```
%   File   : CONT
%
%   Author : Bernard Silver
%   Updated: 5 March 1984
%   Purpose: CONTENTS PAGE FOR LP FOLDER


mec:lp.hlp                          Help File
mec:lp.hol                          Help File for helper
mec:filin                           Make LP
mec:filuti                          Make LPUTIL
------------
lmisc:lp.ref                        Cross Reference
------------
learn:sol                           Solve top level
learn:tl                            Work top level
learn:specia                        Special cases for Work
learn:genrrb                        Problem generator
------------
learn:newmet                        Make new methods
learn:desc                          Make schemas
learn:constr                        List hacking for above
------------
  rn:table                          Table of Conditions and Connections
  rn:cond                           Precondition Definitions
------------
learn:conJ                          Conjecture rewrite rules
learn:confir                        Confirm Conjectures
------------
learn:flag                          Flag settins
learn:char                          Character Input
learn:out                           Output Routines
------------
learn:interp                        Misc interpreted code
learn:comp                          Misc compiled code
------------
learn:method                        Top level method access
learn:imeth                         Some interpreted PRESS methods
learn:polpak                        Polynomial packase
press:collec                        Collection code
learn:poly                          More poly code
learn:los                           Los method
  rn:nasty                          Nasty Function Method
  rn:axioms                         Isolation, Collection and Attraction Axioms
------------
learn:weaknf                        Weak normal form code
learn:func                          Identify new unknown functions
learn:loop                          Loop checker
learn:rew                           Apply rewrite rule
------------
util:util.ops                       Operators for UTIL
util:arith.ops                      Arithmetic operators
learn:ops                           Operators for LP
util:files.pl                       File handlins
learn:imisce.pl                     Misc interpreted utilities
mec:edit.pl                         Interface to TOP
util:type.pl                        Type file from Prolos
util:trysee.pl                      Interface to helper
util:tidy.pl                        Expression tidier
util:writef.pl                      Output
learn:routin.pl                     Misc compiled utilities
```

util:struct.pl	Structure hacking
util:flagro.pl	Flag support
learn:time	Find time of day
learn:file	Check file name syntax
press:match	Pattern matcher
press:poltid	Polynomial tidier
learn:simp.ax	Axioms for tidy
learn:portr	Portray code
util:long.pl	Rational + long integer package
----------	
lmisc:lp.ccl	Load file for XREF
lmisc:lp.def	XREF definition file
lmisc:xref.ini	XREF asserted clause
mec:lp.mic	LP load mic
mec:lputil.mic	LPUTIL load mic
lp.sub	Load file for SUBFIL
lp.cng	Changes to LP
lp.sts	Clause count

Bernard Silver
11 February 1984

HELP FILE FOR LP

P.EXE is in mec:, so any member of the PRESS project can run the program run y typing LP.

P has two basic operations, solving equations and examining worked examples. he two methods are described below. Some of the descriptions talk about urning flags on and off. Unless otherwise indicated, to turn on flag Flag, ype "enable Flag", and to turn it off type "disable Flag". If the flag is lready set to that value a message to that effect is output.

quation Solving

o do the first type "solve(Eqn)", where Eqn is an equation in x, or solve(Eqn,X,Ans)" for the general case as in PRESS.

o start with, the program will not be able to solve many problems, as it does ot have the rules or schemas. If desired the standard rules, examples, and roblems can be read in by typing "wep", to read in the worked examples and ems, and "rules", to read in the rules and predicates. Alternatively, g "examples" loads the examples problems, rules and predicates. The tems are called t1 - t7, and the worked examples are trig1, trig2, trig3. There are more but they are not safe!)

he last equation can be rerun by typing "sredo". This is unlike PRESS in that he last equation is automatically stored without the need for a go predicate.

yping "generate_problem" causes LP to generate a set of equations similar to he one that produced the last schema, (see section on worked examples.) The ser can then indicate that LP should attempt to solve one of these equations. his allows the generality of the schema to be tested.

yping "show schemas" just displays the schemas, without asking if they should e removed.

yping "show solve" outputs all the steps on the solution path of the last quation, this command uses stored assertions and the equation is not rerun.

ed Examples

run a worked example, either use one of the standard problems, or type give_example" and the example can then be entered line by line, typing "end" n a new line to finish. "xredo" will rerun this example. Typing "old_xredo" ill first remove the new schema and any new rules, etc, that have just been dded, before rerunning the example (i.e. it restores the program to the old tate.)

n giving a worked example a few painful conventions have to be used. If a ine of the form A*B=0 occurs the next line must be A=0#B=0. Now the line A=0 or B=0) occurs, followed by the solution of this equation. Then the other actor is given, followed by this equation. For example:

cos(x) + 2 * cos(2 * x) + cos(3 * x) = 0 ,

cos(x * 2) * cos(x) * 2 + cos(x * 2) * 2 = 0,

cos(x * 2) * (cos(x) * 2 + 2) = 0 ,

```
cos(x * 2) = 0 #  cos(x) * 2 + 2 = 0

cos(x * 2) = 0

x =  (n1 * 360 +  90) * (1/2) # x = (n1 * 360 +  -90) * (1/2)

cos(x) * 2 + 2 = 0

x =  n2 * 360 + 180  #  x = n2 * 360 +  -180
```

Similar remarks apply to the Change of Unknown case (where the substitution must be explicitly shown, as must the substitution back) and disjunctions.

)efaults for Flags

lormally the worked example is not output at the beginning of the run. This :an be turned on by turning on the "output" flag, and turning the flag off igain turns off the output.

.P assumes that the user is at a CIFER terminal, use disable cifer to switch his flag off.

hen solving equations, LP uses a loop checker. The default value of the loop lag is "yes". If the program finds that it is trying to solve an equation hat it has met before (on the same run!), it will output a message and ask the ser what to do. This is not always due to an error, sometimes the same quation is arrived at by backtracking, but there is no real loop.

he flag can be set to "no", in which case no loop checking occurs, or it can e set to "warn", in which case if the looping condition occurs a warning essage is output, but the program continues.

hen the flag is set to yes and the user is prompted, the options are ontinuing, aborting, setting the flag to warn or no, or entering a break.

hen LP is given a worked example, it will sometimes produce a new method to xplain the example. This is only allowed if the "new_methods" flag is turned n, which is the default value. Turning the flag off also disables the orresponding method(NewMethod) flags, so that these new methods are not vailable to the program.

he default for other methods is that they are all enabled.

ther commands and flags.

ere are useful commands which apply to both worked examples and equations.

show rules" does the obvious thing, as does "remove rules". emove rule(Name)" remove the rule of that name, and "remove rules(List)". oes the same for a list of names.

show flags" shows the current setting of all the flags.

enable Method" and "disable Method" turn on and off the corresponding ethod(Method) flag. If the method(Method) flag is off, the method Method can ot be used either to solve equations or to process worked examples. If solve s being used, and a method needed in a schema is turned off, interesting

things can happen!

"show methods" lists all currently enabled methods.

"show rules_list" lists the names of the rules associated with the new methods.

"show new_methods" lists the new methods along with their "applicable next method." This is a method that should be applied after the new method.

The prefix command "writeout" is like show, but prompts for a file name and directs output to that file. writeout can be used with "rules", "schemas", and "solve".

disable, enable, remove, show and writeout can be shortened to their first letter, e.g. "s methods".

'give_example", mentioned above, allows the user to input a worked example. 'xredo" reruns this example, sredo reruns the last solve command.

'work_solution" uses the trace produced by the last solve command as a worked xample.

nerate_problem", mentioned above, generates a set of equations that are be relatives of the equation in the last schema produced by a worked xample. The user is asked if LP should attempt to solve one of these quations.

$purpose.

The program LP is intended to be part of a self improving algebra system. LP can solve equations and learn from worked examples. It can only deal with single equations in one unknown, i.e. no simultaneous equations or inequalities. It has two modes of operation, solve and work, where work is the worked example understander. See entries on solve and work for more information on these commands.

$source. The EXE file is DSKB:LP.EXE[400,444]. The files used are listed in ILUTI and FILIN, both in DSKB:[400,421,LP].

$solve. As in PRESS, solve(Eqn,X,Ans) is the basic solve procedure that olves Eqn for X to get answer Ans. The predicate solve(Eqn) can also be used, n this case the unknown is taken to be x. (The answer is printed out at the erminal so there is no need to have a variable to hold it, unless more rocessing is required.)

he operators allowed are *, +, -, /, ^ (exponentiation), & and #
disjunction). Any function can be used, the program knows about the six trig unctions and their inverses, the six hyperbolic functions and their inverses, nd the logarithmic function, log(X,Y) where X is the base, and Y the argument. xample equation is

2*cos(x)^2 - 3*sin(3*x) = 1.

Typing sredo reruns the last equation.

$work. The predicate work(Example,Unknown) is used to give the program a orked example. The example is presented in the form of a list, the lines of he example being separated by a comma. Unknown is the unknown in the example.

n example is

work([sin(x) + sin(3*x) + sin(5*x) = 0, 2*sin(3*x)*cos(x) + sin(3*x) = 0, in(3*x)*(2*cos(x) + 1) = 0,...etc]).

Some predefined examples can be loaded by typing "wep" to LP. (This requires he path learn: = [400,421,LP] to be defined, but this is the case for all ressGang.) The examples are trig1, trig2, trig3, and demo1. Prestored rules nd predicates can be loaded by typing "rules". Typing "examples" loads the mple, rule and predicate files.

work(Example) is a short form of work(Example,x).

See also entries on give_example and generate_problem.

$give_example. This predicate (with no arguments) can be used to give a orked example at the terminal. The program prompts for the example line by ine. The unknown in the example is always taken to be x.

Typing xredo reruns the last worked example input by give_example, old_redo emoves all the methods, schemas etc that have just been added, and reruns the xample again.

$generate_problem. This command is used to test the applicability of program enerated schemas. It should be used after a worked example has been rocessed. The program generates a choice of equations that are similar to the quation solved in the worked example. The user then chooses which of these

equations the program should attempt to solve.

$methods. There are 3 classes of methods: Top level, program created methods, and the lowest level.

The name of the each method is followed by the abbreviation that LP uses.

The top level methods are: Factorization (factorize), Disjunction (disjunct), Isolation (isolate).

When the program starts there are no program created methods.

The lowest level methods are:

Polynomial Methods, Change of Unknown, Apply New Rule
Prepare for Factorization , Collection, Function Stripping,
ttraction, Logarithmic Method and Nasty Function Method

$flags. Some functions of LP depend on the values of flags which can be ltered by the user. Flags are used to determine if:

the worked example should be output (the flag is called output, default is

program can create new methods, (new_methods flag, default yes),

perform loop checking (loop flag, three values, default yes)

the terminal is a cifer, (default yes)

In addition, flags are used to enable and disable equation solving methods.

The basic method of changing the value of flag Flag to yes and no is to type nable Flag or disable Flag respectively. The three valued loop flag, as values yes, no and warn. To change it to es or no the disable and enable methods can be used, but to change its value warn, type flag(loop,warn).

With the loop flag set to yes, if the program finds it has to solve an quation it has already met, it asks the user what to do. The options include ntinuing, aborting, or entering a break. A loop is not always caused by an ror, the program might reach an equation by backtracking, and might not be in oop at all. If the flag is set to warn, the loop checker prints out a rning message, but continues. With the flag set to no, no loop checking is rformed at all.

Trying to change a flag to a value it already has produces an error message.

All methods are originally allowed to be used. method Method can be disabled typing disable Method and enabled again by typing enable Method.

sable and enable can also take a list as argument.

ther_commands. The prefix operator remove is used to remove things from the tabase. 'remove rules ' removes all new rules. 'remove new_methods' and emove schemas' do the obvious things.

'remove rule(Name')' removes the rule of name Name (there should be only one.) emove rules(List)' recurses down list, calling 'remove rule(Name)' on each ne that is a member of List. (i.e. it is checklist('remove rule',List).

The prefix operator show displays to the terminal. 'show schemas' displays the schemas, 'show flags' shows the settings of the flags, and 'show methods' shows which methods are enabled. 'show solve' shows the steps on the solution path of the last equation to be solved. 'show new_methods' displays a list of the new methods and their applicable next methods. 'show rules_list' shows which rules are associated with the new methods.

writeout is another prefix operator. 'writeout rules' and 'writeout schemas' prompt for a file name and write to that file. 'writeout solve' is like 'show solve' but prompts for a file name and writes the trace to that file.

reset reinitializes loop checkers and difficulty markers, useful after aborting a run.

change_method_order allows the user to change the order in which methods are tried. Only the lowest level of methods (see entry on methods) can be changed, and all these methods must be specified. The program prints out the old order.

As mentioned above xredo reruns the previous example given by give_example, and redo reruns the last equation. work_solution uses the last solve trace as input for the worked example. generate_problem creates a new equation for the program to solve.

```
* FILIN :

                                                    Bernard Silver
                                                    Updated: 9 February  984

his file is loaded by the MIC file DSKB:LP.MIC[400,444].
: is read into DSKB:LPUTIL.EXE[400,444].  For the files
sed for LPUTIL.EXE see DSKB:FILUTI[400,421,LP].

ie paths packag, method etc are defined in DSKB:PRESS.PTH[400,444].

,


· compile([

ackag:match',          % Matcher
earn:portr',           % Portray Code
ethod:collec',         % Collection code
earn:weaknf',          % weak normal forms
earn:cond',            % precondition definitions
earn:char',            % Character Input
rn:simp.ax',           % Simplification axioms for tidy
earn:rew',             % Apply rewrite rules
earn:file',            % Check file names
earn:time',            % Print time of day
earn:comp',            % Stuff to compile
earn:polpak'           % Bag addition and multiplication of polynom als

  ]).


  [

earn:tl',              % Top level
earn:out',             % Output
earn:conj',            % conjecture new rules
earn:desc',            % description building
earn:loop',            % Loop Checker
earn:sol',             % Solve code
earn:imeth',           % basic interpreted method defns
earn:confir',          % confirming conjectures
earn:flag',            % Flag setting
earn:method',          % Description of PRESS methods
earn:specia',          % Special top level methods
earn:table',           % Tables and Conditions
earn:constr',          % Constraint prop
earn:func',            % Function Checking
earn:newmet',          % Conjecture new methods
earn:interp',          % Misc clauses from other PRESS files
earn:genprb',          % Problem Generator
earn:nasty',           % Nasty code
earn:axioms',          % Collection and Attraction axioms
earn:poly',            % Polynomial methods
earn:homog',
earn:log',             % Log Method
ackag:poltid'          % Identify and split polynomials


  :- core_image, writehead, reinitialise.
```

```
itehead :-
        ttynl,
        write('Learning PRESS  Mark 4 ('),
        version_date(Date),
        write(Date),
        write(')'),
        ttynl,
        ttynl,
        display('Type ** help. <CR> ** for help'),
        plsys(ppno(P)),
        plsys(date(X,Y,Z)),
        time(T),
        ttynl,
        writef('\n[User %t. Session begins at %w on %t.%t.%t]\n',[P, ,X,Y,Z]),
        ttynl.
```

```
/* FILUTI :

                                                    Bernard Silver
                                                    Updated: 20 December 1983


This file is used to create DSKB:LPUTIL.EXE[400,444], which is
used to form the program DSKB:LP.EXE[400,444]

This file is loaded by the MIC file DSKB:LPUTIL.MIC[400,444].

The util path is

util: = DSKB:[140,143,UTIL].


*/
:- [
                'util:util.ops',        % General operator declarations
                'util:arith.ops',       % Arithmetic operator declarations
                'learn:ops'             % Operators for LP
    ].


    compile([

'util:files.pl',        % Manipulate files
'util:writef.pl',       % Formatted write (writef)
'learn:routin.pl',      % List and routines
'util:flagro.pl',       % Flag handling
'util:struct.pl',       % Structure crunching
'util:long.pl',         % Rational arithmatic package
'util:tidy.pl'          % Expression tidy/evaluator
        ]).

:- [

'learn:misce.pl',       % Miscellaneous
'mec:edit.pl',          % Use TOP to edit
'util:type.pl',         % Type a file
'util:trysee.pl'        % Needed for above
].


core_image :- plsys(core_image).
```

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

        All files of Learning PRESS

  PREDICATE              FILE          CALLED BY


\=/2                     utility       subterms/3 conjecture_steps1/5
                                       find_both_sides_difference/8
                                       obtain_rule1/5 choice_tidy/2 method1/1
                                       find_step_cont/4 find_step_cont1/4
                                       read_conditions/1 read_unknown/1
                                       syntax_check/3

a_s_m1/2                 LEARN:NEWMET  already_suitable_method/2

absent/2                 METHOD:COLLEC exp_match1/5 absent/2

ac_decomp/4              PACKAG:MATCH  decomp/2 ac_decomp/4

ac_op/5                  PACKAG:MATCH  decomp/2 ac_decomp/4 recomp/2 ac_recomp/3
                                       match/2 template_match/3

ac_recomp/3              PACKAG:MATCH  recomp/2 ac_recomp/3

act_ans/8                LEARN:CONFIR  <user> confirm_conj/7

act_option/2             LEARN:CONFIR  <user> know_conditions1/1

action_after_prep_factors/5
                         LEARN:SOL     sort_solve_now/6

action_on_pick/8         LEARN:SOL     suitable_schema/7

action_on_pick1/8        LEARN:SOL     action_on_pick/8

action_on_pick2/6        LEARN:SOL     mapscore_compare/8

add_into_table/2         LEARN:TABLE   add_to_table/2

add_into_table1/3        LEARN:TABLE   add_into_table/2

add_new_method/2         LEARN:INTERP  create_method/13

add_poly/3               LEARN:POLPAK  poly/3 add_poly/3 times_poly/3

add_power/3              LEARN:POLPAK  map_add_power/3

add_to_table/2           LEARN:TABLE   add_to_table/2 <user> add_new_method/2

all_same_type/2          LEARN:GENPRB  problem1/3

all_same_type1/4         LEARN:GENPRB  all_same_type/2 all_same_type1/4
```

```
allowed_new_methods/0    LEARN:FLAG     find_new_method/4

alphanumeric_char/2      LEARN:FILE     alphanumeric_string/6

alphanumeric_string/6    LEARN:FILE     valid_filename/3 filename/3 extension/3
                                        alphanumeric_string/6

already_suitable_method/2
                         LEARN:NEWMET   cond_create/1

analyze/5                LEARN:DESC     construct_method/5

append/3                 utility        nasty_act/5 find_symbols1/4
                                        attract_list/3 strip/3 get_dist/4
                                        read_in_rule_conds_cont/2
                                        read_in_unknown/1

applicable/3             METHOD:COLLEC  old_attract/3 apply_new_rule3/7 collect/3

applicable_next_method/3
                         LEARN:NEWMET   create_method1/13

apply_new_rule1/5        LEARN:REW      method_transform/6 obtain_rule/6
                                        try_parallel/5 try_auto_method/4
                                        try_user_rule/4

apply_new_rule2/8        LEARN:REW      apply_new_rule1/5 try_use_rule/7

apply_new_rule3/7        LEARN:REW      apply_new_rule2/8 apply_new_rule3/7

apply_new_rule4/7        LEARN:REW      apply_new_rule2/8

apply_new_rule5/9        LEARN:REW      apply_new_rule4/7

arbint/1                 LEARN:COMP     isolax/3

assemble_preconds/2      LEARN:OUT      assemble_preconds/3 assemble_preconds/2

assemble_preconds/3      LEARN:OUT      o_s2/1

asserted/1               LMISC:XREF.INI
                                        obtain_rule/6 old_xredo/0

associative/1            LEARN:COMP     find_diff3/8 remove_match1/5
                                        try_use_rule/7

at_least_occ/3           LEARN:COMP     at_least_occ/3 least_dom/2

attract/3                LEARN:IMETH    recurse_attract/3 recurse_attract1/3
                                        try_attract/3

attract_check/3          LEARN:TL       try_attract/3

attract_check1/3         LEARN:TL       attract_check/3

attract_list/3           LEARN:NASTY    find_attract_list/4 attract_list/3

attrax/4                 LEARN:AXIOMS   old_attract/3

auto_next_applicable_method/5
```

	LEARN:SOL	method_transform/6
auto_rule/2	LMISC:XREF.INI	method_transform/6 <user> a_s_m1/2 consider_parallel/7 store_rules_list/2 s_method1/2 try_auto_method/4
bad_name/1	LEARN:SOL	<user> suitable_methods/9
base/2	LEARN:LOG	find_base/2 find_base/4
binary_to_list/5	LEARN:COMP	binary_to_list/5 least_dom/2
binomial/3	LEARN:POLPAK	poly/3 binomial/3
c_f_a/5	LEARN:FUNC	check_ok_args/4 c_f_a/5
c_m_o/2	LEARN:METHOD	change_method_order/0
c_ok_arg1/3	LEARN:FUNC	c_f_a/5
change_method_order/0	LEARN:METHOD	<user>
heck_cond/1	LEARN:INTERP	step_solve_eqn1/7 method_transform/6 possible_schema/5 satisfy_satisfied/2 old_attract/3 check_cond/1 problem/3 applicable_next_method/3 apply_new_rule1/5 apply_new_rule3/7 apply_new_rule4/7 apply_new_rule5/9 try_use_rule/7 find_step/7 find_step1/7
check_contains_solution/1	LEARN:TL	work_main/2
check_example/3	LEARN:CONFIR	work_main/2
check_example1/2	LEARN:CONFIR	check_example/3
check_example2/3	LEARN:CONFIR	<user> check_example1/2
check_expp/1	LEARN:COND	findtype/2 check_expp/1
heck_found_schema/3	LEARN:SOL	store_steps/4
check_free/2	LEARN:GENPRB	check_functor/2 check_free/2
check_functor/2	LEARN:GENPRB	all_same_type1/4 check_functor/2
check_logf/1	LEARN:COND	findtype/2 check_logf/1
check_name/1	LEARN:CONFIR	<user> obtain_the_rules1/10
check_no_conjectures/2	LEARN:CONJ	conjecture_the_steps/6
check_ok_args/4	LEARN:FUNC	check_ok_term/3
check_ok_set/3	LEARN:FUNC	find_functions1/3 check_ok_set/3 c_ok_arg1/3
heck_ok_term/3	LEARN:FUNC	check_ok_set/3

check_order/2	LEARN:METHOD	c_m_o/2
check_power_of/2	LEARN:LOG	find_base/2 check_power_of/2
check_preconds1/1	LEARN:NEWMET	test_preconds/5 check_preconds1/1
check_same_poly/3	LEARN:INTERP	check_same_poly/3 try_poly/3
check_the_preconds/5	LEARN:NEWMET	find_missing_preconds/7
check_tidy/2	LEARN:INTERP	method_transform/6
check_trigf/1	LEARN:COND	findtype/2 check_trigf/1
checkpt/1	LEARN:NASTY	attract_list/3 checkpt/1
choice_tidy/2	LEARN:REW	apply_new_rule3/7 apply_new_rule4/7 apply_new_rule5/9 try_use_rule/7
choice_tidy1/2	LEARN:REW	choice_tidy/2
choose_member_schema/3	LEARN:SOL	f_s1/6
chunk_part/4	LEARN:SPECIA	chunk_part/4 work1/6
clear/0	LEARN:OUT	initialize_screen/0
closeness/3	LEARN:COMP	closer/3 old_attract/3
closer/3	LEARN:COND	attract/3
collax/3	LEARN:AXIOMS	collect/3
collect/3	METHOD:COLLEC	recurse_collect/3 recurse_collect1/3 mod_collect/3 try_collect/3 collect/3
collect_check/3	LEARN:TL	try_collect/3
collect_check1/3	LEARN:TL	collect_check/3
com_ass_idn/2	LEARN:COMP	least_dom/2
command_error/2	LEARN:OUT	remove/1 show/1 writeout/1
comment_name/1	LEARN:OUT	construct_good_example_list/2
common_subterms/3	LEARN:COND	common_subterms/3 applicable_next_method/3
common_subterms_list/3	LEARN:COND	common_subterms/3 common_subterms_list/3
commutative/1	LEARN:COMP	<user>
compatible/2	METHOD:COLLEC	absent/2 list_compatible/2
concat/3	utility	assemble_preconds/2
cond_add_to_list/2	LEARN:CONFIR	store_rule/8

cond_assertz/1	LEARN:INTERP	add_new_method/2
cond_create/1	LEARN:NEWMET	create_method/13
confirm_conj/7	LEARN:CONFIR	conj_steps/4
conj_steps/4	LEARN:CONJ	conjecture_the_steps/6 conj_steps/4
conjecture_steps/7	LEARN:CONJ	work_main/2
conjecture_steps1/5	LEARN:CONJ	conj_steps/4
conjecture_the_steps/6	LEARN:CONJ	conjecture_steps/7
consider_isolation1/3	LEARN:TL	try_to_isolate/3
consider_parallel/7	LEARN:NEWMET	create_method1/13
construct_good_example_list/2	LEARN:OUT	report_steps/4 construct_good_example_list/2
construct_method/5	LEARN:DESC	work_main/2
contains/2	LEARN:COND	filter/4 mod_weak_normal_form2/3 is_product/2 is_disjunct/2 is_sum/2 dominated1/5 subterms/3 old_attract/3 at_least_occ/3 least_dom/4 make_unk_list1/5 problem1/3 get_new_args1/3 work1/6 collect/3 root_nasty/2 exp_nasty/2
contains_nasties/2	LEARN:COND	<user>
convert_case/2	LEARN:METHOD	short_name/2 convert_case/2
convert_chars/2	LEARN:CHAR	read_in_line1/2 read_name/1 mod_know_conditions/1
correspond/4	LEARN:COMP	correspond/4
corresponding_arguments/4	PACKAG:MATCH	old_attract/3 apply_new_rule3/7 corresponding_arguments/4 collect/3
create_method/13	LEARN:NEWMET	find_new_method/4
create_method1/13	LEARN:NEWMET	create_method/13
d/1	LEARN:FLAG	<user>
d_a_m/0	LEARN:FLAG	disable/1
d_a_m1/1	LEARN:FLAG	d_a_m/0
d_a_m2/1	LEARN:FLAG	d_a_m1/1 d_a_m2/1
decide_message/3	LEARN:SOL	method_transform/6
decomp/2	PACKAG:MATCH	weak_normal_form/3 prepfac_terms/6

		mult_decomp/2 common_subterms/3 <user> action_after_prep_factors/5 prep_factors/5 split_disjunct_solve/5 method_transform/6 old_attract/3 tidy_up_disjunction/2 remake_conjecture/4 find_diff1/7 tidy_up/3 problem1/3 return_new_term/2 all_same_type1/4 negative_number_product/2 apply_new_rule3/7 try_use_rule/7 work1/6 is_factorization/4 ortodot/2 try_prep_fact/3 match/2 collect/3 template_match/3 exp_distrib/2 multiply_through/4 prepd/2 mulbag_to_list/2
delete/3	LEARN:COMP	delete/3 tidy_up_disjunction/2 remove_table_entries/0
delete_member/3	LEARN:CONJ	remove_match/4 delete_member/3
delete_one/3	LEARN:COMP	prepfac_terms/6 map_delete_one/3 delete_one/3 match_conj/4 get_unknown_steps/2
elimiter/3	LEARN:FILE	alphanumeric_string/6
diff/2	utility	exp_nasty/2
diff_list/4	LEARN:CONJ	only_diff/3 diff_list/4
difficult_once/2	LMISC:XREF.INI	check_example/3 check_no_conjectures/2
dis_solution/2	LEARN:COND	dis_solution/2 sort_solve/5 sort_solve1a/6 prep_chunk/6 p_r1/11 chunk_part/4 work1/6
disable/1	LEARN:FLAG	d/1 disable/1 d_a_m2/1
disable_new_methods/0	LEARN:METHOD	flag_message/2
discriminant/4	LEARN:POLY	poly_method/4
isguised_linear/1	LEARN:POLY	poly_method/4
disjunct_solve/6	LEARN:SOL	method_transform/6 disjunct_solve/6
disjunct_writef/1	LEARN:OUT	f_s1/6
dist/2	LEARN:NASTY	nas_rule/3 multiply_through/4
dist1/2	LEARN:NASTY	dist/2 dist1/2
div_power/3	LEARN:POLPAK	map_div_power/3
dl_parse/3	LEARN:COND	parse/3 dl_parse/3
dl_parse4/4	LEARN:NASTY	parse4/4 dl_parse4/4
dl_parsef/3	LEARN:FUNC	function_parse/3 dl_parsef/3

dominatable_function/1	LEARN:COND	dominated1/5
dominated/3	LEARN:COND	<user> applicable_next_method/3
dominated1/5	LEARN:COND	dominated/3
domult/3	LEARN:NASTY	mult/3
domult/4	LEARN:NASTY	domult/3 domult/4
dottoand/2	LEARN:COMP	dottoand/2
e/1	LEARN:FLAG	<user>
e_a_m/0	LEARN:FLAG	enable/1
e_a_m1/1	LEARN:FLAG	e_a_m/0
e_a_m2/1	LEARN:FLAG	e_a_m1/1 e_a_m2/1
enable/1	LEARN:FLAG	e/1 enable/1 e_a_m2/1
able_new_methods/0	LEARN:METHOD	<user> flag_message/2
eval/1	utility	odd/1 even/1 negative/1 non_zero/1 <user> great_el/2 base/2 negative_number_product/2 match/2 poly/3 add_poly/3 isolax/3 root_nasty/2 exp_nasty/2 least/3 good_fun/1 get_nasty_type/3 expon_exp/3 expon_inv_exp/3 exp_member/4 warn_if_complex/1
eval/2	utility	gcd/3 inverse_number/3 non_neg/1 same_poly_fac/3 problem1/3 get_new_args/2 return_new_term/2 base/2 powered/3 match/2 poly/3 timesingl/4 binomial/3 get_nasty_type/3 neg_exp/3 domult/4 free_mult/3 remove_neg_powers/4 poly_method/4
ven/1	LEARN:COMP	isolax/3
examine_template/2	LEARN:REW	apply_new_rule1/5 try_use_rule/7
examples/0	LEARN:INTERP	<user>
excludes/2	LEARN:TABLE	m_s1/2
exp_distrib/2	LEARN:POLPAK	poly/3
exp_distrib_list/3	LEARN:POLPAK	exp_distrib/2 exp_distrib_list/3
exp_match/5	METHOD:COLLEC	template_match/3
exp_match1/5	METHOD:COLLEC	exp_match/5
exp_member/4	LEARN:NASTY	domult/4 exp_member/4

exp_nasty/2	LEARN:NASTY	nasty/2 exp_nasty_list/3
exp_nasty_list/3	LEARN:NASTY	<user> contains_nasties/2 try_nasty_method/3 exp_nasty_list/3
exp_term/2	LEARN:LOG	prod_exp_terms/2
expand_constraints/2	LEARN:SOL	suitable_step1/8
explain_g/1	LEARN:NEWMET	<user>
explain_g1/1	LEARN:NEWMET	expand_constraints/2 mod_check_cond1/1 explain_g/1
expon/2	LEARN:NASTY	find_symbols1/4
expon_exp/3	LEARN:NASTY	nas_rule/3
expon_exp1/3	LEARN:NASTY	expon_exp/3
expon_inv_exp/3	LEARN:NASTY	nas_rule/3
expon_inv_exp1/3	LEARN:NASTY	expon_inv_exp/3
xtension/3	LEARN:FILE	valid_filename/3 filename/3
extra_message/1	LEARN:OUT	mod_writef/2
extreme_term/3	LEARN:COMP	identical_subterms/3
extreme_term/5	LEARN:COMP	extreme_term/3 extreme_term/5
f_n1/4	LEARN:INTERP	<user> file_name_check/3
f_n2/3	LEARN:INTERP	<user> f_n1/4
f_s1/6	LEARN:SOL	factor_solve/5 f_s1/6
f_z/3	LEARN:TIME.	fill_zero/2
factor_part/3	LEARN:SPECIA	work1/6
factor_part1/3	LEARN:SPECIA	factor_part/3 factor_part1/3
factor_part2/4	LEARN:SPECIA	factor_part1/3 factor_part2/4
factor_solve/5	LEARN:SOL	action_after_prep_factors/5 split_disjunct_solve/5
file_consult/2	LEARN:CONFIR	act_option/2
file_exists/0	utility	file_name_check/3
file_exists/1	utility	<user> file_consult/2
file_name/2	LEARN:INTERP	f_n2/3 file_name1/2
file_name1/2	LEARN:INTERP	w_schemas/0 w_rules/0 w_solve_steps/1 w_n_m/1

file_name_check/3	LEARN:INTERP	file_name1/2
filename/3	LEARN:FILE	valid_filename/3
fill_in_values/5	LEARN:CONFIR	ob_condition/7
fill_in_values1/5	LEARN:CONFIR	fill_in_values/5
fill_zero/2	LEARN:TIME.	make_time/4
filter/4	LEARN:WEAKNF	weak_normal_form/3 filter/4
find1/2	LEARN:POLY	linear_method/3
find2/3	LEARN:POLY	find_coeffs/4
find_attract_list/4	LEARN:NASTY	nasty_act/5 find_attract_list/4
find_base/2	LEARN:LOG	log_reduce/4
find_base/4	LEARN:LOG	find_base/2 find_base/4
find_both_sides_difference/8	LEARN:CONJ	find_diff_hard/9
find_coeffs/4	LEARN:POLY	poly_method/4
find_diff/7	LEARN:CONJ	conjecture_steps1/5
find_diff1/7	LEARN:CONJ	find_diff/7 find_both_sides_difference/8
find_diff2/7	LEARN:CONJ	find_diff1/7 find_diff3/8
find_diff3/8	LEARN:CONJ	remake_conjecture/4 find_diff1/7
find_diff_hard/9	LEARN:CONJ	conjecture_steps1/5
find_first_hard_step/7	LEARN:NEWMET	find_new_method/4
find_first_hard_step/8	LEARN:NEWMET	find_first_hard_step/7 find_first_hard_step/8
find_first_part/11	LEARN:CONSTR	partition_the_list/8 find_first_part/11
find_functions/3	LEARN:FUNC	show_initialize_and_store/3 get_functions_from_step/2 work_main/2
find_functions1/3	LEARN:FUNC	find_functions/3 find_functions1/3
find_missing_preconds/7	LEARN:NEWMET	find_the_purpose1/6 find_new_method/4
find_missing_preconds1/7	LEARN:NEWMET	find_missing_preconds/7 find_missing_preconds1/7
find_new_method/4	LEARN:NEWMET	analyze/5 find_new_method/4
find_ok_preconds/6	LEARN:NEWMET	find_the_purpose1/6 find_new_method/4
find_ok_preconds1/7	LEARN:NEWMET	find_ok_preconds/6 find_ok_preconds1/7

find_parts_of_equation/5	LEARN:CONJ	conjecture_steps1/5
find_satisfy_unsat/2	LEARN:SOL	suitable_methods/9
find_satisfy_unsat1/2	LEARN:SOL	find_satisfy_unsat/2 find_satisfy_unsat1/2
find_step/7	LEARN:TL	work1/6
find_step1/7	LEARN:TL	find_step/7
find_step_cont/4	LEARN:TL	find_step/7
find_step_cont1/4	LEARN:TL	find_step1/7
find_sub_test/3	LEARN:SOL	mapscore_compare/8 find_sub_test/3
find_subset_same_type/3	LEARN:SOL	action_on_pick/8 find_subset_same_type/3
find_symbols/4	LEARN:NASTY	try_nasty_method/3 find_symbols/4
find_symbols1/4	LEARN:NASTY	find_symbols/4
find_the_purpose/6	LEARN:CONSTR	make_schema/5 find_the_purpose/6
find_the_purpose1/6	LEARN:CONSTR	find_the_purpose/6 find_the_purpose1/6
find_type/3	LEARN:COND	action_on_pick/8 find_subset_same_type/3
find_unknown_steps/4	LEARN:OUT	conjecture_steps/7
find_unknown_steps1/4	LEARN:OUT	find_unknown_steps/4
find_unknown_steps2/4	LEARN:OUT	find_unknown_steps1/4 find_unknown_steps2/4
findtype/2	LEARN:COND	find_type/3
first_method/4	LEARN:DESC	store_schema/4
first_subst/3	LEARN:CONJ	zap_stored_conj/5 first_subst/3
flag/3	utility	action_on_pick1/8 suitable_methods/9 looping/2 loop_action/1 store_rule/8 conjecture_steps1/5 store_schema/4 r_rules/0 still_rules/0 rules_stored/0 allowed_new_methods/0 nocifer/0 flag2/2 show_all_flags/1 loop_flag/1 text_not_used/0 add_new_method/2 known_method/8 disable_new_methods/0 enable_new_methods/0 remove_new_method_markers/0 already_suitable_method/2 show_the_example/3 rule_text1/0 rule_text2/0 rule_text2a/0 submethod_on/3
flag2/2	LEARN:FLAG	enable/1 disable/1

flag_error/1	LEARN:FLAG	flag2/2
flag_message/2	LEARN:FLAG	flag2/2
form_new_equation/2	LEARN:LOG	prod_exp_terms_eqn/3
form_time/2	LEARN:TIME	time/1
free_mult/3	LEARN:NASTY	multiply_through/4 free_mult/3
freeof/2	LEARN:COND	mod_weak_normal_form2/3 <user> contains/2 freeof/3 dis_solution/2 dominated1/5 dl_parse/3 sort_solve/5 sort_solve1a/6 make_unk_list1/5 dl_parsef/3 check_ok_term/3 c_ok_arg1/3 safe_divisor/2 check_functor/2 check_free/2 exp_term/2 prod_decomp/7 work1/6 is_poly/2 poly/3 dl_parse4/4
freeof/3	LEARN:COND	freeof/2 freeof/3
function_parse/3	LEARN:FUNC	find_functions1/3 c_ok_arg1/3
g1/1	LEARN:GENPRB	generate_problem/0
gcd/3	LEARN:COMP	gcd_poly/3
gcd_poly/3	LEARN:POLPAK	gcd_powers/2 gcd_poly/3
gcd_powers/2	LEARN:POLPAK	poly_hidden/3
generate_problem/0	LEARN:GENPRB	<user>
gensym/2	utility	arbint/1 identifier/1 mod_gensym/2
get_dist/4	LEARN:NASTY	prepd1/2 get_dist/4
get_explanation/2	LEARN:TABLE	output_preconds/1
get_functions_from_step/2	LEARN:CONFIR	obtain_the_rules1/10
get_lines/1	LEARN:TL	give_example/0 get_lines1/2
get_lines1/2	LEARN:TL	get_lines/1
get_method_list/2	LEARN:METHOD	known_method_interp/8
get_nasty_type/3	LEARN:NASTY	attract_list/3
get_new_args/2	LEARN:GENPRB	problem1/3 get_new_args/2
get_new_args1/3	LEARN:GENPRB	problem1/3 get_new_args1/3
get_next_element/3	LEARN:SOL	step_cycle/6
get_ops/3	LEARN:NASTY	posl/4
get_preconditions/4	LEARN:NEWMET	first_method/4 find_the_purpose1/6 find_new_method/4

get_problem/4	LEARN:GENPRB	<user> g_p1/1
get_problem1/3	LEARN:GENPRB	get_problem/4
get_problem2/2	LEARN:GENPRB	get_problem1/3
get_unknown_steps/2	LEARN:OUT	find_unknown_steps/4 get_unknown_steps/2
get_user_rule/8	LEARN:REW	apply_new_rule1/5
get_word/2	LEARN:FLAG	flag_message/2
give_example/0	LEARN:TL	<user>
give_help/1	MDHELP.PL	help1/1
go/0	LEARN:INTERP	<user>
good_flag/1	LEARN:SOL	check_found_schema/3
good_fun/1	LEARN:NASTY	nice_list/1
good_subterm/2	LEARN:COND	good_subterm/4 good_subterm/3
good_subterm/3	LEARN:COND	good_subterm/2 good_subterm/3
good_subterm/4	LEARN:COND	<user> identical_subterms/3
great_el/2	LEARN:INTERP	great_el/2
ground/1	LEARN:COMP	ground/2 match_check/2 <user> known_method_tl/8 known_method_interp/8 apply_new_rule1/5 ok_vars/2 apply_new_rule5/9
ground/2	LEARN:COMP	ground/1 ground/2
hard_combine/4	LEARN:IMETH	mod_collect/3 attract/3
hard_tidy_expr/2	LEARN:REW	choice_tidy/2 hard_tidy_expr/2 hard_tidy_expr/3
ırd_tidy_expr/3	LEARN:REW	hard_tidy_expr/2 hard_tidy_expr/3
help/0	LEARN:INTERP	<user>
help1/1	LEARN:INTERP	<user> help/0
ident_operators/2	METHOD:COLLEC	applicable/3
identical_subterms/3	LEARN:COND	prep_chunk/6 applicable_next_method/3
identifier/1	LEARN:COMP	sort_solve_now/6 try_chunk/5
ignore_term/1	LEARN:CONSTR	find_first_part/11 p_r1/11
initialize_screen/0	LEARN:OUT	solve/3 check_example2/3 work/2
insert_word/3	LEARN:COMP	scan_term/3 insert_word/3

integral/1	LMISC:XREF.INI	<user> really_new/2 subintegral/2
interpret_value/2	LEARN:FLAG	show_all_flags/1
inverse_number/3	LEARN:CONJ	shift_numbers/5
is_disjunct/2	LEARN:COND	<user>
is_factorization/4	LEARN:TL	work1/6
is_mod_poly/3	LEARN:COND	<user>
is_poly/2	LEARN:POLPAK	is_mod_poly/3 is_poly/2 simplify/2
is_product/2	LEARN:COND	<user>
is_rules/0	LEARN:FLAG	<user> remove/1 show/1
is_sum/2	LEARN:COND	<user>
_type_rule/6	LEARN:CONFIR	obtain_the_rules1/10
isolate/3	LEARN:IMETH	method_transform/6 try_to_isolate/3 consider_isolation1/3 try_function_stripping/4 try_isolate/3 isolate/4
isolate/4	LEARN:POLY	poly_method/4
isolate1/3	LEARN:IMETH	isolate/3 isolate1/3
isolax/3	LEARN:AXIOMS	isolate1/3
join/3	LEARN:TIME.	make_time/4
just_created/1	LMISC:XREF.INI	a_s_m1/2
k_functor/2	LEARN:FUNC	known_functor/2
ey_method/1	LEARN:METHOD	<user> step_solve_eqn1/7 assemble_preconds/3
know_conditions/1	LEARN:CONFIR	obtain_rule/6 know_conditions/1 mod_know_conditions/1
know_conditions1/1	LEARN:CONFIR	know_conditions/1 act_option/2 <user> file_consult/2
known_functor/2	LEARN:FUNC	check_ok_term/3
known_method/8	LEARN:METHOD	step_solve_eqn1/7 suitable_methods/9 conjecture_steps1/5 applicable_next_method/3 find_step/7 find_step1/7
known_method1/8	LEARN:METHOD	known_method/8

known_method2/8	LEARN:METHOD	known_method_interp/8
known_method_auto/7	LMISC:XREF.INI	auto_next_applicable_method/5 <user> remove/1 known_method1/8 cond_create/1 s_n_m1/1
known_method_interp/8	LEARN:METHOD	known_method1/8
known_method_schema/9	LMISC:XREF.INI	possible_schema/5 store_schema/4 schemas/0 generate_problem/0 problem/3 get_problem1/3 known_method/8 s_k_m_s/0
known_method_tl/8	LEARN:METHOD	known_method1/8
l_text/0	LEARN:LOOP	<user> looping_action/1
last/2	utility	attract_list/3 remove_neg_powers/4 poly_method/4
last_equation/1	LMISC:XREF.INI	<user>
last_example/2	LMISC:XREF.INI	xredo/0
least/3	LEARN:NASTY	find_base/4 member_match/3
least_dom/2	LEARN:COMP	old_attract/3 apply_new_rule3/7 collect/3
least_dom/4	LEARN:COMP	least_dom/2 least_dom/4
less_occ/3	LEARN:COND	<user>
linear/1	LEARN:POLY	poly_method/4
linear_method/3	LEARN:POLY	poly_method/4
list_compatible/2	METHOD:COLLEC	compatible/2 list_compatible/2
listtoset/2	utility	parse/3 tidy_up_disjunction/2 function_parse/3 remove_subsumed/2
log_reduce/4	LEARN:LOG	method_transform/6 remove_logs/4
log_separate/4	LEARN:LOG	log_reduce/4
log_separate/6	LEARN:LOG	log_separate/4
loop_action/1	LEARN:LOOP	<user> looping_action/1
loop_flag/1	LEARN:FLAG	sort_solve/5 sort_solve_one_step/5 get_problem/4
looping/2	LEARN:LOOP	sort_solve_one_step/5 sort_solve1/6
looping_action/1	LEARN:LOOP	<user> looping_check/3
looping_check/3	LEARN:LOOP	looping/2

m/1	LEARN:OUT	<user> w_n_m/1
m_s1/2	LEARN:SOL	maintain_satisfy/2 m_s1/2
m_t1/8	LEARN:SOL	method_transform/6
maintain_satisfy/2	LEARN:SOL	suitable_methods/9
make_arblist/2	LEARN:LOOP	remove_arbs1/3
make_arblist1/3	LEARN:LOOP	make_arblist/2 make_arblist1/3
make_name_atomic/2	LEARN:OUT	o_s2/1
make_poly/3	LEARN:POLPAK	simplify/3
make_reply_from_chars/4	LEARN:CHAR	read_in_rule_conds_cont/2
make_schema/5	LEARN:DESC	construct_method/5
make_subl/3	LEARN:INTERP	remove_arbs1/3 make_subl/3
make_time/4	LEARN:TIME.	form_time/2
make_unk_list/4	LEARN:CONJ	conjecture_steps/7
make_unk_list1/5	LEARN:CONJ	make_unk_list/4 make_unk_list1/5
manoeuvre_sides/3	LEARN:IMETH	isolate/3
map_add_power/3	LEARN:POLPAK	map_add_power/3 remove_neg_powers/4 poly_method/4
map_delete_one/3	LEARN:IMETH	prepfac_terms/6 prepfac_terms1/6 map_delete_one/3
map_div_power/3	LEARN:POLPAK	map_div_power/3 poly_method/4
map_match_member/3	LEARN:IMETH	prepfac_terms/6 prepfac_terms1/6 map_match_member/3
map_mult_decomp/2	LEARN:IMETH	prepfac_terms/6 map_mult_decomp/2
map_prettify/1	LEARN:NEWMET	pretty_print_conds/2 map_prettify/1
map_put/1	LEARN:CHAR	syntax_check/3 map_put/1 make_reply_from_chars/4 convert_chars/2
map_recomp/3	LEARN:IMETH	prepfac_terms/6 map_recomp/3
map_reify/3	LEARN:POLPAK	make_poly/3 map_reify/3
mapscore_compare/8	LEARN:SOL	action_on_pick1/8
maptidy_example/3	LEARN:INTERP	show_the_example/3 is_factorization/4
maptidy_example1/3	LEARN:INTERP	maptidy_example/3 maptidy_example1/3
mark_mod_asserta/1	LEARN:CONFIR	is_type_rule/6

match/2	PACKAG:MATCH	store_schema/4 match_check/2 merge_steps/3 apply_new_rule4/7 factor_part2/4 is_factorization/4 match/2 match_arguments/3 template_match/3 exp_match/5 exp_match1/5 rem_sub/3 member_match/3 expon_exp1/3 expon_inv_exp1/3 neg_exp/3 neg_exp_match/4 exp_member/4
match_arguments/3	PACKAG:MATCH	match/2 match_arguments/3
match_check/2	LEARN:INTERP	mod_weak_normal_form/3 show_initialize_and_store/3 match_member/3 obtain_rule1/5 match_conj/4 log_reduce/4 try_use_rule/7 choice_tidy1/2 work1/6 understood_constants/2 consider_isolation1/3 collect_check/3 collect_check1/3 try_attract/3 attract_check/3 attract_check1/3 try_function_stripping/4 try_prep_fact/3 poly_step/4 remove_nasty/3
match_conj/4	LEARN:DESC	process_method/4
match_member/3	LEARN:COMP	map_match_member/3 match_member/3 apply_new_rule5/9
member/2	utility	mod_weak_normal_form2/3 prepfac_terms/6 action_on_pick/8 action_on_pick1/8 action_on_pick2/6 find_satisfy_unsat1/2 m_s1/2 choose_member_schema/3 subterms/3 zap_stored_conj/5 add_into_table1/3 match_conj/4 remove_table_entries/0 sibling_term/2 sort_of_list/2 known_method_interp/8 get_unknown_steps/2 process_reply1/5 factor_part1/3 try_prep_fact/3 check_contains_solution/1
member_match/3	LEARN:NASTY	rem_sub/3 member_match/3
mberchk/2	utility	trigf/1
merge_steps/3	LEARN:INTERP	maptidy_example/3 merge_steps/3
method/1	LEARN:METHOD	perform_rename/5 enable/1 disable/1 flag_message/2 <user> really_new/2 disable_new_methods/0 enable_new_methods/0 method1/1 show/1 writeout/1
method1/1	LEARN:OUT	short_name/2 <user> method_list_on/2
method_list/2	LEARN:METHOD	get_method_list/2
method_list_on/2	LEARN:OUT	flag_message/2 e_a_m/0 d_a_m/0 show/1
method_transform/6	LEARN:SOL	step_solve_eqn1/7 applicable_next_method/3

might_satisfy/2	LEARN:TABLE	find_satisfy_unsat1/2
mod_ab/1	LEARN:INTERP	mod_abolish/2
mod_abolish/2	LEARN:INTERP	show_initialize_and_store/3 reset1/0
mod_assert/1	LEARN:INTERP	check_ok_term/3 a_s_m1/2 store_rules_list/2 find_step_cont/4 find_step_cont1/4
mod_asserta/1	LEARN:INTERP	sort_solve/5 sort_solve_main/8 mark_mod_asserta/1 add_into_table/2 add_into_table1/3 store_schema/4 cond_create/1 tell_create/1
mod_assertz/1	LEARN:INTERP	store_rule/8 find_step_cont1/4
mod_check_cond/3	LEARN:SOL	report_mod_check_cond/4
mod_check_cond1/1	LEARN:SOL	<user> mod_check_cond/3 mod_check_cond1/1
d_collect/3	LEARN:IMETH	method_transform/6 try_prep_fact/3
mod_gensym/2	LEARN:INTERP	store_schema/4 cond_create/1
mod_iso_trace/1	LEARN:OUT	<user>
mod_know_conditions/1	LEARN:CHAR	read_in_conditions/1
mod_list_listing/1	LEARN:OUT	mod_list_listing/1 w_rules/0
mod_list_new_rules/0	LEARN:OUT	w_rules/0
mod_listing/1	LEARN:OUT	mod_list_listing/1 m/1 w_schemas/0
mod_listing1/2	LEARN:OUT	mod_listing/1
mod_subst_mesg/3	LEARN:OUT	sort_solve_now/6 try_chunk/5
mod_weak_normal_form/3	LEARN:WEAKNF	sort_solve_one_step/5 show_initialize_and_store/3
mod_weak_normal_form1/4	LEARN:WEAKNF	mod_weak_normal_form/3 mod_weak_normal_form1/4 mod_weak_normal_form2/3 method_transform/6 factor_part1/3 try_to_isolate/3 try_function_stripping/4 remove_nasty/3
mod_weak_normal_form2/3	LEARN:WEAKNF	maptidy_example1/3
mod_writef/2	LEARN:OUT	solve/3
mulbag_to_list/2	LEARN:NASTY	domult/3
mult/3	LEARN:NASTY	multiply_through/4 mult/3
mult_decomp/2	LEARN:IMETH	map_mult_decomp/2

mult_occ/2	LEARN:COND	dominated/3 old_attract/3 apply_new_rule3/7 collect_check/3 collect_check1/3 attract_check/3 attract_check1/3 collect/3 simplify/2
multiply_through/4	LEARN:NASTY	try_nasty_method/3
must_satisfy/2	LEARN:TABLE	find_satisfy_unsat1/2 m_s1/2 add_into_table/2 remove_table_entries/0
nas_rule/3	LEARN:NASTY	nasty_act/5
nasty/2	LEARN:NASTY	subnasty/3
nasty_act/5	LEARN:NASTY	try_nasty_method/3
nasty_method/3	LEARN:NASTY	method_transform/6 remove_nasty/3
neg_exp/3	LEARN:NASTY	nas_rule/3 neg_exp/3
neg_exp_match/4	LEARN:NASTY	neg_exp/3
negative/1	LEARN:INTERP	non_neg/1 isolax/3
negative_number_product/2	LEARN:LOG	form_new_equation/2 <user>
new_functor/2	LMISC:XREF.INI	check_ok_term/3
new_rule_stored/1	LMISC:XREF.INI	is_rules/0 s_r_n1/0 s_r_n1a/0 mod_list_new_rules/0
newform/4	METHOD:COLLEC	old_attract/3 apply_new_rule3/7 collect/3
nice/1	LEARN:NASTY	nasty_act/5 nice/1
nice_list/1	LEARN:NASTY	nice/1 nice_list/1
nmember/3	utility	get_problem/4 create_method/13 find_unknown_steps2/4 nasty_act/5 posl/4
no_major_effect/1	LEARN:SOL	step_cycle1/8
no_such_flag/2	LEARN:FLAG	enable/1 disable/1
nocifer/0	LEARN:FLAG	clear/0
non_neg/1	LEARN:INTERP	isolax/3
non_zero/1	LEARN:INTERP	same_poly_fac/3 safe_divisor/2 isolax/3
normal/0	LEARN:INTERP	examples/0
normstore/3	LEARN:LOOP	looping_check/3
not/1	utility	step_solve_eqn1/7 possible_schema/5 suitable_methods/9 <user> mod_check_cond/3 obtain_rule/6 r_nm/0

		really_new/2 form_new_equation/2 known_method_tl/8 find_missing_preconds1/7 find_ok_preconds1/7 a_s_m1/2 consider_parallel/7 store_rules_list/2 isolax/3 root_nasty/2 good_fun/1
num_writef/2	LEARN:OUT	output_real_rule/1 mod_listing1/2 s_r_n/5 s_r_n1a/0
number/1	utility	good_subterm/2 dl_parse/3 position/3 term_size/2 scan_term/3 shift_numbers/5 negative/1 problem1/3 return_new_term/2 find_base/2 find_base/4 negative_number_product/2 powered/3 hard_tidy_expr/2 select_the_number/3 match/2 poly/3 <user> isolax/3 root_nasty/2 exp_nasty/2 expon/2 rem_sub/3 good_fun/1
numbers_between/3	LEARN:GENPRB	g_p1/1 numbers_between/3
s2/1	LEARN:OUT	output_schema1/1 o_s2/1
o_x1/0	LEARN:TL	old_xredo/0
ob_condition/7	LEARN:CONFIR	obtain_rule/6
obtain_rule/6	LEARN:CONFIR	act_ans/8
obtain_rule1/5	LEARN:CONJ	<user> obtain_rule/6
obtain_the_rules1/10	LEARN:CONFIR	ob_condition/7
occ/3	utility	mult_occ/2 single_occ/2 same_occ/3 less_occ/3 identical_subterms/3 good_subterm/4 test/3 fill_in_values/5 obtain_rule1/5 apply_new_rule1/5 apply_new_rule3/7 apply_new_rule4/7 apply_new_rule5/9 try_use_rule/7 consider_isolation1/3
:urs_in/2	utility	<user> s_r_n/5
odd/1	LEARN:COMP	isolax/3
ok_vars/2	LEARN:REW	apply_new_rule1/5
old_attract/3	LEARN:COMP	attract/3 old_attract/3
old_xredo/0	LEARN:TL	<user>
only_diff/3	LEARN:CONJ	find_diff3/8
ops_list/2	METHOD:COLLEC	ops_to_find/2 ops_list/2
ops_to_find/2	METHOD:COLLEC	exp_match1/5 ops_list/2
option_text/0	LEARN:OUT	<user> know_conditions1/1

ortodot/2	LEARN:TL	work1/6
output_diff/1	LEARN:OUT	find_diff/7 output_diff/1
output_example/1	LEARN:OUT	show_the_example/3 output_example/1
output_list_write/1	LEARN:OUT	write_bag/2 output_list_write/1
output_mess/1	LEARN:OUT	report_steps1/3
output_preconds/1	LEARN:NEWMET	explain_g1/1 output_preconds/1
output_real_rule/1	LEARN:CONJ	<user> obtain_rule1/5
output_schema/1	LEARN:OUT	s_k_m_s/0
output_schema1/1	LEARN:OUT	output_schema/1 output_schema1/1
output_solution/2	LEARN:OUT	report_steps1/3 output_solution/2
output_v_list/1	LEARN:OUT	output_valid_list/2 output_v_list/1
output_valid_list/2	LEARN:OUT	no_such_flag/2 command_error/2
p_r1/11	LEARN:CONSTR	find_first_part/11 partition_rest/8 p_r1/11
pairfrom/4	utility	try_use_rule/7
paraphrase_goal/3	LEARN:OUT	m_t1/8
parse/3	LEARN:COND	find_type/3
parse4/4	LEARN:NASTY	<user> contains_nasties/2 try_nasty_method/3
partition_rest/8	LEARN:CONSTR	partition_the_list/8 partition_rest/8
partition_the_list/8	LEARN:CONSTR	make_schema/5
perform_rename/5	LEARN:CONJ	zap_stored_conj/5
perm/2	utility	check_order/2
perm2/4	utility	template_match/3
pol_tidy/2	PACKAG:POLTID	poly_tidy/2 pol_tidy/2
poly/3	LEARN:POLPAK	poly_norm/3 poly/3
poly_hidden/3	LEARN:POLY	poly_method/4
poly_method/4	LEARN:POLY	poly_solve/4 poly_method/4
poly_norm/3	LEARN:POLPAK	try_poly/3 poly_solve/4 simplify/3
poly_solve/4	LEARN:POLY	method_transform/6 try_poly/3 poly_step/4 poly_solve/4
poly_step/4	LEARN:TL	try_poly/3

poly_tidy/2	PACKAG:POLTID	try_poly/3 poly_solve/4 simplify/3
position/3	LEARN:COMP	method_transform/6 position/4 try_to_isolate/3 consider_isolation1/3 nasty_act/5
position/4	LEARN:COMP	position/3 position/4
posl/4	LEARN:NASTY	find_symbols1/4 posl/4
possible_schema/5	LEARN:SOL	<user> suitable_schema/7
powered/3	LEARN:LOG	check_power_of/2
prep_chunk/6	LEARN:SOL	sort_solve_now/6 prep_chunk/6
prep_factors/5	LEARN:SOL	sort_solve_now/6 prep_factors/5
prepd/2	LEARN:NASTY	dist/2 re_dist/3
prepd1/2	LEARN:NASTY	prepd/2
epfac_terms/6	LEARN:IMETH	hard_combine/4
prepfac_terms1/6	LEARN:IMETH	prepfac_terms/6 prepfac_terms1/6
pretty_print_conds/2	LEARN:NEWMET	cond_create/1
principle_functor/2	LEARN:SOL	test/3
problem/3	LEARN:GENPRB	<user> g_p1/1
problem1/3	LEARN:GENPRB	problem/3 problem1/3
process_cm/5	LEARN:DESC	process_conjlist/3
process_conjlist/3	LEARN:DESC	construct_method/5 process_conjlist/3
process_diff/5	LEARN:OUT	conjecture_steps1/5 find_both_sides_difference/8 process_diff/5
process_method/4	LEARN:DESC	construct_method/5 process_method/4
process_reply/5	LEARN:OUT	looping_action/1 confirm_conj/7 know_conditions1/1 check_example1/2 help/0 file_name_check/3 f_n1/4 g_p1/1
process_reply1/5	LEARN:OUT	process_reply/5
prod_decomp/7	LEARN:LOG	log_separate/6 prod_decomp/7
prod_exp_terms/2	LEARN:LOG	prod_exp_terms_eqn/3 prod_exp_terms/2
prod_exp_terms_eqn/3	LEARN:LOG	<user>
pt/1	LEARN:NASTY	checkpt/1
quadratic/1	LEARN:POLY	poly_method/4

r/1	LEARN:FLAG	<user>
r_nm/0	LEARN:FLAG	<user> remove/1
r_rules/0	LEARN:FLAG	<user> remove/1
r_s/0	LEARN:FLAG	<user> remove/1
re_dist/3	LEARN:NASTY	prepd1/2
read_conditions/1	LEARN:CHAR	read_in_conditions/1
read_in_conditions/1	LEARN:CHAR	read_in_rule_conds_cont/2
read_in_line/1	LEARN:CHAR	get_lines/1
read_in_line1/2	LEARN:CHAR	read_in_line/1
read_in_rule/1	LEARN:CHAR	read_in_rule_conds/1
read_in_rule_conds/1	LEARN:CHAR	obtain_rule/6
ad_in_rule_conds_cont/2	LEARN:CHAR	read_in_rule_conds/1
read_in_unknown/1	LEARN:CHAR	read_in_rule_conds_cont/2
read_name/1	LEARN:CHAR	<user> obtain_the_rules1/10
read_rest/2	LEARN:CHAR	read_rule/1 read_in_line1/2
read_rest1/3	LEARN:CHAR	read_rest/2 read_rest1/3
read_rest_cond/2	LEARN:CHAR	read_conditions/1
read_rest_unk/2	LEARN:CHAR	read_unknown/1
read_rest_unk1/2	LEARN:CHAR	read_name/1 read_rest_unk/2 read_rest_cond/2 read_rest_unk1/2
read_rule/1	LEARN:CHAR	read_in_rule/1
read_unknown/1	LEARN:CHAR	read_in_unknown/1
really_new/2	LEARN:INTERP	mod_gensym/2
recomp/2	PACKAG:MATCH	prepfac_terms/6 map_recomp/3 prep_factors/5 method_transform/6 old_attract/3 tidy_up_disjunction/2 remove_match1/5 tidy_up/3 problem1/3 return_new_term/2 negative_number_product/2 apply_new_rule3/7 try_use_rule/7 is_factorization/4 <user> match/2 collect/3 newform/4 exp_distrib/2 make_poly/3 multiply_through/4 get_dist/4 domult/4
recurse_attract/3	LEARN:IMETH	method_transform/6

recurse_attract1/3	LEARN:IMETH	recurse_attract/3 recurse_attract1/3 attract_check/3 attract_check1/3
recurse_collect/3	LEARN:IMETH	method_transform/6
recurse_collect1/3	LEARN:IMETH	recurse_collect/3 recurse_collect1/3 collect_check/3 collect_check1/3
reify/3	LEARN:POLPAK	map_reify/3
rem_sub/3	LEARN:NASTY	remove_subsumed/2 rem_sub/3
remake_conjecture/4	LEARN:CONJ	conjecture_steps1/5
remove/1	LEARN:FLAG	r/1 remove_rules1/1
remove_arbs/2	LEARN:LOOP	normstore/3 try_to_isolate/3 consider_isolation1/3 try_function_stripping/4
remove_arbs1/3	LEARN:LOOP	remove_arbs/2
remove_logs/4	LEARN:TL	<user>
remove_match/4	LEARN:CONJ	find_diff3/8 remove_match/4
remove_match1/5	LEARN:CONJ	find_diff3/8
remove_nasty/3	LEARN:TL	<user>
remove_neg_powers/4	LEARN:POLY	poly_solve/4
remove_new_method_markers/0	LEARN:METHOD	r_nm/0
remove_rules1/1	LEARN:FLAG	<user> remove/1 remove_rules1/1
remove_safe_divisors/3	LEARN:INTERP	prep_factors/5 method_transform/6 remove_safe_divisors/3 is_factcrization/4
remove_subsumed/2	LEARN:NASTY	try_nasty_method/3
remove_table_entries/0	LEARN:FLAG	r_nm/0
report_mod_check_cond/4	LEARN:SOL	suitable_methods/9
report_steps/4	LEARN:OUT	work_main/2
report_steps1/3	LEARN:OUT	report_steps/4 report_steps1/3
reset/0	LEARN:FLAG	sort_solve_main/8 show_initialize_and_store/3 act_ans/8 check_example2/3 conjecture_the_steps/6 check_no_conjectures/2 reset1/0 generate_problem/0 get_problem2/2
reset1/0	LEARN:FLAG	work/2
return_new_term/2	LEARN:GENPRB	get_new_args1/3 sibling_term/2

rev/2	utility	construct_method/5 analyze/5
rhs_zero/1	LEARN:COND	<user>
root_nasty/2	LEARN:NASTY	nasty/2
roots/6	LEARN:POLY	poly_method/4
rule/4	LMISC:XREF.INI	<user> read_in_rule_conds_cont/2
rule_text1/0	LEARN:OUT	obtain_rule/6
rule_text2/0	LEARN:OUT	<user> read_in_conditions/1
rule_text2a/0	LEARN:OUT	<user> read_in_unknown/1
rule_text3/0	LEARN:OUT	<user> obtain_the_rules1/10
rules/0	LEARN:INTERP	normal/0
ules_stored/0	LEARN:FLAG	is_rules/0 <user> remove/1 writeout/1
s/1	LEARN:OUT	<user>
s_k_m_s/0	LEARN:OUT	<user> show/1
s_method/1	LEARN:OUT	<user> show/1
s_method1/2	LEARN:OUT	s_method/1 s_method1/2
s_n_m/1	LEARN:OUT	<user> show/1
s_n_m1/1	LEARN:OUT	s_n_m/1 s_n_m1/1
s_r_n/5	LEARN:OUT	s_rules_new1/0
s_r_n1/0	LEARN:OUT	s_rules_new/0
s_r_n1a/0	LEARN:OUT	s_r_n1/0
_rules_new/0	LEARN:OUT	<user> show/1
s_rules_new1/0	LEARN:OUT	s_rules_new/0
safe_divisor/2	LEARN:INTERP	remove_safe_divisors/3
same_methods/2	LEARN:DESC	<user> store_schema/4 same_methods/2
same_occ/3	LEARN:COND	<user>
same_poly_fac/3	LEARN:INTERP	check_same_poly/3
satisfy_satisfied/2	LEARN:SOL	choose_member_schema/3
scan_list/3	LEARN:COMP	scan_term/3 scan_list/3
scan_term/3	LEARN:COMP	wordsin/2 scan_list/3

schema/1	LMISC:XREF.INI	<user> step_solve_eqn1/7 bad_name/1 s_n_m1/1
schema_used/1	LMISC:XREF.INI	<user> possible_schema/5
schemas/0	LEARN:FLAG	<user> remove/1 r_nm/0 show/1 writeout/1
seen_eqn/1	LMISC:XREF.INI	<user> looping_check/3
select/3	utility	problem1/3 return_new_term/2 sibling/2 negative_number_product/2 select_the_number/3 template_match/3 exp_match1/5 list_compatible/2
select_letter/2	PACKAG:POLTID	simplify/2
select_the_number/3	LEARN:TL	work1/6
shift_numbers/5	LEARN:CONJ	find_diff3/8 shift_numbers/5 tidy_up/3
sort_name/2	LEARN:METHOD	enable/1 disable/1
show/1	LEARN:OUT	s/1
show_all_flags/1	LEARN:FLAG	show_all_flags/1 show/1
show_initialize_and_store/3	LEARN:SOL	solve/3
show_the_example/3	LEARN:OUT	work_main/2
sibling/2	LEARN:GENPRB	problem1/3 sibling_term/2
sibling_term/2	LEARN:GENPRB	sibling/2
simple/1	utility	freeof/2 ground/1
simple_rule/7	LEARN:REW	<user> get_user_rule/8
simplify/2	PACKAG:POLTID	solve/3 hard_tidy_expr/2 singleton_method/3 pol_tidy/2
simplify/3	PACKAG:POLTID	simplify/2
single_occ/2	LEARN:COND	applicable_next_method/3
single_plural/3	LEARN:OUT	expand_constraints/2 explain_g/1 pretty_print_conds/2 write_bag/2
singleton_method/3	LEARN:POLY	poly_method/4
sol_mess/1	LEARN:OUT	output_solution/2
solve/1	LEARN:SOL	sredo/0
solve/3	LEARN:SOL	solve/1

solve_steps/1	LEARN:OUT	<user> show/1 w_solve_steps/1
solve_steps1/1	LEARN:OUT	solve_steps/1 solve_steps1/1
sort_of_list/2	LEARN:CONSTR	partition_the_list/8
sort_solve/5	LEARN:SOL	solve/3 sort_solve_one_step/5 sort_solve_main/8 try_chunk/5 disjunct_solve/6 get_problem2/2
sort_solve1/6	LEARN:SOL	sort_solve_main/8 sort_solve_now/6 split_disjunct_solve/5 f_s1/6 get_problem1/3
sort_solve1a/6	LEARN:SOL	sort_solve1/6
sort_solve_main/8	LEARN:SOL	sort_solve/5
sort_solve_now/6	LEARN:SOL	sort_solve1a/6
sort_solve_one_step/5	LEARN:SOL	sort_solve/5
split_disjunct_solve/5	LEARN:SOL	sort_solve_now/6
split_two_ways/3	PACKAG:MATCH	match/2 split_two_ways/3
sredo/0	LEARN:INTERP	<user>
step_cycle/6	LEARN:SOL	sort_solve_now/6 prep_chunk/6 prep_factors/5 step_cycle1/8
step_cycle1/8	LEARN:SOL	step_cycle/6
step_solve_eqn/6	LEARN:SOL	sort_solve_one_step/5 m_t1/8 suitable_step1/8
step_solve_eqn1/7	LEARN:SOL	step_solve_eqn/6 m_t1/8 suitable_methods/9
still_rules/0	LEARN:FLAG	remove/1
store_rule/8	LEARN:CONFIR	<user> obtain_the_rules1/10
store_rules_list/2	LEARN:NEWMET	cond_add_to_list/2 cond_create/1
store_schema/4	LEARN:DESC	make_schema/5
store_steps/4	LEARN:SOL	solve/3
store_we/1	LMISC:XREF.INI	<user> show/1 writeout/1 work_solution/0
strip/3	LEARN:NASTY	nasty_act/5
strip_num/2	LEARN:COMP	wordsin/2 strip_num/2
subintegral/2	LEARN:NASTY	remove_arbs/2 subintegral/2
submethod_on/3	LEARN:OUT	method_list_on/2 submethod_on/3

subnasty/3	LEARN:NASTY	<user> contains_nasties/2 try_nasty_method/3 subnasty/3
subs1/3	LEARN:INTERP	remove_arbs1/3 subs1/3
subst/3	utility	subst_mesg/3 normstore/3 zap_stored_conj/5 subs1/3 create_method/13 apply_new_rule5/9 work1/6 nasty_act/5
subst_mesg/3	LEARN:COMP	mod_subst_mesg/3
subterms/3	LEARN:COMP	subterms/3 subterms2/3 <user> apply_new_rule4/7
subterms2/3	LEARN:COMP	common_subterms_list/3
suitable_methods/9	LEARN:SOL	suitable_step1/8
suitable_schema/7	LEARN:SOL	sort_solve/5
suitable_step/7	LEARN:SOL	step_cycle1/8
itable_step1/8	LEARN:SOL	suitable_step/7
syntax_check/1	LEARN:CHAR	read_in_rule/1 read_in_unknown/1 read_in_conditions/1 read_in_line1/2 read_name/1
syntax_check/3	LEARN:CHAR	syntax_check/1
take_logs_and_recomp/3	LEARN:LOG	log_reduce/4
tell_create/1	LEARN:NEWMET	cond_create/1
tell_reason/5	LEARN:NEWMET	find_new_method/4
tell_redo/2	LEARN:SOL	sort_solve_one_step/5
template_match/3	METHOD:COLLEC	applicable/3
rm_size/2	LEARN:COMP	extreme_term/3 extreme_term/5 term_size/4
term_size/4	LEARN:COMP	term_size/2 term_size/4
terminate_text/0	LEARN:CONFIR	check_example/3 check_example2/3
test/3	LEARN:SOL	mapscore_compare/8 find_sub_test/3
test_preconds/5	LEARN:NEWMET	check_the_preconds/5
text_not_used/0	LEARN:FLAG	obtain_rule1/5 reset/0
tidy/2	utility	zero_rhs/2 isolate/3 tidy_rhs/2 recurse_collect/3 recurse_collect1/3 recurse_attract/3 recurse_attract1/3 hard_combine/4 f_s1/6 method_transform/6 disjunct_solve/6 show_initialize_and_store/3 old_attract/3 subst_mesg/3 tidy_up_disjunction/2

		remove_arbs/2 remove_match1/5 obtain_rule1/5 negative/1 non_neg/1 non_zero/1 type_tidy/3 tidy_up/3 problem1/3 return_new_term/2 log_reduce/4 form_new_equation/2 negative_number_product/2 choice_tidy1/2 work1/6 consider_isolation1/3 try_collect/3 try_attract/3 try_prep_fact/3 poly_step/4 collect/3 add_poly/3 timesingl/4 reify/3 isolax/3 nasty_method/3 try_nasty_method/3 nasty_act/5 nas_rule/3 multiply_through/4 linear_method/3 discriminant/4 roots/6 pol_tidy/2
tidy_expr/2	utility	weak_normal_form/3 find_both_sides_difference/8 obtain_rule1/5 check_tidy/2 type_tidy/3 choice_tidy1/2 hard_tidy_expr/2 get_lines/1
tidy_ops/2	METHOD:COLLEC	exp_match1/5 tidy_ops/2
idy_rhs/2	LEARN:IMETH	isolate1/3 tidy_rhs/2
tidy_up/3	LEARN:INTERP	tidy_up/3 output_diff/1
tidy_up_disjunction/2	LEARN:COMP	solve/3 f_s1/6 disjunct_solve/6
tidy_withvars/2	utility	obtain_rule/6
time/1	LEARN:TIME.	initialize_screen/0
times_poly/3	LEARN:POLPAK	poly/3 times_poly/3 binomial/3
timesingl/4	LEARN:POLPAK	times_poly/3 timesingl/4
tlar/4	LEARN:LOG	take_logs_and_recomp/3 tlar/4
translate_difference/3	LEARN:NEWMET	tell_reason/5
ree/2	LEARN:GENPRB	sibling_term/2
tree_list/4	LEARN:COMP	wordsin/2 tree_list/4
tree_size/3	LEARN:COMP	closeness/3 tree_size/5
tree_size/5	LEARN:COMP	tree_size/3 tree_size/5
trigf/1	LEARN:COND	check_trigf/1
try_attract/3	LEARN:TL	<user>
try_auto_method/4	LEARN:TL	<user>
try_chunk/5	LEARN:SOL	method_transform/6
try_collect/3	LEARN:TL	<user>
try_function_stripping/4		

	LEARN:TL	<user>
try_isolate/3	LEARN:NASTY	nasty_act/5
try_nasty_method/3	LEARN:NASTY	nasty_method/3
try_parallel/5	LEARN:NEWMET	<user> consider_parallel/7
try_poly/3	LEARN:TL	<user>
try_prep_fact/3	LEARN:TL	<user>
try_to_isolate/3	LEARN:TL	<user>
try_use_rule/7	LEARN:REW	obtain_rule1/5 apply_new_rule1/5
try_user_rule/4	LEARN:TL	<user>
type_of_rhs/2	LEARN:SOL	test/3
type_tidy/3	LEARN:INTERP	mod_weak_normal_form1/4 remove_safe_divisors/3
derstood_constants/2	LEARN:TL	find_step/7
unknown/1	LMISC:XREF.INI	<user> really_new/2 show_the_example/3 factor_part1/3
user_macro/3	LEARN:OUT	remove/1 show/1 writeout/1
user_rule/7	LMISC:XREF.INI	check_name/1 still_rules/0 get_preconditions/4 <user> get_user_rule/8 s_rules_new1/0
valid_commands/2	LEARN:OUT	no_such_flag/2 command_error/2
valid_filename/2	LEARN:FILE	file_name/2
valid_filename/3	LEARN:FILE	valid_filename/2
riable_argument/1	LEARN:FLAG	remove/1 enable/1 disable/1 show/1 writeout/1
w/1	LEARN:OUT	<user>
w_n_m/1	LEARN:OUT	<user> writeout/1
w_rules/0	LEARN:OUT	<user> writeout/1
w_schemas/0	LEARN:OUT	<user> writeout/1
w_solve_steps/1	LEARN:OUT	<user> writeout/1
warn_if_complex/1	LEARN:POLY	roots/6
warn_if_possible_missing_rule/3	LEARN:OUT	report_steps1/3

weak_normal_form/3	LEARN:WEAKNF	mod_weak_normal_form1/4 is_mod_poly/3 method_transform/6 multiply_through/4
wep/0	LEARN:INTERP	normal/0
wordsin/2	LEARN:COMP	mod_weak_normal_form2/3 remove_arbs/2 factor_part1/3 neg_exp/3 simplify/2
work/1	LEARN:TL	<user>
work/2	LEARN:TL	work/1 give_example/0 xredo/0
work1/6	LEARN:TL	work2/6 work2r/6 work3/10 work_main/2 work1/6
work2/6	LEARN:SPECIA	work1/6
work2r/6	LEARN:SPECIA	work2/6 work2r/6
work3/10	LEARN:SPECIA	work1/6
work_main/2	LEARN:TL	check_example2/3 work/2
rk_solution/0	LEARN:TL	<user>
write_bag/2	LEARN:OUT	show/1
write_horiz_list/1	LEARN:OUT	o_s2/1 write_horiz_list/1
write_set/1	LEARN:GENPRB	g_p1/1
write_set1/2	LEARN:GENPRB	write_set/1 write_set1/2
write_step/1	LEARN:OUT	conj_steps/4
write_type/1	LEARN:OUT	is_type_rule/6 cond_add_to_list/2
writef/1	utility	sort_solve1a/6 sort_solve_now/6 <user> action_after_prep_factors/5 prep_chunk/6 split_disjunct_solve/5 f_s1/6 step_cycle1/8 method_transform/6 check_found_schema/3 expand_constraints/2 suitable_methods/9 report_mod_check_cond/4 mod_check_cond1/1 looping_action/1 l_text/0 loop_action/1 confirm_conj/7 act_ans/8 ob_condition/7 check_name/1 act_option/2 terminate_text/0 check_example1/2 check_example2/3 conjecture_steps1/5 obtain_rule1/5 check_no_conjectures/2 store_schema/4 remove/1 r_rules/0 r_s/0 r_nm/0 remove_rules1/1 still_rules/0 flag_message/2 e_a_m1/1 e_a_m2/1 d_a_m1/1 d_a_m2/1 help/0 help1/1 sredo/0 go/0 file_name/2 f_n1/4 f_n2/3 generate_problem/0 g_p1/1 write_set/1 write_set1/2 get_problem/4 get_problem1/3 get_problem2/2 check_order/2 tell_reason/5 explain_g1/1 create_method/13 create_method1/13

		report_steps/4 report_steps1/3 sol_mess/1 output_mess/1 output_example/1 find_unknown_steps1/4 extra_message/1 mod_iso_trace/1 rule_text1/0 rule_text2/0 rule_text2a/0 rule_text3/0 option_text/0 solve_steps/1 solve_steps1/1 s_r_n1/0 show/1 write_bag/2 output_schema/1 output_schema1/1 process_reply1/5 factor_part2/4 chunk_part/4 work1/6 give_example/0 get_lines1/2 xredo/0 old_xredo/0 work_solution/0 check_contains_solution/1 warn_if_complex/1 read_in_rule_conds_cont/2 read_in_rule/1 read_in_unknown/1 read_in_conditions/1 read_conditions/1 read_in_line1/2 read_unknown/1
writef/2	utility	mod_weak_normal_form/3 solve/3 sort_solve/5 tell_redo/2 sort_solve_main/8 <user> action_after_prep_factors/5 m_t1/8 disjunct_solve/6 show_initialize_and_store/3 suitable_step/7 suitable_step1/8 expand_constraints/2 report_mod_check_cond/4 obtain_rule/6 check_name/1 know_conditions1/1 file_consult/2 conjecture_the_steps/6 conj_steps/4 conjecture_steps1/5 find_diff2/7 find_both_sides_difference/8 obtain_rule1/5 store_schema/4 remove/1 flag_message/2 interpret_value/2 flag_error/1 no_such_flag/2 variable_argument/1 check_ok_term/3 file_name/2 file_name_check/3 f_n1/4 maptidy_example1/3 merge_steps/3 g_p1/1 write_set1/2 get_problem/4 get_method_list/2 c_m_o/2 tell_reason/5 explain_g/1 output_preconds/1 cond_create/1 tell_create/1 already_suitable_method/2 a_s_m1/2 pretty_print_conds/2 map_prettify/1 report_steps/4 report_steps1/3 output_mess/1 show_the_example/3 output_example/1 write_step/1 output_diff/1 write_type/1 num_writef/2 paraphrase_goal/3 disjunct_writef/1 mod_writef/2 mod_iso_trace/1 solve_steps1/1 write_bag/2 output_list_write/1 s_method/1 s_method1/2 s_n_m/1 s_n_m1/1 s_k_m_s/0 o_s2/1 write_horiz_list/1 w_schemas/0 w_rules/0 mod_list_new_rules/0 w_solve_steps/1 user_macro/3 command_error/2 output_valid_list/2 output_v_list/1 process_reply1/5 mod_subst_mesg/3 initialize_screen/0 work_main/2 get_lines/1 read_in_rule_conds_cont/2

writeout/1	LEARN:OUT	w/1
xredo/0	LEARN:TL	old_xredo/0 o_x1/0
z_norm/2	LEARN:POLPAK	poly_norm/3 z_norm/2 poly_tidy/2
zap_stored_conj/5	LEARN:CONJ	conjecture_the_steps/6 zap_stored_conj/5
zero_rhs/2	LEARN:WEAKNF	weak_normal_form/3

```
/* SOL :

                                                    Bernard Silver
                                                    Updated: 27 February 1984
*/

 % Solve new equations
solve(Eqn) :- solve(Eqn,x,_).

solve(Eqn,X,Ans) :-
        initialize_screen,
        statistics(runtime,_),
        show_initialize_and_store(X,Eqn,Eqn1),
        sort_solve(Eqn1,X,Ans1,Flag,EqnSteps-[]),
        tidy_up_disjunction(Ans1,Ans2),
        simplify(Ans2,Ans),
        mod_writef('\nAnswer is : %t\n',[Ans]),
        store_steps(Eqn1,X,Flag,EqnSteps),
        statistics(runtime,[_,Time]),
        writef('\n[Problem took %t milliseconds]\n',[Time]),
        !.

solve(E,_,_) :-
        statistics(runtime,[_,T]),
        writef('\nCouldn''t solve the equation\n%t.\n
[The attempt took %t milliseconds]\n',[E,T]).

 % Look for and use suitable schema
sort_solve(Eqn,X,Eqn,win,E-E) :-
        freeof(X,Eqn),
        !,
        writef('\n%t does not contain the unknown %t.\n',[Eqn,X]).

sort_solve(Eqn,X,Eqn,win,T-T) :-
        dis_solution(Eqn,X),
        !.

sort_solve(Eqn,X,Ans,new_flag(Flag,Flag1),EqnSteps) :-
        suitable_schema(Eqn,X,List,Eqn1,Type,Flag,Name),
        !,
        loop_flag(warn1),
        mod_asserta(schema_used(Name)),
writef('\nAttempting to use schema method %t,
enerated for equation\n\n%t.\n',[Name,Eqn1]),
        sort_solve_main(Eqn,X,Ans,List,Type,Flag1,EqnSteps,Name).

sort_solve(Eqn,X,Ans,F,Steps) :-
        sort_solve_one_step(Eqn,X,Ans,F,Steps).

 % Do one step
sort_solve_one_step(Eqn,X,Ans,F,T-T1) :-
        loop_flag(yes),          % Switch on loop check
        mod_weak_normal_form(Eqn,X,Eqn1),
        looping(Eqn1,X),
        !,
        step_solve_eqn(no,Eqn1,X,_,New1,T-T2),
        tell_redo(New1,Eqn1),
        sort_solve(New1,X,Ans,F,T2-T1).

 % Tell if fail to solve new equation
```

```
tell_redo(_,_).
tell_redo(New,Old) :-
        writef('\nFailed to solve equation %t.\n\nBack to %t.\n',[New,Old]),
        fail.

 % Main schema loop
sort_solve_main(Eqn,X,Ans,List,Type,Flag,EqnSteps,Name) :-
        sort_solve1(Eqn,X,Ans,List,Type,EqnSteps),
        (retract(schema_abandon(Name)) -> Flag = bad(Name);Flag = good(Name)),
        !.

 % Start again without the schema
sort_solve_main(Eqn,X,Ans,_,_,bad,EqnSteps,Name) :-
 writef('\nCan''t use schema %t\n\n',[Name]),
        reset,
        mod_asserta(schema_abandon(Name)),
        sort_solve(Eqn,X,Ans,_,EqnSteps).

sort_solve1(Eqn,X,Ans,List,Type,EqnSteps) :-
        looping(Eqn,X),
        sort_solve1a(Eqn,X,Ans,List,Type,EqnSteps),
        !.

 % Equation is solved
  rt_solve1a(Eqn,X,Eqn,_,_,T-T) :-
        dis_solution(Eqn,X),
        !,
        writef('\n[End of solution]\n').

sort_solve1a(Eqn,X,Eqn,_,_,T-T) :-
        freeof(X,Eqn),
        !,
        writef('Equation doesn''t contain the unknown\n'),
        writef('\n[End of solution]\n').

 % Try to apply schema at once
sort_solve1a(Eqn,X,Ans,List,Type,EqnSteps) :-
        sort_solve_now(Eqn,X,Ans,List,Type,EqnSteps),
        !.


sort_solve_now(Eqn,X,Ans,[L1,L2|L3],'Change of Unknown',T-T1) :- !,
        writef('\nAttempting to change the unknown.\n'),
        prep_chunk(Term,Eqn,X,New,L1,T-[NewVar=Term,CVE|T2]),
        identifier(NewVar),
        mod_subst_mesg(Term=NewVar,New,CVE),
        writef('\nTrying to solve the changed variable equation.\n'),
        sort_solve1(CVE,NewVar,NewEqn,L2,'General',T2-[SE|T3]),
        mod_subst_mesg(NewVar=Term,NewEqn,SE),
        writef('\nTrying to solve the substitution equation.\n'),
        split_disjunct_solve(SE,X,Ans,L3,T3-T1),
        !.


sort_solve_now(Eqn,X,Ans,[L1|F],'Factorization',T-T1) :- !,
        writef('\nAttempting to manipulate equations into factors.\n'),
        prep_factors(Eqn,X,New,L1,T-T2),
        action_after_prep_factors(New,X,Ans,F,T2-T1),
        !.
```

```
action_after_prep_factors(New,X,Ans,F,T2-T1) :-
        (decomp(New,[#|_]) ->
        writef('\nFactorizing equation to obtain\n\n%t\n',[New]),
        writef('\nTrying to solve the factors.\n');true),
        factor_solve(New,X,Ans,F,T2-T1),
        !.

sort_solve_now(Eqn,X,Ans,List,Type,T-T1) :-
        step_cycle(Eqn,X,New,List,Rest,T-T2),
        !,
        sort_solve1(New,X,Ans,Rest,Type,T2-T1).


 % Do the initial part of chunk schema
prep_chunk(_,Eqn,X,Eqn,_,T-T) :-
        dis_solution(Eqn,X),
        !,
        writef('\nThe equation is solved.\n').

prep_chunk(Term,Eqn,X,Eqn,_,T-T) :-
        identical_subterms(Eqn,X,Term),
        !.

prep_chunk(Term,Eqn,X,New,List,T-T1) :-
        step_cycle(Eqn,X,Eqn1,List,Rest,T-T2),
        prep_chunk(Term,Eqn1,X,New,Rest,T2-T1),
        !.

 % Do initial part of factor schema
prep_factors(A*B=0,X,New,_,[New1|T]-T) :-
        decomp(A*B,[*|List]),
        remove_safe_divisors(X,List,New),
        length(New,Length),
        Length > 1,
        recomp(New1,[#|New]),
        !.

prep_factors(Eqn,X,New,List,T-T1) :-
        step_cycle(Eqn,X,New1,List,Rest,T-T2),
        prep_factors(New1,X,New,Rest,T2-T1).


 % Split the Change of Unknown case
split_disjunct_solve(A#B,X,Ans,L,Steps) :- !,
        decomp(A#B,[#|List]),
        writef('\nSolving disjuncts.\n'),
        factor_solve(List,X,Ans,L,Steps).

split_disjunct_solve(Eqn,X,Ans,L,Steps) :-
        sort_solve1(Eqn,X,Ans,L,other,Steps).

 % Rest of factor schema
factor_solve(New,X,Ans,Schema,EqnSteps) :-
        f_s1(New,X,Ans,Schema,false,EqnSteps).

f_s1([],_,Ans,_,Acc,S-S) :- !,tidy_up_disjunction(Acc,Ans).
f_s1([H|List],X,Ans,Schema,Acc,[H|T]-T1) :-
        choose_member_schema(H,S,Schema),
```

```
        disjunct_writef([H,X]),
        writef('\nChoosing schema to solve this factor.\n'),
        sort_solve1(H,X,Ans1,S,other,T-T2),
        !,
        tidy(Acc#Ans1,NewAcc),
        f_s1(List,X,Ans,Schema,NewAcc,T2-T1).

step_cycle(Eqn,X,New,List,Rest,Steps) :-
        get_next_element(List,Step,Rest1),
        step_cycle1(Eqn,X,New,List,Rest,Step,Rest1,Steps).

step_cycle1(Eqn,X,New,List,Rest,Step,Rest1,Steps) :-
        suitable_step(X,Eqn,Step,_,New,Steps,Flag),
        (Flag = win -> Rest = Rest1; Rest = List),
        !.

 % Miss out steps if possible
step_cycle1(Eqn,X,New,_,Rest,Step,Rest1,Steps) :-
        no_major_effect(Step),
        writef('\nAttempting to omit step.\n'),
        step_cycle(Eqn,X,New,Rest1,Rest,Steps).

 % Perform one step, using method Name to give New1
 ep_solve_eqn(Flag,Eqn,X,Name,New,EqnSteps) :-
        step_solve_eqn1(Flag,Eqn,X,Name,New,EqnSteps,Mess),
        call(Mess),
        !.

step_solve_eqn1(Flag,Eqn,X,Name,New,EqnSteps,Mess) :-
        known_method(X,Eqn,New,Name,Where,Call,Precond,Post),
        not schema(Name),
        check_cond(Precond),
        (key_method(Name)->!;true),
        method_transform(Flag,Call,Where,Mess,EqnSteps,v),
        ((Flag=no, Name = method(_));check_cond(Post)),
        !.


 % Using the methods from step_solve_eqn

method_transform(_,try_to_isolate(X,Old,New),all,Mess,[New|S]-S,F) :-
        position(X,Old,Posn),
        isolate(Posn,Old,New),
        tidy(New,New1),
        !,
        (F= gag -> Mess=true;Mess = mod_iso_trace(New1)).

method_transform(Type,try_disjunct(X,A#B,Ans),all,true,[A#B|E]-E1,F) :-
        decide_message(Type,F,NewType),
        (NewType=no;
        writef('\nSplitting into disjuncts and solving each in turn.\n')),
        decomp(A#B,[#|List]),
        disjunct_solve(NewType,List,X,Ans,false,E-E1),
        !.

method_transform(Type,try_factorize(X,A*B=0,Ans),all,true,Z-E1,F) :-
        decide_message(Type,F,NewType),
        (NewType=no;
        decomp(A*B,[*|List]),
        remove_safe_divisors(X,List,New),
```

```
        recomp(Disj,[#|New]),
        (length(New,1) -> Z=E;
        Z=[Disj|E],
        writef('\nSplitting into factors and solving each in turn.\n\n'))),
        disjunct_solve(NewType,New,X,Ans,false,E-E1).

 % New Methods without schemas (Messy!!)
method_transform(no,try_auto_method(Method,X,Patt,New),all,true,
[Mid|S]-S1,F) :-
        auto_rule(Method,Name),
        apply_new_rule1(X,Patt,Mid,Name,sol),
        auto_next_applicable_method(Method,NewMethod,X,Mid,Rest),
        check_cond([applicable_next_method(NewMethod,X,Mid)|Rest]),
        m_t1(X,Mid,Name,Method,NewMethod,New,S-S1,F).

m_t1(X,New1,_,_,NewMethod,New,S-S1,gag) :-
        step_solve_eqn1(no,New1,X,NewMethod,New,S-S1,_),
        !.

m_t1(X,New1,Name,Method,NewMethod,New,S-S1,v) :-
writef('\nUsing rule %t to apply new method,\n%t, to obtain\n\n%t.\n',
[Name,Method,New1]),
        paraphrase_goal('Trying',NewMethod,Method),
        (step_solve_eqn(no,New1,X,NewMethod,New,S-S1);
        paraphrase_goal('Failed',NewMethod,Method),fail),
        !.

 % New methods using schemas
method_transform(yes,try_auto_method(Method,X,Patt,New),all,
writef('\nUsing rule %t to apply new method,\n%t, to obtain\n\n%t.\n',
[Name,Method,New]),[New|S]-S,_) :-
        auto_rule(Method,Name),
        apply_new_rule1(X,Patt,New,Name,sol).

method_transform(_,try_prep_fact(X,A+B=0,C*D=0),all,
writef('\nPreparing for Factorization to obtain \n\n%t.\n',[C*D=0]),
[C*D=0|S]-S,_) :-
        mod_collect(X,A+B,New),
        check_tidy(New,C*D=0),
        !.

method_transform(_,try_function_stripping(X,Old,Posn,New),all,
  writef('\nApplying Function Stripping to obtain\n\n%t.\n',[New]),[New|S]-S,_)
        :-
        isolate(Posn,Old,New1),
        tidy(New1,New2),
        mod_weak_normal_form1(New2,expr,X,New),
        !.

method_transform(_,try_collect(X,Old=B,New),part,
writef('\nCollecting Equation to obtain \n\n%t.\n',[New]),[New|S]-S,_) :-
        recurse_collect(X,Old,New1),
        tidy(New1=B,New),
        !.


method_transform(_,try_attract(X,Old=B,New),part,
writef('\nAttracting Equation to obtain \n\n%t.\n',[New]),[New|S]-S,_) :-
        recurse_attract(X,Old,New1),
        tidy(New1=B,New),
```

```
        !.

method_transform(_,try_poly(X,Old,New),all,
writef('\nUsing Polynomial methods to obtain\n\n%t.\n',[New]),[New|S]-S,_) :-
        !,  % Nothing else should be tried
        poly_solve(Old,X,Ans,_),
        tidy(Ans,New).



method_transform(_,try_chunk(X,Old,New,Term),all,true,S-S1,_) :-
        try_chunk(X,Old,New,Term,S-S1),
        !.


 % Log methods
method_transform(_,remove_logs(X,New,Mid,Base),all,
writef('\nApplying Log method, base %t, to obtain\n\n%t.\n',[Base,New]),
        [New|S]-S,_) :-
        log_reduce(Mid,X,Base,New1),
        weak_normal_form(New1,X,New),
        !.

  thod_transform(_,remove_nasty(X,Old,New),all,
writef('\nApplying Nasty method to obtain\n\n%t.\n',[New]),[New|S]-S,_) :-
        nasty_method(Old,X,New1),
        weak_normal_form(New1,X,New),
        !.

 % Change of Unknown

try_chunk(X,Old,Ans,Term,[Var=Term,NewEqn|S]-S1) :-
        !,
        identifier(Var),
        mod_subst_mesg(Term=Var, Old, NewEqn),
        sort_solve(NewEqn,Var,NewAns,_,S-[XAns|S2]),
        mod_subst_mesg(Var=Term, NewAns, XAns),
        sort_solve(XAns,X,Ans,_,S2-S1).

 % Do disjunctions properly
disjunct_solve(_,[],_,Ans,Acc,S-S) :- !,tidy_up_disjunction(Acc,Ans).
 .sjunct_solve(Type,[A|B],X,Ans,Acc,[A|S]-S1) :-
        (Type=no;
        writef('\nSolving factor %t.\n',[A])),
        sort_solve(A,X,Ans1,_,S-S2),
        tidy(Acc#Ans1,NewAcc),
        disjunct_solve(Type,B,X,Ans,NewAcc,S2-S1).


suitable_schema(Eqn,X,Schema,Eqn1,Type,Flag,Name) :-
        setof(p_s(Schema1,Name1,Type1),
                possible_schema(X,Eqn,Schema1,Name1,Type1),Bag),
        action_on_pick(Bag,X,Eqn,Schema,Name,Type,Flag,Eqn1),
        !.

 % Schema method is possible if its preconds are satisfied
possible_schema(X,Eqn,Schema,Name,Type) :-
        known_method_schema(X,Eqn,_,Name,Schema,Type,_,Pre,_),
        not schema_used(Name),
```

```
        check_cond(Pre).


 % Only one schema
action_on_pick([p_s(schema(S,Eqn1,_),N,T)],_,_,S,N,T,only(N),Eqn1) :- !.


 % First tie_break (equation same type as generator)
action_on_pick(Bag,X,Eqn,S,N,T,Flag,Eqn1) :-
        find_type(X,Eqn,EqnType),
        find_subset_same_type(Bag,EqnType,SubBag),
        SubBag \== [],
        !,
        action_on_pick1(SubBag,X,Eqn,S,N,T,Flag,Eqn1).

 % No schema passes first tie break, so choose one at random
action_on_pick(Bag,_,_,S,N,T,random(N),Eqn1) :-
        member(p_s(schema(S,Eqn1,_),N,T),Bag).

find_subset_same_type([],_,[]).
find_subset_same_type([p_s(S,Eqn1,X)|T],Type,[p_s(S,Eqn1,X)|T1]) :-
        find_type(X,Eqn1,Type1),
        Type=Type1,
        !,
        find_subset_same_type(T,Type,T1).

find_subset_same_type([_|T],Type,T1) :-
        find_subset_same_type(T,Type,T1).

 % Only one passes first tie break
action_on_pick1([p_s(schema(S,Eqn1,_),N,T)],_,_,S,N,T,only2(N),Eqn1) :- !.


 % Use second tie break, if allowed
action_on_pick1(SubBag,X,Eqn,S,N,T,Flag,Eqn1) :-
        flag(tiebreak,on,on),
        mapscore_compare(X,Eqn,SubBag,S,N,T,Eqn1,Flag),
        !.

action_on_pick1(Bag,_,_,S,N,T,random2(N),Eqn1) :-
        member(p_s(schema(S,Eqn1,_),N,T),Bag).

mapscore_compare(X,Eqn,Bag,S,N,T,Eqn1,Flag) :-
        test(X,Eqn,TestList),
        find_sub_test(Bag,TestList,SubBag),
        SubBag \== [],
        action_on_pick2(SubBag,S,N,T,Flag,Eqn1).

find_sub_test([],_,[]) :- !.
find_sub_test([p_s(schema(S,Eqn,X),N,T)|R],Te,[p_s(schema(S,Eqn,X),N,T)|U]) :-
        test(X,Eqn,Test),
        Test = Te,
        !,
        find_sub_test(R,Te,U).

find_sub_test([_|T],Test,Sub) :-
        find_sub_test(T,Test,Sub).

action_on_pick2([p_s(schema(S,Eqn1,_),N,T)],S,N,T,only3(N),Eqn1) :- !.
action_on_pick2(Bag,S,N,T,random3(N),Eqn1) :-
```

```
        member(p_s(schema(S,Eqn1,_),N,T),Bag).


test(X,Eqn,[Occ,ZeroRhs,DomFunc]) :-
        occ(X,Eqn,Occ),
        type_of_rhs(Eqn,ZeroRhs),
        principle_functor(Eqn,DomFunc),
        !.

type_of_rhs(_=0,0) :- !.
type_of_rhs(_=_,n) :- !.
type_of_rhs(_,other) :- !.
principle_functor(A=_,F/N) :- !,
        functor(A,F,N).
principle_functor(_#_,#).


show_initialize_and_store(X,Eqn,New) :-
        mod_abolish(asserted,1),
        mod_abolish(schema_used,1),
        asserta(last_equation(Eqn)),
        writef('\nSolving %t for %t.\n',[Eqn,X]),
        tidy(Eqn,New1),
        find_functions([New1],X,check),
        (match_check(Eqn,New1);writef('\nTidying to %t.\n',[New1])),
        mod_weak_normal_form(New1,X,New),
        reset,
        mod_abolish(schema_abandon,1),
        !.


 % Store solution steps
store_steps(Eqn,X,Flag,EqnSteps) :-
        asserta(store_we(work([Eqn|EqnSteps],X))),
        check_found_schema(Eqn,X,Flag).


 % There is an old schema that works
check_found_schema(_,_,new_flag(Flag,Flag1)) :-
        (Flag1 = win;(arg(1,Flag,X),Flag1=good(X))),
        !,
        writef('\n[No problems with schema]\n').


 % Old schema fails
check_found_schema(_,_,new_flag(win(_),bad)) :- !,
        writef('\n[**Old schema for this problem fails**]\n').

 % No schema before , or rough-match schema failed

check_found_schema(_,_,Flag) :-
        (good_flag(Flag);
        writef('\n
[**May need concept learning, rough-match schema failed**]\n')),
        writef('\n[Type work_solution to create a schema for this equation]\n').

good_flag(win).
good_flag(noschema).

 % Get next method and postcond from auto method
```

```
auto_next_applicable_method(Method,NewMethod,X,New1,Rest) :-
        known_method_auto(_,_,_,method(Method),_,_,
[applicable_next_method(NewMethod,X,New1)|Rest]),
        !.

get_next_element([[H|T]|T1],H,[T|T1]) :- !.
get_next_element([H|T],H,T) :- !.

 % Try to apply indicated schema step

suitable_step(X,Eqn,Purpose,Name,New,Steps,Flag) :-
        arg(1,Purpose,Name1),
writef('\nTrying indicated schema step of applying %w.\n',[Name1]),
        suitable_step1(X,Eqn,Purpose,Name,New,Steps,Flag,Name1),
        !.

suitable_step1(X,Eqn,_,Name,New,Steps,win,Name) :-
        step_solve_eqn(yes,Eqn,X,Name,New,Steps),
        !.

 % Try to apply one that satisfies required constraints
suitable_step1(X,Eqn,conditions(_,Sat,UnSat),Name,New,Steps,Flag,N1) :-
writef('\nFailed to apply schema step %w.\n',[N1]),
        expand_constraints(Sat,UnSat),
  uitable_methods(X,Eqn,Sat,UnSat,Name,New,Steps,Flag,N1),
          !.

expand_constraints(sat(Sat,_,_),unsat([],_,_)) :-
        writef('\nNo new constraints are satisfied by schema step.\n'),
       (Sat = [];single_plural(Sat,S,_),
        writef('\nAny new step must still satisfy the following
        precondition%w\n',[S]),
        explain_g1(Sat)),
        !.

expand_constraints(sat(Sat,_,_),unsat(UnSat,_,_)) :-
        single_plural(UnSat,W,_),
        writef('\nTrying to find a method whose result satisfies the
following precondition%w\n', [W]),
        explain_g1(UnSat),
        (Sat=[];single_plural(Sat,W1,_),
        writef('\n\nwhile keeping the following condition%w true\n',[W1]),
        explain_g1(Sat)),
        !.

suitable_methods(X,Eqn,Sat,UnSat,Name,New,Steps,win,N1) :-
        find_satisfy_unsat(UnSat,Name),
        not bad_name(Name),
        Name \== N1,
        maintain_satisfy(Sat,Name),
        flag(method(Name),on,on),
        step_solve_eqn1(yes,Eqn,X,Name,New,Steps,Mess),
        report_mod_check_cond(UnSat,New,X,Name),
        call(Mess),
        !.

suitable_methods(X,Eqn,Sat,UnSat,Name,New,Steps,no_win,N1) :-
        arg(1,UnSat,Arg),
        (Arg =[_|_] ->
        writef('\nCan''t find method to help satisfy new conditions.\n');
```

```
        Arg=[]),
        arg(1,Sat,Arg1),
        Arg1=[_|_],
writef('\nLooking for method that keeps satisfied conditions satisfied.\n'),
        known_method(_,_,_,Name,_,_,_,_),
        Name \== N1,
        not bad_name(Name),
        flag(method(Name),on,on),
        maintain_satisfy(Sat,Name),
        step_solve_eqn1(yes,Eqn,X,Name,New,Steps,Mess),
        report_mod_check_cond(Sat,New,X,Name),
        call(Mess),
        !.

report_mod_check_cond(UnSat,New,X,Name) :-
        writef('\n[Method %w can be applied\n',[Name]),
        mod_check_cond(UnSat,New,X),
        !,
        writef('and satisfies all the conditions]\n').

mod_check_cond(Term,New,X) :-
        not not (arg(1,Term,List),
        arg(2,Term,Unk),
        arg(3,Term,Eqn),
        New = Eqn,
        X = Unk,
        mod_check_cond1(List)).

mod_check_cond1([]) :- !.
mod_check_cond1([H|T]) :-
        call(H),
        !,
        mod_check_cond1(T).

mod_check_cond1([H|_]) :-  % Only report first failure
        writef('but fails to satisfy (at least) the precondition\n'),
        explain_g1([H]),
        writef('\n]\n'),
        !,
        fail.


  nd_satisfy_unsat(unsat(UnSat,_,_),Name) :- find_satisfy_unsat1(UnSat,Name).
  nd_satisfy_unsat1([],_) :- !.
find_satisfy_unsat1([H|T],Name) :-
        (must_satisfy(H,List);might_satisfy(H,List)),
        member(Name,List),
        find_satisfy_unsat1(T,Name).

maintain_satisfy(sat(Sat,_,_),Name) :- m_s1(Sat,Name).
m_s1([],_) :- !.
m_s1([H|_],Name) :-
        excludes(H,List),
        member(Exclude,List),
        must_satisfy(Exclude,List1),
        member(Name,List1),
        !,
        fail.
```

```
m_s1([_|T],Name) :- m_s1(T,Name).

 % Choose a schema for the factors
choose_member_schema(Eqn,S,Schema) :-
        member(S,Schema),
        satisfy_satisfied(Eqn,Schema).

choose_member_schema(_,S,Schema) :-
        member(S,Schema).

satisfy_satisfied(Eqn,[conditions(_,sat(Sat,_,Eqn),_)|_]) :-
        check_cond(Sat),
        !.

decide_message(yes,_,no) :- !.
decide_message(_,gag,no) :- !.
decide_message(_,_,yes) :- !.

bad_name('Factorization') :- !.
bad_name('Change of Unknown') :- !.
bad_name(X) :- schema(X).

no_major_effect(conditions(_,_,unsat([],_,_))).
```

```
/
* TL :

                                                    Bernard Silver
                                                    Updated: 7 March 198
/
% Find steps between lines


% Top level

ork(Example) :- work(Example,x),!.

ork(Example,X) :-
        reset1,
        initialize_screen,
        work_main(Example,X).

ork_main(Example,X) :-
        statistics(runtime,_),
        show_the_example(Example,X,Example1),
        find_functions(Example1,X,check),
        work1(Example1,X,Method1,[],Unksteps,[]),
        report_steps(Example1,X,Method1,Method),
        check_contains_solution(Method1),
        conjecture_steps(Example1,X,Unksteps,Conjlist,UnkList,Method Method2),
        construct_method(Example1,Conjlist,Method2,X,UnkList),
        statistics(runtime,[_,Time]),
        writef('\n[Example took %t milliseconds]\n',[Time]),
        check_example(X,Example1,Example),
        !.

ork_main(_,_) :-
        statistics(runtime,[_,Time]),
        writef('\nCould not do problem.
The attempt took %t milliseconds]\n',[Time]).

        % SPECIAL CASES

ork1([],_,L,L,L1,L1) :- !.
ork1([X=_],X,[message(solution)|L],L,L1,L1) :- !.
ork1([false],_,[message(false)|L],L,L1,L1) :- !.
  1([true],_,[message(true)|L],L,L1,L1) :- !.

        % Superficial Factorization, number removed
ork1([Z*Y=0,Eqn=0|Rest1],X,[message(removecf)|L1],L3,US,L2) :-
        decomp(Z*Y,[*|List]),
        select_the_number(_,List,_),
        !,
        work1([Eqn=0|Rest1],X,L1,L3,US,L2),
        !.

select_the_number(Num,List,Rest) :- select(Num,List,Rest),number(Num ,!.

% Factorization
ork1([A*B=0,A1|Rest],X,[message('Factorization')|T],L1,UnknownSteps L2) :-
        is_factorization(X,A*B,A1,DisList),
        factor_part(DisList,Rest,StepsList),
        !,
        work2(StepsList,X,T,L1,UnknownSteps,L2).
```

```
s_factorization(X,A*B,A1,List1) :-
        decomp(A*B,[*|List]),
        remove_safe_divisors(X,List,New),
        maptidy_example(X,New,New1,work),
        recomp(Term,[#|New1]),
        (decomp(A1,[#|List1])->
        recomp(Term1,[#|List1]); Term1 = A1),
        match(Term,Term1),
        !.

% Disjunction
ork1([A#B|Rest],X,[message(solution)|T],L1,Unknownsteps,L2) :-
        dis_solution(A#B,X),
        !,
        work1(Rest,X,T,L1,Unknownsteps,L2).

ork1([A#B,A|Rest],X,T,L1,UnknownSteps,L2) :-
        ortodot(A#B,C),
        factor_part(C,[A|Rest],StepsList),
        !,
        work2(StepsList,X,T,L1,UnknownSteps,L2).


rtodot(B,C) :- decomp(B,[#|C]),!.
rtodot(B,[B]).

% Change of unknown
ork1([Old,Y=Subs,Neweqn|R],X,[message('Change of Unknown')|T],L1,Un s,L2) :-
        freeof(X,Y),
        contains(X,Subs),
        contains(X,Old),
        subst(Subs=Y,Old,New),
        tidy(New,New1),
        match_check(New1,Neweqn),
        chunk_part(Subs,R,CVsteps,SubsSteps),
        !,
        work3(X,Y,Subs,Neweqn,CVsteps,SubsSteps,T,L1,Unks,L2).

/* Normal case  */

    l([L1,Head|R],X,NewMess,L2,NewUnks,L3) :-
        find_step(L1,Head,X,NewMess,L4,NewUnks,L5),
        writef('\n[Processing]\n'),
        work1([Head|R],X,L4,L2,L5,L3),
        !.

% Case for examples that don't end with solutions
ork1([_],_,[message(nosol)|L1],L1,L,L) :-
        writef('\nNo solution step found.\n').



% find_step(Eqn1,Eqn2,Unk,_,_,_,_) tries to find
% the single PRESS step that transforms Eqn1 to Eqn2, with X as the unknown.

find_step(Old,New,X,[message(Name)|L1],L1,L2,L2) :-
        known_method(X,Old,New,Name,all,Goal,PreCond,PostCond),
        check_cond(PreCond),
        check_cond(PostCond),
```

```
        find_step_cont(Goal,Name,Old,New),
        !.

ind_step_cont(Goal,_,_,_) :-
        call(Goal),
        !.


ind_step_cont(_,Name,Old,New) :-
        Name \= user_rule(_,_,_),
        mod_assert(
arning(writef('\n\t[Possibly missing rule for method %w.]\n',[Name] ,New)
        ),
        mod_assert(p_m_r(Name,Old,New)),
        fail.

ind_step(Eqn1=A,Eqn2=B,X,Mess,L1,Unks,L2) :-
        understood_constants(A,B),
        find_step1(Eqn1,Eqn2=B,X,Mess,L1,Unks,L2),
        !.

% Not understood
   _step(_,New,_,[message(fail)|L1],L1,[unknown_steps(New)|L2],L2)  - !.


nderstood_constants(A,A) :- !.

nderstood_constants(A,B) :- match_check(A,B),!.

ind_step1(Term,Term=_,_,L1,L1,L2,L2) :- !.


ind_step1(Term1,Term2=B,X,[message(Name)|L1],L1,L2,L2) :-
        known_method(X,Term1,Term2,Name,part,Goal,PreCond,PostCond),
        check_cond(PreCond),
        check_cond(PostCond),
        find_step_cont1(Goal,Name,Term1,Term2=B),
        !,
        call(Goal),
        !.


  _step_cont1(Goal,_,_,_) :-
        call(Goal),
        !.


find_step_cont1(_,Name,Old,Eqn) :-
        Name \= user_rule(_,_,_),
        mod_assertz(
warning(writef('\n\t[Possibly missing rule for method %w.]\n',[Name] ,Eqn)
        ),
        mod_assert(p_m_r(Name,Old,Eqn)),
        fail.

 % Isolation
try_to_isolate(X,Eqn,New) :-
        position(X,Eqn,Posn),
        isolate(Posn,Eqn,Isol),
        mod_weak_normal_form1(Isol,expr,X,Isol1),
```

```
        remove_arbs(Isol1,NewIsol),
        consider_isolation1(NewIsol,New,X),
        !.

 % Isolation is complete
:onsider_isolation1(NewIsol,New,_) :-
        remove_arbs(New,New1),
        match_check(NewIsol,New1),
        !.

 % Partial isolation has occurred
:onsider_isolation1(NewIsol,New,X) :-
        occ(X,New,1),
        tidy(New,Tidy),
        position(X,Tidy,Posn),
        isolate(Posn,Tidy,New1),
        remove_arbs(New1,NewEqn),
        match_check(NewEqn,NewIsol),
        !.

 % Collection
:ry_collect(X,Eqn1,Eqn2) :-
        tidy(Eqn1,New1),
        collect(X,New1,New),
        collect_check(X,Eqn2,New),
        !.

 % Collection is complete
:ollect_check(_,Eqn,New) :- match_check(Eqn,New),!.

 % Partial Collection
:ollect_check(X,Eqn,New) :-
        mult_occ(X,New),
        recurse_collect1(X,New,Normal),
        !,
        collect_check1(X,Normal,Eqn).

:ollect_check1(_,Normal,Eqn) :- match_check(Normal,Eqn),!.
:ollect_check1(X,Normal,Eqn) :-
        mult_occ(X,Eqn),
        recurse_collect1(X,Eqn,N1),
        !,
        match_check(N1,Normal).

 % Attraction
try_attract(X,Eqn1,Eqn2) :-
        tidy(Eqn1,New1),
        attract(X,New1,New),
        match_check(Eqn2,New),
        attract_check(X,Eqn2,New),
        !.

 % Attraction is complete
attract_check(_,Eqn,New) :- match_check(Eqn,New),!.

 % Partial Attraction
attract_check(X,Eqn,New) :-
        mult_occ(X,New),
        recurse_attract1(X,New,Normal),
        !,
```

```
        attract_check1(X,Normal,Eqn).

attract_check1(_,Normal,Eqn) :- match_check(Normal,Eqn),!.
attract_check1(X,Normal,Eqn) :-
        mult_occ(X,Eqn),
        recurse_attract1(X,Eqn,N1),
        !,
        match_check(N1,Normal).

 % Function Stripping
try_function_stripping(X,Old,Posn,New) :-
        isolate(Posn,Old,NewIsol),
        mod_weak_normal_form1(NewIsol,expr,X,New1),
        remove_arbs(New1,New2),
        remove_arbs(New,New3),
        match_check(New3,New2),
        !.

 % Prepare for Factorization
try_prep_fact(X,A+B=0,C*D=0) :-
        decomp(C*D,[*|Args]),
        member(Term,Args),
        functor(Term,+,_),
        mod_collect(X,A+B,New),
        tidy(New,New1),
        match_check(New1,C*D),
        !.

 % Polynomial

try_poly(X,Old,New)  :-
        poly_solve(Old,X,Ans,_),
        poly_step(Old,X,New,Ans),
        !.

 % Can't solve poly, but normal form is the same
try_poly(X,A=B,NL=NR) :-
        poly_norm(A+ -1*B,X,Plist),
        poly_norm(NL+ -1*NR,X,Plist1),
        !,
        poly_tidy(Plist,Qlist),
        poly_tidy(Plist1,Qlist1),
        check_same_poly(_,Qlist,Qlist1).

poly_step(_,_,New,Ans) :-
        tidy(Ans,Ans1),
        match_check(Ans1,New),
        !.

 % Solve to solution
poly_step(_,X,A=B,Ans) :-
        poly_solve(A=B,X,Ans1,_),
        tidy(Ans1,Ans2),
        match_check(Ans,Ans2),
        !.

 % New methods


try_auto_method(Method,X,Term1,Term2) :-
```

```
        auto_rule(Method,NewName),
        apply_new_rule1(X,Term1,Term2,NewName,tl),
        !.


emove_logs(X,New,Mid,Base) :-
        log_reduce(Mid,X,Base,New).

ry_homog(X,Eqn,Set,New) :-
        homog(Eqn,X,Set,Homog),
        mod_weak_normal_form1(Homog,expr,X,New2),
        match_check(New2,New).

emove_nasty(X,Eqn,New) :-
        nasty_method(Eqn,X,New1),
        mod_weak_normal_form1(New1,expr,X,New2),
        match_check(New2,New).

:ry_user_rule(X,Term1,Term2,Name) :-
        apply_new_rule1(X,Term1,Term2,Name,tl),
        !.


 % Enter worked example line by line

jive_example :-
        writef('\nEnter example, line by line, using x as the unknow .\n
Terminate with **<CR>** on new line.\n\n'),
        prompt(_,'First line:'),
        get_lines(Example),
        writef('\n
Example has been stored, and can be rerun using **xredo** predicate. n\n'),
        asserta(last_example(Example,x)),
        work(Example,x).

get_lines(New) :-
        read_in_line(Line),
        tidy_expr(Line,NewLine),
        (NewLine=end;writef('\n\n%t.\n\n',[NewLine])),
        !,
        get_lines1(New,NewLine).


get_lines1([],end) :-
        writef('\nEnd of Example Input.\n'),
        !.

get_lines1([Line|Rest],Line) :-
        prompt(_,'Next line:'),
        get_lines(Rest).


 % Run last example again

xredo :-
        last_example(Example,X),
        !,
        writef('\nRerunning Example.\n\n'),
        work(Example,X).
```

```
redo :- writef('\nNo previous example, nothing done.\n\n').

ld_xredo :-
        asserted(_),
        !,
        writef('\n[Retracting added facts]\n'),
        o_x1.

ld_xredo :- xredo.

_x1 :-
        retract(asserted(X)),
        retract(X),
        fail.

_x1 :- xredo.

% Run last solution as an example

ork_solution :-
        store_we(X),
        writef('\nRunning last solution trace.\n'),
        !,
        call(X).

ork_solution :- writef('\nNo solution trace is stored.\n'),!.

heck_contains_solution(List) :-
        member(message(nosol),List),
        !,
ritef('\nNo solution step has been found, the example will not be p ocessed
further.  Please supply a new example with a solution step!\n'),
        fail.

check_contains_solution(_) :- !.
```

SPECIA : Code for special cases of top level

Bernard Silver
Updated: 28 January 1984

Partition steps for factorization case
We require that after factorization step the equations are solved in order, i.e. from A*B*C...*Z=0, the next line is A = 0, (or its tidied form) which is solved, then the next factor B=0 appears, and the example proceeds with the solution of this equation.

Disjunction is similar

```
ctor_part([_|T],List,Ans) :- factor_part1(T,List,Ans).

ctor_part1([],Steps,[Steps]).
ctor_part1([H|T],Steps,[HSteps|Rest]) :-
      wordsin(H,Words),
      member(W,Words),
      unknown(W),
      mod_weak_normal_form1(H,expr,W,H1),
      factor_part2(H1,Steps,HSteps,NewSteps),
      factor_part1(T,NewSteps,Rest).


ctor_part2(H1,[H2|Rest],[],[H2|Rest]) :- match(H1,H2),!.
ctor_part2(H1,[H2|Rest1],[H2|L2],Rest) :-
      factor_part2(H1,Rest1,L2,Rest),
      !.

 Failure
ctor_part2(_,_,_,_) :-
      writef('Failure to find solution of next disjunct.'),
      !.

rk2([[A|Rest]|StepsList],X,
essage(ff)|T],L1,Unk,L2) :-
      work1([A|Rest],X,T,L3,Unk,L4),
      work2r(StepsList,X,L3,L1,L4,L2),
      !.

rk2r([],_,T,T,L,L) :- !.
rk2r([[B|Rest]|Tail],X,[message(nf)|T],L1,Unk,L2) :-
      work1([B|Rest],X,T,L3,Unk,L4),
      work2r(Tail,X,L3,L1,L4,L2),
      !.
```

Change of Unknown
We require the example has a step of the form New = f(Old) and, after solving the changed variable equation, a step f(Old)=Ans, before the substitution equation is solved.

```
 Substitution Equation is found.
unk_part(Subs,[Subs=Ans|R],[],[single(Subs=Ans)|R]) :- !.
```

```
 Disjunctive solution is found
unk_part(Subs,[H|R],[],[mult(H)|R])  :- dis_solution(H,Subs),!.

 Not found yet, so add steps to CVsteps.
unk_part(Subs,[H|R],[H|CVSteps],Substeps) :-
      chunk_part(Subs,R,CVSteps,Substeps),
      !.

 Failure
unk_part(_,_,_,_) :-
      writef('Failure to find solution of substitution equation.'),
      !.

rk3(Old,New,_,Neweqn,CVsteps,[H|SubsSteps],[message(cve)|NewM],L1,Unks,L2) :-
      work1([Neweqn|CVsteps],New,NewM,[message(ses)|L3],Unks,L4),
      arg(1,H,H1),
      work1([H1|SubsSteps],Old,L3,L1,L4,L2),
      !.
```

```
/* GENPRB :

                                                    Bernard Silver
                                                    Updated: 6 February 1984
*/

generate_problem :-
        reset,
        known_method_schema(_,_,_,_,schema(_,Eqn,_),_,_,_,_),
        !,
        g_p1(Eqn).

generate_problem :-
        writef('\nNo schemas are stored, can''t generate problem.\n').

g_p1(Eqn) :-
        writef('\n
Attempting to generate a problem to test schema generated by problem\n%t\n',
[Eqn]),
        setof(N,problem(x,Eqn,N),Set),
        !,
        write_set(Set),
        length(Set,L),
        numbers_between(1,L,Num),
        process_reply([n|Num],get_problem(Eqn,x,Set,Ans),Ans,'Problem:',
writef('\nWhich problem do you want run? Type number or n for none.\n')).

g_p1(_) :-
        writef('\nUnable to generate a suitable problem.\n'),
        !.

problem(X,Eqn,New) :-
        known_method_schema(X,V,_,_,schema(_,Eqn,_),_,_,Precond,_),
        problem1(X,Eqn,New),
        V=New,
        check_cond(Precond).

problem1(X,Eqn=0,New) :-
        decomp(Eqn,[+|Args]),
        all_same_type(X,Args),
        sibling(Args,NewArgs),
        recomp(NewEqn,[+|NewArgs]),
        tidy(NewEqn=0,New).

problem1(X,A*B=0, New) :-
        decomp(A*B,[*|Args]),
        select(Term,Args,Rest),
        contains(X,Term),
        recomp(New2,[*|Rest]),
        problem1(X,Term=0,New1=0),
        tidy(New1*New2=0,New).

problem1(_,Eqn=A,Eqn=B) :-
        number(A),
        !,
        A\==0,
        eval(A+1,B).

problem1(_,Eqn=A,Eqn=B) :-
        A=..[F|Args],
```

```
        get_new_args(Args,NewArgs),
        NewArgs\==Args,
        B=..[F|NewArgs].

problem1(X,Eqn=A,New=A) :-
        Eqn=..[F|Args],
        get_new_args1(X,Args,NewArgs),
        NewArgs \== Args,
        New=..[F|NewArgs].

get_new_args([],[]) :- !.
get_new_args([H|T],[H1|T1]) :-
        eval(H+1,H1),
        get_new_args(T,T1).

get_new_args([H|T],[H|T1]) :-
        get_new_args(T,T1).

get_new_args1(_,[],[]) :- !.
get_new_args1(X,[H|T],[H1|T1]) :-
        contains(X,H),
        return_new_term(H,H1),
        get_new_args1(X,T,T1).

get_new_args1(X,[H|T],[H|T1]) :-
        get_new_args1(X,T,T1).

return_new_term(A*B,New) :- !,
        decomp(A*B,[*|List]),
        select(El,List,Rest),
        number(El),
        eval(El+1,El1),
        recomp(New,[*,El1|Rest]).

return_new_term(A,B) :- tidy(2*A,B).

all_same_type(X,[H|T]) :-
        functor(H,F,N),
        all_same_type1(X,T,F,N).

all_same_type1(_,[],_,_) :- !.
   _same_type1(X,[H|T],F,N) :-
        functor(H,F,N),
        all_same_type1(X,T,F,N).

all_same_type1(X,[H*I|T],F,N) :-
        decomp(H*I,List),
        check_functor(X,List),
        all_same_type1(X,T,F,N).


check_functor(_,[]).
check_functor(X,[H|T]) :-
        freeof(X,H),
        !,
        check_functor(X,T).
check_functor(X,[_|T]) :-
        !,
        check_free(X,T).
```

```
:heck_free(_,[]).
:heck_free(X,[H|T]) :-
        freeof(X,H),
        !,
        check_free(X,T).


sibling(List,[NewTerm|Rest]) :-
        select(Term,List,Rest),
        sibling_term(Term,NewTerm).

sibling_term(H,H1) :-
        tree(_,Tree),
        member(type(Type,H,F1),Tree),
        member(type(Type,_,F2),Tree),
        F1 \== F2,
        H=..[F1|Args],
        H1 =.. [F2|Args].

sibling_term(A*B,New) :- return_new_term(A*B,New).

sibling_term(Term,New) :-
        Term =.. [F|Args],
        sibling(Args,NewArgs),
        New =.. [F|NewArgs].

write_set(Set) :-
        writef('\nThe set of generated problems is:\n'),
        write_set1(Set,1).

write_set1([],_) :- writef('\n[End of set]\n').
write_set1([H|T],N) :-
        writef('\n%t)\t\t%t\n',[N,H]),
        M is N + 1,
        !,
        write_set1(T,M).

numbers_between(L,M,[]) :- L> M,!.
numbers_between(N,M,[N|T]) :-
        M1 is N + 1,
        !,
        numbers_between(M1,M,T).

get_problem(_,_,_,n) :- !,
        writef('\nOK, no test problems will be run.\n').

get_problem(Eqn,X,Set,N) :-
        nmember(New,Set,N),
        writef('\nEquation\n\n%t\n\nselected.\n\nRun begins:\n',[New]),
        loop_flag(warn1),
        get_problem1(X,Eqn,New),
        !.

get_problem1(X,Eqn,New) :-
        known_method_schema(X,New,_,_,schema(_,Eqn,_),_,Call,_,_),
        call(Call),
        !,
        writef('\nSchema solves test problem.\n').

get_problem1(X,_,New) :-
```

```
        writef('\n[**Unable to solve test problem with schema**]\n'),
        writef('\nTrying to solve without schema.\n'),
        !,
        get_problem2(X,New).


get_problem2(X,Eqn) :-
        reset,
        sort_solve(X,Eqn,_,_,_-[]),
        !,
        writef('\n
[**Test problem can be solved without schema, may need concept learning**]\n').


get_problem2(_,_) :- writef('\n[Test problem could not be solved by LP]\n').

 % Args of tree are name and structure.
 % Structure is a list of type(Class,Term,Termname) predicates.

tree(trig_tree,
        [type(sincos,sin(_),sin),
        type(sincos,cos(_),cos),
        type(other,tan(_),tan),
        type(other,sec(_),sec),
        type(other,cosec(_),cosec),
        type(other,cot(_),cot)]).


tree(log_tree,[type(log,log(_,_),log)]).
```

```
/* GENPRB :

                                                    Bernard Silver
                                                    Updated: 29 February 1984
*/

generate_problem :-
        reset,
        known_method_schema(_,_,_,_,schema(_,Eqn,_),_,_,_,_),
        !,
        g_p1(Eqn).

generate_problem :-
        writef('\nNo schemas are stored, can''t generate problem.\n').

g_p1(Eqn) :-
        writef('\n
Attempting to generate a problem to test schema generated by problem\n%t\n',
[Eqn]),
        setof(N,problem(x,Eqn,N),Set),
        !,
        write_set(Set),
        length(Set,L),
        numbers_between(1,L,Num),
        process_reply([n|Num],get_problem(Eqn,x,Set,Ans),Ans,'Problem:',
writef('\nWhich problem do you want run? Type number or n for none.\n')).

g_p1(_) :-
        writef('\nUnable to generate a suitable problem.\n'),
        !.

problem(X,Eqn,New) :-
        known_method_schema(X,V,_,_,schema(_,Eqn,_),_,_,Precond,_),
        problem1(X,Eqn,New),
        V=New,
        check_cond(Precond).

problem1(X,Eqn=0,New) :-
        decomp(Eqn,[+|Args]),
        all_same_type(X,Args),
        sibling(Args,NewArgs),
        recomp(NewEqn,[+|NewArgs]),
        tidy(NewEqn=0,New).

problem1(X,A*B=0, New) :-
        decomp(A*B,[*|Args]),
        select(Term,Args,Rest),
        contains(X,Term),
        recomp(New2,[*|Rest]),
        problem1(X,Term=0,New1=0),
        tidy(New1*New2=0,New).

problem1(_,Eqn=A,Eqn=B) :-
        number(A),
        A\==0,
        eval(A+1,B).

problem1(_,Eqn=A,Eqn=B) :-
        A=..[F|Args],
        get_new_args(Args,NewArgs),
```

```
        NewArgs\==Args,
        B=..[F|NewArgs].

problem1(X,Eqn=A,New=A) :-
        Eqn=..[F|Args],
        get_new_args1(X,Args,NewArgs),
        NewArgs \== Args,
        New=..[F|NewArgs].

get_new_args([],[]) :- !.
get_new_args([H|T],[H1|T1]) :-
        eval(H+1,H1),
        get_new_args(T,T1).

get_new_args([H|T],[H|T1]) :-
        get_new_args(T,T1).

get_new_args1(_,[],[]) :- !.
get_new_args1(X,[H|T],[H1|T1]) :-
        contains(X,H),
        return_new_term(H,H1),
        get_new_args1(X,T,T1).

  t_new_args1(X,[H|T],[H|T1]) :-
        get_new_args1(X,T,T1).

return_new_term(A*B,New) :- !,
        decomp(A*B,[*|List]),
        select(El,List,Rest),
        number(El),
        eval(El+1,El1),
        recomp(New,[*,El1|Rest]).

return_new_term(A,B) :- tidy(2*A,B).

all_same_type(X,[H|T]) :-
        functor(H,F,N),
        all_same_type1(X,T,F,N).

all_same_type1(_,[],_,_) :- !.
all_same_type1(X,[H|T],F,N) :-
        functor(H,F,N),
        all_same_type1(X,T,F,N).

all_same_type1(X,[H*I|T],F,N) :-
        decomp(H*I,List),
        check_functor(X,List),
        all_same_type1(X,T,F,N).


check_functor(_,[]).
check_functor(X,[H|T]) :-
        freeof(X,H),
        !,
        check_functor(X,T).
check_functor(X,[_|T]) :-
        !,
        check_free(X,T).

check_free(_,[]).
```

```
check_free(X,[H|T]) :-
        freeof(X,H),
        !,
        check_free(X,T).


sibling(List,[NewTerm|Rest]) :-
        select(Term,List,Rest),
        sibling_term(Term,NewTerm).

sibling_term(H,H1) :-
        tree(_,Tree),
        member(type(Type,H,F1),Tree),
        member(type(Type,_,F2),Tree),
        F1 \== F2,
        H=..[F1|Args],
        H1 =.. [F2|Args].

sibling_term(A*B,New) :- return_new_term(A*B,New).

sibling_term(Term,New) :-
        Term =.. [F|Args],
        sibling(Args,NewArgs),
        New =.. [F|NewArgs].

write_set(Set) :-
        writef('\nThe set of generated problems is:\n'),
        write_set1(Set,1).

write_set1([],_) :- writef('\n[End of set]\n').
write_set1([H|T],N) :-
        writef('\n%t)\t\t%t\n',[N,H]),
        M is N + 1,
        !,
        write_set1(T,M).

numbers_between(L,M,[]) :- L> M,!.
numbers_between(N,M,[N|T]) :-
        M1 is N + 1,
        !,
        numbers_between(M1,M,T).

 ^_problem(_,_,_,n) :- !,
        writef('\nOK, no test problems will be run.\n').

get_problem(Eqn,X,Set,N) :-
        nmember(New,Set,N),
        writef('\nEquation\n\n%t\n\nselected.\n\nRun begins:\n',[New]),
        loop_flag(warn1),
        get_problem1(X,Eqn,New),
        !.

get_problem1(X,Eqn,New) :-
        known_method_schema(X,New,_,_,schema(Schema,Eqn,_),Type,_,_,_),
        sort_solve1(New,X,Ans,Schema,Type,_),
        !,
        writef('\nSchema solves test problem.\n').

get_problem1(X,_,New) :-
        writef('\n[**Unable to solve test problem with schema**]\n'),
```

```
        writef('\nTrying to solve without schema.\n'),
        !,
        get_problem2(X,New).


get_problem2(X,Eqn) :-
        reset,
        sort_solve(Eqn,X,_,_,_,-[]),
        !,
        writef('\n
[**Test problem can be solved without schema, may need concept learning**]\n').


get_problem2(_,_) :- writef('\n[Test problem could not be solved by LP]\n').

 % Args of tree are name and structure.
 % Structure is a list of type(Class,Term,Termname) predicates.

tree(trig_tree,
        [type(sincos,sin(_),sin),
        type(sincos,cos(_),cos),
        type(other,tan(_),tan),
        type(other,sec(_),sec),
        type(other,cosec(_),cosec),
        type(other,cot(_),cot)]).


tree(log_tree,[type(log,log(_,_),log)]).
```

```
* NEWMET:  Conjecture new methods

                                                    Bernard Silver
                                                    Updated: 23 February 1984
/

* Work backwards through the description of the worked example.
 is the unknown in the example, U is the variable representing the unknown
n the rules.

ind the first hard step, one which we do not know the reason for, although
e know that a certain rule, Name was used to perform the step.
If we just have a fail, or tellequal marker, due to not having the
ules, no new method can be created.  Type is a flag to indicate this.)
his step transforms the equation from From to To (say!), is step N
n the input list.

ind which method was applied next, call this next method NM.
ind the preconditions of NM, making two copies Pre and Pre1.  Also
ake two copies of the pattern that NM expects for input, Patt and Patt1.

ind which of the preconditions Pre, (of NM), are not satisfied by From.
h se are called MP.  ND is a message giving information about MP.

hose preconditions of NM that are satisfied are called Rest.  The
wo copies are needed to avoid  unwanted instantiations.

ow create a new method that makes the bad preconditions satisfied, we
lready know that rule Name does this.  We insist that after application of
he new method all preconditions of NM are satisfied, so that method can be
pplied.

/

ind_new_method(X,List,Method,RList) :-
        allowed_new_methods,
        find_first_hard_step(List,RList,From,To,NM,Type,N),
        get_preconditions(U,NM,Patt1,Pre1),
        get_preconditions(U,NM,Patt,Pre), % Need two copies
        find_missing_preconds(To,U,X,From,MP,Pre,Patt),
        find_ok_preconds(U,X,From,Patt1,Pre1,Rest),
        tell_reason(From,To,NM,MP,ND),
te_method(From,U,Patt1,Patt,NM,ND,List,NewL,N,Rest,Type,Pre,X),
        !,
        find_new_method(X,NewL,Method,RList).

ind_new_method(_,List,List,_).

et_preconditions(X,user(Name),Pattern,PreCond) :- !,
        user_rule(Name,X,Pattern=>_, PreCond,_,_,_).

et_preconditions(X,Method,Pattern,Precond) :-
lause(known_method_tl(X,Pattern,_,Method,_,_,Precond,_),_),
        !.

et_preconditions(X,Method,Pattern,Precond) :-
lause(known_method_auto(X,Pattern,_,Method,_,Precond,_),_),
        !.

et_preconditions(X,Method,Pattern,Precond) :-
```

```
lause(known_method2(X,Pattern,_,Method,_,_,Precond,_),_),
        !.

ind_missing_preconds(To,U,X,Old,Rd,Precond,Pattern) :-
        check_the_preconds(X,U,Precond,Pattern,To),
        find_missing_preconds1(U,X,Old,Pattern,Precond,[],Rd),
        !.



ind_missing_preconds1(_,_,_,_,[],Method,Method) :- !.
ind_missing_preconds1(U,X,Eqn,P1,[H|T],Acc,Method) :-
        not not (U=X,Eqn=P1,call(H)),
        !,
        find_missing_preconds1(U,X,Eqn,P1,T,Acc,Method).

ind_missing_preconds1(U,X,Eqn,P1,[H|T],Acc,Method) :- !,
        find_missing_preconds1(U,X,Eqn,P1,T,[H|Acc],Method).


% These preconditions are satisfied and standardized apart form the others
ind_ok_preconds(U,X,Eqn,P1,List,Method) :-
        find_ok_preconds1(U,X,Eqn,P1,List,[],Method).


ind_ok_preconds1(_,_,_,_,[],Acc,Acc) :- !.
ind_ok_preconds1(U,X,Eqn,P1,[H|T],Acc,Method) :-
        not not (U=X,Eqn=P1,call(H)),
        !,
        find_ok_preconds1(U,X,Eqn,P1,T,[H|Acc],Method).

ind_ok_preconds1(U,X,Eqn,P1,[_|T],Acc,Method) :-!,
        find_ok_preconds1(U,X,Eqn,P1,T,Acc,Method).

heck_the_preconds(X,U,Pre,Pattern,New) :-
        test_preconds(X,U,Pre,Pattern,New),
        fail.    % Undo bindings

heck_the_preconds(_,_,_,_,_).

est_preconds(X,U,Pre,Pattern,New) :-
        X=U,
        New = Pattern,
        !,
        check_preconds1(Pre).

heck_preconds1([]) :- !.
heck_preconds1([H|T]) :- call(H),!,check_preconds1(T).

ind_first_hard_step(L1,L2,From,To,Next,Type,Number) :-
        find_first_hard_step(L1,L2,From,To,Next,Type,Number,2).

ind_first_hard_step([N,fail,_|_],[_,To,From|_],From,To,N,fail,A,A) :-
        !.
ind_first_hard_step([N,user_rule(Name,From,To),_|_],[_,To,From|_],From,To,
        N,ur(Name),A,A) :- !.

ind_first_hard_step([N,tellequal(From,To),_|_],
        [_,To,From|_],From,To,N,te,A,A) :- !.
```

```
ind_first_hard_step([_|T],[_|T1],From,To,Next,Type,A,N) :-
        M is N + 1,
        find_first_hard_step(T,T1,From,To,Next,Type,A,M).



ell_reason(From,To,Me,Difference,NewDiff) :-
        writef('\nTrying to find a reason why the step from\n'),
        writef('\n\t%t\n\nto\n\n\t%t\n\nwas performed.\n',[From,To]),
        translate_difference(Difference,NewDiff,Message),
        !,
ritef('\nReason found is that next method to be applied,\n\n\t%w\n',[Me]),
        call(Message).

% No differences found
ell_reason(_,_,_,_,[]) :- !.

ranslate_difference([H|T],upc([H|T]),explain_g([H|T])) :- !.

ranslate_difference(_,_,_) :- !,fail.


xplain_g(List) :-
        length(List,Length),
        single_plural(List,S,_),
        writef('\n\nrequires its input to satisfy the following unsatisfied
recondition%w:\n\n',[S]),
        explain_g1(List),
        !.

xplain_g1(List) :-
        writef('\nPrecondition\t\t\tExplanation\n'),
        output_preconds(List),
        !.

utput_preconds([]).
utput_preconds([H|T]) :-
        functor(H,F,N),
        get_explanation(F/N,Explan),
        writef('\n%t/%t\t\t\t%w\n',[F,N,Explan]),
        !,
        output_preconds(T).


% Create new method

% From is eqn before rule Name was applied, To is after, X is the unknown.
% OldP, NewP and U are variable forms of these.  NM is the next method
% Type is the type of reason discovered for the step, List is the list
% of steps in the worked example, NL is this list after the step with
% the user rule application has been relabeled with the new method name.
% Pre is the list of preconditions, Rest are those that were satisfied by
% From.

reate_method(From,U,OldP,NewP,NM,Type,List,NL,N,Rest,ur(Name),Pre,X) :- !,
 create_method1(U,OldP,NewP,NM,Type,Method,Rest,Name,Pre,From,X,NewM,List),
        cond_create(NewM),
        nmember(Term,List,N),
        subst(Term= method(Method),List,NL),
        add_new_method(Method,Type),
```

```
        !.

reate_method(_,_,_,_,_,_,_,_,_,_,_,_,_,_) :- !,
        writef('\nNo means of producing new method.\n'),
        fail.

reate_method1(U,OldP,NewP,NMethod,upc([H|T]),Method,Rest,Name,_,_,_,MT,_) :-!,
        MT = [upc([H|T]),Rest,[H|T],U,OldP,NewP,Method,Name,NMethod].

reate_method1(U,OldP,NewP,NMethod,[],Method,Pre,Name,_,From,X,MT,_) :-
        applicable_next_method(NMethod,X,From),
        !,
        MT= [[para(NMethod,Name)],Pre,Pre,U,OldP,NewP,Method,Name,NMethod].

reate_method1(U,Old,_,method(NM),[],Method,Pre,Name,_,From,X,MT,_) :-
        consider_parallel(U,Old,Name,From,X,NM,NName),
        !,
        MT= [[ei(NM,Name,NName)],Pre,Pre,U,Old,_,Method,Name,method(NM)].



reate_method1(U,OldP,NewP,NMethod,[],Method,Pre,Name,_,_,_,MT,_) :- !,
        MT= [[ok(NMethod)],Pre,Pre,U,OldP,NewP,Method,Name,NMethod].

reate_method1(_,_,_,_,_,_,_,_,_,_,_,_,_,_) :-
        writef('\nCan not create a new method.\n'),
        !,
        fail.


% Only create methods if there are not already ones like them.

ond_create([]) :- !.

% Already have method like this

ond_create([_,Pre,Post,U,OldP,NewP,Method,Name,NMethod|_]) :-
        known_method_auto(U,OldP,NewP,method(Method),
ry_auto_method(Method,U,OldP,NewP),Pre,
applicable_next_method(NMethod,U,NewP)|Post]),
        !,
        already_suitable_method(Method,Name).

% Create new method
ond_create([upc([H|T]),Rest,[H|T],U,OldP,NewP,Method,Name,NMethod]) :-
        !,
        mod_gensym(auto,Method),
        pretty_print_conds([H|T],method(Method)),
        store_rules_list(Method,Name),
        mod_asserta(known_method_auto(U,OldP,NewP,method(Method),
ry_auto_method(Method,U,OldP,NewP),Rest,
applicable_next_method(NMethod,U,NewP)|[H|T]])).

ond_create([[Flag],Pre,Pre,U,OldP,NewP,Method,Name,NMethod]) :-
        mod_gensym(auto,Method),
        tell_create(Flag),
        writef('\nCreating new method, named %t.\n',[method(Method)]),
        store_rules_list(Method,Name),
        mod_asserta(known_method_auto(U,OldP,NewP,method(Method),
ry_auto_method(Method,U,OldP,NewP),Pre,
```

```
:applicable_next_method(NMethod,U,NewP)|Pre])).

:ell_create(para(NMethod,Name)) :-
        writef('\nRule %t is applied in parallel with %t.\n',[Name,NMethod]),
        mod_asserta(para(NMethod,Name)),
                !.

:ell_create(ei(NMethod,Name,NName)) :-
        writef('\nMethod %t (rule %t) and %t can be applied in either order.\n',
:NMethod,NName,Name]),
        mod_asserta(ei(NMethod,Name,NName)),
        !.


:ell_create(ok(NMethod)) :-
        writef('\n
The next method, %w, can not be applied unless the step occurs, but
no outstanding preconditions can be found.\n
\ssuming that this step is preparation for %t.\n',[NMethod,NMethod]),
        !.

already_suitable_method(Method,Name) :-
        flag(method(method(Method)),on,on),
        !,
        a_s_m1(Method,Name).


already_suitable_method(M,_) :- !,
        writef('\n[Method %t is disabled, will not create duplicate]\n',[M]).

 % First clause should only be used if using a file of new rules
a_s_m1(Method,Name) :-
        not auto_rule(Method,Name),
        !,
        writef('\nMethod %w is applicable.\n
\dding rule %w to rule list of this method.\n',[Method,Name]),
        mod_assert(auto_rule(Method,Name)),
        !.

a_s_m1(Method,_) :-
        just_created(Method),
        writef('
     is the newly defined method %t again.\nNothing done.\n',[Method]),
        !.

applicable_next_method('Factorization',_,_*_=0) :- !.
applicable_next_method('Split Disjunctions',_,_#_) :- !.
applicable_next_method('Isolation',X,Eqn) :- !,
        single_occ(X,Eqn).
applicable_next_method('Change of Unknown',X,Eqn) :- !,
        identical_subterms(Eqn,X,_).
applicable_next_method('Function Stripping',X,Eqn) :-
        dominated(X,Eqn,_),
        !.
applicable_next_method('Prepare for Factorization',X,A+B=0) :- !,
        common_subterms(X,A+B,_).

applicable_next_method(NMethod,X,NewP) :-   %Not right yet!
        known_method(X,NewP,_,NMethod,_,Call,Pre,Post),
        check_cond(Pre),
```

```
        method_transform(yes,Call,_,_,_,gag),
        check_cond(Post),
        !.

onsider_parallel(U,Old,Name,From,X,NName,NName1) :-
        auto_rule(NName,NName1),
        NName1 \==Name,
        not not try_parallel(U,X,From,Old,NName1),
        !.

ry_parallel(U,X,From,Old,Name) :-
        U=X,
        From=Old,
        apply_new_rule1(X,Old,_,Name,sol),
        !.

% Store the rules list for the new method
tore_rules_list(Method,Name) :-
        not auto_rule(Method,Name),
        !,
        mod_assert(auto_rule(Method,Name)),
        !.

   e_rules_list(_,_).

retty_print_conds(List,Name) :-
        single_plural(List,S,S1),
ritef('\nCreating a new method, named  %t,\n\nthat transforms equation so that the
ollowing precondition%w %w satisfied:\n\n<',[Name,S,S1]),
        map_prettify(List),
        !.

ap_prettify([H]) :- !,
        functor(H,F,N),
        writef('%t/%t>\n',[F,N]),!.
ap_prettify([H|T]) :- !,
        functor(H,F,N),
        writef('%t/%t,  ',[F,N]),
        !,
        map_prettify(T).
```

```
/* DESC:

                                                Bernard Silver
                                                Updated: 26 February 1984
*/

% Produce description of method
construct_method([Eqn|ExampleList],Conjlist1,Method,X,UnkList) :-
        process_conjlist(Conjlist1,[],Conjlist),
        process_method([Eqn|ExampleList],Method,New1,Conjlist),
        analyze(X,New1,Eqn,NewMethod,ExampleList),
        rev(NewMethod,Description),
        make_schema(X,Eqn,Description,[Eqn|ExampleList],UnkList),
        !.

 % Get conjectures
process_conjlist([conj_mess(Type,Comm,E1,E2)|Rest],Acc,Ans) :-
        process_cm(Type,Comm,E1,E2,New),
        process_conjlist(Rest,[New|Acc],Ans).

process_conjlist([],Ans,Ans) :- !.

 % Process conjecture ready for description

   This code is probably not needed
process_cm(fail,_,_,_,at(fail,fail,tellfail)) :- !.

 % Now there is a name for this rule
process_cm(equal,rule(A,B),E1,E2,at(E1,E2,tellequal(A,B))) :- !.
process_cm(old,_,E1,E2,at(E1,E2,old)) :- !.

process_method([],[],[],_)  :- !.
process_method([A,B|T1],[fail|T],[C1|Ans],Conj) :-
        match_conj(A,Conj,C1,NewConj),
        process_method([B|T1],T,Ans,NewConj),
        !.

process_method([_|T1],[H|T],[H|Ans],Conj) :-
        process_method(T1,T,Ans,Conj),
        !.

match_conj(A,Conj,X,NewConj) :-
        member(at(E,A,X),Conj),
        !,
        delete_one(at(E,A,X),Conj,NewConj).

match_conj(A,Conj,X,NewConj) :-
        member(at(E,B,X),Conj),
        match_check(A,B),
        !,
        delete_one(at(E,B,X),Conj,NewConj).


analyze(X,Desc,Eqn,NewMethod,ExampleList) :-
        rev(Desc,Desc2),
        rev([Eqn|ExampleList],RExList),
        find_new_method(X,Desc2,NewMethod,RExList),
        !.
```

```
make_schema(X,Eqn,MethodL,ExampleList,UnkList) :-
        partition_the_list(X,MethodL,P1,ExampleList,Part1,UnkList,Part2,Type),
        find_the_purpose(U,U1,P1,Part1,Part2,Schema),
        store_schema(X,Eqn,Schema,Type),
        !.


% Schema already exists
store_schema(X,Eqn,Schema,Type) :-
        known_method_schema(_,_,_,_,schema(S,Eqn1,X),Type,_,_,_),
        match(Eqn,Eqn1),
        !,
        (same_methods(S,Schema) ->
        writef('\n[Already have this schema for this equation]\n');writef('\n
[**There is already a DIFFERENT schema for this equation**]\n')).


 % Store schema
store_schema(X,Eqn,Schema,Type) :-
        first_method(Schema,Precond,Var,Old),
        mod_gensym(auto,Name),
        writef('\n[Creating new schema method, called %w, for this equation]
\n',[Name]),
        mod_asserta(known_method_schema(Var,Old,New,method(Name),
schema(Schema,Eqn,X),Type,use_schema(Var,Old,New,Type,Eqn,Schema,X),Precond,
[dis_solution(Var,New)])),
        flag(method(method(Name)),_,on),
        mod_asserta(method(method(Name))),
        mod_asserta(schema(method(Name))),
        !.

same_methods([],[]) :- !.
same_methods([[H|T]|T1],[[H1|T2]|T3]) :- !,
        same_methods([H|T],[H1|T2]),
        same_methods(T1,T3).
same_methods([H|T],[H1|T1]) :-
        arg(1,H,E),
        arg(1,H1,E),
        !,
        same_methods(T,T1).

first_method([[conditions(NM,_,_)|_]|_],Pre,Var,H) :- !,
        get_preconditions(Var,NM,H,Pre).

first_method([conditions(NM,_,_)|_],Pre,Var,H) :- !,
        get_preconditions(Var,NM,H,Pre).
```

```
CONSTR :

                                                    Bernard Silver
                                                    Updated: 27 February 1984


; Partition list into seperate solution paths
rtition_the_list(X,List,[Main|P],List1,[Main1|P1],List2,[Main2|P2],Type) :-
        sort_of_list(List,Type),
        find_first_part(Type,X,List,Main,R,List1,Main1,R1,List2,Main2,R2),
        partition_rest(Type,X,R,P,R1,P1,R2,P2).

rt_of_list(List,'Change of Unknown') :- member('Change of Unknown',List),!.
rt_of_list(List,'Factorization') :- member('Factorization',List),!.
rt_of_list(_,'General') :- !.


nd_first_part('Change of Unknown',_,['Change of Unknown'|T],
        ['Change of Unknown'],T,[H|T1],[H],T1,[H1|T2],[H1],T2) :- !.
nd_first_part('Factorization',_,['Factorization'|T],['Factorization'],T,
|T1],[H],T1,[H1|T2],[H1],T2) :- !.
nd_first_part(Ty,X,[H|T],Main,R,[H1|T1],[H1|Main1],R1,[H2|T2],
2|Main2],R2) :-
        ignore_term(H),
        !,
        find_first_part(Ty,X,T,Main,R,T1,Main1,R1,T2,Main2,R2).
nd_first_part('General',X,List,New,[],List1,New1,[],List2,New2,[]) :- !,
        p_r1('General',X,List,New,_,List1,New1,_,List2,New2,_).
nd_first_part(Ty,X,[H|T],[H|Main],R,[H1|T1],[H1|Main1],R1,[H2|T2],
2|Main2],R2) :-
        find_first_part(Ty,X,T,Main,R,T1,Main1,R1,T2,Main2,R2).
nd_first_part(_,_,[],[],[],[],[],[],[],[],[]) :- !.


rtition_rest(_,_,[],[],[],[],[],[]) :- !.
rtition_rest(Type,X,List,[F|P],List1,[F1|P1],List2,[F2|P2]) :-
        p_r1(Type,X,List,F,R,List1,F1,R1,List2,F2,R2),
        !,
        partition_rest(Type,X,R,P,R1,P1,R2,P2).

_r1(_,_,[H|T],[H],T,[H1|T1],[H1],T1,[X|T2],[X],T2) :- dis_solution(H1,X),!.
_r1(Type,X,[H|T],F,R,[H1|T1],[H1|F1],R1,[H2|T2],[H2|F2],R2) :-
        ignore_term(H),
        !,
        p_r1(Type,X,T,F,R,T1,F1,R1,T2,F2,R2).

_r1('Change of Unknown',X,[cve|T],F,R,[_|T1],F1,R1,[_|T2],F2,R2) :- !,
        p_r1('Change of Unknown',X,T,F,R,T1,F1,R1,T2,F2,R2).
_r1('Change of Unknown',X,[ses|T],F,R,[_|T1],F1,R1,[_|T2],F2,R2) :- !,
        p_r1('Change of Unknown',X,T,F,R,T1,F1,R1,T2,F2,R2).
_r1(Type,X,[start|T],F,T1,[_|T2],F1,R1,[_|T3],F2,R2) :- !,
        p_r1(Type,X,T,F,T1,T2,F1,R1,T3,F2,R2).
_r1(Type,X,[H|T],[H|F],R,[H1|T1],[H1|F1],R1,[H2|T2],[H2|F2],R2) :- !,
        p_r1(Type,X,T,F,R,T1,F1,R1,T2,F2,R2).

gnore_term(nd).
gnore_term(ff).
gnore_term(nf).
gnore_term(start).
```

```
nd_the_purpose(_,_,[],[],[],[]) :- !.
nd_the_purpose(U,Unk1,[H|T],[H1|T1],[U1|TU],[H2|T2]) :-
      find_the_purpose1(U,Unk1,H,H1,U1,H2),
      !,
      find_the_purpose(U,Unk1,T,T1,TU,T2).

nd_the_purpose1(_,_,[],_,_,[]) :- !.
nd_the_purpose1(Unk,_,[H],[A|_],[X|_],
      [conditions(H,sat(Sat,Unk,Eqn),unsat([],_,_))]) :- !,
      get_preconditions(Unk,H,Eqn,Pre),
      find_ok_preconds(Unk,X,A,Eqn,Pre,Sat).

nd_the_purpose1(Unk,Unk1,[H,H1|T],[A,B|T1],[X,X|UT],
      [conditions(H,sat(Sat,Unk1,Eqn2),unsat(Unsat,Unk,Eqn1))|T2]) :-
      get_preconditions(Unk,H1,Eqn1,Pre1),
      get_preconditions(Unk1,H1,Eqn2,Pre2),
      find_missing_preconds(B,Unk,X,A,Unsat,Pre1,Eqn1),
      find_ok_preconds(Unk,X,A,Eqn2,Pre2,Sat),
      !,
      find_the_purpose1(Unk,Unk1,[H1|T],[B|T1],[X|UT],T2).
```

```
%   File   : TABLE
%   Author : Bernard Silver
%   Updated: 6 March 1984
%   Purpose: Tables of Connection and Conditions for LP

                        /* Table of Connections */


must_satisfy(less_occ(_,_,_),['Collection','Prepare for Factorization']).
must_satisfy(identical_subterms(_,_,_),['Homogenization']).
must_satisfy(closer(_,_,_),['Attraction']).
must_satisfy(is_product(_,_),['Prepare for Factorization']).
must_satisfy(rhs_zero(_),['Prepare for Factorization']).
must_satisfy(mult_occ(_,_),['Attraction','Logarithmic Method']).

might_satisfy(single_occ(_,_),['Collection']).
might_satisfy(mult_occ(_,_),['Collection','Nasty Function Method',
'Function Stripping','Polynomial Methods']).
might_satisfy(is_mod_poly(_,_,_),['Collection','Attraction',
        'Nasty Function Method','Logarithmic Method','Function Stripping']).
might_satisfy(rhs_zero(_),['Nasty Function Method','Logarithmic Method',
        'Function Stripping']).
might_satisfy(common_subterms(_,_,_),['Collection','Attraction',
'Nasty Function Method','Logarithmic Method','Function Stripping']).


                        /* Table of Conditions */

excludes(single_occ(_,_),[mult_occ(_,_),common_subterms(_,_,_),
        multiple_offenders_set(_,_,_),closer(_,_,_),identical_subterms(_,_,_),
        prod_exp_terms_gen(_,_,_),dominated(_,_,_)]).

excludes(mult_occ(_,_),[single_occ(_,_)]).
excludes(identical_subterms(_,_,_),[single_occ(_,_)]).
excludes(prod_exp_terms_gen(_,_,_),[single_occ(_,_)]).
excludes(multiple_offenders_set(_,_,_),[single_occ(_,_)]).
excludes(common_subterms(_,_,_),[single_occ(_,_)]).
excludes(closer(_,_,_),[single_occ(_,_)]).
excludes(dominated(_,_,_),[single_occ(_,_)]).
excludes(is_product(_,_),[is_sum(_,_),is_disJunct(_,_)]).
excludes(is_sum(_,_),[is_product(_,_),is_disJunct(_,_)]).
excludes(is_disJunct(_,_),[is_product(_,_),is_sum(_,_)]).


set_explanation(common_subterms/3,'Equation has common additive subterms') :- !.
set_explanation(prod_exp_terms_gen/2,'Equation is a product of exponentials')
        :- !.
set_explanation(mult_occ/2,'Unknown occurs more than once in equation') :- !.
set_explanation(single_occ/2,'Unknown occurs exactly once in equation') :- !.
set_explanation(is_mod_poly/3,'Equation is a polynomial') :- !.
set_explanation(rhs_zero/1,'Right hand side of equation is 0') :- !.
set_explanation(is_disJunct/2,'Expression is a disJunction') :- !.
set_explanation(is_product/2,'Left hand side of equation is a product') :- !.
set_explanation(is_sum/2,'Left hand side of equation is a sum') :- !.
set_explanation(dominated/3,'All occurrences of unknown are dominated') :- !.
set_explanation(identical_subterms/3,'Unknowns occur in identical subterms') :-
        !.
set_explanation(multiple_offenders_set/3,'There are multiple offending terms')
        :- !.
```

```
 % Add new entry to table
add_to_table(Method,[H|T]) :-
        add_into_table(Method,H),
        !,
        add_to_table(Method,T).


add_to_table(_,[]).
 % Table already has entry for condition
add_into_table(Method,Cond) :-
        must_satisfy(Cond,List),
        !,
        add_into_table1(Cond,List,Method).


 % Start Table entry for condition
add_into_table(Method,Cond) :-
        mod_asserta(must_satisfy(Cond,[Method])),
        !.

 % Entry already contains method
 _into_table1(_,List,Method) :-
        member(Method,List),
        !.

 % Add new member to entry
 _into_table1(Cond,List,Method) :-
        retract(must_satisfy(Cond,List)),
        !,
        mod_asserta(must_satisfy(Cond,[Method|List])).
```

```
File    : COND
Author : Bernard Silver
Updated: 27 February 1984
Purpose: Conditions for LP

public
        closer/3,
        common_subterms/3,
        contains/2,
        contains_nasties/2,
        dis_solution/2,
        dominated/3,
        find_type/3,
        freeof/2,
        good_subterm/4,
        identical_subterms/3,
        is_disjunct/2,
        is_mod_poly/3,
        is_product/2,
        is_sum/2,
        less_occ/3,
        mult_occ/2,
        parse/3,
        rhs_zero/1,
        same_occ/3,
        single_occ/2,
        trigf/1.

- mode
                check_expp(+),
                check_logf(+),
                check_trigf(+),
                closer(+,+,+),
                common_subterms(+,+,?),
                common_subterms_list(+,+,?),
                contains(+, +),
                contains_nasties(+,+),
                dis_solution(+,+),
                dl_parse(+,?,+),
                dominated(+,+,?),
                dominatable_function(+),
                dominated1(+,+,+,+,?),
                find_type(+,+,?),
                findtype(-,+),
                freeof(+, +),
                freeof(+, +, -),
                good_subterm(+, +, +, -),
                good_subterm(+, -),
                good_subterm(+, +, -),
                identical_subterms(+, +, -),
                is_disjunct(+,+),
                is_mod_poly(+,+,?),
                is_product(+,+),
                is_sum(+,+),
                less_occ(+,+,+),
                mult_occ(+,+),
                parse(+,-,+),
                rhs_zero(+),
                same_occ(+,+,+),
                single_occ(+,+),
```

```
            trigf(+).


  Occurrence clauses.
lt_occ(Term,Exp)      :- occ(Term, Exp, N), N > 1, !.

ngle_occ(X,Eqn) :- occ(X,Eqn,1).

  This provides an alternative implementation of freeof/2 and
  contains/2 which should be faster and use less stack.

ntains(Kernel, Expression) :-
      \+ freeof(Kernel, Expression).

eeof(Kernel, Kernel) :- !,
      fail.
eeof(Kernel, Expression) :-
      simple(Expression), !.
eeof(Kernel, Expression) :-
      functor(Expression, _, Arity), !,
      freeof(Arity, Kernel, Expression).

      freeof(0, Kernel, Expression) :- !.
      freeof(N, Kernel, Expression) :-
              arg(N, Expression, Argument),
              freeof(Kernel, Argument),
              M is N-1, !,
              freeof(M, Kernel, Expression).

me_occ(X,Old,New) :-
      occ(X,Old,M),
      occ(X,New,N),
      !,
      M = N.

ss_occ(X,Old,New) :-
      occ(X,Old,M),
      occ(X,New,N),
      !,
      M > N.

 Pattern Matching
 zero(_=0).

_product(X,A*B=_) :-
      contains(X,A),
      contains(X,B).

_disjunct(X,A#B) :-
      contains(X,A),
      contains(X,B),
      !.

_sum(X,A+B=_) :-
      contains(X,A),
      contains(X,B),
      !.

is_solution(X=A,X) :- !,
      freeof(X,A).
```

```
s_solution(A#B,X) :- !,
        dis_solution(A,X),
        dis_solution(B,X).
s_solution(true,_) :- !.
s_solution(false,_) :- !.

 For F.P.
mmon_subterms(X,Term=_,Subterm) :- !,
        common_subterms(X,Term,Subterm).
mmon_subterms(X,Term,Subterm) :-
        decomp(Term,[+|List]),
        common_subterms_list(X,List,Subterm),
        !.

mmon_subterms_list(_,[],_).
mmon_subterms_list(X,[H|T],Subterm) :-
        subterms2(H,X,Subterm),
        common_subterms_list(X,T,Subterm).

 For Attraction
oser(X,A,B) :-
        closeness(X,A,C1),
        closeness(X,B,C2),
        !,
        C2<C1.


 Function Stripping
minated(X,L=R,[1|Posn]) :-
        mult_occ(X,L),
        functor(L,F,N),
        L =..[F|Args],
        dominated1(N,F,Args,X,Posn),
        !.


minated1(2,F,[A,B],X,[P]) :-
        dominatable_function(F),
        !,
        ((contains(X,A),freeof(X,B),P=1);
        (contains(X,B),freeof(X,A),P=2)),
        !.
  nated1(1,_,_,_,[1]) :- !.

minatable_function(*) :- !.
minatable_function(log) :- !.
minatable_function(^) :- !.

 Defn for Nasty Function Preconditions

ntains_nasties(X,Eqn) :-
        call(parse4(Eqn,X,U,other)),
        call(subnasty(X,U,V)),
        V \== [],
        !.

ntains_nasties(X,Eqn) :-
        call(parse4(Eqn,X,U,neg)),
        call(exp_nasty_list(X,U,V)),
```

```
        V \== [],
        !.



; Recognize poly if equation is not in weak normal form
_mod_poly(X,Pol,A=B) :-
        weak_normal_form(Pol,X,Eqn),
        !,
        Eqn = (A=B),
        is_poly(X,A).

For Change of Unknown

lentical_subterms(Lhs=Rhs, Var, Term) :-
        occ(Var, Lhs, N), N > 1,
        setof(Term, good_subterm(Lhs, Var, N, Term), TermSet),
        extreme_term(TermSet, >, Term),!.




c subterm(Exp, Var, N, Term) :-
        good_subterm(Exp, Term),
        occ(Var, Term, M), M > 0,
        occ(Term, Exp, L), L > 1,
        N is L*M.

        %   good_subterm(Term, Exp) is true when Term is a non-atomic subterm
        %   of Exp.  This enables us to drop the "Term \= Var" requirement in
        %   good_subterm/4.

        good_subterm(Exp, Term) :-
                (   atomic(Exp) ; number(Exp)   ), !, fail.
        good_subterm(Exp, Term) :-
                functor(Exp, _, N),
                good_subterm(N, Exp, Term).

            %   good_subterm(N,E,T) <- T is a good subterm of Exp's Nth argument

                good_subterm(0, Exp, Term) :- !, Term = Exp.
                good_subterm(N, Exp, Term) :-
                        arg(N, Exp, Arg),
                        good_subterm(Arg, Term).
                good_subterm(N, Exp, Term) :-
                        M is N-1, !,
                        good_subterm(M, Exp, Term).


; Type of equation (for schema methods)
ind_type(X,Eqn,Type) :-
        parse(Eqn,Set,X),
        findtype(Type,Set).

rse(Exp,Set,Unk) :- dl_parse(Exp,Set1-[],Unk),listtoset(Set1,Set).

._parse(A=_,L,Unk) :- !,dl_parse(A,L,Unk).
._parse(A#B,L-L1,Unk) :- !,dl_parse(A,L-L2,Unk),dl_parse(B,L2-L1,Unk).
._parse(A+B,L-L1,Unk) :- !,dl_parse(A,L-L2,Unk),dl_parse(B,L2-L1,Unk).
```

```
._parse(A*B,L-L1,Unk) :- !,dl_parse(A,L-L2,Unk),dl_parse(B,L2-L1,Unk).
._parse(A^B,L,Unk) :- number(B), !,dl_parse(A,L,Unk).
._parse(Unk,[Unk|L]-L,Unk) :- !.
._parse(A,L-L,Unk) :- freeof(Unk,A),!.
._parse(A,[A|L]-L,_) :- !.

indtype(trig,L) :- check_trigf(L),!.
indtype(log(_),L) :- check_logf(L),!.
indtype(exp,L) :- check_expp(L),!.


ieck_trigf([]) :- !.
ieck_trigf([H|T]) :- trigf(H),check_trigf(T),!.


ieck_logf([]) :- !.
ieck_logf([log(_,_)|T]) :- check_logf(T).

ieck_expp([]) :- !.
ieck_expp([_^_|T]) :- check_expp(T),!.

-igf(X) :- memberchk(X,[sin(_),cos(_),tan(_),sec(_),cosec(_),cot(_)]).
```

```
/* CONJ:

                                                    Bernard Silver
                                                    Updated: 26 February 1984
*/

conjecture_steps([Eqn|Example1],X,Unksteps,Conjlist,UnkList,Method,Method2) :-
        make_unk_list([Eqn|Example1],X,List1,UnkList),
        find_unknown_steps([Eqn|Example1],List1,Unksteps,NewUnk),
        conjecture_the_steps(Eqn,X,NewUnk,Conjlist,Method,Method2).


  % Find unknowns for each step
make_unk_list(A,X,B,List) :- make_unk_list1(A,X,X,B,List),!.

make_unk_list1([],_,_,[],[]) :- !.

make_unk_list1([false|T],Old,Old,[su(false,Old)|T1],[Old|T2]) :- !,
        make_unk_list1(T,Old,Old,T1,T2).

 % Change of Unknown
make_unk_list1([Y=Sub|T],Old,Unk,[su(Y=Sub,Y)|T1],[Y|T2]) :-
        freeof(Unk,Y),
        contains(Unk,Sub),
        make_unk_list1(T,Old,Y,T1,T2),
        !.

 % Change back
make_unk_list1([Y=X|T],Old,Unk,[su(Y=X,Old)|T1],[Old|T2]) :-
        freeof(Unk,Y),
        freeof(Old,X),
        contains(Old,Y),
        make_unk_list1(T,Old,Old,T1,T2),
        !.

make_unk_list1([H|T],Unk,X,[su(H,X)|T1],[X|T2]) :-
        contains(X,H),
        make_unk_list1(T,Unk,X,T1,T2),
        !.

make_unk_list1([H|T],Unk,X,[su(H,Unk)|T1],[Unk|T2]) :-
        contains(Unk,H),
        make_unk_list1(T,Unk,Unk,T1,T2),
        !.

 % Conjecture reasons for steps

conjecture_the_steps(_,_,[],[],L,L) :- reset,!.
conjecture_the_steps(Eqn,X,A,Conj,L,L1) :- !,
        check_no_conjectures(Eqn,X),
        ((A = [_] -> Flag = '.');Flag='s.'),
        writef('\nTrying to conjecture reasons for unknown step%w\n',[Flag]),
        conj_steps(A,Flag,Conj1,Namelist),
        zap_stored_conj(Conj1,Conj,Namelist,L,L1).

conj_steps([],Flag,[],[]) :- writef('\nEnd of Conjecture%w\n',[Flag]),!.
conj_steps([H|T],Flag,[Conj1|Conjlist],NewNames) :-
        write_step(H),
        conjecture_steps1(H,Conj,Conj1,Ok,Type),
        H = st(A,X,B,_),
```

```
        confirm_conj(Ok,Conj,X,A,B,Type,Name),
        (((var(Type),var(Name)) -> NewNames = Rest);
        NewNames =[change_name(Name,A,B)|Rest]),
        conj_steps(T,Flag,Conjlist,Rest),
        !.



conjecture_steps1(st(A,X,B,X),C,C1,yes,Name) :-
        retract(p_m_r(Name,A,B)),
        (Name = method(Name1);Name=Name1),
        !,
        known_method(_,_,_,Name,Type,_,_,_),
        find_parts_of_equation(Type,A,B,A1,B1),
        Diff =.. [Name1,A1,B1],
        writef('\nStep is an application of %w\n',[Name1]),
        remake_conjecture(X,A,B,NewC),
        (NewC=diff(Old,New);NewC=re(diff(Old,New))),
        writef('\n\nConjecture that\n\n%t\n\n\t\t->\n\n%t\n',[Old,New]),
        !,
        process_diff(diff(Diff),A1,B1,C,C1).

conjecture_steps1(st(A=B,X,C=B,X),Conj,Conj1,yes,Type) :-
        find_diff(X,A,C,Diff,Type,A=B,C=B),
        process_diff(Diff,A=B,C=B,Conj,Conj1),
        !.

conjecture_steps1(st(A,X,B,X),C,C1,F,Type) :-
        !,
        find_diff_hard(A,B,X,F,C,C1,Type,A,B).

conjecture_steps1(st(Old,X,New,Y),chunk,Conj,yes1,_) :-
        flag(method('Change of Unknown'),on,on),
        X\=Y,
        writef('\nSome sort of change of unknown seems to have happened.\n'),
        Conj = conj_mess('Change of Unknown',ch(X,Y),Old,New),
        !.

conjecture_steps1(st(Old,_,New,_),_,Conj,no,_) :-
        writef('\nUnable to conjecture reason for this step.\n'),
        Conj = conj_mess(fail(New),_,Old,New),
        !.


find_parts_of_equation(all,A,B,A,B) :- !.
find_parts_of_equation(part,A=B,C=B,A,C) :- !.
find_parts_of_equation(_,A,B,A,B) :- !.

remake_conjecture(X,A=C,B=C,Conj) :-
        functor(A,F,N),
        functor(B,F,N),
        decomp(A,[F|Arg1]),
        decomp(B,[F|Arg2]),
        find_diff3(X,F,Arg1,Arg2,Conj,_,A=C,B=C),
        !.

remake_conjecture(_,Old,New,diff(Old,New)).

  % Find differences
find_diff(X,Term1,Term2,Diff,Type,Eqn1,Eqn2) :-
```

```
        find_diff1(X,Term1,Term2,Diff,Type,Eqn1,Eqn2),
        output_diff(Diff),
        !.

find_diff1(X,Term1,Term2,Diff,Type,Eqn1,Eqn2) :-
        functor(Term1,F,N),
        functor(Term2,F,N),
        decomp(Term1,[F|Arg1]),
        decomp(Term2,[F|Arg2]),
        find_diff3(X,F,Arg1,Arg2,Diff,Type,Eqn1,Eqn2),
        !.

find_diff1(X,Term1,Term2,Diff,Type,Eqn1,Eqn2) :-
        find_diff2(X,Term1,Term2,Diff,Type,Eqn1,Eqn2),!.


find_diff2(X,Term1,Term2,diff(Diff),Name,Old,New) :-
        retract(p_m_r(Name,Old,New)),
        (Name = method(Name1);Name=Name1),
        writef('\nStep is an application of %w\n',[Name1]),
        !,
        Diff =.. [Name1,Term1,Term2].


  nd_diff2(_,A,B,diff(A,B),_,_,_,_) :- !.

find_diff3(X,F,Arg1,Arg2,re(Diff),Type,Eqn1,Eqn2) :-
        associative(F),
        remove_match(Arg1,Arg2,New1,New2),
        shift_numbers(New1,New2,NewAr1,NewAr2,F),
        remove_match1(NewAr1,NewAr2,A,B,F),
        find_diff2(X,A,B,Diff,Type,Eqn1,Eqn2),
        !.

find_diff3(X,_,Arg1,Arg2,Diff,_,_,_) :-
        only_diff(Arg1,Arg2,Diff),
        !.

 % Args differ in only one entry
only_diff(List1,List2,Diff) :- diff_list(List1,List2,[],[Diff]),!.

diff_list([],[],X,X) :- !.
 iff_list([H|T],[H|T1],Acc,Diff) :- diff_list(T,T1,Acc,Diff),!.
  f_list([H|T],[H1|T1],Acc,Diff) :- diff_list(T,T1,[diff(H,H1)|Acc],Diff),!.

remove_match([],List,[],List) :- !.
remove_match([H|T],List,New1,New2) :-
        delete_member(H,List,New),
        !,
        remove_match(T,New,New1,New2).

remove_match([H|T],List,[H|New1],New2) :-
        remove_match(T,List,New1,New2).

 % Delete term from list to get new list
delete_member(H,[H|T],T) :- !.
delete_member(H,[H1|T],[H1|T1]) :- delete_member(H,T,T1).


remove_match1([A],[B],NewT1,NewT2,_) :-
```

```
        A =.. [F|Arg1],
        B =.. [F|Arg2],
        associative(F),
        recomp(New1,[F|Arg1]),
        tidy(New1,NewT1),
        recomp(New2,[F|Arg2]),
        tidy(New2,NewT2),
        !.

remove_match1(List1,List2,TTerm1,TTerm2,F) :-
        recomp(Term1,[F|List1]),
        tidy(Term1,TTerm1),
        recomp(Term2,[F|List2]),
        tidy(Term2,TTerm2),
        !.

 % Move numbers from left hand side to right hand side of rewrite rule
shift_numbers([],L,[],L,_) :- !.
shift_numbers([H|T],L,L1,[H1|L2],F) :-
        number(H),
        !,
        inverse_number(F,H,H1),
        shift_numbers(T,L,L1,L2,F).

  ift_numbers([H|T],L,[H|L1],L2,F) :-
        shift_numbers(T,L,L1,L2,F).

inverse_number(+,Old,Inv) :- !,eval(-Old,Inv).
inverse_number(*,Old,Inv) :- !,eval(1/Old,Inv).

 % Harder differences

 % Recognise disjunctive solution

 % Work with both sides
find_diff_hard(A=B,C=D,X,yes,Conj,Conj1,Type,E1,E2) :-
        find_both_sides_difference(X,A=B,C=D,Conj,Conj1,Type,E1,E2).

find_both_sides_difference(X,A=B,C=D,Conj,Conj1,Type,E1,E2) :-
        B \==0,  % to stop hidden divisions
        D \==0,
        tidy_expr(A-B=0,New1=0),
        tidy_expr(C-D=0,New2=0),
        find_diff1(X,New1,New2,Diff,Type,E1,E2),
        Diff \= diff(_,_),
        Diff \= re(diff(_,_)),
        !,
        writef('\n\nConjecture that\n\n%t\n\n\t\t=\n\n%t\n',[New1,New2]),
        process_diff(Diff,A=B,C=D,Conj,Conj1).


 % Default
find_both_sides_difference(_,A=B,C=D,Co,Co1,_,E1,E2) :-
        writef('\n\nConjecture that\n\n%t\n\n\t\t->\n\n%t\n',[A=B,C=D]),
        process_diff(re(diff(A=B,C=D)),A=B,C=D,Co,Co1).


 % Process rule obtained from user
obtain_rule1(_,_,Rule,_,_) :-
        functor(Rule,F,_),
```

```
        F \= =>,
        !,
writef('\nYou have used the wrong connective, %t, in the rule.\n',[F]),
        (F=(=) -> output_real_rule(Rule);true),
        !,
        text_not_used,
        writef('\nPlease try again.\n'),
        fail.

output_real_rule(Rule) :-
        Rule =.. [_|Arg],
        R1 =.. [=>|Arg],
        num_writef('\n[Rule\n\n%t\n\nshould probably be\n\n%t]\n',[Rule,R1]),
        !.

obtain_rule1(X,Unk,L=>R,Cond,rule(_,Old,New)) :-
        X=Unk,
        occ(X,New,F),
        try_use_rule(Unk,Old,New1,L=>R,Cond,[],F),
        (tidy_expr(New1,New2);tidy(New1,New2)),
        match_check(New,New2),
        !.

obtain_rule1(_,_,_,_,_) :-
        writef('\n
This rule does not appear to apply to the conjecture, please try again.\n'),
        !,
        fail.


 % Replace information in stored conjectures with rule name

zap_stored_conj(Old,Old,[],L,L) :- !.
zap_stored_conj(Old,New,[change_name(Name,T1,T2)|T],L,L1) :-
        member(conj_mess(OldName,rule(A,B),T1,T2),Old),
        member(fail,L),
        !,
        perform_rename(OldName,NewName,T1,T2,Name),
        subst(conj_mess(OldName,rule(A,B),T1,T2) =
                conj_mess(NewName,rule(A,B),T1,T2),
        Old,Mid),
        first_subst(fail=NewName,L,L2),
        zap_stored_conj(Mid,New,T,L2,L1).


perform_rename(X,X,_,_,_) :- method(X),!.
perform_rename(X,method(X),_,_,_) :- method(method(X)),!.
perform_rename(_,user_rule(Name,T1,T2),T1,T2,Name) :- !.

first_subst(_,[],[]) :- !.
first_subst(A=B,[A|T],[B|T]) :- !.
first_subst(S,[H|T],[H|T1]) :-
        first_subst(S,T,T1).

 % Has this problem been tried before
check_no_conjectures(Eqn,X) :-
        difficult_once(Eqn,X),
        !,
writef('\nStill finding difficulties with this problem, entering break.\n'),
        reset,
```

```
        break,
        fail.

check_no_conjectures(Eqn,X) :- asserta(difficult_once(Eqn,X)).
```

```
/* CONFIR :

                                                    Bernard Silver
                                                    Updated: 26 February 1984
*/

 % Ask user to confirm conjectures
confirm_conj(no,_,_,_,_,_,_) :- !.
confirm_conj(Flag,Step,X,A,B,Type,Name) :-
        process_reply([y,n],act_ans(Ans,Step,X,Flag,A,B,Type,Name),
ns,'Confirm:',writef('\nIs the conjecture correct? (y/n)\n')).

act_ans(y,Step,X,yes,A,B,Type,Name) :- !,
        obtain_rule(Step,X,A,B,Type,Name).

act_ans(y,Step,X,yes1,_,_,_,_) :- !.
act_ans(n,_,_,_,_,_,_,_) :-
        writef('\nCan''t proceed then, entering break.\n'),
        !,
        break,
        fail.

act_ans(a,_,_,_,_,_,_,_) :- reset,writef('\n[Aborting]\n'),abort.


 % Just created new rule accounts for step
obtain_rule(_,X,Old,New,_,Name) :-
        asserted(user_rule(Name,_,_,_,_,_,_)),
        apply_new_rule1(X,Old,New,Name,tl),
        writef('\nRule %w is applicable\n',[Name]),
        !.

 % Ask user for rule
obtain_rule(Step,X,_,_,Type,Name) :-
        rule_text1,
        repeat,
        read_in_rule_conds([Unk1,Rule1,Cond]),
        know_conditions(Cond),
        tidy_withvars(Rule1,Rule),
        not not obtain_rule1(X,Unk,Rule,Cond,Step),
        ob_condition(X,Unk,Rule,Cond,Step,Type,Name),
        !.


 % Fill in conditions if none are given
ob_condition(X,Unk,Rule,Cond,Step,Type,Na) :-
        !,
        fill_in_values(X,Step,Occ1,Occ2,Cond1),
        obtain_the_rules1(X,Unk,Rule,Cond,Step,Occ1,Occ2,Cond1,Type,Na).

ob_condition(_,_,_,_,_,_,_,_) :- !,
        writef('\n[**Can not fill in values!!**]\n'),
        !,
        fail.

obtain_the_rules1(X,Unk,L=>R,Cond,Step,Occ1,Occ2,Cond1,Type,Name) :-
        is_type_rule(Type,X,Unk,L=>R,Cond,Step),
        get_functions_from_step(Step,X),
        ((var(Type);Type = method(_)) ->
        repeat,
```

```
        rule_text3,
        prompt(_,'Name:'),
        read_name(Name),
        check_name(Name),
        !,
        store_rule(Name,Unk,L=>R,Cond,Occ1,Occ2,Cond1,Type);
        true).

get_functions_from_step(rule(_,A,B),X) :-
        find_functions([A,B],X,add).


is_type_rule(Type,X,Unk,L=>R,Cond,rule(_,A=_,B)) :-
        (Type == 'Isolation';Type == "Function Stripping"),
        !,
        write_type('Isolation'),
        mark_mod_asserta((isolax(N,L,R) :- check_cond(Cond))).

is_type_rule(Type,_,Unk,L=>R,Cond,_) :-
        Type == 'Collection',
        write_type('Collection'),
        mark_mod_asserta((collax(Unk,L,R):- check_cond(Cond))).

is_type_rule(Type,_,Unk,L=>R,Cond,_) :-
        Type == 'Attraction',
        write_type('Attraction'),
        mark_mod_asserta((attrax(L&R,L,R,Cond))).  %probably wrong!


is_type_rule(_,_,_,_,_,_,_).

/*
 % Make rule into a homog rule
make_homog_rule(Type,Var,RTerm,Offend,Unk,Old,New,Cond) :-
        subst(Var=NewVar,Offend,Offend1),
        copy_ground(Old=New,Ground,Subst),
        position(Offend1,Ground,Posn),
        isolate(Posn,Ground,L=R),
        subst(Subst,L=R,NewEqn),
        NewEqn = (Offend=R1),
        write_type('Homogenization'),
        mod_asserta((rew_rule(Type,RTerm,Offend,R1,Unk) :- check_cond(Cond))).


check_name(Name) :-
        var(Name),
        !,
        writef('\nYou have used a variable,please try again.\n'),
        fail.

check_name(Name) :-
        user_rule(Name,_,_,_,_,_,_),
        !,
writef('\nThere is already a rule of name %t, please try another name.\n',[Name]),
        fail.

check_name(_).

 % Store the rules
store_rule(Name,Unk,X,Y,Occ1,Occ2,G,Type) :-
```

```
        flag(rules,_,yes),     % Mark that rules are stored
        assert(rule(Name,Unk,X,Y)),
        cond_add_to_list(Type,Name),
        mod_assertz(user_rule(Name,Unk,X,Y,Occ1,Occ2,G)).

:ond_add_to_list(Type,_) :- var(Type),!.
:ond_add_to_list(method(Name),Name1) :-
        write_type(Name),
        store_rules_list(Name,Name1).
 % Are the conditions understood?
:now_conditions([]).
:now_conditions([H|T]) :- know_conditions1(H),know_conditions(T),!.

:now_conditions1(H) :-
        functor(H,F,N),
        functor(New,F,N),
        current_predicate(F,New),
        !.

:now_conditions1(H) :-
        functor(H,F,N),
        writef('\nThe condition %t/%t is not understood.\n',[F,N]),
        process_reply([b,c,u],act_option(A,H),A,'Option:',option_text),
        !.



ict_option(b,H) :- !,
        writef('\n[Entering break]\n'),
        break,
        writef('\n[Leaving break]\n'),
        know_conditions1(H).

ict_option(c,H) :- !,
        repeat,
        writef('\nWhich file?\n'),
        prompt(_,'Filename:'),
        read(File),
        file_consult(File,H),
        !.


   _option(u,H) :- !,
        writef('\nConsulting [user].\n'),
        consult(user),
        know_conditions1(H).

file_consult(File,H) :-
        ((file_exists(File) -> reconsult(File),know_conditions1(H));
        (writef('\n[**File %w does not exist**] Please try again.\n',[File]),
        !,fail)),
        !.


 %  Fill in Occurrence information
fill_in_values(X,rule(_,Old,New),H1,H2,Cond) :-
        occ(X,Old,H3),
        occ(X,New,H4),
        fill_in_values1(H3,H4,H1,H2,Cond),
        !.
```

```
ill_in_values(_,_,_,_,[]).  % Play safe
ill_in_values1(A,A,B,B,[]) :- !. % Old and New have same number of occurrences
                                 % Guess this is always the case

ill_in_values1(A,B,C,D,[eval(C>D)]) :-  % Old greater than New, e.g. collect
        A>B,
        !.

ill_in_values1(A,B,C,D,[eval(C<D)]) :- % Old less than new ,e.g sin double
        A<B,                    % angle.
        !.

ill_in_values1(_,_,_,_,[]).  % Just in case! Should never happen


 % Check example if difficulty was found, to see if things are understood now
heck_example(X,[Eqn|_],Example) :-
        difficult_once(Eqn,X),
        !,
        check_example1(X,Example).

 % No problem last time, so end the output

   k_example(_,_,_) :-  terminate_text.

erminate_text :-
        writef('\n[Type generate_problem to test the schema]\n'),
        writef('\n[End of Output]\n').

heck_example1(X,Example) :-
        process_reply([y,n],check_example2(X,Example,Ans),Ans,'Reply:',
ritef('\nDo you want to run the problem again now? Type y or n.\n')).

heck_example2(_,_,n) :- !,reset,writef('\nOk.\n'),terminate_text.
heck_example2(X,Example,y) :-
        writef('\nRerunning example.\n'),
        initialize_screen,
        work_main(Example,X).


ark_mod_asserta(Rule) :-
        mod_asserta(Rule),
        mod_asserta(new_rule_stored(Rule)).
```

```
%   File   : FLAG
%   Author : Bernard Silver
%   Updated: 6 March 1984
%   Purpose: Flags for LP



 % Initialize flags
:-      flag(rules,_,no),       % Can not be changed by user
        flag(text1,_,notused),
        flag(text2,_,notused),
        flag(text2a,_,notused),
        flag(output,_,no),
        flag(loop,_,yes),
        flag(new_methods,_,yes),
        flag(cifer,_,yes).


 % Initialize method flags
:-      flag(method('Isolation'),_,on),
        flag(method('Factorization'),_,on),
        flag(method('Prepare for Factorization'),_,on),
        flag(method('Split DisJunctions'),_,on),
        flag(method('Polynomial Methods'),_,on),
        flag(method('Change of Unknown'),_,on),
        flag(method('Function Stirrins'),_,on),
        flag(method('Collection'),_,on),
        flag(method('Attraction'),_,on),
        flag(method('Homogenization'),_,on),
        flag(method('Logarithmic Method'),_,on),
        flag(method('Nasty Function Method'),_,on),
        flag(method(user_rule(_,_,_)),_,on).

        /* remove commands */

r(X) :- remove(X).

remove(X) :- var(X),!,variable_argument(remove).


 % Remove stored rules
  ove(rules) :- !,user_macro(is_rules,rules,r_rules).

is_rules :- rules_stored;new_rule_stored(_).

 % Remove specific rules
remove(rules(L)) :- user_macro(rules_stored,rules,remove_rules1(L)).

 % Remove new methods
remove(new_methods) :- !,
        user_macro(known_method_auto(_,_,_,_,_,_,_),'new methods',r_nm).

remove(schemas) :- !,
        user_macro(schemas, 'schemas',r_s).

% Remove rule of specific name, doesn't use user_macro
remove(rule(Name)) :-
        rules_stored,
        retract(user_rule(Name,A,B,C,D,E,F)),
```

```
        !,
        writef('\nRule %t retracted.\n',[Name]),
        still_rules.    % Check if any rules at all are left

remove(rule(Name)) :- !,
        (rules_stored   ->
        writef('\nNo rules of name %t are stored, nothing done.\n',[Name]);
        writef('\nNo rules are stored, nothing done.\n')).

 % Error
remove(X) :- command_error(remove,X).

r_rules :-
        abolish(user_rule,7),
        writef('\nUser rules removed.\n'),
        flas(rules,_,no),
        !.


 % Schemas
r_s :-
        abolish(known_method_schema,9),
        retract(schema(Name)),
        retract(method(Name)),
        fail.

r_s :-  writef('\nSchema methods removed.\n').

 % New methods
r_nm :-
        abolish(known_method_auto,7),
        abolish(ei,3),
        abolish(para,2),
        abolish(auto_rule,2),
        remove_table_entries,
        remove_new_method_markers,
        writef('\nAll new (non-schema) methods removed.\n'),
        (not schemas;
        writef('\n[Use ''remove schemas'' to remove schema methods.]\n')),
        !.


remove_table_entries :-
        must_satisfy(Cond,List),
        member(method(auto(X)),List),
        delete(auto(X),List,New),
        retract(must_satisfy(Cond,List)),
        assert(must_satisfy(Cond,New)),
        fail.

remove_table_entries.

remove_rules1([]) :- writef('\nFinished.\n').
remove_rules1([H|T]) :- remove(rule(H)),remove_rules1(T).



 % Still user rules around?
still_rules :- user_rule(_,_,_,_,_,_,_),!.
still_rules :-  flas(rules,_,no),writef('\n[No rules are left]\n').
```

```
% Define various flags
rules_stored :- flag(rules,yes,yes).

allowed_new_methods :- flag(new_methods,yes,yes).

schemas :- known_method_schema(_,_,_,_,_,_,_,_,_).

nocifer :- flag(cifer,no,no).

% Change flags

e(X) :- enable(X).

enable(Var) :- var(Var),!,variable_argument(enable).

enable([]) :- !.
enable([H|T]) :- !,enable(H),enable(T).

enable(allmethods) :- !, e_a_m.
enable(Flag) :- flag2(Flag,yes),!.
enable(Flag) :- method(method(Flag)),flag2(method(method(Flag)),on),!.
e  ble(Flag) :- method(Flag),flag2(method(Flag),on),!.
   ble(Method) :- short_name(Method,New),!,enable(New).
enable(Error) :- no_such_flag(Error,enable),!.

d(X) :- disable(X).

disable(Var) :- var(Var),!,variable_argument(disable).

disable([]) :- !.
disable([H|T]) :- !,disable(H),disable(T).

disable(allmethods) :- !,d_a_m.
disable(Flag) :- flag2(Flag,no),!.
disable(Flag) :- method(method(Flag)),flag2(method(method(Flag)),off),!.
disable(Flag) :- method(Flag),flag2(method(Flag),off),!.
disable(Method) :- short_name(Method,New),!,disable(New).
disable(Error) :- no_such_flag(Error,disable),!.


  g2(Flag,X) :- flag(Flag,X,X),!,flag_error(X).
flag2(Flag,X) :- flag(Flag,_,X),flag_message(Flag,X),!.


flag_message(loop,warn) :- !,writef('\nLoop checker will now  warn.\n').
flag_message(loop,X) :-
        (X=yes->Word=on;X=no,Word=off),
        writef('\nLoop checker turned %w.\n',[Word]),
        !.

flag_message(output,X) :-
        set_word(X,W),
        writef('\nThe example will now%wbe output.\n',[W]),
        !.

flag_message(cifer,X) :-
        set_word(X,W),
```

```
        writef('\nYour terminal is%wa cifer.\n',[W]),
        !.

flas_messase(new_methods,X) :-
        set_word(X,W),
        writef('\nNew methods are now%wallowed.\n',[W]),
        (X=yes->enable_new_methods;disable_new_methods),
        !.

flas_messase(method(Method),X) :-
        (X=on;X=off),
        method(Method),
        (Method=user_rule(_,_,_) -> N='Apply New Rule';N=Method),
        writef('\nMethod %w turned %t.\n',[N,X]),
        (method_list_on([_|_],_);writef('\n[**No methods are left**]\n')),
        !.

e_a_m :-
        method_list_on(_,ListOff),
        e_a_m1(ListOff),
        !.

e  `_m1([]) :- !,writef('\nNo methods are disabled, nothing done.\n').

e_a_m1(List) :-
        e_a_m2(List),
        !.

e_a_m2([]) :- writef('\nAll methods have been turned on.\n'),!.
e_a_m2([H|T]) :-
        enable(H),
        !,
        e_a_m2(T).

d_a_m :-
        method_list_on(ListOn,_),
        d_a_m1(ListOn),
        !.

d_a_m1([]) :- !,writef('\nNo methods are disabled, nothing done.\n').

r  `_m1(List) :-
        d_a_m2(List),
        !.

d_a_m2([]) :- writef('\nAll methods have been turned off.\n'),!.
d_a_m2([H|T]) :-
        disable(H),
        !,
        d_a_m2(T).


set_word(yes,' ') :- !.
set_word(no,' not ') :- !.


show_all_flass([]) :- !.
show_all_flass([H|T]) :-
        flas(H,Old,Old),
        interpret_value(H,Old),
```

```
        !,
        show_all_flags(T).

interpret_value(rules,Val) :- !,
        (Val=yes -> W=' ';W=' no '),
        writef('\nThere are%wrules stored.\n',[W]).

interpret_value(loop,Val) :- !,

        writef('\nThe loop flag is set to %t.\n',[Val]).

interpret_value(Flag,Val) :-
        (Val=yes->W=enabled;W=disabled),
        !,
        writef('\nThe %t flag is %w.\n',[Flag,W]).

 % Change loop flag when needed by solve, do nothing if flag is set to no
loop_flag(Value) :-
        flag(loop,Old,Old),
        ((Old==no;Old==warn);flag(loop,_,Value)),
        !.

 * Reinitialize runs by removing difficulty marker and stored equations

reset :-
        abolish(seen_eqn,1),
        abolish(p_m_r,3),
        abolish(warning,2),
        text_not_used,
        abolish(just_created,1),
        (retract(difficult_once(_,_));true),
        !.

 % Abolish asserted markers
reset1 :- mod_abolish(asserted,1),reset.

text_not_used :-
        flag(text1,_,notused),
        flag(text2,_,notused),
        flag(text2a,_,notused),
        !.


 % Flag errors

 % Flag already has the new value
flag_error(X) :-
        ((X=yes; X = on) -> W = enabled;W=disabled),
writef('\n Already %w, nothing done.\n',[W]).

 % No flag or method of that name exists
no_such_flag(Err,Ty) :-
        writef('\n[**No such flag or method as %t, %w ignored**]\n',[Err,Ty]),
        valid_commands(Ty,List),
        output_valid_list(Ty,List).

 % Argument is  a variable
variable_argument(Type) :-
        writef('\n[**Variable in %w command**, command ignored]\n',[Type]).
```

```
;   File   : CHAR
;   Author : Bernard Silver
;   Updated: 26 February 1984
;   Purpose: Character Reading for LP


:- public
        convert_chars/2,
        read_in_conditions/1,
        read_in_rule_conds/1,
        read_in_line/1,
        read_in_rule/1,
        read_in_unknown/1,
        read_name/1,
        syntax_check/1.

:- mode
        convert_chars(+,-),
        make_reply_from_chars(+,+,+,-),
        map_put(+),
        mod_know_conditions(+),
        read_conditions(_),
        read_unknown(-),
        read_in_conditions(-),
        read_in_line(-),
        read_in_line1(+,-),
        read_in_rule(-),
        read_in_rule_conds(-),
        read_in_rule_conds_cont(+,-),
        read_in_unknown(-),
        read_name(-),
        read_rule(-),
        read_rest(+,?),
        read_rest1(+,+,?),
        read_rest_cond(+,?),
        read_rest_unk(+,?),
        read_rest_unk1(+,?),
        read_rest1(+,?),
        syntax_check(+),
        syntax_check(+,+,+).

    _in_rule_conds(Reply) :-
        read_in_rule(Rule),
        read_in_rule_conds_cont(Rule,Reply),
        !.

 % List starts "rule(", remainder is Name)
read_in_rule_conds_cont([114,117,108,101,40|L],Reply) :-
        append(NameChars,")",L),
        name(Name,NameChars),
        ((call(rule(Name,Unk,Rule,Cond)),Reply=[Unk,Rule,Cond]);
(writef('\nRule %t not found, please try again.\n\n',[Name]),!,fail)),
        !.

read_in_rule_conds_cont("a",_) :- !,
        writef('\n[Aborting]\n'),
        call(abort).
read_in_rule_conds_cont("b",_) :- !,
```

```
        writef('\n[Entering Break]\n'),
        call(break),
        writef('\n[Leaving Break]\n'),
        fail.
read_in_rule_conds_cont(Rule,Reply) :-
        read_in_unknown(Unk),
        read_in_conditions(Cond),
        !,
        make_reply_from_chars(Rule,Unk,Cond,Reply).

read_in_rule(Rule) :-
        repeat,
        writef('\nEnter Rule\n'),
        prompt(_,'Rule:'),
        read_rule(Rule),
        syntax_check(Rule),
        !.

read_in_unknown(Unk) :-
        repeat,
        writef('\nEnter Unknown\n'),
        call(rule_text2a),
        prompt(_,'Unknown:'),
        read_unknown(Unk),
        append("u(",Unk,Mid),
        append(Mid,")",New),
        syntax_check(New),
        !.

read_in_conditions(Cond) :-
        repeat,
        writef('\nEnter List of conditions\n'),
        call(rule_text2),
        prompt(_,'Conditions:'),
        read_conditions(Cond),
        syntax_check(Cond),
        mod_know_conditions(Cond),
        !.

read_rule(Rule) :-
        get0(C),
        read_rest(C,Rule),
        !.
    _conditions(Cond) :-
        get0(C),
        (C \= 31; C=31,writef('\n[No conditions]\n')),
        read_rest_cond(C,Cond),
        !.
read_in_line(Rule) :-
        get0(C),
        read_in_line1(C,Rule),
        !.

read_in_line1(31,end) :- !.
read_in_line1(97,_) :- !,
        writef('\n[Aborting]\n'),
        call(abort).

read_in_line1(C,Rule) :-
        read_rest(C,Rule1),
```

```
        syntax_check(Rule1),
        convert_chars(Rule1,Rule),
        !.
read_name(Name) :-
        get0(C),
        read_rest_unk1(C,List),
        syntax_check(List),
        convert_chars(List,Name),
        !.

read_unknown(Unk) :-
        get0(C),
        (C\=31;C=31,writef('\n[No distinguished unknown]\n')),
        read_rest_unk(C,Unk),
        !.

read_rest(C,Unk) :-
        get0(C1),
        read_rest1(C,C1,Unk),
        !.

read_rest1(46,31,[]) :- !.  % Delimited by .<cr>
read_rest1(46,32,[]) :- !.  % Delimited by .<space>
read_rest1(C,C1,[C|L]) :-
        get0(C2),
        read_rest1(C1,C2,L).


read_rest_unk(31,"_") :- !.
read_rest_unk(C,Rest) :- read_rest_unk1(C,Rest),!.

read_rest_cond(31,"[]") :- !.
read_rest_cond(C,Rest) :- read_rest_unk1(C,Rest),!.

read_rest_unk1(31,[]) :- !. % delimited by <cr>
read_rest_unk1(46,[]) :- !. % delimited  .
read_rest_unk1(C,[C|L1]) :-
        get0(C1),
        !,
        read_rest_unk1(C1,L1).

syntax_check(Input) :-
        seeing(OldSee),
        telling(OldTell),
        syntax_check(Input,OldSee,OldTell).

syntax_check([31],OldSee,OldTell) :- !,
        fileerrors,
        seen,
        see(OldSee),
        told,
        tell(OldTell).

syntax_check(Input,OldSee,OldTell) :-
        tell('lp1234.tmp'),
        map_put(Input),
        put("."),
        nl,
        told,
        tell(OldTell),
```

```
        nofileerrors,
        see('lp1234.tmp'),
        read(Term),
        Term \= 'end_of_file',
        !,
        rename('lp1234.tmp',[]),
        seen,
        fileerrors,
        see(OldSee),
        !.

syntax_check(_,OldSee,OldTell) :-
        seen,
        see(OldSee),
        told,
        tell(OldTell),
        fileerrors,
        fail.

map_put([]) :- !.
map_put([H|T]) :-
        put(H),
        !,
        map_put(T).

make_reply_from_chars(Rule,Unk,Cond,Reply) :-
        telling(OldTell),
        tell('lp1234.tmp'),
        put("["),
        map_put(Unk),
        put(","),
        map_put(Rule),
        put(","),
        map_put(Cond),
        put("]"),
        put("."),
        nl,
        told,
        tell(OldTell),
        seeing(OldSee),
        see('lp1234.tmp'),
        read(Reply),
        rename('lp1234.tmp',[]),
        seen,
        see(OldSee),
        !.

mod_know_conditions("[]") :- !.
mod_know_conditions(Cond) :-
        convert_chars(Cond,Cond1),
        know_conditions(Cond1),
        !.

convert_chars(List,Structure) :-
        seeing(OS),
        telling(OT),
        tell('LP1234.tmp'),
        map_put(List),
        put("."),
        nl,
```

```
    told,
    tell(OT),
    see('LP1234.tmp'),
    read(Structure),
    seen,
    see(OS),
    !.
```

```
%    File    : OUT
%    Author  : Bernard Silver
%    Updated: 6 March 1984
%    Purpose: Output for LP



 % Output routines etc

 % Report the reasons and get list of method used
report_steps([H|List],X,Method,[start|Good]) :-
        writef('\n\n\tOperator Identification Complete.\n'),
        writef('\n\nSolving\n\t\t%t\nfor %t\n\n',[H,X]),
        writef('\nThe steps were as follows.\n'),
        report_steps1(List,Method,Mid),
        construct_good_example_list(Mid,Good).

report_steps1([],[],[]) :- writef('\n\tEnd of Example.\n'),!.

report_steps1([H|T],[message(M)|Rest],[M|Method]) :-
        output_mess(M),
        warn_if_possible_missing_rule(M,Warn,H),
        writef('\n %t.\n',[H]),
        output_solution(Rest,NewRest),
        report_steps1(T,NewRest,Method),
        !.

 % Check to see if missing rule messages exist for this Eqn

warn_if_possible_missing_rule(fail,Warn,Eqn) :-
        retract(warning(Warn,Eqn)),
        !,
        call(Warn).

warn_if_possible_missing_rule(_,_,_) :- !.



output_solution([],[]).
 utput_solution([message(M)|T],List) :-
        sol_mess(M),
        !,
        output_solution(T,List).

output_solution(T,T).

sol_mess(false) :- !,
        writef('\n[No (real) solutions]\n').


sol_mess(true) :- !,
        writef('\n[Expression is an identity]\n').

sol_mess(nosol) :- !.

sol_mess(solution) :- !,
        writef('\n[Solution]\n').

output_mess(removecf) :- !,
```

```
        writef('\nRemoving constant factor\n').

output_mess(ff) :- !,
        writef('\nSolving first factor\n').

output_mess(nf) :- !,
        writef('\nSolving next factor\n').

output_mess(cve) :- !,
        writef('\nSolving Changed Variable Equation\n').

output_mess(ses) :- !,
        writef('\nSolving Substitution Equation').

output_mess(fail) :- !,
        writef('\nStep not understood, continuing processing\n').

output_mess(user_rule(Name,_,_)) :- !,
        writef('\nStep uses new rule %w\n',[Name]).

output_mess(Name) :-
        writef('\nApplying %w\n',[Name]).

  % Check is example should be output
show_the_example(Example,X,Example1) :-
        asserta(last_example(Example,X)),
        (unknown(X);asserta(unknown(X))),
        (flag(output,no,no);
  (writef('\nWorking on the following example, %t is the unknown.\n' [X]),
        output_example(Example))),
        maptidy_example(X,Example,Example1,work),
        !.

  % Output tidied example

output_example([]) :- writef('\n\t[End of Worked Example]\n'),!.
output_example([H|T]) :-
        writef('\n\t%t.\n',[H]),
        !,
        output_example(T).


 % Find difficulties in the example
find_unknown_steps(Example1,Example2,OldUnk,Steps) :-
        get_unknown_steps(OldUnk,List),
        find_unknown_steps1(Example1,Example2,List,Steps),
        !.

get_unknown_steps(OldUnk,[us(Step)|T]) :-
        member(unknown_steps(Step),OldUnk),
        delete_one(unknown_steps(Step),OldUnk,Rest),
        !,
        get_unknown_steps(Rest,T).

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


find_unknown_steps1(_,_,[],[]) :- writef('\nNo difficulties encounte ed.\n'),!.

find_unknown_steps1(Example1,Example2,List,Steps) :-
```

```
        find_unknown_steps2(Example1,Example2,List,Steps),
        !.

find_unknown_steps2(_,_,[],[]) :- !.
find_unknown_steps2(Ex1,Ex2,[us(Step)|T],[st(Step1,X,Step,Y)|T1]) :-
        nmember(Step,Ex1,N),
        nmember(su(Step,Y),Ex2,N),
        M is N - 1,
        nmember(Step1,Ex1,M),
        nmember(su(Step1,X),Ex2,M),
        !,
        find_unknown_steps2(Ex1,Ex2,T,T1).


 % Output for conjectures
write_step(st(A,_,B,_)) :-
        writef('\nWorking on step from \n\n%t\n\nto\n\n%t\n',[A,B]),
        !.

 % Process conjectures for storing
process_diff(re(diff(X)),T1,T2,rule(F,A,B),
conj_mess(F,rule(A,B),T1,T2)) :-
        functor(X,F,2),
        !,
        arg(1,X,A),
        arg(2,X,B),
        !.
process_diff(diff(X),T1,T2,rule(F,A,B),
conj_mess(F,rule(A,B),T1,T2)) :-
        functor(X,F,2),
        !,
        arg(1,X,A),
        arg(2,X,B),
        !.

process_diff(re(X),T1,T2,Rule,Conj) :- !,process_diff(X,T1,T2,Rule,C nj).


process_diff(diff(A,B),T1,T2,rule(_,A,B),
conj_mess(equal,rule(A,B),T1,T2)) :- !.

   Output conjectures

output_diff(re(diff(A,B))) :-
        tidy_up(A=B,C,D),
        writef('\nConjecture that \n\n%t\n\n\t\t = \n\n%t.\n',[C,D])
        !.

output_diff(re(diff(X))) :-
        functor(X,_,2),
        arg(1,X,A),
        arg(2,X,B),
        tidy_up(A=B,C,D),
        writef('\n%t =>\n\n%t.\n',[C,D]),
        output_diff(diff(X)).

output_diff(diff(X)) :-
        functor(X,F,_),
        writef('\nAppears to be a new %w step.\n',[F]),
        !.
```

```
output_diff(diff(A,B)) :-
        tidy_up(A=B,C,D),
        writef('\nConjecture that \n\n%t\n\n\t\t = \n\n%t.\n',[C,D])
        !.


 % Output message telling where the new rule is being stored
write_type(Ty) :- writef('\nRule is being stored as a %w rule.\n',[T ]).


num_writef(Format,Vars) :-
        numbervars(Vars,0,_),
        writef(Format,Vars),
        fail.

num_writef(_,_).

paraphrase_goal(W,M1,M2) :-
  writef('\n%w to apply the indicated next method of %t,\n %w\n',[W, 2,M1]).

 % Get wording right for disjunct solving
 isjunct_writef([X=A,X]) :-
        %       freeof(X,A),  % Should be anyway
        !,
        writef('\n%t = %t is a solution.\n',[X,A]).

 disjunct_writef([A,X]) :- writef('\nSolving %t for %t.\n',[A,X]).

mod_writef(Ans) :-
        writef('\nAnswer is:\n'),
        mod_writef1(Ans).

mod_writef1(A#B) :- !,
        writef('%t  v \n',[A]),
        mod_writef1(B).
mod_writef1(A) :-
        writef('%t\n\n',[A]),
        extra_message(A),
        !.

 xtra_message(true) :- !,
        writef('\n[Expression is an identity]\n').

extra_message(false) :- !,
        writef('\n[No real values satisfy the equation]\n').

extra_message(_).

  % Get the wording right for the isolate case
mod_iso_trace(false) :- !,
        writef('\nNo real values satisfy the equation.\n').

mod_iso_trace(true) :- !,
        writef('\nExpression is an identity.\n').

mod_iso_trace(New) :-
        writef('\nIsolating Equation to obtain \n\n%t.\n',[New]).
```

```
 % Text for rule input

rule_text1 :-
        flag(text1,used,used),
        !.

rule_text1 :-
        flag(text1,_,used),
        writef('\n
Enter the rewrite rule, in the form Old => New, terminate with a "."
(You will then be prompted for further details)\n').

rule_text1 :-
        flag(text1,_,notused),
        fail.

rule_text2 :-
        flag(text2,used,used),
        !.

rule_text2 :-
        flag(text2,_,used),
        writef('Terminate with <cr> (Just type <cr> if none.)\n').


rule_text2 :-
        flag(text2,_,notused),
        fail.

rule_text2a :-
        flag(text2a,used,used),
        !.

rule_text2a :-
        flag(text2a,_,used),
writef('Terminate with <cr> (Just type <cr> if no distinguished unkn wn.)\n').

rule_text2a :-
        flag(text2a,_,notused),
        fail.

  le_text3 :- writef('\nGive a name for this rule (use an atom).\n')


 % Text for when conditions are not understood
option_text :-
        writef('\nType


        a  : to abort,

        b  : to break,

        c  : to (re)consult the definition from a file,

        u  : to consult user.\n').


 %  Show the steps that led to the solution
solve_steps(L) :-
```

```
        writef('\nSolve steps were as follows:\n'),
        solve_steps1(L).

solve_steps1([]) :- !,writef('\n[End of trace]\n').
solve_steps1([H|T]) :-
        writef('\n%t\n',[H]),
        !,
        solve_steps1(T).

method_list_on(ListOn,ListOff) :-
        bagof(X,method1(X),List1),
        submethod_on(List1,ListOn,ListOff).

method1(X) :- method(X),X\=user_rule(_,_,_).

submethod_on([],[],[]) :- !.
submethod_on([Name|T],[Name|T1],T2) :-
        flag(method(Name),on,on),
        !,
        submethod_on(T,T1,T2).

submethod_on([Name|T],T1,[Name|T2]) :- submethod_on(T,T1,T2).



 % Pretty print listings
mod_list_listing([]) :- !.
mod_list_listing([H|T]) :-
        mod_listing(H),
        !,
        mod_list_listing(T).

mod_listing(X) :-
        current_predicate(X,Goal),
        clause(Goal,Body),
        mod_listing1(Goal,Body),
        fail.

mod_listing(_).

mod_listing1(Goal,true) :- !,
        num_writef('\n%t.\n',[Goal]).

mod_listing1(Goal,Body) :- !,
        num_writef('\n%t :- %t.\n',[Goal,Body]).

 % quick form

m(X) :- mod_listing(X).

 % Show Commands

s(X) :- show(X).

show(X) :- var(X),!,variable_argument(show).

 % Show rules to user
show(rules) :-  !,
        user_macro(is_rules,rules,s_rules_new).
```

```
s_rules_new :-
        s_rules_new1,
        s_r_n1.

s_rules_new1 :-
        user_rule(Name,Unk,L=>R,Cond,_,_,_),
        s_r_n(Name,Unk,L,R,Cond),
        fail.
s_rules_new1.

s_r_n(Name,Unk,L,R,Cond) :-
        (Cond=[]-> C = none;C=Cond),
        (occurs_in(Unk,[L,R,Cond]) ->U=Unk;U='_'),
        num_writef('\nRule %w\n%t => %t\n\nUnknown:%t\n\nConditions: t\n\n',
        [Name,L,R,U,Cond]),
        !.

s_r_n1 :-
        new_rule_stored(_),
        !,
        writef('\nRules added to initial methods\n'),
        s_r_n1a.
  r_n1.

s_r_n1a :-
        new_rule_stored(Rule),
        num_writef('\n%t\n',[Rule]),
        fail.
s_r_n1a.

 % Show which rules are used by each method

show(rules_list) :- !,
        user_macro(setof(X,method(method(X)),S),'new methods',s_meth d(S)).

 % Show schemas
show(schemas) :- !,
        user_macro(schemas,schemas,s_k_m_s).

 % Show new methods
show(new_methods) :- !,
        user_macro(bagof(M,method(method(M)),Bag), 'new methods',s_n m(Bag)).

 % Show trace to user
show(solve) :- user_macro(store_we(worked(L,_)),trace,solve_steps(L) .

 % Show settings of flags  (Rest of code in FLAG)

show(flags) :-
        writef('\nThe flags are set as follows:\n'),
        show_all_flags([cifer,output,loop,new_methods,rules]).


% Show which methods are available
show(methods) :-
        method_list_on(BagOn,BagOff),
        write_bag(enabled,BagOn),
        write_bag(disabled,BagOff),
        !.
```

```
write_bag(enabled,[]) :- !,
        writef('\n[**No methods are enabled**]\n').

write_bag(_,[]) :- !,
        writef('\nNo methods are disabled.\n').

write_bag(Word,Bag) :-
        single_plural(Bag,W,W1),
        writef('\nThe following method%w %w %w:\n\n',[W,W1,Word]),
        output_list_write(Bag),
        !.

output_list_write([]) :- !.
output_list_write([H|T]) :-
        writef('\n\t\t%w\n',[H]),
        !,
        output_list_write(T).

 % Illegal show command
show(X) :- !,command_error(show,X).

s_method(Set) :-
        (length(Set,1) -> W=is,W1=' ';W=are,W1='s '),
        writef('\nThe rule list%w%w as follows:\n',[W1,W]),
        s_method1(Set,W1).

s_method1([],Word) :- writef('\n[End of rule list%w]\n',[Word]),!.
s_method1([H|T],W) :-
        (setof(Rules,auto_rule(H,Rules),Set);Set = none),
        writef('\nMethod %t: %t\n',[H,Set]),
        s_method1(T,W).

s_n_m(Bag) :-
        length(Bag,N),
        (N=1->W=' ',W1='s:';W='s ',W1=':'),
        writef('\nThe following new method%wexist%w\n',[W,W1]),
        s_n_m1(Bag).


s_n_m1([]) :- ttynl,!.
s_n_m1([Method|T]) :-
        (((known_method_auto(_,_,_,method(Method),_,_,[A|_]),
        arg(1,A,Name)) ->
        writef('\nMethod %t has applicable next method %w\n',[Method Name]));
        (schema(method(Method)),
        writef('\nMethod %t is a schema method.\n',[Method]))),
        !,
        s_n_m1(T).

s_k_m_s :-
        known_method_schema(X,Eqn,_,Name,schema(S,E,U),Type,_,_,_),
        writef('\n\t\tSchema Method %t\n\n',[Name]),
        output_schema(S),
        writef('\nType:%w\n\nGenerating Equation:%w\n\nUnknown:%t\n\ ',
[Type,E,U]),
        fail.

s_k_m_s.

output_schema(S) :-
```

```
        writef('\nMethod                                                    Purp se\n'),
        output_schema1(S).

output_schema1([H|T]) :- !,
        o_s2(H),
        writef('\n_____________________________________________________ ____\n'),
        output_schema1(T).

output_schema1([]) :- writef('\n[End of Schema]\n').

o_s2([]) :- !.
o_s2([conditions(H,_,unsat(Cond,_,_))|T]) :-
        assemble_preconds(H,Cond,List),
        make_name_atomic(H,H1),
        writef('\n%25l\t\t<',[H1]),
        write_horiz_list(List),
        !,
        o_s2(T).

write_horiz_list([H]) :- !,
        writef('%w>\n',[H]).
write_horiz_list([H|T]) :-
        writef('%w, ',[H]),
        !,
        write_horiz_list(T).

make_name_atomic(method(H),H) :- !.
make_name_atomic(H,H).

assemble_preconds(M,[],[H]) :-
        (key_method(M) -> H='key method';H=none),
        !.
assemble_preconds(_,List,Conds) :-
        assemble_preconds(List,Conds).

assemble_preconds([],[]).
assemble_preconds([H|T],[Term|T1]) :-
        functor(H,F,N),
        concat(F,/,Mid),
        concat(Mid,N,Term),
        !,
        assemble_preconds(T,T1).

  % Writeout Commands

w(X) :- writeout(X).

writeout(X) :- var(X),!,variable_argument(writeout).

 % Write the trace to a file


writeout(solve) :- user_macro(store_we(worked(L,_)),trace,w_solve_st ps(L)).

 % Write user rules to a file

writeout(rules) :- user_macro(rules_stored,rules,w_rules).

writeout(schemas) :- user_macro(schemas,schemas,w_schemas).
```

```
writeout(new_methods) :- !,
        user_macro(bagof(M,method(method(M)),Bag), 'new methods',w_n m(Bag)).

 % Error
writeout(X) :- command_error(writeout,X).

w_schemas :-
        file_name1(schemas,File),
        tell(File),
        mod_listing(known_method_schema),
        told,
        writef('\nSchemas written to file %t.\n',[File]).

w_rules :-
        file_name1(rules,File),
        tell(File),
        write(':- flag(rules,_,yes).'),
        mod_list_listing([user_rule]),
        mod_list_new_rules,
        told,
        writef('\nRules written to file %t.\n',[File]).



mod_list_new_rules :-
        new_rule_stored(Rule),
        writef('\nasserta((%t)).\n',[Rule]),
        fail.

mod_list_new_rules.

w_solve_steps(L) :-
        file_name1(trace,File),
        tell(File),
        solve_steps(L),
        told,
        writef('\nTrace written to file %t.\n',[File]).

w_n_m(Bag) :-
        file_name1('new methods',File),
        tell(File),
        m known_method_auto,
        told.

 % writeout, show or remove Thing, after calling Test to check all i  OK
 % Thing can be trace, rules, new methods, or schemas.
user_macro(Test,_,Call) :- call(Test),!,call(Call).
user_macro(_,Thing,_) :-
        (Thing=trace -> IsAre = is;IsAre=are),
        writef('\nNo %w %w stored, nothing done.\n',[Thing,IsAre]).

 % Illegal arguments
command_error(Type,X) :-
        writef('\n[**%w is not a valid argument in %w command**]\n', X,Type]),
        valid_commands(Type,List),
        output_valid_list(Type,List).

output_valid_list(Type,List) :-
        writef('\nValid arguments for the %w command are:\n',[Type])
        output_v_list(List).
```

```
output_v_list([]) :- ttynl,!.
output_v_list([H|T]) :-
        writef('\n%w',[H]),
        output_v_list(T).

valid_commands(remove,[rules,'rule(List)','rule(Name)',schemas,
new_methods]).

valid_commands(show,[flags,methods,rules_list,schemas,rules,new_meth ds,solve]).

valid_commands(writeout,[solve,rules,schemas,new_methods]).

valid_commands(enable,[output,cifer,new_methods,allmethods,'MethodNa e']).

valid_commands(disable,[output,cifer,new_methods,allmethods,'MethodN me']).


process_reply(List,Call,Var,Prompt,Text) :-
        repeat,
        call(Text),
        prompt(_,Prompt),
        get0(Var1),
        name(NewVar,[Var1]),
        process_reply1(List,Call,Var,Var1,NewVar),
        !.

process_reply1(_,_,_,Var,_) :-
        Var >= "A", Var =< "Z",
        !,
        writef('\nYou have used a variable, please try again.\n'),
        fail.


process_reply1([H|List],Call,Var,Var1,NewVar) :-
        ((Var1 = 31 -> Var = H,writef('\n[Assuming default value: %w \n',[H]));
        (skip(31),member(NewVar,[H|List]),Var=NewVar)),
        !,
        call(Call),
        !.


 rocess_reply1(_,_,_,_,a) :- !,writef('\n[Aborting]\n'),abort.
process_reply1(_,_,_,_,b) :-
        !,
        writef('\n[Entering break]\n'),
        break,
        writef('\n[Leaving break]\n'),
        fail.

process_reply1(L,_,_,_,_) :-
writef('\nNot a valid response!  Use one of \n%l\n\nPlease try again \n',[L]),
        fail.

mod_subst_mesg(T=Var,Old,New) :-
        subst_mesg(T=Var,Old,New),
writef('\n\nApplying substitution %t = %t to obtain\n\n%t\n\n',[Var, ,New]),
        !.

 % Get plural of word is list contains more than one element
```

```
single_plural([_],"",is) :- !.
single_plural([_,_|_],"s",are) :- !.



 % Construct a list of Methods without extra comments such as "solut on"
construct_good_example_list([],[]) :-!.
construct_good_example_list([M|T],T1) :-
        comment_name(M),
        !,
        construct_good_example_list(T,T1).

construct_good_example_list([H|T],[H|T1]) :- !,
        construct_good_example_list(T,T1).

comment_name(false) :- !.
comment_name(true) :- !.
comment_name(nosol) :- !.
comment_name(solution) :- !.

        % Commands for the cifer terminal

  itialize_screen :-
        clear,
        time(X),
        writef("\n[Run begins at %w]\n",[X]).

clear :-
        (nocifer;
        ttyput(283),                    %   Clear screen ESC-J
        ttyput(330)),
        !.
```

```
/* INTERP :  Misc interpreted code

                                                    Bernard Silver
                                                    Updated: 6 March 198

Includes code by Alan, Lawrence, Leon and Richard

*/
% Get help.
help :-
        writef('
Help is available either at monitor level or from within this progra .
At monitor level see file DSKB:LP.HLP[400,444].\n'),
        process_reply([y,n],help1(Ans),Ans,'More Help:',writef('\n
Do you want to continue with help in this program? (y./n.)\n')),
        !.

help1(y) :- !,
        (current_predicate(give_help,_);
        writef('\n[Loading help file]\n'),
        compile('learn:mdhelp.pl')),
        ttynl,
        !,
        give_help(lp).

help1(n) :-
writef('\nOK.  (LP.HLP[400,444] can also be examined from this progr m by typing
\nty ''mec:lp.hlp''\nin response to the Prolog prompt) \n'),
        !.


                        % Redo last equation

sredo :-
        retract(last_equation(Equation)),
        !,
        solve(Equation).

sredo :- writef('\nNo previous equation, nothing done.\n').

 % File read in
wep :- ['learn:wep'].
rules :- ['learn:rules'].

 % Normal state
normal :- wep,rules.
examples :- normal.

go :-
        writef('\nGo option isn''t supported, type help for help.\n' ,
        !.


 % Quick definitions to replace the interval package
negative(X) :- tidy(X,Y),number(Y),!,eval(Y<0).

non_neg(X) :- negative(X),!,fail.
```

```
non_neg(X) :- tidy(X,Y),eval(Y,0),!.
non_neg(_).

non_zero(X) :- tidy(X,New),eval(New=0),!,fail.
non_zero(_) :- !.

 % Hack for tidying factors
check_tidy(A*B=0,Y) :- !,tidy_expr(A*B=0,Y).
check_tidy(A*B,Y) :- tidy_expr(A*B=0,Y).


 % Two poly normal forms are the same.
check_same_poly(_,[],[]).
check_same_poly(Fac,[H|T],[H1|T1]) :-
        same_poly_fac(Fac,H,H1),
        !,
        check_same_poly(Fac,T,T1).

same_poly_fac(1,polyand(N,M),polyand(N,M)) :- !.
same_poly_fac(Fac,polyand(M,N),polyand(M,N1)) :-
        non_zero(N1),
        eval(N/N1,Fac),
        !.



 % Call a list of conditions
check_cond([]).
check_cond([H|T]) :- call(H),check_cond(T).


 % Get a file name
file_name(Type,File) :-
        repeat,
        writef('\nName the file to write %w to?\n',[Type]),
        prompt(_,'Filename:'),
        read(File),
        (valid_filename(File,_); % Check that a file name is not too long
                                 % (Uses Lincoln's Code in File)
        writef('\nInvalid file name, please try again.\n')),
        !.


file_name_check(File,Type,NewFile) :-
        file_exists,
        !,
        process_reply([y,n],f_n1(Reply,File,Type,NewFile),Reply,'Ove write:',
writef('\nFile %w already exists.  Overwrite it? (y/n)\n',[File])),
        !.

file_name_check(File,_,File) :- !.

f_n1(n,_,Type,File) :- !,
        process_reply([y,n],f_n2(Reply,Type,File),Reply,'Choose new  ile:',
writef('\nChoose new file? (y/n)\n')).

f_n1(y,File,_,File) :- !,
        writef('\nOK, overwriting file %w\n',[File]).
```

```
f_n2(y,Type,File) :- !,
        file_name(Type,File).
f_n2(n,_,_) :- !,
        writef('\nOK, Aborting from writeout command\n'),
        !,
        call(abort).

 % Get and check a file name, query if file already exists
file_name1(Type,File) :-
        file_name(Type,File1),
        file_name_check(File1,Type,File),
        !.

type_tidy(normal,Old,New) :- !,tidy(Old,New).
type_tidy(expr,Old,New) :- !,tidy_expr(Old,New).



 % Split a list of multiplicative factors into a list of
 % equations, removing useless ones

remove_safe_divisors(_,[],[]) :- !.

remove_safe_divisors(X,[H|T],T1) :-
        safe_divisor(X,H),
        !,
        remove_safe_divisors(X,T,T1).

remove_safe_divisors(X,[H|T],[H1|T1]) :-
        type_tidy(expr,H=0,H1),
        remove_safe_divisors(X,T,T1).

safe_divisor(X,H) :- freeof(X,H),non_zero(H).



 % match_check(X,Y) sees if match(X,Y) is true and cuts alternatives
match_check(X,Y) :- ground(X),!,match(X,Y).
match_check(X,Y) :- ground(Y),!,match(Y,X).

 % Maplist tidy
maptidy_example(X,Example,New,Flag) :-
        maptidy_example1(X,Example,Mid),
        !,
        merge_steps(Example,Mid,New,Flag).

maptidy_example1(_,[],[]) :- !.
maptidy_example1(X,[H|T],[H1|T1]) :-
        mod_weak_normal_form2(H,X,H1),
        !,
        maptidy_example1(X,T,T1).

maptidy_example1(X,[H|_],_) :-
        writef('\n
[**Unable to weak_normal_form\n%t\nfor %t.\nFailing**]\n',[H,X]),
        !,
        fail.

 % If two steps are the same after tidying merge them and warn user
```

```
merge_steps([],[],[],_) :- !.
merge_steps([_],[B],[B],_) :- !.
merge_steps([H,H|T],[A,A|C],D,Flag) :- !,
(Flag = sol; writef('\n[**Consecutive step %t, steps merged**]\n',[H ])),
        !,
        merge_steps([H|T],[A|C],D,Flag).

merge_steps([H,H1|T],[A,A1|C],[A|D],Flag) :-
        match(A,A1),
        !,
(Flag = sol;    writef('\n[**Steps\n\t%t\n\nand\n\n\t%t\n
both tidy to \n\n\t%t\n\nsteps merged**]\n',[H,H1,A])),
        !,
        merge_steps(T,C,D,Flag).


merge_steps([_|T],[A|C],[A|D],Flag) :- !,
        merge_steps(T,C,D,Flag).



 % Tidy Conjectures
 .dy_up(A=B,C,D) :-
        functor(A,F,_),
        (F= + ; F = *),
        decomp(A,[F|List]),
        shift_numbers(List,[],Rest,[H|Numbers],F), % [H|Numbers] mus  be
        !,                                         % non_empty
        recomp(Num,[F,B,H|Numbers]),
        recomp(C1,[F|Rest]),
        tidy(C1=Num,New),
        !,
        tidy_up(New,C,D).

tidy_up(A=B,A,B) :- !.


 % Add new auto method
 % General case
add_new_method(Method,Type) :-
        (Type = upc(Post) -> add_to_table(method(Method),Post);true)
        cond_assertz(method(method(Method))),
        cond_assertz(just_created(Method)),
        flag(method(method(Method)),_,on).


cond_assertz(A) :- call(A),!.
cond_assertz(A) :- assert(asserted(A)),assertz(A).



 % Abolish the units rather than every term
mod_abolish(Functor,Arity) :-
        functor(Term,Functor,Arity),
        mod_ab(Term).

mod_ab(Term) :- retract(Term),fail.
mod_ab(_).

mod_asserta(X) :- assert(asserted(X)),asserta(X),!.
```

```
mod_assert(X) :- assert(asserted(X)),assert(X),!.
mod_assertz(X) :- assert(asserted(X)),assertz(X),!.


subs1(Exp,[],Exp) :- !.
subs1(Exp,[H|T],E1) :- subst(H,Exp,E2),!,subs1(E2,T,E1).


make_subl([],[],[]) :- !.
make_subl([X|R],[X|R1],R2) :- !,make_subl(R,R1,R2).
make_subl([Hd|R],[H1|R1],[Hd=H1|R2]) :- !,make_subl(R,R1,R2).


great_el([Hd],Hd) :- !.
great_el([Hd|Tl],Ans) :- great_el(Tl,Hgr),(eval(Hd>Hgr) -> Hd=Ans;Hg =Ans),!.

mod_gensym(Root,Term) :-
        repeat,
        gensym(Root,Term),
        really_new(Root,Term),
        !.

 ally_new(auto,Term) :- !,
        not (method(method(Term))).
really_new(n,Term) :- !,
        not integral(Term).
really_new(x,Term) :- !,
        not unknown(Term).
```

```
/* COMP : Misc code for compilation

                                                    Bernard Silver
                                                    Updated: 27 February 1984

Includes code by Alan, Lawrence, Leon and Richard
*/


:- public
                        arbint/1,
                        associative/1,
                        closeness/3,
                        commutative/1,
                        correspond/4,
                        delete/3,
                        delete_one/3,
                        dottoand/2,
                        even/1,
                        extreme_term/3,
                        gcd/3,
                        ground/1,
                        identical_subterms/3,
                        identifier/1,
                        least_dom/2,
                        match_member/3,
                        odd/1,
                        old_attract/3,
                        position/3,
                        subterms/3,
                        subterms2/3,
                        subst_mesg/3,
                        tidy_up_disjunction/2,
                        wordsin/2.

:- mode
                        arbint(-),
                        associative(+),
                        at_least_occ(+,+,+),
                        binary_to_list(+,+,+,?,?),
                        closeness(+,+,?),
                        com_ass_idn(+,-),
                        commutative(+),
                        correspond(?,+,+,?),
                        delete(+,+,-),
                        delete_one(+,+,?),
                        dottoand(+, -),
                        even(+),
                        extreme_term(+, +, -),
                        extreme_term(+, +, +, +, -),
                        gcd(+, +, -),
                        ground(?),
                        ground(?,+),
                        identifier(-),
                        insert_word(?, +, -),
                        least_dom(+, +),
                        least_dom(+, +, +, +),
                        match_member(+,+,?),
                        odd(+),
```

```
                        old_attract(+,+,-),
                        position(?, +, ?),
                        position(+, ?, +, ?),
                        scan_list(+, ?, -),
                        scan_term(+, ?, -),
                        strip_num(+, -),
                        subterms(+,+,?),
                        subterms2(+,+,?),
                        subst_mesg(+, +, -),
                        term_size(+, -),
                        term_size(+, +, +, -),
                        tidy_up_disjunction(+,-),
                        tree_list(?, +, +, -),
                        tree_size(+,+,-),
                        tree_size(+,+,+,+,-),
                        wordsin(+, -).


 % Delete first occ of H from second arg to get third.
delete_one(_,[],[]).
delete_one(H,[H|T],T) :- !.
delete_one(H,[X|T],[X|Ans]) :- delete_one(H,T,Ans).

   terms(Term,X,Sub) :-
        Term=..[_|Args],
        member(Sub1,Args),
        \+ atomic(Sub1),
        contains(X,Sub1),
        subterms(Sub1,X,Sub).

subterms(Term,X,Term1) :- Term \= X,contains(X,Term),Term1=Term.


subterms2(X,X,X) :- !.
subterms2(Term,X,Sub) :- subterms(Term,X,Sub).

match_member(X,[Y|_],Y) :- functor(X,F,N),functor(Y,F,N),match_check(Y,X).
match_member(X,[_|T],Z) :- match_member(X,T,Z).

closeness(X,Exp,Arcs) :-
        tree_size(X,Exp,Nodes),
        Arcs is Nodes - 1.


tree_size(X, X, 1) :- !.
tree_size(X, Exp, 0) :-
        atomic(Exp), !.
tree_size(X, Exp, Size) :-
        functor(Exp, _, N),                     %  cut not needed after all
        tree_size(N, Exp, X, 0, Size).

tree_size(0, Exp, X, 0, 0) :- !.                %  X doesn't occur in Exp
tree_size(0, Exp, X, M, N) :- !,                %  X does occur in Exp,
        N is M+1.                               %  so count Exp node too.
tree_size(N, Exp, X, Acc, Size) :-
        arg(N, Exp, Arg),
        tree_size(X, Arg, ArgSize),
        NewAcc is Acc+ArgSize,
        M is N-1, !,
```

```
        tree_size(M, Exp, X, NewAcc, Size).

old_attract(X,Old,New1) :-
        closeness(X,Old,Closeness),
        mult_occ(X,Old),
        least_dom(X,Old),
        attrax(U & V,Template,Rewrite,Cond), % Assumes attraction between 2
        applicable(Template,Old,Rest),  % subterms only
        contains(X,U),
        contains(X,V),
        !,
        newform(Old,Rewrite,Rest,New),
        check_cond(Cond),
        tidy(New,New1),
        closeness(X,New1,NewC),
        NewC < Closeness,
        !.

old_attract(X,Old,New) :-
        mult_occ(X,Old),
        decomp(Old,[Fun|Args]),
        corresponding_arguments(Args,Arg,NewArgs,NewArg),
        old_attract(X,Arg,NewArg),
        recomp(New,[Fun|NewArgs]).

dottoand([],true) :-!.
dottoand([Head|Tail], Head & Rest) :-
        dottoand(Tail, Rest).



        binary_to_list(Nil, _, Nil, List, List) :- !.
        binary_to_list(Term, Op, Nil, Head, Tail) :-
                Term =.. [Op, Arg1, Arg2],
                binary_to_list(Arg1, Op, Nil, Head, Middle), !,
                binary_to_list(Arg2, Op, Nil, Middle, Tail).
        binary_to_list(Term, _, _, [Term|Tail], Tail).


%   at_least_occ(List, Term, Limit) is true when List contains at least
%   Limit (>= 0) elements which contain Term.  This is NOT the same as
%   occ(List,Term,N) & N >= Limit, as several instances can be in 1 element.



at_least_occ(_, _, 0) :- !.
at_least_occ([Head|Tail], Term, Limit) :-
        contains(Term, Head),
        Mimit is Limit-1, !,
        at_least_occ(Tail, Term, Mimit).
at_least_occ([_|Tail], Term, Limit) :-
        at_least_occ(Tail, Term, Limit).


%   com_ass_idn(Op,Id) -> Op is a commutative associative operator
%   with identity element Id.  This is a makeshift for keeping the
%   arguments of such operators as bags.

        com_ass_idn(+, 0).              com_ass_idn(*, 1).
```

```
least_dom(Term, Exp) :-
        functor(Exp, Op, 2),
        com_ass_idn(Op, Unit),
        binary_to_list(Exp, Op, Unit, List,[]), !,
        at_least_occ(List, Term, 2).
least_dom(Term, Exp) :-
        functor(Exp, _, N),
        least_dom(N, 0, Term, Exp).

        least_dom(N, 2, Term, Exp) :- !.
        least_dom(0, K, Term, Exp) :- !, fail.
        least_dom(N, K, Term, Exp) :-
                arg(N, Exp, Arg),
                contains(Term, Arg),
                M is N-1, L is K+1, !,
                least_dom(M, L, Term, Exp).
        least_dom(N, K, Term, Exp) :-
                M is N-1, !,
                least_dom(M, K, Term, Exp).


%   position(Term, Exp, Path) is true when Term occurs in Exp at the
%   position defined by Path.  It may be at other places too.

   ition(Term, Term, []).
position(Term, Exp, Path) :-
        (   var(Exp) ; atomic(Exp) ; number(Exp)   ), !, fail.
position(Term, Exp, Path) :-
        functor(Exp, _, N),
        position(N, Term, Exp, Path).

        position(0, Term, Exp, Path) :- !, fail.
        position(N, Term, Exp, [N|Path]) :-
                arg(N, Exp, Arg),
                position(Term, Arg, Path).
        position(N, Term, Exp, Path) :-
                M is N-1, !,
                position(M, Term, Exp, Path).


%   generate intermediate variables, or arbitray integer tokens.

  int(Var) :-
        gensym(n, Var),
        assert(integral(Var)).

identifier(Var) :-
        asserta(unknown(Var)),
        gensym(x, Var),!.

%   correspond(X, Xlist, Ylist, Y) is true when the position of X and Xlist
%   and the position of Y in Ylist (which is as long as Xlist) are the same.

correspond(X, [X|_], [Y|_], Y) :- !.
correspond(X, [_|T], [_|U], Y) :-
        correspond(X, T, U, Y).


%   apply a substitution, tidy the result
```

```
subst_mesg(Substitution, Old, New) :-
        subst(Substitution, Old, Mid),
        tidy(Mid, New).


%   Find the smallest (if C = <) or greatest (if C = >) term in a list of
%   terms, where comparison is by the size of a term.

extreme_term([Head|Tail], C, Term) :-
        term_size(Head, Size),
        extreme_term(Tail, Head, Size, C, Term).

        extreme_term([Head|Tail], Hold, Sold, C, Term) :-
                term_size(Head, Size),
                compare(C, Size, Sold), !,
                extreme_term(Tail, Head, Size, C, Term).
        extreme_term([Head|Tail], Hold, Sold, C, Term) :-
                extreme_term(Tail, Hold, Sold, C, Term).
        extreme_term([],          Term, _,    _, Term).

        term_size(Term, 1) :-
                (   var(Term) ; atomic(Term) ; number(Term)   ), !.
        term_size(Term, Size) :-
                functor(Term, _, N),
                term_size(N, Term, 1, Size).

                term_size(0, Exp, Ans, Ans) :- !.
                term_size(N, Exp, Acc, Ans) :-
                        arg(N, Exp, Arg),
                        term_size(Arg, Size),
                        Nxt is Acc+Size+1, M is N-1, !,
                        term_size(M, Exp, Nxt, Ans).

 % Delete all occurrences of X from list Y to get list Z

delete([], _, []) :- !.
delete([Kill|Tail], Kill, Rest) :- !,
        delete(Tail, Kill, Rest).
delete([Head|Tail], Kill, [Head|Rest]) :- !,
        delete(Tail, Kill, Rest).


   Remove 'false' and duplications in a disjunction

tidy_up_disjunction(Term,Ans) :-
        decomp(Term,[#|List]),
        !,
        listtoset(List,List1),
        delete(List,false,New),
        recomp(Ans,[#|New]).

tidy_up_disjunction(X,Y) :- tidy(X,Y).  % For cases that fall through


wordsin(Term, List) :-
        scan_term(Term, Some, Tree),
        tree_list(Tree, 1, [], Pairs),
        keysort(Pairs, Inorder),
        strip_num(Inorder, List).
```

```
        scan_term(Simp, Old_Tree, Old_Tree) :-
                var(Simp), !.
        scan_term(Simp, Old_Tree, Old_Tree) :-
                number(Simp), !.        %  was integer(Simp)
        scan_term(Atom, Old_Tree, New_Tree) :-
                atom(Atom), !,
                insert_word(Old_Tree, Atom, New_Tree).
        scan_term(List, Old_Tree, New_Tree) :-
                List = [_|_], !,
                scan_list(List, Old_Tree, New_Tree).
        scan_term(Term, Old_Tree, New_Tree) :-
                Term =.. [Functor|Args], !,
                scan_list(Args, Old_Tree, New_Tree).

                insert_word(t(C, W, L, R), W, t(D, W, L, R)) :- !,
                        (   var(C), D = 1
                        ;   integer(C), D is C+1
                        ), !.
                insert_word(t(C, X, L, R), W, t(C, X, M, R)) :-
                        W @< X, !,
                        insert_word(L, W, M).
                insert_word(t(C, X, L, R), W, t(C, X, L, S)) :-
                        W @> X, !,
                        insert_word(R, W, S).

                scan_list([Head|Tail], Old_Tree, New_Tree) :-
                        scan_term(Head, Old_Tree, Mid_Tree), !,
                        scan_list(Tail, Mid_Tree, New_Tree).
                scan_list([],           Old_Tree, Old_Tree).

        tree_list(Tree,           Thresh, Accum, Accum) :-
                var(Tree), !.
        tree_list(t(N, X, L, R), Thresh, Accum, Answer) :-
                N < Thresh,
                tree_list(L, Thresh, Accum, Sofar), !,
                tree_list(R, Thresh, Sofar, Answer).
        tree_list(t(C, W, L, R), Thresh, Accum, Answer) :-
                tree_list(L, Thresh, Accum, Sofar),
                Key is -C, !,
                tree_list(R, Thresh, [Key-W|Sofar], Answer).

        strip_num([Key-Word|Rest], [Word|More]) :- !,
                strip_num(Rest, More).
        strip_num([],              []).

odd(X) :-
        eval(odd(X)).

even(X) :-
        eval(even(X)).

gcd(X, Y, Z) :-
        eval(gcd(X,Y), Z).

commutative(+) :- !.
commutative(*) :- !.

associative(+) :- !.
associative(*) :- !.
```

```
 % Is term ground

ground(Term) :- var(Term),!,fail.
ground(Term) :- simple(Term),!.
ground(Term) :-
        functor(Term,F,N),
        !,
        ground(Term,N).

ground(_,0) :- !.
ground(Term,N) :-
        arg(N,Term,Arg),
        ground(Arg),
        M is N - 1,
        !,
        ground(Term,M).
```

```
  File   : Method
  Author : Bernard Silver
  Updated: 23 February 1984
  Purpose: LP methods


% known_method is the top level call that checks whether the method is
% currently enabled by the user.


 Schema methods

own_method(X,Old,New,Name,schema(S,Type),Call,Pre,Post) :-
      var(New),        % Solve only
      known_method_schema(X,Old,New,Name,S,Type,Call,Pre,Post),
      flag(method(Name),on,on).

 Top level

own_method(X,Old,New,Name,Type,Call,Pre,Post) :-
      known_method1(X,Old,New,Name,Type,Call,Pre,Post),
      flag(method(Name),on,on).


  Top level' methods that should be tried first.  These are
 Factorize, Isolatation and Disjunction, Change of Unknown



own_method1(X,Old,New,Name,Type,Call,Pre,Post) :-
      known_method_tl(X,Old,New,Name,Type,Call,Pre,Post).


 New methods
own_method1(X,Old,New,Name,all,Call,Pre,Post) :-
      known_method_auto(X,Old,New,Name,Call,Pre,Post).

  All other methods
own_method1(X,Old,New,Name,Type,Call,Pre,Post) :-
      known_method_interp(X,Old,New,Name,Type,Call,Pre,Post).

            /*  THE METHODS  */

   method_tl(X,Old,Ans,'Split Disjunctions',all,
y_disjunct(X,Old,Ans),[is_disjunct(X,Old)],[])
            :-
      not ground(Ans).

own_method_tl(X,Old,New,'Isolation',all,
      try_to_isolate(X,Old,New),[single_occ(X,Old)],[]).

% Only to be used for solving, hence not ground(Ans)
own_method_tl(X,Old,Ans,'Factorization',all,
y_factorize(X,Old,Ans),[rhs_zero(Old),mult_occ(X,Old),is_product(X,Old)],[])
            :-
      not ground(Ans).


own_method_tl(X,Eqn,New,'Change of Unknown',all,
y_chunk(X,Eqn,New,Term),[mult_occ(X,Eqn),identical_subterms(Eqn,X,Term)],[])
```

```
            :-
      not ground(New).

 known_method_interp tries the lowest priority methods in the order
 they occur in method_list(MethodList).  This list can be changed by the
 user.  Different for tl and solve.

›wn_method_interp(X,Eqn,New,Name,Type,Call,PreCond,Effect) :-
      (ground(New)->Task=tl;Task=sol),
      get_method_list(Task,MethodList),
      member(Name,MethodList),
      known_method2(X,Eqn,New,Name,Type,Call,PreCond,Effect).


›wn_method2(X,Old,New,'Prepare for Factorization',all,
      try_prep_fact(X,Old,New),
      [is_sum(X,Old),rhs_zero(Old),mult_occ(X,Old),common_subterms(X,Old,_)],
      [is_product(X,New),rhs_zero(New),less_occ(X,Old,New)]).


›wn_method2(X,Old,New,'Function Stripping',all,
/_function_stripping(X,Old,Posn,New),[dominated(X,Old,Posn)],
      [same_occ(X,Old,New),not dominated(X,New,_)]).

›wn_method2(X,Old,New,'Polynomial Methods',all,try_poly(X,WNF,New),
      [is_mod_poly(X,Old,WNF)],[]).

›wn_method2(X,Old,New,'Collection',part,try_collect(X,Old,New),
      [mult_occ(X,Old)],[less_occ(X,Old,New)]).

% Attract
›wn_method2(X,Old,New,'Attraction',part,try_attract(X,Old,New),
      [mult_occ(X,Old)],[same_occ(X,Old,New),closer(X,Old,New)]).


 Log Method
›wn_method2(X,Eqn,New,'Logarithmic Method',all,remove_logs(X,New,Mid,Base),
      [mult_occ(X,Eqn),prod_exp_terms_eqn(Eqn,X,Mid)],[]).

 Nasty
›wn_method2(X,Eqn,New,'Nasty Function Method',all,
    _nasty(X,Eqn,New),[mult_occ(X,Eqn),contains_nasties(X,Eqn)],[]).

 user Rule  (Work only at present)
›wn_method2(X,Eqn,New,user_rule(Name,_,_),_,try_user_rule(X,Eqn,New,Name),
      [rules_stored],[]).


thod('Isolation').
thod('Factorization').
thod('Prepare for Factorization').
thod('Split Disjunctions').
thod('Polynomial Methods').
thod('Change of Unknown').
thod('Collection').
thod('Function Stripping').
thod('Attraction').
thod('Logarithmic Method').
thod('Nasty Function Method').
thod(user_rule(_,_,_)).
```

```
method('homogenization').
method('user_rule').

sable_new_methods :- method(method(X)),flag(method(method(X)),_,off),fail.
sable_new_methods.

able_new_methods :- method(method(X)),flag(method(method(X)),_,on),fail.
able_new_methods.


nove_new_method_markers :-
      retract(method(method(X))),
      flag(method(method(X)),_,off),
      fail.
nove_new_method_markers.

t_method_list(Type,List) :- method_list(Type,List),!.
t_method_list(Type,_) :-
      writef('\n[**Error.  No method_list for type %t present**.]\n',[Type]),
      !,
      fail.

h~d_list(sol,['Prepare for Factorization','Polynomial Methods',
   .ion Stripping','Collection','Attraction',
 homogenization,
ogarithmic Method','Nasty Function Method']).

thod_list(tl,['Prepare for Factorization','Polynomial Methods',
 homogenization,
unction Stripping','Collection','Attraction','Logarithmic Method',
asty Function Method',user_rule(_,_,_)]).

 Key Methods
y_method('Factorization').
y_method('Change of Unknown').
y_method('Isolation').
y_method('Polynomial Methods').
y_method('Split Disjunctions').

 Change method order in method list

    _method_order :-
      c_m_o(solve,sol),
      c_m_o('worked example',tl).

n_o(Word,Type) :-
      writef('\nChange list for %w.\n',[Word]),
      retract(method_list(Type,List)),
      writef('\nCurrent method order is:\n%l\nEnter new order.\n\n',[List]),
      prompt(_,'New List:'),
      repeat,
      read(New),
      check_order(List,New),
      !,
      asserta((method_list(Type,New))).

eck_order(Old,New) :-
      perm(Old,New),
      !,
      writef('\nOK, new order is being stored.\n').
```

```
eck_order(_,_) :-
        writef('\n
e new list is not a permutation of the old one, please try again.\n'),
        fail.

ort_name(chunk,'Change of Unknown') :- !.
ort_name(user_rule,user_rule(_,_,_)) :- !.
ort_name(Short,Method) :-
        atom(Short),
        name(Short,[F1,F2,F3,F4|ShortName]), % Must be at least 4 letters long
        convert_case([F1,F2,F3,F4|ShortName],[L1,L2,L3,L4|_]),
        !,
        method1(Method),
        atom(Method),
        name(Method,MethodName),
        convert_case(MethodName,[L1,L2,L3,L4|_]),
        !.

nvert_case([],[]) :- !.
nvert_case([H|T],[H1|T1]) :-
        H>="A",
        H=<"Z",
        !,
        H1 is H +32,
        !,
        convert_case(T,T1).
nvert_case([H|T],[H|T1]) :-
        convert_case(T,T1).
```

```
%   File   : IMETH
%   Author : Bernard Silver
%   Updated: 23 February 1984
%   Purpose: Some Interpreted Method Code for LP

 % The isolate code

isolate([N|Posn],Exp,Ans) :-
        manoeuvre_sides(N,Exp,NewExp),
        isolate1(Posn,NewExp,Inter),
        tidy(Inter,Ans).


 %  get term to be isolated on Rhs

manoeuvre_sides(1,Exp,Exp) :- !.

manoeuvre_sides(2,A=B,B=A).

 % Perform the Isolation

 % trivial boolean cases

isolate1(_,false,false).
isolate1(_,true,true).

 % deal with each disjunct

isolate1(Posn,Eqn1#Eqn2,Ans1#Ans2) :-
        !,
        isolate1(Posn,Eqn1,Ans1),
        isolate1(Posn,Eqn2,Ans2).


 % expression is already isolated

isolate1([],Ans,Ans) :- !.

 % expression can have isolax rule applied

isolate1([N|Posn],Old,Ans) :- !,
        isolax(N,Old,New),
        tidy_rhs(New,New1),
        isolate1(Posn,New1,Ans).

tidy_rhs(A#B,C#D) :- !,tidy_rhs(A,C),tidy_rhs(B,D).
tidy_rhs(A=B,A=C) :- tidy(B,C).
tidy_rhs(false,false) :- !.
tidy_rhs(true,true) :- !.

 % Attract and Collect applied as much as possible

recurse_collect(X,Eqn,New) :-
        collect(X,Eqn,Mid),
        tidy(Mid,Mid1),
        recurse_collect1(X,Mid1,New).

recurse_collect1(X,Old,New) :-
        collect(X,Old,Mid),
        tidy(Mid,Mid1),
```

```
        recurse_collect1(X,Mid1,New).

recurse_collect1(_,Old,Old).

recurse_attract(X,Old,New) :-
        attract(X,Old,Mid),
        tidy(Mid,Mid1),
        recurse_attract1(X,Mid1,New).

recurse_attract1(X,Old,New) :-
        attract(X,Old,Mid),
        tidy(Mid,Mid1),
        recurse_attract1(X,Mid1,New).

recurse_attract1(_,Old,Old).
 % Modified Collection, call Collection
mod_collect(X,Exp,New) :- collect(X,Exp,New).

mod_collect(X,A+B,New) :- hard_combine(contains(X,T),A+B,T,New),!.

attract(X,Exp,New) :- old_attract(X,Exp,New),closer(X,Exp,New).
attract(X,A+B,New) :-
        hard_combine(true,A+B,_,New),
        !,
        closer(X,A+B,New).



 % In the expression A + B, A and B are products which have a common
 % subterm Y, (which matches Y1) X is the unknown. Term1 And Term2
 % are the rest of A and B.  Keep on applying to remove other
 % stuff as well.

prepfac_terms(Cond,Exp,Y1,NewTerm,Term1,Term2) :-
        decomp(Exp,[+|PlusBag]),
        map_mult_decomp(PlusBag,[List1|ListList]),
        member(Y1,List1),
        call(Cond),
        map_match_member(Y1,ListList,ListY2),
        !,
        delete_one(Y1,List1,New1),
        map_delete_one(ListY2,ListList,New2),
        prepfac_terms1(New1,New2,NewTerm,1,New3,New4),
        recomp(Term1,[*|New3]),
        map_recomp(ListTerm2,New4,*),
        recomp(Term2,[+|ListTerm2]),
        !.

prepfac_terms1([H|List],List2,NewTerm,Acc,Rest1,Rest2) :-
        map_match_member(H,List2,Mult2),
        !,
        map_delete_one(Mult2,List2,Temp2),
        prepfac_terms1(List,Temp2,NewTerm,Acc*H,Rest1,Rest2).

prepfac_terms1([H|List],List2,NewTerm,Acc,[H|New1],New2) :- !,
        prepfac_terms1(List,List2,NewTerm,Acc,New1,New2).

prepfac_terms1([],List,Acc,Acc,[],List) :- !.

map_mult_decomp([],[]) :- !.
```

```
map_mult_decomp([H|T],[H1|T1]) :-
        mult_decomp(H,H1),
        !,
        map_mult_decomp(T,T1).

mult_decomp(A,List) :- decomp(A,[*|List]),!.
mult_decomp(A,[A]).

map_delete_one([],[],[]) :- !.
map_delete_one([Term|Tail],[H|T],[H1|T1]) :-
        delete_one(Term,H,H1),
        !,
        map_delete_one(Tail,T,T1).

map_match_member(_,[],[]) :- !.
map_match_member(Term,[H|T],[H1|T1]) :-
        match_member(Term,H,H1),
        !,
        map_match_member(Term,T,T1).

map_recomp([],[],_).
map_recomp([H|T],[H1|T1],Op) :-
        recomp(H,[Op|H1]),
        map_recomp(T,T1,Op).

 % Use to rewrite
hard_combine(Cond,Exp+Exp1,Mult,New) :-
        prepfac_terms(Cond,Exp+Exp1,Mult,Y,Term1,Term2),
        !,
        tidy(Mult*Y*(Term1+Term2),New).
```

```
POLPAK :    Leon's Code

                                                    Bernard Silver
                                                    Updated: 30 October 82

public

            gcd_powers/2,
            is_poly/2,
            make_poly/3,
            map_add_power/3,
            map_div_power/3,
            map_reify/3,
            poly/4,
            poly_norm/3,
            reify/3,
            z_norm/2.

mode
      add_poly(+,+,?),
      gcd_powers(+,?),
      map_add_power(+,+,?),
      map_reify(+,+,?),
      poly_norm(+,+,-),
      poly(+,+,?,?),
      reify(+,+,-),
      times(+,+,?),
      z_norm(+,?).


Check if Expression is a polynomial */

_poly(X,X) :- !.

_poly(X,X^N) :- integer(N), !.

_poly(X,(X^N)^(-1)) :- integer(N), !.

_poly(X,E) :- freeof(X,E), !.

_poly(X,S+T) :- !, is_poly(X,S), is_poly(X,T).

_poly(X,S*T) :- !, is_poly(X,S), is_poly(X,T).

_poly(X,S^N) :- !, integer(N), N >= 0, is_poly(X,S).


Put polynomials in normal form (succeeds only for polynomials) */

oly_norm(Poly,X,Plist) :-
      poly(X,Poly,Pbag),
      z_norm(Pbag,Plist).

Forms bag of coefficients */

oly(X,X,[polyand(1,1)]) :- !.

oly(X,X^N,[polyand(N,1)]) :- integer(N), !.
```

```
ly(X,(X^N)^(-1),[polyand(N1,1)]) :- integer(N), !, eval(-N,N1).

ly(X,E,[polyand(0,E)]) :- freeof(X,E), !.

ly(X,S+T,Ebag) :-
        !,
        poly(X,S,Sbag),
        poly(X,T,Tbag),
        add_poly(Sbag,Tbag,Ebag).

ly(X,S*T,Ebag) :-
        !,
        poly(X,S,Sbag),
        poly(X,T,Tbag),
        times_poly(Sbag,Tbag,Ebag).

ly(X,S^N,Ebag) :-
        integer(N),
        eval(N > 0),
        !,
        poly(X,S,Sbag),
        binomial(Sbag,N,Ebag).

ly(X,X^K,[polyand(N,1)]) :- !, eval(K,N), number(N).

    X,S^K,Bag) :-
        exp_distrib(S^K,Exp),
        poly(X,Exp,Bag).

* Add two coefficients bags  */

id_poly([],T,T) :- !.

id_poly(S,[],S) :- !.

id_poly([polyand(N,E)|P],[polyand(M,F)|Q],[polyand(N,E)|Y]) :-
                eval(N > M),
                add_poly(P,[polyand(M,F)|Q],Y),
                !.

id_poly([polyand(N,E)|P],[polyand(M,F)|Q],[polyand(N,Y)|Z]) :-
                eval(N = M),
                add_poly(P,Q,Z),
                tidy(E+F,Y),
                !.

id_poly([polyand(N,E)|P],[polyand(M,F)|Q],[polyand(M,F)|Y]) :-   %N < M
                add_poly(Q,[polyand(N,E)|P],Y), !.


* Multiply two coefficient bags - Distributivity of multiplication over
                                        addition assumed        */

imes_poly([],Bag,[]) :- !.

imes_poly(Bag,[],[]) :- !.

imes_poly([polyand(N,E)],S,X) :- timesingl(S,N,E,X), !.

imes_poly([polyand(N,E)|R],S,Z) :-
```

```
        timesingl(S,N,E,X),
        times_poly(R,S,Y),
        add_poly(X,Y,Z),
        !.

imesingl([],N,E,[]) :- !.

imesingl([polyand(M,F)|R],N,E,[polyand(X,Y)|Z]) :-
                eval(M+N,X),
                tidy(F*E,Y),
                timesingl(R,N,E,Z).

 Binomial expansion of coefficient bag */

inomial(Bag, 0, [polyand(0,1)]) :- !.

inomial(Bag, 1, Bag) :- !.

inomial(Sbag, N, Ebag) :-
                !,
                eval(N-1,N1),
                binomial(Sbag,N1,Ebag1),
                times_poly(Sbag,Ebag1,Ebag).

 ove any  terms with zero coefficient

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

_norm([polyand(N,0)|R],Pnorm) :- z_norm(R,Pnorm), !.

_norm([polyand(N,A)|R],[polyand(N,A)|Pnorm]) :- z_norm(R,Pnorm).


 Converted maplists etc

ap_add_power(_,[],[]) :- !.
ap_add_power(N,[Pterm|P],[Qterm|Q]) :-
        add_power(N,Pterm,Qterm),
        map_add_power(N,P,Q).

dd_power(N,polyand(M,Coeff),polyand(MN,Coeff)) :-
        MN is M+N.

ap_div_power(_,[],[]) :- !.
ap_div_power(N,[Pterm|P],[Qterm|Q]) :-
        div_power(N,Pterm,Qterm),
        map_div_power(N,P,Q).

iv_power(Num,polyand(Power,Coeff),polyand(Newpow,Coeff)) :-
        Newpow is Power/Num.

cd_powers([polyand(N,_)|Poly],Gcd) :-
        gcd_poly(Poly,N,Gcd).

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

cd_poly([],Gcd,Gcd) :- !.

cd_poly([polyand(N,_)|Poly],Sofar,Gcd) :-
        gcd(N,Sofar,New),
```

```
        gcd_poly(Poly,New,Gcd).


p_distrib((A*B)^K,Expr) :-
        decomp(A*B,[*|List]),
        exp_distrib_list(K,List,Bag),
        recomp(Expr,[*|Bag]).

p_distrib_list(_,[],[]) :- !.

p_distrib_list(K,[S|Rest],[S^K|NewRest]) :-
        exp_distrib_list(K,Rest,NewRest).


 Reconstitute bag of coefficients into polynomial */

ke_poly(X,Bag1,Poly) :- !,
                map_reify(X,Bag1,Bag2),
                recomp(Poly,[+|Bag2]).

 reify coefficient and power into product */

ify(X,polyand(0,E),E) :- !.

 (X,polyand(1,E),Exp) :- !, tidy(E*X,Exp).

i y(X,polyand(N,E),Exp) :- !, tidy(E*X^N,Exp).


p_reify(_,[],[]) :- !.
p_reify(X,[H|T],[H1|T1]) :- reify(X,H,H1),map_reify(X,T,T1),!.
```

```
  COLLEC      A more efficient version        Leon
                                Updated: 15 February 83
*****************************************/
 COLLECTION ROUTINES*/
*****************************************/
eclarations%
  public                collect/3,
                        applicable/3,
                        newform/4.

 mode                   collect(+,+,-),
                        applicable(+,+,?),
                        newform(+,+,+,?),
                        template_match(+,+,?).


llect(X,Exp,New1) :-
      mult_occ(X,Exp),
      least_dom(X,Exp),                % Expression is in weak normal form
      collax(U,Template,Rewrite),
      applicable(Template,Exp,Rest),
      contains(X,U),
      !,
      newform(Exp,Rewrite,Rest,New),
      tidy(New,New1).

 TRY TO COLLECT WITHIN A SUBTERM*/

llect(X,Old,New) :-
      mult_occ(X,Old),
      decomp(Old,[Fun|Args]),
      corresponding_arguments(Args,Arg,NewArgs,NewArg),
      collect(X,Arg,NewArg),
      recomp(New,[Fun|NewArgs]),
      !.

Does a rewrite rule match an expression?
A more efficient version than relying on the built in commutativity
and associativity of the matcher

plicable(Template,Exp,Rest) :-
      ident_operators(Template,Exp),          % quick test
      template_match(Template,Exp,Rest).

  ident_operators(A,B) :- A=..[Op|_], B=..[Op|_].

mplate_match(Template,Exp,Rest) :-
      Template=..[Op,C,D],
      ac_op(Op,_,_,_,_),
      !,
      decomp(Exp,[Op|Args]),
      select(A,Args,Rem),
      perm2(C,D,Pat1,Pat2),
      exp_match(A,Pat1,Pat2,Rem,Rest),
      !.

mplate_match(Template,Exp,[]) :-
      match(Exp,Template).

p_match(A,C,D,Rem,Rest) :-
```

```
      match(A,C),              % stop match backtracking (the key idea)
      !,
      exp_match1(A,C,D,Rem,Rest).

p_match1(A,C,D,Rem,Rest) :-
      ops_to_find(D,Ops),
      tidy_ops(Ops,Term),
      absent(Term,Rem),
      !,
      fail.

p_match1(A,C,D,Rem,Rest) :-
      match(A,C),
      select(B,Rem,Rest),
      match(B,D),
      !.

s_to_find(Pat,Pat) :- atomic(Pat), !.
s_to_find(Pat,var) :- var(Pat), !.
s_to_find(Pat,Term) :-
      Pat =.. [Op|Args],
      ops_list(Args,NewArgs),
      Term =.. [Op|NewArgs].

s_list([],[]) :- !.
s_list([H|T],[NewH|NewT]) :-
      ops_to_find(H,NewH),
      ops_list(T,NewT).

sent(_,[]) :- !.
sent(Ops,[H|Rest]) :-
      compatible(Term,H),
      !,
      fail.

sent(Term,[_|Rest]) :- absent(Term,Rest).

mpatible(var,H) :- !.
mpatible(Term,H) :-
      Term=..[Op|Args],
      H=..[Op|Terms],
      list_compatible(Args,Terms).

st_compatible([],[]) :- !.
st_compatible([H|T],Terms) :-
      select(A,Terms,Rest),
      compatible(H,A),
      list_compatible(T,Rest),
      !.

form(_,Rewrite,[],Rewrite) :- !.

form(Exp,Rewrite,Rest,New) :-
      Exp=..[Op|_],
      recomp(Term,[Op|Rest]),
      New=..[Op,Rewrite,Term].

dy_ops(var*Term,New) :- !, tidy_ops(Term,New).
dy_ops(Term*var,New) :- !, tidy_ops(Term,New).
dy_ops(var+Term,New) :- !, tidy_ops(Term,New).
```

```
dy_ops(Term+var,New) :- !, tidy_ops(Term,New).
dy_ops(Term,Term).
```

```
* POLY : Leons code

                                                    Bernard Silver
                                                    Updated: 2 November 1983
*/
% Poly_solve is only called when it has been determined that the
% equation is a polynomial equation.
%   i.e. a precondition that the method is called is that is_poly is true

poly_solve(Eqn1#Eqn2,X,Soln1#Soln2,Rules-Diff) :-
        poly_solve(Eqn1,X,Soln1,Rules-Inter),
        poly_solve(Eqn2,X,Soln2,Inter-Diff).

poly_solve(Lhs=Rhs,X,Soln,[Infer,Mult|Rules]-Diff) :-
        poly_norm(Lhs + -1*Rhs,X,Plist),
        poly_tidy(Plist,Qlist),
        remove_neg_powers(X,Qlist,Poly,Mult),          % Remove negative powers
        poly_method(X,Poly,Soln,Rules-Diff).

remove_neg_powers(X,Plist,Qlist,multiply(Mult)) :-
        last(polyand(N,_),Plist),
        N < 0,
        !,
        eval(-N,N1),
        map_add_power(N1,Plist,Qlist).

remove_neg_powers(_,Plist,Plist,nomult).

/*****************************************/
/* ROUTINES FOR POLYNOMIAL EQUATIONS */
/*****************************************/

/* Identities and unsatisfiable equations */

poly_method(_,[],true,[ident|Diff]-Diff) :- !.  % The polynomial has simplified aw

poly_method(X,[Pterm],Ans,[single_term|Diff]-Diff) :-   % Polynomial simplified
        !,                                              % to a single term
        singleton_method(Pterm,X,Ans).

singleton_method(polyand(0,A),_,true) :-
        simplify(A,B),
        B = 0,
        !.

singleton_method(polyand(0,_),_,false) :- !.

singleton_method(polyand(_,_),X,X = 0) :- !.

/* LINEAR EQUATIONS */

poly_method(X,Poly,X=Ans,[linear|Diff]-Diff) :-
        linear(Poly),
        !,
        linear_method(Poly,Ans,_).

linear([polyand(1,_)|_]) :- !.

linear_method([polyand(N,A)|T],Ans,N) :-        % Handles disguised linear also
        find1(T,B),
```

```
        tidy(-B/A,Ans).

find1([polyand(0,B)],B) :- !.
find1([],0) :- !.                % Shouldn't be needed

/* QUADRATIC EQUATIONS*/

poly_method(X,Poly,Soln,[quadratic|Diff]-Diff) :-
        quadratic(Poly),
        !,
        find_coeffs(Poly,A,B,C),
        discriminant(A,B,C,Discr),
        roots(X,A,B,C,Discr,Soln).

quadratic([polyand(2,_)|_]) :- !.

find_coeffs([polyand(2,A)|T],A,B,C) :- find2(T,B,C).

discriminant(A,B,C,Discr) :- tidy(B^2 - 4*A*C,Discr).

roots(X,A,B,_,0,X = Root) :-                    % Only 1 root
        !,
        tidy(-B/(2*A),Root),
        !.

roots(X,A,B,C,Discr,X = Root1 # X = Root2) :-
        warn_if_complex(Discr),
        tidy((-B + Discr^(1/2))/(2*A),Root1),
        tidy((-B - Discr^(1/2))/(2*A),Root2),
        !.

warn_if_complex(Discr)  :-
        eval(Discr < 0),
        writef('\n[Roots are complex.  LP uses only reals so failing.]\n'),
        !,
        fail.

warn_if_complex(_).

find2([polyand(1,B),polyand(0,C)],B,C) :- !.
find2([polyand(1,B)],B,0) :- !.
find2([polyand(0,C)],0,C) :- !.
%       find2([],0,0) :- !.     Shouldn't be needed

/* Polynomial divisible by an integral power of the unknown */

poly_method(X,Plist,X = 0 # Ans,[divide(X^N)|Rules]-Diff) :-
        last(polyand(N,_),Plist),
        N > 0,
        !,
        eval(-N,M),
        map_add_power(M,Plist,Qlist),
        poly_method(X,Qlist,Ans,Rules-Diff).

/* Disguised Linear */

poly_method(X,Poly,Soln,[linear|Rules]-Diff) :-
        disguised_linear(Poly),
        !,
        linear_method(Poly,Ans,N),
```

```
        isolate([1,1],X^N=Ans,Soln,Rules-Diff).

disguised_linear([polyand(_,_),polyand(0,_)]).

/* Disguised polynomial equations  */

poly_method(X,Plist,Ans,Rules-Diff) :-
        poly_hidden(X,Plist,N),          % Disguised polynomial in X^N
        !,
        map_div_power(N,Plist,Qlist),
        poly_method(X^N,Qlist,Inter,Rules-Laws),
        isolate([1,1],Inter,Ans,Laws-Diff). % Maybe needs poly_isolate

poly_hidden(X,Poly,Gcd) :-
        gcd_powers(Poly,Gcd),
        Gcd > 1,
        !.

% isolate hack until code is reformed
isolate(Posn,Eqn,New,[isolate|L]-L) :- isolate(Posn,Eqn,New).
```

```
%   File   : LOG
%   Author : Bernard Silver
%   Updated: 6 March 1984
%   Purpose: Log Code for LP



                        %  Find the appropriate Base and take logs

log_reduce(A=B,X,Base,Eqn) :-
        log_separate(A=B,X,Loglist,Prod),
        find_base(Loglist,Base),
        take_logs_and_recomp(Base,Loglist,Newlhs),
        tidy(Newlhs=log(Base,Prod),Mid),
        match_check(Mid,Eqn).

log_separate(A=B,X,Loglist,Prod) :- log_separate(A=B,X,[],Loglist,1,Prod).

log_separate(A=B,X,Loglist,L,Prod,P) :-
        prod_decomp(B,X,Loglist,Inter,Prod,Intp,rhs),
        prod_decomp(A,X,Inter,L,Intp,P,lhs).

prod_decomp(A*B,X,Log,L,Prod,P,Side) :-
        !,
        prod_decomp(A,X,Log,Intl,Prod,Intp,Side),
        prod_decomp(B,X,Intl,L,Intp,P,Side).

prod_decomp(A,X,L,L,Prod,A*Prod,rhs) :- freeof(X,A).
prod_decomp(A,X,L,L,Prod,Prod/A,lhs) :- freeof(X,A).

prod_decomp(A^B,_,Log,[exp_term(A,B,-1)|Log],P,P,rhs).
prod_decomp(A^B,_,Log,[exp_term(A,B,1)|Log],P,P,lhs).

find_base([exp_term(A,_,_)|Log],Base) :-
        number(A),
        base(A,B),
        find_base(Log,B,Base,BaseList),
        check_power_of([B|BaseList],Base),
        !.

find_base(_,10).          % Logs to base 10 is the default

find_base([exp_term(A,_,_)|Log],B,Base,[Exp|BaseList]) :-
        number(A),
        !,
        base(A,Exp),
        least(Exp,B,NewB),
        find_base(Log,NewB,Base,BaseList).

find_base([],B,B,[]).

base(A,A) :- integer(A), !.
base(A,Denom) :- eval(numer(A)=1),eval(denom(A),Denom).

                        %  Take logs and reconstitute

take_logs_and_recomp(Base,[exp_term(A,B,Sign)|Log],NewLhs) :-
        tlar(Base,Log,Sign*B*log(Base,A),NewLhs).
```

```
.lar(_,[],Lhs,Lhs) :- !.
.lar(Base,[expr_term(A,B,Sign)|Log],Sum,Lhs) :-
        tlar(Base,Log,Sign*B*log(Base,A)+Sum,Lhs).

                        %  Check that the new base is a root of all the
                        %  exponents, otherwise fail and use base 10.
check_power_of([],_).
check_power_of([H|T],Base) :-
        powered(Base,_,H),
        !,
        check_power_of(T,Base).

                        %  Manipulate equation of the form A + B = 0
                        %  to remove negative signs if possible.

form_new_equation(A+B=0,NewEqn) :-
        negative_number_product(A,New),
        !,
        not negative_number_product(B,_),
        tidy(B=New,NewEqn).

form_new_equation(A+B=0,NewEqn) :-
        negative_number_product(B,New),
        !,
        not negative_number_product(A,_),
        tidy(A=New,NewEqn).


negative_number_product(Term,New) :-
        decomp(Term,[*|Args]),
        select(Number,Args,Rest1),
        number(Number),
        eval(Number < 0),
        recomp(Rest,[*|Rest1]),
        tidy(-Number*Rest,New),
        !.

 % powered(A,B,C) if A^B=C,A not equal 1    (From Homos)

powered(1,_,_) :- !,fail.
powered(A,1,A) :- !.
powered(A,N,A^N) :- number(N),!.
powered(A,B,C) :- number(A),number(C),eval(log(A,C),X),!,number(X),B=X.
```

```
/* NASTY :

                                                    Bernard Silver
                                                    Updated: 8 November 82
*/
nasty_method(Eqn,X,Ans) :-
        tidy(Eqn,Eqn1),
        try_nasty_method(Eqn1,X,Ans),
        !.

try_nasty_method(Eqn,X,Neweqn) :-
        parse4(Eqn,X,U,other),
        subnasty(X,U,V),
        find_symbols(Eqn,V,Symbols,Posns),
        nasty_act(Symbols,Posns,Eqn,X,Neweq),
        tidy(Neweq,Neweqn),
        !.

try_nasty_method(Eqn,X,Neweqn) :-
        parse4(Eqn,X,U,neg),
        exp_nasty_list(X,U,V),
        remove_subsumed(V,Termlist),
        multiply_through(Eqn,Termlist,Neweqn,X),
        tidy(Neweqn,New),
        !.

nasty_act(Symbols,[Posn|_],Eqn,X,New) :-
        nice(Symbols),
        append(Posn,[1],Posn1),
        position(Term,Eqn,Posn1),
        try_isolate(Posn1,Eqn,New),
        !.


try_isolate(Posn,Eqn,New) :- isolate(Posn,Eqn,New),!.
try_isolate(_,_,_) :- !,fail.

nasty_act(Symbols,Posns,Eqn,X,New) :-
        find_attract_list(Symbols,N,L,Type),
        nmember(Posn,Posns,N),
        strip(Posn,L,Newp),
        position(Term,Eqn,Newp),
        nas_rule(Term,Nterm,Type),
        subst(Term=Nterm,Eqn,New1),
        tidy(New1,New),
        !.

parse4(A,Unk,Bag,Type) :- dl_parse4(A,Unk,Bag-[],Type).

dl_parse4(A,Unk,L-L,_) :- freeof(Unk,A),!.
dl_parse4(A=B,Unk,L-L1,T) :- !,
        dl_parse4(A,Unk,L-L2,T),
        dl_parse4(B,Unk,L2-L1,T).

dl_parse4(A*B,Unk,L-L1,T) :- !,
        dl_parse4(A,Unk,L-L2,T),
        dl_parse4(B,Unk,L2-L1,T).

dl_parse4(A+B,Unk,L-L1,T) :- !,
        dl_parse4(A,Unk,L-L2,T),
```

```
        dl_parse4(B,Unk,L2-L1,T).

dl_parse4(A^B,Unk,X,other) :- integer(B),B > 0,!,dl_parse4(A,Unk,X,other).
dl_parse4(A,_,[A|L]-L,_) :- !.

nasty(X,Y) :- root_nasty(X,Y),!.
nasty(X,Y) :- exp_nasty(X,Y),!.

root_nasty(X,U^N) :- contains(X,U),number(N),not integer(N),eval(N>0),!.

exp_nasty(X,U^N) :- contains(X,U),number(N),eval(N<0),diff(X,U),!.

exp_nasty_list(_,[],[]) :- !.
exp_nasty_list(X,[H|Rest],[H|RestV]) :-
        exp_nasty(X,H),
        !,
        exp_nasty_list(X,Rest,RestV).

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


-- d_symbols(_,[],[],[]) :- !.
f   _symbols(E,[H|T],[H1|T1],[H2|T2]) :-
        find_symbols1(E,H,H1,H2),
        find_symbols(E,T,T1,T2),
        !.

find_symbols1(Eqn,X,Y,B) :-
        posl(X,Eqn,A,B),
        expon(X,P),
        append(A,[P],Y),
        !.

posl(X,X,[],[]) :- !.
posl(X,E,[Op|L],[N|Pos]) :-
        E=..[Op1,Arg|Args],
        get_ops(Op,Op1,E),
        nmember(T,[Arg|Args],N),
        posl(X,T,L,Pos),
        !.

  _ops(exp(Arg1),_,E) :- E=..[^,_,Arg1|_],!.
et_ops(Op1,Op1,_) :- !.

xpon(U^N,exp(N)) :- number(N),!.

emove_subsumed([],_) :- !, fail.
emove_subsumed(V,Termlist) :-
        listtoset(V,List),              % Cheap test
        rem_sub(List,Termlist,[]).

em_sub([],Termlist,Termlist) :- !.
em_sub([H|Rest],Termlist,Acc) :-
        member_match(H,Acc,NewAcc),
        !,
        rem_sub(Rest,Termlist,NewAcc).
em_sub([H|Rest],Termlist,Acc) :-
        match(H,U^N),
        number(N),
```

```
        rem_sub(Rest,Termlist,[U^N|Acc]).

iember_match(H,[],_) :- !, fail.
iember_match(H,[U^N|Rest],[U^K|Rest]) :-
        match(H,U^M),
        !,
        least(N,M,K).
iember_match(H,[Term|Rest],[Term|NewRest]) :-
        member_match(H,Rest,NewRest).

.east(N,M,N) :- eval(N=<M), !.
.east(N,M,M).

iice([]) :- !.
iice([List|Rest]) :-
        nice_list(List),
        !,
        nice(Rest).

iice_list([]) :- !.
iice_list([Fun|Rest]) :-
        good_fun(Fun),
        !,
        nice_list(Rest).

)o d_fun(+) :- !.
)ood_fun(=) :- !.
)ood_fun(*) :- !.
)ood_fun(exp(N)) :- number(N),not integer(N),eval(numer(N)=1),!.

'ind_attract_list([],_,_,_) :- !,fail.
'ind_attract_list([H|T],1,M,Type) :- attract_list(H,M,Type),!.
'ind_attract_list([_|T],N,M,Type) :-
        find_attract_list(T,N1,M,Type),
        N is N1+1,
        !.

ittract_list([exp(N)|T],K,Type) :-
        integer(N),
        last(exp(M),T),
        get_nasty_type(M,N,Type),
        append(T1,[exp(M)],T),
        checkpt(T1),
        length(T,K),
        !.


ittract_list([_|T],M,Type) :- attract_list(T,M,Type),!.


)et_nasty_type(M,N,root(M)) :- eval(1/N,M),!.
)et_nasty_type(M,N,negroot(M)) :- eval(1/N,-1*M),!.
)et_nasty_type(M,N,neg(M)) :- eval(M<0),!.

)t(*) :- !.
)t(+) :- !.

)as_rule(A^2 ,Exp,root(N)) :- dist(A,A1),tidy(A1,A2),expon_exp(A2^2,N,Exp),!.
)as_rule(A^2,Exp,negroot(N)) :-
        dist(A,A1),
```

```
        tidy(A1,A2),
        expon_inv_exp(A2^2,N,Exp),
        !.

as_rule(A^2,Exp,neg(N)) :- neg_exp(A^2,N,Exp),!.

expon_exp(Old,N,New) :- eval(N=(1/2)),expon_exp1(Old,N,New),!.

expon_inv_exp(Old,N,New) :- eval(N=(-1/2)),expon_inv_exp1(Old,N,New),!.

expon_exp1(A^2,N,C^2 + 2*C*D^N + D) :- match(A,D^N+C),!.
expon_exp1(A^2,N,C^2 + 2*C*E*D^N + D*E^2) :- match(A,C+E*D^N),!.
expon_exp1(A^2,N,C^2*D) :- match(A,C*D^N),!.

expon_inv_exp1(A^2,N,C^2 + 2*C*D^N + D^(-1)) :- match(A,D^N+C),!.
expon_inv_exp1(A^2,N,C^2 + 2*C*E*D^N + D^(-1)*E^2) :- match(A,C+E*D^N),!.
expon_inv_exp1(A^2,N,C^2*D^(-1)) :- match(A,C*D^N),!.

neg_exp(A^2,N,A^2) :- wordsin(A,L),L=[],!.
neg_exp(A^2,N,X*Y) :- match(A,B*C),!,neg_exp(B^2,N,X),neg_exp(C^2,N,Y).
neg_exp(A^2,N,B^E+2*C*B^N +C^2) :-
        match(A,B1+C),
        neg_exp_match(B1,F,B,N),
        eval(2*N,E),
        !.

neg_exp(A^2,_,A^2) :- !.


neg_exp_match(Exp,1,B,N) :- match(Exp,B^N),!.
neg_exp_match(Exp,F,B,N) :- match(Exp,F*B^N),!.

trip(L,N,L1) :- append(L1,List,L),length(List,N),!.

ultiply_through(Lhs=Rhs,List,New,X) :-
        dist(Lhs,Exp),
        decomp(Exp,[+|L]),
        mult(List,L,NewL),
        recomp(NewLhs,[+|NewL]),
        free_mult(List,Rhs,NewRhs),
        weak_normal_form(NewLhs=NewRhs,X,Left=Right),
        tidy(Left=Right,New),
        !.

ist(Old,New) :- prepd(Old,New1),dist1(New1,New),!.

ist1(A+B,C+D) :- !,dist1(A,C),dist1(B,D),!.
ist1((A+B)*C,Y + Z) :- !,dist1(A*C,Y),dist1(B*C,Z).
ist1(C*(A+B),Y+Z) :- !,dist1(A*C,Y),dist1(B*C,Z).
ist1(C*(A+B)*D,Y+Z) :- !,dist1(C*D*A,Y),dist1(C*D*B,Z).
ist1(X,X) :- !.

repd(X,Y) :- decomp(X,[*|L]), prepd1(L,Y),!.
repd(X,X) :- !.

repd1(L,Y) :- get_dist(L,Mult,[],Plus),re_dist(Mult,Plus,Y),!.

et_dist([],_,_,_) :- !,fail.
et_dist([A+B|T],Prod,Acc,A+B) :- !,append(T,Acc,Prod1),recomp(Prod,[*|Prod1]).
et_dist([H|T],Ans,Acc,Plus) :- !,append([H],Acc,Newacc),
```

```
get_dist(T,Ans,Newacc,Plus).


re_dist(M1,P+Q,X+Y) :- prepd(M1*P,X),prepd(M1*Q,Y),!.


nult(Termlist,[],[]) :- !.
nult(Termlist,[H|Rest],[NewH|NewRest]) :-
        domult(Termlist,H,NewH),
        mult(Termlist,Rest,NewRest).

domult(Termlist,H,NewH) :-
        mulbag_to_list(H,Mullist),
        domult(Termlist,Mullist,NewH,1).

domult([],Args,Term*Acc,Acc) :-
        !,
        recomp(Term,[*|Args]).
domult([U^N|Rest],Args,Prod,Acc) :-
        exp_member(U,Args,NewArgs,K),
        eval(K-N,M),
        domult(Rest,NewArgs,Prod,U^M*Acc),
        !.

nulbag_to_list(H,Mullist) :- decomp(H,[*|Mullist]), !.
nulbag_to_list(H,[H]).

exp_member(U,[],[],0) :- !.
exp_member(U,[H|Rest],Rest,1) :- match(H,U), !.
exp_member(U,[H|Rest],Rest,K) :- match(H,U^K),eval(K<0), !. %fix???
exp_member(U,[H|Rest],[H|NewRest],K) :-
        exp_member(U,Rest,NewRest,K).

free_mult(List,0,0) :- !.
free_mult([],Term,Term) :- !.
free_mult([U^N|Rest],Term,NewTerm) :-
        eval(-N,M),
        free_mult(Rest,U^M*Term,NewTerm).


subnasty(_,[],[]) :- !.
subnasty(X,[H|T],[H|T1]) :- nasty(X,H),!,subnasty(X,T,T1).
subnasty(X,[_|T],T1) :- subnasty(X,T,T1),!.

subintegral([],[]) :- !.
subintegral([H|T],[H|T1]) :- integral(H),!,subintegral(T,T1).
subintegral([_|T],T1) :- subintegral(T,T1),!.

checkpt([]) :- !.
checkpt([H|T]) :- pt(H),checkpt(T),!.
```

```
   File    : HOMOG
   Author : Bernard Silver
   Updated: 6 March 1984
   Purpose: Homogenization Code for LP

 Top Level of Homogenization proper

omog(Eqn,Unk,Offend,Homeqn) :-
        findtype(Type,Offend),
        anaz(Type,Eqn,Unk,Offend,Term),
        perform_rewrites(Eqn,Term,Offend,Homeqn,Unk,Type).



% Rewrite the offenders set and obtain new Homogenized equation
erform_rewrites(Eqn,Term,Offend,Homeqn,Unk,Type) :-
        rew(Term,Offend,Sub,Unk,Type),
        subs1(Eqn,Sub,Homeqn).


 Try to choose reduced term . Arguments of anaz are
   pe of offenders set, Equation, the Unknown, the offenders set
   e reduced term.

% Trig case, find the gcd of all angles that occur, then choose fun tor

naz(trig,Eqn,Unk,Offend,Term) :-
        findangle(Unk,Offend,Angle),!,
        anaz1(Eqn,Angle,Offend,Term,Unk,_).


;  Normal log case dealing with terms like log(x,4) and log(2,x) in  he
;  offenders set.

naz(log(_),_,Unk,Offend,Term) :-
        map_laura(Offend,NewList,Arg2,Unk),
        onetest(NewList,Arg1),
        logocc(Arg1,Arg2,X,Offend).

ap_laura([],[],_,_) :- !.
   laura([H|T],[H1|T1],Arg,Unk) :-
        laura(Arg,Unk,H,H1),
        !,
        map_laura(T,T1,Arg,Unk).

%  Other log case where the logs are converted to base 10.

anaz(log(10),_,Unk,Offend,log(10,Term)) :-
        check_laura1(Unk,Term,Offend),
        !.

check_laura1(_,_,[]) :- !.
check_laura1(Unk,Term,[H|T]) :-
        laura1(Unk,Term,H),
        !,
        check_laura1(Unk,Term,T).

% Choose reduced_term  using simplicity metric

anaz(_,_,Unk,Offend,T) :-
```

```
        reduced_term(Offend,Unk,T).


 %   Find gcd of angles in offending set
findangle(Unk,Offend,Angle) :-
        map_anglesize(Offend,List,Unk,Rest),
        form(Rest,List,Angle),
        !.

map_anglesize([],[],_,_) :- !.
map_anglesize([H|T],[H1|T1],Unk,Rest) :-
        angle_size(Unk,Rest,H,H1),
        !,
        map_anglesize(T,T1,Unk,Rest).

angle_size(Unk,Rest,Term,Coeff) :-
        arg(1,Term,Arg),
        angle_size1(Unk,Rest,Arg,Coeff),
        !.

angle_size1(Unk,Rest,Arg,Coeff) :-
        match(Arg,Coeff*Rest),
        number(Coeff),
        contains(Unk,Rest),
        !.

angle_size1(Unk,Rest,Other,1) :-
        not number(Other),
        contains(Unk,Other),
        match(Other,Rest),
        !.

 % Find the reduced term
 % First,see if offending set contains only cos & sin,or sec & tan,
 % or cot & cosec.  If so eliminate (ie choose the other as reduced  erm)
 % the one that occurs to only even powers,if this happens
 % Flag indicates whether sim or solve is the top level

anaz1(Eqn,Ang,Offend,R,X,Flag) :-
        findtype_trig(Type,Offend),
        action(Type,R,Eqn,Ang,X,Flag),
        !.

 % See if equation needs tan(R) as a reduced term because equation c ntains
 % the correct functions.

anaz1(Eqn,Ang,Offend,tan(Ang),X,_) :- tantype(Offend,Ang),taneqn(Eqn X,Ang),!.

 % Otherwise,choose as reduced term the term that occurs most often

anaz1(Eqn,Ang,Offend,R,_,_) :-
        find_common(Offend,Eqn,R1,Ang),
        !,
        makenice(R1,R).

 % If no term occurs more than once,choose according to an order of  iceness

anaz1(_,Ang,Offend,R,_,_) :- anaz2(Ang,Offend,R),(R=tan(Ang) -> ! ;t ue).

 % If resulting equation can't be solved try tan(half_angle) method, hen
```

```
% this method is applicable

naz1(_,Ang,Offend,tan(R),X,_) :-
        map_anglesize(Offend,L1,X,Rest),
        ((match(Ang,M*Rest),number(M));M=1),
        half_angle(M,L1,Ang,R,Rest),
        !.

% Check to see if tan(x/2) method might work

alf_angle(M,List,Angle,Angle,_) :-
        eval(2*M,N),
        member(N,List),
        check_half_angle_check1(M,List),
        !.

alf_angle(M,List,_,A1,Rest) :-
        check_half_angle_check2(M,List),
        form2(M,Rest,A1),
        !.

% Check to see if a term occurs more than once in the equation

ind_common(L1,Eqn,R,Ang) :-
        map_occ(L1,L2,Eqn),
        great_el(L2,Ans),
        Ans>1,
        correspond(R,L1,L2,Ans),
        arg(1,R,x),
        !.

ap_occ([],[],_) :- !.
ap_occ([H|T],[H1|T1],Eqn) :-
        occ(Eqn,H,H1),
        !,
        map_occ(T,T1,Eqn).

% Check for sin_cos etc pairs

findtype_trig(sin_cos,Offend) :-
        memberchk(cos(X),Offend),
        memberchk(sin(X),Offend),
        check_cs(X,Offend),
        !.

findtype_trig(cosec_cot,Offend) :-
        memberchk(cosec(X),Offend),
        memberchk(cot(X),Offend),
        check_cc(X,Offend),
        !.

findtype_trig(sec_tan,Offend) :-
        memberchk(sec(X),Offend),
        memberchk(tan(X),Offend),
        check_st(X,Offend),
        !.


action(Type,R,Eqn,Ang,X,Flag) :-
        parse2(Eqn,X,Offend),
```

```
        action1(Type,R,Offend,Ang,Flag),
        !.

 % If one of pair occurs only to even powers eliminate it
action1(sin_cos,sin(A),Offend,A,_) :-
        map_cosp(Offend,L1,A),
        check_even(L1),
        !.

action1(sin_cos,cos(A),Offend,A,_) :-  !.

action1(sec_tan,tan(A),Offend,A,_) :-
        map_secp(Offend,L1,A),
        check_even(L1),
        !.

action1(sec_tan,sec(A),Offend,A,_) :- !.

action1(cosec_cot,cot(A),Offend,A,_) :-
        map_cosecp(Offend,L1,A),
        check_even(L1),
        !.

action1(cosec_cot,cosec(A),Offend,A,_) :- !.

map_cosp([],[],_) :- !.
map_cosp([H|T],[H1|T1],A) :-
        cosp(A,H,H1),
        !,
        map_cosp(T,T1,A).

map_secp([],[],_) :- !.
map_secp([H|T],[H1|T1],A) :-
        secp(A,H,H1),
        !,
        map_secp(T,T1,A).

map_cosecp([],[],_) :- !.
map_cosecp([H|T],[H1|T1],A) :-
        cosecp(A,H,H1),
        !,
        map_cosecp(T,T1,A).



 % Check for tan case
tantype([],_) :- !.
tantype([H|T],X) :- tantype1(H,X),!,tantype(T,X).

tantype1(tan(_),_) :- !.
tantype1(cot(_),_) :- !.
tantype1(sec(X),Y) :- match(X,Y),!.
tantype1(cosec(X),Y) :- match(X,Y),!.

taneqn(Eqn,X,Ang) :- parse2(Eqn,X,Offend),check_tan(Offend,Ang),!.

check_tan([],_) :- !.
check_tan([H|T],Ang) :- check_tan1(H,Ang),!,check_tan(T,Ang).

check_tan1(tan(_),_) :- !.
```

```
:heck_tan1(cot(_),_) :- !.
:heck_tan1(sec(Ang)^N,Ang1) :- integer(N),even(N),match(Ang,Ang1),!.
:heck_tan1(cosec(Ang)^N,Ang1) :- integer(N),even(N),match(Ang,Ang1), .

 % Choose reduced term in order of niceness

anaz2(Ang,Offend,sin(Ang)) :-
        member(sin(Ang),Offend),
        member(cosec(Ang),Offend),
        !.

anaz2(Ang,Offend,cos(Ang)) :-
        member(cos(Ang),Offend),
        member(sec(Ang),Offend),
        !.

anaz2(Ang,Offend,cos(Ang)) :-
        member(cos(Ang),Offend),
        member(cos(X),Offend),
        diff(X,Ang),
        !.

a  z2(Ang,Offend,sin(Ang)) :- member(sin(Ang),Offend),!.

anaz2(Ang,Offend,cos(Ang)) :- member(cos(Ang),Offend),!.

anaz2(Ang,Offend,cos(Ang)) :- member(sec(Ang),Offend),!.

anaz2(Ang,Offend,sin(Ang)) :- member(cosec(Ang),Offend),!.

anaz2(Ang,Offend,sin(Ang)) :- some(sinfind,Offend),!.

anaz2(Ang,Offend,cos(Ang)) :- some(cosfind,Offend),!.

anaz2(Ang,Offend,sin(Ang)) :- some(cosecfind,Offend),!.

anaz2(Ang,Offend,cos(Ang)) :- some(secfind,Offend),!.

anaz2(Ang,_,tan(Ang)) :- !.


   X,sin(X)) :- !.
os(X,cos(X)) :- !.
cc(X,cot(X)) :- !.
cc(X,cosec(X)) :- !.
st(X,sec(X)) :- !.
st(X,tan(X)) :- !.

sinfind(sin(_)) :- !.
cosfind(cos(_)) :- !.
secfind(sec(_)) :- !.
cosecfind(cosec(_)) :- !.

 % Recognize powers of trig functions in the equation
cosp(Ang,cos(Ang)^N,N) :- integer(N),!.
cosp(Ang,cos(Ang),1) :- !.
cosp(_,_,0) :- !.
secp(Ang,sec(Ang)^N,N) :- integer(N),!.
secp(Ang,sec(Ang),1) :- !.
secp(_,_,0) :- !.
```

```
cosecp(Ang,cosec(Ang)^N,N) :- integer(N),!.
cosecp(Ang,cosec(Ang),1) :- !.
cosecp(_,_,0) :- !.


makenice(cosec(X),sin(X)) :- !.
makenice(sec(X),cos(X)) :- !.
makenice(cot(X),tan(X)) :- !.
makenice(X,X) :- !.

 % expss(P,Q,X,T) expresses sin(Z) in terms of sin(X) where Z/X=Q/P
 % expcs expresses cos(Z) in terms of sin(X) etc.    The 4
 % functions are more or less mutually recursive, but expcc does
 % not depend on the others, though they call it

expss(P,P,X,sin(X)) :- !.

expss(P,Q,X,2*sin(X)*(1-sin(X)^2)^(1/2)) :- eval(Q/P=:=2),!.

expss(P,Q,X,(3*sin(X)-4*sin(X)^3)) :- eval(Q/P=:=3),!.

 % Where Q/P is odd a simple series expansion can be applied
   ss(P,Q,X,A) :- eval(Q/P,N),eval(N mod 2,1),!,sinexp(sin(X),N,0,A)

 % sin(Y) = sin((Y-3*X) + 3*X) = sin(3*X)*cos(Y-3*X) + cos(3*X)*sin( -3*X)
 % We can now express each of these 4 terms in terms of sin(X) as
 % a recursive step. The 4 terms are A,B,C and D below.

expss(P,Q,X,(A*B+C*D)) :-
        eval(3*P,P1),
        eval(Q-3,Q1),
        expss(P,P1,X,A),
        expcs(P,Q1,X,B),
        expcs(P,P1,X,C),
        expss(P,Q1,X,D),
        !.

 % Similarly for sin in terms of cos
expsc(P,P,X,(1-cos(X)^2)^(1/2)) :- !.

expsc(P,Q,X,2*cos(X)*(1-cos(X)^2)^(1/2)) :-eval(Q/P=:=2),!.

  sc(P,Q,X,(4*cos(X)^2-1)*(1-cos(X)^2)^(1/2)) :- eval(Q/P=:=3),!.

expsc(P,Q,X,(A*B+C*D)) :-
        eval(3*P,P1),
        eval(Q-3,Q1),
        expsc(P,P1,X,A),
        expcc(P,Q1,X,B),
        expcc(P,P1,X,C),
        expsc(P,Q1,X,D),
        !.

 %  cos in terms of sin
expcs(P,P,X,(1-sin(X)^2)^(1/2)) :- !.

expcs(P,Q,X,(1-2*sin(X)^2)) :-eval(Q/P=:=2),!.

expcs(P,Q,X,(1-4*sin(X)^2)*(1-sin(X)^2)^(1/2)) :- eval(Q/P=:=3),!.
```

```
expcs(P,Q,X,(A*B-C*D)) :-
        eval(3*P,P1),
        eval(Q-3,Q1),
        expcs(P,P1,X,A),
        expcs(P,Q1,X,B),
        expss(P,P1,X,C),
        expss(P,Q1,X,D),
        !.

 % Series exists for cos in terms of cos
expcc(P,Q,X,Y) :- eval(Q/P,N),cosexp(cos(X),N,0,Y),!.

 % Base case, series complete
cosexp(A,N,R,X) :- eval(2*R,R1),eval(R1+1,R2),(N=R1;N=R2),coeff1(A,N R,X),!.

 % Recurse
cosexp(X1,N,R,X-(Y)) :- coeff1(X1,N,R,X),eval(R+1,R1),!,cosexp(X1,N, 1,Y).

 % Produce the coefficients for the series, very ugly

coeff1(Fang,N,R,X*(ZZ)) :-
        fact(R,R1),
        eval(N-2*R-1,N1),
        eval(N-R-1,N2),
        eval(N1+1,N3),
        fact(N2,Z2),
        fact(N3,Z3),
        eval((2^N1*N*Z2)/(R1*Z3),X),
        form4(Fang,N3,ZZ),
        !.

 % The sin expansion for odd Q/P is very similar to cos cos series
sinexp(X,N,A,B*(Z)) :- eval((-1)^((N-1)/2),B),cosexp(X,N,A,Z),!.

 % Expand tan(n*x) in terms of tan(m*x)  (m < n)
 % Tan produces a numerator and denominator series.

exptt(I,J,X,(Z)/(Y)) :-
        eval(J/I,N),
        tanexp_num(tan(X),N,1,Z),
        tanexp_denom(tan(X),N,0,Y),
        !.

 % Obtain numerator
tanexp_num(A,N,R,X) :- eval(R+1,R1),(N=R1;N=R),coeff2(A,N,R,X),!.
tanexp_num(A,N,R,X-(Y)) :-
        coeff2(A,N,R,X),
        eval(R+2,R1),
        !,
        tanexp_num(A,N,R1,Y).


 % Obtain the denominator
tanexp_denom(A,N,R,X) :- eval(R+1,R1),(N=R1;N=R),coeff2(A,N,R,X),!.
tanexp_denom(A,N,R,X-(Y)) :-
        coeff2(A,N,R,X),
        eval(R+2,R1),
        !,
        tanexp_denom(A,N,R1,Y).
```

```
 % Different coefficients from the other series

coeff2(A,N,R,X*(ZZ)) :- calc_coeff(N,R,X),form4(A,R,ZZ),!.

calc_coeff(N,R,X) :-
        fact(R,Rfact),
        fact(N,Nfact),
        eval(N-R,P),
        fact(P,Pfact),
        eval(Nfact/(Pfact*Rfact),X),
        !.

 % Modified checklists
check_cs(_,[]) :- !.
check_cs(X,[H|T]) :- cs(X,H),check_cs(X,T).


check_cc(_,[]) :- !.
check_cc(X,[H|T]) :- cc(X,H),check_cc(X,T).

check_st(_,[]) :- !.
check_st(X,[H|T]) :- st(X,H),check_st(X,T).


check_half_angle_check1(_,[]) :- !.
check_half_angle_check1(A,[H|T]) :-
        half_angle_check1(A,H),
        check_half_angle_check1(A,T).


check_half_angle_check2(_,[]) :- !.
check_half_angle_check2(A,[H|T]) :-
        half_angle_check2(A,H),
        check_half_angle_check2(A,T).


check_even([]) :- !.
check_even([H|T]) :- even(H),check_even(T).

fact(X,Y) :-
        fact(X,Y,1).

fact(0,X,X).
fact(N,Ans,Acc) :-
        eval(N*Acc,NewAcc),
        eval(N-1,M),
        !,
        fact(M,Ans,NewAcc).


                        % Check is the tan(half-angle) method can b  used
half_angle_check1(M,M) :- !.
half_angle_check1(M,N) :- eval(2*M,N),!.

half_angle_check2(M,M) :- !.

                        % Standard log case
laura(B,X,log(A,B),A) :- freeof(X,A),!.
laura(A,X,log(A,B),B) :- freeof(X,B),!.
```

```
                        %  Convert to log base 10 case
laura1(Unk,Term,log(A,Term)) :-
        number(A),
        contains(Unk,Term),
        !.

laura1(Unk,Term,log(Term,A)) :-
        number(A),
        contains(Unk,Term),
        !.

 % When the terms are being raised to powers the reduced term should
 % be the smallest if all terms are less than one, the largest
 % if they are all greater than one, otherwise unless they are
 % all the same (listtoset is a singleton) fail

onetest(K,A) :- check_moreone(K),least_el(K,A),!.
onetest(K,A) :- check_lessone(K),great_el(K,A),!.
onetest(K,A) :- listtoset(K,[A]),!.

check_moreone([]) :- !.
check_moreone([H|T]) :-
        moreone(H),
        !,
        check_moreone(T).
check_lessone([]) :- !.
check_lessone([H|T]) :-
        lessone(H),
        !,
        check_lessone(T).

 % Choosing the reduced term in the log case, we choose it to have
 % the term containing the unknown as its second argument, whether o
 % not this log term occurred in the original equation
logocc(A,B,log(A,B),L) :- member(log(A,B),L),!.
logocc(A,B,log(B,A),L) :- member(log(B,A),L),!.

 % These form functions put terms together prettily,so 1*A is A for  xample

   m(Unk,K,Z) :-rational_gcd_list(K,Gcd),absol(Gcd,Gcd1),!,form1(Unk Gcd1,Z).

form1(Unk,A,Res) :- tidy(A*Unk,Res),!.

form2(M,Rest,Res) :- !,tidy(Rest*M/2,Res).

form4(_,0,1) :- !.
form4(A,1,A) :- !.
form4(A,N,A^N) :- !.


 % Parser for trig method
parse2(Exp,X,L) :- dl_parse2(Exp,X,L1-[]),!,listtoset(L1,L).

dl_parse2(A=B,X,L-L1) :- !,dl_parse2(A,X,L-L2),dl_parse2(B,X,L2-L1).
dl_parse2(A*B,X,L-L1) :- !,dl_parse2(A,X,L-L2),dl_parse2(B,X,L2-L1).
dl_parse2(A+B,X,L-L1) :- !,dl_parse2(A,X,L-L2),dl_parse2(B,X,L2-L1).
dl_parse2(A^N,_,[A^N|L]-L) :- integer(N),trigf(A),!.
dl_parse2(A^N,X,L) :- number(N),dl_parse2(A,X,L),!.
dl_parse2(A,X,L-L) :- freeof(X,A),!.
```

```
dl_parse2(A,X,[A|L]-L) :- !.

 % Find the "smallest" term in the  offenders set

reduced_term([Unk],Unk,_) :- !,fail.            %Unk can't be the re uced term
reduced_term([A],Unk,A) :- !.
reduced_term(L,Unk,A) :-
        extreme_term(L, <, A),  %  return the smallest
        !,
        A \= Unk.


 % Find the smallest and largest elements of a list of numbers
least_el([Hd],Hd) :- !.
least_el([Hd|Tl],Ans) :- least_el(Tl,Lwr),(eval(Hd < Lwr) -> Hd=Ans; wr=Ans),!.

lessone(A) :- number(A),eval(A < 1),!.

moreone(A) :- number(A),eval(A > 1),!.

 % Absolute value
~bsol(X,X1) :- eval(sign(X)*X,X1),!.

 % Given terms A and B break(A,B,I,J) finds I and J
 % so that A=I*Y,and B=J*Y,if this is possible
break(A,B,1,1) :- match(A,B),!.
break(A,B,1,C) :- number(A),number(B),eval(B/A,C),!.
break(A,B,1,J1) :- match(A,I*Y),number(I),match(B,Y*J),number(J),eva (J/I,J1),!.
break(A,B,1,J) :- match(B,J*A),number(J),!.
break(A,B,J,1) :- match(A,J*B),number(J),!.


/* Try to rewrite each of the terms in the offending set as a
    function of the reduced term */

rew(X,L,Subs,Unk,Type) :-  newtype(Type,New),
        map_rew1(L,L1,New,X,Unk),
        make_subl(L,L1,Subs),
        !.

---_rew1([],[],_,_,_) :- !.
   _rew1([H|T],[H1|T1],New,X,Unk) :-
        rew1(New,X,Unk,H,H1),
        !,
        map_rew1(T,T1,New,X,Unk).

                        %  Kludge for stopping recursive calls of re -rule
                        %  in mixed case, and for getting the log ca e right
newtype(mixed,_) :- !.
newtype(log(X),log) :- X \== 10.
newtype(C,C) :- !.

rew1(_,X,_,X,X) :- !.
rew1(Type,A^B,Unk,Old,New) :- !,rew_rule(Type,A^B,Old,New,Unk).
rew1(Type,X,Unk,A^B,C^D) :- rew1(Type,X,Unk,A,C),rew1(Type,X,Unk,B,D ,!.
rew1(Type,X,Unk,Old,New) :- rew_rule(Type,X,Old,New,Unk),!.

/* rew_rule(Type,Term1,Term2,Exp,Unk) gives Exp as a rewrite of Term  in terms */
/* of Term1,where Unk is the  unknown, and the rule is for type Type */
```

```
/* Special cases   */
rew_rule(_,X,Y,X,_) :- match(X,Y),!.

rew_rule(_,_,Y,Y,Unk) :- freeof(Unk,Y),!.

/* Trignometric Rewrite rules */
rew_rule(T,sin(X),sin(Z),V*cos(C) + V1*sin(C),U) :- T == trig,
        match(Z,B + C),
        contains(U,B),
        freeof(U,C),
        rew_rule(trig,sin(X),sin(B),V,U),
        rew_rule(trig,sin(X),cos(B),V1,U),
        !.

rew_rule(T,sin(X),cos(Z),V*cos(C) - V1*sin(C),U) :- T == trig,
        match(Z,B + C),
        contains(U,B),
        freeof(U,C),
        rew_rule(trig,sin(X),sin(B),V1,U),
        rew_rule(trig,sin(X),cos(B),V,U),
        !.

   _rule(T,cos(X),sin(Z),V*cos(C) + V1*sin(C),U) :- T == trig,
        match(Z,B + C),
        contains(U,B),
        freeof(U,C),
        rew_rule(trig,cos(X),sin(B),V,U),
        rew_rule(trig,cos(X),cos(B),V1,U),
        !.

rew_rule(T,cos(X),cos(Z),V*cos(C) - V1*sin(C),U) :- T == trig,
        match(Z,B + C),
        contains(U,B),
        freeof(U,C),
        rew_rule(trig,cos(X),cos(B),V,U),
        rew_rule(trig,cos(X),sin(B),V1,U),
        !.


rew_rule(trig,sin(X),cos(Z),V,_) :- break(X,Z,P,Q),
        absol(Q,Q1),
        expcs(P,Q1,X,V),
        !.

rew_rule(trig,sin(X),sin(Z),I*(V),_) :- break(X,Z,P,Q),
        absol(Q,Q1),
        eval(sign(Q),I),
        expss(P,Q1,X,V),
        !.

rew_rule(trig,cos(X),sin(Z),I*(V),_) :- break(X,Z,P,Q),
        absol(Q,Q1),
        eval(sign(Q),I),
        expsc(P,Q1,X,V),
        !.

rew_rule(trig,cos(X),cos(Z),V,_) :- break(X,Z,P,Q),
        absol(Q,Q1),
        expcc(P,Q1,X,V),
        !.
```

```
rew_rule(trig,tan(X),sec(X),(1+tan(X)^2)^(1/2),_) :- !.

rew_rule(trig,sec(X),tan(X),(sec(X)^2-1)^(1/2),_) :- !.

rew_rule(trig,cot(X),cosec(X),(1+cot(X)^2)^(1/2),_) :- !.

rew_rule(trig,cosec(X),cot(X),(cosec(X)^2-1)^(1/2),_) :- !.

rew_rule(T,tan(X),tan(Z),(V + tan(C))/(1 - tan(C)*V),U) :- T == trig
        match(Z,B + C),
        contains(U,B),
        freeof(U,C),
        rew_rule(trig,tan(X),tan(B),V,U),
        !.

rew_rule(trig,tan(X),tan(Z),I*(V),_) :- break(X,Z,P,Q),
        absol(Q,Q1),
        eval(sign(Q),I),
        exptt(P,Q1,X,V),
        !.

   _rule(trig,tan(X),cosec(X),(1+tan(X)^2)^(1/2)/tan(X),_) :- !.

rew_rule(trig,tan(X),sin(X),tan(X)/(1+tan(X)^2)^(1/2),_) :- !.

rew_rule(trig,tan(X),cos(X),1/(1+tan(X)^2)^(1/2),_) :- !.

/* Tan half-angle Rewrite rules */
rew_rule(trig,tan(X),sin(Z),2*tan(X)*(1+tan(X)^2)^(-1),_) :-break(X, ,P,Q),
        eval(Q/P=:=2),!.

rew_rule(trig,tan(X),cos(Z),(1-tan(X)^2)*(1+tan(X)^2)^(-1),_) :-brea (X,Z,P,Q),
        eval(Q/P=:=2),!.

/* Reciprocal function Rewrite rules  */
rew_rule(T,X,tan(Z),A*B^ -1,Unk) :- T == trig,
        rew_rule(trig,X,sin(Z),A,Unk),
        rew_rule(trig,X,cos(Z),B,Unk),
        !.

   _rule(T,A,sec(Z),(B)^ -1,Unk) :- T == trig,
        rew_rule(trig,A,cos(Z),B,Unk),
        !.

rew_rule(T,A,cosec(Z),(B)^ -1,Unk) :-  T == trig,
        rew_rule(trig,A,sin(Z),B,Unk),
        !.

rew_rule(T,A,cot(Z),(B)^ -1,Unk) :- T == trig,
        rew_rule(trig,A,tan(Z),B,Unk),
        !.


/* Logarithmic Rewrite rules  */
rew_rule(log,log(X,Y),log(Y,X),log(X,Y)^ -1,_) :- !.

rew_rule(log,log(X,Y),log(Z,Y),N*log(X,Y),_) :- powered(X,N,Z),!.

rew_rule(log,log(X,Y),log(Y,Z),N*log(X,Y)^ -1,_) :- powered(X,N,Z),!
```

```
rew_rule(log,log(X,Y),log(X,Z),N*log(X,Y),_) :- powered(Y,N,Z),!.

rew_rule(log,log(X,Y),log(Z,X),N*log(X,Y)^ -1,_) :- powered(Y,N,Z),!

                          %  Reduced term is log base 10
rew_rule(log(10),log(10,X),log(X,10),log(10,X)^ -1,_) :- !.

rew_rule(log(10),log(10,X),log(A,X),Term,Unk) :-
        number(A),
        tidy(log(10,X)/log(10,A),Term),
        !.

rew_rule(log(10),log(10,X),log(X,A),Term,Unk) :-
        number(A),
        tidy(log(10,A)*(log(10,X)^ -1),Term),
        !.


rational_gcd_list([H|T],Gcd) :-
        eval(numer(H),N),
        eval(denom(H),D),
        rgl(T,N,D,Gcd).

rgl([],N,D,Gcd) :- eval(N/D,Gcd).
rgl([H|T],N,D,Gcd) :-
        eval(numer(H),N1),
        eval(denom(H),D1),
        eval(gcd(N*D1,D*N1),G),
        eval(D*D1,Dnew),
        rgl(T,G,Dnew,Gcd).
```

AXIOMS :

Bernard Silver
Updated: 11 February 1984

```
solax( 1 , -U=V , U= -1*V).
solax( 1 , U+V=W , U=W+(-1)*V).
solax( 2 , V+U=W , U=W+(-1)*V).
solax( 1 , U*V=W , U=W*V1) :- non_zero(V),tidy(1/V,V1).
solax( 2 , V*U=W , U=W*V1) :- non_zero(V),tidy(1/V, V1).
solax( 1 , log(U,1)=0 , U=N) :- arbint(N).
solax( 1 , log(U,V)=W , U=V^W1) :- non_zero(W),tidy(1/W,W1).
solax( 2 , log(U,V)=W , V=U^W).
solax( 1 , U^0 = K , U=N) :- K=1,!,arbint(N).
solax( 1 , _^0 = _ , false) :- !.
solax( 1 , _^N = 0 , false) :- negative(N),!.
solax( 1 , U^N=V , U=V^N1) :- odd(N),tidy(1/N, N1).
solax( 1 , U^N=V , U=V^N1) :-  non_neg(U),even(N),tidy(1/N, N1).

solax( 1 , U^N=V , U=V^N1 # U=(-1)*(V^N1)) :-even(N),tidy(1/N, N1).

   ax( 1 , _^A=V, false) :- negative(V),integer(A),even(A),!.
 ax( 1 , U^A=V, U=V^A1) :- not number(A),tidy(1/A,A1).

solax( 2 , U^V=_ , false) :- (eval(U>0);atom(U)),eval(V=<0),!.
solax( 2 , U^V=W , V=log(U,W)) .
solax( 1 ,sin(U)=0,U=180*N) :- arbint(N).
solax( 1 ,sin(U)=1,U=360*N+90) :- arbint(N).
solax( 1 ,sin(U)= -1,U=360*N-90) :- arbint(N).
solax( 1, sin(_)=V,false) :- (eval(V>1);eval(V< -1)),!.
solax( 1 , sin(U)=V , U=N*180+ (-1)^N*arcsin(V)) :-  arbint(N).

solax( 1 ,cos(U)=1,U=360*N) :- arbint(N).
solax( 1 ,cos(U)= -1,U=360*N+180) :- arbint(N).
solax( 1 ,cos(U)=0,U=180*N+90) :- arbint(N).
solax( 1, cos(_)=V,false) :- (eval(V>1);eval(V< -1)),!.
solax( 1 , cos(U)=V , U=2*N*180+arccos(V) #
                          U=2*N*180+ ((-1)*arccos(V))) :- arbint(N).
solax( 1 , tan(U)=V , U=N*180+arctan(V)) :- arbint(N).

   ax( 1 , cosec(_)=V , false) :- eval(V> -1),eval(V<1),!.
 ax( 1 , cosec(U)=V , U=N*180+ (-1)^N*arcsin(V1)) :- !,
      tidy(1/V,V1),
      arbint(N).

solax( 1 , sec(_)=V , false) :- eval(V> -1),eval(V<1),!.
solax( 1 , sec(U)=V , U=2*N*180+arccos(V1) #
                          U=2*N*180+ ((-1)*arccos(V1))) :- !,
      tidy(1/V,V1),
      arbint(N).

solax( 1 , cot(U) = 0, U = N*180 + 90) :- !,arbint(N).
solax( 1 , cot(U)=V , U=N*180+arctan(V1)) :- tidy(1/V,V1),arbint(N).

solax( 1, arcsin(_)=U,false) :- (eval(U>1);eval(U< -1)),!.

solax( 1 , arcsin(U)=V , U=sin(V)).
solax( 1, arccos(U)=_,false) :- (eval(U>1);eval(U< -1)),!.
```

```
solax( 1 , arccos(U)=V , U=cos(V)).
solax( 1 , arctan(U)=V , U=tan(V)) .
solax( 1 , arccosec(U)=_ , false) :- eval(U> -1),eval(U<1),!.
solax( 1 , arccosec(U)=V , U=V1) :- tidy(1/sin(V),V1),!.
solax( 1 , arcsec(U)=_ , false) :- eval(U> -1),eval(U<1),!.
solax( 1 , arcsec(U)=V , U=V1) :- tidy(1/cos(V),V1),!.
solax( 1 , arccot(U)=V , U=V1) :- tidy(1/tan(V),V1),!.

ollax( W , U*W+V*W , (U+V)*W ) .

ollax( W , W+V*W , (V+1)*W ) .

ollax( W , W+W , 2*W ) .

ollax( U&V , (U+V)*(U+ (-1*V)) , U^2+ -1*(V^2) ) .

ollax( W , W^U*W^V , W^(U+V) ) .

ollax( W , W*W^V , W^(V+1) ) .

ollax( W , W*W , W^2 ) .

 ax( U , sin(U)*cos(U) , sin(2*U)* (1/2) ) .

ollax( U , cos(U)^2+ -1*(sin(U)^2) , cos(2*U) ) .

ollax( U , sin(U)*cos(V)+cos(U)*sin(V) , sin(U+V) ) .

ollax( U&V , sin(U)*cos(V)+ -1*(cos(U)*sin(V)) , sin(U+ (-1*V)) ) .

ollax( U , cos(U)*cos(V)+ -1*(sin(U)*sin(V)) , cos(U+V) ) .

ollax( U , cos(U)*cos(V)+sin(U)*sin(V) , cos(U+ (-1*V)) ) .

ollax( U , cos(U)^2 + sin(U)^2 , 1 ).

ollax( U , log(U,X) + log(U,Y) , log(U,X*Y) ) .

ollax( U , A*log(U,X) + B*log(U,Y),log(U,X^A*Y^B) ).

ollax( U, A*log(U,X) + log(U,Y),log(U,X^A*Y) ).

 ax( U & V , U*W+V*W , (U+V)*W , []) .

:trax( U & V , W^U*W^V , W^(U+V) , [] ) .

:trax( U & V , log(W,U)+log(W,V) , log(W,U*V) , [] ) .

:trax( U & V , A*log(W,U)+B*log(W,V),log(W,U^A*V^B) , [] ) .

:trax( U & V , A*log(W,U)+log(W,V),log(W,U^A*V) , [] ) .

:trax( U & V , U*log(W,V) , log(W,V^U) , [] ) .

:trax( U & V , log(W,V)*log(U,W) , log(U,V) , [] ) .

:trax( U & V , U=V , U+(-1*V)=0 , [] ) .

:trax( V & W , (U^V)^W , U^(V*W) , [] ) .
```

ttrax(U & V , U^(V*W) , (U^V)^W , []) .

```
  File   : WEAKNF
  Author : Bernard Silver
  Updated: 23 February 1984
  Purpose: Weak normal forms for LP

 public
      weak_normal_form/3,
      mod_weak_normal_form/3,
      mod_weak_normal_form1/4,
      mod_weak_normal_form2/3.

 mode

      filter(+, +, -, -),
      mod_weak_normal_form(+,+,?),
      mod_weak_normal_form1(+,+,+,?),
      mod_weak_normal_form2(+,+,?),
      weak_normal_form(+, +, -),
      zero_rhs(+, -).

 Put equation(s) into weak normal form
ak_normal_form(Eqn, Var, New) :-
      tidy_expr(Eqn,Tidy),
      zero_rhs(Tidy, Mid),
      decomp(Mid, [+|Bag]),
      filter(Bag, Var, Lhs, Rhs),
      tidy_expr(Lhs=Rhs, New), !.

ak_normal_form(Eqn, Var, New=0)  :- zero_rhs(Eqn,New),!.

  put an equation Lhs=Rhs into the form New=0.

      zero_rhs(Lhs=0, Lhs) :- !.
      zero_rhs(Lhs=Rhs, New) :- tidy(Lhs-Rhs, New).

  split a sum bag into Lhs, holding all elements containing Var,
  and Rhs, holding all the elements not containing Var.  We are
  free to use '-' in Rhs, as it will be tidied before use.

      filter([Head|Tail], Var, Head+More, Rest) :-
              contains(Var, Head), !,
              filter(Tail, Var, More, Rest).
      filter([Head|Tail], Var, More, Rest-Head) :- !,
              filter(Tail, Var, More, Rest).
      filter([],          Var, 0,    0).


d_weak_normal_form(Exp,X,Ans) :-
      mod_weak_normal_form1(Exp,normal,X,Ans),
      (match_check(Exp,Ans);
      writef('\nPutting in weak normal form to obtain\n\n%t\n',[Ans])),
      !.

d_weak_normal_form(Old,_,Old).

 m_w_n_f1 can be called on its own if we don't want the message output


d_weak_normal_form1(A#B,Type,X,New) :- !,
      mod_weak_normal_form1(A,Type,X,A1),
```

```
      mod_weak_normal_form1(B,Type,X,B1),
      type_tidy(Type,A1#B1,New).

i_weak_normal_form1(Exp,Type,X,Ans) :-
      type_tidy(Type,Exp,Exp1),
      weak_normal_form(Exp1,X,Ans1),
      type_tidy(Type,Ans1,Ans).


 Normal case
i_weak_normal_form2(A=B,X,H1) :-
      contains(X,A),
      !,
      mod_weak_normal_form1(A=B,expr,X,H1).

 A is an atom and so is probably a new variable.  Call m_w_n_f1 with A
 as unknown
i_weak_normal_form2(A=B,_,H1) :-
      atom(A),
      !,
      mod_weak_normal_form1(A=B,expr,A,H1).

    .e expression is an atom so return it
   ak_normal_form2(A,_,A) :- atomic(A),!.

 Exp does not contain the old unknown so guess it!
i_weak_normal_form2(Expr,X,H1) :-
      freeof(X,Expr),
      !,
      wordsin(Expr,Words),
      member(Y,Words),
      !,
      mod_weak_normal_form1(Expr,expr,Y,H1).

 All other cases, e.g Term is a disjunction
i_weak_normal_form2(Term,X,New) :-
      mod_weak_normal_form1(Term,expr,X,New).
```

```
* FUNC : Check LP knows the functions in the problems

                                                    Bernard Silver
                                                    Updated: 9 April 83
/

ind_functions(List,X,Flag) :-
        find_functions1(List,X,Flag),
        abolish(new_functor,2),
        !.

ind_functions1([],_,_).
ind_functions1([H|T],X,F)  :-
        function_parse(H,H1,X),
        check_ok_set(H1,X,F),
        !,
        find_functions1(T,X,F).


unction_parse(Eqn,Set,Unk) :- dl_parsef(Eqn,Set1-[],Unk),listtoset(Set1,Set).

l_parsef(A#B,L-L1,Unk) :- !,dl_parsef(A,L-L2,Unk),dl_parsef(B,L2-L1,Unk).
  rsef(A=B,L-L1,Unk) :- !,dl_parsef(A,L-L2,Unk),dl_parsef(B,L2-L1,Unk).
  arsef(A+B,L-L1,Unk) :- !,dl_parsef(A,L-L2,Unk),dl_parsef(B,L2-L1,Unk).
l_parsef(A*B,L-L1,Unk) :- !,dl_parsef(A,L-L2,Unk),dl_parsef(B,L2-L1,Unk).
l_parsef(A^B,L-L1,Unk) :- !,dl_parsef(A,L-L2,Unk),dl_parsef(B,L2-L1,Unk).
l_parsef(Unk,L-L,Unk) :- !.
l_parsef(A,L-L,Unk) :- freeof(Unk,A),!.
l_parsef(A,[A|L]-L,Unk) :- !.

heck_ok_set([],_,_) :- !.
heck_ok_set([H|T],X,F) :-
        check_ok_term(H,X,F),
        !,
        check_ok_set(T,X,F).

:heck_ok_term(Term,X,_) :- freeof(X,Term),!.
:heck_ok_term(Term,X,F) :-
        known_functor(Term,N),
        check_ok_args(Term,N,X,F).


  k_ok_term(Term,X,check) :-
        functor(Term,F,N),
        (new_functor(F,N);
        (writef('\n[**Warning, LP has no rules for functor %t/%t**]\n',[F,N]),
        asserta(new_functor(F,N)))),
        check_ok_args(Term,N,X,check),
        !.

check_ok_term(Term,_,add) :-
        functor(Term,F,N),
        writef('\n[Adding functor %t/%t to list of known functors]\n',[F,N]),
        mod_assert(k_functor(F,N)),
        !.

known_functor(Term,N) :-
        functor(Term,F,N),
        k_functor(F,N),
        !.
```

```
:_functor(+,2).
:_functor(*,2).
:_functor(^,2).

:_functor(log,2).

:_functor(sin,1).
:_functor(cos,1).
:_functor(tan,1).
:_functor(cot,1).
:_functor(sec,1).
:_functor(cosec,1).

:_functor(arcsin,1).
:_functor(arccos,1).
:_functor(arctan,1).
:_functor(arccot,1).
:_functor(arcsec,1).
:_functor(arccosec,1).

'*
   nctor(sinh,1).
_ nctor(cosh,1).
_functor(tanh,1).
_functor(coth,1).
_functor(sech,1).
_functor(cosech,1).


_functor(arcsinh,1).
_functor(arccosh,1).
_functor(arctanh,1).
_functor(arccoth,1).
_functor(arcsech,1).
_functor(arccosech,1).

/

heck_ok_args(Term,N,Unk,F) :-
        c_f_a(N,0,Unk,Term,F).

  _a(N,N,_,_,_) :- !.
_f_a(N,M,Unk,Term,F)  :-
        K is M + 1,
        arg(K,Term,Arg),
        c_ok_arg1(Unk,Arg,F),
        !,
        c_f_a(N,K,Unk,Term,F).

_ok_arg1(Unk,Arg,F) :- freeof(Unk,Arg),!.


_ok_arg1(Unk,Arg,F) :-
        function_parse(Arg,S,Unk),
        check_ok_set(S,Unk,F),
        !.
```

```
   File   : LOOP
   Author : Bernard Silver
   Updated: 23 February 1984
   Purpose: Loop checking for LP


oping(Eqn,X) :- flag(loop,Old,Old),looping_check(Eqn,X,Old),!.

oping_check(_,_,no) :- !.
oping_check(Eqn,X,Old) :-
       normstore(Eqn,X,Eqn1),
       ((seen_eqn(Eqn1) -> looping_action(Old));
       asserta(seen_eqn(Eqn1))).

oping_action(X) :-
       (X=warn;X=warn1),
       !,
       writef('\n[**Warning. Equation has been seen before**]\n').

oping_action(yes) :- !,
       writef('\n**Looping**.  This equation has been seen before.\n'),
       process_reply([a,b,c,n,w],loop_action(X),X,'Action:',l_text).

text :-
       writef('\nType

       a  : to abort,

       b  : to break,

       c  : to continue,

       n  : to switch off loop test,

       w  : to switch loop test to warn only.\n').

op_action(a) :- !,writef('\n[Aborting]\n'),abort.
op_action(b) :- !,
       writef('\n[Entering break]\n'),
       break,
       writef('\n[Leaving break and continuing]\n').
   action(c) :- !,writef('\n[Continuing]\n').
_action(n) :- !,
       flag(loop,_,no),
       writef('\n[Disabling loop checker]\n').
op_action(w) :- !,
       flag(loop,_,warn),
       writef('\n[Setting loop checker to warn only]').
op_action(_) :- writef('\nNot a valid option! Please try again.\n'),fail.

rmstore(Eqn,X,Eq) :-
       subst(X  = unk,Eqn,Eqn1),
       !,
       remove_arbs(Eqn1,Eq),
       !.



move_arbs(Eqn1,Eqn2) :-
```

```
        wordsin(Eqn1,Words),
        subintegral(Words,Word),
        remove_arbs1(Eqn1,Word,Eqn3),
        tidy(Eqn3,Eqn2),
        !.

:move_arbs1(X,[],X) :- !.
:move_arbs1(X,H,Y) :- make_arblist(H,Z),make_subl(H,Z,Y1),subs1(X,Y1,Y),!.

:ke_arblist(H,Z) :- make_arblist1(H,Z,1),!.

:ke_arblist1([],[],_) :- !.
:ke_arblist1([_|T],[arb(N)|T1],N) :- M is N+1,make_arblist1(T,T1,M),!.
```

REW :

Bernard Silver
Updated: 23 February 1984

```
public          apply_new_rule1/5,
                get_user_rule/8,
                hard_tidy_expr/2,
                simple_rule/7,
                try_use_rule/7.

mode            apply_new_rule1(+,+,?,?,+),
                apply_new_rule2(+,+,+,+,+,+,?,?),
                apply_new_rule3(+,+,+,?,+,?,?),
                apply_new_rule4(+,+,+,+,+,?,?),
                apply_new_rule5(+,+,?,?,+,?,+,?,?),
                choice_tidy(+,?),
                choice_tidy1(+,?),
                examine_template(+,-),
                get_user_rule(?,?,?,?,?,?,?,-,+),
                hard_tidy_expr(+,-),
                hard_tidy_expr(+,+,+),
                simple_rule(?,+,+,?,?,?,-),
                try_use_rule(+,+,?,+,+,+,+,?).


ply_new_rule1(X,Exp,New,Name,Type) :-
      ok_vars(New,Name),  % this is the rule we want or its a worked example
      occ(X,Exp,No1),
      (Type=tl,occ(X,New,No2);Type=sol),
      get_user_rule(Name,X,Template => Rewrite,Cond,No1,No2,Cond1,Type),
      ((nonvar(No2),check_cond(Cond1));var(No2)),
      try_use_rule(X,Exp,New,Template=>Rewrite,Cond,Cond1,No2).

ply_new_rule1(X,Exp,New,Name,Type) :-
      \+ ground(New),
      var(Name),  % Must be careful
      occ(X,Exp,No1),
      get_user_rule(Name,X,Template => Rewrite,Cond,No1,No2,Cond1,Type),
      examine_template(Template,N),
      apply_new_rule2(X,Exp,New,Template=>Rewrite,Cond,N,Cond1,No2).

 One is instantiated
_vars(New,_) :-  ground(New),!.
_vars(_,Y) :- nonvar(Y),!.


 Rule has at least two terms in LHS

ply_new_rule2(X,Exp,New1,Template=>Rewrite,Cond,N,Cond1,F) :-
      N>1,
      apply_new_rule3(X,Exp,New1,Template=>Rewrite,Cond,Cond1,F).

 Rule has only one term on LHS, eg sin(2*x) => 2*sin(x)*cos(x)
ply_new_rule2(X,Exp,New1,Lhs=>Rhs,Cond,_,Cond1,F) :-
      apply_new_rule4(X,Exp,New1,Lhs=>Rhs,Cond,Cond1,F).
```

```
% Basically Collection Code
apply_new_rule3(X,Exp,New1,Template=>Rewrite,Cond,Cond1,F) :-
        mult_occ(X,Exp),
        least_dom(X,Exp),                % Expression is in weak normal form
        applicable(Template,Exp,Rest),
        ((nonvar(F),check_cond(Cond));var(F)),
        newform(Exp,Rewrite,Rest,New),
        check_cond(Cond),
        choice_tidy(New,New1),
        occ(X,New1,F),
        check_cond(Cond1).



 Recursive case of Collection
apply_new_rule3(X,Exp,New1,Template=>Rewrite,Cond,Cond1,F) :-
        mult_occ(X,Exp),
        decomp(Exp,[Fun|Args]),
        corresponding_arguments(Args,Arg,NewArgs,NewArg),
        apply_new_rule3(X,Arg,NewArg,Template=>Rewrite,Cond,Cond1,F),
        recomp(New,[Fun|NewArgs]),
        choice_tidy(New,New1).



 Other case
apply_new_rule4(X,Exp,New,Lhs=>Rhs,Cond,Cond1,F) :-
        match(Exp,Lhs),
        check_cond(Cond),
        choice_tidy(Rhs,New),
        occ(X,New,F),
        check_cond(Cond1).



apply_new_rule4(X,Exp,New,Lhs=>Rhs,Cond,Cond1,F) :-
        setof(Term,subterms(Exp,X,Term),Set),
        apply_new_rule5(Set,X,Lhs,Rhs,Exp,New,Cond,Cond1,F).


apply_new_rule5(Set,X,Lhs,Rhs,Exp,New,C,C1,F) :-
        match_member(Lhs,Set,Term),
        ((ground(F),check_cond(C));\+ground(F)),
        subst(Term=Rhs,Exp,New1),
        check_cond(C),
        choice_tidy(New1,New),
        occ(X,New,F),
        check_cond(C1),
        !.



 What sort of rule is it?
 mine_template(_+_,X) :- !,X=2.
 mine_template(_*_,X) :- !,X=2.
 mine_template(_,X) :- X=1.    % Single term or more complex rule
                               % Collect can't cope with either

 _use_rule(X,Exp=C,New1,Template=C1 =>Rewrite,Cond,Cond1,No2) :-
        match_check(C,C1),
```

```
      functor(Exp,F,_),
      functor(Template,F,_),
      associative(F),
      decomp(Exp,[F|Arg1]),
      decomp(Template,[F|Arg2]),
      length(Arg2,2),
      length(Arg1,N),
      ((N>2,!);N>1),
      pairfrom(Arg1,A,B,Rest),
      recomp(Term,[F,A,B]),
      match_check(Term,Template),
      recomp(Rest1,[F|Rest]),
      New=..[F,Rest1,Rewrite],
      check_cond(Cond),
      choice_tidy(New,New1),
      occ(X,New1,No2),
      check_cond(Cond1).

y_use_rule(X,Exp=C,New1,Template=>Rewrite,Cond,Cond1,No2) :-
      functor(Exp,F,_),
      functor(Template,F,_),
      associative(F),
      decomp(Exp,[F|Arg1]),
      decomp(Template,[F|Arg2]),
      length(Arg2,2),
      length(Arg1,N),
      ((N>2,!);N>1),
      pairfrom(Arg1,A,B,Rest),
      recomp(Term,[F,A,B]),
      match_check(Term,Template),
      recomp(Rest1,[F|Rest]),
      New=..[F,Rest1,Rewrite],
      check_cond(Cond),
      choice_tidy(New=C,New1),
      occ(X,New1,No2),
      check_cond(Cond1).


y_use_rule(X,Exp,New1,Template=>Rewrite,Cond,Cond1,No2) :-
      examine_template(Template,N),
      apply_new_rule2(X,Exp,New1,Template=>Rewrite,Cond,N,Cond1,No2).

   : user_rule of the right type

 Get a tidy type rule, these can loop so only used for tl case
 or if Name is instantiated
:_user_rule(Name,X,Template => Rewrite,Cond,No1,No2,Cond1,Type) :-
      ((nonvar(Name),Type=tl);var(Name)),
      call(simple_rule(Name,X,Template => Rewrite,Cond,No1,No2,Cond1)).

:_user_rule(Name,X,Template => Rewrite,Cond,No1,No2,Cond1,_) :-
      call(user_rule(Name,X,Template => Rewrite,Cond,No1,No2,Cond1)).

ice_tidy(Old,New) :- choice_tidy1(Old,New).

ice_tidy(Old,New) :-
      hard_tidy_expr(Old,Tidy),
      Old \= Tidy,
      choice_tidy1(Tidy,New).
```

```
hoice_tidy1(Old,Mid) :- tidy_expr(Old,New),match_check(Mid,New),!.
hoice_tidy1(Old,Mid) :- tidy(Old,New),match_check(Mid,New),!.

ard_tidy_expr(Exp#Exp1,New) :- !,
        hard_tidy_expr(Exp,Exp2),
        hard_tidy_expr(Exp1,Exp3),
        !,
        tidy_expr(Exp2#Exp3,New),
        !.
ard_tidy_expr(Exp=A,New) :- !,
        hard_tidy_expr(Exp,Exp1),
        simplify(A,A2),
        tidy_expr(A2,A1),
        !,
        tidy_expr(Exp1=A1,New),
        !.

ard_tidy_expr(Exp,Exp) :- (atomic(Exp);number(Exp)),!.

ard_tidy_expr(A+B,New) :- !,
        hard_tidy_expr(A,A1),
        hard_tidy_expr(B,B1),
        tidy_expr(A1+B1,New1),
        simplify(New1,New),
        !.

rd_tidy_expr(A*B,New) :- !,
        hard_tidy_expr(A,A1),
        hard_tidy_expr(B,B1),
        tidy_expr(A1*B1,New1),
        simplify(New1,New),
        !.

 rd_tidy_expr(Exp,New) :-
        functor(Exp,F,N),
        functor(New1,F,N),
        hard_tidy_expr(N,Exp,New1),
        tidy_expr(New1,New).


 rd_tidy_expr(0,_,_) :- !.
 rd tidy_expr(N,Term,New) :-
        arg(N,Term,ArgN),
        hard_tidy_expr(ArgN,ArgN1),
        arg(N,New,ArgN1),
        M is N - 1,
        !,
        hard_tidy_expr(M,Term,New).

  Simplify rules
 mple_rule(sec,_,sec(Y)=>1/cos(Y),[non_zero(cos(Y))],M,M,[]).
 mple_rule(cos,_,cos(Y)=>1/sec(Y),[],M,M,[]).

 mple_rule(cosec,_,cosec(Y)=>1/sin(Y),[non_zero(sin(Y))],M,M,[]).

 mple_rule(sin,_,sin(Y)=>1/cosec(Y),[],M,M,[]).

 mple_rule(cot,_,cot(Y)=>1/tan(Y),[non_zero(tan(Y))],M,M,[]).

 mple_rule(tan,_,tan(Y)=>1/cot(Y),[non_zero(cot(Y))],M,M,[]).
```

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

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 2 August 81
/

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

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

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

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 2 August 81
/

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

```
* OPS :  Operators for LP

                                                Bernard Silver
                                                Updated: 2 November 1983
/

- op(800,xfx,=>).
- op(100,fx,[m,show,remove,enable,disable,writeout,s,r,e,d,w,help]).
```

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

                                                    UTILITY
                                                    Lawrence
                                                    Updated: 2 April 81
/

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


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


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



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

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

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



                        % Succeed if a file exists, otherwise fail

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


                        % Open a file, checking that it exists

)pen(File)
```

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


                        % Open a file and return current file

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



                        % Close file and see old file again

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



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

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

```
/* IMISCE.PL :  Written by Lawrence
                                                    Bernard Silver
                                                    Updated: 6 November 82
*/
:- op(100,fx,l).

  continue.


  \=(X,X) :- !, fail.

  \=(X,Y) :- !.


  clean :- nolog,
           seeing(X),
           see('prolog.log'),
           rename('prolog.log',[]),
           seen,
           see(X),
           log.


  diff(X,X) :- !, fail.

  diff(X,Y) :- !.


 % Default define trace
trace(_,_).

trace(_,_,_).

l(X) :- listing(X).

sb :- save('scra:prolog.bin').


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

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

  not(X) :- !.
```

```
%   File   : EDIT.PL
%   Author : R.A.O'Keefe + Lawrence Byrd
%   Updated: 28 January 1984
%   Purpose: Get from Prolog to TOP and back again.
%   Needs  : append/3 from LISTUT.PL, and latest version of TOP.

%   This module can be compiled or interpreted, whichever you prefer.
%   Luis Jenkins and Bernard Silver added "top" and "top File".

%   Six predicates are defined:
%       edit File       call Top to edit File and return
%       edit            edit the last file mentioned in an edit or redo
%       redo File       call Top to edit File and reconsult the result
%       redo            redo the last file mentioned in an edit or redo
%       top  File       analogous to redo File.
%       top             analogous to redo.
%
%   I have copied Luis Jenkins' idea of making : and . operators so that
%   you can specify a file name without the quotes.

:- op(900, fx, edit).
:- op(900, fx, redo).
:- op(900, fx, top).
:- op(600, xfy, (.)).
   op(800, xfy, (:)).


:- public
        (edit)/0,
        (edit)/1,
        (redo)/0,
        (redo)/1,
        (top)/1,
        (top)/0,
        file_term/2.


:- mode
        edit_file(+),
        file_term(+, -).


edit :-
        'last file'(File), !,
        edit_file(File).
edit :-
        display('! What file?'), ttynl.


edit(File) :-
        file_term(File, FileAtom),
        abolish('last file', 1),
        assert('last file'(FileAtom)),
        edit_file(FileAtom).



redo :-
        'last file'(File), !,
        edit_file(File),
```

```
        reconsult(File).
redo :-
        display('! What file?'), ttynl.


redo(File) :-
        file_term(File, FileAtom),
        abolish('last file', 1),
        assert('last file'(FileAtom)),
        edit_file(FileAtom),
        reconsult(FileAtom).


top File :-
        redo File.


top :-
        redo.


%   file_term takes a file name which may be specified using the : and .
%   operators, and returns an atom.  The lib(_) command should use it as
%   well.  file_term does not test that the result is a well-formed file
    name, nor does it truncate components to 6 letters.  Maybe some code
%   from try_hard_to_see could be adapted.

file_term(Device:FileName.Extension, FileAtom) :- !,
        name(Extension, E),
        name(FileName,  F),
        name(Device,    D),
        append(F, [46|E], FE),
        append(D, [58|FE], DFE),
        name(FileAtom, DFE).
file_term(Device:FileName, FileAtom) :- !,
        name(FileName,  F),
        name(Device,    D),
        append(D, [58|F], DF),
        name(FileAtom, DF).
file_term(FileName.Extension, FileAtom) :- !,
        name(Extension, E),
        name(FileName,  F),
        append(F, [46|E], FE),
        name(FileAtom, FE).
file_term(FileName, FileName) :-
        atom(FileName), !.
file_term(Bogus, _) :-
        display('! Bad file name: '), display(Bogus), ttynl.


%   edit_file has to save and restore Prolog's state, and it has to
%   tell TOP what to do.  There used to be a special hack in TOP to
%   talk to Prolog, but now it uses a system-wide hack: TMP:EDT.
%   A TMP:EDT file contains the following:
%       S <file name> <fn delimiter> [ <program name> <pn delimiter> ]
%   where <fn delimiter> ::= <CR> | <ESC> | }
%   and   <pn delimiter> ::= !
%   What we write is therefore S<filename><ESC>MEC:PROLOG! .  The fact
%   that the editor involved is TOP is, thanks to our use of this Dec-
%   10 convention, immaterial.   To get another editor, say FINE, just
```

```
%    change the editor(_) fact.  The name PROLOG.BIN can *not*  be made
%    different, it is built into Prolog itself.

edit_file(File) :-
        (   save('PROLOG.BIN', 1)
        ;   name(File, FileName),
            append([83|FileName], [27|"MEC:PROLOG!"], Command),
            plsys(tmpcor(tell,edt,Command)),
            editor(Editor),
            plsys(run(Editor, 1))
        ), !,
        see('PROLOG.BIN'),
        rename('PROLOG.BIN', []).


editor('MEC:TOP').
```

```
%   File   : TYPE.PL
%   Author : R.A.O'Keefe
%   Updated: 21 June 1983
%   Purpose: A Tops-10-like "type" command to display files.

%   "type [a,b,c]" will try hard to see a, b, c and will display them
%   on the terminal.  Output redirection will have no effect on it, a
%   more general command can be made by editing this one.  When this
%   is interpreted, you can stop it.  When it is compiled, all you can
%   do is use ^O.  Helper.Pl is needed for try_hard_to_see.

:- public
        (ty)/1,
        (type)/1.

:- mode
        ty(+),
        type(+).

:- op(1150, fx, [ty,type]).


ty(File) :-
        (type File).

(type Var) :-
        var(Var),
        !,
        writef('! variable given as file name\n'),
        fail.
(type [Head|Tail]) :- !,
        (type Head), !,
        (type Tail).
(type File) :-
        seeing(Old),
        try_hard_to_see(File, [eco,mec,util], [pl,hlp,txt,lpt]),
        seeing(New),
        display('File '), display(New), ttynl,
        repeat,
            get0(Ch),
            ttyput(Ch),
            Ch = 26,
        !,
        seen,
        see(Old).
```

```
%   File   : TRYSEE.PL
%   Author : R.A.O'Keefe
%   Updated: 16 December 1983
%   Purpose: Search through several directories/extensions to find a file
%   Needs  : append/3 from Util:ListUtil.Pl

%   try_hard_to_see(FileName, DeviceDefaults, ExtensionDefaults)
%       -- tries all the Extension and Device defaults (varying the
%       -- extensions first) until it succeeds in 'see'ing the file,
%       -- and fails if the file cannot be found.
%   try_hard_to_see(FileName, Devs, Exts, FileFound)
%       -- is like try_hard_to_see/3, but doesn't open the file, it
%       -- just binds FileFound to it.  If no file can be found, it
%       -- just fails.  The other version prints a message.


:- public
        try_hard_to_see/3,
        try_hard_to_see/4.

:- mode
        try_hard_to_see(+, +, +, ?),
        try_hard_to_see(+, +, +),
            expand_file(+, +, +, -),
                parse_file(-, -, -, ?, ?),
                    file_component(-, ?, ?),
                        letter_or_digit(+, -),
                normalise_file_component(+, +, -),
                supply_file_default(+, +, +, -),
                    supply_file_default(+, +, -),
                pack_file_title(+, +, +, -).


try_hard_to_see(Title, DeviceDefaults, ExtensionDefaults, FileFound) :-
        seeing(OldFile),
        nofileerrors,
        (   expand_file(Title, DeviceDefaults, ExtensionDefaults, FullTitle),
            see(FullTitle),
            seeing(FileFound),
            seen
        ;   true
        ),  !,
        see(OldFile),
        fileerrors,
        nonvar(FullTitle).              %   HACK HACK HACK


try_hard_to_see(Title, DeviceDefaults, ExtensionDefaults) :-
        nofileerrors,
        expand_file(Title, DeviceDefaults, ExtensionDefaults, FullTitle),
        see(FullTitle), !,
        fileerrors.
try_hard_to_see(Title, _, _) :-
        fileerrors,
        write('** Can''t see '), writeq(Title), nl, fail.


expand_file(Title, DeviceDefaults, ExtensionDefaults, FullTitle) :-
        atomic(Title),
        name(Title, TitleName), !,
```

```
        expand_file(TitleName, DeviceDefaults, ExtensionDefaults, FullTitle).
expand_file(Title, DeviceDefaults, ExtensionDefaults, FullTitle) :-
        parse_file(Device, FileName, Extension, Title, []),
        normalise_file_component(FileName, 6, TryFileName), !,
        supply_file_default(Device, DeviceDefaults, 6, TryDevice),
        supply_file_default(Extension, ExtensionDefaults, 3, TryExtension),
        pack_file_title(TryDevice, TryFileName, TryExtension, TryTitle),
        name(FullTitle, TryTitle).


        parse_file(Device, FileName, Extension) -->
                (   file_component(Device), ":"
                |   {   Device = ""   }
                ), !,
                file_component(FileName),
                (   ".", file_component(Extension)
                |   {   Extension = ""   }
                ), !.


                file_component([LetDig|Rest]) -->
                        [Char], {   letter_or_digit(Char, LetDig)   }, !,
                        file_component(Rest).
                file_component([]) --> [].

                        letter_or_digit(C, C) :-
                                C >= "0", C =< "9", !.
                        letter_or_digit(C, C) :-
                                C >= "a", C =< "z", !.
                        letter_or_digit(C, D) :-
                                C >= "A", C =< "Z",
                                D is C+("a"-"A").


        normalise_file_component([], _, []) :- !.
        normalise_file_component(Default, Length, TryThis) :-
                atomic(Default),
                name(Default, DefaultName), !,
                normalise_file_component(DefaultName, Length, TryThis).
        normalise_file_component(_, 0, []) :- !.
        normalise_file_component([C|Rest], Length, [LetDig|More]) :-
                letter_or_digit(C, LetDig), !,
                Left is Length-1,
                normalise_file_component(Rest, Left, More).
        normalise_file_component([_|Rest], Length, TryThis) :-
                normalise_file_component(Rest, Length, TryThis).


        supply_file_default(Given, _, Length, TryThis) :-
                normalise_file_component(Given, Length, TryThis).
        supply_file_default([], Defaults, Length, TryThis) :-
                supply_file_default(Defaults, Length, TryThis).

                supply_file_default([Default|_], Length, TryThis) :-
                        normalise_file_component(Default, Length, TryThis).
                supply_file_default([_|Defaults], Length, TryThis) :- !,
                        supply_file_default(Defaults, Length, TryThis).


        pack_file_title([], FileName, Extension, Title) :- !,
```

```
        append(FileName, [46|Extension], Title), !.     % 46 is "."
pack_file_title(Device, FileName, Extension, Title) :-
        pack_file_title([], FileName, Extension, Tail),
        append(Device, [58|Tail], Title), !.            % 58 is ":"
```

/* File: Tidy.Pl Author: R.A.O'Keefe Updated: 18 October 82

This is a new implementation of tidy, written in an attempt to remove some of the deficiencies of the old one. Unfortunately, it has a few of its own. The only completely satisfactory approach seems to be to keep all expressions in bag form all the time.

Tidy has now been split into two parts: tidy_stmt and tidy_expr.

```
    <stmt> ::= <stmt> # <stmt>      %  disjunction
             | <stmt> & <stmt>      %  conjunction
             | <expr> R <expr>      %  equation/inequality
    where R is one of = < > \= >= =<
```

An <expr> is an ordinary algebraic expression. Statements are scanned top-down, and no great effort is expended on them beyond a limited bit of evaluation. Expressions are scanned bottom-up, and are worked hard.

Tidy_stmt works from top down. It doesn't bother putting statements in bag form, although since & (and) and # (or) are both commutative and associative it could well do so. It does however do some flattening of statements: (E1 & E2) & E3 -> E1 & (E2 & E3). This can do no harm. As an experiment, tidy_stmt tries to put constants on the right-hand-sides of equations. E.g. "x+y-3 = 0" -> "x+y = 3". Just how useful this may be remains to be seen. The code for combine_and and combine_or comes almost directly from the original tidy.

The intermediate form makes use of a different representation of bags. A plus (times) bag is stored as +(Tree, Hole, Num) {*(Tree, Hole, Num)}. For example, a+b+c+3 would be stored as

```
    +(      +    , X,  3)
           / \
          +   c
         / \
        +   b
       / \
      X   a
```

```
<expr> ::= <expr> + <expr> | <expr> - <expr> | - <expr>
         | <expr> * <expr> | <expr> / <expr>
         | <expr> ^ <expr> | sqrt(<expr>)
         | <special function>(<expr>,...)
         | <atom>            -- algebraic variable
         | <variable>        -- treated like an atom
         | <number>          -- including rational numbers

<tidy expr> ::= {like <expr>, but only the first column. Also,
        numeric fragments are combined where possible, and sums
        and products are flattened.}

<baggy expr> ::= +(Tree, Hole, Num)
               | *(Tree, Hole, Num)
               | <tidy expr> ^ <baggy expr>
               | <tidy expr>
```

BUG: if the exponent of a term eventually simplifies to 1, the base emerges as a <tidy expr>, rather than a <baggy expr>. Hence some simplifications will be missed. E.g. "(1+x)^(-1)^(-1) + -1" will end up as "(1+x) + -1" rather than as "x". There appears to be no easy way around this problem, though keeping the base as a <baggy expr> may yet prove to be feasible. In any case, the new tidy only has this problem with exponents, which are generally fairly simple.

```
    Tidy requires simple/1 and copy_ground/3 from STRUCT.PL.
*/
```

```
:- public
        tidy/2,                         %  general interface
        tidy_withvars/2,                %  same as tidy_expr
        tidy_expr/2,                    %  tidy an expression
        tidy_stmt/2.                    %  tidy a statement.


:- mode
        bag_to_tidy(+,-),               %  F(T,H,N) -> T'
        bag_to_tidy(+,+,+),
        combine_and(+,+,-),             %  X,Y -> X&Y
        combine_bags(+,-),              %  apply op to baggy arguments
        combine_or(+,+,-),              %  X,Y -> X#Y
        combine_power(+,+,-),           %  X,Y -> X^Y
        combine_plus(+,+,-),            %  X,Y -> X+Y
        combine_rel(+,+,-,-),           %  X(R)Y -> X'(R)Y'
        combine_times(+,+,-),           %  X,Y -> X*Y
        expr_to_bag(+,-),               %  <expr> -> <baggy expr>
        expr_to_bag(+,+,+,+,-),         %  map down args of <expr>
        multiply_exp(+,+,-),            %  X,N -> N*X
        multiply_out(+,+,-),            %  N,X -> N*X
        multiply_out(+,-,+,-),          %  +(T,H),N -> +(T*N,H)
        number_check(+,+,-),            %  maintain number-p accum
        power_out(+,+,-),
        power_out(+,-,+,-),             %  *(T,H),N -> *(T^N,H)
        relop_tidy(-,+,+,+),            %  R,X,Y -> X(R)Y or true/false
        tidy_expr(+,-),                 %  tidy expression
        tidy_stmt(+,-).                 %  tidy statement


tidy(Old, New) :-
        tidy_stmt(Old, Mid), !, New = Mid.      %  which now tries tidy_expr
tidy(Old, Old) :-
        write('** failed: '), write(tidy(Old, '_')), nl.


tidy_withvars(Old, New) :-
        copy_ground(Old, Ground, Subst),
        tidy(Ground, Tidier),
        subst(Subst, Tidier, Mid), !,
        New = Mid.
```

```
tidy_stmt(Var, _) :-                    %  don't do anything with variables
        var(Var), !, fail.
tidy_stmt(OldOne # OldTwo, New) :- !,
        tidy_stmt(OldOne, MidOne),
        tidy_stmt(OldTwo, MidTwo), !,
        combine_or(MidOne, MidTwo, New).
tidy_stmt(OldOne & OldTwo, New) :- !,
        tidy_stmt(OldOne, MidOne),
        tidy_stmt(OldTwo, MidTwo), !,
        combine_and(MidOne, MidTwo, New).
tidy_stmt(Equation, New) :-
        tidy_relop(Equation, Relation, OldLhs, OldRhs),
        expr_to_bag(OldLhs, MidLhs),
        expr_to_bag(OldRhs, MidRhs),
        combine_rel(MidLhs, MidRhs, NewLhs, NewRhs), !,
        relop_tidy(New, Relation, NewLhs, NewRhs).
tidy_stmt(Old, New) :-
        tidy_expr(Old, New).


combine_or(true, Y, true) :- !.         %  zero element
combine_or(false, Y, Y) :- !.           %  unit element
combine_or(X, true, true) :- !.         %  zero element
combine_or(X, false, X) :- !.           %  unit element
combine_or(X, X, X) :- !.               %  merging identical elements
combine_or(W#X, Y, W#(X#Y)) :- !.       %  change association
combine_or(X, Y, X # Y).                %  general case

combine_and(false, Y, false) :- !.      %  zero element
combine_and(true, Y, Y) :- !.           %  unit element
combine_and(X, false, false) :- !.      %  zero element
combine_and(X, true, X) :- !.           %  unit element
combine_and(X, X, X) :- !.              %  merging identical elements
combine_and(W&X, Y, W&(X&Y)) :- !.      %  change association
combine_and(X, Y, X & Y).               %  general case


tidy_relop(X = Y,   =, X, Y) :- !.
tidy_relop(X < Y,   <, X, Y) :- !.
tidy_relop(X > Y,   >, X, Y) :- !.
tidy_relop(X =< Y, =<, X, Y) :- !.
tidy_relop(X >= Y, >=, X, Y) :- !.
tidy_relop(X \= Y, \=, X, Y) :- !.


relop_tidy(Value, Relation, Lhs, Rhs) :-
        number(Lhs), number(Rhs),
        tidy_relop(Goal, Relation, Lhs, Rhs), !,
        eval(Goal, Value).
relop_tidy(Goal, Relation, Lhs, Rhs) :-
        tidy_relop(Goal, Relation, Lhs, Rhs).


combine_rel(+(T1, H1, N1), +(T2, H2, N2), Lhs, Rhs) :- !,
        bag_to_tidy(+(T1, H1, 0), Lhs),
        eval(N2-N1, N3),
        bag_to_tidy(+(T2, H2, N3), Rhs).
combine_rel(+(T1, H1, N1), N2, Lhs, N3) :-
        number(N2), !,
        eval(N2-N1, N3),
```

```
        bag_to_tidy(+(T1, H1, 0), Lhs).
combine_rel(*(T1, H1, N1), *(T2, H2, N2), Lhs,  Rhs) :-
        eval(N1 > 0), !,
        bag_to_tidy(*(T1, H1, 1), Lhs),
        eval(N2/N1, N3),
        bag_to_tidy(*(T2, H2, N3), Rhs).
combine_rel(*(T1, H1, N1), N2, Lhs, N3) :-
        number(N2),
        eval(N1 > 0), !,
        eval(N2/N1, N3),
        bag_to_tidy(*(T1, H1, 1), Lhs).
combine_rel(E1, E2, Lhs, Rhs) :-
        bag_to_tidy(E1, Lhs),
        bag_to_tidy(E2, Rhs).
```

```
tidy_expr(Old, New) :-
        expr_to_bag(Old, Mid), !,
        bag_to_tidy(Mid, New).

expr_to_bag(Var, _) :-                  %  do nothing with variables
        var(Var), !, fail.
expr_to_bag(Old, Old) :-
        simple(Old), !.
expr_to_bag(Old, New) :-
        functor(Old, F, N),
        functor(Mid, F, N),
        expr_to_bag(N, Old, Mid, yes, New).

        expr_to_bag(0, Old, Mid, yes, New) :- !,
                eval(Mid, New).
        expr_to_bag(0, Old, Mid, no,  New) :-
                combine_bags(Mid, New).
        expr_to_bag(N, Old, Mid, EvalP, New) :-
                arg(N, Old, OldN),
                expr_to_bag(OldN, MidN),
                arg(N, Mid, MidN),
                number_check(MidN, EvalP, EvalQ),
                M is N-1, !,
                expr_to_bag(M, Old, Mid, EvalQ, New).

                number_check(N, EvalP, EvalP) :-
                        number(N), !.
                number_check(_, _, no). %  not a number


combine_bags(X+Y, New) :- !,
        combine_plus(X, Y, New).
combine_bags(X-Y, New) :-
        multiply_out(-1, Y, Z), !,
        combine_plus(X, Z, New).
combine_bags(-Y, New) :- !,
        multiply_out(-1, Y, New).
combine_bags(X*Y, New) :- !,
        combine_times(X, Y, New).
combine_bags(X/Y, New) :- !,
        power_out(Y, -1, Z),
        combine_times(X, Z, New).
combine_bags(X^Y, New) :- !,
        combine_power(X, Y, New).
combine_bags(Old, New) :-
        functor(Old, F, N),
        functor(Mid, F, N),
        bag_to_tidy(N, Old, Mid),
        (   simplify_axiom(Mid, New)
        |   New = Mid
        ), !.


bag_to_tidy(0, Old, Mid) :- !.
bag_to_tidy(N, Old, Mid) :-
        arg(N, Old, OldN),
        bag_to_tidy(OldN, MidN),
        arg(N, Mid, MidN),
        M is N-1, !,
        bag_to_tidy(M, Old, Mid).
```

```
bag_to_tidy(+(T+R, R, 0), T) :- !.
bag_to_tidy(+( T , N, N), T) :- !.
bag_to_tidy(*( T,  H, 0), 0) :- !.
bag_to_tidy(*(T*R, R, 1), T) :- !.
bag_to_tidy(*( T , N, N), T) :- !.
bag_to_tidy(B^0,          1) :- !.              %  B^0 = 1
bag_to_tidy(0^X,          0) :- !.              %  0^X = 0
bag_to_tidy(1^X,          1) :- !.              %  1^X = 1
bag_to_tidy(B^1,          B) :- !.              %  B^1 = B (B already <tidy>)
bag_to_tidy(M^ *(T*R,R,N), B^T) :-              %  M^(N*X) = (M^N)^X
        number(M),
        power(M, N, B), !.
bag_to_tidy(B^X,           B^T) :- !,           %  B^X, where X is <baggy>
        bag_to_tidy(X, T).
        bag_to_tidy(*(T, H, N), E).
bag_to_tidy(Old, Old).


combine_plus(+(T1, H1, N1), +(T2, T1, N2), +(T2, H1, N3)) :- !,
        add(N1, N2, N3).
combine_plus(+(T1, H1, N1), N2, +(T1, H1, N3)) :-
        number(N2), !,
        add(N1, N2, N3).
combine_plus(+(T1, H1, N1), E2, +(T1+E4, H1, N1)) :- !,
        bag_to_tidy(E2, E4).
combine_plus(0, E2, E2) :- !.
combine_plus(N1, +(T2, H2, N2), +(T2, H2, N3)) :-
        number(N1), !,
        add(N1, N2, N3).
combine_plus(E1, +(T2, H2, N2), +(T2+E3, H2, N2)) :- !,
        bag_to_tidy(E1, E3).
combine_plus(E1, 0, E1) :- !.
combine_plus(E1, N2, +(H+E3, H, N2)) :-
        number(N2), !,
        bag_to_tidy(E1, E3).
combine_plus(N1, E2, +(H+E4, H, N1)) :-
        number(N1), !,
        bag_to_tidy(E2, E4).
combine_plus(E1, E2, +((H+E3)+E4, H, 0)) :-
        bag_to_tidy(E1, E3),
        bag_to_tidy(E2, E4).


combine_times(*(T1, H1, N1), *(T2, T1, N2), *(T2, H1, N3)) :- !,
        multiply(N1, N2, N3).
combine_times(N1, E2, Ans) :-
        number(N1), !,
        multiply_out(N1, E2, Ans).
combine_times(E1, N2, Ans) :-
        number(N2), !,
        multiply_out(N2, E1, Ans).
combine_times(*(T1, H1, N1), E2, *(T1*E4, H1, N1)) :- !,
        bag_to_tidy(E2, E4).
combine_times(E1, *(T2, H2, N2), *(T2*E3, H2, N2)) :- !,
        bag_to_tidy(E1, E3).
combine_times(E1, E2, *((H*E3)*E4, H, 1)) :-
        bag_to_tidy(E1, E3),
        bag_to_tidy(E2, E4).
```

```
multiply_out(0, Old, 0) :- !.
multiply_out(1, Old, Old) :- !.
/*  The next clause has been replaced by the two following clauses for the
    sake of Press and attraction.  This clause is correct, but alas, when
    attraction moves a number out (N*X+N*X)->N*(X+X) tidy moves it back in.

multiply_out(N, +(OldTree, Hole, OldNum), +(NewTree, Hole, NewNum)) :-
        multiply(N, OldNum, NewNum), !,
        multiply_out(OldTree, Hole, N, NewTree).
*/
multiply_out(-1, +(OldTree, Hole, OldNum), +(NewTree, Hole, NewNum)) :-
        multiply(-1, OldNum, NewNum), !,
        multiply_out(OldTree, Hole, -1, NewTree).
multiply_out(N, +(OldTree, Hole, OldNum), +(NewHole+N*Exp, NewHole, NewNum)) :-
        multiply(N, OldNum, NewNum), !,
        bag_to_tidy(+(OldTree, Hole, 0), Exp).
multiply_out(N, *(OldTree, Hole, OldNum), *(OldTree, Hole, NewNum)) :- !,
        multiply(N, OldNum, NewNum).
multiply_out(N, M, P) :-
        number(M), !,
        multiply(N, M, P).
multiply_out(N, Old, *(Hole*Exp, Hole, N)) :- !,
        bag_to_tidy(Old, Exp).

multiply_out(Bottom, Hole, N, Bottom) :-
        Bottom == Hole.
multiply_out(OldX + OldY, Hole, N, NewX + NewY) :-
        multiply_exp(OldY, N, NewY), !,
        multiply_out(OldX, Hole, N, NewX).

multiply_exp(OldX * OldY, N, NewX * OldY) :- !,
        multiply_exp(OldX, N, NewX).
multiply_exp(OldX + OldY, N, NewX + NewY) :-
        multiply_exp(OldY, N, NewY), !,
        multiply_exp(OldX, N, NewX).
multiply_exp(OldNum, N, NewNum) :-
        number(OldNum), !,
        multiply(N, OldNum, NewNum).
multiply_exp(Old, N, N*Old).


combine_power(B^E1, E2, B^E3) :- !,
        combine_times(E1, E2, E3).
combine_power(B, N2, Ans) :-
        number(N2), !,
        power_out(B, N2, Ans).
combine_power(E1, E2, E3^E4) :-
        bag_to_tidy(E1, E3), !,
        bag_to_tidy(E2, E4).


power_out(B, 0, 1) :- !.
power_out(B, 1, B) :- !.
power_out(B^E1, P, B^E2) :- !,
        multiply_out(P, E1, E2).
power_out(*(H1*T1, H1, 1), P, Ans) :-
        var(H1), !,
        power_out(T1, P, Ans).
power_out(*(T1, H1, N1), P, *(T2, H1, N2)) :-
```

```
        power(N1, P, N2), !,
        power_out(T1, H1, P, T2).
power_out(*(T1, H2*N1, N1), P, *(T2, H2, 1)) :- !,
        power_out(T1, H2, P, T2).
power_out(+(H0+T1, H1, 0), P, Ans) :-
        H0 == H1 /*DRAT*/, !,
        power_out(T1, P, Ans).
power_out(N, P, M) :-
        number(N),
        power(N, P, M), !.
power_out(B, P, E^P) :-
        bag_to_tidy(B, E).


power_out(Bottom, Hole, Num, Bottom) :-
        Bottom == Hole, !.
power_out(OldX * (OldB^OldP), Hole, Num, NewX * NewB) :-
        multiply_exp(OldP, Num, NewP),
        (   NewP = 1, NewB = OldB  |  NewB = OldB^NewP   ), !,
        power_out(OldX, Hole, Num, NewX).
power_out(OldX * OldY, Hole, Num, NewX * (OldY^Num)) :- !,
        power_out(OldX, Hole, Num, NewX).
```

File   : WRITEF.PL
Author : Lawrence + Richard
Updated: 10 September 1983
Purpose: Formatted write routine (and support)

* Compile this module.
* WRITEF requires no other modules.

FIXES

(11 May 81) LB

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

(8 September 82) ROK

Added a clause to writef to allow the format to be a string.
Added the format items nL, nR, nC for atoms/numbers/strings.
Added the %s format code. Made getxxx things grammar rules.

(? September 82) ROK

Fixed long-standing bug in ttyprint: 'tell' was 'see' !!

(22 June 83) ROK

Added fwrite/2 and fwritef/3 by analogy to fprintf.
They are very often useful.
Also added the %i (indirect) format item, and the \e escape (generates ESC) for talking to terminals.

(10 September 1983) ROK

Added the %g (agglutinated) format item. The idea of this is that you can have a term like +(A,B,C,D) written as A + B + C + D. ASA is the only program to use it so far, but since such records are quite a bit more compact than lists, it sems like a good idea.

Added the %x (ignore) format item, so that you can compute a format : get_format(Key,Fmt), writef(Fmt, [List]) where some of the variations don't want to display all the arguments.

Added the \b (backspace) and \f (formfeed) escapes. This wants to be done when strings are read, and wants to be exactly the same as C. Maybe in the next Prolog system...

If the list argument is neither [] nor a list, it will be turned into a list of one element. I keep forgetting to do this in my source code, so writef might as well do it fo me.

Changed uses of & and # as operators to uses as function symbols, so this file can be loaded when you're not using those operators. Also made the logical stuff treat , as conjunction and ; (same as |) as disjunction. Renamed all prexpr's subroutines to prexpr,to remove possible name conflicts. That was a bit dubious, but I also renamed special->wf_char and action->wf_act; those two were *bound* to get in someone's way sooner or later, probably mine.

(15 September 1983) ROK

Added the %v hack, which calls numbervars on the items in the list. This is to make variables come out as letters, which I think looks pretty, and it is harmless, because writef fails anyway!

```
- public
        prconj/1,               %   print conjunction
        prexpr/1,               %   print logical expression
        prlist/1,               %   print list, one per line
        ttyprint/1,             %   print on terminal
        fwritef/2,
        fwritef/3,              %   formatted write to file
        writef/1,
        writef/2.               %   formatted write

- mode
        ttyprint(?),
        prlist(?),
        prconj(?),
        prexpr(?),
                prexpr(+,+,-,?,?),
                prexpr(+,-,-,-),
                prexpr(+,+),
        fwritef(+,+),
        fwritef(+,+,+),
        writef(+),
        writef(+,+),
                wf_act(+,+,-),
                getcode(-,+,-),
                getdigits(+,-,+,-),
                getpad(+,-),
                getpad(+,+,-),
                getpad(-,-,+,-),
                padout(+),
                padout(+,+,+),
                padout(+,+,+,-,-),
                praggl(+,+,+,+),
                wf_char(+,-),
                writelots(?,+),
                writef_nonlist(+,-),
                writefs(+,+).



                        % Print (therefore use pretty printing) onto
                        %  the terminal (no-one uses this routine).

typrint(X) :-           % fwritef(user, '%p', [X])
        telling(Old),
        tell(user),
        print(X),
        tell(Old).
```

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

prlist([]) :- !.
prlist([Head|Tail]) :-
        tab(4), print(Head), nl,
        prlist(Tail).



                        % Print a conjunction, one element per line

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



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

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


prexpr(Term, Nin, Nout, Elements, Z) :-
        prexpr(Term, Conn, A, B), !,
        put("("), prexpr(A, Nin, Nmid, Elements, Rest),
        put(" "), put(Conn),
        put(" "), prexpr(B, Nmid, Nout, Rest, Z),
        put(")").
    prexpr(Term, Nin, Nout, [Term|Z], Z) :-
        Nout is Nin+1,
        put("X"), write(Nin).


        prexpr(&(A,B),  38, A, B).      %  38 is "&"
        prexpr(#(A,B),  35, A, B).      %  35 is "#"
        prexpr((A,B),   38, A, B).      %  38 is "&"
        prexpr((A;B),  124, A, B).      % 124 is "|"


prexpr([Head|Tail], M) :-
        write('    X'), write(M), write(' =  '),
        print(Head), nl,
        N is M+1, !,
        prexpr(Tail, N).
prexpr([], _).
```

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

fwritef(File, Format) :-
        fwritef(File, Format, []).

fwritef(File, Format, List) :-
        telling(Old),
        tell(File),
        writef(Format, List),
        tell(Old).

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


writef(Format, Item) :-
        writef_nonlist(Item, List), !,
        writef(Format, List).
writef([F|String], List) :-
        writefs([F|String], List),
        fail.
   tef(Format, List) :-
        atom(Format),
        name(Format, Fstring),
        writefs(Fstring, List),
        fail.
writef(_, _).


writef_nonlist([], _) :- !, fail.
writef_nonlist([_|_], _) :- !, fail.
writef_nonlist(Item, [Item]).



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

writefs([], List).

   :efs([37,A|Rest], List) :-           %   %<action>
        wf_act(A, List, More), !,
        writefs(Rest, More).

writefs([37,D|Rest], [Head|Tail]) :-    %   %<columns><just>
        "0" =< D, D =< "9",
        getpad(Size, Just, [D|Rest], More),
        padout(Head, Size, Just), !,
        writefs(More, Tail).

writefs([92,C|Rest], List) :-           %   \<special>
        wf_char(C, Char),
        put(Char), !,
        writefs(Rest, List).

writefs([92|Rest], List) :-             %   \<character code in decimal>
        getcode(Char, Rest, More),
```

```
        put(Char), !,
        writefs(More, List).

writefs([Char|Rest], List) :-           %   <ordinary character>
        put(Char), !,
        writefs(Rest, List).



wf_act( 99, [Head|Tail], Tail) :-       %   Conjunction
        nl, !, prconj(Head).

wf_act(100, [Head|Tail], Tail) :-       %   Display
        display(Head).

wf_act(101, [Head|Tail], Tail) :-       %   Expression
        nl, !, prexpr(Head).

wf_act(102, List, List) :-              %   Flush
        ttyflush.

wf_act(103, [Head|Tail], Tail) :-       %   aGglutinated
        functor(Head, F, N),
        praggl(1, N, F, Head).

wf_act(105, [Format,List|Tail], Tail):- %   Indirect
        writef(Format, List).

wf_act(108, [Head|Tail], Tail) :-       %   List
        nl, !, prlist(Head).

wf_act(110, [Char|Tail], Tail) :-       %   iNteger (character)
        put(Char).

wf_act(112,  [Head|Tail], Tail) :-      %   Print
        print(Head).

wf_act(113, [Head|Tail], Tail) :-       %   Quoted
        writeq(Head).

wf_act(114, [Thing,Times|Tail],Tail) :- %   Repeatedly
        writelots(Times, Thing).

   act(115, [Head|Tail], Tail) :-       %   String
        padout(Head).

wf_act(116, [Head|Tail], Tail) :-       %   Term
        print(Head).

wf_act(118, List, List) :-              %   numberVars
        numbervars(List, 0, _).

wf_act(119, [Head|Tail], Tail) :-       %   Write
        write(Head).

wf_act(120, [_|Tail], Tail).            %   X (skip)
```

```
if_char( 37, 37).                   %  %
if_char( 92, 92).                   %  \
if_char( 98,  8).                   %  Backspace
if_char(101, 27).                   %  Escape
if_char(102, 12).                   %  Formfeed
if_char(108, 10).                   %  Linefeed
if_char(110, 31).                   %  Newline
if_char(114, 13).                   %  Return
if_char(116,  9).                   %  Tab


getcode(Char) -->
        getdigits(3, Digits), !,
        {   Digits \== [], name(Char, Digits),  Char < 128    }.

getdigits(Limit, [Digit|Digits]) -->
        {   Limit > 0   },
        [Digit],        {   "0" =< Digit, Digit =< "9"    },
        {   Fewer is Limit-1   }, !,
        getdigits(Fewer, Digits).
getdigits(_, []) --> [].


    :elots(N, T) :-
        N > 0,
        write(T),
        M is N-1, !,
        writelots(M, T).
writelots(_, _).


%   praggl(ArgNo, Arity, Func, Term)
%   prints the arguments of the term one after the other, starting with
%   argument ArgNo.  Arguments are separated by " Func ".  This is meant
%   mainly for ASA, but should be generally useful.

praggl(N, N, _, Term) :- !,
        arg(N, Term, Arg),
        print(Arg).
praggl(L, N, F, Term) :-
        arg(L, Term, Arg),
        print(Arg),
        put(32), write(F), put(32),
        M is L+1, !,
        praggl(M, N, F, Term).


/*  The new formats are %nC, %nL, and %nR for centered, left, and right
    justified output of atoms, integers, and strings.  This is meant to
    simplify the production of tabular output when it is appropriate.
    At least one space will always precede/follow the item written.
*/

getpad(Size, Just) -->
        getdigits(3, Digits),   {   name(Size, Digits)   },
        [Char],                 {   getpad(Char, Just)   }.

        getpad(114, r).         %  right justified
        getpad(108, l).         %  left justified
```

```
        getpad( 99, c).         %  centered
        getpad( 82, r).         %  right justified
        getpad( 76, l).         %  left justified
        getpad( 67, c).         %  centered


                                %   padout(A,S,J) writes the item A in a
                                %   field of S or more characters, justified.

padout(Atom, Size, Just) :-
        atomic(Atom),
        name(Atom, Name), !,
        padout(Name, Size, Just).
padout(String, Size, Just) :-
        length(String, Length),
        padout(Just, Size, Length, Left, Right),
        tab(Left),
        padout(String),
        tab(Right).

                                %   padout(Just,Size,Length,Left,Right)
                                %   calculates the number of spaces to put
                                %   on the Left and Right of an item needing
                                %   Length characters in a field of Size.

  ?ut(l, Size, Length, 0, Right) :-
        Excess is Size-Length, !,
        getpad(Excess, 1, Right).
padout(r, Size, Length, Left, 0) :-
        Excess is Size-Length, !,
        getpad(Excess, 1, Left).
padout(c, Size, Length, Left, Right) :-
        Prefix is (Size-Length)/2,
        getpad(Prefix, 1, Left),
        Remainder is (Size-Length)-Left, !,
        getpad(Remainder, 1, Right).


                                %   getpad(A,B,Max) returns the maximum.

getpad(A, B, A) :- A >= B, !.
getpad(A, B, B).


                                %   padout(Str) writes a string.

padout([Head|Tail]) :-
        put(Head), !,
        padout(Tail).
padout([]).
```

```
/* ROUTIN.PL :  Written by Lawrence & Richard

                                                Bernard Silver
                                                Updated: 17 February 1984

*/


        %%%  Compile this module

 /* EXPORT */

  :- public
            append/3,
            apply/2,
            concat/3,
            gensym/2,
            last/2,
            listtoset/2,
            maplist/3,
            member/2,
            memberchk/2,
            nmember/3,
            occurs_in/2,
            pairfrom/4,
            perm/2,
            perm2/4,
            rev/2,
            select/3,
            some/2,
            union/3.


  /* MODES */

:- mode append(?,?,?),
        apply(+,+),
        concat(+,+,?),
        gensym(+,?),
        last(?,?),
        listtoset(?,?),
        maplist(+,?,?),
        member(?,?),
        memberchk(?,?),
        nmember(?,+,?),
        occurs_in(+, +),
        occurs_in(+, +, +),
        pairfrom(+,?,?,?),
        perm(?,?),
        perm2(?,?,?,?),
        rev(?,?),
        revconc(?,+,?),
        select(?,?,?),
        some(+,?),
        union(?,?,?).


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

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


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

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


                        % X is the N'th member of List

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

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


union([],Ys,Ys).

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

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

append([],L,L).

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

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

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


listtoset([],[]).

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

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


perm([],[]).

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


perm2(X,Y,X,Y).

perm2(X,Y,Y,X).
```

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


revconc([],L,L).

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



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

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


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

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

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

concat(N1,N2,N3)
      :- name(N1,Ls1),
         name(N2,Ls2),
         append(Ls1,Ls2,Ls3),
         name(N3,Ls3).



gensym(Prefix,V)
      :- var(V),
         atom(Prefix),
         flag(gensym(Prefix),N,N),
         N2 is N + 1,
         flag(gensym(Prefix),_,N2),
         concat(Prefix,N2,V),
         !.

apply(P,Eargs)
      :- ( atom(P),
           NP =.. [P|Eargs]   ;   P =.. [Pred|Oargs],
                                  append(Oargs,Eargs,Nargs),
                                  NP =.. [Pred|Nargs]
         ),
         !,
         NP.


maplist(P,[],[]) :- !.

maplist(P,[X|Xs],[Y|Ys])
      :- !,
         apply(P,[X,Y]),
         maplist(P,Xs,Ys).


some(P,[]) :- !, fail.
```

```
  some(P,[X|Xs]) :- apply(P,[X]).

  some(P,[X|Xs]) :- !, some(P,Xs).

occurs_in(Var, Term) :-
        var(Term),
        !,
        Var == Term.
occurs_in(Var, Term) :-
        functor(Term, _, N),
        occurs_in(N, Var, Term).

occurs_in(N, Var, Term) :-
        arg(N, Term, Arg),
        occurs_in(Var, Arg),
        !.
occurs_in(N, Var, Term) :-
        N > 1,
        M is N-1,
        occurs_in(M, Var, Term).
```

```
/*      STRUCT.PL                       General term hacking
        R.A.O'Keefe                     Updated: 1 June 1983

    These routines view a term as a data-structure.  In particular,
they handle Prolog variables in the terms as objects.  This is not
entirely satisfactory.  A proper separations of levels is needed.

*/

:- public
        copy_ground/3,                  %  Term -> GroundCopy,Substitution
        occ/3,                          %  SubTerm,Term -> Occurrences
        simple/1,                       %  simple -vs- structured term check
        subst/3,                        %  Substitution,Term -> ModifiedTerm
        variables/2.                    %  Term -> ListOfVariables


/*  simple(Term) is a generalisation of atomic(Term) which recognises LONG
    numbers as simple objects.  The point of it is to avoid scanning the
    sub-structure of things which conceptually have no sub-structure.  NB
    functor/3 will succeed on an atom!  Should simple accept variables too?
*/


simple(Term) :-
        (   atomic(Term)                %  fast check for integers & atoms
        |   number(Term)                %  other kinds of number (rationals)
        ),  !.                          %  variables are not simple
```

```
%   subst(Substitution, Term, Result) applies a substitution, where
%   <substitution> ::= <OldTerm> = <NewTerm>
%                    |  <Substitution> & <Substitution>
%                    |  <Substitution> # <Substitution>
%   The last two possibilities only make sense when the input Term is
%   an equation, and the substitution is a set of solutions.  The
%   "conjunction" of substitutions really refers to back-substitution,
%   and the order in which the substitutions are done may be crucial.
%   If the substitution is ill-formed, and only then, subst will fail.

:- mode
        subst(+,+,-),           %  Subst,Term -> NewTerm
        subst(+,+,+,-),         %  Lhs,Rhs,Term -> NewTerm
        subst(+,+,+,+,+).       %  ArgNo,Lhs,Rhs,OldTerm, NewTerm


subst(Subst1 & Subst2, Old, New) :-
        subst(Subst1, Old, Mid), !,
        subst(Subst2, Mid, New).
subst(Subst1 # Subst2, Old, New1 # New2) :-
        subst(Subst1, Old, New1), !,
        subst(Subst2, Old, New2).
subst(Lhs = Rhs, Old, New) :- !,
        subst(Lhs, Rhs, Old, New).
   ubst(true, Old, Old).


        subst(Lhs, Rhs, Old, Rhs) :-            %   apply substitution
                Old == Lhs, !.
        subst(Lhs, Rhs, Old, Old) :-            %   copy unchanged
                (   var(Old)
                |   atomic(Old)
        %       |   number(Old)
                ), !.
        subst(Lhs, Rhs, Old, New) :-            %   apply to arguments
                functor(Old, Functor, Arity),
                functor(New, Functor, Arity),
                subst(Arity, Lhs, Rhs, Old, New).


                subst(0, Lhs, Rhs, Old, New) :- !.
                subst(N, Lhs, Rhs, Old, New) :-
                        arg(N, Old, OldArg),
                        subst(Lhs, Rhs, OldArg, NewArg),
                        arg(N, New, NewArg),
                        M is N-1, !,
                        subst(M, Lhs, Rhs, Old, New).
```

```
%   occ(Subterm, Term, Times) counts the number of times that the subterm
%   occurs in the term.  It requires the subterm to be ground.  We have to
%   introduce occ/4, because occ's last argument may already be instantiated.
%   It is useful to do so, because we can use accumulator arguments to make
%   occ/4 and occ/5 tail-recursive.  NB if you merely want to check whether
%   SubTerm occurs in Term or not, it is possible to do better than this.
%   See Util:Occur.Pl .

:- mode
        occ(+,+,?),                     %  SubTerm,Term -> Occurrences
        occ(+,+,+,-),                   %  SubTerm,Term,SoFar -> Total
        occ(+,+,+,+,-).                 %  ArgNo,SubTerm,Term,SoFar -> Total


occ(SubTerm, Term, Occurrences) :-
        occ(SubTerm, Term, 0, Times), !,
        Occurrences = Times.

        occ(SubTerm, Term, SoFar, Total) :-
                Term == SubTerm, !,
                Total is SoFar+1.
        occ(SubTerm, Term, Total, Total) :-
                (   var(Term)
                |   atomic(Term)
        %       |   number(Term)
                ), !.
        occ(SubTerm, Term, SoFar, Total) :-
                functor(Term, Functor, Arity), !,
                occ(Arity, SubTerm, Term, SoFar, Total).

                occ(0, SubTerm, Term, Total, Total) :- !.
                occ(N, SubTerm, Term, SoFar, Total) :-
                        arg(N, Term, Arg),
                        occ(SubTerm, Arg, SoFar, Accum),
                        M is N-1, !,
                        occ(M, SubTerm, Term, Accum, Total).
```

```
%   The previous two predicates operate on ground arguments, and have some
%   pretence of being logical (though at the next level up).  The next one
%   is thoroughly non-logical.  Given a Term,
%       variables(Term, VarList)
%   returns a list whose elements are the variables occuring in Term, each
%   appearing exactly once in the list.  var_member_check(L, V) checks
%   that the variable V is *not* a member of the list L.  The original
%   version of variables/2 had its second argument flagged as "?", but this
%   is actually no use, because the order of elements in the list is not
%   specified, and may change from implementation to implementation.
%   The only application of this routine I have seen is in Lawrence's code
%   for tidy_withvars.  The new version of tidy uses copy_ground (next page).
%   If that is the only use, this routine could be dropped.

:- mode
        variables(+,-),                 %  Term -> VarList
        variables(+,+,-),               %  Term,Accum -> VarList
        variables(+,+,+,-).             %  Arity,Term,Accum -> VarList
        var_member_check(+,-).          %  VarList,Variable ?


variables(Term, VarList) :-
        variables(Term, [], VarList).

        variables(Term, VarList, [Term|VarList]) :-
                var(Term),
                var_member_check(VarList, Term), !.
        variables(Term, VarList, VarList) :-
                (   var(Term)
                |   atomic(Term)
        %       |   number(Term)
                ), !.
        variables(Term, SoFar, VarList) :-
                functor(Term, Functor, Arity), !,
                variables(Arity, Term, SoFar, VarList).

                variables(0, Term, VarList, VarList) :- !.
                variables(N, Term, SoFar, VarList) :-
                        arg(N, Term, Arg),
                        variables(Arg, SoFar, Accum),
                        M is N-1, !,
                        variables(M, Term, Accum, VarList).

                var_member_check([],    Var).
                var_member_check([Head|Tail], Var) :-
                        Var \== Head, !,
                        var_member_check(Tail, Var).
```

```
*   In order to handle statements and expressions which contain variables,
    we have to create a copy of the given data-structure with variables
    replaced by ground terms of some sort, do an ordinary tidy, then put
    the variables back.  Since we can use subst/3 to do this last step, a
    natural choice of working structure in the first step is a substitution
        $VAR(k) = Vk & ... & $VAR(0) = V0 & 9 = 9.
    The rest is straight-forward.  The cost of building the copy is o(E*V)
    where E is the size of the original expression and V is the number of
    variables it contains.  The final substitution is the same order of cost.
    For what it's worth, copy_ground(X,Y,_) & numbervars(X,0,_) => X == Y.
/

- mode
        copy_ground(+,-,-),                 %  Term -> GroundCopy,Substitution
        copy_ground(+,-,+,-),               %  Term->Copy, OldSubst->NewSubst
        copy_ground(+,+,+,+,-),             %  Arity, Term->Copy, OldSubst->NewSubst
        subst_member(+,-,-,-),              %  OldSubst,Var -> Copy,NewSubst
        subst_member(+,-,-).                %  OldSubst,Var -> Copy ?


copy_ground(Term, Copy, Substitution) :-
        copy_ground(Term, Copy, 9=9, Substitution).

        copy_ground(Term, Copy, SubstIn, SubstOut) :-
                var(Term), !,
                subst_member(SubstIn, Term, Copy, SubstOut).
        copy_ground(Term, Term, SubstIn, SubstIn) :-
                simple(Term), !.
        copy_ground(Term, Copy, SubstIn, SubstOut) :-
                functor(Term, Functor, Arity),
                functor(Copy, Functor, Arity), !,
                copy_ground(Arity, Term, Copy, SubstIn, SubstOut).

                copy_ground(0, Term, Copy, SubstIn, SubstIn) :- !.
                copy_ground(N, Term, Copy, SubstIn, SubstOut) :-
                        arg(N, Term, TermN),
                        copy_ground(TermN, CopyN, SubstIn, SubstMid),
                        arg(N, Copy, CopyN),
                        M is N-1, !,
                        copy_ground(M, Term, Copy, SubstMid, SubstOut).

                subst_member(SubstIn, Term, Copy, SubstIn) :-
                        subst_member(SubstIn, Term, Copy), !.
                subst_member(SubstIn, Term, Copy, (Copy = Term) & SubstIn) :-
                        (   SubstIn = (('$VAR'(M) = _) & _),
                                N is M+1                    %  M+1 variables seen
                        |   N = 0                           %  SubstIn = 9=9
                        ), !,
                        Copy = '$VAR'(N).

                        subst_member((Copy = Vrbl) & _, Term, Copy) :-
                                Vrbl == Term, !.
                        subst_member(_ & Rest, Term, Copy) :-
                                subst_member(Rest, Term, Copy).
```

```
    File   : FLAGRO.PL
    Author : Lawrence Byrd + R.A.O'Keefe.
    Updated: 31 October 1983
    Purpose: Flag (global variable) handling.
    Needs  : no other files.

- public
        flag/2,                 %  initialise a flag.
        flag/3.                 %  change a flag.

- mode
        check_valid_flag_name(+),
        flag(+, +),
        flag(+, ?, ?).

/*  Flags are stored in the data base keyed under the Flag itself with
    the information packaged into a compound term as follows:

        Flag -->                '$flag'(Flag, CurrentValue)

    If you only access flags through these routines there will be at
    most one such record per flag.  The flag/2 predicate will clear
    out any records it may find.  The flag/3 predicate maintains the
    flags returning the previous value as Old and updating the flag
    to New.  The code actually checks to see if this updating really
    has to change the data base.  For compatibility with old code, if
    you call flag/3 on a flag which has no record, an old value of 0
    is assumed.  For compatibility with C-Prolog, flags may not be
    integers, but only atoms or compound terms.
*/

check_valid_flag_name(Flag) :-
        nonvar(Flag),
        functor(Flag, Atom, _),
        atom(Atom).
%   There should be a clause to print an error message here.


flag(Flag, InitialValue) :-
        check_valid_flag_name(Flag),
        ( recorded(Flag, '$flag'(Flag,_), Ref), erase(Ref), fail ; true ),
        recorda(Flag, '$flag'(Flag,InitialValue), _).


flag(Flag, OldValue, NewValue) :-
        check_valid_flag_name(Flag),
        (   recorded(Flag, '$flag'(Flag, Old), Ref)  ;  Old = 0   ),
        !,                              %   there should be only one record
        OldValue = Old,                 %   pattern match, may fail
        (   OldValue == NewValue        %   no change needed
        ;   (   var(Ref)  ;  erase(Ref)   ),
            recorda(Flag, '$flag'(Flag,NewValue), _)
        ),  !.
```

```
File    : TIME
Author : Bernard Silver
Updated: 13 January 1984
Purpose: Find current time and converting
milliseconds to hours, mins and seconds.


public
      form_time/2,
      time/1.

mode
      f_z(+,+,-),
      fill_zero(+,-),
      form_time(+,-),
      join(+,+,-),
      make_time(+,+,+,-),
      time(-).


lsys(time(Time)) instantiates Time to the number of MILLIseconds
ince midnight.  time(Time) prints out the hours and mins using the
4 hour clock.



e(NewTime) :-
      plsys(time(Time)),
      form_time(Time,NewTime).

orm_time/2 takes a time in milli-seconds and converts it to
ours, minutes and seconds.
le need to use call as the compiler requires arguments to is/2
o be integers, usually Time is in the form of an xwd due to integer
verflow.
lote that we can't evaluate 60000*60, as this causes overflow,
s/2 reduces the result mod 2^18, and we don't want this to happen.

m_time(Milli,Time) :-
      call(Secs is (Milli/1000) mod 60),
      call(Mins is (Milli/60000) mod 60),
      call(Hours is Milli/(60000*60)),
      make_time(Hours,Mins,Secs,Time).

his call fills in the numbers with leading zeros if necessary,
nd adds the : between the hours, minutes.
nd seconds.

e_time(H,M,S,Time) :-
      fill_zero(H,H1),
      fill_zero(M,M1),
      fill_zero(S,S1),
      join(M1,S1,List1),
      join(H1,List1,List),
      name(Time,List),
      !.

l_zero(X,Fill)  :-
      name(X,X1),
```

```
      f_z(X,X1,Fill).

:(X,X1,X1) :- X > 9,!.
:(_,X1,[48|X1]) :- !.    % 48 is ascii for 0
```

This is a special case of append/3, that also adds ":" to the list

```
n([A,B],[C|D],[A,B,58,C|D]) :- !.     % 58 is ascii for :
n([A],[B|C],[A,58,B|C]) :- !.
```

```
File   : FILE
Author : Lincoln
Updated: 27 January 1984
Purpose: Lincoln's Code for checking File Names for Dec-10 validity


     % valid_filename(+Filename, -Ext) is true if
     %
     %       Filename denotes a file specification of the form:
     %       ========
     %                  specification --> (device):filename.(ext)
     %
     %                  device --> < 11 alphanumeric characters
     %
     %                  filename --> < 7 alphanumeric characters
     %
     %                  ext --> < 4 alphanumeric characters
     %
     %                The device and extension are optional.
     %
     %       Ext        denotes the extension of the input filename.
     %       ===        if no extension is given the atom no_ext is
     %                  returned


public valid_filename/2.
mode valid_filename(+, -),
             valid_filename(-, ?, ?),
                     alphanumeric_string(+, +, ?, -, ?, ?),
                     filename(-, ?, ?),
                     extension(-, ?, ?).

id_filename(UsersFilename, UsersExt) :-
     name(UsersFilename, Ufilename),
     valid_filename(Ext, Ufilename, []),
   ( Ext == no_ext, UsersExt = Ext | name(UsersExt,Ext) ),
     !.

id_filename(Ext) -->
             alphanumeric_string(10, 0, Delim, Length),
           ( {Delim = ":"}, filename(Ext)
            |
             {Length =< 6},
                     ( {Delim = "."}, extension(Ext)
                      |
                       {Delim = no_delim, Ext = no_ext}) ),!.

             filename(Ext) -->
                     alphanumeric_string(6, 0, Delim, _),
                   ( {Delim = "."}, extension(Ext)
                    |
                     {Delim = no_delim, Ext = no_ext} ), !.

                     extension(Ext, Ext, X) :-
                             alphanumeric_string(3, 0, no_delim, _, Ext, X).

hanumeric_string(Bound, N, Delimiter, StringLength) -->
     {N < Bound},
     alphanumeric_char,
     {M is N+1}, !,
```

```
    alphanumeric_string(Bound, M, Delimiter, StringLength).
ıanumeric_string(_, N, Delimiter, N) -->
    delimiter(Delimiter),
    !.
ıanumeric_string(_, N, no_delim, N) --> "".


    alphanumeric_char --> [X], {"a" =< X, X =< "z"}, !.
    alphanumeric_char --> [X], {"A" =< X, X =< "Z"}, !.
    alphanumeric_char --> [X], {"0" =< X, X =< "9"}.

        delimiter(":") --> ":".
        delimiter(".") --> ".".
```

```
 Press:Match.                                  Updated: 14 March 82

:=============================================================================%
                        Pattern Matcher                      24.2.81        %
:=============================================================================%

  Exports...

public
  corresponding_arguments/4,  %   (replaces any1)
  decomp/2,
  match/2,
  recomp/2,
  ac_op/5.

mode
  corresponding_arguments(+,-,-,-),
  decomp(+,?),
      ac_decomp(+,+,?,?),
      ac_op(+,?,?,?,-),
  recomp(?,+),
      ac_recomp(+,+,?),
    tch(+,?),
      match_arguments(+,+,+),
      split_two_ways(+,?,?).

  replace OldA by NewA in one elment of Old, giving New.

responding_arguments([OldA|Tail], OldA, [NewA|Tail], NewA).
responding_arguments([Head|Tail], OldA, [Head|Rest], NewA) :-
      corresponding_arguments(Tail, OldA, Rest, NewA).


-----------------------------------------------------------------------------%

  decomp(Term, List) and recomp(Term, List) are generalisations of univ,
  i.e. Term =.. List, treating the four known associative commutative
  operators as function symbols having any number of arguments.

  They are called in the patterns
      decomp(Old, [Op|Olds]),          %   var(Op)
      any1(<foo>, Olds, News),
      recomp(New, [Op|News]),
  in collect and attract, and elsewhere in the form
      decomp(Old, [+|_])       trig_fac,multiply_through,weaknf
      recomp(New, [+|_])       make_poly.

  ac_op(Op, X, Y, X Op Y, Idn) means that Op is known to be a commutative
  associative operator, that X Op Y =.. [Op,X,Y], and that Idn Op X = X
  i.e. Idn is the identity of Op.  All four operators have an identity.
  The fifth clause is a hack for 1/(X*Y), but is still true.

op(+, X, Y, X+Y, 0)      :- !.
op(*, X, Y, Y*X, 1)      :- !.          %   note reversal!
op(&, X, Y, X&Y, true)   :- !.          %   conjunction
op(#, X, Y, X#Y, false)  :- !.          %   disjunction
c_op(*, X^N, Y^N, (Y*X)^N, 1) :- !.


omp(Term, [Op|Args]) :-
```

```
      functor(Term, Op, 2),
      ac_op(Op, _, _, _, _), !,
      ac_decomp(Term, Op, Args, []).
ecomp((X*Y)^(-1), [*|Args]) :-                  %   special hack
      ac_decomp((X*Y)^(-1), *, Args, []).
omp(Term, List) :-
      Term =.. List.


      ac_decomp(Term, Op, [Term|R], R) :-
              var(Term), !.
      ac_decomp(Term, Op, L, R) :-
              ac_op(Op, X, Y, Term, _), !,
              ac_decomp(X, Op, L, M), !,
              ac_decomp(Y, Op, M, R).
      ac_decomp(Term, Op, [Term|R], R).



omp(Term, [Op|Args]) :-
      ac_op(Op, _, _, _, _), !,
      ac_recomp(Args, Op, Term).
 -(Term, List) :-
      Term =.. List.

      ac_recomp([[]|Args], Op, Term) :- !,
              ac_recomp(Args, Op, Term).
      ac_recomp([Exp], Op, Term) :- !,
              Term = Exp.
      ac_recomp([Exp|Args], Op, Term) :- !,
              ac_op(Op, Exp, Mid, Term, _), !,
              ac_recomp(Args, Op, Mid).
      ac_recomp([], Op, Term) :-
              ac_op(Op, _, _, _, Term).


-------------------------------------------------------------------------%

  match two terms, using the associativity and commutativity of + and *.

:ch(Lhs, Rhs) :-
      functor(Lhs, Op, 2),
      ac_op(Op, Arg1, Arg2, Rhs, _), !,
      decomp(Lhs, [Op|Olds]), !,
      split_two_ways(Olds, [C1|Cs1], [C2|Cs2]),
      recomp(D1, [Op,C1|Cs1]),
      recomp(D2, [Op,C2|Cs2]),
      match(D1, Arg1),
      match(D2, Arg2).

tch(Lhs, Lhs) :-                %   atoms match themselves
      atomic(Lhs), !.

tch(Neg, -1*Pos) :-             %   hack round the representation of
      number(Neg),              %   negative numbers
      eval(Neg < 0),            %   rationals are around now!
      eval(-Neg, Pos), !.
tch(-1*Pos, Neg) :-             %  can't happen if Lhs is tidied first
      number(Neg),
      eval(Neg < 0),
```

```
    eval(-Neg, Pos), !.

ch(Lhs, Rhs) :-
    functor(Lhs, Functor, Arity),
    functor(Rhs, Functor, Arity), !,
    match_arguments(Arity, Lhs, Rhs).


    match_arguments(0, Lhs, Rhs) :- !.
    match_arguments(N, Lhs, Rhs) :-
            arg(N, Lhs, LhsNth),
            arg(N, Rhs, RhsNth),
            match(LhsNth, RhsNth),
            M is N-1,
            match_arguments(M, Lhs, Rhs).


    split_two_ways([Head|Tail], A, B) :-
            split_two_ways(Tail, A1, B1),
            (   A = [Head|A1], B = B1
            |   B = [Head|B1], A = A1
            ).
    split_two_ways([], [], []).
```

Given a Term, discover all the constants, atoms, and functors occuring in it. The Term is known to be ground.

Special code for matching for Leon

```
ctors_in(Term, List) :-
    functors_in(Term, L, []),
    sort(L, List).

    functors_in(Term, [Term|R], R) :-
            atom(Term), !.
    functors_in(Term, [Abso|R], R) :-
            number(Term), !,
            eval(abs(Term), Abso).
    functors_in(Term, [Head|L], R) :-
            functor(Term, Functor, Arity),
            functor(Head, Functor, Arity), !,
            functors_in(Arity, Term, L, R).

            functors_in(0, Term, R, R) :- !.
            functors_in(N, Term, L, R) :-
                    arg(N, Term, Argument),
                    functors_in(Argument, L, M),
                    K is N-1, !,
                    functors_in(K, Term, M, R).
```

```
POLTID : New simplification code using polynomial simplification

                                                    Leon
                                                    Updated: 9 September 82

claration%
public
              simplify/2,
              simplify/3,
              poly_tidy/2.

mode
              simplify(+,-),
              simplify(+,+,-),
              select_letter(-,+),
              poly_tidy(+,-),
              pol_tidy(+,-).

plify(Expr,Expr) :- atomic(Expr), !.

plify(Expr,Simp) :-
      wordsin(Expr,List),
      select_letter(A,List),
      mult_occ(A,Expr),
      is_poly(A,Expr),
      !,
      simplify(Expr,A,Simp).

plify(Expr,Expr).

ect_letter(A,[A|List]).      % Use sorting property of wordsin as heuristic
                             % for selecting letter for simplifying

plify(Expr,Sub,Simp) :-
      poly_norm(Expr,Sub,Pbag),
      poly_tidy(Pbag,Tidy),
      make_poly(Sub,Tidy,Simp).

y_tidy(Pbag,Tidy) :- pol_tidy(Pbag,Qbag), z_norm(Qbag,Tidy).

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

idy([polyand(N,Expr)|Rest],[polyand(N,Simp)|TidyRest]) :-
      tidy(Expr,Tidy),
      simplify(Tidy,Simp),
      pol_tidy(Rest,TidyRest).
```

```
SIMP.AX : Simplification axioms for TIDY

                                                    Bernard Silver
                                                    Updated: 23 February 1984


UBLIC
public
     negative_angle/2,
     simplify_axiom/2.

ODES

mode
     negative_angle(+,-),
     simplify_axiom(+,-).

plify_axiom(U*U^ -1,1).
plify_axiom(U^ -1*U,1).

ogs
plify_axiom(log(U,U^V),V).
plify_axiom(log(A,1),0).
plify_axiom(U^log(U,V),V).
plify_axiom(log(A,A),1) :- A\==1.

ommon trig cases
plify_axiom(sin(arccos(X)),(1-X^2)^(1/2)#(1-X^2)^(1/2)*(-1)).

plify_axiom(cos(arcsin(X)),(1-X^2)^(1/2)#(1-X^2)^(1/2)*(-1)).

plify_axiom(arcsin(cos(X)),90-X).

plify_axiom(arccos(sin(X)),90-X).

plify_axiom(sin(X),sin(X1)* -1) :- negative_angle(X,X1),!.

plify_axiom(cos(X),cos(X1)) :- negative_angle(X,X1),!.

ative_angle(-X,X) :- !.
ative_angle(X,X1) :-
     match(X,A*B),
     number(A),
     !,
     eval(A<0),
     eval(-A,A1),
     tidy(A1*B,X1),
     !.
```

```
/* GPORTR : First stab at a general all level portray handler.

                                                    Richard+Lawrence
                                                    Updated: 26 February 1984

        This was Richard's code for his rational stuff.
        Eventually I must fix these problems by having the 'print'
        routine in the interpreter actually descend level by level
        taking operators into account and calling portray at each
        level to see whether the users wants to handle it.
        NB: this has now been done.  Why is gportr still around?

        The following magic numbers appear in put(N) calls:
        32 = space, 40 = "(", 41 = ")", 44 = ",", 91 = "[", 93 = "]".
        The magic number 1000 also appears; this is the priority of ','.

*/


/* EXPORT */

:- public
    portray/1.


/* MODES */

:- mode
    portray(?),
        prin(+, +),
            prin(+, +, +),
            prnf(+, +, +),
            prna(+, +, +),
            prnp(+, +, +, +),
            printail(+),
            oper(+, ?, ?),
                oper(+, +, ?, ?).


                        % Top level

   tray(Term) :-
        prin(1000, Term).


                        % Print a term taking account of surrounding
                        %  operator priorities.

        prin(Prio, Term) :-
                (   var(Term)           %  _N style of variables
                |   atom(Term)          %  ordinary atoms
                |   Term = '$VAR'(N)    %  A1 style of variables from numbervars
                ), !,
                writeq(Term).           %  quotes around e.g. 'foo baz'
        prin(Prio, Term) :- /*Q'*/
                portray_number(Term),   %  if a number
                !.
```

```
/*  Other user-provided portrayal methods should be called here  */
prin(Prio, [Head|Tail]) :- !,    %  list
        put(91),                 %  "["
        prin(1000, Head),
        printail(Tail).
prin(Prio, Term) :-              %  postfix operator
        functor(Term, Functor, 1),
        oper(Functor, Lp, 0), !,
        prnp(Prio, Lp, 0, 40),
        prna(Lp, Term, 1),
        prnf(Functor, 0, 1),
        prnp(Prio, Lp, 0, 41).
prin(Prio, Term) :-              %  prefix operator
        functor(Term, Functor, 1),
        oper(Functor, 0, Rp), !,
        prnp(Prio, 0, Rp, 40),
        prnf(Functor, 1, 0),
        prna(Rp, Term, 1),
        prnp(Prio, 0, Rp, 41).
prin(Prio, Term) :-              %  infix operator
        functor(Term, Functor, 2),
        oper(Functor, Lp, Rp),
        Lp > 0, Rp > 0, !,
        prnp(Prio, Lp, Rp, 40),
        prna(Lp, Term, 1),
        prnf(Functor, 0, 0),
        prna(Rp, Term, 2),
        prnp(Prio, Lp, Rp, 41).
prin(Prio, Term) :-
        functor(Term, Functor, N),
        writeq(Functor),
        prin(0, N, Term).


                                 % print one argument of a term

        prna(Prio, Term, ArgNo) :-
                arg(ArgNo, Term, Arg),
                prin(Prio, Arg).

                                 % print a functor with spaces

        prnf(',', _, _) :- !,
                write(', ').
        prnf(';', _, _) :- !,
                write('; ').
        prnf('#',L,R) :- !,
                prnp(L, 1, 1, 32),
                write(' v '),
                prnp(R, 1, 1, 32).
        prnf(Functor, L, R) :-
                prnp(L, 1, 1, 32),
                write(Functor),
                prnp(R, 1, 1, 32).

                                 % print the arguments of a term

        prin(0, N, Term) :-
                put(40),                 %  "("
```

```
                prna(1000, Term, 1),
                prin(1, N, Term).
        prin(N, N, Term) :- !,
                put(41).                        %  ")"
        prin(L, N, Term) :-
                M is L+1,
                write(', '),
                prna(1000, Term, M), !,
                prin(M, N, Term).


                                % Print a parenthesis if the priorities
                                %  around the operator require it.

        prnp(Prio, Lp, Rp, Char) :-
                Prio >= Lp, Prio >= Rp, !.
        prnp(Prio, Lp, Rp, Char) :-
                put(Char).


                                % Print the tail of a list, being
                                %  careful about partial instantiation
                                %  at the end.

        printail(List) :-
                nonvar(List), List = [Head|Tail], !,
                write(', '),
                prin(1000, Head), !,
                printail(Tail).
        printail(Tail) :-
                Tail \== [],
                put(124),                       %  "|"
                prin(1000, Tail), !,
                printail([]).
        printail([]) :-
                put(93).                        %  "]"


                % Check for operators.  Return left and right
                %  precedences. These are Richard's conventions.
                %  Note that prefix/postfix ops have 0 for their
                %  other precedence.

oper(Op, Left,Right) :-
        current_op(Prec, Type, Op),
        oper(Type, Prec, Left, Right).


        oper( fx, Prec, 0, Prec).
        oper( fy, Prec, 0, Prec).
        oper(xf , Prec, Prec, 0).
        oper(yf , Prec, Prec, 0).
        oper(xfx, Prec, Prec, Prec).
        oper(xfy, Prec, Prec, More) :- More is Prec+1.
        oper(yfx, Prec, More, Prec) :- More is Prec+1.
```

'* LONG.PL : Arbitrary precision rational arithmetic package.

 Updated: 30 August 82

Designed and written by Richard O'Keefe.
Scenery by Lawrence Byrd.

This package provides arithmetic for arbitrary precision rational numbers. The normal domain of prolog 'integers' is extended to full rational 'numbers'. This domain includes all Prolog integers. The predicate:

```
number(N)
```

will recognise any number in this extended domain.
Rational numbers are produced by using the predicates

```
eval(Command)

eval(Expression,Answer)
```

Expression can involve any form of rational number, whether such numbers can be represented by Prolog integer or not. Any form of number produced as output by "eval" is acceptable as input to it.

For convenience the Answer produced by eval is normalised as follows:

a) Integers X (where |X| <= 99999) are represented as Prolog integers;

b) 1/0, 0/0, -1/0 are represented as infinity, undefined, neginfinity;

c) All other numbers are represented as full rationals in reduced form i.e. numerator and denominator are relatively prime.

In the current representation, one normalised number will unify with another (including an integer) iff the two numbers are equal. But it is better to test for equality between arbitrary numbers by calling

```
eval(N1=:=N2)
```

which also handles infinity & undefined, and is guaranteed to work. Once created, representations of rational numbers can be passed round your program, used with eval, or printed. The predicate

```
portray_number(Number)
```

will pretty-print arbitrary numbers, and will fail for anything else. In particular, it will not evaluate an expression. (But eval(write(Expr)) combines evaluation and printing if you want.) If this is connected up to your general "portray" mechanism, you will never have to see the internal representation of rationals. It is ill-advised to write procedures which assume knowledge of this internal representation as it is subject to change (rarely), not to mention that such activities are against all the principles of abstraction and structured programming.

NB Note that eval/1 and eval/2 will only evaluate fully numeric expressions. If there is some garbage in the expression (such as an atom) then no evaluation at all occurs and the whole input expression is returned untouched. If you want to evaluate

mixed symbolic and numeric expressions then use tidy/2 (from TIDY.PL) which is designed for this purpose.

IXES

3 April 81]

Added the functions numer(_), denom(_) and sign(_) to the evaluator (ie eva2).

8 April 81]

removed choice-points from comq, and corrected sign(.).
replaced the log routine completely.

14 April 81]

changed all XXXr routines to XXXn (for Natural or zero)
changed all XXXs routines to XXXm (for Modified (Natural routines))
changed all XXXl routines to XXXz (for the ring Z of integers)
replaced "digits" by "conn" as I've meant to for some time.
removed experimental 'xwd' code which doesn't work compiled. Eheu.
changed estq,chkq,gest to estd,chkd,estg (estimate division digit, check digit, estimate Gcd) to avoid confusion; they don't use rationals.
rewrote norm_number and renamed it to standardise.
laid the trig routines out in MY style not Lawrence's.
Increased the radix from 10,000 to 100,000 after fixing addn to use unsigned numbers.

21 April 81]

Continued tidying things up.
made 0^(1/N) = 0; this was an oversight.
added new xwd(,) code in eva2, and beautified portray_number.

8 July 81]

fixed mode error bug in eva2(abs(_),_). Foolish oversight.

9 Sept 81]

fixed negative number bugs in arccos and arcsin. How long have these been around without anyone (except Bernard) noticing? Also shunted some cuts around in the same general area.

13 Sept 81]

corrected typo {da=>Da} in gcdq/4.

2 Dec 81]

corrected a benign bug in number/5 (100000 had been written where R should have been), and some minor cosmetic changes. Unified error reporting into long_error[_message]. Added a few mode declarations for trig functions.

9 Dec 81]

when writing eval up for EuroSam, discovered that logs aren't handled properly. Rewrote absq and logq to return 'undefined'

in more cases, instead of failing.

27 July 82]

changed prnq/3 to portray_number/3 and laid it out properly.
changed prin/1 to putn/1 (put Natural) to avoid conflict elsewhere.
Made this stuff call put/1 where it made sense, and used ASCII
codes instead of strings. Don't know if it matters, really.
Also rewrote arctaneval completely, so that it should succeed in a
few more cases. Really, the trig stuff is PITIFUL. Please, will
someone do a proper job of it (preferrably someone PAID to do it).

30 August 82]

fixed bug in gcdq/4 so that gcd(1/2,1/4) = 1/4, not 1/2!

12 July 1983]

arctaneval used to call addn/4, and there isn't any such
predicate. Made it call addn/5.
*/

```
:-  public
        number/1,                   %  number(N) <=> N is a number
        eval/1,                     %  eval(E) => E/rational-eval is true
        eval/2,                     %  eval(E,A) => (A is E)/rational-eval
        portray_number/1,           %  writes rational assumed radix 100000.
;                                   %  Lawrence's Low Level TIDY interface
        add/3,                      %  add(A,B,C) => (C is A+B)/rational-eval
        multiply/3,                 %  similar for *.  NB A,B must be numbers
        power/3.                    %  similar for ^.  NOT general terms.


'* OPERATORS */

:- op(300, xfx, div).              %  integer quotient A div B = fix(A/B).


'* MODES and types */

;   The comments at the right give the argument types for each predicate.
;   The predicates can of course be called with any arguments, but these
;   are the only types they are supposed to work on or deliver.
;   ? = any Prolog term, possibly including variables.
;   E = an arithmetic Expression, a term.
    A = a Prolog atom (but not an integer).
    I = a Prolog integer.  Generally positive, but not always.
    T = a Truth-value, 'true' or 'false'.
;   S = a Sign, '+' or '-'.  {Sometimes can be 0 or *.}
;   R = a Relational operator, {<, =, >; sometimes =<, >=, =/=}
;   N = a long positive (Natural) number.
;   Q = a rational number.

:- mode

;% Top Level %%

    number(+),                                  % Q
    eval(+),                                    % E
    eval(+, -),                                 % E Q
        eva2(+, -),                             % E Q
            relational_op(+, -, -),             % R R T
            combine_ops(+, +, +, -),            % R R T T
    portray_number(+),                          % Q
        portray_number(+, +, +),                % S N N
            putn(+),                            % N

;% Conversions %%

    number(+, +, ?, ?, ?),                      % Q I S N N
        binrad(+, +, -),                        % I I N
    standardise(+, ?),                          % Q Q

;% Low Level %%

    add(+, +, ?),                               % Q Q Q
    multiply(+, +, ?),                          % Q Q Q
    power(+, +, ?),                             % Q Q Q

;% Rational Arithmetic %%

    mod2(+, ?),                                 % Q I
```

```
    intq(+,    +, -),                          % Q    I Q
    gcdq(+, +, +, -),                          % Q Q I Q
/*  invq(+,    +, -),                          % Q    I Q  */
    mulq(+, +, +, -),                          % Q Q I Q
    divq(+, +, +, -),                          % Q Q I Q
    divo(+, +, +, -, -),                       % Q Q I Q Q
    powq(+, +, +, -),                          % Q Q I Q
    negq(+,    +, -),                          % Q    I Q
    addq(+, +, +, -),                          % Q Q I Q
    subq(+, +, +, -),                          % Q Q I Q
    comq(+, +, +, ?),                          % Q Q I R
    nthq(+, +, +, -),                          % I Q I Q
        nthn(+, +, +, -),                      % I N I N
            newton(+, +, +, +, -),             % I N N I N
                newton(+, +, +, +, +, -),      % R I N N I N

  %% Long Arithmetic %%

    addz(+,+, +,+, +, -,-),                    % S N S N I S N
        addn(+, +, +, +, -),                   % N N I I N
            add1(+, +, -),                     % N I N
    comz(+,+, +,+, ?),                         % S N S N R
        comn(+, +, +, ?),                      % N N R R
            com1(+, +, +, -),                  % I I R R
    subz(+,+, +,+, +, -,-),                    % S N S N I S N
        subn(+, +, +, -,-),                    % N N I S N
            subn(+, +, +, +, -,-),             % R N N I S N
                prune(+, -),                   % N N
                subp(+, +, +, +, -),           % N N I I N
                    sub1(+, +, -),             % N I N
    sign(+, +, -),                             % S S S
/*  mulz(+,+, +,+, +, -,-),                    % S N S N I S N  */
        muln(+, +, +, -),                      % N N I N
            muln(+, +, +, +, -),               % N N N I N
                mul1(+, +, +, -),              % N I I N
                    mul1(+, +, +, +, -),       % N I I I N
    powz(+,+, +, +, -,-),                      % S N I I S N
        pown(+, +, +, +, -),                   % I N N I N
    divz(+,+, +,+, +, -,-, -,-),               % S N S N I S N S N
        divn(+, +, +, -, -),                   % N N I N N
            conn(?, ?, ?),                     % I N N
            %   both +, +, - and -, -, + are used.
            div1(+, +, +, -, -),               % N I I N N
            divm(+, +, +, -, -),               % N N I N N
                div2(+, +, +, -, -),           % N N I N N
                    estd(+, +, +, -),          % N N I I
                    chkd(+, +, +, +, +, -, -), % N N I I I I N
/*  gcdz(+,+, +,+, +, -, -,-, -,-),            % S N S N I N S N S N  */
        gcdn(+, +, +, -, -, -),                % N N I N N N
            gcdn(+, +, +, -),                  % N N I N
                gcdn(+, +, +, +, -),           % R N N I N
                    estg(+, +, +, -),          % N N I I

%% Logarithms %%

    logq(+, +, +, -),                          %  Q Q I Q
        logq(+,+, +,+,+,-),                    %  R R Q Q I Q
        absq(+, -, -),                         %  Q S Q
            logq(+, -,-),                      %  S S N
            oneq(+, -, -),                     %  Q R Q
```

```
        ratlog(+, +, +, -),                     %  Q Q I Q
            ratlog(+,+, +,+,+, -),              %  S S Q Q I Q
                lograt(+,+,+, -,-),             %  Q Q I N N
                    loop(+, +, +, -),           %  N N I N
                        loop(+,+,+,+,-),        %  N N N I N
                    logn(+,+,+,+,-),            %  Q I Q Q I I

%% Trigonometry %%

    sineval(+, -),                              %  Q Q
    coseval(+, -),                              %  Q Q
    taneval(+, -),                              %  Q Q
    arcsineval(+, -),                           %  Q Q
    arccoseval(+, -),                           %  Q Q
    arctaneval(+, -),                           %  Q Q
        arctaneval(+, +, -, -),                 %  N N N N
        sineval1(?, ?),                         %  Q Q

%% Error handing %%

    long_error(+, ?),                           %  A ?
        long_error_message(+, -).               %  A A
```

* Implementation

The internal representation for rationals is of the form:

number(Sign, Numerator, Denominator)

where
Sign is in {+,-}
Numerator is a list of (Prolog) integers
Denominator is a list of (Prolog) integers

The lists of Prolog integers represent arbitrary precision unsigned long integers

eg [n0,n1,....,nz]

is n0+R*(n1+R*(....R*nz)...)

where R is the Radix.

The Radix used in the current version is 100000. Most of the code in this module is completely independent of the radix - it all uses the value passed in by the top level procedures. However the printing routine currently assumes that the radix is a power of 10 as this makes things easier. In general the radix must be such that both:

Radix^2 - 1
and Radix*2 + 1

are representable as Prolog integers (which are 18 bit quantities on the DEC10). This is a little restrictive, however, and this implementation only assumes that Radix^2 - 1 is "obtainable" as an intermediate during Prolog arithmetic. On the DEC10 intermediate results can be 36 bit quantities and so 100000 becomes a suitable radix.

The code actually unpacks the number terms into their separate bits for all the low level operations. At this stage the following additional number forms are appropriately converted

<integer> - (Prolog integers)
infinity - represented as +1/0
neginfinity - represented as -1/0
undefined - represented as 0/0

The treatment of these strange things is not supposed to be mathematically beautiful, but sensible things happen using this representation. They are strictly an extension to the rationals and could be removed (with eval failing should 0 denominator numbers ever get produced) if desired.

Results from eval are normalised before being returned. This operation reverses the above transformation except that only integers within the range -99999 to +99999 are turned back into Prolog integers.

k/

```
% TOP LEVEL PREDICATES %%


                        % Number recognition predicate

umber(N)        :- integer(N), !.
umber(number(S,N,D))    :- !.
umber(infinity)         :- !.
umber(neginfinity)      :- !.
umber(undefined)        :- !.


                        % Simple eval interpreter with various features.

val(Var)        :- var(Var), !, long_error(eval, Var).
val(B is Y)     :- !, eval(Y, B).
val(write(Y))   :- !, eval(Y, B), print(B).
val(even(X))    :- !, eva2(X, A), !, mod2(A, 0).
val(odd( X))    :- !, eva2(X, A), !, mod2(A, 1).
val(compare(R,A,B)) :-
                   !, eva2(X, A), eva2(Y, B), comq(A, B, 100000, S), !, R=S.
val(Term)       :- Term =.. [F,X,Y], relational_op(F, R, Flag),
                   !, eva2(X, A), eva2(Y, B), comq(A, B, 100000, S), !,
                   combine_ops(R, S, Flag, true).

        mod2(number(_,_,[]),    M) :- !, fail.
        mod2(number(_,[],_),    0).
        mod2(number(_,[L|_],_), M) :- M is L mod 2.


                        % General evaluation of rational expressions

val(Exp, Ans) :-        %    Hope for the best
        eva2(Exp, N),
        standardise(N, A), !,
        Ans = A.
val(Exp, Exp).          %    Cannot evaluate so leave alone
%       ttynl, display('[Couldn''t evaluate: '),
%       print(Exp), ttyput("]"), ttynl, ttynl.


va2(Var, C)       :- var(Var), !, long_error(eva2, Var).
va2(X+Y, C)       :- !, eva2(X, A), eva2(Y, B), addq(A, B, 100000, C).
va2(X-Y, C)       :- !, eva2(X, A), eva2(Y, B), subq(A, B, 100000, C).
va2( -Y, C)       :- !,             eva2(Y, B), negq(   B, 100000, C).
va2(X*Y, C)       :- !, eva2(X, A), eva2(Y, B), mulq(A, B, 100000, C).
va2(X/Y, C)       :- !, eva2(X, A), eva2(Y, B), divq(A, B, 100000, C).
va2(X div Y, C)   :- !, eva2(X, A), eva2(Y, B), divo(A, B, 100000, C, _).
va2(X mod Y, C)   :- !, eva2(X, A), eva2(Y, B), divo(A, B, 100000, _, C).
va2(X++Y, C)      :- !, eva2((X+Y) mod 360, C).
va2(X--Y, C)      :- !, eva2((X-Y) mod 360, C).
va2(X^Y, C)       :- !, eva2(X, A), eva2(Y, B), powq(A, B, 100000, C).
va2(sqrt(Y), C)   :- !,             eva2(Y, B), nthq(2, B, 100000, C).
va2(pi,number(+,[355],[113])) :- !.
va2(log(X,Y),C)   :- !, eva2(X, A), eva2(Y, B), logq(A, B, 100000, C).
va2(gcd(X,Y),C)   :- !, eva2(X, A), eva2(Y, B), gcdq(A, B, 100000, C).
va2(fix(X), C)    :- !, eva2(X, A),             intq(A,    100000, C).
```

```
eva2(sin(X), C)  :- !, eva2(X, A), sineval(A, C).
eva2(cos(X), C)  :- !, eva2(X, A), coseval(A, C).
eva2(tan(X), C)  :- !, eva2(X, A), taneval(A, C).
eva2(arcsin(X),C):- !, eva2(X, A), arcsineval(A, C).
eva2(arccos(X),C):- !, eva2(X, A), arccoseval(A, C).
eva2(arctan(X),C):- !, eva2(X, A), arctaneval(A, C).
eva2(abs(X),   number(+,N, D )) :- !, eva2(X, A), A = number(_,N,D).
eva2(numer(X), number(+,N,[1])) :- !, eva2(X, A), A = number(_,N,_).
eva2(denom(X), number(+,D,[1])) :- !, eva2(X, A), A = number(_,_,D).
eva2(sign(X),  number(S,B,[1])) :- !, eva2(X, A), A = number(S,N,_),
                                      (N=[], B=[]; B=[1]), !.
eva2(xwd(X,Y),C) :- !, U is !(Y)>>9, V is !(Y)/\511,
                    eva2((X*512+U)*512+V, C).
eva2(X,        C) :- number(X, 100000, S, N, D), !, C = number(S, N, D).
eva2(Term,     C) :- Term =.. [F,X,Y], relational_op(F,R, Flag),
                    !, eva2(X, A), eva2(Y, B), comq(A, B, 100000, S), !,
                    combine_ops(R, S, Flag, C).


        relational_op(  =, =, true).
        relational_op( \=, =, false).
        relational_op(  <, <, true).
        relational_op( >=, <, false).
        relational_op(  >, >, true).
        relational_op( =<, >, false).
        relational_op(=:=, =, true).
        relational_op(=\=, =, false).

        combine_ops(Sign, Sign, Flag,  Ans) :- !, Ans = Flag.
        combine_ops(Sign, Diff, true,false) :- !.
        combine_ops(Sign, Diff, false,true) :- !.



                        % Pretty-Print a number.
                        %  This now always forces parentheses. When a
                        %  proper general portray handler is written
                        %  this could be made cleverer (as it once was).
                        %  The magic numbers are 40 = "(", 41 = ")",
                        %  45 = "-", 47 = "/", 48 = "0" {ASCII codes}.

portray_number(A) :-
        number(A, 100000, S, N, D),     !,
        portray_number(S, N, D).

        portray_number(_,[],  []) :- !,         %  0/0 = undefined
                write(undefined).
        portray_number(+, N,  []) :- !,         % +N/0 = +infinity
                write(infinity).
        portray_number(-, N,  []) :- !,         % -N/0 = -infinity
                write(neginfinity).
        portray_number(+, N, [1]) :- !,         % +N/1 = a +ve integer
                putn(N).
        portray_number(-, N, [1]) :- !,         % -N/1 = a -ve integer
                put(45), putn(N).
        portray_number(+, N,  D ) :- !,         % +N/D = a +ve rational
                put(40), putn(N), put(47), putn(D), put(41).
        portray_number(-, N,  D ) :- !,         % -N/D = a -ve rational
                put(40), put(45), putn(N), put(47), putn(D), put(41).
```

```
putn([]   ) :- !, put(48).
putn([D]  ) :- !, write(D).
putn([D|T]) :-
        putn(T),
        D4 is (D/10000)       +48, put(D4),     % D4*10^4 +
        D3 is (D/1000) mod 10 +48, put(D3),     % D3*10^3 +
        D2 is (D/100) mod 10  +48, put(D2),     % D2*10^2 +
        D1 is (D/10) mod 10   +48, put(D1),     % D1*10^1 +
        D0 is (D) mod 10      +48, put(D0).     % D0*10^0 = D.
```

```
% INTERFACE CONVERSIONS %%

                        % Conversion of a number, of any form, to its
                        %  essential bits.

umber(infinity,         R, +,[1], []) :- !.
umber(neginfinity,      R, -,[1], []) :- !.
umber(undefined,        R, +, [], []) :- !.
umber(number(S, N, D), R, S,  N,  D) :- !.
umber(N, R, +, L, [1]) :- integer(N), N >= 0, !,          binrad(N, R, L).
umber(N, R, -, L, [1]) :- integer(N), N  < 0, !, M is -N, binrad(M, R, L).

        binrad(0, R, [])    :- !.
        binrad(N, R, [M|T]) :- K is N/R, M is N mod R, !, binrad(K, R, T).


                        % Normalise a number

tandardise(number(S,[N],[1]), Ans) :- !,
        (   S = '+', Ans = N
        ;   S = '-', Ans is -N
        ),  !.
tandardise(number(_, [],[1]),  0 ) :- !.
+   lardise(number(S,  N, []), Ans) :- !,
        (   N =  [], Ans = undefined
        ;   S = '+', Ans = infinity
        ;   S = '-', Ans = neginfinity
        ),  !.
tandardise(Number,          Number).



% LOW LEVEL INTERFACE %%


                        % These routines provide a low level interface
                        %  for procedures which want to operate directly
                        %  on pairs of numbers.
                        % Only currently used by TIDY (27/2/81),
                        %  so only those necessary are provided.

   A, B, C) :-          % eval(C is A+B).
        addq(A, B, 100000, X),
        standardise(X, C).

ultiply(A, B, C) :-     % eval(C is A*B).
        mulq(A, B, 100000, X),
        standardise(X, C).

ower(A, N, C) :-        % eval(C is A^B).
        powq(A, N, 100000, X),
        standardise(X, C).
```

```
% BASIC ARITHMETIC OVER RATIONALS %%


                        % Integer part of a rational

ntq(A, R, number(S, Q, [1])) :-
        number(A, R, S, N, D),
        divn(N, D, R, Q, _).



                        %   The greatest common divisor of two numbers is
                        %   defined for all pairs of non-zero rationals.
                        %   gcd(X,Y) = Z iff Z > 0 and there are integers
                        %   M,N relatively prime for which X=MZ & Y=NZ.

cdq(A, B, R, number(+,Nd,Dd)) :-
        number(A, R, _, Na, Da),
        number(B, R, _, Nb, Db),
        gcdn(Da, Db, R, _, Ga, Gb),
        muln(Gb, Na, R, Ma),
        muln(Ga, Nb, R, Mb),
        gcdn(Ma, Mb, R, Nd),
        muln(Gb, Da, R, Dd).

*       The above seems to be right, but I'm not sure.  This IS right.
cdq(A, B, R, number(+,Nd,Dd)) :-
        number(A, R, _, Na, Da),          %  |A| = Na/Da
        number(B, R, _, Nb, Db),          %  |B| = Nb/Db
        muln(Na, Db, R, N1),              %  N1 = Na.Db
        muln(Nb, Da, R, N2),              %  N2 = Nb.Da
        gcdn(N1, N2, R, Nc),              %  Nc = gcd(Na.Db, Nb.Da)
        muln(Da, Db, R, Dc),              %  Dc = Da.Db
        gcdn(Nc, Dc, R, _, Nd, Dd).       %  Nd/Dd = Nc/Dc in standard form
/

*                       % Take the inverse of a rational

nvq(A, R, number(S, D, N)) :-
        number(A, R, S, N, D).

/

                        % Multiplication of two rationals

ulq(A, B, R, number(Sc, Nc, Dc)) :-
        number(A, R, Sa, Na, Da),
        number(B, R, Sb, Nb, Db),
        sign(Sa, Sb, Sc),
        gcdn(Na, Db, R, _, Na1, Db1),
        gcdn(Da, Nb, R, _, Da1, Nb1),
        muln(Na1, Nb1, R, Nc),
        muln(Da1, Db1, R, Dc).



                        % Division of two rationals

ivq(A, B, R, number(Sc, Nc, Dc)) :-
        number(A, R, Sa, Na, Da),
```

```
        number(B, R, Sb, Nb, Db),
        sign(Sa, Sb, Sc),
        gcdn(Na, Nb, R, _, Na1, Nb1),
        gcdn(Da, Db, R, _, Da1, Db1),
        muln(Na1, Db1, R, Nc),
        muln(Da1, Nb1, R, Dc).


                        % Quotient and remainder of two rationals

iivo(A, B, R, number(Sq,Nq,[1]), number(Sx,Nx,Dx)) :-
        number(A, R, Sa, Na, Da),          %  A = Sa.Na/Da
        number(B, R, Sb, Nb, Db),          %  B = Sb.Nb/Db
        muln(Na, Db, R, N1),               %  A/B = (Sa.Na.Db)/(Sb.Nb.Da)
        muln(Nb, Da, R, D1),               %      = (Sa.N1)/(Sb.D1)
        divz(Sa,N1, Sb,D1, R, Sq,Nq, Sx,Ny),
        muln(Da, Db, R, Dy),               %  A/B = Q + (Sx.Ny)/(Sb.Nb.Da)
        gcdn(Ny, Dy, R, _, Nx, Dx).        %  A = Q.B + (Sx.Ny)/Dy


                        % Exponentiation of rationals
                        %  This is always defined for (positive or
                        %   negative) integer powers, however there
                        %   is a current implementation restiction that
                        %   the power be between -99999 and +99999 (ie
                        %   within the current Radix).
                        %  This may be defined for some rational powers
                        %   but since there are results from this which are
                        %   not representable as rationals it will fail
                        %   in such cases.  The code for rational powers
                        %   relies on numerator and denominator being
                        %   relatively prime, which is standard.

owq(A, B, R, C) :-
        number(B, R, S, N, [1]), !,
        powq(S, N, A, R, C).
owq(A, B, R, C) :-
        number(B, R, S, N, [D]),
        nthq(D, A, R, X), !,
        powq(S, N, X, R, C).

        powq(S, [], A, R, number(+,[1],[1])) :- !.
        powq(+,[N], A, R, number(Sc, Nc, Dc)) :- !,
                number(A, R, Sa, Na, Da),
                powz(Sa, Na, N, R, Sc, Nc),
                pown(N,  Da,[1],R,     Dc).
        powq(-,[N], A, R, number(Sc, Nc, Dc)) :- !,
                number(A, R, Sa, Na, Da),
                powz(Sa, Da, N, R, Sc, Nc),
                pown(N,  Na,[1],R,     Dc).


                        % Negate a rational

egq(A, R, number(Sc, Nc, Dc)) :-
        number(A, R, Sa, Nc, Dc),
        (   Nc = [], Dc = [], Sc = +            %  -undefined=undefined
```

```
        ;   sign(Sa, -, Sc)                     %  -0 = -(0) now.
        ),  !.


                        % Addition of two rationals

iddq(A, B, R, number(Sc, Nc, Dc)) :-
        number(A, R, Sa, Na, Da),
        number(B, R, Sb, Nb, Db),
        muln(Na, Db, R, Xa),
        muln(Nb, Da, R, Xb),
        addz(Sa,Xa, Sb,Xb, R, Sc,Xc),
        gcdn(Xc, Da, R, _, Nx, Ya),
        gcdn(Nx, Db, R, _, Nc, Yb),
        muln(Ya, Yb, R, Dc), /*Q'*/ Nc/Dc\==[]/[], !.
iddq(A, B, R, number(Sc, Nc, [])) :- /*Q'*/
        number(A, R, Sa, Na, Da),
        number(B, R, Sb, Nb, Db),
        (   Na\==[], Nb\==[], Sa==Sb, Sc=Sa, Nc=[1]
        ;   Sc= +, Nc=[]
        ),  !.


                        % Subtraction of two rationals

ubq(A, B, R, number(Sc, Nc, Dc)) :-
        number(A, R, Sa, Na, Da),
        number(B, R, Sb, Nb, Db),
        muln(Na, Db, R, Xa),
        muln(Nb, Da, R, Xb),
        subz(Sa,Xa, Sb,Xb, R, Sc,Xc),
        gcdn(Xc, Da, R, _, Nx, Ya),
        gcdn(Nx, Db, R, _, Nc, Yb),
        muln(Ya, Yb, R, Dc), /*Q'*/ Nc/Dc\==[]/[], !.
ubq(A, B, R, number(Sc, Nc, [])) :- /*Q'*/
        number(A, R, Sa, Na, Da),
        number(B, R, Sb, Nb, Db),
        (   Na\==[], Nb\==[], Sa\==Sb, Sc=Sa, Nc=[1]
        ;   Sc= +, Nc=[]
        ),  !.


                        % Comparison of two rationals

omq(A, B, R, S) :-
        number(A, R, Sa, Na, Da), /*Q'*/ Na/Da \== []/[],
        number(B, R, Sb, Nb, Db), /*Q'*/ Nb/Db \== []/[],
        muln(Na, Db, R, Xa),
        muln(Nb, Da, R, Xb),    !,
        comz(Sa, Xa, Sb, Xb, S).


                        % Try to find Nth root
                        %  This will fail in cases where the solution is
                        %  not representable as a rational
```

```
nthq(N, A, R, number(+, Nr, Dr)) :-
        number(A, R, +, Na, Da), !,
        nthn(N, Na, R, Nr),
        nthn(N, Da, R, Dr).
nthq(N, A, R, number(-, Nr, Dr)) :-
        number(A, R, -, Na, Da), !,
        1 is N mod 2,
        nthn(N, Na, R, Nr),
        nthn(N, Da, R, Dr).

        nthn(N,  [], R,  []) :- !.
        nthn(N, [1], R, [1]) :- !.
        nthn(N,   A, R,   S) :-
                newton(N, A, A, R, S), !,
                pown(N, S, [1], R, B), !, B=A.  % check that S^N=A !

                newton(N, A, E, R, S) :-
                        M is N-1,
                        pown(M, E, [1], R, E1), % E1=E^(N-1)
                        mul1(E1,N, R, D2),      % D2=N.E^(N-1)
                        muln(E, E1,R, E2),      % E2=E^N
                        mul1(E2,M, R, N1),      % N1=(N-1).E^N
                        addn(N1,A, 0, R, N2),   % N2=(N-1).E^N+A
                        divn(N2,D2,R, F, _),    % F = {(N-1).E^N+A}div{N.E^(N-1)}
                        comn(F, E, =,  Z), !,   % F Z E
                        newton(Z, N, A, F, R, S).

                        newton(<, N, A, F, R, S) :- !, newton(N, A, F, R, S).
                        newton(=, N, A, F, R, F) :- !.
```

```
                % Take the logarithm of a rational to a rational base.
                % This can be expected to fail for almost every pair
                % of rational numbers.  To keep the search space within


   logq(B, X, R, L) is true iff
       B, X, and L are rationals such that B^L = X.
   This does its best for strange mixtures, like log(-3,-27) = 3.

ogq(B, X, R, L) :-
        absq(B, S, C),  %   B S 0 & |B| = C
        absq(X, T, Y),  %   X T 0 & |X| = Y
        logq(S, T, C, Y, R, L).

        %   absq(A, R, S, B) is true iff
        %       A and B are rationals, |A| = B, and
        %       S = {+,-,0,*} as A {<,=,>} 0 or is undefined.

        absq(number(Sa,[],[]),  *, number(+,[],[]))  :- !.
        absq(number(Sa,[],Da),  0, number(+,[],[1])) :- !.
        absq(number(Sa,Na,Da), Sa, number(+,Na,Da)).

        %   logq(S, T, ...) is just a case analysis of logq.

        logq(+, +, B, X, R, L) :- !,
                ratlog(B, X, R, L).
        logq(-, +, B, X, R, L) :- !,
                ratlog(B, X, R, L),
                mod2(L, 0).                 %  L must be "even"
        logq(-, -, B, X, R, L) :- !,
                ratlog(B, X, R, L), !,
                mod2(L, 1).                 %  L must be "odd"
        logq(+, -, _, _, _, number(+,[],[])) :- !.
        logq(*, _, _, _, _, number(+,[],[])) :- !.
        logq(_, *, _, _, _, number(+,[],[])) :- !.
        logq(0, _, _, _, _, number(+,[],[])) :- !.
        logq(_, 0, B, X, R, number(Z, N,[])) :- !,
                oneq(B, S, _),
                logq(S, Z,N).

                logq(+, -,[1]) :- !.    %  log(B,0) = -inf for 1<B<inf
                logq(-, +,[1]) :- !.    %  log(B,0) = +inf for 0<B<1
                logq(_, +,[]).          %  log(B,0) = ???? otherwise

                %  oneq(A, S, B) is true when A and B are positive
                %  defined rationals, |log A| = log B, and S = sign(log A).

                oneq(number(_, _,[]), *, number(+,[1],[])) :- !.
                oneq(number(_,Na,Na), 0, number(+,Na,Na)) :- !.
                oneq(number(_,Na,Da), +, number(+,Na,Da)) :-
                        comn(Na, Da, =, >), !.
                oneq(number(_,Na,Da), -, number(+,Da,Na)).


                %   ratlog(B, X, R, L) is true iff
                %       B, X > 0 and B^L = X.

                ratlog(B, X, R, L) :-
                        oneq(B, S, C),  %  B S 1 & |log B| = log C
                        oneq(X, T, Y),!,%  X T 1 & |log X| = log Y
```

```
                    ratlog(S, T, C, Y, R, L).

                    %  ratlog(S,T, ...) is just a case analysis

                    ratlog(+, +, B, X, R, number(+,N,D)) :- !,
                            lograt(B, X, R, N, D).
                    ratlog(+, -, B, X, R, number(-,N,D)) :- !,
                            lograt(B, X, R, N, D).
                    ratlog(-, +, B, X, R, number(-,N,D)) :- !,
                            lograt(B, X, R, N, D).
                    ratlog(-, -, B, X, R, number(+,N,D)) :- !,
                            lograt(B, X, R, N, D).
                    ratlog(0, _, _, _, _, number(+,[], [])) :- !.
                    ratlog(_, 0, _, _, _, number(+,[],[1])) :- !.
                    ratlog(+, *, _, _, _, number(+,[1],[])) :- !.
                    ratlog(-, *, _, _, _, number(-,[1],[])) :- !.
                    ratlog(_, *, _, _, _, number(+,[], [])) :- !.
                    ratlog(*, _, _, _, _, number(+,[], [])) :- !.

   lograt(B, X, R, N, D) is true iff
       B > 1, X > 1 are rationals, B^N = X^D, and gcd(N,D) = 1.

ograt(number(+,Nb,Db), number(+,Nx,Dx), R, [N], [D] ) :-
       gcdn(Db, Nx, R, U), !, U = [1],          %  Db co-prime Nx
       gcdn(Nb, Dx, R, V), !, V = [1],          %  Nb co-prime Dx
       loop(Nb, Nx, R, G), !,
       logn(G, 1, G, Nb, R, D), !,              %  D=log(G,Nb)
       logn(G, 1, G, Nx, R, N), !,              %  N=log(G,Nx)
       pown(N, Db, [1], R, K1),
       pown(D, Dx, [1], R, K2), !,
       K1 = K2.                                 %  Db^N = Dx^D

       loop(A, B, R, G) :-
               comn(A, B, =, S), !,
               loop(S, A, B, R, G).

               loop(=, A, B, R, A) :- !.
               loop(<, A, B, R, G) :-
                       divn(B, A, R, Q, X), X = [], !,
                       loop(A, Q, R, G).
               loop(>, A, B, R, G) :-
                       divn(A, B, R, Q, X), X = [], !,
                       loop(Q, B, R, G).

       %  logn(B, N, P, X, R, L) is true iff
       %       X >= B > 1, P = B^N, and X = B^L.

       logn(B, N, X, X, R, N) :- !.
       logn(B, N, P, X, R, L) :-
               comn(P, X, =, <),
               muln(B, P, R, Q),
               M is N+1, !,
               logn(B, M, Q, X, R, L).
```

```
.misc:lp.def
.earn:pps
.earn:file
.earn:weaknf
.earn:imeth
learn:cond
learn:sol
learn:comp
learn:loop
learn:confir
learn:conj
learn:table
learn:desc
learn:flag
learn:func
learn:interp
learn:sempro
learn:log
learn:homog.
learn:constr
learn:method
learn:newmet
learn:rew
l   n:time.
learn:out
learn:specia
learn:tl
lmisc:xref.ini

pack:match
method:collec
learn:polpak
learn:axioms
learn:nasty
learn:poly
learn:char
pack:poltid
```

```
/* LP.DEF :    XREF definition File for LP

                                                    Bernard Silver
                                                    Updated: 1 March 1984
*/

   % Headings
cross_ref_file('lmisc:lp.ref').
title('All files of Learning PRESS').
width(80).
globals_file(no).
update_globals(no).

   % User called predicates
called(change_method_order).
called(commutative(_)).
called(contains_nasties(_,_)).
called(d(_)).
called(dominated(_,_,_)).
called(e(_)).
called(examples).
called(explain_g(_)).
called(generate_problem).
called(give_example).
called(go).
called(help).
called(is_disjunct(_,_)).
called(is_mod_poly(_,_,_)).
called(is_product(_,_)).
called(is_sum(_,_)).
called(last_equation(_)).
called(less_occ(_,_,_)).
called(m(_)).
called(mod_iso_trace(_)).
called(old_xredo).
called(prod_exp_terms_eqn(_,_,_)).
called(r(_)).
called(remove_logs(_,_,_,_)).
called(remove_nasty(_,_,_)).
called(rhs_zero(_)).
called(s(_)).
called(same_occ(_,_,_)).
called(sredo).
called(try_attract(_,_,_)).
called(try_auto_method(_,_,_,_)).
called(try_collect(_,_,_)).
called(try_function_stripping(_,_,_,_)).
called(try_to_isolate(_,_,_)).
called(try_poly(_,_,_)).
called(try_prep_fact(_,_,_)).
called(try_user_rule(_,_,_,_)).
called(w(_)).
called(work(_)).
called(work_solution).

   % Known predicates.

known(copy_ground(_,_,_), utility).
known(file_exists,utility).
known(give_help(_),'MDHELP.PL').
```

```
known(occurs_in(_,_),utility).
known(portray_number(_),utility).
known(simple(_),utility).
known(tidy_expr(_,_),utility).
known(tidy_withvars(_,_),utility).

 % Applies
applies(process_reply(_,B,_,_,_),B).
applies(process_reply(_,_,_,_,E),E).
applies(user_macro(A,_,_),A).
applies(user_macro(_,_,C),C).
```

```
: XREF.INI : Initializations for the benefit of XREF.
      Can also be used to define things for unknown(_,trace).

                                                  Bernard Silver
                                                  Updated: 17 February 1984

; Default definitions for stored things

iserted(_) :- fail.

ito_rule(_,_) :- fail.

ifficult_once(_,_) :- fail.

itegral(_) :- fail.

ist_created(_) :- fail.

iown_method_auto(_,_,_,_,_,_,_) :- fail.

iown_method_schema(_,_,_,_,_,_,_,_,_) :- fail.

   equation(_) :- fail.

ist_example(_,_) :- fail.

ew_functor(_,_) :- fail.

ew_rule_stored(_) :- fail.

ule(_,_,_,_) :- fail.

chema(_) :- fail.

chema_used(_) :- fail.

een_eqn(_) :- fail.

tore_we(_) :- fail.

nknown(_) :- fail.

s_rule(_,_,_,_,_,_,_) :- fail.
```

```
 LP.MIC  -  Load Learning Press   *<SILENCE>

                                Updated: 9 February 1984
       This junk allows for automatic loading believe it or not

       Call as:         /LP              - to load lp (normal use)
                        /LP auto         - used by MAKSYS

on error:backto death
error ?
on operator:backto death
operator !
goto cont
eath::
^C
^C
if ($a = "auto") .let e1 = "error"
 LP.MIC  HALTED
mic return
ont::
let y = $date.["-",20], d = $date.[1,"-"]+" "+$y.[1,"-"]+" "+$y.["-",4]
if ($d.[1] = "0") .let d = $d.[2,20]

 lputil[400,444]  *<revive>
 :- ['*learn:filin*'].
 :- version('*LP Algebra System  Mark 4 (*d)
Copyright (C) 1984 Dept. Artificial Intelligence. Edinburgh*').
 :- asserta( version_date('**d*') ).
 :- ok.
save dskb:lp[400,444]
```

```
 LPUTIL.MIC  -  Load Learning Press  Util *<silence>
                                        Updated: 9 February 1984
       This junk allows for automatic loading believe it or not

       Call as:        /LPUTIL - to load LPUTIL (normal use)
                       /LPUTIL auto    - used by MAKSYS

on error:backto death
error ?
on operator:backto death
operator !
goto cont
eath::
^C
^C
if ($a = "auto") .let e1 = "error"
 LPUTIL.MIC  HALTED
mic return
ont::
let y = $date.["-",20], d = $date.[1,"-"]+" "+$y.[1,"-"]+" "+$y.["-",4]
if ($d.[1] = "0") .let d = $d.[2,20]

    prolog[400,444]  *<revive>
    [**learn:filuti**].
 :- version(**LP UTIL System  (*d)
Copyright (C) 1984 Dept. Artificial Intelligence. Edinburgh**).
 :- core_image,
    display(**LP Utilities Package  (*d)**), ttynl,
    reinitialise.

save lputil[400,444]
```

```
%% BASIC ARITHMÉTIC OVER LONG INTEGERS %%

                    % Addition of two long integers

iddz(+,A, +,B, R, +,C) :- !, addn(A, B, 0, R, C).
iddz(+,A, -,B, R, S,C) :- !, subn(A, B, R, S, C).
iddz(-,A, +,B, R, S,C) :- !, subn(B, A, R, S, C).
iddz(-,A, -,B, R, -,C) :- !, addn(B, A, 0, R, C).

        addn([D1|T1], [D2|T2], Cin, R, [D3|T3]) :-
                Sum is D1+D2+Cin,
                (   !(Sum) >= R, Cout = 1, D3 is !(Sum)-R
                ;   !(Sum) <  R, Cout = 0, D3 =  Sum
                ),  !,
                addn(T1, T2, Cout, R, T3).
        addn([], L, 0, R, L) :- !.
        addn([], L, 1, R, M) :- !, add1(L, R, M).
        addn(L, [], 0, R, L) :- !.
        addn(L, [], 1, R, M) :- !, add1(L, R, M).

                add1([M|T], R, [N|T]) :- N is M+1, N < R, !.
                add1([M|T], R, [0|S]) :- R is M+1, !, add1(T, R, S).
                add1([],    R, [1]).

                    % Comparison of two long integers

omz(_,[],_,[],S) :- !, S = '='.         % -0 = 0 now, alas.
omz(+,A, +,B, S) :- !, comn(A, B, =, S).
omz(+,A, -,B, >).
omz(-,A, +,B, <).
omz(-,A, -,B, S) :- !, comn(B, A, =, S).

        comn([D1|T1], [D2|T2], D, S) :-
                com1(D1, D2, D, N), !,
                comn(T1, T2, N, S).
        comn([],      [],      D, S) :- !, S = D.
        comn([],      L,       D, <) :- !.
        comn(L,       [],      D, >) :- !.

                com1(X, X, D, D) :- !.
                com1(X, Y, D, <) :- X < Y, !.
                com1(X, Y, D, >) :- X > Y, !.


                    % Subtraction of two long integers

ubz(+,A, +,B, R, S,C) :- !, subn(A, B, R, S, C).
ubz(+,A, -,B, R, +,C) :- !, addn(A, B, 0, R, C).
ubz(-,A, +,B, R, -,C) :- !, addn(B, A, 0, R, C).
ubz(-,A, -,B, R, S,C) :- !, subn(B, A, R, S, C).

        subn(A, B, R, S, C) :-
                comn(A, B, =, 0), !,  %  Oh for Ordering
                subn(0, A, B, R, S, C).

                subn(<, A, B, R, -, C) :- !, subp(B, A, 0, R, D), prune(D, C).
```

```
subn(>, A, B, R, +, C) :- !, subp(A, B, 0, R, D), prune(D, C).
subn(=, A, B, R, +,[]) :- !.

        prune([0|L], M ) :- !,
                prune(L, T),
                (T = [], M = []; M = [0|T]).
        prune([D|L], [D|M]) :- !,
                prune(L, M).
        prune([],    []) :- !.

subp([D1|T1], [D2|T2], Bin, R, [D3|T3]) :-
        S is D1-D2-Bin,
        (   S >= 0, Bout = 0, D3 =  S
        ;   S <  0, Bout = 1, D3 is S+R
        ),  !,
        subp(T1, T2, Bout, R, T3).
subp(L, [], 0, R, L) :- !.
subp(L, [], 1, R, M) :- !, sub1(L, R, M).

        sub1([0|T], R, [K|S]) :- !, K is R-1, sub1(T, R, S).
        sub1([N|T], R, [M|T]) :- M is N-1.



        % Multiplication of Signs

sign(S, S, +) :- !.
sign(S, T, -) :- !.



        % Multiplication of two long integers

/*
mulz(S,A, T,B, R, U,C) :-
        sign(S, T, U), !,
        muln(A, B, R, C).
*/
        muln([], B, R, []) :- !.
        muln(A, [], R, []) :- !.
        muln(A,  B, R,  C) :- !, muln(A, B, [], R, C).

        muln([D1|T1], N2, Ac, R, [D3|Pr]) :-
                mul1(N2, D1, R, P2),
                addn(Ac, P2, 0, R, Sm),
                conn(D3, An, Sm), !,
                muln(T1, N2, An,R, Pr).
        muln([],      N2, Ac, R, Ac) :- !.

                mul1(A, 0, R, []) :- !.
                mul1(A, M, R, Pr) :- !,
                        mul1(A, M, 0, R, Pr).

                        mul1([], M, 0, R, []) :- !.
                        mul1([], M, C, R, [C]) :- !.
                        mul1([D1|T1], M, C, R, [D2|T2]) :-
                                D2 is (D1*M+C) mod R,
                                Co is (D1*M+C)  /  R,
                                mul1(T1, M, Co, R, T2).
```

```
                        % Exponentiation of a long integer to a short
                        %  (Prolog) integer. Note that this means the
                        %  power must be less than 100000 (current radix).
                        %  This code should always be called with positive
                        %  powers.

)owz(-,A, N, R, -,C) :-
        N mod 2 =:= 1, !,
        pown(N, A, [1], R, C).
)owz(S,A, N, R, +,C) :- !,
        pown(N, A, [1], R, C).

        pown(0, A, M, R, M) :- !.
        pown(1, A, M, R, P) :- !,
                muln(A, M, R, P).
        pown(N, A, M, R, P) :-
                N1 is N/2,
                (   N mod 2 =:= 0, M1 = M
                ;   N mod 2 =:= 1, muln(A, M, R, M1)
                ),
                muln(A, A, R, A1), !,
                pown(N1, A1, M1, R, P).
```

```
                    % Division of two long integers

divz(S,A, T,B, R, U,Q, S,X) :-
        sign(S, T, U), !,
        divn(A, B, R, Q, X).

        divn(A, [], R, _, _) :- !, fail. % division by 0 is undefined
        divn(A,[1], R, A,[]) :- !.       % a very common special case
        divn(A,[B], R, Q, X) :- !,       % nearly as common a case
                div1(A, B, R, Q, Y),
                conn(Y, [], X).
        divn(A,  B, R, Q, X) :-
                comn(A, B, =, S),
                (   S = '<', Q =  [], X = A
                ;   S = '=', Q = [1], X = []
                ), !.
        divn(A,  B, R, Q, X) :- !,
                divm(A, B, R, Q, X).

                conn(0, [],   []) :- !.
                conn(D, T, [D|T]).

                div1([D1|T1], B1, R, Q1, X1) :- !,
                        div1(T1, B1, R, Q2, X2),
                        D2 is (X2*R+D1)  /  B1,
                        X1 is (X2*R+D1) mod B1,
                        conn(D2, Q2, Q1).
                div1([],      B1, R, [],  0).

% divm(A, B, R, Q, X) is called with A > B > R

                divm([D1|T1], B, R, Q1, X1) :- !,
                        divm(T1, B, R, Q2, X2),
                        conn(D1, X2, T2),
                        div2(T2, B, R, D2, X1),
                        conn(D2, Q2, Q1).
                divm([],      B, R, [], []).

                        div2(A, B, R, Q, X) :-
                                estd(A, B, R, E), !,
                                chkd(A, B, R, E, 0, Q, P), !,
                                subn(A, P, R, S, X).   %  S=+
                        div2(A, B, R, _, _) :-
                                long_error(divq, A/B).

                                estd([A0,A1,A2], [B0,B1], R, E) :-
                                        B1 >= R/2, !,
                                        E is (A2*R+A1)/B1.
                                estd([A0,A1,A2], [B0,B1], R, E) :- !,
                                        L is (A2*R+A1)/(B1+1),
                                        mul1([B0,B1],    L, R, P),
                                        subn([A0,A1,A2], P, R, S, N), !, %S=+
                                        estd(N, [B0,B1], R, M),    !,
                                        E is L+M.
                                estd([A0,A1],    [B0,B1], R, E) :- !,
                                        E is (A1*R+A0+1)/(B1*R+B0).
                                estd([A0],       _,       R, 0) :- !.
                                estd([A0|Ar],    [B0|Br], R, E) :- !,
                                        estd(Ar, Br, R, E).
                                estd([],         _,       R, 0) :- !.
```

```
chkd(A, B, R, E, 3, _, _) :-        !,
        long_error(divq, A/B).
chkd(A, B, R, E, K, E, P) :-
        mul1(B, E, R, P),
        comn(P, A, <, <), !.
chkd(A, B, R, E, K, Q, P) :-
        L is K+1, F is E-1, !,
        chkd(A, B, R, F, L, Q, P).


                % GCD of two long integers

/*
gcdz(S,A, T,B, R, D, S,M, T,N) :- !,
        gcdn(A, B, R, D, M, N).
*/
        gcdn([], [], R, [1],  [],  []) :- !.
        gcdn([],  B, R,   B,  [], [1]) :- !.
        gcdn( A, [], R,   A, [1],  []) :- !.
        gcdn([1], B, R, [1], [1],   B) :- !.    %  common case
        gcdn( A,[1], R, [1],   A, [1]) :- !.    %  common case
        gcdn( A,  B, R,   D,   M,   N) :-       %  A, B > 1
                gcdn(A, B, R, D),
                divn(A, D, R, M, _),
                divn(B, D, R, N, _).

                gcdn(A, B, R, D) :-             %  A, B >= 1  !!
                        comn(A, B, =, S), !,
                        gcdn(S, A, B, R, D).

                        gcdn(<,[], B, R, B) :- !.
                        gcdn(<, A, B, R, D) :-
                                estg(B, A, R, E),
                                muln(E, A, R, P),
                                subn(B, P, R, _, M), !,
                                gcdn(A, M, R, D).
                        gcdn(>, A,[], R, A) :- !.
                        gcdn(>, A, B, R, D) :-
                                estg(A, B, R, E),
                                muln(E, B, R, P),
                                subn(A, P, R, _, M), !,
                                gcdn(M, B, R, D).
                        gcdn(=, A, B, R, A).

                                estg(    A,   [B], R, E) :- !,
                                        div1(A, B, R, Q, X),
                                        (   X*2 =< B, E = Q
                                        ;   add1(Q, R, E)
                                        ), !.
                                estg([_|A], [_|B], R, E) :- !,
                                        estg(A, B, R, E).
```

```
%% TRIGONOMETRIC EVALUATION %%

        % This stuff needs some work done on it, and the mode
        % declarations haven't been written yet.  Taihoa.
        % To do:
        %       Since at this stage all the argumentss are known to be
        %       numbers we shouldn't waste time using the general eval.
        %       Approximations should be used so that the routines work
        %       for ANY argument.  Care is needed, since little is known
        %       about rational approximations, lest the numbers explode.



sineval(X, S) :-
        eval(X < 0),    !,
        eva2(-X, Y),
        sineval(Y, T),
        eva2(-T, S).
sineval(X, S) :-
        eval(X > 90),   !,
        eva2(180-X, Y),
        sineval(Y, S).
sineval(X, S) :-        % 0 <= X <= 90
        sineval1(X, S).


        sineval1(number(+,[],[1]),   number(+,[],[1])).
        sineval1(number(+,[30],[1]), number(+,[1],[2])).
        sineval1(number(+,[45],[1]), number(+,[99],[140])).
        sineval1(number(+,[60],[1]), number(+,[45],[52])).
        sineval1(number(+,[90],[1]), number(+,[1],[1])).



coseval(X, C) :-
        eva2(90-X, Y), !,
        sineval(Y, C).


taneval(X, T) :-
        sineval(X, S),
        coseval(X, C), !,
        eva2(S/C, T).


arcsineval(S, X) :-
        eval(S >= 0), !,
        sineval1(X, S).
arcsineval(S, X) :-
        eval(S < 0),
        eva2(-S, T), !,
        sineval1(Y, T),
        eva2(-Y, X).


arccoseval(C, X) :-
        arcsineval(C, Y), !,
        eva2(90-Y, X).
```

```
rctaneval(number(S,N,D), number(S,M,C)) :-
        arctaneval(N, D, M, C).

rctaneval([],X, [], X) :- !.            %  arctan(0) = 0, arctan(undef) = undef
rctaneval(X, X, [45], [1]) :- !.        %  arctan(1) = 45°
rctaneval(X,[], [90], [1]) :- !.        %  arctan(inf) = 90°
rctaneval(N, D, M, C) :-
        R = 100000,                     %  the common radix
        muln(N, N, R, Nsq),
        muln(D, D, R, Dsq),
        addn(Nsq, Dsq, 0, R, Sq), !,
        nthn(2, Sq, R, Den), !,
        arcsineval(number(+,N,Den), S),
        S = number(+,M,C).



% ERROR HANDLING %%

ong_error(Culprit, Expression) :-
        long_error_message(Culprit, Message),
        display('** '), display(Message), display(': '),
        print(Expression), ttynl,
        break, fail.

ong_error_message(eval, 'EVAL given a variable').
ong_error_message(eva2, 'EVAL given an expression containing a variable').
ong_error_message(divq, 'Unexpected rational division problem').
```

```
lp.sub              ;; this file
wep.
 lunc.
 ortr.
loop.
misce.pl
routin.pl
struct.pl
nasty.
polpak.
mdhelp.pl
ops.
 oly.
simp.ax
axioms.
filuti.
file.
time.
los.
 omos.
conj.
rew.
  '.
   nprb.
confir.
desc.
flgs.
interp.
method.
comp.
table.
char.
newmet.
cond.
imeth.
weaknf.
specia.
constr.
sol.
 ut.
 lilin.
 e.ccl              ;;
  .def              ;;
 ref.ini            ;;
lp.bus              ;;
lp.cns              ;;
```

```
%    File    : LP.CNG
%    Author  : Bernard Silver
%    Updated: 6 March 1984
%    Purpose: List changes to LP
%    Started 24 Februarv 1984 for Mark 4 version

LP mark 1 is described in Notes 123 and 124
LP mark 2 had some planning capability, and is described in Research Paper 184
LP mark 3 produced macros and is described in Research Paper 188
LP mark 4 is described in my thesis

24 February 1984
Corrected syntax errors in various files, corrected bad definition of
dis_solution, added more entries to TABLE.

26 February 1984
Changed various files, added information about failing methods to SOL
mod_check_cond, changed CONFIR and CHAR to allow rule(Name) input and abort
and break as well.  Also changed OUT and FLAG so informative prompts are
issued about the unknown and list of conditions.

26 February 1984
Changed typo in CONSTR that was causing problems, added clause to
  _solution, so it reported if no solution had been stored.
 GPORTR back in LP to overcome bracket problems, added extra clause
o that # prints as v.  New file called PORTR

 March 1984
dded Homogenization to LP, in file HOMOG.  Doesn't handle exponentials or
yperbolics, the former interact with tidy, the latter aren't supported
lsewhere.  This involved changing several files, TABLE, FLAG, METHOD, TL, SOL
ILIN, LP.SUB, LP.CCL, LP.DEF and possibly others.

hange of Unknown solutions sometimes cause duplicate steps in the trace.
 don't seem to be able to fix this, so as a hack, store_steps now does
 meptidy_example before storing.  The latter has been changed so that
t won't print out warning messages in this case.  This involved changing OUT,
NTERP and SOL.
```

File : LP.STS
Author : Bernard Silver
Updated: 28 February 1984
Purpose: Clause count for LP

LPUTIL

il:util.ops	0 clauses	0 predicates.
il:arith.ops	0 clauses	0 predicates.
arn:ops.	0 clauses	0 predicates.
il:files.pl	7 clauses	6 predicates.
il:writef.pl	80 clauses	25 predicates.
arn:routin.ol	39 clauses	20 predicates.
il:flagro.pl	3 clauses	3 predicates.
il:struct.pl	35 clauses	16 predicates.
il:long.pl	260 clauses	83 predicates.
il:tidy.pl	110 clauses	23 predicates.
arn:misce.pl	13 clauses	10 predicates.
:edit.pl	15 clauses	9 predicates.
il:type.pl	4 clauses	2 predicates.
il:trysee.pl	22 clauses	10 predicates.
ol	589 clauses	208 predicates.

Rest of LP

kag:match.	26 clauses	9 predicates.
rn:portr.	29 clauses	9 predicates.
hod:collec.	28 clauses	13 predicates.
rn:weaknf.	16 clauses	6 predicates.
rn:cond.	59 clauses	32 predicates.
rn:char.	37 clauses	22 predicates.
rn:simp.ax	14 clauses	2 predicates.
rn:rew.	37 clauses	14 predicates.
rn:file.	12 clauses	7 predicates.
rn:time.	8 clauses	6 predicates.
rn:comp.	86 clauses	39 predicates.
rn:polpak.	53 clauses	19 predicates.
rn:tl.	67 clauses	36 predicates.
rn:out.	164 clauses	77 predicates.
:conj.	62 clauses	27 predicates.
:desc.	20 clauses	10 predicates.
:loop.	19 clauses	10 predicates.
rn:sol.	115 clauses	56 predicates.
rn:imeth.	37 clauses	18 predicates.
rn:confir.	43 clauses	21 predicates.
rn:flag.	76 clauses	35 predicates.
rn:method.	55 clauses	17 predicates.
rn:specia.	14 clauses	7 predicates.
rn:table.	37 clauses	7 predicates.
rn:constr.	27 clauses	8 predicates.
rn:func.	40 clauses	11 predicates.
rn:newmet.	64 clauses	28 predicates.
rn:interp.	71 clauses	40 predicates.
rn:genprb.	44 clauses	20 predicates.
rn:nasty.	111 clauses	50 predicates.
rn:axioms.	72 clauses	3 predicates.
rn:poly.	31 clauses	16 predicates.

```
arn:log.                32 clauses  16 predicates.
ckag:poltiq.             8 clauses   5 predicates.
=================================================
tal                   1616 clauses 698 predicates.
=================================================
=================================================
and total:            2204 clauses 906 predicates.
=================================================
```