Prof. Dr. H. Stoyan
UNIVERSITÄT ERLANGEN - NÜRNBERG
INSTITUT FÜR MATHEMATISCHE MASCHINEN
U. DATENVERARBEITUNG (INFORMATIK VIII)
AM WEICHSELGARTEN 9 · 8520 ERLANGEN

STANFORD ARTIFICIAL INTELLIGENCE PROJECT December 27, 1963
Memo No. 11

AN ALGEBRAIC SIMPLIFY PROGRAM IN LISP

by Dean Wooldridge Jr.

Abstract: A program which performs "obvious" (non-controversial) simplifying transformations on algebraic expressions (written in LISP prefix notation) is described. Cancellation of inverses and consolidation of sums and products are the basic accomplishments of the program; however, if the user desires to do so, he may request the program to perform special tasks, such as collect common factors from products in sums or expand products. Polynomials are handled by routines which take advantage of the special form of polynomials; in particular, division (not cancellation) is always done in terms of polynomials. The program (run on the IBM 7090) is slightly faster than a human; however the computer does not need to check its work by repeating the simplification.

Although the program is usable - no bugs are known to exist - it is by no means a finished project. A rewriting of the simplify system is anticipated; this will eliminate much of the existing redundancy and other inefficiency, as well as implement an identity-recognizing scheme.

The research reperted here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183)

Contents

LISP SIMPLIFICATION

The formidable lengths and complexities of the algebraic expressions with which engineers and physicists must often deal demonstrates the need for machines which can transform such expressions into forms which may be more conveniently handled. Although heuristics play an important part in simplifying arithmetic expressions, there is a large amount of "dogwork" involved which could quickly and efficiently be done by a computer. In this paper is described a LISP 1.5 program, called Simplify, which applies various simplifying transformations to arbitrary arithmetic expressions.

Simplify and its adjuncts were written to be used on-line in a time-sharing system. The user is not expected to be familiar with LISP. However, this memo consists of a rather detailed description of the program and does, therefore, assume familiarity with the contents of the LISP 1.5 manual (a non-LISP "user's manual" for the simplify program will be available at a later date). In fact, the purpose of this paper is to describe the system in sufficient detail to qualify the reader to improve the system (and, in the likely event that bugs still exist, to debug the system).

Comparison of descriptions in this memo with the listings of the sections of the program being described will frequently yield rather gross discrepancies. If the differences cannot be attributed to the failure of this memo to discuss all the tedious details of bookkeeping, then the differences are related to features of the program which are described elsewhere in the report.

The Language of Simplify

The language understood and used by simplify is the LISP S-expression notation. The meanings of the atoms which represent arithmetic operations are the same as the meanings given in the LISP 1.5 manual; viz. if e^* is the S-expression corresponding to the Algolic expression e, then:

$(a + b + . . . + z)^*$ = (PLUS a^* b^* . . . z^*);

$(-a)^*$ = (MINUS a^*);

$(a - b)^*$ = (DIFFERENCE a^* b^*);

$(a \times b \times . . . \times z)^*$ = (TIMES a^* b^* . . . z^*);

$(a^{-1})^*$ = (RECIP a^*);

$(a / b)^*$ = (QUOTIENT a^* b^*); and

$(a^b)^*$ = (EXPT a^* b^*).

Simplify also has the ability to handle polynomials, which are represented as follows:

$(\sum^n a_k x^k)^*$ = (POLY x^* a_n^* . . . a_0^*).

Polynomials will be discussed in more detail in a later section of this memo.

In this paper, as in the LISP Programmer's Manual, we shall use lower case letters as variables whose values are S-expressions. A symbol whose value is its name is written in upper case, rather than being quoted: e.g., "POLY" corresponds to the S-expression (QUOTE POLY), etc.

Simplify

The main portion of the simplify program consists of two supervisor functions, simplify and simp, and nine "simp" functions: simpatom, simplus, simpminus, simpdifference, simptimes, simprecip, simpquotient, simpexpt, and simpoly.

Simpatom(s) terminates recursion for simplify by returning cons(s;NIL); it has, however, another reason for existence, and this will be discussed in the section on the "descriptor lists" feature of the program.

The argument of simplify is either an atom, in which case simpatom is called, or an expression of the form:

(op a ... z), in which case simplify calls simp:

simp[op;conc[simplify[a]; . . . ;simplify[z]];(op a . . . z)].

Thus, simplify simplies its arguments on all levels, starting at the lowest.

Simp applies to its second argument, which is a list of terms or factors, the function whose name is stored on the property list of op after the indicator FSIMP; e.g. if op = RECIP, then the value of simp is simprecip[(a)]. If the simplified expression turns out to be the same as the original expression, which is carried along as the third argument of simp, then the original expression is returned, rather than clogging the memory with duplicating list structures. If FSIMP does not appear on the property list of op, then simp returns ((op a ... z)).

Each of the simp functions, except simpatom and simpoly, looks for for the occurrence of the atom UNDEFINED in its argument (the simp functions have only one argument), s. If UNDEFINED is found, then the simp function returns

cons[cons[UNDEFINED;cons[op;s]];NIL]. Expressions not containing UNDEFINED which, nevertheless, give rise to UNDEFINED expressions are:

(RECIP 0);

(QUOTIENT x 0); and

(EXPT 0 0).

(Note that, since simplify begins simplifying on the lowest level of the input list, if simplify[x] = (0), then simplify[(RECIP x)] = ((UNDEFINED RECIP 0)). The purpose of the extra pair of parentheses in the outputs of simplify and the simp functions will be discussed in the section on descriptor lists.)

The following descriptions of the operations of the simp functions (excluding simpatom, which was described above and will be discussed again later, and simpoly, which will be described in the section on polynomials, assume that the argument, s, does not contain the atom UNDEFINED.

Simpminus[s] returns the list whose only element is given by:

```
if s = (n) and numberp[n] then minus[n];
if s = ((MINUS a) then a;
if s = ((DIFFERENCE a b)) then
    car[simpdifference[list[b;a]]];
if s = ((POLY x $a_n$ . . . $a_0$)) then
    (POLY x car[simpminus[$a_n$]] . . . car[simpminus[$a_0$]]);
otherwise cons(MINUS;a).
```

Simpdifference[s], where s = (a b), has the value:

simplus[cons[a;simpminus[b]]].

Simprecip[s] returns the list whose only element is:

```
if s = (n) and numberp[n] then:
    if zerop[n] then (UNDEFINED RECIP 0);
    otherwise recip[n];
if s = ((RECIP a) then a;
if s = ((EXPT a b)) then
    (EXPT a car[simpminus[b]]);
if s = ((MINUS a)) then
    car[simpminus[simprecip[cons[a;NIL]]]]
        (the minus sign is brought to the outside of the
        expression);
otherwise cons[RECIP;s].
```

Simpquotient[s], where s = (a b), calls polyquotient to perform the division a / b. If the remainder thus produced is zero (or a zero polynomial; (POLY x 0 0 ... 0)), then simpquotient returns the list whose only element is the quotient. If the division produces a non-zero remainder, then simpquotient[s] = simptimes[cons[a;simprecip[b]]].

In the following discussions of simpexpt, simplus, and simptimes, we shall omit mention of the effects of the "expand" and "factor" features on the operations of these functions. Expand and factor will be discussed in detail later in the paper.

The value of simpexpt[s] is the list whose only element is given by:

```
if s = ((EXPT a b) e f , , , z) then
      car[simpexpt[cons[a;simptimes[list[b;e]]]]];
if s = (0 a) b . . .. z) then
      if zerop[a] then (UNDEFINED EXPT 0 0);
      otherwise 0;
if s = (1 a) then 1;
if s = (a n) and numberp[n] then:
      if zerop[n] then 1;
      if onep[n] then a;
      if numberp[n] then expt[a;n];
      if minusp[n] then
          car[simprecip[simpexpt[list[a;minus[n]]]]];
      otherwise cons[EXPT;s];
  otherwise cons[EXPT; s].
```

Simplus[s] initializes the program variables n and tm by setting them to 0 and NIL, respectively. n is used to store the sum of all numeric terms in the argument of simplus, and tm holds the "term list", which is a list of listed pairs: the car of each pair is a term from the argument of simplus, and the cadr is the number of occurrences of that term in the argument. For example, the term list which would be generated in response to the list (A (TIMES B 3) (PLUS A B C)) is:

((C 1) (B 4) (A 2)). We shall consider in some detail the functions which generate the term list: collectorl and collect.

Collect makes use of the predicate mequal[x;y], which is true if equal[x,y] or if x and y are both sums(or both products-- more. generally, both expressions whose cars are eq and which have the flag COMMUTE on their property lists) and contain the same terms, though not necessarily in the same order. Collect[e;m;s] searches the list s until it finds an element whose car is mequal to e. The cadr of that element is increased by m, and collect terminates. If no such element of s is found, then the value of collect is cons[list[e;m];tm]. Thus, collect brings up to date the record of the number of occurrences of the sub-expression e in the expression being simplified--that record being stored on the term list, tm. For example, if the expression

(PLUS A (TIMES 2 B) A 3 C) is to be simplified (assume that the second level has already been simplified), then simplus initializes n to 0 and tm to NIL, and calls

collectorl[(A (TIMES 2 B) A 3 C);1], which causes

collect[A;1;NIL] to be executed. The result is a new value for tm: ((A 1)). Now collectorl calls itself: collectorl[((TIMES 2 B) A 3 C);1], which again calls collect, etc., until the original list of terms is exhausted. The growth of the term list may be summarized as follows:

collector arguments	collect arguments	results	
		n	tm
(A (TIMES 2 B) A 3 C), 1	A, 1, NIL	0	((A 1))
((TIMES 2 B) A 3 C), 1	(TIMES 2 B), 1, ((A 1))	0	((B 2)(A 1))

(A 3 C), 1	A, 1,((B 2)(A 1))	0	((B 2)(A 2))
(3 C), 1	3, 1, ((B 2)(A 2))	3	((B 2)(A 2))
(C), 1	C, 1, ((B 2)(A 2))	3	((C 1)(B 2)(A 2))
Nil,1	Return to simplus.		

The value of collector1[r;mm], where r is initially the list, s, of the terms of the sum being simplified, is:

```
prog 2[

  (if null[r] then NIL;
   otherwise:
     if numberp[car[r]] then
       setq[n;plus[times[n;mm];car[r]]]
         (recall that the purpose of n is to hold the value of the
         numeric portion of the expression; the purpose of mm will
         be made clear below):
     if atom[car[r]] then collect[car[r];;mm;tm];
     if eq[caar[r];PLUS] then collector[cdar[r];mm]
       (delete nested PLUS operations:
         (PLUS A (PLUS B C)) = (PLUS A B C) );
     if eq[caar[r];MINUS] then
       collector1[cdar[r];minus[mm]]
         (the appearance of (MINUS e) is recorded as -1
         appearance of e rather than as 1 occurrence of
         (MINUS e); thus, (MINUS (MINUS e)) is recorded as
         1 occurrence of e;  this is the reason for the
         argument mm)1;
      if eq[caar[r];TIMES] then
        if numberp[cadr[car[r]]] then
          collect[caddr[car[r]];times[cadar[r];mm];tm]
            (if a numeric factor other than one is involved in a
            product, then simptimes places this factor at the
            front of the list of factors; thus, by the time
            simplus sees the product, if a numeric factor (not 1)
            appears, then it is necessarily the first factor;
            similar remarks may be made with respect to simplus
            and non-zero numeric terms);
       otherwise collect[car[r];mm;tm];
      otherwise collect[car[r];mm;tm].);

     collector1[cdr[r];mm;tm]

 ].
```

After it causes the term list to be generated, simplus must rewrite the term list as a conventional arithmetic expression. For this purpose, collect2 is called. The effect of collect2 is to generate an output list, lo, whose k'th member is car[simptimes[tm_k]], where tm_k is the k'th member of the term list (the argument of simptimes must be a list of factors; each element of tm is a list of two elements: an expression and the number of occurrences of that expression). Collect2 may be summarized by:

mapcon[tm;λ[[x];simptimes[car[x]]]].

The final actions performed by simplus are to cons n onto lo if n is non-zero; to cons PLUS onto the new lo if lo contains more than one element--if the list contains only one element, its car is taken as the new lo; and this last list is cons'ed onto NIL.

The remaining simp function to be discussed in this section, simptimes[s], is similar to simplus: na and tm are initially set to 1 and NIL, respectively; and collectorl is called to generate the term list. However, the tests made by collectorl when this function is called by simptimes differ from the tests described above in the discussion of simplus. Simplus and simptimes actually call collectorl via collector, whose six arguments specify which tests are to be made. The main changes are the following replacements in the descriptions given above: replace plus by times; PLUS by TIMES; MINUS by RECIP; and TIMES by EXPT. One additional test is made; and collectorl, when called by simptimes, has the following appearance:

```
prog2[

   (if null[r] then NIL;
    otherwise:
      if numberp[car[r]] then setq[n;times[n;mm;car[r]]];
      if atom[car[r]] then collect[car[r];mm;tm];
      if eq[caar[r];TIMES] then collector[cdar[r];mm];
      if eq[caar[r];RECIP] then collectorl[cdar[r];minus[mm]]
        (mm again keeps track of inverses--this time inverses
        under multiplication rather than inverses under addition);
      if eq[caar[r];EXPT] then
        collect[cadr[car[r]];car[simptimes[list[mm;
          caddr[car[r]] ]]];tm]
           (the reason for using simptimes, rather than times,
            will be explained below);
      if eq[caar[r];MINUS] then
        prog2[setq[n;minus[n]];collectorl[cdar[r];mm]]
          (this test is not made when collectorl is called by
          simplus; if a factor is preceded by a minus sign, then
          the minus sign is, in effect, transferred to the numeric
          portion of the product; thus, a simplified product contains
          no "negative" factors, except, perhaps, a negative numeric
          factor);
      otherwise collect[car[r];mm;tm].);

    collectorl[cdr[r];mm;tm]

  ].
```

The use of simptimes in the statement which is executed when car[s] is found to be an exponential form allows the collection of non-numeric numbers of occurrences of sub-expressions; e.g. suppose that s = ((EXPT a b)...). Then we may collect b occurrences of a, even if b is not a number. Thus simplify[(TIMES (EXPT A 3)(EXPT A((PLUS B 1)))] = ((EXPT A (PLUS 4 B))). Collect must be modified, but only slightly, in order to accommodate this more general mode of collecting: when collect finds a record of an expression in the term list, instead of adding m to cadr[car[s]] (s is that portion of the term list whose car contains the record of the expression currently being collected), collect will cons m onto cadr[car[s]].

After the term list has been produced, simptimes calls collect2. The normal function of collect2 when called by simptimes is analogous to its function when called by simplus; only one word is changed in the summary. We have: mapcon[tm;λ[[x];simpexpt[car[x]]]] (simpexpt replaces simptimes). However, to make possible collection of like factors with not necessarily numeric exponents, we must modify collect2 when it is called by simptimes:

mapcon[tm;λ[[x];simpexpt[cons[caar[x];simplus[cdr[x]]]]]].

Finally, if n is zero, simptimes returns (0); otherwise, if n is not equal to one, then n is cons'ed onto lo, the output of collect2; if this last list contains only one element, then simptimes returns that list; otherwise simptimes returns cons[cons[TIMES;lo];NIL].

This is the basic simplify program. It has the disadvantages that it permits the user no control over the form of the output of the program (unless, of course, the user is willing to understand LISP and the simplify program well enough to make it behave in a desired manner) and it does not make use of the polynomial format to simplify and speed up symbolic arithmetic operations (particularly division). In order to eliminate these shortcomings, the program has been expanded (by a factor of about three in the number of cards comprising the deck). The new functions of the program are described in the following pages.

Factor

The following question is suggested by the operation of simptimes: if we can collect non-numeric numbers of occurrences of sub-expressions in products (simplify[(TIMES X (EXPT X B))] = ((EXPT X (PLUS 1 B)))), then why don't we collect non-numeric numbers of occurrences of sub-expressions in sums? Then we would have, for example,

simplify[(PLUS (TIMES X A)(TIMES B X))] = ((TIMES X (PLUS B A))).

Brief reflection reveals that this latter trick requires some care because of the commutativity and associativity of multiplication; when we see (EXPT X A), there is doubt that we should collect A occurrences of X; but, when we see (TIMES A B), do we have A occurrences of B or B occurrences of A?

We have bypassed this difficulty by requiring the user of the program to specify what is to be done when collecting in products: before calling simplify, he gives evalquote the arguments FACTOR (r). Factor[r] is the function:

deflist[list[list[FACTOR;cond[[atom[r];NIL];[T;r]]];COLFLAG]].

Factor is defined so that the user need not be concerned with the details of using deflist and manipulating property lists.

If the argument, r, of factor is an atom, then the null list will be put onto the property list of FACTOR after the indicator COLFLAG. In this case simplify will not collect non-numeric numbers of occurrences of expressions in sums.

If r is a list, then the variables in the list will be factored from all sums in which factoring is possible; if factoring of more than one listed variable is possible, the variable which appears first in the factor list, r, will be factored first. For example:

```
    if r = (X Y Z), then simplify[(PLUS (TIMES A X)(TIMES B X))]
        = ((TIMES X (PLUS A B)));
    if r = NIL (or any other atom, or any list not containing X), then
          then simplify[(PLUS (TIMES A X)(TIMES B X))]
        = ((PLUS (TIMES A X)(TIMES B X)));
    if r = (X Y), then
        simplify[(PLUS (TIMES A X)(TIMES B X Y)(TIMES C Y))]
        = ((PLUS (TIMES X (PLUS (TIMES Y B) A))(TIMES Y C)));
    and, if r = (Y X), then
        simplify[(PLUS (TIMES A X)(TIMES B X Y)(TIMES C Y))]
        = ((PLUS (TIMES Y (PLUS (TIMES X B) C))(TIMES X A))).
```

The factoring is accomplished in two stages; we shall first consider the ordering of the factors in products (only products are affected by the ordering procedure). In the process of unwinding the term list generated by collectorl and collect, collect2 calls collect3[x;colist], where x is the arithmetic expression generated by collect2 in correspondence to the first element on the term list and colist is the factor list (as specified by the most recently executed "factor[r]" call). (Colist is a program variable available to collect2.)

Collect3 finds the first element of colist which appears in x, the remainder of colist is dropped; this new list is returned as the value of collect3, and it replaces the old value of colist. Collect3 is executed for its effect as well as its value: if x is found to have an element in common with colist, then x is set aside--stored as the value of the program variable vsav. The next time collect3 is called (x is now the arithmetic expression corresponding to the second element of the term list), if the first element of colist which appears in x turns out to be the last element of colist (which is the first element of the factor list which appeared in the expression corresponding to the first element of the term list), then x is cons'ed onto vsav, the value of collect3 is colist, and control returns to collect2, which eventually calls collect3 with the third arithmetic expression and so on. However, if the first element of colist which appears in x is not the last element of colist, then vsav is cons'ed onto the output list of collect2, lo, x is stored in vsav, and collect3 returns that portion of colist which precedes the element after the first element which appears in x, and control returns to collect2.

If x and colist have no elements in common, then no change is made in vsav, x is cons'ed onto lo, and collect3 returns colist.

Note that at any instant the value of colist is that sublist of the factor list whose last element is the first element of the factor list which appears in any of the expressions generated so far by collect2. Vsav contains a list of expressions generated so far by collect2 which contain the last element of colist. And, lo is a list of expressions which have been studied and which do not contain any element of colist.

When the term list is exhausted, collect2 appends vsav onto lo, producing a list with the property that all elements which contain the first member of the factor list which appears (at any level) in the list are at the beginning of the list.

An example of the operation of the ordering process may help to make the above description more meaningful. Suppose that we execute the two instructions:

```
FACTOR((V W X Y A))
SIMPLIFY((TIMES 3 W (PLUS W Z) X (PLUS 4 Y) X U Z)).
```

The term list which will be generated is:

((Z 1)(U 1)((PLUS 4 Y) 1)((PLUS W Z) 1)(X 2)(W 1)). We shall watch the values of the variables x, colist, vsav, and lo as collect2 unwinds the term list:

x	colist	vsav	lo
-	(V W X Y Z)	NIL	NIL
Z	"	(Z)	"
U	"	"	(U)
(PLUS 4 Y)	(V W X Y)	((PLUS 4 Y))	(Z U)
(PLUS W Z)	(V W)	((PLUS W Z))	((PLUS 4 Y) Z U)
(EXPT X 2)	"	"	((EXPT X 2) (PLUS 4 Y) Z U)
W	"	(W (PLUS W Z))	"

The term list is empty, so collect2 appends vsav to lo:
(W (PLUS W Z)(EXPT X 2)(PLUS 4 Y) Z U). The first element of the factor list which appeared in the term list was W, and all elements of the final list which contain W appear at the front of the list output by collect2. Finally, the numeric factor and the operation indicator are replaced, and the result listed:
simplify[(TIMES 3 W (PLUS W Z) X (PLUS 4 Y) X U Z)]
= ((TIMES 3 W (PLUS W Z)(EXPT X 2)(PLUS 4 Y) Z U)).

The second phase of the factoring process is accomplished by changing collector1, when called by simplus, for the case when eq[caar[r];TIMES] = T. The procedure is straightforward: simplusprime effectively causes the numeric factor of the product (simptimes always puts the numeric factor at the front of the list of factors) to be set aside temporarily; the list of non-numeric factors is examined by simplus2, which determines the first element, say e, of the factor list which appears in the first non-numeric factor of the product; simplus3 separates the list of factors into a list whose elements all contain e and a list whose elements do not contain e (simplus2 and simplus3 make use of the fact that the list of factors has been ordered by collect3--thus, only the first element need be studied in order to determine the first member of the factor list appearing in the list of factors; and a single pass through the list of factors is the maximum amount of effort required to split the list); these two lists are made into legitimate products by simptimes, and the numeric factor of the original product is included in the product not containing e; and the product containing e is collected onto the term list with the product not containing e used as the n number of occurrences.

Perhaps all this can be made more clear by making use of the previous example. We execute:

```
FACTOR((V W X Y Z))
SIMPLIFY((PLUS A (TIMES 3 W (PLUS W Z) X (PLUS 4 Y) X U Z))).
```

After simplification is accomplished on the second level of the expression, simplus is called:

simplus[(A (TIMES 3 W (PLUS W Z)(EXPT X 2)(PLUS 4 Y) Z U))].

Collectorl will see A, set the term list to ((A 1)), and on to the second term. The first element of this term, TIMES, sends control of the program to t the group of functions described above: simplusprime sets 3 off to the side; simplus2 looks at the first element, W, of the remaining list and determines that the first element of the factor list which appears in W is W; simplus3 strips off the elements containing W from the list of factors, forming two lists:

(W (.   W Z)) and ((EXPT X 2) (PLUS 4 Y) Z U); simplusl makes products of these two lists (replacing the numeric factor, 3, in the second list) and, finally, calls:

collect[(TIMES W (PLUS W Z));(TIMES 3 (EXPT X 2)(PLUS 4 Y) Z U); ((A 1))].

It will be noted that the scheme described above will not factor products; for example: simplify[(PLUS X (EXPT X 2))] = ((PLUS X (EXPT X 2))), whether or not X is on the factor list; simplify will not give us the result ((TIMES X (PLUS 1 X))). At first, this seems a rather serious limitation; however, it has one advantage (in addition to the programming difficulties it avoids): the factored output is in a form which may be converted to polynomial notation with considerable ease. Of course, once we have rewritten (PLUS X (EXPT X 2)) in the form (POLY X 1 1 0), it is not difficult to drop the tailing zero and multiply the result by X: (TIMES X (POLY X 1 1)). This, then, may be changed into (TIMES X (PLUS X 1). This scheme appears to be a very roundabout way of factoring, but its implementation is straightforward and makes use of much of the code which already exists; factoring directly would be difficult to code and would require considerably more memory for the program than factoring by means of polynomial manipulation.

One further feature of this factoring mechanism deserves mention: consider a case in which a sum contains several products involving the first element of the factor list, say: (PLUS (TIMES A X Y)((TIMES B X Y)(TIMES C X)), and the factor list is (X Y). Then the term list which will be generated for simplus will be: ((X (C (TIMES Y B)(TIMES Y A)))). Collect2 will use simplus to form a sum of the number of occurrences of X, and simplus will factor Y from the sum:

simplus[(C (TIMES Y B)(TIMES Y A))] = ((PLUS C (TIMES Y (PLUS B A)))). Thus, the factor feature accomplishes factoring whenever possible-- on any level of the expression; and decisions as to which variable should be factored when more than one might be factored are made by referring to the factor list--the first variable on the factor list is factored first, the second variable second, and so on. (Note, in the above example, that if the factor list were (Y X), then:

simplify[(PLUS (TIMES A X Y)(TIMES B X Y)(TIMES C X))]
= ((PLUS (TIMES Y X (PLUS A B))(TIMES X C))).)

Polynomials

A set of functions which manipulate polynomial expressions has been included in simplify; this was done in order to take advantage of the speed and ease of dealing with polynomials. In particular, division of polynomials may be programmed in a straightforward manner.

The definition of simplus, simptimes, and simpquotient have been modified from the forms described earlier by the inclusion of calls to the polynomial functions whenever appropriate.

We shall consider the polynomial functions:

Simpoly[s] is called only by simp; if s is the zero polynomial, (POLY x 0 . . . 0), then simpoly returns (0); otherwise simpoly returns the polynomial derived from s by dropping all leading zeroes: simpoly[(POLY x 0 0 1 1)] = (POLY x 1 1).

Polyunwrite(s) rewrites the polynomial s as a non-polynomial expression; the procedure may be summarized by the program:

```
prog[(u v w);
  setq[u;0];
  setq[v;NIL];
  setq[w;reverse[cddr[s]]];
A setq[v;cons[car[simptimes[list[car[w];car[simpexpt[
      list[cadr[s];u]]] ]]];v]];
  setq[u;add1[u]];
  setq[w;cdr[w]];
  cond[[w;go[A]]];
  return[car[simplus[v]]]
].
```

The actual polyunwrite program is written in the form of recursive and pseudo-recursive functions, rather than in the form of a program; the program rewrites polynomials as non-polynomial expressions on every level of the original expression. After the coefficients of a polynomial have been "polyunwritten," the polynomial itself must be transformed, and this process takes place in the manner described in the program; beginning with the low-order coefficient (car[reverse[cddr[s]]]) and working up to the high-order coefficient (caddr[s]), the program builds up a sum by raising the polynomial variable (cadr[s]) to the same power (u) as the order of the coefficient (car[w]) currently under consideration, then multiplying this expression by the current coefficient, and cons'ing that product to the list (v) of terms. If s is not a polynomial, then polyunwrite will "polyunwrite" any polynomials which appear in s.

If r is a list of atoms, then polywrite[s;r] searches r until it finds an element, x, which appears in s; then polywrite calls itself: polywrite[s;x]. If s and r have no element in common, then polywrite returns s if s is not a polynomial, and returns polywrite[s;cadr[s]] if s is a polynomial.

When x is atomic, polywrite[s;x] writes the expression s as a polynomial in x. If s does not contain x, then polywrite returns (POLY x s). If s is a polynomial in x and none of the coefficients of s contains x, then polywrite[s;x] = s. If neither of these cases is satisfied, then the factor list is set aside, and a new factor list produced which contains the single element x.

Then polywrite calls simplify[polyunwrite[s]]. The output of simplify will almost be a polynomial: no two terms will involve the same power of x, except constant terms; and in every term involving x, the factor containing x will be the first factor. Then, polywrite must rearrange the terms, combine all the constant terms, and drop the factors containing x.

Polywrite calls polywrite3 (via polywrite 1 and polywrite2). Term by term, polywrite3 scans the output of simplify, determines the power of x involved in that term, and gives back a list, (k a), of the power of x and the coefficient. Polywrite2 calls setarray, which simplus'es a onto the k'th element of the coefficient list, harray. When the output of simplify has been scanned, polywritel puts x and POLY on the front of harray, and returns the polynomial to polywrite.

The polynomial format used by the program has no provision for negative powers of x. Therefore, if polywrite3 sees a negative power of x, this term is set aside until the remainder of the simplify output is scanned. Then, polywritel makes a polynomial of the coefficient list, harray, cons'es this polynomial to the list of terms which were set aside by polywrite3, and then cons'es PLUS to that list. Thus, polywrite[(PLUS A (TIMES 2 (EXPT X 2)) B (RECIP X)); X] = (PLUS (POLY X 2 0 (PLUS A B))(RECIP X)).

If setarray is asked to modify the k'th element of harray, but harray's last element is numbered less than k, then setarray will lengthen the list to accommodate the k'th element. Similarly, on the lower end, setarray will extend the list in order to accommodate low-numbered elements. As it is used now, setarray is made to believe that there is a zero'th element, and it is never asked to store data in negative elements (polywrite3 sees to this). However, these restrictions are by no means necessary; setarray was written with the thought in mind that it might prove useful to use a polynomial format such as:

(POLY X -3 A 0 B 0 0 1) = (PLUS (TIMES A (EXPT X 2)) B (EXPT X -3)) (here, the third element of the polynomial specifies the order of the low-order coefficient). Such a format has the advantage that, for example, (PLUS (EXPT X 1000)(EXPT X 998)) may be written as a list of six elements, (POLY X 998 1 0 1), rather than a list of 1003 elements, of which only four are non-zero: (POLY X 1 0 1 0 . . . 0). The polynomial convention which was adopted for simplify was simpler to implement and permits greater speed of manipulation.

Polyplus, polytimes, and polyquotient return 0 if a zero polynomial results from the addition, multiplication, or division, respectively; returns the constant term if the result is a polynomial of zero degree; and returns the polynomial result otherwise. Each of these functions has two arguments--the expressions to be manipulated, and the output is an expression--not a list whose first element is an arithmetic expression, as in the case of the simp functions. The forms of the input and output for the polysimp functions seemed worthwhile simplifications when they were decided upon; however, they render impossible the use of descriptor lists in the polynomial package (see the section on descriptor lists). (This mistake will not be repeated in future versions of simplify).

If neither y nor z is a polynomial, then polyplus[y;z] = car[simplus[list[y;z]]]. Otherwise, if z is a polynomial, then y is rewritten as a polynomial in terms of the variable of z; if z is not a polynomial, then z is rewritten in terms of cadr[y]. Now polyplus has two polynomials in the same variable; the coefficient lists are reversed and given to polyplusl, which adds corresponding coefficients, from low-order to high-order, until one list is exhausted; then whatever remains of the other list is (effectively) reversed and appended to the sum coefficient list. The output of polyplus1 is the list of coefficients (in descending order) of the sum polynomial, to which polyplus need only append the list (POLY/x), where x is the variable of the polynomials.

If neither argument of polytimes[y;z] is a polynomial, then the product of y by z is expanded by polyexpand1: both y and z are expressed as lists of terms of a sum (if y is not a sum--i.e., eq(car(y);PLUS) = NIL--then the list of terms contains one element: y; similarly for z); then each element of each list of terms is multiplied (by simptimes) by each element of the other list, and the resulting products are summed by simplus. Thus,

polytimes[(PLUS A B);(DIFFERENCE A B)] = (PLUS (EXPT A 2)(MINUS (EXPT B 2))).

When at least one argument of polytimes[y;z] is a polynomial: if both y and z are polynomials, then y is rewritten in terms of cadr[z]; otherwise, the non-polynomial argument is made into a polynomial in the variable of the other argument. The coefficient lists of the two polynomials are then "multiplied" together by a succession of maplists and "shifting" of the partial product coefficient list. The coefficient list of z is multiplied (using simptimes), term by term, by the low-order coefficient of y, and the resulting list is the initial value of the partial product. Zero is cons'ed onto the partial product; the coefficient list of z is multiplied by the second low-order coefficient of y, and the resulting list is added, term by term, to the partial product list, forming a new partial product, to which is cons'ed zero, and the multiplication is repeated, using the third low-order coefficient of y; and so on, until the product of the coefficient list of z by the high-order coefficient of y has been included in the partial product. The variable of the polynomials and POLY are then successively cons'ed onto the partial product list, and polytimes returns the resulting polynomial.

Polyquotient[y;z] writes both its arguments as polynomials in the first member of the factor list which appears in either y or z. If no member of the factor list appears in y or z, or if the factor list is null, then: if z is a polynomial, then y is rewritten as a polynomial in the variable of z; if y is a polynomial, then z is rewritten in terms of the variable of y; otherwise, polyquotient returns a list of the form (TIMES y car[simprecip[list[z]]]) (if car[y] = TIMES, then polyquotient returns nconc[y;simprecip[list[z]]]).

The rewriting of y and z (when possible) is performed under the direction of polyquotient1, which gives the rewritten arguments to polyquotient2. Polyquotient2 initializes the program variable check to NIL; if y is a lower-degree polynomial than z, then check will be set to T by polyquotient6, and (TIMES y (RECIP z)) will be returned by polyquotient. (fl is also set to NIL by polyquotient2--this is necessary to prevent errors from occurring when various simp functions, which use the program variable fl, are called by the polyquotient functions--fl is described in the section on descriptor lists.)

Polyquotient2 gives the coefficient lists of y and z to polyquotient4, which initializes the program variables signal and ldcof to T and NIL, respectively, and calls polyquotient5. Signal will be set to NIL by polyquotient6 when the division process is in its next-to-last iteration--the last iteration is performed differently from the others; after the last iteration, the value of the signal is set to the list of coefficients of the quotient polynomial.

During each iteration, ldcof stores the quotient of the first coefficient of the reduced numerator currently considered by the first coefficient of z; ldcof will be multiplied by each of the other coefficients of the reduced numerator, and the resulting list will be subtracted, term by term, from the reduced numerator. Consider the quotient:

$(3a^2 + 6ab + abc + 2b^2c)/(a + 2b)$; or, in polynomial notation:

poly(a;3,6b+bc,$2b^2c$)/poly(a;1,2b). We shall follow the progress of the division:

ldcof	denominator coefficients	numerator coefficients
	1,2b	3,6b+bc,$2b^2c$
3/1 = 3		- 3,6b
		0, bc,$2b^2c$
bc/1 = bc		- bc,$2b^2c$
		0, 0

Thus, ldcof contains the coefficient of the quotient list which was generated by the most recent iteration; the quotient coefficient list is obtained by listing all the values taken on by ldcof during the division: in the example, this list is (3 bc), and in the quotient polynomial is poly(a;3,bc), or 3a + bc. After the last iteration, the value of ldcof is set to the remainder coefficient list; in the example above, this list is (0 0).

The reduction of the numerator is accomplished by polyquotient6, which carries out the process of "long division", as is indicated in the example above.

The output of polyquotient4 is a listed pair; the first member is the list of coefficients of the quotient polynomial; the second is the list obtained from the remainder polynomial coefficient list by dropping all leading zeroes. In the example, the output of polyquotient4 would be: ((3 (TIMES bc)) NIL). This list is given to polyquotient3.

Polyquotient3 returns (TIMES y (RECIP z)) if check has been set to T. If the remainder list is null, then polyquotient3 makes a polynomial of the quotient list and returns. When the division yields a non-zero remainder, polyquotient3 will return an expression of the form (PLUS q (QUOTIENT r z)) (simplifications are made where possible), where q, r, and z are the quotient, remainder, and divisor, respectively, if a non-null list is stored on the property list of the atom SIMPLIFY after the indicator REMAINDER. If no such list or indicator exists, then polyquotient3 returns an expression of the form: (TIMES y car[simprecip[list[z]]]).

The function polyremainder is provided to enable the user to store the an expression on the property list of SIMPLIFY after the indicator REMAINDER. Polyremainder[NO] has the same effect as polyremainder[NIL]--polyquotient will return a quotient only if the remainder is zero.

Psav

We now have the facility to deal with either polynomial or non-polynomial expressions. There seems to be no convenient way of dealing with the two types of expressions simultaneously; thus, if a polynomial and a non-polynomial both appear in a sum, we might either polyunwrite the polynomial or polywrite the non-polynomial, and then simplify. However, another scheme has been adopted--one which minimizes the amount of polywriting and polyunwriting--we simply handle the two types of expressions separately as long as possible, and, in the final stages of simplus and simptimes we shall give to polyplus and polytimes, respectively, the job of making a simplified polynomial of the sum or product.

The program variable psav is set to 0 (1) by simplus (simptimes). Collect2, then, tests car[simplus[car[tm]]] (or car[simptimes[car[tm]]]) to see if it is a polynomial--if so, the expression is added (multiplied) by polyplus (polytimes)into psav, rather than being cons'ed onto the output list. If this test fails, then collect2 acts in the manner described previously (other modifications to collect2 will be discussed later, but we assume for now that they d do not exist). When the term list is exhausted, psav contains the sum (product) of all polynomial expressions which were represented in the term list, and the output list contains all the non-polynomial expressions derived from the term list.

When simplus (simptimes) prepares the final simplified expression it checks the value of psav. If this value is still 0 (1), then the numeric portion and the PLUS (TIMES) are put into the output list, according to the rules previously described. However, if psav is not still 0 (1), then polyplus (polytimes) is called to add (multiply) psav into the non-polynomial part of the output expression, and the numeric part and PLUS (TIMES) are listed with the resulting polynomial expression if necessary.

Thus, any sum or product involving a polynomial term or factor will be simplified to a polynomial. It is, however, a simple matter to cause such sums or products to be simplified to expressions involving no polynomials--we need only call simplify[polyunwrite[e]], rather than simplify[e].

Recipmode

The fact that the reciprocal of an expression, e, may be represented as (RECIP e) or as (EXPT e -1) leads us to add another feature to the simplify program: recipmode. This function sets the constant RECIPMODE to T or NIL, according as the single argument of recipmode is R or some other S-expression, respectively.

When the value of RECIPMODE is T, then simpexpt changes all exponential expressions with negative exponents (a symbolic expression whose first element is MINUS is considered by simpexpt to be negative) into reciprocals of expressions with non-negative exponents. For example, when the value of RECIPMODE is T, simpexpt[(A -3)] = ((RECIP (EXPT A 3))); and simpexpt[(A (MINUS B))] = ((RECIP (EXPT A B))).

If recipmode[x], x $\neq$ R, is executed, then subsequent calls of simpexpt will cause any expression whose first element is RECIP and which is to be raised to a power to be relieved of the prefix RECIP and to be raised to the negative of the original exponent: simpexpt[(A -3)] = ((EXPT A -3)); and simpexpt [((RECIP A) -3)] = ((RECIP A 3)).

Expand

It is sometimes convenient to be able to express a product of sums as a sum of products; e.g. (a + b)(c + d) = ac + ad +bc + bd. Collect2 has been modified to accomplish such expansion, when desired.

The "expand list," which is stored on the property list of SIMPLIFY after the indicator EXPAND, is a list of atomic symbols. The occurrence of at least one of the elements of the expand list in a product of sums initiates the expansion process. The expand list may be set by the user by means of the function expand[x], where x is the desired expand list.

Since the expansion of products of sums is applicable only when collect2 is called by simptimes, we consider the unwinding of the term list generated by simptimes (actually, of course, the term list is generated by collector when called by simptimes). Collect2 compares the algebraic expression produced by the call simpexpt[car[tm]] with the expand list. If no element of the expand list occurs in the expression ,then the expression is cons'ed onto the output list in the usual manner. However, if the expression does contain one or more elements of the expand list, then the expression is multiplied (by polytimes) by the current value of psav, and the resulting product replaces the old value of psav. The machinery which handles polynomial factors also handles the task of putting the expanded products into the output list.

The result of making this very simple addition to collect2 is demonstrated by the following example: if we execute expand[(X)], then simplify[(PLUS X (TIMES (PLUS A B)(PLUS C X)))] = ((PLUS X (TIMES A C)(TIMES A X) (TIMES B C)(TIMES B X)))= and simplify[(PLUS X (TIMES (PLUS A B)(PLUS C D)))] = = ((PLUS X (TIMES (PLUS A B)(PLUS C D)))) (in this second case, the single element, X, of the expand list does not appear in the product of sums).

A problem may sometimes arise in the event that both the expand and factor lists contain an element which may be factored from a sum of products; an example best describes the difficulty: suppose that expand[(Y)] and factor [(Y)] have been executed, and we want to compute simplify[(PLUS (TIMES Y A)(TIMES Y B))]. After simplification on the two lowest levels (Y, A, Y, B, (TIMES Y A), and (TIMES Y B)), simplus[((TIMES Y A)(TIMES Y B))] is called. Simplus generates the term list representing (PLUS B A) occurrences of Y, then collect2, in the process of generating the output list, calls simptimes[(Y (PLUS B A))]. But, because Y appears in both the argument of simptimes and in the expand list, simptimes calls polytimes[Y;(PLUS B A)], which expands its arguments into a list of products, which is made into a sum by simplus. Thus, polytimes[Y;(PLUS B A)] = simplus[((TIMES Y B)(TIMES Y A))]; the evaluation of this expression requires evaluation of simplus[((TIMES Y A)(TIMES Y B))], which requires the value of simplus[((TIMES Y B)(TIMES Y A))], etc.

This non-terminating recursion problem has been avoided by simply turning off the expand feature before collect2 calls simptimes (when collect2 is called by simplus): the expand list is saved, the null list put in its place on the property list of SIMPLIFY, and, after simptimes is executed, the expand list is replaced.

The expand list also affects the output of simpexpt; if a product containing an element of the expand list is raised to a power, then simpexpt will return the product of powers of factors: expand[(X)] and simplify[(EXPT (TIMES A X) 3)] yield ((TIMES (EXPT A 3)(EXPT X 3))). Furthermore, if the exponent is a fixed-point number greater than zero, say n, and the expression to be raised to the n'th power is a sum, then simptimes will be called n-1 times by simpexpt in order to generate a sum of products: expand[(X)] and simplify[(EXPT (TIMES (PLUS A X) B) 2)] yield ((TIMES (PLUS (EXPT A 2)(TIMES 2 A X)(EXPT X 2))(EXPT B 2))).

Consideration of expand leads us to another feature of collect2: the handling of sums of reciprocals which involve members of the expand list. If recipmode[R] has been executed, then when collect2 is called by simplus, the program variable rsav (initialized by simplus) is used to store the product of all expressions seen by collect2 which involve at least one member of the expand list and which are preceded by RECIP. For example, if the expand list is (B C), then simplus[(A (RECIP B) D (RECIP C)(RECIP B))] will generate the term list: (((RECIP C) 1)(D 1)((RECIP B) 2)(A 1)). We shall follow the development of rsav and tm (the term list) as the output list is generated:

rsav	tm	lo
NIL	(((RECIP C) 1) (D 1)((RECIP B) 2)(A 1))	NIL
(C)	((D 1)((RECIP B) 2)(A 1))	(1)
(C)	(((RECIP B) 2)(A 1))	((TIMES D C) 1)
(B C)	((A 1))	((TIMES 2 C)(TIMES B D C) B)
(B C)	NIL	((TIMES A B C)(TIMES 2 C)(TIMES B D C) B)

Each time a reciprocal expression is found in the term list each element of the output list is multiplied by the expression (after deleting RECIP from the expression by taking its cadr), the product of rsav by the number of occurrences of the reciprocal expression is cons'ed onto the output list, and the expression which appeared reciprocated is cons'ed onto rsav. Furthermore, every term to be included in the output list, but which does not result from a reciprocal expression in the term list, is multiplied by the current value of rsav, and,this product is cons'd onto lo. Thus, when the term list is exhausted, the product of the terms of lo is the numerator of the final result, and the product of the terms of rsav is the denominator of the final answer. In the process of putting the finishing touches on the simplified expression, simplus replaces rsav by car[simptimes[rsav]]; puts the numeric term and PLUS onto the output list generated by collect2, if necessary; then gives these two lists to polyquotient, whose output is returned by simplus. We have, then:

simplus[(A (RECIP B) D (RECIP C)(RECIP B))] = ((TIMES (PLUS (TIMES A B C)
(TIMES 2 C)(TIMES B D C) B) (RECIP (TIMES B C)))).

Another function of collect2, one which was not demonstrated by the example, is to multiply psav by the reciprocated expression (after, of course, deleting RECIP) when the elements of lo are multiplied by the reciprocated expression.

When collect2 is called by simptimes, and if RECIPMODE = R, then rsav is also used, to store a list of all reciprocated expressions found on the term list. In this case, however, the elements of lo are not modified when a reciprocal expression appears. The final phase of simptimes generates the quotient of the product of the terms of lo and the product of the terms of rsav: simptimes[(A (RECIP B) D (RECIP C)(RECIP B))] = ((TIMES A D (RECIP (TIMES C (EXPT B 2))))). The handling of reciprocal expressions when simptimes calls collect2 is independent of the expand list.

If RECIPMODE ≠ R, then simpexpt will change reciprocals into exponential expressions. Collect2 will not put such expressions on rsav, and the final result will not have the form of an expression multiplied by the reciprocal of another expression.

Descriptor Lists

Our discussion of what the simplify program does is now complete (the operator package, which will be considered later, is not considered to be part of the simplify program, but is an auxiliary program). However, a study of the listing of the program reveals that some things are done in strange ways--namely, simplify (and the simpfunctions) returns a list whose only element is the simplified expression; these lists are generated by the function trigsimp, rather than by a straightforward cons or list; and the function descmap (in simp) operates on the output of the simpfunctions.

These features of the program have been incorporated in order to make possible the addition of an identity-recognizing routine (in this connection, the trigonometric identities immediately come to mind--hence, the name trigsimp).

It is intended that, after simplification of a sub-expression, trigsimp will be given an opportunity to study the output expression by comparing its properties with a list of properties stored on the property list of the main connective of the simplified expression. Paired with each set of properties on this list will be a message, or descriptor, which indicates conditions to be looked for on higher levels of simplification of the original expression. If the expression seen by trigsimp has any of the properties on the appropriate list of property-descriptor pairs, then the expression is list'ed with the corresponding descriptor, and this list is returned.

For example, suppose we wish to substitute 1 for (PLUS (EXPT (SIN x) 2)(EXPT (COS x) 2)) whenever possible, where x is an arbitrary expression. Then the following information (in some convenient representation) will be stored on the property list of EXPT:

> If the expression is of the form (EXPT (SIN x) 2) (or (EXPT (COS x) 2)), then look for PLUS as the main connective on the next-higher level and for another term of the form (EXPT (COS x) 2) (or (EXPT (SIN x) 2)) in the sum; then, if these conditions are also met, subtract one occurrence of (EXPT (SIN x) 2) and of (EXPT (COS x) 2) from that sum and add 1.

(There may, of course, be other property-descriptor pairs in this list). The portion of this statement following the first "then" is the descriptor; for brevity, let this descriptor be denoted by D. Now consider the simplification of (PLUS (EXPT (COS A) 2)(EXPT (SIN A) 2)). Simpexpt is called twice; in both cases, the output expressions from simpexpt satisfy properties which appear on the list of property-descriptor pairs stored on the property list of EXPT. Thus, the outputs of simpexpt are ((EXPT (COS A) 2) d) and ((EXPT (SIN A) 2) D). On the next-higher level of simplification, simplify calls simp, which saves the descriptors (this is why the second argument of simp must be a list of lists of expressions, contrary to the simplified description given previously) from each of the elements of its second argument; simp lists the car's of the elements of its second argument, sends this list to the appropriate simpfunction, saves the new descriptors generated by the simpfunction, and then calls descmap ("descriptor map" function), which sees if any of the conditions of the original descriptors are satisfied by the expression output by the simpfunction--if so, then the instructions given in the descriptor are obeyed.

In our example, simp saves the descriptors D and D; then calls simplus, which returns ((PLUS (EXPT (COS A) 2) (EXPT (SIN A) 2))); no new descriptors are generated by simplus, and descmap is called. Descmap finds that both (SIN A) and (COS A) raised to the second powers appear in the sum. Therefore, descmap obeys the instructions indicated in the descriptor list: subtract (EXPT (SIN A) 2) and (EXPT (COS A) 2) and add 1. The result is 1, which is cons'ed onto the "new descriptor list" which was saved by simp before descmap was called--in this case, the list is NIL--and the output of simp is (1).

The instructions followed by descmap may require that no substitutionsbe made, but that a new descriptorbe cons'ed onto the "new descriptor list." Thus, it is possible to recognize identities involving arbitrarily complex expressions.

The polynomial functions were written under the belief that the descriptor list feature would not be necessary when dealing with polynomials, and that writing and understanding the code would be simpler without including the descriptor lists in the outputs of the polynomial functions. Belatedly, it was recognized that this was a serious omission--that the usefulness of the identity-recognizing scheme is seriously impaired by not being able to recognize special forms when they appear in polynomial expressions. The changes which are required in the polynomial package are, in essence, simple and straightforward. However, since many changes are required, this task has been postponed in favor of debugging the remainder of the simplify system and preparing this report.

A set of programs exists (complete with built-in bugs) which translates identities into descriptor-instruction pairs, stores these pairs on the property lists of the appropriate atoms, recognizes identities, and makes the indicated substitutions.

The first of these functions, equiv(s), is given a list of listed triplets, ((x x_f e r). . .). The first element, x_f, of a triplet specifies which of the atoms appearing in the second element, e, is to be regarded as "free"--i.e. representing an arbitrary expression. Because of the difficulty of recognizing identities increases much faster than the number of free variables in the general form, we restrict ourselves to such substitutions as "f(x) is to be replaced by g(x)," but "f(x, y,...,z) is to be replaced by g(x,y,...,z)" is outside the scope of our program. Thus, each triplet involves only one free variable. The third element, r, of the triplet is an expression which is to replace e whenever e occurs. More precisely, e and r are forms; if, for any S-expression, x, e(x) occurs in an expression being simplified, then e(x) is to be replaced by r(x).

Equiv processes its list of triplets in the following manner:

Step 1: The lowest level of the expression e is found, and the descriptor-instruction pair which will subsequently be generated will be stored on the property list of the main connective of the first sub-expression of e which occurs on the lowest level. This main connective need not be one of the arithmetic operation atoms.

Step 2: All variables (including the free variable, x_f, if it occurs) which appear in the lowest-level sub-expression found in step 1 are listed, and this list becomes the first element of the descriptor-instruction pair. The free variable is specially-marked so that it will be recognized as free when the descriptor list is used later by trigsimp. This list of variables is listed.

Step 3: In working its way out of the recursion which took it to the lowest-level sub-expression, equiv proceeds as follows on each level except the last:

a) The main connective of the next-higher level sub-expression, x_f, and the cdr of the next-higher level sub-expression are listed.

b) This list becomes the last element of the lowest level sublist of the partial descriptor-instruction list sent up from the previous level of recursion of equiv--if that partial descriptor-instruction list contained more than just the list of variables generated in step 1. Otherwise the variable list is listed with the list generated in part a of this step.

Step 4: The final step in the generation of the descriptor-instruction pair is to form the list (RPLAC r) and to place this list in the partial descriptor-instruction list according to part b of step 3.

Consider the examples:

1) We wish to replace (EXPT I 2) by -1. I is a particular atom, not a free variable. In this case, the first element of the triplet may be any atom, say x. Equiv[((X (EXPT I 2) -1))] generates the descriptor-instruction list, which we denote by di:

Step 1: The lowest-level sub-expression is the expression itself: (EXPT I 2).
Step 2: di = ((I 2)).
Step 3: Since the lowest-level sub-expression is also the highest-level sub-expression, step 3 is omitted.
Step 4: di = ((I 2)(RPLAC -1)). This list goes onto the property list of EXPT.

2) We wish to replace (RECIP (TAN x)) by (QUOTIENT (COS x)(SIN x)). This time x is free. Equiv[((X (RECIP (TAN X))(QUOTIENT (COS X)(SIN X))))] generates di as follows:

Step 1: The lowest- level sub-expression is (TAN X).
Step 2: di = ((x_f)).
Step 3: di = ((x_f)(RECIP x_f ((TAN x_f)))).
Step 4: di = ((x_f)(RECIP x_f ((TAN x_f))(RPLAC (QUOTIENT (COS x_f) (SIN x_f))))), and this list goes onto the property list of TAN.

The function trigsimp and its satellites perform the comparison of expressions with the descriptors and make substitutions when called for. All the work is done by trigsimp, and descmap is no longer needed; however the equiv-trigsimp package represents only one possible use of the descriptor list feature--the fact that these functions exist in punched card form is not to be construed as a final commitment to this particular scheme--therefore, descmap has not been deleted from the simplify system.

When a simpfunction wants to prepare an output expression whose car is the name of the arithmetic operation corresponding to that simpfunction (when simplus wants to return ((PLUS x y . . . z)), when simpminus wants to return ((MINUS x)), etc.), then trigsimp[op;e] computes the value of the simpfunction. Op is the name of the arithmetic operation corresponding to the simpfunction, and e is a list of arguments of that operation. As the simplify system now stands, trigsimp[op;e] = list[cons[op;e]]. The identity-recognizing trigsimp, however, proceeds in the following manner:

Step 1: The descriptor lists from the previous level of simplification are stored as fs.
Step 2: Fs is searched for an entry whose car is op. If such an entry is found, then the sublist of fs which follows that entry replaces fs, and control goes to step 3. Otherwise, fs is set to NIL and control goes to Step 4.
Step 3: The selected entry from fs is one of the instruction lists generated by equiv, but with the second element, x_f, replaced by an expression: (op ae as is). x_f is replaced by ae whenever x_f occurs in as, and this new list is compared, by compare, which will be described below, with e. If the comparison yields a positive result, then the program goes to step 5. Otherwise, control returns to step 2.
Step 4: The descriptor-instruction pairs list from the property list of op is searched for an entry whose car (the car of each descriptor-instruction pair is the variable list constructed in step 2 of equiv) compares positively with e. If such a descriptor-instruction pair is found, then is is set to the cadr of that pair (the instruction list of the pair), ae is set to the value of compare, and the program goes to step 5. Otherwise, trigsimp returns list[cons[op;e]].
Step 5: If is is of the form (RPLAC r), then trigsimp substitutes ae for x_f whenever x_f occurs in r, simplifies the result, and returns the output of simplify. Otherwise, is has the form (op' x_f as' is'). In this case, trigsimp returns cons[cons[op;e];list[op';ae;as';is']].

In discussing compare[op;s1;s2], we consider two cases (var is a program variable):

1) The flag COMMUTE appears on the property list of op. In this case, we have:

Step 1: Set var to NIL.

```
Step 2:  If null[s2], then return null[s1];
     Otherwise, if null[s1], then:
            If null[var], then return NIL;
            Otherwise, return
               car[simp]op;maplist[s2;λ[[x];list[car[x]]]];
                                                 NIL]].
Step 3:  If car[s1] = x_f, then set var to T and s1 to cdr[s1];
              then go to step 2.
Step 4:  If an expression appears in s2 which is arithmetically
     equivalent to car[s1], then delete that expression from s2,
     set s2 equal to the new list thus obtained;  set s1 to
     cdr[s1], and go to step 2.  If no such expression appears
     in s2, then return NIL.
```

2) The flag COMMUTE does not appear on the property list of op. Then:

```
Step 1:  Set var to NIL.
Step 2:  If null[s1], then:
           If null[s2], then return T;
           Otherwise, return NIL.
         If null[s2], then return NIL.
Step 3:  If car[s1] = x_f, then:
           If var = NIL, then set var to car[s2] and
              to to step 4.
           Otherwise:
              If car[s2] and var are arithmetically equivalent,
                   then go to step 4.
              Otherwise, return NIL.
         If car[s1] ≠ x_f, then go to step 5.
Step 4:  Set s1 to cdr[s1]; set s2 to cdr[s2].
Step 5:  If car[s1] and car[s2] are arithmetically equivalent,
     then set s1 to cdr[s1], set s2 to cdr[s2], and go to
     step 2.  Otherwise, return NIL.
```

s1 and s2 are "arithmetically equivalent" if and only if simp[DIFFERENCE;list[list[s1];list[s2]];NIL] = (0).

Thus, if op represents a commutative operation, then compare checks for set equality (that is, the order of the elements is of no importance) between s1 and s2; and, if x_f appears in s1, then compare returns the expression that must be substituted for x_f in order to obtain set equality, if such an expression exists. Furthermore, if op represents a non-commutative operation, then compare checks for equality of n-tuples; and, if x_f appears in s1, then compare returns the expression (if it exists) which must replace x_f in order to obtain n-tuple equality.

The descriptions of trigsimp and compare seem to leave quite ample opportunity for confusion. Therefore, we offer the demonstration of their operations:

We are dealing with a function, f, and wish to recognize that f(x,x+A) = x. We inform the simplify system of this identity by calling equiv[((X (F X(PLUS X A)) X))], which places on the property list of PLUS the descriptor-instruction pair list: $(((x_f\ A)(F\ x_f\ (PLUS\ x_f\ A))(RPLAC\ x_f))))$. We shall watch the progress of trigsimp as the expression (PLUS (F B (PLUS X A))(F(PLUS B C)(PLUS A B C))) is simplified.

On the third level of simplification, while dealing with (F B (PLUS X A)), simplus calls trigsimp[PLUS;(X A)]:

Step 1: fs is set to NIL, since the next-lower level of simplification produced no descriptor list.
Step 2: fs = NIL, so go to step 4.
Step 3: The car of the only element of the descriptor-instruction pairs list on the property list of PLUS is found by compare to be equal (in the set sense) to (X A) if x_f is replaced by X; thus, is is set to (F x_f (x_f (PLUS x_f A))(RPLAC x_f)) and ae is set to X.
Step 5: is is not of the form (RPLAC r), so trigsimp returns ((PLUS X A) F X (x_f (PLUS x_f A))(RPLAC x_f)), and this is the value of simplus.

On the next level of simplification, simp saves the descriptor list that trigsimp has just generated; the section of simp which handles expressions to which none of the simpfunctions is applicable calls trigsimp[F;(B (PLUS X A))]:

Step 1: fs is set to ((F X (x_f (PLUS x_f A))(RPLAC x_f))).
Step 2: The car of the element of fs is F, so set fs to cdr[fs] -- in this case, NIL.
Step 3: Replace x_f by X in (x_f (PLUS x_f A)), yielding (X (PLUS X A)). Compare[F;(X (PLUS X A));(B (PLUS X A))] = NIL, so return to step 2.
Step 2: fs is NIL, so go to step 4.
Step 4: There is no descriptor-instruction pairs list on the property list of F, so trigsimp returns ((F B (PLUS X A))), which is the value of simp.

Simplify goes on now to the second term in the sum: (F (PLUS B C) (PLUS A B C)). On the second-lowest level of simplification, simplus calls trigsimp[PLUS;(A B C)]:

Step 1: No descriptor lists were generated on the atomic level of simplification, so set fs to NIL.
Step 2: fs = NIL, so go to step 4.
Step 4: The car of the only element of the descriptor-instruction pairs list for PLUS is (x_f A), so is is set to (F (x_f (PLUS x_f A)) (RPLAC x_f)), and compare[PLUS;(x_f A);(A B C)] is called. Compare proceeds as follows (PLUS has COMMUTE on its property list):

Step 1: Set var to NIL.
Step 2: Neither s1 or s2 is NIL, so go on to the next step.
Step 3: Car(s1) = x_f; set var to T; set s1 to (A); to to step 2.
Step 2: Go on to the next step.
Step 3: Car(s1) $\neq x_f$, so go on to the next step.

Step 4: Set s2 to (B C); set s1 to NIL; go to step 2.

Step 2: s1 is NIL; return car[simp[PLUS;((B)(C));NIL]] = (PLUS B C).

Trigsimp sets ae to (PLUS B C).

Step 5: is does not have the form (RPLAC r), so trigsimp returns ((PLUS A B C) F (PLUS B C)(x_f (PLUS x_f A)) (RPLAC x_f)), and this is the value of simplus.

On the next level of simplification, simp saves the descriptor list just generated by trigsimp. Since F is not represented by a simpfunction, simp handles the expression (F (PLUS B C)(PLUS A B C)) and calls trigsimp[F;((PLUS B C) (PLUS A B C))]:

Step 1: fs is set to ((F (PLUS B C)(x_f (PLUS x_f A)) (RPLAC x_f))).

Step 2: The car of the element of fs is F, so set fs to cdr[fs]-- in this case, NIL.

Step 3: Replace x_f by (PLUS B C) in (x_f (PLUS x_f A)), producing ((PLUS B C)(PLUS B C A)) (the subst1 function, which is used by trigsimp to make this replacement, simplifies the results--thus, the redundant PLUS is omitted from (PLUS (PLUS B C) A).) Compare[F;[[PLUS B C)(PLUS B C A));(PLUS B C)(PLUS A B C))] is called, and returns T,so go to step 5.

Step 5: is = (RPLAC x_f), so substitute (PLUS B C) for x_f in x_f, yielding (PLUS B C). This is simplified, and the result, ((PLUS B C)), is returned by trigsimp, and this is the value of simp.

Finally, on the top level of simplification, simplus calls trigsimp[PLUS;((F B (PLUS X A)) B C)]. There are no descriptors left over from lower-levels of simplification, and the second argument of trigsimp cannot be made to match the car of the descriptor-instruction pair for PLUS, so trigsimp, and, therefore, simplus and simplify, returns ((PLUS (F B (PLUS X A)) B C)).

Operators

Simplify has provision for applying three special operators -- differentiation and two substitution operators--to its argument. The response of simplify to an expression which requires the use of one of these operators is quite different from simplify's normal operation; for example, if simplify[(PLUS e_1 e_2)] is called, then e_1 and e_2 are simplified and then simplus is called; however, if simplify[(DIFF X e)] is executed, the function diffl[(X e)] is immediately called, before simplification of X and e. This is the reason for our distinction between operations and operators.

We shall first consider the differentiation operator: suppose that simplify[(DIFF X e)] is executed. Simplify finds that the indicator OPERATOR appears on the property list of DIFF, and that DIFFl follows the indicator, so simplify calls diffl[(X e)]. Diffl simply calls diff[X;car[simplify[e]]]. Diff[X;e] performs a straightforward differentiation:

If e is atomic, then diff returns 1 or 0, according as e is or is not eq to x.

If e is a sum, then diff differentiates each term with respect to X and returns the sum of the differentiated terms.

If e is a product, then diff differentiates each factor with respect to X and returns the appropriate sum of products.

If car[e] is an atom other than PLUS or TIMES, then diff fetches from the property list of car[e] a list which describes the form that the differentiated expression must have (this list is stored after the indicator GRADIENT). This list contains a list of variables in the general form (of which e is a special case) and each of the terms which must be summed in order to obtain the differentiated expression (one term for each variable). For example, the list stored on the property list of QUOTIENT after the indicator GRADIENT is:

((U V)(QUOTIENT V (EXPT V 2))(QUOTIENT U (EXPT V 2)))
or
((U V)(RECIP V)(QUOTIENT U (EXPT V 2))). (If the indicator GRADIENT does not appear on the property list of car[e], then diff calls error.) If the number of arguments of car[e] (that is, the length of cdr[e]) disagrees with the number of arguments in the general form, then error is called. Otherwise, diff substitutes the expressions in cdr[e] for the corresponding variables in the general form, differentiates each expression of cdr[e], with respect to X, multiplies each expression in the cdr of the general form by the expression resulting from differentiating the corresponding element of cdr[e], and returns the simplified sum of these products. Thus, diff does exactly what people do when differentiating.

If, car[e] is not an atom, then it is evaluated and its value applied to cdr[e]. Thus, it is possible to use compound differential operators.

The substitution cperators are called when simplify sees SUBST or SUBLIS as the main connective of the input expression. The only difference between the effects of these operators and the effects of the LISP functions with the same names is that the operators call simplify at appropriate times-- so that the outputs of the substitution operators are in simplified form.

In effect, simplify[(SUBST x Y e)] (the Y must be atomic) is equivalent to simplify[subst[x;Y;e]]; i.e. all occurrences of Y in e are replaced by x and the result is simplified. The actual operation of the substitution operators is not quite so straightforward, however, as an attempt is made to minimize the amount of simplifying that must be done by simplifying some expressions at intermediate stages of the substitution process.

Simplify[(SUBLIS ((Y_1 x_1)(Y_2 x_2) . . . (Y_n x_n)) e)] is equivalent to simplify[(SUBST x_1 Y_1 (SUBST x_2 Y_2 (. . . (SUBST x_n Y_n e) . . .)))].

The set of operators is by no means restricted to those we have discussed. To illustrate the ease with which new operators maybe added to the system, we propose a print operator which will cause the expression resulting from simplification of the expression immediately following the atom PRINT in the input to simplify to be printed out; but which has no other effect on the operation of the system.

We define: printop[s] = print[simplify[car[s]]]. The linkage to the simplify system is accomplished by merely executing deflist[((PRINT PRINTOP)); OPERATOR]. With these additions to the system, we have that simplify[(PLUS A (EXPT (PRINT (EXPT A 2) 3)) 2))] not only causes the simplified expression, ((PLUS A (EXPT A 12))), to be generated, but also causes the intermediate result (EXPT A 6) to be printed as soon as it is generated.

A user of the system who is well-acquainted with his problem might wish to turn on and off the recipmode, factor, expand, polyremainder, and store features by means of operator calls inserted judiciously into his input expression.

in retrospect (with some thoughts for the future)

We have seen many examples of simplify's capabilities. We have, however, restricted our attention to the right things that the system does. We consider now some of the wrong (or, at least questionable) things that can occur.

Although the program is able to collect common symbolic factors from a sum of products, it cannot collect common numeric factors. In particular, forms such as (PLUS (MINUS e_1) (MINUS e_2)) appear disturbingly often in the output of simplify as factors in products. This particular example might be simplified by causing collect2, when called by simplus, to minimize the number of "negative" terms in the output sum by negating each term and sending a message to simplus indicating that the sum should be negated. The general case, (PLUS (TIMES n e_1)(TIMES n e_2) . . .), might be handled by simply keeping track of the numeric factors occurring in the terms of the sum. If all terms have the same numeric factor, then take it outside the sum. A variation of this approach would allow minimizing the number of terms in the output expression which contain numeric factors; thus, simplify[(PLUS (TIMES 4 A)(TIMES 4 B) C)] = ((TIMES 4 (PLUS A B (TIMES 0.25 C))).

If the factor list is non-null, then polyquotient will always succeed in rewriting its arguments as polynomials. The results are sometimes unsatisfying. For example, if the factor list contains X, then simplify[(PLUS A (RECIP X))] = ((TIMES (POLY X A)(RECIP (POLY X 1 0)))), rather than ((TIMES (PLUS A X)(RECIP X))), which might be considered a simpler form. This difficulty may be avoided by having simplus call simpquotient, instead of polyquotient, when quotients must be simplified by simplus (see simplusreturnal).

The polyquotient function (and its satellites) was carefully written without the use of reverse. A result of this economy is that trailing zeroes common to both numerator and denominator are not, in general, cancelled. Thus, polyquotient[(POLY X 1 0);(POLY X 1 0 0)] = (TIMES (POLY X 1 0)(RECIP (POLY X 1 0 0))); not (RECIP (POLY X 1 0)), as might be expected. A straightforward (but brute force) modification would correct this situation; however, considerable reworking of polyquotient is necessary in order to achieve the desired result with minimum decrease in speed.

These examples emphasize the fact that the current program is not to be considered a finished product. It is the first draft--the next version will probably be written in LISP 2.

There is some doubt about the efficiency of using special routines to handle polynomials. It appears likely that all arithmetic (except, perhaps, division) will be done by the main simplify program in the next version of the system. At any rate, the polynomial programs should be less independent of the main simplify package if maximum economy is to be attained.

The problem of recognizing identities leads to the thought of using a table-lookup (or syntax compiler) scheme for simplification. Then identities would no longer be special cases. However, the difficulties presented by the associative and commutative operations case this scheme in a somewhat unfavorable light.

Hash coding for simplified expressions may be a feature of the next generation of simplify. There is no doubt that a large proportion of the time spent simplifying is devoted to repeating simplifications already done.

Perhaps we at Stanford are too involved with this particular simplify scheme to be able to view it without bias. Therefore, criticisms of this paper or of the program and suggestions for the new simplify program will be gratefully received.

APPENDIX A

```
        REMARK  SIMPLIFY SYSTEM OF 17 DECEMBER 1963
        SET     SIMPLIFY I -- SIMPLIFY THROUGH SIMPQUOTIENT

SPEAK NIL
LAP ((
(ALIST SUBR 0)
  (CLA $ALIST)
  (TRA 1 4)
)NIL)

COMMON((COLF RECIPMODE))
SPECIAL((FL AL COLIST TM N RSAV PSAV MM VSAV OP INVOP COL OPF
  KN L LO U E R IND1 IND2 NUFL))
SPECIAL((LI))
SPECIAL((X))


(LAMBDA (X)(COMPILE (DEFINE X)))((

(SIMPLIFY  (LAMBDA (X) (COND
  ((ATOM X) (SIMPATOM X))
  ((AND (ATOM (CAR X))) (FLAGP (CAR X) (QUOTE OPERATOR)))
    (APPLY (GET (CAR X) (QUOTE OPERATOR)) (LIST (CDR X)) (ALIST)))
  (T (SIMP (CAR X) (MAPLIST (CDR X) (FUNCTION
    (LAMBDA (J) (SIMPLIFY (CAR J)))
  )) (LIST X)))
)))

(SIMP (LAMBDA (OP LA OX) (PROG (FL AL R)
  (SETQ FL NIL)
  (SETQ AL NIL)
  (SIMP1 LA)
  (SETQ R (APPLY (COND
    ((ATOM OP) (CAR (PROP OP (QUOTE FSIMP)
      (FUNCTION (LAMBDA NIL (LIST
        (FUNCTION (LAMBDA (X)(TRIGSIMP OP X)))
      )))
    )))
    (T (FUNCTION (LAMBDA (X)(CONS (CONS OP X) NIL))))
  ) (LIST AL) (ALIST)))
  (SETQ R (DESCMAP R NIL NIL))
  (RETURN (COND
    ((EQUAL (CAR R) (CAR OX))(CONS (CAR OX) (CDR R)))
    (T R)
  ))
)))
```

```
(DESCMAP (LAMBDA (L IND1 IND2)(PROG (NUFL R)
  (SETQ NUFL NIL)
  (MAP FL (FUNCTION (LAMBDA (J) (COND
    ((EQ (CAAR J) (QUOTE FUNCTION)) (
      (LAMBDA (X) (SETQ R (COND
        ((NULL X) L)(T X)
      )))
      (APPLY (CADAR J)(COND
        ((NULL (CDDAR J))(LIST L))
        (T (LIST L (CADDAR J)))
      )(ALIST))
    ))
    ((OR IND1 IND2)(CONS (CAR J) NUFL))
    (T NIL)
  ))))
  (SETQ FL NUFL)
  (RETURN (COND (R R)(T L)))
)))

(SIMP1 (LAMBDA (L) (CONC
  ((NULL L) NIL)
  (T (PROG NIL
    (SIMP1 (CDR L))
    (SETQ FL (APPENC (CDAR L) FL))
    (SETQ AL (CONS (CAAR L) AL))
    (RETURN NIL)
  ))
)))

(SIMPCALL (LAMBI A (NAME L)(PROG (FL)
  (SETQ FL NIL)
  (RETURN (APPLY NAME (CONS L NIL)(ALIST)))
)))

(SIMPATOM (LAMBDA (A)(COND
  ((NUMBERP A)(CONS A NIL))
  (T ((LAMBDA (X)(COND
    ((EQ (CAAR X)(QUOTE RPLAC))(CDAR X))
    (T (CONS A X))
  ))(GET A (QUOTE VSIMP))))
)))
```

```
(SIMPLUS  (LAMBDA (L)(PROG (N TM COLIST VSAV U PSAV RSAV)
  (COND ((INN (QUOTE UNDEFINED) L)(RETURN
    (CONS (CONS (QUOTE UNDEFINED)(CONS (QUOTE PLUS) L)) NIL)
  )))
  (SETQ RSAV NIL)
  (SETQ PSAV 0)
  (SETQ VSAV NIL)
  (SETQ N 0)
  (SETQ TM NIL)
  (COLLECTOR L (QUOTE PLUS)(QUOTE MINUS)(QUOTE TIMES)(QUOTE PLUS)
      (FUNCTION (LAMBDA (X OP DUM INVOP COL OPF COLF MM)
        ((LAMBDA (Z)
          (SIMPLUS1 (CONS (CAR Z)
            (SIMPLUS3 (SIMPLUS2 (CADR Z)
              (GET (QUOTE FACTOR)(QUOTE COLFLAG))
            )(LIST NIL (CDR Z)))
          ))
        )(SIMPLUSPRIME (CDR X)) )
      ))
  )
  (RETURN (SIMPLUSRETURNA (DESCMAP TM NIL T)))
)))

(SIMPLUSRETURNA (LAMBDA (X)(PROG ()
  (SETQ COLIST NIL)
  (SETQ TM (COLLECT2 X NIL (QUOTE TIMES) 0))
  (COND ((NOT (ZEROP N)) (SETQ TM (CONS N TM))))
  (SETQ TM (SIMPLUSRETURN))
  (COND (RSAV (SETQ RSAV (CAR (SIMPCALL (QUOTE SIMPTIMES) RSAV)))))
  (RETURN (COND
    ((EQUAL PSAV 0)(SIMPLUSRETURNA1 (CAR TM) RSAV))
    (T (SIMPLUSRETURNA1 (POLYPLUS (COND
      (RSAV (POLYTIMES PSAV RSAV))(T PSAV)
    )(CAR TM)) RSAV))
  ))
)))

(SIMPLUSRETURNA1 (LAMBDA (X Y)(PROG (U)
  (COND ((NULL Y)(RETURN (CONS X (CDR TM)))))
  (SETQ U (GET (QUOTE SIMPLIFY)(QUOTE REMAINDER)))
  (POLYREMAINDER NIL)
  (SETQ RSAV (POLYQUOTIENT X Y))
  (POLYREMAINDER U)
  (RETURN (CONS RSAV (CDR TM)))
)))
```

```
(SIMPLUSRETURN (LAMBDA ()(COND
  ((NULL TM) (CONS 0 NIL))
  ((NULL (CDR TM)) (CONS (CAR TM) NIL))
  (RSAV (SIMPCALL (QUOTE SIMPLUS) TM))
  (T (TRIGSIMP (QUOTE PLUS) TM))
)))

(SIMPLUSPRIME (LAMBDA (Y)(COND
 ((NUMBERP (CAR Y))(CONS (TIMES MM (CAR Y))(CDR Y)))
 (T (CONS MM Y))
)))
(SIMPLUS1 (LAMBDA (Y)(COND
 ((NULL (CADR Y))(COLLECT
  (CAR (SIMPTIMES1 (CADDR Y)))(CAR Y) TM
 ))
  (T (COLLECT (SIMPLUS4 (CADR Y))
   (SIMPLUS4 (CONS (CAR Y)(CADDR Y))) TM
 ))
)))

(SIMPLUS2 (LAMBDA (X Y)(COND
 ((NULL Y) NIL)
 ((INN (CAR Y) X) (CAR Y))
 (T (SIMPLUS2 X (CDR Y)))
)))

(SIMPLUS3 (LAMBDA (X Y) (COND
 ((NULL X) Y)
 ((NULL (CADR Y)) Y)
 ((INN X (CAADR Y))((LAMBDA (X Y)(LIST (CONS X (CAR Y))(CADR Y)))
   (CAADR Y)(SIMPLUS3 X (LIST NIL (CDADR Y)))
 ))
 (T Y)
)))

(SIMPLUS4 (LAMBDA (X)(COND
 ((NULL (CDR X))(CAR X))
 (T (CONS (QUOTE TIMES) X))
)))
```

```
(SIMPMINUS  (LAMBDA (L)(COND
  ((INN (QUOTE UNDEFINED) L)
    (CONS (CONS (QUOTE UNDEFINED) (CONS (QUOTE MINUS) L)) NIL)
  )
  ((NUMBERP (CAR L)) (CONS (MINUS (CAR L)) NIL))
  ((EQ (CAAR L) (QUOTE MINUS)) (CONS (CADAR L) NIL))
  ((EQ (CAAR L)(QUOTE POLY))
      (CONS (CONS (QUOTE POLY)(CONS (CADAR L)
           (MAPLIST (CDDAR L)(FUNCTION (LAMBDA (X)
             (CAR (SIMPMINUS (CONS (CAR X) NIL)))
           )))
      )) NIL)
    )
  (T (TRIGSIMP (QUOTE MINUS) L))
)))

(SIMPDIFFERENCE (LAMBDA (L)(COND
  ((INN (QUOTE UNDEFINED) L)
    (CONS (CONS (QUOTE UNDEFINED)(CONS (QUOTE DIFFERENCE) L)) NIL)
  )
  (T (SIMP (QUOTE PLUS)(LIST (CONS (CAR L) NIL)
    (SIMPMINUS (CDR L))
  ) NIL))
)))
```

```
(SIMPTIMES  (LAMBDA (L)(PROG (N TM COLIST VSAV U PSAV RSAV)
  (COND ((INN (QUOTE UNDEFINED) L)(RETURN
    (CONS (CONS (QUOTE UNDEFINED)(CONS (QUOTE TIMES) L)) NIL)
  )))
  (SETQ RSAV NIL)
  (SETQ PSAV 1)
  (SETQ COLIST (GET (QUOTE FACTOR)(QUOTE COLFLAG)))
  (SETQ VSAV NIL)
  (SETQ N 1)
  (SETQ TM NIL)
  (COLLECTOR L (QUOTE TIMES) (QUOTE RECIP)(QUOTE EXPT)(QUOTE TIMES)
    (FUNCTION (LAMBDA (X OP DUM INVOP COL OPF COLF MM)(COND
      ((EQ (CAADR X)(QUOTE RECIP))(
        (LAMBDA (P)(COLLECT (CADADR X)
          (COND
            ((NUMBERP P)(MINUS (TIMES P MM)))
            (T (CAR (SIMPTIMES (LIST (MINUS MM) P))))
          )
          TM
        ))
        (CADDR X)
      ))
      (T (COLLECT (CADR X)(COND
          ((ONEP MM)(CADDR X))
          (T (CAR (SIMPCALL (QUOTE SIMPMINUS)(CDDR X))))
        ) TM))
    )
    ))
  )
  (SETQ TM (SIMPTIMESRETURN (COLLECT2
    (DESCMAP TM T T) NIL (QUOTE EXPT) 1
  )))
  (COND (RSAV (SETQ RSAV (CAR (SIMPCALL (QUOTE SIMPTIMES) RSAV)))))
  (SETQ TM (COND
    ((EQUAL PSAV 1)(SIMPLUSRETURN1 (CAR TM) RSAV))
    (T (SIMPLUSRETURN1 (POLYTIMES PSAV (CAR TM)) RSAV))
  ))
  (RETURN TM)
)))

(SIMPTIMESRETURN (LAMBDA (TM)(COND
    ((ZEROP N) (CONS 0 NIL))
    ((NULL TM)(CONS N NIL))
    ((MINUSP N)(COND
      ((EQUAL N -1)(SIMPMINUS (SIMPTIMES1 TM)) )
      (T (SIMPMINUS (TRIGSIMP (QUOTE TIMES)(CONS (MINUS N) TM))))
      ))
    ((ONEP N)(SIMPTIMES1 TM))
    (T (SIMPTIMES1 (CONS N TM)))
)))
```

```
(SIMPTIMES1 (LAMBDA (X)(COND
  ((NULL (CDR X))(CONS (CAR X) NIL))
  (T (TRIGSIMP (QUOTE TIMES) X))
)))

(SIMPRECIP  (LAMBDA (L)(COND
  ((POLYZERO (CAR L))
    (CONS (LIST (QUOTE UNDEFINED)(QUOTE RECIP) 0) NIL)
  )
  ((INN (QUOTE UNDEFINED) L)
    (CONS (CONS (QUOTE UNDEFINED)(CONS (QUOTE RECIP) L)) NIL)
  )
  ((NUMBERP (CAR L)) (CONS (RECIP (PLUS (CAR L) 0.0)) NIL))
  ((EQ (CAAR L) (QUOTE RECIP)) (CONS (CADAR L) NIL))
    ((EQ (CAAR L)(QUOTE EXPT))(COND
      (RECIPMODE (TRIGSIMP (QUOTE RECIP) L))
      (T (SIMP (QUOTE EXPT)
        (LIST (CONS (CADAR L) NIL)(SIMPMINUS (CDDAR L)))
      ))
    ))
  ((EQ (CAAR L)(QUOTE MINUS))(SIMP (QUOTE MINUS)
    (LIST (SIMPRECIP (CDAR L))) NIL
  ))
  (T (TRIGSIMP (QUOTE RECIP) L))
)))

(SIMPQUOTIENT (LAMBDA (L)(COND
  ((EQUAL (CADR L) 0)
    (CONS (CONS (QUOTE UNDEFINED)(CONS (QUOTE QUOTIENT) L)) NIL)
  )
  (T (PROG (U V)
    (COND ((NOT (OR
      (GET (QUOTE FACTOR)(QUOTE COLFLAG))
      (EQ (CAAR L)(QUOTE POLY))
      (EQ (CAADR L)(QUOTE POLY))
    ))(RETURN (SIMP (QUOTE TIMES)(LIST (CONS (CAR L) NIL)
      (SIMPRECIP (CDR L))
    ) NIL))))
    (SETQ U (GET (QUOTE SIMPLIFY)(QUOTE REMAINDER)))
      (POLYREMAINDER NIL)
      (SETQ V (POLYQUOTIENT (CAR L)(CADR L)))
      (POLYREMAINDER U)
      (RETURN (CONS V NIL))
))
)))

))
STOP)))
```

```
        SET     SIMPLIFY II -- SIMPEXPT THROUGH TRIGSIMP (AND SET-UP)
(LAMBDA (X)(COMPILE (DEFINE X)))((

(SIMPEXPT  (LAMBDA (L)(COND
  ((INN (QUOTE UNDEFINED) L)(CONS
    (LIST (QUOTE UNDEFINED)(QUOTE EXPT)(CAR L)(CADR L))
  NIL))
  ((AND (EQ (CAAR L)(QUOTE TIMES))
    (EXPANSION (CAR L)(GET (QUOTE SIMPLIFY)(QUOTE EXPAND)))
  )(SIMP (QUOTE TIMES)(MAPLIST (CDAR L)
    (FUNCTION (LAMBDA (X)(SIMPEXPT (LIST (CAR X)(CADR L)))))
  ) NIL))
  ((EQ (CAAR L)(QUOTE EXPT))(SIMP (QUOTE EXPT) (LIST
    (CONS (CADAR L) NIL) (SIMPTIMES (LIST
      (CADDAR L)(CADR L)
    ))
  ) NIL))
  ((AND (EQ (CAAR L)(QUOTE RECIP))(NOT RECIPMODE))
    (SIMP (QUOTE EXPT)(LIST (CONS (CADAR L) NIL)
      (SIMPMINUS (CDR L))
    ) NIL)
  )
  ((AND (EQ (CAADR L)(QUOTE MINUS)) RECIPMODE)
    (SIMP (QUOTE RECIP)(SIMP (QUOTE EXPT)(LIST
      (CONS (CADAR L) NIL)(SIMPMINUS (CDR L))
    ) NIL) NIL)
  )
  ((EQUAL (CAR L) 0)(CONS (COND
    ((EQUAL (CADR L) 0)
      (LIST (QUOTE UNDEFINED)(QUOTE EXPT) 0 0)
    )
    (T 0)
  ) NIL))
  ((EQUAL (CAR L) 1)(CONS 1 NIL))
  ((NUMBERP (CADR L))(COND
    ((ZEROP (CADR L))(CONS 1 NIL))
    ((ONEP (CADR L))(CONS (CAR L) NIL))
    ((NUMBERP (CAR L))(CONS (EXPT (CAR L)(CADR L)) NIL))
    ((AND (MINUSP (CADR L)) RECIPMODE)
      (SIMP (QUOTE RECIP)(LIST (SIMPEXPT (LIST
        (CAR L)(MINUS (CADR L))
      ))) NIL)
    )
    ((AND (FIXP (CADR L))(GREATERP (CADR L) 0)
      (OR (EQ (CAAR L)(QUOTE POLY))(EQ (CAAR L)(QUOTE PLUS)))
      (EXPANSION (CAR L)(GET (QUOTE SIMPLIFY)(QUOTE EXPAND)))
  )(CONS (POLYTIMES (CAR L)(CAR (SIMPEXPT (LIST (CAR L)
    (SUB1 (CADR L))
  )))) NIL))
```

```
    ((AND (FIXP (CADR L))(EQ (CAAR L)(QUOTE MINUS)))(COND
      ((ZEROP (REMAINDER (CADR L) 2))
        (SIMPEXPT (CONS (CADAR L)(CDR L)))
      )
      (T (SIMP (QUOTE MINUS)(LIST (SIMPEXPT (CONS
        (CADAR L)(CDR L)
      ))) NIL))
    ))
    (T (TRIGSIMP (QUOTE EXPT) L))
  ))
  (T (TRIGSIMP (QUOTE EXPT) L))
)))

(COLLECTOR (LAMBDA (L OP INVOP COL OPF COLF)(COLLECTOR1 L 1)))

(COLLECTOR1 (LAMBDA (L MM)(COND
  ((NULL L) NIL)
  (T (PROG2 (COLLECTOR1A L)(COLLECTOR1 (CDR L) MM)))
)))

(COLLECTOR1A (LAMBDA (L)(COND
  ((NUMBERP (CAR L))
    (SETQ N (EVAL (LIST OPF (TIMES N MM)(CAR L)) NIL))
  )
  ((ATOM (CAR L))(COLLECT (CAR L) MM TM))
  ((EQ (CAAR L) OP)(COLLECTOR1 (CDAR L) MM))
  ((EQ (CAAR L) INVOP)(COLLECTOR1 (CDAR L)(MINUS MM)))
    ((EQ (CAAR L) COL)
      (COLF (CAR L) OP (QUOTE DUM) INVOP COL OPF COLF MM)
    )
    ((EQ (CAAR L)(QUOTE MINUS))(PROG2
      (SETQ N (MINUS N))
      (COLLECTOR1 (CDAR L) MM)
    ))
  (T (COLLECT (CAR L) MM TM))
)))

(COLLECT (LAMBDA (E KN L)(COND
  ((AND (EQUAL KN 1)(EQ (CAR E) OP))(COLLECTOR1 (CDR E) MM))
  (T (COLLECT1 L))
)))
```

```
(COLLECT1 (LAMBDA (L)(COND
     ((NULL L)(SETQ TM (CONS (LIST E KN) TM)))
     ((MEQUAL E (CAAR L))(RPLACD (CAR L)(FPLUS KN (CDAR L))))
     ((AND (NUMBERP KN)(FIXP KN)(EQ OP (QUOTE TIMES))
         (NEGEQUAL E (CAAR L)) )(PROG2
       (COND
         ((ZEROP (REMAINDER KN 2)) NIL)
         (T (SETQ N (MINUS N)))
       )
       (RPLACD (CAR L)(FPLUS KN (CDAR L)))
     ))
     (T (COLLECT1 (CDR L)))
)))
```

```
(COLLECT2 (LAMBDA (LI LO OP I) (COND
  ((NULL LI)(APPEND VSAV LO))
  (T (COLLECT2 (CDR LI) (
    (LAMBDA (X) (COND
      ((EQUAL X I) LO)
      ((EQ (CAR X)(QUOTE DONTCOLLECT2THIS))(CDR X))
      ((AND (EQ (CAR X)(QUOTE POLY))(EQ OP (QUOTE TIMES)))(PROG2
        (SETQ PSAV (POLYPLUS PSAV (COND
          (RSAV (POLYTIMES RSAV X)) (T X)
        )))
        LO
      ))
      ((EQ (CAR X)(QUOTE POLY))(PROG2
        (SETQ PSAV (POLYTIMES PSAV X)) LO
      ))
      ((AND (EQ (CAR X)(QUOTE PLUS))(EQ OP (QUOTE EXPT))
        (EXPANSION X (GET (QUOTE SIMPLIFY)(QUOTE EXPAND))))
      )(PROG2
        (SETQ PSAV (POLYTIMES PSAV X))
        LO
      ))
      ((AND (EQ (CAR X)(QUOTE MINUS))(EQ (CAADR X)(QUOTE PLUS))
        (EQ OP (QUOTE EXPT))
        (EXPANSION X (GET (QUOTE SIMPLIFY)(QUOTE EXPAND)))
      )(PROG2
        (SETQ PSAV (CAR (SIMPMINUS (POLYTIMES PSAV X))))
        LO
      ))
      ((AND (EQ (CAR X)(QUOTE RECIP))(EQ OP (QUOTE EXPT)))
        (PROG2
          (SETQ RSAV (CONS (CADR X) RSAV))
          LO
        )
      )
      (T (PROG2
        (SETQ COLIST (COLLECT3 X COLIST))
        U
      ))
    ))
(COLLECT2PRIME (SIMPCALL (QUOTE SIMPLUS)(CDAR LI)))
  ) OP I ))
)))
```

```
(COLLECT2PRIME (LAMBDA (XX)(COND
      ((AND (EQ (CAAAR LI)(QUOTE RECIP))(EQ OP (QUOTE TIMES))
        (EXPANSION (CADAAR LI)(GET (QUOTE SIMPLIFY)(QUOTE EXPAND)))
      )(CONS (QUOTE DONTCOLLECT2THIS)(CONS (COND
          (RSAV (CAR (SIMPTIMES (CONS
            (CAR XX)(CDR (SETQ RSAV (CONS (CADAAR LI) RSAV)))
        ))))
        (T (PROG2 (SETQ RSAV (CONS (CADAAR LI) RSAV))(CAR XX) ))
      )(PROG2
        (SETQ PSAV (POLYTIMES PSAV (CADAAR LI)))
        (MAPLIST LO (FUNCTION (LAMBDA (X)
          (CAR (SIMPCALL (QUOTE SIMPTIMES)(LIST
            (CAR X)(CADAAR LI)
          )))
        )))
      ))))
    (T (COLLECT2PRIME1 (COND
        ((EQUAL 1 (CAR XX))(CAAR LI))
        (T (PROG (U V)
          (SETQ U (GET (QUOTE SIMPLIFY)(QUOTE EXPAND)))
          (COND ((EQ OP (QUOTE TIMES))(EXPAND NIL)))
          (SETQ V (CAR (SIMPCALL (GET OP (QUOTE FSIMP))
            (LIST (CAAR LI)(CAR XX))
          )))
          (EXPAND U)
          (RETURN V)
        ))
      )))
)))

(COLLECT2PRIME1 (LAMBDA (YY)(COND
      ((AND RSAV (EQ OP (QUOTE TIMES)))
        (CAR (SIMPCALL (QUOTE SIMPTIMES)(CONS YY RSAV)))
      )
      (T YY)
)))

(COLLECT3 (LAMBDA (X Y)(COND
  ((NULL Y)(PROG2 (SETQ U (CONS X LO)) NIL))
  ((INN (CAR Y) X)(PROG ()
    (COND ((CDR Y)(PROG2
      (SETQ U (APPEND VSAV LO))
      (SETQ VSAV NIL)
    )))
    (SETQ VSAV (CONS X VSAV))
  (RETURN (LIST (CAR Y)))
  ))
  (T (CONS (CAR Y)(COLLECT3 X (CDR Y))))
)))
```

```
(MEQUAL (LAMBDA (LA LB) (COND
  ((ATOM LA) (EQUAL LA LB))
  ((EQUAL (CAR LA) (CAR LB)) (COND
    ((AND (ATOM (CAR LA)) (FLAGP (CAR LA) (QUOTE COMMUTE)))
      (MEQUAL2 (CDR LA) (CDR LB)))
    (T (MEQUAL1 (CDR LA) (CDR LB)))
  ))
  (T F)
)))

(MEQUAL1 (LAMBDA (LA LB) (COND
  ((ATOM LA) (EQUAL LA LB))
  (T (AND (MEQUAL (CAR LA) (CAR LB)) (MEQUAL1 (CDR LA) (CDR LB))))
)))

(MEQUAL2 (LAMBDA (LA LB) (COND
  ((ATOM LA) (EQUAL  LA LB))
  ((MEQUAL3 (CAR LA) LB) (MEQUAL2 (CDR LA) (MEQUAL4 (CAR LA) LB)))
  (T F)
)))

(MEQUAL3 (LAMBDA (E S) (COND
  ((ATOM S) F)
  ((MEQUAL E (CAR S)) T)
  (T (MEQUAL3 E (CDR S)))
)))

(MEQUAL4 (LAMBDA (E S) (COND
  ((NULL S) NIL)
  ((MEQUAL E (CAR S)) (CDR S))
  (T (CONS (CAR S) (MEQUAL4 E (CDR S))))
)))

(ISEQUAL (LAMBDA (X1 X2)(OR
  (MEQUAL X1 X2)
  (POLYZERO (CAR (SIMPCALL (QUOTE SIMPDIFFERENCE)(LIST X1 X2))))
)))

(INEQUAL (LAMBDA (X1 X2)(COND
  ((EQ OP (QUOTE PLUS))(NEGEQUAL X1 X2))
  ((EQ OP (QUOTE TIMES))(RECEQUAL X1 X2))
  (T F)
)))

(RECEQUAL (LAMBDA (X1 X2)(EQUAL 1 ((LAMBDA (X)(COND
  ((EQ (CAR X)(QUOTE POLY))(CDDR X))(T X)
))(CAR (SIMPCALL (QUOTE SIMPTIMES)(LIST X1 X2))) ))) )
```

```
(NEGEQUAL (LAMBDA (T1 T2)
  (POLYZERO (CAR (SIMPCALL (QUOTE SIMPLUS)(LIST T1 T2))
))

(DELETE1 (LAMBDA (E S) (COND
  ((NULL S) NIL)
  ((EQUAL E (CAR S)) (CDR S))
  (T (CONS (CAR S) (DELETE1 E (CDR S))))
)))

(FPLUS (LAMBDA (N1 N2)(COND
  ((AND (NUMBERP N1)(NUMBERP (CAR N2)))(CONS (PLUS N1 (CAR N2)) NIL))
  (T (CONS N1 N2))
)))

(FLAGP (LAMBDA (A P) (COND
  ((NULL A) F)
    ((EQ (CAR A) P) T)
  (T (FLAGP (CDR A) P))
)))

(INN (LAMBDA (X L)(OR
  (EQUAL X L)
  (AND (NOT (ATOM L))
    (OR (INN X (CAR L))(INN X (CDR L)))
  )
)))

(POLYZERO (LAMBDA (Y)(COND
  ((OR (NULL Y)(AND (NUMBERP Y)(ZEROP Y))) T)
  ((ATOM Y) F)
  ((EQ (CAR Y)(QUOTE POLY))(POLYZERO (CDDR Y)))
  (T (AND (POLYZERO (CAR Y))(POLYZERO (CDR Y))))
)))

(FACTOR (LAMBDA (Y)(DEFLIST (LIST (LIST
  (QUOTE FACTOR)
  (COND
    ((ATOM Y) NIL)
    (T Y)
  )
))(QUOTE COLFLAG)))))

(EXPAND (LAMBDA (X)(SIMPFLAG X (QUOTE EXPAND))))

(POLYREMAINDER (LAMBDA (X)(SIMPFLAG X (QUOTE REMAINDER))))

(POLY (LAMBDA (X)(SIMPFLAG X (QUOTE POLYNOMIAL))))
(SIMPFLAG (LAMBDA (X Y)(DEFLIST (LIST (LIST
  (QUOTE SIMPLIFY)
  (COND
    ((EQ X (QUOTE YES))(QUOTE ALL))
    ((EQ X (QUOTE NO)) NIL)
    (T X)
  )
)) Y)))

(RECIPMODE (LAMBDA (X)(CSETQ RECIPMODE (EQ X (QUOTE R)))))

(TRIGSIMP (LAMBDA (OP L)(CONS (CONS OP L) NIL)))

))
RECLAIM NIL


FACTOR(NIL)
DEFLIST ((
(PLUS SIMPLUS)
(MINUS SIMPMINUS)
(DIFFERENCE SIMPDIFFERENCE)
(TIMES SIMPTIMES)
(RECIP SIMPRECIP)
(QUOTIENT SIMPQUOTIENT)
(EXPT SIMPEXPT)
(POLY SIMPOLY)
)FSIMP)
FLAG ((PLUS TIMES )COMMUTE )
RECIPMODE(R)
SPEAK NIL
UNSPECIAL((FL AL COLIST TM N RSAV PSAV MM VSAV OP INVOP COL OPF
   KN L LO U E R IND1 IND2 NUFL))
UNSPECIAL((LI))
UNSPECIAL((X))
UNCOMMON((COLF RECIPMODE))
STOP)) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) )) ))
```

```
       SET      POLYNOMIAL PACKAGE
SPECIAL((CHECK U E X R S SIGNAL LDCOF HARRAY COUNT V ZZZ YYY))
SPECIAL((Z))
SPECIAL((Y))


(LAMBDA (X)(COMPILE (DEFINE X)))((

(SIMPOLY (LAMBDA (L)(COND
((POLYZERO (CDR L))(CONS 0 NIL))
  (T (CONS (POLYWRITE (CONS (QUOTE POLY)(CONS (CAR L)(UNZERO (CDR L)))))
    (GET (QUOTE FACTOR)(QUOTE COLFLAG))
  ) NIL))
)))

(POLYPLUS (LAMBDA (Y Z)((LAMBDA (X)(COND
  ((POLYZERO X) 0)
  ((AND (EQ (CAR X)(QUOTE POLY))(NULL (CDDDR X)))(CADDR X))
  (T X)
))(COND
  ((NOT (EQ (CAR Z)(QUOTE POLY)))(COND
    ((EQ (CAR Y)(QUOTE POLY))(POLYPLUS Z Y))
    (T (CAR (SIMPCALL (QUOTE SIMPLUS) (LIST Y Z))))
  ))
  (T (CONS (QUOTE POLY)(CONS (CADR Z)
    (PROG (FL)
      (SETQ FL NIL)
      (RETURN (POLYPLUS1
        (REVERSE (CDDR (POLYWRITE Y (CADR Z))))
        (REVERSE (CDDR Z))
        NIL
      ))
    )
  )))
))))

(POLYPLUS1 (LAMBDA (Y Z X)(COND
  ((NULL Y)(COND
    ((NULL Z) X)
    (T (POLYPLUS1 (CDR Z) NIL (CONS (CAR Z) X)))
  ))
  ((NULL Z)(POLYPLUS1 (CDR Y) NIL (CONS (CAR Y) X)))
  (T (POLYPLUS1 (CDR Y)(CDR Z)
    (CONS (CAR (SIMPLUS (LIST (CAR Y)(CAR Z)))) X)
  ))
)))
```

```
(POLYTIMES (LAMBDA (Y Z)((LAMBDA (X)(COND
  ((POLYZERO X) 0)
  ((AND (EQ (CAR X)(QUOTE POLY))(NULL (CDDDR X)))(CADDR X))
  (T X)
))(PROG (FL)
  (SETQ FL NIL)
  (RETURN (COND
  ((NOT (EQ (CAR Z)(QUOTE POLY)))(COND
    ((EQ (CAR Y)(QUOTE POLY))(POLYTIMES Z Y))
    ((OR (EQ (CAR Y)(QUOTE PLUS))(EQ (CAR Z)(QUOTE PLUS)))
    (CAR (SIMPLUS (POLYEXPAND1 (POLYEXPAND Y)(POLYEXPAND Z))))
    )
    (T (CAR (SIMPTIMES (LIST Y Z))))
  ))
  (T ((LAMBDA (YY ZZ)(PROG (U YYY ZZZ)
    (SETQ YYY YY)
    (SETQ ZZZ (REVERSE (CDR ZZ)))
    (SETQ U (MAPLIST YYY (FUNCTION (LAMBDA (X)
      (CAR (SIMPTIMES (LIST (CAR X)(CAR ZZZ))))
    ))))
    (RETURN (CONS (QUOTE POLY)(CONS (CAR ZZ)(PROG2
      (MAP (CDR ZZZ)(FUNCTION (LAMBDA (X)
        (SETQ U (POLYTIMES1 YYY (CONS 0 U)))
      )))
      U
    ))))
  ))(CDDR (POLYWRITE Y (CADR Z)))(CDR Z)))
  ))
))))

(POLYTIMES1 (LAMBDA (Y Z)(COND
  ((NULL Y) Z)
  (T (CONS (CAR (SIMPLUS (CONS (CAR Z)
    (SIMPTIMES (LIST (CAR X)(CAR Y)))
  )))(POLYTIMES1 (CDR Y)(CDR Z))))
)))

(POLYEXPAND (LAMBDA (Y)(COND
  ((EQ (CAR Y)(QUOTE PLUS))(CDR Y))
  (T (CONS Y NIL))
)))

(POLYEXPAND1 (LAMBDA (Y Z)(COND
  ((NULL Y) NIL)
  (T (CONC
    (MAPLIST Z (FUNCTION (LAMBDA (X)
      (CAR (SIMPTIMES (LIST (CAR Y)(CAR X))))
    )))
    (POLYEXPAND1 (CDR Y) Z)
  ))
)))
```

```
(POLYQUOTIENT (LAMBDA (Y Z)((LAMBDA (X)(COND
  ((POLYZERO X) 0)
  ((AND (EQ (CAR X)(QUOTE POLY))(NULL (CDDDR X)))(CADDR X))
  (T X)
))
  (POLYQUOTIENT1 Y Z (GET (QUOTE FACTOR)(QUOTE COLFLAG)))
)))

(POLYQUOTIENT1 (LAMBDA (Y Z X)(COND
  ((POLYZERO Z)(LIST (QUOTE UNDEFINED)(QUOTE POLYQUOTIENT) Y Z))
  (X ((LAMBDA (S)(POLYQUOTIENT2 (POLYWRITE Y S)(POLYWRITE Z S)))
    ((LAMBDA (R)(COND
     (R R)
      ((EQ (CAR Z)(QUOTE POLY))(CADR Z))
      ((EQ (CAR Y)(QUOTE POLY))(CADR Y))
      (T NIL)
    ))(SIMPLUS2 (LIST Y Z) X))
  ))
  ((EQ (CAR Z)(QUOTF POLY))
    (POLYQUOTIENT2 (POLYWRITE Y (CADR Z))(POLYWRITE Z (CADR Z)))
  )
  ((EQ (CAR Y)(QUOTE POLY))
    (POLYQUOTIENT2 (POLYWRITE Y (CADR Y))(POLYWRITE Z (CADR Y)))
  )
  (T (POLYQUOTIENTRETURN Y (SIMPCALL (QUOTE SIMPRECIP)(CONS Z NIL))))
)))

(POLYQUOTIENT2 (LAMBDA (R S)(COND
  ((LESSP (LENGTH Y)(LENGTH Z))
    (POLYQUOTIENTRETURN Y (SIMPCALL (QUOTE SIMPRECIP)(LIST Z)))
  )
  ((EQ (CAR R)(QUOTE POLY))(PROG (CHECK FL)
  (SETQ CHECK NIL)
  (SETQ FL NIL)
  (RETURN (POLYQUOTIENT3 (CADR R)(POLYQUOTIENT4 (CDDR R)(CDDR S))))
  ))
  (T (POLYQUOTIENTRETURN Y
    (SIMPCALL (QUOTE SIMPRECIP)(CONS Z NIL))
  ))
)))

(POLYQUOTIENTRETURN (LAMBDA (Y S)(COND
  ((EQUAL Y 1) (CAR S))
  ((EQUAL S (QUOTE (1))) Y)
  ((OR (EQUAL Y 0)(EQUAL S (QUOTE (0)))) 0)
  ((EQ (CAR Y)(QUOTE TIMES))(NCONC Y S))
  ((EQ (CAR Y)(QUOTE MINUS))
    (LIST (QUOTE MINUS)(POLYQUOTIENTRETURN (CADR Y) S))
  )
  (T (LIST (QUOTE TIMES) Y (CAR S)))
)))
```

```
(POLYQUOTIENT3 (LAMBDA (X Q)(COND
  (CHECK (LIST (QUOTE TIMES) R (LIST (QUOTE RECIP) S)))
  ((POLYZERO ((ADR Q))(CONS (QUOTE POLY)(CONS X (CAR Q))))
  ((GET (QUOTE SIMPLIFY)(QUOTE REMAINDER))
    (LIST (QUOTE PLUS)
      (COND
        ((CDAR Q)(CONS (QUOTE POLY)(CONS X (CAR Q))))
        (T (CAAR Q))
      )
      ((LAMBDA (Z)(COND
        ((EQUAL (QUOTE (1))(CADR Q)) Z)
        (T (LIST (QUOTE TIMES)(COND
          ((CDADR Q)(CONS (QUOTE POLY)(CONS X (CADR Q))))
          (T (CAADR Q))
        ) Z))
      ))(LIST (QUOTE RECIP) S))
    )
  )
  (T (LIST (QUOTE TIMES) R (LIST (QUOTE RECIP) S)))
)))

(POLYQUOTIENT4 (LAMBDA (R S)(PROG (SIGNAL LDCOF)
  (SETQ SIGNAL (NOT (EQUAL (LENGTH R)(LENGTH S))))
  (SETQ SIGNAL (POLYQUOTIENT5 R S))
  (RETURN (LIST SIGNAL (UNZERO LDCOF)))
)))

(POLYQUOTIENT5 (LAMBDA (LN LD)(COND
  (SIGNAL (CONS
    (SETQ LDCOF (CAR (SIMPQUOTIENT (LIST (CAR LN)(CAR LD)))))
    (POLYQUOTIENT5 (POLYQUOTIENT6 (CDR LN)(CDR LD)) LD)
  ))
  (T (PROG ()
    (SETQ SIGNAL (SIMPQUOTIENT (LIST (CAR LN)(CAR LD))))
    (SETQ LDCOF (CAR SIGNAL))
    (SETQ LDCOF (POLYQUOTIENT6 (CDR LN)(CDR LD)))
    (RETURN SIGNAL)
  ))
)))
```

```
(POLYQUOTIENT6 (LAMBDA (LN LD)(COND
  ((NULL LD)(PROG2
    (COND
      ((CDR LN) NIL)
      (T (SETQ SIGN/L NIL))
    )
    LN
  ))
  ((NULL LN)(SETQ CHECK T))
  (T (CONS (CAR (SIMPDIFFERENCE
    (LIST (CAR LN)(CAR (SIMPTIMES (LIST LDCOF (CAR LD)))))
  ))(POLYQUOTIENT6 (CDR LN)(CDR LD))))
)))


(POLYWRITE (LAMBDA (L X)(COND
  ((NULL X)(COND
    ((EQ (CAR L)(QUOTE POLY))(POLYWRITE L (CADR L)))
    (T L)
  ))
  ((ATOM L)(COND
    ((EQ L X)(LIST (QUOTE POLY) X 1 0))
    (T (LIST (QUOTE POLY) X L))
  ))
  ((NOT (ATOM X))(COND
    ((INN (CAR X) L)(POLYWRITE L (CAR X)))
    (T (POLYWRITE L (CDR X)))
  ))
  ((OR (EQ (CAR L)(QUOTE RECIP))(EQ (CAR L)(QUOTE MINUS)))
    (CAR (SIMP (CAR L)
      (LIST (LIST (POLYWRITE (CADR L) X))) NIL
    ))
  )
  ((AND (EQ (CAR L)(QUOTE POLY))(EQUAL (CADR L) X)
    (NOT (INN X (CDDR L)))
  ) L)
  (T (PROG (U V W)
    (SETQ U (GET (QUOTE FACTOR)(QUOTE COLFLAG)))
    (FACTOR (LIST X))
    (SETQ W (GET (QUOTE SIMPLIFY)(QUOTE EXPAND)))
    (EXPAND (QUOTE ALL))
    (SETQ V (POLYWRITE1 (CAR (SIMPLIFY (POLYUNWRITE L))) X))
    (EXPAND W)
    (FACTOR U)
    (RETURN V)
  ))
)))
```

```
(POLYWRITE1 (LAMBDA (L X)(COND
  ((EQ (CAR L)(QUOTE RECIP))
    (LIST (QUOTE POLY) X (LIST (QUOTE RECIP)(POLYWRITE1 (CADR L) X)))
  )
  (T (PROG (COUNT HARRAY V U)
    (SETQ COUNT -1)
    (SETQ HARRAY NIL)
    (SETQ V X)
    (SETQ U NIL)
    (COND
      ((EQ (CAR L)(QUOTE PLUS))
        (MAP (CDR L)(FUNCTICN (LAMBDA (X)
          (SETQ HARRAY (POLYWRITE2 (POLYWRITE3 (CAR X))))
        )))
      )
      (T (SETQ HARRAY (POLYWRITE2 (POLYWRITE3 L))))
    )
    (SETQ HARRAY (CONS (QUOTE POLY)(CONS V HARRAY)))
      (RETURN (COND
        ((NULL U) HARRAY)
        (T (LIST (QUOTE PLUS) U HARRAY))
      ))
  ))
)))

(POLYWRITE2 (LAMBDA (X)(COND
    ((NULL X) HARRAY)
    (T (SETARRAY (DIFFERENCE COUNT (CAR X))(CAR X)(CDR X)))
)))
```

```
(POLYWRITE3 (LAMBDA (Z)(COND
      ((EQ (CAR Z)(QUOTE MINUS))(
        (LAMBDA (P)(COND
          (P (CONS (CAR P)
            (CAR (SIMP (QUOTE MINUS)(LIST (LIST (CDR P))) NIL))
          ))
        (T (SETQ U (CONS
            (CAR (SIMP (QUOTE MINUS)(LIST (LIST (CAR U))) NIL))
            (CDR U)
        )))
        ))
        (POLYWRITE3 (CADR Z))
      ))
  ((AND (EQ (CAR Z)(QUOTE TIMES))(NUMBERP (CADR Z)))
    (POLYWRITE3 (CONS (QUOTE TIMES)(CONS
      (CADDR Z)(CONS (CADR Z)(CDDDR Z))
    )))
  )
    ((EQUAL Z V)(CONS 1 1))
    ((AND (EQ (CAR Z)(QUOTE EXPT))(EQUAL (CADR Z) V)
      (NUMBERP (CADDR Z))(FIXP (CADDR Z))
    )(COND
      ((MINUSP (CADDR Z))(PROG2
        (SETQ U (CONS Z U))  NIL
      ))
      (T (CONS (CADDR Z) 1))
    ))
    ((AND (EQ (CAR Z)(QUOTE TIMES))(EQUAL (CADR Z) V))
      (CONS 1 (CAR (SIMPCALL (QUOTE SIMPTIMES)(CDDR Z))))
    )
    ((AND (EQ (CAR Z)(QUOTE TIMES))
        (EQ (CAADR Z)(QUOTE EXPT))(EQUAL (CADADR Z) V)
        (NUMBERP (CADDR (CADR Z)))(FIXP (CADDR (CADR Z)))
    )(COND
      ((MINUSP (CADDR (CADR Z)))(PROG2
        (SETQ U (CONS Z U))  NIL
      ))
      (T (CONS (CADDR (CADR Z))
        (CAR (SIMPCALL (QUOTE SIMPTIMES)(CDDR Z)))
      ))
    ))
    ((INN V Z)(PROG2 (SETQ U (CONS Z U)) NIL))
    (T (CONS 0 Z))
)))
```

```
(SETARRAY (LAMBDA (NM M E)(COND
  ((MINUSP NM)(PROG2
    (SETQ COUNT M)
    (CONS E (SETARRAY1 (DIFFERENCE -1 NM)))
  ))
  (T (SETARRAY2 NM HARRAY))
)))

(SETARRAY1 (LAMBDA (N)(COND
  ((ZEROP N) HARRAY)
  (T (CONS 0 (SETARRAY1 (SUB1 N))))
)))

(SETARRAY2 (LAMBDA (N H)(COND
((ZEROP N)
    (CONS (CAR (SIMPCALL (QUOTE SIMPLUS)(LIST E (CAR H))))(CDR H))
  )
  (T (CONS (CAR H)(SETARRAY2 (SUB1 N)(COND
    ((CDR H)(CDR H))
    (T (CONS 0 NIL))
  ))))
)))

(POLYUNWRITE (LAMBDA (L)(COND
  ((ATOM L) L)
  ((EQ (CAR L)(QUOTE POLY))
    (CAR (SIMPCALL (QUOTE SIMPLUS)
      (POLYUNWRITE1 (POLYUNWRITE (CDDR L))(CADR L))
    ))
  )
  (T (CONS (POLYUNWRITE (CAR L))(POLYUNWRITE (CDR L))))
)))

(POLYUNWRITE1 (LAMBDA (Y X)(POLYUNWRITE2  (REVERSE Y) 0)))

(POLYUNWRITE2 (LAMBDA (Y N)(COND
  ((NULL Y) NIL)
  (T (CONS (CAR (SIMP (QUOTE TIMES) (LIST
    (SIMP (QUOTE EXPT)(LIST (LIST X)(LIST  N)) NIL)
    (CONS (CAR Y) NIL)
  ) NIL))(POLYUNWRITE2 (CDR Y)(ADD1 N)) ))
)))

(UNZERO (LAMBDA (L)(COND
  ((AND (NOT (ATOM L))(EQUAL 0 (CAR L)))(UNZERO (CDR L)))
  (T L)
)))
```

```
(EXPANSION (LAMBDA (L X)(OR
  (EQ X (QUOTE ALL))
  (AND X (OR (INN (CAR X) L)(EXPANSION L (CDR X))))
)))

))
UNSPECIAL((CHECK U E X R S SIGNAL LDCOF HARRAY COUNT V ZZZ YYY))
UNSPECIAL((Z))
UNSPECIAL((Y))
  STOP)) )) )) )) )) ) )) )) ))))
```

```
        SET     OPERATOR PACKAGE -- DIFFERENTIATE AND SUBSTITUTE
SPECIAL((E V J L))


(LAMBDA (X)(COMPILE (DEFINE X)))((

(DIFF (LAMBDA (V E) (COND
  ((EQ E V) (SIMPATOM 1))
  ((ATOM E) (SIMPATOM 0))
  ((EQ (CAR E) (QUOTE PLUS)) (SIMP (QUOTE PLUS)
    (MAPLIST (CDR E) (FUNCTION  (LAMBDA (J) (DIFF V (CAR J)))))
    (LIST E)
  ))
  ((EQ (CAR E) (QUOTE TIMES)) (SIMP (QUOTE PLUS)
    (MAPLIST (CDR E) (FUNCTION (LAMBDA (J)
      (SIMP (QUOTE TIMES) (MAPLIST (CDR E) (FUNCTION (LAMBDA (K) (COND
        ((EQ K J) (DIFF V (CAR K)))
        (T (SIMPLIFY (CAR K)))
      )))) NIL )
    )))
    (LIST E)
  ))
  ((ATOM (CAR E)) (
    (LAMBDA (GRAD) (SIMP (QUOTE PLUS) (DIFF2
      (PAIR (CAR GRAD) (MAPLIST (CDR E) (FUNCTION
        (LAMBDA (J) (SIMPLIFY (CAR J)))
      )))
      (CDR E)
      (CDR GRAD)
    ) (LIST E)))
    (CAR (PROP (CAR E) (QUOTE GRADIENT) (FUNCTION
     (LAMBDA NIL (ERROR (LIST (QUOTE (NO GRADIENT)) E)))
    )))
  ))
  (T (APPLY (CAR L) (LIST (CDR L)) (ALIST)))
)))

(DIFF1 (LAMBDA (L) (DIFF (CAR L) (CAR (SIMPLIFY (CADR L))))))

(DIFF2 (LAMBDA (VL EL GL) (COND
  ((NULL GL) (COND
    ((NULL EL) NIL)
    (T (ERROR (LIST (QUOTE $$.WRONG NUMBER OF ARGS FOR.) E)))
  ))
  (T (CONS (SIMP (QUOTE TIMES) (LIST
    (MATHSUB VL (CAR GL))
    (DIFF V (CAR EL))
  ) NIL )(DIFF2 VL (CDR EL) (CDR GL))))
)))
(MATHSUB (LAMBDA (L E) (MATHS2 (MAPLIST L (FUNCTION
  (LAMBDA (J) (CONS (CAAR J) (SIMPLIFY (CADAR J))))
)) E )))

(MATHS1 (LAMBDA (L) (SIMPLIFY (CAR (MATHSUB (CAR L) (CADR L))))))

(MATHS2 (LAMBDA (L E) (COND
  ((ATOM E) (MATHS3 L))
  (T (SIMP (CAR (MATHS2 L (CAR E))) (MAPLIST (CDR E)
    (FUNCTION (LAMBDA (J) (MATHS2 L (CAR J))))
  ) (LIST E)))
)))

(MATHS3 (LAMBDA (L) (COND
  ((NULL L) (SIMPATOM E))
  ((EQ (CAAR L) E) (CDAR L))
  (T (MATHS3 (CDR L)))
)))

(MATHS4 (LAMBDA (L) (SIMPLIFY
  (CAR (MATHSUB (LIST (LIST (CADR L) (CAR L))) (CADDR L)))
)))

))
UNSPECIAL((E V J L)).
DEFLIST ((
(SUBLIS MATHS1)
(DIFF DIFF1)
(SUBST MATHS4)
)OPERATOR)
DEFLIST((
(MINUS ((X) -1))
(DIFFERENCE ((X Y) 1 -1))
(RECIP ((X) (MINUS (RECIP (EXPT X 2)))))
(QUOTIENT ((U V) (QUOTIENT V (EXPT V 2)) (MINUS (QUOTIENT U (EXPT V 2)))
 ))
(EXPT ((U V)
  (TIMES V (EXPT U (DIFFERENCE V 1)))
  (TIMES (EXPT U V) (LOG U))
))
(LOG ((X) (RECIP X)))
(SIN ((X) (COS X)))
(COS ((X) (MINUS (SIN X))))
) GRADIENT)

STOP)))))
```