

# Maxima Manual

MAXIMA is a fairly complete computer algebra system.

This system MAXIMA is a COMMON LISP implementation due to William F. Schelter, and is based on the original implementation of Macsyma at MIT, as distributed by the Department of Energy. I now have permission from DOE to make derivative copies, and in particular to distribute it under the GNU public license. See the file COPYING included in the distribution. Thus these files may now be redistributed under the terms of GNU public license.



# 1 Introduction to MAXIMA

Start MAXIMA with the command "maxima". MAXIMA will display version information and a prompt. End each MAXIMA command with a semicolon. End the session with the command "quit();". Here's a sample session:

```
sonia$ maxima
GCL (GNU Common Lisp) Version(2.3) Tue Mar 21 14:15:15 CST 2000
Licensed under GNU Library General Public License
Contains Enhancements by W. Schelter
Maxima 5.4 Tue Mar 21 14:14:45 CST 2000 (enhancements by W. Schelter)
Licensed under the GNU Public License (see file COPYING)
(C1) factor(10!);

                                     8 4 2
(D1)                                     2 3 5 7
(C2) expand((x+y)^6);

          6      5      2 4      3 3      4 2      5      6
(D2)      y  + 6 x y  + 15 x y  + 20 x y  + 15 x y  + 6 x y  + x
(C3) factor(x^6-1);

(D3)      (x - 1) (x + 1) (x2 - x + 1) (x2 + x + 1)
(C4) quit();

sonia$
```

MAXIMA can search the info pages. Use the *describe* command to show all the commands and variables containing a string, and optionally their documentation:

```
(C1) describe(factor);

0: DONTFACTOR :(maxima.info)Definitions for Matrices and ..
1: EXPANDWRT_FACTORED :Definitions for Simplification.
2: FACTOR :Definitions for Polynomials.
3: FACTORFLAG :Definitions for Polynomials.
4: FACTORIAL :Definitions for Number Theory.
5: FACTOROUT :Definitions for Polynomials.
6: FACTORSUM :Definitions for Polynomials.
7: GCFACTOR :Definitions for Polynomials.
8: GFACTOR :Definitions for Polynomials.
9: GFACTORSUM :Definitions for Polynomials.
10: MINFACTORIAL :Definitions for Number Theory.
11: NUMFACTOR :Definitions for Special Functions.
12: SAVEFACTORS :Definitions for Polynomials.
13: SCALEFACTORS :Definitions for Miscellaneous Options.
14: SOLVEFACTORS :Definitions for Equations.
Enter n, all, none, or multiple choices eg 1 3 : 2 8;
```

Info from file /d/linux/local/lib/maxima-5.4/info/maxima.info:

- Function: FACTOR (EXP)

factors the expression exp, containing any number of variables or functions, into factors irreducible over the integers.  
 FACTOR(exp, p) factors exp over the field of integers with an element adjoined whose minimum polynomial is p. FACTORFLAG[FALSE] 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. SAVEFACTORS[FALSE] if TRUE causes the factors of an expression which is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors. BERLEFACT[TRUE] if FALSE then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used. INTFACLIM[1000] is the largest divisor which will be tried when factoring a bignum integer. If set to FALSE (this is the case when the user calls FACTOR explicitly), or if the integer is a fixnum (i.e. fits in one machine word), complete factorization of the integer will be attempted. The user's setting of INTFACLIM is used for internal calls to FACTOR. Thus, INTFACLIM may be reset to prevent MACSYMA from taking an inordinately long time factoring large integers. NEWFAC[FALSE] may be set to true to use the new factoring routines. Do EXAMPLE(FACTOR); for examples.

- Function: GFACTOR (EXP)

factors the polynomial exp over the Gaussian integers (i. e. with  $\text{SQRT}(-1) = \%I$  adjoined). This is like FACTOR(exp,A\*\*2+1) where A is %I.

(C1) GFACTOR(X\*\*4-1);

(D1) (X - 1) (X + 1) (X + %I) (X - %I)

(D1) FALSE

To use a result in later calculations, you can assign it to a variable or refer to it by its automatically supplied label. In addition, % refers to the most recent calculated result:

(C2) u:expand((x+y)^6);

(D2)  $y^6 + 6xy^5 + 15x^2y^4 + 20x^3y^3 + 15x^4y^2 + 6x^5y + x^6$

(C3) diff(u,x);

(D3)  $6y^5 + 30x^4y^4 + 60x^2y^3 + 60x^3y^2 + 30x^4y + 6x^5$

(C4) factor(d3);

$$(D4) \quad 6 (y + x)$$

MAXIMA knows about complex numbers and numerical constants:

$$(C6) \text{ cos}(\%pi);$$

$$(D6) \quad - 1$$

$$(C7) \%e^{(i*\%pi)};$$

$$(D7) \quad - 1$$

MAXIMA can do differential and integral calculus:

$$(C8) \text{ u:expand}((x+y)^6);$$

$$(D8) \quad y^6 + 6 x y^5 + 15 x^2 y^4 + 20 x^3 y^3 + 15 x^4 y^2 + 6 x^5 y + x^6$$

$$(C9) \text{ diff}(\%,x);$$

$$(D9) \quad 6 y^5 + 30 x y^4 + 60 x^2 y^3 + 60 x^3 y^2 + 30 x^4 y + 6 x^5$$

$$(C10) \text{ integrate}(1/(1+x^3),x);$$

$$(D10) \quad - \frac{\text{LOG}(x^2 - x + 1)}{6} + \frac{\text{ATAN}\left(\frac{2x - 1}{\text{SQRT}(3)}\right)}{\text{SQRT}(3)} + \frac{\text{LOG}(x + 1)}{3}$$

MAXIMA can solve linear systems and cubic equations:

$$(C11) \text{ linsolve}([3*x + 4*y = 7, 2*x + a*y = 13], [x,y]);$$

$$(D11) \quad \left[ x = \frac{7a - 52}{3a - 8}, y = \frac{25}{3a - 8} \right]$$

$$(C12) \text{ solve}(x^3 - 3*x^2 + 5*x = 15, x);$$

$$(D12) \quad [x = -\text{SQRT}(5) \%I, x = \text{SQRT}(5) \%I, x = 3]$$

MAXIMA can solve nonlinear sets of equations. Note that if you don't want a result printed, you can finish your command with \$ instead of ;.

$$(C13) \text{ eq1: } x^2 + 3*x*y + y^2 = 0\$$$

$$(C14) \text{ eq2: } 3*x + y = 1\$$$

$$(C15) \text{ solve}([\text{eq1}, \text{eq2}]);$$

$$(D15) \quad \left[ \left[ y = -\frac{3 \text{SQRT}(5) + 7}{2}, x = \frac{\text{SQRT}(5) + 3}{2} \right], \right.$$

$$\left. \left[ y = \frac{3 \text{SQRT}(5) - 7}{2}, x = \frac{\text{SQRT}(5) - 3}{2} \right] \right]$$

$$\left. \left[ y = \frac{\text{SQRT}(5) - 3}{2}, x = \frac{3 \text{SQRT}(5) - 7}{2} \right] \right]$$

$$[y = \frac{\text{-----}}{2}, x = -\frac{\text{-----}}{2}]$$

Under the X window system, MAXIMA can generate plots of one or more functions:

```
(C13) plot2d(sin(x)/x, [x, -20, 20]);
```

```
(YMIN -3.0 YMAX 3.0 0.29999999999999999)
```

```
(D13) 0
```

```
(C14) plot2d([atan(x), erf(x), tanh(x)], [x, -5, 5]);
```

```
(YMIN -3.0 YMAX 3.0 0.29999999999999999)
```

```
(YMIN -3.0 YMAX 3.0 0.29999999999999999)
```

```
(YMIN -3.0 YMAX 3.0 0.29999999999999999)
```

```
(D14) 0
```

```
(C15) plot3d(sin(sqrt(x^2+y^2))/sqrt(x^2+y^2), [x, -12, 12], [y, -12, 12]);
```

```
(D15) 0
```

Moving the cursor to the top left corner of the plot window will pop up a menu that will, among other things, let you generate a PostScript file of the plot. (By default, the file is placed in your home directory.) You can rotate a 3D plot.

## 2 Help

### 2.1 Introduction to Help

The most useful online help command is DESCRIBE which obtains help on all commands containing a particular string. Here by command we mean a built in operator such as INTEGRATE or FACTOR etc. As a typing short cut you may type ? fact in lieu of describe("fact")

```
(C3) ? inte;
```

```
0: (maxima.info)Integration.
1: Introduction to Integration.
2: Definitions for Integration.
3: INTERRUPTS.
4: ASKINTEGER :Definitions for Simplification.
5: DISPLAY_FORMAT_INTERNAL :Definitions for Input and Output.
6: INTEGRERP :Definitions for Miscellaneous Options.
7: INTEGRATE :Definitions for Integration.
8: INTEGRATION_CONSTANT_COUNTER :Definitions for Integration.
9: INTERPOLATE :Definitions for Numerical.
Enter n, all, none, or multiple choices eg 1 3 : 7 8;
```

```
Info from file /d/linux2/local/share/info/maxima.info:
```

```
- Function: INTEGRATE (EXP, VAR)
  integrates exp with respect to var or returns an integral
  expression (the noun form) if it cannot perform the integration
  (see note 1 below). Roughly speaking three stages are used:
...

```

In the above the user said he wanted items 7 and 8. Note the ; following the two numbers. He might have typed all to see help on all the items.

### 2.2 Lisp and Maxima

All of Maxima is of course written in lisp. There is a naming convention for functions and variables: All symbols which begin with a "\$" sign at lisp level, are read with the "\$" sign stripped off at Macsyma level. For example, there are two lisp functions TRANSLATE and \$TRANSLATE. If at macsyma level you enter TRANSLATE(FOO); the function which is called is the \$translate function. To access the other function you must prefix with a "?". Note you may *not* put a space after the ? since that would indicate you were looking for help!

```
(C1) ?TRANSLATE(FOO);
```

Of course, this may well not do what you wanted it to do since it is a completely different function.

To enter a lisp command you may use



```
(C1) :lisp (foo 1 2)
```

or to get a lisp prompt use `to_lisp()`; or alternately type `Ctrl-c` to enter into a debug break. This will cause a lisp break loop to be entered. You could now evaluate `$d2` and view the value of the line label `D2`, in its internal lisp format. Typing `:q` will quit to top level, if you are in a debug break. If you had exited maxima with `to_lisp()`; then you should type

```
MAXIMA>(run)
```

at the lisp prompt, to restart the Maxima session.

If you intend to write lisp functions to be called at macsyma level you should name them by names beginning with a "\$". Note that all symbols typed at lisp level are automatically read in upper case, unless you do something like `|$odeSolve|` to force the case to be respected. Maxima interprets symbols as mixed case, if the symbol has already been read before or at the time it was first read there was not an already existing symbol with the same letters but upper case only. Thus if you type

```
(C1) Integrate;
(D1) INTEGRATE
(C2) Integ;
(D2) Integ
```

The symbol `Integrate` already existed in upper case since it is a Maxima primitive, but `INTEG`, does not already exist, so the `Integ` is permitted. This may seem a little bizarre, but we wish to keep old maxima code working, which assumes that Maxima primitives may be in upper or lower case. An advantage of this system is that if you type in lower case, you will immediately see which are the maxima keywords and functions.

To enter Maxima forms at lisp level, you may use the `#$` macro.

```
(setq $foo #$(x,y)$)
```

This will have the same effect as entering

```
(C1)F00: [X,Y];
```

except that `foo` will not appear in the `VALUES` list. In order to view `foo` in macsyma printed format you may type

```
(displa $foo)
```

In this documentation when we wish to refer to a macsyma symbol we shall generally omit the `$` just as you would when typing at macsyma level. This will cause confusion when we also wish to refer to a lisp symbol. In this case we shall usually try to use lower case for the lisp symbol and upper case for the macsyma symbol. For example `LIST` for `$list` and `list` for the lisp symbol whose printname is "list".

Since functions defined using the MAXIMA language are not ordinary lisp functions, you must use `mfuncall` to call them. For example:

```
(D2) F00(X, Y) := X + Y + 3
```

then at lisp level

```
CL-MAXIMA>>(mfuncall '$foo 4 5)
12
```

A number of lisp functions are shadowed in the maxima package. This is because their use within maxima is not compatible with the definition as a system function. For example `typep` behaves differently common lisp than it did in Maclisp. If you want to refer to the zeta lisp `typep` while in the maxima package you should use `global:typep` (or `cl:typep` for common lisp). Thus

```
(macsyma:typep '(1 2)) ==> 'list
(lisp:typep '(1 2))==> error (lisp:type-of '(1 2))==> 'cons
```

To see which symbols are shadowed look in "src/maxima-package.lisp" or do a `describe` of the package at lisp level.

## 2.3 Garbage Collection

Symbolic computation tends to create a good deal of garbage, and effective handling of this can be crucial to successful completion of some programs.

Under GCL, on UNIX systems where the `mprotect` system call is available (including SUN OS 4.0 and some variants of BSD) a stratified garbage collection is available. This limits the collection to pages which have been recently written to. See the GCL documentation under `ALLOCATE` and `GBC`. At the lisp level doing `(setq si::*notify-gbc* t)` will help you determine which areas might need more space.

## 2.4 Documentation

The source for the documentation is in `.texi` texinfo format. From this format we can produce the info files used by the online commands `?`  and `describe`. Also `html` and `pdf` files can be produced.

Additionally there are examples so that you may do

```
example(integrate);
(C4) example(integrate);
(C5) test(f):=BLOCK([u],u:INTEGRATE(f,x),RATSIMP(f-DIFF(u,x)));
(D5) test(f) := BLOCK([u], u :
      INTEGRATE(f, x), RATSIMP(f - DIFF(u, x)));
(C6) test(SIN(x));
(D6)      0
(C7) test(1/(x+1));
(D7)      0
(C8) test(1/(x^2+1));
(D8)      0
(C9) INTEGRATE(SIN(X)^3,X);
...

```

## 2.5 Definitions for Help

**DEMO** (*file*) Function  
 this is the same as BATCH but pauses after each command line and continues when a space is typed (you may need to type ; followed by a newline, if running under xmaxima). The demo files have suffix `.dem`

**DESCRIBE** (*cmd*) Function  
 This command prints documentation on all commands which contain the substring "cmd". Thus

```
(C1) describe("integ");
0: (maxima.info)Integration.
1: Introduction to Integration.
2: Definitions for Integration.
3: ASKINTEGER :Definitions for Simplification.
..
Enter n, all, none, or multiple choices eg 1 3 : 2 3;
Info from file /d/linux2/local/share/info/maxima.info:
Definitions for Integration
=====

- Function: CHANGEVAR (EXP,F(X,Y),Y,X)
...
```

see [Section 2.1 \[Introduction to Help\], page 7](#)

**EXAMPLE** (*command*) Function  
 will start up a demonstration of how command works on some expressions. After each command line it will pause and wait for a space to be typed, as in the DEMO command.

## 3 Command Line

### 3.1 Introduction to Command Line

**%TH** (*i*) Function

is the *i*th previous computation. That is, if the next expression to be computed is  $D(j)$  this is  $D(j-i)$ . This is useful in BATCH files or for referring to a group of D expressions. For example, if SUM is initialized to 0 then FOR I:1 THRU 10 DO SUM:SUM+%TH(I) will set SUM to the sum of the last ten D expressions.

**"' "** operator

- (single quote) has the effect of preventing evaluation. E.g. '(F(X)) means do not evaluate the expression F(X). 'F(X) means return the noun form of F applied to [X].

**"" "** operator

- (two single quotes) causes an extra evaluation to occur. E.g. "c4; will re-execute line C4. "(F(X)) means evaluate the expression F(X) an extra time. "F(X) means return the verb form of F applied to [X].

### 3.2 Definitions for Command Line

**ALIAS** (*newname1, oldname1, newname2, oldname2, ...*) Function

provides an alternate name for a (user or system) function, variable, array, etc. Any even number of arguments may be used.

**DEBUG** () Function

LISPDEBUGMODE(); DEBUGPRINTMODE(); and DEBUG(); make available to the user debugging features used by systems programmers. These tools are powerful, and although some conventions are different from the usual macsyma level it is felt their use is very intuitive. [Some printout may be verbose for slow terminals, there are switches for controlling this.] These commands were designed for the user who must debug translated macsyma code, as such they are a boon. See MACDOC;TRDEBG USAGE for more information. For more help, consult GJC.

**DEBUGMODE** Variable

default: [FALSE] - causes MACSYMA to enter a MACSYMA break loop whenever a MACSYMA error occurs if it is TRUE and to terminate that mode if it is FALSE. If it is set to ALL then the user may examine BACKTRACE for the list of functions currently entered.

**DEBUGPRINTMODE** () Function

LISPDEBUGMODE(); DEBUGPRINTMODE(); and DEBUG(); make available to the user debugging features used by systems programmers. These tools are powerful,

and although some conventions are different from the usual macsyima level it is felt their use is very intuitive. [Some printout may be verbose for slow terminals, there are switches for controlling this.] These commands were designed for the user who must debug translated macsyima code, as such they are a boon. See MACDOC;TRDEBG USAGE for more information. For more help, consult GJC.

**EV** (*exp, arg1, ..., argn*) Function

is one of MACSYMA's most powerful and versatile commands. It evaluates the expression *exp* in the environment specified by the *argi*. This is done in steps, as follows:

- (1) First the environment is set up by scanning the *argi* which may be as follows: **SIMP** causes *exp* to be simplified regardless of the setting of the switch **SIMP** which inhibits simplification if **FALSE**. **NOEVAL** suppresses the evaluation phase of **EV** (see step (4) below). This is useful in conjunction with the other switches and in causing *exp* to be resimplified without being reevaluated. **EXPAND** causes expansion. **EXPAND(m,n)** causes expansion, setting the values of **MAXPOSEX** and **MAXNEGEX** to *m* and *n* respectively. **DETOUT** causes any matrix inverses computed in *exp* to have their determinant kept outside of the inverse rather than dividing through each element. **DIFF** causes all differentiations indicated in *exp* to be performed. **DERIVLIST(var1,...,vark)** causes only differentiations with respect to the indicated variables. **FLOAT** causes non-integral rational numbers to be converted to floating point. **NUMER** causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in *exp* which have been given numerals to be replaced by their values. It also sets the **FLOAT** switch on. **PRED** causes predicates (expressions which evaluate to **TRUE** or **FALSE**) to be evaluated. **EVAL** causes an extra post-evaluation of *exp* to occur. (See step (5) below.) **E** where **E** is an atom declared to be an **EVFLAG** causes **E** to be bound to **TRUE** during the evaluation of *exp*. **V:expression** (or alternately **V=expression**) causes **V** to be bound to the value of *expression* during the evaluation of *exp*. Note that if **V** is a MACSYMA option, then *expression* is used for its value during the evaluation of *exp*. If more than one argument to **EV** is of this type then the binding is done in parallel. If **V** is a non-atomic expression then a substitution rather than a binding is performed. **E** where **E**, a function name, has been declared to be an **EVFUN** causes **E** to be applied to *exp*. Any other function names (e.g. **SUM**) cause evaluation of occurrences of those names in *exp* as though they were verbs. In addition a function occurring in *exp* (say **F(args)**) may be defined locally for the purpose of this evaluation of *exp* by giving **F(args):=body** as an argument to **EV**. If an atom not mentioned above or a subscripted variable or subscripted expression was given as an argument, it is evaluated and if the result is an equation or assignment then the indicated binding or substitution is performed. If the result is a list then the members of the list are treated as if they were additional arguments given to **EV**. This permits a list of equations to be given (e.g. [**X=1, Y=A\*\*2**]) or a list of names of equations (e.g. [**E1,E2**] where **E1** and **E2** are equations) such as that returned by **SOLVE**. The *argi* of **EV** may be given in any order with the exception of substitution equations which are handled in sequence, left to right, and **EVFUNS** which are composed, e.g. **EV(exp,RATSIMP,REALPART)**

is handled as `REALPART(RATSIMP(exp))`. The `SIMP`, `NUMER`, `FLOAT`, and `PRED` switches may also be set locally in a block, or globally at the "top level" in MACSYMA so that they will remain in effect until being reset. If `exp` is in CRE form then `EV` will return a result in CRE form provided the `NUMER` and `FLOAT` switches are not both `TRUE`.

- (2) During step (1), a list is made of the non-subscripted variables appearing on the left side of equations in the `argi` or in the value of some `argi` if the value is an equation. The variables (both subscripted variables which do not have associated array functions, and non-subscripted variables) in the expression `exp` are replaced by their global values, except for those appearing in this list. Usually, `exp` is just a label or `%` (as in (C2) below), so this step simply retrieves the expression named by the label, so that `EV` may work on it.
- (3) If any substitutions are indicated by the `argi`, they are carried out now.
- (4) The resulting expression is then re-evaluated (unless one of the `argi` was `NOEVAL`) and simplified according to the `argi`. Note that any function calls in `exp` will be carried out after the variables in it are evaluated and that `EV(F(X))` thus may behave like `F(EV(X))`.
- (5) If one of the `argi` was `EVAL`, steps (3) and (4) are repeated.

#### Examples

(C1) `SIN(X)+COS(Y)+(W+1)**2+'DIFF(SIN(W),W);`

(D1)  $\cos(Y) + \sin(X) + \frac{d}{dW} \sin(W) + (W + 1)^2$

(C2) `EV(% ,SIN,EXPAND,DIFF,X=2,Y=1);`

(D2)  $\cos(W) + W^2 + 2W + \cos(1) + 1.90929742$

An alternate top level syntax has been provided for `EV`, whereby one may just type in its arguments, without the `EV()`. That is, one may write simply

`exp, arg1, ..., argn.`

This is not permitted as part of another expression, i.e. in functions, blocks, etc.

(C4) `X+Y,X:A+Y,Y:2;`

(D4)  $Y + A + 2$

(Notice the parallel binding process)

(C5) `2*X-3*Y=3$`

(C6) `-3*X+2*Y=-4$`

(C7) `SOLVE([D5,D6]);`

SOLUTION

(E7)  $Y = -\frac{1}{5}$

(E8)  $X = -\frac{6}{5}$

(D8) `[E7, E8]`

(C9) `D6,D8;`

```

(D9)          - 4 = - 4
(C10) X+1/X > GAMMA(1/2);
          1
(D10)      X + - > SQRT(%PI)
          X
(C11) %,NUMER,X=1/2;
(D11)          2.5 > 1.7724539
(C12) %,PRED;
(D12)          TRUE

```

**EVFLAG**

Variable

default: [] - the list of things known to the EV function. An item will be bound to TRUE during the execution of EV if it is mentioned in the call to EV, e.g. EV(% ,numer);. Initial evflags are

```

FLOAT, PRED, SIMP, NUMER, DETOUT, EXPONENTIALIZE, DEMOIVRE,
KEEPFLOAT, LISTARITH, TRIGEXPAND, SIMPSUM, ALGEBRAIC,
RATALGDENOM, FACTORFLAG, %EMODE, LOGARC, LOGNUMBER,
RADEXPAND, RATSIMPEXPONS, RATMX, RATFAC, INFEVAL, %ENUMER,
PROGRAMMODE, LOGNEGINT, LOGABS, LETRAT, HALFANGLES,
EXPTISOLATE, ISOLATE_WRT_TIMES, SUMEXPAND, CAUCHYSUM,
NUMER_PBRANCH, M1PBRANCH, DOTSCRULES, and LOGEXPAND.

```

**EVFUN**

Variable

- the list of functions known to the EV function which will get applied if their name is mentioned. Initial evfuns are FACTOR, TRIGEXPAND, TRIGREDUCE, BFLOAT, RATSIMP, RATEXPAND, RADCAN, LOGCONTRACT, RECTFORM, and POLARFORM.

**INFEVAL**

special symbol

leads to an "infinite evaluation" mode. EV repeatedly evaluates an expression until it stops changing. To prevent a variable, say X, from being evaluated away in this mode, simply include X='X as an argument to EV. Of course expressions such as EV(X,X=X+1,INFEVAL); will generate an infinite loop. CAVEAT EVALUATOR.

**KILL** (*arg1, arg2, ...*)

Function

eliminates its arguments from the MACSYMA system. If *argi* is a variable (including a single array element), function, or array, the designated item with all of its properties is removed from core. If *argi*=LABELS then all input, intermediate, and output lines to date (but not other named items) are eliminated. If *argi*=CLABELS then only input lines will be eliminated; if *argi*=ELABELS then only intermediate E-lines will be eliminated; if *argi*=DLABELS only the output lines will be eliminated. If *argi* is the name of any of the other information lists (the elements of the MACSYMA variable INFOLISTS), then every item in that class (and its properties) is KILLED and if *argi*=ALL then every item on every information list previously defined as well as LABELS is KILLED. If *argi*=a number (say *n*), then the

last  $n$  lines (i.e. the lines with the last  $n$  line numbers) are deleted. If  $\text{arg}_i$  is of the form  $[m,n]$  then all lines with numbers between  $m$  and  $n$  inclusive are killed. Note that  $\text{KILL}(\text{VALUES})$  or  $\text{KILL}(\text{variable})$  will not free the storage occupied unless the labels which are pointing to the same expressions are also KILLED. Thus if a large expression was assigned to  $X$  on line  $C7$  one should do  $\text{KILL}(D7)$  as well as  $\text{KILL}(X)$  to release the storage occupied.  $\text{KILL}(\text{ALLBUT}(\text{name}_1, \dots, \text{name}_k))$  will do a  $\text{KILL}(\text{ALL})$  except it will not KILL the names specified. (Note:  $\text{name}_i$  means a name such as  $U, V, F, G$ , not an infolist such as  $\text{FUNCTIONS}$ .)  $\text{KILL}$  removes all properties from the given argument thus  $\text{KILL}(\text{VALUES})$  will kill all properties associated with every item on the  $\text{VALUES}$  list whereas the  $\text{REMOVE}$  set of functions ( $\text{REVALUE}, \text{REMFUNCTION}, \text{REMARRAY}, \text{REMRULE}$ ) remove a specific property. Also the latter print out a list of names or  $\text{FALSE}$  if the specific argument doesn't exist whereas  $\text{KILL}$  always has value  $\text{"DONE"}$  even if the named item doesn't exist. Note that killing expressions will not help the problem which occurs on  $\text{MC}$  indicated by  $\text{"NO CORE - FASLOAD"}$  which results when either too many  $\text{FASL}$  files have been loaded in or when allocation level has gotten too high. In either of these cases, no amount of killing will cause the size of these spaces to decrease. Killing expressions only causes some spaces to get emptied out but not made smaller.

- LABELS** (*char*) Function  
 takes a char  $C, D$ , or  $E$  as arg and generates a list of all  $C$ -labels,  $D$ -labels, or  $E$ -labels, respectively. If you've generated many  $E$ -labels via  $\text{SOLVE}$ , then  
 $\text{FIRST}(\text{REST}(\text{LABELS}(C)))$   
 reminds you what the last  $C$ -label was.  $\text{LABELS}$  will take as arg any symbolic name, so if you have reset  $\text{INCHAR}, \text{OUTCHAR},$  or  $\text{LINECHAR}$ , it will return the list of labels whose first character matches the first character of the arg you give to  $\text{LABELS}$ . The variable,  $\text{LABELS}$ , default:  $[],$  is a list of  $C, D,$  and  $E$  lines which are bound.
- LASTTIME** Variable  
 - the time to compute the last expression in milliseconds presented as a list of  $\text{"time"}$  and  $\text{"gctime"}$ .
- LINENUM** Variable  
 - the line number of the last expression.
- MYOPTIONS** Variable  
 default:  $[],$  - all options ever reset by the user (whether or not they get reset to their default value).
- NOLABELS** Variable  
 default:  $[\text{FALSE}]$  - if  $\text{TRUE}$  then no labels will be bound except for  $E$  lines generated by the solve functions. This is most useful in the  $\text{"BATCH"}$  mode where it eliminates the need to do  $\text{KILL}(\text{LABELS})$  in order to free up storage.
- OPTIONSET** Variable  
 default:  $[\text{FALSE}]$  - if  $\text{TRUE}$ ,  $\text{MACSYMA}$  will print out a message whenever a  $\text{MACSYMA}$  option is reset. This is useful if the user is doubtful of the spelling of some



option and wants to make sure that the variable he assigned a value to was truly an option variable.

**PLAYBACK** (*arg*) Function

"plays back" input and output lines. If  $arg=n$  (a number) the last  $n$  expressions ( $C_i$ ,  $D_i$ , and  $E_i$  count as 1 each) are "played-back", while if  $arg$  is omitted, all lines are. If  $arg=INPUT$  then only input lines are played back. If  $arg=[m,n]$  then all lines with numbers from  $m$  to  $n$  inclusive are played-back. If  $m=n$  then  $[m]$  is sufficient for  $arg$ .  $Arg=SLOW$  places **PLAYBACK** in a slow-mode similar to **DEMO**'s (as opposed to the "fast" **BATCH**). This is useful in conjunction with **SAVE** or **STRINGOUT** when creating a secondary-storage file in order to pick out useful expressions. If  $arg=TIME$  then the computation times are displayed as well as the expressions. If  $arg=GCTIME$  or **TOTALTIME**, then a complete breakdown of computation times are displayed, as with **SHOWTIME:ALL**;.  $Arg=STRING$  strings-out (see **STRING** function) all input lines when playing back rather than displaying them. If  $ARG=GRIND$  "grind" mode can also be turned on (for processing input lines) (see **GRIND**). One may include any number of options as in **PLAYBACK**([5,10],20,TIME,SLOW).

**PRINTPROPS** (*a, i*) Function

will display the property with the indicator  $i$  associated with the atom  $a$ .  $a$  may also be a list of atoms or the atom **ALL** in which case all of the atoms with the given property will be used. For example, **PRINTPROPS**([F,G],ATVALUE). **PRINTPROPS** is for properties that cannot otherwise be displayed, i.e. for **ATVALUE**, **ATOMGRAD**, **GRADEF**, and **MATCHDECLARE**.

**PROMPT** Variable

default: [-] is the prompt symbol of the **DEMO** function, **PLAYBACK**(**SLOW**) mode, and (**MACSYMA-BREAK**).

**QUIT** () Function

kills the current **MACSYMA** but doesn't affect the user's other jobs; equivalent to exiting to **DCL** and stopping the **MACSYMA** process. One may "quit" to **MACSYMA** top-level by typing **Control-C** **Control-G**; **Control-C** gets **NIL**'s interrupt prompt, at which one types either **Control-G** or just **G**. Typing **X** at the Interrupt prompt will cause a quit in a computation started within a **MACSYMA-BREAK** without disrupting the suspended main computation.

**REMFUNCTION** (*f1, f2, ...*) Function

removes the user defined functions  $f_1, f_2, \dots$  from **MACSYMA**. If there is only one argument of **ALL** then all functions are removed.

**RESET** () Function

causes all **MACSYMA** options to be set to their default values. (Please note that this does not include features of terminals such as **LINEL** which can only be changed by assignment as they are not considered to be computational features of **MACSYMA**.)

**RESTORE** (*file-specification*) Function  
 reinitializes all quantities filed away by a use of the SAVE or STORE functions, in a prior MACSYMA session, from the file given by file-specification without bringing them into core.

**SHOWTIME** Variable  
 default: [FALSE] - if TRUE then the computation time will be printed automatically with each output expression. By setting SHOWTIME:ALL, in addition to the cpu time MACSYMA now also prints out (when not zero) the amount of time spent in garbage collection (gc) in the course of a computation. This time is of course included in the time printed out as "time=" . (It should be noted that since the "time=" time only includes computation time and not any intermediate display time or time it takes to load in out-of-core files, and since it is difficult to ascribe "responsibility" for gc's, the gctime printed will include all gctime incurred in the course of the computation and hence may in rare cases even be larger than "time=").

**SSTATUS** (*feature,package*) Function  
 - meaning SET STATUS. It can be used to SSTATUS( FEATURE, HACK\_PACKAGE) so that STATUS( FEATURE, HACK\_PACKAGE) will then return TRUE. This can be useful for package writers, to keep track of what FEATURES they have loaded in.

**TOBREAK** () Function  
 causes the MACSYMA break which was left by typing TOPLEVEL; to be re-entered. If TOBREAK is given any argument whatsoever, then the break will be exited, which is equivalent to typing TOBREAK() immediately followed by EXIT;.

**TOPLEVEL** () Function  
 During a break one may type TOPLEVEL;. This will cause top-level MACSYMA to be entered recursively. Labels will now be bound as usual. Everything will be identical to the previous top-level state except that the computation which was interrupted is saved. The function TOBREAK() will cause the break which was left by typing TOPLEVEL; to be re-entered. If TOBREAK is given any argument whatsoever, then the break will be exited, which is equivalent to typing TOBREAK() immediately followed by EXIT;.

**TO\_LISP** () Function  
 enters the LISP system under MACSYMA. This is useful on those systems where control-uparrow is not available for this function.

**TTYINTFUN** Variable  
 default: [FALSE] - Governs the function which will be run whenever the User-interrupt-character is typed. To use this feature, one sets TTYINTFUN (default FALSE meaning feature not in use) to a function of no arguments. Then whenever (e.g.) ^U (control-U) is typed, this function is run. E.g. suppose you have a FOR statement loop which increments I, and you want an easy way of checking on the

value of I while the FOR statement is running. You can do: `TTYINTFUN:PRINTI$  
PRINTI():=PRINT(I)$` , then whenever you type (e.g.) `^U` you get the check you want.

## **TTYINTNUM**

Variable

default: [21] (the ascii value of Control-U (`^U`), U being the 21st letter of the alphabet). This controls what character becomes the User-interrupt-character. `^U` was chosen for it mnemonic value. Most users should not reset `TTYINTNUM` unless they are already using `^U` for something else.

## **VALUES**

Variable

default: [] - all bound atoms, i.e. user variables, not MACSYMA Options or Switches, (set up by `:` , `::` , or functional binding).

## 4 Operators

### 4.1 NARY

- An NARY operator is used to denote a function of any number of arguments, each of which is separated by an occurrence of the operator, e.g.  $A+B$  or  $A+B+C$ . The `NARY("x")` function is a syntax extension function to declare `x` to be an NARY operator. Do `DESCRIBE(SYNTAX)`; for more details. Functions may be DECLARED to be NARY. If `DECLARE(J,NARY)`; is done, this tells the simplifier to simplify, e.g.  $J(J(A,B),J(C,D))$  to  $J(A, B, C, D)$ .

### 4.2 NOFIX

- NOFIX operators are used to denote functions of no arguments. The mere presence of such an operator in a command will cause the corresponding function to be evaluated. For example, when one types "exit;" to exit from a MACSYMA break, "exit" is behaving similar to a NOFIX operator. The function `NOFIX("x")` is a syntax extension function which declares `x` to be a NOFIX operator. Do `DESCRIBE(SYNTAX)`; for more details.

### 4.3 OPERATOR

- See OPERATORS

### 4.4 POSTFIX

- POSTFIX operators like the PREFIX variety denote functions of a single argument, but in this case the argument immediately precedes an occurrence of the operator in the input string, e.g.  $3!$ . The `POSTFIX("x")` function is a syntax extension function to declare `x` to be a POSTFIX operator. Do `DESCRIBE(SYNTAX)`; for details.

### 4.5 PREFIX

- A PREFIX operator is one which signifies a function of one argument, which argument immediately follows an occurrence of the operator. `PREFIX("x")` is a syntax extension function to declare `x` to be a PREFIX operator. Do `DESCRIBE(SYNTAX)`; for more details.

### 4.6 Definitions for Operators

"!"

operator

The factorial operator, which is the product of all the integers from 1 up to its argument. Thus  $5! = 1*2*3*4*5 = 120$ . The value of /the option `FACTLIM` (default: [-1]) gives the highest factorial which is automatically expanded. If it is -1 then all integers are expanded. See also the `FACTORIAL`, `MINFACTORIAL`, and `FACTCOMB` commands.

"!!"	operator
Stands for double factorial which is defined as the product of all the consecutive odd (or even) integers from 1 (or 2) to the odd (or even) argument. Thus 8!! is $2*4*6*8 = 384$ .	
"#"	operator
The logical operator "Not equals".	
"."	operator
The dot operator, for matrix (non-commutative) multiplication. When "." is used in this way, spaces should be left on both sides of it, e.g. A . B. This distinguishes it plainly from a decimal point in a floating point number. Do APROPOS(DOT); for a list of the switches which affect the dot operator. DESCRIBE(switch-name); will explain them.	
":"	operator
The assignment operator. E.g. A:3 sets the variable A to 3.	
"::"	operator
Assignment operator. :: assigns the value of the expression on its right to the value of the quantity on its left, which must evaluate to an atomic variable or subscripted variable.	
"::="	operator
The "::=" is used instead of "!=" to indicate that what follows is a macro definition, rather than an ordinary functional definition. See DESCRIBE(MACROS).	
"!="	operator
The function definition operator. E.g. F(X):=SIN(X) defines a function F.	
"="	operator
denotes an equation to MACSYMA. To the pattern matcher in MACSYMA it denotes a total relation that holds between two expressions if and only if the expressions are syntactically identical.	

**ADDITIVE**

special symbol

- If DECLARE(F,ADDITIVE) has been executed, then: (1) If F is univariate, whenever the simplifier encounters F applied to a sum, F will be distributed over that sum. I.e.  $F(Y+X)$ ; will simplify to  $F(Y)+F(X)$ . (2) If F is a function of 2 or more arguments, additivity is defined as additivity in the first argument to F, as in the case of 'SUM or 'INTEGRATE, i.e.  $F(H(X)+G(X),X)$ ; will simplify to  $F(H(X),X)+F(G(X),X)$ . This simplification does not occur when F is applied to expressions of the form  $SUM(X[I],I,lower-limit,upper-limit)$ .

**ALLBUT**

keyword

works with the PART commands (i.e. PART, INPART, SUBSTPART, SUBSTINPART, DPART, and LPART). For example,

```
if EXPR is E+D+C+B+A,
then PART(EXPR,[2,5]);
==> D+A
```

while

```
PART(EXPR,ALLBUT(2,5))=>E+C+B
```

It also works with the KILL command,

```
KILL(ALLBUT(name1,...,namek))
```

will do a KILL(ALL) except it will not KILL the names specified. Note: namei means a name such as function name such as U, F, FOO, or G, not an infolist such as FUNCTIONS.

**ANTISYMMETRIC**

declaration

- If DECLARE(H,ANTISYMMETRIC); is done, this tells the simplifier that H is antisymmetric. E.g. H(X,Z,Y) will simplify to - H(X, Y, Z). That is, it will give  $(-1)^n$  times the result given by SYMMETRIC or COMMUTATIVE, where n is the number of interchanges of two arguments necessary to convert it to that form.

**CABS** (*exp*)

Function

returns the complex absolute value (the complex modulus) of exp.

**COMMUTATIVE**

declaration

- If DECLARE(H,COMMUTATIVE); is done, this tells the simplifier that H is a commutative function. E.g. H(X,Z,Y) will simplify to H(X, Y, Z). This is the same as SYMMETRIC.

**ENTIER** (*X*)

Function

largest integer  $\leq X$  where X is numeric. FIX (as in FIXnum) is a synonym for this, so FIX(X); is precisely the same.

**EQUAL** (*expr1,expr2*)

Function

used with an "IS", returns TRUE (or FALSE) if and only if expr1 and expr2 are equal (or not equal) for all possible values of their variables (as determined by RAT-SIMP). Thus IS(EQUAL((X+1)\*\*2,X\*\*2+2\*X+1)) returns TRUE whereas if X is unbound IS((X+1)\*\*2=X\*\*2+2\*X+1) returns FALSE. Note also that IS(RAT(0)=0) gives FALSE but IS(EQUAL(RAT(0),0)) gives TRUE. If a determination can't be made with EQUAL then a simplified but equivalent form is returned whereas = always causes either TRUE or FALSE to be returned. All variables occurring in exp are presumed to be real valued. EV(exp,PRED) is equivalent to IS(exp).

```
(C1) IS(X**2 >= 2*X-1);
(D1)                                     TRUE
(C2) ASSUME(A>1);
(D2)                                     DONE
```

```
(C3) IS(LOG(LOG(A+1)+1)>0 AND A^2+1>2*A);
(D3)                                     TRUE
```

**EVAL** Function  
causes an extra post-evaluation of *exp* to occur.

**EVENP** (*exp*) Function  
is TRUE if *exp* is an even integer. FALSE is returned in all other cases.

**FIX** (*x*) Function  
a synonym for ENTIER(*X*) - largest integer  $\leq X$  where *X* is numeric.

**FULLMAP** (*fn*, *exp1*, ...) Function  
is similar to MAP but it will keep mapping down all subexpressions until the main operators are no longer the same. The user should be aware that FULLMAP is used by the MACSYMA simplifier for certain matrix manipulations; thus, the user might see an error message concerning FULLMAP even though FULLMAP was not explicitly called by the user.

```
(C1) A+B*C$
(C2) FULLMAP(G,%);
(D2) G(B) G(C) + G(A)
(C3) MAP(G,D1);
(D3) G(B C) + G(A)
```

**FULLMAPL** (*fn*, *list1*, ...) Function  
is similar to FULLMAP but it only maps onto lists and matrices

```
(C1) FULLMAPL("+", [3, [4, 5]], [[A, 1], [0, -1.5]]);
(D1) [[A + 3, 4], [4, 3.5]]
```

**IS** (*exp*) Function  
attempts to determine whether *exp* (which must evaluate to a predicate) is provable from the facts in the current data base. IS returns TRUE if the predicate is true for all values of its variables consistent with the data base and returns FALSE if it is false for all such values. Otherwise, its action depends on the setting of the switch PREDERROR (default: TRUE). IS errs out if the value of PREDERROR is TRUE and returns UNKNOWN if PREDERROR is FALSE.

**ISQRT** (*X*) Function  
takes one integer argument and returns the "integer Sqrt" of its absolute value.

- MAX** (*X1, X2, ...*) Function  
yields the maximum of its arguments (or returns a simplified form if some of its arguments are non-numeric).
- MIN** (*X1, X2, ...*) Function  
yields the minimum of its arguments (or returns a simplified form if some of its arguments are non-numeric).
- MOD** (*poly*) Function  
converts the polynomial *poly* to a modular representation with respect to the current modulus which is the value of the variable **MODULUS**. **MOD**(*poly,m*) specifies a **MODULUS** *m* to be used for converting *poly*, if it is desired to override the current global value of **MODULUS**. See **DESCRIBE**(**MODULUS**); .
- ODDP** (*exp*) Function  
is **TRUE** if *exp* is an odd integer. **FALSE** is returned in all other cases.
- PRED** operator  
(**EVFLAG**) causes predicates (expressions which evaluate to **TRUE** or **FALSE**) to be evaluated.
- RANDOM** (*X*) Function  
returns a random integer between 0 and *X*-1. If no argument is given then a random integer between  $-2^{(29)}$  and  $2^{(29)} - 1$  is returned. If *X* is **FALSE** then the random sequence is restarted from the beginning. Note that the range of the returned result when no argument is given differs in **NIL MACSYMA** from that of **PDP-10** and **Multics MACSYMA**, which is  $-2^{(35)}$  to  $2^{(35)} - 1$ . This range is the range of the **FIXNUM** datatype of the underlying **LISP**.
- SIGN** (*exp*) Function  
attempts to determine the sign of its specified expression on the basis of the facts in the current data base. It returns one of the following answers: **POS** (positive), **NEG** (negative), **ZERO**, **PZ** (positive or zero), **NZ** (negative or zero), **PN** (positive or negative), or **PNZ** (positive, negative, or zero, i.e. nothing known).
- SIGNUM** (*X*) Function  
if  $X < 0$  then -1 else if  $X > 0$  then 1 else 0. If *X* is not numeric then a simplified but equivalent form is returned. For example, **SIGNUM**(-*X*) gives **-SIGNUM**(*X*).
- SORT** (*list, optional-predicate*) Function  
sorts the list using a suitable optional-predicate of two arguments (such as "**<**" or **ORDERLESSP**). If the optional-predicate is not given, then **MACSYMA**'s built-in ordering predicate is used.



**SQRT** (*X*) Function  
 the square root of *X*. It is represented internally by  $X^{(1/2)}$ . Also see ROOTSCONTRACT. RADEXPAND[TRUE] - if TRUE will cause *n*th roots of factors of a product which are powers of *n* to be pulled outside of the radical, e.g. SQRT(16\*X^2) will become 4\*X only if RADEXPAND is TRUE.

**SQRDISPFLAG** Variable  
 default: [TRUE] - if FALSE causes SQRT to display with exponent 1/2.

**SUBLIS** (*list,expr*) Function  
 allows multiple substitutions into an expression in parallel. Sample syntax:  

$$\text{SUBLIS}([A=B, B=A], \text{SIN}(A) + \text{COS}(B));$$

$$\Rightarrow \text{SIN}(B) + \text{COS}(A)$$

The variable SUBLIS\_APPLY\_LAMBDA[TRUE] controls simplification after SUBLIS. For full documentation, see the file SHARE2;SUBLIS INFO.

**SUBLIST** (*L,F*) Function  
 returns the list of elements of the list *L* for which the function *F* returns TRUE. E.g., SUBLIST([1,2,3,4],EVENP); returns [2,4].

**SUBLIS\_APPLY\_LAMBDA** Variable  
 default:[TRUE] - controls whether LAMBDA's substituted are applied in simplification after SUBLIS is used or whether you have to do an EV to get things to apply. TRUE means do the application.

**SUBST** (*a, b, c*) Function  
 substitutes *a* for *b* in *c*. *b* must be an atom, or a complete subexpression of *c*. For example,  $X+Y+Z$  is a complete subexpression of  $2*(X+Y+Z)/W$  while  $X+Y$  is not. When *b* does not have these characteristics, one may sometimes use SUBSTPART or RATSUBST (see below). Alternatively, if *b* is of the form  $e/f$  then one could use SUBST( $a*f,e,c$ ) while if *b* is of the form  $e**(1/f)$  then one could use SUBST( $a**f,e,c$ ). The SUBST command also discerns the  $X^Y$  in  $X^{-Y}$  so that SUBST(A,SQRT(X),1/SQRT(X)) yields 1/A. *a* and *b* may also be operators of an expression enclosed in "s or they may be function names. If one wishes to substitute for the independent variable in derivative forms then the AT function (see below) should be used. Note: SUBST is an alias for SUBSTITUTE. SUBST(eq1,exp) or SUBST([eq1,...,eqk],exp) are other permissible forms. The eqi are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression exp. EXPDSUBST[FALSE] if TRUE permits substitutions like  $Y$  for  $\%E**X$  in  $\%E**(A*X)$  to take place. OPSUBST[TRUE] if FALSE, SUBST will not attempt to substitute into the operator of an expression. E.g. (OPSUBST:FALSE, SUBST(X^2,R,R+R[0])); will work.

(C1) SUBST(A, X+Y, X+(X+Y)\*\*2+Y);

(D1) 
$$Y + X + A^2$$

(C2) SUBST(-%I,%I,A+B\*%I);

(D2)  $A - \%I B$

(Note that C2 is one way of obtaining the complex conjugate of an analytic expression.) For further examples, do EXAMPLE(SUBST);

**SUBSTINPART** (*x, exp, n1, ...*) Function

is like SUBSTPART but works on the internal representation of exp.

(C1) X.'DIFF(F(X),X,2);

(D1) 
$$X \cdot \left( \frac{d^2}{dX^2} F(X) \right)$$

(C2) SUBSTINPART(D\*\*2,%,2);

(D2) 
$$X \cdot D^2$$

(C3) SUBSTINPART(F1,F[1](X+1),0);

(D3) F1(X + 1)

Additional Information

If the last argument to a part function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus

(C1) PART(X+Y+Z,[1,3]);

(D1) Z+X

PIECE holds the value of the last expression selected when using the part functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below.

If PARTSWITCH[FALSE] is set to TRUE then END is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

(C1) 27\*Y\*\*3+54\*X\*Y\*\*2+36\*X\*\*2\*Y+Y+8\*X\*\*3+X+1;

(D1)  $27 Y^3 + 54 X Y^2 + 36 X^2 Y + Y + 8 X^3 + X + 1$

(C2) PART(D1,2,[1,3]);

(D2)  $54 Y^2$

(C3) SQRT(PIECE/54);

(D3)  $Y$

(C4) SUBSTPART(FACTOR(PIECE),D1,[1,2,3,5]);

(D4)  $(3 Y + 2 X)^3 + Y + X + 1$

(C5) 1/X+Y/X-1/Z;

(D5) 
$$-\frac{1}{Z} + \frac{Y}{X} - \frac{1}{X}$$

(C6) SUBSTPART(XTHRU(PIECE),%, [2,3]);

(D6) 
$$\frac{Y + 1}{X} - \frac{1}{Z}$$

Also, setting the option INFLAG to TRUE and calling PART/SUBSTPART is the same as calling INPART/SUBSTINPART.

**SUBSTPART** (*x, exp, n1, ..., nk*) Function  
 substitutes *x* for the subexpression picked out by the rest of the arguments as in PART. It returns the new value of *exp*. *x* may be some operator to be substituted for an operator of *exp*. In some cases it needs to be enclosed in "s (e.g. SUBSTPART("+",A\*B,0); -> B + A ).

(C1) 1/(X\*\*2+2);

(D1) 
$$\frac{1}{X^2 + 2}$$

(C2) SUBSTPART(3/2,%,2,1,2);

(D2) 
$$\frac{1}{X^{3/2} + 2}$$

(C3) A\*X+F(B,Y);

(D3) A X + F(B, Y)

(C4) SUBSTPART("+",%,1,0);

(D4) X + F(B, Y) + A

Also, setting the option INFLAG to TRUE and calling PART/SUBSTPART is the same as calling INPART/SUBSTINPART.

**SUBVARP** (*exp*) Function  
 is TRUE if *exp* is a subscripted variable, for example A[I].

**SYMBOLP** (*exp*) Function  
 returns TRUE if "exp" is a "symbol" or "name", else FALSE. I.e., in effect, SYMBOLP(X):=ATOM(X) AND NOT NUMBERP(X)\$ .

**UNORDER** () Function  
 stops the aliasing created by the last use of the ordering commands ORDERGREAT and ORDERLESS. ORDERGREAT and ORDERLESS may not be used more than one time each without calling UNORDER. Do DESCRIBE(ORDERGREAT); and DESCRIBE(ORDERLESS);, and also do EXAMPLE(UNORDER); for specifics.

**VECTORPOTENTIAL** (*givencurl*) Function  
 Returns the vector potential of a given curl vector, in the current coordinate system. POTENTIALZEROLOC has a similar role as for POTENTIAL, but the order of the left-hand sides of the equations must be a cyclic permutation of the coordinate variables.

**XTHRU** (*exp*)

Function

combines all terms of *exp* (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.

(C1)  $((X+2)**20-2*Y)/(X+Y)**20+(X+Y)**-19-X/(X+Y)**20;$

$$(D1) \quad \frac{1}{(Y+X)^{19}} - \frac{X}{(Y+X)^{20}} + \frac{(X+2)^{20} - 2Y}{(Y+X)^{20}}$$

(C2) XTHRU(%);

$$(D2) \quad \frac{(X+2)^{20} - Y}{(Y+X)^{20}}$$

**ZEROEQUIV** (*exp,var*)

Function

tests whether the expression *exp* in the variable *var* is equivalent to zero. It returns either TRUE, FALSE, or DONTKNOW. For example ZEROEQUIV(SIN(2\*X) - 2\*SIN(X)\*COS(X),X) returns TRUE and ZEROEQUIV(%E^X+X,X) returns FALSE. On the other hand ZEROEQUIV(LOG(A\*B) - LOG(A) - LOG(B),A) will return DONTKNOW because of the presence of an extra parameter. The restrictions are: (1) Do not use functions that MACSYMA does not know how to differentiate and evaluate. (2) If the expression has poles on the real line, there may be errors in the result (but this is unlikely to occur). (3) If the expression contains functions which are not solutions to first order differential equations (e.g. Bessel functions) there may be incorrect results. (4) The algorithm uses evaluation at randomly chosen points for carefully selected subexpressions. This is always a somewhat hazardous business, although the algorithm tries to minimize the potential for error.



## 5 Expressions

### 5.1 Introduction to Expressions

There are a number of reserved words which cannot be used as variable names. Their use would cause a possibly cryptic syntax error.

INTEGRATE	NEXT	FROM	DIFF
IN	AT	LIMIT	SUM
FOR	AND	ELSEIF	THEN
ELSE	DO	OR	IF
UNLESS	PRODUCT	WHILE	THRU
STEP			

Most things in MAXIMA are expressions. A sequence of expressions can be made into an expression by separating them by commas and putting parentheses around them. This is similar to the C *comma expression*.

```
(C29) x:3$
(D30) 16
(C31) joe:(if (x >17) then 2 else 4);
(D31) 4
(C32) joe:(if (x >17) then x:2 else joe:4,joe+x);
(D32) 20
```

Even loops in maxima are expressions, although the value they return is the not to useful  
DONE

```
(C33) joe:(x:1,for i from 1 thru 10 do (x:x*i));
(D33) DONE
```

whereas what you really want is probably is to include a third term in the *comma expression* which actually gives back the value.

```
(C34) joe:(x:1,for i from 1 thru 10 do (x:x*i),x);
(D34) 3628800
```

### 5.2 ASSIGNMENT

- There are two assignment operators in MACSYMA, `:` and `::`. E.g. `A:3` sets the variable A to 3. `::` assigns the value of the expression on its right to the value of the quantity on its left, which must evaluate to an atomic variable or subscripted variable.

### 5.3 COMPLEX

- A complex expression is specified in MACSYMA by adding the real part of the expression to `%I` times the imaginary part. Thus the roots of the equation  $X^2-4X+13=0$  are  $2+3\%I$  and  $2-3\%I$ . Note that simplification of products of complex expressions can be effected by expanding the product. Simplification of quotients, roots, and other functions of complex expressions can usually be accomplished by using the `REALPART`, `IMAGPART`, `RECTFORM`, `POLARFORM`, `ABS`, `CARG` functions.

## 5.4 INEQUALITY

- MACSYMA has the usual inequality operators: less than: < greater than: > greater than or equal to: >= less than or equal to: <=

## 5.5 SYNTAX

- It is possible to add new operators to MACSYMA (infix, prefix, postfix, unary, or matchfix with given precedences), to remove existing operators, or to redefine the precedence of existing operators. While MACSYMA's syntax should be adequate for most ordinary applications, it is possible to define new operators or eliminate predefined ones that get in the user's way. The extension mechanism is rather straightforward and should be evident from the examples below.

```
(C1) PREFIX("DDX")$
(C2) DDX Y$
      /* means                "DDX"(Y) */
(C3) INFIX("<-")$
(C4) A<-DDX Y$
      /* means                "<-"(A,"DDX"(Y)) */
```

For each of the types of operator except SPECIAL, there is a corresponding creation function that will give the lexeme specified the corresponding parsing properties. Thus "PREFIX("DDX")" will make "DDX" a prefix operator just like "-" or "NOT". Of course, certain extension functions require additional information such as the matching keyword for a matchfix operator. In addition, binding powers and parts of speech must be specified for all keywords defined. This is done by passing additional arguments to the extension functions. If a user does not specify these additional parameters, MACSYMA will assign default values. The six extension functions with binding powers and parts of speech defaults (enclosed in brackets) are summarized below. PREFIX(operator, rbp[180], rpos[ANY], pos[ANY]) POSTFIX(operator, lbp[180], lpos[ANY], pos[ANY]) INFIX(operator, lbp[180], rbp[180], lpos[ANY], rpos[ANY], pos[ANY]) NARY(operator, bp[180], argpos[ANY], pos[ANY]) NOFIX(operator, pos[ANY]) MATCHFIX(operator, match, argpos[ANY], pos[ANY]) The defaults have been provided so that a user who does not wish to concern himself with parts of speech or binding powers may simply omit those arguments to the extension functions. Thus the following are all equivalent. PREFIX("DDX",180,ANY,ANY)\$ PREFIX("DDX",180)\$ PREFIX("DDX")\$ It is also possible to remove the syntax properties of an operator by using the functions REMOVE or KILL. Specifically, "REMOVE("DDX",OP)" or "KILL("DDX")" will return "DDX" to operand status; but in the second case all the other properties of "DDX" will also be removed.

```
(C20) PREFIX("DDX",180,ANY,ANY)$
(C21) DDXYZ;
(D21)      DDX YZ
```

(C26) "ddx"(u) := u + 4;

(D26)                    DDX u := u + 4

(C27) ddx 8;

(D27)                    12

## 5.6 Definitions for Expressions

**AT** (*exp*, *list*) Function  
 will evaluate *exp* (which may be any expression) with the variables assuming the values as specified for them in the list of equations or the single equation similar to that given to the ATVALUE function. If a subexpression depends on any of the variables in *list* but it hasn't had an atvalue specified and it can't be evaluated then a noun form of the AT will be returned which will display in a two-dimensional form. Do EXAMPLE(AT); for an example.

**BOX** (*expr*) Function  
 returns *expr* enclosed in a box. The box is actually part of the expression.  
       BOX(*expr*, *label*)  
 encloses *expr* in a labelled box. *label* is a name which will be truncated in display if it is too long. BOXCHAR["] - is the character used to draw the box in this and in the DPART and LPART functions.

**BOXCHAR** Variable  
 default: ["] is the character used to draw the box in the BOX and in the DPART and LPART functions.

**CONSTANT** special operator  
 - makes *ai* a constant as is %PI.

**CONSTANTP** (*exp*) Function  
 is TRUE if *exp* is a constant (i.e. composed of numbers and %PI, %E, %I or any variables bound to a constant or DECLARED constant) else FALSE. Any function whose arguments are constant is also considered to be a constant.

**CONTRACT** (*exp*) Function  
 carries out all possible contractions in *exp*, which may be any well-formed combination of sums and products. This function uses the information given to the DEFCON function. Since all tensors are considered to be symmetric in all indices, the indices are sorted into alphabetical order. Also all dummy indices are renamed using the symbols !1,!2,... to permit the expression to be simplified as much as possible by reducing equivalent terms to a canonical form. For best results *exp* should be fully expanded. RATEXPAND is the fastest way to expand products and powers of sums if there are no variables in the denominators of the terms. The GCD switch should be FALSE if gcd cancellations are unnecessary.



**DECLARE** (*a1, f1, a2, f2, ...*) Function

gives the atom *ai* the flag *fi*. The *ai*'s and *fi*'s may also be lists of atoms and flags respectively in which case each of the atoms gets all of the properties. The possible flags and their meanings are:

CONSTANT - makes *ai* a constant as is %PI.

MAINVAR - makes *ai* a MAINVAR. The ordering scale for atoms: numbers < constants (e.g. %E,%PI) < scalars < other variables < mainvars.

SCALAR - makes *ai* a scalar.

NONSCALAR - makes *ai* behave as does a list or matrix with respect to the dot operator.

NOUN - makes the function *ai* a noun so that it won't be evaluated automatically.

EVFUN - makes *ai* known to the EV function so that it will get applied if its name is mentioned. Initial evfuns are

FACTOR, TRIGEXPAND,  
TRIGREDUCE, BFLOAT, RATSIMP, RATEXPAND, and RADCAN

EVFLAG - makes *ai* known to the EV function so that it will be bound to TRUE during the execution of EV if it is mentioned. Initial evflags are

FLOAT, PRED, SIMP, NUMER, DETOUT, EXPONENTIALIZE, DEMOIVRE,  
KEEPFLOAT, LISTARITH, TRIGEXPAND, SIMPSUM, ALGEBRAIC,  
RATALGDENOM, FACTORFLAG, %EMODE, LOGARC, LOGNUMBER,  
RADEXPAND, RATSIMPEXPONS, RATMX, RATFAC, INFEVAL, %ENUMER,  
PROGRAMMODE, LOGNEGINT, LOGABS, LETRAT, HALFANGLES,  
EXPTISOLATE, ISOLATE\_WRT\_TIMES, SUMEXPAND, CAUCHYSUM,  
NUMER\_PBRANCH, M1PBRANCH, DOTSCRULES, and LOGEXPAND

BINDTEST - causes *ai* to signal an error if it ever is used in a computation unbound.

DECLARE([*var1, var2, ...*], BINDTEST) causes MACSYMA to give an error message whenever any of the *vari* occur unbound in a computation. MACSYMA currently recognizes and uses the following features of objects:

EVEN, ODD, INTEGER, RATIONAL, IRRATIONAL, REAL, IMAGINARY,  
and COMPLEX

the useful features of functions include:

INCREASING,  
DECREASING, ODDFUN (odd function), EVENFUN (even function),  
COMMUTATIVE (or SYMMETRIC), ANTISYMMETRIC, LASSOCIATIVE and  
RASSOCIATIVE

DECLARE(F,INCREASING) is in all respects equivalent to

ASSUME(KIND(F, INCREASING))

The *ai* and *fi* may also be lists of objects or features. The command

FEATUREP(object,feature)

may be used to determine if an object has been DECLARED to have "feature". See DESCRIBE(FEATURES); .

**DISOLATE** (*exp, var1, var2, ..., varN*) Function  
 is similar to ISOLATE(*exp, var*) (Do DESCRIBE(ISOLATE);) except that it enables the user to isolate more than one variable simultaneously. This might be useful, for example, if one were attempting to change variables in a multiple integration, and that variable change involved two or more of the integration variables. To access this function, do LOAD(DISOL)\$ . A demo is available by DEMO("disol"); .

**DISPFORM** (*exp*) Function  
 returns the external representation of *exp* (wrt its main operator). This should be useful in conjunction with PART which also deals with the external representation. Suppose EXP is -A . Then the internal representation of EXP is "\*"(-1,A), while the external representation is "-"(A). DISPFORM(*exp,ALL*) converts the entire expression (not just the top-level) to external format. For example, if EXP:SIN(SQRT(X)), then FREEOF(SQRT,EXP) and FREEOF(SQRT,DISPFORM(EXP)) give TRUE, while FREEOF(SQRT,DISPFORM(EXP,ALL)) gives FALSE.

**DISTRIB** (*exp*) Function  
 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. For example,  
 DISTRIB((A+B)\*(C+D)) -> A C + A D + B C + B D MULTTHRU ((A+B)\*(C+D))  
 -> (A + B) C + (A + B) D DISTRIB (1/((A+B)\*(C+D))) -> 1/ ((A+B) \*(C+D))  
 EXPAND(1/((A+B)\*(C+D)),1,0) -> 1/(A C + A D + B C + B D)

**DPART** (*exp, n1, ..., nk*) Function  
 selects the same subexpression as PART, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. The box is actually part of the expression.

(C1) DPART(X+Y/Z\*\*2,1,2,1);

(D1) 
$$\frac{Y}{2} + X$$
 \*\*\*\*\*  
 \* Z \*  
 \*\*\*\*\*

**EXP** (*X*) Function  
 the exponential function. It is represented internally as %E^X. DEMOIVRE[FALSE] - if TRUE will cause %E^(A+B\*I) to become %E^A\*(COS(B)+%I\*SIN(B)) if B is free of %I. A and B are not expanded. %EMODE[TRUE] - when TRUE %E^(%PI\*I\*X) will be simplified as follows: it will become COS(%PI\*X)+%I\*SIN(%PI\*X) if X is an integer or a multiple of 1/2, 1/3, 1/4, or 1/6 and thus will simplify further. For other numerical X it will become %E^(%PI\*I\*Y) where Y is X-2\*k for some integer k such that ABS(Y)<1. If %EMODE is FALSE no simplification of %E^(%PI\*I\*X) will take place. %ENUMER[FALSE] - when TRUE will cause %E to be converted

into 2.718... whenever NUMER is TRUE. The default is that this conversion will take place only if the exponent in  $\%E^X$  evaluates to a number.

**EXPTISOLATE**

Variable

default: [FALSE] if TRUE will cause ISOLATE(expr,var); to examine exponents of atoms (like %E) which contain var.

**EXPTSUBST**

Variable

default: [FALSE] if TRUE permits substitutions such as Y for  $\%E^{**X}$  in  $\%E^{**}(A^*X)$  to take place.

**FREEOF** (*x1, x2, ..., exp*)

Function

yields TRUE if the xi do not occur in exp and FALSE otherwise. The xi are atoms or they may be subscripted names, functions (e.g. SIN(X) ), or operators enclosed in "s. If 'var' is a "dummy variable" of 'exp', then FREEOF(var,exp); will return TRUE. "Dummy variables" are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable. Example: FREEOF(I,'SUM(F(I),I,0,N)); returns TRUE. Do EXAMPLE(FREEOF); for more examples.

**GENFACT** (*X, Y, Z*)

Function

is the generalized factorial of X which is:  $X*(X-Z)*(X-2*Z)*...*(X-(Y-1)*Z)$ . Thus, for integral X, GENFACT(X,X,1)=X! and GENFACT(X,X/2,2)=X!!

**IMAGPART** (*exp*)

Function

returns the imaginary part of the expression exp.

**INDICES** (*exp*)

Function

returns a list of two elements. The first is a list of the free indices in exp (those that occur only once); the second is the list of dummy indices in exp (those that occur exactly twice).

**INFIX** (*op*)

Function

- INFIX operators are used to denote functions of two arguments, one given before the operator and one after, e.g.  $A^2$ . The INFIX("x") function is a syntax extension function to declare x to be an INFIX operator. Do DESCRIBE(SYNTAX); for more details.

**INFLAG**

Variable

default: [FALSE] if set to TRUE, the functions for part extraction will look at the internal form of exp. Note that the simplifier re-orders expressions. Thus FIRST(X+Y) will be X if INFLAG is TRUE and Y if INFLAG is FALSE. (FIRST(Y+X) gives the same results). Also, setting INFLAG to TRUE and calling PART/SUBSTPART is the same as calling INPART/SUBSTINPART. Functions affected by the setting of INFLAG are: PART, SUBSTPART, FIRST, REST, LAST, LENGTH, the FOR ... IN construct, MAP, FULLMAP, MAPLIST, REVEAL and PICKAPART.

**INPART** (*exp, n1, ..., nk*) Function

is similar to PART but works on the internal representation of the expression rather than the displayed form and thus may be faster since no formatting is done. Care should be taken with respect to the order of subexpressions in sums and products (since the order of variables in the internal form is often different from that in the displayed form) and in dealing with unary minus, subtraction, and division (since these operators are removed from the expression). PART(X+Y,0) or INPART(X+Y,0) yield +, though in order to refer to the operator it must be enclosed in "s. For example ...IF INPART(D9,0)="+" THEN ...

```
(C1) X+Y+W*Z;
(D1)           W Z + Y + X
(C2) INPART(D1,3,2);
(D2)           Z
(C3) PART(D1,1,2);
(D3)           Z
(C4) 'LIMIT(F(X)**G(X+1),X,0,MINUS);
(D4)           G(X + 1)
              LIMIT F(X)
              X ->0-
(C5) INPART(%,1,2);
(D5)           G(X + 1)
```

**ISOLATE** (*exp, var*) Function

returns exp with subexpressions which are sums and which do not contain var replaced by intermediate expression labels (these being atomic symbols like E1, E2, ...). This is often useful to avoid unnecessary expansion of subexpressions which don't contain the variable of interest. Since the intermediate labels are bound to the subexpressions they can all be substituted back by evaluating the expression in which they occur. EXPTISOLATE[FALSE] if TRUE will cause ISOLATE to examine exponents of atoms (like %E) which contain var. ISOLATE\_WRT\_TIMES[FALSE] if TRUE, then ISOLATE will also isolate wrt products. E.g. compare both settings of the switch on ISOLATE(EXPAND((A+B+C)^2),C); . Do EXAMPLE(ISOLATE); for examples.

**ISOLATE\_WRT\_TIMES** Variable

default: [FALSE] - if set to TRUE, then ISOLATE will also isolate wrt products. E.g. compare both settings of the switch on ISOLATE(EXPAND((A+B+C)^2),C); .

**LISTCONSTVARS** Variable

default: [FALSE] - if TRUE will cause LISTOFVARS to include %E, %PI, %I, and any variables declared constant in the list it returns if they appear in the expression LISTOFVARS is called on. The default is to omit these.

**LISTDUMMYVARS** Variable

default: [TRUE] - if FALSE, "dummy variables" in the expression will not be included in the list returned by LISTOFVARS. (The meaning of "dummy variables" is

as given in DESCRIBE(FREEOF): "Dummy variables" are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable.) Example: LISTOFVARS('SUM(F(I),I,0,N)); gives [I,N] if LISTDUMMYVARS is TRUE, and [N] if LISTDUMMYVARS is FALSE.

**LISTOFVARS** (*exp*) Function  
yields a list of the variables in *exp*. LISTCONSTVARS[FALSE] if TRUE will cause LISTOFVARS to include %E, %PI, %I, and any variables declared constant in the list it returns if they appear in *exp*. The default is to omit these.

(C1) LISTOFVARS(F(X[1]+Y)/G\*\*(2+A));  
(D1) [X[1], Y, A, G]

**LOPOW** (*exp, v*) Function  
the lowest exponent of *v* which explicitly appears in *exp*. Thus

LOPOW((X+Y)\*\*2+(X+Y)\*\*A,X+Y) ==> MIN(A,2)

**LPART** (*label, expr, n1, ..., nk*) Function  
is similar to DPART but uses a labelled box. A labelled box is similar to the one produced by DPART but it has a name in the top line.

**MULTTHRU** (*exp*) Function  
multiplies a factor (which should be a sum) of *exp* by the other factors of *exp*. That is *exp* is  $f_1 * f_2 * \dots * f_n$  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(*exp1, exp2*) multiplies each term in *exp2* (which should be a sum or an equation) by *exp1*. If *exp1* is not itself a sum then this form is equivalent to MULTTHRU(*exp1\*exp2*).

(C1) X/(X-Y)\*\*2-1/(X-Y)-F(X)/(X-Y)\*\*3;

(D1) 
$$-\frac{1}{X-Y} + \frac{X}{(X-Y)^2} - \frac{F(X)}{(X-Y)^3}$$

(C2) MULTTHRU((X-Y)\*\*3,%);

(D2) 
$$-(X-Y)^2 + X(X-Y) - F(X)$$

(C3) RATEXPAND(D2);

(D3) 
$$-Y^2 + XY - F(X)$$

(C4) ((A+B)\*\*10\*S\*\*2+2\*A\*B\*S+(A\*B)\*\*2)/(A\*B\*S\*\*2);  
(D3) 
$$\frac{(B+A)^{10}S^2 + 2AB S + A^2 B^2}{A^2 B^2 S^2}$$

(D4) 
$$\frac{A^2 B^2 S}{A^2 B^2 S}$$

(C5) `MULTTHRU(%);`

(D5) 
$$\frac{2 A B (B + A)^{10}}{S^2 A B}$$

(notice that  $(B+A)**10$  is not expanded)

(C6) `MULTTHRU(A.(B+C.(D+E)+F));`

(D6) 
$$A . F + A . (C . (E + D)) + A . B$$

(compare with similar example under `EXPAND`)

**NOUNIFY** (*f*) Function

returns the noun form of the function name *f*. This is needed if one wishes to refer to the name of a verb function as if it were a noun. Note that some verb functions will return their noun forms if they can't be evaluated for certain arguments. This is also the form returned if a function call is preceded by a quote.

**NTERMS** (*exp*) Function

gives the number of terms that *exp* would have if it were fully expanded out and no cancellations or combination of terms occurred. Note that expressions like `SIN(E)`, `SQRT(E)`, `EXP(E)`, etc. count as just one term regardless of how many terms *E* has (if it is a sum).

**OPTIMIZE** (*exp*) Function

returns an expression that produces the same value and side effects as *exp* but does so more efficiently by avoiding the recomputation of common subexpressions. `OPTIMIZE` also has the side effect of "collapsing" its argument so that all common subexpressions are shared. Do `EXAMPLE(OPTIMIZE);` for examples.

**OPTIMPREFIX** Variable

default: [%] - The prefix used for generated symbols by the `OPTIMIZE` command.

**ORDERGREAT** (*V1, ..., Vn*) Function

sets up aliases for the variables *V1, ..., Vn* such that  $V1 > V2 > \dots > Vn >$  any other variable not mentioned as an argument. See also `ORDERLESS`. Caveat: do `EXAMPLE(ORDERGREAT);` for some specifics.

**ORDERGREATP** (*exp1,exp2*) Function

returns `TRUE` if *exp2* precedes *exp1* in the ordering set up with the `ORDERGREAT` function (see `DESCRIBE(ORDERGREAT);`).

**ORDERLESS** ( $V_1, \dots, V_n$ ) Function

sets up aliases for the variables  $V_1, \dots, V_n$  such that  $V_1 < V_2 < \dots < V_n <$  any other variable not mentioned as an argument. Thus the complete ordering scale is: numerical constants  $<$  declared constants  $<$  declared scalars  $<$  first argument to ORDERLESS  $< \dots <$  last argument to ORDERLESS  $<$  variables which begin with A  $< \dots <$  variables which begin with Z  $<$  last argument to ORDERGREAT  $< \dots <$  first argument to ORDERGREAT  $<$  declared MAINVARs. Caveat: do EXAMPLE(ORDERLESS); for some specifics. For another ordering scheme, see DESCRIBE(MAINVAR);.

**ORDERLESSP** ( $exp_1, exp_2$ ) Function

returns TRUE if  $exp_1$  precedes  $exp_2$  in the ordering set up by the ORDERLESS command (see DESCRIBE(ORDERLESS);).

**PART** ( $exp, n_1, \dots, n_k$ ) Function

deals with the displayed form of  $exp$ . It obtains the part of  $exp$  as specified by the indices  $n_1, \dots, n_k$ . First part  $n_1$  of  $exp$  is obtained, then part  $n_2$  of that, etc. The result is part  $n_k$  of  $\dots$  part  $n_2$  of part  $n_1$  of  $exp$ . Thus PART( $Z+2*Y, 2, 1$ ) yields 2. PART can be used to obtain an element of a list, a row of a matrix, etc. If the last argument to a Part function is a list of indices then several subexpressions are picked out, each one corresponding to an index of the list. Thus PART( $X+Y+Z, [1, 3]$ ) is  $Z+X$ . PIECE holds the last expression selected when using the Part functions. It is set during the execution of the function and thus may be referred to in the function itself as shown below. If PARTSWITCH[FALSE] is set to TRUE then END is returned when a selected part of an expression doesn't exist, otherwise an error message is given. For examples, do EXAMPLE(PART);

**PARTITION** ( $exp, var$ ) Function

returns a list of two expressions. They are (1) the factors of  $exp$  (if it is a product), the terms of  $exp$  (if it is a sum), or the list (if it is a list) which don't contain  $var$  and, (2) the factors, terms, or list which do.

```
(C1) PARTITION(2*A*X*F(X), X);
(D1)          [ 2 A , X F(X) ]
(C2) PARTITION(A+B, X);
(D2)          [ A + B , 0 ]
(C3) PARTITION([A, B, F(A), C], A);
(D3)          [[B, C], [A, F(A)]]
```

**PARTSWITCH** Variable

default: [FALSE] - if set to TRUE then END is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

**PICKAPART** ( $exp, depth$ ) Function

will assign E labels to all subexpressions of  $exp$  down to the specified integer depth. This is useful for dealing with large expressions and for automatically assigning parts of an expression to a variable without having to use the part functions.

(C1) EXP: (A+B)/2+SIN(X^2)/3-LOG(1+SQRT(X+1));

(D1) 
$$- \text{LOG}(\text{SQRT}(X + 1) + 1) + \frac{\text{SIN}(X^2)}{3} + \frac{B + A}{2}$$

(C2) PICKAPART(% , 1);

(E2) 
$$- \text{LOG}(\text{SQRT}(X + 1) + 1)$$

(E3) 
$$\frac{\text{SIN}(X^2)}{3}$$

(E4) 
$$\frac{B + A}{2}$$

(D4) 
$$E4 + E3 + E2$$

## PIECE

Variable

- holds the last expression selected when using the Part functions. It is set during the execution of the function and thus may be referred to in the function itself.

## POWERS (expr, var)

Function

gives the powers of var occurring in expr. To use it, do LOAD(POWERS);. For details on usage, do PRINTFILE("powers.usg");.

## PRODUCT (exp, ind, lo, hi)

Function

gives the product of the values of exp as the index ind varies from lo to hi. The evaluation is similar to that of SUM. No simplification of products is available at this time. If hi is one less than lo, we have an "empty product" and PRODUCT returns 1 rather than erring out. Also see DESCRIBE(PRODHACK).

(C1) PRODUCT(X+I\*(I+1)/2,I,1,4);

(D1)  $(X + 1) (X + 3) (X + 6) (X + 10)$

## REALPART (exp)

Function

gives the real part of exp. REALPART and IMAGPART will work on expressions involving trigonometric and hyperbolic functions, as well as SQRT, LOG, and exponentiation.

## RECTFORM (exp)

Function

returns an expression of the form  $A + B*\%I$ , where A and B are purely real.

## REMBOX (expr, arg)

Function

removes boxes from expr according to arg. If arg is UNLABELED then all unlabelled boxes are removed. If arg is the name of some label then only boxes with that label are removed. If arg is omitted then all boxes labelled and unlabelled are removed.



**SUM** (*exp, ind, lo, hi*) Function

performs a summation of the values of *exp* as the index *ind* varies from *lo* to *hi*. If the upper and lower limits differ by an integer then each term in the sum is evaluated and added together. Otherwise, if the SIMPSUM [FALSE] is TRUE the result is simplified. This simplification may sometimes be able to produce a closed form. If SIMPSUM is FALSE or if 'SUM is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics. If *hi* is one less than *lo*, we have an "empty sum" and SUM returns 0 rather than erring out. Sums may be differentiated, added, subtracted, or multiplied with some automatic simplification being performed. Also see DESCRIBE(SUMHACK). CAUCHYSUM[FALSE] when TRUE causes the Cauchy product to be used when multiplying sums together rather than the usual product. In the Cauchy product the index of the inner summation is a function of the index of the outer one rather than varying independently. GENINDEX[I] is the alphabetic prefix used to generate the next variable of summation. GENSUMNUM[0] is the numeric suffix used to generate the next variable of summation. If it is set to FALSE then the index will consist only of GENINDEX with no numeric suffix. Do EXAMPLE(SUM); for examples. See also SUMCONTRACT, INTOSUM, BASHINDICES, and NICEINDICES.

**LSUM** (*exp, ind, list*) Function

performs the sum of EXP for each element IND of the LIST.

```
(C10) lsum(x^i,i,[1,2,7]);
```

```
(D10)      7  2
          x  + x  + x
```

If the last element LIST argument does not evaluate, or does not evaluate to a Maxima list then the answer is left in noun form

```
(C13) lsum(i^2,i,rootsof(x^3-1));
```

```
====
\      2
(D13)  >  i
/
====
3
i in ROOTSOF(x  - 1)
```

**VERB** special symbol

- the opposite of "noun", i.e. a function form which "does something" ("action" - for most functions the usual case). E.g. INTEGRATE integrates a function, unless it is DECLARED to be a "noun", in which case it represents the INTEGRAL of the function. See NOUN, NOUNIFY, and VERBIFY.

**VERBIFY** (*f*) Function

returns the function name *f* in its verb form (See also VERB, NOUN, and NOUNIFY).

## 6 Simplification

### 6.1 Definitions for Simplification

**APPLY\_NOUNS** (*exp*) Function  
 causes the application of noun forms in an expression. E.g. `EXP:'DIFF(X^2/2,X);`  
`APPLY_NOUNS(EXP);` gives X. This gives the same result as `EV(EXP,NOUNS);`  
 except that it can be faster and use less storage. It also can be used in translated  
 code, where `EV` may cause problems. Note that it is called `APPLY_NOUNS`, not  
`EV_NOUNS`, because what it does is to `APPLY` the rules corresponding to the noun-  
 form operators, which is not evaluation.

**ASKEXP** Variable  
 default: `[]` contains the expression upon which `ASKSIGN` is called. A user may enter  
 a MACSYMA break with `^A` and inspect this expression in order to answer questions  
 asked by `ASKSIGN`.

**ASKINTEGER** (*exp*,<*optional-arg*>) Function  
*exp* is any valid macsyms expression and *optional-arg* is `EVEN`, `ODD`, `INTEGER` and  
 defaults to `INTEGER` if not given. This function attempts to determine from the  
 data-base whether *exp* is `EVEN`, `ODD` or just an `INTEGER`. It will ask the user  
 if it cannot tell otherwise and attempt to install the information in the data-base if  
 possible.

**ASKSIGN** (*exp*) Function  
 first attempts to determine whether the specified expression is positive, negative, or  
 zero. If it cannot, it asks the user the necessary questions to complete its deduc-  
 tion. The user's answer is recorded in the data base for the duration of the current  
 computation (one "C-line"). The value of `ASKSIGN` is one of `POS`, `NEG`, or `ZERO`.

**DEMOIVRE** Variable  
 default: `[FALSE]` if `TRUE` will cause  

$$\%E^{(A+B*\%I)} ==> \%E^A * (\%COS(B) + \%I * \%SIN(B))$$
 if B is free of `%I`. A and B are not expanded. `DEMOIVRE:TRUE;` is the way to  
 reverse the effect of `EXPONENTIALIZE:TRUE;`  
`DEMOIVRE(exp)` will cause the conversion without setting the switch or having to  
 re-evaluate the expression with `EV`.

**DOMAIN** Variable  
 default: `[REAL]` - if set to `COMPLEX`, `SQRT(X^2)` will remain `SQRT(X^2)` instead  
 of returning `ABS(X)`. The notion of a "domain" of simplification is still in its infancy,  
 and controls little more than this at the moment.

**EXPAND** (*exp*) Function

will cause products of sums and exponentiated sums to be multiplied out, numerators of rational expressions which are sums to be split into their respective terms, and multiplication (commutative and non-commutative) to be distributed over addition at all levels of *exp*. For polynomials one should usually use RATEXPAND which uses a more efficient algorithm (see DESCRIBE(RATEXPAND);). MAXNEGEX[1000] and MAXPOSEX[1000] control the maximum negative and positive exponents, respectively, which will expand. EXPAND(*exp*,*p*,*n*) expands *exp*, using *p* for MAXPOSEX and *n* for MAXNEGEX. This is useful in order to expand part but not all of an expression. EXPON[0] - the exponent of the largest negative power which is automatically expanded (independent of calls to EXPAND). For example if EXPON is 4 then  $(X+1)**(-5)$  will not be automatically expanded. EXPOP[0] - the highest positive exponent which is automatically expanded. Thus  $(X+1)**3$ , when typed, will be automatically expanded only if EXPOP is greater than or equal to 3. If it is desired to have  $(X+1)**N$  expanded where *N* is greater than EXPOP then executing EXPAND( $(X+1)**N$ ) will work only if MAXPOSEX is not less than *N*. The EXPAND flag used with EV (see EV) causes expansion. The file SHARE1;FACEXP FASL contains several related functions (FACSUM and COLLECTTERMS are two) that provide the user with the ability to structure expressions by controlled expansion. Brief function descriptions are available in SHARE1;FACEXP USAGE. A demo is available by doing BATCH("facexp.mc")\$ .

**EXPANDWRT** (*exp*,*var1*,*var2*,...) Function

expands *exp* with respect to the *vari*. All products involving the *vari* appear explicitly. The form returned will be free of products of sums of expressions that are not free of the *vari*. The *vari* may be variables, operators, or expressions. By default, denominators are not expanded, but this can be controlled by means of the switch EXPANDWRT\_DENOM. Do LOAD(STOPEX); to use this function.

**EXPANDWRT\_DENOM** Variable

default:[FALSE] controls the treatment of rational expressions by EXPANDWRT. If TRUE, then both the numerator and denominator of the expression will be expanded according to the arguments of EXPANDWRT, but if EXPANDWRT\_DENOM is FALSE, then only the numerator will be expanded in that way. Do LOAD(STOPEX) to use.

**EXPANDWRT\_FACTORED** (*exp*, *var1*, *var2*, ..., *varN*) Function

is similar to EXPANDWRT, but treats expressions that are products somewhat differently. EXPANDWRT\_FACTORED will perform the required expansion only on those factors of *exp* that contain the variables in its argument list. Do LOAD(STOPEX) to use this function.

**EXPON** Variable

default: [0] - the exponent of the largest negative power which is automatically expanded (independent of calls to EXPAND). For example if EXPON is 4 then  $(X+1)**(-5)$  will not be automatically expanded.

**EXPONENTIALIZE**

Variable

default: [FALSE] if TRUE will cause all circular and hyperbolic functions to be converted to exponential form. (Setting DEMOIVRE:TRUE; will reverse the effect.) EXPONENTIALIZE(exp) will cause the conversion to exponential form of an expression without setting the switch or having to re-evaluate the expression with EV.

**EXPOP**

Variable

default: [0] - the highest positive exponent which is automatically expanded. Thus  $(X+1)^{**3}$ , when typed, will be automatically expanded only if EXPOP is greater than or equal to 3. If it is desired to have  $(X+1)^{**n}$  expanded where n is greater than EXPOP then executing EXPAND( $(X+1)^{**n}$ ) will work only if MAXPOSEX is not less than n.

**FACTLIM**

Variable

default: [-1] gives the highest factorial which is automatically expanded. If it is -1 then all integers are expanded.

**INTOSUM** (*expr*)

Function

will take all things that a summation is multiplied by, and put them inside the summation. If the index is used in the outside expression, then the function tries to find a reasonable index, the same as it does for SUMCONTRACT. This is essentially the reverse idea of the OUTATIVE property of summations, but note that it does not remove this property, it only bypasses it. In some cases, a SCANMAP(MULTTHRU,expr) may be necessary before the INTOSUM.

**LASSOCIATIVE**

declaration

- If DECLARE(G,LASSOCIATIVE); is done, this tells the simplifier that G is left-associative. E.g.  $G(G(A,B),G(C,D))$  will simplify to  $G(G(G(A,B),C),D)$ .

**LINEAR**

declaration

- One of MACSYMA's OPPROPERTIES. For univariate f so declared, "expansion"  $F(X+Y) \rightarrow F(X)+F(Y)$ ,  $F(A*X) \rightarrow A*F(X)$  takes place where A is a "constant". For functions F of  $\geq 2$  args, "linearity" is defined to be as in the case of 'SUM or 'INTEGRATE, i.e.  $F(A*X+B,X) \rightarrow A*F(X,X)+B*F(1,X)$  for A,B FREEOF X. (LINEAR is just ADDITIVE + OUTATIVE.)

**MAINVAR**

declaration

- 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( $(X+Y)^4$ ); . (Note: Care should be taken if you elect to use the above feature. E.g. if you subtract an expression in which X is a MAINVAR from one in which X isn't a MAINVAR, resimplification e.g. with EV(expression,SIMP) may be necessary if cancellation is to occur. Also, if you SAVE an expression in which X is a MAINVAR, you probably should also SAVE X.)

**MAXAPPLYDEPTH** Variable  
 default: [10000] - the maximum depth to which APPLY1 and APPLY2 will delve.

**MAXAPPLYHEIGHT** Variable  
 default: [10000] - the maximum height to which APPLYB1 will reach before giving up.

**MAXNEGEX** Variable  
 default: [1000] - the largest negative exponent which will be expanded by the EXPAND command (see also MAXPOSEX).

**MAXPOSEX** Variable  
 default: [1000] - the largest exponent which will be expanded with the EXPAND command (see also MAXNEGEX).

**MULTIPLICATIVE** declaration  
 - If DECLARE(F,MULTIPLICATIVE) has been executed, then: (1) If F is univariate, whenever the simplifier encounters F applied to a product, F will be distributed over that product. I.e.  $F(X*Y)$ ; will simplify to  $F(X)*F(Y)$ . (2) If F is a function of 2 or more arguments, multiplicativity is defined as multiplicativity in the first argument to F, i.e.  $F(G(X)*H(X),X)$ ; will simplify to  $F(G(X),X)*F(H(X),X)$ . This simplification does not occur when F is applied to expressions of the form  $PRODUCT(X[I],I,lower-limit,upper-limit)$ .

**NEGDISTRIB** Variable  
 default: [TRUE] - when TRUE allows -1 to be distributed over an expression. E.g.  $-(X+Y)$  becomes  $-Y-X$ . Setting it to FALSE will allow  $-(X+Y)$  to be displayed like that. This is sometimes useful but be very careful: like the SIMP flag, this is one flag you do not want to set to FALSE as a matter of course or necessarily for other than local use in your MACSYMA.

**NEGSUMDISPFLAG** Variable  
 default: [TRUE] - when TRUE,  $X-Y$  displays as  $X-Y$  instead of as  $-Y+X$ . Setting it to FALSE causes the special check in display for the difference of two expressions to not be done. One application is that thus  $A+%I*B$  and  $A-%I*B$  may both be displayed the same way.

**NOEVAL** special symbol  
 - suppresses the evaluation phase of EV. This is useful in conjunction with other switches and in causing expressions to be resimplified without being reevaluated.

**NOUN** declaration  
 - One of the options of the DECLARE command. It makes a function so DECLARED a "noun", meaning that it won't be evaluated automatically.

- NOUNDISP** Variable  
 default: [FALSE] - if TRUE will cause NOUNs to display with a single quote. This switch is always TRUE when displaying function definitions.
- NOUNS** special symbol  
 (EVFLAG) when used as an option to the EV command, converts all "noun" forms occurring in the expression being EV'd to "verbs", i.e. evaluates them. See also NOUN, NOUNIFY, VERB, and VERBIFY.
- NUMER** special symbol  
 causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in exp which have been given numerals to be replaced by their values. It also sets the FLOAT switch on.
- NUMERVAL** (*var1, exp1, var2, exp2, ...*) Function  
 declares vari to have a numeral of expi which is evaluated and substituted for the variable in any expressions in which the variable occurs if the NUMER flag is TRUE. (see the EV function).
- OPPROPERTIES** Variable  
 - the list of the special operator-properties handled by the MACSYMA simplifier: LINEAR, ADDITIVE, MULTIPLICATIVE, OUTATIVE, EVENFUN, ODDFUN, COMMUTATIVE, SYMMETRIC, ANTISYMMETRIC, NARY, LASSOCIATIVE, and RASSOCIATIVE.
- OPSUBST** Variable  
 default:[TRUE] - if FALSE, SUBST will not attempt to substitute into the operator of an expression. E.g. (OPSUBST:FALSE, SUBST(X<sup>2</sup>,R,R+R[0])); will work.
- OUTATIVE** declaration  
 - If DECLARE(F,OUTATIVE) has been executed, then: (1) If F is univariate, whenever the simplifier encounters F applied to a product, that product will be partitioned into factors that are constant and factors that are not and the constant factors will be pulled out. I.e. F(A\*X); will simplify to A\*F(X) where A is a constant. Non-atomic constant factors will not be pulled out. (2) If F is a function of 2 or more arguments, outativity is defined as in the case of 'SUM or 'INTEGRATE, i.e. F(A\*G(X),X); will simplify to A\*F(G(X),X) for A free-of X. Initially, 'SUM, 'INTEGRATE, and 'LIMIT are declared to be OUTATIVE.
- POSFUN** declaration  
 - POSitive FUNction, e.g. DECLARE(F,POSFUN); IS(F(X)>0); -> TRUE.
- PRODHACK** Variable  
 default: [FALSE] - if set to TRUE then PRODUCT(F(I),I,3,1); will yield 1/F(2), by the identity PRODUCT(F(I),I,A,B) = 1/PRODUCT(F(I),I,B+1,A-1) when A>B.

**RADCAN** (*exp*) Function

simplifies *exp*, 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 will not necessarily have the same appearance, but their difference will be simplified by RADCAN to zero. For some expressions RADCAN can be 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. %E\_TO\_NUMLOG (default: [FALSE]) - when set to TRUE, for "r" some rational number, and "x" some expression, %E<sup>(r\*LOG(x))</sup> will be simplified into x<sup>r</sup>. RADEXPAND[TRUE] when set to FALSE will inhibit certain transformations: RADCAN(SQRT(1-X)) will remain SQRT(1-X) and will not become %I SQRT(X-1). RADCAN(SQRT(X<sup>2</sup>-2\*X+1)) will remain SQRT(X<sup>2</sup>-2\*X + 1) and will not be transformed to X- 1. Do EXAMPLE(RADCAN); for examples.

**RADEXPAND** Variable

default: [TRUE] - if set to ALL will cause 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<sup>2</sup>) will become 4\*X. More particularly, consider SQRT(X<sup>2</sup>). (a) If RADEXPAND is ALL or ASSUME(X>0) has been done, SQRT(X<sup>2</sup>) will become X. (b) If RADEXPAND is TRUE and DOMAIN is REAL (its default), SQRT(X<sup>2</sup>) will become ABS(X). (c) If RADEXPAND is FALSE, or RADEXPAND is TRUE and DOMAIN is COMPLEX, SQRT(X<sup>2</sup>) will be returned. (The notion of DOMAIN with settings of REAL or COMPLEX is still in its infancy. Note that its setting here only matters when RADEXPAND is TRUE.)

**RADPRODEXPAND** Variable

- this switch has been renamed RADEXPAND.

**RADSUBSTFLAG** Variable

default: [FALSE] - if TRUE permits RATSUBST to make substitutions such as U for SQRT(X) in X.

**RASSOCIATIVE** declaration

- If DECLARE(G,RASSOCIATIVE); is done, this tells the simplifier that G is right-associative. E.g. G(G(A,B),G(C,D)) will simplify to G(A,G(B,G(C,D))).

**SCSIMP** (*exp,rule1, rule2,...,rulen*) Function

Sequential Comparative Simplification [Stoute]) takes an expression (its first argument) and a set of identities, or rules (its other arguments) and tries simplifying. If a smaller expression is obtained, the process repeats. Otherwise after all simplifications are tried, it returns the original answer. For examples, try EXAMPLE(SCSIMP); .

**SIMP** Function

causes *exp* to be simplified regardless of the setting of the switch SIMP which inhibits simplification if FALSE.

**SIMPSUM**

Variable

default: [FALSE] - if TRUE, the result of a SUM is simplified. This simplification may sometimes be able to produce a closed form. If SIMPSUM is FALSE or if 'SUM is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics.

**SUMCONTRACT** (*expr*)

Function

will combine all sums of an addition that have upper and lower bounds that differ by constants. The result will be an expression containing one summation for each set of such summations added to all appropriate extra terms that had to be extracted to form this sum. SUMCONTRACT will combine all compatible sums and use one of the indices from one of the sums if it can, and then try to form a reasonable index if it cannot use any supplied. It may be necessary to do an INTOSUM(*expr*) before the SUMCONTRACT.

**SUMEXPAND**

Variable

default: [FALSE] if TRUE, products of sums and exponentiated sums are converted into nested sums. For example:

```
SUMEXPAND:TRUE$
SUM(F(I),I,0,M)*SUM(G(J),J,0,N); ->
'SUM('SUM(F(I1)*G(I2),I2,0,N),I1,0,M)
SUM(F(I),I,0,M)^2; -> 'SUM('SUM(F(I3)*F(I4),I4,0,M),I3,0,M)
```

If FALSE, they are left alone. See also CAUCHYSUM.

**SUMHACK**

Variable

default: [FALSE] - if set to TRUE then SUM(F(I),I,3,1); will yield -F(2), by the identity  $\text{SUM}(F(I),I,A,B) = -\text{SUM}(F(I),I,B+1,A-1)$  when  $A > B$ .

**SUMSPLITFACT**

Variable

default: [TRUE] - if set to FALSE will cause MINFACTORIAL to be applied after a FACTCOMB.

**SYMMETRIC**

declaration

- If DECLARE(H,SYMMETRIC); is done, this tells the simplifier that H is a symmetric function. E.g.  $H(X,Z,Y)$  will simplify to  $H(X, Y, Z)$ . This is the same as COMMUTATIVE.

**UNKNOWN** (*exp*)

Function

returns TRUE iff *exp* contains an operator or function not known to the built-in simplifier.





## 7 Plotting

### 7.1 Definitions for Plotting

**IN\_NETMATH** [FALSE] Variable  
 If not nil, then plot2d will output a representation of the plot which is suitable for openplot functions.

**OPENPLOT\_CURVES** *list rest-options* Function  
 Takes a list of curves such as  

$$[[x_1, y_1, x_2, y_2, \dots], [u_1, v_1, u_2, v_2, \dots], \dots]$$
 or  

$$[[[x_1, y_1], [x_2, y_2], \dots], \dots]$$

and plots them. This is similar to xgraph\_curves, but uses the open plot routines. Additional symbol arguments may be given such as "{xrange -3 4}" The following plots two curves, using big points, labeling the first one jim and the second one jane.

```
openplot_curves(["{plotpoints 1} {pointsize 6} {label jim}
  {text {xlabel {joe is nice}}}" ,
  [1,2,3,4,5,6,7,8],
  "{label jane} {color pink } "], [3,1,4,2,5,7]);
```

Some other special keywords are xfun, color, plotpoints, linecolors, pointsize, nolines, bargraph, labelposition, xlabel, and ylabel.

**PLOT2D** (*expr,range,....,options,..*) Function

**PLOT2D** (*[expr1,expr2,..,exprn],xrange,....,options,..*) Function

EXPR is an expression to be plotted on y axis as a function of 1 variable. RANGE is of the form [var,min,max] and expr is assumed to be an expression to be plotted against VAR. In the second form of the function a list of expressions may be given to plot against VAR. Truncation in the y direction will be performed, for the default y range. It may be specified as an option or using SET\_PLOT\_OPTION.

```
plot2d(sin(x), [x, -5, 5]);
plot2d(sec(x), [x, -2, 2], [y, -20, 20], [nticks, 200]);
```

**xgraph\_curves(list)** Function

graphs the list of 'point sets' given in list by using xgraph.

A point set may be of the form

```
[x0, y0, x1, y1, x2, y2, ...] or
[[x0, y0], [x1, y1], ...]
```

A point set may also contain symbols which give labels or other information.

```
xgraph_curves([pt_set1, pt_set2, pt_set3]);
```

would graph the three point sets as three curves.

```
pt_set:append(["NoLines: True","LargePixels: true"],
             [x0,y0,x1,y1,...])
```

would make the point set [and subsequent ones], have no lines between points, and to use large pixels. See the man page on xgraph for more options to specify.

```
pt_set:append([concat("\", "x^2+y"), [x0,y0,x1,y1,...])
```

would make there be a "label" of "x^2+y" for this particular point set. The " at the beginning is what tells xgraph this is a label.

```
pt_set:append([concat("TitleText: Sample Data")], [x0,...])
```

would make the main title of the plot be "Sample Data" instead of "Maxima PLOT".

To make a bar graph with bars which are .2 units wide, and to plot two possibly different such bar graphs:

```
xgraph_curves(
  [append(["BarGraph: true","NoLines: true","BarWidth: .2"],
         create_list([i-.2,i^2],i,1,3)),
    append(["BarGraph: true","NoLines: true","BarWidth: .2"],
         create_list([i+.2,.7*i^2],i,1,3))
  ]);
```

A temporary file 'xgraph-out' is used.

## PLOT\_OPTIONS

Variable

Members of this list indicate defaults for plotting. They may be altered using SET\_PLOT\_OPTION

```
[X, - 3, 3]
[Y, - 3, 3]
```

are the x range and y range respectively.

[TRANSFORM\_XY, FALSE] if not false, should be the output of

```
make_transform([x,y,z], [f1(x,y,z),f2(x,y,z),f3(x,y,z)])
```

which produces a transformation from 3 space to 3 space, which will be applied to the graph. A built in one is polar\_xy which gives the same as

```
make_transform([r,th,z], [r*cos(th),r*sin(th),z])
```

[RUN\_VIEWER,TRUE] if not false, means run the viewer software - don't just output a data file.

[GRID,30,30] means plot3d should divide the x range into 30 intervals and similarly the y range.

[COLOUR\_Z,false] applies to colouring done with plot\_format ps.

[PLOT\_FORMAT,OPENMATH] is for plot3d and currently OPENMATH, GNUPLOT, PS, and GEOMVIEW are supported.

There are good quality public domain viewers for these formats. They are openmath, izic, gnuplot, ghostview, and geomview.

The Openmath viewer is in the distribution, and is based on tcl/tk. The executable is 'maxima/bin/omplotdata'. The viewer lets you zoom in, slide around, and rotate

(if 3 dimensional). This format is also the one used by netmath, for making plots with Netmath. (see '<http://www.ma.utexas.edu/users/wfs/netmath.html>')

geomview is from the Geometry Center at the University of Minnesota, and is available from '<http://www.geom.umn.edu/software/download/geomview.html>' or by anonymous ftp from '<ftp://ftp.geom.umn.edu/pub/software/geomview/>'. It is currently not quite as pretty as izic, but provides excellent support for multiple objects and multiple lights.

gnuplot is everywhere as is ghostview. We also provide mgnuplot, a tcl interface for gnuplot, which lets you rotate the plots using the mouse and a scale.

izic is available by ftp from zenon.inria.fr. Contact one of {fournier,kajler,mourrain}@sophia.inria.fr.

It has beautiful colour gouraud shading, and very fast wireframe. It runs on X windows.

**PLOT3D** (*expr,xrange,yrange,...,options,..*) Function

**PLOT3D** (*[expr1,expr2,expr3],xrange,yrange,...,options,..*) Function

```
plot3d(2^(-u^2+v^2), [u,-5,5], [v,-7,7]);
```

would plot  $z = 2^{-(u^2+v^2)}$  with  $u$  and  $v$  varying in  $[-5,5]$  and  $[-7,7]$  respectively, and with  $u$  on the  $x$  axis, and  $v$  on the  $y$  axis.

An example of the second pattern of arguments is

```
plot3d([cos(x)*(3+y*cos(x/2)), sin(x)*(3+y*cos(x/2)), y*sin(x/2)],
       [x,-%pi,%pi], [y,-1,1], ['grid,50,15])
```

which will plot a moebius band, parametrized by the 3 expressions given as the first argument to plot3d. An additional optional argument [grid,50,15] gives the grid number of rectangles in the  $x$  direction and  $y$  direction.

```
/* REal part of z ^ 1/3 */
plot3d(r^.33*cos(th/3), [r,0,1], [th,0,6*%pi],
       ['grid,12,80], ['PLOT_FORMAT,ps],
       ['TRANSFORM_XY,POLAR_TO_XY], ['VIEW_DIRECTION,1,1,1.4],
       ['COLOUR_Z,true])
```

Here the View\_direction indicates the direction from which we take a projection. We actually do this from infinitely far away, but parallel to the line from view\_direction to the origin. This is currently only used in 'ps' plot\_format, since the other viewers allow interactive rotating of the object.

Another example is a moebius band:

```
plot3d([cos(x)*(3+y*cos(x/2)),
       sin(x)*(3+y*cos(x/2)), y*sin(x/2)],
       [x,-%pi,%pi], [y,-1,1], ['grid,50,15]);
```

or a klein bottle:

```
plot3d([5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0) - 10.0,
       -5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0),
       5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))],
       [x,-%pi,%pi], [y,-%pi,%pi], ['grid,40,40])
```

or a torus

```
plot3d([cos(y)*(10.0+6*cos(x)),
       sin(y)*(10.0+6*cos(x)),
       -6*sin(x)], [x,0,2*%pi],[y,0,2*%pi],
       ['grid,40,40])
```

We can output to gnuplot too:

```
plot3d(2^(x^2-y^2), [x,-1,1], [y,-2,2], [plot_format,gnuplot])
```

Sometimes you may need to define a function to plot the expression. All the arguments to plot3d are evaluated before being passed to plot3d, and so trying to make an expression which does just what you want may be difficult, and it is just easier to make a function.

```
M:MATRIX([1,2,3,4],[1,2,3,2],[1,2,3,4],[1,2,3,3])$
f(x,y):=float(M[?round(x),?round(y)]);
plot3d(f,[x,1,4],[y,1,4],['grid,4,4]);
```

**PLOT2D\_PS** (*expr,range*) Function  
 writes to pstream a sequence of postscript commands which plot EXPR for RANGE. EXPR should be an expression of 1 variable. RANGE should be of the form [variable,min,max] over which to plot expr. see CLOSEPS.

**CLOSEPS** () Function  
 This should usually be called at the end of a sequence of plotting commands. It closes the current output stream PSTREAM, and sets it to nil. It also may be called at the start of a plot, to ensure pstream is closed if it was open. All commands which write to pstream, open it if necessary. CLOSEPS is separate from the other plotting commands, since we may want to plot 2 ranges or superimpose several plots, and so must keep the stream open.

**SET\_PLOT\_OPTION** (*option*) Function  
 option is of the format of one of the elements of the PLOT\_OPTIONS list. Thus  

```
SET_PLOT_OPTION(['grid,30,40])
```

 would change the default grid used by plot3d. Note that if the symbol grid has a value, then you should quote it here:  

```
SET_PLOT_OPTION(['grid,30,40])
```

 so that the value will not be substituted.

**PSDRAW\_CURVE** (*ptlist*) Function  
 Draws a curve connecting the points in PTLIST. The latter may be of the form [x0,y0,x1,y1,...] or [[x0,y0],[x1,y1],...] The function JOIN is handy for taking a list of x's and a list of y's and splicing them together. PSDRAW\_CURVE simply invokes the more primitive function PSCURVE. Here is the definition:

```
(defun $psdraw_curve (lis)
  (p "newpath")
  ($pscurve lis)
  (p "stroke"))
```

?DRAW2D may also be used to produce a list

```
points1:?draw2d(1/x,[.05,10],.03)
```

**PSCOM** (*com*)

Function

COM will be inserted in the poscript file eg

```
pscom("4.5 72 mul 5.5 72 mul translate 14 14 scale");
```



## 8 Input and Output

### 8.1 Introduction to Input and Output

### 8.2 FILES

- A file is simply an area on a particular storage device which contains data or text. The only storage devices which are used on the MC machine are disks and tapes. Files on the disks are figuratively grouped into "directories". A directory is just a list of all the files stored under a given user name. Do DESCRIBE(FILEOP); for details of how you may inspect your files using MACSYMA. Other commands which deal with files are: SAVE, FASSAVE, STORE, LOAD, LOADFILE, RESTORE, UNSTORE, STRINGOUT, BATCH, BATCON, DEMO, WRITEFILE, CLOSEFILE, DELFILE, REMFILE, and APPENDFILE.

### 8.3 PLAYBACK

It is possible to play back the input lines in a temporary scroll down window, and so not lose ones current work. This can be done by typing Function E. A numeric argument tells it the line number to start at, otherwise it will go back about 40 lines.

### 8.4 Definitions for Input and Output

**%** Variable  
The last D-line computed by MACSYMA (whether or not it was printed out). (See also %%.)

**%%** Variable  
The value of the last computation performed while in a (MACSYMA-BREAK). Also may be used in compound statements in the nth statement to refer to the value of the (n-1)th statement. E.g.  $F(N):=(\text{INTEGRATE}(X^N,X),\text{SUBST}(3,X,%%)-\text{SUBST}(2,X,%%))$ ; is in essence equivalent to  $F(N):=\text{BLOCK}([%%], %%:\text{INTEGRATE}(X^N,X), \text{SUBST}(3,X,%%)-\text{SUBST}(2,X,%%))$ ; This will also work for communicating between the (n-1)th and nth (non-atomic) BLOCK statements.

**%EDISPFLAG** Variable  
default: [FALSE] - if TRUE, MACSYMA displays %E to a negative exponent as a quotient, i.e.  $\%E^{-X}$  as  $1/\%E^X$ .

**%TH (i)** Function  
is the ith previous computation. That is, if the next expression to be computed is D(j) this is D(j-i). This is useful in BATCH files or for referring to a group of D expressions. For example, if SUM is initialized to 0 then FOR I:1 THRU 10 DO SUM:SUM+%TH(I) will set SUM to the sum of the last ten D expressions.



**"?"** special symbol  
 - As prefix to a function or variable name, signifies that the function or variable is a LISP token, not a MACSYMA token. Two question marks typed together, ??, will flush the current MACSYMA command line.

**ABSBOXCHAR** Variable  
 default: [!] is the character used to draw absolute value signs around expressions which are more than a single line high.

**APPENDFILE** (*filename1, filename2, DSK, directory*) Function  
 is like WRITEFILE(DSK,directory) but appends to the file whose name is specified by the first two arguments. A subsequent CLOSEFILE will delete the original file and rename the appended file.

**BACKUP** () Function  
 To "back up" and see what you did, see PLAYBACK.

**BATCH** (*file-specification*) Function  
 reads in and evaluates MACSYMA command lines from a file - A facility for executing command lines stored on a disk file rather than in the usual on-line mode. This facility has several uses, namely to provide a reservoir for working command lines, for giving error-free demonstrations, or helping in organizing one's thinking in complex problem-solving situations where modifications may be done via a text editor. A batch file consists of a set of MACSYMA command lines, each with its terminating ; or \$, which may be further separated by spaces, carriage- returns, form-feeds, and the like. The BATCH function calls for reading in the command lines from the file one at a time, echoing them on the user console, and executing them in turn. Control is returned to the user console only when serious errors occur or when the end of the file is met. Of course, the user may quit out of the file-processing by typing control-G at any point. BATCH files may be created using a text editor or by use of the STRINGOUT command. Do DESCRIBE(STRINGOUT) for details DESCRIBE(FILE); and DESCRIBE(FILES); have additional information on how the file argument is interpreted, and files in general.

**BATCHKILL** Variable  
 default: [FALSE] if TRUE then the effect of all previous BATCH files is nullified because a KILL(ALL) and a RESET() will be done automatically when the next one is read in. If BATCHKILL is bound to any other atom then a KILL of the value of BATCHKILL will be done.

**BATCHLOAD** (*file-specification*) Function  
 Batches in the file silently without terminal output or labels.

**BATCON** (*argument*) Function  
 continues BATCHing in a file which was interrupted.

**BATCOUNT**

Variable

default: [0] may be set to the number of the last expression BATCHed in from a file. Thus BATCON(BATCOUNT-1) will resume BATCHing from the expression before the last BATCHed in from before.

**BOTHCASES**

Variable

default: [TRUE] if TRUE will cause MAXIMA to retain lower case text as well as upper case. Note, however, that the names of any MAXIMA special variables or functions are in upper case. The default is now TRUE since it makes code more readable, allowing users to have names like SeriesSolve.

Because of this we make the system variables and functions all upper case, and users may enter them however they like (in upper or lower). But all other variables and functions are case sensitive. When you print out your program using for example grind(fundef(f)) then you will see that the symbols like 'IF', 'SIN',... all appear in upper case whereas non system symbols appear in the case which you used.

This is implemented as follows: If the symbol is being encountered for the first time, if the upper case version is in the package and has a nontrivial function or property list, then the upper case symbol is used, and it is recorded on the mixed case one, that the upper case should be used in future. If a symbol is already in the package then it is just used.

In effect this means that most old programs should continue to work, and that new ones may write sIn, Sin, SIN, sin etc and they will all be interpreted as SIN. However if they write MySin this will be different from MYSIN, because MYSIN is not a system function or variable.

```
SeriesSolve(f,x):=
  if (f = sin) ...
```

and this is read as

```
SeriesSolve(f,x):=
  IF (f = SIN) ...
```

**CHANGE\_FILEDEFAULTS**

Variable

default: [TRUE] on PDP10 systems, and FALSE elsewhere. Controls whether the user doing a LOADFILE or BATCH has his file defaults changed to the file LOADFILEd or BATCHed. The TRUE setting is for people who like DDT-style file defaulting. The FALSE setting is for people who like the conventions of other operating systems, who like LISP-style file defaulting, or who write packages which do LOADFILEs or BATCHes which should not interfere with their user's file defaults.

**CLOSEFILE** (*filename1*, *filename2*)

Function

closes a file opened by WRITEFILE and gives it the name filename1 filename2. (On a Lisp Machine one need only say CLOSEFILE();.) Thus to save a file consisting of the display of all input and output during some part of a session with MACSYMA the user issues a WRITEFILE, transacts with MACSYMA, then issues a CLOSEFILE. The user can also issue the PLAYBACK function after a WRITEFILE to save the

display of previous transactions. (Note that what is saved this way is a copy of the display of expressions not the expressions themselves). To save the actual expression in internal form the SAVE function may be used. The expression can then be brought back into MACSYMA via the LOADFILE function. To save the expression in a linear form which may then be BATCHed in later, the STRINGOUT function is used.

**COLLAPSE** (*expression*) Function

collapses" its argument by causing all of its common (i.e. equal) subexpressions to share (i.e. use the same cells), thereby saving space. (COLLAPSE is a subroutine used by the OPTIMIZE command.) Thus, calling COLLAPSE may be useful before using FASSAVE or after loading in a SAVE file. You can collapse several expressions together by using COLLAPSE([*expr1*,...,*exprN*])\$. Similarly, you can collapse the elements of the array A by doing COLLAPSE(LISTARRAY('A))\$.

**CONCAT** (*arg1, arg2, ...*) Function

evaluates its arguments and returns the concatenation of their values resulting in a name or a quoted string the type being given by that of the first argument. Thus if X is bound to 1 and D is unbound then CONCAT(X,2)="12" and CONCAT(D,X+1)=D2.

**SCONCAT** (*arg1, arg2, ...*) Function

evaluates its arguments and concatenates them into a string. Unlike CONCAT, the arguments do NOT need to be atoms. The result is a Common Lisp String.

```
(C5) sconcat("xx[" ,3,"]:" ,expand((x+y)^3));
(D5)  xx[3]:Y^3+3*X*Y^2+3*X^2*Y+X^3
```

The resulting string could be used in conjunction with print.

**CURSORDISP** Variable

default: [TRUE] If TRUE, causes expressions to be drawn by the displayer in logical sequence. This only works with a console which can do cursor movement. If FALSE, expressions are simply printed line by line. CURSORDISP is FALSE when a WRITEFILE is in effect.

**DIREC** Variable

- The value of this variable is the default file directory for SAVE, STORE, FASSAVE, and STRINGOUT. It is initialized to the user's login name, if he has a disk directory, and to one of the USERSi directories otherwise. DIREC determines to what directory disk files will be written.

**DISP** (*expr1,expr2, ...*) Function

is like DISPLAY but only the value of the arguments are displayed rather than equations. This is useful for complicated arguments which don't have names or where only the value of the argument is of interest and not the name.

**DISPCON** (*tensor1,tensor2,...*) Function

displays the contraction properties of the tensori as were given to DEFCON. DISPCON(ALL) displays all the contraction properties which were defined.

**DISPLAY** (*expr1, expr2, ...*) Function

displays equations whose left side is *expr1* unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and FOR statements in order to have intermediate results displayed. The arguments to DISPLAY are usually atoms, subscripted variables, or function calls. (see the DISP function)

(C1) DISPLAY(B[1,2]);

$$B_{1,2} = X - X^2$$

(D1)

DONE

**DISPLAY2D** Variable

default: [TRUE] - if set to FALSE will cause the standard display to be a string (1-dimensional) form rather than a display (2-dimensional) form. This may be of benefit for users on printing consoles who would like to conserve paper.

**DISPLAY\_FORMAT\_INTERNAL** Variable

default: [FALSE] - if set to TRUE will cause expressions to be displayed without being transformed in ways that hide the internal mathematical representation. The display then corresponds to what the INPART command returns rather than the PART command. Examples:

User	PART	INPART
a-b;	A - B	A + (- 1) B
	A	- 1
a/b;	-	A B
	B	
		1/2
sqrt(x);	SQRT(X)	X
	4 X	4
X*4/3;	---	- X
	3	3

**DISPTERMS** (*expr*) Function

displays its argument in parts one below the other. That is, first the operator of 'expr' is displayed, then each term in a sum, or factor in a product, or part of a more general expression is displayed separately. This is useful if *expr* is too large to be otherwise displayed. For example if P1, P2, ... are very large expressions then the display program may run out of storage space in trying to display P1+P2+... all at once. However, DISPTERMS(P1+P2+...) will display P1, then below it P2, etc. When not using DISPTERMS, if an exponential expression is too wide to be displayed as A\*\*B it will appear as EXPT(A,B) (or as NCEXPT(A,B) in the case of A^^B).

**DSKALL**

Variable

default: [] If TRUE will cause values, functions, arrays, and rules to be written periodically onto the disk in addition to labelled expressions. TRUE is the default value whereas if DSKALL is FALSE then only labelled expressions will be written.

**ERROR\_SIZE**

Variable

default: [20 for a display terminal, 10 for others]. controls the size of error messages. For example, let  $U:(C^D^E+B+A)/(\text{COS}(X-1)+1)$ ; . U has an error size of 24. So if ERROR\_SIZE has value < 24 then

```
(C1) ERROR("The function", F00,"doesn't like", U,"as input.");
prints as:
The function F00 doesn't like ERREXP1 as input.
If ERROR_SIZE>24 then as:
```

```

                                E
                                D
                                C  + B + A
The function F00 doesn't like ----- as input.
                                COS(X - 1) + 1
```

Expressions larger than ERROR\_SIZE are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the user-settable list

```
ERROR_SYMS: [ERREXP1,ERREXP2,ERREXP3]
```

. The default value of this switch might change depending on user experience. If you find the defaults either too big or too small for your tastes, send mail to MACSYMA.

**ERROR\_SYMS**

Variable

default: [ERREXP1,ERREXP2,ERREXP3] - In error messages, expressions larger than ERROR\_SIZE are replaced by symbols, and the symbols are set to the expressions. The symbols are taken from the list ERROR\_SYMS, and are initially ERREXP1, ERREXP2, ERREXP3, etc. After an error message is printed, e.g. "The function FOO doesn't like ERREXP1 as input.", the user can type ERREXP1; to see the expression. ERROR\_SYMS may be set by the user to a different set of symbols, if desired.

**EXPT (A,B)**

Function

if an exponential expression is too wide to be displayed as  $A^B$  it will appear as EXPT(A,B) (or as NCEXPT(A,B) in the case of  $A^{^B}$ ).

**EXPTDISPFLAG**

Variable

default: [TRUE] - if TRUE, MACSYMA displays expressions with negative exponents using quotients e.g.,  $X^{*(-1)}$  as  $1/X$ .

**FASSAVE (args)**

Function

is similar to SAVE but produces a FASL file in which the sharing of subexpressions which are shared in core is preserved in the file created. Hence, expressions which have common subexpressions will consume less space when loaded back from a file created

by FASSAVE rather than by SAVE. Files created with FASSAVE are reloaded using LOADFILE, just as files created with SAVE. FASSAVE returns a list of the form [*<name of file>*,*<size of file in blocks>*,...] where ... are the things saved. Warnings are printed out in the case of large files. FASSAVE may be used while a WRITEFILE is in progress.

**FILEDEFAULTS** ()

Function

returns the current default filename, in whatever format is used by the specific Macsyma implementation. (See DESCRIBE(FILE) for what that format is.) This is the file specification used by LOADFILE, BATCH, and a number of other file-accessing commands. FILEDEFAULTS('file) - sets the current filedefaults to "file". The argument to FILEDEFAULTS is evaluated as it is anticipated that the command will be used mainly in programs. The "file" need not be a real file, so one can use this function e.g. if one's real purpose is to set only the "device" field to something, where one does not care about the settings of the other fields.

**FILENAME**

Variable

default: [] - The value of this variable is the first name of the files which are generated by the automatic disk storage scheme. The default value is the first three characters of the user's login name concatenated with the lowest unused integer, e.g. ECR1.

**FILENAME\_MERGE** ("*filename1*", "*filename2*", ...)

Function

; merges together filenames. What this means is that it returns "filename1" except that missing components come from the corresponding components of "filename2", and if they are missing there, then from "filename3".

**FILENUM**

Variable

default: [0] - The default second file name for files generated by SAVE, STRINGOUT, or FASSAVE if no file names are specified by the user. It is an integer, and is incremented by one each time a new file is written.

**FILE\_SEARCH**

Variable

- this is a list of files naming directories to search by LOAD and a number of other functions. The default value of this is a list of the various SHARE directories used by Macsyma. FILE\_SEARCH("filename"); searches on those directories and devices specified by the FILE\_SEARCH\_LISP, FILE\_SEARCH\_MAXIMA and FILE\_SEARCH\_DEMO variables, and returns the name of the first file it finds. This function is invoked by the LOAD function, which is why LOAD("FFT") finds and loads share/fft.mac. You may add a path to the appropriate list. Note that the format of the paths allows specifying multiple extensions and multiple paths.

```
"/home/wfs/###.{o,lisp,mac,mc}"
"/home/{wfs,joe}/###.{o,lisp,mac,mc}"
```

The '###' is replaced by the actual filename passed. File\_SEARCH first checks if the actual name passed exists, before substituting it in the various patterns.

- FILE\_STRING\_PRINT** Variable  
 default: [FALSE] on MC, [TRUE] elsewhere. If TRUE, filenames are output as strings; if FALSE, as lists. For example, the message when an out of core file is loaded into MACSYMA (e.g. the LIMIT package), appears on MC in list format as LIMIT FASL DSK MACSYM being loaded and in string format as: DSK:MACSYM;LIMIT FASL being loaded The string format is like the top level (DDT) file specifications.
- FILE\_TYPE** ("filename") Function  
 ; returns FASL, LISP, or MACSYMA, depending on what kind of file it is. FASL means a compiled Lisp file, which normally has an extension of .VAS in NIL.
- GRIND** (arg) Function  
 prints out arg in a more readable format than the STRING command. It returns a D-line as value. The GRIND switch, default: [FALSE], if TRUE will cause the STRING, STRINGOUT, and PLAYBACK commands to use "grind" mode instead of "string" mode. For PLAYBACK, "grind" mode can also be turned on (for processing input lines) by specifying GRIND as an option.
- IBASE** Variable  
 default: [10] - the base for inputting numbers.
- INCHAR** Variable  
 default: [C] - the alphabetic prefix of the names of expressions typed by the user.
- LDISP** (expr1,expr2,...) Function  
 is like DISP but also generates intermediate labels.
- LDISPLAY** (expr1,expr2,...) Function  
 is like DISPLAY but also generates intermediate labels.
- LINECHAR** Variable  
 default: [E] - the alphabetic prefix of the names of intermediate displayed expressions.
- LINEDISP** Variable  
 default: [TRUE] - Allows the use of line graphics in the drawing of equations on those systems which support them (e.g. the Lisp Machine). This can be disabled by setting LINEDISP to FALSE. It is automatically disabled during WRITEFILE.
- LINEL** Variable  
 default: [] - the number of characters which are printed on a line. It is initially set by MACSYMA to the line length of the type of terminal being used (as far as is known) but may be reset at any time by the user. The user may have to reset it in DDT with :TCTYP as well.

**LOAD** ("filename") Function

; takes one argument, a filename represented as a "string" (i.e. inside quotation marks), or as list (e.g. inside square brackets), and locates and loads in the indicated file. If no directory is specified, it then searches the SHAREi directories and any other directories listed in the FILE\_SEARCH variable and loads the indicated file. LOAD("EIGEN") will load the eigen package without the need for the user to be aware of the details of whether the package was compiled, translated, saved, or fassaved, i.e. LOAD will work on both LOADFILEable and BATCHable files. Note: LOAD will use BATCHLOAD if it finds the file is BATCHable (which means that it will BATCH the file in "silently" without terminal output or labels). Other MACSYMA commands to load in files are: LOADFILE, RESTORE, BATCH, and DEMO. Do DESCRIBE(command); for details. LOADFILE and RESTORE work for files written with SAVE; BATCH and DEMO for those files written with STRINGOUT or created as lists of commands with a text editor. If load can't find the file, check the value FILE\_SEARCH to make sure that it contains an appropriate template.

```
(C4) load("eigen");
MACSYMA BUG: Unknown file type NIL
```

```
Error: macsyma error
Error signalled by MEVAL1.
Broken at $LOAD. Type :H for Help.
MAXIMA>>:q
```

By examining the file system we find the file is actually in /public/maxima/share/eigen.mc. So we add that to the file\_search path. This can be done at start up (see init.lsp) or, else it can be done and then the system resaved once it has been customized for local directories and pathnames.

At lisp level we would do

```
(in-package "MAXIMA")
(setq $file_search ($append (list '(mlist)
    "/tmp/foo.mac" "/tmp/foo.mc") $file_search))
```

and at maxima level:

```
(C5) file_search:append(["/public/maxima/share/foo.mc"],
    file_search)$
(C6) load("eigen");
```

```
batching /usr/public/maxima/share/eigen.mc
```

```
(D6) #/public/maxima/share/eigen.mc
```

```
(C7) eigenvalues(matrix([a,b],[c,d]));
```

```
      2      2
      - Sqrt(D  - 2 A D + 4 B C + A ) + D + A
(D7) [[-----,
      2
```

```
      2      2
      Sqrt(D  - 2 A D + 4 B C + A ) + D + A
```



-----], [1, 1]]  
2

**LOADFILE** (*filename*) Function

loads a file as designated by its arguments. This function may be used to bring back quantities that were stored from a prior MACSYMA session by use of the SAVE or STORE functions. Specify the pathname as on your operating system. For unix this would be "/home/wfs/foo.mc" for example.

**LOADPRINT** Variable

default: [TRUE] - governs the printing of messages accompanying loading of files. The following options are available: TRUE means always print the message; 'LOADFILE means print only when the LOADFILE command is used; 'AUTOLOAD means print only when a file is automatically loaded in (e.g. the integration file SIN FASL); FALSE means never print the loading message.

**NOSTRING** (*arg*) Function

displays all input lines when playing back rather than STRINGing them. If arg is GRIND then the display will be in a more readable format. One may include any number of options as in PLAYBACK([5,10],20,TIME,SLOW).

**OBASE** Variable

default: [10] the base for display of numbers.

**OUTCHAR** Variable

default: [D] - the alphabetic prefix of the names of outputted expressions.

**PACKAGEFILE** Variable

default:[FALSE] - Package designers who use SAVE, FASSAVE, or TRANSLATE to create packages (files) for others to use may want to set PACKAGEFILE:TRUE\$ to prevent information from being added to MACSYMA's information-lists (e.g. VALUES, FUNCTIONS) except where necessary when the file is loaded in. In this way, the contents of the package will not get in the user's way when he adds his own data. Note that this will not solve the problem of possible name conflicts. Also note that the flag simply affects what is output to the package file. Setting the flag to TRUE is also useful for creating MACSYMA init files.

**PARSEWINDOW** Variable

default:[10] - the maximum number of "lexical tokens" that are printed out on each side of the error-point when a syntax (parsing) error occurs. This option is especially useful on slow terminals. Setting it to -1 causes the entire input string to be printed out when an error occurs.

**PFEFORMAT** Variable

default: [FALSE] - if TRUE will cause rational numbers to display in a linear form and denominators which are integers to display as rational number multipliers.

**PRINT** (*exp1, exp2, ...*) Function  
 evaluates and displays its arguments one after the other "on a line" starting at the leftmost position. If *exp1* is unbound or is preceded by a single quote or is enclosed in "s then it is printed literally. For example, PRINT("THE VALUE OF X IS ",X). The value returned by PRINT is the value of its last argument. No intermediate lines are generated. (For "printing" files, see the PRINTFILE function.)

**SPRINT** (*exp1, exp2, ...*) Function  
 evaluates and displays its arguments one after the other "on a line" starting at the leftmost position. The numbers are printed with the '-' right next to the number, and it disregards line length.

**TCL\_OUTPUT** (*LIST INDEX &optional-skip*) Function  
 prints a TCL list based on LIST extracting the INDEX slot. Here skip defaults to 2, meaning that every other element will be printed if the argument is of the form a list of numbers, rather than a list of lists. For example:

```
TCL_OUTPUT([x1,y1,x2,y2,x3,y3],1) --> {x1 x2 x3 }
TCL_OUTPUT([x1,y1,x2,y2,x3,y3],2) --> {y1 y2 y3 }
TCL_OUTPUT([1,2,3,4,5,6],1,3) --> {1 4}
TCL_OUTPUT([1,2,3,4,5,6],2,3) --> {2 5}
```

**READ** (*string1, ...*) Function  
 prints its arguments, then reads in and evaluates one expression. For example:  
 A:READ("ENTER THE NUMBER OF VALUES").

**READONLY** (*string1,...*) Function  
 prints its arguments, then reads in an expression (which in contrast to READ is not evaluated).

**REVEAL** (*exp,depth*) Function  
 will display *exp* to the specified integer depth with the length of each part indicated. Sums will be displayed as Sum(n) and products as Product(n) where n is the number of subparts of the sum or product. Exponentials will be displayed as Expt.

```
(C1) INTEGRATE(1/(X^3+2),X)$
(C2) REVEAL(%,2);
(D2)                               Negterm + Quotient + Quotient
(C3) REVEAL(D1,3);
(D3)                               ATAN          LOG
- Quotient + ----- + -----
                    Product(2)   Product(2)
```

**RMXCHAR** Variable  
 default: []] - The character used to display the (right) delimiter of a matrix (see also LMXCHAR).

**SAVE** (*filename, arg1, arg2, ..., argi*) Function  
 saves quantities described by its arguments on disk and keeps them in core also. The *arg*'s are the expressions to be SAVED. ALL is the simplest, but note that saving ALL will save the entire contents of your MACSYMA, which in the case of a large computation may result in a large file. VALUES, FUNCTIONS, or any other items on the INFOLISTS (do DESCRIBE(INFOLISTS); for the list) may be SAVED, as may functions and variables by name. C and D lines may also be saved, but it is better to give them explicit names, which may be done in the command line, e.g. SAVE(RES1=D15); Files saved with SAVE should be reloaded with LOADFILE. SAVE returns the pathname where the items were saved.

**SAVEDEF** Variable  
 default: [TRUE] - if TRUE will cause the MACSYMA version of a user function to remain when the function is TRANSLATED. This permits the definition to be displayed by DISPFUN and allows the function to be edited. If SAVEDEF is FALSE, the names of translated functions are removed from the FUNCTIONS list.

**SHOW** (*exp*) Function  
 will display *exp* with the indexed objects in it shown having covariant indices as subscripts, contravariant indices as superscripts. The derivative indices will be displayed as subscripts, separated from the covariant indices by a comma.

**SHOWRATVARS** (*exp*) Function  
 returns a list of the RATVARS (CRE variables) of *exp*.

**STARDISP** Variable  
 default: [FALSE] - if TRUE will cause multiplication to be displayed explicitly with an \* between operands.

**STRING** (*expr*) Function  
 converts *expr* to MACSYMA's linear notation (similar to FORTRAN's) just as if it had been typed in and puts *expr* into the buffer for possible editing (in which case *expr* is usually *C<sub>i</sub>*) The STRING'ed expression should not be used in a computation.

**STRINGOUT** (*args*) Function  
 will output an expression to a file in a linear format. Such files are then used by the BATCH or DEMO commands. STRINGOUT(*file-specification, A1, A2, ...*) outputs to a file given by *file-specification* ([*filename1, filename2, DSK, directory*]) the values given by *A1, A2, ..* in a MACSYMA readable format. The *file-specification* may be omitted, in which case the default values will be used. The *A<sub>i</sub>* are usually C labels or may be INPUT meaning the value of all C labels. Another option is to make *ai* FUNCTIONS which will cause all of the user's function definitions to be strungout (i.e. all those retrieved by DISPFUN(ALL)). Likewise the *ai* may be VALUES, and all the variables to which the user has assigned values will be strungout. *ai* may also be a list [*m, n*] which means to stringout all labels in the range *m* through *n* inclusive. This function may be used to create a file of FORTRAN statements by doing some

simple editing on the strungout expressions. If the GRIND switch is set to TRUE, then STRINGOUT will use GRIND format instead of STRING format. Note: a STRINGOUT may be done while a WRITEFILE is in progress.

**TEX** (*expr*) Function  
**TEX**(*expr, filename*) Function  
**TEX**(*label, filename*) Function

In the case of a label, a left-equation-number will be produced. in case a file-name is supplied, the output will be appended to the file.

```
(C1) integrate(1/(1+x^3),x);
```

$$(D1) \frac{\frac{2x-1}{2} \operatorname{ATAN}\left(\frac{2x-1}{\sqrt{3}}\right) + \frac{\log(x+1)}{3}}{6\sqrt{3}} + \frac{\log(x+1)}{3}$$

```
(C2) tex(d1);
```

```
$$-{\frac{\log \left(x^2-x+1\right)}{6}}+{\frac{\arctan \left(\frac{2x-1}{\sqrt{3}}\right)}{\sqrt{3}}+{\frac{\log \left(x+1\right)}{3}}\leqno{\tt (D1)}
```

```
(D2) (D1)
```

```
(C6) tex(integrate(sin(x),x));
```

```
$$-\cos x$$
```

```
(D6) FALSE
```

```
(C7) tex(d1,"/tmp/jo.tex");
```

```
(D7) (D1)
```

where the last expression will be appended to the file '/tmp/jo.tex'

**SYSTEM**(*command*) Function

Execute COMMAND as a subprocess. The command will be passed to the default shell for execution. System is not supported by all operating systems, but generally exists in the unix environment. if hist is a list of frequencies which you wish to plot as a bar graph using xgraph.

```
(C1) (with_stdout("_hist.out",
      for i:1 thru length(hist) do (
        print(i,hist[i]))),
      system("xgraph -bar -brw .7 -nl < _hist.out"));
```

In order to make the plot be done in the background (returning control to maxima) and remove the temporary file after it is done do:

```
system("(xgraph -bar -brw .7 -nl < _hist.out; rm -f _hist.out)&")
```

**TTYOFF** Variable

default: [FALSE] - if TRUE stops printing output to the console.

**WITH\_STDOUT**(file,stmt1,stmt2,...) macro

Opens file and then evaluates stmt1, stmt2, .... Any printing to standard output goes to the file instead of the terminal. It always returns FALSE. Note the binding of display2d to be false, otherwise the printing will have things like "- 3" instead of "-3".

```
mygnuplot(f,var,range,number_ticks):=
  block([numer:true,display2d:false],
    with_stdout("/tmp/gnu",
      for x:range[1] thru range[2] step
        (range[2]-range[1])/number_ticks
        do (print(x,at(f,var=x)))),
    system("echo \"set data style lines; set title '",
      f,"' ;plot '/tmp/gnu'
;pause 10 \" | gnuplot"));
```

```
(C8) with_stdout("/home/wfs/joe",
      n:10,
      for i:8 thru n
        do(print("factorial(",i,") gives ",i!)));
(D8)      FALSE
(C9) system("cat /home/wfs/joe");
factorial( 8 ) gives 40320
factorial( 9 ) gives 362880
factorial( 10 ) gives 3628800
(D9)      0
```

**WRITEFILE** (DSK, directory) Function

opens up a file for writing. On a Lisp Machine one uses WRITEFILE("filename"). All interaction between the user and MACSYMA is then recorded in this file, just as it is on the console. Such a file is a transcript of the session, and is not reloadable or batchable into MACSYMA again. (See also CLOSEFILE.)

## 9 Floating Point

### 9.1 Definitions for Floating Point

- BFFAC** (*exp,n*) Function  
 BFLOAT version of the Factorial (shifted Gamma) function. The 2nd argument is how many digits to retain and return, it's a good idea to request a couple of extra. This function is available by doing `LOAD(BFFAC);` .
- ALGEPSILON** Variable  
 The default value is  $10^{-8}$ . The value of ALGEPSILON is used by ALGSYS.
- BFLOAT** (*X*) Function  
 converts all numbers and functions of numbers to bigfloat numbers. Setting `FP-PREC[16]` to *N*, sets the bigfloat precision to *N* digits. If `FLOAT2BF[FALSE]` is `FALSE` a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).
- BFLOATP** (*exp*) Function  
 is `TRUE` if *exp* is a bigfloat number else `FALSE`.
- BFPSI** (*n,z,fpprec*) Function  
 gives polygammas of real arg and integer order. For digamma, `BFPSI0(z,fpprec)` is more direct. Note `-BFPSI0(1,fpprec)` provides BFLOATed `%GAMMA`. To use this do `LOAD(BFFAC);`
- BFTORAT** Variable  
 default: `[FALSE]` controls the conversion of bfloats to rational numbers. If `BFTORAT:FALSE` `RATEPSILON` will be used to control the conversion (this results in relatively small rational numbers). If `BFTORAT:TRUE` , the rational number generated will accurately represent the bfloat.
- BFTRUNC** Variable  
 default: `[TRUE]` causes trailing zeroes in non-zero bigfloat numbers not to be displayed. Thus, if `BFTRUNC:FALSE`, `BFLOAT(1);` displays as `1.0000000000000000B0`. Otherwise, this is displayed as `1.0B0`.
- CBFAC** (*z,fpprec*) Function  
 a factorial for complex bfloats. It may be used by doing `LOAD(BFAC);` For more details see `share2/bfac.usg`.

**FLOAT** (*exp*) Function  
 converts integers, rational numbers and bigfloats in *exp* to floating point numbers. It is also an EVFLAG, FLOAT causes non-integral rational numbers and bigfloat numbers to be converted to floating point.

**FLOAT2BF** Variable  
 default: [FALSE] if FALSE, a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

**FLOATDEFUNK** Function  
 - is a utility for making floating point functions from mathematical expression. It will take the input expression and FLOAT it, then OPTIMIZE it, and then insert MODE\_DECLARATIONS for all the variables. This is THE way to use ROMBERG, PLOT2, INTERPOLATE, etc. e.g. EXP:some-hairy-macsyma-expression;  
`FLOATDEFUNK('F, ['X], EXP);`  
 will define the function F(X) for you. (Do PRINTFILE(MCOMPI,DOC,MAXDOC); for more details.)

**FLOATNUMP** (*exp*) Function  
 is TRUE if *exp* is a floating point number else FALSE.

**FPPREC** Variable  
 default: [16] - Floating Point PRECISION. Can be set to an integer representing the desired precision.

**FPPRINTPREC** Variable  
 default: [0] - The number of digits to print when printing a bigfloat number, making it possible to compute with a large number of digits of precision, but have the answer printed out with a smaller number of digits. If FPPRINTPREC is 0 (the default), or  $\geq$  FPPREC, then the value of FPPREC controls the number of digits used for printing. However, if FPPRINTPREC has a value between 2 and FPPREC-1, then it controls the number of digits used. (The minimal number of digits used is 2, one to the left of the point and one to the right. The value 1 for FPPRINTPREC is illegal.)

**?ROUND** (*x*,&optional-divisor) Function  
 round the floating point X to the nearest integer. The argument must be a regular system float, not a bigfloat. The ? beginning the name indicates this is normal common lisp function.

```
(C3) ?round(-2.8);
(D3)      - 3
```

**?TRUNCATE** (*x*,&optional-divisor) Function  
 truncate the floating point X towards 0, to become an integer. The argument must be a regular system float, not a bigfloat. The ? beginning the name indicates this is normal common lisp function.

```
(C4) ?truncate(-2.8);
(D4)      - 2
(C5) ?truncate(2.4);
(D5)      2
(C6) ?truncate(2.8);
(D6)      2
```

**ZUNDERFLOW**

Variable

default: [TRUE] - if FALSE, an error will be signaled if floating point underflow occurs. Currently in NIL Macsyma, all floating-point underflow, floating-point overflow, and division-by-zero errors signal errors, and this switch is ignored.





## 10 Contexts

### 10.1 Definitions for Contexts

**ACTIVATE** (*cont1, cont2, ...*) Function

causes the specified contexts *cont*i to be activated. The facts in these contexts are used in making deductions and retrieving information. The facts in these contexts are not listed when **FACTS**(*cont*); is done. The variable **ACTIVECONTEXTS** is the list of contexts which are active by way of the **ACTIVATE** function.

**ACTIVECONTEXTS** Variable

default: [] is a list of the contexts which are active by way of the **ACTIVATE** function, as opposed to being active because they are subcontexts of the current context.

**ASSUME** (*pred1, pred2, ...*) Function

First checks the specified predicates for redundancy and consistency with the current data base. If the predicates are consistent and non-redundant, they are added to the data base; if inconsistent or redundant, no action is taken. **ASSUME** returns a list whose entries are the predicates added to the data base and the atoms **REDUNDANT** or **INCONSISTENT** where applicable.

**ASSUMESCALAR** Variable

default: [TRUE] - helps govern whether expressions *exp* for which

**NONSCALARP**(*exp*) is FALSE

are assumed to behave like scalars for certain transformations as follows: Let *exp* represent any non-list/non-matrix, and [1,2,3] any list or matrix.

*exp*. [1,2,3] ; ==>

[*exp*, 2\**exp*, 3\**exp*]

if **ASSUMESCALAR** is TRUE or **SCALARP**(*exp*) is TRUE or **CONSTANTP**(*exp*) is TRUE. If **ASSUMESCALAR** is TRUE, such expressions will behave like scalars only for the commutative operators, but not for ".". If **ASSUMESCALAR** is FALSE, such expressions will behave like non-scalars. If **ASSUMESCALAR** is ALL, such expressions will behave like scalars for all the operators listed above.

**ASSUME\_POS** Variable

default:[FALSE] - When using **INTEGRATE**, etc. one often introduces parameters which are real and positive or one's calculations can often be constructed so that this is true. There is a switch **ASSUME\_POS** (default FALSE) such that if set to TRUE, **MACSYMA** will assume one's parameters are positive. The intention here is to cut down on the number of questions **MACSYMA** needs to ask. Obviously, **ASSUME** information or any contextual information present will take precedence. The user can control what is considered to be a parameter for this purpose. Parameters by default are those which satisfy **SYMBOLP**(*x*) OR **SUBVARP**(*x*). The user can change this by setting the option **ASSUME\_POS\_PRED** [default FALSE] to the name of a

predicate function of one argument. E.g. if you want only symbols to be parameters, you can do `ASSUME_POS:TRUE$ ASSUME_POS_PRED:'SYMBOLP$ SIGN(A); -> POS, SIGN(A[1]); -> PNZ.`

### **ASSUME\_POS\_PRED**

Variable

default:[FALSE] - may be set to one argument to control what will be considered a parameter for the "assumptions" that INTEGRATE will make... see ASSUME and ASSUME\_POS .

### **CONTEXT**

Variable

default: INITIAL. Whenever a user assumes a new fact, it is placed in the context named as the current value of the variable CONTEXT. Similarly, FORGET references the current value of CONTEXT. To change contexts, simply bind CONTEXT to the desired context. If the specified context does not exist it will be created by an invisible call to NEWCONTEXT. The context specified by the value of CONTEXT is automatically activated. (Do DESCRIBE(CONTEXTS); for a general description of the CONTEXT mechanism.)

### **CONTEXTS**

Variable

default: [INITIAL,GLOBAL] is a list of the contexts which currently exist, including the currently active context. The context mechanism makes it possible for a user to bind together and name a selected portion of his data base, called a context. Once this is done, the user can have MACSYMA assume or forget large numbers of facts merely by activating or deactivating their context. Any symbolic atom can be a context, and the facts contained in that context will be retained in storage until the user destroys them individually by using FORGET or destroys them as a whole by using KILL to destroy the context to which they belong. Contexts exist in a formal hierarchy, with the root always being the context GLOBAL, which contains information about MACSYMA that some functions need. When in a given context, all the facts in that context are "active" (meaning that they are used in deductions and retrievals) as are all the facts in any context which is an inferior of that context. When a fresh MACSYMA is started up, the user is in a context called INITIAL, which has GLOBAL as a subcontext. The functions which deal with contexts are: FACTS, NEWCONTEXT, SUPCONTEXT, KILLCONTEXT, ACTIVATE, DEACTIVATE, ASSUME, and FORGET.

### **DEACTIVATE** (*cont1, cont2, ...*)

Function

causes the specified contexts *cont*<sub>*i*</sub> to be deactivated.

### **FACTS** (*item*)

Function

If '*item*' is the name of a context then FACTS returns a list of the facts in the specified context. If no argument is given, it lists the current context. If '*item*' is not the name of a context then it returns a list of the facts known about '*item*' in the current context. Facts that are active, but in a different context, are not listed.

**FEATURES**

declaration

- MACSYMA has built-in properties which are handled by the data base. These are called FEATURES. One can do DECLARE(N,INTEGER), etc. One can also DECLARE one's own FEATURES by e.g. DECLARE( INCREASING, FEATURE); which will then allow one to say DECLARE(F, INCREASING);. One can then check if F is INCREASING by using the predicate FEATUREP via FEATUREP(F, INCREASING). There is an infolist FEATURES which is a list of known FEATURES. At present known FEATURES are: INTEGER, NONINTEGER, EVEN, ODD, RATIONAL, IRRATIONAL, REAL, IMAGINARY, COMPLEX, ANALYTIC, INCREASING, DECREASING, ODDFUN, EVENFUN, POSFUN, COMMUTATIVE, LASSOCIATIVE, RASSOCIATIVE, SYMMETRIC, and ANTISYMMETRIC. [Note: system "features" may be checked with STATUS(FEATURE, ...); See DESCRIBE(STATUS); or DESCRIBE(FEATURE); for details.]

**FORGET** (*pred1, pred2, ...*)

Function

removes relations established by ASSUME. The predicates may be expressions equivalent to (but not necessarily identical to) those previously ASSUMEd. FORGET(list) is also a legal form.

**KILLCONTEXT** (*context1, context2, ..., contextn*)

Function

kills the specified contexts. If one of them is the current context, the new current context will become the first available subcontext of the current context which has not been killed. If the first available unkilld context is GLOBAL then INITIAL is used instead. If the INITIAL context is killed, a new INITIAL is created, which is empty of facts. KILLCONTEXT doesn't allow the user to kill a context which is currently active, either because it is a subcontext of the current context, or by use of the function ACTIVATE.

**NEWCONTEXT** (*name*)

Function

creates a new (empty) context, called name, which has GLOBAL as its only subcontext. The new context created will become the currently active context.

**SUPCONTEXT** (*name, context*)

Function

will create a new context (called name) whose subcontext is context. If context is not specified, the current context will be assumed. If it is specified, context must exist.



# 11 Polynomials

## 11.1 Introduction to Polynomials

Polynomials are stored in maxima either in General Form or as Canonical Rational Expressions (CRE) form. The latter is a standard form, and is used internally by operations such as factor, ratsimp, and so on.

Canonical Rational Expressions constitute a kind of representation which is especially suitable for expanded polynomials and rational functions (as well as for partially factored polynomials and rational functions when `RATFAC[FALSE]` is set to `TRUE`). In this CRE form an ordering of variables (from most to least main) is assumed for each expression. Polynomials are represented recursively by a list consisting of the main variable followed by a series of pairs of expressions, one for each term of the polynomial. The first member of each pair is the exponent of the main variable in that term and the second member is the coefficient of that term which could be a number or a polynomial in another variable again represented in this form. Thus the principal part of the CRE form of  $3X^2-1$  is `(X 2 3 0 -1)` and that of  $2XY+X-3$  is `(Y 1 (X 1 2) 0 (X 1 1 0 -3))` assuming Y is the main variable, and is `(X 1 (Y 1 2 0 1) 0 -3)` assuming X is the main variable. "Main"-ness is usually determined by reverse alphabetical order. The "variables" of a CRE expression needn't be atomic. In fact any subexpression whose main operator is not `+` `-` `*` `/` or `^` with integer power will be considered a "variable" of the expression (in CRE form) in which it occurs. For example the CRE variables of the expression  $X+\sin(X+1)+2\sqrt{X}+1$  are X, `SQRT(X)`, and `SIN(X+1)`. If the user does not specify an ordering of variables by using the `RATVARS` function `MACSYMA` will choose an alphabetic one. In general, CRE's represent rational expressions, that is, ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. The internal form is essentially a pair of polynomials (the numerator and denominator) preceded by the variable ordering list. If an expression to be displayed is in CRE form or if it contains any subexpressions in CRE form, the symbol `/R/` will follow the line label. See the `RAT` function for converting an expression to CRE form. An extended CRE form is used for the representation of Taylor series. The notion of a rational expression is extended so that the exponents of the variables can be positive or negative rational numbers rather than just positive integers and the coefficients can themselves be rational expressions as described above rather than just polynomials. These are represented internally by a recursive polynomial form which is similar to and is a generalization of CRE form, but carries additional information such as the degree of truncation. As with CRE form, the symbol `/T/` follows the line label of such expressions.

## 11.2 Definitions for Polynomials

### ALGEBRAIC

Variable

default: `[FALSE]` must be set to `TRUE` in order for the simplification of algebraic integers to take effect.

### BERLEFACT

Variable

default: `[TRUE]` if `FALSE` then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used.

**BEZOUT** (*p1, p2, var*) Function  
 an alternative to the RESULTANT command. It returns a matrix. DETERMINANT of this matrix is the desired resultant.

**BOTHCOEF** (*exp, var*) Function  
 returns a list whose first member is the coefficient of var in exp (as found by RATCOEF if exp is in CRE form otherwise by COEFF) and whose second member is the remaining part of exp. That is, [A,B] where  $\text{exp} = A * \text{var} + B$ .

```
(C1) ISLINEAR(EXP,VAR):=BLOCK([C],
      C:BOTHCOEF(RAT(EXP,VAR),VAR),
      IS(FREEOF(VAR,C) AND C[1]#0))$
(C2) ISLINEAR((R**2-(X-R)**2)/X,X);
(D2)                                     TRUE
```

**COEFF** (*exp, v, n*) Function  
 obtains the coefficient of  $v^n$  in exp. n may be omitted if it is 1. v may be an atom, or complete subexpression of exp e.g., X, SIN(X), A[I+1], X+Y, etc. (In the last case the expression (X+Y) should occur in exp). Sometimes it may be necessary to expand or factor exp in order to make  $v^n$  explicit. This is not done automatically by COEFF.

```
(C1) COEFF(2*A*TAN(X)+TAN(X)+B=5*TAN(X)+3,TAN(X));
(D1)                                     2 A + 1 = 5
(C2) COEFF(Y+X*%E**X+1,X,0);
(D2)                                     Y + 1
```

**COMBINE** (*exp*) Function  
 simplifies the sum exp by combining terms with the same denominator into a single term.

**CONTENT** (*p1, var1, ..., varn*) Function  
 returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial p1 in the variable varn (this is the content) and whose second element is the polynomial p1 divided by the content.

```
(C1) CONTENT(2*X*Y+4*X**2*Y**2,Y);
(D1) [2*X, 2*X*Y**2+Y].
```

**DENOM** (*exp*) Function  
 returns the denominator of the rational expression exp.

**DIVIDE** ( $p1, p2, var1, \dots, varn$ ) Function  
 computes the quotient and remainder of the polynomial  $p1$  divided by the polynomial  $p2$ , in a main polynomial variable,  $varn$ . The other variables are as in the `RATVARS` function. The result is a list whose first element is the quotient and whose second element is the remainder.

```
(C1) DIVIDE(X+Y,X-Y,X);
(D1)                                     [1, 2 Y]
(C2) DIVIDE(X+Y,X-Y);
(D2)                                     [-1, 2 X]
```

(Note that  $Y$  is the main variable in  $C2$ )

**ELIMINATE** ( $[eq1,eq2,\dots,eqn],[v1,v2,\dots,vk]$ ) Function  
 eliminates variables from equations (or expressions assumed equal to zero) by taking successive resultants. This returns a list of  $n-k$  expressions with the  $k$  variables  $v1,\dots,vk$  eliminated. First  $v1$  is eliminated yielding  $n-1$  expressions, then  $v2$  is, etc. If  $k=n$  then a single expression in a list is returned free of the variables  $v1,\dots,vk$ . In this case `SOLVE` is called to solve the last resultant for the last variable. Example:

```
(C1) EXP1:2*X^2+Y*X+Z;
(D1)                                     2
                                     Z + X Y + 2 X
(C2) EXP2:3*X+5*Y-Z-1;
(D2)                                     - Z + 5 Y + 3 X - 1
(C3) EXP3:Z^2+X-Y^2+5;
(D3)                                     2    2
                                     Z - Y + X + 5
(C4) ELIMINATE([EXP3,EXP2,EXP1],[Y,Z]);
(D3) [7425 X8 - 1170 X7 + 1299 X6 + 12076 X5 + 22887 X4
      - 5154 X3 - 1291 X2 + 7688 X + 15376]
```

**EZGCD** ( $p1, p2, \dots$ ) Function  
 gives a list whose first element is the g.c.d of the polynomials  $p1,p2,\dots$  and whose remaining elements are the polynomials divided by the g.c.d. This always uses the `EZGCD` algorithm.

**FACEXPAND** Variable  
 default: `[TRUE]` controls whether the irreducible factors returned by `FACTOR` are in expanded (the default) or recursive (normal CRE) form.

**FACTCOMB** ( $exp$ ) Function  
 tries to combine the coefficients of factorials in  $exp$  with the factorials themselves by converting, for example,  $(N+1)*N!$  into  $(N+1)!$ . `SUMSPLITFACT[TRUE]` if set to `FALSE` will cause `MINFACTORIAL` to be applied after a `FACTCOMB`.



```
(C1) (N+1)^B*N!^B;
      B   B
(D1) (N + 1)  N!
(C2) FACTCOMB(%);
```

**FACTOR** (*exp*) Function

factors the expression *exp*, containing any number of variables or functions, into factors irreducible over the integers. **FACTOR**(*exp*, *p*) factors *exp* over the field of integers with an element adjoined whose minimum polynomial is *p*. **FACTORFLAG**[**FALSE**] 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. **SAVEFACTORS**[**FALSE**] if **TRUE** causes the factors of an expression which is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors. **BERLEFACT**[**TRUE**] if **FALSE** then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used. **INTFACLIM**[1000] is the largest divisor which will be tried when factoring a bignum integer. If set to **FALSE** (this is the case when the user calls **FACTOR** explicitly), or if the integer is a fixnum (i.e. fits in one machine word), complete factorization of the integer will be attempted. The user's setting of **INTFACLIM** is used for internal calls to **FACTOR**. Thus, **INTFACLIM** may be reset to prevent MACSYMA from taking an inordinately long time factoring large integers. **NEWFAC**[**FALSE**] may be set to true to use the new factoring routines. Do **EXAMPLE(FACTOR)**; for examples.

**FACTORFLAG** Variable

default: [**FALSE**] if **FALSE** suppresses the factoring of integer factors of rational expressions.

**FACTOROUT** (*exp*, *var1*, *var2*, ...) Function

rearranges the sum *exp* into a sum of terms of the form  $f(\text{var1}, \text{var2}, \dots) * g$  where *g* is a product of expressions not containing the *var*'s and *f* is factored.

**FACTORSUM** (*exp*) Function

tries to group terms in factors of *exp* which are sums into groups of terms such that their sum is factorable. It can recover the result of **EXPAND**((*X*+*Y*)<sup>2</sup>+(*Z*+*W*)<sup>2</sup>) but it can't recover **EXPAND**((*X*+1)<sup>2</sup>+(*X*+*Y*)<sup>2</sup>) because the terms have variables in common.

```
(C1) (X+1)*((U+V)^2+A*(W+Z)^2),EXPAND;
      2      2      2      2
(D1) A X Z  + A Z  + 2 A W X Z + 2 A W Z + A W  X + V  X
      2      2      2      2
```

$$\begin{aligned}
 &+ 2 U V X + U^2 X + A W^2 + V^2 + 2 U V + U^2 \\
 \text{(C2) FACTORSUM(\%);} \\
 \text{(D2)} & \quad (X + 1) (A (Z + W)^2 + (V + U)^2)
 \end{aligned}$$

**FASTTIMES** (*p1, p2*)

Function

multiplies the polynomials *p1* and *p2* by using a special algorithm for multiplication of polynomials. They should be multivariate, dense, and nearly the same size. Classical multiplication is of order  $N \cdot M$  where  $N$  and  $M$  are the degrees. FASTTIMES is of order  $\text{MAX}(N, M)^{1.585}$ .

**FULLRATSIMP** (*exp*)

Function

When non-rational expressions are involved, one call to RATSIMP followed as is usual by non-rational ("general") simplification may not be sufficient to return a simplified result. Sometimes, more than one such call may be necessary. The command FULLRATSIMP makes this process convenient. FULLRATSIMP repeatedly applies RATSIMP followed by non-rational simplification to an expression until no further change occurs. For example, consider For the expression EXP:  $(X^{(A/2)+1})^2 \cdot (X^{(A/2)-1})^2 / (X^A - 1)$ . RATSIMP(EXP); gives  $(X^{(2 \cdot A) - 2} \cdot X^{A+1}) / (X^A - 1)$ . FULLRATSIMP(EXP); gives  $X^A - 1$ . The problem may be seen by looking at RAT(EXP); which gives  $((X^{(A/2)})^4 \cdot 2 \cdot (X^{(A/2)})^2 + 1) / (X^A - 1)$ . FULLRATSIMP(*exp, var1, ..., varn*) takes one or more arguments similar to RATSIMP and RAT.

**FULLRATSUBST** (*a, b, c*)

Function

is the same as RATSUBST except that it calls itself recursively on its result until that result stops changing. This function is useful when the replacement expression and the replaced expression have one or more variables in common. FULLRATSUBST will also accept its arguments in the format of LRATSUBST. That is, the first argument may be a single substitution equation or a list of such equations, while the second argument is the expression being processed. There is a demo available by DEMO("lrats.dem"); .

**GCD** (*p1, p2, var1, ...*)

Function

computes the greatest common divisor of *p1* and *p2*. The flag GCD[SPMOD] determines which algorithm is employed. Setting GCD to EZ, EEZ, SUBRES, RED, or SPMOD selects the EZGCD, New EEZ GCD, subresultant PRS, reduced, or modular algorithm, respectively. If GCD:FALSE then GCD(*p1, p2, var*) will always return 1 for all *var*. Many functions (e.g. RATSIMP, FACTOR, etc.) cause gcd's to be taken implicitly. For homogeneous polynomials it is recommended that GCD:SUBRES be used. To take the gcd when an algebraic is present, e.g. GCD( $X^2 - 2 \cdot \text{SQRT}(2) \cdot X + 2, X - \text{SQRT}(2)$ ); , ALGEBRAIC must be TRUE and GCD must not be EZ. SUBRES is a new algorithm, and people who have been using the RED setting should probably change it to SUBRES. The GCD flag, default: [SPMOD], if FALSE will also prevent the greatest common divisor from being taken when expressions are converted to CRE form. This will sometimes speed the calculation if gcds are not required.

**GCFACTOR** (*n*) Function  
 factors the gaussian integer *n* over the gaussians, i.e. numbers of the form  $a + b i$  where *a* and *b* are rational integers (i.e. ordinary integers). Factors are normalized by making *a* and *b* non-negative.

**GFACTOR** (*exp*) Function  
 factors the polynomial *exp* over the Gaussian integers (i. e. with  $\text{SQRT}(-1) = \%I$  adjoined). This is like  $\text{FACTOR}(\text{exp}, A^{**2}+1)$  where *A* is  $\%I$ .

```
(C1) GFACTOR(X**4-1);
(D1)      (X - 1) (X + 1) (X + %I) (X - %I)
```

**GFACTORSUM** (*exp*) Function  
 is similar to  $\text{FACTORSUM}$  but applies  $\text{GFACTOR}$  instead of  $\text{FACTOR}$ .

**HIPOW** (*exp*, *v*) Function  
 the highest explicit exponent of *v* in *exp*. Sometimes it may be necessary to expand *exp* since this is not done automatically by  $\text{HIPOW}$ . Thus  $\text{HIPOW}(Y^{**3}X^{**2}+X*Y^{**4}, X)$  is 2.

**INTFACLIM** Variable  
 default: [1000] is the largest divisor which will be tried when factoring a bignum integer. If set to FALSE (this is the case when the user calls  $\text{FACTOR}$  explicitly), or if the integer is a fixnum (i.e. fits in one machine word), complete factorization of the integer will be attempted. The user's setting of  $\text{INTFACLIM}$  is used for internal calls to  $\text{FACTOR}$ . Thus,  $\text{INTFACLIM}$  may be reset to prevent MACSYMA from taking an inordinately long time factoring large integers.

**KEEPFLOAT** Variable  
 default: [FALSE] - if set to TRUE will prevent floating point numbers from being rationalized when expressions which contain them are converted to CRE form.

**LRATSUBST** (*list*, *exp*) Function  
 is analogous to  $\text{SUBST}(\text{list\_of\_equations}, \text{exp})$  except that it uses  $\text{RATSUBST}$  instead of  $\text{SUBST}$ . The first argument of  $\text{LRATSUBST}$  must be an equation or a list of equations identical in format to that accepted by  $\text{SUBST}$  (see  $\text{DESCRIBE}(\text{SUBST})$ ). The substitutions are made in the order given by the list of equations, that is, from left to right. A demo is available by doing  $\text{DEMO}(\text{"lrats.dem"})$ ; .

**MODULUS** Variable  
 default: [FALSE] - if set to a positive prime *p*, then all arithmetic in the rational function routines will be done modulo *p*. That is all integers will be reduced to less than  $p/2$  in absolute value (if  $p=2$  then all integers are reduced to 1 or 0). This is the so called "balanced" modulus system, e.g.  $N \text{ MOD } 5 = -2, -1, 0, 1, \text{ or } 2$ . Warning: If  $\text{EXP}$  is already in CRE form when you reset  $\text{MODULUS}$ , then you may need to

re-rat EXP, e.g. EXP:RAT(RATDISREP(EXP)), in order to get correct results. (If MODULUS is set to a positive non-prime integer, this setting will be accepted, but a warning will be given.)

**NEWFAC** Variable  
 default: [FALSE], if TRUE then FACTOR will use the new factoring routines.

**NUM** (*exp*) Function  
 obtains the numerator, exp1, of the rational expression exp = exp1/exp2.

**QUOTIENT** (*p1, p2, var1, ...*) Function  
 computes the quotient of the polynomial p1 divided by the polynomial p2.

**RAT** (*exp, v1, ..., vn*) Function  
 converts exp to CRE form by expanding and combining all terms over a common denominator and cancelling out the greatest common divisor of the numerator and denominator as well as converting floating point numbers to rational numbers within a tolerance of RATEPSILON[2.0E-8]. The variables are ordered according to the v1,...,vn as in RATVARS, if these are specified. RAT does not generally simplify functions other than +, -, \*, /, and exponentiation to an integer power whereas RATSIMP does handle these cases. Note that atoms (numbers and names) in CRE form are not the same as they are in the general form. Thus RAT(X)- X results in RAT(0) which has a different internal representation than 0. RATFAC[FALSE] when TRUE invokes a partially factored form for CRE rational expressions. During rational operations the expression is maintained as fully factored as possible without an actual call to the factor package. This should always save space and may save some time in some computations. The numerator and denominator are still made relatively prime (e.g. RAT((X^2 -1)^4/(X+1)^2); yields (X-1)^4\*(X+1)^2), but the factors within each part may not be relatively prime. RATPRINT[TRUE] if FALSE suppresses the printout of the message informing the user of the conversion of floating point numbers to rational numbers. KEEPFLOAT[FALSE] if TRUE prevents floating point numbers from being converted to rational numbers. (Also see the RATEXPAND and RATSIMP functions.)

(C1) ((X-2\*Y)\*\*4/(X\*\*2-4\*Y\*\*2)\*\*2+1)\*(Y+A)\*(2\*Y+X)  
 /(4\*Y\*\*2+X\*\*2);

$$(Y + A) (2 Y + X) \left( \frac{(X - 2 Y)^4}{(X^2 - 4 Y^2)^2} + 1 \right)$$

(D1) -----  
 4 Y^2 + X^2

(C2) RAT(%,Y,A,X);

(D2) /R/

$$\frac{2 A + 2 Y}{X + 2 Y}$$

**RATALGDENOM**

Variable

default: [TRUE] - if TRUE allows rationalization of denominators wrt. radicals to take effect. To do this one must use CRE form in algebraic mode.

**RATCOEF** (*exp*, *v*, *n*)

Function

returns the coefficient, C, of the expression  $v^{**n}$  in the expression *exp*. *n* may be omitted if it is 1. C will be free (except possibly in a non-rational sense) of the variables in *v*. If no coefficient of this type exists, zero will be returned. RATCOEF expands and rationally simplifies its first argument and thus it may produce answers different from those of COEFF which is purely syntactic. Thus RATCOEF((X+1)/Y+X,X) returns (Y+1)/Y whereas COEFF returns 1. RATCOEF(*exp*,*v*,0), viewing *exp* as a sum, gives a sum of those terms which do not contain *v*. Therefore if *v* occurs to any negative powers, RATCOEF should not be used. Since *exp* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

```
(C1) S:A*X+B*X+5$
(C2) RATCOEF(S,A+B);
(D2)                                X
```

**RATDENOM** (*exp*)

Function

obtains the denominator of the rational expression *exp*. If *exp* is in general form then the DENOM function should be used instead, unless one wishes to get a CRE result.

**RATDENOMDIVIDE**

Variable

default: [TRUE] - if FALSE will stop the splitting up of the terms of the numerator of RATEXPANDED expressions from occurring.

**RATDIFF** (*exp*, *var*)

Function

differentiates the rational expression *exp* (which must be a ratio of polynomials or a polynomial in the variable *var*) with respect to *var*. For rational expressions this is much faster than DIFF. The result is left in CRE form. However, RATDIFF should not be used on factored CRE forms; use DIFF instead for such expressions.

```
(C1) (4*X**3+10*X-11)/(X**5+5);
```

```
(D1)                                3
                                4 X  + 10 X - 11
                                -----
                                5
                                X
```

```
(C2) MODULUS:3$
```

```
(C3) MOD(D1);
```

```
(D3)                                2
                                X  + X - 1
                                -----
```

$$\begin{array}{r}
 \text{(C4) RATDIFF(D1,X);} \\
 \begin{array}{r}
 X^4 + X^3 + X^2 + X + 1 \\
 \\
 X^5 - X^4 - X^3 + X^2 - 1 \\
 \hline
 X^8 - X^7 + X^5 - X^4 + X^3 - X + 1
 \end{array}
 \end{array}$$

**RATDISREP** (*exp*)

Function

changes its argument from CRE form to general form. This is sometimes convenient if one wishes to stop the "contagion", or use rational functions in non-rational contexts. Most CRE functions will work on either CRE or non-CRE expressions, but the answers may take different forms. If RATDISREP is given a non-CRE for an argument, it returns its argument unchanged. See also TOTALDISREP.

**RATEPSILON**

Variable

default: [2.0E-8] - the tolerance used in the conversion of floating point numbers to rational numbers.

**RATEXPAND** (*exp*)

Function

expands *exp* by multiplying out products of sums and exponentiated sums, combining fractions over a common denominator, cancelling the greatest common divisor of the numerator and denominator, then splitting the numerator (if a sum) into its respective terms divided by the denominator. This is accomplished by converting *exp* to CRE form and then back to general form. The switch RATEXPAND, default: [FALSE], if TRUE will cause CRE expressions to be fully expanded when they are converted back to general form or displayed, while if it is FALSE then they will be put into a recursive form. (see RATSIMP) RATDENOMDIVIDE[TRUE] - if FALSE will stop the splitting up of the terms of the numerator of RATEXPANDED expressions from occurring. KEEPFLOAT[FALSE] if set to TRUE will prevent floating point numbers from being rationalized when expressions which contain them are converted to CRE form.

$$\begin{array}{r}
 \text{(C1) RATEXPAND((2*X-3*Y)**3);} \\
 \begin{array}{r}
 - 27 Y^3 + 54 X Y^2 - 36 X^2 Y + 8 X^3 \\
 \text{(D1)} \\
 \text{(C2) (X-1)/(X+1)**2+1/(X-1);} \\
 \begin{array}{r}
 X - 1 \quad 1 \\
 \hline
 (X + 1)^2 \quad X - 1
 \end{array} \\
 \text{(D2)} \\
 \text{(C3) EXPAND(D2);} \\
 \begin{array}{r}
 X \quad 1 \quad 1 \\
 \hline
 X^2 \quad X^2 \quad X - 1
 \end{array} \\
 \text{(D3)}
 \end{array}$$

$$(C4) \text{ RATEXPAND}(D2);$$

$$(D4) \frac{X^2 + 2X + 1}{X^3 + X^2 - X - 1} + \frac{X^2 + 2X + 1}{X^3 + X^2 - X - 1}$$

**RATFAC**

Variable

default: [FALSE] - when TRUE invokes a partially factored form for CRE rational expressions. During rational operations the expression is maintained as fully factored as possible without an actual call to the factor package. This should always save space and may save some time in some computations. The numerator and denominator are still made relatively prime, for example  $\text{RAT}((X^2 - 1)^4 / (X + 1)^2)$ ; yields  $(X - 1)^4 (X + 1)^2$ , but the factors within each part may not be relatively prime. In the CTENSR (Component Tensor Manipulation) Package, if RATFAC is TRUE, it causes the Ricci, Einstein, Riemann, and Weyl tensors and the Scalar Curvature to be factored automatically. \*\* This should only be set for cases where the tensorial components are known to consist of few terms \*\*. Note: The RATFAC and RATWEIGHT schemes are incompatible and may not both be used at the same time.

**RATNUMER** (*exp*)

Function

obtains the numerator of the rational expression *exp*. If *exp* is in general form then the NUM function should be used instead, unless one wishes to get a CRE result.

**RATNUMP** (*exp*)

Function

is TRUE if *exp* is a rational number (includes integers) else FALSE.

**RATP** (*exp*)

Function

is TRUE if *exp* is in CRE or extended CRE form else FALSE.

**RATPRINT**

Variable

default: [TRUE] - if FALSE suppresses the printout of the message informing the user of the conversion of floating point numbers to rational numbers.

**RATSIMP** (*exp*)

Function

rationaly" simplifies (similar to RATEXPAND) the expression *exp* and all of its subexpressions including the arguments to non-rational functions. The result is returned as the quotient of two polynomials in a recursive form, i.e. the coefficients of the main variable are polynomials in the other variables. Variables may, as in RATEXPAND, include non-rational functions (e.g.  $\text{SIN}(X^{**2} + 1)$ ) but with RATSIMP, the arguments to non-rational functions are rationally simplified. Note that RATSIMP is affected by some of the variables which affect RATEXPAND. RATSIMP(*exp*, *v1*, *v2*, ..., *vn*) - enables rational simplification with the specification of variable ordering as in RATVARS. RATSIMPEXPONS[FALSE] - if TRUE will cause exponents of expressions to be RATSIMPed automatically during simplification.

```

(C1) SIN(X/(X^2+X))=%E^((LOG(X)+1)**2-LOG(X)**2);
(D1)          X          (LOG(X) + 1)2 - LOG(X)2
      SIN(-----) = %E
            2
            X + X
(C2) RATSIMP(%);
(D2)          1          2
      SIN(-----) = %E X
            X + 1
(C3) ((X-1)**(3/2)-(X+1)*SQRT(X-1))/SQRT((X-1)*(X+1));
(D3)          3/2
      (X - 1) - SQRT(X - 1) (X + 1)
      -----
            SQRT(X - 1) SQRT(X + 1)
(C4) RATSIMP(%);
(D4)          2
      - -----
            SQRT(X + 1)
(C5) X**(A+1/A),RATSIMPEXPONS:TRUE;
(D5)          2
      A + 1
      -----
            A
      X

```

**RATSIMPEXPONS**

Variable

default: [FALSE] - if TRUE will cause exponents of expressions to be RATSIMPed automatically during simplification.

**RATSUBST** (*a*, *b*, *c*)

Function

substitutes *a* for *b* in *c*. *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 would return Z+A. RATSUBSTFLAG[FALSE] if TRUE permits RATSUBST to make substitutions like U for SQRT(X) in X. Do EXAMPLE(RATSUBST); for examples.

**RATVARS** (*var1*, *var2*, ..., *varn*)

Function

forms its *n* arguments into a list in which the rightmost variable *varn* will be the main variable of future rational expressions in which it occurs, and the other variables will follow in sequence. If a variable is missing from the RATVARS list, it will be given lower priority than the leftmost variable *var1*. The arguments to RATVARS can be either variables or non-rational functions (e.g. SIN(X)). The variable RATVARS is a list of the arguments which have been given to this function.



**RATWEIGHT** ( $v1, w1, \dots, vn, wn$ ) Function

assigns a weight of  $w_i$  to the variable  $v_i$ . This causes a term to be replaced by 0 if its weight exceeds the value of the variable `RATWTLVL` [default is `FALSE` which means no truncation]. The weight of a term is the sum of the products of the weight of a variable in the term times its power. Thus the weight of  $3*v1^{**2}*v2$  is  $2*w1+w2$ . This truncation occurs only when multiplying or exponentiating CRE forms of expressions.

```
(C5) RATWEIGHT(A,1,B,1);
(D5)                                     [[B, 1], [A, 1]]
(C6) EXP1:RAT(A+B+1)$
(C7) %**2;

(D7)/R/
      2          2
      B  + (2 A + 2) B + A  + 2 A + 1
(C8) RATWTLVL:1$
(C9) EXP1**2;
(D9)/R/
      2 B + 2 A + 1
```

Note: The `RATFAC` and `RATWEIGHT` schemes are incompatible and may not both be used at the same time.

**RATWEIGHTS** Variable

- a list of weight assignments (set up by the `RATWEIGHT` function), `RATWEIGHTS`; or `RATWEIGHT()`; will show you the list.

```
KILL(...,RATWEIGHTS)
```

and

```
SAVE(...,RATWEIGHTS);
```

both work.

**RATWEYL** Variable

default: `[]` - one of the switches controlling the simplification of components of the Weyl conformal tensor; if `TRUE`, then the components will be rationally simplified; if `FACRAT` is `TRUE` then the results will be factored as well.

**RATWTLVL** Variable

default: `[FALSE]` - used in combination with the `RATWEIGHT` function to control the truncation of rational (CRE form) expressions (for the default value of `FALSE`, no truncation occurs).

**REMAINDER** ( $p1, p2, var1, \dots$ ) Function

computes the remainder of the polynomial  $p1$  divided by the polynomial  $p2$ .

**RESULTANT** ( $p1, p2, var$ ) Function

computes the resultant of the two polynomials  $p1$  and  $p2$ , eliminating the variable  $var$ . The resultant is a determinant of the coefficients of  $var$  in  $p1$  and  $p2$  which equals zero if and only if  $p1$  and  $p2$  have a non-constant factor in common. If  $p1$  or  $p2$  can be factored, it may be desirable to call `FACTOR` before calling `RESULTANT`. `RESULTANT[SUBRES]` - controls which algorithm will be used to compute the resultant.

SUBRES for subresultant prs [the default], MOD for modular resultant algorithm, and RED for reduced prs. On most problems SUBRES should be best. On some large degree univariate or bivariate problems MOD may be better. Another alternative is the BEZOUT command which takes the same arguments as RESULTANT and returns a matrix. DETERMINANT of this matrix is the desired resultant.

**SAVEFACTORS**

Variable

default: [FALSE] - if TRUE causes the factors of an expression which is a product of factors to be saved by certain functions in order to speed up later factorizations of expressions containing some of the same factors.

**SQFR** (*exp*)

Function

is similar to FACTOR except that the polynomial factors are "square-free." That is, they have factors only of degree one. This algorithm, which is also used by the first stage of FACTOR, utilizes the fact that a polynomial has in common with its  $n$ th derivative all its factors of degree  $> n$ . Thus by taking gcds with the polynomial of the derivatives with respect to each variable in the polynomial, all factors of degree  $> 1$  can be found.

(C1)  $\text{SQFR}(4*X^{**4}+4*X^{**3}-3*X^{**2}-4*X-1);$

(D1)  $(X^2 - 1) (2 X + 1)^2$

**TELLRAT** (*poly*)

Function

adds to the ring of algebraic integers known to MACSYMA, the element which is the solution of the polynomial with integer coefficients. MACSYMA initially knows about %I and all roots of integers. TELLRAT(X); means substitute 0 for X in rational functions. There is a command UNTELLRAT which takes kernels and removes TELLRAT properties. When TELLRATing a multivariate polynomial, e.g. TELLRAT(X<sup>2</sup>-Y<sup>2</sup>);, there would be an ambiguity as to whether to substitute Y<sup>2</sup> for X<sup>2</sup> or vice versa. The system will pick a particular ordering, but if the user wants to specify which, e.g. TELLRAT(Y<sup>2</sup>=X<sup>2</sup>); provides a syntax which says replace Y<sup>2</sup> by X<sup>2</sup>. TELLRAT and UNTELLRAT both can take any number of arguments, and TELLRAT(); returns a list of the current substitutions. Note: When you TELLRAT reducible polynomials, you want to be careful not to attempt to rationalize a denominator with a zero divisor. E.g. TELLRAT(W<sup>3</sup>-1)\$ ALGEBRAIC:TRUE\$ RAT(1/(W<sup>2</sup>-W)); will give "quotient by zero". This error can be avoided by setting RATALGDENOM:FALSE\$. ALGEBRAIC[FALSE] must be set to TRUE in order for the simplification of algebraic integers to take effect. Do EXAMPLE(TELLRAT); for examples.

**TOTALDISREP** (*exp*)

Function

converts every subexpression of *exp* from CRE to general form. If *exp* is itself in CRE form then this is identical to RATDISREP but if not then RATDISREP would return *exp* unchanged while TOTALDISREP would "totally disrep" it. This is useful for ratdisrepping expressions e.g., equations, lists, matrices, etc. which have some subexpressions in CRE form.

**UNTELLRAT** (*x*)

Function

takes kernels and removes TELLRAT properties.

## 12 Constants

### 12.1 Definitions for Constants

<b>E</b>	- The base of natural logarithms, $e$ , is represented in MACSYMA as %E.	Variable
<b>FALSE</b>	- the Boolean constant, false. (NIL in LISP)	Variable
<b>MINF</b>	- real minus infinity.	Variable
<b>PI</b>	- "pi" is represented in MACSYMA as %PI.	Variable
<b>TRUE</b>	- the Boolean constant, true. (T in LISP)	Variable



## 13 Logarithms

### 13.1 Definitions for Logarithms

- LOG** (*X*) Function  
 the natural logarithm.  
**LOGEXPAND**[TRUE] - causes  $\text{LOG}(A^B)$  to become  $B \cdot \text{LOG}(A)$ . If it is set to ALL,  $\text{LOG}(A \cdot B)$  will also simplify to  $\text{LOG}(A) + \text{LOG}(B)$ . If it is set to SUPER, then  $\text{LOG}(A/B)$  will also simplify to  $\text{LOG}(A) - \text{LOG}(B)$  for rational numbers  $a/b$ ,  $a \neq 1$ . ( $\text{LOG}(1/B)$ , for  $B$  integer, always simplifies.) If it is set to FALSE, all of these simplifications will be turned off.  
**LOGSIMP**[TRUE] - if FALSE then no simplification of  $\%E$  to a power containing LOG's is done.  
**LOGNUMER**[FALSE] - if TRUE then negative floating point arguments to LOG will always be converted to their absolute value before the log is taken. If NUMER is also TRUE, then negative integer arguments to LOG will also be converted to their absolute value.  
**LOGNEGINT**[FALSE] - if TRUE implements the rule  $\text{LOG}(-n) \rightarrow \text{LOG}(n) + \%i \cdot \%pi$  for  $n$  a positive integer.  
**\%E\_TO\_NUMLOG**[FALSE] - when TRUE, " $r$ " some rational number, and " $x$ " some expression,  $\%E^{(r \cdot \text{LOG}(x))}$  will be simplified into  $x^r$ . It should be noted that the RADCAN command also does this transformation, and more complicated transformations of this ilk as well. The LOGCONTRACT command "contracts" expressions containing LOG.
- LOGABS** Variable  
 default: [FALSE] - when doing indefinite integration where logs are generated, e.g. INTEGRATE(1/X,X), the answer is given in terms of LOG(ABS(...)) if LOGABS is TRUE, but in terms of LOG(...) if LOGABS is FALSE. For definite integration, the LOGABS:TRUE setting is used, because here "evaluation" of the indefinite integral at the endpoints is often needed.
- LOGARC** Variable  
 default: [FALSE] - if TRUE will cause the inverse circular and hyperbolic functions to be converted into logarithmic form. LOGARC(exp) will cause this conversion for a particular expression without setting the switch or having to re-evaluate the expression with EV.
- LOGCONCOEFFP** Variable  
 default:[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)).

**LOGCONTRACT** (*exp*) Function

recursively scans an *exp*, transforming subexpressions of the form  $a_1 \cdot \text{LOG}(b_1) + a_2 \cdot \text{LOG}(b_2) + c$  into  $\text{LOG}(\text{RATSIMP}(b_1^{a_1} \cdot b_2^{a_2})) + c$

(C1)  $2*(A*\text{LOG}(X) + 2*A*\text{LOG}(Y))\$$

(C2)  $\text{LOGCONTRACT}(\%);$

(D3)  $A \text{LOG}(X^2 Y^4)$

If you do  $\text{DECLARE}(N, \text{INTEGER});$  then  $\text{LOGCONTRACT}(2*A^N*\text{LOG}(X));$  gives  $A*\text{LOG}(X^{2*N})$ . The coefficients that "contract" in this manner are those such as the 2 and the N here which satisfy  $\text{FEATUREP}(\text{coeff}, \text{INTEGER})$ . The user can control which coefficients are contracted by setting the option  $\text{LOGCONCOEFFP}[\text{FALSE}]$  to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do  $\text{LOGCONCOEFFP}:\text{LOGCONFUN}\$$   $\text{LOGCONFUN}(M):=\text{FEATUREP}(M, \text{INTEGER}) \text{ OR } \text{RATNUMP}(M)\$$  . Then  $\text{LOGCONTRACT}(1/2*\text{LOG}(X));$  will give  $\text{LOG}(\text{SQRT}(X))$ .

**LOGEXPAND** Variable

default:  $[\text{TRUE}]$  - causes  $\text{LOG}(A^B)$  to become  $B*\text{LOG}(A)$ . If it is set to **ALL**,  $\text{LOG}(A*B)$  will also simplify to  $\text{LOG}(A)+\text{LOG}(B)$ . If it is set to **SUPER**, then  $\text{LOG}(A/B)$  will also simplify to  $\text{LOG}(A)-\text{LOG}(B)$  for rational numbers  $a/b$ ,  $a \neq 1$ . ( $\text{LOG}(1/B)$ , for B integer, always simplifies.) If it is set to **FALSE**, all of these simplifications will be turned off.

**LOGNEGINT** Variable

default:  $[\text{FALSE}]$  - if **TRUE** implements the rule  $\text{LOG}(-n) \rightarrow \text{LOG}(n) + \%i*\%pi$  for n a positive integer.

**LOGNUMER** Variable

default:  $[\text{FALSE}]$  - if **TRUE** then negative floating point arguments to **LOG** will always be converted to their absolute value before the log is taken. If **NUMER** is also **TRUE**, then negative integer arguments to **LOG** will also be converted to their absolute value.

**LOGSIMP** Variable

default:  $[\text{TRUE}]$  - if **FALSE** then no simplification of  $\%E$  to a power containing **LOG**'s is done.

**PLOG** (*X*) Function

the principal branch of the complex-valued natural logarithm with  $-\%PI < \text{CARG}(X) \leq +\%PI$  .

**POLARFORM** (*exp*) Function

returns  $R*\%E^{( \%I*\text{THETA} )}$  where R and THETA are purely real.

## 14 Trigonometric

### 14.1 Introduction to Trigonometric

- MACSYMA has many Trig functions defined. Not all Trig identities are programmed, but it is possible for the user to add many of them using the pattern matching capabilities of the system. The Trig functions defined in MACSYMA are: ACOS, ACOSH, ACOT, ACOTH, ACSC, ACSCH, ASEC, ASECH, ASIN, ASINH, ATAN, ATANH, COS, COSH, COT, COth, CSC, CSCH, SEC, SECH, SIN, SINH, TAN, and TANH. There are a number of commands especially for handling Trig functions, see TRIGEXPAND, TRIGREDUCE, and the switch TRIGSIGN. Two SHARE packages extend the simplification rules built into MACSYMA, NTRIG and ATRIG1. Do DESCRIBE(cmd) for details.

### 14.2 Definitions for Trigonometric

<b>ACOS</b> - Arc Cosine	Function
<b>ACOSH</b> - Hyperbolic Arc Cosine	Function
<b>ACOT</b> - Arc Cotangent	Function
<b>ACOTH</b> - Hyperbolic Arc Cotangent	Function
<b>ACSC</b> - Arc Cosecant	Function
<b>ACSCH</b> - Hyperbolic Arc Cosecant	Function
<b>ASEC</b> - Arc Secant	Function
<b>ASECH</b> - Hyperbolic Arc Secant	Function
<b>ASIN</b> - Arc Sine	Function
<b>ASINH</b> - Hyperbolic Arc Sine	Function



<b>ATAN</b>	Function
- Arc Tangent	
<b>ATAN2 (Y,X)</b>	Function
yields the value of ATAN(Y/X) in the interval $-\%PI$ to $\%PI$ .	
<b>ATANH</b>	Function
- Hyperbolic Arc Tangent	
<b>ATRIG1</b>	Function
- SHARE1;ATRIG1 FASL contains several additional simplification rules for inverse trig functions. Together with rules already known to Macsyma, the following angles are fully implemented: 0, $\%PI/6$ , $\%PI/4$ , $\%PI/3$ , and $\%PI/2$ . Corresponding angles in the other three quadrants are also available. Do LOAD(ATRIG1); to use them.	
<b>COS</b>	Function
- Cosine	
<b>COSH</b>	Function
- Hyperbolic Cosine	
<b>COT</b>	Function
- Cotangent	
<b>COTH</b>	Function
- Hyperbolic Cotangent	
<b>CSC</b>	Function
- Cosecant	
<b>CSCH</b>	Function
- Hyperbolic Cosecant	
<b>HALFANGLES</b>	Variable
default: [FALSE] - if TRUE causes half-angles to be simplified away.	
<b>SEC</b>	Function
- Secant	
<b>SECH</b>	Function
- Hyperbolic Secant	
<b>SIN</b>	Function
- Sine	

**SINH** Function  
- Hyperbolic Sine

**TAN** Function  
- Tangent

**TANH** Function  
- Hyperbolic Tangent

**TRIGEXPAND** (*exp*) Function  
expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in *exp*. For best results, *exp* should be expanded. To enhance user control of simplification, this function expands only one level at a time, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the switch TRIGEXPAND:TRUE. TRIGEXPAND default: [FALSE] - if TRUE causes expansion of all expressions containing SINS and COSs occurring subsequently. HALFANGLES[FALSE] - if TRUE causes half-angles to be simplified away. TRIGEXPANDPLUS[TRUE] - controls the "sum" rule for TRIGEXPAND, expansion of sums (e.g. SIN(X+Y)) will take place only if TRIGEXPANDPLUS is TRUE. TRIGEXPANDTIMES[TRUE] - controls the "product" rule for TRIGEXPAND, expansion of products (e.g. SIN(2\*X)) will take place only if TRIGEXPANDTIMES is TRUE.

(C1)  $X + \frac{\sin(3X)}{\sin(X)}$ , TRIGEXPAND=TRUE, EXPAND;

(D1)  $-\sin(X) + 3\cos(X) + X$

(C2) TRIGEXPAND(SIN(10\*X+Y));

(D2)  $\cos(10X)\sin(Y) + \sin(10X)\cos(Y)$

**TRIGEXPANDPLUS** Variable  
default: [TRUE] - controls the "sum" rule for TRIGEXPAND. Thus, when the TRIGEXPAND command is used or the TRIGEXPAND switch set to TRUE, expansion of sums (e.g. SIN(X+Y)) will take place only if TRIGEXPANDPLUS is TRUE.

**TRIGEXPANDTIMES** Variable  
default: [TRUE] - controls the "product" rule for TRIGEXPAND. Thus, when the TRIGEXPAND command is used or the TRIGEXPAND switch set to TRUE, expansion of products (e.g. SIN(2\*X)) will take place only if TRIGEXPANDTIMES is TRUE.

**TRIGINVERSES** Variable  
default: [ALL] - controls the simplification of the composition of trig and hyperbolic functions with their inverse functions: If ALL, both e.g. ATAN(TAN(X)) and TAN(ATAN(X)) simplify to X. If TRUE, the arcfunction(function(x)) simplification is turned off. If FALSE, both the arcfun(fun(x)) and fun(arcfun(x)) simplifications are turned off.

**TRIGREDUCE** (*exp, var*) Function

combines products and powers of trigonometric and hyperbolic SINS and COSs of var into those of multiples of var. It also tries to eliminate these functions when they occur in denominators. If var is omitted then all variables in exp are used. Also see the POISSIMP function (6.6).

```
(C4) TRIGREDUCE(-SIN(X)^2+3*COS(X)^2+X);
```

```
(D4)          2 COS(2 X) + X + 1
```

The trigonometric simplification routines will use declared information in some simple cases. Declarations about variables are used as follows, e.g.

```
(C5) DECLARE(J, INTEGER, E, EVEN, 0, ODD)$
```

```
(C6) SIN(X + (E + 1/2)*%PI)$
```

```
(D6)          COS(X)
```

```
(C7) SIN(X + (0 + 1/2) %PI);
```

```
(D7)          - COS(X)
```

**TRIGSIGN** Variable

default: [TRUE] - if TRUE permits simplification of negative arguments to trigonometric functions. E.g., SIN(-X) will become -SIN(X) only if TRIGSIGN is TRUE.

**TRIGSIMP** (*expr*) Function

employs the identities  $\sin(x)^2 + \cos(x)^2 = 1$  and  $\cosh(x)^2 - \sinh(x)^2 = 1$  to simplify expressions containing tan, sec, etc. to sin, cos, sinh, cosh so that further simplification may be obtained by using TRIGREDUCE on the result. Some examples may be seen by doing DEMO("trgsmp.dem"); . See also the TRIGSUM function.

**TRIGRAT** (*trigexp*) Function

gives a canonical simplified quasilinear form of a trigonometrical expression; trigexp is a rational fraction of several sin, cos or tan, the arguments of them are linear forms in some variables (or kernels) and  $\%pi/n$  (n integer) with integer coefficients. The result is a simplified fraction with numerator and denominator linear in sin and cos. Thus TRIGRAT linearize always when it is possible.(written by D. Lazard).

```
(c1) trigrat(sin(3*a)/sin(a+%pi/3));
```

```
(d1)          sqrt(3) sin(2 a) + cos(2 a) - 1
```

Here is another example (for which the function was intended); see [Davenport, Siret, Tournier, Calcul Formel, Masson (or in english, Addison-Wesley), section 1.5.5, Morley theorem). Timings are on VAX 780.

```
(c4)  c:%pi/3-a-b;
```

```
(d4)  %pi
      - b - a + ---
           3
```

```
(c5)  bc:sin(a)*sin(3*c)/sin(a+b);
```

```

      sin(a) sin(3 b + 3 a)
(d5)  -----
      sin(b + a)

(c6)  ba:bc,c=a,a=c$

(c7)  ac2:ba^2+bc^2-2*bc*ba*cos(b);

      2      2
      sin (a) sin (3 b + 3 a)
(d7)  -----
      2
      sin (b + a)

%pi
  2 sin(a) sin(3 a) cos(b) sin(b + a - ---) sin(3 b + 3 a)
  3
  -----
  %pi
  sin(a - ---) sin(b + a)
  3

      2      2      %pi
      sin (3 a) sin (b + a - ---)
      3
+ -----
      2      %pi
      sin (a - ---)
      3

(c9)  trigrat(ac2);
Totaltime= 65866 msec.  Gctime= 7716 msec.

(d9)
- (sqrt(3) sin(4 b + 4 a) - cos(4 b + 4 a)
- 2 sqrt(3) sin(4 b + 2 a)
+ 2 cos(4 b + 2 a) - 2 sqrt(3) sin(2 b + 4 a) + 2 cos(2 b + 4 a)
+ 4 sqrt(3) sin(2 b + 2 a) - 8 cos(2 b + 2 a) - 4 cos(2 b - 2 a)
+ sqrt(3) sin(4 b) - cos(4 b) - 2 sqrt(3) sin(2 b) + 10 cos(2 b)
+ sqrt(3) sin(4 a) - cos(4 a) - 2 sqrt(3) sin(2 a) + 10 cos(2 a)
- 9)/4

```



## 15 Special Functions

### 15.1 Introduction to Special Functions

`[index](expr)` - *Bessel Funct 1st Kind (in SPECINT)* %J

`[index](expr)` %K  
 Bessel Funct 2nd Kind (in SPECINT)  
 Constant, in ODE2

### 15.2 GAMALG

- A Dirac gamma matrix algebra program which takes traces of and does manipulations on gamma matrices in n dimensions. It may be loaded into MACSYMA by `LOAD-FILE("gam")`; A preliminary manual is contained in the file `SHARE;GAM USAGE` and may be printed using `PRINTFILE(GAM,USAGE,SHARE)`;

### 15.3 SPECINT

- The Hypergeometric Special Functions Package HYPGEO is still under development. At the moment it will find the Laplace Transform or rather, the integral from 0 to INF of some special functions or combinations of them. The factor, `EXP(-P*var)` must be explicitly stated. The syntax is as follows: `SPECINT(EXP(-P*var)*expr,var)`; where var is the variable of integration and expr may be any expression containing special functions (at your own risk). Special function notation follows:

<code>%J[index](expr)</code>	Bessel Funct 1st Kind
<code>%K[index](expr)</code>	" " 2nd Kind
<code>%I[ ]( )</code>	Modified Bessel
<code>%HE[ ]( )</code>	Hermite Poly
<code>%P[ ]( )</code>	Legendre Funct
<code>%Q[ ]( )</code>	Legendre of second kind
<code>HSTRUVE[ ]( )</code>	Struve H Function
<code>LSTRUVE[ ]( )</code>	" L Function
<code>%F[ ]([ ], [ ], expr)</code>	Hypergeometric Function
<code>GAMMA()</code>	
<code>GAMMAGREEK()</code>	
<code>GAMMAINCOMPLETE()</code>	
<code>SLOMMEL</code>	
<code>%M[ ]( )</code>	Whittaker Funct 1st Kind
<code>%W[ ]( )</code>	" " 2nd "

For a better feeling for what it can do, do `DEMO(HYPGEO,DEMO,SHARE1)`; .

## 15.4 Definitions for Special Functions

- AIRY** ( $X$ ) Function  
 returns the Airy function  $Ai$  of real argument  $X$ . The file `SHARE1;AIRY FASL` contains routines to evaluate the Airy functions  $Ai(X)$ ,  $Bi(X)$ , and their derivatives  $dAi(X)$ ,  $dBi(X)$ .  $Ai$  and  $Bi$  satisfy the AIRY eqn  $\text{diff}(y(x),x,2)-x*y(x)=0$ . Read `SHARE1;AIRY USAGE` for details.
- ASYMP** Function  
 - A preliminary version of a program to find the asymptotic behavior of Feynman diagrams has been installed on the `SHARE1;` directory. For further information, see the file `SHARE1;ASYMP USAGE`. (For Asymptotic Analysis functions, see `ASYMPA`.)
- ASYMPA** Function  
 - Asymptotic Analysis - The file `SHARE1;ASYMPA >` contains simplification functions for asymptotic analysis, including the big-O and little-o functions that are widely used in complexity analysis and numerical analysis. Do `BATCH("asympa.mc");` . (For asymptotic behavior of Feynman diagrams, see `ASYMP`.)
- BESSEL** ( $Z,A$ ) Function  
 returns the Bessel function  $J$  for complex  $Z$  and real  $A > 0.0$  . Also an array `BESSELARRAY` is set up such that `BESSELARRAY[I] = J[I+A- ENTIER(A)](Z)`.
- BETA** ( $X, Y$ ) Function  
 same as `GAMMA(X)*GAMMA(Y)/GAMMA(X+Y)`.
- GAMMA** ( $X$ ) Function  
 the gamma function. `GAMMA(I)=(I-1)!` for  $I$  a positive integer. For the Euler-Mascheroni constant, see `%GAMMA`. See also the `MAKEGAMMA` function. The variable `GAMMALIM[1000000]` (which see) controls simplification of the gamma function.
- GAMMALIM** Variable  
 default: `[1000000]` controls simplification of the gamma function for integral and rational number arguments. If the absolute value of the argument is not greater than `GAMMALIM`, then simplification will occur. Note that the `FACTLIM` switch controls simplification of the result of `GAMMA` of an integer argument as well.
- INTOPOIS** ( $A$ ) Function  
 converts  $A$  into a Poisson encoding.
- MAKEFACT** ( $exp$ ) Function  
 transforms occurrences of binomial, gamma, and beta functions in `exp` to factorials.
- MAKEGAMMA** ( $exp$ ) Function  
 transforms occurrences of binomial, factorial, and beta functions in `exp` to gamma functions.

**NUMFACTOR** (*exp*) Function  
 gives the numerical factor multiplying the expression *exp* which should be a single term. If the gcd of all the terms in a sum is desired the CONTENT function may be used.

```
(C1) GAMMA(7/2);
(D1)          15 Sqrt(%PI)
          -----
                8

(C2) NUMFACTOR(%);
(D2)          15
          --
                8
```

**OUTOFOPOIS** (*A*) Function  
 converts *A* from Poisson encoding to general representation. If *A* is not in Poisson form, it will make the conversion, i.e. it will look like the result of OUTOFOPOIS(INTOPOIS(*A*)). This function is thus a canonical simplifier for sums of powers of SIN's and COS's of a particular type.

**POISDIFF** (*A*, *B*) Function  
 differentiates *A* with respect to *B*. *B* must occur only in the trig arguments or only in the coefficients.

**POISEXPT** (*A*, *B*) Function  
*B* a positive integer) is functionally identical to INTOPOIS(*A*\*\**B*).

**POISINT** (*A*, *B*) Function  
 integrates in a similarly restricted sense (to POISDIFF). Non-periodic terms in *B* are dropped if *B* is in the trig arguments.

**POISLIM** Variable  
 default: [5] - determines the domain of the coefficients in the arguments of the trig functions. The initial value of 5 corresponds to the interval  $[-2^{(5-1)+1}, 2^{(5-1)}]$ , or  $[-15, 16]$ , but it can be set to  $[-2^{(n-1)+1}, 2^{(n-1)}]$ .

**POISMAP** (*series*, *sinfn*, *cosfn*) Function  
 will map the functions *sinfn* on the sine terms and *cosfn* on the cosine terms of the poisson series given. *sinfn* and *cosfn* are functions of two arguments which are a coefficient and a trigonometric part of a term in series respectively.

**POISPLUS** (*A*, *B*) Function  
 is functionally identical to INTOPOIS(*A*+*B*).

**POISSIMP** (*A*) Function  
 converts *A* into a Poisson series for *A* in general representation.



**POISSON**

special symbol

- The Symbol /P/ follows the line label of Poisson series expressions.

**POISSUBST** (A, B, C)

Function

substitutes A for B in C. C is a Poisson series. (1) Where B is a variable U, V, W, X, Y, or Z then A must be an expression linear in those variables (e.g.  $6*U+4*V$ ). (2) Where B is other than those variables, then A must also be free of those variables, and furthermore, free of sines or cosines. POISSUBST(A, B, C, D, N) is a special type of substitution which operates on A and B as in type (1) above, but where D is a Poisson series, expands COS(D) and SIN(D) to order N so as to provide the result of substituting A+D for B in C. The idea is that D is an expansion in terms of a small parameter. For example, POISSUBST(U,V,COS(V),E,3) results in  $COS(U)*(1-E^2/2) - SIN(U)*(E-E^3/6)$ .

**POISTIMES** (A, B)

Function

is functionally identical to INTOPOIS(A\*B).

**POISTRIM** ()

Function

is a reserved function name which (if the user has defined it) gets applied during Poisson multiplication. It is a predicate function of 6 arguments which are the coefficients of the U, V, ..., Z in a term. Terms for which POISTRIM is TRUE (for the coefficients of that term) are eliminated during multiplication.

**PRINTPOIS** (A)

Function

prints a Poisson series in a readable format. In common with OUTOFFPOIS, it will convert A into a Poisson encoding first, if necessary.

**PSI** (X)

Function

derivative of LOG(GAMMA(X)). At this time, MACSYMA does not have numerical evaluation capabilities for PSI. For information on the PSI[N](X) notation, see POLYGAMMA.

## 16 Orthogonal Polynomials

### 16.1 Introduction to Orthogonal Polynomials

The `specfun` package, located in the `share` directory, contains Maxima code for the evaluation of all orthogonal polynomials listed in Chapter 22 of Abramowitz and Stegun. These include Chebyshev, Laguerre, Hermite, Jacobi, Legendre, and ultraspherical (Gegenbauer) polynomials. Additionally, `specfun` contains code for spherical Bessel, spherical Hankel, and spherical harmonic functions.

The following table lists each function in `specfun`, its Maxima name, restrictions on its arguments ( $m$  and  $n$  must be integers), and a reference to the algorithm `specfun` uses to evaluate it. With few exceptions, `specfun` follows the conventions of Abramowitz and Stegun. Before you use `specfun`, check that `specfun`'s conventions match your expectations.

A&S refers to Abramowitz and Stegun, *Handbook of Mathematical Functions* (10th printing, December 1972), G&R to Gradshteyn and Ryzhik, *Table of Integrals, Series, and Products* (1980 corrected and enlarged edition), and Merzbacher to *Quantum Mechanics* (2ed, 1970).

<i>Function</i>	<i>Maxima Name</i>	<i>Restrictions</i>	<i>Reference(s)</i>
Chebyshev T	<code>chebyshev_t(n, x)</code>	$n > -1$	A&S 22.5.31
Chebyshev U	<code>chebyshev_u(n, x)</code>	$n > -1$	A&S 22.5.32
generalized Laguerre	<code>gen_laguerre(n,a,x)</code>	$n > -1$	A&S page 789
Laguerre	<code>laguerre(n,x)</code>	$n > -1$	A&S 22.5.67
Hermite	<code>hermite(n,x)</code>	$n > -1$	A&S 22.4.40, 22.5.41
Jacobi	<code>jacobi_p(n,a,b,x)</code>	$n > -1, a, b > -1$	A&S page 789
associated Legendre P	<code>assoc_legendre_p(n,m,x)</code>	$n > -1$	A&S 22.5.37, 8.6.6, 8.2.5
associated Legendre Q	<code>assoc_legendre_q(n,m,x)</code>	$n > -1, m > -1$	G & R 8.706
Legendre P	<code>legendre_p(n,m,x)</code>	$n > -1$	A&S 22.5.35
Legendre Q	<code>legendre_q(n,m,x)</code>	$n > -1$	A&S 8.6.19
spherical Hankel 1st	<code>spherical_hankel1(n, x)</code>	$n > -1$	A&S 10.1.36
spherical Hankel 2nd	<code>spherical_hankel2(n, x)</code>	$n > -1$	A&S 10.1.17
spherical Bessel J	<code>spherical_bessel_j(n,x)</code>	$n > -1$	A&S 10.1.8, 10.1.15
spherical Bessel Y	<code>spherical_bessel_y(n,x)</code>	$n > -1$	A&S 10.1.9, 10.1.15
spherical harmonic	<code>spherical_harmonic(n,m,x,y)</code>	$1,  m  < = n$	Merzbacher 9.64
ultraspherical (Gegenbauer)	<code>ultraspherical(n,a,x)</code>	$n > -1$	A&S 22.5.27

The `specfun` package is primarily intended for symbolic computation. It is hoped that it gives accurate floating point results as well; however, no claims are made that the algorithms are well suited for numerical evaluation. Some effort, however, has been made to provide good numerical performance. When all arguments, except for the order, are floats (but not bfloats), many functions in `specfun` call a float modedecleared version of the Jacobi function. This greatly speeds floating point evaluation of the orthogonal polynomials.

specfun handles most domain errors by returning an unevaluated function. No attempt has been made to define simplification rules (based on recursion relations) for unevaluated functions. Users should be aware that it is possible for an expression involving sums of unevaluated special functions to vanish, yet Maxima is unable to reduce it to zero. Be careful.

To access functions in specfun, you must first load specfun.o. Alternatively, you may append autoload statements to your init.lisp file (located in your working directory). To autoload the hermite function, for example, append

```
(defprop |$hermite| #"specfun.o" autoload)
(add2lnc '|$hermite| $props)
```

to your init.lisp file. An example use of specfun is

```
(c1) load("specfun.o")$
(c2) [hermite(0,x),hermite(1,x),hermite(2,x)];
(d2) [1,2*x,-2*(1-2*x^2)]
(c3) diff(hermite(n,x),x);
(d3) 2*n*hermite(n-1,x)
```

When using the compiled version of specfun, be especially careful to use the correct number of function arguments; calling them with too few arguments may generate a fatal error messages. For example

```
(c1) load("specfun")$
/* chebyshev_t requires two arguments. */
(c2) chebyshev_t(8);
Error: Caught fatal error [memory may be damaged]
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by MMAPCAR.
Broken at SIMPLIFY. Type :H for Help.
```

Maxima code translated into Lisp handles such errors more gracefully. If specfun.LISP is installed on your machine, the same computation results in a clear error message. For example

```
(c1) load("specfun.LISP")$
(c2) chebyshev_t(8);
Error: Expected 2 args but received 1 args
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by MACSYMA-TOP-LEVEL.
Broken at |$CHEBYSHEV_T|. Type :H for Help.
```

Generally, compiled code runs faster than translated code; however, translated code may be better for program development.

For some functions, when the order is symbolic but has been declared to be an integer, specfun will return a series representation. (The series representation is not used by specfun for any computations.) You may use this feature to find symbolic values for special values orthogonal polynomials. An example:

```
(c1) load("specfun")$
(c2) legendre_p(n,1);
(d2) legendre_p(n, 1)
/* Declare n to be an integer; now legendre_p(n,1) evaluates to 1. */
(c3) declare(n,integer)$
```

```
(c4) legendre_p(n,1);
(d4)      1
(c5) ultraspherical(n,3/2,1);
(d4)      (n+1)*gamma (n+3) / (2*gamma (n+2))
```

Although the preceding example doesn't show it, two terms of the sum are added outside the summation. Removing these two terms avoids errors associated with  $0^0$  terms in a sum that should evaluate to 1, but evaluate to 0 in a Maxima summation. Because the sum index runs from 1 to  $n - 1$ , the lower sum index will exceed the upper sum index when  $n = 0$ ; setting `sumhack` to true provides a fix. For example:

```
(c1) load("specfun.o")$
(c2) declare(n,integer)$
(c3) e : legendre_p(n,x)$
(c4) ev(e,sum,n=0);
Lower bound to SUM: 1
is greater than the upper bound: - 1
-- an error. Quitting. To debug this try DEBUGMODE(TRUE);)
(c5) ev(e,sum,n=0),sumhack : true;
(d5)      1
```

Most functions in `specfun` have a `gradef` property; derivatives with respect to the order or other function parameters aren't unevaluated.

The `specfun` package and its documentation were written by Barton Willis of the University of Nebraska at Kearney. It is released under the terms of the General Public License (GPL). Send bug reports and comments on this package to `willisb@unk.edu`. In your report, please include Maxima and `specfun` version information. The `specfun` version may be found using `get`:

```
(c2) get('specfun,'version);
(d2)      110
```

## 16.2 Definitions for Orthogonal Polynomials

**ASSOC\_LEGENDRE\_P** ( $n, m, x$ ) Function  
 [specfun package] return the associated Legendre function of the first kind for integers  $n > -1$  and  $m > -1$ . When  $|m| > n$  and  $n \geq 0$ , we have  $assoc\_legendre\_p(n, m, x) = 0$ . Reference: A&S 22.5.37 page 779, A&S 8.6.6 (second equation) page 334, and A&S 8.2.5 page 333. To access this function, `load("specfun")`. See [\[ASSOC\\_LEGENDRE\\_Q\]](#), [page 107](#), [\[LEGENDRE\\_P\]](#), [page 109](#), and [\[LEGENDRE\\_Q\]](#), [page 109](#).

**ASSOC\_LEGENDRE\_Q** ( $n, m, x$ ) Function  
 [specfun package] return the associated Legendre function of the second kind for integers  $n > -1$  and  $m > -1$ .  
 Reference: Gradshteyn and Ryzhik 8.706 page 1000.  
 To access this function, `load("specfun")`.  
 See also `ASSOC_LEGENDRE_P`, `LEGENDRE_P`, and `LEGENDRE_Q`.

- CHEBYSHEV\_T** ( $n, x$ ) Function  
 [specfun package] return the Chebyshev function of the first kind for integers  $n > -1$ .  
 Reference: A&S 22.5.31 page 778 and A&S 6.1.22 page 256.  
 To access this function, load("specfun").  
 See also CHEBYSHEV\_U.
- CHEBYSHEV\_U** ( $n, x$ ) Function  
 [specfun package] return the Chebyshev function of the second kind for integers  $n > -1$ .  
 Reference: A&S, 22.8.3 page 783 and A&S 6.1.22 page 256.  
 To access this function, load("specfun").  
 See also CHEBYSHEV\_T.
- GEN\_LAGUERRE** ( $n, a, x$ ) Function  
 [specfun package] return the generalized Laguerre polynomial for integers  $n > -1$ .  
 To access this function, load("specfun").  
 Reference: table on page 789 in A&S.
- HERMITE** ( $n, x$ ) Function  
 [specfun package] return the Hermite polynomial for integers  $n > -1$ .  
 To access this function, load("specfun").  
 Reference: A&S 22.5.40 and 22.5.41, page 779.
- JACOBLP** ( $n, a, b, x$ ) Function  
 [specfun package] return the Jacobi polynomial for integers  $n > -1$  and  $a$  and  $b$  symbolic or  $a > -1$  and  $b > -1$ . (The Jacobi polynomials are actually defined for all  $a$  and  $b$ ; however, the Jacobi polynomial weight  $(1-x)^a(1+x)^b$  isn't integrable for  $a < -1$  or  $b < -1$ .)  
 When  $a, b$ , and  $x$  are floats (but not bfloats) specfun calls a special modedeclared version of *jacobi\_p*. For numerical values, the modedeclared version is much faster than the other version. Many functions in specfun are computed as a special case of the Jacobi polynomials; they also enjoy the speed boost from the modedeclared version of *jacobi*.  
 If  $n$  has been declared to be an integer, *jacobi\_p*( $n, a, b, x$ ) returns a summation representation for the Jacobi function. Because Maxima simplifies  $0^0$  to 0 in a sum, two terms of the sum are added outside the summation.  
 To access this function, load("specfun").  
 Reference: table on page 789 in A&S.
- LAGUERRE** ( $n, x$ ) Function  
 [specfun package] return the Laguerre polynomial for integers  $n > -1$ .  
 Reference: A&S 22.5.16, page 778 and A&S page 789.  
 To access this function, load("specfun").  
 See also GEN\_LAGUERRE.

- LEGENDRE\_P** ( $n, x$ ) Function  
 [specfun package] return the Legendre polynomial of the first kind for integers  $n > -1$ .  
 Reference: A&S 22.5.35 page 779.  
 To access this function, load("specfun").  
 See [\[LEGENDRE\\_Q\]](#), page 109.
- LEGENDRE\_Q** ( $n, x$ ) Function  
 [specfun package] return the Legendre polynomial of the first kind for integers  $n > -1$ .  
 Reference: A&S 8.6.19 page 334.  
 To access this function, load("specfun").  
 See also LEGENDRE\_P.
- SPHERICAL\_BESSEL\_J** ( $n, x$ ) Function  
 [specfun package] return the spherical Bessel function of the first kind for integers  $n > -1$ .  
 Reference: A&S 10.1.8 page 437 and A&S 10.1.15 page 439.  
 To access this function, load("specfun").  
 See also SPHERICAL\_HANKEL1, SPHERICAL\_HANKEL2, and SPHERICAL\_BESSEL\_Y.
- SPHERICAL\_BESSEL\_Y** ( $n, x$ ) Function  
 [specfun package] return the spherical Bessel function of the second kind for integers  $n > -1$ .  
 Reference: A&S 10.1.9 page 437 and 10.1.15 page 439.  
 To access this function, load("specfun").  
 See also SPHERICAL\_HANKEL1, SPHERICAL\_HANKEL2, and SPHERICAL\_BESSEL\_Y.
- SPHERICAL\_HANKEL1** ( $n, x$ ) Function  
 [specfun package] return the spherical hankel function of the first kind for integers  $n > -1$ .  
 Reference: A&S 10.1.36 page 439.  
 To access this function, load("specfun").  
 See also SPHERICAL\_HANKEL2, SPHERICAL\_BESSEL\_J, and SPHERICAL\_BESSEL\_Y.
- SPHERICAL\_HANKEL2** ( $n, x$ ) Function  
 [specfun package] return the spherical hankel function of the second kind for integers  $n > -1$ .  
 Reference: A&S 10.1.17 page 439.  
 To access this function, load("specfun").  
 See also SPHERICAL\_HANKEL1, SPHERICAL\_BESSEL\_J, and SPHERICAL\_BESSEL\_Y.

**SPHERICAL\_HARMONIC** ( $n, m, x, y$ ) Function

[specfun package] return the spherical harmonic function for integers  $n > -1$  and  $|m| \leq n$ .

Reference: Merzbacher 9.64.

To access this function, load("specfun").

See also ASSOC\_LEGENDRE\_P

**ULTRASPHERICAL** ( $n, a, x$ ) Function

[specfun package] return the ultraspherical polynomials for integers  $n > -1$ . The ultraspherical polynomials are also known as Gegenbauer polynomials.

Reference: A&S 22.5.27

To access this function, load("specfun").

See also JACOBI\_P.

## 17 Limits

### 17.1 Definitions for Limits

#### **LHOSPITALIM**

Variable

default: [4] - the maximum number of times L'Hospital's rule is used in LIMIT. This prevents infinite looping in cases like  $\text{LIMIT}(\text{COT}(X)/\text{CSC}(X), X, 0)$ .

#### **LIMIT** (*exp, var, val, dir*)

Function

finds the limit of *exp* as the real variable *var* approaches the value *val* from the direction *dir*. *Dir* may have the value PLUS for a limit from above, MINUS for a limit from below, or may be omitted (implying a two-sided limit is to be computed). For the method see Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation" - Ph.D. Thesis - MAC TR-92 October 1971. LIMIT uses the following special symbols: INF (positive infinity) and MINF (negative infinity). On output it may also use UND (undefined), IND (indefinite but bounded) and INFINITY (complex infinity). LHOSPITALIM[4] is the maximum number of times L'Hospital's rule is used in LIMIT. This prevents infinite looping in cases like  $\text{LIMIT}(\text{COT}(X)/\text{CSC}(X), X, 0)$ . TLIMSWITCH[FALSE] when true will cause the limit package to use Taylor series when possible. LIMSUBST[FALSE] prevents LIMIT from attempting substitutions on unknown forms. This is to avoid bugs like  $\text{LIMIT}(F(N)/F(N+1), N, \text{INF})$ ; giving 1. Setting LIMSUBST to TRUE will allow such substitutions. Since LIMIT is often called upon to simplify constant expressions, for example, INF-1, LIMIT may be used in such cases with only one argument, e.g.  $\text{LIMIT}(\text{INF}-1)$ ; Do EXAMPLE(LIMIT); for examples.

#### **TLIMIT** (*exp, var, val, dir*)

Function

is just the function LIMIT with TLIMSWITCH set to TRUE.

#### **TLIMSWITCH**

Variable

default: [FALSE] - if true will cause the limit package to use Taylor series when possible.





## 18 Differentiation

### 18.1 Definitions for Differentiation

**ANTID** ( $G, X, U(X)$ ) Function

A routine for evaluating integrals of expressions involving an arbitrary unspecified function and its derivatives. It may be used by `LOAD(ANTID);`, after which, the function `ANTIDIFF` may be used. E.g. `ANTIDIFF(G,X,U(X));` where  $G$  is the expression involving  $U(X)$  ( $U(X)$  arbitrary) and its derivatives, whose integral with respect to  $X$  is desired. The functions `NONZEROANDFREEOF` and `LINEAR` are also defined, as well as `ANTID`. `ANTID` is the same as `ANTIDIFF` except that it returns a list of two parts, the first part is the integrated part of the expression and the second part of the list is the non-integrable remainder.

**ANTIDIFF** - Function  
See `ANTID`.

**ATOMGRAD** property  
- the atomic gradient property of an expression. May be set by `GRADEF`.

**ATVALUE** (*form, list, value*) Function  
enables the user to assign the boundary value *value* to *form* at the points specified by *list*.

(C1) `ATVALUE(F(X,Y), [X=0,Y=1], A**2)$`  
The form must be a function,  $f(v_1, v_2, \dots)$ , or a derivative,

$\text{DIFF}(f(v_1, v_2, \dots), v_i, n_i, v_j, n_j, \dots)$  in which the functional arguments explicitly appear ( $n_i$  is the order of differentiation with respect to  $v_i$ ). The list of equations determine the "boundary" at which the value is given; *list* may be a list of equations, as above, or a single equation,  $v_i = \text{expr}$ . The symbols `@1, @2, ...` will be used to represent the functional variables  $v_1, v_2, \dots$  when atvalues are displayed. `PRINTPROPS([f1, f2, ...], ATVALUE)` will display the atvalues of the functions  $f_1, f_2, \dots$  as specified in previously given uses of the `ATVALUE` function. If the list contains just one element then the element can be given without being in a list. If a first argument of `ALL` is given then atvalues for all functions which have them will be displayed. Do `EXAMPLE(ATVALUE);` for an example.

**CARTAN** - Function  
The exterior calculus of differential forms is a basic tool of differential geometry developed by Elie Cartan and has important applications in the theory of partial differential equations. The present implementation is due to F.B. Estabrook and H.D. Wahlquist. The program is self-explanatory and can be accessed by doing `batch("cartan");` which will give a description with examples.

**DELTA** (*t*)

Function

This is the Dirac Delta function. Currently only LAPLACE knows about the DELTA function:

(C1) LAPLACE(DELTA(T-A)\*SIN(B\*T),T,S);

Is A positive, negative or zero?

POS;

(D1) 
$$\int_{-\infty}^{\infty} \delta(T-A) \sin(BT) e^{-AS} dT$$

**DEPENDENCIES**

Variable

default: [] - the list of atoms which have functional dependencies (set up by the DEPENDS or GRADEF functions). The command DEPENDENCIES has been replaced by the DEPENDS command. Do DESCRIBE(DEPENDS);

**DEPENDS** (*funlist1,varlist1,funlist2,varlist2,...*)

Function

declares functional dependencies for variables to be used by DIFF.

DEPENDS([F,G],[X,Y],[R,S],[U,V,W],U,T)

informs DIFF that F and G depend on X and Y, that R and S depend on U,V, and W, and that U depends on T. The arguments to DEPENDS are evaluated. The variables in each funlist are declared to depend on all the variables in the next varlist. A funlist can contain the name of an atomic variable or array. In the latter case, it is assumed that all the elements of the array depend on all the variables in the succeeding varlist. Initially, DIFF(F,X) is 0; executing DEPENDS(F,X) causes future differentiations of F with respect to X to give dF/dX or Y (if DERIVABBREV:TRUE).

(C1) DEPENDS([F,G],[X,Y],[R,S],[U,V,W],U,T);

(D1) 
$$[F(X, Y), G(X, Y), R(U, V, W), S(U, V, W), U(T)]$$

(C2) DEPENDENCIES;

(D2) 
$$[F(X, Y), G(X, Y), R(U, V, W), S(U, V, W), U(T)]$$

(C3) DIFF(R.S,U);

(D3) 
$$\frac{dR}{dU} \cdot S + R \cdot \frac{dS}{dU}$$

Since MACSYMA knows the chain rule for symbolic derivatives, it takes advantage of the given dependencies as follows:

(C4) DIFF(R.S,T);

(D4) 
$$\left(\frac{dR}{dU} \frac{dU}{dT}\right) \cdot S + R \cdot \left(\frac{dS}{dU} \frac{dU}{dT}\right)$$

If we set

(C5) DERIVABBREV:TRUE;

(D5) 
$$\text{TRUE}$$

then re-executing the command C4, we obtain

(C6) ''C4;

(D6) 
$$\left(\frac{R}{U} \frac{U}{T}\right) \cdot S + R \cdot \left(\frac{S}{U} \frac{U}{T}\right)$$

To eliminate a previously declared dependency, the REMOVE command can be used. For example, to say that R no longer depends on U as declared in C1, the user can type

```
REMOVE(R,DEPENDENCY)
```

This will eliminate all dependencies that may have been declared for R.

```
(C7) REMOVE(R,DEPENDENCY);
(D7)                                     DONE
(C8) ' 'C4;
(D8)                                     R . (S U )
                                         U   T
```

CAVEAT: DIFF is the only MACSYMA command which uses DEPENDENCIES information. The arguments to INTEGRATE, LAPLACE, etc. must be given their dependencies explicitly in the command, e.g., INTEGRATE(F(X),X).

### DERIVABBREV

Variable

default: [FALSE] if TRUE will cause derivatives to display as subscripts.

### DERIVDEGREE (*exp, dv, iv*)

Function

finds the highest degree of the derivative of the dependent variable dv with respect to the independent variable iv occurring in exp.

```
(C1) 'DIFF(Y,X,2)+'DIFF(Y,Z,3)*2+'DIFF(Y,X)*X**2$
(C2) DERIVDEGREE(% ,Y,X);
(D2)                                     2
```

### DERIVLIST (*var1,...,vark*)

Function

causes only differentiations with respect to the indicated variables, within the EV command.

### DERIVSUBST

Variable

default: [FALSE] - controls non-syntactic substitutions such as

```
SUBST(X, 'DIFF(Y,T) , 'DIFF(Y,T,2));
```

If DERIVSUBST is set to true, this gives 'DIFF(X,T).

### DIFF

special symbol

[flag] for ev causes all differentiations indicated in exp to be performed.

### DIFF (*exp, v1, n1, v2, n2, ...*)

Function

DIFF differentiates exp with respect to each vi, ni times. If just the first derivative with respect to one variable is desired then the form DIFF(exp,v) may be used. If the noun form of the function is required (as, for example, when writing a differential equation), 'DIFF should be used and this will display in a two dimensional format. DERIVABBREV[FALSE] if TRUE will cause derivatives to display as subscripts.

DIFF(exp) gives the "total differential", that is, the sum of the derivatives of exp with respect to each of its variables times the function DEL of the variable. No further simplification of DEL is offered.

(C1) DIFF(EXP(F(X)),X,2);

(D1) 
$$\%E \frac{F(X)^2}{2} \frac{d}{dX} (F(X)) + \%E \frac{F(X)}{dX} \frac{d}{dX} (F(X))^2$$

(C2) DERIVABBREV:TRUE\$

(C3) 'INTEGRATE(F(X,Y),Y,G(X),H(X));

(D3) 
$$\frac{H(X)}{\int F(X, Y) dY}$$

(C4) DIFF(%,X);

(D4) 
$$\frac{H(X)}{\int F(X, Y) dY + F(X, H(X)) \frac{H(X)}{X} - F(X, G(X)) \frac{G(X)}{X}}$$

For the tensor package, the following modifications have been incorporated: 1) the derivatives of any indexed objects in exp will have the variables vi appended as additional arguments. Then all the derivative indices will be sorted. 2) the vi may be integers from 1 up to the value of the variable DIMENSION[default value: 4]. This will cause the differentiation to be carried out wrt the vith member of the list COORDINATES which should be set to a list of the names of the coordinates, e.g., [x,y,z,t]. If COORDINATES is bound to an atomic variable, then that variable subscripted by vi will be used for the variable of differentiation. This permits an array of coordinate names or subscripted names like X[1], X[2],... to be used. If COORDINATES has not been assigned a value, then the variables will be treated as in 1) above.

**DSCALAR** (*function*)

Function

applies the scalar d'Alembertian to the scalar function.

(C41) DEPENDENCIES(FIELD(R));

(D41) [FIELD(R)]

(C42) DSCALAR(FIELD);

(D43)

$$\%E \left( \frac{\partial^M}{\partial R^M} \left( \frac{\partial^N}{\partial R^N} - \frac{\partial^M}{\partial R^M} + 2 \frac{\partial^2}{\partial R^2} \right) R + 4 \frac{\partial}{\partial R} \right)$$

**EXPRESS** (*expression*) Function

The result uses the noun form of any derivatives arising from expansion of the vector differential operators. To force evaluation of these derivatives, the built-in EV function can be used together with the DIFF evflag, after using the built-in DEPENDS function to establish any new implicit dependencies.

**GENDIFF** Function

Sometimes DIFF(E,X,N) can be reduced even though N is symbolic.

```
batch("gendif.mc")$
```

and you can try, for example,

```
DIFF(%E^(A*X),X,Q)
```

by using GENDIFF rather than DIFF. Unevaluable items come out quoted. Some items are in terms of "GENFACT", which see.

**GRADEF** ( $f(x_1, \dots, x_n), g_1, \dots, g_n$ ) Function

defines the derivatives of the function  $f$  with respect to its  $n$  arguments. That is,  $df/dx_i = g_i$ , etc. If fewer than  $n$  gradients, say  $i$ , are given, then they refer to the first  $i$  arguments of  $f$ . The  $x_i$  are merely dummy variables as in function definition headers and are used to indicate the  $i$ th argument of  $f$ . All arguments to GRADEF except the first are evaluated so that if  $g$  is a defined function then it is invoked and the result is used. Gradients are needed when, for example, a function is not known explicitly but its first derivatives are and it is desired to obtain higher order derivatives. GRADEF may also be used to redefine the derivatives of MACSYMA's predefined functions (e.g. `GRADEF(SIN(X),SQRT(1-SIN(X)**2))`). It is not permissible to use GRADEF on subscripted functions. GRADEFS is a list of the functions which have been given gradients by use of the GRADEF command (i.e. `GRADEF(f(x1, ..., xn), g1, ..., gn)`). `PRINTPROPS([f1,f2,...],GRADEF)` may be used to display the gradeFs of the functions  $f_1, f_2, \dots$ . `GRADEF(a,v,exp)` may be used to state that the derivative of the atomic variable  $a$  with respect to  $v$  is  $exp$ . This automatically does a `DEPENDS(a,v)`. `PRINTPROPS([a1,a2,...],ATOMGRAD)` may be used to display the atomic gradient properties of  $a_1, a_2, \dots$ .

**GRADEFS** Variable

default: [] - a list of the functions which have been given gradients by use of the GRADEF command (i.e. `GRADEF(f(x1, ..., xn), g1, ..., gn)`.)

**LAPLACE** (*exp, ovar, lvar*) Function

takes the Laplace transform of  $exp$  with respect to the variable  $ovar$  and transform parameter  $lvar$ .  $Exp$  may only involve the functions EXP, LOG, SIN, COS, SINH, COSH, and ERF. It may also be a linear, constant coefficient differential equation in which case ATVALUE of the dependent variable will be used. These may be supplied either before or after the transform is taken. Since the initial conditions must

be specified at zero, if one has boundary conditions imposed elsewhere he can impose these on the general solution and eliminate the constants by solving the general solution for them and substituting their values back. Exp may also involve convolution integrals. Functional relationships must be explicitly represented in order for LAPLACE to work properly. That is, if F depends on X and Y it must be written as F(X,Y) wherever F occurs as in LAPLACE('DIFF(F(X,Y),X),X,S). LAPLACE is not affected by DEPENDENCIES set up with the DEPENDS command.

(C1) LAPLACE(%E\*\*(2\*T+A)\*SIN(T)\*T,T,S);

$$(D1) \quad \frac{e^{2T} (S - 2)}{(S - 2)^2 + 1}$$

### UNDIFF (*exp*)

Function

returns an expression equivalent to *exp* but with all derivatives of indexed objects replaced by the noun form of the DIFF function with arguments which would yield that indexed object if the differentiation were carried out. This is useful when it is desired to replace a differentiated indexed object with some function definition and then carry out the differentiation by saying EV(...,DIFF).

# 19 Integration

## 19.1 Introduction to Integration

MACSYMA has several routines for handling integration. The INTEGRATE command makes use of most of them. There is also the ANTID package, which handles an unspecified function (and its derivatives, of course). For numerical uses, there is the ROMBERG function, and the IMSL version of Romberg, DCADRE. There is also an adaptive integrator which uses the Newton-Cotes 8 panel quadrature rule, called QUANC8. Hypergeometric Functions are being worked on, do DESCRIBE(SPECINT); for details. Generally speaking, MACSYMA only handles integrals which are integrable in terms of the "elementary functions" (rational functions, trigonometrics, logs, exponentials, radicals, etc.) and a few extensions (error function, dilogarithm). It does not handle integrals in terms of unknown functions such as  $g(x)$  and  $h(x)$ .

## 19.2 Definitions for Integration

**CHANGEVAR** (*exp,f(x,y),y,x*) Function  
 makes the change of variable given by  $f(x,y) = 0$  in all integrals occurring in *exp* with integration with respect to *x*; *y* is the new variable.

```
(C1) 'INTEGRATE(%E**SQRT(A*Y),Y,0,4);
      4
      /
      [  SQRT(A) SQRT(Y)
(D1)  I (%E          ) dY
      ]
      /
      0
(C2) CHANGEVAR(D1,Y-Z^2/A,Z,Y);
      2 SQRT(A)
      /
      [          Z
      2 I          Z %E  dZ
      ]
      /
      0
(D4) -----
      A
```

CHANGEVAR may also be used to changes in the indices of a sum or product. However, it must be realized that when a change is made in a sum or product, this change must be a shift, i.e.  $I=J+ \dots$ , not a higher degree function. E.g.

```
(C3) SUM(A[I]*X^(I-2),I,0,INF);
      INF
      =====
      \          I - 2
```



```

(D3)          >  A X
              /  I
              ====
              I = 0

(C4) CHANGEVAR(% , I-2-N, N, I);

              INF
              ====
              \
              >  A      X
              /    N + 2
              ====
              N = - 2

```

**DBLINT** ('F,'R,'S,a,b)

Function

a double-integral routine which was written in top-level macsyma and then translated and compiled to machine code. Use `LOAD(DBLINT)`; to access this package. It uses the Simpson's Rule method in both the x and y directions to calculate  $\int \int F(X,Y) \frac{DY}{S(X)} \frac{DX}{R(X)}$ . The function  $F(X,Y)$  must be a translated or compiled function of two variables, and  $R(X)$  and  $S(X)$  must each be a translated or compiled function of one variable, while  $a$  and  $b$  must be floating point numbers. The routine has two global variables which determine the number of divisions of the  $x$  and  $y$  intervals: `DBLINT_X` and `DBLINT_Y`, both of which are initially 10, and can be changed independently to other integer values (there are  $2*DBLINT\_X+1$  points computed in the  $x$  direction, and  $2*DBLINT\_Y+1$  in the  $y$  direction). The routine subdivides the  $X$  axis and then for each value of  $X$  it first computes  $R(X)$  and  $S(X)$ ; then the  $Y$  axis between  $R(X)$  and  $S(X)$  is subdivided and the integral along the  $Y$  axis is performed using Simpson's Rule; then the integral along the  $X$  axis is done using Simpson's Rule with the function values being the  $Y$ -integrals. This procedure may be numerically unstable for a great variety of reasons, but is reasonably fast: avoid using it on highly oscillatory functions and functions with singularities (poles or branch points in the region). The  $Y$  integrals depend on how far apart  $R(X)$  and  $S(X)$  are, so if the distance  $S(X)-R(X)$  varies rapidly with  $X$ , there may be substantial errors arising from truncation with different step-sizes in the various  $Y$  integrals. One can increase `DBLINT_X` and `DBLINT_Y` in an effort to improve the coverage of the region, at the expense of computation time. The function values are not saved, so if the function is very time-consuming, you will have to wait for re-computation if you change anything (sorry). It is required that the functions  $F$ ,  $R$ , and  $S$  be either translated or compiled prior to calling `DBLINT`. This will result in orders of magnitude speed improvement over interpreted code in many cases! Ask LPH (or GJC) about using these numerical aids. The file `SHARE1;DBLINT DEMO` can be run in batch or demo mode to illustrate the usage on a sample problem; the file `SHARE1;DBLNT DEMO1` is an extension of the `DEMO` which also makes use of other numerical aids, `FLOATDEFUNK` and `QUANC8`. Please send all bug notes and questions to LPH

- DEFINT** (*exp, var, low, high*) Function  
 DEFinite INTEgration, the same as INTEGRATE(*exp,var,low,high*). This uses symbolic methods, if you wish to use a numerical method try ROMBERG(*exp,var,low,high*).
- ERF** (*X*) Function  
 the error function, whose derivative is:  $2*\text{EXP}(-X^2)/\text{SQRT}(\%PI)$ .
- ERFFLAG** Variable  
 default: [TRUE] if FALSE prevents RISCH from introducing the ERF function in the answer if there were none in the integrand to begin with.
- ERRINTSCE** Variable  
 default: [TRUE] - If a call to the INTSCE routine is not of the form  
 $\text{EXP}(A*X+B)*\text{COS}(C*X)^N*\text{SIN}(C*X)$   
 then the regular integration program will be invoked if the switch ERRINTSCE[TRUE] is TRUE. If it is FALSE then INTSCE will err out.
- ILT** (*exp, lvar, ovar*) Function  
 takes the inverse Laplace transform of *exp* with respect to *lvar* and parameter *ovar*. *exp* must be a ratio of polynomials whose denominator has only linear and quadratic factors. By using the functions LAPLACE and ILT together with the SOLVE or LINSOLVE functions the user can solve a single differential or convolution integral equation or a set of them.
- (C1) 'INTEGRATE(SINH(A\*X)\*F(T-X),X,0,T)+B\*F(T)=T\*\*2;  
           T  
           /  
           [  
 (D1)      I (SINH(A X) F(T - X)) dX + B F(T) = T<sup>2</sup>  
           ]  
           /  
           0
- (C2) LAPLACE(%,T,S);  
 (D2)      A LAPLACE(F(T), T, S)  
           -----  
           2      2  
           S - A  
                                   2  
           + B LAPLACE(F(T), T, S) = --  
                                   3  
                                   S
- (C3) LINSOLVE([%], ['LAPLACE(F(T),T,S)]);  
 SOLUTION
- (E3)      LAPLACE(F(T), T, S) = -----  
                                   2      2  
                                   2 S - 2 A

$$\begin{aligned}
 & \text{(D3)} \quad \text{[E3]} \quad B^5 S^2 + (A - A^2 B) S^3 \\
 & \text{(C4)} \quad \text{ILT(E3,S,T);} \\
 & \text{IS } A B (A B - 1) \text{ POSITIVE, NEGATIVE, OR ZERO?} \\
 & \text{POS;} \\
 & \text{(D4)} \quad F(T) = \frac{\text{SQRT}(A) \text{SQRT}(A B^2 - B) T}{2 \text{COSH}\left(\frac{\quad}{B}\right)} \\
 & \quad \quad \quad \frac{A}{\quad} \\
 & \quad \quad \quad + \frac{A T^2}{A B - 1} + \frac{2}{A^3 B^2 - 2 A^2 B + A}
 \end{aligned}$$

**INTEGRATE** (*exp*, *var*)

Function

integrates *exp* with respect to *var* or returns an integral expression (the noun form) if it cannot perform the integration (see note 1 below). Roughly speaking three stages are used:

- (1) **INTEGRATE** sees if the integrand is of the form  $F(G(X)) \cdot \text{DIFF}(G(X), X)$  by testing whether the derivative of some subexpression (i.e.  $G(X)$  in the above case) divides the integrand. If so it looks up  $F$  in a table of integrals and substitutes  $G(X)$  for  $X$  in the integral of  $F$ . This may make use of gradients in taking the derivative. (If an unknown function appears in the integrand it must be eliminated in this stage or else **INTEGRATE** will return the noun form of the integrand.)
- (2) **INTEGRATE** tries to match the integrand to a form for which a specific method can be used, e.g. trigonometric substitutions.
- (3) If the first two stages fail it uses the Risch algorithm. Functional relationships must be explicitly represented in order for **INTEGRATE** to work properly. **INTEGRATE** is not affected by **DEPENDENCIES** set up with the **DEPENDS** command. **INTEGRATE**(*exp*, *var*, *low*, *high*) finds the definite integral of *exp* with respect to *var* from *low* to *high* or returns the noun form if it cannot perform the integration. The limits should not contain *var*. Several methods are used, including direct substitution in the indefinite integral and contour integration. Improper integrals may use the names **INF** for positive infinity and **MINF** for negative infinity. If an integral "form" is desired for manipulation (for example, an integral which cannot be computed until some numbers are substituted for some parameters), the noun form **'INTEGRATE** may be used and this will display with an integral sign. (See Note 1 below.) The function **LDEFINT** uses **LIMIT** to evaluate the integral at the lower and upper limits. Sometimes during

integration the user may be asked what the sign of an expression is. Suitable responses are POS;, ZERO;, or NEG;.

(C1) INTEGRATE(SIN(X)\*\*3,X);

(D1) 
$$\frac{\cos^3(X)}{3} - \cos(X)$$

(C2) INTEGRATE(X\*\*A/(X+1)\*\*(5/2),X,0,INF);

IS A + 1 POSITIVE, NEGATIVE, OR ZERO?

POS;

IS 2 A - 3 POSITIVE, NEGATIVE, OR ZERO?

NEG;

(D2) 
$$\text{BETA}(A + 1, -\frac{3}{2} - A)$$

(C3) GRADEF(Q(X),SIN(X\*\*2));

(D3) 
$$Q(X)$$

(C4) DIFF(LOG(Q(R(X))),X);

(D4) 
$$\frac{\frac{d}{dX} \left( \frac{1}{Q(R(X))} \right) \sin(R(X)^2)}{Q(R(X))}$$

(C5) INTEGRATE(%,X);

(D5) 
$$\text{LOG}(Q(R(X)))$$

(Note 1) The fact that MACSYMA does not perform certain integrals does not always imply that the integral does not exist in closed form. In the example below the integration call returns the noun form but the integral can be found fairly easily. For example, one can compute the roots of  $X^3+X+1 = 0$  to rewrite the integrand in the form

$$1/((X-A)*(X-B)*(X-C))$$

where A, B and C are the roots. MACSYMA will integrate this equivalent form although the integral is quite complicated.

(C6) INTEGRATE(1/(X^3+X+1),X);

(D6) 
$$\int \frac{1}{X^3 + X + 1} dX$$

## INTEGRATION\_CONSTANT\_COUNTER

Variable

- a counter which is updated each time a constant of integration (called by MACSYMA, e.g., "INTEGRATIONCONSTANT1") is introduced into an expression by indefinite integration of an equation.

**INTEGRATE\_USE\_ROOTSOF**

Variable

default: [false] If not false then when the denominator of a rational function cannot be factored, we give the integral in a form which is a sum over the roots of the denominator:

(C4) `integrate(1/(1+x+x^5),x);`

$$(D4) \frac{\int \frac{dx}{x^3 - x^2 + 1}}{7} - \frac{2x + 1}{14} \operatorname{LOG}(x^2 + x + 1) + \frac{5 \operatorname{ATAN}\left(\frac{2x + 1}{\sqrt{3}}\right)}{7 \sqrt{3}}$$

but now we set the flag to be true and the first part of the integral will undergo further simplification.

(C5) `INTEGRATE_USE_ROOTSOF:true;`

(D5) `TRUE`

(C6) `integrate(1/(1+x+x^5),x);`

$$(D6) \frac{\int \frac{dx}{x^3 - x^2 + 1}}{7} - \frac{(\%R1^2 - 4 \%R1 + 5) \operatorname{LOG}(x - \%R1)}{3 \%R1^2 - 2 \%R1}$$

%R1 in ROOTSOF(x<sup>3</sup> - x<sup>2</sup> + 1)

$$- \frac{\operatorname{LOG}(x^2 + x + 1)}{14} + \frac{5 \operatorname{ATAN}\left(\frac{2x + 1}{\sqrt{3}}\right)}{7 \sqrt{3}}$$

Note that it may be that we want to approximate the roots in the complex plane, and then provide the function factored, since we will then be able to group the roots and their complex conjugates, so as to give a better answer.

**INTSCE** (*expr,var*)

Function

INTSCE LISP contains a routine, written by Richard Bogen, for integrating products of sines,cosines and exponentials of the form

$$\operatorname{EXP}(A*X+B) * \operatorname{COS}(C*X)^N * \operatorname{SIN}(C*X)^M$$

The call is INTSCE(*expr,var*) *expr* may be any expression, but if it is not in the above form then the regular integration program will be invoked if the switch ERRINTSCE[TRUE] is TRUE. If it is FALSE then INTSCE will err out.

**LDEFINT** (*exp, var, ll, ul*) Function  
 yields the definite integral of *exp* by using **LIMIT** to evaluate the indefinite integral of *exp* with respect to *var* at the upper limit *ul* and at the lower limit *ll*.

**POTENTIAL** (*givengradient*) Function  
 The calculation makes use of the global variable  
`POTENTIALZERoloc[0]`  
 which must be **NONLIST** or of the form  
`[indeterminatej=expressionj, indeterminatek=expressionk, ...]`  
 the former being equivalent to the nonlist expression for all right-hand sides in the latter. The indicated right-hand sides are used as the lower limit of integration. The success of the integrations may depend upon their values and order. **POTENTIALZERoloc** is initially set to 0.

**QQ** Function  
 - The file `SHARE1;QQ FASL` (which may be loaded with `LOAD("QQ");`) contains a function **QUANC8** which can take either 3 or 4 arguments. The 3 arg version computes the integral of the function specified as the first argument over the interval from *lo* to *hi* as in `QUANC8('function name,lo,hi);` . The function name should be quoted. The 4 arg version will compute the integral of the function or expression (first arg) with respect to the variable (second arg) over the interval from *lo* to *hi* as in `QUANC8(<f(x) or expression in x>,x,lo,hi)`. The method used is the Newton-Cotes 8th order polynomial quadrature, and the routine is adaptive. It will thus spend time dividing the interval only when necessary to achieve the error conditions specified by the global variables `QUANC8_RELEERR` (default value=1.0e-4) and `QUANC8_ABSERR` (default value=1.0e-8) which give the relative error test:  $|\text{integral}(\text{function}) - \text{computed value}| < \text{quanc8\_relerr} * |\text{integral}(\text{function})|$  and the absolute error test:  $|\text{integral}(\text{function}) - \text{computed value}| < \text{quanc8\_abserr}$ . Do `PRINT-FILE(QQ,USAGE,SHARE1)` for details.

**QUANC8** (*'function name, lo, hi*) Function  
 An adaptive integrator, available in `SHARE1;QQ FASL`. `DEMO` and `USAGE` files are provided. The method is to use Newton-Cotes 8-panel quadrature rule, hence the function name **QUANC8**, available in 3 or 4 arg versions. Absolute and relative error checks are used. To use it do `LOAD("QQ");` For more details do `DESCRIBE(QQ);` .

**RESIDUE** (*exp, var, val*) Function  
 computes the residue in the complex plane of the expression *exp* when the variable *var* assumes the value *val*. The residue is the coefficient of  $(\text{var}-\text{val})^{*-1}$  in the Laurent series for *exp*.

```
(C1) RESIDUE(S/(S**2+A**2),S,A*%I);
      1
(D1)
      -
      2
(C2) RESIDUE(SIN(A*X)/X**4,X,0);
      3
```

$$(D2) \quad - \frac{A}{6}$$

**RISCH** (*exp, var*)

Function

integrates *exp* with respect to *var* using the transcendental case of the Risch algorithm. (The algebraic case of the Risch algorithm has not been implemented.) This currently handles the cases of nested exponentials and logarithms which the main part of INTEGRATE can't do. INTEGRATE will automatically apply RISCH if given these cases. ERFFLAG[TRUE] - if FALSE prevents RISCH from introducing the ERF function in the answer if there were none in the integrand to begin with.

```
(C1) RISCH(X^2*ERF(X),X);
```

$$(D1) \quad \frac{\%E^{-X^2} (\%E^{X^2} \sqrt{\%PI} X^3 \operatorname{ERF}(X) + X^2 + 1)}{3 \sqrt{\%PI}}$$

```
(C2) DIFF(%,X),RATSIMP;
```

$$(D2) \quad X^2 \operatorname{ERF}(X)$$

**ROMBERG** (*exp,var,ll,ul*)

Function

or ROMBERG(*exp,ll,ul*) - Romberg Integration. You need not load in any file to use ROMBERG, it is autoloading. There are two ways to use this function. The first is an inefficient way like the definite integral version of INTEGRATE: ROMBERG(<integrand>,<variable of integration>,<lower limit>,<upper limit>);

Examples:

```
ROMBERG(SIN(Y),Y,1,%PI);
      TIME= 39 MSEC.          1.5403023
F(X):=1/(X^5+X+1);
ROMBERG(F(X),X,1.5,0);
      TIME= 162 MSEC.        - 0.75293843
```

The second is an efficient way that is used as follows:

```
ROMBERG(<function name>,<lower limit>,<upper limit>);
Example:
F(X):=(MODE_DECLARE([FUNCTION(F),X],FLOAT),1/(X^5+X+1));
TRANSLATE(F);
ROMBERG(F,1.5,0);
      TIME= 13 MSEC.          - 0.75293843
```

The first argument must be a TRANSLATED or compiled function. (If it is compiled it must be declared to return a FLONUM.) If the first argument is not already TRANSLATED, ROMBERG will not attempt to TRANSLATE it but will give an error. The accuracy of the integration is governed by the global variables

ROMBERGTOL (default value 1.E-4) and ROMBERGIT (default value 11). ROMBERG will return a result if the relative difference in successive approximations is less than ROMBERGTOL. It will try halving the stepsize ROMBERGIT times before it gives up. The number of iterations and function evaluations which ROMBERG will do is governed by ROMBERGABS and ROMBERGMIN, do DESCRIBE(ROMBERGABS,ROMBERGMIN); for details. ROMBERG may be called recursively and thus can do double and triple integrals.

Example:

```
INTEGRATE(INTEGRATE(X*Y/(X+Y),Y,0,X/2),X,1,3);
          13/3 (2 LOG(2/3) + 1)
%,NUMER;
          0.81930233
DEFINE_VARIABLE(X,0.0,FLOAT,"Global variable in function F")$
F(Y):=(MODE_DECLARE(Y,FLOAT), X*Y/(X+Y) )$
G(X):=ROMBERG('F,0,X/2)$
ROMBERG(G,1,3);
          0.8193023
```

The advantage with this way is that the function F can be used for other purposes, like plotting. The disadvantage is that you have to think up a name for both the function F and its free variable X. Or, without the global:

```
G1(X):=(MODE_DECLARE(X,FLOAT), ROMBERG(X*Y/(X+Y),Y,0,X/2))$
ROMBERG(G1,1,3);
          0.8193023
```

The advantage here is shortness.

```
Q(A,B):=ROMBERG(ROMBERG(X*Y/(X+Y),Y,0,X/2),X,A,B)$
Q(1,3);
          0.8193023
```

It is even shorter this way, and the variables do not need to be declared because they are in the context of ROMBERG. Use of ROMBERG for multiple integrals can have great disadvantages, though. The amount of extra calculation needed because of the geometric information thrown away by expressing multiple integrals this way can be incredible. The user should be sure to understand and use the ROMBERGTOL and ROMBERGIT switches. (The IMSL version of Romberg integration is now available in Macsyma. Do DESCRIBE(DCADRE); for more information.)

## ROMBERGABS

Variable

default: [0.0] (0.0B0) Assuming that successive estimates produced by ROMBERG are  $Y[0], Y[1], Y[2]$  etc., then ROMBERG will return after  $N$  iterations if (roughly speaking)  $(\text{ABS}(Y[N]-Y[N-1]) \leq \text{ROMBERGABS OR } \text{ABS}(Y[N]-Y[N-1]) / (\text{IF } Y[N]=0.0 \text{ THEN } 1.0 \text{ ELSE } Y[N]) \leq \text{ROMBERGTOL})$  is TRUE. (The condition on the number of iterations given by ROMBERGMIN must also be satisfied.) Thus if ROMBERGABS is 0.0 (the default) you just get the relative error test. The usefulness of the additional variable comes when you want to perform an integral, where the dominant contribution comes from a small region. Then you can do the integral over the small dominant region first, using the relative accuracy check, followed by the integral over the rest of the region using the absolute accuracy check. Example: Suppose you want to compute



```
Integral(exp(-x),x,0,50)
```

(numerically) with a relative accuracy of 1 part in 10000000. Define the function. N is a counter, so we can see how many function evaluations were needed.

```
F(X):=(MODE_DECLARE(N,INTEGER,X,FLOAT),N:N+1,EXP(-X))$
TRANSLATE(F)$
/* First of all try doing the whole integral at once */
BLOCK([ROMBERGTOL:1.E-6,ROMBERABS:0.],N:0,ROMBERG(F,0,50));
==> 1.00000003
N; ==> 257 /* Number of function evaluations*/
```

Now do the integral intelligently, by first doing `Integral(exp(-x),x,0,10)` and then setting `ROMBERGABS` to  $1.E-6$ \*(this partial integral).

```
BLOCK([ROMBERGTOL:1.E-6,ROMBERGABS:0.,SUM:0.],
N:0,SUM:ROMBERG(F,0,10),ROMBERGABS:SUM*ROMBERGTOL,ROMBERGTOL:0.,
SUM+ROMBERG(F,10,50)); ==> 1.00000001 /* Same as before */
N; ==> 130
```

So if `F(X)` were a function that took a long time to compute, the second method would be about 2 times quicker.

## ROMBERGIT

Variable

default: [11] - The accuracy of the ROMBERG integration command is governed by the global variables `ROMBERGTOL`[1.E-4] and `ROMBERGIT`[11]. ROMBERG will return a result if the relative difference in successive approximations is less than `ROMBERGTOL`. It will try halving the stepsize `ROMBERGIT` times before it gives up.

## ROMBERGMIN

Variable

default: [0] - governs the minimum number of function evaluations that ROMBERG will make. ROMBERG will evaluate its first arg. at least  $2^{(ROMBERGMIN+2)+1}$  times. This is useful for integrating oscillatory functions, when the normal converge test might sometimes wrongly pass.

## ROMBERGTOL

Variable

default: [1.E-4] - The accuracy of the ROMBERG integration command is governed by the global variables `ROMBERGTOL`[1.E-4] and `ROMBERGIT`[11]. ROMBERG will return a result if the relative difference in successive approximations is less than `ROMBERGTOL`. It will try halving the stepsize `ROMBERGIT` times before it gives up.

## TLDEFINT (*exp,var,ll,ul*)

Function

is just `LDEFINT` with `TLIMSWITCH` set to `TRUE`.

## 20 Equations

### 20.1 Definitions for Equations

#### **%RNUM\_LIST**

Variable

default: [] - When %R variables are introduced in solutions by the ALGSYS command, they are added to %RNUM\_LIST in the order they are created. This is convenient for doing substitutions into the solution later on. It's recommended to use this list rather than doing CONCAT('%R,J).

#### **ALGEXACT**

Variable

default: [FALSE] affects the behavior of ALGSYS as follows: If ALGEXACT is TRUE, ALGSYS always calls SOLVE and then uses REALROOTS on SOLVE's failures. If ALGEXACT is FALSE, SOLVE is called only if the eliminant was not univariate, or if it was a quadratic or biquadratic. Thus ALGEXACT:TRUE doesn't guarantee only exact solutions, just that ALGSYS will first try as hard as it can to give exact solutions, and only yield approximations when all else fails.

#### **ALGSYS** ([*exp1*, *exp2*, ...], [*var1*, *var2*, ...])

Function

solves the list of simultaneous polynomials or polynomial equations (which can be non-linear) for the list of variables. The symbols %R1, %R2, etc. will be used to represent arbitrary parameters when needed for the solution (the variable %RNUM\_LIST holds these). In the process described below, ALGSYS is entered recursively if necessary. The method is as follows: (1) First the equations are FACTORed and split into subsystems. (2) For each subsystem  $S_i$ , an equation  $E$  and a variable  $var$  are selected (the  $var$  is chosen to have lowest nonzero degree). Then the resultant of  $E$  and  $E_j$  with respect to  $var$  is computed for each of the remaining equations  $E_j$  in the subsystem  $S_i$ . This yields a new subsystem  $S'_i$  in one fewer variables ( $var$  has been eliminated). The process now returns to (1). (3) Eventually, a subsystem consisting of a single equation is obtained. If the equation is multivariate and no approximations in the form of floating point numbers have been introduced, then SOLVE is called to find an exact solution. (The user should realize that SOLVE may not be able to produce a solution or if it does the solution may be a very large expression.) If the equation is univariate and is either linear, quadratic, or bi-quadratic, then again SOLVE is called if no approximations have been introduced. If approximations have been introduced or the equation is not univariate and neither linear, quadratic, or bi-quadratic, then if the switch REALONLY[FALSE] is TRUE, the function REALROOTS is called to find the real-valued solutions. If REALONLY:FALSE then ALLROOTS is called which looks for real and complex-valued solutions. If ALGSYS produces a solution which has fewer significant digits than required, the user can change the value of ALGEPSEILON[10<sup>-8</sup>] to a higher value. If ALGEXACT[FALSE] is set to TRUE, SOLVE will always be called. (4) Finally, the solutions obtained in step (3) are re-inserted into previous levels and the solution process returns to (1). The user should be aware of several caveats: When ALGSYS encounters a multivariate equation which contains floating point approximations (usually due to its failing to find exact solutions at an earlier

stage), then it does not attempt to apply exact methods to such equations and instead prints the message: "ALGSYS cannot solve - system too complicated." Interactions with RADCAN can produce large or complicated expressions. In that case, the user may use PICKAPART or REVEAL to analyze the solution. Occasionally, RADCAN may introduce an apparent %I into a solution which is actually real-valued. Do EXAMPLE(ALGSYS); for examples.

**ALLROOTS** (*poly*)

Function

finds all the real and complex roots of the real polynomial poly which must be univariate and may be an equation, e.g. poly=0. For complex polynomials an algorithm by Jenkins and Traub is used (Algorithm 419, Comm. ACM, vol. 15, (1972), p. 97). For real polynomials the algorithm used is due to Jenkins (Algorithm 493, TOMS, vol. 1, (1975), p.178). The flag POLYFACTOR[FALSE] when true causes ALLROOTS to factor the polynomial over the real numbers if the polynomial is real, or over the complex numbers, if the polynomial is complex. ALLROOTS may give inaccurate results in case of multiple roots. (If poly is real and you get inaccurate answers, you may want to try ALLROOTS(%I\*poly);) Do EXAMPLE(ALLROOTS); for an example. ALLROOTS rejects non-polynomials. It requires that the numerator after RATting should be a polynomial, and it requires that the denominator be at most a complex number. As a result of this ALLROOTS will always return an equivalent (but factored) expression, if POLYFACTOR is TRUE.

**BACKSUBST**

Variable

default: [TRUE] if set to FALSE will prevent back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions. (On MC this could cause storage capacity to be exceeded.)

**BREAKUP**

Variable

default: [TRUE] if FALSE will cause SOLVE to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default. BREAKUP:TRUE only works when PROGRAM-MODE is FALSE.

**DIMENSION** (*equation or list of equations*)

Function

The file "share1/dimen.mc" contains functions for automatic dimensional analysis. LOAD(DIMEN); will load it up for you. There is a demonstration available in share1/dimen.dem. Do DEMO("dimen"); to run it.

**DISPFLAG**

Variable

default: [TRUE] if set to FALSE within a BLOCK will inhibit the display of output generated by the solve functions called from within the BLOCK. Termination of the BLOCK with a dollar sign, \$, sets DISPFLAG to FALSE.

**FUNCSOLVE** (*eqn,g(t)*)

Function

gives [g(t) = ...] or [], depending on whether or not there exists a rational fcn g(t) satisfying eqn, which must be a first order, linear polynomial in (for this case) g(t) and g(t+1).

$$(C1) \text{ FUNCSOLVE}((N+1)*\text{FOO}(N) - (N+3)*\text{FOO}(N+1)/(N+1) = \\ (N-1)/(N+2), \text{FOO}(N));$$

$$(D1) \quad \text{FOO}(N) = \frac{N}{(N+1)(N+2)}$$

Warning: this is a very rudimentary implementation—many safety checks and obvious generalizations are missing.

### GLOBALSOLVE

Variable

default: [FALSE] if set to TRUE then variables which are SOLVED for will be set to the solution of the set of simultaneous equations.

### IEQN (*ie, unk, tech, n, guess*)

Function

Integral Equation solving routine. Do LOAD(INTEQN); to access it. CAVEAT: To free some storage, a KILL(LABELS) is included in this file. Therefore, before loading the integral equation package, the user should give names to any expressions he wants to keep. *ie* is the integral equation; *unk* is the unknown function; *tech* is the technique to be tried from those given above (*tech* = FIRST means: try the first technique which finds a solution; *tech* = ALL means: try all applicable techniques); *n* is the maximum number of terms to take for TAYLOR, NEUMANN, FIRSTKINDSERIES, or FREDSERIES (it is also the maximum depth of recursion for the differentiation method); *guess* is the initial guess for NEUMANN or FIRSTKINDSERIES. Default values for the 2nd thru 5th parameters are: *unk*: P(X), where P is the first function encountered in an integrand which is unknown to MACSYMA and X is the variable which occurs as an argument to the first occurrence of P found outside of an integral in the case of SECONDKIND equations, or is the only other variable besides the variable of integration in FIRSTKIND equations. If the attempt to search for X fails, the user will be asked to supply the independent variable; *tech*: FIRST; *n*: 1; *guess*: NONE, which will cause NEUMANN and FIRSTKINDSERIES to use F(X) as an initial guess.

### IEQNPRINT

Variable

default: [TRUE] - governs the behavior of the result returned by the IEQN command (which see). If IEQNPRINT is set to FALSE, the lists returned by the IEQN function are of the form [SOLUTION, TECHNIQUE USED, NTERMS, FLAG] where FLAG is absent if the solution is exact. Otherwise, it is the word APPROXIMATE or INCOMPLETE corresponding to an inexact or non-closed form solution, respectively. If a series method was used, NTERMS gives the number of terms taken (which could be less than the *n* given to IEQN if an error prevented generation of further terms).

### LHS (*eqn*)

Function

the left side of the equation *eqn*.

### LINSOLVE ([*exp1, exp2, ...*], [*var1, var2, ...*])

Function

solves the list of simultaneous linear equations for the list of variables. The *exp*i must each be polynomials in the variables and may be equations. If GLOBAL-SOLVE[FALSE] is set to TRUE then variables which are SOLVED for will be set

to the solution of the set of simultaneous equations. `BACKSUBST[TRUE]` if set to `FALSE` will prevent back substitution after the equations have been triangularized. This may be necessary in very big problems where back substitution would cause the generation of extremely large expressions. (On MC this could cause the storage capacity to be exceeded.) `LINSOLVE_PARAMS[TRUE]` If `TRUE`, `LINSOLVE` also generates the `%Ri` symbols used to represent arbitrary parameters described in the manual under `ALGSYS`. If `FALSE`, `LINSOLVE` behaves as before, i.e. when it meets up with an under-determined system of equations, it solves for some of the variables in terms of others.

```
(C1) X+Z=Y$
(C2) 2*A*X-Y=2*A**2$
(C3) Y-2*Z=2$
(C4) LINSOLVE([D1,D2,D3],[X,Y,Z]),GLOBALSOLVE:TRUE;
SOLUTION
(E4) X : A + 1
(E5) Y : 2 A
(E6) Z : A - 1
(D6) [E4, E5, E6]
```

**LINSOLVEWARN**

Variable

default: `[TRUE]` - if `FALSE` will cause the message "Dependent equations eliminated" to be suppressed.

**LINSOLVE\_PARAMS**

Variable

default: `[TRUE]` - If `TRUE`, `LINSOLVE` also generates the `%Ri` symbols used to represent arbitrary parameters described in the manual under `ALGSYS`. If `FALSE`, `LINSOLVE` behaves as before, i.e. when it meets up with an under-determined system of equations, it solves for some of the variables in terms of others.

**MULTIPLICITIES**

Variable

default: `[NOT_SET_YET]` - will be set to a list of the multiplicities of the individual solutions returned by `SOLVE` or `REALROOTS`.

**NROOTS** (*poly*, *low*, *high*)

Function

finds the number of real roots of the real univariate polynomial `poly` in the half-open interval `[low,high]`. The endpoints of the interval may also be `MINF,INF` respectively for minus infinity and plus infinity. The method of Sturm sequences is used.

```
(C1) POLY1:X**10-2*X**4+1/2$
(C2) NROOTS(POLY1,-6,9.1);
RAT REPLACED 0.5 BY 1/2 = 0.5
(D2)
```

**NTHROOT** ( $p,n$ ) Function

where  $p$  is a polynomial with integer coefficients and  $n$  is a positive integer returns  $q$ , a polynomial over the integers, such that  $q^n=p$  or prints an error message indicating that  $p$  is not a perfect  $n$ th power. This routine is much faster than FACTOR or even SQFR.

**PROGRAMMODE** Variable

default: [TRUE] - when FALSE will cause SOLVE, REALROOTS, ALLROOTS, and LINSOLVE to print E-labels (intermediate line labels) to label answers. When TRUE, SOLVE, etc. return answers as elements in a list. (Except when BACKSUBST is set to FALSE, in which case PROGRAMMODE:FALSE is also used.)

**REALONLY** Variable

default: [FALSE] - if TRUE causes ALGSYS to return only those solutions which are free of %I.

**REALROOTS** (*poly, bound*) Function

finds all of the real roots of the real univariate polynomial  $poly$  within a tolerance of  $bound$  which, if less than 1, causes all integral roots to be found exactly. The parameter  $bound$  may be arbitrarily small in order to achieve any desired accuracy. The first argument may also be an equation. REALROOTS sets MULTIPLICITIES, useful in case of multiple roots. REALROOTS( $poly$ ) is equivalent to REALROOTS( $poly,ROOTSEPSILON$ ). ROOTSEPSILON[1.0E-7] is a real number used to establish the confidence interval for the roots. Do EXAMPLE(REALROOTS); for an example.

**RHS** (*eqn*) Function

the right side of the equation  $eqn$ .

**ROOTSCONMODE** Variable

default: [TRUE] - Determines the behavior of the ROOTS CONTRACT command. Do DESCRIBE(ROOTS CONTRACT); for details.

**ROOTS CONTRACT** (*exp*) Function

converts products of roots into roots of products. For example,

$$\text{ROOTS CONTRACT}(\text{SQRT}(X)*Y^{(3/2)}) \implies \text{SQRT}(X*Y^3)$$

When RADEXPAND is TRUE and DOMAIN is REAL (their defaults), ROOTS CONTRACT converts ABS into SQRT, e.g.

$$\text{ROOTS CONTRACT}(\text{ABS}(X)*\text{SQRT}(Y)) \implies \text{SQRT}(X^2*Y)$$

There is an option ROOTSCONMODE (default value TRUE), affecting ROOTS CONTRACT as follows:

Problem	Value of ROOTSCONMODE	Result of applying ROOTS CONTRACT
---------	--------------------------	--------------------------------------

$X^{1/2} * Y^{3/2}$	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	$(X^3 * Y^2)^{1/6}$

The above examples and more may be tried out by typing

```
EXAMPLE(ROOTSCONTRACT);
```

When `ROOTSCONMODE` is `FALSE`, `ROOTSCONTRACT` contracts only wrt 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 lcm (least common multiple) of the denominators of the exponents. `ROOTSCONTRACT` uses `RATSIMP` in a manner similar to `LOGCONTRACT` (see the manual).

## ROOTSEPSILON

Variable

default: `[1.0E-7]` - a real number used to establish the confidence interval for the roots found by the `REALROOTS` function.

## SOLVE (*exp*, *var*)

Function

solves the algebraic equation `exp` for the variable `var` and returns a list of solution equations in `var`. If `exp` is not an equation, it is assumed to be an expression to be set equal to zero. `Var` may be a function (e.g. `F(X)`), or other non-atomic expression except a sum or product. It may be omitted if `exp` contains only one variable. `Exp` may be a rational expression, and may contain trigonometric functions, exponentials, etc. The following method is used: Let `E` be the expression and `X` be the variable. If `E` is linear in `X` then it is trivially solved for `X`. Otherwise if `E` is of the form  $A * X^N + B$  then the result is  $(-B/A)^{1/N}$  times the `N`th roots of unity. If `E` is not linear in `X` then the gcd of the exponents of `X` in `E` (say `N`) is divided into the exponents and the multiplicity of the roots is multiplied by `N`. Then `SOLVE` is called again on the result. If `E` factors then `SOLVE` is called on each of the factors. Finally `SOLVE` will use the quadratic, cubic, or quartic formulas where necessary. In the case where `E` is a polynomial in some function of the variable to be solved for, say `F(X)`, then it is first solved for `F(X)` (call the result `C`), then the equation `F(X)=C` can be solved for `X` provided the inverse of the function `F` is known. `BREAKUP[TRUE]` if `FALSE` will cause `SOLVE` to express the solutions of cubic or quartic equations as single expressions rather than as made up of several common subexpressions which is the default. `MULTIPLICITIES[NOT_SET_YET]` - will be set to a list of the multiplicities of the individual solutions returned by `SOLVE`, `REALROOTS`, or `ALLROOTS`. Try `APROPOS(SOLVE)` for the switches which affect `SOLVE`. `DESCRIBE` may then be used on the individual switch names if their purpose is not clear. `SOLVE([eq1, ..., eqn], [v1, ..., vn])` solves a system of simultaneous (linear or non-linear) polynomial equations by calling `LINSOLVE` or `ALGSYS` and returns a list of the solution lists in the variables. In the case of `LINSOLVE` this list would contain a single list of solutions. It takes two lists as arguments. The first list (`eqi`,

$i=1,\dots,n$ ) represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted. For linear systems if the given equations are not compatible, the message `INCONSISTENT` will be displayed (see the `SOLVE_INCONSISTENT_ERROR` switch); if no unique solution exists, then `SINGULAR` will be displayed. For examples, do `EXAMPLE(SOLVE)`;

**SOLVEDECOMPOSES** Variable  
 default: `[TRUE]` - if `TRUE`, will induce `SOLVE` to use `POLYDECOMP` (see `POLYDECOMP`) in attempting to solve polynomials.

**SOLVEEXPLICIT** Variable  
 default: `[FALSE]` - if `TRUE`, inhibits `SOLVE` from returning implicit solutions i.e. of the form  $F(x)=0$ .

**SOLVEFACTORS** Variable  
 default: `[TRUE]` - if `FALSE` then `SOLVE` will not try to factor the expression. The `FALSE` setting may be desired in some cases where factoring is not necessary.

**SOLVENULLWARN** Variable  
 default: `[TRUE]` - if `TRUE` the user will be warned if he calls `SOLVE` with either a null equation list or a null variable list. For example, `SOLVE([],[])`; would print two warning messages and return `[]`.

**SOLVERADCAN** Variable  
 default: `[FALSE]` - if `TRUE` then `SOLVE` will use `RADCAN` which will make `SOLVE` slower but will allow certain problems containing exponentials and logs to be solved.

**SOLVETRIGWARN** Variable  
 default: `[TRUE]` - if set to `FALSE` will inhibit printing by `SOLVE` of the warning message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.

**SOLVE\_INCONSISTENT\_ERROR** Variable  
 default: `[TRUE]` - If `TRUE`, `SOLVE` and `LINSOLVE` give an error if they meet up with a set of inconsistent linear equations, e.g. `SOLVE([A+B=1,A+B=2])`. If `FALSE`, they return `[]` in this case. (This is the new mode, previously gotten only by calling `ALGSYS`.)

**ZRPOLY** Function  
 - The IMSL `ZRPOLY` routine for finding the zeros of simple polynomials (single variable, real coefficients, non-negative integer exponents), using the Jenkins-Traub technique. To use it, do: `LOADFILE("imsl");` The command is `POLYROOTS(polynomial);` For more info, do: `PRINTFILE("zrpoly.usg");` For a demo do: `DEMO("zrpoly.dem");` For general info on `MACSYMA-IMSL` packages, `PRINTFILE(IMSL,USAGE,SHARE2);`



**ZSOLVE**

Function

- For those who can make use of approximate numerical solutions to problems, there is a package which calls a routine which has been translated from the IMSL fortran library to solve N simultaneous non-linear equations in N unknowns. It uses black-box techniques that probably aren't desirable if an exact solution can be obtained from one of the smarter solvers (LINSOLVE, ALGSYS, etc). But for things that the other solvers don't attempt to handle, this can probably give some very useful results. For documentation, do `PRINTFILE("zsolve.usg");`. For a demo do `batch("zsolve.mc")$`

## 21 Differential Equations

### 21.1 Definitions for Differential Equations

**DESOLVE** ([eq1,...,eqn],[var1,...,varn]) Function

where the eq's are differential equations in the dependent variables var1,...,varn. The functional relationships must be explicitly indicated in both the equations and the variables. For example

(C1) 'DIFF(F,X,2)=SIN(X)+'DIFF(G,X);

(C2) 'DIFF(F,X)+X^2-F=2\*'DIFF(G,X,2);

is NOT the proper format. The correct way is:

(C3) 'DIFF(F(X),X,2)=SIN(X)+'DIFF(G(X),X);

(C4) 'DIFF(F(X),X)+X^2-F(X)=2\*'DIFF(G(X),X,2);

The call is then DESOLVE([D3,D4],[F(X),G(X)]);

If initial conditions at 0 are known, they should be supplied before calling DESOLVE by using ATVALUE.

(C11) 'DIFF(F(X),X)='DIFF(G(X),X)+SIN(X);

(D11) 
$$-- F(X) = -- G(X) + SIN(X)$$

(C12) 'DIFF(G(X),X,2)='DIFF(F(X),X)-COS(X);

(D12) 
$$--- G(X) = -- F(X) - COS(X)$$

(C13) ATVALUE('DIFF(G(X),X),X=0,A);

(D13) 
$$A$$

(C14) ATVALUE(F(X),X=0,1);

(D14) 
$$1$$

(C15) DESOLVE([D11,D12],[F(X),G(X)]);

(D16) 
$$[F(X)=A \%E^{-X} - A + 1, G(X) = \cos(X) + A \%E^{-X} - A + G(0) - 1]$$

/\* VERIFICATION \*/

(C17) [D11,D12],D16,DIFF;

(D17) 
$$[A \%E^{-X} = A \%E^{-X}, A \%E^{-X} - \cos(X) = A \%E^{-X} - \cos(X)]$$

If DESOLVE cannot obtain a solution, it returns "FALSE".

**IC1** (exp,var,var) Function

In order to solve initial value problems (IVPs) and boundary value problems (BVPs), the routine IC1 is available in the ODE2 package for first order equations, and IC2 and BC2 for second order IVPs and BVPs, respectively. Do LOAD(ODE2) to access these. They are used as in the following examples:

```

(C3) IC1(D2,X=%PI,Y=0);
(D3)          COS(X) + 1
            Y = - ----
                  3
                  X
(C4) 'DIFF(Y,X,2) + Y*'DIFF(Y,X)^3 = 0;
(D4)          d Y      dY 3
            --- + Y (--) = 0
              2      dX
            dX
(C5) ODE2(%Y,X);
(D7)          3
            Y  - 6 %K1 Y - 6 X
            ----- = %K2
              3
(C8) RATSIMP(IC2(D7,X=0,Y=0,'DIFF(Y,X)=2));
(D9)          3
            2 Y  - 3 Y + 6 X
            ----- = 0
              3
(C10) BC2(D7,X=0,Y=1,X=1,Y=3);
(D11)         3
            Y  - 10 Y - 6 X
            ----- = - 3
              3

```

**ODE** (*equation,y,x*)

Function

a pot-pourri of Ordinary Differential solvers combined in such a way as to attempt more and more difficult methods as each fails. For example, the first attempt is with ODE2, so therefore, a user using ODE can assume he has all the capabilities of ODE2 at the very beginning and if he has been using ODE2 in programs they will still run if he substitutes ODE (the returned values, and calling sequence are identical). In addition, ODE has a number of user features which can assist an experienced ODE solver if the basic system cannot handle the equation. The equation is of the same form as required for ODE2 (which see) and the y and x are dependent and independent variables, as with ODE2. For more details, do `PRINTFILE(ODE,USAGE,SHARE)`;

**ODE2** (*exp,dvar,ivar*)

Function

takes three arguments: an ODE of first or second order (only the left hand side need be given if the right hand side is 0), the dependent variable, and the independent variable. When successful, it returns either an explicit or implicit solution for the dependent variable. %C is used to represent the constant in the case of first order equations, and %K1 and %K2 the constants for second order equations. If ODE2 cannot obtain a solution for whatever reason, it returns FALSE, after perhaps printing out an error message. The methods implemented for first order equations in

the order in which they are tested are: linear, separable, exact - perhaps requiring an integrating factor, homogeneous, Bernoulli's equation, and a generalized homogeneous method. For second order: constant coefficient, exact, linear homogeneous with non-constant coefficients which can be transformed to constant coefficient, the Euler or equidimensional equation, the method of variation of parameters, and equations which are free of either the independent or of the dependent variable so that they can be reduced to two first order linear equations to be solved sequentially. In the course of solving ODEs, several variables are set purely for informational purposes: METHOD denotes the method of solution used e.g. LINEAR, INTFACTOR denotes any integrating factor used, ODEINDEX denotes the index for Bernoulli's method or for the generalized homogeneous method, and YP denotes the particular solution for the variation of parameters technique.



## 22 Numerical

### 22.1 Introduction to Numerical

### 22.2 DCADRE

The following is obsolete. To make an interface to fortran libraries in the current MAXIMA look at the examples in "maxima/src/fortdef.lsp" - The IMSL version of Romberg integration is now available in Macsyma. For documentation, Do `PRINTFILE(DCADRE,USAGE,IMSL1);` . For a demo, do `batch("dcadre.mc");` This is a numerical integration package using cautious, adaptive Romberg extrapolation. The DCADRE package is written to call the IMSL fortran library routine DCADRE. This is documentation for that program. Send bugs/comments to KMP To load this package, do

```
LOADFILE("imsl")$
```

For a demo of this package, do

```
batch("dcadre.mc");
```

The worker function takes the following syntax: `IMSL_ROMBERG(fn,low,hi)` where `fn` is a function of 1 argument; `low` and `hi` should be the lower and upper bounds of integration. `fn` must return floating point values. `IMSL_ROMBERG(exp,var,low,hi)` where `exp` should be integrated over the range `var=low` to `hi`. The result of evaluating `exp` must always be a floating point number. `FAST_IMSL_ROMBERG(fn,low,hi)` This function does no error checking but may achieve a speed gain over the `IMSL_ROMBERG` function. It expects that `fn` is a Lisp function (or translated Macsyma function) which accepts a floating point argument and that it always returns a floating point value.

Returns either `[SUCCESS, answer, error]` where `answer` is the result of the integration and `error` is the estimated bound on the absolute error of the output, DCADRE, as described in PURPOSE below. or `[WARNING, n, answer, error]` where `n` is a warning code, `answer` is the answer, and `error` is the estimated bound on the absolute error of the output, DCADRE, as described in PURPOSE below. The following warnings may occur: 65 = One or more singularities were successfully handled. 66 = In some subinterval(s), the estimate of the integral was accepted merely because the estimated error was small, even though no regular behavior was recognized. or `[ERROR, errorcode]` where `error code` is the IMSL-generated error code. The following error codes may occur: 131 = Failure due to insufficient internal working storage. 132 = Failure. This may be due to too much noise in function (relative to the given error requirements) or due to an ill-behaved integrand. 133 = RERR is greater than 0.1 or less than 0.0 or is too small for the precision of the machine.

The following flags have an influence upon the operation of `IMSL_ROMBERG` –

`ROMBERG_AERR` [Default 1.0E-5] – Desired absolute error in answer.

`ROMBERG_RERR` [Default 0.0] – Desired relative error in the answer.

Note: If IMSL signals an error, a message will be printed on the user's console stating the nature of the error. (This error message may be suppressed by setting `IMSLVERBOSE` to FALSE.)

Note: Because this uses a translated Fortran routine, it may not be recursively invoked. It does not call itself, but the user should be aware that he may not type ^A in the middle of an IMSL\_ROMBERG computation, begin another calculation using the same package, and expect to win – IMSL\_ROMBERG will complain if it was already doing one project when you invoke it. This should cause minimal problems.

Purpose (modified version of the IMSL documentation)

DCADRE attempts to solve the following problem: Given a real-valued function F of one argument, two real numbers A and B, find a number

DCADRE such that:

$$\left| \frac{\int_A^B F(x) dx - \text{DCADRE}}{\int_A^B F(x) dx} \right| \leq \max \left[ \text{ROMBERG\_AERR}, \text{ROMBERG\_RERR} * \frac{\int_A^B F(x) dx}{B - A} \right]$$

Algorithm (modified version of the IMSL documentation)

This routine uses a scheme whereby DCADRE is computed as the sum of estimates for the integral of F(x) over suitably chosen subintervals of the given interval of integration. Starting with the interval of integration itself as the first such subinterval, cautious Romberg extrapolation is used to find an acceptable estimate on a given subinterval. If this attempt fails, the subinterval is divided into two subintervals of equal length, each of which is considered separately. Programming Notes (modified version of the IMSL documentation)

- 1. DCADRE (the translated-Fortran base for IMSL\_ROMBERG) can, in many cases, handle jump discontinuities and certain algebraic discontinuities. See reference for full details.
- 2. The relative error parameter ROMBERG\_RERR must be in the interval [0.0,0.1]. For example, ROMBERG\_RERR=0.1 indicates that the estimate of the intergral is to be correct to one digit, where as ROMBERG\_RERR=1.0E-4 calls for four digits of accuracy. If DCADRE determines that the relative accuracy requirement cannot be satisfied, IER is set to 133 (ROMBERG\_RERR should be large enough that, when added to 100.0, the result is a number greater than 100.0 (this will not be true of very tiny floating point numbers due to the nature of machine arithmetic)).
- 3. The absolute error parameter, ROMBERG\_AERR, should be nonnegative. In order to give a reasonable value for ROMBERG\_AERR, the user must know the approximate magnitude of the integral being computed. In many cases, it is satisfactory to use AERR=0.0. In this case, only the relative error requirement is satisfied in the computation.
- 4. We quote from the reference, “A very cautious man would accept DCADRE only if IER [the warning or error code] is 0 or 65. The merely reasonable man would keep the faith even if IER is 66. The adventurous man is quite often right in accepting DCADRE even if the IER is 131 or 132.” Even when IER is not 0, DCADRE returns the best estimate that has been computed.

For references on this technique, see de Boor, Calr, “CADRE: An Algorithm for Numerical Quadrature,” Mathematical Software (John R. Rice, Ed.), New York, Academic Press, 1971, Chapter 7.

## 22.3 ELLIPT

- A package on the SHARE directory for Numerical routines for Elliptic Functions and Complete Elliptic Integrals. (Notation of Abramowitz and Stegun, Chs 16 and 17) Do `LOAD(ELLIPT)`; to use this package. At present all arguments **MUST** be floating point. You'll get nonsense otherwise. Be warned. The functions available are: Jacobian elliptic functions

```
AM(U,M) - amplitude with modulus M
AM1(U,M1) - amplitude with complementary modulus M1
AM(U,M):=AM1(U,1-M); so use AM1 if M ~ 1
SN(U,M):=SIN(AM(U,M));
CN(U,M):=COS(AM(U,M));
DN(U,M):=SQRT(1-M*SN(U,M)^2);
(These functions come defined like this. Others CD, NS etc. may be
similarly defined.)
Complete Elliptic Integrals
ELLIPTK(M) - Complete elliptic integral of first kind
ELLIPTK1(M1) - Same but with complementary modulus.
ELLIPTK(M):=ELLIPTK1(1-M); so use if M ~ 1
ELLIPTE(M) - Complete elliptic integral of second kind
ELLIPTE1(M1) - Same but with complementary modulus.
ELLIPTE(M):=ELLIPTE1(1-M); so use if M ~ 1
```

## 22.4 FOURIER

- There is a Fast Fourier Transform package, do `DESCRIBE(FFT)` for details. There is also a Fourier Series package. It may be loaded with `LOAD(FOURIE)`. It will also calculate Fourier integral coefficients and has various other functions to do such things as replace all occurrences of `F(ARG)` by `ARG` in expression (like changing `ABS(a*x+b)` to `a*x+b`). Do `PRINTFILE(FOURIE,USAGE,DSK,SHARE1)`; for a list of the functions included.

## 22.5 NDIFFQ

a package residing on the SHARE directory for numerical solutions of differential equations. `LOAD("NDIFFQ")`; will load it in for use. An example of its use would be:

```
Define_Variable(N,0.3,FLOAT);
Define_Variable(H,0.175,FLOAT);
F(X,E):=(Mode_Declare([X,E],FLOAT),N*EXP(X)/(E+X^(2*H)*EXP(H*X)));
Compile(F);
Array([X,E],FLOAT,35);
Init_Float_Array(X,1.0E-3,6.85); /* Fills X with the interval */
E[0]:5.0; /* Initial condition */
Runge_Kutta(F,X,E); /* Solve it */
Graph2(X,E); /* Graph the solution */
```

p.s. `Runge_Kutta(F,X,E,E-Prime)` would be the call for a second-order equation.



## 22.6 Definitions for Numerical

**FFT** (*real-array, imag-array*) Function

Fast Fourier Transform. This package may be loaded by doing `LOAD(FFT)`; There is also an `IFT` command, for Inverse Fourier Transform. These functions perform a (complex) fast fourier transform on either 1 or 2 dimensional `FLOATING-POINT` arrays, obtained by:

```
ARRAY(<ary>,FLOAT,<dim1>); or
ARRAY(<ary>,FLOAT,<dim1>,<dim2>);
```

For 1D arrays

```
<dim1> = 2^n-1
```

and for 2D arrays

```
<dim1>=<dim2>=2^n-1
```

(i.e. the array is square). (Recall that MACSYMA arrays are indexed from a 0 origin so that there will be  $2^n$  and  $(2^n)^2$  arrays elements in the above two cases.) This package also contains two other functions, `POLARTORECT` and `RECTTOPOLAR`. Do `DESCRIBE(cmd)` for details. For details on the implementation, do `PRINT-FILE(FFT,USAGE,SHARE)`; .

**FORTINDENT** Variable

default: [0] - controls the left margin indentation of expressions printed out by the FORTRAN command. 0 gives normal printout (i.e. 6 spaces), and positive values will causes the expressions to be printed farther to the right.

**FORTMX** (*name,matrix*) Function

converts a MACSYMA matrix into a sequence of FORTRAN assignment statements of the form `name(i,j)=<corresponding matrix element>`. This command is now obsolete. `FORTMX(name,matrix)`; may now be done as `FORTRAN(name=matrix)`; (If "name" is bound, `FORTRAN('name=matrix)`; may be necessary.) Please convert code that uses the `FORTMX` command as it may be flushed some day.

**FORTRAN** (*exp*) Function

converts `exp` into a FORTRAN linear expression in legal FORTRAN with 6 spaces inserted at the beginning of each line, continuation lines, and `**` rather than `^` for exponentiation. When the option `FORTSPACES[FALSE]` is `TRUE`, the FORTRAN command fills out to 80 columns using spaces. If FORTRAN is called on a bound symbolic atom, e.g. `FORTRAN(X)`; where `X:A*B$` has been done, then `X={value of X}`, e.g. `X=A*B` will be generated. In particular, if e.g. `M:MATRIX(...)`; has been done, then `FORTRAN(M)`; will generate the appropriate assignment statements of the form `name(i,j)=<corresponding matrix element>`. `FORTINDENT[0]` controls the left margin of expressions printed out, 0 is the normal margin (i.e. indented 6 spaces), increasing it will cause the expression to be printed further to the right.

**FORTSPACES** Variable

default: [FALSE] - if `TRUE`, the FORTRAN command fills out to 80 columns using spaces.

**HORNER** (*exp, var*)

Function

will convert *exp* into a rearranged representation as in Horner's rule, using *var* as the main variable if it is specified. *Var* may also be omitted in which case the main variable of the CRE form of *exp* is used. **HORNER** sometimes improves stability if *expr* is to be numerically evaluated. It is also useful if **MACSYMA** is used to generate programs to be run in **FORTRAN** (see **DESCRIBE**(**STRINGOUT**);)

```
(C1) 1.0E-20*X^2-5.5*X+5.2E20;
      2
      1.0E-20 X  - 5.5 X + 5.2E+20
(D1)
(C2) HORNER(%,X),KEEPFLOAT:TRUE;
      X (1.0E-20 X - 5.5) + 5.2E+20
(D2)
(C3) D1,X=1.0E20;
      ARITHMETIC OVERFLOW
(C4) D2,X=1.0E20;
      6.9999999E+19
(D4)
```

**IFT** (*real-array, imag-array*)

Function

Inverse Fourier Transform. Do **LOAD**(**FFT**); to load in this package. These functions (**FFT** and **IFT**) perform a (complex) fast fourier transform on either 1 or 2 dimensional **FLOATING-POINT** arrays, obtained by: **ARRAY**(<ary>, **FLOAT**, <dim1>); or **ARRAY**(<ary>, **FLOAT**, <dim1>, <dim2>); For 1D arrays <dim1> must equal  $2^n-1$ , and for 2D arrays <dim1>=<dim2>= $2^n-1$  (i.e. the array is square). (Recall that **MACSYMA** arrays are indexed from a 0 origin so that there will be  $2^n$  and  $(2^n)^2$  arrays elements in the above two cases.) For details on the implementation, do **PRINTFILE**(**FFT**, **USAGE**, **SHARE**); .

**INTERPOLATE** (*func,x,a,b*)

Function

finds the zero of *func* as *x* varies. The last two args give the range to look in. The function must have a different sign at each endpoint. If this condition is not met, the action of the of the function is governed by **INTPOLERROR**[**TRUE**]). If **INTPOLERROR** is **TRUE** then an error occurs, otherwise the value of **INTPOLERROR** is returned (thus for plotting **INTPOLERROR** might be set to 0.0). Otherwise (given that **MACSYMA** can evaluate the first argument in the specified range, and that it is continuous) **INTERPOLATE** is guaranteed to come up with the zero (or one of them if there is more than one zero). The accuracy of **INTERPOLATE** is governed by **INTPOLABS**[0.0] and **INTPOLREL**[0.0] which must be non-negative floating point numbers. **INTERPOLATE** will stop when the first arg evaluates to something less than or equal to **INTPOLABS** or if successive approximants to the root differ by no more than **INTPOLREL** \* <one of the approximants>. The default values of **INTPOLABS** and **INTPOLREL** are 0.0 so **INTERPOLATE** gets as good an answer as is possible with the single precision arithmetic we have. The first arg may be an equation. The order of the last two args is irrelevant. Thus

```
INTERPOLATE(SIN(X)=X/2,X,%PI,.1);
      is equivalent to
INTERPOLATE(SIN(X)=X/2,X,.1,%PI);
```

The method used is a binary search in the range specified by the last two args. When it thinks the function is close enough to being linear, it starts using linear interpolation. An alternative syntax has been added to `interpolate`, this replaces the first two arguments by a function name. The function **MUST** be TRANSLATED or compiled function of one argument. No checking of the result is done, so make sure the function returns a floating point number.

```
F(X):=(MODE_DECLARE(X,FLOAT),SIN(X)-X/2.0);
INTERPOLATE(SIN(X)-X/2,X,0.1,%PI)      time= 60 msec
INTERPOLATE(F(X),X,0.1,%PI);           time= 68 msec
TRANSLATE(F);
INTERPOLATE(F(X),X,0.1,%PI);           time= 26 msec
INTERPOLATE(F,0.1,%PI);                 time=  5 msec
```

There is also a Newton method interpolation routine, do `DESCRIBE(NEWTON);` .

### INTPOLABS

Variable

default: [0.0] - The accuracy of the `INTERPOLATE` command is governed by `INTPOLABS[0.0]` and `INTPOLREL[0.0]` which must be non-negative floating point numbers. `INTERPOLATE` will stop when the first arg evaluates to something less than or equal to `INTPOLABS` or if successive approximants to the root differ by no more than `INTPOLREL * <one of the approximants>`. The default values of `INTPOLABS` and `INTPOLREL` are 0.0 so `INTERPOLATE` gets as good an answer as is possible with the single precision arithmetic we have.

### INTPOLERROR

Variable

default: [TRUE] - Governs the behavior of `INTERPOLATE`. When `INTERPOLATE` is called, it determines whether or not the function to be interpolated satisfies the condition that the values of the function at the endpoints of the interpolation interval are opposite in sign. If they are of opposite sign, the interpolation proceeds. If they are of like sign, and `INTPOLERROR` is TRUE, then an error is signaled. If they are of like sign and `INTPOLERROR` is not TRUE, the value of `INTPOLERROR` is returned. Thus for plotting, `INTPOLERROR` might be set to 0.0.

### INTPOLREL

Variable

default: [0.0] - The accuracy of the `INTERPOLATE` command is governed by `INTPOLABS[0.0]` and `INTPOLREL[0.0]` which must be non-negative floating point numbers. `INTERPOLATE` will stop when the first arg evaluates to something less than or equal to `INTPOLABS` or if successive approximants to the root differ by no more than `INTPOLREL * <one of the approximants>`. The default values of `INTPOLABS` and `INTPOLREL` are 0.0 so `INTERPOLATE` gets as good an answer as is possible with the single precision arithmetic we have.

### NEWTON (*exp,var,X0,eps*)

Function

The file `NEWTON` 1 on the `SHARE` directory contains a function which will do interpolation using Newton's method. It may be accessed by `LOAD(NEWTON);` . The Newton method can do things that `INTERPOLATE` will refuse to handle, since `INTERPOLATE` requires that everything evaluate to a flonum. Thus `NEWTON(x^2-`

$a^2, x, a/2, a^2/100$ ); will say that it can't tell if  $\text{flonum} * a^2 < a^2/100$ . Doing `ASSUME(a>0)`; and then doing `NEWTON` again works. You get  $x = a + \text{<small flonum>} * a$  which is symbolic all the way. `INTERPOLATE(x^2 - a^2, x, a/2, 2*a)`; complains that  $.5 * a$  is not flonum... An adaptive integrator which uses the Newton-Cotes 8 panel quadrature rule is available in `SHARE1;QQ FASL`. Do `DESCRIBE(QQ)` for details.

**POLARTORECT** (*magnitude-array, phase-array*) Function

converts from magnitude and phase form into real and imaginary form putting the real part in the magnitude array and the imaginary part into the phase array

```
<real>=<magnitude>*COS(<phase>) ==>
<imaginary>=<magnitude>*SIN(<phase>)
```

This function is part of the FFT package. Do `LOAD(FFT)`; to use it. Like `FFT` and `IFT` this function accepts 1 or 2 dimensional arrays. However, the array dimensions need not be a power of 2, nor need the 2D arrays be square.

**RECTTOPOLAR** (*real-array, imag-array*) Function

undoes `POLARTORECT`. The phase is given in the range from  $-\%PI$  to  $\%PI$ . This function is part of the FFT package. Do `LOAD(FFT)`; to use it. Like `FFT` and `IFT` this function accepts 1 or 2 dimensional arrays. However, the array dimensions need not be a power of 2, nor need the 2D arrays be square.



## 23 Statistics

### 23.1 Definitions for Statistics

**GAUSS** (*mean, sd*) Function  
returns a random floating point number from a normal distribution with mean MEAN and standard deviation SD. This is part of the BESSEL function package, do LOAD(BESSEL); to use it.



## 24 Arrays and Tables

### 24.1 Definitions for Arrays and Tables

**ARRAY** (*name, dim1, dim2, ..., dimk*) Function

This sets up a k-dimensional array. A maximum of five dimensions may be used. The subscripts for the *i*th dimension are the integers running from 0 to *dimi*. If the user assigns to a subscripted variable before declaring the corresponding array, an undeclared array is set up. If the user has more than one array to be set up the same way, they may all be set up at the same time, by `ARRAY([list-of-names],dim1, dim2, ..., dimk)`. Undeclared arrays, otherwise known as hashed arrays (because hash coding is done on the subscripts), are more general than declared arrays. The user does not declare their maximum size, and they grow dynamically by hashing as more elements are assigned values. The subscripts of undeclared arrays need not even be numbers. However, unless an array is rather sparse, it is probably more efficient to declare it when possible than to leave it undeclared. The `ARRAY` function can be used to transform an undeclared array into a declared array.

**ARRAYAPPLY** (*array,[sub1, ... ,subk]*) Function

is like `APPLY` except the first argument is an array.

**ARRAYINFO** (*a*) Function

returns a list of information about the array *a*. For hashed arrays it returns a list of "HASHED", the number of subscripts, and the subscripts of every element which has a value. For declared arrays it returns a list of "DECLARED", the number of subscripts, and the bounds that were given the the `ARRAY` function when it was called on *a*. Do `EXAMPLE(ARRAYINFO)`; for an example.

**ARRAYMAKE** (*name,[i1,i2,...]*) Function

returns `name[i1,i2,...]`.

**ARRAYS** Variable

default: `[]` a list of all the arrays that have been allocated, both declared and undeclared. Functions which deal with arrays are: `ARRAY`, `ARRAYAPPLY`, `ARRAYINFO`, `ARRAYMAKE`, `FILLARRAY`, `LISTARRAY`, and `REARRAY`.

**BASHINDICES** (*expr*) Function

- transforms the expression *expr* by giving each summation and product a unique index. This gives `CHANGEVAR` greater precision when it is working with summations or products. The form of the unique index is `J<number>`. The quantity `<number>` is determined by referring to `GENSUMNUM`, which can be changed by the user. For example, `GENSUMNUM:0$` resets it.



**FILLARRAY** (*array,list-or-array*) Function

fills array from list-or-array. If array is a floating-point (integer) array then list-or-array should be either a list of floating-point (integer) numbers or another floating-point (integer) array. If the dimensions of the arrays are different array is filled in row-major order. If there are not enough elements in list-or-array the last element is used to fill out the rest of array. If there are too many the remaining ones are thrown away. FILLARRAY returns its first argument.

**GETCHAR** (*a, i*) Function

returns the *i*th character of the quoted string or atomic name *a*. This function is useful in manipulating the LABELS list.

**LISTARRAY** (*array*) Function

returns a list of the elements of a declared or hashed array. the order is row-major. Elements which you have not defined yet will be represented by #####.

**MAKE\_ARRAY** (*type,dim1,dim2,...,dimn*) Function

- creates an array. "type" may be 'ANY, 'FLONUM, 'FIXNUM, 'HASHED or 'FUNCTIONAL. This is similar to the ARRAY command, except that the created array is a functional array object. The advantage of this over ARRAY is that it doesn't have a name, and once a pointer to it goes away, it will also go away. e.g. Y:MAKE\_ARRAY(...); Y now points to an object which takes up space, but do Y:FALSE, and Y no longer points to that object, so the object will get garbage collected. Note: the "dim*i*" here are different from the ARRAY command, since they go from 0 to *i*-1, i.e. a "dimension" of 10 means you have elements from 0 to 9. Y:MAKE\_ARRAY('FUNCTIONAL,'F,'HASHED,1) - The second argument to MAKE\_ARRAY in this case is the function to call to calculate array elements, and the rest of the arguments are passed recursively to MAKE\_ARRAY to generate the "memory" for the array function object.

**REARRAY** (*array,dim1, ... ,dimk*) Function

can be used to change the size or dimensions of an array. The new array will be filled with the elements of the old one in row-major order. If the old array was too small, FALSE, 0.0 or 0 will be used to fill the remaining elements, depending on the type of the array. The type of the array cannot be changed.

**REMARRAY** (*name1, name2, ...*) Function

removes arrays and array associated functions and frees the storage occupied. If name is ALL then all arrays are removed. It may be necessary to use this function if it is desired to redefine the values in a hashed array.

**USE\_FAST\_ARRAYS** Variable

[TRUE on LispM] - If TRUE then only two types of arrays are recognized.

1) The art-q array (t in common lisp) which may have several dimensions indexed by integers, and may hold any lisp or maxcyma object as an entry. To construct such an

array, enter `A:MAKE_ARRAY(ANY,3,4)`; then `A` will have as value, an array with twelve slots, and the indexing is zero based.

2) The `Hash_table` array which is the default type of array created if one does `B[X+1]:Y^2` (and `B` is not already an array, a list, or a matrix— if it were one of these an error would be caused since `x+1` would not be a valid subscript for an array, a list or a matrix ). Its indices (also known as keys) may be any object. It only takes ONE KEY at a time (`B[X+1,U]:Y` would ignore the `u`) Referencing is done by `B[X+1]==> Y^2`. Of course the key may be a list, eg `B[[x+1,u]]:y` would be valid. This is incompatible with the old Macsyma hash arrays, but saves consing.

An advantage of storing the arrays as values of the symbol is that the usual conventions about local variables of a function apply to arrays as well. The `Hash_table` type also uses less consing and is more efficient than the old type of macsyma hashar. To obtain consistent behaviour in translated and compiled code set `TRANSLATE_FAST_ARRAYS [TRUE]` to be `TRUE`.



## 25 Matrices and Linear Algebra

### 25.1 Introduction to Matrices and Linear Algebra

#### 25.1.1 DOT

- . The dot operator, for matrix (non-commutative) multiplication. When "." is used in this way, spaces should be left on both sides of it, e.g.  $A \cdot B$ . This distinguishes it plainly from a decimal point in a floating point number. Do `APROPOS(DOT)`; for a list of the switches which affect the dot operator.

#### 25.1.2 VECTORS

- The file `SHARE;VECT >` contains a vector analysis package, `share/vect.dem` contains a corresponding demonstration, and `SHARE;VECT ORTH` contains definitions of various orthogonal curvilinear coordinate systems. `LOAD(VECT)`; will load this package for you. The vector analysis package can combine and simplify symbolic expressions including dot products and cross products, together with the gradient, divergence, curl, and Laplacian operators. The distribution of these operators over sums or products is under user control, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There is also a capability for deriving the scalar or vector potential of a field. The package contains the following commands: `VECTORSIMP`, `SCALEFACTORS`, `EXPRESS`, `POTENTIAL`, and `VECTORPOTENTIAL`. Do `DESCRIBE(cmd)` on these command names, or `PRINTFILE(VECT,USAGE,SHARE)`; for details. Warning: The `VECT` package declares "." to be a commutative operator.

### 25.2 Definitions for Matrices and Linear Algebra

**ADDCOL** ( $M, list1, list2, \dots, listn$ ) Function  
appends the column(s) given by the one or more lists (or matrices) onto the matrix  $M$ .

**ADDROW** ( $M, list1, list2, \dots, listn$ ) Function  
appends the row(s) given by the one or more lists (or matrices) onto the matrix  $M$ .

**ADJOINT** ( $matrix$ ) Function  
computes the adjoint of a matrix.

**AUGCOEFMATRIX** ( $[eq1, \dots], [var1, \dots]$ ) Function  
the augmented coefficient matrix for the variables  $var1, \dots$  of the system of linear equations  $eq1, \dots$ . This is the coefficient matrix with a column adjoined for the constant terms in each equation (i.e. those not dependent upon  $var1, \dots$ ). Do `EXAMPLE(AUGCOEFMATRIX)`; for an example.

- CHARPOLY** ( $M, var$ ) Function  
 computes the characteristic polynomial for Matrix  $M$  with respect to  $var$ . That is, `DETERMINANT(M - DIAGMATRIX(LENGTH(M),var))`. For examples of this command, do `EXAMPLE(CHARPOLY)`; .
- COEFMATRIX** ( $[eq1, \dots], [var1, \dots]$ ) Function  
 the coefficient matrix for the variables  $var1, \dots$  of the system of linear equations  $eq1, \dots$
- COL** ( $M, i$ ) Function  
 gives a matrix of the  $i$ th column of the matrix  $M$ .
- COLUMNVECTOR** ( $X$ ) Function  
 a function in the `EIGEN` package. Do `LOAD(EIGEN)` to use it. `COLUMNVECTOR` takes a `LIST` as its argument and returns a column vector the components of which are the elements of the list. The first element is the first component, ...etc... (This is useful if you want to use parts of the outputs of the functions in this package in matrix calculations.)
- CONJUGATE** ( $X$ ) Function  
 a function in the `EIGEN` package on the `SHARE` directory. It returns the complex conjugate of its argument. This package may be loaded by `LOAD(EIGEN)`; . For a complete description of this package, do `PRINTFILE("eigen.usg")`; .
- COPYMATRIX** ( $M$ ) Function  
 creates a copy of the matrix  $M$ . This is the only way to make a copy aside from recreating  $M$  elementwise. Copying a matrix may be useful when `SETELMX` is used.
- DETERMINANT** ( $M$ ) Function  
 computes the determinant of  $M$  by a method similar to Gaussian elimination. The form of the result depends upon the setting of the switch `RATMX`. There is a special routine for dealing with sparse determinants which can be used by setting the switches `RATMX:TRUE` and `SPARSE:TRUE`.
- DETOUT** Variable  
 default: `[FALSE]` if `TRUE` will cause the determinant of a matrix whose inverse is computed to be kept outside of the inverse. For this switch to have an effect `DOALLMXOPS` and `DOSCMXOPS` should be `FALSE` (see their descriptions). Alternatively this switch can be given to `EV` which causes the other two to be set correctly.
- DIAGMATRIX** ( $n, x$ ) Function  
 returns a diagonal matrix of size  $n$  by  $n$  with the diagonal elements all  $x$ . An identity matrix is created by `DIAGMATRIX(n,1)`, or one may use `IDENT(n)`.
- DOALLMXOPS** Variable  
 default: `[TRUE]` if `TRUE` all operations relating to matrices are carried out. If it is `FALSE` then the setting of the individual `DOT` switches govern which operations are performed.

- DOMXEXPT** Variable  
 default: [TRUE] if TRUE,  

$$\%E^{\text{MATRIX}([1,2],[3,4])} \Rightarrow \text{MATRIX}([\%E, \%E^2], [\%E^3, \%E^4])$$
 In general, this transformation affects expressions of the form  $\langle \text{base} \rangle^{\langle \text{power} \rangle}$  where  $\langle \text{base} \rangle$  is an expression assumed scalar or constant, and  $\langle \text{power} \rangle$  is a list or matrix. This transformation is turned off if this switch is set to FALSE.
- DOMXMXOPS** Variable  
 default: [TRUE] if TRUE then all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this switch is FALSE they are not.
- DOMXNCTIMES** Variable  
 default: [FALSE] Causes non-commutative products of matrices to be carried out.
- DONTFACTOR** Variable  
 default: [] 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.
- DOSCMXOPS** Variable  
 default: [FALSE] if TRUE then scalar-matrix operations are performed.
- DOSCMXPLUS** Variable  
 default: [FALSE] if TRUE will cause SCALAR + MATRIX to give a matrix answer. This switch is not subsumed under DOALLMXOPS.
- DOT0NSCSIMP** Variable  
 default: [TRUE] Causes a non-commutative product of zero and a nonscalar term to be simplified to a commutative product.
- DOT0SIMP** Variable  
 default: [TRUE] Causes a non-commutative product of zero and a scalar term to be simplified to a commutative product.
- DOT1SIMP** Variable  
 default: [TRUE] Causes a non-commutative product of one and another term to be simplified to a commutative product.
- DOTASSOC** Variable  
 default: [TRUE] when TRUE causes  $(A.B).C$  to simplify to  $A.(B.C)$
- DOTCONSTRULES** Variable  
 default: [TRUE] Causes a non-commutative product of a constant and another term to be simplified to a commutative product. Turning on this flag effectively turns on DOT0SIMP, DOT0NSCSIMP, and DOT1SIMP as well.

**DOTDISTRIB** Variable  
 default: [FALSE] if TRUE will cause A.(B+C) to simplify to A.B+A.C

**DOTEXPTSIMP** Variable  
 default: [TRUE] when TRUE causes A.A to simplify to A<sup>2</sup>

**DOTIDENT** Variable  
 default: [1] The value to be returned by X<sup>0</sup>.

**DOTSCRULES** Variable  
 default: [FALSE] when TRUE will cause A.SC or SC.A to simplify to SC\*A and A.(SC\*B) to simplify to SC\*(A.B)

**ECHELON** (*M*) Function  
 produces the echelon form of the matrix M. That is, M with elementary row operations performed on it such that the first non-zero element in each row in the resulting matrix is a one and the column elements under the first one in each row are all zero.

```
(D2)          [ 2  1 - A  -5 B ]
              [                ]
              [ A    B    C    ]

(C3) ECHELON(D2);
          [      A - 1      5 B      ]
          [ 1  - ----  - ----  ]
          [      2          2          ]
(D3)      [                ]
          [      2 C + 5 A B ]
          [ 0    1      ----- ]
          [                2      ]
          [      2 B + A  - A ]
```

**EIGENVALUES** (*mat*) Function

There is a package on the SHARE; directory which contains functions for computing EIGENVALUES and EIGENVECTORS and related matrix computations. For information on it do PRINTFILE(EIGEN,USAGE,SHARE); . EIGENVALUES(mat) takes a MATRIX as its argument and returns a list of lists the first sublist of which is the list of eigenvalues of the matrix and the other sublist of which is the list of the multiplicities of the eigenvalues in the corresponding order. [ The MACSYMA function SOLVE is used here to find the roots of the characteristic polynomial of the matrix. Sometimes SOLVE may not be able to find the roots of the polynomial; in that case nothing in this package except CONJUGATE, INNERPRODUCT, UNITVECTOR, COLUMNVECTOR and GRAMSCHMIDT will work unless you know the eigenvalues. In some cases SOLVE may generate very messy eigenvalues. You may want to simplify the answers yourself before you go on. There are provisions for this and

they will be explained below. ( This usually happens when SOLVE returns a not-so-obviously real expression for an eigenvalue which is supposed to be real...) The EIGENVALUES command is available directly from MACSYMA. To use the other functions you must have loaded in the EIGEN package, either by a previous call to EIGENVALUES, or by doing LOADFILE("eigen"); .

**EIGENVECTORS** (*MAT*)

Function

takes a MATRIX as its argument and returns a list of lists the first sublist of which is the output of the EIGENVALUES command and the other sublists of which are the eigenvectors of the matrix corresponding to those eigenvalues respectively. This function will work directly from MACSYMA, but if you wish to take advantage of the flags for controlling it (see below), you must first load in the EIGEN package from the SHARE; directory. You may do that by LOADFILE("eigen");. The flags that affect this function are: NONDIAGONALIZABLE[FALSE] will be set to TRUE or FALSE depending on whether the matrix is nondiagonalizable or diagonalizable after an EIGENVECTORS command is executed. HERMITIANMATRIX[FALSE] If set to TRUE will cause the degenerate eigenvectors of the hermitian matrix to be orthogonalized using the Gram-Schmidt algorithm. KNOWNEIGVALS[FALSE] If set to TRUE the EIGEN package will assume the eigenvalues of the matrix are known to the user and stored under the global name LISTEIGVALS. LISTEIGVALS should be set to a list similar to the output of the EIGENVALUES command. ( The MACSYMA function ALGSYS is used here to solve for the eigenvectors. Sometimes if the eigenvalues are messy, ALGSYS may not be able to produce a solution. In that case you are advised to try to simplify the eigenvalues by first finding them using EIGENVALUES command and then using whatever marvelous tricks you might have to reduce them to something simpler. You can then use the KNOWNEIGVALS flag to proceed further. )

**EMATRIX** (*m, n, x, i, j*)

Function

will create an m by n matrix all of whose elements are zero except for the i,j element which is x.

**ENTERMATRIX** (*m, n*)

Function

allows one to enter a matrix element by element with MACSYMA requesting values for each of the m\*n entries.

```
(C1) ENTERMATRIX(3,3);
Is the matrix 1. Diagonal 2. Symmetric 3. Antisymmetric
4. General
```

```
Answer 1, 2, 3 or 4
1;
Row 1 Column 1: A;
Row 2 Column 2: B;
Row 3 Column 3: C;
Matrix entered.
```

```
[ A 0 0 ]
[           ]
```



$$(D1) \quad \begin{bmatrix} 0 & B & 0 \\ & & \\ 0 & 0 & C \end{bmatrix}$$

**GENMATRIX** (*array, i2, j2, i1, j1*) Function  
 generates a matrix from the array using array(i1,j1) for the first (upper-left) element and array(i2,j2) for the last (lower-right) element of the matrix. If j1=i1 then j1 may be omitted. If j1=i1=1 then i1 and j1 may both be omitted. If a selected element of the array doesn't exist a symbolic one will be used.

(C1)  $H[I, J] := 1/(I+J-1)$   
 (C2) GENMATRIX(H, 3, 3);

$$(D2) \quad \begin{bmatrix} 1 & 1 \\ 1 & - \\ 2 & 3 \\ & \\ 1 & 1 & 1 \\ - & - & - \\ 2 & 3 & 4 \\ & \\ 1 & 1 & 1 \\ - & - & - \\ 3 & 4 & 5 \end{bmatrix}$$

**GRAMSCHMIDT** (*X*) Function  
 a function in the EIGEN package. Do LOAD(EIGEN) to use it. GRAMSCHMIDT takes a LIST of lists the sublists of which are of equal length and not necessarily orthogonal (with respect to the innerproduct defined above) as its argument and returns a similar list each sublist of which is orthogonal to all others. (Returned results may contain integers that are factored. This is due to the fact that the MACSYMA function FACTOR is used to simplify each substage of the Gram-Schmidt algorithm. This prevents the expressions from getting very messy and helps to reduce the sizes of the numbers that are produced along the way.)

**HACH** (*a, b, m, n, l*) Function  
 An implementation of Hacijan's linear programming algorithm is available by doing BATCH("kach.mc"\$. Details of use are available by doing BATCH("kach.dem");

**IDENT** (*n*) Function  
 produces an n by n identity matrix.

**INNERPRODUCT** (*X, Y*) Function  
 a function in the EIGEN package. Do LOAD(EIGEN) to use it. INNERPRODUCT takes two LISTS of equal length as its arguments and returns their inner (scalar) product defined by (Complex Conjugate of X).Y (The "dot" operation is the same as the usual one defined for vectors).

**INVERT** (*matrix*) Function

finds the inverse of a matrix using the adjoint method. This allows a user to compute the inverse of a matrix with bfloat entries or polynomials with floating pt. coefficients without converting to cre-form. The DETERMINANT command is used to compute cofactors, so if RATMX is FALSE (the default) the inverse is computed without changing the representation of the elements. The current implementation is inefficient for matrices of high order. The DETOUT flag if true keeps the determinant factored out of the inverse. Note: the results are not automatically expanded. If the matrix originally had polynomial entries, better appearing output can be generated by EXPAND(INVERT(mat)),DETOUT. If it is desirable to then divide through by the determinant this can be accomplished by XTHRU(%) or alternatively from scratch by EXPAND(ADJOINT(mat))/EXPAND(DETERMINANT(mat)). INVERT(mat):=ADJOINT(mat)/DETERMINANT(mat). See also DESCRIBE("^^"); for another method of inverting a matrix.

**LMXCHAR** Variable

default: [[] - The character used to display the (left) delimiter of a matrix (see also RMXCHAR).

**MATRIX** (*row1, ..., rown*) Function

defines a rectangular matrix with the indicated rows. Each row has the form of a list of expressions, e.g. [A, X\*\*2, Y, 0] is a list of 4 elements. There are a number of MACSYMA commands which deal with matrices, for example: DETERMINANT, CHARPOLY, GENMATRIX, ADDCOL, ADDROW, COPYMATRIX, TRANSPOSE, ECHELON, and RANK. There is also a package on the SHARE directory for computing EIGENVALUES. Try DESCRIBE on these for more information. Matrix multiplication is effected by using the dot operator, ".", which is also convenient if the user wishes to represent other non-commutative algebraic operations. The exponential of the "." operation is "^^" . Thus, for a matrix A, A.A = A^^2 and, if it exists, A^^-1 is the inverse of A. The operations +, -, \*, \*\* are all element-by-element operations; all operations are normally carried out in full, including the . (dot) operation. Many switches exist for controlling simplification rules involving dot and matrix-list operations. Options Relating to Matrices: LMXCHAR, RMXCHAR, RATMX, LISTARITH, DETOUT, DOALLMXOPS, DOMXEXPT DOMXXMOPS, DOSCMXOPS, DOSCMXPLUS, SCALARMATRIX, and SPARSE. Do DESCRIBE(option) for details on them.

**MATRIXMAP** (*fn, M*) Function

will map the function fn onto each element of the matrix M.

**MATRIXP** (*exp*) Function

is TRUE if exp is a matrix else FALSE.

**MATRIX\_ELEMENT\_ADD** Variable

default: [+] - May be set to "?"; may also be the name of a function, or a LAMBDA expression. In this way, a rich variety of algebraic structures may be simulated. For more details, do DEMO("matrix.dem1"); and DEMO("matrix.dem2");.

**MATRIX\_ELEMENT\_MULT** Variable  
 default: [\*] - May be set to "."; may also be the name of a function, or a LAMBDA expression. In this way, a rich variety of algebraic structures may be simulated. For more details, do DEMO("matrix.dem1"); and DEMO("matrix.dem2");

**MATRIX\_ELEMENT\_TRANSPOSE** Variable  
 default: [FALSE] - Other useful settings are TRANSPOSE and NONSCALARS; may also be the name of a function, or a LAMBDA expression. In this way, a rich variety of algebraic structures may be simulated. For more details, do DEMO("matrix.dem1"); and DEMO("matrix.dem2");

**MATTRACE** (*M*) Function  
 computes the trace [sum of the elements on the main diagonal] of the square matrix *M*. It is used by NCHARPOLY, an alternative to MACSYMA's CHARPOLY. It is used by doing LOADFILE("nchrpl");

**MINOR** (*M, i, j*) Function  
 computes the *i,j* minor of the matrix *M*. That is, *M* with row *i* and column *j* removed.

**NCEXPT** (*A,B*) Function  
 if an (non-commutative) exponential expression is too wide to be displayed as  $A^B$  it will appear as NCEXPT(*A,B*).

**NCHARPOLY** (*M,var*) Function  
 finds the characteristic polynomial of the matrix *M* with respect to *var*. This is an alternative to MACSYMA's CHARPOLY. NCHARPOLY works by computing traces of powers of the given matrix, which are known to be equal to sums of powers of the roots of the characteristic polynomial. From these quantities the symmetric functions of the roots can be calculated, which are nothing more than the coefficients of the characteristic polynomial. CHARPOLY works by forming the determinant of  $VAR * IDENT [N] - A$ . Thus NCHARPOLY wins, for example, in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether. It may be used by doing LOADFILE("nchrpl");

**NEWDET** (*M,n*) Function  
 also computes the determinant of *M* but uses the Johnson-Gentleman tree minor algorithm. *M* may be the name of a matrix or array. The argument *n* is the order; it is optional if *M* is a matrix.

**NONSCALAR** declaration  
 - makes *ai* behave as does a list or matrix with respect to the dot operator.

**NONSCALARP** (*exp*) Function  
 is TRUE if *exp* is a non-scalar, i.e. it contains atoms declared as non-scalars, lists, or matrices.

- PERMANENT** ( $M, n$ ) Function  
 computes the permanent of the matrix  $M$ . A permanent is like a determinant but with no sign changes.
- RANK** ( $M$ ) Function  
 computes the rank of the matrix  $M$ . That is, the order of the largest non-singular subdeterminant of  $M$ . Caveat: RANK may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.
- RATMX** Variable  
 default: [FALSE] - if FALSE will cause determinant and matrix addition, subtraction, and multiplication to be performed in the representation of the matrix elements and will cause the result of matrix inversion to be left in general representation. If it is TRUE, the 4 operations mentioned above will be performed in CRE form and the result of matrix inverse will be in CRE form. Note that this may cause the elements to be expanded (depending on the setting of RATFAC) which might not always be desired.
- ROW** ( $M, i$ ) Function  
 gives a matrix of the  $i$ th row of matrix  $M$ .
- SCALARMATRIXP** Variable  
 default: [TRUE] - if TRUE, then whenever a 1 x 1 matrix is produced as a result of computing the dot product of matrices it will be converted to a scalar, namely the only element of the matrix. If set to ALL, then this conversion occurs whenever a 1 x 1 matrix is simplified. If set to FALSE, no conversion will be done.
- SETELMX** ( $x, i, j, M$ ) Function  
 changes the  $i, j$  element of  $M$  to  $x$ . The altered matrix is returned as the value. The notation  $M[i, j]:x$  may also be used, altering  $M$  in a similar manner, but returning  $x$  as the value.
- SIMILARITYTRANSFORM** ( $MAT$ ) Function  
 a function in the EIGEN package. Do LOAD(EIGEN) to use it. SIMILARITYTRANSFORM takes a MATRIX as its argument and returns a list which is the output of the UNITEEIGENVECTORS command. In addition if the flag NONDIAGONALIZABLE is FALSE two global matrices LEFTMATRIX and RIGHTMATRIX will be generated. These matrices have the property that LEFTMATRIX.MAT.RIGHTMATRIX is a diagonal matrix with the eigenvalues of  $MAT$  on the diagonal. If NONDIAGONALIZABLE is TRUE these two matrices will not be generated. If the flag HERMITIANMATRIX is TRUE then LEFTMATRIX is the complex conjugate of the transpose of RIGHTMATRIX. Otherwise LEFTMATRIX is the inverse of RIGHTMATRIX. RIGHTMATRIX is the matrix the columns of which are the unit eigenvectors of  $MAT$ . The other flags (see DESCRIBE(EIGENVALUES); and DESCRIBE(EIGENVECTORS);) have the same effects since SIMILARITYTRANSFORM calls the other functions in the package in order to be able to form RIGHTMATRIX.

- SPARSE** Variable  
 default: [FALSE] - if TRUE and if RATMX:TRUE then DETERMINANT will use special routines for computing sparse determinants.
- SUBMATRIX** ( $m1, \dots, M, n1, \dots$ ) Function  
 creates a new matrix composed of the matrix M with rows  $m_i$  deleted, and columns  $n_i$  deleted.
- TRANSPOSE** ( $M$ ) Function  
 produces the transpose of the matrix M.
- TRIANGULARIZE** ( $M$ ) Function  
 produces the upper triangular form of the matrix M which needn't be square.
- UNITEIGENVECTORS** ( $MAT$ ) Function  
 a function in the EIGEN package. Do LOAD(EIGEN) to use it. UNITEIGENVECTORS takes a MATRIX as its argument and returns a list of lists the first sublist of which is the output of the EIGENVALUES command and the other sublists of which are the unit eigenvectors of the matrix corresponding to those eigenvalues respectively. The flags mentioned in the description of the EIGENVECTORS command have the same effects in this one as well. In addition there is a flag which may be useful : KNOWNEIGVECTS[FALSE] - If set to TRUE the EIGEN package will assume that the eigenvectors of the matrix are known to the user and are stored under the global name LISTEIGVECTS. LISTEIGVECTS should be set to a list similar to the output of the EIGENVECTORS command. (If KNOWNEIGVECTS is set to TRUE and the list of eigenvectors is given the setting of the flag NONDIAGONALIZABLE may not be correct. If that is the case please set it to the correct value. The author assumes that the user knows what he is doing and will not try to diagonalize a matrix the eigenvectors of which do not span the vector space of the appropriate dimension...)
- UNITVECTOR** ( $X$ ) Function  
 a function in the EIGEN package. Do LOAD(EIGEN) to use it. UNITVECTOR takes a LIST as its argument and returns a unit list. (i.e. a list with unit magnitude).
- VECTORSIMP** (*vectorexpression*) Function  
 This function employs additional simplifications, together with various optional expansions according to the settings of the following global flags:  
 EXPANDALL, EXPANDDOT, EXPANDDOTPLUS, EXPANDCROSS, EXPANDCROSSPLUS, EXPANDCROSSCROSS, EXPANDGRAD, EXPANDGRADPLUS, EXPANDGRADPROD, EXPANDDIV, EXPANDDIVPLUS, EXPANDDIVPROD, EXPANDCURL, EXPANDCURLPLUS, EXPANDCURLCURL, EXPANDLAPLACIAN, EXPANDLAPLACIANPLUS, EXPANDLAPLACIANPROD.
- All these flags have default value FALSE. The PLUS suffix refers to employing additivity or distributivity. The PROD suffix refers to the expansion for an operand that is any kind of product. EXPANDCROSSCROSS refers to replacing  $\tilde{p}(q\tilde{r})$  with  $(p.r)*q-(p.q)*r$ , and EXPANDCURLCURL refers to replacing CURL CURL

$p$  with  $\text{GRAD DIV } p + \text{DIV GRAD } p$ .  $\text{EXPANDCROSS:TRUE}$  has the same effect as  $\text{EXPANDCROSSPLUS:EXPANDCROSSCROSS:TRUE}$ , etc. Two other flags,  $\text{EXPANDPLUS}$  and  $\text{EXPANDPROD}$ , have the same effect as setting all similarly suffixed flags true. When  $\text{TRUE}$ , another flag named  $\text{EXPANDLAPLACIANTODIVGRAD}$ , replaces the  $\text{LAPLACIAN}$  operator with the composition  $\text{DIV GRAD}$ . All of these flags are initially  $\text{FALSE}$ . For convenience, all of these flags have been declared  $\text{EVFLAG}$ . For orthogonal curvilinear coordinates, the global variables  $\text{COORDINATES}[[X,Y,Z]]$ ,  $\text{DIMENSION}[3]$ ,  $\text{SF}[[1,1,1]]$ , and  $\text{SFPROD}[1]$  are set by the function invocation

<b>VECT_CROSS</b>	Variable
default:[FALSE] - If $\text{TRUE}$ allows $\text{DIFF}(X\sim Y,T)$ to work where $\sim$ is defined in $\text{SHARE;VECT}$ (where $\text{VECT\_CROSS}$ is set to $\text{TRUE}$ , anyway.)	
<b>ZEROMATRIX</b> ( $m,n$ )	Function
takes integers $m,n$ as arguments and returns an $m$ by $n$ matrix of 0's.	
"["	special symbol
- [ and ] are the characters which $\text{MACSYMA}$ uses to delimit a list.	



## 26 Affine

### 26.1 Definitions for Affine

**FAST\_LINSOLVE** (*eqns,variables*) Function  
 Solves the linear system of equations EQNS for the variables VARIABLES and returns a result suitable to SUBLIS. The function is faster than linsolve for system of equations which are sparse.

**GROBNER\_BASIS** (*eqns*) Function  
 Takes as argument a macsyma list of equations and returns a grobner basis for them. The function POLYSIMP may now be used to simplify other functions relative to the equations.  
 GROBNER\_BASIS([3\*X^2+1,Y\*X])\$  
 POLYSIMP(Y^2\*X+X^3\*9+2)==> -3\*x+2  
 Polysimp(f)==> 0 if and only if f is in the ideal generated by the EQNS ie. if and only if f is a polynomial combination of the elements of EQNS.

**SET\_UP\_DOT\_SIMPLIFICATIONS** (*eqns,[check-thru-degree]*) Function  
 The eqns are polynomial equations in non commutative variables. The value of CURRENT\_VARIABLES is the list of variables used for computing degrees. The equations must be homogeneous, in order for the procedure to terminate.  
 If you have checked overlapping simplifications in DOT\_SIMPLIFICATIONS above the degree of f, then the following is true: DOTSIMP(f)==> 0 if and only if f is in the ideal generated by the EQNS ie. if and only if f is a polynomial combination of the elements of EQNS.  
 The degree is that returned by NC\_DEGREE. This in turn is influenced by the weights of individual variables.

**DECLARE\_WEIGHT** (*var1,wt1,var2,wt2,...*) Function  
 Assigns VAR1 weight WT1, VAR2 weight wt2.. These are the weights used in computing NC\_DEGREE.

**NC\_DEGREE** (*poly*) Function  
 Degree of a non commutative polynomial. See DECLARE\_WEIGHTS.

**DOTSIMP** (*f*) Function  
 ==> 0 if and only if f is in the ideal generated by the EQNS ie. if and only if f is a polynomial combination of the elements of EQNS.

**FAST\_CENTRAL\_ELEMENTS** (*variables,degree*) Function  
 if SET\_UP\_DOT\_SIMPLIFICATIONS has been previously done, finds the central polynomials in the variables in the given degree, For example:



```

set_up_dot_simplifications([y.x+x.y],3);
fast_central_elements([x,y],2);
[y.y,x.x];

```

**CHECK\_OVERLAPS** (*degree,add-to-simps*) Function

checks the overlaps thru degree, making sure that you have sufficient simplification rules in each degree, for dotsimp to work correctly. This process can be speeded up if you know before hand what the dimension of the space of monomials is. If it is of finite global dimension, then HILBERT should be used. If you don't know the monomial dimensions, do not specify a RANK\_FUNCTION. An optional third argument RESET, false says don't bother to query about resetting things.

**MONO** (*vari,n*) Function

VARI is a list of variables. Returns the list of independent monomials relative to the current dot\_simplifications, in degree N

**MONOMIAL\_DIMENSIONS** (*n*) Function

Compute the hilbert series through degree n for the current algebra.

**EXTRACT\_LINEAR\_EQUATIONS** (*List\_nc\_polys,monoms*) Function

Makes a list of the coefficients of the polynomials in list\_nc\_polys of the monoms. MONOMS is a list of noncommutative monomials. The coefficients should be scalars. Use LIST\_NC\_MONOMIALS to build the list of monoms.

**LIST\_NC\_MONOMIALS** (*polys-or-list*) Function

returns a list of the non commutative monomials occurring in a polynomial or a collection of polynomials.

**PCOEFF** (*poly monom [variables-to-exclude-from-cof (list-variables monom)]*) Function

This function is called from lisp level, and uses internal poly format.

```

CL-MAXIMA>>(setq me (st-rat #x^2*u+y+1$))
(#:Y 1 1 0 (#:X 2 (#:U 1 1) 0 1))

```

```

CL-MAXIMA>>(pcoeff me (st-rat #x^2$))
(#:U 1 1)

```

Rule: if a variable appears in monom it must be to the exact power, and if it is in variables to exclude it may not appear unless it was in monom to the exact power. (pcoeff pol 1 ..) will exclude variables like substituting them to be zero.

**NEW-DISREP** (*poly*) Function

From lisp this returns the general maxima format for an arg which is in st-rat form:

```

(displa(new-disrep (setq me (st-rat #x^2*u+y+1$))))

```

```

      2
Y + U X + 1

```

**CREATE\_LIST** (*form,var1,list1,var2,list2,...*) Function

Create a list by evaluating FORM with VAR1 bound to each element of LIST1, and for each such binding bind VAR2 to each element of LIST2,... The number of elements in the result will be  $\text{length}(\text{list1}) \times \text{length}(\text{list2}) \times \dots$ . Each VARn must actually be a symbol—it will not be evaluated. The LISTn args will be evaluated once at the beginning of the iteration.

```
(C82) create_list1(x^i,i,[1,3,7]);
(D82) [X,X^3,X^7]
```

With a double iteration:

```
(C79) create_list([i,j],i,[a,b],j,[e,f,h]);
(D79) [[A,E],[A,F],[A,H],[B,E],[B,F],[B,H]]
```

Instead of LISTn two args maybe supplied each of which should evaluate to a number. These will be the inclusive lower and upper bounds for the iteration.

```
(C81) create_list([i,j],i,[1,2,3],j,1,i);
(D81) [[1,1],[2,1],[2,2],[3,1],[3,2],[3,3]]
```

Note that the limits or list for the j variable can depend on the current value of i.

**ALL\_DOTSIMP\_DENOMS** Variable

if its value is FALSE the denominators encountered in getting dotsimps will not be collected. To collect the denoms

```
ALL_DOTSIMP_DENOMS: [];
```

and they will be nconc'd onto the end of the list.



## 27 Tensor

### 27.1 Introduction to Tensor

- Indicial Tensor Manipulation package. It may be loaded by `LOADFILE("itensr");` A manual for the Tensor packages is available in `share/tensor.descr`. A demo is available by `DEMO("itenso.dem1");` (and additional demos are in `("itenso.dem2")`, `("itenso.dem3")` and following).

- There are two tensor packages in MACSYMA, CTENSR and ITENSR. CTENSR is Component Tensor Manipulation, and may be accessed with `LOAD(CTENSR);` . ITENSR is Indicial Tensor Manipulation, and is loaded by doing `LOAD(ITENSR);` A manual for CTENSR AND ITENSR is available from the LCS Publications Office. Request MIT/LCS/TM-167. In addition, demos exist on the TENSOR; directory under the filenames CTENSO DEMO1, DEMO2, etc. and ITENSO DEMO1, DEMO2, etc. Do `DEMO("ctenso.dem1");` or `DEMO("itenso.dem2");` Send bugs or comments to RP or TENSOR.

### 27.2 Definitions for Tensor

**CANFORM** (*exp*) Function

[Tensor Package] Simplifies *exp* by renaming dummy indices and reordering all indices as dictated by symmetry conditions imposed on them. If ALLSYM is TRUE then all indices are assumed symmetric, otherwise symmetry information provided by DECSYM declarations will be used. The dummy indices are renamed in the same manner as in the RENAME function. When CANFORM is applied to a large expression the calculation may take a considerable amount of time. This time can be shortened by calling RENAME on the expression first. Also see the example under DECSYM. Note: CANFORM may not be able to reduce an expression completely to its simplest form although it will always return a mathematically correct result.

**CANTEN** (*exp*) Function

[Tensor Package] Simplifies *exp* by renaming (see RENAME) and permuting dummy indices. CANTEN is restricted to sums of tensor products in which no derivatives are present. As such it is limited and should only be used if CANFORM is not capable of carrying out the required simplification.

**CARG** (*exp*) Function

returns the argument (phase angle) of *exp*. Due to the conventions and restrictions, principal value cannot be guaranteed unless *exp* is numeric.

**COUNTER** Variable

default: [1] determines the numerical suffix to be used in generating the next dummy index in the tensor package. The prefix is determined by the option DUMMYX[#].

**DEFCON** (*tensor1,<tensor2,tensor3>*) Function

gives tensor1 the property that the contraction of a product of tensor1 and tensor2 results in tensor3 with the appropriate indices. If only one argument, tensor1, is given, then the contraction of the product of tensor1 with any indexed object having the appropriate indices (say tensor) will yield an indexed object with that name, i.e.tensor, and with a new set of indices reflecting the contractions performed. For example, if METRIC: G, then DEFCON(G) will implement the raising and lowering of indices through contraction with the metric tensor. More than one DEFCON can be given for the same indexed object; the latest one given which applies in a particular contraction will be used. CONTRACTIONS is a list of those indexed objects which have been given contraction properties with DEFCON.

**FLUSH** (*exp,tensor1,tensor2,...*) Function

Tensor Package - will set to zero, in exp, all occurrences of the tensori that have no derivative indices.

**FLUSHD** (*exp,tensor1,tensor2,...*) Function

Tensor Package - will set to zero, in exp, all occurrences of the tensori that have derivative indices.

**FLUSHND** (*exp,tensor,n*) Function

Tensor Package - will set to zero, in exp, all occurrences of the differentiated object tensor that have n or more derivative indices as the following example demonstrates.

(C1) SHOW(A([I],[J,R],K,R)+A([I],[J,R,S],K,R,S));

(D1) 
$$\begin{array}{ccc} & J & R & S & & J & R \\ A & & & & + & A & \\ & I, & K & R & S & & I, & K & R \end{array}$$

(C2) SHOW(FLUSHND(D1,A,3));

(D2) 
$$\begin{array}{ccc} & & J & R \\ A & & & \\ & & I, & K & R \end{array}$$

**KDELTA** (*L1,L2*) Function

is the generalized Kronecker delta function defined in the Tensor package with L1 the list of covariant indices and L2 the list of contravariant indices. KDELTA([i],[j]) returns the ordinary Kronecker delta. The command EV(EXP,KDELTA) causes the evaluation of an expression containing KDELTA([],[]) to the dimension of the manifold.

**LC** (*L*) Function

is the permutation (or Levi-Civita) tensor which yields 1 if the list L consists of an even permutation of integers, -1 if it consists of an odd permutation, and 0 if some indices in L are repeated.

- LORENTZ** (*exp*) Function  
 imposes the Lorentz condition by substituting 0 for all indexed objects in *exp* that have a derivative index identical to a contravariant index.
- MAKEBOX** (*exp*) Function  
 will display *exp* in the same manner as **SHOW**; however, any tensor d'Alembertian occurring in *exp* will be indicated using the symbol  $\square$ . For example,  $\square P([M],[N])$  represents  $G(\square,[I,J])*P([M],[N],[I,J])$ .
- METRIC** (*G*) Function  
 specifies the metric by assigning the variable **METRIC:G**; in addition, the contraction properties of the metric **G** are set up by executing the commands **DEFCON(G)**, **DEFCON(G,G,KDELTA)**. The variable **METRIC**, default:  $\square$ , is bound to the metric, assigned by the **METRIC(g)** command.
- NTERMSG** () Function  
 gives the user a quick picture of the "size" of the Einstein tensor. It returns a list of pairs whose second elements give the number of terms in the components specified by the first elements.
- NTERMSRCI** () Function  
 returns a list of pairs, whose second elements give the number of terms in the **RICCI** component specified by the first elements. In this way, it is possible to quickly find the non-zero expressions and attempt simplification.
- NZETA** (*Z*) Function  
 returns the complex value of the Plasma Dispersion Function for complex *Z*.  

$$\text{NZETAR}(Z) ==> \text{REALPART}(\text{NZETA}(Z))$$

$$\text{NZETAI}(Z) \text{ returns } \text{IMAGPART}(\text{NZETA}(Z)).$$
 This function is related to the complex error function by  

$$\text{NZETA}(Z) = \%I*\text{SQRT}(\%PI)*\text{EXP}(-Z^2)*(1-\text{ERF}(-\%I*Z)).$$
- RAISERIEMANN** (*dis*) Function  
 returns the contravariant components of the Riemann curvature tensor as array elements **UR[I,J,K,L]**. These are displayed if *dis* is **TRUE**.
- RATEINSTEIN** Variable  
 default:  $\square$  - if **TRUE** rational simplification will be performed on the non-zero components of Einstein tensors; if **FACRAT:TRUE** then the components will also be factored.
- RATRIEMAN** Variable  
 - This switch has been renamed **RATRIEMANN**.

- RATRIEMANN** Variable  
 default: [] - one of the switches which controls simplification of Riemann tensors; if TRUE, then rational simplification will be done; if FACRAT:TRUE then each of the components will also be factored.
- REMCON** (*tensor1, tensor2, ...*) Function  
 removes all the contraction properties from the tensori. REMCON(ALL) removes all contraction properties from all indexed objects.
- RICCICOM** (*dis*) Function  
 Tensor package) This function first computes the covariant components LR[i,j] of the Ricci tensor (LR is a mnemonic for "lower Ricci"). Then the mixed Ricci tensor is computed using the contravariant metric tensor. If the value of the argument to RICCICOM is TRUE, then these mixed components, RICCI[i,j] (the index i is covariant and the index j is contravariant), will be displayed directly. Otherwise, RICCICOM(FALSE) will simply compute the entries of the array RICCI[i,j] without displaying the results.
- RINVARIANT** () Function  
 Tensor package) forms the invariant obtained by contracting the tensors  

$$R[i, j, k, l] * UR[i, j, k, l].$$
 This object is not automatically simplified since it can be very large.
- SCURVATURE** () Function  
 returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.
- SETUP** () Function  
 this has been renamed to TSETUP(); Sets up a metric for Tensor calculations.
- WEYL** (*dis*) Function  
 computes the Weyl conformal tensor. If the argument dis is TRUE, the non-zero components W[I,J,K,L] will be displayed to the user. Otherwise, these components will simply be computed and stored. If the switch RATWEYL is set to TRUE, then the components will be rationally simplified; if FACRAT is TRUE then the results will be factored as well.

## 28 Ctensor

### 28.1 Introduction to Ctensor

- Component Tensor Manipulation Package. To use the CTENSR package, type TSETUP(); which automatically loads it from within MACSYMA (if it is not already loaded) and then prompts the user to input his coordinate system. The user is first asked to specify the dimension of the manifold. If the dimension is 2, 3 or 4 then the list of coordinates defaults to [X,Y], [X,Y,Z] or [X,Y,Z,T] respectively. These names may be changed by assigning a new list of coordinates to the variable OMEGA (described below) and the user is queried about this. \*\* Care must be taken to avoid the coordinate names conflicting with other object definitions \*\*. Next, the user enters the metric either directly or from a file by specifying its ordinal position. As an example of a file of common metrics, see TENSOR;METRIC FILE. The metric is stored in the matrix LG. Finally, the metric inverse is computed and stored in the matrix UG. One has the option of carrying out all calculations in a power series. A sample protocol is begun below for the static, spherically symmetric metric (standard coordinates) which will be applied to the problem of deriving Einstein's vacuum equations (which lead to the Schwarzschild solution) as an example. Many of the functions in CTENSR will be displayed for the standard metric as examples.

```
(C2) TSETUP();
```

```
Enter the dimension of the coordinate system:
4;
Do you wish to change the coordinate names?
N;
Do you want to
1. Enter a new metric?
2. Enter a metric from a file?
3. Approximate a metric with a Taylor series?
Enter 1, 2 or 3
1;
Is the matrix 1. Diagonal 2. Symmetric 3. Antisymmetric 4. General
Answer 1, 2, 3 or 4
1;
Row 1 Column 1: A;
Row 2 Column 2: X^2;
Row 3 Column 3: X^2*SIN(Y)^2;
Row 4 Column 4: -D;
Matrix entered.
Enter functional dependencies with the DEPENDS function or 'N' if none
DEPENDS([A,D],X);
Do you wish to see the metric?
Y;
      [ A  0      0      0 ]
      [
      [   2
      [ 0  X      0      0 ]
```





$$\begin{aligned}
 \text{CURVATURE} &= - \text{CHR2} & - \text{CHR2} & \text{CHR2} & + \text{CHR2} \\
 & \begin{matrix} h \\ i \ j \ k \end{matrix} & \begin{matrix} h \\ i \ k, j \\ h \\ \%1 \ k \end{matrix} & \begin{matrix} h \\ \%1 \ j \\ \%1 \\ i \ j \end{matrix} & \begin{matrix} \%1 \\ i \ k \\ i \ j, k \end{matrix}
 \end{aligned}$$

**DIAGMETRIC**

Variable

default:[] - An option in the CTENSR (Component Tensor Manipulation) package. If DIAGMETRIC is TRUE special routines compute all geometrical objects (which contain the metric tensor explicitly) by taking into consideration the diagonality of the metric. Reduced run times will, of course, result. Note: this option is set automatically by TSETUP if a diagonal metric is specified.

**DIM**

Variable

default:[4] - An option in the CTENSR (Component Tensor Manipulation) package. DIM is the dimension of the manifold with the default 4. The command DIM:N; will reset the dimension to any other integral value.

**EINSTEIN** (*dis*)

Function

A function in the CTENSR (Component Tensor Manipulation) package. EINSTEIN computes the mixed Einstein tensor after the Christoffel symbols and Ricci tensor have been obtained (with the functions CHRISTOF and RICCIOM). If the argument *dis* is TRUE, then the non-zero values of the mixed Einstein tensor  $G[i,j]$  will be displayed where  $j$  is the contravariant index. RATEINSTEIN[TRUE] if TRUE will cause the rational simplification on these components. If RATEFAC[FALSE] is TRUE then the components will also be factored.

**LRICCIOM** (*dis*)

Function

A function in the CTENSR (Component Tensor Manipulation) package. LRICCIOM computes the covariant (symmetric) components  $LR[i,j]$  of the Ricci tensor. If the argument *dis* is TRUE, then the non-zero components are displayed.

**MOTION** (*dis*)

Function

A function in the CTENSR (Component Tensor Manipulation) package. MOTION computes the geodesic equations of motion for a given metric. They are stored in the array EM[i]. If the argument *dis* is TRUE then these equations are displayed.

**OMEGA**

Variable

default:[] - An option in the CTENSR (Component Tensor Manipulation) package. OMEGA assigns a list of coordinates to the variable. While normally defined when the function TSETUP is called, one may redefine the coordinates with the assignment OMEGA:[j1,j2,...jn] where the  $j$ 's are the new coordinate names. A call to OMEGA will return the coordinate name list. Also see DESCRIBE(TSETUP); .

**RIEMANN** (*dis*) Function

A function in the CTENSR (Component Tensor Manipulation) Package. RIEMANN computes the Riemann curvature tensor from the given metric and the corresponding Christoffel symbols. If *dis* is TRUE, the non-zero components  $R[i,j,k,l]$  will be displayed. All the indicated indices are covariant. As with the Einstein tensor, various switches set by the user control the simplification of the components of the Riemann tensor. If RATTRIEMAN[TRUE] is TRUE then rational simplification will be done. If RATFAC[FALSE] is TRUE then each of the components will also be factored.

**TRANSFORM** Function

- The TRANSFORM command in the CTENSR package has been renamed to TTRANSFORM.

**TSETUP** () Function

A function in the CTENSR (Component Tensor Manipulation) package which automatically loads the CTENSR package from within MACSYMA (if it is not already loaded) and then prompts the user to make use of it. Do DESCRIBE(CTENSR); for more details.

**TTRANSFORM** (*matrix*) Function

A function in the CTENSR (Component Tensor Manipulation) package which will perform a coordinate transformation upon an arbitrary square symmetric matrix. The user must input the functions which define the transformation. (Formerly called TRANSFORM.)

## 29 Series

### 29.1 Introduction to Series

Maxima contains functions `Taylor` and `PowerSeries` for finding the series of differentiable functions. It also has tools such as `Nusum` capable of finding the closed form of some series. Operations such as addition and multiplication work as usual on series. This section presents the various global variables which control the expansion.

### 29.2 Definitions for Series

#### CAUCHYSUM

Variable

default: [FALSE] - When multiplying together sums with INF as their upper limit, if SUMEXPAND is TRUE and CAUCHYSUM is set to TRUE then the Cauchy product will be used rather than the usual product. In the Cauchy product the index of the inner summation is a function of the index of the outer one rather than varying independently. That is:  $\text{SUM}(F(I), I, 0, \text{INF}) * \text{SUM}(G(J), J, 0, \text{INF})$  becomes  $\text{SUM}(\text{SUM}(F(I) * G(J-I), I, 0, J), J, 0, \text{INF})$

#### DEFTAYLOR (*function, exp*)

Function

allows the user to define the Taylor series (about 0) of an arbitrary function of one variable as *exp* which may be a polynomial in that variable or which may be given implicitly as a power series using the SUM function. In order to display the information given to DEFTAYLOR one can use POWERSERIES(F(X),X,0). (see below).

```
(C1) DEFTAYLOR(F(X), X**2 + SUM(X**I / (2**I * I!**2),
      I, 4, INF));
```

```
(D1) [F]
(C2) TAYLOR(%E**SQRT(F(X)), X, 0, 4);
```

```
(D2) /R/      2      3      4
           X    3073 X    12817 X
           2    18432    307200
1 + X + --- + ----- + ----- + . . .
```

#### MAXTAYORDER

Variable

default: [TRUE] - if TRUE, then during algebraic manipulation of (truncated) Taylor series, TAYLOR will try to retain as many terms as are certain to be correct.

#### NICEINDICES (*expr*)

Function

will take the expression and change all the indices of sums and products to something easily understandable. It makes each index it can "I", unless "I" is in the internal expression, in which case it sequentially tries J, K, L, M, N, I0, I1, I2, I3, I4, ... until it finds a legal index.

**NICEINDICESPREF**

Variable

default: [I,J,K,L,M,N] - the list which NICEINDICES uses to find indices for sums and products. This allows the user to set the order of preference of how NICEINDICES finds the "nice indices". E.g. NICEINDICESPREF:[Q,R,S,T,INDEX]\$. Then if NICEINDICES finds that it cannot use any of these as indices in a particular summation, it uses the first as a base to try and tack on numbers. Here, if the list is exhausted, Q0, then Q1, etc, will be tried.

**NUSUM** (*exp,var,low,high*)

Function

performs indefinite summation of *exp* with respect to *var* using a decision procedure due to R.W. Gosper. *exp* and the potential answer must be expressible as products of *n*th powers, factorials, binomials, and rational functions. The terms "definite" and "indefinite summation" are used analogously to "definite" and "indefinite integration". To sum indefinitely means to give a closed form for the sum over intervals of variable length, not just e.g. 0 to inf. Thus, since there is no formula for the general partial sum of the binomial series, NUSUM can't do it.

**PADE** (*taylor-series,num-deg-bound,denom-deg-bound*)

Function

returns a list of all rational functions which have the given taylor-series expansion where the sum of the degrees of the numerator and the denominator is less than or equal to the truncation level of the power series, i.e. are "best" approximants, and which additionally satisfy the specified degree bounds. Its first argument must be a univariate taylor-series; the second and third are positive integers specifying degree bounds on the numerator and denominator. PADE's first argument can also be a Laurent series, and the degree bounds can be INF which causes all rational functions whose total degree is less than or equal to the length of the power series to be returned. Total degree is num-degree + denom-degree. Length of a power series is "truncation level" + 1 - minimum(0,"order of series").

```
(C15) ff:taylor(1+x+x^2+x^3,x,0,3);
```

```
(D15)/T/
1 + X + X2 + X3 + . . .
```

```
(C16) pade(ff,1,1);
```

```
(D16) [- -----]
          1
          X - 1
```

```
(c1) ff:taylor(-(83787*X^10-45552*X^9-187296*X^8
+387072*X^7+86016*X^6-1507328*X^5
+1966080*X^4+4194304*X^3-25165824*X^2
+67108864*X-134217728)
/134217728,x,0,10);
```

```
(C25) PADE(ff,4,4);
```

```
(D25) []
```

There is no rational function of degree 4 numerator/denominator, with this power series expansion. You must in general have degree of the numerator and degree of the denominator adding up to at least the degree of the power series, in order to have enough unknown coefficients to solve.

```
(C26) PADE(ff,5,5);
```

$$(D26) \frac{[-(520256329X^5 - 96719020632X^4 - 489651410240X^3 - 1619100813312X^2 - 2176885157888X - 2386516803584)]}{[(47041365435X^5 + 381702613848X^4 + 1360678489152X^3 + 2856700692480X^2 + 3370143559680X + 2386516803584)]}$$

**POWERDISP**

Variable

default: [FALSE] - if TRUE will cause sums to be displayed with their terms in the reverse order. Thus polynomials would display as truncated power series, i.e., with the lowest power first.

**POWERSERIES** (*exp, var, pt*)

Function

generates the general form of the power series expansion for *exp* in the variable *var* about the point *pt* (which may be INF for infinity). If POWERSERIES is unable to expand *exp*, the TAYLOR function may give the first several terms of the series. VERBOSE[FALSE] - if TRUE will cause comments about the progress of POWERSERIES to be printed as the execution of it proceeds.

```
(C1) VERBOSE:TRUE$
(C2) POWERSERIES(LOG(SIN(X)/X),X,0);
Can't expand
```

$$\text{LOG(SIN(X))}$$

So we'll try again after applying the rule:

$$\text{LOG(SIN(X))} = \int \frac{\frac{d}{dx} (\text{SIN(X)})}{\text{SIN(X)}} dx$$

In the first simplification we have returned:

$$\frac{\int \text{COT(X)} dx - \text{LOG(X)}}{\int_1^{\text{INF}} \frac{(-1)^{I1} \sqrt[2]{\text{BERN}(2 I1) X}}{I1 (2 I1)!} dx}$$

(D2)

**PSEXPAND**

Variable

default: [FALSE] - if TRUE will cause extended rational function expressions to display fully expanded. (RATEXPAND will also cause this.) If FALSE, multivariate expressions will be displayed just as in the rational function package. If PSEXPAND:MULTI, then terms with the same total degree in the variables are grouped together.

**REVERT** (*expression, variable*)

Function

Does reversion of Taylor Series. "Variable" is the variable the original Taylor expansion is in. Do LOAD(REVERT) to access this function. Try

```
REVERT2(expression, variable, hipower)
```

also. REVERT only works on expansions around 0.

**SRRAT** (*exp*)

Function

this command has been renamed to TAYTORAT.

**TAYLOR** (*exp, var, pt, pow*)

Function

expands the expression *exp* in a truncated Taylor series (or Laurent series, if required) in the variable *var* around the point *pt*. The terms through  $(\text{var}-\text{pt})^{\text{pow}}$  are generated. If *exp* is of the form  $f(\text{var})/g(\text{var})$  and  $g(\text{var})$  has no terms up to degree *pow* then TAYLOR will try to expand  $g(\text{var})$  up to degree  $2*\text{pow}$ . If there are still no non-zero terms TAYLOR will keep doubling the degree of the expansion of  $g(\text{var})$  until reaching  $\text{pow}*2^{**}n$  where *n* is the value of the variable TAYLORDEPTH[3]. If MAXTAYORDER[FALSE] is set to TRUE, then during algebraic manipulation of (truncated) Taylor series, TAYLOR will try to retain as many terms as are certain to be correct. Do EXAMPLE(TAYLOR); for examples. TAYLOR(*exp*, [*var1*, *pt1*, *ord1*], [*var2*, *pt2*, *ord2*], ...) returns a truncated power series in the variables *vari* about the points *pti*, truncated at *ordi*. PSEXPAND[FALSE] if TRUE will cause extended rational function expressions to display fully expanded. (RATEXPAND will also cause this.) If FALSE, multivariate expressions will be displayed just as in the rational function package. If PSEXPAND:MULTI, then terms with the same total degree in the variables are grouped together. TAYLOR(*exp*, [*var1*, *var2*, . . .], *pt*, *ord*) where each of *pt* and *ord* may be replaced by a list which will correspond to the list of variables. that is, the *n*th items on each of the lists will be associated together. TAYLOR(*exp*, [*x*, *pt*, *ord*, ASYMP]) will give an expansion of *exp* in negative powers of  $(x-\text{pt})$ . The highest order term will be  $(x-\text{pt})^{(-\text{ord})}$ . The ASYMP is a syntactic device and not to be assigned to. See also the TAYLOR\_LOGEXPAND switch for controlling expansion.

**TAYLORDEPTH**

Variable

default: [3] - If there are still no non-zero terms TAYLOR will keep doubling the degree of the expansion of  $g(\text{var})$  until reaching  $\text{pow}*2^{**}n$  where *n* is the value of the variable TAYLORDEPTH[3].

**TAYLORINFO** (*exp*)

Function

returns FALSE if *exp* is not a Taylor series. Otherwise, a list of lists is returned describing the particulars of the Taylor expansion. For example,

```

(C3) TAYLOR((1-Y^2)/(1-X),X,0,3,[Y,A,INF]);
      2          2
(D3)/R/ 1 - A  - 2 A (Y - A) - (Y - A)
      2          2          2
      + (1 - A  - 2 A (Y - A) - (Y - A) ) X
      2          2          2
+ (1 - A  - 2 A (Y - A) - (Y - A) ) X
      2          2          2
      + (1 - A  - 2 A (Y - A) - (Y - A) ) X
      + . . .
(C4) TAYLORINFO(D3);
(D4) [[Y, A, INF], [X, 0, 3]]

```

**TAYLORP** (*exp*) Function  
 a predicate function which returns TRUE if and only if the expression 'exp' is in Taylor series representation.

**TAYLOR\_LOGEXPAND** Variable  
 default: [TRUE] controls expansions of logarithms in TAYLOR series. When TRUE all log's are expanded fully so that zero-recognition problems involving logarithmic identities do not disturb the expansion process. However, this scheme is not always mathematically correct since it ignores branch information. If TAYLOR\_LOGEXPAND is set to FALSE, then the only expansion of log's that will occur is that necessary to obtain a formal power series.

**TAYLOR\_ORDER\_COEFFICIENTS** Variable  
 default: [TRUE] controls the ordering of coefficients in the expression. The default (TRUE) is that coefficients of taylor series will be ordered canonically.

**TAYLOR\_SIMPLIFIER** Function  
 - A function of one argument which TAYLOR uses to simplify coefficients of power series.

**TAYLOR\_TRUNCATE\_POLYNOMIALS** Variable  
 default: [TRUE] When FALSE polynomials input to TAYLOR are considered to have infinite precision; otherwise (the default) they are truncated based upon the input truncation levels.

**TAYTORAT** (*exp*) Function  
 converts exp from TAYLOR form to CRE form, i.e. it is like RAT(RATDISREP(exp)) although much faster.

**TRUNC** (*exp*) Function  
 causes exp which is in general representation to be displayed as if its sums were truncated Taylor series. E.g. compare EXP1: $X^2+X+1$ ; with EXP2:TRUNC( $X^2+X+1$ ); . Note that IS(EXP1=EXP2); gives TRUE.



**UNSUM** (*fun,n*)

Function

is the first backward difference  $\text{fun}(n) - \text{fun}(n-1)$ .(C1)  $G(P) := P \cdot 4^N / \text{BINOMIAL}(2 \cdot N, N)$ ;(D1) 
$$G(P) := \frac{P^4}{\text{BINOMIAL}(2N, N)}$$
(C2)  $G(N^4)$ ;(D2) 
$$\frac{N^4}{\text{BINOMIAL}(2N, N)}$$
(C3)  $\text{NUSUM}(D2, N, 0, N)$ ;(D3) 
$$\frac{2(N+1)(63N^4 + 112N^3 + 18N^2 - 22N + 3)4^N}{693 \text{BINOMIAL}(2N, N)} - \frac{2}{3 \cdot 11^7}$$
(C4)  $\text{UNSUM}(\%, N)$ ;(D4) 
$$\frac{N^4}{\text{BINOMIAL}(2N, N)}$$
**VERBOSE**

Variable

default: [FALSE] - if TRUE will cause comments about the progress of POWERSERIES to be printed as the execution of it proceeds.

## 30 Number Theory

### 30.1 Definitions for Number Theory

**BERN** (*x*) Function  
 gives the Xth Bernoulli number for integer X. ZEROBERN[TRUE] if set to FALSE excludes the zero BERNOULLI numbers. (See also BURN).

**BERNPOLY** (*v, n*) Function  
 generates the nth Bernoulli polynomial in the variable v.

**BFZETA** (*exp, n*) Function  
 BFLOAT version of the Riemann Zeta function. The 2nd argument is how many digits to retain and return, it's a good idea to request a couple of extra. This function is available by doing LOAD(BFFAC); .

**BGZETA** (*S, FPPREC*) Function  
 BGZETA is like BZETA, but avoids arithmetic overflow errors on large arguments, is faster on medium size arguments (say S=55, FPPREC=69), and is slightly slower on small arguments. It may eventually replace BZETA. BGZETA is available by doing LOAD(BFAC);.

**BHZETA** (*S, H, FPPREC*) Function  
 gives FPPREC digits of  

$$\text{SUM}((K+H)^{-S}, K, 0, \text{INF})$$
 This is available by doing LOAD(BFFAC);.

**BINOMIAL** (*X, Y*) Function  
 the binomial coefficient  $X*(X-1)*...*(X-Y+1)/Y!$ . If X and Y are integers, then the numerical value of the binomial coefficient is computed. If Y, or the value X-Y, is an integer, the binomial coefficient is expressed as a polynomial.

**BURN** (*N*) Function  
 is like BERN(N), but without computing all of the uncomputed Bernoullis of smaller index. So BURN works efficiently for large, isolated N. (BERN(402) takes about 645 secs vs 13.5 secs for BURN(402). BERN's time growth seems to be exponential, while BURN's is about cubic. But if next you do BERN(404), it only takes 12 secs, since BERN remembers all in an array, whereas BURN(404) will take maybe 14 secs or maybe 25, depending on whether MACSYMA needs to BFLOAT a better value of %PI.) BURN is available by doing LOAD(BFFAC);. BURN uses an observation of WGD that (rational) Bernoulli numbers can be approximated by (transcendental) zetas with tolerable efficiency.

**BZETA** Function  
 - This function is obsolete, see BFZETA.

**CF** (*exp*) Function

converts *exp* into a continued fraction. *exp* is an expression composed of arithmetic operators and lists which represent continued fractions. A continued fraction  $a+1/(b+1/(c+\dots))$  is represented by the list  $[a,b,c,\dots]$ .  $a,b,c,\dots$  must be integers. *Exp* may also involve  $\text{SQRT}(n)$  where  $n$  is an integer. In this case *CF* will give as many terms of the continued fraction as the value of the variable  $\text{CFLENGTH}[1]$  times the period. Thus the default is to give one period. (*CF* binds  $\text{LISTARITH}$  to  $\text{FALSE}$  so that it may carry out its function.)

**CFDISREP** (*list*) Function

converts the continued fraction represented by list into general representation.

```
(C1) CF([1,2,-3]+[1,-2,1]);
(D1)      [1, 1, 1, 2]
(C2) CFDISREP(%);

(D2)      1
          1 + -----
                1
            1 + -----
                    1
                1 + -
                    2
```

**CFEXPAND** (*x*) Function

gives a matrix of the numerators and denominators of the next-to-last and last convergents of the continued fraction *x*.

```
(C1) CF(SQRT(3));
(D1)      [1, 1, 2, 1, 2, 1, 2, 1]
(C2) CFEXPAND(%);

(D2)      [71  97]
          [    ]
          [41  56]

(C3) D2[1,2]/D2[2,2],NUMER;
(D3)      1.7321429
```

**CFLENGTH** Variable

default:  $[1]$  controls the number of terms of the continued fraction the function *CF* will give, as the value  $\text{CFLENGTH}[1]$  times the period. Thus the default is to give one period.

**CGAMMA** Function

- The Gamma function in the complex plane. Do  $\text{LOAD}(\text{CGAMMA})$  to use these functions. Functions *Cgamma*, *Cgamma2*, and *LogCgamma2*. These functions evaluate the Gamma function over the complex plane using the algorithm of Kuki, CACM

algorithm 421. Calculations are performed in single precision and the relative error is typically around 1.0E-7; evaluation at one point costs less than 1 msec. The algorithm provides for an error estimate, but the Macsyma implementation currently does not use it. Cgamma is the general function and may be called with a symbolic or numeric argument. With symbolic arguments, it returns as is; with real floating or rational arguments, it uses the Macsyma Gamma function; and for complex numeric arguments, it uses the Kuki algorithm. Cgamma2 of two arguments, real and imaginary, is for numeric arguments only; LogCgamma2 is the same, but the log-gamma function is calculated. These two functions are somewhat more efficient.

- CGAMMA2** Function  
- See CGAMMA.
- DIVSUM** ( $n,k$ ) Function  
adds up all the factors of  $n$  raised to the  $k$ th power. If only one argument is given then  $k$  is assumed to be 1.
- EULER** ( $X$ ) Function  
gives the  $X$ th Euler number for integer  $X$ . For the Euler-Mascheroni constant, see %GAMMA.
- FACTORIAL** ( $X$ ) Function  
The factorial function.  $\text{FACTORIAL}(X) = X!$ . See also MINFACTORIAL and FACTCOMB. The factorial operator is  $!$ , and the double factorial operator is  $!!$ .
- FIB** ( $X$ ) Function  
the  $X$ th Fibonacci number with  $\text{FIB}(0)=0$ ,  $\text{FIB}(1)=1$ , and  $\text{FIB}(-N)=(-1)^{(N+1)} * \text{FIB}(N)$ . PREVFIIB is  $\text{FIB}(X-1)$ , the Fibonacci number preceding the last one computed.
- FIBTOPHI** ( $exp$ ) Function  
converts  $\text{FIB}(n)$  to its closed form definition. This involves the constant %PHI ( $= (\text{SQRT}(5)+1)/2 = 1.618033989$ ). If you want the Rational Function Package to know About %PHI do  $\text{TELLRAT}(\%PHI^2-\%PHI-1)\$ \text{ALGEBRAIC:TRUE}\$$ .
- INRT** ( $X,n$ ) Function  
takes two integer arguments,  $X$  and  $n$ , and returns the integer  $n$ th root of the absolute value of  $X$ .
- JACOBI** ( $p,q$ ) Function  
is the Jacobi symbol of  $p$  and  $q$ .
- LCM** ( $exp1,exp2,\dots$ ) Function  
returns the Least Common Multiple of its arguments. Do  $\text{LOAD}(\text{FUNCTS})$ ; to access this function.

**MAXPRIME** Variable  
 default: [489318] - the largest number which may be given to the PRIME(n) command, which returns the nth prime.

**MINFACTORIAL** (*exp*) Function  
 examines *exp* for occurrences of two factorials which differ by an integer. It then turns one into a polynomial times the other. If *exp* involves binomial coefficients then they will be converted into ratios of factorials.

```
(C1) N!/(N+1)!;
(D1)
      N!
-----
(N + 1)!
(C2) MINFACTORIAL(%);
(D2)
      1
-----
N + 1
```

**PARTFRAC** (*exp*, *var*) Function  
 expands the expression *exp* in partial fractions with respect to the main variable, *var*. PARTFRAC does a complete partial fraction decomposition. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out. See EXAMPLE(PARTFRAC); for examples.

**PRIME** (*n*) Function  
 gives the nth prime. MAXPRIME[489318] is the largest number accepted as argument. Note: The PRIME command does not work in maxima, since it required a large file of primes, which most users do not want. PRIMEP does work however.

**PRIMEP** (*n*) Function  
 returns TRUE if *n* is a prime, FALSE if not.

**QUNIT** (*n*) Function  
 gives the principal unit of the real quadratic number field Sqrt(*n*) where *n* is an integer, i.e. the element whose norm is unity. This amounts to solving Pell's equation  $A^{**2} - n*B^{**2} = 1$ .

```
(C1) QUNIT(17);
(D1)
      Sqrt(17)+4
(C2) EXPAND(%*(Sqrt(17)-4));
(D2)
      1
```

**TOTIENT** (*n*) Function  
 is the number of integers less than or equal to *n* which are relatively prime to *n*.

- ZEROBERN** Variable  
default: [TRUE] - if set to FALSE excludes the zero BERNOULLI numbers. (See the BERN function.)
- ZETA (X)** Function  
gives the Riemann zeta function for certain integer values of X.
- ZETA%PI** Variable  
default: [TRUE] - if FALSE, suppresses ZETA(n) giving  $\text{coeff}^*\%PI^n$  for n even.



## 31 Symmetries

### 31.1 Definitions for Symmetries

**COMP2PUI** ( $n, l$ ) Function  
 réalise le passage des fonctions symétriques complètes, données dans la liste  $l$ , aux fonctions symétriques élémentaires de  $0$  à  $n$ . Si la liste  $l$  contient moins de  $n+1$  éléments les valeurs formelles viennent la compléter. Le premier élément de la liste  $l$  donne le cardinal de l'alphabet si il existe, sinon on le met égal à  $n$ .

```
COMP2PUI(3, [4, g]);
      2      3
[4, g, - g  + 2 h2, g  - 3 h2 g + 3 h3]
```

**CONT2PART** ( $pc, lvar$ ) Function  
 rend le polynôme partitionné associé à la forme contractée  $pc$  dont les variables sont dans  $lvar$ .

```
pc : 2*a^3*b*x^4*y + x^5$
CONT2PART(pc, [x, y]);
      3
[[2 a  b, 4, 1], [1, 5]]
```

Autres fonctions de changements de représentations :

CONTRACT, EXPLOSE, PART2CONT, PARTPOL, TCONTRACT, TPARTPOL.

**CONTRACT** ( $psym, lvar$ ) Function  
 rend une forme contractée (i.e. un monoôme par orbite sous l'action du groupe symétrique) du polynôme  $psym$  en les variables contenues dans la liste  $lvar$ . La fonction EXPLOSE réalise l'opération inverse. La fonction TCONTRACT teste en plus la symétrie du polynôme.

```
psym : EXPLOSE(2*a^3*b*x^4*y, [x, y, z]);
      3      4      3      4      3      4
2 a  b y z + 2 a  b x z + 2 a  b y z
      3      4      3      4      3      4
+ 2 a  b x z + 2 a  b x y + 2 a  b x y

CONTRACT(psym, [x, y, z]);
      3      4
2 a  b x y
```

Autres fonctions de changements de représentations :

CONT2PART, EXPLOSE, PART2CONT, PARTPOL, TCONTRACT, TPARTPOL.



**DIRECT** ( $[P_1, \dots, P_n], y, f, [lvar_1, \dots, lvar_n]$ ) Function

calcul l'image directe (voir M. GIUSTI, D. LAZARD et A. VALIBOUZE, ISSAC 1988, Rome) associée à la fonction  $f$ , en les listes de variables  $lvar_1, \dots, lvar_n$ , et aux polynômes  $P_1, \dots, P_n$  d'une variable  $y$ . L'arité de la fonction  $f$  est importante pour le calcul. Ainsi, si l'expression de  $f$  ne dépend pas d'une variable, non seulement il est inutile de donner cette variable mais cela diminue considérablement les calculs si on ne le fait pas.

```
DIRECT([z^2 - e1*z + e2, z^2 - f1*z + f2], z, b*v + a*u,
      [[u, v], [a, b]]);
```

$$z^2 - e_1 f_1 z - 4 e_2 f_2 + e_1^2 f_2 + e_2^2 f_1$$

```
DIRECT([z^3 - e1*z^2 + e2*z - e3, z^2 - f1*z + f2], z, b*v + a*u,
      [[u, v], [a, b]]);
```

$$Y^6 - 2 E_1 F_1 Y^5 - 6 E_2 F_2 Y^4 + 2 E_1^2 F_2 Y^4 + 2 E_2 F_1 Y^4$$

$$+ E_1^2 F_1 Y^4$$

$$+ 9 E_3 F_1 F_2 Y^3 + 5 E_1 E_2 F_1 F_2 Y^3 - 2 E_1^3 F_1 F_2 Y^3 - 2 E_3 F_1^3 Y^3$$

$$- 2 E_1^3 E_2 F_1 Y^3 + 9 E_2^2 F_2 Y^2 - 6 E_1^2 E_2 F_2 Y^2 + E_1^4 F_2 Y^2$$

$$- 9 E_1^2 E_3 F_1 F_2 Y^2 - 6 E_2^2 F_1 F_2 Y^2 + 3 E_1^2 E_2 F_1 F_2 Y^2$$

$$+ 2 E_1^4 E_3 F_1 Y^2$$

$$+ E_2^2 F_1^4 Y^2 - 27 E_2 E_3 F_1 F_2 Y^2 + 9 E_1^2 E_3 F_1 F_2 Y^2$$

$$+ 3 E_1^2 E_2^2 F_1 F_2 Y^2$$

$$- E_1^3 E_2 F_1 F_2 Y^2 + 15 E_2 E_3 F_1 F_2 Y^2 - 2 E_1^2 E_3 F_1 F_2 Y^2$$

$$- E_1^2 E_2^3 F_1 F_2 Y^2$$

$$- 2 E_2 E_3 F_1^5 Y^2 - 27 E_3^2 F_2^3 + 18 E_1^3 E_2 E_3 F_2^3 - 4 E_1^3 E_3 F_2^3$$

$$\begin{aligned}
& - 4 E_2^3 F_2^3 \\
& + E_1^2 E_2^2 F_2^3 + 27 E_3^2 F_1^2 F_2^2 - 9 E_1 E_2 E_3 F_1^2 F_2^2 + E_1^2 E_3^3 F_1^2 F_2^2 \\
& + E_2^3 F_1^2 F_2^2 - 9 E_3^2 F_1^4 F_2^2 + E_1^4 E_2 E_3 F_1^2 F_2^2 + E_3^2 F_1^6
\end{aligned}$$

Recherche du polynôme dont les racines sont les somme  $a+u$  ou  $a$  est racine de  $z^2 - e_1 z + e_2$  et  $u$  est racine de  $z^2 - f_1 z + f_2$

```
DIRECT([z^2 - e1*z + e2,z^2 - f1*z + f2], z,a+u,[[u],[a]]);
```

$$\begin{aligned}
& Y^4 - 2 F_1^3 Y^3 - 2 E_1^3 Y^3 + 2 F_2^2 Y^2 + F_1^2 Y^2 + 3 E_1 F_1 Y^2 + 2 E_2 Y^2 \\
& + E_1^2 Y^2 \\
& - 2 F_1 F_2 Y^2 - 2 E_1 F_2 Y^2 - E_1^2 F_1 Y^2 - 2 E_2 F_1 Y^2 - E_1^2 F_1 Y^2 \\
& - 2 E_1 E_2 Y^2 + F_2^2 \\
& + E_1 F_1 F_2 - 2 E_2 F_2 + E_1^2 F_2 + E_2 F_1^2 + E_1 E_2 F_1 + E_2^2
\end{aligned}$$

DIRECT peut prendre deux drapeaux possibles : ELEMENTAIRES et PUISSANCES (valeur par de'faut) qui permettent de de'composer les polynômes syme'triques apparaissant dans ce calcul par les fonctions syme'triques e'l'ementaires ou les fonctions puissances respectivement.

fonctions de SYM utilis'ees dans cette fonction :

```
MULTI_ORBIT (donc ORBIT), PUI_DIRECT, MULTI_ELEM
(donc ELEM), MULTI_PUI (donc PUI), PUI2ELE, ELE2PUI
(si le drapeau DIRECT est a' PUISSANCES).
```

### ELE2COMP ( $m, l$ )

Function

passé des fonctions syme'triques e'l'ementaires aux fonctions comple'tes. Similaire a' COMP2ELE et COMP2PUI.

autres fonctions de changements de bases :

```
COMP2ELE, COMP2PUI, ELE2PUI, ELEM, MON2SCHUR, MULTI_ELEM,
MULTI_PUI, PUI, PUI2COMP, PUI2ELE, PUIREDUC, SCHUR2COMP.
```

**ELE2POLYNOME** (*l,z*)

Fonction

donne le polynôme en  $z$  dont les fonctions symétriques élémentaires des racines sont dans la liste  $l$ .  $l=[n,e_1,\dots,e_n]$  où  $n$  est le degré du polynôme et  $e_i$  la  $i$ -ième fonction symétrique élémentaire.

```
ele2polynome([2,e1,e2],z);
```

$$Z^2 - E1 Z + E2$$

```
polynome2ele(x^7-14*x^5 + 56*x^3 - 56*x + 22,x);
```

```
[7, 0, - 14, 0, 56, 0, - 56, - 22]
ele2polynome([7, 0, - 14, 0, 56, 0, - 56, - 22],x);
```

$$X^7 - 14 X^5 + 56 X^3 - 56 X + 22$$

la réciproque : POLYNOME2ELE( $p,z$ )

autres fonctions à voir :

POLYNOME2ELE, PUI2POLYNOME.

**ELE2PUI** (*m, l*)

Fonction

passer des fonctions symétriques élémentaires aux fonctions complètes. Similaire à COMP2ELE et COMP2PUI.

autres fonctions de changements de bases :

```
COMP2ELE, COMP2PUI, ELE2COMP, ELEM, MON2SCHUR, MULTI_ELEM,
MULTI_PUI, PUI, PUI2COMP, PUI2ELE, PUIREDUC, SCHUR2COMP.
```

**ELEM** (*ele,sym,lvar*)

Fonction

décompose le polynôme symétrique  $sym$ , en les variables contenues de la liste  $lvar$ , par les fonctions symétriques élémentaires contenues dans la liste  $ele$ . Si le premier élément de  $ele$  est donné ce sera le cardinal de l'alphabet sinon on prendra le degré du polynôme  $sym$ . Si il manque des valeurs à la liste  $ele$  des valeurs formelles du type "ei" sont rajoutées. Le polynôme  $sym$  peut être donné sous 3 formes différentes : contractée (ELEM doit alors valoir 1 sa valeur par défaut), partitionnée (ELEM doit alors valoir 3) ou étendue (i.e. le polynôme en entier) (ELEM doit alors valoir 2). L'utilisation de la fonction PUI se réalise sur le même mode.

Sur un alphabet de cardinal 3 avec  $e_1$ , la première fonction symétrique élémentaire, valant 7, le polynôme symétrique en 3 variables dont la forme contractée (ne dépendant ici que de deux de ses variables) est  $x^4-2*x*y$  se décompose ainsi en les fonctions symétriques élémentaires :

```
ELEM([3,7],x^4-2*x*y,[x,y]);
```

$$28 e_3 + 2 e_2^2 - 198 e_2 + 2401$$

autres fonctions de changements de bases :

COMP2ELE, COMP2PUI, ELE2COMP, ELE2PUI, MON2SCHUR, MULTLELEM, MULTLPUI, PUI, PUI2COMP, PUI2ELE, PUIREDUC, SCHUR2COMP.

**EXPLOSE** (*pc,lvar*) Function  
rend le polynôme symétrique associé à la forme contractée pc. La liste lvar contient les variables.

```
EXPLOSE(a*x +1,[x,y,z]);
```

$$(x + y + z) a + 1$$

Autres fonctions de changements de représentations :

CONTRACT, CONT2PART, PART2CONT, PARTPOL, TCONTRACT, TPARTPOL.

**KOSTKA** (*part1,part2*) Function  
écrite par P. ESPERET) calcule le nombre de kostka associé aux partitions part1 et part2

```
kostka([3,3,3],[2,2,2,1,1,1]);
```

6

**LGTREILLIS** (*n,m*) Function  
rend la liste des partitions de poids n et de longueur m.

```
LGTREILLIS(4,2);
```

```
[[3, 1], [2, 2]]
```

Voir également : LTREILLIS, TREILLIS et TREINAT.

**LTREILLIS** (*n,m*) Function  
rend la liste des partitions de poids n et de longueur inférieure ou égale à m.

```
ltreillis(4,2);
```

```
[[4, 0], [3, 1], [2, 2]]
```

Voir également : LGTREILLIS, TREILLIS et TREINAT.

**MON2SCHUR** (*l*)

Function

la liste *l* représente la fonction de Schur  $S_l$  : On a  $l=[i_1, i_2, \dots, i_q]$  avec  $i_1 \leq i_2 \leq \dots \leq i_q$ . La fonction de Schur est  $S_{[i_1, i_2, \dots, i_q]}$  est le mineur de la matrice infinie  $(h_{i-j})_{i \geq 1, j \geq 1}$  composé des  $q$  premières lignes et des colonnes  $i_1+1, i_2+2, \dots, i_q+q$ .

On écrit cette fonction de Schur en fonction des formes monomiales en utilisant les fonctions TREINAT et KOSTKA. La forme rendue est un polynôme symétrique dans une de ses représentations contractées avec les variables  $x_1, x_2, \dots$

```
mon2schur([1,1,1]);
```

$$x_1 x_2 x_3$$

```
mon2schur([3]);
```

$$x_1^2 x_2^2 x_3^2 + x_1^3 x_2^3 + x_1^3$$

```
MON2SCHUR([1,2]);
```

$$x_1^2 x_2^2 x_3^2 + x_1^2 x_2^2$$

ce qui veut dire que pour 3 variables cela donne :

$$x_1^2 x_2^2 x_3^2 + x_1^2 x_2^2 + x_2^2 x_1^2 + x_1^2 x_3^2 + x_3^2 x_1^2 + x_2^2 x_3^2 + x_3^2 x_2^2$$

autres fonctions de changements de bases :

```
COMP2ELE, COMP2PUI, ELE2COMP, ELE2PUI, ELEM, MULTI_ELEM,
MULTI_PUI, PUI, PUI2COMP, PUI2ELE, PUIREDUC, SCHUR2COMP.
```

**MULTI\_ELEM** (*l\_elem, multi\_pc, l\_var*)

Function

decompose un polynôme multi-symétrique sous la forme multi-contractée *multi\_pc* en les groupes de variables contenue dans la liste de listes *l\_var* sur les groupes de fonctions symétriques élémentaires contenues dans *l\_elem*.

```
MULTI_ELEM([[2,e1,e2],[2,f1,f2]],a*x+a^2+x^3,[[x,y],[a,b]]);
```

$$2 x^3 - 2 f_2 + f_1 + e_1 f_1 - 3 e_1 e_2 + e_1$$

autres fonctions de changements de bases :

```
COMP2ELE, COMP2PUI, ELE2COMP, ELE2PUI, ELEM,
MON2SCHUR, MULTI_PUI, PUI, PUI2COMP, PUI2ELE,
PUIREDUC, SCHUR2COMP.
```

**MULTI\_ORBIT** (*P, [lvar1, lvar2, ..., lvarp]*)

Function

*P* est un polynôme en l'ensemble des variables contenues dans les listes *lvar1, lvar2, ..., lvarp*. Cette fonction ramène l'orbite du polynôme *P* sous l'action du produit des groupes symétriques des ensembles de variables représentés par ces *p* LISTES.

```

MULTI_ORBIT(a*x+b*y, [[x,y], [a,b]]);
      [b y + a x, a y + b x]
multi_orbit(x+y+2*a, [[x,y], [a,b,c]]);
      [Y + X + 2 C, Y + X + 2 B, Y + X + 2 A]

```

voir e'galement : ORBIT pour l'action d'un seul groupe syme'trique

### MULTI\_PUI

Function

est a' la fonction PUI ce que la fonction MULTIELEM est a' la fonction ELEM.

```

MULTI_PUI([[2,p1,p2], [2,t1,t2]], a*x+a^2+x^3, [[x,y], [a,b]]);

```

$$T2 + P1 T1 + \frac{3 P1 P2}{2} - \frac{P1}{2}$$

### MULTINOMIAL (r,part)

Function

ou' r est le poids de la partition part. Cette fonction rame'ne le coefficient multinomial associe' : si les parts de la partitions part sont i1, i2, ..., ik, le re'sultat de MULTINOMIAL est  $r!/(i1!i2!...ik!)$ .

### MULTSYM (ppart1, ppart2,N)

Function

re'alise le produit de deux polyno^mes syme'triques de N variables en ne travaillant que modulo l'action du groupe syme'trique d'ordre N. Les polyno^mes sont dans leur repre'sentation partitionne'e.

Soient les 2 polyno^mes syme'triques en x, y :  $3*(x+y) + 2*x*y$  et  $5*(x^2+y^2)$  dont les formes partitionne'es sont respectivement  $[[3,1],[2,1,1]]$  et  $[[5,2]]$ , alors leur produit sera donne' par :

```

MULTSYM([[3,1], [2,1,1]], [[5,2]], 2);
      [[10, 3, 1], [15, 2, 1], [15, 3, 0]]

```

soit  $10*(x^3*y+y^3*x)+15*(x^2*y +y^2*x) +15(x^3+y^3)$

Fonctions de changements de repre'sentations d'un polyno^me syme'trique :

CONTRACT, CONT2PART, EXPLOSE, PART2CONT, PARTPOL, TCONTRACT, TPARTPOL.

**ORBIT** ( $P, lvar$ ) Function  
 calcul l'orbite du polynôme  $P$  en les variables de la liste  $lvar$  sous l'action du groupe symétrique de l'ensemble des variables contenues dans la liste  $lvar$ .

```
orbit(a*x+b*y, [x,y]);

orbit(2*x+x^2, [x,y]);
```

$$[A Y + B X, B Y + A X]$$

$$[Y^2 + 2 Y, X^2 + 2 X]$$

voir également : MULTLORBIT pour l'action d'un produit de groupes symétriques sur un polynôme.

**PART2CONT** ( $ppart, lvar$ ) Function  
 passe de la forme partitionnée à la forme contractée d'un polynôme symétrique. La forme contractée est rendue avec les variables contenues dans  $lvar$ .

```
PART2CONT([[2*a^3*b,4,1]], [x,y]);
```

$$2 a^3 b x^4 y$$

Autres fonctions de changements de représentations :

CONTRACT, CONT2PART, EXPLOSE, PARTPOL, TCONTRACT, TPARTPOL.

**PARTPOL** ( $psym, lvar$ ) Function  
 $psym$  est un polynôme symétrique en les variables de  $lvar$ . Cette fonction ramène sa représentation partitionnée.

```
PARTPOL(-a*(x+y)+3*x*y, [x,y]);
```

$$[[3, 1, 1], [- a, 1, 0]]$$

Autres fonctions de changements de représentations :

CONTRACT, CONT2PART, EXPLOSE, PART2CONT, TCONTRACT, TPARTPOL.

**PERMUT** ( $l$ ) Function  
 ramène la liste des permutations de la liste  $l$ .

**POLYNOME2ELE** ( $p, x$ ) Function  
 donne la liste  $l=[n,e1,\dots,en]$  où  $n$  est le degré du polynôme  $p$  en la variable  $x$  et  $e_i$  la  $i$ -ième fonction symétrique élémentaire des racines de  $p$ .

```
POLYNOME2ELE(x^7-14*x^5 + 56*x^3 - 56*x + 22,x);
[7, 0, - 14, 0, 56, 0, - 56, - 22]
ELE2POLYNOME( [7, 0, - 14, 0, 56, 0, - 56, - 22],x);
      7      5      3
X  - 14 X  + 56 X  - 56 X + 22
```

la re'ciproque : ELE2POLYNOME(1,x)

### PRODRAC ( $L,K$ )

Function

$L$  est une liste contenant les fonctions syme'triques e'le'mentaires sur un ensemble  $A$ . PRODRAC rend le polyno^me dont les racines sont les produits  $K$  a'  $K$  des e'le'ments de  $A$ .

### PUI ( $pui,sym,lvar$ )

Function

de'compose le polyno^me syme'trique  $sym$ , en les variables contenues de la liste  $lvar$ , par les fonctions puissances contenues dans la liste  $pui$ . Si le premier e'le'ment de  $pui$  est donne' ce sera le cardinal de l'alphabet sinon on prendra le degre' du polyno^me  $sym$ . Si il manque des valeurs a' la liste  $pui$ , des valeurs formelles du type "pi" sont rajoute'es. Le polyno^me  $sym$  peut etre donne' sous 3 formes diffe'rentes : contracte'e (PUI doit alors valoir 1 sa valeur par de'faut), partitionne'e (PUI doit alors valoir 3) ou e'tendue (i.e. le polyno^me en entier) (PUI doit alors valoir 2). La fonction ELEM s'utilise de la me^me manie're.

```
PUI;
      1
PUI([3,a,b],u*x*y*z,[x,y,z]);
      3
(a  - 3 b a + 2 p3) u
      -----
      6
```

autres fonctions de changements de bases :

```
COMP2ELE, COMP2PUI, ELE2COMP, ELE2PUI, ELEM, MON2SCHUR,
MULTI_ELEM, MULTI_PUI, PUI2COMP, PUI2ELE, PUIREDUC,
SCHUR2COMP.
```

### PUI2COMP ( $N,LPUI$ )

Function

rend la liste des  $N$  premie'res fonctions comple'tes (avec en te^te le cardinal) en fonction des fonctions puissance donne'es dans la liste  $LPUI$ . Si la liste  $LPUI$  est vide le cardinal est  $N$  sinon c'est son premier e'le'ment similaire a' COMP2ELE et COMP2PUI.



```

PUI2COMP(2, []);

      2
      p1  + p2
      [2, p1, -----]
      2

PUI2COMP(3, [2, a1]);

      2      3
      a1  + p2 a1  + 3 p2 a1 + 2 p3
      [2, a1, -----, -----]
      2      6

```

Autres fonctions de changements de bases :

```

COMP2ELE, COMP2PUI, ELE2COMP, ELE2PUI, ELEM,
MON2SCHUR, MULTI_ELEM, MULTI_PUI, PUI, PUI2ELE,
PUIREduc, SCHUR2COMP.

```

### **PUI2ELE** (*N,LPUI*)

Function

re'alise le passage des fonctions puissances aux fonctions syme'triques e'le'mentaires. Si le drapeau PUI2ELE est GIRARD, on re'cupe're la liste des fonctions syme'triques e'le'mentaires de 1 a' N, et s'il est e'gal a' CLOSE, la Nie'me fonction syme'trique e'le'mentaire.

Autres fonctions de changements de bases :

```

COMP2ELE, COMP2PUI, ELE2COMP, ELE2PUI, ELEM,
MON2SCHUR, MULTI_ELEM, MULTI_PUI, PUI, PUI2COMP,
PUIREduc, SCHUR2COMP.

```

### **PUI2POLYNOME** (*X,LPUI*)

Function

calcul le polyno^me en X dont les fonctions puissances des racines sont donne'es dans la liste LPUI.

```

(C6) polynome2ele(x^3-4*x^2+5*x-1,x);
(D6) [3, 4, 5, 1]
(C7) ele2pui(3,%);
(D7) [3, 4, 6, 7]
(C8) pui2polynome(x,%);
(D8) X^3 - 4 X^2 + 5 X - 1

```

Autres fonctions a' voir :

POLYNOME2ELE, ELE2POLYNOME.

**PUI\_DIRECT** (*ORBITE*, [*lvar1*, ..., *lvarn*], [*d1*, *d2*, ..., *dn*])

Function

Soit  $f$  un polynome en  $n$  blocs de variables  $lvar1, \dots, lvarn$ . Soit  $ci$  le nombre de variables dans  $lvari$ . Et  $SC$  le produit des  $n$  groupes symétriques de degré  $c1, \dots, cn$ . Ce groupe agit naturellement sur  $f$ . La liste *ORBITE* est l'orbite, notée  $SC(f)$ , de la fonction  $f$  sous l'action de  $SC$ . (Cette liste peut être obtenue avec la fonction : *MULTI\_ORBIT*). Les  $di$  sont des entiers tels que  $c1 \leq d1$ ,  $c2 \leq d2, \dots, cn \leq dn$ . Soit  $SD$  le produit des groupes symétriques  $S_{d1} \times S_{d2} \times \dots \times S_{dn}$ .

la fonction *pui\_direct* ramène les  $N$  premières fonctions puissances de  $SD(f)$  de suites des fonctions puissances de  $SC(f)$  où  $N$  est le cardinal de  $SD(f)$ .

Le résultat est rendu sous forme multi-contractée par rapport à  $SD$ . i.e. on ne conserve qu'un élément par orbite sous l'action de  $SD$ .

$L: [[x, y], [a, b]]$

*PUI\_DIRECT*(*MULTI\_ORBIT*( $a*x+b*y$ ,  $L$ ),  $L$ , [2,2]);

$$[a^2 x^2, 4 a b x y + a^2 x^2]$$

*PUI\_DIRECT*(*MULTI\_ORBIT*( $a*x+b*y$ ,  $L$ ),  $L$ , [3,2]);

$$[2 A^2 X, 4 A B X Y + 2 A^2 X^2, 3 A^2 B X Y + 2 A^3 X^3, \\ 12 A^2 B X Y + 4 A^3 B X Y + 2 A^4 X^4, \\ 10 A^3 B X Y + 5 A^4 B X Y + 2 A^5 X^5, \\ 40 A^3 B X Y + 15 A^4 B X Y + 6 A^5 B X Y + 2 A^6 X^6]$$

*PUI\_DIRECT*( $[y+x+2*c, y+x+2*b, y+x+2*a]$ ,  $[[x, y], [a, b, c]]$ , [2,3]);

$$[3 x^2 + 2 a, 6 x y + 3 x^2 + 4 a x + 4 a^2, \\ 9 x^2 y + 12 a x y + 3 x^3 + 6 a x^2 + 12 a^2 x + 8 a^3]$$

*PUI\_DIRECT*( $[y+x+2*c, y+x+2*b, y+x+2*a]$ ,  $[[x, y], [a, b, c]]$ , [3,4]);

**PUIREDUC** ( $N, LPUI$ )

Function

$LPUI$  est une liste dont le premier élément est un entier  $M$ . *PUIREDUC* donne les  $N$  premières fonctions puissances en fonction des  $M$  premières.

```

PUIREduc(3, [2]);

3
  3 p1 p2 - p1
  [2, p1, p2, -----]

2

```

**RESOLVANTE** ( $p, x, f, [x_1, \dots, x_d]$ ) Function

calcule la re'solvante du polynome  $p$  de la variable  $x$  et de degre'  $n \geq d$  par la fonction  $f$  exprime'e en les variables  $x_1, \dots, x_d$ . Il est important pour l'efficacite' des calculs de ne pas mettre dans la liste  $[x_1, \dots, x_d]$  les variables n'intervenant pas dans la fonction de transformation  $f$ .

Afin de rendre plus efficaces les calculs on peut mettre des drapeaux a' la variable RESOLVANTE afin que des algorithmes adequates soient utilise's :

Si la fonction  $f$  est unitaire :

- un polynome d'une variable,
- line'aire ,
- alterne'e,
- une somme de variables,
- syme'trique en les variables qui apparaissent dans son expression,
- un produit de variables,
- la fonction de la re'solvante de Cayley (utilisable qu'en degre' 5)

$$\frac{(x_1 x_2 + x_2 x_3 + x_3 x_4 + x_4 x_5 + x_5 x_1 - (x_1 x_3 + x_3 x_5 + x_5 x_2 + x_2 x_4 + x_4 x_1))^2}{\text{generale,}}$$

le drapeau de RESOLVANTE pourra e'tre respectivement :

- unitaire,
- lineaire,
- alternee,
- somme,
- produit,
- cayley,
- generale.

```

resolvante:unitaire;
resolvante(x^7-14*x^5 + 56*x^3 - 56*x + 22,x,x^3-1,[x]);

7      6      5      4      3      2
Y  + 7 Y  - 539 Y  - 1841 Y  + 51443 Y  + 315133 Y  + 376999 Y

```

```

+ 125253

resolvante : lineaire;
resolvante(x^4-1,x,x1+2*x2+3*x3,[x1,x2,x3]);

      24      20      16      12      8      4
Y  + 80 Y  + 7520 Y  + 1107200 Y  + 49475840 Y  + 344489984 Y
+ 655360000
Meme solution pour :
resolvante : general;
resolvante(x^4-1,x,x1+2*x2+3*x3,[x1,x2,x3]);
resolvante(x^4-1,x,x1+2*x2+3*x3,[x1,x2,x3,x4])
direct([x^4-1],x,x1+2*x2+3*x3,[[x1,x2,x3]]);

resolvante:lineaire$
resolvante(x^4-1,x,x1+x2+x3,[x1,x2,x3]);

      4
Y  - 1

resolvante:symetrique$

resolvante(x^4-1,x,x1+x2+x3,[x1,x2,x3]);

      4
Y  - 1
resolvante(x^4+x+1,x,x1-x2,[x1,x2]);
      12      8      6      4      2
Y  + 8 Y  + 26 Y  - 112 Y  + 216 Y  + 229

resolvante:alternee$
resolvante(x^4+x+1,x,x1-x2,[x1,x2]);

      12      8      6      4      2
Y  + 8 Y  + 26 Y  - 112 Y  + 216 Y  + 229

resolvante:produit;
resolvante(x^7-7*x+3,x,x1*x2*x3,[x1,x2,x3]);

      35      33      29      28      27      26      24
Y  - 7 Y  - 1029 Y  + 135 Y  + 7203 Y  - 756 Y  + 1323 Y

      23      22      21      20      19
+ 352947 Y  - 46305 Y  - 2463339 Y  + 324135 Y  - 30618 Y

      18
- 453789 Y

```

```

      17          15          14          12
- 40246444 Y + 282225202 Y - 44274492 Y + 155098503 Y

      11
+ 12252303 Y

      10          9          8          7          6
+ 2893401 Y - 171532242 Y + 6751269 Y + 2657205 Y - 94517766 Y

      5          3
- 3720087 Y + 26040609 Y + 14348907

  resolvante:symetrique$
  resolvante(x^7-7*x+3,x,x1*x2*x3,[x1,x2,x3]);

      35          33          29          28          27          26          24
Y - 7 Y - 1029 Y + 135 Y + 7203 Y - 756 Y + 1323 Y

      23          22          21          20          19
+ 352947 Y - 46305 Y - 2463339 Y + 324135 Y - 30618 Y

      18
- 453789 Y

      17          15          14          12
- 40246444 Y + 282225202 Y - 44274492 Y + 155098503 Y

      11
+ 12252303 Y

      10          9          8          7          6
+ 2893401 Y - 171532242 Y + 6751269 Y + 2657205 Y - 94517766 Y

      5          3
- 3720087 Y + 26040609 Y + 14348907

  resolvante:cayley$
  resolvante(x^5-4*x^2+x+1,x,a,[]);

" resolvante de Cayley "

      6          5          4          3          2
X - 40 X + 4080 X - 92928 X + 3772160 X + 37880832 X + 93392896

```

Pour la re'solvante de Cayley, les 2 derniers arguments sont neutres et le polyno^me donne' en entre'e doit ne'cessairement e^tre de degre' 5.

Voir e'galement :

```

RESOLVANTE_BIPARTITE, RESOLVANTE_PRODUI_T_SYM,
RESOLVANTE_UNITAIRE, RESOLVANTE_ALTERNEE1, RESOLVANTE_KLEIN,

```

RESOLVANTE\_KLEIN3, RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE.

**RESOLVANTE\_ALTERNEE1** ( $p, x$ ) Function

calcule la transformation de  $p(x)$  de degre  $n$  par la fonction  $\prod_{1 \leq i < j \leq n-1} (x_i - x_j)$ .

Voir e'galement :

RESOLVANTE\_PRODUIT\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,  
RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE, RESOLVANTE\_BIPARTITE.

**RESOLVANTE\_BIPARTITE** ( $p, x$ ) Function

calcule la transformation de  $p(x)$  de degre  $n$  ( $n$  pair) par la fonction  $x_1 x_2 \dots x_{n/2} + x_{n/2+1} \dots x_n$

Voir e'galement :

RESOLVANTE\_PRODUIT\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,  
RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE, RESOLVANTE\_ALTERNEE1  
RESOLVANTE\_BIPARTITE( $x^6+108, x$ );

$$Y^{10} - 972 Y^8 + 314928 Y^6 - 34012224 Y^4$$

Voir e'galement :

RESOLVANTE\_PRODUIT\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,  
RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE,  
RESOLVANTE\_ALTERNEE1.

**RESOLVANTE\_DIEDRALE** ( $p, x$ ) Function

calcule la transformation de  $p(x)$  par la fonction  $x_1 x_2 + x_3 x_4$ .

resolvante\_diedrale( $x^5-3*x^4+1, x$ );

$$\begin{aligned} X^{15} &- 21 X^{12} - 81 X^{11} - 21 X^{10} + 207 X^9 + 1134 X^8 + 2331 X^7 - 945 X^6 \\ &- 4970 X^5 - 18333 X^4 - 29079 X^3 - 20745 X^2 - 25326 X - 697 \end{aligned}$$

Voir e'galement :

RESOLVANTE\_PRODUIT\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE\_ALTERNEE1, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,  
RESOLVANTE\_VIERER, RESOLVANTE.

**RESOLVANTE\_KLEIN** ( $p,x$ )

Function

calcule la transformation de  $p(x)$  par la fonction  $x \rightarrow x_1x_2x_4+x_4$ .

Voir e'galement :

RESOLVANTE\_PRODUIT\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE\_ALTERNEE1, RESOLVANTE, RESOLVANTE\_KLEIN3,  
RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE.

**RESOLVANTE\_KLEIN3** ( $p,x$ )

Function

calcule la transformation de  $p(x)$  par la fonction  $x \rightarrow x_1x_2x_4+x_4$ .

Voir e'galement :

RESOLVANTE\_PRODUIT\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE\_ALTERNEE1, RESOLVANTE\_KLEIN, RESOLVANTE,  
RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE.

**RESOLVANTE\_PRODUIT\_SYM** ( $p,x$ )

Function

calcule la liste toutes les r'esolvantes produit du polynome  $p(x)$ .

```

resolvante_produit_sym(x^5+3*x^4+2*x-1,x);

      5      4      10      8      7      6      5      4
[Y  + 3 Y  + 2 Y - 1, Y  - 2 Y  - 21 Y  - 31 Y  - 14 Y  - Y
+ 14 Y
      2      10      8      7      6      5      4      3      2
+ 3 Y  + 1, Y  + 3 Y  + 14 Y  - Y  - 14 Y  - 31 Y  - 21 Y  - 2 Y
+ 1, Y  - 2 Y  - 3 Y - 1, Y - 1]

resolvante:produit$
resolvante(x^5+3*x^4+2*x-1,x,a*b*c,[a,b,c]);

      10      8      7      6      5      4      3      2
Y  + 3 Y  + 14 Y  - Y  - 14 Y  - 31 Y  - 21 Y  - 2 Y  + 1

```

Voir e'galement :

RESOLVANTE, RESOLVANTE\_UNITAIRE,  
RESOLVANTE\_ALTERNEE1, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,

RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE.

**RESOLVANTE\_UNITAIRE** ( $p, q, x$ ) Function

calcule la r'esolvante du polyn\^ome  $p(x)$  par le polyn\^ome  $q(x)$ .

Voir e'galement :

RESOLVANTE\_PRODUI\_T\_SYM, RESOLVANTE,  
RESOLVANTE\_ALTERNEE1, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,  
RESOLVANTE\_VIERER, RESOLVANTE\_DIEDRALE.

**RESOLVANTE\_VIERER** ( $p, x$ ) Function

calcule la transformation de  $p(x)$  par la fonction  $x \rightarrow x^2 - x^3 + x^4$ .

Voir e'galement :

RESOLVANTE\_PRODUI\_T\_SYM, RESOLVANTE\_UNITAIRE,  
RESOLVANTE\_ALTERNEE1, RESOLVANTE\_KLEIN, RESOLVANTE\_KLEIN3,  
RESOLVANTE, RESOLVANTE\_DIEDRALE.

**SCHUR2COMP** ( $P, l\_var$ ) Function

:  $P$  est un polyno^me en les variables contenues dans la liste  $l\_var$ . Chacune des variables de  $l\_var$  repre'sente une fonction syme'trique comple'te. On repre'sente dans  $l\_var$  la i'e'me fonction syme'trique comple'te comme la concate'nation de la lettre  $h$  avec l'entier  $i$  :  $h_i$ . Cette fonction donne l'expression de  $P$  en fonction des fonctions de Schur.

SCHUR2COMP( $h1*h2-h3, [h1, h2, h3]$ );

s  
1, 2

SCHUR2COMP( $a*h3, [h3]$ );

s a  
3

**SOMRAC** ( $liste, K$ ) Function

la liste contient les fonctions syme'triques e'le'mentaires d'un polyno^me  $P$ . On calcul le polyno^mes dont les racines sont les sommes  $K$  a'  $K$  distinctes des racines de  $P$ .

Voir e'galement PRODRAC.

**TCONTRACT** ( $pol, lvar$ ) Function

teste si le polyno^me  $pol$  est syme'trique en les variables contenues dans la liste  $lvar$ . Si oui il rend une forme contracte'e comme la fonction CONTRACT.



Autres fonctions de changements de representations :

CONTRACT, CONT2PART, EXPLOSE, PART2CONT, PARTPOL, TPARTPOL.

**TPARTPOL** (*pol,lvar*)

Function

teste si le polynome  $pol$  est syme'trique en les variables contenues dans la liste  $lvar$ .  
Si oui il rend sa forme partionne'e comme la fonction PARTPOL.

Autres fonctions de changements de representations :

CONTRACT, CONT2PART, EXPLOSE, PART2CONT, PARTPOL, TCONTRACT.

**TREILLIS** (*n*)

Function

rame'ne toutes les partitions de poids  $n$ .

```
treillis(4);
```

```
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

Voir e'galement : LGTREILLIS, LTREILLIS et TREINAT.

**TREINAT**

Function

TREINAT(*part*) rame'ne la liste des partitions infe'rieures a' la partition *part* pour l'ordre naturel.

```
treinat([5]);
```

```
[[5]]
```

```
treinat([1,1,1,1,1]);
```

```
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],
```

```
[1, 1, 1, 1, 1]]
```

```
treinat([3,2]);
```

```
[[5], [4, 1], [3, 2]]
```

Voir e'galement : LGTREILLIS, LTREILLIS et TREILLIS.

## 32 Groups

### 32.1 Definitions for Groups

**TODD\_COXETER** (*relations, subgroup*) Function

Find the order of  $G/H$  where  $G$  is the Free Group modulo RELATIONS, and  $H$  is the subgroup of  $G$  generated by SUBGROUP. SUBGROUP is an optional argument, defaulting to []. In doing this it produces a multiplication table for the right action of  $G$  on  $G/H$ , where the cosets are enumerated  $[H, Hg_2, Hg_3, \dots]$ . This can be seen internally in the \$todd\_coxeter\_state. The multiplication tables for the variables are in table:todd\_coxeter\_state[2]. Then table[i] gives the table for the  $i$ th variable. `mulcoset(coset,i) := table[varnum][coset];`

Example:

```
(C1) symet(n):=create_list(if (j - i) = 1 then (p(i,j))3 else
    if (not i = j) then (p(i,j))2 else p(i,i) , j,1,n-1,i,1,j);
    <3>
(D1) SYMET(N) := CREATE_LIST(IF J - I = 1 THEN P(I, J)
    <2>
    ELSE (IF NOT I = J THEN P(I, J)
    ELSE P(I, I)), J, 1, N - 1, I, 1, J)
(C2) p(i,j) :=concat(x,i).concat(x,j);
(D2) P(I, J) := CONCAT(X, I) . CONCAT(X, J)
(C3) symet(5);
(D3) [X1 . X1, (X1 . X2)<3>, X2 . X2, (X1 . X3)<2>, (X2 . X3)<3>,
    X3 . X3, (X1 . X4)<2>, (X2 . X4)<2>, (X3 . X4)<3>, X4 . X4]
(C4) todd_coxeter(d3);

Rows tried 426
(D4) 120
(C5) todd_coxeter(d3, [x1]);

Rows tried 213
(D5) 60
(C6) todd_coxeter(d3, [x1, x2]);

Rows tried 71
(D6) 20
(C7) table:todd_coxeter_state[2]$
(C8) table:todd_coxeter_state[2]$
(C9) table[1];
(D9) {Array: FIXNUM #(0 2 1 3 7 6 5 4 8 11 17 9 12 14 13 20
    16 10 18 19 15 0 0 0 0 0 0 0 0 0 0 0)}
```

Note only the elements 1 thru 20 of this array d9 are meaningful. `table[1][4] = 7` indicates `coset4.var1 = coset7`

## 33 Runtime Environment

### 33.1 Introduction for Runtime Environment

- A file which is loaded automatically for you when you start up a MACSYMA, to customize MACSYMA for you. It is possible to have an init file written as a BATCH file of macsyms commands. We hope this makes it easier for users to customize their macsyms environment. Here is an example init file

```
/*-*-macsyms-*/
setup_autoload("share\bessel",j0,j1,jn);
showtime:all; comgrind:true;
```

The strange looking comment at the top of the file "/\*-\*-macsyms-\*/" tells that it is a macsyms-language file. Also: "SETUP\_AUTOLOAD" can be used to make functions in BATCH files autoloading, meaning that you can then use (for instance, here) the functions J0, J1 and Jn from the BESSEL package directly because when you use the function the BESSEL package will be loaded in for you automatically. If the second file name in the argument to SETUP\_AUTOLOAD is not specified (the preferred usage) then the standard search for second file names of "FASL", "TRLISP", and ">" is done.

### 33.2 INTERRUPTS

- There are several ways the user can interrupt a MACSYMA computation, usually with a control character. Do DESCRIBE(CHARACTERS); for details. MACSYMA will also be interrupted if ^Z (control-Z) is typed, as this will exit back to Unix shell level Usually Control-C interrupts the computation putting you in a break loop. Typing :t should give you top level maxima back again.

### 33.3 Definitions for Runtime Environment

**ALARMCLOCK** (*arg1, arg2, arg3*) Function

will execute the function of no arguments whose name is *arg3* when the time specified by *arg1* and *arg2* elapses. If *arg1* is the atom "TIME" then *arg3* will be executed after *arg2* seconds of real-time has elapsed while if *arg1* is the atom "RUNTIME" then *arg3* will be executed after *arg2* milliseconds of cpu time. If *arg2* is negative then the *arg1* timer is shut off.

**ALLOC** Function

takes any number of arguments which are the same as the replies to the "run out of space" question. It increases allocations accordingly. E.g. If the user knows initially that his problem will require much space, he can say ALLOC(4); to allocate the maximum amount initially. See also the DYNAMALLOC switch.

**BUG** ("*message*") Function

similar to mail, sends a message to MACSYMA Mail. This may be used for reporting bugs or suspected bugs in MACSYMA. Expressions may be included by referring to

them, outside double quotes, e.g. `BUG("I am trying to integrate",D3,"but it asks for more list space. What should I do?");`

**CLEARSCREEN** () Function

Clears the screen. The same as typing control-L.

**CONTINUE** Function

- Control-`~` typed while in MACSYMA causes LISP to be entered. The user can now type any LISP S-expression and have it evaluated. Typing (CONTINUE) or `^G` (control-G) causes MACSYMA to be re-entered.

**DDT** () Function

Exits from MACSYMA to the operating system level. (The same as control-Z on ITS, or control-C on Tops-20.)

**DELFILE** (*file-specification*) Function

will delete the file given by the file-specification (i.e. filename, secondname, device, user) from the given device.

**DISKFREE** () Function

With no args or an arg of TRUE, will return the total number of free blocks of disk space in the system. With an arg of 0, 1, or 13, it will return the number of free blocks of disk space on the respective disk pack. With an arg of SECONDARY or PRIMARY, it will return the total number of free blocks of disk space on the secondary or primary disk pack respectively.

**FEATURE** declaration

- A nice adjunct to the system. STATUS(FEATURE) gives you a list of system features. At present the list for MC is: MACSYMA, NOLDMSG, MACLISP, PDP10, BIGNUM, FASLOAD, HUNK, FUNARG, ROMAN, NEWIO, SFA, PAGING, MC, and ITS. Any of these "features" may be given as a second argument to STATUS(FEATURE,...); If the specified feature exists, TRUE will be returned, else FALSE. Note: these are system features, and not really "user related". See also DESCRIBE(features); for more user-oriented features.

**FEATUREP** (*a,f*) Function

attempts to determine whether the object *a* has the feature *f* on the basis of the facts in the current data base. If so, it returns TRUE, else FALSE. See DESCRIBE(FEATURES); .

(C1) DECLARE(J,EVEN)\$

(C2) FEATUREP(J,INTEGER);

(D2)

TRUE

**ROOM** () Function  
 types out a verbose description of the state of storage and stack management in the Macsyma. This simply utilizes the Lisp ROOM function. ROOM(FALSE) - types out a very terse description, containing most of the same information.

**STATUS** (*arg*) Function  
 will return miscellaneous status information about the user's MACSYMA depending upon the arg given. Permissible arguments and results are as follows:

- TIME - the time used so far in the computation.
- DAY - the day of the week.
- DATE - a list of the year, month, and day.
- DAYTIME - a list of the hour, minute, and second.
- RUNTIME - accumulated cpu time times the atom "MILLISECONDS" in the current MACSYMA.
- REALTIME - the real time (in sec) elapsed since the user started up his MACSYMA.
- GCTIME - the garbage collection time used so far in the current computation.
- TOTALGCTIME - gives the total garbage collection time used in MACSYMA so far.
- FREECORE - the number of blocks of core your MACSYMA can expand before it runs out of address space. (A block is 1024 words.) Subtracting that value from 250\*BLOCKS (the maximum you can get on MC) tells you how many blocks of core your MACSYMA is using up. (A MACSYMA with no "fix" file starts at approx. 191 blocks.)
- FEATURE - gives you a list of system features. At present the list for MC is: MACSYMA, NOLDMSG, MACLISP, PDP10, BIGNUM, FASLOAD, HUNK, FUNARG, ROMAN, NEWIO, SFA, PAGING, MC, and ITS. Any of these "features" may be given as a second argument to STATUS(FEATURE,...); If the specified feature exists, TRUE will be returned, else FALSE. Note: these are system features, and not really "user related".

For information about your files, see the FILEDEFAULTS(); command.

**TIME** (*Di1, Di2, ...*) Function  
 gives a list of the times in milliseconds taken to compute the Di. (Note: the Variable SHOWTIME, default: [FALSE], may be set to TRUE to have computation times printed out with each D-line.)



## 34 Miscellaneous Options

### 34.1 Introduction to Miscellaneous Options

In this section various options are discussed which have a global effect on the operation of maxima. Also various lists such as the list of all user defined functions, are discussed.

### 34.2 SHARE

- The SHARE directory on MC or on a DEC20 version of MACSYMA contains programs, information files, etc. which are considered to be of interest to the MACSYMA community. Most files on SHARE; are not part of the MACSYMA system per se and must be loaded individually by the user, e.g. `LOADFILE("array");`. Many files on SHARE; were contributed by MACSYMA users. Do `PRINTFILE(SHARE,USAGE,SHARE);` for more details and the conventions for contributing to the SHARE directory. For an annotated "table of contents" of the directory, do: `PRINTFILE(SHARE,>,SHARE);`

### 34.3 Definitions for Miscellaneous Options

#### ALIASES

Variable

default: [] atoms which have a user defined alias (set up by the ALIAS, ORDER-GREAT, ORDERLESS functions or by DECLAREing the atom a NOUN).

#### ALLSYM

Variable

default: [TRUE] - If TRUE then all indexed objects are assumed symmetric in all of their covariant and contravariant indices. If FALSE then no symmetries of any kind are assumed in these indices. Derivative indices are always taken to be symmetric.

#### ALPHABETIC

declaration

Adds to MACSYMA's alphabet which initially contains the letters A-Z, % and -. Thus, `DECLARE("~",ALPHABETIC)` enables `NEW~VALUE` to be used as a name.

#### APROPOS (*string*)

Function

takes a character string as argument and looks at all the MACSYMA names for ones with that string appearing anywhere within them. Thus, `APROPOS(EXP);` will return a long list of all the flags and functions which have EXP as part of their names, such as EXPAND, EXP, EXPONENTIALIZE. Thus if you can only remember part of the name of something you can use this command to find the rest of the name. Similarly, you could say `APROPOS(TR.);` to find a list of many of the switches relating to the TRANSLATOR (most of which begin with TR.).

#### ARGS (*exp*)

Function

returns a list of the args of exp. I.e. it is essentially equivalent to

```
SUBSTPART("[",exp,0)
```

Both ARGS and SUBSTPART depend on the setting of INFLAG.



- DUMMY** (*i1,i2,...*) Function  
 will set each index *i1,i2,...* to name of the form !*n* where *n* is a positive integer. This guarantees that dummy indices which are needed in forming expressions will not conflict with indices already in use. COUNTER[default 1] determines the numerical suffix to be used in generating the next dummy index. The prefix is determined by the option DUMMYX[!].
- GENINDEX** Variable  
 default: [I] is the alphabetic prefix used to generate the next variable of summation when necessary.
- GENSUMNUM** Variable  
 [0] is the numeric suffix used to generate the next variable of summation. If it is set to FALSE then the index will consist only of GENINDEX with no numeric suffix.
- INF** Variable  
 - real positive infinity.
- INFINITY** Variable  
 - complex infinity, an infinite magnitude of arbitrary phase angle. (See also INF and MINF.)
- INFOLISTS** Variable  
 default: [] a list of the names of all of the information lists in MACSYMA. These are: LABELS - all bound C,D, and E labels. VALUES - all bound atoms, i.e. user variables, not MACSYMA Options or Switches, (set up by : , :: , or functional binding). FUNCTIONS - all user defined functions (set up by f(x):=...). ARRAYS - declared and undeclared arrays (set up by : , :: , or :=...) MACROS - any Macros defined by the user. MYOPTIONS - all options ever reset by the user (whether or not they get reset to their default value). RULES - user defined pattern matching and simplification rules (set up by TELLSIMP, TELLSIMPAFTER, DEFMATCH, or, DEFRULE.) ALIASES - atoms which have a user defined alias (set up by the ALIAS, ORDERGREAT, ORDERLESS functions or by DECLAREing the atom a NOUN). DEPENDENCIES - atoms which have functional dependencies (set up by the DEPENDS or GRADEF functions). GRADEFS - functions which have user defined derivatives (set up by the GRADEF function). PROPS - atoms which have any property other than those mentioned above, such as atvalues, matchdeclares, etc. as well as properties specified in the DECLARE function. LET\_RULE\_PACKAGES - a list of all the user-defined let rule packages plus the special package DEFAULT\_LET\_RULE\_PACKAGE. (DEFAULT\_LET\_RULE\_PACKAGE is the name of the rule package used when one is not explicitly set by the user.)
- INTEGERP** (*exp*) Function  
 is TRUE if *exp* is an integer else FALSE.

**M1PBRANCH**

Variable

default: [FALSE] - "principal branch for -1 to a power". Quantities such as  $(-1)^{(1/3)}$  [i.e. "odd" rational exponent] and  $(-1)^{(1/4)}$  [i.e. "even" rational exponent] are now handled as indicated in the following chart:

DOMAIN:REAL(default)

$(-1)^{(1/3)}$ :        -1  
 $(-1)^{(1/4)}$ :     $(-1)^{(1/4)}$

DOMAIN:COMPLEX

M1PBRANCH:FALSE(default)	M1PBRANCH:TRUE
$(-1)^{(1/3)}$	$1/2+\%i*\text{sqrt}(3)/2$
$(-1)^{(1/4)}$	$\text{sqrt}(2)/2+\%i*\text{sqrt}(2)/2$

**NUMBERP** (*exp*)

Function

is TRUE if *exp* is an integer, a rational number, a floating point number or a bigfloat else FALSE.

**PROPERTIES** (*a*)

Function

will yield a list showing the names of all the properties associated with the atom *a*.

**PROPS**

special symbol

- atoms which have any property other than those explicitly mentioned in INFOLISTS, such as *atvalues*, *matchdeclares*, etc. as well as properties specified in the *DECLARE* function.

**PROPVARS** (*prop*)

Function

yields a list of those atoms on the PROPS list which have the property indicated by *prop*. Thus *PROPVARS(ATVALUE)* will yield a list of atoms which have *atvalues*.

**PUT** (*a, p, i*)

Function

associates with the atom *a* the property *p* with the indicator *i*. This enables the user to give an atom any arbitrary property.

**QPUT** (*a, p, i*)

Function

is similar to *PUT* but it doesn't have its arguments evaluated.

**REM** (*a, i*)

Function

removes the property indicated by *i* from the atom *a*.

**REMOVE** (*args*)

Function

will remove some or all of the properties associated with variables or functions. *REMOVE(a1, p1, a2, p2, ...)* removes the property *pi* from the atom *ai*. *Ai* and *pi* may also be lists as with *DECLARE*. *Pi* may be any property e.g. *FUNCTION*, *MODE.DECLARE*, etc. It may also be *TRANSFUN* implying that the translated LISP version of the function is to be removed. This is useful if one wishes to have

the MACSYMA version of the function executed rather than the translated version. Pi may also be OP or OPERATOR to remove a syntax extension given to ai (see Appendix II). If ai is "ALL" then the property indicated by pi is removed from all atoms which have it. Unlike the more specific remove functions (REVALUE, REMARRAY, REMFUNCTION, and REMRULE) REMOVE does not indicate when a given property is non-existent; it always returns "DONE".

**REVALUE** (*name1, name2, ...*) Function

removes the values of user variables (which can be subscripted) from the system. If name is ALL then the values of all user variables are removed. Values are those items given names by the user as opposed to those which are automatically labeled by MACSYMA as Ci, Di, or Ei.

**RENAME** (*exp*) Function

returns an expression equivalent to exp but with the dummy indices in each term chosen from the set [!1,!2,...]. Each dummy index in a product will be different; for a sum RENAME will try to make each dummy index in a sum the same. In addition, the indices will be sorted alphanumerically.

**RNCOMBINE** (*exp*) Function

transforms exp by combining all terms of exp 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 will achieve 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. Bugs to ASB.

**SCALARP** (*exp*) Function

is TRUE if exp is a number, constant, or variable DECLARED SCALAR, or composed entirely of numbers, constants, and such variables, but not containing matrices or lists.

**SCALEFACTORS** (*coordinatetransform*) Function

Here coordinatetransform evaluates to the form [[expression1, expression2, ...], indeterminate1, indeterminate2, ...], where indeterminate1, indeterminate2, etc. are the curvilinear coordinate variables and where a set of rectangular Cartesian components is given in terms of the curvilinear coordinates by [expression1, expression2, ...]. COORDINATES is set to the vector [indeterminate1, indeterminate2,...], and DIMENSION is set to the length of this vector. SF[1], SF[2], ..., SF[DIMENSION] are set to the coordinate scale factors, and SFPROD is set to the product of these scale factors. Initially, COORDINATES is [X, Y, Z], DIMENSION is 3, and SF[1]=SF[2]=SF[3]=SFPROD=1, corresponding to 3-dimensional rectangular Cartesian coordinates. To expand an expression into physical components in the current coordinate system, there is a function with usage of the form

**SETUP\_AUTOLOAD** (*file,func1,...,funcN*) Function

which takes two or more arguments: a file specification, and one or more function names, "funcI", and which indicates that if a call to "funcI" is made and "funcI" is not defined, that the file specified by "file" is to be automatically loaded in via LOAD, which file should contain a definition for "funcI". (This is the process by which calling e.g. INTEGRATE in a fresh MACSYMA causes various files to be loaded in.) As with the other file-handling commands in MACSYMA, the arguments to SETUP\_AUTOLOAD are not evaluated. Example: SETUP\_AUTOLOAD("bessel")\$ J1(0.0); . Note: SETUP\_AUTOLOAD does not work for array functions.



## 35 Rules and Patterns

### 35.1 Introduction to Rules and Patterns

This section discusses user defined pattern matching and simplification rules (set up by TELLSIMP, TELLSIMPATER, DEFMATCH, or, DEFRULE.) You may affect the main simplification procedures, or else have your rules applied explicitly using APPLY1 and APPLY2. There are additional mechanisms for polynomials rules under TELLRAT, and for commutative and non commutative algebra in chapter on AFFINE.

### 35.2 Definitions for Rules and Patterns

**APPLY1** (*exp, rule1, ..., ruln*) Function  
 repeatedly applies the first rule to *exp* until it fails, then repeatedly applies the same rule to all subexpressions of *exp*, left-to-right, until the first rule has failed on all subexpressions. Call the result of transforming *exp* in this manner *exp'*. Then the second rule is applied in the same fashion starting at the top of *exp'*. When the final rule fails on the final subexpression, the application is finished.

**APPLY2** (*exp, rule1, ..., ruln*) Function  
 differs from APPLY1 in that if the first rule fails on a given subexpression, then the second rule is repeatedly applied, etc. Only if they all fail on a given subexpression is the whole set of rules repeatedly applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule. MAXAPPLYDEPTH[10000] is the maximum depth to which APPLY1 and APPLY2 will delve.

**APPLYB1** (*exp, rule1, ..., ruln*) Function  
 is similar to APPLY1 but works from the "bottom up" instead of from the "top down". That is, it processes the smallest subexpression of *exp*, then the next smallest, etc. MAXAPPLYHEIGHT[10000] - is the maximum height to which APPLYB1 will reach before giving up.

**CURRENT\_LET\_RULE\_PACKAGE** Variable  
 default:[DEFAULT\_LET\_RULE\_PACKAGE] - the name of the rule package that is presently being used. The user may reset this variable to the name of any rule package previously defined via the LET command. Whenever any of the functions comprising the let package are called with no package name the value of  
     CURRENT\_LET\_RULE\_PACKAGE  
 is used. If a call such as LETSIMP(*expr,rule\_pkg\_name*); is made, the rule package *rule\_pkg\_name* is used for that LETSIMP command only, i.e. the value of CURRENT\_LET\_RULE\_PACKAGE is not changed.

**DEFAULT\_LET\_RULE\_PACKAGE**

Variable

- the name of the rule package used when one is not explicitly set by the user with LET or by changing the value of CURRENT\_LET\_RULE\_PACKAGE.

**DEFMATCH** (*progname, pattern, parm1, ..., parmn*)

Function

creates a function of n+1 arguments with the name progname which tests an expression to see if it can match a particular pattern. The pattern is some expression containing pattern variables and parameters. The parms are given explicitly as arguments to DEFMATCH while the pattern variables (if supplied) were given implicitly in a previous MATCHDECLARE function. The first argument to the created function progname, is an expression to be matched against the "pattern" and the other n arguments are the actual variables occurring in the expression which are to take the place of dummy variables occurring in the "pattern". Thus the parms in the DEFMATCH are like the dummy arguments to the SUBROUTINE statement in FORTRAN. When the function is "called" the actual arguments are substituted. For example:

```
(C1) NONZEROANDFREEOF(X,E) := IF E#0 AND FREEOF(X,E)
      THEN TRUE ELSE FALSE$
```

(IS(E#0 AND FREEOF(X,E)) is an equivalent function definition)

```
(C2) MATCHDECLARE(A,NONZEROANDFREEOF(X),B,FREEOF(X))$
```

```
(C3) DEFMATCH(LINEAR,A*X+B,X)$
```

This has caused the function LINEAR(exp,var1) to be defined. It

tests exp to see if it is of the form  $A*\text{var1}+B$  where A and B do not contain var1 and A is not zero. DEFMATCHed functions return (if the match is successful) a list of equations whose left sides are the pattern variables and parms and whose right sides are the expressions which the pattern variables and parameters matched. The pattern variables, but not the parameters, are set to the matched expressions. If the match fails, the function returns FALSE. Thus LINEAR( $3*Z+(Y+1)*Z+Y**2,Z$ ) would return [B=Y\*\*2, A=Y+4, X=Z]. Any variables not declared as pattern variables in MATCHDECLARE or as parameters in DEFMATCH which occur in pattern will match only themselves so that if the third argument to the DEFMATCH in (C4) had been omitted, then LINEAR would only match expressions linear in X, not in any other variable. A pattern which contains no parameters or pattern variables returns TRUE if the match succeeds. Do EXAMPLE(DEFMATCH); for more examples.

**DEFRULE** (*rulename, pattern, replacement*)

Function

defines and names a replacement rule for the given pattern. If the rule named rulename is applied to an expression (by one of the APPLY functions below), every subexpression matching the pattern will be replaced by the replacement. All variables in the replacement which have been assigned values by the pattern match are assigned those values in the replacement which is then simplified. The rules themselves can be treated as functions which will transform an expression by one operation of the pattern match and replacement. If the pattern fails, the original expression is returned.

**DISPRULE** (*rulename1, rulename2, ...*) Function  
 will display rules with the names *rulename1, rulename2*, as were given by `DEFRULE`, `TELLSIMP`, or `TELLSIMPAFTER` or a pattern defined by `DEFMATCH`. For example, the first rule modifying `SIN` will be called `SINRULE1`. `DISPRULE(ALL)`; will display all rules.

**LET** (*prod, repl, predname, arg1, arg2, ..., argn*) Function  
 defines a substitution rule for `LETSIMP` such that *prod* gets replaced by *repl*. *prod* is a product of positive or negative powers of the following types of terms:

- (1) Atoms which `LETSIMP` will search for literally unless previous to calling `LETSIMP` the `MATCHDECLARE` function is used to associate a predicate with the atom. In this case `LETSIMP` will match the atom to any term of a product satisfying the predicate.
- (2) Kernels such as `SIN(X)`, `N!`, `F(X,Y)`, etc. As with atoms above `LETSIMP` will look for a literal match unless `MATCHDECLARE` is used to associate a predicate with the argument of the kernel. A term to a positive power will only match a term having at least that power in the expression being `LETSIMP`ed. A term to a negative power on the other hand will only match a term with a power at least as negative. In the case of negative powers in "product" the switch `LETRAT` must be set to `TRUE` (see below). If a predicate is included in the `LET` function followed by a list of arguments, a tentative match (i.e. one that would be accepted if the predicate were omitted) will be accepted only if `predname(arg1',...,argn')` evaluates to `TRUE` where `argi'` is the value matched to `argi`. The `argi` may be the name of any atom or the argument of any kernel appearing in *prod*. *repl* may be any rational expression. If any of the atoms or arguments from *prod* appear in *repl* the appropriate substitutions will be made.

`LETRAT[FALSE]` when `FALSE`, `LETSIMP` will simplify the numerator and denominator of *expr* independently and return the result. Substitutions such as `N!/N` goes to `(N-1)!` will fail. To handle such situations `LETRAT` should be set to `TRUE`, then the numerator, denominator, and their quotient will be simplified in that order. These substitution functions allow you to work with several rulepackages at once. Each rulepackage can contain any number of `LET`ed rules and is referred to by a user supplied name. To insert a rule into the rulepackage *name*, do `LET([prod,repl,pred,arg1,...],name)`. To apply the rules in rulepackage *name*, do `LETSIMP(expr, name)`. The function `LETSIMP(expr,name1,name2,...)` is equivalent to doing `LETSIMP(expr,name1)` followed by `LETSIMP(% ,name2)` etc. `CURRENT_LET_RULE_PACKAGE` is the name of the rule package that is presently being used. The user may reset this variable to the name of any rule package previously defined via the `LET` command. Whenever any of the functions comprising the `let` package are called with no package name the value of `CURRENT_LET_RULE_PACKAGE` is used. If a call such as `LETSIMP(expr,rule_pkg_name)`; is made, the rule package `rule_pkg_name` is used for that `LETSIMP` command only, i.e. the value of `CURRENT_LET_RULE_PACKAGE` is not changed. There is a `DEFAULT_LET_RULE_PACKAGE` which is assumed when no other name is supplied to any of the functions. Whenever a `LET` includes a rulepackage name that is used as the `CURRENT_LET_RULE_PACKAGE`.



**LETRAT** Variable

default: [FALSE] - when FALSE, LETSIMP will simplify the numerator and denominator of expr independently and return the result. Substitutions such as  $N!/N$  goes to  $(N-1)!$  will fail. To handle such situations LETRAT should be set to TRUE, then the numerator, denominator, and their quotient will be simplified in that order.

**LETRULES** () Function

displays the rules in the current rulepackage. LETRULES(name) displays the rules in the named rulepackage. The current rulepackage is the value of

CURRENT\_LET\_RULE\_PACKAGE

The initial value of the rules is

DEFAULT\_LET\_RULE\_PACKAGE

**LETSIMP** (exp) Function

will continually apply the substitution rules previously defined by the function LET until no further change is made to exp. LETSIMP(expr,rule\_pkg\_name); will cause the rule package rule\_pkg\_name to be used for that LETSIMP command only, i.e. the value of CURRENT\_LET\_RULE\_PACKAGE is not changed.

**LET\_RULE\_PACKAGES** Variable

default:[DEFAULT\_LET\_RULE\_PACKAGE] - The value of LET\_RULE\_PACKAGES is a list of all the user-defined let rule packages plus the special package

DEFAULT\_LET\_RULE\_PACKAGE

This is the name of the rule package used when one is not explicitly set by the user.

**MATCHDECLARE** (patternvar, predicate, ...) Function

associates a predicate with a pattern variable so that the variable will only match expressions for which the predicate is not FALSE. (The matching is accomplished by one of the functions described below). For example after

MATCHDECLARE(Q,FREEOF(X,%E))

is executed, Q will match any expression not containing X or %E. If the match succeeds then the variable is set to the matched expression. The predicate (in this case FREEOF) is written without the last argument which should be the one against which the pattern variable is to be tested. Note that the patternvar and the arguments to the predicate are evaluated at the time the match is performed. The odd numbered argument may also be a list of pattern variables all of which are to have the associated predicate. Any even number of arguments may be given. For pattern matching, predicates refer to functions which are either FALSE or not FALSE (any non FALSE value acts like TRUE). MATCHDECLARE(var,TRUE) will permit var to match any expression.

**MATCHFIX** Function

- MATCHFIX operators are used to denote functions of any number of arguments which are passed to the function as a list. The arguments occur between the main

operator and its "matching" delimiter. The MATCHFIX("x",...) function is a syntax extension function which declares x to be a MATCHFIX operator. The default binding power is 180, and the ARGS inside may be anything.

```
(C1) matchfix("|","|");

(D1)      "|"
(C2) |a|+b;

(D2)      b + (|a|)
(C3) |(a,b)|;

(D3)      |b|
(C4) |[a,b]|;

(D4)      |[a, b]|

(C9) |x|:=IF NUMBERP(x) THEN ABS(x)
      ELSE (IF LISTP(x) AND APPLY("and",MAP(NUMBERP,x))
      THEN SUM(x[i]^2,i,1,LENGTH(x))^0.5 ELSE BUILDQ([u:x],|u|))$

(C10) |[1,2,3]|;

(D10)      3.741657386773941

(C18) |-7|;

(D18)      7
(C19) |[a,b]|;

(D19)      |[a, b]|
```

**REMLET** (*prod, name*) Function  
 deletes the substitution rule, *prod* → *repl*, most recently defined by the LET function. If *name* is supplied the rule is deleted from the rule package *name*. REMLET() and REMLET(ALL) delete all substitution rules from the current rulepackage. If the name of a rulepackage is supplied, e.g. REMLET(ALL,*name*), the rulepackage, *name*, is also deleted. If a substitution is to be changed using the same product, REMLET need not be called, just redefine the substitution using the same product (literally) with the LET function and the new replacement and/or predicate name. Should REMLET(*product*) now be called the original substitution rule will be revived.

**REMRULE** (*function, rulename*) Function  
 will remove a rule with the name *rulename* from the function which was placed there by DEFRULE, DEFMATCH, TELLSIMP, or TELLSIMPAFTER. If *rule-name* is ALL, then all rules will be removed.

**TELLSIMP** (*pattern, replacement*) Function  
is similar to TELLSIMPAFTER but places new information before old so that it is applied before the built-in simplification rules. TELSIMP is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier "knows" something about the expression, but what it returns is not to your liking. If the simplifier "knows" something about the main operator of the expression, but is simply not doing enough for you, you probably want to use TELLSIMPAFTER. The pattern may not be a sum, product, single variable, or number. RULES is a list of names having simplification rules added to them by DEFRULE, DEFMATCH, TELSIMP, or TELLSIMPAFTER. Do EXAMPLE(TELLSIMP); for examples.

**TELLSIMPAFTER** (*pattern, replacement*) Function  
defines a replacement for pattern which the MACSYMA simplifier uses after it applies the built-in simplification rules. The pattern may be anything but a single variable or a number.

## 36 Lists

### 36.1 Introduction to Lists

Lists are the basic building block for maxima and lisp. All data types other than arrays, hash tables, numbers are represented as lisp lists, These lisp lists have the form

```
((mplus) $A 2)
```

to indicate an expression  $A+2$ . At maxima level one would see the infix notation  $A+2$ . Maxima also has lists which are printed as

```
[1, 2, 7, x+y]
```

for a list with 4 elements. Internally this corresponds to a lisp list of the form

```
((mlist) 1 2 7 ((mplus) $X $Y ))
```

The flag which denotes the type field of the maxima expression is a list itself, since after it has been through the simplifier the list would become

```
((mlist simp) 1 2 7 ((mplus simp) $X $Y))
```

### 36.2 Definitions for Lists

**APPEND** (*list1, list2, ...*) Function  
 returns a single list of the elements of list1 followed by the elements of list2,... APPEND also works on general expressions, e.g. APPEND(F(A,B), F(C,D,E)); -> F(A,B,C,D,E). Do EXAMPLE(APPEND); for an example.

**ATOM** (*exp*) Function  
 is TRUE if exp is atomic (i.e. a number or name) else FALSE. Thus ATOM(5) is TRUE while ATOM(A[1]) and ATOM(SIN(X)) are FALSE. (Assuming A[1] and X are unbound.)

**CONS** (*exp, list*) Function  
 returns a new list constructed of the element exp as its first element, followed by the elements of list. CONS also works on other expressions, e.g. CONS(X, F(A,B,C)); -> F(X,A,B,C).

**COPYLIST** (*L*) Function  
 creates a copy of the list L.

**DELETE** (*exp1, exp2*) Function  
 removes all occurrences of exp1 from exp2. Exp1 may be a term of exp2 (if it is a sum) or a factor of exp2 (if it is a product).

```
(C1) DELETE(SIN(X), X+SIN(X)+Y);
(D1)          Y + X
```

DELETE(exp1, exp2, integer) removes the first integer occurrences of exp1 from exp2. Of course, if there are fewer than integer occurrences of exp1 in exp2 then all occurrences will be deleted.

**ENDCONS** (*exp, list*) Function  
 returns a new list consisting of the elements of list followed by exp. ENDCONS also works on general expressions, e.g. ENDCONS(X, F(A,B,C)); -> F(A,B,C,X).

**FIRST** (*exp*) Function  
 yields the first part of exp which may result in the first element of a list, the first row of a matrix, the first term of a sum, etc. Note that FIRST and its related functions, REST and LAST, work on the form of exp which is displayed not the form which is typed on input. If the variable INFLAG [FALSE] is set to TRUE however, these functions will look at the internal form of exp. Note that the simplifier re-orders expressions. Thus FIRST(X+Y) will be X if INFLAG is TRUE and Y if INFLAG is FALSE. (FIRST(Y+X) gives the same results).

**GET** (*a, i*) Function  
 retrieves the user property indicated by i associated with atom a or returns FALSE if a doesn't have property i.

```
(C1) PUT(%E, 'TRANSCENDENTAL, 'TYPE);
(D1)      TRANSCENDENTAL
(C2) PUT(%PI, 'TRANSCENDENTAL, 'TYPE)$
(C3) PUT(%I, 'ALGEBRAIC, 'TYPE)$
(C4) TYPEOF(EXP) := BLOCK([Q],
                          IF NUMBERP(EXP)
                          THEN RETURN('ALGEBRAIC),
                          IF NOT ATOM(EXP)
                          THEN RETURN(MAPLIST('TYPEOF, EXP)),
                          Q : GET(EXP, 'TYPE),
                          IF Q=FALSE
THEN ERRCATCH(ERROR(EXP,"is not numeric.)) ELSE Q)$
(C5) TYPEOF(2*%E+X*%PI);
X is not numeric.
(D5)      [[TRANSCENDENTAL, []], [ALGEBRAIC, TRANSCENDENTAL]]
(C6) TYPEOF(2*%E+%PI);
(D6)      [TRANSCENDENTAL, [ALGEBRAIC, TRANSCENDENTAL]]
```

**LAST** (*exp*) Function  
 yields the last part (term, row, element, etc.) of the exp.

**LENGTH** (*exp*) Function  
 gives (by default) the number of parts in the external (displayed) form of exp. For lists this is the number of elements, for matrices it is the number of rows, and for sums it is the number of terms. (See DISPFORM). The LENGTH command is affected by the INFLAG switch [default FALSE]. So, e.g. LENGTH(A/(B\*C)); gives 2 if INFLAG is FALSE (Assuming EXPTDISPFLAG is TRUE), but 3 if INFLAG is TRUE (the internal representation is essentially  $A*B^{-1}*C^{-1}$ ).

**LISTARITH**

Variable

default: [TRUE] - if FALSE causes any arithmetic operations with lists to be suppressed; when TRUE, list-matrix operations are contagious causing lists to be converted to matrices yielding a result which is always a matrix. However, list-list operations should return lists.

**LISTP** (*exp*)

Function

is TRUE if *exp* is a list else FALSE.

**MAKELIST** (*exp,var,lo,hi*)

Function

returns a list as value. MAKELIST may be called as MAKELIST(*exp,var,lo,hi*) ["lo" and "hi" must be integers], or as MAKELIST(*exp,var,list*). In the first case MAKELIST is analogous to SUM, whereas in the second case MAKELIST is similar to MAP. Examples:

```
MAKELIST(CONCAT(X,I),I,1,6) yields [X1,X2,X3,X4,X5,X6]
MAKELIST(X=Y,Y,[A,B,C]) yields [X=A,X=B,X=C]
```

**MEMBER** (*exp, list*)

Function

returns TRUE if *exp* occurs as a member of *list* (not within a member). Otherwise FALSE is returned. Member also works on non-list expressions, e.g. MEMBER(B, F(A,B,C)); -> TRUE.

**REST** (*exp, n*)

Function

yields *exp* with its first *n* elements removed if *n* is positive and its last -*n* elements removed if *n* is negative. If *n* is 1 it may be omitted. *Exp* may be a list, matrix, or other expression.

**REVERSE** (*list*)

Function

reverses the order of the members of the list (not the members themselves). REVERSE also works on general expressions, e.g. REVERSE(A=B); gives B=A. REVERSE default: [FALSE] - in the Plotting functions, if TRUE cause a left-handed coordinate system to be assumed.



## 37 Function Definition

### 37.1 Introduction to Function Definition

### 37.2 FUNCTION

- To define a function in MACSYMA you use the `:=` operator. E.g.

```
F(X):=SIN(X)
```

defines a function F. Anonymous functions may also be created using LAMBDA. For example

```
lambda([i,j], ... )
```

can be used instead of F where

```
F(I,J):=BLOCK([], ... );
MAP(LAMBDA([I],I+1),L)
```

would return a list with 1 added to each term.

You may also define a function with a variable number of arguments, by having a final argument which is assigned to a list of the extra arguments:

```
(C8) f([u]):=u;
(C9) f(1,2,3,4);
(D9) [1, 2, 3, 4]
(C11) f(a,b,[u]):=[a,b,u];
(C12) f(1,2,3,4,5,6);
(D12) [1, 2, [3, 4, 5, 6]]
```

The right hand side of a function is an expression. Thus if you want a sequence of expressions, you do

```
f(x):=(expr1,expr2,....,exprn);
```

and the value of `exprn` is what is returned by the function.

If you wish to make a `return` from some expression inside the function then you must use `block` and `return`.

```
block([],expr1,...,if(a>10) then return(a),...exprn)
```

is itself an expression, and so could take the place of the right hand side of a function definition. Here it may happen that the `return` happens earlier than the last expression.

The first `[]` in the `block`, may contain a list of variables and variable assignments, such as `[a:3,b,c:[]]`, which would cause the three variables `a`, `b`, and `c` to not refer to their global values, but rather have these special values for as long as the code executes inside the `block`, or inside functions called from inside the `block`. This is called *dynamic* binding, since the variables last from the start of the block to the time it exits. Once you return from the `block`, or throw out of it, the old values (if any) of the variables will be restored. It is certainly a good idea to protect your variables in this way. Note that the assignments in the block variables, are done in parallel. This means, that if you had used `c:a` in the above, the value of `c` would have been the value of `a` at the time you just entered the block, but before `a` was bound. Thus doing something like



```
block([a:a],expr1,... a:a+3,...exprn)
```

will protect the external value of `a` from being altered, but would let you access what that value was. Thus the right hand side of the assignments, is evaluated in the entering context, before any binding occurs. Using just `block([x],..` would cause the `x` to have itself as value, just as if it would have if you entered a fresh **MAXIMA** session.

The actual arguments to a function are treated in exactly same way as the variables in a block. Thus in

```
f(x):=(expr1,...exprn);
and
f(1);
```

we would have a similar context for evaluation of the expressions as if we had done

```
block([x:1],expr1,...exprn)
```

Inside functions, when the right hand side of a definition, may be computed at runtime, it is useful to use `define` and possibly `buildq`.

## 37.3 MACROS

**BUILDQ**([varlist],expression); Function  
 EXPRESSION is any single MAXIMA expression and VARLIST is a list of elements of the form <atom> or <atom>:<value>

### 37.3.1 Semantics

The <value>s in the <varlist> are evaluated left to right (the syntax <atom> is equivalent to <atom>:<atom>). then these values are substituted into <expression> in parallel. If any <atom> appears as a single argument to the special form `SPLICE` (i.e. `SPLICE(<atom>)` ) inside <expression>, then the value associated with that <atom> must be a macsyma list, and it is spliced into <expression> instead of substituted.

### 37.3.2 SIMPLIFICATION

The arguments to `BUILDQ` need to be protected from simplification until the substitutions have been carried out. This code should affect that by using `'`.

`buildq` can be useful for building functions on the fly. One of the powerful things about **MAXIMA** is that you can have your functions define other functions to help solve the problem. Further below we discuss building a recursive function, for a series solution. This defining of functions inside functions usually uses `define`, which evaluates its arguments. A number of examples are included under `splice`.

**SPLICE** (atom) Function

This is used with `buildq` to construct a list. This is handy for making argument lists, in conjunction with `BUILDQ`

```
MPRINT([X]) ::= BUILDQ([U : x],
  if (debuglevel > 3) print(splice(u)));
```

Including a call like

```
MPRINT("matrix is ",MAT,"with length",LENGTH(MAT))
```

is equivalent to putting in the line

```
IF DEBUGLEVEL > 3
  THEN PRINT("matrix is ",MAT,"with length",
            LENGTH(MAT))
```

A more non trivial example would try to display the variable values AND their names.

```
MSHOW(A,B,C)
```

should become

```
PRINT('A',"=",A,"",', 'B',"=",B,"", and", 'C',"=",C)
```

so that if it occurs as a line in a program we can print values.

```
(C101) foo(x,y,z):=mshow(x,y,z);
(C102) foo(1,2,3);
X = 1 , Y = 2 , and Z = 3
```

The actual definition of mshow is the following. Note how buildq lets you build 'QUOTED' structure, so that the 'u lets you get the variable name. Note that in macros, the RESULT is a piece of code which will then be substituted for the macro and evaluated.

```
MSHOW([lis]) := BLOCK([ans: [], N:LENGTH(lis)],
  FOR i THRU N DO
    (ans:APPEND(ans,
      BUILDQ([u:lis[i]],
        ['u',"=",u])),
  IF i < N
    THEN ans
    :APPEND(ans,
      IF i < N-1 THEN [","]
      ELSE [", and"])),
  BUILDQ([U:ans], PRINT(SPLICE(u))))
```

The splice also works to put arguments into algebraic operations:

```
(C108) BUILDQ([A:' [B,C,D]], +SPLICE(A));
(D108) D+C+B
```

Note how the simplification only occurs AFTER the substitution, The operation applying to the splice in the first case is the + while in the second it is the \*, yet logically you might think `splice(a)+splice(A)` could be replaced by `2*splice(A)`. No simplification takes place with the buildq To understand what SPLICE is doing with the algebra you must understand that for MAXIMA, a formula an operation like  $A+B+C$  is really internally similar to  $+(A,B,C)$ , and similarly for multiplication. Thus  $*(2,B,C,D)$  is  $2*B*C*D$

```
(C114) BUILDQ([A:' [B,C,D]], +SPLICE(A));
(D114) D+C+B
```

```
(C111) BUILDQ([A:' [B,C,D]], SPLICE(A)+SPLICE(A));
(D111) 2*D+2*C+2*B
but
(C112) BUILDQ([A:' [B,C,D]], 2*SPLICE(A));
```

(D112)  $2*B*C*D$

Finally the `buildq` can be invaluable for building recursive functions. Suppose your program is solving a differential equation using the series method, and has determined that it needs to build a recursion relation

$$F[N] := (-((N^2-2*N+1)*F[N-1]+F[N-2]+F[N-3]))/(N^2-N)$$

and it must do this on the fly inside your function. Now you would really like to add `expand`.

$$F[N] := \text{EXPAND}((-((N^2-2*N+1)*F[N-1]+F[N-2]+F[N-3]))/(N^2-N));$$

but how do you build this code. You want the `expand` to happen each time the function runs, NOT before it.

```
kill(f),
val: (-((N^2-2*N+1)*F[N-1]+F[N-2]+F[N-3]))/(N^2-N),
define(f[n], buildq([u:val], expand(u))),
```

does the job. This might be useful, since when you do

```
With the Expand
(C28) f[6];
(D28) -AA1/8-13*AA0/180
```

where as without it is kept unsimplified, and even after 6 terms it becomes:

```
(C25) f[6];
(D25) (5*(-4*(-3*(-2*(AA1+AA0)+AA1+AA0)/2
      -(AA1+AA0)/2+AA1)
/3
-(-2*(AA1+AA0)+AA1+AA0)/6+(-AA1-AA0)/2)
/4
+(-3*(-2*(AA1+AA0)+AA1+AA0)/2
-(AA1+AA0)/2+AA1)
/12-(2*(AA1+AA0)-AA1-AA0)/6)
/30
```

The expression quickly becomes complicated if not simplified at each stage, so the simplification must be part of the definition. Hence the `buildq` is useful for building the form.

## 37.4 OPTIMIZATION

When using `TRANSLATE` and generating code with `MACSYMA`, there are a number of techniques which can save time and be helpful. Do `DEMO("optimu.dem")` for a demonstration. In particular, the function `FLOATDEFUNK` from `TRANSL;OPTIMU FASL`, creates a function definition from a math-like expression, but it optimizes it (with `OPTIMIZE`) and puts in the `MODE_DECLARE`ations needed to `COMPILE` correctly. (This can be done by hand, of course). The demo will only run in a fresh `macsyma`.

## 37.5 Definitions for Function Definition

### **APPLY** (*function, list*)

Function

gives the result of applying the function to the list of its arguments. This is useful when it is desired to compute the arguments to a function before applying that function. For example, if L is the list [1, 5, -10.2, 4, 3], then APPLY(MIN,L) gives -10.2. APPLY is also useful when calling functions which do not have their arguments evaluated if it is desired to cause evaluation of them. For example, if FILESPEC is a variable bound to the list [TEST, CASE] then APPLY(CLOSEFILE,FILESPEC) is equivalent to CLOSEFILE(TEST,CASE). In general the first argument to APPLY should be preceded by a ' to make it evaluate to itself. Since some atomic variables have the same name as certain functions the values of the variable would be used rather than the function because APPLY has its first argument evaluated as well as its second.

### **BINDTEST** (*ai*)

Function

causes ai to signal an error if it ever is used in a computation unbound.

### **BLOCK** (*[v1,...,vk], statement1,...,statementj*)

Function

Blocks in MACSYMA are somewhat analogous to subroutines in FORTRAN or procedures in ALGOL or PL/I. Blocks are like compound statements but also enable the user to label statements within the block and to assign "dummy" variables to values which are local to the block. The  $v_i$  are variables which are local to the BLOCK and the  $stmt_i$  are any MACSYMA expressions. If no variables are to be made local then the list may be omitted. A block uses these local variables to avoid conflict with variables having the same names used outside of the block (i.e. global to the block). In this case, upon entry to the block, the global values are saved onto a stack and are inaccessible while the block is being executed. The local variables then are unbound so that they evaluate to themselves. They may be bound to arbitrary values within the block but when the block is exited the saved values are restored to these variables. The values created in the block for these local variables are lost. Where a variable is used within a block and is not in the list of local variables for that block it will be the same as the variable used outside of the block. If it is desired to save and restore other local properties besides VALUE, for example ARRAY (except for complete arrays), FUNCTION, DEPENDENCIES, ATVALUE, MATCHDECLARE, ATOMGRAD, CONSTANT, and NONSCALAR then the function LOCAL should be used inside of the block with arguments being the names of the variables. The value of the block is the value of the last statement or the value of the argument to the function RETURN which may be used to exit explicitly from the block. The function GO may be used to transfer control to the statement of the block that is tagged with the argument to GO. To tag a statement, precede it by an atomic argument as another statement in the BLOCK. For example: BLOCK([X],X:1,LOOP,X:X+1,...,GO(LOOP),...). The argument to GO must be the name of a tag appearing within the BLOCK. One cannot use GO to transfer to a tag in a BLOCK other than the one containing the GO. Blocks typically appear on the right side of a function definition but can be used in other places as well.

**BREAK** (*arg1*, ...) Function

will evaluate and print its arguments and will then cause a (MACSYMA-BREAK) at which point the user can examine and change his environment. Upon typing EXIT; the computation resumes. Control-A (^A) will enter a MACSYMA-BREAK from any point interactively. EXIT; will continue the computation. Control-X may be used inside the MACSYMA-BREAK to quit locally, without quitting the main computation.

**BUILDQ** Macro

- See DESCRIBE(MACROS); .

**CATCH** (*exp1*,...,*expn*) Function

evaluates its arguments one by one; if the structure of the *exp1* leads to the evaluation of an expression of the form THROW(*arg*), then the value of the CATCH is the value of THROW(*arg*). This "non-local return" thus goes through any depth of nesting to the nearest enclosing CATCH. There must be a CATCH corresponding to a THROW, else an error is generated. If the evaluation of the *exp1* does not lead to the evaluation of any THROW then the value of the CATCH is the value of *expn*.

```
(C1) G(L) := CATCH(MAP(LAMBDA([X],
    IF X < 0 THEN THROW(X) ELSE F(X)), L));
(C2) G([1, 2, 3, 7]);
(D2) [F(1), F(2), F(3), F(7)]
(C3) G([1, 2, -3, 7]);
(D3) - 3
```

The function G returns a list of F of each element of L if L consists only of non-negative numbers; otherwise, G "catches" the first negative element of L and "throws" it up.

**COMPILE** (*[filespec]*, *f1*, *f2*, ..., *fn*) Function

Compiles functions *fi* into the file "*filespec*". For convenience, see the COMPILE function.

**COMPGRIND** Variable

default: [FALSE] when TRUE function definitions output by COMPILE are pretty-printed.

**COMPILE** (*f*) Function

The COMPILE command is a convenience feature in macsyma. It handles the calling of the function COMPILE, which translates macsyma functions into lisp, the calling of the lisp compiler on the file produced by COMPILE, and the loading of the output of the compiler, known as a FASL file, into the macsyma. It also checks the compiler comment listing output file for certain common errors. Do PRINT-FILE(MCOMPI,DOC,MAXDOC); for more details. COMPILE(); causes macsyma to prompt for arguments. COMPILE(function1,function2,...); compiles the functions, it uses the name of function1 as the first name of the file to put the lisp output. COMPILE(ALL); or COMPILE(FUNCTIONS); will compile all functions. COMPILE([file-name],function1,function2,...); N.B. all arguments are evaluated, just like a normal function (it is a normal function!). Therefore, if you have variables with the same name as part of the file you can not ignore that fact.

**COMPILE\_LISP\_FILE** (*"input filename"*) Function

which takes an optional second argument of "output filename," can be used in conjunction with

```
TRANSLATE_FILE("filename").
```

For convenience you might define

```
Compile_and_load(FILENAME) :=
  LOAD(COMPILE_LISP_FILE(TRANSLATE_FILE(FILENAME) [2])) [2];
```

These file-oriented commands are to be preferred over the use of COMPILE, COMPFILE, and the TRANSLATE SAVE combination.

**DEFINE** (*f(x1, ...), body*) Function

is equivalent to  $f(x1, \dots) := \text{body}$  but when used inside functions it happens at execution time rather than at the time of definition of the function which contains it.

**DEFINE\_VARIABLE** Function

(*name, default-binding, mode, optional-documentation*)

introduces a global variable into the MACSYMA environment. This is for user-written packages, which are often translated or compiled. Thus

```
DEFINE_VARIABLE(FOO, TRUE, BOOLEAN);
```

does the following:

- (1) MODE\_DECLARE(FOO, BOOLEAN); sets it up for the translator.
- (2) If the variable is unbound, it sets it: FOO:TRUE.
- (3) DECLARE(FOO, SPECIAL); declares it special.
- (4) Sets up an assign property for it to make sure that it never gets set to a value of the wrong mode. E.g. FOO:44 would be an error once FOO is defined BOOLEAN.

See DESCRIBE(MODE\_DECLARE); for a list of the possible "modes". The optional 4th argument is a documentation string. When TRANSLATE\_FILE is used on a package which includes documentation strings, a second file is output in addition to the LISP file which will contain the documentation strings, formatted suitably for use in manuals, usage files, or (for instance) DESCRIBE. With any variable which has been DEFINE\_VARIABLE'd with mode other than ANY, you can give a VALUE\_CHECK property, which is a function of one argument called on the value the user is trying to set the variable to.

```
PUT('G5, LAMBDA([U], IF U#'G5 THEN ERROR("Don't set G5")),
    'VALUE_CHECK);
```

Use DEFINE\_VARIABLE(G5, 'G5, ANY\_CHECK, "this ain't supposed to be set by anyone but me.") ANY\_CHECK is a mode which means the same as ANY, but which keeps DEFINE\_VARIABLE from optimizing away the assign property.

**DISPFUN** (*f1, f2, ...*) Function

displays the definition of the user defined functions f1, f2, ... which may also be the names of array associated functions, subscripted functions, or functions with constant subscripts which are the same as those used when the functions were defined. DISPFUN(ALL) will display all user defined functions as given on the FUNCTIONS and ARRAYS lists except subscripted functions with constant subscripts. E.g. if the user has defined a function F(x), DISPFUN(F); will display the definition.

- FUNCTIONS** Variable  
 default: [] - all user defined functions (set up by f(x):=...).
- FUNDEF** (*functionname*) Function  
 returns the function definition associated with "functionname". FUNDEF(fnname);  
 is similar to DISPFUN(fnname); except that FUNDEF does not invoke display.
- FUNMAKE** (*name*,[*arg1*,...,*argn*]) Function  
 returns name(*arg1*,...,*argn*) without calling the function name.
- LOCAL** (*v1*, *v2*, ... ) Function  
 causes the variables *v1*,*v2*,... to be local with respect to all the properties in the statement in which this function is used. LOCAL may only be used in BLOCKs, in the body of function definitions or LAMBDA expressions, or in the EV function and only one occurrence is permitted in each. LOCAL is independent of CONTEXT.
- MACROEXPANSION** Variable  
 default:[FALSE] - Controls advanced features which affect the efficiency of macros. Possible settings: FALSE – Macros expand normally each time they are called. EXPAND – The first time a particular call is evaluated, the expansion is "remembered" internally, so that it doesn't have to be recomputed on subsequent calls making subsequent calls faster. The macro call still GRINDs and DISPLAYs normally, however extra memory is required to remember all of the expansions. DISPLACE – The first time a particular call is evaluated, the expansion is substituted for the call. This requires slightly less storage than when MACROEXPANSION is set to EXPAND and is just as fast, but has the disadvantage that the original macro call is no longer remembered and hence the expansion will be seen if DISPLAY or GRIND is called. See documentation for TRANSLATE and MACROS for more details.
- MODE\_CHECKP** Variable  
 default: [TRUE] - If TRUE, MODE\_DECLARE checks the modes of bound variables.
- MODE\_CHECK\_ERRORP** Variable  
 default: [FALSE] - If TRUE, MODE\_DECLARE calls error.
- MODE\_CHECK\_WARNP** Variable  
 default: [TRUE] - If TRUE, mode errors are described.
- MODE\_DECLARE** (*y1*, *mode1*, *y2*, *mode2*, ... ) Function  
 MODE\_DECLARE is a synonym for this. MODE\_DECLARE is used to declare the modes of variables and functions for subsequent translation or compilation of functions. Its arguments are pairs consisting of a variable *yi*, and a mode which is one of BOOLEAN, FIXNUM, NUMBER, RATIONAL, or FLOAT. Each *yi* may also be a list of variables all of which are declared to have *modei*. If *yi* is an array, and if every element of the array which is referenced has a value then ARRAY(*yi*, COMPLETE, *dim1*, *dim2*, ...) rather than

```
ARRAY(yi, dim1, dim2, ...)
```

should be used when first declaring the bounds of the array. If all the elements of the array are of mode FIXNUM (FLOAT), use FIXNUM (FLOAT) instead of COMPLETE. Also if every element of the array is of the same mode, say m, then

```
MODE_DECLARE(COMPLETEARRAY(yi),m)
```

should be used for efficient translation. Also numeric code using arrays can be made to run faster by declaring the expected size of the array, as in:

```
MODE_DECLARE(COMPLETEARRAY(A[10,10]),FLOAT)
```

for a floating point number array which is 10 x 10. Additionally one may declare the mode of the result of a function by using FUNCTION(F1,F2,...) as an argument; here F1,F2,... are the names of functions. For example the expression,

```
MODE_DECLARE([FUNCTION(F1,F2,...),X],FIXNUM,Q,
             COMPLETEARRAY(Q),FLOAT)
```

declares that X and the values returned by F1,F2,... are single-word integers and that Q is an array of floating point numbers. MODE\_DECLARE is used either immediately inside of a function definition or at top-level for global variables. Do PRINTFILE(MCOMPI,DOC,MAXDOC); for some examples of the use of MODE\_DECLARE in translation and compilation.

### MODE\_IDENTITY (*arg1, arg2*)

Function

A special form used with MODE\_DECLARE and MACROS to declare, e.g., a list of lists of flonums, or other compound data object. The first argument to MODE\_IDENTITY is a primitive value mode name as given to MODE\_DECLARE (i.e. [FLOAT, FIXNUM, NUMBER, LIST, ANY]), and the second argument is an expression which is evaluated and returned as the value of MODE\_IDENTITY. However, if the return value is not allowed by the mode declared in the first argument, an error or warning is signalled. The important thing is that the MODE of the expression as determined by the MACSYMA to Lisp translator, will be that given as the first argument, independent of anything that goes on in the second argument. E.g. X:3.3; MODE\_IDENTITY(FIXNUM,X); is an error. MODE\_IDENTITY(FLONUM,X) returns 3.3 . This has a number of uses, e.g., if you knew that FIRST(L) returned a number then you might write MODE\_IDENTITY(NUMBER, FIRST(L)). However, a more efficient way to do it would be to define a new primitive,

```
FIRSTNUMB(X) ::= BUILDQ([X], MODE_IDENTITY(NUMBER, X));
```

and use FIRSTNUMB every time you take the first of a list of numbers.

### TRANSBIND

Variable

default: [FALSE] - if TRUE removes global declarations in the local context. This applies to variables which are formal parameters to functions which one is TRANSLATE-ing from MACSYMA code to LISP.

### TRANSCOMPILE

Variable

default:[FALSE] - if true, TRANSLATE will generate the declarations necessary for possible compilation. The COMPFILE command uses TRANSCOMPILE:TRUE;.



**TRANSLATE** (*f1, f2, ...*)

Function

translates the user defined functions *f1, f2, ...* from the MACSYMA language to LISP (i.e. it makes them EXPRs). This results in a gain in speed when they are called. There is now a version of macsyima with the macsyima to lisp translator pre-loaded into it. It is available by typing :TM (for TranslateMacsyima) at DDT level. When given a file name, E.g. :TM GJC;TMTEST > , it gives that file to the function TRANSLATE\_FILE, and proceeds without further user interaction. If no file name is given, :TM gives a regular macsyima "(C1)" line. P.s. A user init file with second name "TM" will be loaded if it exists. You may just want to link this to your macsyima init file. Functions to be translated should include a call to MODE\_DECLARE at the beginning when possible in order to produce more efficient code. For example:

```
F(X1,X2,...):=BLOCK([v1,v2,...],
  MODE_DECLARE(v1,mode1,v2,mode2,...),...)
```

where the *X1, X2, ...* are the parameters to the function and the *v1, v2, ...* are the local variables. The names of translated functions are removed from the FUNCTIONS list if SAVEDEF is FALSE (see below) and are added to the PROPS lists. Functions should not be translated unless they are fully debugged. Also, expressions are assumed simplified; if they are not, correct but non-optimal code gets generated. Thus, the user should not set the SIMP switch to FALSE which inhibits simplification of the expressions to be translated. The switch TRANSLATE, default: [FALSE], If TRUE, causes automatic translation of a user's function to LISP. Note that translated functions may not run identically to the way they did before translation as certain incompatibilities may exist between the LISP and MACSYMA versions. Principally, the RAT function with more than one argument and the RATVARS function should not be used if any variables are MODE\_DECLAREd CRE. Also the PREDERROR:FALSE setting will not translate. SAVEDEF[TRUE] - if TRUE will cause the MACSYMA version of a user function to remain when the function is TRANSLATED. This permits the definition to be displayed by DISPFUN and allows the function to be edited. TRANSRUN[TRUE] - if FALSE will cause the interpreted version of all functions to be run (provided they are still around) rather than the translated version. One can translate functions stored in a file by giving TRANSLATE an argument which is a file specification. This is a list of the form [fn1,fn2,DSK,dir] where fn1 fn2 is the name of the file of MACSYMA functions, and dir is the name of a file directory. The result returned by TRANSLATE is a list of the names of the functions TRANSLATED. In the case of a file translation the corresponding element of the list is a list of the first and second new file names containing the LISP code resulting from the translation. This will be fn1 LISP on the disk directory dir. The file of LISP code may be read into MACSYMA by using the LOADFILE function.

**TRANSLATE\_FILE** (*file*)

Function

translates a file of MACSYMA code into a file of LISP code. It takes one or two arguments. The first argument is the name of the MACSYMA file, and the optional second argument is the name of the LISP file to produce. The second argument defaults to the first argument with second file name the value of TR\_OUTPUT\_FILE\_DEFAULT which defaults to TRLISP. For example: TRANSLATE\_FILE("test.mc"); will trans-

late "test.mc" to "test.LISP". Also produced is a file of translator warning messages of various degrees of severity. The second file name is always UNLISP. This file contains valuable (albeit obscure for some) information for tracking down bugs in translated code. Do APROPOS(TR\_) to get a list of TR (for TRANSLATE) switches. In summary, TRANSLATE\_FILE("foo.mc"), LOADFILE("foo.LISP") is "=" to BATCH("foo.mc") modulo certain restrictions (the use of " and % for example).

**TRANSRUN** Variable

default: [TRUE] - if FALSE will cause the interpreted version of all functions to be run (provided they are still around) rather than the translated version.

**TR\_ARRAY\_AS\_REF** Variable

default: [TRUE] - If TRUE runtime code uses the value of the variable as the array.

**TR\_BOUND\_FUNCTION\_APPLY** Variable

default: [TRUE] - Gives a warning if a bound variable is found being used as a function.

**TR\_FILE\_TTY\_MESSAGESP** Variable

default: [FALSE] - Determines whether messages generated by TRANSLATE\_FILE during translation of a file will be sent to the TTY. If FALSE (the default), messages about translation of the file are only inserted into the UNLISP file. If TRUE, the messages are sent to the TTY and are also inserted into the UNLISP file.

**TR\_FLOAT\_CAN\_BRANCH\_COMPLEX** Variable

default: [TRUE] - States whether the arc functions might return complex results. The arc functions are SQRT, LOG, ACOS, etc. e.g. When it is TRUE then ACOS(X) will be of mode ANY even if X is of mode FLOAT. When FALSE then ACOS(X) will be of mode FLOAT if and only if X is of mode FLOAT.

**TR\_FUNCTION\_CALL\_DEFAULT** Variable

default: [GENERAL] - FALSE means give up and call MEVAL, EXPR means assume Lisp fixed arg function. GENERAL, the default gives code good for MEXPRS and MLEXPRS but not MACROS. GENERAL assures variable bindings are correct in compiled code. In GENERAL mode, when translating F(X), if F is a bound variable, then it assumes that APPLY(F,[X]) is meant, and translates a such, with appropriate warning. There is no need to turn this off. With the default settings, no warning messages implies full compatibility of translated and compiled code with the macsyms interpreter.

**TR\_GEN\_TAGS** Variable

default: [FALSE] - If TRUE, TRANSLATE\_FILE generates a TAGS file for use by the text editor.

- TR\_NUMER** Variable  
 default: [FALSE] - If TRUE numer properties are used for atoms which have them, e.g. %PI.
- TR\_OPTIMIZE\_MAX\_LOOP** Variable  
 default: [100] - The maximum number of times the macro-expansion and optimization pass of the translator will loop in considering a form. This is to catch MACRO expansion errors, and non-terminating optimization properties.
- TR\_OUTPUT\_FILE\_DEFAULT** Variable  
 default: [TRLISP] - This is the second file name to be used for translated lisp output.
- TR\_PREDICATE\_BRAIN\_DAMAGE** Variable  
 default: [FALSE] - If TRUE, output possible multiple evaluations in an attempt to interface to the COMPARE package.
- TR\_SEMICOPILE** Variable  
 default: [FALSE] - If TRUE TRANSLATE\_FILE and COMPILE output forms which will be macroexpanded but not compiled into machine code by the lisp compiler.
- TR\_STATE\_VARS** Variable  
 default:  
 [TRANSCOMPILER, TR\_SEMICOPILE,  
 TR\_WARN\_UNDECLARED, TR\_WARN\_MEVAL, TR\_WARN\_FEXPR, TR\_WARN\_MODE,  
 TR\_WARN\_UNDEFINED\_VARIABLE, TR\_FUNCTION\_CALL\_DEFAULT,  
 TR\_ARRAY\_AS\_REF, TR\_NUMER]
- The list of the switches that affect the form of the translated output. This information is useful to system people when trying to debug the translator. By comparing the translated product to what should have been produced for a given state, it is possible to track down bugs.
- TR\_TRUE\_NAME\_OF\_FILE\_BEING\_TRANSLATED** Variable  
 default: [FALSE] is bound to the quoted string form of the true name of the file most recently translated by TRANSLATE\_FILE.
- TR\_VERSION** Variable  
 - The version number of the translator.
- TR\_WARNINGS\_GET ()** Function  
 Prints a list of warnings which have been given by the translator during the current translation.
- TR\_WARN\_BAD\_FUNCTION\_CALLS** Variable  
 default: [TRUE] - Gives a warning when when function calls are being made which may not be correct due to improper declarations that were made at translate time.

- TR\_WARN\_FEXPR** Variable  
 default: [COMPILE] - Gives a warning if any FEXPRs are encountered. FEXPRs should not normally be output in translated code, all legitimate special program forms are translated.
- TR\_WARN\_MEVAL** Variable  
 default: [COMPILE] - Gives a warning if the function MEVAL gets called. If MEVAL is called that indicates problems in the translation.
- TR\_WARN\_MODE** Variable  
 default: [ALL] - Gives a warning when variables are assigned values inappropriate for their mode.
- TR\_WARN\_UNDECLARED** Variable  
 default: [COMPILE] - Determines when to send warnings about undeclared variables to the TTY.
- TR\_WARN\_UNDEFINED\_VARIABLE** Variable  
 default: [ALL] - Gives a warning when undefined global variables are seen.
- TR\_WINDY** Variable  
 default: [TRUE] - Generate "helpfull" comments and programming hints.
- UNDECLAREDWARN** Variable  
 default: [COMPILE] - A switch in the Translator. There are four relevant settings: SETTING | ACTION ————— FALSE  
 | never print warning messages. COMPILE | warn when in COMPILE TRANSLATE | warn when in TRANSLATE and when TRANSLATE:TRUE ALL | warn in COMPILE and TRANSLATE ————— Do  
 MODE\_DECLARE(<variable>,ANY) to declare a variable to be a general macsyma variable (i.e. not limited to being FLOAT or FIXNUM). The extra work in declaring all your variables in code to be compiled should pay off.
- COMPILE\_FILE** (*filename,&optional-outfile*) Function  
 It takes filename which contains macsyma code, and translates this to lisp and then compiles the result. It returns a list of four files (the original file,translation, notes on translation and the compiled code).
- DECLARE\_TRANSLATED** (*FN1,FN2..*) Function  
 When translating a file of macsyma code to lisp, it is important for the translator to know which functions it sees in the file are to be called as translated or compiled functions, and which ones are just macsyma functions or undefined. Putting this declaration at the top of the file, lets it know that although a symbol does which does not yet have a lisp function value, will have one at call time. (MFUNCTION-CALL fn arg1 arg2.. ) is generated when the translator does not know fn is going to be a lisp function.



## 38 Program Flow

### 38.1 Introduction to Program Flow

MACSYMA provides a DO loop for iteration, as well as more primitive constructs such as GO.

### 38.2 Definitions for Program Flow

#### BACKTRACE

Variable

default: [] (when DEBUGMODE:ALL has been done) has as value a list of all functions currently entered.

#### DO

special operator

- The DO statement is used for performing iteration. Due to its great generality the DO statement will be described in two parts. First the usual form will be given which is analogous to that used in several other programming languages (FORTRAN, ALGOL, PL/I, etc.); then the other features will be mentioned. 1. There are three variants of this form that differ only in their terminating conditions. They are:

- (a) FOR variable : initial-value STEP increment THRU limit DO body
- (b) FOR variable : initial-value STEP increment WHILE condition DO body
- (c) FOR variable : initial-value STEP increment UNLESS condition DO body

(Alternatively, the STEP may be given after the termination condition or limit.) The initial-value, increment, limit, and body can be any expressions. If the increment is 1 then "STEP 1" may be omitted. The execution of the DO statement proceeds by first assigning the initial-value to the variable (henceforth called the control-variable). Then: (1) If the control-variable has exceeded the limit of a THRU specification, or if the condition of the UNLESS is TRUE, or if the condition of the WHILE is FALSE then the DO terminates. (2) The body is evaluated. (3) The increment is added to the control-variable. The process from (1) to (3) is performed repeatedly until the termination condition is satisfied. One may also give several termination conditions in which case the DO terminates when any of them is satisfied. In general the THRU test is satisfied when the control-variable is greater than the limit if the increment was non-negative, or when the control-variable is less than the limit if the increment was negative. The increment and limit may be non-numeric expressions as long as this inequality can be determined. However, unless the increment is syntactically negative (e.g. is a negative number) at the time the DO statement is input, MACSYMA assumes it will be positive when the DO is executed. If it is not positive, then the DO may not terminate properly. Note that the limit, increment, and termination condition are evaluated each time through the loop. Thus if any of these involve much computation, and yield a result that does not change during all the executions of the body, then it is more efficient to set a variable to their value prior to the DO and use this variable in the DO form. The value normally returned by a DO statement is the atom DONE, as every statement in MACSYMA returns a value. However, the

function RETURN may be used inside the body to exit the DO prematurely and give it any desired value. Note however that a RETURN within a DO that occurs in a BLOCK will exit only the DO and not the BLOCK. Note also that the GO function may not be used to exit from a DO into a surrounding BLOCK. The control-variable is always local to the DO and thus any variable may be used without affecting the value of a variable with the same name outside of the DO. The control-variable is unbound after the DO terminates.

```
(C1)  FOR A:-3 THRU 26 STEP 7 DO LDISPLAY(A)$
(E1)      A = -3
(E2)      A = 4
(E3)      A = 11
(E4)      A = 18
(E5)      A = 25
```

The function LDISPLAY generates intermediate labels; DISPLAY does not.

```
(C6)  S:0$
(C7)  FOR I:1 WHILE I<=10 DO S:S+I;
(D7)      DONE
(C8)  S;
(D8)      55
```

Note that the condition in C7 is equivalent to UNLESS I > 10 and also THRU 10

```
(C9)  SERIES:1$
(C10) TERM:EXP(SIN(X))$
(C11) FOR P:1 UNLESS P>7 DO
      (TERM:DIFF(TERM,X)/P,
      SERIES:SERIES+SUBST(X=0,TERM)*X^P)$
(C12) SERIES;
      7      6      5      4      2
(D12)  X      X      X      X      X
      -- - --- - -- - -- + -- + X + 1
      96  240  15  8  2
```

which gives 8 terms of the Taylor series for  $e^{\sin(x)}$ .

```
(C13) POLY:0$
(C14) FOR I:1 THRU 5 DO
      FOR J:I STEP -1 THRU 1 DO
      POLY:POLY+I*X^J$
(C15) POLY;
      5      4      3      2
(D15)  5 X + 9 X + 12 X + 14 X + 15 X
(C16) GUESS:-3.0$
(C17) FOR I:1 THRU 10 DO (GUESS:SUBST(GUESS,X,.5*(X+10/X)),
      IF ABS(GUESS^2-10)<.00005 THEN RETURN(GUESS));
(D17)  - 3.1622807
```

This example computes the negative square root of 10 using the Newton- Raphson iteration a maximum of 10 times. Had the convergence criterion not been met the value returned would have been "DONE". Additional Forms of the DO Statement Instead of always adding a quantity to the control-variable one may sometimes wish to change it in some other way for each iteration. In this case one may use "NEXT

expression" instead of "STEP increment". This will cause the control-variable to be set to the result of evaluating expression each time through the loop.

```
(C1) FOR COUNT:2 NEXT 3*COUNT THRU 20
      DO DISPLAY(COUNT)$
          COUNT = 2
          COUNT = 6
          COUNT = 18
```

As an alternative to FOR variable:value ...DO... the syntax FOR variable FROM value ...DO... may be used. This permits the "FROM value" to be placed after the step or next value or after the termination condition. If "FROM value" is omitted then 1 is used as the initial value. Sometimes one may be interested in performing an iteration where the control-variable is never actually used. It is thus permissible to give only the termination conditions omitting the initialization and updating information as in the following example to compute the square-root of 5 using a poor initial guess.

```
(C1) X:1000;
(C2) THRU 10 WHILE X#0.0 DO X:.5*(X+5.0/X)$
(C3) X;
(D3) 2.236068
```

If it is desired one may even omit the termination conditions entirely and just give "DO body" which will continue to evaluate the body indefinitely. In this case the function RETURN should be used to terminate execution of the DO.

```
(C1) NEWTON(F,GUESS):=
      BLOCK([NUMER,Y],
          LOCAL(DF),
          NUMER:TRUE,
          DEFINE(DF(X),DIFF(F(X),X)),
          DO(Y:DF(GUESS),
              IF Y=0.0 THEN ERROR("Derivative at:",GUESS," is zero."),
              GUESS:GUESS-F(GUESS)/Y,
              IF ABS(F(GUESS))<5.0E-6 THEN RETURN(GUESS)))$
(C2) SQR(X):=X^2-5.0$
(C3) NEWTON(SQR,1000);
(D3) 2.236068
```

(Note that RETURN, when executed, causes the current value of GUESS to be returned as the value of the DO. The BLOCK is exited and this value of the DO is returned as the value of the BLOCK because the DO is the last statement in the block.) One other form of the DO is available in MACSYMA. The syntax is:

```
FOR variable IN list [end-tests] DO body
```

The members of the list are any expressions which will successively be assigned to the variable on each iteration of the body. The optional end-tests can be used to terminate execution of the DO; otherwise it will terminate when the list is exhausted or when a RETURN is executed in the body. (In fact, list may be any non-atomic expression, and successive parts are taken.)



```

(C1) FOR F IN [LOG, RHO, ATAN] DO LDISP(F(1))$
(E1)                                0
(E2)                                RHO(1)
                                   %PI
(E3)                                ---
                                   4
(C4) EV(E3,NUMER);
(D4)                                0.78539816

```

**ERRCATCH** (*exp1, exp2, ...*) Function

evaluates its arguments one by one and returns a list of the value of the last one if no error occurs. If an error occurs in the evaluation of any arguments, **ERRCATCH** "catches" the error and immediately returns [] (the empty list). This function is useful in BATCH files where one suspects an error might occur which would otherwise have terminated the BATCH if the error weren't caught.

**ERREXP** Variable

default: [ERREXP] When an error occurs in the course of a computation, MACSYMA prints out an error message and terminates the computation. **ERREXP** is set to the offending expression and the message "ERREXP contains the offending expression" is printed. The user can then type **ERREXP**; to see this and hopefully find the problem.

**ERROR** (*arg1, arg2, ...*) Function

will evaluate and print its arguments and then will cause an error return to top level MACSYMA or to the nearest enclosing **ERRCATCH**. This is useful for breaking out of nested functions if an error condition is detected, or wherever one can't type control-^ . The variable **ERROR** is set to a list describing the error, the first of it being a string of text, and the rest the objects in question. **ERRORMSG()**; is the preferred way to see the last error message. **ERRORFUN** default: [FALSE] - if set to the name of a function of no arguments will cause that function to be executed whenever an error occurs. This is useful in BATCH files where the user may want his MACSYMA killed or his terminal logged out if an error occurs. In these cases **ERRORFUN** would be set to **QUIT** or **LOGOUT**.

**ERRORFUN** Variable

default: [FALSE] - if set to the name of a function of no arguments will cause that function to be executed whenever an error occurs. This is useful in BATCH files where the user may want his MACSYMA killed or his terminal logged out if an error occurs. In these cases **ERRORFUN** would be set to **QUIT** or **LOGOUT**.

**ERRORMSG** () Function

reprints the last error message. This is very helpful if you are using a display console and the message has gone off the screen. The variable **ERROR** is set to a list describing the error, the first of it being a string of text, and the rest the objects in question. **TTYINTFUN:LAMBDA([],ERRORMSG(),PRINT(""))\$** will set up the user-interrupt character (^U) to reprint the message.

**FOR** special operator  
 - Used in iterations, do DESCRIBE("DO"); for a description of MACSYMA's iteration facilities.

**GO** (*tag*) Function  
 is used within a BLOCK to transfer control to the statement of the block which is tagged with the argument to GO. To tag a statement, precede it by an atomic argument as another statement in the BLOCK. For example:

```
BLOCK([X], X:1, LOOP, X+1, ..., GO(LOOP), ...)
```

. The argument to GO must be the name of a tag appearing in the same BLOCK. One cannot use GO to transfer to tag in a BLOCK other than the one containing the GO.

**IF** special operator  
 - The IF statement is used for conditional execution. The syntax is:

```
IF condition THEN expression1 ELSE expression2.
```

The result of an IF statement is expression1 if condition is true and expression2 if it is false. expression1 and expression2 are any MACSYMA expressions (including nested IF statements), and condition is an expression which evaluates to TRUE or FALSE and is composed of relational and logical operators which are as follows:

Operator name	Symbol	Type
greater than	>	relational infix
equal to	= , EQUAL	" "
not equal to	#	" "
less than	<	" "
greater than or equal to	>=	" "
less than or equal to	<=	" "
and	AND	logical infix
or	OR	" "
not	NOT	logical prefix

**LISPDEBUGMODE** () Function  
 LISPDEBUGMODE(); DEBUGPRINTMODE(); and DEBUG(); make available to the user debugging features used by systems programmers. These tools are powerful, and although some conventions are different from the usual macsyms level it is felt their use is very intuitive. [Some printout may be verbose for slow terminals, there are switches for controlling this.] These commands were designed for the user who must debug translated macsyms code, as such they are a boon. See MACDOC;TRDEBG USAGE for more information. For more help, consult GJC.

**MAP** (*fn, exp1, exp2, ...*) Function  
 returns an expression whose leading operator is the same as that of the expi but whose subparts are the results of applying fn to the corresponding subparts of the expi. Fn

is either the name of a function of  $n$  arguments (where  $n$  is the number of `expi`) or is a LAMBDA form of  $n$  arguments. MAPERROR[TRUE] - if FALSE will cause all of the mapping functions to (1) stop when they finish going down the shortest `expi` if not all of the `expi` are of the same length and (2) apply `fn` to [`exp1`, `exp2`,...] if the `expi` are not all the same type of object. If MAPERROR is TRUE then an error message will be given in the above two instances. One of the uses of this function is to MAP a function (e.g. PARTFRAC) onto each term of a very large expression where it ordinarily wouldn't be possible to use the function on the entire expression due to an exhaustion of list storage space in the course of the computation.

```
(C1) MAP(F,X+A*Y+B*Z);
(D1)          F(B Z) + F(A Y) + F(X)
(C2) MAP(LAMBDA([U],PARTFRAC(U,X)),X+1/(X^3+4*X^2+5*X+2));
(D2)          1      1      1
          ----- - ----- + ----- + X
          X + 2   X + 1          2
                                (X + 1)
(C3) MAP(RATSIMP, X/(X^2+X)+(Y^2+Y)/Y);
(D3)          1
          Y + ----- + 1
          X + 1
(C4) MAP("=", [A,B], [-0.5,3]);
(D4)          [A = - 0.5, B = 3]
```

**MAPATOM** (*expr*)

Function

is TRUE if and only if `expr` is treated by the MAPping routines as an "atom", a unit. "Mapatoms" are atoms, numbers (including rational numbers), and subscripted variables.

**MAPERROR**

Variable

default: [TRUE] - if FALSE will cause all of the mapping functions, for example

```
MAP(fn,exp1,exp2,...)
```

to (1) stop when they finish going down the shortest `expi` if not all of the `expi` are of the same length and (2) apply `fn` to [`exp1`, `exp2`,...] if the `expi` are not all the same type of object. If MAPERROR is TRUE then an error message will be given in the above two instances.

**MAPLIST** (*fn, exp1, exp2, ...*)

Function

yields a list of the applications of `fn` to the parts of the `expi`. This differs from MAP(`fn,exp1,exp2,...`) which returns an expression with the same main operator as `expi` has (except for simplifications and the case where MAP does an APPLY). `Fn` is of the same form as in MAP.

**PREDERROR**

Variable

default: [TRUE] - If TRUE, an error message is signalled whenever the predicate of an IF statement or an IS function fails to evaluate to either TRUE or FALSE.

If FALSE, UNKNOWN is returned instead in this case. The PREDERROR:FALSE mode is not supported in translated code.

**RETURN** (*value*) Function  
 may be used to exit explicitly from a BLOCK, bringing its argument. Do DESCRIBE(BLOCK); for more information.

**SCANMAP** (*function,exp*) Function  
 recursively applies function to exp, in a "top down" manner. This is most useful when "complete" factorization is desired, for example:

(C1) EXP: (A<sup>2</sup>+2\*A+1)\*Y + X<sup>2</sup>  
 (C2) SCANMAP(FACTOR,EXP);

(D2)  $(A + 1)^2 Y + X^2$

Note the way in which SCANMAP applies the given function FACTOR to the constituent subexpressions of exp; if another form of exp is presented to SCANMAP then the result may be different. Thus, D2 is not recovered when SCANMAP is applied to the expanded form of exp:

(C3) SCANMAP(FACTOR,EXPAND(EXP));

(D3)  $A^2 Y + 2 A Y + Y + X^2$

Here is another example of the way in which SCANMAP recursively applies a given function to all subexpressions, including exponents:

(C4) EXPR : U\*V<sup>(A\*X+B)</sup> + C

(C5) SCANMAP('F, EXPR);

(D5) F(F(F(U) F(F(V) F(F(F(A) F(X)) + F(B))))) + F(C)

SCANMAP(function,expression,BOTTOMUP) applies function to exp in a "bottom-up" manner. E.g., for undefined F,

```
SCANMAP(F,A*X+B) ->
  F(A*X+B) -> F(F(A*X)+F(B)) -> F(F(F(A)*F(X))+F(B))
SCANMAP(F,A*X+B,BOTTOMUP) -> F(A)*F(X)+F(B)
-> F(F(A)*F(X))+F(B) ->
  F(F(F(A)*F(X))+F(B))
```

In this case, you get the same answer both ways.

**THROW** (*exp*) Function  
 evaluates exp and throws the value back to the most recent CATCH. THROW is used with CATCH as a structured nonlocal exit mechanism.





The actual file /tmp/joe.mac is the following:

```
foo(x,y):=(
  x:x+2,
  y:y+2,
  x:joe(y),
  x+y);

joe(y):=block([u:y^2],
  u:u+3,
  u:u^2,
  u);
```

If you are running in Gnu Emacs then if you are looking at the file joe.mac, you may set a break point at a certain line of that file by typing **C-x space**. This figures out which function your cursor is in, and then it sees which line of that function you are on. If you are on say line 2 of joe, then it will insert in the other window **:br joe 2** the command to break joe at its second line. To have this enabled you must have maxima-mode.el on in the window in which the file joe.mac is visiting. There are additional commands available in that file window, such as evaluating the function into the maxima, by typing **Alt-Control-x**

## 39.2 Keyword Commands

Break commands start with **:'**. Thus to evaluate a lisp form you may type **:lisp** followed by the argument which is the form to be evaluated.

```
(C3) :lisp (+ 2 3)
5
```

The number of arguments taken depends on the particular command. Also you need not type the whole command, just enough to be unique among the keyword commands. Thus **:br** would suffice for **:break**. The current commands are:

<b>:break</b>	Set a breakpoint in the specified FUNCTION at the specified LINE offset from the beginning of the function. If FUNCTION is given as a string, then it is presumed to be a FILE and LINE is the offset from the beginning of the file.
<b>:bt</b>	Undocumented
<b>:continue</b>	Continue the computation.
<b>:delete</b>	Delete all breakpoints, or if arguments are supplied delete the specified breakpoints
<b>:disable</b>	Disable the specified breakpoints, or all if none are specified
<b>:enable</b>	Enable the specified breakpoints, or all if none are specified
<b>:frame</b>	With an argument print the selected stack frame. Otherwise the current frame.
<b>:help</b>	Print help on a break command or with no arguments on all break commands
<b>:info</b>	Undocumented

**:lisp** Evaluate the lisp form following on the line  
**:lisp-quiet** Evaluate its arg as a lisp form without printing a prompt.  
**:next** Like **:step**, except that subroutine calls are stepped over  
**:quit** Quit this level  
**:resume** Continue the computation.  
**:step** Step program until it reaches a new source line  
**:top** Throw to top level

### 39.3 Definitions for Debugging

**REFCHECK** Variable  
 default: [FALSE] - if TRUE causes a message to be printed each time a bound variable is used for the first time in a computation.

**RETRACE** () Function  
 This function is no longer used with the new TRACE package.

**SETCHECK** Variable  
 default: [FALSE] - if set to a list of variables (which can be subscripted) will cause a printout whenever the variables, or subscripted occurrences of them, are bound (with **:** or **::** or function argument binding). The printout consists of the variable and the value it is bound to. SETCHECK may be set to ALL or TRUE thereby including all variables. Note: No printout is generated when a SETCHECKed variable is set to itself, e.g. X:'X.

**SETCHECKBREAK** Variable  
 default: [FALSE] - if set to TRUE will cause a (MACSYMA-BREAK) to occur whenever the variables on the SETCHECK list are bound. The break occurs before the binding is done. At this point, SETVAL holds the value to which the variable is about to be set. Hence, one may change this value by resetting SETVAL.

**SETVAL** Variable  
 - holds the value to which a variable is about to be set when a SETCHECKBREAK occurs. Hence, one may change this value by resetting SETVAL. (See SETCHECK-BREAK).

**TIMER** (F) Function  
 will put a timer-wrapper on the function F, within the TRACE package, i.e. it will print out the time spent in computing F.

**TIMER\_DEVALUE** Variable  
 default: [FALSE] - when set to TRUE then the time charged against a function is the time spent dynamically inside the function devalued by the time spent inside other TIMED functions.



**TIMER\_INFO** (*F*) Function  
 will print the information on timing which is stored also as GET('F,'CALLS);  
 GET('F,'RUNTIME); and GET('F,'GCTIME); . This is a TRACE package  
 function.

**TRACE** (*name1, name2, ...*) Function  
 gives a trace printout whenever the functions mentioned are called. TRACE() prints  
 a list of the functions currently under TRACE. On MC see MACDOC;TRACE US-  
 AGE for more information. Also, DEMO("trace.dem"); . To remove tracing, see  
 UNTRACE.

**TRACE\_OPTIONS** (*F,option1,option2,...*) Function  
 gives the function F the options indicated. An option is either a keyword or an  
 expression. The possible Keywords are: Keyword Meaning of return value ———  
 —————- NOPRINT If TRUE do no printing. BREAK If TRUE give  
 a breakpoint. LISP\_PRINT If TRUE use lisp printing. INFO Extra info to print.  
 ERRORCATCH If TRUE errors are caught. A keyword means that the option is  
 in effect. Using a keyword as an expression, e.g. NOPRINT(predicate\_function)  
 means to apply the predicate\_function (which is user-defined) to some arguments to  
 determine if the option is in effect. The argument list to this predicate\_function is  
 always [LEVEL, DIRECTION, FUNCTION, ITEM] where LEVEL is the recursion  
 level for the function. DIRECTION is either ENTER or EXIT. FUNCTION is the  
 name of the function. ITEM is either the argument list or the return value. On MC  
 see DEMO("trace.dem"); for more details.

**UNTRACE** (*name1, ...*) Function  
 removes tracing invoked by the TRACE function. UNTRACE() removes tracing from  
 all functions.

## 40 Indices



## Appendix A Function and Variable Index

<b>%</b>		AIRY . . . . .	102
% . . . . .	55	ALARMCLOCK . . . . .	211
%% . . . . .	55	ALGEBRAIC . . . . .	77
%EDISPFLAG . . . . .	55	ALGEPSILON . . . . .	69
%RNUM_LIST . . . . .	129	ALGEXACT . . . . .	129
%TH . . . . .	11, 55	ALGSYS . . . . .	129
		ALIAS . . . . .	11
		ALIASES . . . . .	215
<b>?</b>		ALL_DOTSIMP_DENOMS . . . . .	169
?ROUND . . . . .	70	ALLBUT . . . . .	21
?TRUNCATE . . . . .	70	ALLOC . . . . .	211
		ALLROOTS . . . . .	130
		ALLSYM . . . . .	215
<b>[</b>		ALPHABETIC . . . . .	215
[index] (expr) . . . . .	101	ANTID . . . . .	113
		ANTIDIFF . . . . .	113
		ANTISYMMETRIC . . . . .	21
<b>"</b>		APPEND . . . . .	227
"!!" . . . . .	20	APPENDFILE . . . . .	56
"!" . . . . .	19	APPLY . . . . .	235
"#" . . . . .	20	APPLY_NOUNS . . . . .	41
">>" . . . . .	11	APPLY1 . . . . .	221
">" . . . . .	11	APPLY2 . . . . .	221
". " . . . . .	20	APPLYB1 . . . . .	221
": :=" . . . . .	20	APROPOS . . . . .	215
": ." . . . . .	20	ARGS . . . . .	215
": =" . . . . .	20	ARRAY . . . . .	151
": " . . . . .	20	ARRAYAPPLY . . . . .	151
"=" . . . . .	20	ARRAYINFO . . . . .	151
"?" . . . . .	56	ARRAYMAKE . . . . .	151
"[" . . . . .	165	ARRAYS . . . . .	151
		ASEC . . . . .	95
<b>A</b>		ASECH . . . . .	95
ABSBOXCHAR . . . . .	56	ASIN . . . . .	95
ACOS . . . . .	95	ASINH . . . . .	95
ACOSH . . . . .	95	ASKEXP . . . . .	41
ACOT . . . . .	95	ASKINTEGER . . . . .	41
ACOTH . . . . .	95	ASKSIGN . . . . .	41
ACSC . . . . .	95	ASSOC_LEGENDRE_P . . . . .	107
ACSCH . . . . .	95	ASSOC_LEGENDRE_Q . . . . .	107
ACTIVATE . . . . .	73	ASSUME . . . . .	73
ACTIVECONTEXTS . . . . .	73	ASSUME_POS . . . . .	73
ADDCOL . . . . .	155	ASSUME_POS_PRED . . . . .	74
ADDITIVE . . . . .	20	ASSUMESCALAR . . . . .	73
ADDRROW . . . . .	155	ASYMP . . . . .	102
ADJOINT . . . . .	155	ASYMPA . . . . .	102
		AT . . . . .	31
		ATAN . . . . .	96

ATAN2	96
ATANH	96
ATOM	227
ATOMGRAD	113
ATRIG1	96
ATVALUE	113
AUGCOEFMATRIX	155

## B

BACKSUBST	130
BACKTRACE	245
BACKUP	56
BASHINDICES	151
BATCH	56
BATCHKILL	56
BATCHLOAD	56
BATCON	56
BATCOUNT	57
BERLEFACT	77
BERN	185
BERNPOLY	185
BESSEL	102
BETA	102
BEZOUT	78
BFFAC	69
BFLOAT	69
BFLOATP	69
BFPSI	69
BFTORAT	69
BFTRUNC	69
BFZETA	185
BGZETA	185
BHZETA	185
BINDTEST	235
BINOMIAL	185
BLOCK	235
BOTHCASES	57
BOTHCOEF	78
BOX	31
BOXCHAR	31
BREAK	236
BREAKUP	130
BUG	211
BUILDQ	236
BUILDQ([varlist], expression);	232
BURN	185
BZETA	185

## C

CABS	21
CANFORM	171
CANTEN	171
CARG	171
CARTAN	113
CATCH	236
CAUCHYSUM	179
CBFAC	69
CF	186
CFDISREP	186
CFEXPAND	186
CFLENGTH	186
CGAMMA	186
CGAMMA2	187
CHANGE_FILEDEFAULTS	57
CHANGEVAR	119
CHARPOLY	156
CHEBYSHEV_T	108
CHEBYSHEV_U	108
CHECK_OVERLAPS	168
CHR1	176
CHR2	176
CHRISTOF	176
CLEARSCREEN	212
CLOSEFILE	57
CLOSEPS	52
COEFF	78
COEFMATRIX	156
COL	156
COLLAPSE	58
COLUMNVECTOR	156
COMBINE	78
COMMUTATIVE	21
COMP2PUI	191
COMPFILE	236
COMPGRIND	236
COMPILE	236
COMPILE_FILE	243
COMPILE_LISP_FILE	237
CONCAT	58
CONJUGATE	156
CONS	227
CONSTANT	31
CONSTANTP	31
CONT2PART	191
CONTENT	78

CONTEXT	74	DERIVLIST	115
CONTEXTS	74	DERIVSUBST	115
CONTINUE	212	DESCRIBE	10
CONTRACT	31, 191	DESOLVE	137
COPYLIST	227	DETERMINANT	156
COPYMATRIX	156	DETOUT	156
COS	96	DIAGMATRIX	156
COSH	96	DIAGMETRIC	177
COT	96	DIFF	115
COTH	96	DIM	177
COUNTER	171	DIMENSION	130
COVDIFF	176	DIREC	58
CREATE_LIST	169	DIRECT	192
CSC	96	DISKFREE	212
CSCH	96	DISOLATE	33
CURRENT_LET_RULE_PACKAGE	221	DISP	58
CURSORDISP	58	DISPCON	58
CURVATURE	176	DISPFLAG	130
<b>D</b>			
DBLINT	120	DISPFUN	237
DDT	212	DISPLAY	59
DEACTIVATE	74	DISPLAY_FORMAT_INTERNAL	59
DEBUG	11	DISPLAY2D	59
DEBUGMODE	11	DISPRULE	222
DEBUGPRINTMODE	11	DISPTERMS	59
DECLARE	32	DISTRIB	33
DECLARE_TRANSLATED	243	DIVIDE	79
DECLARE_WEIGHT	167	DIVSUM	187
DEFAULT_LET_RULE_PACKAGE	222	DO	245
DEFCON	172	DOALLMXOPS	156
DEFINE	237	DOMAIN	41
DEFINE_VARIABLE	237	DOMXEXPT	157
DEFINT	120	DOMXMXOPS	157
DEFMATCH	222	DOMXNCTIMES	157
DEFRULE	222	DONTFACTOR	157
DEFTAYLOR	179	DOSCMXOPS	157
DELETE	227	DOSCMXPLUS	157
DELFILE	212	DOTONSCSIMP	157
DELTA	114	DOTOSIMP	157
DEMO	9	DOT1SIMP	157
DEMOIVRE	41	DOTASSOC	157
DENOM	78	DOTCONSTRULES	157
DEPENDENCIES	114	DOTDISTRIB	158
DEPENDS	114	DOTEXPTSIMP	158
DERIVABBREV	115	DOTIDENT	158
DERIVDEGREE	115	DOTSCRULES	158
		DOTSIMP	167
		DPART	33

DSCALAR .....	116	EXPT .....	60
DSKALL .....	59	EXPTDISPFLAG .....	60
DUMMY .....	216	EXPTISOLATE .....	34
<b>E</b>		EXPTSUBST .....	34
E .....	91	EXTRACT_LINEAR_EQUATIONS .....	168
ECHELON .....	158	EZGCD .....	79
EIGENVALUES .....	158	<b>F</b>	
EIGENVECTORS .....	159	FACEXPAND .....	79
EINSTEIN .....	177	FACTCOMB .....	79
ELE2COMP .....	193	FACTLIM .....	43
ELE2POLYNOME .....	193	FACTOR .....	80
ELE2PUI .....	194	FACTORFLAG .....	80
ELEM .....	194	FACTORIAL .....	187
ELIMINATE .....	79	FACTOROUT .....	80
EMATRIX .....	159	FACTORSUM .....	80
ENDCONS .....	228	FACTS .....	74
ENTERMATRIX .....	159	FALSE .....	91
ENTIER .....	21	FASSAVE .....	60
EQUAL .....	21	FAST_CENTRAL_ELEMENTS .....	167
ERF .....	121	FAST_LINSOLVE .....	167
ERFFLAG .....	121	FASTTIMES .....	81
ERRCATCH .....	248	FEATURE .....	212
ERREXP .....	248	FEATUREP .....	212
ERRINTSCE .....	121	FEATURES .....	75
ERROR .....	248	FFT .....	144
ERROR_SIZE .....	60	FIB .....	187
ERROR_SYMS .....	60	FIBTOPHI .....	187
ERRORFUN .....	248	FILE_SEARCH .....	61
ERRORMSG .....	248	FILE_STRING_PRINT .....	62
EULER .....	187	FILE_TYPE .....	62
EV .....	12	FILEDEFAULTS .....	61
EVAL .....	22	FILENAME .....	61
EVENP .....	22	FILENAME_MERGE .....	61
EVFLAG .....	14	FILENUM .....	61
EVFUN .....	14	FILLARRAY .....	152
EXAMPLE .....	10	FIRST .....	228
EXP .....	33	FIX .....	22
EXPAND .....	42	FLOAT .....	70
EXPANDWRT .....	42	FLOAT2BF .....	70
EXPANDWRT_DENOM .....	42	FLOATDEFUNK .....	70
EXPANDWRT_FACTORED .....	42	FLOATNUMP .....	70
EXPLOSE .....	195	FLUSH .....	172
EXPON .....	42	FLUSHD .....	172
EXPONENTIALIZE .....	43	FLUSHND .....	172
EXPOP .....	43	FOR .....	249
EXPRESS .....	117	FORGET .....	75

FORTINDENT .....	144
FORTMX .....	144
FORTTRAN .....	144
FORTSPACES .....	144
FPPREC .....	70
FPPRINTPREC .....	70
FREEOF .....	34
FULLMAP .....	22
FULLMAPL .....	22
FULLRATSIMP .....	81
FULLRATSUBST .....	81
FUNCSOLVE .....	130
FUNCTIONS .....	238
FUNDEF .....	238
FUNMAKE .....	238

**G**

GAMMA .....	102
GAMMALIM .....	102
GAUSS .....	149
GCD .....	81
GCFACTOR .....	82
GEN_LAGUERRE .....	108
GENDIFF .....	117
GENFACT .....	34
GENINDEX .....	216
GENMATRIX .....	160
GENSUMNUM .....	216
GET .....	228
GETCHAR .....	152
GFACTOR .....	82
GFACTORSUM .....	82
GLOBALSOLVE .....	131
GO .....	249
GRADEF .....	117
GRADEFS .....	117
GRAMSCHMIDT .....	160
GRIND .....	62
GROBNER_BASIS .....	167

**H**

HACH .....	160
HALFANGLES .....	96
HERMITE .....	108
HIPOW .....	82
HORNER .....	145

**I**

IBASE .....	62
IC1 .....	137
IDENT .....	160
IEQN .....	131
IEQNPRINT .....	131
IF .....	249
IFT .....	145
ILT .....	121
IMAGPART .....	34
IN_NETMATH .....	49
INCHAR .....	62
INDICES .....	34
INF .....	216
INFEVAL .....	14
INFINITY .....	216
INFIX .....	34
INFLAG .....	34
INFOLISTS .....	216
INNERPRODUCT .....	160
INPART .....	35
INRT .....	187
INTEGERP .....	216
INTEGRATE .....	122
INTEGRATE_USE_ROOTSOF .....	123
INTEGRATION_CONSTANT_COUNTER .....	123
INTERPOLATE .....	145
INTFACLIM .....	82
INTOPOIS .....	102
INTOSUM .....	43
INTPOLABS .....	146
INTPOLERROR .....	146
INTPOLREL .....	146
INTSCE .....	124
INVERT .....	161
IS .....	22
ISOLATE .....	35
ISOLATE_WRT_TIMES .....	35
ISQRT .....	22

**J**

JACOBI .....	187
JACOBI_P .....	108



**K**

KDELTA	172
KEEPFLOAT	82
KILL	14
KILLCONTEXT	75
KOSTKA	195

**L**

LABELS	15
LAGUERRE	108
LAPLACE	117
LASSOCIATIVE	43
LAST	228
LASTTIME	15
LC	172
LCM	187
LDEFINT	125
LDISP	62
LDISPLAY	62
LEGENDRE_P	109
LEGENDRE_Q	109
LENGTH	228
LET	223
LET_RULE_PACKAGES	224
LETRAT	224
LETRULES	224
LETSIMP	224
LGTREILLIS	195
LHOSPITALIM	111
LHS	131
LIMIT	111
LINEAR	43
LINECHAR	62
LINEDISP	62
LINEL	62
LINENUM	15
LINSOLVE	131
LINSOLVE_PARAMS	132
LINSOLVEWARN	132
LISPDEBUGMODE	249
LIST_NC_MONOMIALS	168
LISTARITH	229
LISTARRAY	152
LISTCONSTVARS	35
LISTDUMMYVARS	35
LISTOFVARS	36

LISTP	229
LMXCHAR	161
LOAD	63
LOADFILE	64
LOADPRINT	64
LOCAL	238
LOG	93
LOGABS	93
LOGARC	93
LOGCONCOEFFP	93
LOGCONTRACT	94
LOGEXPAND	94
LOGNEGINT	94
LOGNUMBER	94
LOGSIMP	94
LOPOW	36
LORENTZ	173
LPART	36
LRATSUBST	82
LRICCOM	177
LSUM	40
LTREILLIS	195

**M**

M1PBRANCH	217
MACROEXPANSION	238
MAINVAR	43
MAKE_ARRAY	152
MAKEBOX	173
MAKEFACT	102
MAKEGAMMA	102
MAKELIST	229
MAP	249
MAPATOM	250
MAPERROR	250
MAPLIST	250
MATCHDECLARE	224
MATCHFIX	224
MATRIX	161
MATRIX_ELEMENT_ADD	161
MATRIX_ELEMENT_MULT	162
MATRIX_ELEMENT_TRANSPOSE	162
MATRIXMAP	161
MATRIXP	161
MATTRACE	162
MAX	23
MAXAPPLYDEPTH	44

MAXAPPLYHEIGHT	44	NONSCALAR	162
MAXNEGEX	44	NONSCALARP	162
MAXPOSEX	44	NOSTRING	64
MAXPRIME	188	NOUN	44
MAXTAYORDER	179	NOUNDISP	45
MEMBER	229	NOUNIFY	37
METRIC	173	NOUNS	45
MIN	23	NROOTS	132
MINF	91	NTERMS	37
MINFACTORIAL	188	NTERMSG	173
MINOR	162	NTERMSRCI	173
MOD	23	NTHROOT	133
MODE_CHECK_ERRORP	238	NUM	83
MODE_CHECK_WARNP	238	NUMBERP	217
MODE_CHECKP	238	NUMER	45
MODE_DECLARE	238	NUMERVAL	45
MODE_IDENTITY	239	NUMFACTOR	103
MODULUS	82	NUSUM	180
MON2SCHUR	196	NZETA	173
MONO	168		
MONOMIAL_DIMENSIONS	168		
MOTION	177		
MULTI_ELEM	196	<b>O</b>	
MULTI_ORBIT	196	OBASE	64
MULTI_PUI	197	ODDP	23
MULTINOMIAL	197	ODE	138
MULTIPLICATIVE	44	ODE2	138
MULTIPLICITIES	132	OMEGA	177
MULTSYM	197	OPENPLOT_CURVES	49
MULTTHRU	36	OPPROPERTIES	45
MYOPTIONS	15	OPSUBST	45
		OPTIMIZE	37
<b>N</b>		OPTIMPREFIX	37
NC_DEGREE	167	OPTIONSET	15
NCEXPT	162	ORBIT	198
NCHARPOLY	162	ORDERGREAT	37
NEGDISTRIB	44	ORDERGREATP	37
NEGSUMDISPFLAG	44	ORDERLESS	38
NEW-DISREP	168	ORDERLESSP	38
NEWCONTEXT	75	OUTATIVE	45
NEWDET	162	OUTCHAR	64
NEWFAC	83	OUTOFPOIS	103
NEWTON	146		
NICEINDICES	179		
NICEINDICESPREF	180		
NOEVAL	44		
NOLABELS	15		

**P**

PACKAGEFILE	64
PADE	180
PARSEWINDOW	64
PART	38
PART2CONT	198
PARTFRAC	188
PARTITION	38
PARTPOL	198
PARTSWITCH	38
PCOEFF	168
PERMANENT	163
PERMUT	198
PFEFORMAT	64
PI	91
PICKAPART	38
PIECE	39
PLAYBACK	16
PLOG	94
PLOT_OPTIONS	50
PLOT2D	49
PLOT2D_PS	52
PLOT3D	51
POISDIFF	103
POISEXPT	103
POISINT	103
POISLIM	103
POISMAP	103
POISPLUS	103
POISSIMP	103
POISSON	104
POISSUBST	104
POISTIMES	104
POISTRIM	104
POLARFORM	94
POLARTORECT	147
POLYNOME2ELE	198
POSFUN	45
POTENTIAL	125
POWERDISP	181
POWERS	39
POWERSERIES	181
PRED	23
PREDERROR	250
PRIME	188
PRIMEP	188
PRINT	65
PRINTPOIS	104
PRINTPROPS	16
PRODHACK	45
PRODRAC	199
PRODUCT	39
PROGRAMMODE	133
PROMPT	16
PROPERTIES	217
PROPS	217
PROPVARS	217
PSCOM	53
PSDRAW_CURVE	52
PSEXPAND	181
PSI	104
PUI	199
PUI_DIRECT	201
PUI2COMP	199
PUI2ELE	200
PUI2POLYNOME	200
PUIREduc	201
PUT	217
<b>Q</b>	
QPUT	217
QQ	125
QUANC8	125
QUIT	16
QUNIT	188
QUOTIENT	83
<b>R</b>	
RADCAN	46
RADEXPAND	46
RADPRODEXPAND	46
RADSUBSTFLAG	46
RAISERiemann	173
RANDOM	23
RANK	163
RASSOCIATIVE	46
RAT	83
RATALGDENOM	84
RATCOEF	84
RATDENOM	84
RATDENOMDIVIDE	84
RATDIFF	84
RATDISREP	85

RATEINSTEIN	173	RESOLVANTE_PRODUI_T_SYM	206
RATEPSILON	85	RESOLVANTE_UNITAIRE	207
RATEXPAND	85	RESOLVANTE_VIERER	207
RATFAC	86	REST	229
RATMX	163	RESTORE	17
RATNUMER	86	RESULTANT	88
RATNUMP	86	RETURN	251
RATP	86	REVEAL	65
RATPRINT	86	REVERSE	229
RATRIEMAN	173	REVERT	182
RATRIEMANN	174	RHS	133
RATSIMP	86	RICCICOM	174
RATSIMPEXPONS	87	RIEMANN	178
RATSUBST	87	RINVARIANT	174
RATVARS	87	RISCH	126
RATWEIGHT	87	RMXCHAR	65
RATWEIGHTS	88	RNCOMBINE	218
RATWEYL	88	ROMBERG	126
RATWTLVL	88	ROMBERGABS	127
READ	65	ROMBERGIT	128
READONLY	65	ROMBERGMIN	128
REALONLY	133	ROMBERGTOL	128
REALPART	39	ROOM	213
REALROOTS	133	ROOTSCONMODE	133
REARRAY	152	ROOTSCONTRACT	133
RECTFORM	39	ROOTSEPSILON	134
RECTTOPOLAR	147	ROW	163
REFCHECK	255		
REM	217	<b>S</b>	
REMAINDER	88	SAVE	66
REMARRAY	152	SAVEDEF	66
REMBOX	39	SAVEFACTORS	89
REMCN	174	SCALARMATRIXP	163
REMFUNCTIION	16	SCALARP	218
REMLET	225	SCALEFACTORS	218
REMOVE	217	SCANMAP	251
REMRULE	225	SCHUR2COMP	207
REMRACE	255	SCONCAT	58
REMVAlUE	218	SCSIMP	46
RENAME	218	SCURVATURE	174
RESET	16	SEC	96
RESIDUE	125	SECH	96
RESOLVANTE	202	SET_PLOT_OPTION	52
RESOLVANTE_ALTERNEE1	205	SET_UP_DOT_SIMPLIFICATIONS	167
RESOLVANTE_BIPARTITE	205	SETCHECK	255
RESOLVANTE_DIEDRALE	205	SETCHECKBREAK	255
RESOLVANTE_KLEIN	206	SETELMX	163
RESOLVANTE_KLEIN3	206		

SETUP	174	SUM	40
SETUP_AUTOLOAD	219	SUMCONTRACT	47
SETVAL	255	SUMEXPAND	47
SHOW	66	SUMHACK	47
SHOWRATVARS	66	SUMSPLITFACT	47
SHOWTIME	17	SUPCONTEXT	75
SIGN	23	SYMBOLP	26
SIGNUM	23	SYMMETRIC	47
SIMILARITYTRANSFORM	163	SYSTEM(command)	67
SIMP	46		
SIMPSUM	47	<b>T</b>	
SIN	96	TAN	97
SINH	97	TANH	97
SOLVE	134	TAYLOR	182
SOLVE_INCONSISTENT_ERROR	135	TAYLOR_LOGEXPAND	183
SOLVEDECOMPOSES	135	TAYLOR_ORDER_COEFFICIENTS	183
SOLVEEXPLICIT	135	TAYLOR_SIMPLIFIER	183
SOLVEFACTORS	135	TAYLOR_TRUNCATE_POLYNOMIALS	183
SOLVENULLWARN	135	TAYLORDEPTH	182
SOLVERADCAN	135	TAYLORINFO	182
SOLVETRIGWARN	135	TAYLORP	183
SOMRAC	207	TAYTORAT	183
SORT	23	TCL_OUTPUT	65
SPARSE	164	TCONTRACT	207
SPHERICAL_BESSEL_J	109	TELLRAT	89
SPHERICAL_BESSEL_Y	109	TELLSIMP	226
SPHERICAL_HANKEL1	109	TELLSIMPATER	226
SPHERICAL_HANKEL2	109	TEX	67
SPHERICAL_HARMONIC	110	TEX(expr,filename)	67
SPLICE	232	TEX(label,filename)	67
SPRINT	65	THROW	251
SQFR	89	TIME	213
SQRT	24	TIMER	255
SQRDISPFLAG	24	TIMER_DEVALUE	255
SRRAT	182	TIMER_INFO	256
SSTATUS	17	TLDEFINT	128
STARDISP	66	TLIMIT	111
STATUS	213	TLIMSWITCH	111
STRING	66	TO_LISP	17
STRINGOUT	66	TOBREAK	17
SUBLIS	24	TODD_COXETER	209
SUBLIS_APPLY_LAMBDA	24	TOPELVEL	17
SUBLIST	24	TOTALDISREP	89
SUBMATRIX	164	TOTIENT	188
SUBST	24	TPARTPOL	208
SUBSTINPART	25	TR_ARRAY_AS_REF	241
SUBSTPART	26	TR_BOUND_FUNCTION_APPLY	241
SUBVARP	26		

TR_FILE_TTY_MESSAGESP	241
TR_FLOAT_CAN_BRANCH_COMPLEX	241
TR_FUNCTION_CALL_DEFAULT	241
TR_GEN_TAGS	241
TR_NUMER	242
TR_OPTIMIZE_MAX_LOOP	242
TR_OUTPUT_FILE_DEFAULT	242
TR_PREDICATE_BRAIN_DAMAGE	242
TR_SEMICOPILE	242
TR_STATE_VARS	242
TR_TRUE_NAME_OF_FILE_BEING_TRANSLATED	242
TR_VERSION	242
TR_WARN_BAD_FUNCTION_CALLS	242
TR_WARN_FEXPR	243
TR_WARN_MEVAL	243
TR_WARN_MODE	243
TR_WARN_UNDECLARED	243
TR_WARN_UNDEFINED_VARIABLE	243
TR_WARNINGS_GET	242
TR_WINDY	243
TRACE	256
TRACE_OPTIONS	256
TRANSBIND	239
TRANSCOMPIL	239
TRANSFORM	178
TRANSLATE	240
TRANSLATE_FILE	240
TRANSPOSE	164
TRANSRUN	241
TREILLIS	208
TREINAT	208
TRIANGULARIZE	164
TRIGEXPAND	97
TRIGEXPANDPLUS	97
TRIGEXPANDTIMES	97
TRIGINVERSES	97
TRIGRAT	98
TRIGREDUCE	98
TRIGSIGN	98
TRIGSIMP	98
TRUE	91
TRUNC	183
TSETUP	178
TTRANSFORM	178
TTYINTFUN	17
TTYINTNUM	18
TTYOFF	67
<b>U</b>	
ULTRASPHERICAL	110
UNDECLAREDWARN	243
UNDIFF	118
UNITEIGENVECTORS	164
UNITVECTOR	164
UNKNOWN	47
UNORDER	26
UNSUM	184
UNTELLRAT	90
UNTRACE	256
USE_FAST_ARRAYS	152
<b>V</b>	
VALUES	18
VECT_CROSS	165
VECTORPOTENTIAL	26
VECTORSIMP	164
VERB	40
VERBIFY	40
VERBOSE	184
<b>W</b>	
WEYL	174
WITH_STDOUT(file,stmt1,stmt2,...)	68
WRITEFILE	68
<b>X</b>	
xgraph_curves(list)	49
XTHRU	27
<b>Z</b>	
ZEROBERN	189
ZEROEQUIV	27
ZEROMATRIX	165
ZETA	189
ZETA%PI	189
ZRPOLY	135
ZSOLVE	136
ZUNDERFLOW	71



## Short Contents

.....	1
1 Introduction to MAXIMA .....	3
2 Help .....	7
3 Command Line .....	11
4 Operators .....	19
5 Expressions .....	29
6 Simplification .....	41
7 Plotting .....	49
8 Input and Output .....	55
9 Floating Point .....	69
10 Contexts .....	73
11 Polynomials .....	77
12 Constants .....	91
13 Logarithms .....	93
14 Trigonometric .....	95
15 Special Functions .....	101
16 Orthogonal Polynomials .....	105
17 Limits .....	111
18 Differentiation .....	113
19 Integration .....	119
20 Equations .....	129
21 Differential Equations .....	137
22 Numerical .....	141
23 Statistics .....	149
24 Arrays and Tables .....	151
25 Matrices and Linear Algebra .....	155
26 Affine .....	167
27 Tensor .....	171
28 Ctensor .....	175
29 Series .....	179
30 Number Theory .....	185
31 Symmetries .....	191
32 Groups .....	209
33 Runtime Environment .....	211
34 Miscellaneous Options .....	215



35	Rules and Patterns . . . . .	221
36	Lists . . . . .	227
37	Function Definition . . . . .	231
38	Program Flow . . . . .	245
39	Debugging . . . . .	253
40	Indices . . . . .	257
	Appendix A Function and Variable Index . . . . .	259

# Table of Contents

.....	<b>1</b>
<b>1 Introduction to MAXIMA.....</b>	<b>3</b>
<b>2 Help.....</b>	<b>7</b>
2.1 Introduction to Help.....	7
2.2 Lisp and Maxima.....	7
2.3 Garbage Collection.....	9
2.4 Documentation.....	9
2.5 Definitions for Help.....	9
<b>3 Command Line.....</b>	<b>11</b>
3.1 Introduction to Command Line.....	11
3.2 Definitions for Command Line.....	11
<b>4 Operators.....</b>	<b>19</b>
4.1 NARY.....	19
4.2 NOFIX.....	19
4.3 OPERATOR.....	19
4.4 POSTFIX.....	19
4.5 PREFIX.....	19
4.6 Definitions for Operators.....	19
<b>5 Expressions.....</b>	<b>29</b>
5.1 Introduction to Expressions.....	29
5.2 ASSIGNMENT.....	29
5.3 COMPLEX.....	29
5.4 INEQUALITY.....	30
5.5 SYNTAX.....	30
5.6 Definitions for Expressions.....	31
<b>6 Simplification.....</b>	<b>41</b>
6.1 Definitions for Simplification.....	41
<b>7 Plotting.....</b>	<b>49</b>
7.1 Definitions for Plotting.....	49

<b>8</b>	<b>Input and Output</b> .....	<b>55</b>
8.1	Introduction to Input and Output .....	55
8.2	FILES .....	55
8.3	PLAYBACK .....	55
8.4	Definitions for Input and Output .....	55
<b>9</b>	<b>Floating Point</b> .....	<b>69</b>
9.1	Definitions for Floating Point .....	69
<b>10</b>	<b>Contexts</b> .....	<b>73</b>
10.1	Definitions for Contexts .....	73
<b>11</b>	<b>Polynomials</b> .....	<b>77</b>
11.1	Introduction to Polynomials .....	77
11.2	Definitions for Polynomials .....	77
<b>12</b>	<b>Constants</b> .....	<b>91</b>
12.1	Definitions for Constants .....	91
<b>13</b>	<b>Logarithms</b> .....	<b>93</b>
13.1	Definitions for Logarithms .....	93
<b>14</b>	<b>Trigonometric</b> .....	<b>95</b>
14.1	Introduction to Trigonometric .....	95
14.2	Definitions for Trigonometric .....	95
<b>15</b>	<b>Special Functions</b> .....	<b>101</b>
15.1	Introduction to Special Functions .....	101
15.2	GAMALG .....	101
15.3	SPECINT .....	101
15.4	Definitions for Special Functions .....	102
<b>16</b>	<b>Orthogonal Polynomials</b> .....	<b>105</b>
16.1	Introduction to Orthogonal Polynomials .....	105
16.2	Definitions for Orthogonal Polynomials .....	107
<b>17</b>	<b>Limits</b> .....	<b>111</b>
17.1	Definitions for Limits .....	111
<b>18</b>	<b>Differentiation</b> .....	<b>113</b>
18.1	Definitions for Differentiation .....	113

<b>19</b>	<b>Integration</b> . . . . .	<b>119</b>
	19.1 Introduction to Integration . . . . .	119
	19.2 Definitions for Integration . . . . .	119
<b>20</b>	<b>Equations</b> . . . . .	<b>129</b>
	20.1 Definitions for Equations . . . . .	129
<b>21</b>	<b>Differential Equations</b> . . . . .	<b>137</b>
	21.1 Definitions for Differential Equations . . . . .	137
<b>22</b>	<b>Numerical</b> . . . . .	<b>141</b>
	22.1 Introduction to Numerical . . . . .	141
	22.2 DCADRE . . . . .	141
	22.3 ELLIPT . . . . .	143
	22.4 FOURIER . . . . .	143
	22.5 NDIFFQ . . . . .	143
	22.6 Definitions for Numerical . . . . .	144
<b>23</b>	<b>Statistics</b> . . . . .	<b>149</b>
	23.1 Definitions for Statistics . . . . .	149
<b>24</b>	<b>Arrays and Tables</b> . . . . .	<b>151</b>
	24.1 Definitions for Arrays and Tables . . . . .	151
<b>25</b>	<b>Matrices and Linear Algebra</b> . . . . .	<b>155</b>
	25.1 Introduction to Matrices and Linear Algebra . . . . .	155
	25.1.1 DOT . . . . .	155
	25.1.2 VECTORS . . . . .	155
	25.2 Definitions for Matrices and Linear Algebra . . . . .	155
<b>26</b>	<b>Affine</b> . . . . .	<b>167</b>
	26.1 Definitions for Affine . . . . .	167
<b>27</b>	<b>Tensor</b> . . . . .	<b>171</b>
	27.1 Introduction to Tensor . . . . .	171
	27.2 Definitions for Tensor . . . . .	171
<b>28</b>	<b>Ctensor</b> . . . . .	<b>175</b>
	28.1 Introduction to Ctensor . . . . .	175
	28.2 Definitions for Ctensor . . . . .	176
<b>29</b>	<b>Series</b> . . . . .	<b>179</b>
	29.1 Introduction to Series . . . . .	179
	29.2 Definitions for Series . . . . .	179

<b>30</b>	<b>Number Theory</b> .....	<b>185</b>
	30.1 Definitions for Number Theory .....	185
<b>31</b>	<b>Symmetries</b> .....	<b>191</b>
	31.1 Definitions for Symmetries .....	191
<b>32</b>	<b>Groups</b> .....	<b>209</b>
	32.1 Definitions for Groups .....	209
<b>33</b>	<b>Runtime Environment</b> .....	<b>211</b>
	33.1 Introduction for Runtime Environment .....	211
	33.2 INTERRUPTS .....	211
	33.3 Definitions for Runtime Environment .....	211
<b>34</b>	<b>Miscellaneous Options</b> .....	<b>215</b>
	34.1 Introduction to Miscellaneous Options .....	215
	34.2 SHARE .....	215
	34.3 Definitions for Miscellaneous Options .....	215
<b>35</b>	<b>Rules and Patterns</b> .....	<b>221</b>
	35.1 Introduction to Rules and Patterns .....	221
	35.2 Definitions for Rules and Patterns .....	221
<b>36</b>	<b>Lists</b> .....	<b>227</b>
	36.1 Introduction to Lists .....	227
	36.2 Definitions for Lists .....	227
<b>37</b>	<b>Function Definition</b> .....	<b>231</b>
	37.1 Introduction to Function Definition .....	231
	37.2 FUNCTION .....	231
	37.3 MACROS .....	232
	37.3.1 Semantics .....	232
	37.3.2 SIMPLIFICATION .....	232
	37.4 OPTIMIZATION .....	234
	37.5 Definitions for Function Definition .....	235
<b>38</b>	<b>Program Flow</b> .....	<b>245</b>
	38.1 Introduction to Program Flow .....	245
	38.2 Definitions for Program Flow .....	245
<b>39</b>	<b>Debugging</b> .....	<b>253</b>
	39.1 Source Level Debugging .....	253
	39.2 Keyword Commands .....	254
	39.3 Definitions for Debugging .....	255

40	Indices .....	257
Appendix A	Function and Variable Index ...	259

