```
/* cubic.
Read in files for specializing general cubic to z form
Alan Bundy 31.3.81 */
```

use Pres.

```
:- [
-poly,                  % new method called from poly_method
-int,                   % non_zero patch
equate,                 % new method
probs                   % call of general cubic
].
```

include gik

```
/* equate
Specialize general polynomial
Alan Bundy 31.3.81 */


/* specialize general polynomial by removing coefficients */
/*-----------------------------------------------------------*/

specialize(GenBas,X,SpecBas,Y,NewCoeffs) :-
        degree(GenBas,N), N1 is N-1,
        spec(N,N1,GenBas,SpecBas,NewCoeffs),
        sensym(y,Y).


/* specialize bas */

spec(N,N1,[],[],[]) :- !.       % basis

spec(N,N1,[pair(N,C)|GBas],[pair(N,1)|SBas],NCs) :- !,  % leading coeff set to
        spec(N,N1,GBas,SBas,NCs).

spec(N,N1,[pair(N1,C)|GBas],SBas,NCs) :- !,             % 2nd term dropped
        spec(N,N1,GBas,SBas,NCs).

spec(N,N1,[pair(M,GC)|GBas],[pair(M,SC)|SBas],[SC|NCs]) :- !,
        sensym(c,SC),                                   % other terms retained
        spec(N,N1,GBas,SBas,NCs).


/* Degree of Polynomial */

degree([],0) :- !.              % basis

degree([pair(N1,C)|Bas],N) :- !,                % step
        degree(Bas,N2), max(N1,N2,N).


/* maximum of two numbers */

max(N1,N2,N1) :-
        N1 >= N2, !.

max(N1,N2,N2).



/* Equate coefficients of two polynomials */
/*----------------------------------------*/

equate_coeffs(SNF,SUnk,TNF,TUnk,Subst,Eqns) :-
        make_poly(TUnk,TNF,TEqn),
        subst_mess(Subst,TEqn,TEqn1),
        poly_norm(SUnk,TEqn1,TNF1),
        pair_off(SNF,TNF1,Eqns).


/* Form equations by equating coefficients */

pair_off([],[],[]) :- !.                % both empty
```

```
pair_off([],TNF,Eqns) :- !,                    % SNF empty
        maplist(eq_zero,TNF,Eqns).

pair_off([pair(N,SC)|SNF1], TNF, [SC=TC|Eqns]) :-
        select(pair(N,TC),TNF,TNF1), !,                    % pair off
        pair_off(SNF1,TNF1,Eqns).                          % recurse

pair_off([pair(N,SC)|SNF], TNF, [SC=0|Eqns]) :- !,         % no pair
        pair_off(SNF,TNF,Eqns).                            % recurse

/* set coefficient equal to zero */
eq_zero(pair(N,TC), TC=0).
```

```
/*                POLY              File 9  19.2.81 */
/*****************************************/
/* ROUTINES FOR POLYNOMIAL EQUATIONS */
/*****************************************/

/* Identities and unsatisfiable equations */
/*-----------------------------------------*/

poly_method( _, [], true) :- !.
poly_method( _, [pair(0,A)], false) :- !.


/* Linear Equation */
/*-----------------*/

poly_method(X,Plist, X=Ans) :-
        subset(Plist,[pair(1,A),pair(0,B)]),
        not var(A), !,
        default(B),
        tidy( (-1)*B*A^ -1, Ans).

/* Quadratic Equation */
/*--------------------*/

poly_method(X,Plist, X=Ans1 # X=Ans2) :-
        subset(Plist,[pair(2,A),pair(1,B),pair(0,C)]),
        not var(A), !,
        default(B), default(C),
        tidy((B^2+ -4*A*C)^(2^ -1),Sqrt),
        tidy((-1*B+Sqrt)*(2*A)^ -1,Ans1),
        tidy((-1*B+(-1*Sqrt))*(2*A)^ -1,Ans2).



/* Default unbound coefficient to zero */

default(C) :- var(C), !, C=0.
default(C) :- !.



/* Polynomial with Negative Powers */
/*---------------------------------*/

poly_method(X,Plist,Ans) :-
        member(pair(N,_),Plist), N<0, !,
        N1 is -N,
        trace('Multiply through by %t to get a polynomial\n',[X^N1],1),
        maplist(add_n(N1),Plist,NPlist),
        poly_method(X,NPlist,Ans).



/* Multiply through pairs list by N */

add_n(N,pair(M,Coeff),pair(MN,Coeff)) :- !,
        MN is M+N.
```

```
/* Solve general equation by eliminating coefficients */
/*-----------------------------------------------------------------*/

poly_method(X,GenBas,Ans) :-
        checklist(atom_coeff,GenBas),
        specialize(GenBas,X,SpecBas,Y,NewCoeffs),
        gensym(a,A), gensym(b,B),
        equate_coeffs(GenBas,X,SpecBas,Y,Y=A*X+B,EqnList),
        dottoand(EqnList,Eqns),
        simsolve(Eqns,[A,B|NewCoeffs],Solns),
        make_poly(Y,SpecBas,SpecPoly),
        solve(SpecPoly,Y,Ans1),
        subst_mess(Y=A*X+B & Solns, Ans1, Ans2),
        solve(Ans2,X,Ans).

/* coefficient is an atom */
atom_coeff(pair(N,C)) :- atom(C).
```

```
/*                POLYIS          19.2.81  */
/*****************************
        POLYNOMIAL NORMAL FORM
*****************************/


/* Use polynomial form for simplification (always succeeds) */

poly_form(true,true).
poly_form(false,false).

poly_form(Exp,Poly) :- !,
        poly_form1(Exp,New),
        tidy(New,Poly).

/* Look for terms to simplify */

poly_form1(Exp,Poly) :-
        Exp=..[Sym|Args], ispred(Sym), !,
        maplist(poly_form1,Args,PArgs),
        Poly=..[Sym|PArgs].

/* Apply to term */

poly_form1(Exp,Poly) :- !,
        wordsin(Exp,Vars),
        sublist(mult_occ(Exp),Vars,Vars1),
        poly_form(Vars1,Exp,Poly).



/* Test for predicate or logical connective */

ispred(&).      ispred($).      ispred(=).
ispred(>).      ispred(>=).     ispred(<).      ispred(=<).



/* Put term in polynomial normal form with respect to list of variables*/

poly_form([],Exp,Exp) :- !.

poly_form([Var|Vars],Exp,Poly) :- !,
        poly(Var,Exp,Ebas1,simp),
        maplist(half_poly(Vars),Ebas1,Ebas2),
        make_poly(Var,Ebas2,Poly).


/* Apply poly_form to coeffs */

half_poly(Vars,pair(N,E1), pair(N,E2)) :- !,
        poly_form(Vars,E1,E2).


/* Put polynomials in normal form (succeeds only for polynomials) */

poly_norm(X,Poly,Pbas1) :- !,
        poly(X,Poly,Pbas,poly),
        maplist(poly_form_coeff,Pbas,Pbas1).
```

```
/* Tidy coefficients */
poly_form_coeff(pair(N,E),pair(N,E1)) :- poly_form(E,E1).


/* Forms bas of coefficients */

poly(X,X,[pair(1,1)],flas) :- !.

poly(X,X^N,[pair(N,1)],poly) :-
        inteser(N), !.

poly(X,X^N,[pair(N,1)],simp) :-
        inteser(N), !.

poly(X,(X^N)^(-1),[pair(N1,1)],flas) :-
        inteser(N), !,
        N1 is -N.

poly(X,E,[pair(0,E)],flas) :-
        freeof(X,E), !.

poly(X,S+T,Ebas,flas) :-!,
        poly(X,S,Sbas,flas), poly(X,T,Tbas,flas),
        add(Sbas,Tbas,Ebas).

poly(X,S*T,Ebas,flas) :- !,
        poly(X,S,Sbas,flas), poly(X,T,Tbas,flas),
        times(Sbas,Tbas,Ebas).

poly(X,S^N,Ebas,flas) :-
        inteser(N),N > 0, !,
        poly(X,S,Sbas,flas),
        binomial(Sbas,N,Ebas).

poly(X,E,[pair(0,E1)],simp) :- !,
        E=..[Sym|Arss],
        maplist(poly_form1,Arss,Arss1),
        E1=..[Sym|Arss1].

/* Add two coefficients bass - code in POLPAK */

/* Multiply two coefficient bass - code in POLPAK */

/* Binomial expansion of coefficient bas */


binomial(Bas, 0, [pair(0,1)]) :- !.

binomial(Bas, 1, Bas) :- !.

binomial(Sbas, N, Ebas) :- !,
        N1 is N-1,
        binomial(Sbas,N1,Ebas1),
        times(Sbas,Ebas1,Ebas).


/* Reconstitute bas of coefficients into polynomial */

make_poly(X,Bas1,Poly) :- !,
        maplist(reify(X),Bas1,Bas2),
```

```
        recomp(Poly,[T|Bas2]).

/* reify coefficient and power into product */

reify(X,pair(0,E),E) :- !.
reify(X,pair(1,E),E*X) :- !.
reify(X,pair(N,E),E*X^N) :- !.
```

```
/* probs.
CURRENT PROBLEMS*/

/* interval and eval problems */

bbb(I) :- find_int(11/(-m1),I).

abb :- zeute(x*2).
quadratic(A) :- solve(a*x^2 + b*x + c = 0, x, A).

cubic(A) :- solve(a*x^3 + b*x^2 + c*x + d = 0, x, A).

quartic(A) :- solve(a*x^4 + b*x^3 + c*x^2 + d*x + e = 0, x, A).

non_zero(a).
```

```
  QQQQQQ     UU        UU     AAAAAA     DDDDDDDD                  SSSSSSSS     000000     LL
  QQQQQQ     UU        UU     AAAAAA     DDDDDDDD                  SSSSSSSS     000000     LL
QQ       QQ  UU        UU  AA      AA   DD      DD                SS           00    00    LL
QQ       QQ  UU        UU  AA      AA   DD      DD                SS           00    00    LL
QQ       QQ  UU        UU  AA      AA   DD      DD                SS           00    00    LL
QQ       QQ  UU        UU  AA      AA   DD      DD                SS           00    00    LL
QQ       QQ  UU        UU  AA      AA   DD      DD                  SSSSSS     00    00    LL
QQ       QQ  UU        UU  AA      AA   DD      DD                  SSSSSS     00    00    LL
QQ  QQ   QQ  UU        UU  AAAAAAAAAA   DD      DD                        SS   00    00    LL
QQ  QQ   QQ  UU        UU  AAAAAAAAAA   DD      DD                        SS   00    00    LL
QQ     QQ    UU        UU  AA      AA   DD      DD        ....            SS   00    00    LL
QQ     QQ    UU        UU  AA      AA   DD      DD        ....            SS   00    00    LL
  QQQQ  QQ   UUUUUUUUUU  AA      AA   DDDDDDDD          ....    SSSSSSSS     000000     LLLLLLLLLL
  QQQQ  QQ   UUUUUUUUUU  AA      AA   DDDDDDDD          ....    SSSSSSSS     000000     LLLLLLLLLL


44    44       000000       000000                  44    44       000000    555555555
44    44       000000       000000                  44    44       000000    555555555
44    44    00       00  00       00                44    44    00       00  55
44    44    00       00  00       00                44    44    00       00  55
44    44    00     0000  00     0000                44    44    00     0000  555555
44    44    00     0000  00     0000                44    44    00     0000  555555
4444444444  00  00  00  00  00  00                 4444444444  00  00  00        55
4444444444  00  00  00  00  00  00                 4444444444  00  00  00        55
      44    0000    00  0000    00                       44    0000    00          55
      44    0000    00  0000    00                       44    0000    00          55
      44    00      00  00      00      ,,,,             44    00      00  55      55
      44    00      00  00      00      ,,,,             44    00      00  55      55
      44      000000      000000          ,,             44      000000      555555
      44      000000      000000          ,,             44      000000      555555


BBBB  U    U N    N DDDD  Y    Y                H   H PPPP   SSSS
B   B U    U N    N D   D Y    Y                H   H P   P S
B   B U    U NN   N D    D  Y Y                 H   H P   P S
BBBB  U    U N N  N D    D   Y                  HHHHH PPPP   SSS
B   B U    U N  NN D    D   Y                  H   H P         S
B   B U    U N    N D   D    Y                  H   H P         S
BBBB  UUUUU N    N DDDD     Y                  H   H P     SSSS
```

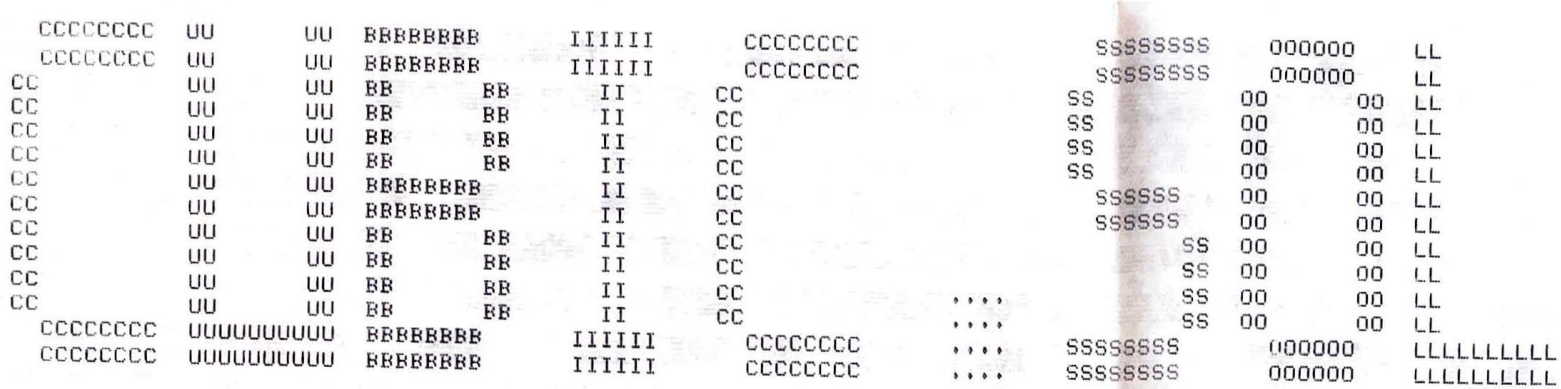
START User BUNDY HPS [400,405] Job QUAD Seq. 6472 Date 20-May-81 11:59:54 Monitor ERCC ICF DEC10 7.01(047) *START*
File: DSKA:QUAD.SOL<005>[400,405,MYPRES,CUBIC,SPEC] Created: 20-May-81 11:58:33 Printed: 20-May-81 12:00:10
QUEUE Switches: /FILE:ASCII /COPIES:1 /SPACING:1 /LIMIT:56 /FORMS:NORMAL

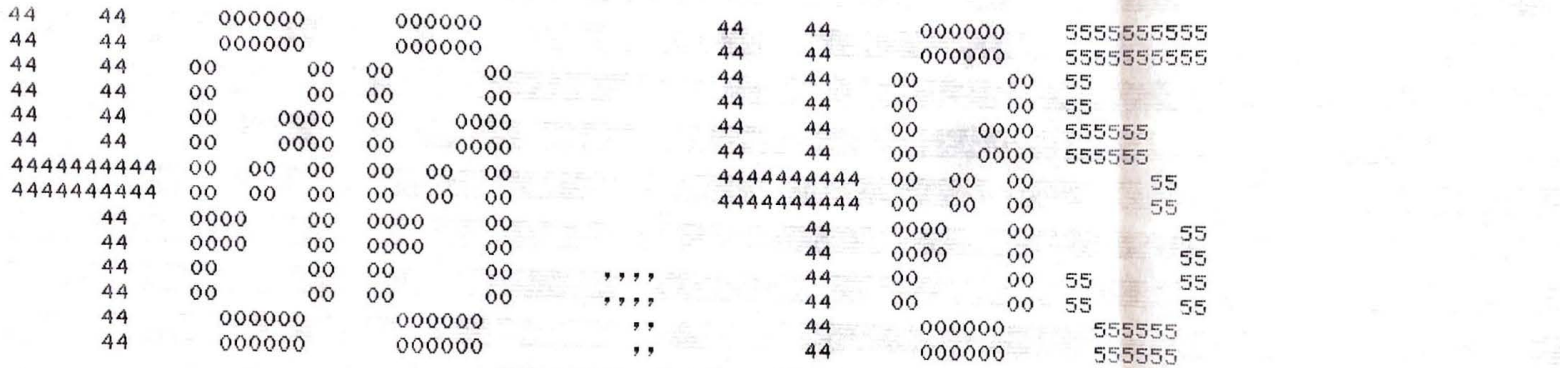
```
PRESS - PRolog Equation Solving System
Dept. Artificial Intelligence, University of Edinburgh
    Version of 19 may 81

Prolog-10  version 3.2
Copyright (C) 1981 by D. Warren, F. Pereira and L. Byrd

| ?- [spec].

poly reconsulted   390 words      0.24 sec.

simeq reconsulted   326 words      0.25 sec.

equate consulted   552 words      0.23 sec.

probs consulted   168 words      0.06 sec.

spec consulted   1448 words      0.90 sec.

yes
| ?- quadratic(A).
Solving a * x ^ 2 + b * x + c = 0 for x
Trying to specialize polynomial to eliminate x^1
term and reduce coefficient of x^2 to 1

Formed equation c1 + 1 * y1 ^ 2

Applying substitution
    y1 = a1 * x + b1

  to    :
    c1 + 1 * y1 ^ 2

  gives :
    c1 + (a1 * x + b1) ^ 2

Equating coefficients of x

Simultaneously solving :
    a = a1 ^ 2
    b = a1 * b1 * 2
    c = c1 + b1 ^ 2
For [a1, b1, c1].
Solving a = a1 ^ 2 for a1
I assume a1 positive.

    a1 = a ^ (1/2)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  a1 = a ^ (1/2)

Applying substitution
    a1 = a ^ (1/2)

  to    :
    b = a1 * b1 * 2
    c = c1 + b1 ^ 2
```

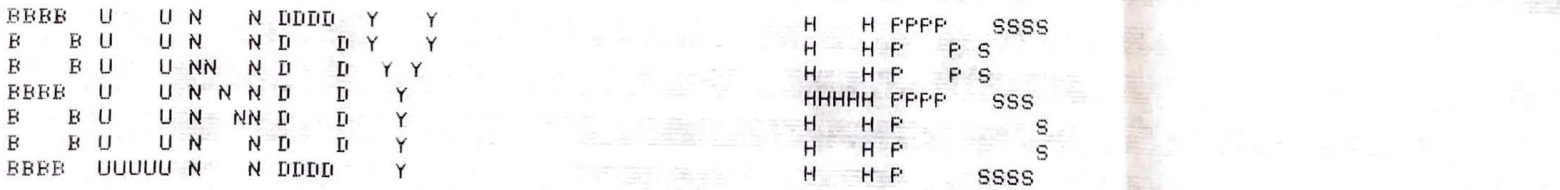
```
  gives :
    b = a ^ (1/2) * b1 * 2
    c = c1 + b1 ^ 2

Solving b = a ^ (1/2) * b1 * 2 for b1

    a ^ (1/2) * b1 = b * (1/2)
        (by Isolation)
I assume a positive.

    b1 = b * (1/2) * a ^ (-1/2)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  b1 = b * a ^ (-1/2) * (1/2)

Applying substitution
    b1 = b * a ^ (-1/2) * (1/2)

  to    :
    c = c1 + b1 ^ 2

  gives :
    c = c1 + (b * a ^ (-1/2) * (1/2)) ^ 2

Solving c = c1 + (b * a ^ (-1/2) * (1/2)) ^ 2 for c1

    c1 = c + -1 * (b * a ^ (-1/2) * (1/2)) ^ 2
        (by Isolation)

Answer is :
X1
  where :
    X1 =  c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1

Applying substitution
    c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1

  to    :

  gives :

Substituting back in c1 solution
Applying substitution

  to    :
    c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1

  gives :
    c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1

Substituting back in b1 solution
Applying substitution
    true
    c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1

  to    :
```

```
    b1 = b * a ^ (-1/2) * (1/2)

  gives :
    b1 = b * a ^ (-1/2) * (1/2)

Substituting back in a1 solution
Applying substitution
    true & c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1
    b1 = b * a ^ (-1/2) * (1/2)

  to    :
    a1 = a ^ (1/2)

  gives :
    a1 = a ^ (1/2)


Final Answers are :
( ( X1 & X2 ) & X3 )
  where :
    X1 =  c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1
    X2 =  b1 = b * a ^ (-1/2) * (1/2)
    X3 =  a1 = a ^ (1/2)
Solving c1 + 1 * y1 ^ 2 = 0 for y1

    y1 ^ 2 = 0 + -1 * c1
        (by Isolation)

    y1 = (0 + -1 * c1) ^ (1/2) # y1 = -1 * (0 + -1 * c1) ^ (1/2)
        (by Isolation)

Answer is :
( X1 # X2 )
  where :
    X1 =  y1 = (c1 * -1) ^ (1/2)
    X2 =  y1 = (c1 * -1) ^ (1/2) * -1

Applying substitution
    y1 = a1 * x + b1
    c1 = c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1 & b1 = b * a ^ (-1/2) * (1/2)
    a1 = a ^ (1/2)

  to    :
    y1 = (c1 * -1) ^ (1/2) # y1 = (c1 * -1) ^ (1/2) * -1

  gives :
    a ^ (1/2) * x + b * a ^ (-1/2) * (1/2) = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) # a ^ (1/2) * x + b * a ^ (-1/2)
 * (1/2) = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * -1

Solving a ^ (1/2) * x + b * a ^ (-1/2) * (1/2) = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) # a ^ (1/2) * x + b * a ^ (-
1/2) * (1/2) = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * -1 for x

    a ^ (1/2) * x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) + -1 * (b * a ^ (-1/2) * (1/2))
        (by Isolation)

    x = (((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) + -1 * (b * a ^ (-1/2) * (1/2))) * a ^ (-1/2)
        (by Isolation)

    a ^ (1/2) * x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * -1 + -1 * (b * a ^ (-1/2) * (1/2))
        (by Isolation)
```

```
    x = (((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * -1 + -1 * (b * a ^ (-1/2) * (1/2))) * a ^ (-1/2)
        (by Isolation)

Answer is :
( X1 # X2 )
  where :
    X1 =  x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2)
    X2 =  x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) * -1 + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2)

 x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2) # x = ((c + (b * a ^
 (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) * -1 + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2) is a solution

Answer is :
( X1 # X2 )
  where :
    X1 =  x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2)
    X2 =  x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) * -1 + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2)


A = x = ((c + (b * a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2) # x = ((c + (b *
a ^ (-1/2) * (1/2)) ^ 2 * -1) * -1) ^ (1/2) * a ^ (-1/2) * -1 + a ^ (-1/2) * a ^ (-1/2) * b * (-1/2)

yes
| ?- core     97280  (68096 lo-ses + 29184 hi-ses)
heap     62976 =  60926 in use +   2050 free
global    1187 =     16 in use +   1171 free
local     1024 =     16 in use +   1008 free
trail      511 =      0 in use +    511 free
    0.02 sec. for 1 GCs gaining 312 words
    1.00 sec. for 37 local shifts and 41 trail shifts
    7.81 sec. runtime
```







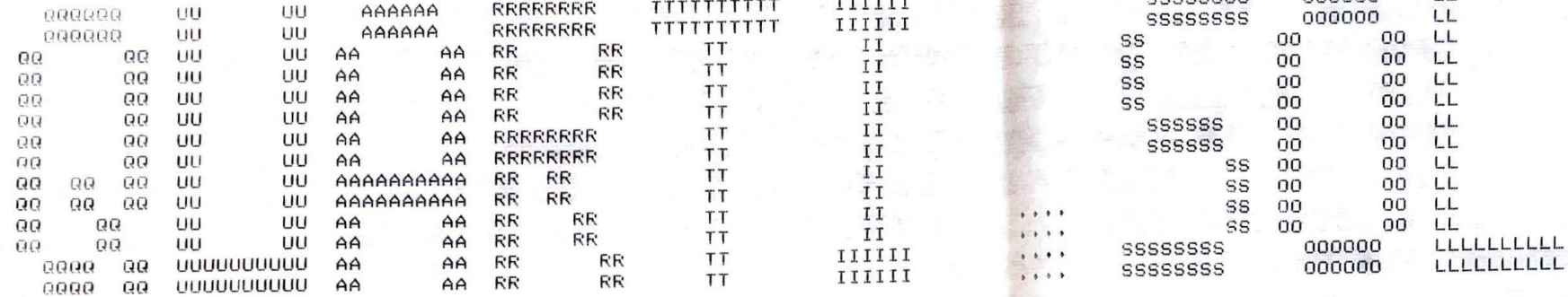
START User BUNDY HPS [400,405] Job CUBIC Seq. 6486 Date 20-May-81 13:29:36 Monitor ERCC ICF DEC10 7.01(047) *START*
File: DSKA:CUBIC.SOL<005>[400,405,MYPRES,CUBIC,SPEC] Created: 20-May-81 13:23:41 Printed: 20-May-81 13:29:53
QUEUE Switches: /FILE:ASCII /COPIES:1 /SPACING:1 /LIMIT:56 /FORMS:NORMAL

```
yes
| ?- cubic(A).
Solving a * x ^ 3 + b * x ^ 2 + c * x + d = 0 for x
Trying to specialize polynomial to eliminate x^2
term and reduce coefficient of x^3 to 1

Formed equation c2 + c1 * y1 + 1 * y1 ^ 3

Applying substitution
    y1 = a1 * x + b1

  to    :
    c2 + c1 * y1 + 1 * y1 ^ 3

  gives :
    c2 + c1 * (a1 * x + b1) + (a1 * x + b1) ^ 3

Equating coefficients of x

Simultaneously solving :
    a = a1 ^ 3
    b = b1 * a1 ^ 2 * 3
    c = c1 * a1 + a1 * b1 ^ 2 * 3
    d = c2 + c1 * b1 + b1 ^ 3
For [a1, b1, c1, c2].
Solving a = a1 ^ 3 for a1

    a1 = a ^ (1/3)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  a1 = a ^ (1/3)

Applying substitution
    a1 = a ^ (1/3)

  to    :
    b = b1 * a1 ^ 2 * 3
    c = c1 * a1 + a1 * b1 ^ 2 * 3
    d = c2 + c1 * b1 + b1 ^ 3

  gives :
    b = b1 * a ^ (2/3) * 3
    c = c1 * a ^ (1/3) + a ^ (1/3) * b1 ^ 2 * 3
    d = c2 + c1 * b1 + b1 ^ 3

Solving b = b1 * a ^ (2/3) * 3 for b1

    b1 * a ^ (2/3) = b * (1/3)
        (by Isolation)
I assume a positive.

    b1 = b * (1/3) * a ^ (-2/3)
        (by Isolation)

Answer is :
X1
```

```
    where :
      X1 =  b1 = b * a ^ (-2/3) * (1/3)

  Applying substitution
      b1 = b * a ^ (-2/3) * (1/3)

    to    :
      c = c1 * a ^ (1/3) + a ^ (1/3) * b1 ^ 2 * 3
      d = c2 + c1 * b1 + b1 ^ 3

    gives :
      c = c1 * a ^ (1/3) + a ^ (1/3) * (b * a ^ (-2/3) * (1/3)) ^ 2 * 3
      d = c2 + c1 * b * a ^ (-2/3) * (1/3) + (b * a ^ (-2/3) * (1/3)) ^ 3

 Solving c = c1 * a ^ (1/3) + a ^ (1/3) * (b * a ^ (-2/3) * (1/3)) ^ 2 * 3 for c1

     c1 * a ^ (1/3) = c + -1 * (a ^ (1/3) * (b * a ^ (-2/3) * (1/3)) ^ 2 * 3)
         (by Isolation)

     c1 = (c + -1 * (a ^ (1/3) * (b * a ^ (-2/3) * (1/3)) ^ 2 * 3)) * a ^ (-1/3)
         (by Isolation)

 Answer is :
 X1
   where :
     X1 =  c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3)

 Applying substitution
     c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3)

   to    :
     d = c2 + c1 * b * a ^ (-2/3) * (1/3) + (b * a ^ (-2/3) * (1/3)) ^ 3

   gives :
     d = c2 + (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3) * b * a ^ (-2/3) * (1/3) + (b * a ^ (-2/3) * (1
/3)) ^ 3

Solving d = c2 + (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3) * b * a ^ (-2/3) * (1/3) + (b * a ^ (-2/3)
* (1/3)) ^ 3 for c2

    c2 + (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3) * b * a ^ (-2/3) * (1/3) = d + -1 * (b * a ^ (-2/3)
 * (1/3)) ^ 3
        (by Isolation)

    c2 = d + -1 * (b * a ^ (-2/3) * (1/3)) ^ 3 + -1 * ((c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3) * b *
 a ^ (-2/3) * (1/3))
        (by Isolation)

Answer is :
X1
  where :
    X1 =  c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2/
3) * a ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3

Applying substitution
    c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2/3) * a
 ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3

  to    :
```

```
    gives :

  Substituting back in c2 solution
  Applying substitution

    to    :
      c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2/3) * a
  ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3

    gives :
      c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2/3) * a
  ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3

  Substituting back in c1 solution
  Applying substitution
      true
      c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2/3) * a
  ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3

    to    :
      c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3)

    gives :
      c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3)

  Substituting back in b1 solution
  Applying substitution
      true & c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2
  /3) * a ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3
      c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3)

    to    :
      b1 = b * a ^ (-2/3) * (1/3)

    gives :
      b1 = b * a ^ (-2/3) * (1/3)

  Substituting back in a1 solution
  Applying substitution
      (true & c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-
  2/3) * a ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3) & c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a
  ^ (-1/3)
      b1 = b * a ^ (-2/3) * (1/3)

    to    :
      a1 = a ^ (1/3)

    gives :
      a1 = a ^ (1/3)


  Final Answers are :
  ( ( ( X1 & X2 ) & X3 ) & X4 )
    where :
      X1 =  c2 = d + c * a ^ (-1/3) * a ^ (-2/3) * b * (-1/3) + (a ^ (-2/3) * a ^ (-2/3) * a ^ (-2/3) * (-1/27) + a ^ (1/3) * a ^ (-2/
  3) * a ^ (-2/3) * a ^ (-1/3) * a ^ (-2/3) * (1/9)) * b ^ 3
      X2 =  c1 = (c + a ^ (1/3) * b * a ^ (-2/3) * b * a ^ (-2/3) * (-1/3)) * a ^ (-1/3)
      X3 =  b1 = b * a ^ (-2/3) * (1/3)
      X4 =  a1 = a ^ (1/3)
  Solving c2 + c1 * y1 + 1 * y1 ^ 3 = 0 for y1
```

```
no
| ?- core      97280  (68096 lo-segs + 29184 hi-segs)
heap      62976 =  60936 in use +   2040 free
global     1187 =     16 in use +   1171 free
local      1024 =     16 in use +   1008 free
trail       511 =      0 in use +    511 free
     0.02 sec. for 1 GCs gaining 938 words
     1.33 sec. for 36 local shifts and 42 trail shifts
     7.80 sec. runtime
```



START User BUNDY HPS [400,405] Job QUARTI Seq. 6475 Date 20-May-81 12:07:46 Monitor ERCC ICF DEC10 7.01(047) *START*
File: DSKA:QUARTI.SOL<005>[400,405,MYPRES,CUBIC,SPEC] Created: 20-May-81 12:00:29 Printed: 20-May-81 12:08:01
QUEUE Switches: /FILE:ASCII /COPIES:1 /SPACING:1 /LIMIT:62 /FORMS:NORMAL

```
yes
| ?- quartic(A).
Solving a * x ^ 4 + b * x ^ 3 + c * x ^ 2 + d * x + e = 0 for x
Trying to specialize polynomial to eliminate x^3
term and reduce coefficient of x^4 to 1

Formed equation c3 + c2 * y1 + c1 * y1 ^ 2 + 1 * y1 ^ 4

Applying substitution
    y1 = a1 * x + b1

  to    :
    c3 + c2 * y1 + c1 * y1 ^ 2 + 1 * y1 ^ 4

  gives :
    c3 + c2 * (a1 * x + b1) + c1 * (a1 * x + b1) ^ 2 + (a1 * x + b1) ^ 4

Equating coefficients of x

Simultaneously solving :
    a = a1 ^ 4
    b = b1 * a1 ^ 3 * 4
    c = (c1 + b1 ^ 2 * 6) * a1 ^ 2
    d = c2 * a1 + c1 * a1 * b1 * 2 + a1 * b1 ^ 3 * 4
    e = c3 + c2 * b1 + c1 * b1 ^ 2 + b1 ^ 4
For [a1, b1, c1, c2, c3].
Solving a = a1 ^ 4 for a1
I assume a1 positive.

    a1 = a ^ (1/4)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  a1 = a ^ (1/4)

Applying substitution
    a1 = a ^ (1/4)

  to    :
    b = b1 * a1 ^ 3 * 4
    c = (c1 + b1 ^ 2 * 6) * a1 ^ 2
    d = c2 * a1 + c1 * a1 * b1 * 2 + a1 * b1 ^ 3 * 4
    e = c3 + c2 * b1 + c1 * b1 ^ 2 + b1 ^ 4

  gives :
    b = b1 * a ^ (3/4) * 4
    c = (c1 + b1 ^ 2 * 6) * a ^ (1/2)
    d = c2 * a ^ (1/4) + c1 * a ^ (1/4) * b1 * 2 + a ^ (1/4) * b1 ^ 3 * 4
    e = c3 + c2 * b1 + c1 * b1 ^ 2 + b1 ^ 4

Solving b = b1 * a ^ (3/4) * 4 for b1

    b1 * a ^ (3/4) = b * (1/4)
        (by Isolation)
I assume a positive.

    b1 = b * (1/4) * a ^ (-3/4)
```

```
          (by Isolation)

   Answer is :
   X1
     where :
       X1 =  b1 = b * a ^ (-3/4) * (1/4)

   Applying substitution
       b1 = b * a ^ (-3/4) * (1/4)

     to    :
       c = (c1 + b1 ^ 2 * 6) * a ^ (1/2)
       d = c2 * a ^ (1/4) + c1 * a ^ (1/4) * b1 * 2 + a ^ (1/4) * b1 ^ 3 * 4
       e = c3 + c2 * b1 + c1 * b1 ^ 2 + b1 ^ 4

     gives :
       c = (c1 + (b * a ^ (-3/4) * (1/4)) ^ 2 * 6) * a ^ (1/2)
       d = c2 * a ^ (1/4) + c1 * a ^ (1/4) * b * a ^ (-3/4) * (1/2) + a ^ (1/4) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4
       e = c3 + c2 * b * a ^ (-3/4) * (1/4) + c1 * (b * a ^ (-3/4) * (1/4)) ^ 2 + (b * a ^ (-3/4) * (1/4)) ^ 4

 Solving c = (c1 + (b * a ^ (-3/4) * (1/4)) ^ 2 * 6) * a ^ (1/2) for c1

     c1 + (b * a ^ (-3/4) * (1/4)) ^ 2 * 6 = c * a ^ (-1/2)
         (by Isolation)

     c1 = c * a ^ (-1/2) + -1 * ((b * a ^ (-3/4) * (1/4)) ^ 2 * 6)
         (by Isolation)

 Answer is :
 X1
   where :
     X1 =  c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)

 Applying substitution
     c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)

   to    :
     d = c2 * a ^ (1/4) + c1 * a ^ (1/4) * b * a ^ (-3/4) * (1/2) + a ^ (1/4) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4
     e = c3 + c2 * b * a ^ (-3/4) * (1/4) + c1 * (b * a ^ (-3/4) * (1/4)) ^ 2 + (b * a ^ (-3/4) * (1/4)) ^ 4

   gives :
     d = c2 * a ^ (1/4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * a ^ (1/4) * b * a ^ (-3/4) * (1/2) + a ^ (1/4
 ) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4
     e = c3 + c2 * b * a ^ (-3/4) * (1/4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4) * (1/4)) ^
 2 + (b * a ^ (-3/4) * (1/4)) ^ 4

 Solving d = c2 * a ^ (1/4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * a ^ (1/4) * b * a ^ (-3/4) * (1/2) + a ^
 (1/4) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4 for c2

     c2 * a ^ (1/4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * a ^ (1/4) * b * a ^ (-3/4) * (1/2) = d + -1 * (a
 ^ (1/4) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4)
         (by Isolation)

     c2 * a ^ (1/4) = d + -1 * (a ^ (1/4) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4) + -1 * ((c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/
 4) * (-3/8)) * a ^ (1/4) * b * a ^ (-3/4) * (1/2))
         (by Isolation)

     c2 = (d + -1 * (a ^ (1/4) * (b * a ^ (-3/4) * (1/4)) ^ 3 * 4) + -1 * ((c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)
 ) * a ^ (1/4) * b * a ^ (-3/4) * (1/2))) * a ^ (-1/4)
         (by Isolation)
```

```
Answer is :
X1
  where :
     X1 =  c2 = d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-
3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3

Applying substitution
     c2 = d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) *
 a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3

  to    :
     e = c3 + c2 * b * a ^ (-3/4) * (1/4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4) * (1/4)) ^
2 + (b * a ^ (-3/4) * (1/4)) ^ 4

  gives :
     e = c3 + (d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3
/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3) * b * a ^ (-3/4) * (1/
4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4) * (1/4)) ^ 2 + (b * a ^ (-3/4) * (1/4)) ^ 4

Solving e = c3 + (d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^
 (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3) * b * a ^ (-3/4) *
 (1/4) + (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4) * (1/4)) ^ 2 + (b * a ^ (-3/4) * (1/4)) ^ 4 f
or c3

    c3 + (d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4)
* a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3) * b * a ^ (-3/4) * (1/4) +
 (c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4) * (1/4)) ^ 2 = e + -1 * (b * a ^ (-3/4) * (1/4)) ^ 4
       (by Isolation)

    c3 + (d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4)
* a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3) * b * a ^ (-3/4) * (1/4) =
 e + -1 * (b * a ^ (-3/4) * (1/4)) ^ 4 + -1 * ((c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4) * (1/4)
) ^ 2)
       (by Isolation)

    c3 = e + -1 * (b * a ^ (-3/4) * (1/4)) ^ 4 + -1 * ((c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)) * (b * a ^ (-3/4)
 * (1/4)) ^ 2) + -1 * ((d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4)
 * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3) * b * a ^ (-
3/4) * (1/4))
       (by Isolation)

Answer is :
X1
  where :
     X1 =  c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1/
4) * a ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a ^
 (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4)
* a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4

Applying substitution
     c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1/4) * a
 ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a ^ (-3/4
) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^
(-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4

  to    :

  gives :
```

```
Substituting back in c3 solution
Applying substitution

  to    :
    c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1/4) * a
 ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a ^ (-3/4
) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^
(-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4

  gives :
    c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1/4) * a
 ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a ^ (-3/4
) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^
(-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4

Substituting back in c2 solution
Applying substitution
    true
    c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1/4) * a
 ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a ^ (-3/4
) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^
(-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4

  to    :
    c2 = d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) *
 a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3

  gives :
    c2 = d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) *
 a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3

Substituting back in c1 solution
Applying substitution
    true & c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1
/4) * a ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a
 ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4)
 * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4
    c2 = d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) *
 a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3

  to    :
    c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)

  gives :
    c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)

Substituting back in b1 solution
Applying substitution
    (true & c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (
1/4) * a ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a
 ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4
) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4) & c2 = d * a ^ (-1/4) + c * a ^ (-1/2
) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a
^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3
    c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)

  to    :
    b1 = b * a ^ (-3/4) * (1/4)

  gives :
```

```
    b1 = b * a ^ (-3/4) * (1/4)

Substituting back in a1 solution
Applying substitution
    ((true & c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^
(1/4) * a ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) +
a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/
4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4) & c2 = d * a ^ (-1/4) + c * a ^ (-1/
2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a
 ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3) & c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)
    b1 = b * a ^ (-3/4) * (1/4)

  to    :
    a1 = a ^ (1/4)

  gives :
    a1 = a ^ (1/4)


Final Answers are :
( ( ( ( ( X1 & X2 ) & X3 ) & X4 ) & X5 )
  where :
    X1 =  c3 = e + d * a ^ (-1/4) * a ^ (-3/4) * b * (-1/4) + (a ^ (-1/2) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-1/2) * a ^ (1/
4) * a ^ (-3/4) * a ^ (-1/4) * a ^ (-3/4) * (1/8)) * c * b ^ 2 + (a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/256) + a ^
 (-3/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (3/128) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4)
* a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * a ^ (-3/4) * (-1/4)) * b ^ 4
    X2 =  c2 = d * a ^ (-1/4) + c * a ^ (-1/2) * a ^ (1/4) * a ^ (-3/4) * a ^ (-1/4) * b * (-1/2) + (a ^ (1/4) * a ^ (-3/4) * a ^ (-
3/4) * a ^ (-3/4) * (-1/16) + a ^ (-3/4) * a ^ (-3/4) * a ^ (1/4) * a ^ (-3/4) * (3/16)) * a ^ (-1/4) * b ^ 3
    X3 =  c1 = c * a ^ (-1/2) + b * a ^ (-3/4) * b * a ^ (-3/4) * (-3/8)
    X4 =  b1 = b * a ^ (-3/4) * (1/4)
    X5 =  a1 = a ^ (1/4)
Solving c3 + c2 * y1 + c1 * y1 ^ 2 + 1 * y1 ^ 4 = 0 for y1

no
| ?- core     97280  (68096 lo-segs + 29184 hi-segs)
heap     62976 =  60938 in use +   2038 free
global    1187 =     16 in use +   1171 free
local     1024 =     16 in use +   1008 free
trail      511 =      0 in use +    511 free
    2.76 sec. for 2 GCs gaining 84917 words
    1.81 sec. for 56 local shifts and 72 trail shifts
   23.56 sec. runtime
```

```
/* mymatc.
FILE IN POWERFUL MATCHER */

/* new procedures */
 :- consult(match: 'match.pl'),
    consult(match: 'hard.pl'),
    consult(match: 'bass.pl'),
    consult(match: 'expr.pl'),
    consult(match: 'memo.pl'),
    consult(match: 'pick.pl'),
    consult(match: 'fuzzy.pl'),
    consult(match: 'trform.pl'),
    consult(match: 'portra.pl'),
    consult(match: 'misc.pl'),
    consult(match: 'featur.pl'),
    consult(match: 'inst.pl'),
    consult('test.pl').

/* additions to PRESS */
 :- consult('collec.pl'),
    consult(match: 'tidy.pl'),
    consult(match: 'isolax.pl'),
    consult(match: 'sort.pl').

/* patches to PRESS */
 :- reconsult('collax.pat'),
    reconsult(match: 'decomp.pat'),
    reconsult(match: 'interv.pat').


 ok :- core_image, display('Borning matcher'), ttynl, '$reinit'.
```

collec.pl

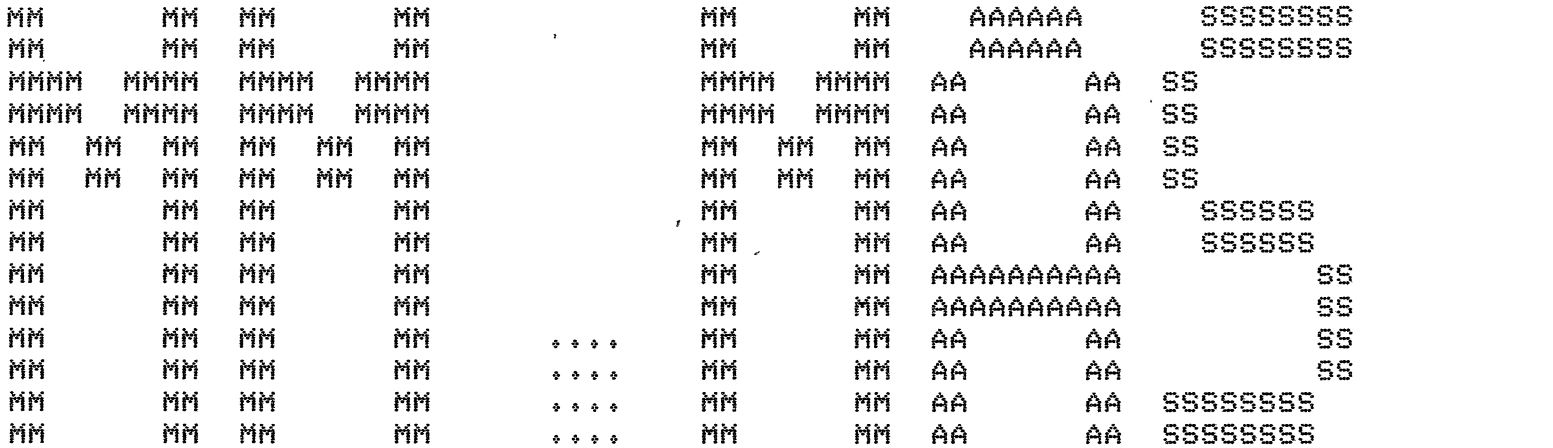
```
/* COLLECTION ROUTINE TO INVOKE POWERFUL MATCHER */


collect(X,Old,New1) :-
/*  least_dom(X,Old),   */
    flag(try_hard,true,true),
    trace('\ntrying to use powerful matcher to collect %p in %c\n',
      [X,Old],3),
    features(Old,X,FExpr),
    /* select a collection axiom */
    trace('features of expression are %p\n',[FExpr],4),
    trace('looking for a collection rule with matching features\n\n',4),
    collax(Us,LHS,RHS),
    /* bind all rule variables to random atoms */
    instantiate(LHS,PatternVars),
   /* chose a collection variable in rule and work out features wrt it */
    wordsin(Us,Ul), member(U,Ul),
    features(LHS,U,FRule),
    /* make sure the features of the expression and the rule match */
    match_cut(FExpr,FRule,X,U,Subst,NewPVars), !,
    append(NewPVars,PatternVars,PatternVars1),
   /* apply resulting substitution to rule and put in rnf */
    subst(Subst,LHS,LHS1), subst(Subst,RHS,RHS1),
    poly_form([X],LHS1,LHS2),
    /* prepare for and apply rule */
    make_description(Old,X,[],Old_D),
    make_description(LHS2,X,PatternVars1,LHS_D),
    make_description(RHS1,X,PatternVars1,RHS_D),
    apply_rule( Old_D , rule(LHS_D,RHS_D) , New_D),
    expr(New_D,New),
    tidy(New,New1),
    trace('%p collected in %p gives  %c\n',[X,Old,New1],2).



try_hard_to_solve(Eqn,Unknown,Ans) :-
   /* solve the equation using powerful matcher */
   flag(try_hard,Old,true),
   solve(Eqn,Unknown,Ans),
   flag(try_hard,_,Old).

:- flag(try_hard,_,false).


/* Match features of expr and rule and return substitution */

/* FExpr are identical except for the difference in variable */
match_cut(FExpr, FRule, X, U, U=X, []) :-
        subst(U=X, FRule, FExpr).

/* Special hack for cubic */
match_cut(FExpr, FRule, X, U, cos(U)=X*Q^(-1) & U=arccos(X*Q^(-1)), [Q] ) :-
        sensym(q,Q),
        subst(cos(U)=X, FRule, FExpr).
```

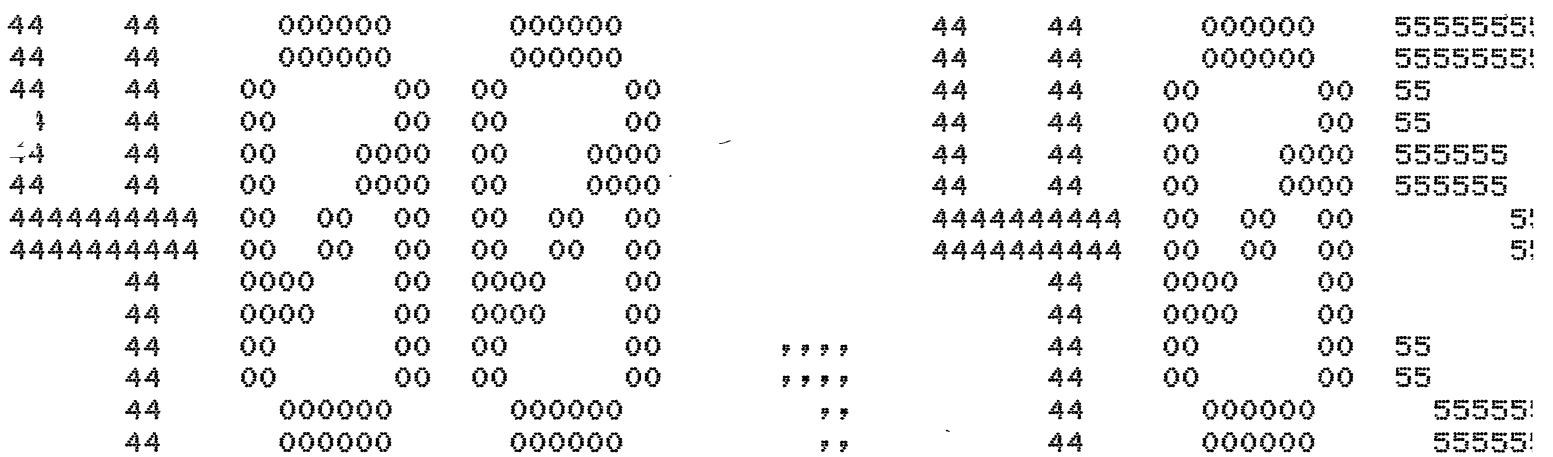
collax.pat

```
collax(U , cos(U)^3 + (-1)*3*(4^(-1))*cos(U), 4^(-1)*cos(3*U) ),

collax( W , U*W+V*W , (U+V)*W ) ,

/* collax( W , W+V*W , (V+1)*W ) ,   */

/* collax( W , V*W+(-W) , (V+(-1))*W ) ,   */

/* collax( W , W+W , 2*W ) ,   */

collax( U&V , (U+V)*(U+ (-1*V)) , U^2+ -1*(V^2) ) ,

collax( W , W^U*W^V , W^(U+V) ) ,

collax( W , W*W^V , W^(V+1) ) ,

collax( W , W*W , W^2 ) ,

collax( U , sin(U)*cos(U) , sin(2*U)*2^ (-1) ) ,

 ollax( U , cos(U)^2+ -1*(sin(U)^2) , cos(2*U) ) ,

collax( U , sin(U)*cos(V)+cos(U)*sin(V) , sin(U+V) ) ,

collax( U&V , sin(U)*cos(V)+ -1*(cos(U)*sin(V)) , sin(U+ (-1*V)) ) ,

collax( U , cos(U)*cos(V)+ -1*(sin(U)*sin(V)) , cos(U+V) ) ,

collax( U , cos(U)*cos(V)+sin(U)*sin(V) , cos(U+ (-1*V)) ) ,

collax( U , sin(U)*cos(U)^(-1) , tan(U) ),

collax( U , cos(U)*sin(U)^(-1) , cot(U) ),

collax( U , U^2+2*U*V+V^2 , (U+V)^2 ),

collax( U, U^3 + 3*U^2*V + 3*U*V^2 + V^3 , (U+V)^3 ),
```

test.p

```
/* tests for powerful matcher */

 :- tlim(5).

spec_cubic :- try_hard_to_solve(z^3 + 3*h*z + s = 0 , z , Ans).

quadratic :- try_hard_to_solve( a*x^2+b*x+c=0, x, Ans).

cubic :- try_hard_to_solve( a*x^3+b*x^2+c*x+d=0, x, Ans).

distrib1 :- try_hard_to_solve( a*x+x = x^2, x, Ans).

distrib2 :- try_hard_to_solve( a*x-x = x^2, x, Ans).

distrib3 :- try_hard_to_solve( x+x = x^2, x, Ans).

tris1 :- try_hard_to_solve( a*sin(x)+b*cos(x)=c, x, Ans).

chanse1 :- try_hard_to_solve( 5^(2*y) + 5^y + 5 = 0, y, Ans).

 hanse2 :- try_hard_to_solve( 5^(2*y) - 5^(y+1) + 6 = 0, y, Ans).

chanse3 :- try_hard_to_solve( 3^(2*y) - 2*3^(y+2) + 81 = 0, y, Ans).

/* the followins two examples are from McArthur & Keith,
      Intermediate Alsebra */

chanse4 :- try_hard_to_solve( (3*y^2-2*y-2)^2 = 21*y^2 - 14*y - 20, y, Ans).

chanse5 :- try_hard_to_solve( y^6 + 7*a^3*y^3 - 8*a^6 = 0, y, Ans).
```





START User BUNDY HPS [400,405] Job MM Seq. 3069 Date 05-Feb-81 14:37:09
File: DSKA:MM.MAS<005>[400,421,MATCH] Created: 26-Jan-81 10:19:05 Printed: 05-F
QUEUE Switches: /FILE:ASCII /COPIES:1 /SPACING:1 /LIMIT:81 /FORMS:NORMAL

SUBFILE: MM.SUB @18:47 5-AUG-1980 <005> (55)

```
mm.sub
filin
match.pl
hard.pl
bass.pl
expr.pl
memo.pl
pick.pl
fuzzy.pl
trform.pl
portra.pl
misc.pl
featur.pl
inst.pl
test.pl
learn.pl
ltest.pl
collec.pl
change.pl
tidy.pl
 solax.pl
sort.pl
collax.pat
decomp.pat
interv.pat
poly.pat
los.pl
paths.pl


\\\\\
```

```
SUBFILE: FILIN. @13:5 21-APR-1980 <005> (163)
/* FILE IN POWERFUL MATCHER */

/* new procedures */
 :- consult('match.pl'),
    consult('hard.pl'),
    consult('bass.pl'),
    consult('expr.pl'),
    consult('memo.pl'),
    consult('pick.pl'),
    consult('fuzzy.pl'),
    consult('trform.pl'),
    consult('portra.pl'),
    consult('misc.pl'),
    consult('featur.pl'),
    consult('inst.pl'),
    consult('test.pl').

/* stuff to learn how to solve specialized kinds of equations */
 :- consult('learn.pl'),
    consult('ltest.pl').

/* additions to PRESS */
 :- consult('collec.pl'),
    consult('change.pl'),
    consult('tidy.pl'),
    consult('isolax.pl'),
    consult('sort.pl').

/* patches to PRESS */
 :- reconsult('collax.pat'),
    reconsult('decomp.pat'),
    reconsult('interv.pat'),
    reconsult('poly.pat').
```

} added in to press?

\\\\\

```
SUBFILE: MATCH.PL @18:9 23-APR-1980 <005> (338)
/* POWERFUL ALGEBRAIC MATCHER */


/* The arguments to "apply_rule" are as follows:

     Expr - the expression being transformed
     Rule - the rewrite rule being applied
     New_Expr - the result of applying the rule to Expr
   The symbolic quantities in the expression and the rule are assumed
   to be standardized apart. */

apply_rule( Expr , rule(Pattern,Replacement) , New_Expr ) :-
   expr(Expr,EE), expr(Pattern,EP), expr(Replacement,ER),
   trace('trying to apply rule %p -> %p\n   to %p\n\n',[EP,ER,EE],4),
   match(Expr,Pattern,Transform),
   apply_transform(Transform,Replacement,New_Expr),
   !.


/* match is called as follows:
     match(Expr,Pattern,Transform)
        where
     Expr is the expression or subexpression being matched
     Pattern is the left hand side of the rule (or a subpart of it)
     Transform is returned - it is a transformation (functions to be applied,
        substitutions, and possibly change of unknown)
        that makes Expr=Pattern      */



/* SIMPLE CASES -- IMMEDIATE MATCH OR SIMPLE SUBSTITUTION */
match(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,E),
   null_transform(Transform),
   trace('trivially matching %p and %p\n',[E,E],4),
   !.

match(Expr,Pattern,Transform) :-
   expr(Pattern,Var),
   atom(Var),
   pattern_vars(Pattern,PatternVars),
   member(Var,PatternVars),
   expr(Expr,E),
   make_substitution_transform(Var=E,Transform),
   trace('matching %p and %p by using substitution\nreturning %p\n',
      [E,Var,Transform],4),
   !.



/* HARD MATCH - USE MEMO */
match(Expr,Pattern,Transform) :-
   memo( hard_match(Expr,Pattern,Transform) ),
   !.
```

```
SUBFILE: HARD.PL @16:45 30-APR-1980 <005> (652)
/* PROCEDURES FOR NON-TRIVIAL MATCHES */

/* There are two ways of accomplishing a hard match:
      by matching subexpressions ("match1") -- this may involve
         base matches, and applying functions to the RHS of the rule;
      or by solving for a variable in the rule ("match2").
   The subgoals match1 and match2 are used to prevent backtracking
   among the cases of match1. */

hard_match(Expr,Pattern,Transform) :-
   match1(Expr,Pattern,Transform),
   expr(Expr,E), expr(Pattern,P),
   trace('match succeeded on expression %p and pattern %p\nreturning %p\n\n',
      [E,P,Transform],4).

hard_match(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,P),
   trace('match failed on %p and %p\n',[E,P],4),
   match2(Expr,Pattern,Transform),
   trace('solving for a variable succeeded in matching expression %p\n',
      [E],4),
   trace('   and pattern %p\nreturning %p\n\n',[P,Transform],4).



/* match1 procedures to convert to base for + and *  */

match1(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,P),
   ( E=_+_ ; P=_+_ ),
   !,
   convert_and_match(+,Expr,Pattern,Transform).

match1(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,P),
   ( E=_*_ ; P=_*_ ),
   !,
   convert_and_match(*,Expr,Pattern,Transform).



/* MATCHING OTHER KINDS OF FUNCTIONS */

match1(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,P),
   trace('trying to match expression %p and pattern %p\n',
      [E,P],4),
   /* Expr and Pattern must have the same functor */
   functor(E,F,N), functor(P,F,N),
   match_parts(Expr,Pattern,1,N,Transform).


match_parts(Expr,Pattern,J,N,Transform) :-
   J>N,
```

```
    null_transform(Transform),
    !.

match_parts(Expr,Pattern,J,N,T3) :-
    subpart(Expr,J,E1), subpart(Pattern,J,P1),
    match(E1,P1,T1),
    apply_transform(T1,Pattern,P2),
    J1 is J+1,
    match_parts(Expr,P2,J1,N,T2),
    concat_transforms(T1,T2,T3).




/* "MATCH2" -- SOLVING FOR A VARIABLE IN THE RULE */

match2(Expr,Pattern,_) :-
    /* don't allow solving for a variable at top level */
    owners(Expr,[]),
    !,
    fail.

match2(Expr,Pattern,Transform) :-
    pattern_var(Pattern,V),
    expr(Expr,E), expr(Pattern,P),
    contains(V,P),
    trace('\ntrying to solve for a variable\n',4),
    trace('calling equation solver to solve for %p in %p\n\n',[V,E=P],4),
    /* assert that a particular rather than a general solution
       for V is desired */
    assert(particular_solution(V)),
    solve(E=P,V,SS),
    or_to_list(SS,SList),
    select(V=Soln,SList,_),
    make_substitution_transform(V=Soln,T1),
    /* If the unknown in Pattern is the same as the unknown in Expr (i.e.
       there is no new unknown), then the solution must be free of the
       unknown; if solving for the new unknown then the solution must
       contain the unknown; otherwise, the solution must be free of both
       the unknown and the new unknown */
    unknown(Expr,UExpr), unknown(Pattern,UPattern),
    (UPattern=UExpr -> freeof(UExpr,Soln), Transform=T1 ;
     V=UPattern -> contains(UExpr,Soln), change_unknown(UExpr,T1,Transform) ;
     freeof(UExpr,Soln), freeof(UPattern,Soln), Transform=T1 ),
    trace('using solution %p\n',[V=Soln],4).



\\\\\
```

```
SUBFILE: BAGS.PL @14:40 13-MAY-1980 <005> (659)
/* BAG PROCEDURES FOR POWERFUL ALGEBRAIC MATCHER */


convert_and_match(Op,Expr,Pattern,Transform) :-
   to_bag(Op,Expr,ExprBag),
   to_bag(Op,Pattern,PatternBag),
   bag_match(ExprBag,PatternBag,Transform).


/* TRIVIAL CASE - EMPTY BAGS */
bag_match(Expr,Pattern,Transform) :-
   expr( Expr , bag(_,[]) ),
   expr( Pattern , bag(_,[]) ),
   null_transform(Transform),
   !.


/* USE MEMO FOR OTHER CASES */
bag_match(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,P),
   E=bag(Op,_),
   (Op= + -> Name=plus ; Op= * -> Name=times ),
   trace('trying to match %t bags for expression %p\n',[Name,E],4),
   trace('   and pattern %p\n',[P],4),
   memo( bag_match1(Expr,Pattern,Transform) ).



/* TRY PICKING A TERM FROM EACH BAG AND MATCHING THESE TERMS */
bag_match1(Expr,Pattern,Transform) :-
   pick_terms(Expr,Pattern,E,P,ERest,PRest),
   trace(
'picking terms from expression & pattern bags and trying to match them\n',4),
   match(E,P,T1),
   apply_transform(T1,PRest,PR1),
   (null_transform(T1) -> true ;
    expr(PR1,RR),
    trace('applying transform to remaining terms in pattern bag\n',4),
    trace('   yielding %p\n\n',[RR],4) ),
   bag_match(ERest,PR1,T2),
   concat_transforms(T1,T2,Transform).



/* IF THERE IS JUST A VARIABLE LEFT, TRY MAKING IT THE IDENTITY ELEMENT FOR
   THE BAG.  This may not work, so be prepared to backtrack. */
bag_match1(Expr,Pattern,Transform) :-
   expr(Expr,bag(Op,[])), expr(Pattern,bag(Op,[V])),
   atom(V),
   pattern_vars(Pattern,PatternVars), member(V,PatternVars),
   unknown(Pattern,PUnknown), V\==PUnknown,
   identity(Op,Ident),
   make_substitution_transform(V=Ident,Transform),
   trace('trying making %p the bag identity element %p\n',[V,Ident],4).
```

```
/* SEE IF THERE'S JUST A PATTERN VARIABLE IN THE PATTERN, AND RANDOM
   JUNK LEFT IN THE EXPRESSION THAT'S FREE OF THE UNKNOWN */
bas_match1(Expr,Pattern,Transform) :-
   expr(Expr,Bas), expr(Pattern,bas(Op,[V])),
   pattern_var(Pattern,V), not unknown(Pattern,V),
   unknown(Expr,EUnknown), freeof(EUnknown,Bas),
   from_bas(Expr,Junk), expr(Junk,J),
   make_substitution_transform(V=J,Transform),
   trace('substituting %p for %p\n',[J,V],4).



/* TRY ELIMINATING A TERM FROM EITHER THE EXPRESSION OR THE RULE BY MOVING
   IT OR ITS INVERSE TO THE OTHER SIDE OF THE RULE */
bas_match1(Expr,Pattern,Transform) :-
   perm2(Expr,Pattern,Try,_),
   select_term(Try,T,Rest),
   op_distributes(T),
   expr(T,TT),
   unknown(Expr,EUnknown), freeof(EUnknown,TT),
   unknown(Pattern,PUnknown), freeof(PUnknown,TT),
   trace('dealing with term %p\n', [TT],4),
   trace('   by applying a function to each side of the rule\n',4),
   expr(Try,bas(Op,_)),
   (Try=Expr ->
      make_function_transform(Op,TT,T1), !, bas_match(Rest,Pattern,T2) ;
      make_inv_function_transform(Op,TT,T1), !, bas_match(Expr,Rest,T2) ),
   concat_transforms(T1,T2,Transform).


/* TRY INVOKING SOLVE-FOR-VARIABLE MATCH */
bas_match1(Expr,Pattern,Transform) :-
   from_bas(Expr,E), from_bas(Pattern,P),
   memo( match2(E,P,Transform) ).


/* FAILURE - OUTPUT A MESSAGE */
bas_match1(Expr,Pattern,Transform) :-
   expr(Expr,E), expr(Pattern,P),
   trace('bas match failed on %p and %p\n',[E,P],4),
   fail.



\\\\\
```

```
SUBFILE: EXPR.PL @14:45 13-MAY-1980 <005> (799)
/* EXPRESSION DESCRIPTIONS

An expression description is a data structure for describing expressions
and subexpressions for the powerful matcher, along with some associated
access procedures.

data structure format:
   expr_description(Expr,Root,Unknown,PatternVars,Owners)
      where
   Expr is the current expression
   Root is the root of the expression tree, of which Expr is a subexpression
   Unknown is the current unknown
   PatternVars is a list of the pattern variables in Root
   Owners is a list of owners of Expr (sort of like a path from the root)

*/


/* ACCESS TO PARTS */

expr( expr_description(Expr,Root,Unknown,PatternVars,Owners) , Expr) :- !.

root( expr_description(Expr,Root,Unknown,PatternVars,Owners) , Root) :- !.

unknown( expr_description(Expr,Root,Unknown,PatternVars,Owners) ,
      Unknown) :- !.

symbols( expr_description(Expr,Root,Unknown,PatternVars,Owners) , Symbols) :-
   wordsin(Root,Symbols),
   !.

pattern_vars( expr_description(Expr,Root,Unknown,PatternVars,Owners) ,
      PatternVars) :- !.

owners( expr_description(Expr,Root,Unknown,PatternVars,Owners) , Owners) :- !.


/* REPLACING PARTS */

new_expr( expr_description(Expr,Root,Unknown,PatternVars,Owners) ,
      New_Expr ,
      expr_description(New_Expr,Root,Unknown,PatternVars,Owners) ) :- !.

new_owners( expr_description(Expr,Root,Unknown,PatternVars,Owners) ,
      New_Owners ,
      expr_description(Expr,Root,Unknown,PatternVars,New_Owners) ) :- !.

new_unknown( expr_description(Expr,Root,Unknown,PatternVars,Owners) ,
      New_Unknown ,
      expr_description(Expr,Root,New_Unknown,PatternVars,Owners) ) :- !.

new_pattern_vars( expr_description(Expr,Root,Unknown,PatternVars,Owners) ,
      New_PatternVars ,
      expr_description(Expr,Root,Unknown,New_PatternVars,Owners) ) :- !.
```

```
/* TEST IF SOMETHING'S A PATTERN VARIABLE,
   OR RETURN ONE NONDETERMINISTICALLY */
pattern_var( expr_description(Expr,Root,Unknown,PatternVars,Owners) , V ) :-
   member(V,PatternVars).

/* TEST FOR EXPR THAT'S AN EMPTY BAG */
empty( expr_description(bag(_,[]),Root,Unknown,PatternVars,Owners) ).



/* MAKE A DESCRIPTION GIVEN AN EXPRESSION AND AN UNKNOWN */

make_description( Expr, Unknown , PatternVars ,
      expr_description(Expr,Expr,Unknown,PatternVars,[]) ) :- !.



subpart( expr_description(Expr,Root,Unknown,PatternVars,Owners) , N ,
      expr_description(SubExpr,Root,Unknown,PatternVars,NewOwners) ) :-
   ars(N,Expr,SubExpr),
   add_owner(Owners,Expr,NewOwners),
   !.


/* SELECT NONDETERMINISTICALLY A TERM FROM A BAG */

select_term(Expr,T,Rest) :-
   Expr = expr_description( E, Root,Unknown,PatternVars,Owners),
   E = bag(Op,Args),
   select(A,Args,ARest),
   add_owner(Owners,E,New_Owners),
   T = expr_description( A, Root,Unknown,PatternVars,New_Owners),
   Rest = expr_description( bag(Op,ARest),
 Root,Unknown,PatternVars,New_Owners).



/* ROUTINES TO KEEP TRACK OF OWNERS
As the matcher is recursively called on expressions, it keeps track
of the enclosing expressions in a list of owners.  Each item in the list
is a pair such as pair(first,t), pair(other,t), or pair(first,sin).  "first"
or "other" indicates whether the term being considered is the first element
of a bag being matched. */


/* procedures for bags */
add_owner( [pair(_,Op)|Rest], bag(Op,_), [pair(other,Op)|Rest] ) :- !.
add_owner( Owners, bag(Op,_), [pair(first,Op)|Owners] ) :- !.

/* procedures for other expressions */
add_owner( Owners, Expr, [pair(first,Op)|Owners] ) :-
   Expr=..[Op|_],
   !.


op_distributes(Expr) :-
   owners(Expr,Owners),
   distributes_over_owners(Owners),
```

```
    !.


distributes_over_owners([]) :- !.

distributes_over_owners([P]) :- !.

distributes_over_owners([P1,P2|Rest]) :-
    dist1(P1,P2),
    distributes_over_owners([P2|Rest]).


dist1( pair(_,Op1) , pair(first,Op2) ) :-
    distributes(Op1,Op2).

distributes(*,+).



\\\\\
```

```
SUBFILE: MEMO.PL @18:18 11-AUG-1980 <005> (198)
/* PROCEDURE THAT REMEMBERS PREVIOUSLY COMPUTED RESULTS

   memo(Pred)
      Pred is the predicate being evaluated */


memo(Pred) :-
   recorded(Pred,memo(Pred,Result),_),
   ( Result=fail ->
        trace('looking up failure for\n   %P\n',[Pred],4), !, fail ;
     trace('looking up result for\n   %P\n',[Pred],4) ).

memo(Pred) :-
   call(Pred),
   /* Record result in data base if not there already.
      If it is already there, fail and try for another answer.
      This check is necessary -- the predicate may have been called
      previously without all possible results (including the final fail) being
      generated and recorded.  In this case, the previously recorded results
      will be re-generated before new ones.  Mumble mumble. */
   ( recorded(Pred,memo(Pred,Result),_) -> fail ;
       recordz(Pred,memo(Pred,true),_) ).

memo(Pred) :-
   /* all calls have failed -- record failure */
   recordz(Pred,memo(Pred,fail),_),
   !, fail.



\\\\\
```

```
SUBFILE: PICK.PL @14:49 13-MAY-1980 <005> (484)
/* AUXILIARY PROCEDURES FOR POWERFUL MATCHER
   SELECT BEST TERMS TO MATCH FROM BAGS */

pick_terms(Expr,Pattern,E,P,ERest,PRest) :-
   pick_term(Expr,E,ERest),
   features(E,EFeatures),
   select_term(Pattern,P,PRest),
   features(P,PFeatures),
   unknown(Expr,EUnknown), unknown(Pattern,PUnknown),
   fuzzy_match(EFeatures,PFeatures,EUnknown,PUnknown),
   /* extra check for polynomials -- check that powers are the same */
   expr(E,EE), expr(P,PP),
   power(EE,EUnknown,N1),
   power(PP,PUnknown,N2),
   (EUnknown=PUnknown, integer(N1), integer(N2) -> N1=N2 ; true),
   /* reject if the match is non-trivial and
      moving the terms to the other side would succeed */
   (PFeatures=mumble, op_distributes(E), not(match(EE,PP)) -> fail ; true).



/* PICK THE BEST TERM FROM A BAG TO TRY MATCHING NEXT */

pick_term(Expr,T,Rest) :-
   expr(Expr,E), unknown(Expr,Unknown),
   pick1(E,Unknown,TT,RR),
   owners(Expr,Owners),
   add_owner(Owners,E,New_Owners),
   new_expr(Expr,TT,T1), new_owners(T1,New_Owners,T),
   new_expr(Expr,RR,R1), new_owners(R1,New_Owners,Rest).



pick1( bag(Op,[Term]) , Unknown , Term , bag(Op,[]) ) :- !.

pick1( bag(Op,[T1,T2|Others]) , Unknown , Term , bag(Op,[TBad|Rest]) ) :-
   pick_from_pair(T1,T2,Unknown,TGood,TBad),
   pick1( bag(Op,[TGood|Others]) , Unknown , Term , bag(Op,Rest) ).


/* Crock to handle polynomials - just pick term with unknown
   to highest power.  This also handles terms free of the unknown. */

pick_from_pair(T1,T2,Unknown,TGood,TBad) :-
   power(T1,Unknown,P1),
   power(T2,Unknown,P2),
   (integer(P1), integer(P2), P1<P2 ->
          TGood=T2, TBad=T1; TGood=T1, TBad=T2).


/* power(Term,Unknown,N) unifies N with the highest power to which
   unknown occurs in term if the unknown is to an integer power, or
   to "mumble" if to a non-integer power.
```

```
   All the cuts and junk are to prevent unwanted backtrackins. */

power(Unknown,Unknown,N) :-
   !,
   N=1.

power(X,Unknown,N) :-
   atomic(X),
   !,
   N=0.

power(Unknown^N,Unknown,N1) :-
   !,
   (inteser(N) -> N1=N ; N1=mumble).

power([],Unknown,N) :-
   !,
   N=0.

power([H|T],Unknown,N) :-
   !,
   power(H,Unknown,P1),
   power(T,Unknown,P2),
   (P1=mumble -> N=mumble; P2=mumble -> N=mumble;
    P1>P2 -> N=P1; N=P2).

power(Expr,Unknown,N) :-
   !,
   Expr=..[Op|Arss],
   power(Arss,Unknown,N).



\\\\\
```

```
SUBFILE: FUZZY.PL @14:40 14-MAY-1980 <005> (169)
/* FUZZY MATCHER FOR USE WITH "FEATURE" STUFF */

fuzzy_match(Expr1,Expr2,Unknown,New_Unknown) :-
   New_Unknown=false,
   !,`
   match(Expr1,Expr2),
   !.

fuzzy_match(Expr1,Expr2,Unknown,New_Unknown) :-
   freeof(New_Unknown,Expr1),
   freeof(New_Unknown,Expr2),
   !,
   match(Expr1,Expr2),
   !.

fuzzy_match(Expr1,Expr2,Unknown,New_Unknown) :-
   /* At this point, one and only one of the expr's should contain
      New_Unknown.  For fuzzy match, just see that the other
      expression contains Unknown */
   perm2(Expr1,Expr2,E1,E2),
   contains(New_Unknown,E2),
   contains(Unknown,E1),
   !.


/* "features" will retain inteser powers -- make sure that the match`
   "mumble" */

:- asserta(( match(I,M) :- inteser(I), atom(M), M=mumble )).
:- asserta(( match(M,I) :- inteser(I), atom(M), M=mumble )).



\\\\\
```

```
SUBFILE: TRFORM.PL @17:27 23-APR-1980 <005> (669)
/* TRANSFORMS */

/* Transforms are data structures that represent functions, substitutions,
   and possibly a change of unknown to be applied to an expression.
   Format:
     transform(FunctionList,SubstitutionList,New_Unknown)
   New_Unknown is "false" if the unknown isn't being changed */


/* CREATING TRANSFORMS */

null_transform( transform([],[],false) ).

make_substitution_transform( S , transform([],[S],false) ).

make_function_transform( Op , Expr ,
      transform( [function(Op,Expr)] , [] , false) ).

make_inv_function_transform( + , Expr ,
      transform( [function(+,NExpr)] , [] , false) ) :-
   tidy(-1*Expr,NExpr).

make_inv_function_transform( * , Expr ,
      transform( [function(*,InvExpr)] , [] , false) ) :-
   tidy(Expr^ -1,InvExpr).


change_unknown( New_Unknown , transform(FList,SList,false) ,
      transform(FList,SList,New_Unknown) ).


apply_transform( transform(FList,SList,New_Unknown) , Descr1 , Descr6 ) :-
   expr(Descr1,E1),
   apply_functions(FList,E1,E2),
   new_expr(Descr1,E2,Descr2),
   /*remove pattern vars that have been substituted for */
   subst_vars(SList,SVars), pattern_vars(Descr2,PVars),
   subtract(PVars,SVars,New_PVars),
   apply_substitutions(SList,Descr2,Descr3),
   new_pattern_vars(Descr3,New_PVars,Descr4),
   expr(Descr4,E4),
   tidy(E4,E5),
   new_expr(Descr4,E5,Descr5),
   (New_Unknown=false -> Descr6=Descr5 ;
      new_unknown(Descr5,New_Unknown,Descr6) ),
   !.

concat_transforms( transform(F1,S1,U1) , transform(F2,S2,U2) ,
      transform(F3,S3,U3) ) :-
   append(F1,F2,F3), append(S1,S2,S3),
   (U1=false -> U3=U2 ; U2=false -> U3=U1 ; fail),
   !.
```

```
/*    apply_functions(Functions,Expr,New)
   takes a list of functions "Functions"  and an expression "Expr".
   Returns the result of applying the functions to the expression in "New".
   The functions are of the form function(+,Args) or function(*,Args). */

apply_functions([],Expr,Expr) :- !.

apply_functions([H|T],Expr,New) :-
   apply_function(H,Expr,E1),
   apply_functions(T,E1,New).


/* These clauses handle bags.  If the function has the same operator as the
   bag, just add the new argument to the bag.  If the function's operator
   distributes over the bag, apply the function to each element.
   Otherwise fail. */

apply_function( function(Op,Func_Args), bag(Op,Argss),
   bag(Op,[Func_Args|Argss]) ) :- !.


apply_function( function(Func_Op,Func_Args), bag(Bag_Op,[]),
      bag(Bag_Op,[]) ) :-
   distributes(Func_Op,Bag_Op),
   !.

apply_function( F, bag(Bag_Op,[Args1|Argss]),
      bag(Bag_Op,[New_Args1|New_Argss]) ) :-
   F=function(Func_Op,_),
   !,  /* fail completely if Func_Op doesn't distribute */
   distributes(Func_Op,Bag_Op),
   apply_function(F,Args1,New_Args1),
   apply_function(F,bag(Bag_Op,Argss),bag(Bag_Op,New_Argss)).



/* now clauses to handle expressions not in bag form */

apply_function( function(+,Args), Expr, Expr+Args).

apply_function( function(*,Args), Expr, Expr*Args).




apply_substitutions([],X,X) :- !.

apply_substitutions([H|T],X,Z) :-
   subst(H,X,Y),
   apply_substitutions(T,Y,Z),
   !.



/* return a list of the variables from a substitution list */

subst_vars( [] , []).
subst_vars( [V=_|Rest] , [V|VList] ) :-
```

```
    subst_vars(Rest,VList).


\\\\\
```

```
SUBFILE: PORTRA.PL @16:57 23-APR-1980 <005> (176)
/* PORTRAY */

portray( bas(Op,[A]) ) :-
   write(A),
   !.

portray( bas(Op,[A|Rest]) ) :-
   write(A),
   write(Op),
   portray( bas(Op,Rest) ),
   !.

portray( bas(Op,[]) ) :-
   write('<empty bas>'),
   !.



portray( transform(FList,SList,New_Unknown) ) :-
   writef('transform:\n'),
   portray_functions(FList),
   portray_subst(SList),
   (New_Unknown=false -> true ;
    writef('   change unknown to %t\n',[New_Unknown]) ),
   !.

portray_functions([]) :- !.
portray_functions( [function(*,A)|TL] ) :-
   writef('   * %p\n',[A]),
   portray_functions(TL),
   !.
portray_functions( [function(+,A)|TL] ) :-
   writef('   + %p\n',[A]),
   portray_functions(TL),
   !.

 ortray_subst([]) :- !.
portray_subst( [Var=E|TL] ) :-
   writef('   %p  ->  %p\n',[Var,E]),
   portray_subst(TL),
   !.



portray(X) :-
   write(X),
   !.



\\\\\
```

```
SUBFILE: MISC.PL @17:15 21-APR-1980 <005> (128)
/* MISCELLANEOUS ROUTINES FOR POWERFUL ALGEBRAIC MATCHER */


identity(+,0).
identity(*,1).



to_bas(Op,Descr1,Descr2) :-
   expr(Descr1,Expr),
   decomp(Expr,[Op|A1]),
   rev(A1,Args),
   new_expr(Descr1,bas(Op,Args),Descr2),
   !.

to_bas(Op,Descr1,Descr2) :-
   expr(Descr1,Expr),
   new_expr(Descr1,bas(Op,[Expr]),Descr2),
   !.


from_bas(Descr1,Descr2) :-
   expr(Descr1,bas(Op,A1)),
   rev(A1,Args),
   recomp(Expr,[Op|Args]),
   new_expr(Descr1,Expr,Descr2),
   !.


match_cut(A,B) :-
   match(A,B),
   !.


or_to_list(H#T,[H|TT]) :-
   !,
   or_to_list(T,TT).

or_to_list(Expr,[Expr]) :- !.



\\\\\
```

```
SUBFILE: FEATUR.PL @11:24 18-APR-1980 <005> (189)
/* PROCEDURES FOR EXTRACTING FEAURES FROM EXPRESSIONS
   calling protocol:
      features(Expr,Features)
 */


features(Expr,Features) :-
   expr(Expr,E), unknown(Expr,U),
   features(E,U,Features),
   !.



features([],U,[]) :- !.

features([H|T],U,[FH|FT]) :-
   features(H,U,FH),
   features(T,U,FT),
   !.


features(U,U,U) :- !.

features(Expr,U,mumble) :-
   freeof(U,Expr),
   !.

features(E^N,U,F^N) :-
   integer(N),
   features(E,U,F),
   !.

features(E1+E2,U,Features) :-
   features(E1,U,F1),
   features(E2,U,F2),
   (F1=mumble -> Features=F2 ;
    F2=mumble -> Features=F1 ;
    Features=F1+F2),
   !.

features(E1*E2,U,Features) :-
   features(E1,U,F1),
   features(E2,U,F2),
   (F1=mumble -> Features=F2 ;
    F2=mumble -> Features=F1 ;
    Features=F1*F2),
   !.

features(Expr,U,Features) :-
   Expr=..[Op|Args],
   features(Args,U,FArgs),
   Features=..[Op|FArgs],
   !.
```

```
SUBFILE: INST.PL @16:23 23-APR-1980 <005> (59)
/* kludge to instantiate all the variables in a rule
   generated names are of the form $1, $2, etc */

instantiate(Expr,Vars) :-
   variables(Expr,Vars),
   bind_list(Vars,1),
   !.


bind_list([],N).

bind_list([H|T],N) :-
   concat($,N,H),
   N1 is N+1,
   bind_list(T,N1).



\\\\\
```

```
SUBFILE: TEST.PL @10:0 21-APR-1980 <005> (169)
/* tests for powerful matcher */

 :- tlim(5).

quadratic :- try_hard_to_solve( a*x^2+b*x+c=0, x, Ans).

cubic :- try_hard_to_solve( a*x^3+b*x^2+c*x+d=0, x, Ans).

distrib1 :- try_hard_to_solve( a*x+x = x^2, x, Ans).

distrib2 :- try_hard_to_solve( a*x-x = x^2, x, Ans).

distrib3 :- try_hard_to_solve( x+x = x^2, x, Ans).

trig1 :- try_hard_to_solve( a*sin(x)+b*cos(x)=c, x, Ans).

change1 :- try_hard_to_solve( 5^(2*y) + 5^y + 5 = 0, y, Ans).

change2 :- try_hard_to_solve( 5^(2*y) - 5^(y+1) + 6 = 0, y, Ans).

change3 :- try_hard_to_solve( 3^(2*y) - 2*3^(y+2) + 81 = 0, y, Ans).

/* the following two examples are from McArthur & Keith,
      Intermediate Algebra */

change4 :- try_hard_to_solve( (3*y^2-2*y-2)^2 = 21*y^2 - 14*y - 20, y, Ans).

change5 :- try_hard_to_solve( y^6 + 7*a^3*y^3 - 8*a^6 = 0, y, Ans).



\\\\\
```

```
SUBFILE: LEARN.PL @12:52 21-APR-1980 <005> (576)
/* PROCEDURE FOR LEARNING TO SOLVE PARTICULAR FORMS OF EQUATIONS */


learn_to_solve(Form,Unknown,Eqn,Conditions) :-
   trace('trying to learn to solve %p for %p\n\n',[Eqn,Unknown],1),
 /* solve the equation using powerful matcher */
   try_hard_to_solve(Eqn,Unknown,A1),
 /* convert Eqn to the normal form */
   C=..[Form,Unknown,Eqn,Norm_Eqn],
   call(C),
 /* change symbols in equation etc. to variables */
   wordsin(A1,Symbols),
   variablize( [Eqn,Norm_Eqn,Unknown,A1,Symbols,Conditions] ,
        [EqnVar,NormVar,UnknownVar,AnsVar,SymVars,CondVars] ),
 /* make up a conversion command to execute when the new method is run */
   Convert=..[Form,UnknownVar,E1,E2],
 /* assert the new method */
   trace('asserting new method for solving %p for %p\n\n',[Eqn,Unknown],1),
   asserta((
      solve1(E1,UnknownVar,Ans) :-
          Convert,
          match(E2,NormVar),
          trace(
           'using learned method for solving equations of the form  %p\n\n',
           [Eqn],1),
          CondVars,
          tidy(AnsVar,Ans),
          !
      )),
   !.



variablize(A,B) :-
   wordsin(A,W),
   variablize(A,W,B),
   !.

variablize(A,[],A) :- !.

variablize(A,[H|T],B) :-
   /* crock - don't variablize arbitrary integers */
   (integral(H) -> A1=A ; var_subst( _=H , A, A1) ),
   variablize(A1,T,B),
   !.


/* SUBSTITUTE THAT DOESN'T BIND OLD VARS */
var_subst(Var=Const,Old,Old) :-
   var(Old),
   !.

var_subst(Var=Const,Const,Var) :- !.

var_subst(Var=Const,X,X) :-
   atomic(X),
```

```
    !.

var_subst(Var=Const,[],[]) :- !.

var_subst(Var=Const,[H|T],[H1|T1]) :-
    var_subst(Var=Const,H,H1),
    var_subst(Var=Const,T,T1),
    !.

var_subst(Var=Const,Old,New) :-
    Old=..[Op|Args],
    var_subst(Var=Const,Args,NArgs),
    New=..[Op|NArgs],
    !.



polynomial(X, L=R, poly_eqn(X,PList)) :-
    poly_norm(X, L+ -1*R, P1),
    poly_sort(P1,P2),
    tidy(P2,PList),    /* kludge - clean up after normalization */
    !.


/* bubble sort for polynomial coefficients */
poly_sort(P1,P3) :-
    poly_sort1(P1,P2),
    (P1=P2 -> P3=P2 ; poly_sort1(P2,P3)).

poly_sort1( [A,B|Rest] , [X|S] ) :-
    perm2(A,B,X,Y),
    X=pair(NX,_), Y=pair(NY,_),
    NX>NY,
    poly_sort1( [Y|Rest] , S ),
    !.

poly_sort1(P1,P1) :- !.



'* general class doesn't change the expression */
general(X,Exp,Exp) :- !.


/* add a clause for matching stuff in polynomial normal form */
:- asserta((
match(poly_eqn(X,L1),poly_eqn(X,L2)) :-
    !, poly_match(L1,L2), !
)).



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

poly_match( [pair(N,0)] , [] ) :- !.

poly_match( [] , [pair(N,0)] ) :- !.
```

```
poly_match( [pair(N1,C1)|R1], [pair(N2,C2)|R2] ) :-
    (N1=N2 -> !, C1=C2, poly_match(R1,R2) ;
     N1>N2 -> !, C1=0, poly_match(R1,[pair(N2,C2)|R2]) ;
     /* N1<N2 */ !, C2=0, poly_match([pair(N1,C1)|R1],R2) ),
    !.
```

\\\\\

```
SUBFILE: LTEST.PL @10:1 21-APR-1980 <005> (81)
learn_quad :- learn_to_solve( polynomial, x, a*x^2+b*x+c=0 , non_zero(a)  ).

qtest1 :- solve( x^2=9 , x, Ans ).

qtest2 :- solve( x^2-x-6=0, x, Ans).

qtest3 :- solve( (x+3)*(x+2)= 6, x, Ans).


learn_tris :- learn_to_solve( general, x, a*sin(x)+b*cos(x)=c ,
      non_zero(a) ).

ttest1 :- solve( 1*sin(x)+0*cos(x)=1 , x, Ans ).

ttest2 :- solve( 1*sin(x)+1*cos(x)=2^ (2^ -1) , x, Ans ).



.\\\\
```

```
SUBFILE: COLLEC.PL @16:26 23-APR-1980 <005> (262)
/* COLLECTION ROUTINE TO INVOKE POWERFUL MATCHER */


collect(X,Old,New1) :-
/*  least_dom(X,Old),   */
    flag(try_hard,true,true),
    trace('\ntrying to use powerful matcher to collect %p in %c\n',
      [X,Old],3),
    features(Old,X,FExpr),
    /* select a collection axiom */
    trace('features of expression are %p\n',[FExpr],4),
    trace('looking for a collection rule with matching features\n\n',4),
    collax(U,LHS,RHS),
    /* take one of the variables it collects and bind it to the unknown */
    variables(U,List),
    select(X,List,_),
    /* bind all other variables to random atoms */
    instantiate(LHS,PatternVars),
    features(LHS,_,FRule),
    /* make sure the features of the expression and the rule match */
    match_sub(FExpr,FRule),
    make_description(Old,X,[],Old_D),
    make_description(LHS,X,PatternVars,LHS_D),
    make_description(RHS,X,PatternVars,RHS_D),
    apply_rule( Old_D , rule(LHS_D,RHS_D) , New_D),
    expr(New_D,New),
    contains(X,U), !, tidy(New,New1),
    trace('%p collected in %p gives  %c\n',[X,Old,New1],2).



try_hard_to_solve(Eqn,Unknown,Ans) :-
   /* solve the equation using powerful matcher */
   flag(try_hard,Old,true),
   solve(Eqn,Unknown,Ans),
   flag(try_hard,_,Old).

:- flag(try_hard,_,false).



\\\\\
```

```
SUBFILE: CHANGE.PL @14:47 14-MAY-1980 <005> (363)
/* CHANGE OF UNKNOWN ROUTINE USING POWERFUL MATCHER
   tries to change equation to a quadratic */

solve1(LHS=RHS,Unknown,Ans) :-
   /* move everything in equation to LHS and put in poly form */
   poly_form(LHS+ -1*RHS,Expr),
   /* cheap test to see if change of unknown is appropriate */
   quad_test(Expr,Unknown),
   trace('trying change of unknown to make equation into a quadratic\n',[],4),
   /* match against the general quadratic equation
      The "_zzz" junk is to ensure that the names in the expr and
      the rule are standardized apart. */
   make_description(Expr,Unknown,[],EDescr),
   make_description( a_zzz*x_zzz^2 + b_zzz*x_zzz + c_zzz ,
      x_zzz , [x_zzz,a_zzz,b_zzz,c_zzz] , Q ),
   match(EDescr,Q,Transform),
   /* substitute for a,b,c,x in solution to general quadratic */
   Sqrt = (b_zzz^2+ -4*a_zzz*c_zzz)^(2^ -1),
   Sol1 = (x_zzz=(-1*b_zzz+Sqrt)*(2*a_zzz)^ -1),
   Sol2 = (x_zzz=(-1*b_zzz+(-1*Sqrt))*(2*a_zzz)^ -1),
   make_description( Sol1 , x_zzz , [] , Sol1Descr ),
   make_description( Sol2 , x_zzz , [] , Sol2Descr ),
   apply_transform(Transform,Sol1Descr,New1Descr),
   apply_transform(Transform,Sol2Descr,New2Descr),
   expr(New1Descr,New1), expr(New2Descr,New2),
   trace(
'\napplying transform to solution to quadratic equation yielding %e\n',
      [New1#New2],4),
   try_hard_to_solve(New1#New2,Unknown,Ans).


/* Test if the expression could be made into a quadratic with
   a change of unknown.  This test consists of seeing if the
   expression is a sum, with two terms containing the unknown,
   and one of them involving exponentiation. */
quad_test(Expr,Unknown) :-
   decomp(Expr,[+|Terms]),
   select(T1,Terms,Rest),
   contains(Unknown,T1),
   subterm(A^B,T1),
   select(T2,Rest,_),
   contains(Unknown,T2),
   !.



\\\\\
```

```
SUBFILE: TIDY.PL @9:42 11-MAR-1980 <005> (79)
/* ADDITIONS TO TIDY */


/* additional tidy axioms */

nt_tidyax( (U^V)^W , U^X ) :- poly_form(V*W,X).

nt_tidyax( U^(N^ -1) , Ans ) :- eval( U^(N^ -1) , Ans).



/* new bas flushins procedure to combine like items to powers -
   put in before other procedures */
:- asserta((
f12([*|L],New) :- twofrom(L,X1^A,X2^B,R), match(X1,X2), tidy(A+B,C),
    !, f12([*,X1^C|R],New)
)).



\\\\\
```

```
SUBFILE: ISOLAX.PL @16:40 20-APR-1980 <005> (138)
/* ISOLATION AXIOMS THAT RETURN PARTICULAR SOLUTIONS
   When solving for a variable in a rule using the powerful matcher,
   particular rather than general solutions are desired.  */

:- asserta((
isolax( 1 , sin(U)=V , U=arcsin(V) , particular_solution(U) )
)).

:- asserta((
isolax( 1 , cos(U)=V , U=arccos(V) , particular_solution(U) )
)).

:- asserta((
isolax( 1 , tan(U)=V , U=arctan(V) , particular_solution(U) )
)).

:- asserta((
isolax( 1 , cosec(U)=V , U=arccosec(V) , particular_solution(U) )
)).

:- asserta((
isolax( 1 , sec(U)=V , U=arcsec(V) , particular_solution(U) )
)).

:- asserta((
isolax( 1 , cot(U)=V , U=arccot(V) , particular_solution(U) )



\\\\\
```

```
SUBFILE: SQRT.PL @11:51 20-MAR-1980 <005> (163)
/* SQUARE ROOT EVALUATION */

/* put the new eval before the old ones */
:- asserta((
eval( X^(N^ -1), Ans) :-
   eval(X,X1),
   eval(N,N1),
   inteser(X1),
   inteser(N1),
   !,
   (N1=0 -> Ans=X ;
    N1<0 -> N2 is -N1, eval(X^(N2^ -1),A1), eval(A1^ -1,Ans) ;
    /* N1 > 0 */
    remove_powers(X1,N1,2,IPart,Residue),
    (Residue=1 -> Ans=IPart ;
     IPart=1 -> Ans=Residue^(N1^ -1) ;
     Ans=IPart*Residue^(N1^ -1) )),
   !
)),

remove_powers(X,Power,J,1,X) :-
   intexp(J,Power,A),
   A>X,
   !.

remove_powers(X,Power,J,IPart,Residue) :-
   intexp(J,Power,A),
   0 is X mod A,
   X1 is X/A,
   remove_powers(X1,Power,J,IP1,Residue),
   IPart is IP1*J,
   !.

remove_powers(X,Power,J,IPart,Residue) :-
   J1 is J+1,
   remove_powers(X,Power,J1,IPart,Residue),
   !.



\\\\\
```

```
SUBFILE: COLLAX.PAT @10:35 16-APR-1980 <005> (175)
collax( W , U*W+V*W , (U+V)*W ) ,

/* collax( W , W+V*W , (V+1)*W ) ,   */

/* collax( W , V*W+(-W) , (V+(-1))*W ) ,   */

/* collax( W , W+W , 2*W ) ,   */

collax( U&V , (U+V)*(U+ (-1*V)) , U^2+ -1*(V^2) ) ,

collax( W , W^U*W^V , W^(U+V) ) ,

collax( W , W*W^V , W^(V+1) ) ,

collax( W , W*W , W^2 ) ,

collax( U , sin(U)*cos(U) , sin(2*U)*2^ (-1) ) ,

collax( U , cos(U)^2+ -1*(sin(U)^2) , cos(2*U) ) ,

collax( U , sin(U)*cos(V)+cos(U)*sin(V) , sin(U+V) ) ,

collax( U&V , sin(U)*cos(V)+ -1*(cos(U)*sin(V)) , sin(U+ (-1*V)) ) ,

collax( U , cos(U)*cos(V)+ -1*(sin(U)*sin(V)) , cos(U+V) ) ,

collax( U , cos(U)*cos(V)+sin(U)*sin(V) , cos(U+ (-1*V)) ) ,

collax( U , sin(U)*cos(U)^(-1) , tan(U) ),

collax( U , cos(U)*sin(U)^(-1) , cot(U) ),

collax( U , U^2+2*U*V+V^2 , (U+V)^2 ),

collax( U, U^3 + 3*U^2*V + 3*U*V^2 + V^3 , (U+V)^3 ),


\\\\\
```

```
SUBFILE: DECOMP.PAT @17:8 21-APR-1980 <005> (101)
decomp(A+(B+C),L) :- !, decomp(A+B+C,L).
decomp(A+B+C,[+,C|L]) :- !, decomp(A+B,[+|L]).
decomp(A+B,[+,B,A]) :- !.

decomp(A*(B*C),L) :- !, decomp(A*B*C,L).
decomp(A*B*C,[*,C|L]) :- !, decomp(A*B,[*|L]).
decomp(A*B,[*,B,A]) :- !.

decomp(A&(B&C),L) :- !, decomp(A&B&C,L).
decomp(A&B&C,[&,C|L]) :- !, decomp(A&B,[&|L]).
decomp(A&B,[&,B,A]) :- !.

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


decomp(E,F) :- E=..F, !.



\\\\\
```

```
SUBFILE: INTERV.PAT @17:48 10-APR-1980 <005> (33)
/* KLUDGES!  interval stuff is screwing up - just bypass it */

non_zero(X) :- !.

acute(X) :- !, fail.

non_reflex(X) :- !, fail.

non_nes(X) :- !, fail.



\\\\\
```

```
SUBFILE: POLY.PAT @10:39 18-MAR-1980 <005> (33)
/* disable existing method for linear and quadratic equations so that
   the program can try to learn them */

poly_method(X,Plist, X=Ans) :-
   !,
   fail.



\\\\\
```

```
SUBFILE: LOG.PL @10:8 21-MAR-1980 <005> (171)
/* LOGARITHM EVALUATION */

:- asserta((
eval1(los(Base,X),Ans) :-
    loseval(Base,X,IPart,Fraction),
    (IPart=0 -> Ans=Fraction ;
     Fraction=0 -> Ans=IPart ;
     /* return an improper fraction */
     Fraction=Denominator^ -1,
     Numerator is IPart*Denominator+1,
     Ans=Numerator*Denominator^ -1),
    !
)).



/* loseval will succeed if the logarithm can be expressed as an integer
   plus 1 over an integer */
loseval(Base,1,0,0) :- !.

loseval(Base,X,IPart,Fraction) :-
    0 is X mod Base,
    X1 is X/Base,
    loseval(Base,X1,I1,Fraction),
    IPart is I1+1,
    !.

loseval(Base,X,0,Power^ -1) :-
    X<Base,
    findpower(X,Base,2,Power),
    !.



findpower(X,Base,Test,Power) :-
    intexp(X,Test,K),
    (K<Base -> T1 is Test+1, findpower(X,Base,T1,Power), ! ;
     K=Base -> Power=Test, ! ;
     !, fail).



\\\\\
```

```
SUBFILE: PATHS.PL @18:50 26-MAR-1980 <005> (106)
/* PATHS */
/* Paths are a way of describing subparts of things.
   A path consists of a list of part-extracting functions to be applied to
   some object.  The functions are either integers (arg numbers), or lists
   consisting of a functor plus n-1 arguments.  The last argument is supplied
   by apply_path, and is a variable to hold the result */

apply_path([],Expr,Expr) :- !.

apply_path([H|T],E1,E3) :-
   (integer(H) -> arg(H,E1,E2) ;
      append(H,[E2],L), C=..L, call(C) ),
   apply_path(T,E2,E3),
   !.



\\\\\
```

```
/* cardan.
Files for solving cubic, etc by Cardan's method
Alan Bundy 13.5.81 */

:- [
-decomp,                    % Replace old decomp and recomp
-frmeqn,                    % Form Auxiliary Equation, using Pack and Unpack
packax,                     % rewrite rules for packing and unpacking
probs                       % test problems
].
```

```
rmean,
m Auxiliary Equations for Cardan method
en Bundy 13.5.81 */

* Cardan Method */
/*---------------*/

/* New polynomial method clause */
poly_method(Z,PolyBas,Ans) :-
        make_poly(Z,PolyBas,ZPoly),
        gen_term(PQTerm,NewUnks),
        subst_mess(Z=PQTerm,ZPoly,Changed),
        gensym(cardan,Name),
        form_eqn(Name,NewUnks,Changed=0,Improved,Auxiliary),
        simsolve(Auxiliary & Improved, NewUnks, Ans1),
        sensible(Ans1),
        subst_mess(Ans1,Z=PQTerm,Ans).


/* Generate a suitable change of unknown substitution */
gen_term(pte,[P,Q]).    % Change this when other things work

 * Is answer sensible? */
sensible(Ans) :-
        writef('sensible called on %t\n\n',[Ans]),
        Ans\==false.            % add extra bits as they occur

/* Form Auxiliary equations */
/*--------------------------*/

/* Suitable factor exists */
form_eqn(Name, NewUnks, Changed=0, Improved=0, Auxiliary=0) :-
        decomp(Changed, [+|Summands]),
        select(Distinguished,Summands,Rest),
        decomp(Distinguished,[*|Factors]),
        member(Auxiliary,Factors),
        is_new(Name,aux,Auxiliary),
        test(NewUnks,Auxiliary),
        recomp(Improved,[+|Rest]),
        test(NewUnks,Improved).

* Try Packing Summands */
form_eqn(Name, NewUnks, Changed=0, Improved, Auxiliary) :-
        decomp(Changed,[+|Summands]),
        select(X,Summands,Rest1), select(Y,Rest1,Rest),
        rewrite(packax,X+Y,Z),
        recomp(New,[+,Z|Rest]),
        is_new(Name,changed,New),
        form_eqn(Name,NewUnks,New=0,Improved,Auxiliary).

/* Try Unpacking Summands */
form_eqn(Name, NewUnks, Changed=0, Improved, Auxiliary) :-
        decomp(Changed,[+|Summands]),
        select(X,Summands,Rest),
        rewrite(unpackax,X,Y),
        decomp(Y,[+|Bas]),
        append(Bas,Rest,NewBas),
        recomp(New,[+|NewBas]),
        is_new(Name,changed,New),
        form_eqn(Name,NewUnks,New=0,Improved,Auxiliary).
```

```
/* Test that new equations are ok */
test([P|Qs], Eqn) :-
        contains(P,Eqn), !.

test([P|Qs], Eqn) :-
        test(Qs, Eqn), !.

/* New equation, really is new */
is_new(Name, Type, Eqn1) :- !,
        not (store(Name, Type, Eqn2) & match(Eqn1,Eqn2)),
        assert(store(Name, Type, Eqn1)).
```

```
/* packax.
Packins and Unpackins Rewrite Rules
Alan Bundy 13.5.81 */

/* Unpack Rules */

unpackax((U+V)^2, U^2 + 2*U*V + V^2).

unpackax((U+V)^3, U^3 + 3*U^2*V + 3*U*V^2 + V^3).

unpackax((U+V)^4, U^4 + 4*U^3*V + 6*U^2*V^2 + 4*U*V^3 + V^4).

unpackax(U*(V+W), U*V + U*W).

unpackax(U*(1+W), U + U*W).

unpackax(2*U, U + U).

unpackax(3*U, U + U + U).

unpackax(4*U, U + U + U + U).


/* Pack is Unpack backwards */
packax(X,Y) :- !, unpackax(Y,X).
```

```
/* decomp.
Decomposition and recomposition - experimental version for Cardan
Alan Bundy 13.5.81 */

/* Decomposition */
decomp(E-F,L) :- !, decomp(E+(-F),L).

decomp((-(-E)),L) :- !, decomp(E,L).
decomp(-(E+F),L) :- !, decomp((-E)+(-F),L).
decomP(-(E-F),L) :- !, decomp((-E)+F,L).

decomp(E+(X+Y),L) :- !, decomp(E+X+Y,L).
decomp(E+X+Y,[+,Y|L]) :- !, decomp(E+X,[+|L]).
decomp(E+X,[+,X,E]) :- !.

decomp((X*Y)^(-1),L) :- !, decomp(X^(-1)*Y^(-1),L).
decomp(E^2,L) :- !, decomp(E*E,L).
decomp(E^3,L) :- !, decomp(E*E*E,L).
decomp(E^4,L) :- !, decomp(E*E*E*E,L).

decomp(X*(Y*E),L) :- !, decomp(X*Y*E,L).
 comp(E*X*Y,[*,Y|L]) :- !, decomp(E*X,[*|L]).
_comp(E*Y,[*,Y,E]) :- !.

decomp((E&X)&Y,L) :- !, decomp(E&X&Y,L).
decomp(Y&E&X,[&,Y|L]) :- !, decomp(E&X,[&|L]).
decomp(X&E,[&,X,E]) :- !.

decomp((E#X)#Y,L) :- !, decomp(E#X#Y,L).
decomp(Y#E#X,[#,Y|L]) :- !, decomp(E#X,[#|L]).
decomp(X#E,[#,X,E]) :- !.


decomp(E,F) :- E=..F, !.


recomp(E,[+,E]) :- !.
recomp(E,[+,[]|L]) :- !,recomp(E,[+|L]).
recomp(E+X,[+,X|L]) :- !, recomp(E,[+|L]).
recomp(0,[+]) :- !.

 ecomp(E,[*,E]) :- !.
recomp(E*X,[*,X|L]) :- !, recomp(E,[*|L]).
recomp(1,[*]) :- !.

recomp(E,[&,E]) :- !.
recomp(X&E,[&,X|L]) :- !, recomp(E,[&|L]).
recomp(true,[&]) :- !.

recomp(E,[#,E]) :- !.
recomp(X#E,[#,X|L]) :- !, recomp(E,[#|L]).
recomp(false,[#]) :- !.


recomp(E,F) :- E=..F, !.
```

```
/* probs.
Test problems for Cardan Method
Alan Bundy 13.5.81 */

test1(Imp,Aux) :-
        sensym(cubic,Name),
        form_eqn(Name, [p,q], (p+q)^3 + 3*h*(p+q) + s = 0, Imp, Aux).

test2(Ans) :-
        solve(z^3 + h*z + s = 0, z, Ans).
```

```
/*                    PARTIAD          15.5.81  */
/*************************************************/
/* SIMULTANEOUS EQUATIONS ROUTINES*/
/*************************************************/

/* Susar for old type simsolve - assume that general solution required */
simsolve(Eans,Unks,Ans) :- simsolve(Eans,Unks,Ans,gen).


/*simultaneous solution with messages*/
simsolve(Eans,Us,Ans,Type)
        :- trace('Simultaneously solving : %cFor %t,\n',[Eans,Us],1),
           simsolve1(Eans,Us,Ans1,Type), tidy(Ans1,Ans),
           trace('\nFinal Answers are : %e', [Ans],1).



/* Solve conJunction of equations for set of unknowns */
simsolve1(Eans, [], Eans,Type).          % no unknowns to solve for.

 imsolve1(Eans, [X|Unks], Ans1,Type) :-          % regular case
        pick_xeqn(Eans,X,XEqn,Rest),
        solve(XEqn,X,Ans),
        distribute(Ans,Rest,Eans1),
        simsolve2(Eans1,Unks,Ans1,Type).


/*Pick equation to solve for x, and return the remainder */
pick_xeqn(EqnC,X,XEqn,RestC) :-
        andtodot(EqnC,EqnL),
        sublist(contains(X),EqnL,XEqnL),
        (XEqnL=[] -> fail_mes(X); true),
        subtract(EqnL,XEqnL,NonXRestL),
        select(XEqn,XEqnL,XRestL),
        append(XRestL,NonXRestL,RestL),
        dottoand(RestL,RestC), !.

/* Distribute Or over And */
distribute(Sub1 # Sub2, Exp, Ans1 # Ans2) :- !, % disJunction case
        distribute(Sub1,Exp,Ans1),
        distribute(Sub2,Exp,Ans2).

distribute(Sub, Exp, Sub & Ans) :- !,   % conJunction or single equation case
        subst_mess(Sub,Exp,Ans).


/* Call simsolve1 recursively and substitute back */
% Solve disJunction
simsolve2(Eans1 # Eans2, Unks, Ans1 # Ans2,gen) :- !,   % general solution wante
        simsolve2(Eans1,Unks,Ans1,gen),
        simsolve2(Eans2,Unks,Ans2,gen).

simsolve2(Eans1 # Eans2, Unks, Ans1, part) :-   % particular solution wanted
        simsolve2(Eans1,Unks,Ans1,part).

simsolve2(Eans1 # Eans2, Unks, Ans2, part) :-   % particular solution wanted
        simsolve2(Eans2,Unks,Ans2,part).


simsolve2(X=Ans1 & Eans, Unks, Ans3,Type) :- !, % Discount already solved equati
```

```
        simsolve1(Eqns, Unks, Ans2,Type),
        trace('Substituting back in %t solution\n',[X],1),
        distribute(Ans2,X=Ans1,Ans3).

/* Failure message */
fail_mess(X) :- !,
        trace('No equations containing %t\n',[X],1), fail.


/* Problems
        4. Reject silly answers as required by Cardan. (??)
*/
```

```
/* probs.
Problems for Simultaneous Equation Solving
Alan Bundy 15.5.81 */

prb1(Ans) :- simsolve(x=4 & y=x+1, [y,x], Ans,sen).

prb2(Ans) :- simsolve(x^2=4 & y=x+1, [x,y], Ans,sen).

prb3(Ans) :- simsolve(x^2=4 & y=x+1, [y,x], Ans,sen).

prb4(Ans) :- simsolve(x^2=4 & y=x+1, [x,y], Ans,part).
```

```
| ?- test2(A).
Solving z ^ 3 + h * z + g = 0 for z
Applying substitution
    z = p + q

  to    :
    g + h * z + 1 * z ^ 3

  gives :
    g + h * (p + q) + (p + q) ^ 3

Trying p + q as new  auxiliary equation
Trying h as new  auxiliary equation
Trying g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3 as new  changed equation
Trying q ^ 2 as new  auxiliary equation
Trying p as new  auxiliary equation
Trying 3 as new  auxiliary equation
Trying q as new  auxiliary equation
Trying p ^ 2 as new  auxiliary equation
Trying g + h * (p + q) + p ^ 3 + q ^ 3 + 3 * q * p * (q + p) as new  changed equation
Trying g + p ^ 3 + q ^ 3 + (q + p) * (3 * q * p + h) as new  changed equation
Trying 3 * q * p + h as new  auxiliary equation
   (1) 17 Call : match(q ^ 3, g + h * (p + q) + p ^ 3) ? g

          Ancestor list
          -------------

   (-) 0  test2(_24)
   (-) 1  solve(z ^ 3 + h * z + g = 0, z, _24)
   (-) 2  solve1(z ^ 3 + h * z + g = 0, z, _84)
   (-) 3  poly_solve(z, [pair(3, 1), pair(1, h), pair(0, g)], _84)
   (-) 4  poly_method(z, [pair(3, 1), pair(1, h), pair(0, g)], _84)
   (-) 5  form_eqn(cardan1, [p, q], g + h * (p + q) + (p + q) ^ 3 = 0, _1447, _1448)
   (-) 6  form_eqn(cardan1, [p, q], g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3 = 0, _1447, _1448)
   (-) 7  form_eqn(cardan1, [p, q], g + h * (p + q) + p ^ 3 + q ^ 3 + 3 * q * p * (q + p) = 0, _1447, _1448)
   (-) 8  form_eqn(cardan1, [p, q], g + p ^ 3 + q ^ 3 + (q + p) * (3 * q * p + h) = 0, _1447, _1448)
   (-) 9  is_new(cardan1, changed, g + p ^ 3 + q ^ 3 + (q + p) * (p * 3 * q) + (q + p) * h)
   (-) 10  not (store(cardan1, changed, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3) & match(g + p ^ 3 + q ^ 3 +
 (q + p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3))
   (-) 11  call(store(cardan1, changed, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3) & match(g + p ^ 3 + q ^ 3 +
 (q + p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3))
   (-) 12  store(cardan1, changed, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3) & match(g + p ^ 3 + q ^ 3 + (q +
 p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 + q ^ 3)
   (-) 13  call(match(g + p ^ 3 + q ^ 3 + (q + p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^
 2 + q ^ 3))
   (-) 14  match(g + p ^ 3 + q ^ 3 + (q + p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2 +
q ^ 3)
   (-) 15  match(g + q ^ 3 + (q + p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q + 3 * p * q ^ 2)
   (-) 16  match(q ^ 3 + (q + p) * (p * 3 * q) + (q + p) * h, g + h * (p + q) + p ^ 3 + 3 * p ^ 2 * q)

   (1) 17 Call : match(q ^ 3, g + h * (p + q) + p ^ 3) ?   Action (h for help) ? [ Execution aborted ]
| ?- core      97280  (68096 lo-seg + 29184 hi-seg)
heap      62976 =  60938 in use +   2038 free
global     1187 =     16 in use +   1171 free
local      1024 =     16 in use +   1008 free
trail       511 =      0 in use +    511 free
    0.02 sec. for 1 GCs gaining 348 words
    0.12 sec. for 6 local shifts and 14 trail shifts
   55.34 sec. runtime
```

Master

Note 84

Alan Bundy
11 May 1981

SPECIALIZING THE GENERAL CUBIC

1. The Implemented Method

This note reports the implementation of the Specialization Method described in note 69 for preprocessing the general cubic

$$ax^3 + bx^2 + cx + d = 0$$

to the special form

$$z^3 + hz + g = 0$$

The method is called as the last poly_method clause in a version of file POLY in my area [400,405,mypres,cubic].

poly_method gets 3 arguments: the unknown, X; a polynomial in bag form, GenBag; and a variable to bind the answer to, Ans. The new clause works as follows:

1. GenBag is checked to see that it is a general polynomial. This test is currently crude in that it succeeds iff each coefficient is an atom. Without the test the method loops.

2. Next a new polynomial in bag form, SpecBag, is made. This is like GenBag except that it has a new unknown, Y; the leading coefficient is 1; the next coefficient is 0; and the remaining coefficients are all new constants created by gensym, NewCoeffs.

3. Y is related to X by the linear substitution, Y=A*X+B, where A and B are newly created constants. SpecBag is reified into a regular polynomial, the substitution applied, the resulting polynomial put back into bag form and its coefficients paired off with those of GenBag to form a conjunction of simultaneous equations.

4. These equations are solved for A, B and NewCoeffs to produce the solutions, Solns.

5. SpecBag is reified into a regular polynomial and solved for Y with solution Ans1.

6. Y=A*X+B & Solns are substituted in Ans1 to produce Ans2 and Ans2 is solved for X to produce Ans.

All the above steps, except steps 2 and 3: the specialization of GenBag and the equating of coefficients; only require standard PRESS and UTIL procedures. The procedures for steps 2 and 3 are in file EQUATE [400,405,mypres,cubic]. Both are very straightforward.

The specialization of the leading coefficient to 1 seems reasonable as does the specialization of one of the remaining coefficents to 0. This is currently constrained to be the next to leading term, which seems rather arbitrary. I plan to libarate this constraint and do a little search on which term to eliminate.

I improved the poly_norm procedure so that it poly_forms the coefficients of its bag members. This was an efficient and powerful simplification technique. I suggest it be adopted in regular PRESS.

The specialization method has been applied to the quadratic, cubic and quartic. The quadratic is reduced to a polynomial with only a squared and constant term, which is then Isolated. Thus the specialization technique effectively solves the quadratic. The cubic is successfully Specialized, so that Collection + the Borning Matcher could then solve it as per note 74. The quartic blows up due to an intermediate expression explosion when the equated coefficient equations are being simultaneously solved. Some of the equations are quadratics. PRESS looks for general solutions (when only particular ones are required) and gets bogged down in the resulting disjunctions. There seems some hope here if the simultanous equation solving could be cleaned up.

2. An Alternative Method

At the Mecho meeting of 8.5.81, Leon suggested applying Cardan's method (see note 83) to the problem of Specializing the cubic. This section records that idea.

Suppose, for sake of simplicity, that the leading coefficient has been simplified to 1, giving the equation

$$y^3 + b'y^2 + c'y + d' = 0$$

Apply the change of unknown substitution

$$y = z + e$$

to yield the changed equation

$$(z+e)^3 + b'(z+e)^2 + c'(z+e) + d' = 0$$

Unpack this to

$$z^3 + 3z^2e + 3ze^2 + e^3 + b'z^2 + 2b'ze + b'e^2 + c'z + c'e + d' = 0$$

Collect coefficients of powers of z by Packing to get

$$z^3 + (3e+b')z^2 + (3e^2+2b'e+c')z + e^3 + b'e^2 + c'e + d' = 0$$

and form auxiliary equation

$$3e + b' = 0$$

to get improved equation

$$z^3 + (3e^2+2b'e+c')z + e^3 + b'e^2 + c'e + d' = 0$$

The fact that Cardan's method is applicable to this rather different problem gives support to its generality, i.e. tends to suggest that it is not just an ad hoc patch to solve the cubic.

The obvious next question which arises is: can the method of section 1 above be applied to the problem of solving the special cubic? I have looked at this briefly, but without much success. The specialization could be a quadratic, but to make the method work, the substitution has to be something awful. Thus Cardan's method currently seems more general than Specialization.

Master

Note 83

Alan Bundy
8 May 1981

CARDAN'S METHOD

This note is a sequel to note 69: it goes into more detail about how one of the methods of solving the general cubic described there, Cardan's Method, might be implemented in PRESS. The first observation to make is that the method in note 69 headed 'Sterling's Method', is actually a simplified version of Cardan's Method, and it is this version we will actually consider here.

1. A Worked Example

As a starting point we take the problem to solve

$$z^3 + 3hz + g = 0$$

for z. The first step, which still remains rather 'magic' is to consider the change of unknown substitution

$$z = p + q \qquad \text{(i)}$$

where p and q are new unknowns. Cardan uses $z = \sqrt[3]{p} + \sqrt[3]{q}$ here, but we can make do with (i), and it is simpler and hence less magic, so we will use it in preference.

A change of unknown, where an atom (z) is replaced by a term which contains 2 new unknowns (p and q), is non-standard on both counts. However, the replacement of 2 unknowns for 1 means that we have introduced redundancy, which can be cancelled at any stage by introducing an auxiliary equation connecting p and q. And indeed this is what we will do. The only question that remains is: can we find an equation which simplifies the problem from finding a cubic solution to finding a quadratic solution or simpler.

The change of unknown yields the changed equation

$$(p+q)^3 + 3h(p+q) + g = 0 \qquad \text{(ii)}$$

which can be rewritten as

$$p^3 + q^3 + 3(pq+h)(p+q) + g = 0 \qquad \text{(iii)}$$

On what basis this transformation can be affected I am not sure. Trying to model it is my immediate goal. The binomial expansion of $(p+q)^3$ may be understood as fairly straightforward simplification/normal-forming, but the applications of the distributive law, to remove the common factor of 3pq from $3p^2q$ and $3pq^2$ and to remove the common factor of 3(p+q) from 3pq(p+q) and 3h(p+q), is less obvious. And how are we to avoid the expansion of 3h(p+q) to 3hp and 3hq: or is this step to be made and then undone?

Step (iii) is highly suggestive of what auxiliary equation to introduce between p and q to simplify matters. Either equation

$$pq + h = 0 \qquad \text{(iv)}$$

or

$$p + q = 0 \qquad \text{(v)}$$

would simplify (iii) to

$$p^3 + q^3 + g = 0 \qquad \text{(vi)}$$

We call this the improved equation. However, (v) would make z=0, so that is no good: it has to be (iv). Why is the improved equation, (vi), any better than (iii), apart from being syntactically simpler? It is still a cubic. Cardan's change of unknown substitution is better here, since (vi) would by now be a linear equation in p and q. However, early stages of the solution would have been harder, and in particular, (iv) would be full of cube roots.

(vi) and (iv) form a pair of simultaneous equations in p and q, which rapidly deteriorate into a quadratic. How long can we wait to discover this news? Should we search ahead for different ways of simplifying (iii) (indeed (ii)) and hope for the best, or is there some cheap test we could apply to (vi) and (iv) to see that they are a disguised quadratic?

Solving (iv) for q yields

$$q = -h/p$$

and substituting this in (vi) yields

$$p^3 + (-h/p)^3 + g = 0$$

which would be recognised by Poly Method as a disguised quadratic in p^3. The solution to this will give a value for z.

2. Overview of the Method

This note reflects coffee room discussions with Leon and Bernard, as a result of which I feel that I now have a much better understanding of Cardan's Method. I hope this better understanding is reflected above and that the magic bits have been further isolated form the routine. These magic bits now seem localised enough that they could be tackled by search, and that is the line I propose to take. The basic framework is:

1. The application of the novel change of unknown technique, with different substitutions for z being attempted and backtracked on. Among these substitutions would be p+q. The number of new unknowns in the substitution would determine how many auxiliary equations connecting them could be introduced, to improve the changed equation.

2. A non-deterministic simplification with a view to gaining suggestions on introducing the auxiliary equation(s). This is probably the most ambitious part of the program, and will be my first goal (as stated above).

3. The introduction of the auxiliary equation(s) and the solution of the resulting simultaneous set. This is straightforward.

Search is involved at steps 1 and 2: the trying of different substitutions, the trying of different simplifications and the introduction of auxiliary equations. Back up would be forced if step 3 were not easier than the original problem. (Does this require a difficulty measure?) I am anxious to find additional tests to suggest back up.

3. Non-Deterministic Simplification

In this section we explore some ideas about how the 2nd step above: the non-deterministic simplification of the changed equation: could be implemented. This step would be handled by three sub-methods: an auxiliary equation forming method; a summand packer; and a summand unpacker. We described these in turn.

The job of the auxiliary equation forming submethod, Form-Eqn, is to examine the form of the changed equation and use it to suggest auxiliary equations relating p and q (e.g. (iv) and (v) above). A generalization of the technique used in the worked example above is to look for the following pattern: a sum of a few terms, all but one of which are syntactically simple. The distinguished summand is a product of terms at least one of which is a sum, e.g. pq+h or p+q. If one of these sums is set equal to zero then the distinguised summand will disappear from the changed equation, so they are the candidate auxiliary equations.

They are subjected to the following test. The three simultaneous equations: the improved equation, the auxiliary equation and the change of unknown substitution, should not imply a value for any of the original constants (z, g and h), which is independant of the new unknowns (p and q).

Consider how this works on our running example. The original changed equation is

$$(p+q)^3 + 3h(p+q) + g = 0$$

This fits the pattern we are searching for, with $(p+q)^3$ as the distinguished summand, being a product of terms of the form p+q. Thus there is only one candidate auxiliary equation,

$$p + q = 0$$

Solving this simultaneously with the change of unknown substitution,

$$z = p + q$$

will yield z=0 as the solution for z. But this does not contain p or q, so the candidate is rejected and the attempt to form an equation fails. It could also be rejected on the grounds that the improved equation is g=0: a value for g which does not contain p or q. If 3h(p+q) is tried as a distinguished summand then it will be rejected on similar grounds.

When the first submethod fails, we drop down to the second and third submethods, whose job is to shake up the changed equation to produce some more candidates. In particular, we want to apply variants of the distributive law to pack more summands into one big distinguished summand. This will be done by the second submethod, which we will call Pack. Appying Pack above would yield

$$(p+q)\{(p+q)^2 + 3hpq\} + g = 0$$

with

$$p + q = 0 \quad \text{and} \quad (p+q)^2 + 3hpq = 0$$

as candidate auxiliary equations. However the improved equation is g = 0, which is a solution for g independant of the new unknowns, so both candidates are rejected.

To find more candidates, more shaking is required. If no further Packing of the current equation, is possible then progress may be made by Unpacking and re-Packing. Unpack is the third submethod. It works by applying variants of the distributive law in reverse, so that Pack may work on a different grouping of summands. The binomial expansion is such a variant of the distributive law. It would Unpack (ii) to

$$p^3 + 3p^2q + 3pq^2 + q^3 + 3h(p+q) + g = 0$$

Pack could then group the 2nd and 3rd summands to form

$$p^3 + 3pq(p+q) + 3h(p+q) + g = 0$$

and again to form

$$p^3 + 3(p+q)(pq+h) + q^3 + g = 0$$

as required.

If the first submethod, Form-Eqn, is applied to this it will pick $3(p+q)(pq+h)$ as the distinguished summand and $(p + q = 0$ and $pq + h = 0$ as candidate auxiliary equations. We have already seen that the first of these will be rejected and that the second leads to success.

A list of rearrangements needs to be kept to prevent looping, as also does a list of candidate auxiliary equations tried and failed. I am still considering the ordering of these submethods. Is it best to try Form-Eqn first, followed by Pack and Unpack, as Isolate, Collection and Attraction are ordered, or should one Unpack completely; form all rePackings and then Form-Eqn each of these?

Note 74

Alan Bundy
15 December 1981

ADAPTING THE BORNING MATCHER TO HANDLE THE CUBIC

This note is a sequel to note 69. In that note we pointed out that for Press to be able to solve the cubic it was sufficient to Borning-match the Collection rule,

$$\cos^3 u - 3/4 \cos u \Rightarrow 1/4 \cos 3u \quad \text{(i)}$$

to the term

$$z^3 + 3hz \quad \text{(ii)}$$

generating the substitution

$$z = 2\sqrt{-h} \cos u$$

as a side effect. In this note we speculate as to how this might be done.

The normal first step of the Borning matcher when matching a rule, e.g.

$$u^2 + 2uv + v^2 \Rightarrow (u+v)^2$$

to a term, e.g.

$$ax^2 + bx$$

is to instantiate one of the variables in the rule, e.g. u to the unknown in the term, e.g. x, in such a way that the lhs of the rule and the term have the same features, e.g. $x^2 + x$ if u is instantiated to x, and hence that they will pass the fuzzy match test.

Instantiating u to z is not a sensible procedure in the case of the cubic match above. The instantiated rule,

$$\cos^3 z - 3/4 \cos z \Rightarrow 1/4 \cos 3z$$

does not have the same features as the term, (ii), it is to be matched to and it will not match that term. The question that arises is: 'Can the Borning matcher be adapted to recognise that

$$\cos^3 u - 3/4 \cos u$$

and

$$z^3 + 3hz$$

can be made to have the same features, namely $z^3 + z$, provided that u is instantiated not to z, but to arccos z/v, i.e. such that

$$z = v \cos u \ ?'$$

The status of v is the same as in the quadratic example: it is a variable remaining to be instantiated during the course of the match. In fact it will receive the instantiation of $2\sqrt{-h}$ during the matching of the linear terms.

The simplest instantiation which would give both lhs side of rule, (i), and the term, (ii), the same features is $z = \cos u$. One reason for preferring $z = v \cos u$ is that the range of cos u is only -1 to 1 and the v factor can bring this up to $-\infty$ to ∞.

The matching will proceed as follows.

match $z^3/v^3 - 3/4\ z/v \Rightarrow 1/4 \cos 3.\arccos z/v$
to $z^3 + 3hz$

matching cubic terms produces multiplicative factor v^3

matching linear term $-3/4\ v^2 z$ to $3hz$
generates an equation for v whose solution is

$$v = 2\sqrt{-h}$$

The match succeeds and the rewritten term is

$$2\sqrt{-h^3} \cos 3.\arccos z/2\sqrt{-h}$$

as required.

Note 69

Master

Alan Bundy
12 September 1980

SOLVING THE GENERAL CUBIC

In response to a challenge by Woody Bledsoe to prove the idea of Meta-Level Inference by 'solving hard problems', I have been looking at the solution to the general cubic. This was a milestone in the history of algebra and equation solving and is a hard problem - I was unable to solve it myself unaided.

Altogether I have found 5 independant solutions to the cubic - 3 in [Burnside and Panton 81], 1 in [Barnard and Child 36] and 1 communicated to me by Leon Sterling. In this note I describe all 5 solutions together with annotations discussing the difficulties of getting Press to produce these solutions. So far Press has managed to do a crucial bit of the solution in [Barnard and Child 36] and I also describe this. My goal is to enrich Press with new methods until it is capable of finding all 5 solutions and then to test Press on the general quadratic, quartic and other problems to be found in the references below.

The discussion in [Burnside and Panton 81] is particularly valuable and I commend it to you. They show how each of the 3 methods of solving the cubic can also be applied to the quadratic and the quartic. In fact, they start by illustrating the methods on the quadratic and then apply them to the other problems.

All 5 methods contain some element of 'magic', i.e. some step in which an unmotivated expression is conjured out of the air. I have some hope that 'meta-level inference' can help here because:

- in one case at least, the use of an identity in the [Barnard and Child 36] solution, the magic is reduced by recognising the identity as a Collection rule;

- if the magic bits are seen as evidence of random search then the search space is hopeless at the object level, but not unreasonable at the meta-level.

Preprocessing - Reducing the Equation

All three solution methods start the same way, by the reduction of the general form

$b' = 3b$

$$ax^3 + 3bx^2 + 3cx + d = 0 \qquad (1)$$

by eliminating the quadratic term and specializing the leading coefficient to 1, to the special form

$z = ax + b$
$= ax + b'/3$

$$z^3 + 3hz + g = 0 \qquad (2)$$

This is done by linear substitution, e.g. by putting

$$z = Ex + F$$

substituting this in equation 2 and equating coefficients with equation 1 to find the values of E and F which will do the trick. These turn out to be a and b, respectively. (The neatness of this substitution turns, in part, on the '3' coefficients in 1 and 2 above. If they were omitted then the method would work but more messily.)

This method of 'linear substitution' and 'equating coefficients' turns up again and again in these methods. It should certainly be part of Press. It seems particularly easy to implement.

The magic part of the above is deciding to aim for the form 2. It is not too bad if we assume that preprocessing any polynomial by specializing its leading coefficient and eliminating one or more of its terms is always a good first move. Leading coefficients can always be specialized to 1, but only some terms can be eliminated. In the case of the cubic, the elimination of the cubic or constant term are impossible, since that would reduce the cubic to the quadratic. However, either the quadratic or the linear term can be eliminated. It would be interesting to explore the application of the methods below to the second of these special forms.

Method 1 - Guessing the Format

We now explore methods of solving equation 2. The first of these can be found in both [Burnside and Panton 81] and [Barnard and Child 36] and is due to Cardan (see [Cardan 68]). The key step is to guess the form of the answer. This is the 'magic' part. This guess is then used to form a new cubic similar to 2 then the two equations have their coefficients equated in order to put more flesh on the original guess.

Cardan's guess is that the final answer has the form

$$z = \sqrt[3]{P} + \sqrt[3]{Q} \qquad (3)$$

I don't know what was going on in Cardan's mind[1] when he came up with this particular form. Presumably the $\sqrt[3]{}$s are suggested by the cubes and the analogy with the quadratic case where the format is

$$z = P + \sqrt{Q}$$

By cubing 3 we get the cubic

$$z^3 = \sqrt[3]{P^3} + 3\sqrt[3]{P^2}\sqrt[3]{Q} + 3\sqrt[3]{P}\sqrt[3]{Q^2} + \sqrt[3]{Q^3}$$

which can be manipulated by simplification and resubstitution of 3 into the form

$$z^3 - 3\sqrt[3]{P}\sqrt[3]{Q}z - (P+Q) = 0$$

This is now closely analogous to 2. Why this particular simplification and resubstitution is done is one of the magic parts of this method. Note that there are an infinite number of ways of forming cubics analogous to 2, but this one yields nice values for P and Q after the equating of coefficients, that is equating linear and constant terms produces the equations

$$\sqrt[3]{P}.\sqrt[3]{Q} = -h \text{ and}$$
$$P+Q = -g$$

These define a quadratic in P (or Q) which can be solved in terms of g and h, namely

---

[1] Or rather Tartaglia's. See [Cardan 68] for an interesting discussion of the attribution of the general solution.

$$P^2 + gP - h^3 = 0^2$$

and substituting these values back in 3 gives the answer to 2.

Despite the two magic bits, this solution looks one of the most promising for mechanisation because it is so short.

This is really Cardan.

Method 2 - Sterling's Method

The next method I will consider is the one communicated to me by Leon Sterling, because it bears some similarity to the method outlined above. It starts by guessing the format of the answer, but in this case a slightly less specific format, namely

$$z = P + Q$$

Instead of using this to form another cubic, it is substituted directly into 2, yielding

$$P^3 + Q^3 + 3PQ(P+Q) + 3h(P+Q) + g = 0 \qquad (5)$$

The second piece of magic in this method involves spotting that a further specilization of P and Q, by letting

$$PQ = -h$$

will simplify 5 considerably to

$$P^3 + Q^3 + g = 0$$

In fact this is a disguised quadratic in P^3, namely

$$(P^3)^2 + gP^3 - h^3 = 0$$

basically the same quadratic as 4 in the first method. Solving this yields values for P and Q which can be substituted back in the original substitution for z.

I do not have any feel for how general this method is and thus how hard it would be to implement. A general digression :- It is always possible to write code which will lead Press through a particular solution. This is only worth doing if the code implements a general method, i.e. if the same code will solve several problems. If you have several such solutions in mind you can seperate the common element from them. Getting Press to produce the correct uncommon elements in each case then becomes the challenge. In this case it is not at all clear what the common and uncommon parts are.

Method 3 - Factorization

The third method we will consider can be found in both [Burnside and Panton 81] and [Barnard and Child 36]. It involves trying to factorize the equation and splitting into two simpler equations; one for each factor. A first step to update Press to deal with this method would be to implement a Factorization method in which the equation were put in the form s = 0 and rewrite rules of the form

---

[2]This quadratic turns up in 4 of the methods in some form or other. (4)

lhs => r1.r2

are then applied to s.

Such a rewrite rule with a fuzzy match to 2 is

$u^3 - v^3 \Rightarrow (u-v).(u^2 + uv + v^2)$

This method consists of the attempt to apply this rule. The hard bit is to rewrite 2 into the difference of two cubes so the rule will really match. The method works by guessing (magic) a more complex format than the difference of two cubes, then multiplying out and equating coefficients to flesh out the format completely. The solution in [Barnard and Child 36] puts less structure on the magic expression, so we will follow that.

We assume that we can find A, B, K and L such that $z^3 + 3hz + g$ can be put in the form

$$A(z-K)^3 - B(z-L)^3 \qquad (6)$$

Multiplying out, equating coefficients with 2, and solving the resulting four simultaneous equations for A, B, K and L in terms of h and g will yield the appropriate transformation of 2. (Incidently, the solution of the simultaneous equations generates the quadratic 4 as a byproduct.)

Where does the magic expression 6 come from? The situation is not as hopeless as it looks. The expressions to be matched against u and v must be at worst linear or the two resulting equations will not be simpler than the original: $A(z-K)^3$ is a trivial transformation of the cube of a general linear expression. A further check is that the equating of coefficients can be expected to yield four simultaneous equations, so the magic expression should introduce at most four variables to be solved for.

Method 4 - Symmetric Functions of the Roots

If l, m and n are the roots of 2 then they are connected to the coefficients 3h and g by the symmetric root equations:

$l+m+n = 0$
$lm + mn + nl = 3h$
$lmn = -g$

Trying to solve these equations for l, m and n will do no good: it will yield the original equation 2 with l, m and n substituted for z. Method 4 consists of manipulating the above equations into 3 linear simultaneous equations in l, m and n and then solving these (which does work!).

We already have one linear equation, namely

$l + m + n = 0$

The trick is to find another 2. The first piece of magic here is to pick the linear equations

$l + wm + w^2n = P$
$l + w^2m + wn = Q$

where w is a cube root of unity and P and Q are expressions in h and g to be determined. Where this particular choice comes from I have no idea, except that the combination of the 3 equations is particularly easy to solve

simultaneously.

Solving for P and Q is another piece of magic, which can be handwaved if we start from the assumption that we are going to end up solving a quadratic in P. In this case, if P and Q are the roots of the quadratic then $-(P+Q)$ and PQ will be the linear and constant coefficients respectively. In fact, PQ is independant of w, but $P+Q$ is not. However, P^3+Q^3 is independant, so we can get a quadratic in P^3. The actual equations are

$$P^3 + Q^3 = -27g/l^3$$
$$PQ = -9h/l^2$$

Hence, P^3 and Q^3 are the roots of the quadratic

$$t^2 + (3/l)^3gt - (9h/l^2)^3 = 0$$

which is essentially 4. These values can be substituted back in the linear simultaneous equations above and these in turn solved for l, m and n.

This seems the most mysterious of the 5 methods, and I hold no hope of ever automating it.

Method 6 - Collection

We finish with the method in [Barnard and Child 36], which I have not found elsewhere. This is our big success story. On careful examination I discovered that it was essentially an application of Collection requiring some heavy pattern matching. So I tried Alan Borning's pattern matcher on it and succeeded modulo a magic substitution. What follows is the Press version of events.

Since 2 contains two occurrences of z we will try to collect them into one. A suitable Collection rewrite rule is

$$\cos^3 u - 3/4.\cos u \Rightarrow 1/4 \cos 3u$$

Apart from the coses, this has a strong resemblance to 2. The cos signs would almost certainly make it fail the fuzzy match and neither will it Borning match to 2 as it stands. What is needed is the magic substitution

$$v = w.\cos u$$

into the rule.[3] This produces the new rule

$$v^3/w^3 - 3v/4w \Rightarrow 1/4 \cos 3.\arccos v/w$$

This rule both fuzzy matches and Borning matches 2 and armed with it Press selects it as the first thing to try; Collects successfully and goes on to Isolate the resulting equation. Traces on request.

[Barnard and Child 36] say that this only produces solutions for the cases when all three roots of the cubic are real. This is true provided you stick to defining cos only on the reals, but if you extend this definition in the obvious way to the complex numbers then the method yields solutions for all types of roots.

---

[3] [Barnard and Child 36] make the substitution $z = w.\cos u$ into 2 and this possibility also needs exploring.

As for eliminating the magic substitution I am open to suggestions. One obvious route would be to try to extend the Borning matcher so that it can use the rule in its original trigonometric form. This might happen on a second pass after the original rule had floped the fuzzy match (mumble test) on the first pass.

Conclusion

The next step seems to be to implement equating coefficients and the preprocessing reduction. Following this I will worry about the magic part of the Collection method and implement one of the other methods, e.g.the Factorization one. After such a good start I am optimisitic about final victory.

REFERENCES

[Barnard and Child 36]
Barnard and Child.
Higher Algebra.
MacMillan, 1936.

[Burnside and Panton 81]
Burnside, W.S. and Panton, A.W.
The theory of equations.
Longmans, Green & Co., 1881.

[Cardan 68]
Cardano G.
The Great Art or the rules of Algebra.
The MIT Press, 1968.
Translated from the Italian (Ars Magna 1545) by Witmer,

Feb 18
Here's a copy of a net message I sent you yesterday, in case you can't log into the net for a while. Alan

Hi Alan,

I got your net message, and also your letter along with a copy of Note 74; and have mailed a draft of the IJCAI paper off to you on the 13th.

I've been thinking about how to get PRESS to discover the magic substitution needed to solve the cubic using the trig collection rule. One idea is to allow a change of unknown during fuzzy matching. This will result in a guesss as to the form of the substitution, the details to be filled in during the full match. This idea is still half baked, but seems worth looking into.

The computation might proceed as follows. Let PRESS first try to solve the equation using the regular matcher, and then using fuzzy matching and the powerful matcher. If these fail, then try fuzzy matching the expression in question against each collection rule, allowing a change of unknown in either the expression or the pattern (at the fuzzy match level!). Suppose that the cubic has already been reduced to z^3 + 3hz + g = 0. The features term of the LHS is z^3 + z. Let PRESS try a fuzzy match against the trig rule cos^3 u - 3/4 cos u -> 1/4 cos 3u, whose features term is cos^3 u + cos u, with a change of unknown allowed. Find each proper subexpression of the trig expression such that the subexpression has a single occurence of u and such that there is more than one occurence of the subexpression in the pattern. Try substituting the subexpression for the unknown in the original expression, and see if the features match. One such subexpression of the pattern is cos u. If we substitute z=cos u into the cubic expression, then it fuzzy matches the pattern part of the rule.

The next step is to find the actual substitution. The features extraction algorithm drops "mumble" factors and terms from products and sums. Therefore, let us allow for these by postulating the substitution

z = c*cos(a*u+b) + d.

(The program should be able to leave out some of these pattern variables - see below.) Substitute this into the original equation, resulting in

(c*cos(a*u+b)+d)^3 + 3*h*(c*cos(a*u+b)+d).

After normalization this is

c^3*cos^3(a*u+b) + 3*c^2*cos^2(a*u+b)*d + (3*c*d^2+3*h*c)*cos(a*u+b) + 3*h*d + d^3.

Using the powerful matcher to match this against the LHS of the collection rule, first match the two terms involving cos^3. Divide both sides of the rule by c^3, and substitute a=1 and b=0. Since there is no cos^2 term in the rule, set d=0 to handle the second term. By this time the third term is (3*h*c/c^3)*cos(u). Match this and -3/4 cos u by solving algebraically for c, resulting in c = 2*sqrt(-h). This completes the match.

It would be better not to introduce superfluous pattern variables in the change of unknown. Of the four new variables in z = c*cos(a*u+b) + d, the program might reason that "a" and "b" are unnecessary, since the cos terms in the pattern are both of the form cos u.
Or, the features extraction algorithm could be changed so that it didn't drop "mumble" terms and factors inside of other functions, since these will in general not be easy to dispose of. If this is done, then the program wouldn't introduce the additional variables a and b.

In the above solution, the change of unknown is applied to the expression rather than to the pattern. This could be done the other way as well, but more simplification would be needed, in that the arccos(cos(u)) terms would need to be simplified before the match would work. Also, more care would be needed in avoiding the introduction of superfluous pattern variables (otherwise the simplification doesn't go

through.)

The most dubious part of the solution is the random search for possible substitutions ... a mitigating factor is that the search takes place at the fuzzy level, where the expressions are simpler and hence the combinatorics are not as bad.

An alternative way of approaching this might be to expand the definition of "features", allowing an expression to have several possible features terms. The program could check for possible changes of unknown in computing different features terms. In the case at hand, the LHS of the collection rule might have as its features cos^3 u + cos u, and also

    z^3 + z where z=cos u.

Something I neglected to mention in the cover letter with the IJCAI paper draft ... there is an inconsistency in regard to pattern variables. In the normal PRESS matcher, these are PROLOG variables; but in the powerful matcher, they are instantiated to generated symbols (e.g. g1) so that the matcher can solve algebraically for the value of a pattern variable. I thought it better not to burden the reader with a description of this hack, and therefore talked about even the regular PRESS matcher in terms of substitution, ignoring the use of PROLOG variables and unification. The alternatives are to describe the system as it is (which is harder to understand), or to always use PROLOG variables for pattern variables (in which case it's hard to see how to solve for them algebraically). I am not completely satisfied with any of these alternatives -- what do you think?

Best regards,
  Alan
@

Alan

DEPARTMENT OF ARTIFICIAL INTELLIGENCE
UNIVERSITY OF EDINBURGH

DAI Working Paper No. 67
Date: May 1980

Title: A Powerful Matcher for Algebraic Equation Solving

Author: Alan Borning

Abstract

This paper describes a powerful algebraic matcher for applying rewrite rules in equation solving. The matcher knows about the commutativity and associativity of addition and multiplication, will provide defaults for missing summands and factors, and if necessary will solve algebraically for the value of pattern variables.

Keywords

equation solving, symbolic manipulation, matching, rewrite rules

Acknowledgements

The author would like to thank all the members of the Mecho group for help and encouragement with this research. Computing resources were provided by SRC grant number GR/A 57954. The author is supported by a NATO Postdoctoral Fellowship from the National Science Foundation, USA.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

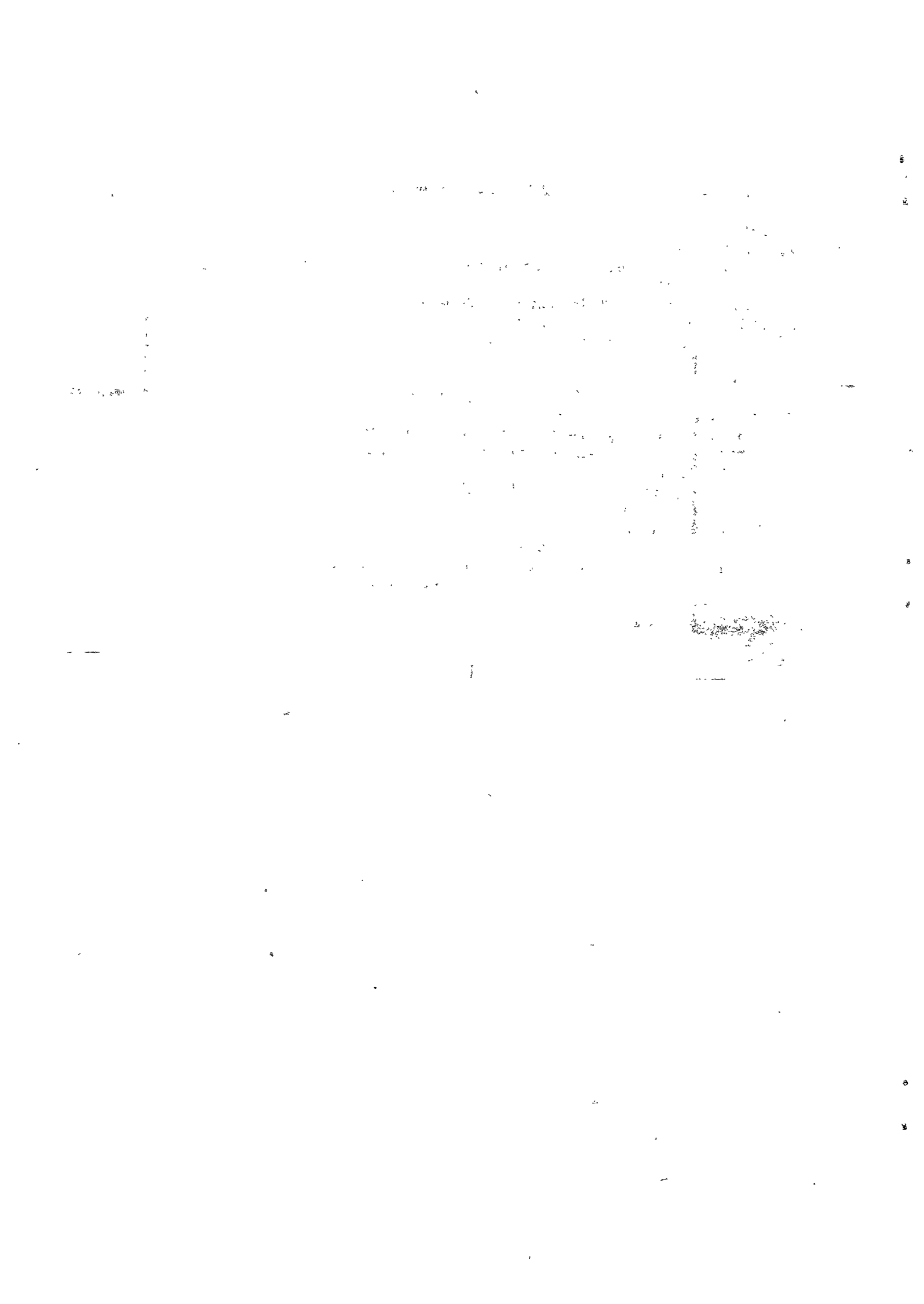


Table of Contents

1. Introduction

This paper describes a powerful matcher for use in algebraic equation solving.

The matcher is an extension to the PRESS algebra system, a computer program for solving equations and inequalities and for simplifying expressions [Bundy and Welham 79], [Bundy and Welham ng]. PRESS, as well as the matcher described here, are written in PROLOG [Pereira et al 78]. PRESS uses multiple sets of rewrite rules, selectively applied. Some of the important rewrite rule sets are:

isolation — PRESS tries applying isolation rules when there is a single occurence of the unknown in the equation. Isolation rules are applied to strip away surrounding functions and operators from the unknown, finally resulting in an equation with the unknown on one side by itself, and some expression (free of the unknown) on the other. A typical isolation rule is

```
arcsin(X)=E  ->  X=sin(E).
```

collection — Collection rules serve to reduce the number of occurences of the unknown, so that isolation can be applied. A typical collection rule is

```
U*W + V*W  ->  (U+V)*W
```

which collects relative to W.

attraction — Attraction rules move occurences of the unknown closer together in the expression tree, so that perhaps a collection rule can be applied. A sample rule is

```
log(B,U)+log(B,V)  ->  log(B,U*V).
```

To apply a rewrite rule to an expression, PRESS uses a matcher that extends PROLOG unification by building in the commutativity and associativity of addition and multiplication. For example, to apply the collection rule

```
U*W + V*W  ->  (U+V)*W
```

to the expression

```
x*y + z*(3*x)
```

in order to collect the two occurences of x, the PRESS matcher would unify W with x, U with y, and V with 3*z. The result of applying the rule would be

```
(y+3*z)*x.
```

The matcher used the commutativity and associativity of multiplication in accomplishing the match. (Note: following the standard PROLOG convention, names beginning with a capital letter represent variables, while those starting with a lower case letter represent atoms.)

An application of collection is the crucial step in the solution of some equations. Reflecting this, human mathematicians will try quite hard to find and apply a collection rule to an expression. For example, the standard solution of the general quadratic equation

```
a*x^2 + b*x + c = 0
```

uses the collection rule

```
U^2 + 2*U*V + V^2  -> (U+V)^2.
```

All the other steps of the solution are either preparations for applying the rule, or subsequent isolation steps. (However, a different terminology is usually used -- rather than talking about rewrite rules, mathematicians usually talk about identities. Also, the process of applying the above collection rule is often presented in "compiled form" as the operation of completing the square.) Similarly, the standard solution of the trigonomtric equation

```
a*sin(x) + b*cos(x) = c
```

depends critically on the use of the rule

```
cos(U)*sin(V) + sin(U)*cos(V)  ->  sin(U+V).
```

(Descriptions of the solutions of these equations may be found in [Tranter 70].)

However, the application of these rules -- matching the left hand side of a rule with an expression -- cannot be accomplished simply by using pattern matching and information about associativity and commutativity. What additional techniques are required? As part of an investigation of this question, an experimental matcher has been embedded in PRESS that can, among other things, solve both of the above equations.

2. Two Examples

Before plunging into the details of the matcher, its operation will be illustrated by two examples. Annotations to the program trace added by the author are preceeded by five asterisks.

2.1. Deriving the Solution for the General Quadratic Equation

```
| ?- try_hard_to_solve( a*x^2+b*x+c=0 , x , Ans ).

***** The user asks the program to solve the general quadratic
***** equation.  The "try_hard_to_solve" command informs the program
***** that it can use the powerful matcher in the solution process.

Solving a*x^2+b*x+c=0 for x

trying to use powerful matcher to collect x in
    a*x^2+b*x+c
```

***** PRESS notices that there are two occurences of the unknown in
***** the equation. It tries to collect the two occurences of x.

features of expression are x^2+x
looking for a collection rule with matching features

***** The program looks for an applicable collection rule. It first
***** does a cheap "fuzzy match" between the expression and the
***** pattern part of a potentially applicable collection rule. To do
***** this, it extracts some features from the expression, and
***** searches for a rule with matching features.

trying to apply rule x^2+2*x*g1+g1^2 -> (x+g1)^2
   to a*x^2+b*x+c

***** One of the collection rules is
*****   U^2 + 2*U*V + V^2 -> (U+V)^2
***** which collects relative to U. When U is unified with the unknown
***** x, and V with the generated symbol g1, the features of the rule
***** are also
*****   x^2+x.
***** The program selects this rule and tries to apply it to the left
***** hand side of the equation. Note: the symbol g1 was generated by
***** the program. V can't be left as a PROLOG variable, since the
***** matcher may have to solve algebraically for g1, and the equation
***** solver wouldn't work with a non-ground expression. This is a
***** hack ...

trying to match plus bags for expression a*x^2+b*x+c
   and pattern x^2+2*x*g1+g1^2
picking terms from expression & pattern bags and trying to match them

***** The powerful matcher is invoked to match the pattern part of the
***** rule with the quadratic expression. Since the principal
***** operator of both the expression and pattern is +, the matcher
***** converts to a bag representation. There are a number of ways in
***** which two bags can be matched. The matcher tries one of these
***** methods: picking a term from each bag and matching those two
***** terms. To pick the pair of terms to try matching, the matcher
***** first uses a simple complexity metric to choose the most complex
***** term from the expression. It finds the features of this term,
***** and then chooses a term with matching features from the rule.

trying to match times bags for expression a*x^2
   and pattern x^2

***** The matcher has now been called recursively on the first terms
***** in each sum. Since the principal operator of a*x^2 is times,
***** the matcher converts both terms to a bag representation. (The
***** x^2 term is converted to a times bag with one element.)

picking terms from expression & pattern bags and trying to match them

```
trivially matching x^2 and x^2 .
trying to match times bags for expression a
   and pattern <empty bag>
dealing with term a
   by applying a function to each side of the rule
match succeeded on expression a*x^2 and pattern x^2
returning transform:
   * a

***** The matcher picks the x^2 terms from each bag, and matches them
***** trivially. After that, however, it must match the expression
***** bag, which still has the "a" left in it, with the now empty
***** pattern bag. The previously used strategy of picking a term
***** from each product is no longer applicable. Instead, the matcher
***** decides that the "a" should be dealt with by multiplying both
***** sides of the rule by "a". This result is returned as a
***** transform.

applying transform to remaining terms in pattern bag
   yielding 2*x*g1*a+g1^2*a

***** The two remaining terms in the pattern bag are multiplied by
***** "a". The matcher is now called recursively on the remaining
***** parts of the expression and rule.

trying to match plus bags for expression b*x+c
   and pattern 2*x*g1*a+g1^2*a
picking terms from expression & pattern bags and trying to match them
trying to match times bags for expression b*x
   and pattern 2*x*g1*a

***** The matcher now tries matching the two terms containing x.

picking terms from expression & pattern bags and trying to match them
trivially matching x and x
trying to match times bags for expression b
   and pattern 2*g1*a

***** The two x's have been trivially matched. The matcher will now
***** make several unsuccessful attempts to match b with a term from
***** the pattern, before trying the method of solving algebraically
***** for the value of g1. (See section 4.1 for a discussion of this
***** search.) The strategy previously employed of multiplying both
***** sides of the rule by some expression can no longer be used,
***** since doing so would invalidate the previously established match
***** of the x^2 terms.

picking terms from expression & pattern bags and trying to match them
trying to match expression b and pattern 2
match failed on b and 2
picking terms from expression & pattern bags and trying to match them
```

```
matching b and g1 by using substitution
returning transform:
   g1  ->  b

applying transform to remaining terms in pattern bag
   yielding 2*a

trying to match times bags for expression <empty bag>
   and pattern 2*a
bag match failed on <empty bag> and 2*a
picking terms from expression & pattern bags and trying to match them
trying to match expression b and pattern a
match failed on b and a

trying to solve for a variable
calling equation solver to solve for g1 in b=2*g1*a

Solving b=2*g1*a for g1

    2*g1=b*a^-1
        (by Isolation)

    g1=b*a^-1*2^-1
        (by Isolation)

Answer is :
X1
  where :
    X1 =  g1=b*a^-1*2^-1

using solution g1=b*a^-1*2^-1
match succeeded on expression b*x and pattern 2*x*g1*a
returning transform:
   g1  ->  b*a^-1*2^-1

***** The matcher has used the other principal matching strategy,
***** algebraically solving for the value of a pattern variable.  The
***** transform indicates a substitution for g1.  Now the first two
***** terms in the bags have been matched.

applying transform to remaining terms in pattern bag
   yielding (b*a^-1*2^-1)^2*a

trying to match plus bags for expression c
   and pattern (b*a^-1*2^-1)^2*a
dealing with term c
   by applying a function to each side of the rule
trying to match plus bags for expression <empty bag>
   and pattern (b*a^-1*2^-1)^2*a
dealing with term (b*a^-1*2^-1)^2*a
   by applying a function to each side of the rule
```

```
***** The last two terms don't match each other.  However, the matcher
***** can complete the match by adding the c term in the expression to
***** each side of the rule, and subtracting the (b*a^-1*2^-1)^2*a
***** term.

match succeeded on expression a*x^2+b*x+c and pattern x^2+2*x*g1+g1^2
returning transform:
   * a
   + c
   + -1*(b*a^-1*2^-1)^2*a
   g1  ->  b*a^-1*2^-1

***** The match is now complete.  The pattern will match the
***** expression if the transform listed is applied to it (multiply
***** each side of the rule by a, add c to each side, add the term
***** -1*(b*a^-1*2^-1)^2*a to each side, and substitute b*a^-1*2^-1
***** for g1).  The rule remains a valid collection rule after the
***** transform has been applied to each side of it.  So the transform
***** is applied to the replacement part of the rule, and the result
***** is substituted for the expression
*****     a*x^2+b*x+c.
***** After this substitution, there will be only one occurence of "x"
***** in the equation, which may therefore be solved by isolation.

x collected in a*x^2+b*x+c gives
    (x+b*a^-1*2^-1)^2*a+c+-1*(b*a^-1*2^-1)^2*a


    (x+b*a^-1*2^-1)^2*a+c+-1*(b*a^-1*2^-1)^2*a=0


    (x+b*a^-1*2^-1)^2*a+c=0+-1*(-1*(b*a^-1*2^-1)^2*a)
        (by Isolation)

    (x+b*a^-1*2^-1)^2*a=0+-1*(-1*(b*a^-1*2^-1)^2*a)+-1*c
        (by Isolation)

    (x+b*a^-1*2^-1)^2=(0+-1*(-1*(b*a^-1*2^-1)^2*a)+-1*c)*a^-1
        (by Isolation)

    x+b*a^-1*2^-1=((0+-1*(-1*(b*a^-1*2^-1)^2*a)+-1*c)*a^-1)^2^-1#
    x+b*a^-1*2^-1=-1*((0+-1*(-1*(b*a^-1*2^-1)^2*a)+-1*c)*a^-1)^2^-1
        (by Isolation)

    x=((0+-1*(-1*(b*a^-1*2^-1)^2*a)+-1*c)*a^-1)^2^-1+-1*(b*a^-1*2^-1)
        (by Isolation)

    x=-1*((0+-1*(-1*(b*a^-1*2^-1)^2*a)+-1*c)*a^-1)^2^-1
          +-1*(b*a^-1*2^-1)
        (by Isolation)

Answer is :
```

```
( X1 # X2 )
  where :
    X1 =  x=2^-1*-1*a^-1*b+(a^-2*b^2*2^-2+-1*c*a^-1)^2^-1
    X2 =  x=2^-1*-1*a^-1*b+-1*(a^-2*b^2*2^-2+-1*c*a^-1)^2^-1

[END OF TRACE]
```

The program has now solved the equation. The two roots, written in two-dimensional notation, are:

$$x = -\frac{b}{2a} + \sqrt{\frac{b^2}{2^2 a^2} - \frac{c}{a}}$$

or

$$x = -\frac{b}{2a} - \sqrt{\frac{b^2}{2^2 a^2} - \frac{c}{a}}$$

If the fractions in the root are put over a common demoninator, the answers simplify to the usual expressions. (The current program doesn't simplify fractions in this way, because nobody has gotten around to writing the necessary package.)

2.2. Solving a Trigonometric Equation

Another example will now be presented. Since there are many similarities to the previous one, there are only a few annotations, pointing out some interesting features of the solution.

```
| ?- try_hard_to_solve( a*sin(x)+b*cos(x)=c , x , Ans ).

Solving a*sin(x)+b*cos(x)=c for x
Trying to collect x in
    a*sin(x)+b*cos(x)
```

```
trying to use powerful matcher to collect x in
    a*sin(x)+b*cos(x)

features of expression are sin(x)+cos(x)
looking for a collection rule with matching features

trying to apply rule sin(x)*cos(g1)+cos(x)*sin(g1) -> sin(x+g1)
   to a*sin(x)+b*cos(x)

trying to match plus bags for expression a*sin(x)+b*cos(x)
   and pattern sin(x)*cos(g1)+cos(x)*sin(g1)
picking terms from expression & pattern bags and trying to match them
trying to match times bags for expression a*sin(x)
   and pattern sin(x)*cos(g1)
picking terms from expression & pattern bags and trying to match them
trivially matching sin(x) and sin(x)
trying to match times bags for expression a
   and pattern cos(g1)
dealing with term a
   by applying a function to each side of the rule
trying to match times bags for expression <empty bag>
   and pattern cos(g1)
dealing with term cos(g1)
   by applying a function to each side of the rule
match succeeded on expression a*sin(x) and pattern sin(x)*cos(g1)
returning transform:
   * a
   * cos(g1)^-1

***** The first term from the expression has been matched with the
***** first term from the pattern in much the same fashion as in the
***** quadratic example.

applying transform to remaining terms in pattern bag
   yielding cos(x)*sin(g1)*a*cos(g1)^-1

trying to match plus bags for expression b*cos(x)
   and pattern cos(x)*sin(g1)*a*cos(g1)^-1
picking terms from expression & pattern bags and trying to match them
trying to match times bags for expression b*cos(x)
   and pattern cos(x)*sin(g1)*a*cos(g1)^-1
picking terms from expression & pattern bags and trying to match them
trivially matching cos(x) and cos(x)
trying to match times bags for expression b
   and pattern sin(g1)*a*cos(g1)^-1

***** The cos(x) factors have been matched trivially. There will now
***** be several unsuccessful attempts to match b with a factor from
***** the pattern, before solving b=sin(g1)*a*cos(g1)^-1 for g1.
```

```
picking terms from expression & pattern bags and trying to match them
trying to match expression b and pattern sin(g1)
match failed on b and sin(g1)

trying to solve for a variable
calling equation solver to solve for g1 in b=sin(g1)

Solving b=sin(g1) for g1

    g1=arcsin(b)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  g1=arcsin(b)

using solution g1=arcsin(b)
solving for a variable succeeded in matching expression b
   and pattern sin(g1)
returning transform:
   g1  ->  arcsin(b)


applying transform to remaining terms in pattern bag
   yielding a*cos(arcsin(b))^-1

trying to match times bags for expression <empty bag>
   and pattern a*cos(arcsin(b))^-1
bag match failed on <empty bag> and a*cos(arcsin(b))^-1
picking terms from expression & pattern bags and trying to match them
trying to match expression b and pattern a
match failed on b and a
picking terms from expression & pattern bags and trying to match them
trying to match expression b and pattern cos(g1)^-1
match failed on b and cos(g1)^-1

trying to solve for a variable
calling equation solver to solve for g1 in b=cos(g1)^-1

Solving b=cos(g1)^-1 for g1

    cos(g1)=b^-1^-1
        (by Isolation)

    g1=arccos(b^-1^-1)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  g1=arccos(b^-1)
```

```
using solution g1=arccos(b^-1)
solving for a variable succeeded in matching expression b
   and pattern cos(g1)^-1
returning transform:
   g1  ->  arccos(b^-1)


applying transform to remaining terms in pattern bag
   yielding sin(arccos(b^-1))*a

trying to match times bags for expression <empty bag>
   and pattern sin(arccos(b^-1))*a
bag match failed on <empty bag> and sin(arccos(b^-1))*a

***** A successful attempt finally begins.

trying to solve for a variable
calling equation solver to solve for g1 in b=sin(g1)*a*cos(g1)^-1

Solving b=sin(g1)*a*cos(g1)^-1 for g1
Trying to collect g1 in
    sin(g1)*a*cos(g1)^-1

g1 collected in sin(g1)*a*cos(g1)^-1 gives
    a*tan(g1)

    b=a*tan(g1)


    tan(g1)=b*a^-1
        (by Isolation)

    g1=arctan(b*a^-1)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  g1=arctan(b*a^-1)

using solution g1=arctan(b*a^-1)
match succeeded on expression b*cos(x)
   and pattern cos(x)*sin(g1)*a*cos(g1)^-1
returning transform:
   g1  ->  arctan(b*a^-1)

*****  The  two  cos(x) terms have now been matched by solving for the
***** value of g1.    The  equation  solver  used  collection  in  the
***** process!  Note that the matcher used a particular rather than a
***** general solution for g1 (see section 5.4).
```

```
applying transform to remaining terms in pattern bag
   yielding <empty bag>

match succeeded on expression a*sin(x)+b*cos(x) and pattern
 sin(x)*cos(g1)+cos(x)*sin(g1)
returning transform:
   * a
   * cos(g1)^-1
   g1  ->  arctan(b*a^-1)


x collected in a*sin(x)+b*cos(x) gives
    sin(x+arctan(b*a^-1))*a*cos(arctan(b*a^-1))^-1


    sin(x+arctan(b*a^-1))*a*cos(arctan(b*a^-1))^-1=c


    sin(x+arctan(b*a^-1))*a=c*(cos(arctan(b*a^-1))^-1)^-1
        (by Isolation)

    sin(x+arctan(b*a^-1))=c*(cos(arctan(b*a^-1))^-1)^-1*a^-1
        (by Isolation)
        where n1 denotes an arbitrary integer.

    x+arctan(b*a^-1)=n1*180+-1^n1*
           arcsin(c*(cos(arctan(b*a^-1))^-1)^-1*a^-1)
        (by Isolation)

    x=n1*180+-1^n1*arcsin(c*(cos(arctan(b*a^-1))^-1)^-1*a^-1)+
          -1*arctan(b*a^-1)
        (by Isolation)

Answer is :
X1
  where :
    X1 =  x=-1^n1*arcsin(c*cos(arctan(b*a^-1))*a^-1)+
          -1*arctan(b*a^-1)+180*n1

[END OF TRACE]
```

The equation has now been solved. As often occurs, the solution gives spurious values for x, as well the correct values. Thus the solutions to a specific equation would need to be checked by substituting them back into the original equation.

These spurious solutions were introduced in the collection step

```
    sin(g1)*cos(g1)^-1  ->  tan(g1)
```

when solving for the value of g1. The standard textbook derivation avoids this. In the standard solution, one **simultaneously** finds k and theta such that

```
    a = k*cos(theta)  &  b = k*sin(theta).
```

Suitable (particular!) values are

```
    k = sqrt(a^2+b^2)  &  theta = arctan(b/a).
```

Thus

```
    a*sin(x)+b*cos(x) = k*cos(theta)*sin(x)+k*sin(theta)*cos(x)
        = k*sin(x+theta).
```

The equation

```
    k*sin(x+theta) = c
```

may now be solved by isolation.

For the matcher to do this would require that it be extended to solve simultaneously for the values of several pattern variables. Only a few modifications to the existing parts of the matcher would be required. However, it might be rather difficult to write methods for finding simple, particular solutions for simultaneous equations -- this requires more investigation.

3. Objects used by the Matcher

The matcher makes use of several kinds of objects (i.e. data structures and associated procedures.) Other parts of the program create and access these objects through the procedures, thus providing a degree of data abstraction.

3.1. Expression Descriptions

An **expression description** consists of an algebraic expression, along with some other descriptive information. The data structure for an expression description has the following format:

```
expr_description(Expr,Root,Unknown,PatternVars,Path)
 where
   Expr is the expression being considered
   Root is the root of the expression tree of which Expr
      is a subexpression
   Unknown is the unknown
   PatternVars is a list of the pattern variables in Root
   Path describes the path from Root to Expr
      It is used to decide when it is permissible to add
      or multiply each side of the rule by a term.
```

For example, the description

```
    expr_description( x^2 , a*x^2+b*x+c , x ,
        [] , [pair(first,*),pair(first,+)] )
```

represents the subexpression x^2 of the expression a*x^2+b*x+c in the unknown x. There are no pattern variables. The path indicates that the subexpression was the first term selected from a product, which was in turn the first term selected from a sum. (See section 5.3.)

Along with the data structure, procedures have been defined for accessing the parts of an expression description, creating new descriptions, returning copies of existing descriptions modified in various ways, and so forth.

3.2. Transforms

A **transform** is an object that represents functions, substitutions, and possibly a change of unknown to be applied to an expression description. Its data structure format is:

```
transform(FunctionList,SubstitutionList,New_Unknown)
 where
   FunctionList is a list of functions to be applied
      to the expression
   SubstutitionList is a list of substitutions to be applied
      to the expression
   New_Unknown is the new unknown if the transform includes
      a change of unknown, and is otherwise "false"
```

Again, procedures have been defined for accessing the parts of a transform, creating new ones, copying existing transforms, concatenating two transforms to form a third, and applying transforms to expression descriptions to yield a new description. (The operations within the transform are performed left to right.)

4. Search Control

The matcher has available a considerable range of strategies for accomplishing a match; some of these strategies, such as algebraically solving for the value of a pattern variable, can be expensive to use. Therefore, it is important that the search involved in accomplishing a match be tightly controlled. The main technique for doing this is the use of the fuzzy matcher to perform a preliminary check before invoking the full matcher. Fuzzy matching is used both for the initial selection of a collection rule, and for the selection of a pair of terms to match from two bags. Another technique for controlling search is the complexity heuristic for deciding which term in a bag to look at next. These techniques have proven to be quite powerful. As illustrated by the preceeding examples, most spurious matches are rejected during fuzzy matching, and little search is done using the full matcher.

To handle the search that does occur, the matcher uses the depth-first search provided by the built-in PROLOG backtracking mechanism, along with a memo procedure to save the results of matches in case they are needed again.

The current search control methods are for the most part adequate for matches that are eventually successful, and for matches that can't succeed (and are detected as such by the fuzzy matcher). The matcher takes considerably longer on matches that pass the fuzzy matcher, but eventually fail. For example, if one asks the system to find the solution to the general cubic equation

```
a*x^3 + b*x^2 + c*x + d = 0,
```

it will (reasonably enough) attempt to apply the collection rule

```
U^3 + 3*U^2*V +3*U*V^2 + V^3  ->  (U+V)^3.
```

This match eventually fails, but only after considerable backtracking.

A direction for future research with the matcher would be to explore other search control strategies, e.g. agendas and resource allocation mechanisms. The need for such strategies would become more acute if more alternatives for matching two expressions were added to the matcher, such as the method of simultaneously solving for the value of several pattern variables.

4.1. Reducing Search -- Some Unanswered Questions

One might argue that, at least in the two examples presented so far, no search at all involving the full matcher was really needed. In the quadratic example, such search took place when trying to match b with 2*gl*a. Matching b with gl was obviously silly, since the matcher would in any case be unable to deal with the remaining 2*a. At one point the program was in fact modified to take this into account. In picking a pair of terms from two bags, the matcher insisted that not only did the pair of terms match fuzzily, but the remainder of the two bags match fuzzily as well. With this modification, the search in question was eliminated, since the fuzzy match between <empty bag> and 2*a failed. The problem with this was that a great deal of additional information had to be built into the fuzzy matcher, since for example an empty bag would match a non-empty bag if each side of the rule could be multiplied by the appropriate terms, or if the non-emtpy bag contained a pattern variable. This information duplicated that in the bag matching procedures, making things very unmodular. Also, it was contrary to the goal of keeping the fuzzy matcher comparatively simple. So this modification was removed.

The author is unsure as to what ought to be done about this, if anything. One idea is to do more re-ordering of subgoals (in this case, to try matching <empty bag> with 2*a before matching b with gl).

5. The Matching Algorithm

The arguments to the matcher are as follows:

match(Expression_Description, Pattern_Description, Transform)

Expression_Description and Pattern_Description are descriptions of the expression and pattern to be matched. If the match is successful, Transform is unified with a transform that, if applied to the pattern, would make it algebraically equal to the expression.

When called, the matcher first checks for simple cases. If the expression and pattern are identical, the match succeeds trivially, and the null transform is returned. If the pattern consists solely of a pattern variable, the match succeeds again, and a transform consisting of the single substitution "variable -> expr" is returned.

Otherwise, the matcher must try harder. The matcher has two ways of accomplishing a non-trivial match: by recursively matching corresponding parts of the expression and the pattern, or by algebraically solving for the value of a pattern variable.

5.1. Preserving the Properties of the Rule

The matcher should preserve the properties of the rule whose pattern is being matched. This is handled in a somewhat ad hoc way at present -- the matcher always leaves the same number of occurences of the unknown on each side of the rule. Thus, in particular, collection rules will always remain valid collection rules after being matched (i.e. after the transform returned from a match has been applied to both sides).

5.2. Recursively Matching Parts of Expressions

In general, to match two complex expressions, the matcher will first check that the principal operators or functions are the same, and will then match the corresponding arguments. For example, consider matching the expression log(e,a), i.e. the log of a to the base e; with the pattern log(e,g2), where g2 is a pattern variable. The matcher first checks that the functions log are the same, and then calls itself recursively to match e with e, and a with g2.

The matcher knows about the commutativity and associativity of addition and multiplication. When matching two sums or products, the matcher puts all the terms in each sum or product into an unordered bag. It then has available the following alternatives in matching the two bags:

- If both bags are empty, the match succeeds trivially.

- The matcher can pick an term from each bag and call itself

recursively to match the two terms. In using this alternative, the matcher will pick the most complex term from one bag (using a simple complexity metric), and then will pick an appropriate term from the other bag. This other term is selected by performing a fuzzy match between the term from the first bag and candidate terms from the other bag.

- If there is just a pattern variable left in the pattern, and the expression bag is empty, then the matcher can try setting the variable to the identity element for the bag (0 for plus bags, 1 for times bags). This option is logically redundant, as the alternative of solving for a pattern variable would accomplish the same end. It is included at this point, however, as a cheap option to be tried before others such as adding or multiplying each side of the rule by some term.

- If there is just a pattern variable left in the pattern, and the expression bag contains some random terms, all free of the unknown, then the matcher can try substituting the expression's terms for the pattern variable. Again, this alternative is logically redundant, but is included here so that it will be tried before the others that follow.

- If the term in either the expression or the pattern is free of the unknown, the matcher can permit the match to succeed by adding or multiplying each side of the rule by a term, if applying the operation will not invalidate previously matched parts of the expression and pattern. This strategy was used, for example, to match the a*x^2 and x^2 terms in the quadratic. Whether or not this option can be used is determined by inspecting the path from the root to the expression. (See section 5.3.)

- If the pattern contains a pattern variable, the matcher can try to solve for its value algebraically. (See section 5.4 for more details.)

When matching a sum against any other expression (including a product), the matcher will convert the other expression into a plus bag with just the one element. Matching a product against any other expression (except a sum) is handled analogously.

5.3. Paths

A preliminary word of warning: this was one of the most difficult parts of the matcher to design and debug, and the whole thing could use re-thinking. It is only with reluctance that the author describes its current kludgiferous state.

As described in section 3.1, one of the parts of an expression description is a path from the root to the current expression. The path is represented as a list of pairs, each pair consisting of "first" or "other", followed by the expression's principal operator or function. As each new subexpression is considered, another pair is put on the front of the list from the previous description. One of the peculiarites of these paths is that they reflect the order in which terms are selected from bags by the matcher, rather than the order in which they occur in the original expression. In the quadratic example, the path to the root expression a*x^2+b*x+c is of course []. The path to a*x^2 is [pair(first,+)], since this is the first term selected from a plus bag. Similarly, the path to x^2 is [pair(first,*),pair(first,+)], since this is the first term selected from a times bag, which was the first term from a plus bag. The path to b*x is [pair(other,+)], because this was not the first term selected. Finally, the path to the x in b*x is [pair(first,*),pair(other,+)].

To decide whether a term can be added to both sides of the rule, or if both sides can be multiplied by something, the matcher looks at the path. The addition or multiplication is allowed if each operator on the path distributes over its successor on the path. (Naturally, multiplication distributes over addition.)

There was an attempt (probably misguided) to make this design general -- for example, the whole thing should work with logical operators "and" and "or".

5.4. Algebraically Solving for the Value of a Pattern Variable

The other principal technique for accomplishing a match is to solve algebraically for the value of a pattern variable. An equation is constructed whose two sides are the epxression and pattern to be matched.

In solving equations of this kind, a **particular** rather than a general solution is wanted. The equation solver is told about this by adding the assertion

```
particular_solution(PVar)
```

to the data base, where PVar is the pattern variable being solved for. An instance of this was seen in the trigonometic example given in section 2.2. There, when solving b=sin(g1)*a*cos(g1)^-1 for g1, the particular solution g1=arctan(b*a^-1) was used, rather than the general solution g1=n1*180+arctan(b*a^-1).

As mentioned previously, the unknown is given a special status, and this is reflected in some restrictions on this method. If no change of unknown is involved, the expression to be substituted for the pattern variable must be free of the unknown. (Otherwise, the rule would probably no longer serve as e.g. a valid collection rule.) On the other hand, if a change of unknown is involved, the variable being

solved for must be the new unknown, and the expression to be substituted for it must contain the new unknown. (The change of unknown example in section 7.3 will illustrate this.)

6. The Fuzzy Matcher

The fuzzy matcher is used to perform an inexpensive preliminary check on whether an expression and pattern may match. Thus, it must always succeed if the expression and pattern can be matched by the full matcher, and should fail on cases on which the full matcher would "obviously" fail. The fuzzy matcher computes the **features terms** of the expression and pattern, and then matches these using the normal PRESS matcher (which is comparatively inexpensive).

The algorithm used for extracting a feature term is as follows.

- If the expression is the unknown itself, then its feature term is the unknown as well.
- If the expression is free of the unknown, its feature term is the expression "mumble".
- To compute the feature term of a sum or product bag, the features of each term in the bag are found. All "mumbles" are discarded; the feature term is then a bag consisting of the remaining feature terms.
- Integer exponents of expressions not free of the unknown remain themselves. (Otherwise a fuzzy match of e.g. x^3+x and x^2+x would succeed, since the feature terms of both would be x^mumble+x. This was not desired.)
- The features of any other complex term are found by computing the features of each argument, and returning a new term with the arguments replaced by their corresponding features.

Here are some examples. In all cases the unknown is x.

Expression	Features
x	x
3*sin(b)+cos(a)	mumble
3*a*b*x^2 + 5	x^2
2*sin(x)+cos(x)^2	sin(x)+cos(x)^2

The algorithm for extracting a features term reflects the fact that

the matcher can often deal with miscellaneous expressions that are free of the unknown.

7. Applications of the Matcher

Several applications of the matcher have been programmed. Two are described in this section: compiling specialized methods for solving equations, and solving equations using a change of unknown.

The matcher has also been tried as the standard matcher in PRESS for applying collection rules. When run on a set of standard problems, the program solved them all, but ran at 1/3 the speed of the regular program. (Much of this is due to a kludgey interface between the matcher and the rest of PRESS, and could be reduced considerably.) In any case, the matcher is probably more powerful than necessary for most equation solving applications.

7.1. Compiling Specialized Methods for Solving Equations

The first of these applications is a procedure that compiles specialized methods for solving certain kinds of equations. The user gives the program the general form of an equation. The program solves the equation using the powerful matcher, and then asserts a new PROLOG procedure for solving instances of that equation.

The following command is used:

```
learn_to_solve(Normal_Form, Unknown, Equation, Conditions)
 where
   Normal_Form is the name of the normal form of the equation
   Unknown is the unknown
   Equation is the equation
   Conditions is a list of conditions on the symbols
        of the equation
```

The new procedure asserted by the program will execute as follows. First, the incoming equation will be put into the specified normal form. The normalized equation will then be matched against the normalized general equation using the standard PRESS matcher, and the conditions will be evaluated. If all this is successful, then the procedure simply tidies the solution to the general equation and returns it. (The versions of the general equation and its solution included in the asserted procedure have all symbolic quantities replaced by PROLOG variables, so that the matching and substitution into the solution are accomplished using PROLOG unification.)

For example, suppose the user issues the command

```
learn_to_solve( polynomial , x , a*x^2+b*x+c=0 , non_zero(a) )
```

The program will solve the equation as previously described, and will then assert a new procedure for solving quadratic equations. The use of polynomial normal form (rather than general normal form) ensures that equations such as

```
(x-3)*(x-4)=0
```

are recognized as quadratics, and also provides correct defaults for missing coefficients.

7.2. Problems Involving a Change of Unknown

Another application uses the matcher in solving equations using a change of unknown. In this method, the equation is matched against another equation whose solution is known. The match will involve a change of unknown. The method currently tries only to match against quadratics, but there should be no particular problem in extending it to try other kinds of equations.

First, an expression is constructed by subtracting the right hand side of the equation from its left hand side, and distributing products over sums. An inexpensive test is then performed to see if the match against the general quadratic might be successful, to weed out obviously losing attempts. The test consists of checking that the expression is a sum, with two terms containing the unknown, and one of them involving exponentiation. If the expression passes the test, it is then matched against the quadratic expression

```
a*x^2 + b*x +c
```

using the powerful matcher. The transform returned by the matcher can use only substitution and a change of unknown, but not function application. If the match is successful, the same transform is applied to the two solutions to the quadratic equation. Each of the resulting pair of equations will then have only one occurence of the original unknown, and is then solved by isolation.

This method is currently programmed in a rather counterintuitive fashion, in that the transform returned by the matcher will take the quadratic pattern and make it equal to the original expression. Humans, on the other hand, usually do it the other way round -- they take an equation and transform it into a quadratic. However, the end result is similar. Modelling the human behaviour would require that the matcher be able to return transforms that apply to the expression (if it's an equation), as well as to the pattern.

This change of unknown method is complementary to the change of unknown method used in the standard PRESS program. In that method, the program searches for a proper subexpression of the equation being solved such that there is more than one occurence of the subexpression in the equation, and such that all occurences of the unknown are contained within these subexpressions. If these conditions hold, the program will generate a new unknown, substitute it for each occurence of the subexpression, solve the new equation, and finally use this

solution to solve for the original unknown. Thus, the method makes no presuppositions about the form of the new equation. However, the subexpressions containing the unknown must each be exactly the same. (Thus the problem presented in section 7.3 could not be handled by this method.) Conversely, the method using the powerful matcher will only match the original equation against one of a particular form, but is more flexible in the transformations it can use.

7.3. A Change of Unknown Example

```
| ?- solve( 5^(2*y)-5^(y+1)+6=0 , y , Ans ).

Solving 5^(2*y)-5^(y+1)+6=0 for y

trying to use powerful matcher to collect y in
    5^(2*y)+-1*5^(y+1)+6

features of expression are mumble^y+mumble^y
looking for a collection rule with matching features

***** The program has just unsuccessfully attempted to find a suitable
***** collection rule.

trying change of unknown to make equation into a quadratic
trying to match plus bags for expression 5^(2*y)+-1*5^(y+1)+6
   and pattern a_zzz*x_zzz^2+b_zzz*x_zzz+c_zzz
picking terms from expression & pattern bags and trying to match them
trying to match times bags for expression 5^(2*y)
   and pattern a_zzz*x_zzz^2

***** The matcher has selected the first term from each plus bag.
***** When a change of unknown is involved, the fuzzy matcher's checks
***** are relaxed: two expressions match fuzzily if either the
***** expression contains the unknown and the pattern the new unknown,
***** or neither contain the unknown or new unknown.

picking terms from expression & pattern bags and trying to match them
trying to match expression 5^(2*y) and pattern x_zzz^2
matching 5 and x_zzz by using substitution
returning transform:
   x_zzz  ->  5

match failed on 5^(2*y) and x_zzz^2

trying to solve for a variable
calling equation solver to solve for x_zzz in 5^(2*y)=x_zzz^2

Solving 5^(2*y)=x_zzz^2 for x_zzz

    x_zzz=(5^(2*y))^2^-1#x_zzz=-1*(5^(2*y))^2^-1
```

```
        (by Isolation)

Answer is :
( X1 # X2 )
  where :
    X1 =  x_zzz=5^y
    X2 =  x_zzz=-1*5^y

using solution x_zzz=5^y
solving for a variable succeeded in matching expression 5^(2*y)
   and pattern x_zzz^2
returning transform:
   x_zzz  ->  5^y
   change unknown to y


applying transform to remaining terms in pattern bag
   yielding a_zzz

trying to match times bags for expression <empty bag>
   and pattern a_zzz
trying making a_zzz the bag identity element 1
match succeeded on expression 5^(2*y) and pattern a_zzz*x_zzz^2
returning transform:
   x_zzz  ->  5^y
   a_zzz  ->  1
   change unknown to y

***** The first terms have now been matched. The unknown in the
***** pattern is changed from x_zzz to y.

applying transform to remaining terms in pattern bag
   yielding b_zzz*5^y+c_zzz

trying to match plus bags for expression -1*5^(y+1)+6
   and pattern b_zzz*5^y+c_zzz
picking terms from expression & pattern bags and trying to match them
trying to match times bags for expression -1*5^(y+1)
   and pattern b_zzz*5^y
picking terms from expression & pattern bags and trying to match them
trying to match expression 5^(y+1) and pattern 5^y
trivially matching 5 and 5
trying to match plus bags for expression y+1
   and pattern y
picking terms from expression & pattern bags and trying to match them
trivially matching y and y
trying to match plus bags for expression 1
   and pattern <empty bag>
bag match failed on 1 and <empty bag>
bag match failed on y+1 and y
match failed on y+1 and y
match failed on 5^(y+1) and 5^y
```

```
trying to solve for a variable
calling equation solver to solve for b_zzz in -1*5^(y+1)=b_zzz*5^y

Solving -1*5^(y+1)=b_zzz*5^y for b_zzz

    b_zzz=-1*5^(y+1)*(5^y)^-1
        (by Isolation)

Answer is :
X1
  where :
    X1 =  b_zzz=-5

using solution b_zzz=-5
match succeeded on expression -1*5^(y+1) and pattern b_zzz*5^y
returning transform:
   b_zzz  ->  -5

***** The second terms have now been matched.

applying transform to remaining terms in pattern bag
   yielding c_zzz

trying to match plus bags for expression 6
   and pattern c_zzz
substituting 6 for c_zzz
match succeeded on expression 5^(2*y)+-1*5^(y+1)+6 and pattern
 a_zzz*x_zzz^2+b_zzz*x_zzz+c_zzz
returning transform:
   x_zzz  ->  5^y
   a_zzz  ->  1
   b_zzz  ->  -5
   c_zzz  ->  6
   change unknown to y



applying transform to solution to quadratic equation yielding
( X1 # X2 )
  where :
    X1 =  5^y=3
    X2 =  5^y=2


Solving 5^y=2#5^y=3 for y

    y=log(5,2)
        (by Isolation)

    y=log(5,3)
        (by Isolation)
```

```
Answer is :
( X1 # X2 )
  where :
    X1 =  y=log(5,3)
    X2 =  y=log(5,2)

[END OF TRACE]
```

8. Related Work

To the best of the author's knowledge, no other algebraic matcher has been implemented that compares in power to the one described here.

A powerful matcher, upon which the present work is based, is proposed in [Bundy 75]. (This paper also describes many of the basic ideas in the PRESS program.) The PRESS program as currently implemented includes a matcher that knows about the commutativity and associativity of addition and multiplication. The matcher in MACSYMA ([Fateman 72], [Mathlab 77]) knows about commutativity and associativity. It also provides defaults for missing summands, factors, and exponents; and will distribute products over sums to accomplish a match. However, in the author's opinion, the normal form mechanisms of PRESS provide a cleaner way of specifying these additional features. For efficiency, the MACSYMA matcher compiles patterns into LISP programs, rather than using an interpreter.

A survey of the state of the art in matching and unification problems is in [Raulefs et al 78]. An algorithm for matching under commutativity that improves over the naive solution is presented in [Siekmann 79], along with some formal discussion.

REFERENCES

[Bundy and Welham 79]
Bundy, A. and Welham, B.
Using meta-level descriptions for selective application of multiple rewrite rules in algebraic manipulation.
Working Paper 55, Dept. of Artificial Intelligence, Edinburgh, May, 1979.

[Bundy and Welham ng]
Bundy, A. and Welham, B.
Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation.
Artificial Intelligence , forthcoming.

[Bundy 75]
Bundy, A.
Analysing Mathematical Proofs (or reading between the lines).
In Winston, P., editor, Procs of the fourth. IJCAI, Georgia, 1975.
An expanded version is available from Edinburgh as DAI Research Report No. 2.

[Fateman 72]
Fateman, R.J.
Essays in Algebraic Simplification.
PhD thesis, MIT, April, 1972.
also available as MAC TR-95.

[Mathlab 77]
Mathlab Group.
MACSYMA Reference Manual.
Technical Report, MIT, 1977.

[Pereira et al 78]
Pereira, L.M., Pereira, F.C.N. and Warren, D.H.D.
User's guide to DECsystem-10 PROLOG.
Internal Memo, Dept. of Artificial Intelligence, Edinburgh, 1978.

[Raulefs et al 78]
Raulefs, P.,Siekmann, J.,Szabo, P. and Unvericht, E.
A short survey on the state of the art in matching and unification problems.
AISB Quarterly issue 32:pp17-21, December, 1978.

[Siekmann 79]
Siekmann, J.
Unification of Commutative Terms.
In Ng, E.W., editor, Symbolic and Algebraic Computation, pages 531-545. Springer-Verlag, 1979.

[Tranter 70]
Tranter, C.J.
Advanced Level Pure Mathematics.
English Universities Press, 1970.