(Note: The current spelling Maxima is a corruption of the original name, Macsyma, used by Massachusetts Institute of Technology (MIT), Project MAC, which sold commercial rights to a version of this program. In order to avoid possible legal disputes, the public open-source version of the program was renamed. In the text below, most statements are true without change in Maxima. However, Macsyma, along with its original version of Lisp, does not distinguish upper and lower case, and so the programs and internal data may look like they are shouted out. Maxima uses lower case by default. -- RJF 2016)

Macsyma's General Simplifier: Philosophy and Operation

Richard J. Fateman
Computer Science Division
Electrical Engineering and Computer Sciences Dept.
University of California
Berkeley California

From: *Macsyma Users Conference 1979*

1. Introduction

Ideally the transformations performed by Macsyma's simplification program on algebraic expressions correspond to those simplifications desired by each user and each program. Since it is impossible for a program to intuit all users' requirements simultaneously, explicit control of the simplifier is necessary to override default transformations. A model of the simplification process is helpful in controlling this large and complex program.

Having examined several algebraic simplification programs, it appears that to date no program has been written which combines a conceptually simple and useful view of simplification with a program nearly as powerful as Macsyma's. *{Note: 1979. Not clear this would be different in 2001. RJF}*

Rule-directed transformation schemes struggle to approach the power of the varied control structures in more usual program schemes [Fenichel, 68]. *{Note: Mathematica pushes rules further. RJF}*

It is our belief that a thorough grasp of the decision and data structures of the Macsyma simplifier program itself is the most direct way of understanding its potential for algebraic expression transformation. This is an unfortunate admission to have to make, but it appears to reflect the state of the art in dealing with formalizations of complex programs. Simplification is a perplexing task. Because of this, we feel it behooves the "guardians of the simplifier" to try to meet the concerned Macsyma users part-way by documenting the program as it has evolved. We hope this paper continues to grow to reflect a reasonably accurate, complete, and current description. Of course Lisp program details are available to the curious, but even for those without a working knowledge of the Lisp language (in which the simplifier is written) we expect this paper to be of some help in answering questions which arise perennially as to why Macsyma deals with same particular class of expressions in some unanticipated fashion, or is inefficient in performing some set of transformations. Most often difficulties such as these are accounted for by implicit design decisions which are not evident from mere descriptions of what is done in the anticipated and usual cases. We also hope that improvements or revisions of the simplifier will benefit from the more centralized treatment of issues given here.

We also provide additional commentary which reflects our current outlook on how simplification programs should be written, and what capabilities they should have.

2. The ground rules

The general simplification package of Macsyma is a set of programs written in MacLisp, a dialect of Lisp 1.5. This package allows simplification and manipulation of algebraic expressions in the "general" tree-like form used for most of Macsyma. Several special purpose simplifiers, such as those for rational functions, Taylor series, trigonometric forms, factorial forms, etc., are not included in this discussion except peripherally. The general simplifier is the heart of Macsyma and has the unenviable task of dealing with the multiplicity of data types in Macsyma. It must also respond to a large number of "flag" settings in the environment in directing the simplification process. It must be sufficiently flexible so that additions, alterations, and deletions from its repertoire are possible, since the concept of simplicity is highly context dependent, and the user may seek to modify the program to conform to his specifications. An excellent discussion of the various meanings of simplification is given in [Moses, 71].

When we use the word "simplified" in this paper, we mean the result of running the simplification program on an expression. The fact that the "simplified" form may not appear to be simpler to the reader is usually irrelevant. The form is a consequence of the myriad transformations incorporated into the program over the years. Among the (less desirable) properties of the form is that two mathematically equivalent expressions may not simplify into identical results. -- Given the generality of the class of expressions handled, this is to be expected, in fact, necessary since the zero-equivalence problem is "recursively undecidable".

Many programs are written in such a fashion as to be resistant to misbehavior on invalid inputs. So-called "robust" programs thrive on the ability to detect and ward off illegal inputs. It is clearly of value to have a robust simplifier in Macsyma. However, as far as possible Macsyma attempts to avoid specific checks for bad input until absolutely necessary. In this way an input not originally envisioned by the author of the main program can percolate down to a routine, added later, which makes sense of it. The intermediate programs will not notice the problem at all. This is a very effective way of working in a growing system but obviously has its pitfalls if truly unexpected errors percolate down to a routine unable to handle them. Furthermore, the messages that can be generated at a low level are not usually able to reflect the global causes Thus "division by zero" can sometimes be very uninformative, when the user has typed a command with no division in it at all. We will return to these issues in particular contexts.

Historically, the Macsyma simplifier is a modification of an "On-line Algebraic Simplify Program" written by Knut Korsvold [Korsvold, 65] Some of the programs remain only as shells of their original form, but many names are the same. (Some variable names have a definite Norwegian flavor.) Major additions were made along lines not possible in Korsvold's original program because of space limitations in his Q-32 Lisp system. Major deletions from his simplifier structure include his provisions for substitution and "rule-directed" simplification, and for polynomial canonical forms, including greatest-common-divisor calculation. These functions are provided in considerably different form in Macsyma.

3. Internal forms of data in Macsyma

There are two external forms of data in Macsyma visible to the user, and three (sometimes more) internal forms. The external form for input is a string of ASCII characters as typed in by the user or read from a file. (For "example, the string "y+3/x+y"). The output display of characters on a two-dimensional "grid" is the other external form, for example,

```
          3
   2y + ---.
          x
```

Between the external forms there are several internal forms. In actuality each is a Lisp "S-expression," a tree-like nested structure of list cells, pointers, and atomic names. The printed forms of these, expressed in the Lisp language, will usually look like parenthesized prefix strings. The different internal forms are:

(1) The output from the parser, which is handed to the command interpreter.

(2) The simplified form which most commands obtain by evaluating and simplifying their arguments.

(3) The formatted form, which the display program uses to translate from the simplified form to a more "user oriented" expression. In the Berkeley VAX/UNIX version of the simplifier/display, a fourth form, namely typesetter code, is also used. (There can be other forms, e.g. TeX form, rational form..)

(Note: Other forms, some related to the MathML representation for World-Wide Web display are used in various front-ends for the Maxima version of Macsyma. RJF 2016)

Some commands, as have been previously noted, impose different notions of simplification on expressions. We will not deal with these in any detail, although it is part of the overall philosophy of Macsyma that alternative forms are allowed to co-exist, or can be converted to one another, in order to provide the most storage-efficient data structure and/or provide the fastest type of computation.

By and large the data represents algebraic expressions, but sometimes represents programs, program fragments, messages, names of variables, files, or other information.

The simplification program, SIMPLIFYA, is usually invoked by manipulatory functions such as the integration package. The input is an algebraic expression written in a type of parenthesized prefix notation. The input may have been partly simplified in the past, or may be "raw" from the parser.

An expression is either an integer or floating-point number, an atomic "indeterminate," or a Lisp S-expression of the form (OPERATOR . ARGLIST). The ARGLIST is a list in the Lisp meaning, which contains the appropriate number of arguments for the associated OPERATOR. Each argument is itself (with a few exceptions having to do with the OPERATORs MRAT and $POIS) an algebraic expression. Many operators have a fixed number of arguments. The exceptions include "n-ary" operators like PLUS, TIMES and DERIVATIVE.

There is an initial collection of OPERATORs known to the simplifier; these may be augmented by using the "TELLSIMP" commands in Macsyma discussed in section B.

The symbols in the OPERATORs are a mixture of historical conventions, and several have two or more variants which are intended to signify the difference between a "verb" (e.g. Integrate this expression) and a "noun" (e.g. Consider the integral ...). The first part of each OPERATOR (the Lisp CAR) is indicative of the algebraic meaning. The rest of the OPERATOR consists of "flags" describing the arguments, or modifying the meaning of the OPERATOR. We make these notions more definite by examples in the next sections.

3.1. The internal form generated by the parser

The following table indicates the "raw" output of the parser which corresponds to various input strings. The "input" column consists of character strings which (if followed by ";" or "$") are accepted by the top-level Macsyma parser. Since the parser is extensible, it is possible for the user to extend this table. An attentive reader may wonder if this table indicates all the possible forms of input to the simplifier. In fact, it does not, because some programs can and do generate forms which cannot be typed in directly by the user. Some of these forms may be generated by components of the simplifier itself.

Some of the parser input is included primarily for completeness, and indicates various subtleties in the current Macsyma "top-level" language which we will not explain here.

The mysterious prefixing system involving the translation of single-quote (') to % and & originated in an attempt to separate the user's name space from the Lisp-programmer's name space. The `?` prefix is an attempt to thwart this separation. The separation is not effective in any case since using a Macsyma- loaded package written by someone else (or yourself!) again provides a potential for name-conflict. There are other techniques for "automatic prefixing" to achieve proper results. The prefixes are also used to separate related "noun" and "verb" forms, where they can both exist.

(Note: whether the ANSI CL programming language with its package mechanism provides a useful alternative is not entirely clear. The Maxima implementation makes only slight use of the package system in CL and retains the single-character prefix tradition. RJF 2016)

Macsyma Syntax and Internal Representation

Input String	**Parser Output**
`a`	`$a`
`?a`	`a`
`"a"`	`&a`
`` `a ``	`((mquote) $a)`
`x+y`	`((mplus) $x $y)`
`x-y`	`((mplus) $x ((mminus) $y)`
`x*y`	`((mtimes) $x $y)`
`a(x)`	`((sa) $x)`
`a[1,2]`	`(($a array) 12)`
`a[1,2](x)`	`((mqapply) (($a array) 1 2) $x)`
`sin(x)`	`((%sin) $x)`
`x/y`	`((mquotient) $x $y)`
`x.y`	`((mnctimes) $x $y)`
`x^^2 or x2`	`((mexpt) $x 2)`
`x^^2`	`((mncexpt) $x 2)`
`[a,b,c]`	`((mlist) $a $b $c)`
`(a,b,c)`	`((dolist) $a $b $c)`
`if a then b`	`((mcond) $a $b t $false)`
`if a then b else c`	`((mcond) $a $b t $c)`
`for i:a thru b step c unless q do f(i)`	`((mdo) $i $a $c nil $b $q (($f) $i))`
`for i:a next n unless q do f(i)`	`((mdo) $i $a nil $n nil $q (($f) $i))`
`for i in L do f(i)`	`((mdoin) $i $L nil nil nil nil (($f) $i))`
`diff(y,x)`	`(($diff) $y $x 1)`
`diff(y,x,2,z,1)`	`(($diff) $y $x 2 $z 1)`
`` `diff(y.x) ``	`((%derivative) $y $x 1)`
`integrate(a,b,c,d)`	`((sintegrate) $a $b $c $d)`
`` `integrate(a,b,c,d) ``	`((%integrate) $a $b $c $d)`
`block([l1,l2], s1,s2)`	`((mprog) ((must) $l1 $l2) $si $s2)`
`block(s1,s2)`	`((mprog) ((mlist)) $s1 $s2)`
`not a`	`((mnot) $a)`
`a or b`	`((mor) $a $b)`
`a and b`	`((mand) $a $b)`
`a=b`	`((mequal) $a $b)`
`a>b`	`((mgreaterp) $a $b)`

```
a>=b                    ((mgeqp) $a $b)
a<b                     ((mlessp) $a $b
a<=b                    ((mleqp) $a $b)
a#b                     ((mnotequal) $a $b)
a:b                     ((msetq) $a $b)
a::b                    ((mset) $a $b)
a(x):f                  ((mdefine) (($a) $x) $f)
```

3.2. The operators and flags of the internal form

In each of the following subsections we list the basic form of the OPERATOR, followed by possible variants, and an explanation of the meaning and usage of that OPERATOR. This listing is not complete. An exhaustive listing of all the built- in operators would be substantially larger. Under the discussion of MPLUS we give more information about the flags. Much of this information is true of other operators also.

(MPLUS) (MPLUS SIMP) (MPLUS SIMP RATSIMP) (MPLUS SIMP FACTORED) (MPLUS SIMP IRREDUCIBLE FACTORED) (MPLUS SIMP SQFRED) (MPLUS SIMP TRUNC)

This is addition. As an example, ((MPLUS SIMP) 3 $X) might be the result of simplification of X+3. The ARGLIST in this example is (3 $X). In general, the associated ARGLIST may contain zero or more arguments.

In the normal course of operation, the parser provides an expression like ((MPLUS) $X $Y). The $X represents an "X" typed in by the user, the $Y, a "Y" typed in. A simplified form of this would be ((MPLUS SIMP) $X $Y), that is, no change except to place the "SIMP" flag on the OPERATOR's "property list".

The SIMP flag is used to indicate to interested parties that all subexpressions in the ARGLIST, and the expression itself are already simplified. In the case of large subexpressions, this prevents the simplifier from duplicating efforts when such (previously simplified) expressions are re-used. (This is discussed in more detail in section 5.2).

The flag RATSIMP is included on expressions which have been processed by the RATSIMP command. That is, they have been rationally simplified in an attempt to impose a set of criteria for providing a "simplest" form.

The flag FACTORED indicates that this is the top-level operator resulting from a call to the command FACTOR. Thus FACTOR(FACTOR(...)) does not require two factorizations and is rendered surprisingly fast; about as fast as FACTOR(...).

The flag IRREDUCIBLE indicates that this is a subexpression which the factoring program found irreducible (usually over the integers).

The SQFRED means this subexpression is "square-free", or has no multiple factors, considered as a polynomial over the integers. The SQFR factoring command produces this.

The TRUNC flag merely indicates a display convention: ((MPLUS TRUNC) 1 $X) displays as 1+X+... whereas without that flag, it would display as X+1. A Lisp programmer is able to place other flags on this list, although resimplifying an expression will usually wipe out such flags if the form of the expression is changed at all. This is necessary in general because the simplifier does not know what properties are preserved by its transformations. It would be desirable for Macsyma-level users to be able to set and access flags, but no facility has been provided for this, and it has been acknowledged as a problem in several circumstances. The only flag that can appear on an unsimplified operator is ARRAY. This would be rather an abuse of notation, but "+"[5,8](X,Y) is parsable. *(Many built-in operators have a prefix form: "+"(5,8) is the same as 5+8, and simplifies to 13. RJF 2016.)*

((MPLUS)) simplifies to 0, ((MPLUS) x) to x.

(MTIMES) (MTIMES SIMP) (MTIMES SIMP RATSIMP) (MTIMES FACTORED)

This is multiplication. The non-commutative "multiplication" operation is signified by MNCTIMES, and the treatment of that operator parallels MT1MES. The associated ARCLIST may contain zero or more arguments. The comments concerning the extra flags on MPLUS are relevant here too. ((MTIMES)) simplifies to 1, ((MTIMES) x) is x. The FACTORED flag indicates that the multiplicands were produced by the FACTOR or SQFR program, and (depending on some Macsyma flag settings) are irreducible (or powers of irreducible) polynomials over the integers. This flag prevents certain operations from taking place in the simplifier: e.g. ((MTIMES) 2 3) simplifies to 8, but ((MTIMES SIMP FACTORED) 2 3) is simplified. (MTIMES SIMP IRREDUCIBLE) presumably cannot be generated.

(MEXPT) (MEXPT SIMP) (MEXPT SIMP RATSIMP) (MEXPT FACTORED)

This is powering (exponentiation). MNCEXPT is the non-commutative version. The associated ARGLIST must contain exactly to arguments. The program ordinarily checks for this: no program calling the simplifier should provide such a badly-formed expression, but there are devious ways to con struct such an expression. (SUBST("-",F,F(X)), for example produces an error "Wrong number of args to -" and exits to Macsyma top-level. If debugging aids are enabled, the context of the error is saved and can be examined using Lisp). The SIMP, RATSIMP, and FACTORED flags have the same significance as previously indicated.

(MMINUS)

This is unary minus. It has no simplified form because ((MMINUS) x) is simplified to ((MTIMES SIMP) -1 x). Its transitory existence would seem superfluous if the parser were a bit cleverer; its justification is perhaps in the fact that prior to display, a program called NFORMAT introduces unary minus, (also difference and quotient) so that the user sees "-X" rather than "-1*X". Section 3.3 discusses this further.

(MQUOTIENT)

This is quotient (of two arguments). It also has no simplified form. ((MQUOTIENT x y) is changed to ((MTIMES) x ((MEXPT) y -1)), then simplified.

(%DERIVATIVE) ($DERIVATIVE SIMP)

This is the derivative operator, as used to stand for unevaluated derivatives as one sees, for example, in a differential equation. The ARCLIST consists of an odd number of arguments, representing the expression being differentiated, and the indeterminates it is being differentiated with respect to. Each indeterminate is followed by a number, indicating the order of the derivative.

(RAT) (RAT SIMP) (RAT SIMP FACTORED)

This is the operator for "rational number." It must have exactly two arguments, each an integer. Its existence is an alleged efficiency kludge which has led to certain unfortunate problems, many of which have been programmed around. Partly explained: 1/3 might be ((MEXPT) 3 -1) or ((RAT) 1 3). 1/(3*X) might be ((MEXPT) ((MTIMES 3 $X) -1) or ((MTIMES) ((RAT) 1 3)((MEXPT) $X -1)). It is. in fact the latter, and if you wish to detect subexpressions of the form 3X, the latter form is rather tricky.

(MRAT SIMP varlist genvars ...)

This operator signifies that the expression is in an internal canonical form for rational expressions (ratios of polynomials). The ARGLIST in this case has a special significance, and cannot be treated as sub-trees in an expression tree. There is no unsimplified form of this, since it is impossible to type it in directly as input and there is no concept of an unsimplified canonical rational expression. The (MRAT ...) OPERATOR, because it wards off the effects of simplification on subexpressions is also used for Taylor series and truncated power series in Macsyma, but the flags are somewhat different. Korsvold's original simplifier had a POLY operator with a similar significance, although the format of the polynomial
representation was different. Here "varlist" is a list of Macsyma variables or non-rational "kernels" such as ((%SIN SIMP) $X) which are potentially contained in the body of the rational expression; "genvar" is a corresponding list of generated symbols used internally for naming those "kernels." The details of the polynomial representation will not be discussed here.

(%SIN) (%SIN SIMP) (%SIN SIMP IRREDUCIBLE) (%SIN SIMP RATSIMP IRREDUCIBLE)

This is the sine (or SIN) operator. It is representative of a selection of other trigonometric and hyperbolic operators (and inverses) known to Macsyma: %COS, %TAN, %COTI %CSC, %SEC, ZSINHI %COSH. %TANH, %COTH, %CSCH, %SECH, %ASIN, %ACOS, %ATAN, ZACOT. %ACSC, ZASEC, %ASINH. %ACOSH, %ATANH.

(BFLOAT SIMP n)

This is the arbitrary-precision floating point number representation. The default value of n is 58, the number of binary digits of significance carried by bigfloats. The operands in the arglist are the exponent and significand.

($POIS SIMP)

This is used by the Poisson series package, and is another exception to the "tree" interpretation of algebraic expressions. The simplifier keeps out of its innards.

($usernamed....function) ($usernamedfunction SIMP)

More-or-less arbitrary names devised by the user can be introduced into the system, and the simplifier will treat them basically by simplifying their arguments, if any, and leaving them otherwise unchanged, but with the SIMP flag now included. Several possible ways are available to change this mode of operation as indicated in section 8.

3.3. The pre-display formatted form

The internal form of the simplified expressions is not ordered for most appealing display, nor does it correspond to most users' intuitions about the order in which objects are stored. As a prelude to the display and to the PART command, the program NFORMAT is used to restore some of the redundant operators. It takes its cues from various flag settings of the user. For example, the MQUOTIENT operator is re-introduced, and used to format ((RAT SIMP) 1 3) as ((MQUOTIENT) 1 3). Depending on switch settings, various effects can be achieved. For example, setting EXPTDISPFLAG to TRUE favors X^(-1) over 1/X. Normally however, ((MEXPT) $X -1) is altered to ((MQUOTIENT) 1 $X) for display.

Other switches which affect the formatter/display combination include BFTRUNC, %EDISPFLAG, NOUNDISP, POWERDISP, STARDISP. SQRTDISPFLAG.

4. Program structure of the simplifier

The simplifier basically walks the expression tree of its input, simplifying subtrees using the simplification program associated with its root node. That is, subtrees of the form ((MPLUS ...)...) are simplified by the program SIMPLUS, ((MTIMES ...)...) by SIMPTIMES, and in a completely analogous fashion, we have SIMPEXPT and many others. The central program which dispatches on the operator is SIMPLIFYA. This program takes two arguments, the expression, and a flag (T or NIL) which specifies whether subexpressions (if any) should also be simplified. T specifies that simplification has already been done on subexpressions. In the case of an atom, SIMPLIFYA is the identity function except that when NUMER is set to TRUE, some atoms (like %PI), may be converted to approximate numerical equivalents. Otherwise, it operates by examining the operator of the expression; if it is MPLUS, MTIMES, or MEXPT, a dispatch is provided to the appropriate simplification routine. In the case of the less popular operators, the Lisp property list of the atom is queried for property "OPERATORS" for an appropriate Lisp simplification program. Thus (GET `%COS `OPERATORS) returns S1MP-%COS, the cosine simplifier. If there is no OPERATORS property, simplification proceeds to the subtrees. Finally, the declared operator properties (commutative, etc.) are checked, and applied to the expression.

Several of the built-in simplification programs deserve careful examination. In the following sections we describe these, defend their operation to the extent we can, and comment on alternatives where relevant.

4.1. SIMPLUS

In this subsection we provide a bit more detail than in others to reveal the typical program organization of a commuting operator such as MPLUS or TIMES.

Initially SIMPLUS saves (binds) the value of its input X to the program variable CHECK. CHECK is consequently a Macsyma expression whose highest operator is "+" (i.e. MPLUS). This value CHECK is used at the end to see whether the simplification process led to an identical expression, in which case SIMPLUS returns the original list structure, possibly' inserting a SIMP flag indicating that the expression is in simplified form.

As SIMPLUS proceeds through its main loop, picking operands off the expression it is given, it builds up various "subtotals" in various program variables:

```
EQNFLAG     is the sum of all equations
MATRIXFLAG  is the sum of all matrices and lists
SUMFLAG     is the sum of all sigma notation sums
RES         is the sum of all scalar quantities
```
(i.e. everything else)

Scalars (namely rational numbers, integers) are added by the routine PLS as they are encountered. The right and left hand sides of equations are added by invoking the simplifier on some suitably constructed lists. Sigma notation sums are added by SUMPLS.

As SIMPLUS proceeds through the loop, if it discovers any summand to be in CRE form, it will invoke the rule that CRE form is contagious over the other summands (both those already dealt with and those further down the list). An exception to this rule is made in the case that one or more of the other operands is a matrix, list, equation or sigma sum, in which case SIMPLUS the running sum.

A similar contagion is implemented for BFLOATs, to which other numeric values are coerced. At the end of the loop SIMPLUS first consolidates the sigma sum with the sum of the other scalars. It then makes the decision based on prevailing flags whether or not to add the scalar to every element of a list or matrix, or simply leave it as "+" of two unlike quantities. (For example, if DOSCMXPLUS is TRUE, then a scalar + matrix yields a matrix.)

The result is then added to both sides of whatever equations have been summed during the simplification process. That is, the simplified form of any Macsyma sum with an equation as a summand is always an equation.

A pervasive part of the simplification process is ordering the objects in the list of operands. This is described in section 5.3. Among the flags which affect the operation of SIMPLUS and its underlings are DOALLMXOPS, DOMXMXOPS, DOSCMXPLUS, LISTARITH.

4.2. SIMPTIMES

SIMPTIMES in many respects has the analogous tasks to perform, but has additional responsibilities. In earlier incarnations, the expansion of products of sums (EXPAND) was intertwined in the SIMPTJMES program and its immediate dependents. Its complexity was worsened in part by the various "extras" introduced by Macsyma to the concept of expansion: non-commuting "dot" product expansion is performed, as is derivative expansion. The interdependency was decreased in an attempt to boest efficiency and remove bugs. At the same time (about 1972. I would guess) the command RATEXPAND and some variants of it were introduced in order to provide a facility which did exactly "polynomial expansion" far more rapidly. One decision which had to be made in SIMPTIMES and in SIMPEXPT described below, was how to utilize the simplifier internally: what tools should be used to simplify sums which occur as part of the processing. This is important, since the process of resimplifying can be very costly, yet, forgetting some nuance of processing could be difficult to detect. Since the user can by means of TELLSIMP (see section 6), alter the manner in which sums are simplified, a call from this program directly to SIMPLUS or SIMPTIMES is not permitted. SIMPTIMES must use the proper path (i.e. by calling SIMPLIFYA, and possibly looking on the OPERATORS property of the atom MPLUS) to determine the true PLUS simplifier. Among the flags which affect the operation of SIMPTIMES and its principal subroutines are: MXOSIMP, OUTSUM, EXPOP, EXPON, DOSCMXOPS, DOMXMXOPS, DOALLMXOPS, DOMXTIMES, NUMBERP, RATMX.

4.3. SIMPEXPT

SIMPEXPT inherits much of the complexity of SIMPTIMES, although the exponentiation operator can have only two arguments. SIMPEXPT has to deal with the relationships of logarithms in the exponent, expansions of sums or products to powers, and the multitude of possible interpretations of matrices, equations, sigma notations, conversion from float to bigfloat, etc. SIMPEXPT is one of the longest LISP program in the Macsyma system. (For the curious, most programs are about 10 lines long, SIMPEXPT is about 120). A guided tour through SIMPEXPT, depending on your outlook, either reveals the glory of a program which attempts to do everything, or illustrates the essential bankruptcy of such an idea.

SIMPEXPT must itself deal with details of logarithms in the exponents. and transformations which cancel numerator and denominator terms (recall that there is no true denominator, only a factor in a product with a negative exponent), and other complications attributable to mixtures of floating point numbers, bigfloats. CRE's etc. The logarithm simplification program (SIMPLN) is actually quite simple since it has much less to worry about within its own purview.

The flags examined by SIMPEXPT include RATSIMPEXPONS, RADPRODEXPAND, LOGSIMP, DEMOIVRE, %EMODE, DOALLMXOPS, DOSCMXOPS, DOMXEXPT, NUMER. In EXPTRL, a substantial subroutine of SIMPEXPT, the additional flags checked are FLOAT, KEEPFLOAT, and RATPRINT.

4.4. Others

Of the other main programs in the simplifier structure, it is hard to pick out a few for special treatment. There are a large number of trigonometric routines which share a common analysis routine for determining special angle reductions and simplifications based on arguments of the form A+B*%PI for various integer or fractional integer values of B (and where A may be zero). We leave as exercises for the reader other simplifiers such as absolute value (SIMPABS). These can figured out in detail from listings once the reader has an understanding of the main routines noted above, which influence the availability of facilities used by these other operator simplifications.

5. Storage and time efficiency

5.1. Common subexpressions

Scattered throughout the major and minor simplification routines there are invocations of the function EQTEST and reference to a variable CHECK. These are part of a scheme to conserve memory and take advantage of common subexpressions.

During the process of simplifying an expression, new expressions naturally arise, and can clutter up things relatively quickly. Therefore, when entering simplification functions, the variable CHECK is set to the (initial) expression to be simplified, and when the function is finished, the result is compared by EQTEST for isomorphism to the original. In case they are isomorphic, the original is returned, and the "copy" will eventually be converted to free list space.

The simplified expression often equals the original in cases where the transformations of the algebraic program are relatively local, and large parts of the expression remain unchanged. For example, substitution may leave subexpressions unchanged if the pattern being replaced is absent from some sub-tree. Nevertheless, the simplifier may not immediately recognize such a situation. The principal effect of EQTEST, then, is to provide, to the extent possible, sharing of common subexpressions: new copies identical to the old will not be generated.

5.2. "Already simplified" flags

The SIMP flags or their equivalent have been found invaluable to several algebra systems, [Tobey, 85] not only Macsyma. The alternative combinatorial growth in time, as a tree-like expression is repeatedly re-simplified in vain, is disastrous. Except in unusual circumstances, a tree labeled with a SIMP flag need not be simplified again. Unfortunately, there is no information associated with the SIMP flag which gives an indication of the environment in which the earlier simplification took place. Thus combinations of expanded and unexpanded expressions, which could not be considered simplified consistently, can co-exist. Placing more information, such as a simplification context, at places in the expression tree, can hardly be of much use unless the manipulatory programs make some use of that. It is a difficult problem to characterize the minimal additional work that must be done to re-simplify an expression in context A given that is already simplified with respect to context B. An analogous problem which appears in Macsyma in the context of evaluation is concerned with how partially evaluated expressions are treated. This is touched upon in section 6.3.

In fact, something similar to a simplification context is in use with respect to special "canonical" simplifiers such as the polynomial CRE (MRAT) form, and the Poisson series form. By carrying their contexts as part of the data type, and writing the necessary conversion programs, significant savings in data representation and algorithm efficiency are achieved (also noted by [Hearn, 76]). Perhaps simplifier designs of the future can take this approach: the main line simplifier is a rule-directed transformer of combinations of canonical forms.

5.3. Ordering

An extremely expensive requirement on the simplifier is that it must impose a unique ordering on subexpressions so that for example, the expression a+b-a+b can be simplified to 2*b. The function GREAT provides an ordering of atoms, functions, constants, etc. The detailed comparisons are based on considerations of declarations of indeterminates. Some represent constants either by system convention (%PI, %I, %E) or by user declarations. Some represent scalars, and some "non-scalars". Because this requires repeated re-examination of attributes of atoms and expressions, it is important that the ordering be done as few times as possible. In the case of functions, comparisons are made on the basis of ordering of arguments. The principal functions used are ORDFN, for comparing two functions, ORDFNA for function/atom comparisons, and ALIKE1, which tests for equality except for SIMP flags on the operator, and the like.

Because of the incessant checking for declarations in these routines, it might be appropriate to strip out much of this from the standard system and apply a much simpler ordering function based on (for example) hash-coding of expressions. This more elaborate ordering function could then be substituted for the faster one in contexts where it would do some good, e.g. only after the user has declared constants or introduced other complications. (It is particularly appealing to advocate simplifying a simplifier.)

6. Modifying the simplifier at the user Level

When the user of a system like Macsyma introduces new functions or uses old functions in a way that is unfamiliar to the system, the user may discover one of two situations: Either the system simplifies some expression in the "wrong" way for the application, or the system ignores the particular properties of the functions.

The user can "turn off" the simplifier by setting SIMP to FALSE, and then program from scratch. All in all, this seems like a less attractive alternative than just writing the simplifier desired entirely in Lisp (or some other language). In fact, we do not mean to exclude this as a realistic option: We hope to see further work on program packages which deal with so-called "modes" in Macsyma, REDUCE, and SCRATCHPAD, or their host Lisp systems, to aid in writing special purpose simplifiers, using the tools in the algebra system.

Alternatively, the user can try to modify the simplifier. There are two general methods for this. One simple method is to use declarations for certain predictable properties, and the second, more flexible technique is to use the TELLSIMP commands.

6.1. Declarations to the simplifier

The simplifier can be told that a newly introduced function is linear, additive, multiplicative, commutative (or symmetric), antisymmetric, outative, right associative, or left associative, by means of the DECLARE command in Macsyma. For example, if the command DECLARE(F, COMMUTATIVE) is executed, then F(A,B)-F(B,A) simplifies to zero (F's arguments commute, and are therefore reordered: F(A,B)-F(A,B) is then simplified to zero.) Meanings of the other properties are indicated in the table below. In each example f is declared to have the indicated property.

`additive`	`f(a+b) ==> f(a)+f(b)`
`antisymmetric`	`f(a,b)+f(b, a) ==> 0`
`commutative`	`f(a,b)-f(b,a)    > 0`
`multiplicative`	`f(a*b) ==> f(a)f(b)`
`outative`	`f(3*a)   > 3*f(pj~)`
`linear`	`f(a+3*b) ==> f(a)+3*f(b) ~or instead of "3", any constant.~`
`associative`	`f(f(a,b),f(c,d))-f(a,f(b,f(c,d))) ==> 0`
`rassociative`	`f(a,f(b,c)) ==> f(f(a,b),c)`
`lassociative`	`f(f(a,b),c) ==> f(b,f(a,c))`
`evenfun`	`f(-g) ==> f(g)`
`oddfun`	`f(-g) ==> -f(g)`
`nary`	`f(f(a,b),f(c,d)) => f(a,b, c,d)`

In the case of linear, the first argument is used if f has two or more arguments. Thus the noun forms of limit, sum, and integrate are linear in their first arguments, with respect to their second arguments. That is, the definition of "constant" means "free of the second argument."

Some of these declarations are not used directly in the simplifier as we have delimited it here, but because of the arbitrarily extended boundaries of the simplifier, can be considered within its realm. For example, the program which "simplifies" limits, can use information about increasing or decreasing functions, and the integration program can reduce certain problems considerably by determining that the integrand is an odd function integrated over a symmetric domain or that a parameter is not an integer. A rudimentary inference capability allows for some deductions, and is sometimes used by commands in an attempt to determine the sign of an expression.

Additional declarations are available for objects which are not functions. Some of the uses are indicated below, where the indeterminate x has been declared to have the property indicated in the left-hand column.

`even`	`cos(x*%pi) ==> 1`
`odd`	`cos(x*%pi) ==> 0`
`integer`	`cos((x+1/2)*%pi) ==> 0`
`noninteger`	`used by integration`
`rational`	
`irrational`	
`real`	
`imaginary`	
`constant`	`linearfunction(x*y) = => x*linearfunction(y)`
`complex`	

```
scalar         used by matrix manipulation
```

The question arises: given all the possible functional simplifications described in these tables, how much of the simplifier needs to be hard-wired? It is clear that some of the reliance on switches would be difficult to consider within the framework of such declarations. One possibility would be to allow declarations to be conditional, depending on switch settings.

A major question of efficiency emerges in any system which is interpretive or rule directed. Attempts to compile rule-sets into functions have appeared at various times. (See [Jenks, 76] for what is probably the best description.) We remain skeptical about efficiency compared to conventional programs, both in data structure and program structure. Recently SCRATCHPAD seems to have conceded this point [Jenks, 79], and REDUCE has for some time permitted both types of program construction: rule transformations via general pattern matches and a program mode. (see for example, [Hearn, 73] and [Hearn, 76]).

We would like to see progress toward a more ambitious goal of "automatic programming" of algebraic manipulation programs. This would entail a combination of mathematical properties of the objects to be handled, and programming and data-structure expertise. Current algebra systems are most notable for their syntactic features and particular expertise in certain mathematical problems. These do not typically include algorithm selection or data-structure design.

An exploration of just how much of a simplifier can be constructed out of a skeleton and declarations is worth embarking on. Clearly REDUCE (see, for example, [Hearn, 73], and more recently [Hearn, 76]) has attempted to provide such a facility, but without the elaboration of as many built-in properties (linearity being one which REDUCE does include). The consequent reliance on a fairly general pattern matcher in REDUCE for so much of the simplification process doubtless exacts a penalty in speed.

6.2. The TELLSIMP constructions

The objective of this section is not to provide a tutorial in how to use the two commands TELLSIMP and TELLSIMPAFTER, but in how their use affects the behavior of the simplifier.

As has been indicated earlier, most operators in Macsyma have a built-in simplification program, indicated by the OPERATOR property of the operator-name. A newly introduced operator has no OPERATOR property. Thus by uttering F(X); the user introduces the operator "$F". The TELLSIMP and TELLSIMPAFTER commands allow the user to insert a simplifier program on the property list of the atom $F where SIMPLIFY will use it. This fits in to the usual simplifier procedures. If an operator already has a simplifier, say "oldsimp-$F", and then TELLSIMP(F(<pattern of args>), <replacement>) is executed, a new program replaces the old, having the following general outline:

```
newsimp-$F(args): =
 if new <pattern of args> matches
   then return (simplify( <replacement>))
   else return(oldsimp-$F(args));
```

Note that this can be recursively redone for an "evennewersimp$F" ad infinitum and also that the invocation of "simplify" can cause "newsimp-$F" to be invoked again. (Sometimes leading to an infinitely recursive scheme, if the replacement still consists of an instance of the same pattern.)

By maintaining auxiliary information about the order of TELLSIMP commands, the simplifier can be restored to earlier pristine states.

The TELLSIMPAFTER command is subtly different: the new program replacing the old follows the outline:

```
newsimp-$F(args): =
temp: oldsimp-$F(args);
 if temp still has leading operator "F" and
   temp's args match <pattern of args> then
     return(simplify*(<replacement> ) );
```

Where "simplify*" is similar to the ordinary simplification program, but has this TELLSIMPAFTER program on "F" disabled This last restriction turns out to be fairly natural. In effect TELLSIMPAFTER(F(<pattern>), replacement) means, "If all previous efforts to simplify F(args) have not removed the F as principal operator, see if <pattern> matches args. If no match, leave it alone, otherwise return <replacement>."

The sophisticated user of Macsyma who wishes to see exact programs for these advice-taking systems should study the easily-accessed Lisp programs produced by these commands. A design criterion for the pattern matching programs originally written by Fateman was that the patterns should actually be compilable and executable Lisp. An interpretive version, which has a speed and space advantage if in fact the pattern program is interpreted rather than compiled, is under construction. *(Note: The rules are currently interpreted. RJF 2016)*

6.3. Other semantic alterations to the system

The user is faced with a complex set of tools for specifying knowledge in Macsyma. In addition to simplification, there is a process of evaluation, drawing upon function definitions and substitution of values for variables. In this section we illustrate some of these alternatives briefly, so as to distinguish them from the objectives of the simplifier modification techniques described above.

For example, F may be defined as a function, e.g.

```
F(X):=IF X=0 THEN 1 ELSE 0$
```

(this may not have desired effect: F('R) evaluates to 0 since R is not syntactically identical to 0) or

```
F(X):=IF EQUAL(X,0) THEN 1 ELSE 0$
```

(this may not be desired: F('R) evaluates to an error condition: "Macsyma was unable to evaluate the predicate EQUAL(R,0)")

F may be left undefined, but with certain properties, for example, derivatives, or simplification rules.

```
TELLSIMP(F(0), i)$
```

causes F(0) to be replaced by 1, anywhere it occurs, but leaves F('R) untouched. This last situation is very similar to interspersing

```
SUBST(1,F(0),lastexpression)
```

frequently during the course of a calculation. A more subtle situation is replacing any expression F(<non-zero>) by 0. This can be programmed by

```
MATCHDECLARE(NZ,NONZERO)$
NONZERO(X):=IS(X#0)$
TELLSIMP(F(NZ),0)$
```

which cannot be easily simulated by calls to SUBST.

Without dwelling on the details here, we merely wish to indicate that in Macsyma the evaluator is another layer of interface which attempts to intuit the users' needs, and which coexists, sometimes uncomfortably, with the simplifier. The modeling of the evaluation process, which in some algebraic manipulation systems is equated with simplification (see [Hearn, 78]), represents another challenging area for study. In Macsyma, evaluation is more directly related to programming language semantics than algebraic transformations. We would like to see it move even more in that direction, and away from simplification.

7. Other simplifiers

One of the principal suggestions we wish to see adopted is a move to strengthen simplification programs by taking advantage of canonical simplifiers, and other special purpose programs, which in their contexts, can be relatively sure of achieving the desired effect. We here do no more than list the sections which we hope can be expanded upon in a future version of this paper, but whose contents can in fact be derived from previously published manuals or papers. The power of these programs represents the strongest argument for continuing to build upon the structure of Macsyma, rather than starting ab initio in writing a new system attempting to do everything one more time around, right.
(Note: This consists of the titles of subsections that might appropriately be added to this paper. Expanded version not written as of 2016. RJF)

7.1. Canonical rational expressions {*added 7/12/2017 RJF}*

There is a subsystem within Maxima called ``CRE" for Canonical Rational Expression, in which many operations can be done much more efficiently. In particular, polynomials in one or more variables, or ratios of those polynomials, can be compactly represented and efficiently operated on. Ordinarily the CRE form of a polynomial is in an expanded form, recursively in the list of variables in the polynomial. The ordering of the variables can be set by the user via the ratvars() command. If the expression can be expressed as a ratio of polynomials, all common factors will be cancelled.

The word ``canonical" here is intended to convey the idea that all equivalent polynomial expressions will be mapped into the same (canonical) format.

Maxima tries to do as much manipulation as possible of CRE forms entirely within the CRE system, if you give it a starting point.

In particular, once an expression is converted to this form by rat(expression), it is in CRE form, and Maxima will, to the extent possible, convert anything it touches to CRE form. For example, (rat(x)+1)^2 is immediately expanded to x^2+2*x+1.

It is possible to convert out of this form by ratdisrep().

For human consumption, CRE form is displayed in the conventional display format, but there should be a /R/ notation near the output label if all or part of the displayed expression is in CRE form.

There are a number of options regarding CRE form. One flag, ratfac, is normally false, but if true, allows CRE form to include factored or partially factored expressions to be maintained. Thus, (1+z)^3 will retain its form. However, adding 1 to that will force it to expand.

If floating point numbers are put in CRE form, a rational number nearby is used. See ratepsilon, ratprint, keepfloat.

Ratsimp, a popular simplification routine, essentially is ratdisrep(rat(..)); Fullratsimp traverses down into the non-rational kernels in a CRE form and simplifies those too. For example, cos((x^2-1)/(x-1)) to cos(x+1).

The notion of this rational field allows for algebraic extensions, and it is possible to adjoin algebraic elements by first setting algebraic:true and using tellrat. For example, tellrat(z^3=k) defines z as the cube root of k. (Also set ratfac:false if you were experimenting with that just now.)

rat(z+1)^5 returns (10+k)*z^2+(5+5*k)*z+10*k+1 because z^3, z^4, z^5 were reduced modulo z^3.

Another feature of Maxima that uses CRE form is the ratsubst() command, which seems superficially like subst, but will allow you to substitute for a form F that is only evident in an expression E if you try to divide E by F and look at the quotient and remainder.

Ratcoef() picks out coefficients from a polynomial CRE form especially rapidly.

Finally, some of the internal programs like factor() and integrate() use CRE form, out of the user's sight.

(Note: The following subsections have not been written as of August 2017)

7.2. RATSIMP and RADCAN

7.3. Simplification of sums

7.4. Taylor series

7.5. Trigonometric expressions

7.5.1. Poisson series

7.5.2. TRIGEXPAND

7.5.3. TRIGREDUCE

7.5.4. TRIGSIMP

8. Summary and conclusions

In this presentation we have suggested that a good algebraic simplification program be structured to allow for disciplined growth. Restructuring of "working" code may very well be desirable. More powerful simplification routines may be most easily constructed as canonical simplifiers for particular classes of expressions, whose applications are controlled, perhaps, by rule-directed context switching. The tools are at hand for storing "contexts of simplification," if we can resolve an adequate model for manipulation of these contexts. We hope to see more progress in "automatic programming" as an aid to the construction of simplification programs and related data structures. Regardless of the techniques used for achieving this goal, it is necessary that the user of an algebraic manipulation system be able to comprehend and to some extent alter, the default transformations of that system. This is an important argument for keeping things simple.

For the present, the Macsyma simplifier works for all of the people
some of the time, and some of the people, all of the time. While this crude characterization is not likely to change, we can provide much better service to those not immediately satisfied by providing better-documented and more consistent facilities in a framework which reflects a more systematic structure. To be successful, this structure must reflect the underlying mathematics and the algorithmic nature of the simplification process. The current program makes good use of a operator-operand tree model of algebraic expressions, which however, fails to make use of operator models [Doohovskoy, 1977]. It makes possible the use of canonical form simplifiers, but does not take full advantage of them in an entirely systematic way. While it provides schemes for altering the

default behavior of the program via TELLSIMP, DECLAREs, and user-settable flags, it is quite difficult to model the effects of these changes in the large.

Overall, we consider the Macsyma simplifier an impressive program, in spite of our criticisms. Working on and with it has given many of us insights into the challenges of simplification. It has helped us to redefine our objectives in view of many user requirements. We hope that we have also started to refine our techniques for constructing such programs in the future.

9. References

[Doohovskoy, 77] Doohovskoy, A. "Varieties of operator manipulation," Proc. Of the 1977 Macsyma Users' Conf. NASA CP-2012, July, 1977, (473-490).

[Fenichel, 68] R. Fenichel, "An On-line System for Algebraic Manipulation," doctoral dissertation, Harvard University, July, 1968, also Report MAC-TR-35, Project MAC, M.I.T., available from the Clearinghouse, document AD-857-282.

[Hearn, 73] A. C. Hearn, Reduce 2 User's Manual University of Utah Computational Physics Group Report No. UCP-19, March 1973.

[Hearn, 78] A. C. Hearn, "A new REDUCE model for algebraic simplification," Proc. 1976 ACM Symposium on Symbolic and Algebraic Computation, August, 1976, (46-50).

[Jenks. 76] R. D. Jenks, "A pattern compiler," Proc. 1976 ACM Symposium on
Symbolic and Algebraic Computation, August, 1978, (60-65).

[Jenks, 79] R. D. Jenks, "SCRATCHPAD/380: Reflections on a language design," SICSAM Bulletin 13, no. 1, Feb., 1979, (18-26).

[Korsvold, 65] Knut Korsvold, "On-Line Algebraic Simplify Program," Stanford A.I. Project Memo 37, Nov. 1965, 30 p.

[Moses, 71] Joel Moses, "Algebraic simplification, a guide for the perplexed," Comm. A.C.M. 14, no. 8, Aug., 1971, (527-538).

[Tobey, 65] R. G. Tobey, R. J. Bobrow, and S. N. Ziles, "Automatic Simplification in Formac," Proc. AFIPS 1965 Fall Joint Comput. Conf., (1965) (37-52).

(Footnote: Work reported herein was supported in part by the U. S. Department of Energy. Contract DE-ATO3-76SF00034, Project Agreement DE-ASO3-79ER10358, and the National Science Foundation under Grant No. UCS 7807291 and by the Laboratory for Computer Science at M.I.T. currently supported by the United States Department of Energy under contract E(11-1)-a3070, and the National Aeronautics and Space Administration under grant NSG 1323.)