Maxima by Example: Ch. 2, Algebra, Part 1 [*]

Edwin L. Woollett

June 18, 2008

Contents

[*]Uses Maxima 5.14. This is a live document. Check `http://www.csulb.edu/~woollett/` for the latest version of these notes. Send comments and suggestions to `woollett@charter.net`

2 Algebra, Part 1

2.1 Introduction

There are many ways to manipulate algebraic expressions in Maxima.

One motivation for massaging an expression is to try to get the expression in a form (or close to a form) such that it will be easy to compare with other results.

Another motivation is to try to get the expression in the simplest (usually that means the most compact) form. However, simplicity is somewhat in the eye of the beholder. Sometimes, the pursuit of simplicity is a false path which really leads to no real progress, and one must be satisfied with partial simplification.

Another motivation for manipulating an expression is to make it easier to replace parts of the expression with replacements.

One often resorts to using a computer algebra system (CAS) like Maxima because the algebra one must deal with in a given problem is long and tedious, and it is easy for small errors to creep in when the algebra is done ”by hand”. In this case, the CAS provides a check on possible mistakes, as long as there are no ”bugs” in the CAS functions which affect your calculation. In any case, one should always make as many consistency checks on a result as possible, looking at convenient limits of parameters, for example, or looking at numerical spot checks. One way to be extra careful is to make numerical checks with the first raw form of the expression, and compare with the numerical values of the simplified expression, but this would only be recommended if there are other indications something has gone wrong.

In chapter 1, Getting Started, we presented the notion of Maxima ”expressions”. In this chapter, we will present tools for and examples of expanding, simplifying, and factoring expressions, and finally, making substitutions in expressions. In the next chapter, we present examples of extracting parts of an expression for subsequent use with other commands, give examples of using and simplifying expressions which involve trigonometric functions, and also discuss maxima tools for evaluations of summations. There will be, inevitably, some overlap and circularity in the use of these tools, and these two chapters on algebraic manipulations should be read at least twice.

2.2 Expanding Expressions

Maxima provides several expansion related commands, each of which expands its argument in a different way. The **expand**(expression) function, for example, multiplies out product sums (ie., distributing sums over products) and exponentiated sums, expanding sub-expressions on all levels of the expression. The **distrib** function also distributes sums over products, but works only at the top level of the expression. (*vide infra*)

This section presents the following commands:

- **expand** expands the given expression by multiplying out products of sums and exponentiated sums at all levels of the expression. The option variables **logexpand** and **radexpand** are presented.
- **ratexpand** should normally be used for expanding polynomials, rather than **expand**, since **ratexpand** is much more efficient for high order polynomials.
- **multthru** multiplies a term or terms through a sum or equation.
- **distrib** expands the given expression by distributing sums over products.
- **partfrac** does a complete partial fraction decomposition, expanding an expression in partial fractions with respect to a given main variable.

2.2.1 expand, logexpand, radexpand

Function: **expand** `(expr)`
Expands the expression `expr`.
Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplication (commutative and non-commutative) is distributed over addition at all levels of expr.
For polynomials one should usually use **ratexpand** which uses a more efficient algorithm.
Expansion occurs only for terms with "internal representation exponents" n such that **maxnegex** $\leq n \leq$ **maxposex**. These option variables are described below.

Function: **expand** `(expr, p, n)`
Expands `expr`, using `p` for **maxposex** and `n` for **maxnegex**. Both p and n should be non-negative integers. This is useful in order to expand part but not all of an expression.

maxposex *default:* 1000
Controls the action of **expand**. If a term has an internal representation exponent more positive than **maxposex**, expansion is inhibited.

maxnegex *default:* 1000
Controls the action of **expand**. If a term has an internal representation exponent more negative than −**maxnegex**, expansion is inhibited.

expop *default:* **0**
expop is the highest positive exponent which is automatically expanded. By default, `(x+1)^3`, when entered, will not automatically be expanded. The term will be automatically expanded only if `expop` is greater than or equal to 3.

```
(%i4) (x+1)^3;
                                            3
(%o4)                              (x + 1)
(%i5) expop;
(%o5)                                 0
(%i6) expop:3$
(%i7) (x+1)^3;
                                3      2
(%o7)                          x  + 3 x  + 3 x + 1
```

expon *default:* **0**
By default `(x+1)^(-5)` will not be automatically expanded. Only if **expon** $\geq$ **5**, will this term be automatically expanded.

```
(%i8) expon;
(%o8)                                 0
(%i9) (1+x)^(-5);
                                   1
(%o9)                          --------
                                      5
                               (x + 1)
```

```
(%i10) expon:6$
(%i11) (1+x)^(-5);
                                          1
(%o11)               -----------------------------------
                      5      4       3       2
                     x  + 5 x  + 10 x  + 10 x  + 5 x + 1
(%i12) [expop:0,expon:0]$
(%i13) [(1+x)^3, (1+x)^(-5) ];
                                        3      1
(%o13)                        [(x + 1) , --------]
                                                5
                                         (x + 1)
```

The `expand` flag used with `ev` causes expansion. With the default values of 0 for **expop** and **expon**, we begin with an expression with no automatic expansion performed:

```
(%i1) expr1: (1/(a + b)^2 + x/(a - b)^3 ) ^2 ;
                                   x          1      2
(%o1)                        (-------- + --------)
                                     3          2
                               (a - b)    (b + a)
```

Maxima effectively holds this definition in an internal form

```
        ( x*(a - b)^(-3) + (a + b)^(-2) )^2
```

when deciding if an exponent is positive or negative. Now expand $expr1$, assigning the result to $expr2$.

```
(%i2) expr2 : expand(expr1);
                                      2
                                     x
(%o2) ----------------------------------------------------------------
        6        5        2  4        3  3        4  2       5       6
       b  - 6 a b  + 15 a  b  - 20 a  b  + 15 a  b  - 6 a  b + a
                             2 x
 + -------------------------------------------
      5      4       2  3       3  2     4      5
   - b  + a b  + 2 a  b  - 2 a  b  - a  b + a
                          1
 + -----------------------------------
    4        3       2  2       3      4
   b  + 4 a b  + 6 a  b  + 4 a  b + a
```

With additional arguments, **expand** multiplies out specific terms only; `(%i3)` expands those terms of expr1 which have positive exponents between 2 and 0 and expands no term which has a negative exponent.

```
(%i3) expr3 : expand(expr1, 2, 0 );
                             2
                            x               2 x              1
(%o3)                   -------- + ----------------- + --------
                               6          3        2          4
                         (a - b)    (a - b)  (b + a)    (b + a)
```

Now expand only the negative exponents of expr1 through either (-2) or through (-3):

```
(%i5) expand(expr1, 0, 2 );
                                x                1        2
(%o5)                      (-------- + ---------------)
                                    3    2           2
                             (a - b)    b  + 2 a b + a
(%i6) expand(expr1, 0, 3 );
                                x                         1        2
(%o6)           (------------------------- + ---------------)
                     3        2       2       3    2            2
                 - b  + 3 a b  - 3 a  b + a    b  + 2 a b + a
```

You can declare a variable to be a ”main variable” as a way to alter the output from **expand**. The default ordering is described in the Maxima manual section on Simplification under the entry ”Declaration: **mainvar**”, which reads:

> Declaration: **mainvar**
> You may declare variables to be `mainvar`. The ordering scale for atoms is essentially:
> `numbers < constants (e.g., %e, %pi) < scalars < other variables < mainvars.`
> E.g., compare `expand( (x + y)^4 )` with `(declare (x, mainvar), expand ((x+y)^4))`.

Here is that comparison:

```
(%i1) expand( (x+y)^4 );
                           4        3       2  2       3      4
(%o1)                     y  + 4 x y  + 6 x  y  + 4 x  y + x
(%i2) ( declare(x, mainvar), expand( (x + y)^4 ) );
                           4        3       2  2       3      4
(%o2)                     x  + 4 y x  + 6 y  x  + 4 y  x + y
```

You can set option variables to control how much and what kind of expansions are to take place. The option variable **logexpand** controls the expansion of logarithms of products and powers, and **radexpand** controls the expansion of expressions containing radicals. The Maxima manual entry for **logexpand** is:

> Option variable: **logexpand**
> Default value: `true`
> Causes `log(a^b)` to become `b*log(a)`. If it is set to `all`, `log(a*b)` will also simplify to `log(a)+log(b)`. and `log(a/b)` will also simplify to `log(a)-log(b)` for rational numbers `a/b, a#1`. `(log(1/b)`, for integer b, always simplifies.) If it is set to `false`, all of these simplifications will be turned off.

The **log** command gives the natural log of its argument in base e.

```
(%i1) log(%e);
(%o1)                                   1
```

When **logexpand** is **true**, Maxima does not simplify the logarithms of products and quotients.

```
(%i2) logexpand;
(%o2)                                 true
(%i3) log(a*b);
(%o3)                               log(a b)
(%i4) log(a/b);
                                           a
(%o4)                                log(-)
                                           b
```

Resetting **logexpand** to **all** tells Maxima to simplify these logarithms.

```
(%i5) logexpand : all$
(%i6) log(a*b);
(%o6)                          log(b) + log(a)
(%i7) log(a/b);
(%o7)                          log(a) - log(b)
```

The Maxima manual entry for **radexpand** is:

> Option variable: `radexpand`
> Default value: `true`
> `radexpand` controls some simplifications of radicals. When `radexpand` is `all`, causes nth roots of factors of a product which are powers of n to be pulled outside of the radical. E.g. if `radexpand` is `all`, `sqrt (16*x^2)` simplifies to `4*x`. More particularly, consider `sqrt (x^2)`.
>
> - If `radexpand` is `all` or `assume (x > 0)` has been executed, `sqrt(x^2)` simplifies to `x`.
> - If `radexpand` is `true` and `domain` is `real` (its default), `sqrt(x^2)` simplifies to `abs(x)`.
> - If `radexpand` is `false`, or `radexpand` is `true` and `domain` is `complex`, `sqrt(x^2)` is not simplified.
>
> Note that `domain` only matters when `radexpand` is `true`.

When **radexpand** is **true**, Maxima does not simplify radicals containing products, quotients, and powers.

```
(%i8) radexpand;
(%o8)                               true
(%i9) sqrt(x^y);
                                        y
(%o9)                            sqrt(x )
(%i10) sqrt(x*y);
(%o10)                           sqrt(x y)
(%i11) sqrt(x/y);
                                        x
(%o11)                            sqrt(-)
                                        y
```

Resetting **radexpand** to **all** tells Maxima to simplify these radicals.

```
(%i12) radexpand : all$
(%i13) sqrt(x^y);
                                    y/2
(%o13)                             x
(%i14) sqrt(x*y);
(%o14)                        sqrt(x) sqrt(y)
(%i15) sqrt(x/y);
                                  sqrt(x)
(%o15)                            -------
                                  sqrt(y)
(%i16) sqrt(x^2);
(%o16)                               x
```

For more information on the kinds of option variables that affect expansion, type `?? expand` (no semi-colon, just press Enter).

2.2.2 ratexpand

The Maxima function **ratexpand** is faster than **expand** for polynomials and also works harder to find and cancel common factors in rational expressions. Here are two examples which compare **expand** with **ratexpand**, the first from the manual, the second from the mailing list.

```
(%i1) e1 : (x - 1)/(x + 1)^2  + 1/(x-1);
                                   x - 1       1
(%o1)                             -------- + -----
                                         2   x - 1
                                   (x + 1)
(%i2) expand(e1);
                           x             1           1
(%o2)                 ------------ - ------------ + -----
                       2              2             x - 1
                      x  + 2 x + 1   x  + 2 x + 1
(%i3) ratexpand(e1);
                                 2
                              2 x              2
(%o3)                  --------------- + ---------------
                        3    2            3    2
                       x  + x  - x - 1   x  + x  - x - 1
(%i4) e2:(a+b+c)/a^2/(b+1)^2;
                                    c + b + a
(%o4)                              -----------
                                    2        2
                                   a  (b + 1)
(%i5) expand(e2);
               c                    b                     a
(%o5)   ------------------- + ------------------- + -------------------
         2  2      2      2    2  2      2      2     2  2      2      2
        a  b  + 2 a  b + a    a  b  + 2 a  b + a    a  b  + 2 a  b + a
(%i6) ratexpand(e2);
                c                     b                   1
(%o6)    ------------------- + ------------------- + ----------------
          2  2      2      2    2  2      2      2       2
         a  b  + 2 a  b + a    a  b  + 2 a  b + a    a b  + 2 a b + a
```

2.2.3 multthru

The Maxima manual entry is:

> Function: `multthru (expr)`
> Function: `multthru (expr_1, expr_2)`
> Multiplies a factor (which should be a sum) of `expr` by the other factors of `expr`.
>
> That is, `expr` is `f_1 f_2 ... f_n` where at least one factor, say `f_i`, is a sum of terms. Each term in that sum is multiplied by the other factors in the product. (Namely all the factors except `f_i`). `multthru` does not expand exponentiated sums. This function is the fastest way to distribute products (commutative or noncommutative) over sums. Since quotients are represented as products `multthru` can be used to divide sums by products as well.
>
> `multthru (expr_1, expr_2)` multiplies each term in `expr_2` (which should be a sum or an equation) by `expr_1`. If `expr_1` is not itself a sum then this form is equivalent to `multthru (expr_1*expr_2)`.

Here is an example of the use of **multthru**:

```
(%i1) expr1: (1/(a + b)^2 + x/(a - b)^3 ) ^2 ;
                                      x          1     2
(%o1)                           (-------- + --------)
                                          3         2
                                   (a - b)    (b + a)
(%i2) expr3 : expand(expr1, 2, 0 );
                           2
                          x              2 x             1
(%o2)                 -------- + ----------------- + --------
                             6          3        2          4
                      (a - b)    (a - b)  (b + a)    (b + a)
```

Multiply each term in the sum expr3 by `(a - b)^4`.

```
(%i3) multthru( (a - b)^4, expr3 );
                              2                            4
                             x       2 (a - b) x    (a - b)
(%o3)                    -------- + ----------- + --------
                                2             2          4
                         (a - b)      (b + a)      (b + a)
```

As a second example,let’s write down an equation:

```
(%i9) eq1 : a = 23*x^2 + y^2 - y^3/(a-b);
                                         3
                                        y       2       2
(%o9)                          a = - ----- + y  + 23 x
                                      a - b
```

We now multiply this equation by `(a-b)`.

```
(%i10) multthru( (a-b),eq1 );
                                      3            2                2
(%o10)              a (a - b) = - y  + (a - b) y  + 23 (a - b) x
```

As a third example, consider the following expr4 which contains factors of `(s - t)^n`.

```
(%i11) expr4 : ( (b+a)^10*(s-t)^2 + 2*a*b*(s-t) +
                  a^2*b^2*(s-t) )/a/b/(s-t)^4 ;
                    10        2    2  2
              (b + a)   (s - t)  + a  b  (s - t) + 2 a b (s - t)
(%o11)        --------------------------------------------------
                                              4
                                  a b (s - t)
```

Decompose expr4 into a sum of terms, in each of which cancellation of factors of `(s - t)` will occur.

```
(%i12) multthru(expr4);
                                10
                         (b + a)          a b          2
(%o12)                ------------ + -------- + --------
                                  2          3          3
                      a b (s - t)    (s - t)    (s - t)
```

Remember the denominator of expr4 is internally held as a product, so

```
        (A   + B + C)/D   ==>   (A   + B   + C)*D^(-1)
```

and the effect of **multthru**(expr4) is to produce a sum of terms which is

```
         A*D^(-1)   +   B*D^(-1)   + C*D^(-1)
```

For the simplest examples, there will be no difference in the result of using **exand** or **multthru**:

```
(%i13) multthru( a*b*(c + d) );
(%o13)                          a b d + a b c
(%i14) expand( a*b*(c + d) );
(%o14)                          a b d + a b c
```

2.2.4 distrib

The Maxima manual entry is:

> Function: **distrib** (expr)
> Distributes sums over products. It differs from `expand` in that it works at only the top level of an expression, i.e., it doesn't recurse and it is faster than `expand`. It differs from `multthru` in that it expands all sums at that level.

The following table illustrates the differences in behavior of **distrib**, **multthru**, and **expand**,

$expr =$	$(a+b)(c+d)$
distrib(expr)	$bd+ad+bc+ac$
multthru(expr)	$(b+a)d+(b+a)c$
expand(expr)	$bd+ad+bc+ac$
$expr =$	$(b(d+c)+1)(f+a)$
distrib(expr)	$b(d+c)f+f+ab(d+c)+a$
multthru(expr)	$(b(d+c)+1)f+a(b(d+c)+1)$
expand(expr)	$bdf+bcf+f+abd+abc+a$
$expr =$	$1/\left((a+b)(c+d)\right)$
distrib(expr)	$1/((b+a)(d+c))$
multthru(expr)	$1/((b+a)(d+c))$
expand(expr)	$1/(bd+ad+bc+ac)$

Table 1: A Comparison of **distrib**, **multthru**, and **expand**

Here is an example which compares the effects of **distrib**, **multthru**, and **expand** when the expression is $(x+1)((u+v)^2+a(w+z)^2)$.

```
(%i15) expr5 : (x + 1)*( (u + v)^2 + a*(w + z)^2 ) ;
                                              2           2
(%o15)                   (x + 1) (a (z + w)   + (v + u) )
```

```
(%i16) distrib(expr5);
                           2            2           2             2
(%o16)          a x (z + w)  + a (z + w)  + (v + u)  x + (v + u)
/****************************************************************/

(%i17) multthru(expr5);
                                       2           2
(%o17)                a (x + 1) (z + w)  + (v + u)  (x + 1)
/****************************************************************/

(%i18) expand(expr5);
            2      2                            2      2                2
(%o18) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x + 2 u v x + u  x
                                                            2    2            2
                                                       + a w  + v  + 2 u v + u
```

See the Maxima manual entry for **ratexpand** for an example which shows some differences in using **ratexpand** as compared to **expand** with an expression containing negative exponents.

Given an expression which is a ratio of two expressions, how does one expand only the numerator, or only the denominator? One way is to make homemade functions which pick apart the numerator and denominator, apply **expand** to the part desired, and reconstruct the whole expression.

```
(%i1) numexpand(expr) := expand(num(expr))/denom(expr);
                                        expand(num(expr))
(%o1)                numexpand(expr) := -----------------
                                           denom(expr)
(%i2) denomexpand(expr) := num(expr)/expand(denom(expr));
                                             num(expr)
(%o2)              denomexpand(expr) := -------------------
                                        expand(denom(expr))
(%i3) e1 : 1/a + 1/b + 1/c;
                                   1   1   1
(%o3)                              - + - + -
                                   c   b   a
(%i4) e2 : ratsimp(e1);
                                (b + a) c + a b
(%o4)                           ---------------
                                     a b c
(%i5) e3 : numexpand(e2);
                                b c + a c + a b
(%o5)                           ---------------
                                     a b c
(%i6) ratexpand(e2);
                                   1   1   1
(%o6)                              - + - + -
                                   c   b   a
(%i7) ratexpand(e3);
                                   1   1   1
(%o7)                              - + - + -
                                   c   b   a
```

```
(%i8) e4 : a/(b*(c + d));
                                        a
(%o8)                               ---------
                                    b (d + c)
(%i9) e5 : denomexpand(e4);
                                        a
(%o9)                               ---------
                                    b d + b c
```

We have used the function **ratsimp** to express `e1` as the quotient of two polynomials, the denominator being the least common denominator. We will see more examples of **ratsimp** in the next section. The function **ratexpand** (or **expand**) returns the quotient back into a sum of terms.

2.2.5 partfrac

The **partfrac**(exp, var) command performs a complete partial fraction decomposition, expanding the expression exp in partial fractions with respect to the main variable var. This function rewrites a rational expression as a sum of terms with minimal denominators, treating all variables other than `var` as constants. The Maxima manual has the example:

```
(%i1) e1 : 1/(1+x)^2 - 2/(1+x) + 2/(2+x);
                                2       2        1
(%o1)                         ----- - ----- + --------
                              x + 2   x + 1          2
                                              (x + 1)
(%i2) e2 : ratsimp(e1);
                                           x
(%o2)                           - -------------------
                                     3      2
                                    x  + 4 x  + 5 x + 2
(%i3) e3 : partfrac(e2,x);
                                2       2        1
(%o3)                         ----- - ----- + --------
                              x + 2   x + 1          2
                                              (x + 1)
```

A second example of **partfrac**:

```
(%i1) e1 : (5*x^2 - 4*x +16)/( (x^2-x+1)^2 * (x-3) ) ;
                                          2
                                       5 x  - 4 x + 16
(%o1)                           ---------------------
                                          2        2
                                (x - 3) (x  - x + 1)
(%i2) partfrac(e1,x);
                           - x - 2         - 2 x - 3        1
(%o2)                     ---------- + ------------- + -----
                           2                2         2   x - 3
                          x  - x + 1    (x  - x + 1)
```

2.3 Simplifying Expressions

Maxima provides many commands that simplify expressions. The following simplification commands are described in this section.

- **ratsimp** simplifies an expression by combining the rational functions in the expression, then canceling out the greatest common divisor in the numerator and denominator.
- **radcan** simplifies expressions containing radicals, logarithms, and exponentials.
- **rootscontract** converts products of roots into roots of products.
- **logcontract** does things like $a * log(b) \rightarrow log(b^a)$.
- **scsimp** implements the sequential comparative simplifier, which applies given identities to an expression in an effort to obtain a smaller expression.
- **combine** and **rncombine** group the terms in a sum that have the same denominator into a single term.
- **xthru** combines the terms of a sum over a common denominator without expanding them first.
- **map** can apply a given function, such as a simplification command, to each term of a very large expression. This can be useful when applying the function to the entire expression would be inefficient.

This section also introduces the option variable **algebraic**.

2.3.1 ratsimp

The function **ratsimp**(exp) simplifies the expression `exp` and all of its subexpressions, including the arguments to non-rational functions. With additional arguments, **ratsimp**(exp, var1,var2,...,varn) specifies the ordering of each variable `vari` as well. **ratsimp** first combines the sum of rational functions (quotients of polynomials) into one rational function, then cancels out the greatest common divisor of this new function's numerator and denominator.

```
(%i1) e1 : -(x^5 - x)/(x - 1) +x+x^2 +x^3 + x^4 + (a+b+c)^3;
                          5
                     x - x     4    3    2                 3
(%o1)                ------ + x  + x  + x  + x + (c + b + a)
                     x - 1
(%i2) ratsimp(e1);
       3                2       2                2       3        2      2      3
(%o2) c  + (3 b + 3 a) c  + (3 b  + 6 a b + 3 a ) c + b  + 3 a b  + 3 a  b + a
(%i3) ratsimp(e1,a);
       3        2               2        2   2                   2      3    3
(%o3) c  + a (3 c  + 6 b c + 3 b ) + 3 b c  + a  (3 c + 3 b) + 3 b  c + b  + a
```

Maxima simplified the terms containing the variable x to zero, but it returned the expanded result of $(a+b+c)^3$, which is the canonical form of the polynomial.

```
(%i4) e2 : d*(w+a)*x + c*(w+a)*x + b*d + b*c + c*d;
(%o4)               d (w + a) x + c (w + a) x + c d + b d + b c
(%i5) ratsimp(e2);
(%o5)               ((d + c) w + a d + a c) x + (c + b) d + b c
(%i6) ratsimp(e2, c, d);
(%o6)                d ((w + a) x + c + b) + c ((w + a) x + b)
```

Input `(%i5)` produces the default using Maxima's normal ordering of variables. Input `(%i6)` orders d first and c second. With some practice, you will learn to use the various simplification commands to transform an expression into the form you want.

The option variable **algebraic** has the default value **false**. Temporarily setting its value to **true** during the operation of **ratsimp** allows extra simplification to occur.

```
(%i1) algebraic;
(%o1)                              false
(%i2) e1 : 1/(sqrt(x) - 2);
                                     1
(%o2)                           -----------
                                sqrt(x) - 2
(%i3) ratsimp(e1);
                                     1
(%o3)                           -----------
                                sqrt(x) - 2
(%i4) ratsimp(e1),algebraic:true;
                                sqrt(x) + 2
(%o4)                           -----------
                                   x - 4
```

2.3.2 radcan

The Maxima function **radcan**(expr) may be useful for simplifying expressions which contain logs, exponentials, and radicals. This function has the manual entry:

> Function: **radcan** (expr)
> Simplifies expr, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, radcan produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero.
>
> For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.
>
> When `%e_to_numlog` is `true`, `%e^(r*log(expr))` simplifies to `expr^r` if r is a rational number.
>
> When `radexpand` is `false`, certain transformations are inhibited.
> `radcan (sqrt (1-x))` remains `sqrt (1-x)` and is not simplified to `%i sqrt (x-1)`.
> `radcan (sqrt (x^2 - 2*x + 11))` remains `sqrt (x^2 - 2*x + 1)` and is not simplified to $x - 1$.
>
> `example (radcan)` displays some examples.

Here is the result of running **example(radcan)**.

```
(%i5) example(radcan);
(%i6) (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
                                    2                a
                              (log(x  + x) - log(x))
(%o6)                         -----------------------
                                               a/2
                                   log(x + 1)
(%i7) radcan(%)
                                               a/2
(%o7)                              log(x + 1)
(%i8) log(1+2*a^x+a^(2*x))/log(1+a^x)
                                    2 x      x
                               log(a    + 2 a  + 1)
(%o8)                          --------------------
                                         x
                                   log(a  + 1)
(%i9) radcan(%)
(%o9)                                     2
(%i10) (%e^x-1)/(1+%e^(x/2))
                                         x
                                      %e  - 1
(%o10)                               ---------
                                        x/2
                                     %e    + 1
(%i11) radcan(%)
                                        x/2
(%o11)                               %e    - 1
```

2.3.3 radcan and ratsimp with algebraic:true

If **radcan** (perhaps followed by **factor**) does not succeed, you should try **radcan**`(expr)`,**algebraic**`;`, which is, of course equivalent to **radcan**`(expr)`,**algebraic:true**`;`.
Here is an example from the mailing list. We use this as an opportunity to compare the effects of ”turning on” algebraic:true on the behavior of both **radcan** and **ratsimp**.

```
(%i1) ex1:  1/(sqrt(x^2+1)+x)+sqrt(x^2+1)+x;
                               1                2
(%o1)                  ---------------- + sqrt(x  + 1) + x
                             2
                       sqrt(x  + 1) + x
(%i2) ex2 : 1/sqrt(1-x^2);
                                         1
(%o2)                              ------------
                                             2
                                   sqrt(1 - x )
(%i3) algebraic;
(%o3)                                  false
```

```
(%i4) radcan(ex1);
                                    2           2
                          2 x sqrt(x  + 1) + 2 x  + 2
(%o4)                     ---------------------------
                                     2
                               sqrt(x  + 1) + x
(%i5) radcan(ex1),algebraic;
                                        2
(%o5)                           2 sqrt(x  + 1)
(%i6) algebraic;
(%o6)                                false
(%i7) ratsimp(ex2);
                                      1
(%o7)                            ------------
                                           2
                                 sqrt(1 - x )
(%i8) ratsimp(ex2),algebraic;
                                            2
                                  sqrt(1 - x )
(%o8)                           - ------------
                                      2
                                     x  - 1
```

2.3.4 rootscontract

The Maxima manual has the entry:

> Function: **rootscontract** `(expr)`
> Converts products of roots into roots of products. For example, `rootscontract (sqrt(x)*y^(3/2))` yields `sqrt(x*y^3)`.
> When `radexpand` is `true` and `domain` is `real`, `rootscontract` converts `abs` into `sqrt`, e.g., `rootscontract (abs(x)*sqrt(y))` yields `sqrt(x^2*y)`.
> There is an option `rootsconmode` affecting `rootscontract` as follows:

Problem	Value of	Result of applying
	rootsconmode	rootscontract
x^(1/2)*y^(3/2)	false	(x*y^3)^(1/2)
x^(1/2)*y^(1/4)	false	x^(1/2)*y^(1/4)
x^(1/2)*y^(1/4)	true	(x*y^(1/2))^(1/2)
x^(1/2)*y^(1/3)	true	x^(1/2)*y^(1/3)
x^(1/2)*y^(1/4)	all	(x^2*y)^(1/4)
x^(1/2)*y^(1/3)	all	(x^3*y^2)^(1/6)

> When `rootsconmode` is `false`, `rootscontract` contracts only with respect to rational number exponents whose denominators are the same. The key to the `rootsconmode: true` examples is simply that `2` divides into `4` but not into `3`. `rootsconmode: all` involves taking the least common multiple of the denominators of the exponents.
> `rootscontract` uses `ratsimp` in a manner similar to `logcontract`.

By default, **radexpand** is **true**, and **domain** is **real**. Here are examples displayed by the **example** function. Note that the default value of **rootsconmode** is **true**.

```
(%i1) example(rootscontract);
(%i2) rootsconmode:false$
(%i3) rootscontract(x^(1/2)*y^(3/2));
                                             3
(%o3)                            sqrt(x y )
(%i4) rootscontract(x^(1/2)*y^(1/4));
                                             1/4
(%o4)                           sqrt(x) y
(%i5) rootsconmode:true$
(%i6) rootscontract(x^(1/2)*y^(1/4));
(%o6)                          sqrt(x sqrt(y))
(%i7) rootscontract(x^(1/2)*y^(1/3));
                                             1/3
(%o7)                           sqrt(x) y
(%i8) rootsconmode:all$
(%i9) rootscontract(x^(1/2)*y^(1/4));
                                        2   1/4
(%o9)                             (x   y)
(%i10) rootscontract(x^(1/2)*y^(1/3));
                                       3   2 1/6
(%o10)                           (x   y )
(%i11) rootsconmode:false$
(%i12) rootscontract(sqrt(sqrt(x)+sqrt(1+x))*sqrt(sqrt(1+x)-sqrt(x)));
(%o12)                                1
(%i13) rootsconmode:true$
(%i14) rootscontract(sqrt(5+sqrt(5))-5^(1/4)*sqrt(1+sqrt(5)));
(%o14)                                0
```

2.3.5 logcontract

The manual has the description

> Function: **logcontract** `(expr)`
> Recursively scans the expression `expr`, transforming subexpressions of the form
> `a1*log(b1) + a2*log(b2) + c` into `log(ratsimp(b1^a1 * b2^a2)) + c`
>
> ```
> (%i1) 2*(a*log(x) + 2*a*log(y))$
> (%i2) logcontract(%);
>                                      2   4
> (%o2)                          a log(x   y )
> ```
>
> If you do `declare(n,integer);` then `logcontract(2*a*n*log(x));` gives `a*log(x^(2*n))`.
> The coefficients that "contract" in this manner are those such as the `2` and the `n` here which satisfy `featurep(coeff,inte`
> The user can control which coefficients are contracted by setting the option `logconcoeffp` to the name of a predicate function of one argument.
>
> E.g. if you like to generate SQRTs, you can do `logconcoeffp:'logconfun$`
> `logconfun(m):=featurep(m,integer)` or `ratnump(m)$`.
>
> Then `logcontract(1/2*log(x));` will give `log(sqrt(x))`.

The manual description of the global option variable **logconcoeffp** is:

> Option variable: **logconcoeffp**
> Default value: **false**
> Controls which coefficients are contracted when using `logcontract`. It may be set to the name of a predicate function of one argument.
> E.g. if you like to generate SQRTs, you can do
> `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer)` or `ratnump(m)$`.
> Then `logcontract(1/2*log(x));` will give `log(sqrt(x))`.

The **example** function gives

```
(%i1) example(logcontract)$
(%i2) 2*(2*a*log(y)+a*log(x));
(%o2)                     2 (2 a log(y) + a log(x))
(%i3) logcontract(%);
                                         2  4
(%o3)                           a log(x   y )
(%i4) logcontract(log(sqrt(1+x)-sqrt(x))+log(sqrt(x)+sqrt(1+x)));
(%o4)                                  0
```

Here we explore some of the options available

```
(%i1) logconcoeffp;
(%o1)                                false
(%i2) logcontract(2*a*n*log(x) );
                                         2
(%o2)                            a n log(x )
(%i3) featurep(n,integer);
(%o3)                                false
(%i4) declare(n,integer)$
(%i5) featurep(n,integer);
(%o5)                                true
(%i6) logcontract(2*a*n*log(x) );
                                        2 n
(%o6)                            a log(x   )
(%i7) logconfun;
(%o7)                             logconfun
(%i8) logconcoeffp: 'logconfun$
(%i9) logconfun(m) := featurep(m,integer) or ratnump(m);
(%o9)         logconfun(m) := featurep(m, integer) or ratnump(m)
(%i10) featurep(1/2,integer);
(%o10)                               false
(%i11) ratnump(1/2);
(%o11)                               true
(%i12) logconfun(1/2);
(%o12)                               true
(%i13) logcontract(2*a*n*log(x) );
                                        2 n
(%o13)                           a log(x   )
(%i14) logcontract(1/2*log(x) );
(%o14)                          log(sqrt(x))
```

```
(%i15) 1/2*log(x);
                                  log(x)
(%o15)                            ------
                                    2
(%i16) logcontract(1/n*log(x));
                                  log(x)
(%o16)                            ------
                                    n
(%i17) ratnump(n);
(%o17)                             false
```

We see that even though the symbol `n` was declared to be an integer, `n` does not pass the **ratnumbp** test, and hence **logcontract** `(1/n*log(x) )` is not changed. The function **ratnump** has the manual entry

> Function: **ratnump** `(expr)`
> Returns `true` if `expr` is a literal integer or ratio of literal integers, otherwise `false`.

The reason **ratnump** `(n)` returned **false** is that `n` is a declared symbolic integer, not a literal integer like `2`.

2.3.6 scsimp

The manual entry is

> Function: **scsimp** `(expr, rule_1, ..., rule_n)`
> Sequential Comparative Simplification (method due to Stoute). `scsimp` attempts to simplify expr according to the rules `rule_1, ..., rule_n`. If a smaller expression is obtained, the process repeats. Otherwise after all simplifications are tried, it returns the original answer.
>
> `example (scsimp)` displays some examples.

Here is the result of running **example(scsimp)**:

```
(%i2) example(scsimp)$
(%i3) exp:-k^2*l^2*m^2*n^2-k^2*l^2*n^2+k^2*m^2*n^2+k^2*n^2
                    2  2  2  2    2  2  2    2  2  2    2  2
(%o3)            - k  l  m  n  + k  m  n  - k  l  n  + k  n
(%i4) eq1:l^2+k^2 = 1
                                 2    2
(%o4)                           l  + k  = 1
(%i5) eq2:n^2-m^2 = 1
                                 2    2
(%o5)                           n  - m  = 1
(%i6) scsimp(exp,eq1,eq2)
                                     4  4
(%o6)                               k  n
(%i7) exq:(-k2*k3-k1*k2+k1*k4)/k3^2
                            k1 k4 - k2 k3 - k1 k2
(%o7)                       ---------------------
                                       2
                                     k3
(%i8) eq3:k1*k4-k2*k3 = 0
(%o8)                         k1 k4 - k2 k3 = 0
(%i9) eq4:k3*k4+k1*k2 = 0
(%o9)                         k3 k4 + k1 k2 = 0
```

```
(%i10) scsimp(exq,eq3,eq4)
                                          k4
(%o10)                                    --
                                          k3
```

2.3.7 combine, rncombine

The manual entry for **combine** is:

> Function: **combine** (expr)
> Simplifies the sum expr by combining terms with the same denominator into a single term.

The manual entry for **rncombine** is:

> Function: **rncombine** (expr)
> Transforms expr by combining all terms of expr that have identical denominators or denominators that differ from each other by numerical factors only. This is slightly different from the behavior of `combine`, which collects terms that have identical denominators.
> Setting `pfeformat: true` and using `combine` yields results similar to those that can be obtained with `rncombine`, but `rncombine` takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possibility of recognizing some cancellations.

In order to use **rncombine**, one must load a package with **load(rncomb)**. The function **rncombine** is presently broken, is not reliable, and this a known bug in Maxima.

The manual entry on **pfeformat** is:

> Option variable: **pfeformat**
> Default value: **false**
> When `pfeformat` is `true`, a ratio of integers is displayed with the solidus (forward slash) character, and an integer denominator `n` is displayed as a leading multiplicative term `1/n`.

Here is the manual example of **pfeformat**:

```
(%i1) pfeformat;
(%o1)                                    false
(%i2) 2^16/7^3;
                                         65536
(%o2)                                    -----
                                          343
(%i3) (a+b)/8;
                                         b + a
(%o3)                                    -----
                                           8
(%i4) pfeformat:true$
(%i5) 2^16/7^3;
(%o5)                                  65536/343
(%i6) (a+b)/8;
(%o6)                                 1/8 (b + a)
```

Here is the single example Maxima has for **combine**:

```
(%i7) example(combine);
(%i8) combine(b/y+a/y+b/x+a/x)
                                 b + a   b + a
(%o8)                            ----- + -----
                                   y       x
```

Here is a comparison of **combine** and **rncombine**:

```
(%i1) load(rncomb);
(%o1)
 C:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/share/simplification/rncomb.mac
(%i2) e1 : x/a + y/a + z/(2*a);
                                  z     y    x
(%o2)                            --- + - + -
                                 2 a    a    a
(%i3) combine(e1);
                                  z     y + x
(%o3)                            --- + -----
                                 2 a      a
(%i4) rncombine(e1);
                                z + 2 y + 2 x
(%o4)                           -------------
                                     2 a
(%i5) pfeformat:true$
(%i6) combine(e1);
                                1/2 z + y + x
(%o6)                           -------------
                                      a
(%i7) rncombine(1/x + 1/y);
                                    1
(%o7)                               - + 1
                                    y
(%i8) rncombine(x + y/x);
                                    y
(%o8)                               - + x
                                    x
(%i9) rncombine(y + x/y);
(%o9)                               y + x
```

Ouputs 7 and 9 are not correct.

There is a demo file: `5.14.0/share/simplification/rncomb.dem` which is run, as usual, with the Maxima **demo** function, with the syntax: **demo**("rncomb"). This demo file has more examples:

```
(%i8) demo ("rncomb");
batching #pC:/PROGRA~1/MAXIMA~3.0/share/maxima/5.14.0/
                                share/simplification/rncomb.dem

 At the _ prompt, type ';' followed by enter to get next demo
(%i9)                           load(rncomb)
_;
                                     z         x
(%i10)                    exp1 : ----- + ---------
                                 y + x   2 (y + x)
                                z         x
(%o10)                        ----- + ---------
                              y + x   2 (y + x)
;
```

```
(%i11)                             combine(exp1)
                                    z         x
(%o11)                           ----- + ---------
                                 y + x   2 (y + x)

_
(%i12)                             rncombine(%)
                                      2 z + x
(%o12)                              ---------
                                    2 (y + x)

_;
                                      d   c   b   a
(%i13)                         exp2 : - + - + - + -
                                      3   3   2   2
                                   d   c   b   a
(%o13)                             - + - + - + -
                                   3   3   2   2

_;
(%i14)                             combine(exp2)
                               2 d + 2 c + 3 (b + a)
(%o14)                         ---------------------
                                         6

_;
(%i15)                           rncombine(exp2)
                               2 d + 2 c + 3 b + 3 a
(%o15)                         ---------------------
                                         6

_;
(%i16)
```

The function **rncombine** should be used with caution and cross checks.

2.3.8 xthru

The manual entry for **xthru** is:

> Function: **xthru** (expr)
> Combines all terms of expr (which should be a sum) over a common denominator without expanding products and exponentiated sums as `ratsimp` does. `xthru` cancels common factors in the numerator and denominator of rational expressions but only if the factors are explicit. Sometimes it is better to use `xthru` before ratsimping an expression in order to cause explicit factors of the gcd of the numerator and denominator to be canceled thus simplifying the expression to be ratsimped.

Here is **example(xthru)** which is also in the manual:

```
(%i1) example(xthru);
(%i2) (-x)/(y+x)^20+1/(y+x)^19+((2+x)^20-2*y)/(y+x)^20
                                                 20
                         1        (x + 2)    - 2 y        x
(%o2)                --------- + --------------- - ---------
                            19               20           20
                     (y + x)          (y + x)       (y + x)
```

```
(%i3) xthru(%)
                                         20
                                  (x + 2)    - y
(%o3)                             -------------
                                           20
                                    (y + x)
```

Here is another example of using **xthru** after using **multthru** to distribute a product (a denominator times a numerator) over a sum (the numerator being the sum):

```
(%i1) e1 : ((b+a)^10*(s-t)^2+2*a*b*(s-t)+a^2*b^2*(s-t))/a/b/(s-t)^4;
                    10        2    2  2
              (b + a)   (s - t)  + a  b  (s - t) + 2 a b (s - t)
(%o1)         ------------------------------------------------
                                                  4
                                       a b (s - t)
(%i2) me1 : multthru(e1);
                             10
                      (b + a)         a b         2
(%o2)                ------------ + -------- + --------
                                2          3          3
                     a b (s - t)    (s - t)    (s - t)
(%i3) e2 : xthru(me1);
                             10
                      (b + a)   (s - t) + a b (a b + 2)
(%o3)                 ---------------------------------
                                                3
                                     a b (s - t)
```

2.3.9 Using map with Simplification Functions

We discussed the use of **map** with lists in chapter 1, Getting Started. Here we want to use **map** for applying some simplifying function to an expression. Let f be some operator or function, at present undefined. We compare the action of f when applied to a list or a sum of terms.

```
(%i1) map(f,[a,b,c]);
(%o1)                          [f(a), f(b), f(c)]
(%i2) map(f,a+b+c);
(%o2)                          f(c) + f(b) + f(a)
```

Here we use **map** to apply the simplification function **ratsimp** to each term of a sum, and we compare that result with simply applying the simplification function to the whole expression.

```
(%i1) e1 : x/(x^2+x) + (y^2+y)/y ;
                                     2
                                    y  + y     x
(%o1)                               ------ + ------
                                      y       2
                                             x  + x
(%i2) ratsimp(e1);
                                  (x + 1) y + x + 2
(%o2)                            -----------------
                                        x + 1
```

```
(%i3) map(ratsimp,e1);
                                                 1
(%o3)                                   y + ----- + 1
                                              x + 1
```

Other more specialized forms of the **map** function are the Maxima functions **fullmap**, **scanmap**, and **outermap**.

2.4 Factoring Expressions

Not covered (yet) in this section are examples of the **facexp** package which contains specialized factoring functions like **facsum**. There is a demo you can run which has a long list of elaborate examples:
type " **demo**(facexp)$ ".

2.4.1 factor and gfactor

Here is the beginning of the Maxima manual entry for **factor**.

> Function: **factor** (expr)
> Factors the expression expr, containing any number of variables or functions, into factors irreducible over the integers.
> `factor (expr, p)` factors expr over the field of integers with an element adjoined whose minimum polynomial is p.
> `factor` uses `ifactors` function for factoring integers.
> `factorflag` if false suppresses the factoring of integer factors of rational expressions
> `dontfactor` may be set to a list of variables with respect to which factoring is not to occur. (It is initially empty).
> Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the `dontfactor` list.

And here is the manual entry for **gfactor**.

> Function: **gfactor** (expr)
> Factors the polynomial expr over the Gaussian integers (that is, the integers with the imaginary unit `%i` adjoined). This is like `factor (expr, a^2+1)` where `a` is `%i`.
> Example:

```
(%i1) gfactor (x^4 - 1);
(%o1)            (x - 1) (x + 1) (x - %i) (x + %i)
```

Here is an example of using the Maxima function **factor**. First we **expand** the expression $(a + b\,x)^5$:

```
(%i1) expr1 : expand( (1+x^2)^5 );
                         10      8       6       4      2
(%o1)                   x   + 5 x  + 10 x  + 10 x  + 5 x  + 1
```

The **factor** function returns the expanded expression to its original form.

```
(%i2) factor( expr1 );
                                          2     5
(%o2)                                   (x  + 1)
```

We can get further factorization of `expr1` by using **gfactor**:

```
(%i3) gfactor( expr1 );
                                            5          5
(%o3)                             (x - %i)   (x + %i)
```

The direct use of **factor** does not always get the complete factorization expected. Here is an example involving a high order polynomial:

```
(%i1) e1 : x^28 + 1;
                                         28
(%o1)                                   x   + 1
(%i2) factor(e1);
                  4         24     20     16     12     8     4
(%o2)           (x  + 1) (x   - x   + x   - x   + x  - x  + 1)
```

To recover the desired x^{28} we subtract 1 and use **ratsimp**:

```
(%i3) e2 : % - 1;
                4         24     20     16     12     8     4
(%o3)         (x  + 1) (x   - x   + x   - x   + x  - x  + 1) - 1
(%i4) ratsimp(e2);
                                           28
(%o4)                                     x
```

Here is an example which illustrates the limitations of **factor**

```
(%i7) e3 : a*x^2 - 2*a$
(%i8) factor(e3);
                                          2
(%o8)                                a (x  - 2)
```

To get `e3` in completely factored form, we need to use the expanded form **factor**`(expr, p)`, where `p` is a polynomial whose solutions will allow further factorization of `expr` involving numbers `a` which are not integers. We see that those will be the numbers $\sqrt{2}$ and $-\sqrt{2}$, which are the solutions of the equation $q^2 - 2 = 0$.

```
(%i9) factor(e3,q^2 - 2);
(%o9)                             a (x - q) (x + q)
(%i10) solns : solve(q^2 - 2);
(%o10)                       [q = - sqrt(2), q = sqrt(2)]
(%i11) subst( solns[2],%o9 );
(%o11)                       a (x - sqrt(2)) (x + sqrt(2))
```

We can also use **map** to separately factor terms of a sum.

```
(%i5) e2 : (x^2 - 2*x +1)/a + (x^2 -7*x +12)/b;
                          2              2
                         x  - 2 x + 1   x  - 7 x + 12
(%o5)                    ------------ + -------------
                              a               b
(%i6) factor(e2);
                       2      2
                    b x  + a x  - 2 b x - 7 a x + b + 12 a
(%o6)               --------------------------------------
                                     a b
```

```
(%i7) map(factor,e2);
                                        2
                               (x - 1)     (x - 4) (x - 3)
(%o7)                          -------- + ---------------
                                  a              b
```

The Maxima manual **factor** entry has the following involved example which eventually produces a nicer looking partial fraction expansion:

```
(%i1) e1 : (2 + x)/(3 + x)/(b + x)/(c + x)^2;
                                      x + 2
(%o1)                          ------------------------
                                                      2
                               (x + 3) (x + b) (x + c)
(%i2) e2 : ratsimp(e1);
                  4                  3     2                       2
(%o2) (x + 2)/(x  + (2 c + b + 3) x  + (c  + (2 b + 6) c + 3 b) x
                                                     2                  2
                                       + ((b + 3) c  + 6 b c) x + 3 b c )
(%i3) e3 : partfrac (e2, x);
         2                   4                 3     2                2
(%o3) - (c  - 4 c - b + 6)/((c  + (- 2 b - 6) c  + (b  + 12 b + 9) c
        2                2                            c - 2
 + (- 6 b  - 18 b) c + 9 b ) (x + c)) - --------------------------------
                                             2                         2
                                           (c  + (- b - 3) c + 3 b) (x + c)
                            b - 2
 + -------------------------------------------------
               2             2       3      2
   ((b - 3) c  + (6 b - 2 b ) c + b  - 3 b ) (x + b)
                            1
 - ----------------------------------------------
               2
   ((b - 3) c  + (18 - 6 b) c + 9 b - 27) (x + 3)
(%i4) e4 : map(factor,e3);
               2
             c  - 4 c - b + 6                 c - 2
(%o4) - ------------------------- - ------------------------
                2        2                                2
         (c - 3)  (c - b)  (x + c)    (c - 3) (c - b) (x + c)
                                     b - 2                        1
                             + ----------------------- - ------------------------
                                             2                          2
                               (b - 3) (c - b)  (x + b)   (b - 3) (c - 3)  (x + 3)
```

The manual has the following example of the use of the **dontfactor** list, which is assigned the value `[x]` locally inside the **block** structure.

```
(%i1) dontfactor;
(%o1)                                  []
(%i2) -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2;
                         2  2         2    2    2
(%o2)                   x  y  + 2 x y  + y  - x  - 2 x - 1
```

```
(%i3) block ([dontfactor: [x]], factor (%/36/(1 + 2*y + y^2)));
                                2
                              (x  + 2 x + 1) (y - 1)
(%o3)                         ----------------------
                                   36 (y + 1)
(%i4) dontfactor;
(%o4)                                   []
```

2.4.2 collectterms

Syntax: **collectterms** `(expr, arg1, arg2, ...)`

This Maxima function is best explained with an example. Consider the expression:

```
(%i2) ex1 : expand( a1*(b + c/2)^2 + a2*(d + e/3)^3 );
                3        2                          2
          a2 e    a2 d e        2          3   a1 c                2
(%o2)     ----- + ------- + a2 d  e + a2 d  + ----- + a1 b c + a1 b
           27        3                           4
```

How can we return this expanded expression to the original form? We first use **collectterms** `(expr,arg1,arg2,...)`:

```
(%i3) ex2 : collectterms(ex1,a1,a2);
                       3      2                     2
                      e    d e     2      3        c            2
(%o3)            a2 (-- + ---- + d  e + d ) + a1 (-- + b c + b )
                      27    3                      4
```

We then **map** the function **factor** on to ex2:

```
(%i4) map(factor,ex2);
                                         3               2
                          a2 (e + 3 d)     a1 (c + 2 b)
(%o4)                     ------------- + -------------
                               27                4
```

Maxima's core simplification rules prevent us from getting the $3^3 = 27$ into the numerator of the first term, and also from getting the $2^2 = 4$ into the numerator of the second term.

2.4.3 factorsum

The manual description of **factorsum** is:

> Function: **factorsum** (expr)
> Tries to group terms in factors of expr which are sums into groups of terms such that their sum is factorable. `factorsum` can recover the result of `expand ((x + y)^2 + (z + w)^2)` but it can't recover `expand ((x + 1)^2 + (x + y)^2)` because the terms have variables in common.

Here is the example presented in the manual:

```
(%i1) e1: (x+1)*( (u+v)^2 + a*(w+z)^2 )$
(%i2) e2: expand(e1);
           2       2                             2       2                 2
(%o2) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x + 2 u v x + u  x
                                                              2     2            2
                                                       + a w  + v  + 2 u v + u
(%i3) factor(e2);
                          2                 2     2            2
(%o3)         (x + 1) (a z  + 2 a w z + a w  + v  + 2 u v + u )
(%i4) factorsum(e2);
                                        2           2
(%o4)                   (x + 1) (a (z + w)  + (v + u) )
```

2.4.4 factorout

Here is the brief manual description:

> Function: **factorout** `(expr, x_1, x_2, ...)`
> Rearranges the sum `expr` into a sum of terms of the form `f (x_1, x_2, ...)*g` where `g` is a product of expressions not containing any `x_i` and `f` is factored.

Here is a simple example which compares various factoring tools.

```
(%i1) e1 : a*z + b*z + 1$
(%i2) factor(e1);
(%o2)                          b z + a z + 1
(%i3) factorsum(e1);
(%o3)                          (b + a) z + 1
(%i4) factorout(e1,z);
(%o4)                          b z + a z + 1
(%i5) factorout(e1,a,b);
(%o5)                          (b + a) z + 1
```

2.5 Making Substitutions

Maxima allows you to perform many kinds of substitutions, such as substituting one expression for another in a third.

This section presents the following commands:

- **ev** evaluates a given expression in the specified environment.
- **subst** makes replacements by substitution, with some restrictions on the expression being replaced.
- **ratsubst** is similar to **subst**, but without the restrictions on the expression being replaced.

2.5.1 Using ev for Substitutions

The Maxima manual has a long description of the **ev** function, which begins:

> Function: **ev** `(expr, arg_1, ..., arg_n)`
> Evaluates the expression `expr` in the environment specified by the arguments `arg_1, ..., arg_n`.
> The arguments are switches (Boolean flags), assignments, equations, and functions.
> `ev` returns the result (another expression) of the evaluation.
> The evaluation is carried out in steps, as follows.
> (continues...)

A simple example of using **ev** for making a substitution follows. We first define a simple function of `z`.

```
(%i1) e1 : z*%e^z;
                                          z
(%o1)                                  z %e
(%i2) ev( e1, z = x^2 );
                                            2
                                        2  x
(%o2)                                  x  %e
```

The form shown works in both interactive work and in code loaded in as a package of Maxima functions. A simpler way to get the same "evaluation" process when working iteractively, is as follows:

```
(%i3) e1, z = x^2;
                                            2
                                        2  x
(%o3)                                  x  %e
```

The symbol `e1` is still bound to the original quantity defined in `(%i1)`.

```
(%i4) e1;
                                          z
(%o4)                                  z %e
```

A possible use of such a evaluation would be to define some related function or expression. We first define `f1` to be an expression which depends on `x`. We then define `f2(x)` to be an actual "function" of `x` in the usual sense that if `x` is replaced by, say, `y`, we get the same functional form, but with `x` replaced by `y`. Note the use of two single quotes (used to force extra evaluation) in the definition of `f2(x)`.

```
(%i5) f1 : e1,z = x^2$
(%i6) f1;
                                            2
                                        2  x
(%o6)                                  x  %e
(%i7) f2(x) := ''f1;
                                                2
                                            2  x
(%o7)                            f2(x) := x  %e
(%i8) f2(y);
                                            2
                                        2  y
(%o8)                                  y  %e
```

Even if we had first defined an expression, say `f3`, directly as

```
(%i9) f3 : x^2*%e^(x^2);
                                                 2
                                             2  x
(%o9)                                       x  %e
```

we would still need the double quote to get a proper function definition for `f4(x)`:

```
(%i10) f4(x) := f3;
(%o10)                              f4(x) := f3
(%i11) f4(x) := ''f3;
                                                     2
                                                 2  x
(%o11)                           f4(x) := x  %e
(%i12) f4(y);
                                                 2
                                             2  y
(%o12)                                      y  %e
```

2.5.2 Using subst

To make replacements by substitution, you can use the command **subst** `(a, b, c)` where `a` is the expression you want to substitute for expression `b` in expression `c`.
The alternative form of the **subst** syntax is `subst(b = a,c)` which has the same effect. The argument `b` must be an atom (ie., a number, a string, or a symbol) or a complete subexpression of `c`. When `b` does not have these characteristics, use **ratsubst** `(a, b, c)` instead.

The Maxima manual describes the function **atom** which will return **true** or **false** depending on whether or not the argument supplied fits the technical definition of an atom. Here is the Maxima manual description:

> Function: **atom** `(expr)`
> Returns `true` if `expr` is atomic (i.e. a number, name or string) else `false`. Thus `atom(5)` is `true` while `atom(a[1])` and `atom(sin(x))` are `false` (asuming `a[1]` and `x` are unbound).

Here we use **subst** to substitute x^2 for z in `e1` (the actual value of `e1` does not change). We also check the action of **atom**.

```
(%i1) e1 : z*%e^z;
                                                z
(%o1)                                       z %e
(%i2) [subst(x^2,z,e1), subst(z = x^2,e1)];
                                            2         2
                                       2   x     2   x
(%o2)                             [x  %e   , x  %e   ]
(%i3) e1;
                                                z
(%o3)                                       z %e
(%i4) map(atom, [z, 5, sin(z)] );
(%o4)                             [true, true, false]
```

We see that the binding of `e1` has not been altered.

Here we use the **subst** function to change the binding of the symbol `e1`:

```
(%i5) e1 : subst( a^2*b^3, z, e1 );
                                                 2  3
                                         2  3   a  b
(%o5)                                   a  b  %e
(%i6) e1;
                                                 2  3
                                         2  3   a  b
(%o6)                                   a  b  %e
```

2.5.3 Using ratsubst

We now define `e2` to be an expression which contains four atoms: `a`, `b`, `c`, and `d`, and two complete subexpressions: `d` and `c + b + a`.

```
(%i7) e2 : (a + b + c)/d;
                                          c + b + a
(%o7)                                     ---------
                                              d
```

The **subst** function cannot replace `a + b` because it is not a complete subexpression of `e1`.

```
(%i8) subst(d, a + b, e2 );
                                          c + b + a
(%o8)                                     ---------
                                              d
```

However, **ratsubst** can be used instead.

```
(%i9) ratsubst(d, a + b, e2 );
                                            d + c
(%o9)                                       -----
                                              d
```

The Maxima manual description of **ratsubst** is:

> Function: **ratsubst** `(a, b, c)`
> Substitutes `a` for `b` in `c` and returns the resulting expression.
> `b` may be a sum, product, power, etc.
> `ratsubst` knows something of the meaning of expressions whereas `subst` does a purely syntactic substitution.
> Thus `subst (a, x + y, x + y + z)` returns `x + y + z` whereas `ratsubst` returns `z + a`.
> When `radsubstflag` is `true`, `ratsubst` makes substitutions for radicals in expressions which don't explicitly contain them.

The manual examples for **ratsubst** are:

```
(%i1) e1 : cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1$
(%i2) ratsubst( 1 - sin(x)^2, cos(x)^2, e1 );
                    4           2                      2
(%o2)            sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
(%i3) ratsubst( 1 - cos(x)^2, sin(x)^2, sin(x)^4 );
                                    4           2
(%o3)                            cos (x) - 2 cos (x) + 1
```

```
(%i4) radsubstflag;
(%o4)                               false
(%i5) ratsubst (u, sqrt(x), x);
(%o5)                                 x
(%i6) radsubstflag : true$
(%i7) ratsubst (u, sqrt(x), x);
                                       2
(%o7)                                 u
```