin  sram/4 number/3
\textit{parameter}/2 mycin  check_ans/5
question/1  mycin  setans/3
question/2  mycin  askfor/3
readans/2   mycin  askfor/3 check_ans/5
report/2    mycin  check_for_query/4
rewrite/2   mycin  senheader/2 rewrite/2 setans/3
rule/6      mycin  rule_check/4
rule_check/4 mycin  \texttt{<user>} deduce/3
ruJ.eno/1  mycin  rule/6
same/4  mycin  sram/4 space/2 word/3 number/3
space/2  mycin  start_session/0
start_session/0  mycin  

test/4  mycin  check_again_for_query/2
therapy_required/0  mycin  start_session/0
translate/2  mycin  report/2 explain/1
translate/2  mycin  

write_known/1  mycin  
write_list/1  mycin  check_ans/5 write_list/1
writeprems/1  mycin  
writeval/1  mycin  write_known/1 writeprems/1 translate/2
% File: mycin
% Declarations
\:- Public
% Imports:
% End%
% Mycin(Prolog) — a Prolog rational reconstruction.
% Waterloo Prolog version by Peter Hammond 1980.
% converted to Dec-10 Prolog by Richard O’Keefe 1981.

% A consultation session is initiated by the goal start_session.

start_session :- therapy_required, !, consult.
start_session :- write('The patient needs no therapy.'), nl.

therapy_required :- f(setans(0, initiator, Ans)), Ans = yes.
consult :- setdata(0, PatientData), write(PatientData), nl.

% The hierarchical nature of the structure of PatientData is reflected
% in the recursive definition of setdata below.

setdata(Class4, Hypos) :-
    Class3 is Class4-1,
    current(Class3, Entity),
    last_class(Class4, Attribute), !,
    sethypos(Entity, Attribute, Hypos).

setdata(Class1, [Class1DataItem : OtherClass1Data]) :-
    Class1 < 4,
    sennew(Class1, Entity),
    setdescr(Class1, Entity, Descr),
    Class2 is Class1+1,
    setdata(Class2, Class2Data),
    Class1DataItem = w(Entity, Descr, Class2Data),
    check_for_more(Class1, OtherClass1Data).

% sennew generates a new entity of the required class and also
% prints a heading announcing the new entity.

sennew(Class, Entity) :-
    senno(Class, NewNo),
    data_class(Class, ClassName, Details),
    Entity = ClassName-NewNo, % e.g. patient-1
    senheader(Class, Entity).

% Each time a new entity is required senno generates a suitable
% cardinal for the entity.

senno(Class, NewNo) :-
    retract(number(Class, OldNo)), !,
    NewNo is OldNo+1,
    assert(number(Class, NewNo)).

senno(Class, 1) :-
    \+ number(Class, _),
    assert(number(Class, 1)).

senheader(ClassNo, Entity) :-
    repwrite(' ', ClassNo), repwrite('#', 8),
    write(Entity), repwrite(' ', 8), nl.

repwrite(Object, Times) :-
    Times > 0, Left is Times-1,
    write(Object), !,
    repwrite(Object, Left).

repwrite(Object, 0).
% setdescr asks the user to supply information of a background nature
% for a particular entity, and can also cause a message to be printed
% announcing the name of the first entity in the next class in the hierarchy.

setdescr(Class, Entity, Descr) :-
  data_class(Class, Name, Details),
  setlist(Class, Details, Descr),
  senmess(Class, Entity).

setlist(Class, [Item | OtherItems], [Ans | OtherAns]) :-
  setans(Class, Item, Ans),
  setlist(Class, OtherItems, OtherAns).

setlist(Class, [], []).% current returns the current entity of a particular class.
current(Class, Entity) :-
  number(Class, No),
  data_class(Class, ClassName, Details),
  Entity = ClassName-No.

% setans asks the user for and reads the value of an item of background
data. Each question is preceded by a new question number.

setans(Class, Item, Ans) :-
  senno(question, Q),
  rewrite(' ', Class),
  display('('), display(Q), display(')'),
  question(Item), ttyflush,
  read(Ans), nl.

% check_for_more asks the user if there is another entity of a
% particular class to consider.

check_for_more(Class, OtherData) :-
  data_class(Class, ClassName, Details),
  display('Is there another '),
  display(ClassName), display('? '), ttyflush,
  read(Ans),
  consider(Ans, Class, OtherData).

consider(no, Class, []).
consider(yes, Class, OtherData) :- setdata(Class, OtherData).

% same causes evidence to be gathered which bears on a particular
% value of a clinical parameter, and succeeds if the CF supporting
% this value is greater than 200.
same(Entity, Attribute, ReqVal, CFmi) :-
  set(Entity, Attribute, ReqVal, CFmi), !,
  CFmi > 200.

% set collects together the hypotheses relevant to determining the
% value of a clinical parameter and finds the largest of the CF-s
% supporting the possible values.

set(Entity, Attribute, ReqVal, CFmi) :-
  sethysos(Entity, Attribute, Hysos),
  intersect(Hysos, ReqVal, Intersect),
  maxhyp(Intersect, CFmi).
sethypos(Entity, Attribute, Hypo) :-
    (  know(Entity, Attribute, Hypo)
    ;  deduce(Entity, Attribute, Hypo)
    ;  askfor(Entity, Attribute, Hypo)
    ),
    Hypo \== [].

% deduce collects all the evidence for the value of a parameter, % merses evidence for the same value into one hypothesis, and % stores the information obtained.

deduce(Entity, Attribute, Hypo) :-
    basof(v(Val, Cf), rule_check(Entity, Attribute, Val, Cf), H),
    merge(H, Hypo),
    assert(know(Entity, Attribute, Hypo)).

% rule_check investigates a rule and calculates the Cf of the % deduction when the rule succeeds.

rule_check(Entity, Attribute, Value, Cf) :-
    rule(RuleNo, Entity, Attribute, Value, C, Tally),
    Cf is Tally*C/1000.

merge([], []),
merge([v(unk,1000)], []).
merge([H : Rest], [H1 : Rest1]) :-
    compare(H, Rest, H1, Tser1),
    merge(Tser1, Rest1).

compare(R, [], R, []).
compare(v(Val, Cf1), [v(Val, Cf2) : U], R, W) :-
    Cf3 is (1000-Cf1)*Cf2/1000 + Cf1,
    compare(v(Val, Cf3), U, R, W).

compare(v(Val1, Cf1), [v(Val2, Cf2) : U], R, [v(Val2, Cf2) : W]) :-
    Val1 \== Val2,
    compare(v(Val1, Cf1), U, R, W).

intersect([v(Val, Cf) : H], [Val : Rest], [v(Val, Cf) : H]) :- !,
    intersect(H, Rest, H1),
    intersect([v(Val, Cf) : H], [Alt : Rest], H1) :- !,
    intersect(H, [Alt], X),
    intersect([v(Val, Cf) : H], Rest, Y),
    union(X, Y, H1).

union([], List, []).
union([R], Y, [R|Y]).

maxhyp([], 0).
maxhyp([v(Val, Cf) : H], CFmi) :-
    maxhyp(H, C1),
    (C1 > Cf, !, CFmi = C1; CFmi = Cf).

% askfor causes the user to be asked for the value of a clinical % parameter. The answer is read and checked to see if it is one % of the question's possible answers.
askfor(Entity, Attribute, \[v(ActVal, Cf)\]) :-  
  question(Entity, Attribute),  
  reads(Answer1, Cf1),  
  check_for-query(Answer1, Cf1, A, C),  
  check_ans(Attribute, A, C, ActVal, Cf),  
  assert(know(Entity, Attribute, \[v(ActVal, Cf)\])).

check_ans(Attribute, A1, C1, A1, C1) :-  
  parameter(Attribute, Expected),  
  member(A1, Expected), 
  .

check_ans(Attribute, A1, C1, A, C) :-  
  parameter(Attribute, Expected),  
  display('Please enter one of the following:', nl,  
  write_list(Expected),  
  reads(Answer2, Cf2),  
  check_ans(Attribute, Answer2, Cf2, A, C).

question(Entity, Attribute) :-  
  display('Please enter the'),  
  display(Attribute),  
  display(' of'),  
  write(Entity),  
  display(': '),  
  ttyflush.

readans is used to read the user's reply to a request for the value of a clinical parameter. The value and its certainty factor are both read. The default Cf is 1000.

readans(Answer, Cf) :-  
  read(Reply),  
  name(Reply, Text),  
  sram(Answer, Cf, Text, []).

  sram(Answer, Cf) --> space, word(A), \{name(Answer, A)\},  
  \{"," , space, number(C) , ','\} , space,  
  \{name(Cf , C) ; \{Cf = 1000\}\}.

  space --> ['C'], \{C =< 32\} , space ; ['].

  word(['C:R']) --> ['C'], \{C > 32, C =\= 'C'\} , word(R).

  word([' ']) --> space.

  number(['C:R']) --> ['C'], \{C >= '0' , C =< '9'\} , number(R).

  number([' ']) --> space.

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

write_list([X|Y]) :-  
  write(X), nl, write_list(Y).

write_list([]) :-  
  nl.

min([X], X),
min([X,Y], Z) :-  
  min(Y, U),  
  \{ X < U, !, Z=X ; Z=U \}.

THE EXPLANATION SYSTEM

check_for_query is called after the user is asked to give a parameter value. If he said WHY or RULE the explanation system is entered.
check_for_query(Answer, Cf, A, C) :-
  ( Answer = why ; Answer = rule ), !,
  answer_query(Answer),
  set_nearest(Rule),
  report(Answer, Rule),
  check_again_for_query(A, C),
  check_for_query(Answer, Cf, Answer, Cf).

answer_query(why) :- % he can't be serious!
  display('To determine the senus of the orsanism,'), nl.
answer_query(rule) :-
  display('The current rule is!'), nl.

check_again_for_query(A, C) :-
  f(readans(A1, C1)),
  test(A1, C1, A, C).

test(why, 1000, A, C) :- !, fail.
test(A, C, A, C) :- A \== why.

-t_nearest(rule(N, E, A, R, C, T)) :-
  ancestors(AllOfThem),
  member(rule(N, E, A, R, C, T), AllOfThem).

report(why, Rule) :- explain(Rule).
report(rule, Rule) :- clause(Rule, Body), translate(Rule, Body).

% explain prints the known parameter values in a rule, those still % to be determined, and the deduction which could result.

explain(Rule) :-
  clause(Rule, Body),
  divide(Body, KnownPrems, UnknownPrems),
  write_known(KnownPrems),
  translate(Head, UnknownPrems),
  divide(',',CA,B>, ',',(A,0therKnown), Unknown) :-
    known(A),
    divide(B, OtherKnown, Unknown),
  divide(',',true, ',',(A,B)) :-
    \+ known(A),
  divide(A, true, A) :-
    \+ known(A),

write_known(true).
write_known(',',(A,B)) :-
  display(' It is known that!'), nl,
  writePrems(',',(A,B)),
  display(' therefore...'), nl.

writePrems(true).
writePrems(',',(A,B)) :-
  transPred(A),
  writePrems(B).
writePrems(A) :-
  transPred(A).

% translate gives a simple english translation of a rule.
translate(Head, Body) :-
    display('If :'), nl,
    writePrems(Body),
    display('Then:•>'), nl,
    transPred(Head),

transPred(min(M, N)).
transPred(same(E, A, R, C)) :-
    current(3, Entity),
    display('the '), write(A),
    display('of '), write(Entity),
    display('is '), write(R), nl. % the genus of organism-1 is [staph]
transPred(rule(N, E, A, V < C, T)) :-
    current(3, Entity),
    display('there is '),
    sive_evidence(C),
    display('evidence that the '), nl,
    display('of '), write(A),
    display('is '), write(V), nl,
    display(' (rule '),
    write(N), display(')'), nl.

known(same(E, A, R, C)) :-
    know(E, A, Hypos),
    intersect(Hypos, R, I),
    man~hyp(I, CFmi),
    CFmi > 200.

ruleNo(N) :-
    display('(Rule '),
    write(N), display(')'), nl.

writeval([M]) :- write(M).
sive_evidence(C) :-
    C > 800, display('strongly suggestive')
; C > 400, display('suggestive')
; C < 401, display('weakly suggestive')
.

THE KNOWLEDGE BASE
%
data_class lets the user declare the classes of data objects,
% their names, and the background details required.
data_class(0, patient, [name, sex, age]),
data_class(1, infection, [inftype, infdate]),
data_class(2, culture, [cultsite, cultdate]),
data_class(3, organism, []),
last_class(4, genus).

senmess(0, Entity),
senmess(1, Entity) :-
    display('The most recent culture associated with '),
    write(Entity),
    display('will be referred to as:'), nl.
senmess(2, Entity) :-
    display('The first significant organism from '),
    write(Entity),
display('will be referred to as:'), nl.

demess(3, Entity).

% question defines the questions which will elicit the values of
% particular details from the user.

question(name) :-
    display('Patient's name: '),
question(sex) :-
    display('Patient's sex: '),
question(age) :-
    display('Patient's age: ')

question(initiator) :-
    display('Have you been able to obtain positive cultures from a
    site at which the patient has an infection? ')

question(inftype) :-
    display('What is the infection? ')

question(inftype) :-
    display('When did this infection first appear? ')

question(cultsite) :-
    display('What site did the specimen of this culture come from? ')

question(cultdate) :-
    display('When was this culture obtained? ')

% parameter records the possible values of a clinical parameter.

parameter(senus, [unk, strep, neiss, bact, staph, coryn]),

parameter(sramstain, [unk, pos, nes]),

parameter(morPhology, [unk, rod, coccus]),

parameter(conformation, [unk, singles, longchains, shortchains]),

parameter(aerobicity, [unk, anaerobic, facul]).

% THE RULE BASE

rule(35, Entity, senus, bact, 600, Tally) :-
    same(Entity, sramstain, [nes], Cf1),
    same(Entity, morphology, [rod], Cf2),
    same(Entity, aerobicity, [anaerobic], Cf3),
    min(Cf1, Cf2, Cf3, Tally).

rule(9, Entity, senus, neiss, 800, Tally) :-
    same(Entity, sramstain, [nes], Cf1),
    same(Entity, morphology, [coccus], Cf2),
    min(Cf1, Cf2, Tally).

rule(306, Entity, senus, staph, 700, Tally) :-
    same(Entity, sramstain, [pos], Cf1),
    same(Entity, morphology, [coccus], Cf2),
    same(Entity, conformation, [singles], Cf3),
    min(Cf1, Cf2, Cf3, Tally).

rule(412, Entity, senus, strept, 950, Tally) :-
    same(Entity, sramstain, [pos], Cf1),
    same(Entity, morphology, [coccus], Cf2),
    same(Entity, conformation, [longchains], Cf3),
    min(Cf1, Cf2, Cf3, Tally).

rule(413, Entity, senus, strept, 800, Tally) :-
    same(Entity, sramstain, [pos], Cf1),
same(Entity, morphology, [coccus], Cf2),
same(Entity, conformation, [shortchains], Cf3),
min([Cf1, Cf2, Cf3], Tally).

f(X) :- call(X), !. % awful name!

% THE * END *