An expression formatter for MACSYMA

Bruce R. Miller*

July 6, 1995

Abstract

A package for formatting algebraic expressions in MACSYMA[1] is described. It provides facilities for user-directed hierarchical structuring of expressions, as well as for directing simplifications to selected subexpressions. It emphasizes a semantic rather than syntactic description of the desired form. The package also provides utilities for obtaining efficiently the coefficients of polynomials, trigonometric sums and power series. Similar capabilities would be useful in other computer algebra systems.
Keywords: algebraic structure, computer algebra, MACSYMA, simplification, software, transformation.

1 Introduction

In a general purpose Computer Algebra System (CAS), any particular mathematical expression can take on a variety of forms: expanded form, factored form or anything in between. Each form may have advantages; a given form may be more compact than another, or allow clear expression of certain algorithms. Or it may simply be more informative, particularly if it has physical significance.

A CAS contains many tools for transforming expressions. However, most are like MACSYMA's[1] `factor` and `expand`, operating only on the entire expression or its top level. At the other extreme are operations like `substpart` which extract a specific part of an expression, then transform and replace it. Unfortunately, the means of specifying the piece of interest is purely syntactic, requiring the user to keep close watch on the form of the arguments to avoid error.

The package described here gives users of MACSYMA more control over the structure of expressions, and it does so using a more semantic, almost algebraic, language describing the desired structure. It also provides a semantic means of addressing parts of an expression for particular simplifications. For example, to

*Email:miller@cam.nist.gov
[1]The use of commercial products or their names implies no endorsement by NIST, the Dept. of Commerce or the U.S. Government.

rearrange an expression into a series in `eps` through order 5, whose terms will be polynomials in the `x` and `y`, whose coefficients, in turn, will be trigonometric sums in `l` and `g` with factored coefficients one uses the command:

```
format(foo, %series(eps, 5), %poly(x, y), %trig(l, g), %factor);
```

— more easily invoked than described.

An expression 'formatting' tool for a general purpose system was reported in [4] for Scratchpad, predating the user-specified canonical representations of AXIOM and the author's system MAO. Jeffrey Golden[3] proposed a similar system for MACSYMA— although never implemented, his design provided inspiration and a good naming convention for the package described here. A different addressing scheme for directing simplifications in Mathematica was reported in [5]. Thus the general idea behind these tools is not new, yet the tools themselves are not commonly available in most CAS. Further, we feel that our synthesis is unique. And while our syntax may be a bit baroque, including many keywords, we have found the package to be an indispensable tool in practice.

Two modules are documented in this report. The principal tool, `format`, is described in Section 2. It uses procedures in `coeflist` which obtain coefficients of polynomials, trigonometric sums and power series. The latter module can be useful alone; it is documented in section 3. An appendix discusses implementation issues. The LISP source code may be obtained from the author.

2 FORMAT; Formatting expressions

`format`(*expr*, $template_1$, ...) *Function*

Recursively arranges *expr* according to the *chain* of templates, $template_i$.

Each *template* indicates the desired form for an expression; either the expected form or that into which it will be transformed. At the same time, the indicated form implies a set of *pieces*; the next template in the chain applies to those pieces. For example, `%poly(x)` specifies the transformation into a polynomial in x, with the pieces being the coefficients. The passive `%frac` treats the expression as a fraction; the pieces are the numerator and denominator.

Whereas the next template formats all pieces of the previous layer, positional *subtemplates* may be used to specify formats for each piece individually. This is most useful when the pieces have unique roles and need to be treated differently, such as a fraction's numerator and denominator.

The full syntax of a template is

$$keyword(parameter, \ldots)[subtemplate, \ldots].$$

The recognized keywords are described in Table 1. The parameters (if not needed) and subtemplates (along with parentheses and brackets) are optional.

In addition to the keyword templates, arithmetic patterns are recognized. This is an expression involving addition, multiplication and exponentiation containing a single instance of a keyword template. In effect, the system 'solves' the expression to be formatted for the corresponding part, formats it accordingly and reinserts it. Eg. , `format(X,a+%factor)` is equivalent to `a+factor(X-a)`.

Any other template is assumed to be a function to be applied to the expression; the result is then formatted according to the rest of the template chain.

Examples

```
(c1) format((a+b*x)*(c-x)^2,%poly(x),%factor);
        3                2                        2
(d1) b x  - (2 b c - a) x  + c (b c - 2 a) x + a c

(c2) format((1+2*e*(q+r*cos(g))^2)^4,%series(e,2),%trig(g),%f);
                2      2                2
(d2) 1 + e (4 (r  + 2 q ) + 4 cos(2 g) r  + 16 cos(g) q r)

    2        4       2  2      4                4
 + e  (3 (3 r  + 24 q  r  + 8 q ) + 3 cos(4 g) r

                    3                     2      2
   + 24 cos(3 g) q r  + 24 cos(g) q r (3 r  + 4 q )

                  2   2      2
   + 12 cos(2 g) r  (r  + 6 q )) + ...

(c3) format((1+2*a+a^2)*b + a*(1+2*b+b^2),%sum,%product,%factor);
              2          2
(d3) a (b + 1)  + (a + 1)  b

(c4) format(expand((1+x^4)*y^2+(1+x^8)*y^4),%p(y),%f(a^2-2));
       4      2        4      2       4
(d4) (x  - a x  + 1) (x  + a x  + 1) y

       2              2             2
   + (x  - a x + 1) (x  + a x + 1) y

(c5) format(expand((a+x)^3-a^3),%f-a^3);
             3    3
(d5) (x + a)  - a
```

`format` can also be used to focus simplifications on manageable pieces of large expressions.

Class	*Template (w/abbrev.)*	*Coercion to*	*Pieces & ordering*
Algebraic[a]	`%poly(`$x_1, \ldots$`)`, `%p`	Polynomial in x_i	coefficients (ascending exps.)
	`%series(`ϵ, n`)`, `%s`	Series in ϵ through order n	"
	`%Taylor(`ϵ, n`)`	Taylor in ϵ through order n	"
	`%monicpoly(`$x_1, \ldots$`)`, `%mp`	Monic polynomial in x_i	leading coef. then coefficients
	`%trig(`$x_1, \ldots$`)`, `%t`	Trigonometric sum in x_i	sin coefs (ascending mult), then cos
	`%coeff(`v, n`)`	Polynomial in v	Coefficient of v^n and remainder.
Sums	`%sum`	*passive*[b]	terms (`inpart` order[c])
	`%partfrac(`x`)`, `%pf`	Partial fraction decomp. in x	"
Products	`%product`, `%prod`	*passive*	factors (`inpart` order)
	`%factor`, `%f`	factored form	"
	`%factor(`*minpoly*`)`, `%f`	factored with element adjoined	"
	`%sqfr`, `%sf`	square-free factored form	"
Fractions	`%frac`	*passive*	numerator & denominator
	`%ratsimp`, `%r`	rationally simplified	"
Complex	`%rectform`, `%g`	Gaussian form	real & imaginary parts
	`%polarform`	Polar form	magnitude & phase
'Bags'[d]	`%equation`, `%eq`	equation	left-hand side & right-hand side
	`%relation(`r`)`, `%rel`	Relation; $r \in \{=, >, \geq, <, \leq, \neq\}$	"
	`%list`	list	elements
	`%matrix`	matrix	rows (use `%list` to target elements)
General	`%expression`, `%expr`	*passive*	the operands (`inpart` order)
	`%preformat(`$T_1, \ldots$`)`	Format according to chain T_i	the result, (**not!** the parts)

[a] See section 3 for the interpretation of polynomials, trigonometric sums and series used by this package.

[b] A passive keyword does not transform the expression but treats it as a sum, fraction or whatever. Note that `%sum` (`%product`) treats an expression which does not have "+" (resp. "*") as its main operator as a single term (factor).

[c] The order of the pieces corresponds to the internal ordering; subtemplate usage may be awkward.

[d] See the documentation of `coerce_bag` for a description of the coercions used.

Table 1: Template keywords.

Table 1: Template keywords continued.

Class	*Template (w/abbrev.)*	*Function*
Targeting[a]	`%arg(n)`	Formats the n-th argument
	`%lhs(`r`)`	Formats the lefthand side of an equation or relation (default `"="`).
	`%rhs(`r`)`	Formats the righthand side.
	`%element(i,..)`,`%el`	Formats an element of a matrix.
	`%num`,`%denom`	Formats the numerator or denominator of a fraction.
	`%match(`P`)`	Formats all subexpressions for which $P(expr)$ returns True.
Control	`%if(`$P_1,\ldots$`)[`$T_1\ldots,T_{n+1}$`]`	Find first $P_i(expr)$ →`True`, then format expr using T_i, else T_{n+1}.
Subtemplate Aids	`%noop`	Does nothing; used to fill a subtemplate slot.
	`[`$T_1,T_2,\ldots$`]`	Creates a template chain where an individual template was expected.
	`%ditto(`T`)`	Repeats the template so that it applies to following pieces.
Convenience	`%subst(`*eqns*,...`)`	Substitutes *eqns* into expression; result is formatted at next layer
	`%ratsubst(`*eqns*,...`)`	`lratsubst`'s *eqns* into expression; result is formatted at next layer

[a]Targetting templates are basically shorthand equivalents of structuring templates using subtemplates.

```
(c6) foo:X^2*SIN(Y)^4-2*X^2*SIN(Y)^2+X^4*COS(Y)^4
           -2*X^4*COS(Y)^2+X^4+X^2+1$
(c7) trigsimp(foo);
        4    2     4          4    2       4
(d7) (x  + x ) cos (y) - 2 x  cos (y) + x  + 1

(c8) format(foo,%p(x),trigsimp);
       4    4       2    4
(d8) x  sin (y) + x  cos (y) + 1
```

The following examples illustrate the use of subtemplates

```
(c9) l1:[1+2*a+a^2,1+2*b+b^2,1+2*c+c^2]$
(c10) format(l1,%list,%f);
              2         2         2
(d10) [(a + 1) , (b + 1) , (c + 1) ]

(c11) format(l1,%list[%noop,%f]);
        2                   2   2
(d11) [a  + 2 a + 1, (b + 1) , c  + 2 c + 1]

(c12) format(l1,%list[%noop,%ditto(%f)]);
        2                   2         2
(d12) [a  + 2 a + 1, (b + 1) , (c + 1) ]
```

The following examples illustrate the usage with 'bags.'

```
(c13) format([a=b,c=d,e=f],%equation);
(d13) [a, c, e] = [b, d, f]

(c14) format(%,%list);
(d14) [a = b, c = d, e = f]

(c15) m1:matrix([a^2+2*a+1=q,b^2+2*b+1=r],
                [c^2+2*c+1=s,d^2+2*d+1=t])$
(c16) format(m1,%equation,%matrix[%noop,%list[%noop,%factor]]);
       [  2             2           ]
       [ a  + 2 a + 1  b  + 2 b + 1 ]   [ q  r ]
(d16)  [                            ] = [      ]
       [  2                     2   ]   [ s  t ]
       [ c  + 2 c + 1    (d + 1)    ]
```

The more concise `format(m1,%eq,%el(2,2),%f);` obtains the same result.

And a more involved example:

```
(c17) sqrtp(f):=not(atom(f)) and op(f)='sqrt$
```

```
(c18) first(solve(a*x^2+b*x-(b-2*a)/4,x));
                    2           2
          sqrt(b  + a b - 2 a ) + b
(d18) x = - -------------------------
                      2 a

(c19) format(%,%rhs,%preformat(%p(match(sqrtp))),
                   %match(sqrtp),%arg(1),%f);
          sqrt((b - a) (b + 2 a))    b
(d19) x = - ----------------------- - ---
                      2 a            2 a
```

2.1 User defined templates

New templates can be defined by giving the template keyword the property `formatter`; the value should be a function (or lambda expression) of the expression to be formatted and any parameters for the template.

For example, `%rectform` and `%if` could be defined as

```
put(%rectform,lambda([c],
                 block([r:rectformlist(c)],
                    format_piece(r[1]) +%I* format_piece(r[2]))),
    formatter)$
put(%if, lambda([x,test],
           if test(x) then format_piece(x,1)
           else            format_piece(x,2)),
     formatter)$
```

Tools useful for defining templates are the following.

`format_piece`(*piece*,{*nth*}) *Function*

> Format a given piece of an expression, automatically accounting for subtemplates and the remaining template chain. A specific subtemplate, rather than the next one, can be selected by specifying *nth*.

`coerce_bag`(*op*,*expr*) *Function*

> Attempts to coerce *expr* into an expression with *op* (one of `"="`, `"#"`, `"<"`, `"<="`, `">"`, `">="`, `"["` or `matrix`) as the top-level operator. It coerces the expression by swapping operands between layers – but only if adjacent layers are also lists, matrices or relations. This model assumes that a list of equations, for example, can be viewed as an equation whose sides are lists. Certain combinations, particularly those involving inequalities may not be meaningful, however, so some caution is advised.

3 COEFLIST; Determining coefficients

We define the 'algebras' of polynomials, trigonometric sums and power series to be those expressions that can be cast into the following forms.

$$\begin{aligned}
\mathcal{P}(v_1,\ldots) &= \left\{ P \;\middle|\; P = \sum_i c_i v_1^{p_{1,i}} v_2^{p_{2,i}} \cdots \right\}, \\
\mathcal{T}(v_1,\ldots) &= \left\{ T \;\middle|\; T = \sum_i c_i \cos(m_{1,i} v_1 + \cdots) + \sum_i s_i \sin(m'_{1,i} v_1 + \cdots) \right\}, \\
\mathcal{S}(v,\mathcal{O}) &= \left\{ S \;\middle|\; S = \sum_i^n c_i v^{p_i}; p_n \leq \mathcal{O} \right\}.
\end{aligned}$$

The variables v_i may be any atomic expression in the sense of `ratvars`[1]. The shorthands `operator(op)` and `match(predicate)` may be used to specify all subexpressions having op as an operator, or that pass the predicate, respectively.

The coefficients c_i and s_i are general MACSYMA expressions. In principle they would be independent of the variables v_i, but in practice they may contain non-polynomial dependence (or non-trigonometric, in the trigonometric case). These non-polynomial cases would include expressions like $(1+x)^n$, where n is symbolic. Likewise, $(x^a)^b$ is, in general, multivalued; unless $a = 1$ or $b \in Z$ or `radexpand=all`, it will not be interpreted as $x^{ab} \in \mathcal{P}$. Furthermore, we extend the algebras to include lists, vectors, matrices and equations, by interpreting a list of polynomials, say, as a polynomial with lists as coefficients.

The exponents p_i in series are restricted to numbers, but the exponents $p_{j,i}$ and multiples $m_{j,i}$ for polynomials and trigonometric sums may be general expressions (excluding bags).

The following functions construct a list of the coefficients and 'keys', that is, the exponents or multiples. Note that these are sparse representations — no coefficients are zero.

$$\begin{aligned}
\texttt{coeffs}(P, v_1, \ldots) &\rightarrow [[\%\texttt{poly}, v_1, \ldots], [c_1, p_{1,1}, \ldots], \ldots] \\
\texttt{trig_coeffs}(T, v_1, \ldots) &\rightarrow \\
&[[\%\texttt{trig}, v_1, \ldots], [[c_1, m_{1,1}, \ldots], \ldots], [[s_1, m'_{1,1}, \ldots], \ldots]] \\
\texttt{series_coeffs}(S, v, \mathcal{O}) &\rightarrow [[\%\texttt{series}, v, \mathcal{O}], [c_1, p_1], \ldots, [c_n, p_n]] \\
\texttt{Taylor_coeffs}(S, v, \mathcal{O}) &\rightarrow [[\%\texttt{Taylor}, v, \mathcal{O}], [c_1, p_1], \ldots, [c_n, p_n]]
\end{aligned}$$

The latter two functions both expand an expression through order $\mathcal{O}$, but the series version only carries expands arithmetic operations and is often considerably faster than `Taylor_coeffs`.

Examples:

```
(c20) cl1:coeffs((a+b*x)*(c-x)^2,x);
                        2        2
(d20) [[%poly,x],[a c ,0],[b c  - 2 a c,1],[a - 2 b c,2],[b,3]]

(c21) map('first,rest(coeffs(
                        (a+b*x)*(c-x)^2=q0+q1*x+q2*x^2+q3*x^3,x)));
          2          2
(d21) [a c  = q0, b c  - 2 a c = q1, a - 2 b c = q2, b = q3]

(c22) trig_coeffs(2*(a+cos(x))*cos(x+3*y),x,y);
(d22) [[%trig,x,y],[],[[1,0,3],[2 a,1,3],[1,2,3]]]

(c23) series_coeffs((a+b*x)*(c-x)^2,x,2);
                            2        2
(d23) [[%series,x,2],[a c ,0],[b c  - 2 a c,1],[a - 2 b c,2]]

(c24) coeffs((a+b*x)*sin(x),x);
(d24) [[%poly,x],[a sin(x),0],[b sin(x),1]]

(c25) coeffs((a+log(b)*x)*(c-log(x))^2,operator(log));
                                   2       2
(d25) [[%poly,log(x),log(b)],[a c ,0,0],[c  x,0,1],[- 2 a c,1,0],
                               [- 2 c x,1,1],[a,2,0],[x,2,1]]
```

3.1 Related functions

`get_coef`(*clist*,$k_1, \ldots$) *Function*

Gets the coefficient from the coefficient list *clist* corresponding to the keys k_i. The keys are matched to variable powers when *clist* is a `%poly`, `%series` or `%Taylor` form. If *clist* is a `%trig` then k_1 should be `sin` or `cos` and the remaining keys are matched to multipliers.

`uncoef`(*clist*) *Function*

Reconstructs the expression from a coefficient list *clist*. The coefficient list can be any of the coefficient list forms.

`partition_poly`(*expr*,*test*,$v_1, \ldots$) *Function*

Partitions *expr* into two polynomials; the first is made of those monomials for which the function *test* returns `true` and the second is the remainder. The test function is called on the powers of the v_i.

`partition_trig`(*expr*,*sintest*,*costest*,$v_1, \ldots$) *Function*

Trigonometric analog to partition_poly; The functions *sintest* and *costest* select sine and co¡sine terms, respectively; each are called on the multipliers of the v_i.

partition_series(*expr,test,v,$\mathcal{O}$*) *Function*

partition_Taylor(*expr,test,v,$\mathcal{O}$*) *Function*

Analog to partition_poly for series.

Examples:

```
(c26) get_coef(CL1,2);
(d26) a - 2 b c

(c27) uncoef(cl1);
          3                   2         2                 2
(d27) b x  + (a - 2 b c) x  + (b c  - 2 a c) x + a c

(c28) partition_poly((a+b*x)*(c-x)^2,'evenp,x);
                         2       2      3        2
(d28) [(a - 2 b c) x  + a c , b x  + (b c  - 2 a c) x]
```

3.2 Support functions

matching_parts(*expr, predicate, args...*) *Function*

Returns a list of all subexpressions of *expr* for which the application predicate(piece,args...) returns True.

function_calls(*expr,functions...*) *Function*

Returns a list of all calls in *expr* involving any of *functions*.

function_arguments(*expr,functions...*) *Function*

Returns a list of all argument lists for calls to *functions* in *expr*.

Examples:

```
(c29) t2:(a+log(b)*x)*(c-log(x))^2$
(c30) matching_parts(t2,constantp);
(d30) [2, - 1]

(c31) function_calls(t2,log);
(d31) [log(x), log(b)]
```

4 Availability

This package has been tested in Macsyma Inc.'s versions 418.85 for Genera 8.3 and 418.1 for Sparc computers under SunOS 4.1.3, as well as the DOE 'maxima' version 4.155. The LISP source code is available from the author.

Acknowledgments

The author wishes to thank Jeffrey Golden (Macsyma, Inc.) for sharing his ideas which led to expanding the scope of the package. We thank Richard Fateman (U. C. Berkeley) for teaching us much about the internals of MACSYMA.

References

[1] Symbolics Inc., *Macsyma Reference Manual*, Symbolics Inc., Burlington MA., 1988.

[2] Knuth, Donald, *Seminumerical Algorithms, The art of computer programming*, Vol. 2. Addison-Wesley, 1969.

[3] Golden, Jeffrey P., private communication.

[4] Griesmer, J. H., Jenks, R. D. and Yun, Y. Y., "A FORMAT statement in SCRATCHPAD," SIGSAM Bulletin (9), 1975, pp. 24–25.

[5] Barnett, M. P. and Perry, K. R. , "Hierarchical Addressing in Symbolic Computation," Computers Math. Applic. , to appear.

A Implementation

In this appendix, we describe some of the most important elements of the implementation. It is not our intention to describe every facet in detail, rather, we offer it as an overview to the lisp code, and as a guide to anyone wishing to implement similar facilities for another CAS.

A.1 Coefficient Lists

The fundamental algorithm for converting polynomials, poisson series, etc. into canonical representations, such as the coefficient lists defined here, is as follows. First, an 'arithmetic' is implemented for the new representation. That is, the code to add, multiply and exponentiate (at least) objects in the new form is written (See [2] for algorithms). An expression is then converted recursively; depending on the main operator of the expression, its arguments are first converted and then they are combined appropriately. Atoms are converted in whatever way is appropriate for the representation.

This is the method used internally by the CRE and Poisson facilities of MACSYMA. An issue for us was whether it was best to leverage these existing facilities by transforming first to CRE or Poisson representations and from there into coefficient lists, or whether we should reimplement the methods for conversions directly into coefficient list form.

In the end, we decided to reimplement the method for polynomial and series arithmetic. The primary reason is that the CRE (and Taylor) transforms the entire expression into CRE form, including what will become the coefficients. This is unnecessary work for our purposes, and in the application to `format`, the work may immediately be undone at the next step. Indeed, if an expression had already been `format`'d, the current code may leave the coefficients in the correct form.

The Poisson package does not carry out any transformations of the coefficients and, so, was suitable for use in conversion to trigonometric coefficient lists. Ultimately, we rewrote much of the existing poisson package anyway. This was both to add flexibility (particularly to allow non-integral multipliers) that would be useful both here and to users of the Poisson package, and also to remedy a long standing limitation of the package — it failed to detect when encoded trigonometric arguments exceeded the predeclared bounds resulting in spurious computations. Contact the author for information about this alternate Poisson package. However, we have an implementation of `trig_coeffs` that avoids using Poisson, should our alternative Poisson package be unacceptable for whatever reason.

`Taylor_coeffs`, the alternative conversion to series coefficient lists, does use `Taylor` as described above; it is useful when full Taylor expansions are needed.

A.2 Format

The basic operation of the formatting program is relatively simple; it is data-driven by the templates. The first template in the chain is examined and if it is a known formatting template, format binds the remaining template chain and the subtemplates. It then calls the function associated with the template on the expression and any parameters given to the template. Each template function transforms the expression appropriately and then calls `format_piece` on the appropriate pieces.

The function `format_piece` determines if there is a subtemplate that should be applied to a given piece or if the next template in the chain should be used. It then recursively invokes `format` to format the given piece with the selected template.