Maxima Manual

Maxima is a computer algebra system, implemented in Lisp.

Maxima is derived from the Macsyma system, developed at MIT in the years 1968 through 1982 as part of Project MAC. MIT turned over a copy of the Macsyma source code to the Department of Energy in 1982; that version is now known as DOE Macsyma. A copy of DOE Macsyma was maintained by Professor William F. Schelter of the University of Texas from 1982 until his death in 2001. In 1998, Schelter obtained permission from the Department of Energy to release the DOE Macsyma source code under the GNU Public License, and in 2000 he initiated the Maxima project at SourceForge to maintain and develop DOE Macsyma, now called Maxima.

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:

```
[wfs@chromium]$ maxima
Maxima 5.9.1 http://maxima.sourceforge.net
Using Lisp CMU Common Lisp 19a
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1) factor(10!);
                            8  4  2
(%o1)                      2  3  5  7
(%i2) expand ((x + y)^6);
       6        5       2  4       3  3       4  2      5      6
(%o2) y  + 6 x y  + 15 x  y  + 20 x  y  + 15 x  y  + 6 x  y + x
(%i3) factor (x^6 - 1);
                                  2             2
(%o3)       (x - 1) (x + 1) (x  - x + 1) (x  + x + 1)
(%i4) quit();
[wfs@chromium]$
```

Maxima can search the info pages. Use the *describe* command to show all the commands and variables containing a string, and optionally their documentation. The question mark `?` is an abbreviation for `describe`:

```
(%i1) ? integ

 0: (maxima.info)Introduction to Elliptic Functions and Integrals.
 1: Definitions for Elliptic Integrals.
 2: Integration.
 3: Introduction to Integration.
 4: Definitions for Integration.
 5: askinteger :Definitions for Simplification.
 6: integerp :Definitions for Miscellaneous Options.
 7: integrate :Definitions for Integration.
 8: integrate_use_rootsof :Definitions for Integration.
 9: integration_constant_counter :Definitions for Integration.
Enter space-separated numbers, ‘all’ or ‘none’: 6 5

Info from file /usr/local/info/maxima.info:
 - Function: integerp (<expr>)
     Returns ‘true’ if <expr> is an integer, otherwise ‘false’.


 - Function: askinteger (expr, integer)
 - Function: askinteger (expr)
 - Function: askinteger (expr, even)
```

```
 - Function: askinteger (expr, odd)
     `askinteger (expr, integer)' attempts to determine from the
     `assume' database whether `expr' is an integer.  `askinteger' will
     ask the user if it cannot tell otherwise, and attempt to install
     the information in the database if possible.  `askinteger (expr)'
     is equivalent to `askinteger (expr, integer)'.

     `askinteger (expr, even)' and `askinteger (expr, odd)' likewise
     attempt to determine if `expr' is an even integer or odd integer,
     respectively.

(%o1)                                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:

```
(%i1) u: expand ((x + y)^6);
       6        5       2  4       3  3       4  2      5      6
(%o1) y  + 6 x y  + 15 x  y  + 20 x  y  + 15 x  y  + 6 x  y + x
(%i2) diff (u, x);
         5         4       2  3       3  2       4        5
(%o2) 6 y  + 30 x y  + 60 x  y  + 60 x  y  + 30 x  y + 6 x
(%i3) factor (%o2);
                                    5
(%o3)                       6 (y + x)
```

Maxima knows about complex numbers and numerical constants:

```
(%i1) cos(%pi);
(%o1)                                 - 1
(%i2) exp(%i*%pi);
(%o2)                                 - 1
```

Maxima can do differential and integral calculus:

```
(%i1) u: expand ((x + y)^6);
       6        5       2  4       3  3       4  2      5      6
(%o1) y  + 6 x y  + 15 x  y  + 20 x  y  + 15 x  y  + 6 x  y + x
(%i2) diff (%, x);
         5         4       2  3       3  2       4        5
(%o2) 6 y  + 30 x y  + 60 x  y  + 60 x  y  + 30 x  y + 6 x
(%i3) integrate (1/(1 + x^3), x);
                                     2 x - 1
                   2            atan(-------)
              log(x  - x + 1)        sqrt(3)    log(x + 1)
(%o3)       - --------------- + ------------- + ----------
                     6             sqrt(3)          3
```

Maxima can solve linear systems and cubic equations:

```
(%i1) linsolve ([3*x + 4*y = 7, 2*x + a*y = 13], [x, y]);
                           7 a - 52        25
(%o1)                 [x = --------, y = -------]
                           3 a - 8       3 a - 8
(%i2) solve (x^3 - 3*x^2 + 5*x = 15, x);
```

```
(%o2)        [x = - sqrt(5) %i, x = 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 `;`.

```
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
                3 sqrt(5) + 7      sqrt(5) + 3
(%o3) [[y = - -------------, x = -----------],
                      2                2

                                 3 sqrt(5) - 7        sqrt(5) - 3
                            [y = -------------, x = - -----------]]
                                       2                   2
```

Maxima can generate plots of one or more functions:

```
(%i1) eq_1: x^2 + 3*x*y + y^2 = 0$
(%i2) eq_2: 3*x + y = 1$
(%i3) solve ([eq_1, eq_2]);
                3 sqrt(5) + 7      sqrt(5) + 3
(%o3) [[y = - -------------, x = -----------],
                      2                2

                                 3 sqrt(5) - 7        sqrt(5) - 3
                            [y = -------------, x = - -----------]]
                                       2                   2
(%i4) kill(labels);
(%o0)                        done
(%i1) plot2d (sin(x)/x, [x, -20, 20]);
(%o1)
(%i2) plot2d ([atan(x), erf(x), tanh(x)], [x, -5, 5]);
(%o2)
(%i3) plot3d (sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2), [x, -12, 12], [y, -12, 12]);
(%o3)
```

2 Bug Detection and Reporting

2.1 Introduction to Bug Detection and Reporting

Like all large programs, Maxima contains both known and unknown bugs. This chapter describes the built-in facilities for running the Maxima test suite as well as reporting new bugs.

2.2 Definitions for Bug Detection and Reporting

run_testsuite () Function
run_testsuite (*boolean*) Function
run_testsuite (*boolean*, *boolean*) Function

Run the Maxima test suite. Tests producing the desired answer are considered "passes," as are tests that do not produce the desired answer, but are marked as known bugs.

`run_testsuite ()` displays only tests that do not pass.

`run_testsuite (true)` displays tests that are marked as known bugs, as well as failures.

`run_testsuite (true, true)` displays all tests.

`run_testsuite` changes the Maxima environment. Typically a test script executes `kill` to establish a known environment (namely one without user-defined functions and variables) and then defines functions and variables appropriate to the test.

`run_testsuite` returns `done`.

bug_report () Function

Prints out Maxima and Lisp version numbers, and gives a link to the Maxima project bug report web page. The version information is the same as reported by `build_info`.

When a bug is reported, it is helpful to copy the Maxima and Lisp version information into the bug report.

`bug_report` returns an empty string `""`.

build_info () Function

Prints out a summary of the parameters of the Maxima build.

`build_info` returns an empty string `""`.

3 Help

3.1 Introduction to Help

The primary on-line help function is `describe`, which is typically invoked by the question mark `?` at the interactive prompt. `? foo` (with a space between `?` and `foo`) is equivalent to `describe ("foo")`, where `foo` is the name or part of the name of a function or topic; `describe` then finds all documented items which contain the string `foo` in their titles. If there is more than one such item, Maxima asks the user to select an item or items to display.

```
(%i1) ? integ
 0: (maxima.info)Introduction to Elliptic Functions and Integrals.
 1: Definitions for Elliptic Integrals.
 2: Integration.
 3: Introduction to Integration.
 4: Definitions for Integration.
 5: askinteger :Definitions for Simplification.
 6: integerp :Definitions for Miscellaneous Options.
 7: integrate :Definitions for Integration.
 8: integrate_use_rootsof :Definitions for Integration.
 9: integration_constant_counter :Definitions for Integration.
Enter space-separated numbers, `all' or `none': 7 8

Info from file /use/local/maxima/doc/info/maxima.info:
 - Function: integrate (expr, var)
 - Function: integrate (expr, var, a, b)
     Attempts to symbolically compute the integral of `expr' with
     respect to `var'.  `integrate (expr, var)' is an indefinite
     integral, while `integrate (expr, var, a, b)' is a definite
     integral, [...]
```

In this example, items 7 and 8 were selected. All or none of the items could have been selected by entering `all` or `none`, which can be abbreviated `a` or `n`, respectively.

3.2 Lisp and Maxima

Maxima is written in Lisp, and it is easy to access Lisp functions and variables from Maxima and vice versa. Lisp and Maxima symbols are distinguished by a naming convention. A Lisp symbol which begins with a dollar sign `$` corresponds to a Maxima symbol without the dollar sign. A Maxima symbol which begins with a question mark `?` corresponds to a Lisp symbol without the question mark. For example, the Maxima symbol `foo` corresponds to the Lisp symbol `$foo`, while the Maxima symbol `?foo` corresponds to the Lisp symbol `foo`, Note that `?foo` is written without a space between `?` and `foo`; otherwise it might be mistaken for `describe ("foo")`.

Hyphen `-`, asterisk `*`, or other special characters in Lisp symbols must be escaped by backslash `\` where they appear in Maxima code. For example, the Lisp identifier `*foo-bar*` is written `?\*foo\-bar\*` in Maxima.

Lisp code may be executed from within a Maxima session. A single line of Lisp (containing one or more forms) may be executed by the special command `:lisp`. For example,

```
(%i1) :lisp (foo $x $y)
```

calls the Lisp function `foo` with Maxima variables `x` and `y` as arguments. The `:lisp` construct can appear at the interactive prompt or in a file processed by `batch` or `demo`, but not in a file processed by `load`, `batchload`, `translate_file`, or `compile_file`.

The function `to_lisp()` opens an interactive Lisp session. Entering `(to-maxima)` closes the Lisp session and returns to Maxima.

Lisp functions and variables which are to be visible in Maxima as functions and variables with ordinary names (no special punctuation) must have Lisp names beginning with the dollar sign `$`.

Maxima is case-sensitive, distinguishing between lowercase and uppercase letters in identifiers, while Lisp is not. There are some rules governing the translation of names between Lisp and Maxima.

1. A Lisp identifier not enclosed in vertical bars corresponds to a Maxima identifier in lowercase. Whether the Lisp identifier is uppercase, lowercase, or mixed case, is ignored. E.g., Lisp `$foo`, `$FOO`, and `$Foo` all correspond to Maxima `foo`.
2. A Lisp identifier which is all uppercase or all lowercase and enclosed in vertical bars corresponds to a Maxima identifier with case reversed. That is, uppercase is changed to lowercase and lowercase to uppercase. E.g., Lisp `|$FOO|` and `|$foo|` correspond to Maxima `foo` and `FOO`, respectively.
3. A Lisp identifier which is mixed uppercase and lowercase and enclosed in vertical bars corresponds to a Maxima identifier with the same case. E.g., Lisp `|$Foo|` corresponds to Maxima `Foo`.

The `#$` Lisp macro allows the use of Maxima expressions in Lisp code. `#$`*expr*`$` expands to a Lisp expression equivalent to the Maxima expression *expr*.

```
(msetq $foo #$[x, y]$)
```

This has the same effect as entering

```
(%i1) foo: [x, y];
```

The Lisp function `displa` prints an expression in Maxima format.

```
(%i1) :lisp #$[x, y, z]$
((MLIST SIMP) $X $Y $Z)
(%i1) :lisp (displa '((MLIST SIMP) $X $Y $Z))
[x, y, z]
NIL
```

Functions defined in Maxima are not ordinary Lisp functions. The Lisp function `mfuncall` calls a Maxima function. For example:

```
(%i1) foo(x,y) := x*y$
(%i2) :lisp (mfuncall '$foo 'a 'b)
((MTIMES SIMP) A B)
```

Some Lisp functions are shadowed in the Maxima package, namely the following.

`complement`, `continue`, `//`, `float`, `functionp`, `array`, `exp`, `listen`, `signum`, `atan`, `asin`, `acos`, `asinh`, `acosh`, `atanh`, `tanh`, `cosh`, `sinh`, `tan`, `break`, and `gcd`.

3.3 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.

3.4 Documentation

The Maxima on-line user's manual can be viewed in different forms. From the Maxima interactive prompt, the user's manual is viewed as plain text by the `?` command (i.e., the `describe` function). The user's manual is viewed as `info` hypertext by the `info` viewer program and as a web page by any ordinary web browser.

`example` displays examples for many Maxima functions. For example,

```
(%i1) example (integrate);
```

yields

```
(%i2) test(f):=block([u],u:integrate(f,x),ratsimp(f-diff(u,x)))
(%o2) test(f) := block([u], u : integrate(f, x),

                                           ratsimp(f - diff(u, x)))
(%i3) test(sin(x))
(%o3)                         0
(%i4) test(1/(x+1))
(%o4)                         0
(%i5) test(1/(x^2+1))
(%o5)                         0
```

and additional output.

3.5 Definitions for Help

demo (*filename*) Function

Evaluates Maxima expressions in *filename* and displays the results. `demo` pauses after evaluating each expression and continues after the user enters a carriage return. (If running in Xmaxima, `demo` may need to see a semicolon `;` followed by a carriage return.)

`demo` searches the list of directories `file_search_demo` to find `filename`. If the file has the suffix `dem`, the suffix may be omitted. See also `file_search`.

`demo` evaluates its argument. `demo` returns the name of the demonstration file.

Example:

```
(%i1) demo ("disol");

batching /home/wfs/maxima/share/simplification/disol.dem
```

```
 At the _ prompt, type ';' followed by enter to get next demo
(%i2)                          load(disol)


_
(%i3)              exp1 : a (e (g + f) + b (d + c))
(%o3)                  a (e (g + f) + b (d + c))


_
(%i4)                   disolate(exp1, a, b, e)
(%t4)                            d + c

(%t5)                            g + f

(%o5)                      a (%t5 e + %t4 b)


_
(%i5) demo ("rncomb");

batching /home/wfs/maxima/share/simplification/rncomb.dem
 At the _ prompt, type ';' followed by enter to get next demo
(%i6)                         load(rncomb)


_
                                z          x
(%i7)                  exp1 : ----- + ---------
                              y + x   2 (y + x)
                             z          x
(%o7)                      ----- + ---------
                           y + x   2 (y + x)


_
(%i8)                         combine(exp1)
                             z          x
(%o8)                      ----- + ---------
                           y + x   2 (y + x)


_
(%i9)                         rncombine(%)
                                2 z + x
(%o9)                          ---------
                               2 (y + x)


_
                                d   c   b   a
(%i10)                   exp2 : - + - + - + -
                                3   3   2   2
                             d   c   b   a
(%o10)                       - + - + - + -
                             3   3   2   2
```

```
_
(%i11)                         combine(exp2)
                          2 d + 2 c + 3 (b + a)
(%o11)                    ---------------------
                                    6

_
(%i12)                        rncombine(exp2)
                          2 d + 2 c + 3 b + 3 a
(%o12)                    ---------------------
                                    6

_
(%i13)
```

describe (*string*) Function

Finds all documented items which contain *string* in their titles. If there is more than one such item, Maxima asks the user to select an item or items to display. At the interactive prompt, `? foo` (with a space between `?` and `foo`) is equivalent to `describe ("foo")`.

`describe ("")` yields a list of all topics documented in the on-line manual.

`describe` quotes its argument. `describe` always returns `false`.

Example:

```
(%i1) ? integ
 0: (maxima.info)Introduction to Elliptic Functions and Integrals.
 1: Definitions for Elliptic Integrals.
 2: Integration.
 3: Introduction to Integration.
 4: Definitions for Integration.
 5: askinteger :Definitions for Simplification.
 6: integerp :Definitions for Miscellaneous Options.
 7: integrate :Definitions for Integration.
 8: integrate_use_rootsof :Definitions for Integration.
 9: integration_constant_counter :Definitions for Integration.
Enter space-separated numbers, ‘all’ or ‘none’: 7 8

Info from file /use/local/maxima/doc/info/maxima.info:
 - Function: integrate (expr, var)
 - Function: integrate (expr, var, a, b)
     Attempts to symbolically compute the integral of ‘expr’ with
     respect to ‘var’.  ‘integrate (expr, var)’ is an indefinite
     integral, while ‘integrate (expr, var, a, b)’ is a definite
     integral, [...]
```

In this example, items 7 and 8 were selected. All or none of the items could have been selected by entering `all` or `none`, which can be abbreviated `a` or `n`, respectively.

see Section 3.1 [Introduction to Help], page 9

example (*topic*) Function
example () Function

`example (`*topic*`)` displays some examples of *topic*, which is a symbol (not a string). Most topics are function names. `example ()` returns the list of all recognized topics.

The name of the file containing the examples is given by the global variable `manual_demo`, which defaults to `"manual.demo"`.

`example` quotes its argument. `example` returns `done` unless there is an error or there is no argument, in which case `example` returns the list of all recognized topics.

Examples:

```
(%i1) example (append);
(%i2) append([x+y,0,-3.2],[2.5E+20,x])
(%o2)             [y + x, 0, - 3.2, 2.5E+20, x]
(%o2)                         done
(%i3) example (coeff);
(%i4) coeff(b+tan(x)+2*a*tan(x) = 3+5*tan(x),tan(x))
(%o4)                      2 a + 1 = 5
(%i5) coeff(1+x*%e^x+y,x,0)
(%o5)                         y + 1
(%o5)                         done
```

4 Command Line

4.1 Introduction to Command Line

"'" Operator

The single quote operator ' prevents evaluation.

Applied to a symbol, the single quote prevents evaluation of the symbol.

Applied to a function call, the single quote prevents evaluation of the function call, although the arguments of the function are still evaluated (if evaluation is not otherwise prevented). The result is the noun form of the function call.

Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression. E.g., `'(f(x))` means do not evaluate the expression `f(x)`. `'f(x)` (with the single quote applied to `f` instead of `f(x)`) means return the noun form of `f` applied to `[x]`.

The single quote does not prevent simplification.

When the global flag `noundisp` is `true`, nouns display with a single quote. This switch is always `true` when displaying function definitions.

See also the quote-quote operator `''` and `nouns`.

Examples:

Applied to a symbol, the single quote prevents evaluation of the symbol.

```
(%i1) aa: 1024;
(%o1)                         1024
(%i2) aa^2;
(%o2)                        1048576
(%i3) 'aa^2;
                                 2
(%o3)                          aa
(%i4) ''%;
(%o4)                        1048576
```

Applied to a function call, the single quote prevents evaluation of the function call. The result is the noun form of the function call.

```
(%i1) x0: 5;
(%o1)                           5
(%i2) x1: 7;
(%o2)                           7
(%i3) integrate (x^2, x, x0, x1);
                               218
(%o3)                          ---
                                3
(%i4) 'integrate (x^2, x, x0, x1);
                             7
                            /
                            [   2
(%o4)                       I  x  dx
```

```
                              ]
                              /
                               5
(%i5) %, nouns;
                                 218
(%o5)                            ---
                                  3
```

Applied to a parenthesized expression, the single quote prevents evaluation of all symbols and function calls in the expression.

```
(%i1) aa: 1024;
(%o1)                           1024
(%i2) bb: 19;
(%o2)                            19
(%i3) sqrt(aa) + bb;
(%o3)                            51
(%i4) '(sqrt(aa) + bb);
(%o4)                        bb + sqrt(aa)
(%i5) ''%;
(%o5)                            51
```

The single quote does not prevent simplification.

```
(%i1) sin (17 * %pi) + cos (17 * %pi);
(%o1)                            - 1
(%i2) '(sin (17 * %pi) + cos (17 * %pi));
(%o2)                            - 1
```

"''" Operator

The `''` (double single quotes) operator causes an extra evaluation to occur. E.g., `''%i4` will re-evaluate input line `%i4`. `''(f(x))` means evaluate the expression `f(x)` an extra time. `''f(x)` (with the double single quotes applied to `f` instead of `f(x)`) means return the verb form of `f` applied to `[x]`.

4.2 Definitions for Command Line

alias (*new_name_1, old_name_1, ..., new_name_n, old_name_n*) Function

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

debugmode Option variable

Default value: `false`

When a Maxima error occurs, Maxima will start the debugger if `debugmode` is `true`. The user may enter commands to examine the call stack, set breakpoints, step through Maxima code, and so on. See `debugging` for a list of debugger commands.

Enabling `debugmode` will not catch Lisp errors.

ev (*expr, arg_1, ..., arg_n*) Function

Evaluates the expression *expr* in the environment specified by the arguments *arg_1*, ..., *arg_n*. The arguments are switches (Boolean flags), assignments, equations, and functions. `ev` returns the result (another expression) of the evaluation.

The evaluation is carried out in steps, as follows.

1. First the environment is set up by scanning the arguments which may be any or all of the following.
   - `simp` causes *expr* to be simplified regardless of the setting of the switch `simp` which inhibits simplification if `false`.
   - `noeval` supresses the evaluation phase of `ev` (see step (4) below). This is useful in conjunction with the other switches and in causing *expr* to be resimplified without being reevaluated.
   - `nouns` causes the evaluation of noun forms (typically unevaluated functions such as `'integrate` or `'diff`) in *expr*.
   - `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 *expr* to have their determinant kept outside of the inverse rather than dividing through each element.
   - `diff` causes all differentiations indicated in *expr* to be performed.
   - `derivlist (x, y, z, ...)` 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 *expr* which have been given numervals 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 *expr* to occur. (See step (5) below.)
   - `A` where `A` is an atom declared to be an evaluation flag (see `evflag`) causes `A` to be bound to `true` during the evaluation of *expr*.
   - `V: expression` (or alternately `V=expression`) causes `V` to be bound to the value of `expression` during the evaluation of *expr*. Note that if `V` is a Maxima option, then `expression` is used for its value during the evaluation of *expr*. 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.
   - `F` where `F`, a function name, has been declared to be an evaluation function (see `evfun`) causes `F` to be applied to *expr*.
   - Any other function names (e.g., `sum`) cause evaluation of occurrences of those names in *expr* as though they were verbs.

- In addition a function occurring in *expr* (say `F(x)`) may be defined locally for the purpose of this evaluation of *expr* by giving `F(x) := expression` 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., `[%t1, %t2]` where `%t1` and `%t2` are equations) such as that returned by `solve`.

The arguments of `ev` may be given in any order with the exception of substitution equations which are handled in sequence, left to right, and evaluation functions which are composed, e.g., `ev (`*expr*`, ratsimp, realpart)` is handled as `realpart (ratsimp (`*expr*`))`.

The `simp`, `numer`, `float`, and `pred` switches may also be set locally in a block, or globally in Maxima so that they will remain in effect until being reset.

If *expr* is a canonical rational expression (CRE), then the expression returned by `ev` is also a CRE, 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 arguments or in the value of some arguments if the value is an equation. The variables (subscripted variables which do not have associated array functions as well as non-subscripted variables) in the expression *expr* are replaced by their global values, except for those appearing in this list. Usually, *expr* is just a label or `%` (as in `%i2` in the example 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 arguments, they are carried out now.
4. The resulting expression is then re-evaluated (unless one of the arguments was `noeval`) and simplified according to the arguments. Note that any function calls in *expr* 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 arguments was `eval`, steps (3) and (4) are repeated.

Examples

```
(%i1) sin(x) + cos(y) + (w+1)^2 + 'diff (sin(w), w);
                                     d                    2
(%o1)             cos(y) + sin(x) + -- (sin(w)) + (w + 1)
                                     dw
(%i2) ev (%, sin, expand, diff, x=2, y=1);
                          2
(%o2)           cos(w) + w  + 2 w + cos(1) + 1.909297426825682
```

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

expr, *arg_1*, ..., *arg_n*

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

Notice the parallel binding process in the following example.

```
(%i3) programmode: false;
(%o3)                                false
(%i4) x+y, x: a+y, y: 2;
(%o4)                              y + a + 2
(%i5) 2*x - 3*y = 3$
(%i6) -3*x + 2*y = -4$
(%i7) solve ([%o5, %o6]);
Solution

                                         1
(%t7)                             y = - -
                                         5

                                        6
(%t8)                               x = -
                                        5
(%o8)                            [[%t7, %t8]]
(%i8) %o6, %o8;
(%o8)                              - 4 = - 4
(%i9) x + 1/x > gamma (1/2);
                                   1
(%o9)                          x + - > sqrt(%pi)
                                   x
(%i10) %, numer, x=1/2;
(%o10)                      2.5 > 1.772453850905516
(%i11) %, pred;
(%o11)                               true
```

evflag Property

Some Boolean flags have the `evflag` property. `ev` treats such flags specially. A flag with the `evflag` property will be bound to `true` during the execution of `ev` if it is mentioned in the call to `ev`. For example, `demoivre` and `ratfac` are bound to `true` during the call `ev (%, demoivre, ratfac)`.

The flags which have the `evflag` property are: `algebraic`, `cauchysum`, `demoivre`, `dotscrules`, `%emode`, `%enumer`, `exponentialize`, `exptisolate`, `factorflag`, `float`, `halfangles`, `infeval`, `isolate_wrt_times`, `keepfloat`, `letrat`, `listarith`, `logabs`, `logarc`, `logexpand`, `lognegint`, `lognumer`, `m1pbranch`, `numer_pbranch`, `programmode`, `radexpand`, `ratalgdenom`, `ratfac`, `ratmx`, `ratsimpexpons`, `simp`, `simpsum`, `sumexpand`, and `trigexpand`.

The construct `:lisp (putprop '|$foo| t 'evflag)` gives the `evflag` property to the variable `foo`, so `foo` is bound to `true` during the call `ev (%, foo)`. Equivalently, `ev (%, foo:true)` has the same effect.

evfun Property

Some functions have the `evfun` property. `ev` treats such functions specially. A function with the `evfun` property will be applied during the execution of `ev` if it is mentioned in the call to `ev`. For example, `ratsimp` and `radcan` will be applied during the call `ev (%, ratsimp, radcan)`.

The functions which have the `evfun` property are: `bfloat`, `factor`, `fullratsimp`, `logcontract`, `polarform`, `radcan`, `ratexpand`, `ratsimp`, `rectform`, `rootscontract`, `trigexpand`, and `trigreduce`.

The construct `:lisp (putprop '|$foo| t 'evfun)` gives the `evfun` property to the function `foo`, so that `foo` is applied during the call `ev (%, foo)`. Equivalently, `foo (ev (%))` has the same effect.

infeval Option variable

Enables "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.

kill (*symbol_1, ..., symbol_n*) Function
kill (*labels*) Function
kill (*clabels, dlabels, elabels*) Function
kill (*n*) Function
kill ([*m, n*]) Function
kill (*values, functions, arrays, ...*) Function
kill (*all*) Function
kill (*allbut* (*symbol_1, ..., symbol_n*)) Function

Removes all bindings (value, function, array, or rule) from the arguments *symbol_1*, ..., *symbol_n*. An argument may be a single array element or subscripted function.

Several special arguments are recognized. Different kinds of arguments may be combined, e.g., `kill (clabels, functions, allbut (foo, bar))`.

`kill (labels)` unbinds all input, output, and intermediate expression labels created so far. `kill (clabels)` unbinds only input labels which begin with the current value of `inchar`. Likewise, `kill (dlabels)` unbinds only output labels which begin with the current value of `outchar`, and `kill (elabels)` unbinds only intermediate expression labels which begin with the current value of `linechar`.

`kill (`*n*`)`, where *n* is an integer, unbinds the *n* most recent input and output labels.

`kill ([`*m*`, `*n*`])` unbinds input and output labels *m* through *n*.

`kill (`*infolist*`)`, where *infolist* is any item in `infolists` (such as `values`, `functions`, or `arrays`) unbinds all items in `infolist`. See also `infolists`.

`kill (all)` unbinds all items on all infolists. `kill (all)` does not reset global variables to their default values; see `reset` on this point.

`kill (allbut (`*symbol_1*`, ..., `*symbol_n*`))` unbinds all items on all infolists except for *symbol_1*, ..., *symbol_n*. `kill (allbut (`*infolist*`))` unbinds all items except for the ones on *infolist*, where *infolist* is `values`, `functions`, `arrays`, etc.

The memory taken up by a bound property is not released until all symbols are unbound from it. In particular, to release the memory taken up by the value of a symbol, one unbinds the output label which shows the bound value, as well as unbinding the symbol itself.

`kill` quotes its arguments. The double single quotes operator, `''`, defeats the quotation.

`kill` (*symbol*) unbinds all properties of *symbol*. In contrast, `remvalue`, `remfunction`, `remarray`, and `remrule` unbind a specific property.

`kill` always returns `done`, even if an argument has no binding.

labels (*symbol*) Function
labels System variable

Returns the list of input, output, or intermediate expression labels which begin with *symbol*. Typically *symbol* is the value of `inchar`, `outchar`, or `linechar`. The label character may be given with or without a percent sign, so, for example, `i` and `%i` yield the same result.

If no labels begin with *symbol*, `labels` returns an empty list.

The function `labels` quotes its argument. The double single quotes operator `''` defeats quotation. For example, `labels (''inchar)` returns the input labels which begin with the current input label character.

The variable `labels` is the list of input, output, and intermediate expression labels, including all previous labels if `inchar`, `outchar`, or `linechar` were redefined.

By default, Maxima displays the result of each user input expression, giving the result an output label. The output display is suppressed by terminating the input with `$` (dollar sign) instead of `;` (semicolon). An output label is generated, but not displayed, and the label may be referenced in the same way as displayed output labels. See also `%`, `%%`, and `%th`.

Intermediate expression labels can be generated by some functions. The flag `programmode` controls whether `solve` and some other functions generate intermediate expression labels instead of returning a list of expressions. Some other functions, such as `ldisplay`, always generate intermediate expression labels.

`first (rest (labels (''inchar)))` returns the most recent input label.

See also `inchar`, `outchar`, `linechar`, and `infolists`.

linenum System variable

The line number of the current pair of input and output expressions.

myoptions System variable

Default value: `[]`

`myoptions` is the list of all options ever reset by the user, whether or not they get reset to their default value.

nolabels Option variable

Default value: `false`

When `nolabels` is `true`, input and output labels are generated but not appended to `labels`, the list of all input and output labels. `kill (labels)` kills the labels on the `labels` list, but does not kill any labels generated since `nolabels` was assigned `true`. It seems likely this behavior is simply broken.

See also `batch`, `batchload`, and `labels`.

optionset Option variable

Default value: `false`

When `optionset` is `true`, Maxima prints out a message whenever a Maxima 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 () Function
playback (*n*) Function
playback ([*m*, *n*]) Function
playback ([*m*]) Function
playback (*input*) Function
playback (*slow*) Function
playback (*time*) Function
playback (*grind*) Function

Displays input, output, and intermediate expressions, without recomputing them. `playback` only displays the expressions bound to labels; any other output (such as text printed by `print` or `describe`, or error messages) is not displayed. See also `labels`.

`playback` quotes its arguments. The double single quotes operator, `''`, defeats quotation. `playback` always returns `done`.

`playback ()` (with no arguments) displays all input, output, and intermediate expressions generated so far. An output expression is displayed even if it was suppressed by the `$` terminator when it was originally computed.

`playback (`*n*`)` displays the most recent *n* expressions. Each input, output, and intermediate expression counts as one.

`playback ([`*m*`, `*n*`])` displays input, output, and intermediate expressions with numbers from *m* through *n*, inclusive.

`playback ([`*m*`])` is equivalent to `playback ([`*m*`, `*m*`])`; this usually prints one pair of input and output expressions.

`playback (input)` displays all input expressions generated so far.

`playback (slow)` pauses between expressions and waits for the user to press `enter`. This behavior is similar to `demo`. `playback (slow)` is useful in conjunction with `save` or `stringout` when creating a secondary-storage file in order to pick out useful expressions.

`playback (time)` displays the computation time for each expression.

`playback (grind)` displays input expressions in the same format as the `grind` function. Output expressions are not affected by the `grind` option. See `grind`.

Arguments may be combined, e.g., `playback ([5, 10], grind, time, slow)`.

printprops (*a*, *i*) Function
printprops ([*a_1*, ..., *a_n*], *i*) Function
printprops (*all*, *i*) Function

Displays 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 Option variable

Default value: `_`

`prompt` is the prompt symbol of the `demo` function, `playback (slow)` mode, and the Maxima break loop (as invoked by `break`).

quit () Function

Terminates the Maxima session. Note that the function must be invoked as `quit();` or `quit()$`, not `quit` by itself.

To stop a lengthy computation, type `control-C`. The default action is to return to the Maxima prompt. If `*debugger-hook*` is `nil`, `control-C` opens the Lisp debugger. See also `debugging`.

remfunction (*f_1, ..., f_n*) Function
remfunction (*all*) Function

Removes the user defined functions *f_1*, ..., *f_n* from Maxima. `remfunction (all)` removes all functions.

reset () Function

Resets many global variables and options, and some other variables, to their default values.

`reset` processes the variables on the Lisp list `*variable-initial-values*`. The Lisp macro `defmvar` puts variables on this list (among other actions). Many, but not all, global variables and options are defined by `defmvar`, and some variables defined by `defmvar` are not global variables or options.

showtime Option variable

Default value: `false`

When `showtime` is `true`, the computation time and elapsed time is printed with each output expression.

See also `time`, `timer`, and `playback`.

sstatus (*feature, package*) Function

Sets the status of *feature* in *package*. After `sstatus (`*feature*`, `*package*`)` is executed, `status (`*feature*`, `*package*`)` returns `true`. This can be useful for package writers, to keep track of what features they have loaded in.

to_lisp () Function

Enters the Lisp system under Maxima. `(to-maxima)` returns to Maxima.

values System variable

Initial value: `[]`

`values` is a list of all bound user variables (not Maxima options or switches). The list comprises symbols bound by `:` , `::`, or `:=`.

5 Operators

5.1 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. 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)`.

See also `syntax`.

5.2 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 Maxima 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.

See also `syntax`.

5.3 operator

See `operators`.

5.4 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.

See also `syntax`.

5.5 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.

See also `syntax`.

5.6 Definitions for Operators

"!" Operator

The factorial operator. For any complex number `x` (including integer, rational, and real numbers) except for negative integers, `x!` is defined as `gamma(x+1)`.

For an integer `x`, `x!` simplifies to the product of the integers from 1 to `x` inclusive. `0!` simplifies to 1. For a floating point number `x`, `x!` simplifies to the value of `gamma (x+1)`. For `x` equal to `n/2` where `n` is an odd integer, `x!` simplifies to a rational factor times `sqrt (%pi)` (since `gamma (1/2)` is equal to `sqrt (%pi)`). If `x` is anything else, `x!` is not simplified.

The variables `factlim`, `minfactorial`, and `factcomb` control the simplification of expressions containing factorials.

The functions `gamma`, `bffac`, and `cbffac` are varieties of the gamma function. `makegamma` substitutes `gamma` for factorials and related functions.

See also `binomial`.

- The factorial of an integer, half-integer, or floating point argument is simplified unless the operand is greater than `factlim`.

```
(%i1) factlim: 10$
(%i2) [0!, (7/2)!, 4.77!, 8!, 20!];
                105 sqrt(%pi)
(%o2) [1, -------------, 81.44668037931193, 40320, 20!]
                     16
```

- The factorial of a complex number, known constant, or general expression is not simplified. Even so it may be possible simplify the factorial after evaluating the operand.

```
(%i1) [(%i + 1)!, %pi!, %e!, (cos(1) + sin(1))!];
(%o1)    [(%i + 1)!, %pi!, %e!, (sin(1) + cos(1))!]
(%i2) ev (%, numer, %enumer);
(%o2) [(%i + 1)!, 7.188082728976031, 4.260820476357003,

                                        1.227580202486819]
```

- The factorial of an unbound symbol is not simplified.

```
(%i1) kill (foo)$
(%i2) foo!;
(%o2)                         foo!
```

- Factorials are simplified, not evaluated. Thus `x!` may be replaced even in a quoted expression.

```
(%i1) '([0!, (7/2)!, 4.77!, 8!, 20!]);
                105 sqrt(%pi)
(%o1) [1, -------------, 81.44668037931193, 40320, 20!]
                     16
```

"!!" Operator

The double factorial operator.

For an integer, float, or rational number `n`, `n!!` evaluates to the product `n (n-2) (n-4) (n-6) ... (n - 2 (k-1))` where `k` is equal to `entier (n/2)`, that is, the largest integer less than or equal to `n/2`. Note that this definition does not coincide with other published definitions for arguments which are not integers.

For an even (or odd) integer `n`, `n!!` evaluates to the product of all the consecutive even (or odd) integers from 2 (or 1) through `n` inclusive.

For an argument `n` which is not an integer, float, or rational, `n!!` yields a noun form `genfact (n, n/2, 2)`.

"#" Operator

Represents the negation of syntactic equality `=`.

Note that because of the rules for evaluation of predicate expressions (in particular because `not` *expr* causes evaluation of *expr*), `not` *a* `=` *b* is not equivalent to *a* `#` *b* in some cases.

Examples:

```
(%i1) a = b;
(%o1)                         a = b
(%i2) is (a = b);
(%o2)                         false
(%i3) a # b;
(%o3)                         a # b
(%i4) not a = b;
(%o4)                         true
(%i5) is (a # b);
(%o5)                         true
(%i6) is (not a = b);
(%o6)                         true
```

"." 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.

See also `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dotscrules`.

":" 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 `macros`.

":=" Operator

The function definition operator. E.g. `f(x):=sin(x)` defines a function `f`.

"=" Operator

denotes an equation to Maxima. To the pattern matcher in Maxima it denotes a total relation that holds between two expressions if and only if the expressions are syntactically identical.

The negation of `=` is represented by `#`. Note that because of the rules for evaluation of predicate expressions (in particular because `not` *expr* causes evaluation of *expr*), `not` *a* `=` *b* is not equivalent to *a* `#` *b* in some cases.

and Operator

The logical conjunction operator. `and` is an n-ary infix operator; its operands are Boolean expressions, and its result is a Boolean value.

`and` forces evaluation (like `is`) of one or more operands, and may force evaluation of all operands.

Operands are evaluated in the order in which they appear. `and` evaluates only as many of its operands as necessary to determine the result. If any operand is `false`, the result is `false` and no further operands are evaluated.

The global flag `prederror` governs the behavior of `and` when an evaluated operand cannot be determined to be `true` or `false`. `and` prints an error message when `prederror` is `true`. Otherwise, `and` returns `unknown`.

`and` is not commutative: `a and b` might not be equal to `b and a` due to the treatment of indeterminate operands.

or Operator

The logical disjunction operator. `or` is an n-ary infix operator; its operands are Boolean expressions, and its result is a Boolean value.

`or` forces evaluation (like `is`) of one or more operands, and may force evaluation of all operands.

Operands are evaluated in the order in which they appear. `or` evaluates only as many of its operands as necessary to determine the result. If any operand is `true`, the result is `true` and no further operands are evaluated.

The global flag `prederror` governs the behavior of `or` when an evaluated operand cannot be determined to be `true` or `false`. `or` prints an error message when `prederror` is `true`. Otherwise, `or` returns `unknown`.

`or` is not commutative: `a or b` might not be equal to `b or a` due to the treatment of indeterminate operands.

not Operator

The logical negation operator. `not` is a prefix operator; its operand is a Boolean expression, and its result is a Boolean value.

`not` forces evaluation (like `is`) of its operand.

The global flag `prederror` governs the behavior of `not` when its operand cannot be determined to be `true` or `false`. `not` prints an error message when `prederror` is `true`. Otherwise, `not` returns `unknown`.

abs (*expr*) Function

Returns the absolute value *expr*. If *expr* is complex, returns the complex modulus of *expr*.

additive Keyword

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,

```
(%i1) expr: e+d+c+b+a$
(%i2) part (expr, [2, 5]);
(%o2)                          d + a
```

while

```
(%i3) part (expr, allbut (2, 5));
(%o3)                        e + c + b
```

It also works with the `kill` command,

```
kill (allbut (name_1, ..., name_k))
```

will do a `kill (all)` except it will not `kill` the names specified. Note: `name_i` 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 (*expr*) Function

Returns the complex absolute value (the complex modulus) of *expr*.

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

Returns the largest integer less than or equal to *x* where *x* is numeric. `fix` (as in `fixnum`) is a synonym for this, so `fix(x)` is precisely the same.

equal (*expr_1*, *expr_2*) Function

Used with an `is`, returns `true` (or `false`) if and only if *expr_1* and *expr_2* are equal (or not equal) for all possible values of their variables (as determined by `ratsimp`). 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)` yields `false` but `is (equal (rat(0), 0))` yields `true`.

If a determination can't be made, then `is (equal (a, b))` returns a simplified but equivalent expression, whereas `is (a=b)` always returns either `true` or `false`.

All variables occurring in *expr_1* and *expr_2* are presumed to be real valued.

The negation of `equal` is `notequal`. Note that because of the rules for evaluation of predicate expressions (in particular because `not` *expr* causes evaluation of *expr*), `notequal` is not equivalent to `not equal` in some cases.

`ev (`*expr*`, pred)` is equivalent to `is (`*expr*`)`.

```
(%i1) is (x^2 >= 2*x - 1);
(%o1)                          true
(%i2) assume (a > 1);
(%o2)                         [a > 1]
(%i3) is (log (log (a+1) + 1) > 0 and a^2 + 1 > 2*a);
(%o3)                          true
```

notequal (*expr_1*, *expr_2*) Function

Represents the negation of `equal (`*expr_1*`, `*expr_2*`)`.

Note that because of the rules for evaluation of predicate expressions (in particular because `not` *expr* causes evaluation of *expr*), `notequal` is not equivalent to `not equal` in some cases.

Examples:

```
(%i1) equal (a, b);
(%o1)                      equal(a, b)
(%i2) maybe (equal (a, b));
(%o2)                        unknown
(%i3) notequal (a, b);
(%o3)                    notequal(a, b)
(%i4) not equal (a, b);
`macsyma' was unable to evaluate the predicate:
equal(a, b)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) maybe (notequal (a, b));
(%o5)                        unknown
(%i6) maybe (not equal (a, b));
(%o6)                        unknown
(%i7) assume (a > b);
(%o7)                         [a > b]
(%i8) equal (a, b);
(%o8)                      equal(a, b)
(%i9) maybe (equal (a, b));
(%o9)                          false
```

```
(%i10) notequal (a, b);
(%o10)                  notequal(a, b)
(%i11) not equal (a, b);
(%o11)                       true
(%i12) maybe (notequal (a, b));
(%o12)                       true
(%i13) maybe (not equal (a, b));
(%o13)                       true
```

eval Operator

As an argument in a call to `ev (`*expr*`)`, `eval` causes an extra evaluation of *expr*. See `ev`.

evenp (*expr*) Function

Returns `true` if *expr* is an even integer. `false` is returned in all other cases.

fix (*x*) Function

A synonym for `entier (`*x*`)`.

fullmap (*f, expr_1, ...*) Function

Similar to `map`, but `fullmap` keeps mapping down all subexpressions until the main operators are no longer the same.

`fullmap` is used by the Maxima simplifier for certain matrix manipulations; thus, Maxima sometimes generates an error message concerning `fullmap` even though `fullmap` was not explicitly called by the user.

```
(%i1) a + b*c$
(%i2) fullmap (g, %);
(%o2)                  g(b) g(c) + g(a)
(%i3) map (g, %th(2));
(%o3)                    g(b c) + g(a)
```

fullmapl (*f, list_1, ...*) Function

Similar to `fullmap`, but `fullmapl` only maps onto lists and matrices.

```
(%i1) fullmapl ("+", [3, [4, 5]], [[a, 1], [0, -1.5]]);
(%o1)               [[a + 3, 4], [4, 3.5]]
```

is (*expr*) Function

Attempts to determine whether the predicate *expr* is provable from the facts in the `assume` database.

If the predicate is provably `true` or `false`, `is` returns `true` or `false`, respectively. Otherwise, the return value is controlled by the global flag `prederror`. When `prederror` is `false`, `is` returns `unknown` for a predicate which cannot be proven nor disproven, and reports an error otherwise.

See also `assume`, `facts`, and `maybe`.

Examples:

`is` causes evaluation of predicates.

```
(%i1) %pi > %e;
(%o1)                          %pi > %e
(%i2) is (%pi > %e);
(%o2)                            true
```

`is` attempts to derive predicates from the `assume` database.

```
(%i1) assume (a > b);
(%o1)                           [a > b]
(%i2) assume (b > c);
(%o2)                           [b > c]
(%i3) is (a < b);
(%o3)                            false
(%i4) is (a > c);
(%o4)                            true
(%i5) is (equal (a, c));
(%o5)                            false
```

If `is` can neither prove nor disprove a predicate from the `assume` database, the global flag `prederror` governs the behavior of `is`.

```
(%i1) assume (a > b);
(%o1)                           [a > b]
(%i2) prederror: true$
(%i3) is (a > 0);
`macsyma' was unable to evaluate the predicate:
a > 0
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) prederror: false$
(%i5) is (a > 0);
(%o5)                           unknown
```

maybe (*expr*) Function

Attempts to determine whether the predicate *expr* is provable from the facts in the `assume` database.

If the predicate is provably `true` or `false`, `maybe` returns `true` or `false`, respectively. Otherwise, `maybe` returns `unknown`.

`maybe` is functionally equivalent to `is` with `prederror: false`, but the result is computed without actually assigning a value to `prederror`.

See also `assume`, `facts`, and `is`.

Examples:

```
(%i1) maybe (x > 0);
(%o1)                           unknown
(%i2) assume (x > 1);
(%o2)                           [x > 1]
(%i3) maybe (x > 0);
(%o3)                            true
```

isqrt (*x*) Function

Returns the "integer square root" of the absolute value of *x*, which is an integer.

max (*x_1, x_2, ...*) Function

Returns the maximum of its arguments (or returns a simplified form if some of its arguments are non-numeric).

min (*x_1, x_2, ...*) Function

Returns the minimum of its arguments (or returns a simplified form if some of its arguments are non-numeric).

mod (*p*) Function
mod (*p, m*) Function

Converts the polynomial *p* to a modular representation with respect to the current modulus which is the value of the variable `modulus`.

`mod (`*p*`, `*m*`)` specifies a modulus *m* to be used instead of the current value of `modulus`.

See `modulus`.

oddp (*expr*) Function

is `true` if *expr* is an odd integer. `false` is returned in all other cases.

pred Operator

As an argument in a call to `ev (`*expr*`)`, `pred` causes predicates (expressions which evaluate to `true` or `false`) to be evaluated. See `ev`.

make_random_state (*n*) Function
make_random_state (*s*) Function
make_random_state (*true*) Function
make_random_state (*false*) Function

A random state object represents the state of the random number generator. The state comprises 627 32-bit words.

`make_random_state (`*n*`)` returns a new random state object created from an integer seed value equal to *n* modulo 2^32. *n* may be negative.

`make_random_state (`*s*`)` returns a copy of the random state *s*.

`make_random_state (true)` returns a new random state object, using the current computer clock time as the seed.

`make_random_state (false)` returns a copy of the current state of the random number generator.

set_random_state (*s*) Function

Copies *s* to the random number generator state.

`set_random_state` always returns `done`.

random (*x*) Function

Returns a pseudorandom number. If *x* is an integer, `random (`*x*`)` returns an integer from 0 through `x - 1` inclusive. If *x* is a floating point number, `random (`*x*`)` returns a nonnegative floating point number less than *x*. `random` complains with an error if *x* is neither an integer nor a float, or if *x* is not positive.

The functions `make_random_state` and `set_random_state` maintain the state of the random number generator.

The Maxima random number generator is an implementation of the Mersenne twister MT 19937.

Examples:

```
(%i1) s1: make_random_state (654321)$
(%i2) set_random_state (s1);
(%o2)                          done
(%i3) random (1000);
(%o3)                           768
(%i4) random (9573684);
(%o4)                         7657880
(%i5) random (2^75);
(%o5)                11804491615036831636390
(%i6) s2: make_random_state (false)$
(%i7) random (1.0);
(%o7)                   .2310127244107132
(%i8) random (10.0);
(%o8)                   4.394553645870825
(%i9) random (100.0);
(%o9)                   32.28666704056853
(%i10) set_random_state (s2);
(%o10)                         done
(%i11) random (1.0);
(%o11)                  .2310127244107132
(%i12) random (10.0);
(%o12)                  4.394553645870825
(%i13) random (100.0);
(%o13)                  32.28666704056853
```

sign (*expr*) Function

Attempts to determine the sign of *expr* 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

For numeric *x*, returns 0 if *x* is 0, otherwise returns -1 or +1 as *x* is less than or greater than 0, respectively.

If *x* is not numeric then a simplified but equivalent form is returned. For example, `signum(-x)` gives `-signum(x)`.

sort (*list*, *p*) Function
sort (*list*) Function

Sorts *list* according to a predicate `p` of two arguments, such as `"<"` or `orderlessp`.

`sort (`*list*`)` sorts *list* according to Maxima's built-in ordering.

list may contain numeric or nonnumeric items, or both.

sqrt (*x*) Function

The square root of *x*. It is represented internally by *x*`^(1/2)`. See also `rootscontract`.

`radexpand` if `true` will cause nth 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`.

sqrtdispflag Option variable

Default value: `true`

When `sqrtdispflag` is `false`, causes `sqrt` to display with exponent 1/2.

sublis (*list, expr*) Function

Makes multiple parallel substitutions into an expression.

The variable `sublis_apply_lambda` controls simplification after `sublis`.

Example:

```
(%i1) sublis ([a=b, b=a], sin(a) + cos(b));
(%o1)                    sin(b) + cos(a)
```

sublist (*list, p*) Function

Returns the list of elements of *list* for which the predicate `p` returns `true`.

Example:

```
(%i1) L: [1, 2, 3, 4, 5, 6]$
(%i2) sublist (L, evenp);
(%o2)                       [2, 4, 6]
```

sublis_apply_lambda Option variable

Default value: `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 double-quotes `"` 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.

`subst` is an alias for `substitute`.

`subst (`*eq_1*`,` *expr*`)` or `subst ([`*eq_1*`, ...,` *eq_k*`],` *expr*`)` are other permissible forms. The *eq_i* are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression *expr*.

`exptsubst` if `true` permits substitutions like `y` for `%e^x` in `%e^(a*x)` to take place.

When `opsubst` is `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.

Examples:

```
(%i1) subst (a, x+y, x + (x+y)^2 + y);
                                                 2
(%o1)                    y + x + a
(%i2) subst (-%i, %i, a + b*%i);
(%o2)                     a - %i b
```

For further examples, do `example (subst)`.

substinpart (*x, expr, n_1, ..., n_k*) Function

Similar to `substpart`, but `substinpart` works on the internal representation of *expr*.

```
(%i1) x . 'diff (f(x), x, 2);
                              2
                             D
(%o1)                   x . --- (f(x))
                              2
                             dx
(%i2) substinpart (d^2, %, 2);
                                  2
(%o2)                        x . d
(%i3) substinpart (f1, f[1](x+1), 0);
(%o3)                       f1(x + 1)
```

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

```
(%i1) part (x+y+z, [1, 3]);
(%o1)                         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` is set to `true` then `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

```
(%i1) expr: 27*y^3 + 54*x*y^2 + 36*x^2*y + y + 8*x^3 + x + 1;
              3         2       2            3
(%o1)     27 y  + 54 x y  + 36 x  y + y + 8 x  + x + 1
(%i2) part (expr, 2, [1, 3]);
                                  2
(%o2)                         54 y
(%i3) sqrt (piece/54);
(%o3)                        abs(y)
(%i4) substpart (factor (piece), expr, [1, 2, 3, 5]);
                               3
(%o4)               (3 y + 2 x)  + y + x + 1
(%i5) expr: 1/x + y/x - 1/z;
                             1   y   1
(%o5)                      - - + - + -
```

```
                                   z    x    x
(%i6) substpart (xthru (piece), expr, [2, 3]);
                                  y + 1   1
(%o6)                             ----- - -
                                    x     z
```

Also, setting the option `inflag` to `true` and calling `part` or `substpart` is the same as calling `inpart` or `substinpart`.

substpart (*x, expr, n_1, ..., n_k*) Function

Substitutes *x* for the subexpression picked out by the rest of the arguments as in `part`. It returns the new value of *expr*. *x* may be some operator to be substituted for an operator of *expr*. In some cases *x* needs to be enclosed in double-quotes " (e.g. `substpart ("+", a*b, 0)` yields `b + a`).

```
(%i1) 1/(x^2 + 2);
                                    1
(%o1)                            ------
                                  2
                                 x  + 2
(%i2) substpart (3/2, %, 2, 1, 2);
                                    1
(%o2)                            --------
                                  3/2
                                 x    + 2
(%i3) a*x + f (b, y);
(%o3)                       a x + f(b, y)
(%i4) substpart ("+", %, 1, 0);
(%o4)                      x + f(b, y) + a
```

Also, setting the option `inflag` to `true` and calling `part` or `substpart` is the same as calling `inpart` or `substinpart`.

subvarp (*expr*) Function

Returns `true` if *expr* is a subscripted variable, for example `a[i]`.

symbolp (*expr*) Function

Returns `true` if *expr* is a symbol, else `false`. In effect, `symbolp(x)` is equivalent to the predicate `atom(x) and not numberp(x)`.

See also `Identifiers`.

unorder () Function

Disables 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`. See also `ordergreat` and `orderless`.

```
(%i1) unorder();
(%o1)                              []
(%i2) b*x + a^2;
                                       2
(%o2)                            b x + a
```

```
(%i3) ordergreat (a);
(%o3)                          done
(%i4) b*x + a^2;
                                2
(%o4)                          a  + b x
(%i5) %th(1) - %th(3);
                                2    2
(%o5)                          a  - a
(%i6) unorder();
(%o6)                           [a]
```

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 (*expr*) Function

Combines all terms of *expr* (which should be a sum) over a common denominator without expanding products and exponentiated sums as `ratsimp` does. `xthru` cancels common factors in the numerator and denominator of rational expressions but only if the factors are explicit.

Sometimes it is better to use `xthru` before `ratsimp`ing 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 `ratsimp`ed.

```
(%i1) ((x+2)^20 - 2*y)/(x+y)^20 + (x+y)^(-19) - x/(x+y)^20;
                                     20
                 1           (x + 2)   - 2 y       x
(%o1)        --------- + --------------- - ---------
                    19             20             20
             (y + x)        (y + x)        (y + x)
(%i2) xthru (%);
                                 20
                          (x + 2)   - y
(%o2)                     -------------
                                   20
                            (y + x)
```

zeroequiv (*expr*, *v*) Function

Tests whether the expression *expr* in the variable *v* is equivalent to zero, returning `true`, `false`, or `dontknow`.

`zeroequiv` has these restrictions:

1. Do not use functions that Maxima 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.

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)` returns `dontknow` because of the presence of an extra parameter `b`.

6 Expressions

6.1 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*.

```
(%i1) x: 3$
(%i2) (x: x+1, x: x^2);
(%o2)                          16
(%i3) (if (x > 17) then 2 else 4);
(%o3)                           4
(%i4) (if (x > 17) then x: 2 else y: 4, y+x);
(%o4)                          20
```

Even loops in Maxima are expressions, although the value they return is the not too useful `done`.

```
(%i1) y: (x: 1, for i from 1 thru 10 do (x: x*i))$
(%i2) y;
(%o2)                         done
```

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

```
(%i3) y: (x: 1, for i from 1 thru 10 do (x: x*i), x)$
(%i4) y;
(%o4)                        3628800
```

6.2 Assignment

There are two assignment operators in Maxima, `:` 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.

6.3 Complex

A complex expression is specified in Maxima by adding the real part of the expression to `%i` times the imaginary part. Thus the roots of the equation `x^2 - 4*x + 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.

6.4 Nouns and Verbs

Maxima distinguishes between operators which are "nouns" and operators which are "verbs". A verb is an operator which can be executed. A noun is an operator which appears as a symbol in an expression, without being executed. By default, function names are verbs. A verb can be changed into a noun by quoting the function name or applying the `nounify` function. A noun can be changed into a verb by applying the `verbify` function. The evaluation flag `nouns` causes `ev` to evaluate nouns in an expression.

The verb form is distinguished by a leading dollar sign `$` on the corresponding Lisp symbol. In contrast, the noun form is distinguished by a leading percent sign `%` on the corresponding Lisp symbol. Some nouns have special display properties, such as `'integrate` and `'derivative` (returned by `diff`), but most do not. By default, the noun and verb forms of a function are identical when displayed. The global flag `noundisp` causes Maxima to display nouns with a leading quote mark `'`.

See also `noun`, `nouns`, `nounify`, and `verbify`.

Examples:

```
(%i1) foo (x) := x^2;
                                         2
(%o1)                     foo(x) := x
(%i2) foo (42);
(%o2)                         1764
(%i3) 'foo (42);
(%o3)                       foo(42)
(%i4) 'foo (42), nouns;
(%o4)                         1764
(%i5) declare (bar, noun);
(%o5)                         done
(%i6) bar (x) := x/17;
                                         x
(%o6)                    ''bar(x) := --
                                         17
(%i7) bar (52);
(%o7)                       bar(52)
(%i8) bar (52), nouns;
                                52
(%o8)                           --
                                17
(%i9) integrate (1/x, x, 1, 42);
(%o9)                       log(42)
(%i10) 'integrate (1/x, x, 1, 42);
                              42
                             /
                             [   1
(%o10)                       I   - dx
```

```
                                ]    x
                                /
                                 1
(%i11) ev (%, nouns);
(%o11)                           log(42)
```

6.5 Identifiers

Maxima identifiers may comprise alphabetic characters, plus the numerals 0 through 9, plus any special character preceded by the backslash \ character.

A numeral may be the first character of an identifier if it is preceded by a backslash. Numerals which are the second or later characters need not be preceded by a backslash.

A special character may be declared alphabetic by the `declare` function. If so declared, it need not be preceded by a backslash in an identifier. The alphabetic characters are initially `A` through `Z`, `a` through `z`, `%`, and `_`.

Maxima is case-sensitive. The identifiers `foo`, `FOO`, and `Foo` are distinct. See Section 3.2 [Lisp and Maxima], page 9 for more on this point.

A Maxima identifier is a Lisp symbol which begins with a dollar sign `$`. Any other Lisp symbol is preceded by a question mark `?` when it appears in Maxima. See Section 3.2 [Lisp and Maxima], page 9 for more on this point.

Examples:

```
(%i1) %an_ordinary_identifier42;
(%o1)               %an_ordinary_identifier42
(%i2) embedded\ spaces\ in\ an\ identifier;
(%o2)           embedded spaces in an identifier
(%i3) symbolp (%);
(%o3)                          true
(%i4) [foo+bar, foo\+bar];
(%o4)                 [foo + bar, foo+bar]
(%i5) [1729, \1729];
(%o5)                     [1729, 1729]
(%i6) [symbolp (foo\+bar), symbolp (\1729)];
(%o6)                     [true, true]
(%i7) [is (foo\+bar = foo+bar), is (\1729 = 1729)];
(%o7)                    [false, false]
(%i8) baz\~quux;
(%o8)                       baz~quux
(%i9) declare ("~", alphabetic);
(%o9)                         done
(%i10) baz~quux;
(%o10)                      baz~quux
(%i11) [is (foo = FOO), is (FOO = Foo), is (Foo = foo)];
(%o11)                [false, false, false]
(%i12) :lisp (defvar *my-lisp-variable* '$foo)
*MY-LISP-VARIABLE*
(%i12) ?\*my\-lisp\-variable\*;
(%o12)                         foo
```

6.6 Inequality

Maxima has the inequality operators `<`, `<=`, `>=`, `>`, `#`, and `notequal`. See `if` for a description of conditional expressions.

6.7 Syntax

It is possible to define new operators with specified precedence, to undefine existing operators, or to redefine the precedence of existing operators. An operator may be unary prefix or unary postfix, binary infix, n-ary infix, matchfix, or nofix. "Matchfix" means a pair of symbols which enclose their argument or arguments, and "nofix" means an operator which takes no arguments. As examples of the different types of operators, there are the following.

unary prefix
: negation `- a`

unary postfix
: factorial `a!`

binary infix
: exponentiation `a^b`

n-ary infix
: addition `a + b`

matchfix
: list construction `[a, b]`

(There are no built-in nofix operators; for an example of such an operator, see `nofix`.)

The mechanism to define a new operator is straightforward. It is only necessary to declare a function as an operator; the operator function might or might not be defined.

An example of user-defined operators is the following. Note that the explicit function call `"dd" (a)` is equivalent to `dd a`, likewise `"<-" (a, b)` is equivalent to `a <- b`. Note also that the functions `"dd"` and `"<-"` are undefined in this example.

```
(%i1) prefix ("dd");
(%o1)                          dd
(%i2) dd a;
(%o2)                         dd a
(%i3) "dd" (a);
(%o3)                         dd a
(%i4) infix ("<-");
(%o4)                          <-
(%i5) a <- dd b;
(%o5)                      a <- dd b
(%i6) "<-" (a, "dd" (b));
(%o6)                      a <- dd b
```

The Maxima functions which define new operators are summarized in this table, stating the default left and right binding powers (lbp and rbp, respectively). (Binding power determines operator precedence. However, since left and right binding powers can differ, binding power is somewhat more complicated than precedence.) Some of the operation definition functions take additional arguments; see the function descriptions for details.

`prefix` rbp=180

`postfix` lbp=180

`infix` lbp=180, rbp=180

`nary` lbp=180, rbp=180

`matchfix` (binding power not applicable)

`nofix` (binding power not applicable)

For comparison, here are some built-in operators and their left and right binding powers.

```
Operator   lbp     rbp

  :        180     20
  ::       180     20
  :=       180     20
  ::=      180     20
  !        160
  !!       160
  ^        140     139
  .        130     129
  *        120
  /        120     120
  +        100     100
  -        100     134
  =        80      80
  #        80      80
  >        80      80
  >=       80      80
  <        80      80
  <=       80      80
  not              70
  and      65
  or       60
  ,        10
  $        -1
  ;        -1
```

`remove` and `kill` remove operator properties from an atom. `remove ("a", op)` removes only the operator properties of *a*. `kill ("a")` removes all properties of *a*, including the operator properties. Note that the name of the operator must be enclosed in quotation marks.

```
(%i1) infix ("@");
(%o1)                          @
(%i2) "@" (a, b) := a^b;
                                    b
(%o2)                     a @ b := a
(%i3) 5 @ 3;
(%o3)                         125
(%i4) remove ("@", op);
```

```
(%o4)                         done
(%i5) 5 @ 3;
Incorrect syntax: @ is not an infix operator
5 @
 ^
(%i5) "@" (5, 3);
(%o5)                          125
(%i6) infix ("@");
(%o6)                           @
(%i7) 5 @ 3;
(%o7)                          125
(%i8) kill ("@");
(%o8)                         done
(%i9) 5 @ 3;
Incorrect syntax: @ is not an infix operator
5 @
 ^
(%i9) "@" (5, 3);
(%o9)                        @(5, 3)
```

6.8 Definitions for Expressions

at (*expr*, [*eqn_1*, ..., *eqn_n*]) Function
at (*expr*, *eqn*) Function

Evaluates the expression *expr* with the variables assuming the values as specified for them in the list of equations `[`*eqn_1*`, ..., `*eqn_n*`]` or the single equation *eqn*.

If a subexpression depends on any of the variables for which a value is specified but there is no atvalue specified and it can't be otherwise evaluated, then a noun form of the `at` is returned which displays in a two-dimensional form.

`at` carries out multiple substitutions in series, not parallel.

See also `atvalue`. For other functions which carry out substitutions, see also `subst` and `ev`.

Examples:

```
(%i1) atvalue (f(x,y), [x = 0, y = 1], a^2);
                                  2
(%o1)                            a
(%i2) atvalue ('diff (f(x,y), x), x = 0, 1 + y);
(%o2)                         @2 + 1
(%i3) printprops (all, atvalue);
                                  !
                 d                !
                --- (f(@1, @2))!       = @2 + 1
                d@1               !
                                  !@1 = 0

                                      2
```

```
                              f(0, 1) = a

(%o3)                            done
(%i4) diff (4*f(x, y)^2 - u(x, y)^2, x);
                  d                            d
(%o4)  8 f(x, y) (-- (f(x, y))) - 2 u(x, y) (-- (u(x, y)))
                  dx                           dx
(%i5) at (%, [x = 0, y = 1]);
                                         !
              2              d           !
(%o5)     16 a  - 2 u(0, 1) (-- (u(x, y))!            )
                             dx          !
                                         !x = 0, y = 1
```

box (*expr*) Function

box (*expr*, *a*) Function

Returns *expr* enclosed in a box. The return value is an expression with `box` as the operator and *expr* as the argument. A box is drawn on the display when `display2d` is `true`.

`box (`*expr*`,` *a*`)` encloses *expr* in a box labelled by the symbol *a*. The label is truncated if it is longer than the width of the box.

A boxed expression does not evaluate to its content, so boxed expressions are effectively excluded from computations.

`boxchar` is the character used to draw the box in `box` and in the `dpart` and `lpart` functions.

Examples:

boxchar Option variable

Default value: `"`

`boxchar` is the character used to draw the box in the `box` and in the `dpart` and `lpart` functions.

All boxes in an expression are drawn with the current value of `boxchar`; the drawing character is not stored with the box expression.

carg (*z*) Function

Returns the complex argument of *z*. The complex argument is an angle `theta` in `(-%pi, %pi]` such that `r exp (theta %i) =` *z* where `r` is the magnitude of *z*.

`carg` is a computational function, not a simplifying function.

`carg` ignores the declaration `declare (`*x*`, complex)`, and treats *x* as a real variable. This is a bug.

See also `abs` (complex magnitude), `polarform`, `rectform`, `realpart`, and `imagpart`.

Examples:

```
(%i1) carg (1);
(%o1)                               0
(%i2) carg (1 + %i);
```

```
                                %pi
(%o2)                           ---
                                 4
(%i3) carg (exp (%i));
(%o3)                            1
(%i4) carg (exp (%pi * %i));
(%o4)                           %pi
(%i5) carg (exp (3/2 * %pi * %i));
                                 %pi
(%o5)                          - ---
                                  2
(%i6) carg (17 * exp (2 * %i));
(%o6)                            2
```

constant Special operator

`declare (a, constant)` declares *a* to be a constant. See `declare`.

constantp (*expr*) Function

Returns `true` if *expr* is a constant expression, otherwise returns `false`.

An expression is considered a constant expression if its arguments are numbers (including rational numbers, as displayed with `/R/`), symbolic constants such as `%pi`, `%e`, and `%i`, variables bound to a constant or declared constant by `declare`, or functions whose arguments are constant.

`constantp` evaluates its arguments.

Examples:

declare (*a_1, f_1, a_2, f_2, ...*) Function

Assigns the atom *a_i* the flag *f_i*. The *a_i*'s and *f_i*'s may also be lists of atoms and flags respectively in which case each of the atoms gets all of the properties.

`declare` quotes its arguments. `declare` always returns `done`.

The possible flags and their meanings are:

`constant` makes *a_i* a constant as is `%pi`.

`mainvar` makes *a_i* a `mainvar`. The ordering scale for atoms: numbers `<` constants (e.g. `%e`, `%pi`) `<` scalars `<` other variables `<` mainvars.

`scalar` makes *a_i* a scalar.

`nonscalar` makes *a_i* behave as does a list or matrix with respect to the dot operator.

`noun` makes the function *a_i* a noun so that it won't be evaluated automatically.

`evfun` makes *a_i* known to the `ev` function so that it will get applied if its name is mentioned. See `evfun`.

`evflag` makes *a_i* known to the `ev` function so that it will be bound to `true` during the execution of `ev` if it is mentioned. See `evflag`.

`bindtest` causes *a_i* to signal an error if it ever is used in a computation unbound.

Maxima 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
```

The *a_i* and *f_i* may also be lists of objects or features.

`featurep (`*object*`, `*feature*`)` determines if *object* has been declared to have *feature*.

See also `features`.

disolate (*expr, x_1, ..., x_n*) Function

is similar to `isolate (`*expr*`, `*x*`)` 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. This function is autoloaded from '`simplification/disol.mac`'. A demo is available by `demo("disol")$`.

dispform (*expr*) Function

Returns the external representation of *expr* with respect to its main operator. This should be useful in conjunction with `part` which also deals with the external representation. Suppose *expr* is -A . Then the internal representation of *expr* is `"*"`(-1,A), while the external representation is `"-"`(A). `dispform (`*expr*`, all)` converts the entire expression (not just the top-level) to external format. For example, if `expr: sin (sqrt (x))`, then `freeof (sqrt, expr)` and `freeof (sqrt, dispform (expr))` give `true`, while `freeof (sqrt, dispform (expr, all))` gives `false`.

distrib (*expr*) 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.

Examples:

```
(%i1) distrib ((a+b) * (c+d));
(%o1)                 b d + a d + b c + a c
(%i2) multthru ((a+b) * (c+d));
(%o2)                 (b + a) d + (b + a) c
(%i3) distrib (1/((a+b) * (c+d)));
                                1
(%o3)                    ---------------
                         (b + a) (d + c)
(%i4) expand (1/((a+b) * (c+d)), 1, 0);
                                1
(%o4)                 ---------------------
                      b d + a d + b c + a c
```

dpart (*expr, n_1, ..., n_k*) 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.

```
(%i1) dpart (x+y/z^2, 1, 2, 1);
                                  y
(%o1)                            ---- + x
                                    2
                                 """
                                 "z"
                                 """
```

exp (*x*) Function

Represents the exponential function. Instances of `exp (x)` in input are simplified to `%e^x`; `exp` does not appear in simplified expressions.

`demoivre` if `true` causes `%e^(a + b %i)` to simplify to `%e^(a (cos(b) + %i sin(b)))` if `b` is free of `%i`. See `demoivre`.

`%emode`, when `true`, causes `%e^(%pi %i x)` to be simplified. See `%emode`.

`%enumer`, when `true` causes `%e` to be replaced by 2.718... whenever `numer` is `true`. See `%enumer`.

%emode Option variable

Default value: `true`

When `%emode` is `true`, `%e^(%pi %i x)` is simplified as follows.

`%e^(%pi %i x)` simplifies to `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 then further simplified.

For other numerical `x`, `%e^(%pi %i x)` simplifies to `%e^(%pi %i y)` where `y` is `x - 2 k` for some integer `k` such that `abs(y) < 1`.

When `%emode` is `false`, no special simplification of `%e^(%pi %i x)` is carried out.

%enumer Option variable

Default value: `false`

When `%enumer` is `true`, `%e` is replaced by its numeric value 2.718... whenever `numer` is `true`.

When `%enumer` is `false`, this substitution is carried out only if the exponent in `%e^x` evaluates to a number.

See also `ev` and `numer`.

exptisolate Option variable

Default value: `false`

`exptisolate`, when `true`, causes `isolate (expr, var)` to examine exponents of atoms (such as `%e`) which contain `var`.

exptsubst Option variable

Default value: `false`

`exptsubst`, when `true`, permits substitutions such as `y` for `%e^x` in `%e^(a x)`.

freeof (*x_1, ..., x_n, expr*) Function

`freeof (`*x_1*`, `*expr*`)` Returns `true` if no subexpression of *expr* is equal to *x_1* or if *x_1* occurs only as a dummy variable in *expr*, and returns `false` otherwise.

`freeof (`*x_1*`, ..., `*x_n*`, `*expr*`)` is equivalent to `freeof (`*x_1*`, `*expr*`) and ... and freeof (`*x_n*`, `*expr*`)`.

The arguments *x_1, ..., x_n* may be names of functions and variables, subscripted names, operators (enclosed in double quotes), or general expressions. `freeof` evaluates its arguments.

`freeof` operates only on *expr* as it stands (after simplification and evaluation) and does not attempt to determine if some equivalent expression would give a different result. In particular, simplification may yield an equivalent but different expression which comprises some different elements than the original form of *expr*.

A variable is a dummy variable in an expression if it has no binding outside of the expression. Dummy variables recognized by `freeof` are the index of a sum or product, the limit variable in `limit`, the integration variable in the definite integral form of `integrate`, the original variable in `laplace`, formal variables in `at` expressions, and arguments in `lambda` expressions. Local variables in `block` are not recognized by `freeof` as dummy variables; this is a bug.

The indefinite form of `integrate` is *not* free of its variable of integration.

- Arguments are names of functions, variables, subscripted names, operators, and expressions. `freeof (a, b, expr)` is equivalent to `freeof (a, expr) and freeof (b, expr)`.

```
(%i1) expr: z^3 * cos (a[1]) * b^(c+d);
                                      d + c  3
(%o1)                   cos(a ) b      z
                             1
(%i2) freeof (z, expr);
(%o2)                         false
(%i3) freeof (cos, expr);
(%o3)                         false
(%i4) freeof (a[1], expr);
(%o4)                         false
(%i5) freeof (cos (a[1]), expr);
(%o5)                         false
(%i6) freeof (b^(c+d), expr);
(%o6)                         false
(%i7) freeof ("^", expr);
(%o7)                         false
(%i8) freeof (w, sin, a[2], sin (a[2]), b*(c+d), expr);
(%o8)                         true
```

- `freeof` evaluates its arguments.

```
(%i1) expr: (a+b)^5$
(%i2) c: a$
(%i3) freeof (c, expr);
(%o3)                          false
```

- `freeof` does not consider equivalent expressions. Simplification may yield an equivalent but different expression.

```
(%i1) expr: (a+b)^5$
(%i2) expand (expr);
          5        4        2  3        3  2       4      5
(%o2)    b  + 5 a b  + 10 a  b  + 10 a  b  + 5 a  b + a
(%i3) freeof (a+b, %);
(%o3)                          true
(%i4) freeof (a+b, expr);
(%o4)                          false
(%i5) exp (x);
                                  x
(%o5)                           %e
(%i6) freeof (exp, exp (x));
(%o6)                          true
```

- A summation or definite integral is free of its dummy variable. An indefinite integral is not free of its variable of integration.

```
(%i1) freeof (i, 'sum (f(i), i, 0, n));
(%o1)                          true
(%i2) freeof (x, 'integrate (x^2, x, 0, 1));
(%o2)                          true
(%i3) freeof (x, 'integrate (x^2, x));
(%o3)                          false
```

genfact (*x*, *y*, *z*) Function

Returns the generalized factorial, defined as `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 (*expr*) Function

Returns the imaginary part of the expression *expr*.

`imagpart` is a computational function, not a simplifying function.

See also `abs`, `carg`, `polarform`, `rectform`, and `realpart`.

infix (*op*) Function
infix (*op*, *lbp*, *rbp*) Function
infix (*op*, *lbp*, *rbp*, *lpos*, *rpos*, *pos*) Function

Declares *op* to be an infix operator. An infix operator is a function of two arguments, with the name of the function written between the arguments. For example, the subtraction operator `-` is an infix operator.

`infix (op)` declares *op* to be an infix operator with default binding powers (left and right both equal to 180) and parts of speech (left and right both equal to `any`).

`infix (`*op*`,` *lbp*`,` *rbp*`)` declares *op* to be an infix operator with stated left and right binding powers and default parts of speech (left and right both equal to `any`).

`infix (`*op*`,` *lbp*`,` *rbp*`,` *lpos*`,` *rpos*`,` *pos*`)` declares *op* to be an infix operator with stated left and right binding powers and parts of speech.

The precedence of *op* with respect to other operators derives from the left and right binding powers of the operators in question. If the left and right binding powers of *op* are both greater the left and right binding powers of some other operator, then *op* takes precedence over the other operator. If the binding powers are not both greater or less, some more complicated relation holds.

The associativity of *op* depends on its binding powers. Greater left binding power (*lbp*) implies an instance of *op* is evaluated before other operators to its left in an expression, while greater right binding power (*rbp*) implies an instance of *op* is evaluated before other operators to its right in an expression. Thus greater *lbp* makes *op* right-associative, while greater *rbp* makes *op* left-associative. If *lbp* is equal to *rbp*, *op* is left-associative.

See also `Syntax`.

Examples:

- If the left and right binding powers of *op* are both greater the left and right binding powers of some other operator, then *op* takes precedence over the other operator.

```
(%i1) "@"(a, b) := sconcat("(", a, ",", b, ")")$
(%i2) :lisp (get '$+ 'lbp)
100
(%i2) :lisp (get '$+ 'rbp)
100
(%i2) infix ("@", 101, 101)$
(%i3) 1 + a@b + 2;
(%o3)                        (a,b) + 3
(%i4) infix ("@", 99, 99)$
(%i5) 1 + a@b + 2;
(%o5)                        (a+1,b+2)
```

- Greater *lbp* makes *op* right-associative, while greater *rbp* makes *op* left-associative.

```
(%i1) "@"(a, b) := sconcat("(", a, ",", b, ")")$
(%i2) infix ("@", 100, 99)$
(%i3) foo @ bar @ baz;
(%o3)                     (foo,(bar,baz))
(%i4) infix ("@", 100, 101)$
(%i5) foo @ bar @ baz;
(%o5)                     ((foo,bar),baz)
```

inflag Option variable

Default value: `false`

When `inflag` is `true`, functions for part extraction inspect the internal form of `expr`.

Note that the simplifier re-orders expressions. Thus `first (x + y)` returns `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` or `substpart` is the same as calling `inpart` or `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 (*expr, n_1, ..., n_k*) 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 (%o9,0) = "+" then ...`.

Examples:

```
(%i1) x + y + w*z;
(%o1)                        w z + y + x
(%i2) inpart (%, 3, 2);
(%o2)                              z
(%i3) part (%th (2), 1, 2);
(%o3)                              z
(%i4) 'limit (f(x)^g(x+1), x, 0, minus);
                                     g(x + 1)
(%o4)                 limit   f(x)
                      x -> 0-
(%i5) inpart (%, 1, 2);
(%o5)                         g(x + 1)
```

isolate (*expr, x*) Function

Returns *expr* with subexpressions which are sums and which do not contain var replaced by intermediate expression labels (these being atomic symbols like `%t1`, `%t2`, ...). 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` (default value: `false`) if `true` will cause `isolate` to examine exponents of atoms (like `%e`) which contain var.

`isolate_wrt_times` if `true`, then `isolate` will also isolate wrt products. See `isolate_wrt_times`.

Do `example (isolate)` for examples.

isolate_wrt_times Option variable

Default value: `false`

When `isolate_wrt_times` is `true`, `isolate` will also isolate wrt products. E.g. compare both settings of the switch on

```
(%i1) isolate_wrt_times: true$
(%i2) isolate (expand ((a+b+c)^2), c);

(%t2)                          2 a


(%t3)                          2 b


                          2            2
(%t4)                    b  + 2 a b + a

                     2
(%o4)               c  + %t3 c + %t2 c + %t4
(%i4) isolate_wrt_times: false$
(%i5) isolate (expand ((a+b+c)^2), c);
                     2
(%o5)               c  + 2 b c + 2 a c + %t4
```

listconstvars Option variable

Default value: `false`

When `listconstvars` is `true`, it 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 Option variable

Default value: `true`

When `listdummyvars` is `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 `freeof`. "Dummy variables" are mathematical things like the index of a sum or product, the limit variable, and the definite integration variable.) Example:

```
(%i1) listdummyvars: true$
(%i2) listofvars ('sum(f(i), i, 0, n));
(%o2)                        [i, n]
(%i3) listdummyvars: false$
(%i4) listofvars ('sum(f(i), i, 0, n));
(%o4)                          [n]
```

listofvars (*expr*) Function

Returns a list of the variables in *expr*.

`listconstvars` if `true` causes `listofvars` to include `%e`, `%pi`, `%i`, and any variables declared constant in the list it returns if they appear in *expr*. The default is to omit these.

```
(%i1) listofvars (f (x[1]+y) / g^(2+a));
(%o1)                     [g, a, x , y]
                                  1
```

lfreeof (*list, expr*) Function

For each member *m* of list, calls `freeof (m, expr)`. It returns `false` if any call to `freeof` does and `true` otherwise.

lopow (*expr, x*) Function

Returns the lowest exponent of *x* which explicitly appears in *expr*. Thus

```
(%i1) lopow ((x+y)^2 + (x+y)^a, x+y);
(%o1)                          min(a, 2)
```

lpart (*label, expr, n_1, ..., n_k*) 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 (*expr*) Function

multthru (*expr_1, expr_2*) Function

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

`multthru (expr_1, expr_2)` multiplies each term in *expr_2* (which should be a sum or an equation) by *expr_1*. If *expr_1* is not itself a sum then this form is equivalent to `multthru (expr_1*expr_2)`.

```
(%i1) x/(x-y)^2 - 1/(x-y) - f(x)/(x-y)^3;
                        1        x         f(x)
(%o1)               - ----- + -------- - --------
                      x - y          2          3
                              (x - y)    (x - y)
(%i2) multthru ((x-y)^3, %);
                              2
(%o2)               - (x - y)  + x (x - y) - f(x)
(%i3) ratexpand (%);
                              2
(%o3)                      - y  + x y - f(x)
(%i4) ((a+b)^10*s^2 + 2*a*b*s + (a*b)^2)/(a*b*s^2);
                         10  2              2  2
                (b + a)   s  + 2 a b s + a  b
(%o4)           ------------------------------
                                  2
                             a b s
(%i5) multthru (%);  /* note that this does not expand (b+a)^10 */
                                           10
                       2   a b   (b + a)
(%o5)                  - + --- + ---------
                       s    2       a b
```

```
                     s
(%i6) multthru (a.(b+c.(d+e)+f));
(%o6)            a . f + a . c . (e + d) + a . b
(%i7) expand (a.(b+c.(d+e)+f));
(%o7)        a . f + a . c . e + a . c . d + a . b
```

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 (*expr*) Function

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

op (*expr*) Function

Returns the main operator of the expression *expr*. `op (`*expr*`)` is equivalent to `part (`*expr*`, 0)`.

`op` returns a string if the main operator is a built-in or user-defined prefix, binary or n-ary infix, postfix, matchfix, or nofix operator. Otherwise `op` returns a symbol.

`op` observes the value of the global flag `inflag`.

`op` evaluates it argument.

See also `args`.

Examples:

```
(%i1) ?stringdisp: true$
(%i2) op (a * b * c);
(%o2)                           "*"
(%i3) op (a * b + c);
(%o3)                           "+"
(%i4) op ('sin (a + b));
(%o4)                           sin
(%i5) op (a!);
(%o5)                           "!"
(%i6) op (-a);
(%o6)                           "-"
(%i7) op ([a, b, c]);
(%o7)                           "["
(%i8) op ('(if a > b then c else d));
(%o8)                          "if"
(%i9) op ('foo (a));
(%o9)                           foo
(%i10) prefix (foo);
(%o10)                         "foo"
(%i11) op (foo a);
(%o11)                         "foo"
```

operatorp (*expr, op*) Function
operatorp (*expr*, [*op_1, ..., op_n*]) Function

`operatorp (expr, op)` returns `true` if *op* is equal to the operator of *expr*.

`operatorp (expr, [op_1, ..., op_n])` returns `true` if some element *op_1*, ..., *op_n* is equal to the operator of *expr*.

optimize (*expr*) Function

Returns an expression that produces the same value and side effects as *expr* 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 Option variable

Default value: `%`

`optimprefix` is the prefix used for generated symbols by the `optimize` command.

ordergreat (*v_1, ..., v_n*) Function

Sets up aliases for the variables *v_1*, ..., *v_n* such that *v_1* > *v_2* > ... > *v_n*, and *v_n* > any other variable not mentioned as an argument.

See also `orderless`.

ordergreatp (*expr_1, expr_2*) Function

Returns `true` if *expr_2* precedes *expr_1* in the ordering set up with the `ordergreat` function.

orderless (*v_1, ..., v_n*) Function

Sets up aliases for the variables *v_1*, ..., *v_n* such that *v_1* < *v_2* < ... < *v_n*, and *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` < ... < last argument to `orderless` < variables which begin with A < ... < variables which begin with Z < last argument to `ordergreat` < ... < first argument to `ordergreat` < declared `mainvars`.

See also `ordergreat` and `mainvar`.

orderlessp (*expr_1, expr_2*) Function

Returns `true` if *expr_1* precedes *expr_2* in the ordering set up by the `orderless` command.

part (*expr, n_1, ..., n_k*) Function

Returns parts of the displayed form of `expr`. It obtains the part of `expr` as specified by the indices *n_1*, ..., *n_k*. First part *n_1* of `expr` is obtained, then part *n_2* of that, etc. The result is part *n_k* of ... part *n_2* of part *n_1* of `expr`.

`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` is set to `true` then `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

Example: `part (z+2*y, 2, 1)` yields 2.

`example (part)` displays additional examples.

partition (*expr, x*) Function

Returns a list of two expressions. They are (1) the factors of *expr* (if it is a product), the terms of *expr* (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.

```
(%i1) partition (2*a*x*f(x), x);
(%o1)                     [2 a, x f(x)]
(%i2) partition (a+b, x);
(%o2)                      [b + a, 0]
(%i3) partition ([a, b, f(a), c], a);
(%o3)                 [[b, c], [a, f(a)]]
```

partswitch Option variable

Default value: `false`

When `partswitch` is `true`, `end` is returned when a selected part of an expression doesn't exist, otherwise an error message is given.

pickapart (*expr, n*) Function

Assigns intermediate expression labels to subexpressions of *expr* at depth *n*, an integer. Subexpressions at greater or lesser depths are not assigned labels. `pickapart` returns an expression in terms of intermediate expressions equivalent to the original expression *expr*.

See also `part`, `dpart`, `lpart`, `inpart`, and `reveal`.

Examples:

```
(%i1) expr: (a+b)/2 + sin (x^2)/3 - log (1 + sqrt(x+1));
                                          2
                                     sin(x )   b + a
(%o1)       - log(sqrt(x + 1) + 1) + ------- + -----
                                        3        2
(%i2) pickapart (expr, 0);

                                          2
                                     sin(x )   b + a
(%t2)       - log(sqrt(x + 1) + 1) + ------- + -----
                                        3        2
```

```
(%o2)                                %t2
(%i3) pickapart (expr, 1);

(%t3)                - log(sqrt(x + 1) + 1)


                                      2
                                 sin(x )
(%t4)                            -------
                                    3


                                  b + a
(%t5)                             -----
                                    2

(%o5)                        %t5 + %t4 + %t3
(%i5) pickapart (expr, 2);

(%t6)                  log(sqrt(x + 1) + 1)


                                      2
(%t7)                            sin(x )


(%t8)                             b + a

                             %t8   %t7
(%o8)                        --- + --- - %t6
                              2     3
(%i8) pickapart (expr, 3);

(%t9)                        sqrt(x + 1) + 1


                                    2
(%t10)                             x

                  b + a              sin(%t10)
(%o10)            ----- - log(%t9) + ---------
                    2                    3
(%i10) pickapart (expr, 4);

(%t11)                         sqrt(x + 1)

                       2
                 sin(x )   b + a
```

```
(%o11)          ------- + ----- - log(%t11 + 1)
                   3        2
(%i11) pickapart (expr, 5);

(%t12)                        x + 1

                  2
             sin(x )   b + a
(%o12)       ------- + ----- - log(sqrt(%t12) + 1)
                3        2
(%i12) pickapart (expr, 6);
                 2
            sin(x )   b + a
(%o12)      ------- + ----- - log(sqrt(x + 1) + 1)
               3        2
```

piece System 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.

polarform (*expr*) Function

Returns an expression `r %e^(%i theta)` equivalent to *expr*, such that `r` and `theta` are purely real.

powers (*expr, x*) Function

Gives the powers of *x* occuring in *expr*.

`load (powers)` loads this function.

product (*expr, i, i_0, i_1*) Function

Returns the product of the values of `expr` as the index *i* varies from *i_0* to *i_1*. The evaluation is similar to that of `sum`.

If *i_1* is one less than *i_0*, the product is an "empty product" and `product` returns 1 rather than reporting an error. See also `prodhack`.

Maxima does not simplify products.

Example:

```
(%i1) product (x + i*(i+1)/2, i, 1, 4);
(%o1)          (x + 1) (x + 3) (x + 6) (x + 10)
```

realpart (*expr*) Function

Returns the real part of *expr*. `realpart` and `imagpart` will work on expressions involving trigonometic and hyperbolic functions, as well as square root, logarithm, and exponentiation.

rectform (*expr*) Function

Returns an expression `a + b %i` equivalent to *expr*, such that *a* and *b* are purely real.

rembox (*expr, unlabelled*) Function
rembox (*expr, label*) Function
rembox (*expr*) Function

Removes boxes from *expr*.

`rembox (`*expr*`, unlabelled)` removes all unlabelled boxes from *expr*.

`rembox (`*expr*`, `*label*`)` removes only boxes bearing *label*.

`rembox (`*expr*`)` removes all boxes, labelled and unlabelled.

Boxes are drawn by the `box`, `dpart`, and `lpart` functions.

Examples:

```
(%i1) expr: (a*d - b*c)/h^2 + sin(%pi*x);
                                  a d - b c
(%o1)                sin(%pi x) + ---------
                                      2
                                     h
(%i2) dpart (dpart (expr, 1, 1), 2, 2);
                        """""""    a d - b c
(%o2)               sin("%pi x") + ---------
                        """""""      """"
                                     " 2"
                                     "h "
                                     """"
(%i3) expr2: lpart (BAR, lpart (FOO, %, 1), 2);
                  FOO"""""""""""   BAR""""""""
                  "    """"""" "   "a d - b c"
(%o3)             "sin("%pi x")" + "---------"
                  "    """"""" "   "  """"   "
                  """"""""""""""   "  " 2"   "
                                   "  "h "   "
                                   "  """"   "
                                   """""""""""
(%i4) rembox (expr2, unlabelled);
                                  BAR""""""""
                   FOO"""""""""   "a d - b c"
(%o4)              "sin(%pi x)" + "---------"
                   """"""""""""   "    2    "
                                  "   h     "
                                  """""""""""
(%i5) rembox (expr2, FOO);
                                  BAR""""""""
                       """""""    "a d - b c"
(%o5)              sin("%pi x") + "---------"
                       """""""    "  """"   "
                                  "  " 2"   "
                                  "  "h "   "
                                  "  """"   "
                                  """""""""""
(%i6) rembox (expr2, BAR);
                   FOO"""""""""""
```

```
                     "    """"""" "   a d - b c
(%o6)                "sin("%pi x")" + ---------
                     "    """"""" "      """"
                     """"""""""""""      " 2"
                                         "h "
                                         """"
(%i7) rembox (expr2);
                                    a d - b c
(%o7)                 sin(%pi x) + ---------
                                        2
                                       h
```

sum (*expr, i, i_0, i_1*) Function

Represents a summation of the values of *expr* as the index *i* varies from *i_0* to *i_1*. Sums may be differentiated, added, subtracted, or multiplied with some automatic simplification being performed. The noun form `'sum` is displayed in sigma notation.

If the upper and lower limits differ by an integer, the summand *expr* is evaluated for each value of the summation index *i*, and the results are added together.

Otherwise, if the `simpsum` is `true` the summation is simplified. Simplification may sometimes yield a closed form. If the evaluation flag `simpsum` is `false` or simplification fails, the result is a noun form `'sum`.

`sum` evaluates *i_0* and *i_1* and quotes *i*. The summand *expr* is quoted under some circumstances, or evaluated to greater or lesser degree in others.

If *i_1* is one less than *i_0*, the sum is a considered an "empty sum" and `sum` returns 0 rather than reporting an error. See also `sumhack`.

When the evaluation flag `cauchysum` is `true`, the product of summations is expressed as a Cauchy product, in which the index of the inner summation is a function of the index of the outer one, rather than varying independently.

The global variable `genindex` is the alphabetic prefix used to generate the next index of summation, when an automatically generated index is needed.

`gensumnum` is the numeric suffix used to generate the next index of summation, when an automatically generated index is needed. When `gensumnum` is `false`, an automatically-generated index is only `genindex` with no numeric suffix.

See also `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns`, and `evflag`.

Examples:

```
(%i1) sum (i^2, i, 1, 7);
(%o1)                          140
(%i2) sum (a[i], i, 1, 7);
(%o2)           a  + a  + a  + a  + a  + a  + a
                 7    6    5    4    3    2    1
(%i3) sum (a(i), i, 1, 7);
(%o3)    a(7) + a(6) + a(5) + a(4) + a(3) + a(2) + a(1)
(%i4) sum (a(i), i, 1, n);
                            n
                           ====
                           \
```

```
(%o4)                          >    a(i)
                              /
                              ====
                              i = 1
(%i5) ev (sum (2^i + i^2, i, 0, n), simpsum);
                                    3      2
                   n + 1   2 n  + 3 n  + n
(%o5)             2      + --------------- - 1
                                  6
(%i6) ev (sum (1/3^i, i, 1, inf), simpsum);
                                 1
(%o6)                            -
                                 2
(%i7) ev (sum (i^2, i, 1, 4) * sum (1/i^2, i, 1, inf), simpsum);
                                   2
(%o7)                          5 %pi
```

lsum (*expr*, *x*, *L*) Function

Represents the sum of *expr* for each element *x* in *L*.

A noun form `'lsum` is returned if the argument *L* does not evaluate to a list.

Examples:

```
(%i1) lsum (x^i, i, [1, 2, 7]);
                               7    2
(%o1)                         x  + x  + x
(%i2) lsum (i^2, i, rootsof (x^3 - 1));
                        ====
                        \      2
(%o2)                    >    i
                        /
                        ====
                                      3
                        i in rootsof(x  - 1)
```

verbify (*f*) Function

Returns the verb form of the function name *f*.

See also `verb`, `noun`, and `nounify`.

Examples:

```
(%i1) verbify ('foo);
(%o1)                               foo
(%i2) :lisp $%
$FOO
(%i2) nounify (foo);
(%o2)                               foo
(%i3) :lisp $%
%FOO
```

7 Simplification

7.1 Definitions for Simplification

askexp System variable

When `asksign` is called, `askexp` is the expression `asksign` is testing.

At one time, it was possible for a user to inspect `askexp` by entering a Maxima break with control-A.

askinteger (*expr, integer*) Function
askinteger (*expr*) Function
askinteger (*expr, even*) Function
askinteger (*expr, odd*) Function

`askinteger (`*expr*`, integer)` attempts to determine from the `assume` database whether *expr* is an integer. `askinteger` prompts the user if it cannot tell otherwise, and attempt to install the information in the database if possible. `askinteger (`*expr*`)` is equivalent to `askinteger (`*expr*`, integer)`.

`askinteger (`*expr*`, even)` and `askinteger (`*expr*`, odd)` likewise attempt to determine if *expr* is an even integer or odd integer, respectively.

asksign (*expr*) 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 deduction. The user's answer is recorded in the data base for the duration of the current computation. The return value of `asksign` is one of `pos`, `neg`, or `zero`.

demoivre (*expr*) Function
demoivre Option variable

The function `demoivre (expr)` converts one expression without setting the global variable `demoivre`.

When the variable `demoivre` is `true`, complex exponentials are converted into equivalent expressions in terms of circular functions: `exp (a + b*%i)` simplifies to `%e^a * (cos(b) + %i*sin(b))` if `b` is free of `%i`. `a` and `b` are not expanded.

The default value of `demoivre` is `false`.

`exponentialize` converts circular and hyperbolic functions to exponential form. `demoivre` and `exponentialize` cannot both be true at the same time.

domain Option variable

Default value: `real`

When `domain` is set to `complex`, `sqrt (x^2)` will remain `sqrt (x^2)` instead of returning `abs(x)`.

expand (*expr*) Function
expand (*expr, p, n*) Function

Expand expression *expr*. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplication (commutative and non-commutative) are distributed over addition at all levels of *expr*.

For polynomials one should usually use `ratexpand` which uses a more efficient algorithm.

`maxnegex` and `maxposex` control the maximum negative and positive exponents, respectively, which will expand.

`expand (`*expr*`, `*p*`, `*n*`)` expands *expr*, using *p* for `maxposex` and *n* for `maxnegex`. This is useful in order to expand part but not all of an expression.

`expon` - 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` - 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` causes expansion.

The file '`simplification/facexp.mac`' contains several related functions (in particular `facsum`, `factorfacsum` and `collectterms`, which are autoloaded) and variables (`nextlayerfactor` and `facsum_combine`) that provide the user with the ability to structure expressions by controlled expansion. Brief function descriptions are available in '`simplification/facexp.usg`'. A demo is available by doing `demo("facexp")`.

expandwrt (*expr, x_1, ..., x_n*) Function

Expands expression `expr` with respect to the variables *x_1*, ..., *x_n*. All products involving the variables appear explicitly. The form returned will be free of products of sums of expressions that are not free of the variables. *x_1*, ..., *x_n* may be variables, operators, or expressions.

By default, denominators are not expanded, but this can be controlled by means of the switch `expandwrt_denom`.

This function is autoloaded from '`simplification/stopex.mac`'.

expandwrt_denom Option variable

Default value: `false`

`expandwrt_denom` 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.

expandwrt_factored (*expr, x_1, ..., x_n*) Function

is similar to `expandwrt`, but treats expressions that are products somewhat differently. `expandwrt_factored` expands only on those factors of `expr` that contain the variables *x_1, ..., x_n*.

This function is autoloaded from '`simplification/stopex.mac`'.

expon Option variable

Default value: 0

`expon` is 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 (*expr*) Function

exponentialize Option variable

The function `exponentialize (expr)` converts circular and hyperbolic functions in *expr* to exponentials, without setting the global variable `exponentialize`.

When the variable `exponentialize` is `true`, all circular and hyperbolic functions are converted to exponential form. The default value is `false`.

`demoivre` converts complex exponentials into circular functions. `exponentialize` and `demoivre` cannot both be true at the same time.

expop Option variable

Default value: 0

`expop` is 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 Option variable

Default value: -1

`factlim` specifies the highest factorial which is automatically expanded. If it is -1 then all integers are expanded.

intosum (*expr*) Function

Moves multiplicative factors outside a summation to inside. 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

`declare (g, lassociative)` tells the Maxima 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 Maxima's operator properties. For univariate `f` so declared, "expansion" `f(x + y)` yields `f(x) + f(y)`, `f(a*x)` yields `a*f(x)` takes place where `a` is a "constant". For functions of two or more arguments, "linearity" is defined to be as in the case of `sum` or `integrate`, i.e., `f (a*x + b, x)` yields `a*f(x,x) + b*f(1,x)` for `a` and `b` free of `x`.

`linear` is equivalent to `additive` and `outative`. See also `opproperties`.

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 (expr, 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 Option variable

Default value: 10000

`maxapplydepth` is the maximum depth to which `apply1` and `apply2` will delve.

maxapplyheight Option variable

Default value: 10000

`maxapplyheight` is the maximum height to which `applyb1` will reach before giving up.

maxnegex Option variable

Default value: 1000

`maxnegex` is the largest negative exponent which will be expanded by the `expand` command (see also `maxposex`).

maxposex Option variable

Default value: 1000

`maxposex` is the largest exponent which will be expanded with the `expand` command (see also `maxnegex`).

multiplicative Declaration

`declare (f, multiplicative)` tells the Maxima simplifier that `f` is multiplicative.

1. If `f` is univariate, whenever the simplifier encounters `f` applied to a product, `f` distributes over that product. E.g., `f(x*y)` simplifies 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`, e.g., `f (g(x) * h(x), x)` simplifies 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, m, n)`.

negdistrib Option variable

Default value: `true`

When `negdistrib` is `true`, -1 distributes 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 Maxima.

negsumdispflag Option variable

Default value: `true`

When `negsumdispflag` is `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

`noeval` 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

`noun` is 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 Option variable

Default value: `false`

When `noundisp` is `true`, nouns display with a single quote. This switch is always `true` when displaying function definitions.

nouns Special symbol

`nouns` is an `evflag`. When used as an option to the `ev` command, `nouns` 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

`numer` causes some mathematical functions (including exponentiation) with numerical arguments to be evaluated in floating point. It causes variables in `expr` which have been given numervals to be replaced by their values. It also sets the `float` switch on.

numerval (*x_1, expr_1, ..., var_n, expr_n*) Function

Declares the variables `x_1`, ..., *x_n* to have numeric values equal to `expr_1`, ..., `expr_n`. The numeric value is evaluated and substituted for the variable in any expressions in which the variable occurs if the `numer` flag is `true`. See also `ev`.

The expressions `expr_1`, ..., `expr_n` can be any expressions, not necessarily numeric.

opproperties System variable

`opproperties` is the list of the special operator properties recognized by the Maxima simplifier: `linear`, `additive`, `multiplicative`, `outative`, `evenfun`, `oddfun`, `commutative`, `symmetric`, `antisymmetric`, `nary`, `lassociative`, `rassociative`.

opsubst Option variable

Default value: `true`

When `opsubst` is `false`, `subst` does not attempt to substitute into the operator of an expression. E.g., `(opsubst: false, subst (x^2, r, r+r[0]))` will work.

outative Declaration

`declare (f, outative)` tells the Maxima simplifier that constant factors in the argument of `f` can be pulled out.

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. E.g., `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`.

`sum`, `integrate`, and `limit` are all `outative`.

posfun Declaration

`declare (f, posfun)` declares `f` to be a positive function. `is (f(x) > 0)` yields `true`.

prodhack Option variable

Default value: `false`

When `prodhack` is `true`, the identity `product (f(i), i, a, b) = 1/product (f(i), i, b+1, a-1)` is applied if `a` is greater than `b`. For example, `product (f(i), i, 3, 1)` yields `1/f(2)`.

radcan (*expr*) Function

Simplifies *expr*, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, `radcan` produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by `radcan` to zero.

For some expressions `radcan` is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.

When `%e_to_numlog` is `true`, `%e^(r*log(expr))` simplifies to `expr^r` if `r` is a rational number.

When `radexpand` is `false`, certain transformations are inhibited. `radcan (sqrt (1-x))` remains `sqrt (1-x)` and is not simplified to `%i sqrt (x-1)`. `radcan (sqrt (x^2 - 2*x + 11))` remains `sqrt (x^2 - 2*x + 1)` and is not simplified to `x - 1`.

`example (radcan)` displays some examples.

radexpand Option variable

Default value: `true`

`radexpand` controls some simplifications of radicals.

When `radexpand` is `all`, causes nth roots of factors of a product which are powers of n to be pulled outside of the radical. E.g. if `radexpand` is `all`, `sqrt (16*x^2)` simplifies to `4*x`.

More particularly, consider `sqrt (x^2)`.

- If `radexpand` is `all` or `assume (x > 0)` has been executed, `sqrt(x^2)` simplifies to `x`.
- If `radexpand` is `true` and `domain` is `real` (its default), `sqrt(x^2)` simplifies to `abs(x)`.
- If `radexpand` is `false`, or `radexpand` is `true` and `domain` is `complex`, `sqrt(x^2)` is not simplified.

Note that `domain` only matters when `radexpand` is `true`.

radsubstflag Option variable

Default value: `false`

`radsubstflag`, if `true`, permits `ratsubst` to make substitutions such as `u` for `sqrt (x)` in `x`.

rassociative Declaration

`declare (g, rassociative)` tells the Maxima simplifier that `g` is right-associative. E.g., `g(g(a, b), g(c, d))` simplifies to `g(a, g(b, g(c, d)))`.

scsimp (*expr*, *rule_1*, ..., *rule_n*) Function

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

`example (scsimp)` displays some examples.

simpsum Option variable

Default value: `false`

When `simpsum` is `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 the quoted form `'sum` is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics.

sumcontract (*expr*) Function

Combines all sums of an addition that have upper and lower bounds that differ by constants. The result is 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` combines all compatible sums and uses 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 Option variable

Default value: `false`

When `sumexpand` is `true`, products of sums and exponentiated sums simplify to nested sums.

See also `cauchysum`.

Examples:

```
(%i1) sumexpand: true$
(%i2) sum (f (i), i, 0, m) * sum (g (j), j, 0, n);
                     m      n
                    ====   ====
                    \      \
(%o2)                >      >     f(i1) g(i2)
                    /      /
                    ====   ====
                    i1 = 0 i2 = 0
(%i3) sum (f (i), i, 0, m)^2;
                     m      m
                    ====   ====
                    \      \
(%o3)                >      >     f(i3) f(i4)
                    /      /
                    ====   ====
                    i3 = 0 i4 = 0
```

sumhack Option variable

Default value: `false`

When `sumhack` is `true`, the identity `sum (f(i), i, a, b) = - sum (f(i), i, b+1, a-1)` is applied if `a` is greater than `b`. For example, `(sumhack: true, sum (f(i), i, 3, 1))` yields `-f(2)`.

sumsplitfact Option variable

Default value: `true`

When `sumsplitfact` is `false`, `minfactorial` is applied after a `factcomb`.

symmetric Declaration

`declare (h, symmetric)` tells the Maxima simplifier that `h` is a symmetric function. E.g., `h (x, z, y)` simplifies to `h (x, y, z)`.

`commutative` is synonymous with `symmetric`.

unknown (*expr*) Function

Returns `true` if and only if *expr* contains an operator or function not recognized by the Maxima simplifier.

8 Plotting

8.1 Definitions for Plotting

in_netmath Option variable

Default value: `false`

When `in_netmath` is `true`, `plot3d` prints OpenMath output to the console if `plot_format` is `openmath`; otherwise `in_netmath` (even if `true`) has no effect.

`in_netmath` has no effect on `plot2d`.

openplot_curves (*list*, *rest_options*) Function

Takes a list of curves such as

```
[[x1, y1, x2, y2, ...], [u1, v1, u2, v2, ...], ..]
```

or

```
[[[x1, y1], [x2, y2], ...], ...]
```

and plots them. This is similar to xgraph_curves, but uses the open plot routines. Addtional 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 {xaxislabel {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`, `xaxislabel`, and `yaxislabel`.

plot2d (*expr*, *range*, ..., *options*, ...) Function
plot2d (*parametric_expr*) Function
plot2d (*discrete_expr*) Function
plot2d ([*expr_1*, ..., *expr_n*], *x_range*, *y_range*) Function
plot2d ([*expr_1*, ..., *expr_n*], *x_range*) Function
plot2d (*expr*, *x_range*, *y_range*) Function
plot2d (*expr*, *x_range*) Function

Displays a plot of one or more expressions as a function of one variable.

In all cases, *expr* is an expression to be plotted on the vertical axis as a function of one variable. *x_range*, the range of the horizontal axis, is a list of the form `[`*variable*`,` *min*`,` *max*`]`, where *variable* is a variable which appears in *expr*. *y_range*, the range of the vertical axis, is a list of the form `[y,` *min*`,` *max*`]`.

`plot2d (`*expr*`,` *x_range*`)` plots *expr* as a function of the variable named in *x_range*, over the range specified in *x_range*. If the vertical range is not otherwise specified by `set_plot_option`, it is chosen automatically. All options are assumed to have default values unless otherwise specified by `set_plot_option`.

`plot2d (`*expr*`,` *x_range*`,` *y_range*`)` plots *expr* as a function of the variable named in *x_range*, over the range specified in *x_range*. The vertical range is set to *y_range*. All options are assumed to have default values unless otherwise specified by `set_plot_option`.

`plot2d ([`*expr_1*`, ..., `*expr_n*`], `*x_range*`)` plots *expr_1*, ..., *expr_n* as a function of the variable named in *x_range*, over the range specified in *x_range*. If the vertical range is not otherwise specified by `set_plot_option`, it is chosen automatically. All options are assumed to have default values unless otherwise specified by `set_plot_option`.

`plot2d ([`*expr_1*`, ..., `*expr_n*`], `*x_range*`, `*y_range*`)` plots *expr_1*, ..., *expr_n* as a function of the variable named in *x_range*, over the range specified in *x_range*. The vertical range is set to *y_range*. All options are assumed to have default values unless otherwise specified by `set_plot_option`.

Examples:

```
(%i1) plot2d (sin(x), [x, -5, 5])$
(%i2) plot2d (sec(x), [x, -2, 2], [y, -20, 20], [nticks, 200])$
```

Anywhere there may be an ordinary expression, there may be a parametric expression: *parametric_expr* is a list of the form `[parametric, `*x_expr*`, `*y_expr*`, `*t_range*`, `*options*`]`. Here *x_expr* and *y_expr* are expressions of 1 variable *var* which is the first element of the range *trange*. The plot is of the path traced out by the pair `[`*x_expr*`, `*y_expr*`]` as *var* varies in *trange*.

In the following example, we plot a circle, then we do the plot with only a few points used, so that we get a star, and finally we plot this together with an ordinary function of X.

Examples:

- Plot a circle with a parametric plot.

```
(%i1) plot2d ([parametric, cos(t), sin(t), [t, -%pi*2, %pi*2],
        [nticks, 80]])$
```

- Plot a star: join eight points on the circumference of a circle.

```
(%i2) plot2d ([parametric, cos(t), sin(t), [t, -%pi*2, %pi*2],
        [nticks, 8]])$
```

- Plot a cubic polynomial with an ordinary plot and a circle with a parametric plot.

```
(%i3) plot2d ([x^3 + 2, [parametric, cos(t), sin(t), [t, -5, 5],
        [nticks, 80]]], [x, -3, 3])$
```

Discrete expressions may also be used instead or ordinary or parametric expressions: *discrete_expr* is a list of the form `[discrete, `*x_list*`, `*y_list*`]` or `[discrete, `*xy_list*`]`, where *xy_list* is a list of `[`*x*`,`*y*`]` pairs.

Examples:

- Create some lists.

```
(%i1) xx:makelist(x,x,0,10)$
(%i2) yy:makelist(exp(-x*1.0),x,0,10)$
(%i3) xy:makelist([x,x*x],x,0,5)$
```

- Plot with line segments.

```
(%i4) plot2d([discrete,xx,yy])$
```

- Plot with line segments, using a list of pairs.

```
(%i5) plot2d([discrete,xy])$
```

- Plot with points.

```
(%i6) plot2d([discrete,xx,yy],[gnuplot_curve_styles,["with points"]])$
```

- Plot the curve `cos(x)` using lines and (*xx*,*yy*) using points.

```
plot2d([cos(x),[discrete,xx,yy]],[x,0,10],[gnuplot_curve_styles,["with line
```

See also `plot_options`, which describes plotting options and has more examples.

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]);
```

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 0.2 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 System variable

Elements of this list state the default options for plotting. If an option is present in a `plot2d` or `plot3d` call, that value takes precedence over the default option. Otherwise, the value in `plot_options` is used. Default options are assigned by `set_plot_option`.

Each element of `plot_options` is a list of two or more items. The first item is the name of an option, and the remainder comprises the value or values assigned to the option. In some cases the, the assigned value is a list, which may comprise several items.

The plot options which are recognized by `plot2d` and `plot3d` are the following:

- Option: plot_format determines which plotting package is used by plot2d and plot3d.
  - Default value: gnuplot Gnuplot is the default, and most advanced, plotting package. It requires an external gnuplot installation.
  - Value: mgnuplot Mgnuplot is a Tk-based wrapper around gnuplot. It is included in the Maxima distribution. Mgnuplot offers a rudimentary GUI for gnuplot, but has fewer overall features than the plain gnuplot interface. Mgnuplot requires an external gnuplot installation and Tcl/Tk.
  - Value: openmath Openmath is a Tcl/Tk GUI plotting program. It is included in the Maxima distribution.
  - Value: ps Generates simple PostScript files directly from Maxima. Much more sophisticated PostScript output can be generated from gnuplot, by leaving the option plot_format unspecified (to accept the default), and setting the option gnuplot_term to ps.
- Option: run_viewer controls whether or not the appropriate viewer for the plot format should be run.
  - Default value: true Execute the viewer program.
  - Value: false Do not execute the viewer program.
- gnuplot_term Sets the output terminal type for gnuplot.
  - Default value: default Gnuplot output is displayed in a separate graphical window.
  - Value: dumb Gnuplot output is displayed in the Maxima console by an "ASCII art" approximation to graphics.
  - Value: ps Gnuplot generates commands in the PostScript page description language. If the option gnuplot_out_file is set to *filename*, gnuplot writes the PostScript commands to *filename*. Otherwise, the commands are printed to the Maxima console.
- Option: gnuplot_out_file Write gnuplot output to a file.
  - Default value: false No output file specified.
  - Value: *filename* Example: [gnuplot_out_file, "myplot.ps"] This example sends PostScript output to the file myplot.ps when used in conjunction with the PostScript gnuplot terminal.
- Option: x The default horizontal range.

  ```
  [x, - 3, 3]
  ```

  Sets the horizontal range to [-3, 3].
- Option: y The default vertical range.

  ```
  [y, - 3, 3]
  ```

  Sets the vertical range to [-3, 3].
- Option: t The default range for the parameter in parametric plots.

  ```
  [t, 0, 10]
  ```

  Sets the parametric variable range to [0, 10].
- Option: nticks Initial number of points used by the adaptive plotting routine.

```
[nticks, 20]
```

The default for `nticks` is 10.

- Option: `adapt_depth` The maximum number of splittings used by the adaptive plotting routine.

  ```
  [adapt_depth, 5]
  ```

  The default for `adapt_depth` is 10.

- Option: `grid` Sets the number of grid points to use in the x- and y-directions for three-dimensional plotting.

  ```
  [grid, 50, 50]
  ```

  sets the grid to 50 by 50 points. The default grid is 30 by 30.

- Option: `transform_xy` Allows transformations to be applied to three-dimensional plots.

  ```
  [transform_xy, false]
  ```

  The default `transform_xy` is `false`. If it is not `false`, it should be the output of

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

  The `polar_xy` transformation is built in. It gives the same transformation as

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

- Option: `colour_z` is specific to the `ps` plot format.

  ```
  [colour_z, true]
  ```

  The default value for `colour_z` is `false`.

- Option: `view_direction` Specific to the `ps` plot format.

  ```
  [view_direction, 1, 1, 1]
  ```

  The default `view_direction` is [1, 1, 1].

There are several plot options specific to gnuplot. All of these options (except `gnuplot_pm3d`) are raw gnuplot commands, specified as strings. Refer to the gnuplot documentation for more details.

- Option: `gnuplot_pm3d` Controls the usage PM3D mode, which has advanced 3D features. PM3D is only available in gnuplot versions after 3.7. The default value for `gnuplot_pm3d` is `false`.

  Example:

  ```
  [gnuplot_pm3d, true]
  ```

- Option: `gnuplot_preamble` Inserts gnuplot commands before the plot is drawn. Any valid gnuplot commands may be used. Multiple commands should be separated with a semi-colon. The example shown produces a log scale plot. The default value for `gnuplot_preamble` is the empty string `""`.

  Example:

  ```
  [gnuplot_preamble, "set log y"]
  ```

- Option: `gnuplot_curve_titles` Controls the titles given in the plot key. The default value is `default`, which automatically sets the title of each curve to the function plotted. If not `default`, `gnuplot_curve_titles` should contain a list of strings. (To disable the plot key entirely, add "set nokey" to gnuplot_preamble.)

Example:

```
[gnuplot_curve_titles, ["my first function", "my second function"]]
```

- Option: gnuplot_curve_styles A list of strings controlling the appearance of curves, i.e., color, width, dashing, etc., to be sent to the gnuplot plot command. The default value is ["with lines 3", "with lines 1", "with lines 2", "with lines 5", "with lines 4", "with lines 6", "with lines 7"], which cycles through different colors. See the gnuplot documentation for plot for more information.

  Example:

  ```
  [gnuplot_curve_styles, ["with lines 7", "with lines 2"]]
  ```

- Option: gnuplot_default_term_command The gnuplot command to set the terminal type for the default terminal. The default value is the empty string "", i.e., use gnuplot's default.

  Example:

  ```
  [gnuplot_default_term_command, "set term x11"]
  ```

- Option: gnuplot_dumb_term_command The gnuplot command to set the terminal type for the dumb terminal. The default value is "set term dumb 79 22", which makes the text output 79 characters by 22 characters.

  Example:

  ```
  [gnuplot_dumb_term_command, "set term dumb 132 50"]
  ```

- Option: gnuplot_ps_term_command The gnuplot command to set the terminal type for the PostScript terminal. The default value is "set size 1.5, 1.5;set term postscript eps enhanced color solid 24", which sets the size to 1.5 times gnuplot's default, and the font size to 24, among other things. See the gnuplot documentation for set term postscript for more information.

  Example:

  ```
  [gnuplot_ps_term_command, "set term postscript eps enhanced color solid 18"
  ```

Examples:

- Saves a plot of sin(x) to the file sin.eps.

  ```
  plot2d (sin(x), [x, 0, 2*%pi], [gnuplot_term, ps], [gnuplot_out_file, "sin.eps
  ```

- Uses the y option to chop off singularities and the gnuplot_preamble option to put the key at the bottom of the plot instead of the top.

  ```
  plot2d ([gamma(x), 1/gamma(x)], [x, -4.5, 5], [y, -10, 10], [gnuplot_preamble,
  ```

- Uses a very complicated gnuplot_preamble to produce fancy x-axis labels. (Note that the gnuplot_preamble string must be entered without any line breaks.)

  ```
  my_preamble: "set xzeroaxis; set xtics ('-2pi' -6.283, '-3pi/2' -4.712, '-pi'
  plot2d ([cos(x), sin(x), tan(x), cot(x)], [x, -2*%pi, 2*%pi],
      [y, -2, 2], [gnuplot_preamble, my_preamble]);
  ```

- Uses a very complicated gnuplot_preamble to produce fancy x-axis labels, and produces PostScript output that takes advantage of the advanced text formatting available in gnuplot. (Note that the `gnuplot_preamble` string must be entered without any line breaks.)

```
my_preamble: "set xzeroaxis; set xtics ('-2{/Symbol p}' -6.283, '-3{/Symbol p}
plot2d ([cos(x), sin(x), tan(x)], [x, -2*%pi, 2*%pi], [y, -2, 2],
    [gnuplot_preamble, my_preamble], [gnuplot_term, ps], [gnuplot_out_file, "t
```

- A three-dimensional plot using the gnuplot pm3d terminal.

```
plot3d (atan (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4], [grid, 50, 50], [gnuplot_
```

- A three-dimensional plot without a mesh and with contours projected on the bottom plane.

```
my_preamble: "set pm3d at s;unset surface;set contour;set cntrparam levels 20;
plot3d (atan (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4], [grid, 50, 50],
    [gnuplot_pm3d, true], [gnuplot_preamble, my_preamble])$
```

- A plot where the z-axis is represented by color only. (Note that the `gnuplot_preamble` string must be entered without any line breaks.)

```
plot3d (cos (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
    [gnuplot_preamble, "set view map; unset surface"], [gnuplot_pm3d, true], [
```

plot3d (*expr, x_range, y_range, ..., options, ...*) Function
plot3d (*[expr_1, expr_2, expr_3], x_range, y_range, ..., options, ...*) Function

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

plots `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 plots a Moebius band, parametrized by the three 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.

This example shows a plot of the 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` option 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 Klein bottle:

```
expr_1: 5*cos(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y) + 3.0) - 10.0;
expr_2: -5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y) + 3.0);
expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y));

plot3d ([expr_1, expr_2, expr_3], [x, -%pi, %pi], [y, -%pi, %pi], ['grid, 40, 
```

or a torus

```
expr_1: cos(y)*(10.0+6*cos(x));
expr_2: sin(y)*(10.0+6*cos(x));
expr_3: -6*sin(x);

plot3d ([expr_1, expr_2, expr_3], [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])$
```

See `plot_options` for more examples.

make_transform (*vars, fx, fy, fz*) Function

Returns a function suitable for the transform function in plot3d. Use with the plot option `transform_xy`.

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

is a transformation to polar coordinates.

plot2d_ps (*expr, range*) Function

Writes to pstream a sequence of PostScript commands which plot *expr* over *range*.

expr is an expression. *range* is a list of the form `[`*x*`,` *min*`,` *max*`]` in which *x* is a variable which appears in *expr*.

See also `closeps`.

closeps () Function

This should usually becalled 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

Assigns one of the global variables for plotting. *option* is specified as a list of two or more elements, in which the first element is one of the keywords on the `plot_options` list.

`set_plot_option` evaluates its argument. `set_plot_option` returns `plot_options` (after modifying one of its elements).

See also `plot_options`, `plot2d`, and `plot3d`.

Examples:

Modify the `grid` and `x` values. When a `plot_options` keyword has an assigned value, quote it to prevent evaluation.

```
(%i1) set_plot_option ([grid, 30, 40]);
(%o1) [[x, - 1.755559702014E+305, 1.755559702014E+305],
[y, - 1.755559702014E+305, 1.755559702014E+305], [t, - 3, 3],
[grid, 30, 40], [view_direction, 1, 1, 1], [colour_z, false],
[transform_xy, false], [run_viewer, true],
[plot_format, gnuplot], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]],
[gnuplot_curve_styles, [with lines 3, with lines 1,
with lines 2, with lines 5, with lines 4, with lines 6,
with lines 7]], [gnuplot_default_term_command, ],
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript #
eps enhanced color solid 24]]
(%i2) x: 42;
(%o2)                          42
(%i3) set_plot_option (['x, -100, 100]);
(%o3) [[x, - 100.0, 100.0], [y, - 1.755559702014E+305,
1.755559702014E+305], [t, - 3, 3], [grid, 30, 40],
[view_direction, 1, 1, 1], [colour_z, false],
[transform_xy, false], [run_viewer, true],
[plot_format, gnuplot], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]],
[gnuplot_curve_styles, [with lines 3, with lines 1,
with lines 2, with lines 5, with lines 4, with lines 6,
with lines 7]], [gnuplot_default_term_command, ],
[gnuplot_dumb_term_command, set term dumb 79 22],
[gnuplot_ps_term_command, set size 1.5, 1.5;set term postscript #
eps enhanced color solid 24]]
```

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"))
```

pscom (*cmd*) Function

cmd is inserted in the PostScript file. Example:

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

9 Input and Output

9.1 Introduction to Input and Output

9.2 Files

A file is simply an area on a particular storage device which contains data or text. Files on the disks are figuratively grouped into "directories". A directory is just a list of files. Commands which deal with files are: `save`, `load`, `loadfile`, `stringout`, `batch`, `demo`, `writefile`, `closefile`, and `appendfile`.

9.3 Definitions for Input and Output

_ System variable

`_` is the most recent input expression (e.g., `%i1`, `%i2`, `%i3`, ...).

`_` is assigned the input before the input is simplified or evaluated. However, the value of `_` is simplified (but not evaluated) when it is displayed.

`_` is recognized by `batch`, but not by `load`.

See also `%`.

Examples:

```
(%i1) 13 + 29;
(%o1)                        42
(%i2) :lisp $_
((MPLUS) 13 29)
(%i2) _;
(%o2)                        42
(%i3) sin (%pi/2);
(%o3)                         1
(%i4) :lisp $_
((%SIN) ((MQUOTIENT) $%PI 2))
(%i4) _;
(%o4)                         1
(%i5) a: 13$
(%i6) b: 29$
(%i7) a + b;
(%o7)                        42
(%i8) :lisp $_
((MPLUS) $A $B)
(%i8) _;
(%o8)                       b + a
(%i9) a + b;
(%o9)                        42
(%i10) ev (_);
(%o10)                       42
```

% — System variable

% is the output expression (e.g., %o1, %o2, %o3, ...) most recently computed by Maxima, whether or not it was displayed.

% is recognized by `batch`, but not by `load`.

See also `_`, `%%`, and `%th`.

%% — System variable

In a compound statement comprising two or more statements, `%%` is the value of the previous statement. For example,

```
block (integrate (x^5, x), ev (%%, x=2) - ev (%%, x=1));
block ([prev], prev: integrate (x^5, x), ev (prev, x=2) - ev (prev, x=1));
```

yield the same result, namely `21/2`.

A compound statement may comprise other compound statements. Whether a statement be simple or compound, `%%` is the value of the previous statement. For example,

```
block (block (a^n, %%*42), %%/6)
```

yields `7*a^n`.

Within a compound statement, the value of `%%` may be inspected at a break prompt, which is opened by executing the `break` function. For example, at the break prompt opened by

```
block (a: 42, break ())$
```

entering `%%;` yields `42`.

At the first statement in a compound statement, or outside of a compound statement, `%%` is undefined.

`%%` is recognized by both `batch` and `load`.

See also `%`.

%edispflag — Option variable

Default value: `false`

When `%edispflag` is `true`, Maxima displays `%e` to a negative exponent as a quotient. For example, `%e^-x` is displayed as `1/%e^x`.

%th (*i*) — Function

The value of the *i*'th previous output expression. That is, if the next expression to be computed is the *n*'th output, `%th (`*m*`)` is the (*n* - *m*)'th output.

`%th` is useful in `batch` files or for referring to a group of output expressions. For example,

```
block (s: 0, for i:1 thru 10 do s: s + %th (i))$
```

sets `s` to the sum of the last ten output expressions.

`%th` is recognized by `batch`, but not by `load`.

See also `%`.

"?" Special symbol

As prefix to a function or variable name, `?` signifies that the name is a Lisp name, not a Maxima name. For example, `?round` signifies the Lisp function `ROUND`. See Section 3.2 [Lisp and Maxima], page 9 for more on this point.

The notation `? word` (a question mark followed a word, separated by whitespace) is equivalent to `describe ("word")`.

absboxchar Option variable

Default value: `!`

`absboxchar` is the character used to draw absolute value signs around expressions which are more than one line tall.

appendfile (*filename*) Function

Appends a console transcript to *filename*. `appendfile` is the same as `writefile`, except that the transcript file, if it exists, is always appended.

`closefile` closes the transcript file opened by `appendfile` or `writefile`.

batch (*filename*) Function

Reads Maxima expressions from *filename* and evaluates them. `batch` searches for *filename* in the list `file_search_maxima`. See `file_search`.

filename comprises a sequence of Maxima expressions, each terminated with `;` or `$`. The special variable `%` and the function `%th` refer to previous results within the file. The file may include `:lisp` constructs. Spaces, tabs, and newlines in the file are ignored. A suitable input file may be created by a text editor or by the `stringout` function.

`batch` reads each input expression from *filename*, displays the input to the console, computes the corresponding output expression, and displays the output expression. Input labels are assigned to the input expressions and output labels are assigned to the output expressions. `batch` evaluates every input expression in the file unless there is an error. If user input is requested (by `asksign` or `askinteger`, for example) `batch` pauses to collect the requisite input and then continue.

It may be possible to halt `batch` by typing `control-C` at the console. The effect of `control-C` depends on the underlying Lisp implementation.

`batch` has several uses, such as to provide a reservoir for working command lines, to give error-free demonstrations, or to help organize one's thinking in solving complex problems.

`batch` evaluates its argument. `batch` has no return value.

See also `load`, `batchload`, and `demo`.

batchload (*filename*) Function

Reads Maxima expressions from *filename* and evaluates them, without displaying the input or output expressions and without assigning labels to output expressions. Printed output (such as produced by `print` or `describe`) is displayed, however.

The special variable `%` and the function `%th` refer to previous results from the interactive interpreter, not results within the file. The file cannot include `:lisp` constructs.

`batchload` returns the path of *filename*, as a string. `batchload` evaluates its argument.

See also `batch` and `load`.

closefile () Function

Closes the transcript file opened by `writefile` or `appendfile`.

collapse (*expr*) Function

Collapses *expr* 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 after loading in a `save` file. You can collapse several expressions together by using `collapse ([`*expr_1*`, ...,` *expr_n*`])`. Similarly, you can collapse the elements of the array `A` by doing `collapse (listarray ('A))`.

concat (*arg_1, arg_2, ...*) Function

Concatenates its arguments. The arguments must evaluate to atoms. The return value is a symbol if the first argument is a symbol and a Maxima string otherwise.

`concat` evaluates its arguments. The single quote `'` prevents evaluation.

```
(%i1) y: 7$
(%i2) z: 88$
(%i3) concat (y, z/2);
(%o3)                        744
(%i4) concat ('y, z/2);
(%o4)                        y44
```

A symbol constructed by `concat` may be assigned a value and appear in expressions. The `::` (double colon) assignment operator evaluates its left-hand side.

```
(%i5) a: concat ('y, z/2);
(%o5)                        y44
(%i6) a:: 123;
(%o6)                        123
(%i7) y44;
(%o7)                        123
(%i8) b^a;
                             y44
(%o8)                       b
(%i9) %, numer;
                             123
(%o9)                       b
```

Note that although `concat (1, 2)` looks like a number, it is a Maxima string.

```
(%i10) concat (1, 2) + 3;
(%o10)                     12 + 3
```

sconcat (*arg_1, arg_2, ...*) Function

Concatenates its arguments into a string. Unlike `concat`, the arguments do *not* need to be atoms.

The result is a Lisp string.

```
(%i1) sconcat ("xx[", 3, "]:", expand ((x+y)^3));
(%o1)                xx[3]:y^3+3*x*y^2+3*x^2*y+x^3
```

disp (*expr_1, expr_2, ...*) 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 (*tensor_1, tensor_2, ...*) Function

dispcon (*all*) Function

Displays the contraction properties of its arguments as were given to `defcon`. `dispcon (all)` displays all the contraction properties which were defined.

display (*expr_1, expr_2, ...*) Function

Displays equations whose left side is *expr_i* 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 also `disp`.

```
(%i1) display(B[1,2]);
                                          2
                         B     = X - X
                          1, 2
(%o1)                            done
```

display2d Option variable

Default value: `true`

When `display2d` is `false`, the console display is a string (1-dimensional) form rather than a display (2-dimensional) form.

display_format_internal Option variable

Default value: `false`

When `display_format_internal` is `true`, expressions are displayed without being transformed in ways that hide the internal mathematical representation. The display then corresponds to what `inpart` returns rather than `part`.

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 *expr* 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 + ...)` displays `P1`, then below it `P2`, etc. When not using `dispterms`, if an exponential expression is too wide to be displayed as `A^B` it appears as `expt (A, B)` (or as `ncexpt (A, B)` in the case of `A^^B`).

error_size Option variable

Default value: 10

`error_size` modifies error messages according to the size of expressions which appear in them. If the size of an expression (as determined by the Lisp function `ERROR-SIZE`) is greater than `error_size`, the expression is replaced in the message by a symbol, and the symbol is assigned the expression. The symbols are taken from the list `error_syms`.

Otherwise, the expression is smaller than `error_size`, and the expression is displayed in the message.

See also `error` and `error_syms`.

Example:

The size of `U`, as determined by `ERROR-SIZE`, is 24.

```
(%i1) U: (C^D^E + B + A)/(cos(X-1) + 1)$

(%i2) error_size: 20$

(%i3) error ("Example expression is", U);

Example expression is errexp1
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i4) errexp1;
                            E
                           D
                          C   + B + A
(%o4)                    --------------
                         cos(X - 1) + 1
(%i5) error_size: 30$

(%i6) error ("Example expression is", U);

                         E
                        D
                       C   + B + A
Example expression is --------------
                      cos(X - 1) + 1
 -- an error.  Quitting.  To debug this try debugmode(true);
```

error_syms Option variable

Default value: `[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`. The first too-large expression is replaced by `error_syms[1]`, the second by `error_syms[2]`, and so on.

If there are more too-large expressions than there are elements of `error_syms`, symbols are constructed automatically, with the *n*-th symbol equivalent to `concat ('errexp, n)`.

See also `error` and `error_size`.

expt (*a, b*) Function

If an exponential expression is too wide to be displayed as *a*^*b* it appears as `expt (a, b)` (or as `ncexpt (a, b)` in the case of *a*^^*b*).

`expt` and `ncexpt` are not recognized in input.

exptdispflag Option variable

Default value: `true`

When `exptdispflag` is `true`, Maxima displays expressions with negative exponents using quotients, e.g., `X^(-1)` as `1/X`.

filename_merge (*path, filename*) Function

Constructs a modified path from *path* and *filename*. If the final component of *path* is of the form `###.`*something*, the component is replaced with *filename*`.`*something*. Otherwise, the final component is simply replaced by *filename*.

file_search (*filename*) Function
file_search (*filename, pathlist*) Function

`file_search` searches for the file *filename* and returns the path to the file (as a string) if it can be found; otherwise `file_search` returns `false`. `file_search (filename)` searches in the default search directories, which are specified by the `file_search_maxima`, `file_search_lisp`, and `file_search_demo` variables.

`file_search` first checks if the actual name passed exists, before attempting to match it to "wildcard" file search patterns. See `file_search_maxima` concerning file search patterns.

The argument *filename* can be a path and file name, or just a file name, or, if a file search directory includes a file search pattern, just the base of the file name (without an extension). For example,

```
file_search ("/home/wfs/special/zeta.mac");
file_search ("zeta.mac");
file_search ("zeta");
```

all find the same file, assuming the file exists and `/home/wfs/special/###.mac` is in `file_search_maxima`.

`file_search (filename, pathlist)` searches only in the directories specified by *pathlist*, which is a list of strings. The argument *pathlist* supersedes the default search directories, so if the path list is given, `file_search` searches only the ones specified, and not any of the default search directories. Even if there is only one directory in *pathlist*, it must still be given as a one-element list.

The user may modify the default search directories. See `file_search_maxima`.

`file_search` is invoked by `load` with `file_search_maxima` and `file_search_lisp` as the search directories.

file_search_maxima Option variable
file_search_lisp Option variable
file_search_demo Option variable

These variables specify lists of directories to be searched by `load`, `demo`, and some other Maxima functions. The default values of these variables name various directories in the Maxima installation.

The user can modify these variables, either to replace the default values or to append additional directories. For example,

```
file_search_maxima: ["/usr/local/foo/###.mac",
    "/usr/local/bar/###.mac"]$
```

replaces the default value of `file_search_maxima`, while

```
file_search_maxima: append (file_search_maxima,
    ["/usr/local/foo/###.mac", "/usr/local/bar/###.mac"])$
```

appends two additional directories. It may be convenient to put such an expression in the file `maxima-init.mac` so that the file search path is assigned automatically when Maxima starts.

Multiple filename extensions and multiple paths can be specified by special "wildcard" constructions. The string `###` expands into the sought-after name, while a comma-separated list enclosed in curly braces `{foo,bar,baz}` expands into multiple strings. For example, supposing the sought-after name is `neumann`,

```
"/home/{wfs,gcj}/###.{lisp,mac}"
```

expands into `/home/wfs/neumann.lisp`, `/home/gcj/neumann.lisp`, `/home/wfs/neumann.mac`, and `/home/gcj/neumann.mac`.

file_type (*filename*) Function

Returns a guess about the content of *filename*, based on the filename extension. *filename* need not refer to an actual file; no attempt is made to open the file and inspect the content.

The return value is a symbol, either `object`, `lisp`, or `maxima`. If the extension starts with `m` or `d`, `file_type` returns `maxima`. If the extension starts with `l`, `file_type` returns `lisp`. If none of the above, `file_type` returns `object`.

grind (*expr*) Function
grind Option variable

The function `grind` prints *expr* to the console in a form suitable for input to Maxima. `grind` always returns `done`.

See also `string`, which returns a string instead of printing its output. `grind` attempts to print the expression in a manner which makes it slightly easier to read than the output of `string`.

When the variable `grind` is `true`, the output of `string` and `stringout` has the same format as that of `grind`; otherwise no attempt is made to specially format the output of those functions. The default value of the variable `grind` is `false`.

`grind` can also be specified as an argument of `playback`. When `grind` is present, `playback` prints input expressions in the same format as the `grind` function. Otherwise, no attempt is made to specially format input expressions.

ibase Option variable

Default value: 10

Integers entered into Maxima are interpreted with respect to the base `ibase`.

`ibase` may be assigned any integer between 2 and 35 (decimal), inclusive. When `ibase` is greater than 10, the numerals comprise the decimal numerals 0 through 9 plus capital letters of the alphabet A, B, C, ..., as needed. The numerals for base 35, the largest acceptable base, comprise 0 through 9 and A through Y.

See also `obase`.

inchar Option variable

Default value: `%i`

`inchar` is the prefix of the labels of expressions entered by the user. Maxima automatically constructs a label for each input expression by concatenating `inchar` and `linenum`. `inchar` may be assigned any string or symbol, not necessarily a single character.

```
(%i1) inchar: "input";
(%o1)                                input
(input1) expand ((a+b)^3);
                            3        2      2      3
(%o1)                      b  + 3 a b  + 3 a  b + a
(input2)
```

See also `labels`.

ldisp (*expr_1*, ..., *expr_n*) Function

Displays expressions *expr_1*, ..., *expr_n* to the console as printed output. `ldisp` assigns an intermediate expression label to each argument and returns the list of labels.

See also `disp`.

```
(%i1) e: (a+b)^3;
                                   3
(%o1)                       (b + a)
(%i2) f: expand (e);
                     3        2      2      3
(%o2)               b  + 3 a b  + 3 a  b + a
(%i3) ldisp (e, f);
                                   3
```

```
(%t3)                          (b + a)

                         3        2      2      3
(%t4)                   b  + 3 a b  + 3 a  b + a

(%o4)                         [%t3, %t4]
(%i4) %t3;
                                      3
(%o4)                          (b + a)
(%i5) %t4;
                         3        2      2      3
(%o5)                   b  + 3 a b  + 3 a  b + a
```

ldisplay (*expr_1*, ..., *expr_n*) Function

Displays expressions *expr_1*, ..., *expr_n* to the console as printed output. Each expression is printed as an equation of the form `lhs = rhs` in which `lhs` is one of the arguments of `ldisplay` and `rhs` is its value. Typically each argument is a variable. `ldisp` assigns an intermediate expression label to each equation and returns the list of labels.

See also `display`.

```
(%i1) e: (a+b)^3;
                                      3
(%o1)                          (b + a)
(%i2) f: expand (e);
                         3        2      2      3
(%o2)                   b  + 3 a b  + 3 a  b + a
(%i3) ldisplay (e, f);
                                        3
(%t3)                        e = (b + a)

                             3        2      2      3
(%t4)                   f = b  + 3 a b  + 3 a  b + a

(%o4)                         [%t3, %t4]
(%i4) %t3;
                                        3
(%o4)                        e = (b + a)
(%i5) %t4;
                             3        2      2      3
(%o5)                   f = b  + 3 a b  + 3 a  b + a
```

linechar Option variable

Default value: `%t`

`linechar` is the prefix of the labels of intermediate expressions generated by Maxima. Maxima constructs a label for each intermediate expression (if displayed) by concatenating `linechar` and `linenum`. `linechar` may be assigned any string or symbol, not necessarily a single character.

Intermediate expressions might or might not be displayed. See `programmode` and `labels`.

linel Option variable

Default value: 79

`linel` is the assumed width (in characters) of the console display for the purpose of displaying expressions. `linel` may be assigned any value by the user, although very small or very large values may be impractical. Text printed by built-in Maxima functions, such as error messages and the output of `describe`, is not affected by `linel`.

lispdisp Option variable

Default value: `false`

When `lispdisp` is `true`, Lisp symbols are displayed with a leading question mark `?`. Otherwise, Lisp symbols are displayed with no leading mark.

Examples:

```
(%i1) lispdisp: false$
(%i2) ?foo + ?bar;
(%o2)                      foo + bar
(%i3) lispdisp: true$
(%i4) ?foo + ?bar;
(%o4)                     ?foo + ?bar
```

load (*filename*) Function

Evaluates expressions in *filename*, thus bringing variables, functions, and other objects into Maxima. The binding of any existing object is clobbered by the binding recovered from *filename*. To find the file, `load` calls `file_search` with `file_search_maxima` and `file_search_lisp` as the search directories. If `load` succeeds, it returns the name of the file. Otherwise `load` prints an error message.

`load` works equally well for Lisp code and Maxima code. Files created by `save`, `translate_file`, and `compile_file`, which create Lisp code, and `stringout`, which creates Maxima code, can all be processed by `load`. `load` calls `loadfile` to load Lisp files and `batchload` to load Maxima files.

See also `loadfile`, `batch`, `batchload`, and `demo`. `loadfile` processes Lisp files; `batch`, `batchload`, and `demo` process Maxima files.

See `file_search` for more detail about the file search mechanism.

`load` evaluates its argument.

loadfile (*filename*) Function

Evaluates Lisp expressions in *filename*. `loadfile` does not invoke `file_search`, so `filename` must include the file extension and as much of the path as needed to find the file.

`loadfile` can process files created by `save`, `translate_file`, and `compile_file`. The user may find it more convenient to use `load` instead of `loadfile`.

`loadfile` quotes its argument, so `filename` must be a literal string, not a string variable. The double-single-quote operator defeats quotation.

loadprint Option variable

Default value: `true`

`loadprint` tells whether to print a message when a file is loaded.

- When `loadprint` is `true`, always print a message.
- When `loadprint` is `'loadfile`, print a message only if a file is loaded by the function `loadfile`.
- When `loadprint` is `'autoload`, print a message only if a file is automatically loaded. See `setup_autoload`.
- When `loadprint` is `false`, never print a message.

obase Option variable

Default value: 10

`obase` is the base for integers displayed by Maxima.

`obase` may be assigned any integer between 2 and 35 (decimal), inclusive. When `obase` is greater than 10, the numerals comprise the decimal numerals 0 through 9 plus capital letters of the alphabet A, B, C, ..., as needed. The numerals for base 35, the largest acceptable base, comprise 0 through 9, and A through Y.

See also `ibase`.

outchar Option variable

Default value: `%o`

`outchar` is the prefix of the labels of expressions computed by Maxima. Maxima automatically constructs a label for each computed expression by concatenating `outchar` and `linenum`. `outchar` may be assigned any string or symbol, not necessarily a single character.

```
(%i1) outchar: "output";
(output1)                          output
(%i2) expand ((a+b)^3);
                          3        2      2      3
(output2)                b  + 3 a b  + 3 a  b + a
(%i3)
```

See also `labels`.

packagefile Option variable

Default value: `false`

Package designers who use `save` or `translate` to create packages (files) for others to use may want to set `packagefile: true` to prevent information from being added to Maxima'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 Maxima init files.

pfeformat Option variable

Default value: `false`

When `pfeformat` is `true`, a ratio of integers is displayed with the solidus (forward slash) character, and an integer denominator `n` is displayed as a leading multiplicative term `1/n`.

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

print (*expr_1, ..., expr_n*) Function

Evaluates and displays *expr_1*, ..., *expr_n* one after another, from left to right, starting at the left edge of the console display.

The value returned by `print` is the value of its last argument. `print` does not generate intermediate expression labels.

See also `display`, `disp`, `ldisplay`, and `ldisp`. Those functions display one expression per line, while `print` attempts to display two or more expressions per line.

To display the contents of a file, see `printfile`.

```
(%i1) r: print ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is", radcan (log
             3        2      2      3
(a+b)^3 is b  + 3 a b  + 3 a  b + a  log (a^10/b) is

                                                   10 log(a) - log(b)
(%i2) r;
(%o2)                      10 log(a) - log(b)
(%i3) disp ("(a+b)^3 is", expand ((a+b)^3), "log (a^10/b) is", radcan (log (a^
                            (a+b)^3 is

                         3        2      2      3
                        b  + 3 a b  + 3 a  b + a

                            log (a^10/b) is

                          10 log(a) - log(b)
```

tcl_output (*list, i0, skip*) Function
tcl_output (*list, i0*) Function
tcl_output ([*list_1, ..., list_n*], *i*) Function

Prints elements of a list enclosed by curly braces `{ }`, suitable as part of a program in the Tcl/Tk language.

`tcl_output (`*list*`, `*i0*`, `*skip*`)` prints *list*, beginning with element *i0* and printing elements *i0* `+` *skip*, *i0* `+ 2` *skip*, etc.

`tcl_output (`*list*`, `*i0*`)` is equivalent to `tcl_output (`*list*`, `*i0*`, 2)`.

`tcl_output ([`*list_1*`, ..., `*list_n*`], `*i*`)` prints the *i*'th elements of *list_1*, ..., *list_n*.

Examples:

```
(%i1) tcl_output ([1, 2, 3, 4, 5, 6], 1, 3)$

 {1.000000000     4.000000000
 }
(%i2) tcl_output ([1, 2, 3, 4, 5, 6], 2, 3)$

 {2.000000000     5.000000000
 }
(%i3) tcl_output ([3/7, 5/9, 11/13, 13/17], 1)$

 {((RAT SIMP) 3 7) ((RAT SIMP) 11 13)
 }
(%i4) tcl_output ([x1, y1, x2, y2, x3, y3], 2)$

 {$Y1 $Y2 $Y3
 }
(%i5) tcl_output ([[1, 2, 3], [11, 22, 33]], 1)$

 {SIMP 1.000000000     11.00000000
 }
```

read (*expr_1, ..., expr_n*) Function

Prints *expr_1*, ..., *expr_n*, then reads one expression from the console and returns the evaluated expression. The expression is terminated with a semicolon `;` or dollar sign `$`.

See also `readonly`.

```
(%i1) foo: 42$
(%i2) foo: read ("foo is", foo, " -- enter new value.")$
foo is 42  -- enter new value.
(a+b)^3;
(%i3) foo;
                                                 3
(%o3)                                     (b + a)
```

readonly (*expr_1*, ..., *expr_n*) Function

Prints *expr_1*, ..., *expr_n*, then reads one expression from the console and returns the expression (without evaluation). The expression is terminated with a `;` (semicolon) or `$` (dollar sign).

```
(%i1) aa: 7$
(%i2) foo: readonly ("Enter an expression:");
Enter an expression:
2^aa;
                                  aa
(%o2)                            2
(%i3) foo: read ("Enter an expression:");
Enter an expression:
2^aa;
(%o3)                            128
```

See also `read`.

reveal (*expr*, *depth*) Function

Replaces parts of *expr* at the specified integer *depth* with descriptive summaries.

- Sums and differences are replaced by `sum(`*n*`)` where *n* is the number of operands of the sum.
- Products are replaced by `product(`*n*`)` where *n* is the number of operands of the product.
- Exponentials are replaced by `expt`.
- Quotients are replaced by `quotient`.
- Unary negation is replaced by `negterm`.

When *depth* is greater than or equal to the maximum depth of *expr*, `reveal (`*expr*`,` *depth*`)` returns *expr* unmodified.

`reveal` evaluates its arguments. `reveal` returns the summarized expression.

Example:

```
(%i1) e: expand ((a - b)^2)/expand ((exp(a) + exp(b))^2);
                          2            2
                         b  - 2 a b + a
(%o1)               -------------------------
                        b + a     2 b     2 a
                    2 %e      + %e    + %e
(%i2) reveal (e, 1);
(%o2)                       quotient
(%i3) reveal (e, 2);
                             sum(3)
(%o3)                        ------
                             sum(3)
(%i4) reveal (e, 3);
                     expt + negterm + expt
(%o4)               ------------------------
                    product(2) + expt + expt
```

```
(%i5) reveal (e, 4);
                      2                 2
                     b  - product(3) + a
(%o5)        ------------------------------------
                        product(2)     product(2)
             2 expt + %e           + %e
(%i6) reveal (e, 5);
                        2           2
                       b  - 2 a b + a
(%o6)             --------------------------
                      sum(2)     2 b     2 a
                  2 %e       + %e    + %e
(%i7) reveal (e, 6);
                         2           2
                        b  - 2 a b + a
(%o7)              -------------------------
                       b + a     2 b     2 a
                   2 %e      + %e    + %e
```

rmxchar Option variable

Default value: `]`

`rmxchar` is the character drawn on the right-hand side of a matrix.

See also `lmxchar`.

save (*filename*, *name_1*, *name_2*, *name_3*, ...) Function
save (*filename*, *values*, *functions*, *labels*, ...) Function
save (*filename*, [*m*, *n*]) Function
save (*filename*, *name_1*=*expr_1*, ...) Function
save (*filename*, *all*) Function

Stores the current values of *name_1*, *name_2*, *name_3*, ..., in *filename*. The arguments are the names of variables, functions, or other objects. If a name has no value or function associated with it, it is ignored. `save` returns *filename*.

`save` stores data in the form of Lisp expressions. The data stored by `save` may be recovered by `load (`*filename*`)`. The effect of executing `save` when *filename* already exists depends on the underlying Lisp implementation; the file may be clobbered, or `save` may complain with an error message.

The special form `save (`*filename*`, values, functions, labels, ...)` stores the items named by `values`, `functions`, `labels`, etc. The names may be any specified by the variable `infolists`. `values` comprises all user-defined variables.

The special form `save (`*filename*`, [`*m*`, `*n*`])` stores the values of input and output labels *m* through *n*. Note that *m* and *n* must be literal integers or double-quoted symbols. Input and output labels may also be stored one by one, e.g., `save ("foo.1", %i42, %o42)`. `save (`*filename*`, labels)` stores all input and output labels. When the stored labels are recovered, they clobber existing labels.

The special form `save (`*filename*`, `*name_1*`=`*expr_1*`, `*name_2*`=`*expr_2*`, ...)` stores the values of *expr_1*, *expr_2*, ..., with names *name_1*, *name_2*, ... It is useful to apply this form to input and output labels, e.g., `save ("foo.1", aa=%o88)`. The right-hand side of the equality in this form may be any expression, which is evaluated. This form does not introduce the new names into the current Maxima environment, but only stores them in *filename*.

These special forms and the general form of `save` may be mixed at will. For example, `save (`*filename*`, aa, bb, cc=42, functions, [11, 17])`.

The special form `save (`*filename*`, all)` stores the current state of Maxima. This includes all user-defined variables, functions, arrays, etc., as well as some automatically defined items. The saved items include system variables, such as `file_search_maxima` or `showtime`, if they have been assigned new values by the user; see `myoptions`.

`save` quotes its arguments. *filename* must be a string, not a string variable. The first and last labels to save, if specified, must be integers. The double quote operator evaluates a string variable to its string value, e.g., `s: "foo.1"$ save (''s, all)$`, and integer variables to their integer values, e.g., `m: 5$ n: 12$ save ("foo.1", [''m, ''n])$`.

savedef Option variable

Default value: `true`

When `savedef` is `true`, the Maxima version of a user function is preserved when the function is translated. This permits the definition to be displayed by `dispfun` and allows the function to be edited.

When `savedef` is `false`, the names of translated functions are removed from the `functions` list.

show (*expr*) Function

Displays `expr` with the indexed objects in it shown having covariant indices as subscripts, contravariant indices as superscripts. The derivative indices are displayed as subscripts, separated from the covariant indices by a comma.

showratvars (*expr*) Function

Returns a list of the canonical rational expression (CRE) variables in expression `expr`.

See also `ratvars`.

stardisp Option variable

Default value: `false`

When `stardisp` is `true`, multiplication is displayed with an asterisk `*` between operands.

string (*expr*) Function

Converts `expr` to Maxima's linear notation just as if it had been typed in.

The return value of `string` is a string, and thus it cannot be used in a computation.

stringdisp Lisp variable

Default value: `false`

When `?stringdisp` is `true`, strings are displayed enclosed in double quote marks. Otherwise, quote marks are not displayed.

`?stringdisp` is always `true` when displaying a function definition.

`?stringdisp` is a Lisp variable, so it must be written with a leading question mark `?`.

Examples:

```
(%i1) ?stringdisp: false$
(%i2) "This is an example string.";
(%o2)              This is an example string.
(%i3) foo () := print ("This is a string in a function definition.");
(%o3) foo() :=
              print("This is a string in a function definition.")
(%i4) ?stringdisp: true$
(%i5) "This is an example string.";
(%o5)             "This is an example string."
```

stringout (*filename, expr_1, expr_2, expr_3, ...*) Function
stringout (*filename,* [*m, n*]) Function
stringout (*filename, input*) Function
stringout (*filename, functions*) Function
stringout (*filename, values*) Function

`stringout` writes expressions to a file in the same form the expressions would be typed for input. The file can then be used as input for the `batch` or `demo` commands, and it may be edited for any purpose. `stringout` can be executed while `writefile` is in progress.

The general form of `stringout` writes the values of one or more expressions to the output file. Note that if an expression is a variable, only the value of the variable is written and not the name of the variable. As a useful special case, the expressions may be input labels (`%i1`, `%i2`, `%i3`, ...) or output labels (`%o1`, `%o2`, `%o3`, ...).

If `grind` is `true`, `stringout` formats the output using the `grind` format. Otherwise the `string` format is used. See `grind` and `string`.

The special form `stringout (`*filename*`, [`*m*`, `*n*`])` writes the values of input labels m through n, inclusive.

The special form `stringout (`*filename*`, input)` writes all input labels to the file.

The special form `stringout (`*filename*`, functions)` writes all user-defined functions (named by the global list `functions`) to the file.

The special form `stringout (`*filename*`, values)` writes all user-assigned variables (named by the global list `values`) to the file. Each variable is printed as an assignment statement, with the name of the variable, a colon, and its value. Note that the general form of `stringout` does not print variables as assignment statements.

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

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

```
(%i1) integrate (1/(1+x^3), x);
                                          2 x - 1
                     2               atan(-------)
                log(x  - x + 1)           sqrt(3)    log(x + 1)
(%o1)         - --------------- + ------------- + ----------
                       6             sqrt(3)          3
(%i2) tex (%o1);
$$-{{\log \left(x^2-x+1\right)}\over{6}}+{{\arctan \left({{2\,x-1
 }\over{\sqrt{3}}}\right)}\over{\sqrt{3}}}+{{\log \left(x+1\right)
 }\over{3}}\leqno{\tt (\%o1)}$$
(%o2)                          (\%o1)
(%i3) tex (integrate (sin(x), x));
$$-\cos x$$
(%o3)                           false
(%i4) tex (%o1, "foo.tex");
(%o4)                          (\%o1)
```

system (*command*) Function

Executes *command* as a separate process. The command is passed to the default shell for execution. `system` is not supported by all operating systems, but generally exists in Unix and Unix-like environments.

Supposing `_hist.out` is a list of frequencies which you wish to plot as a bar graph using `xgraph`.

```
(%i1) (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 Option variable

Default value: `false`

When `ttyoff` is `true`, output expressions are not displayed. Output expressions are still computed and assigned labels. See `labels`.

Text printed by built-in Maxima functions, such as error messages and the output of `describe`, is not affected by `ttyoff`.

with_stdout (*filename, expr_1, expr_2, expr_3, ...*) Macro

Opens *filename* and then evaluates *expr_1*, *expr_2*, *expr_3*, ... The values of the arguments are not stored in *filename*, but any printed output generated by evaluating the arguments (from `print`, `display`, `disp`, or `grind`, for example) goes to *filename* instead of the console.

`with_stdout` returns the value of its final argument.

See also `writefile`.

```
(%i1) with_stdout ("tmp.out", for i:5 thru 10 do print (i, "! yields", i!))$
(%i2) printfile ("tmp.out")$
5 ! yields 120
6 ! yields 720
7 ! yields 5040
8 ! yields 40320
9 ! yields 362880
10 ! yields 3628800
```

writefile (*filename*) Function

Begins writing a transcript of the Maxima session to *filename*. All interaction between the user and Maxima is then recorded in this file, just as it appears on the console.

As the transcript is printed in the console output format, it cannot be reloaded into Maxima. To make a file containing expressions which can be reloaded, see `save` and `stringout`. `save` stores expressions in Lisp form, while `stringout` stores expressions in Maxima form.

The effect of executing `writefile` when *filename* already exists depends on the underlying Lisp implementation; the transcript file may be clobbered, or the file may be appended. `appendfile` always appends to the transcript file.

It may be convenient to execute `playback` after `writefile` to save the display of previous interactions. As `playback` displays only the input and output variables (`%i1`, `%o1`, etc.), any output generated by a print statement in a function (as opposed to a return value) is not displayed by `playback`.

`closefile` closes the transcript file opened by `writefile` or `appendfile`.

10 Floating Point

10.1 Definitions for Floating Point

bffac (*expr*, *n*) Function

Bigfloat version of the factorial (shifted gamma) function. The second argument is how many digits to retain and return, it's a good idea to request a couple of extra.

`load ("bffac")` loads this function.

algepsilon Option variable

Default value: 10^8

`algepsilon` is used by `algsys`.

bfloat (*expr*) Function

Converts all numbers and functions of numbers in *expr* to bigfloat numbers. The number of significant digits in the resulting bigfloats is specified by the global variable `fpprec`.

When `float2bf` 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 (*expr*) Function

Returns `true` if *expr* is a bigfloat number, otherwise `false`.

bfpsi (*n*, *z*, *fpprec*) Function
bfpsi0 (*z*, *fpprec*) Function

`bfpsi` is the polygamma function of real argument *z* and integer order *n*. `bfpsi0` is the digamma function. `bfpsi0 (`*z*`,` *fpprec*`)` is equivalent to `bfpsi (0,` *z*`,` *fpprec*`)`.

These functions return bigfloat values. *fpprec* is the bigfloat precision of the return value.

`load ("bffac")` loads these functions.

bftorat Option variable

Default value: `false`

`bftorat` controls the conversion of bfloats to rational numbers. When `bftorat` is `false`, `ratepsilon` will be used to control the conversion (this results in relatively small rational numbers). When `bftorat` is `true`, the rational number generated will accurately represent the bfloat.

bftrunc Option variable

Default value: `true`

`bftrunc` causes trailing zeroes in non-zero bigfloat numbers not to be displayed. Thus, if `bftrunc` is `false`, `bfloat (1)` displays as `1.000000000000000B0`. Otherwise, this is displayed as `1.0B0`.

cbffac (*z, fpprec*) Function

Complex bigfloat factorial.

`load ("bffac")` loads this function.

float (*expr*) Function

Converts integers, rational numbers and bigfloats in *expr* 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 Option variable

Default value: `false`

When `float2bf` 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).

floatnump (*expr*) Function

Returns `true` if *expr* is a floating point number, otherwise `false`.

fpprec Option variable

Default value: 16

`fpprec` is the number of significant digits for arithmetic on bigfloat numbers. `fpprec` does not affect computations on ordinary floating point numbers.

See also `bfloat` and `fpprintprec`.

fpprintprec Option variable

Default value: 0

`fpprintprec` is 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.

When `fpprintprec` is 0, or greater than or equal to `fpprec`, then the value of `fpprec` controls the number of digits used for printing.

When `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*) Lisp function

?round (*x, divisor*) Lisp function

Round the floating point *x* to the nearest integer. The argument must be an ordinary float, not a bigfloat. The `?` beginning the name indicates this is a Lisp function.

```
(%i1) ?round (-2.8);
(%o1)                          - 3
```

?truncate (*x*) Lisp function

?truncate (*x, divisor*) Lisp function

Truncate the floating point *x* towards 0, to become an integer. The argument must be an ordinary float, not a bigfloat. The `?` beginning the name indicates this is a Lisp function.

```
(%i1) ?truncate (-2.8);
(%o1)                           - 2
(%i2) ?truncate (2.4);
(%o2)                            2
(%i3) ?truncate (2.8);
(%o3)                            2
```

11 Contexts

11.1 Definitions for Contexts

activate (*context_1, ..., context_n*) Function

Activates the contexts *context_1*, ..., *context_n*. The facts in these contexts are then available to make deductions and retrieve information. The facts in these contexts are not listed by `facts ()`.

The variable `activecontexts` is the list of contexts which are active by way of the `activate` function.

activecontexts System variable

Default value: `[]`

`activecontexts` 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 (*pred_1, ..., pred_n*) Function

Adds predicates *pred_1*, ..., *pred_n* to the current database, after checking for redundancy and inconsistency. 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 elements are the predicates added to the database and the atoms `redundant` or `inconsistent` where applicable.

assumescalar Option variable

Default value: `true`

`assumescalar` helps govern whether expressions `expr` for which `nonscalarp (expr)` is `false` are assumed to behave like scalars for certain transformations.

Let `expr` represent any expression other than a list or a matrix, and let `[1, 2, 3]` represent any list or matrix. Then `expr . [1, 2, 3]` yields `[expr, 2 expr, 3 expr]` if `assumescalar` is `true`, or `scalarp (expr)` is `true`, or `constantp (expr)` is `true`.

If `assumescalar` is `true`, such expressions will behave like scalars only for commutative operators, but not for noncommutative multiplication `.`.

When `assumescalar` is `false`, such expressions will behave like non-scalars.

When `assumescalar` is `all`, such expressions will behave like scalars for all the operators listed above.

assume_pos Option variable

Default value: `false`

When `assume_pos` is `true` and the sign of a parameter *x* cannot be determined from the `assume` database or other considerations, `sign` and `asksign (x)` return `true`. This may forestall some automatically-generated `asksign` queries, such as may arise from `integrate` or other computations.

By default, a parameter is *x* such that `symbolp (x)` or `subvarp (x)`. The class of expressions considered parameters can be modified to some extent via the variable `assume_pos_pred`.

`sign` and `asksign` attempt to deduce the sign of expressions from the sign of operands within the expression. For example, if `a` and `b` are both positive, then `a + b` is also positive.

However, there is no way to bypass all `asksign` queries. In particular, when the `asksign` argument is a difference *x* `-` *y* or a logarithm `log(`*x*`)`, `asksign` always requests an input from the user, even when `assume_pos` is `true` and `assume_pos_pred` is a function which returns `true` for all arguments.

assume_pos_pred Option variable

Default value: `false`

When `assume_pos_pred` is assigned the name of a function or a lambda expression of one argument *x*, that function is called to determine whether *x* is considered a parameter for the purpose of `assume_pos`. `assume_pos_pred` is ignored when `assume_pos` is `false`.

The `assume_pos_pred` function is called by `sign` and `asksign` with an argument *x* which is either an atom, a subscripted variable, or a function call expression. If the `assume_pos_pred` function returns `true`, *x* is considered a parameter for the purpose of `assume_pos`.

By default, a parameter is *x* such that `symbolp (x)` or `subvarp (x)`.

See also `assume` and `assume_pos`.

Examples:

```
(%i1) assume_pos: true$
(%i2) assume_pos_pred: symbolp$
(%i3) sign (a);
(%o3)                         pos
(%i4) sign (a[1]);
(%o4)                         pnz
(%i5) assume_pos_pred: lambda ([x], display (x), true)$
(%i6) asksign (a);
                             x = a

(%o6)                         pos
(%i7) asksign (a[1]);
                            x = a
                                 1

(%o7)                         pos
(%i8) asksign (foo (a));
                           x = foo(a)

(%o8)                         pos
(%i9) asksign (foo (a) + bar (b));
                           x = foo(a)
```

```
                              x = bar(b)

(%o9)                           pos
(%i10) asksign (log (a));
                               x = a

Is  a - 1  positive, negative, or zero?

p;
(%o10)                          pos
(%i11) asksign (a - b);
                               x = a

                               x = b

                               x = a

                               x = b

Is  b - a  positive, negative, or zero?

p;
(%o11)                          neg
```

context Option variable

Default value: `initial`

`context` names the collection of facts maintained by `assume` and `forget`. `assume` adds facts to the collection named by `context`, while `forget` removes facts.

Binding `context` to a name *foo* changes the current context to *foo*. If the specified context *foo* does not yet exist, it is created automatically by a call to `newcontext`. The specified context is activated automatically.

See `context` for a general description of the context mechanism.

contexts Option variable

Default value: `[initial, global]`

`contexts` 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 database, called a context. Once this is done, the user can have Maxima 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 destroyed one by one by calling `forget` or destroyed as a whole by calling `kill` to destroy the context to which they belong.

Contexts exist in a hierarchy, with the root always being the context `global`, which contains information about Maxima 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 a subcontext of the active context.

When a fresh Maxima is started up, the user is in a context called `initial`, which has `global` as a subcontext.

See also `facts`, `newcontext`, `supcontext`, `killcontext`, `activate`, `deactivate`, `assume`, and `forget`.

deactivate (*context_1*, ..., *context_n*) Function

Deactivates the specified contexts *context_1*, ..., *context_n*.

facts (*item*) Function
facts () Function

If *item* is the name of a context, `facts (`*item*`)` returns a list of the facts in the specified context.

If *item* is not the name of a context, `facts (`*item*`)` 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.

`facts ()` (i.e., without an argument) lists the current context.

features Declaration

Maxima recognizes certain mathematical properties of functions and variables. These are called "features".

`declare (`*x*`, `*foo*`)` gives the property *foo* to the function or variable `x`.

`declare (`*foo*`, feature)` declares a new feature *foo*. For example, `declare ([red, green, blue], feature)` declares three new features, `red`, `green`, and `blue`.

The predicate `featurep (`*x*`, `*foo*`)` returns `true` if *x* has the *foo* property, and `false` otherwise.

The infolist `features` is a list of known features. These are `integer`, `noninteger`, `even`, `odd`, `rational`, `irrational`, `real`, `imaginary`, `complex`, `analytic`, `increasing`, `decreasing`, `oddfun`, `evenfun`, `posfun`, `commutative`, `lassociative`, `rassociative`, `symmetric`, and `antisymmetric`, plus any user-defined features.

`features` is a list of mathematical features. There is also a list of non-mathematical, system-dependent features. See `status`.

forget (*pred_1*, ..., *pred_n*) Function
forget (*L*) Function

Removes predicates established by `assume`. The predicates may be expressions equivalent to (but not necessarily identical to) those previously assumed.

`forget (`*L*`)`, where *L* is a list of predicates, forgets each item on the list.

killcontext (*context_1*, ..., *context_n*) Function

Kills the contexts *context_1*, ..., *context_n*.

If one of the contexts 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 unkilled context is `global` then `initial` is used instead. If the `initial` context is killed, a new, empty `initial` context is created.

`killcontext` refuses 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`.

`killcontext` evaluates its arguments. `killcontext` returns `done`.

newcontext (*name*) Function

Creates a new, empty context, called *name*, which has `global` as its only subcontext. The newly-created context becomes the currently active context.

`newcontext` evaluates its argument. `newcontext` returns *name*.

supcontext (*name, context*) Function
supcontext (*name*) Function

Creates a new context, called *name*, which has *context* as a subcontext. *context* must exist.

If *context* is not specified, the current context is assumed.

12 Polynomials

12.1 Introduction to Polynomials

Polynomials are stored in Maxima either in General Form or as Cannonical 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 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 3*X^2-1 is (X 2 3 0 -1) and that of 2*X*Y+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 Maxima 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.

12.2 Definitions for Polynomials

algebraic Option variable

Default value: `false`

`algebraic` must be set to `true` in order for the simplification of algebraic integers to take effect.

berlefact Option variable

Default value: `true`

When `berlefact` is `false` then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used.

bezout (*p1*, *p2*, *x*) Function

an alternative to the `resultant` command. It returns a matrix. `determinant` of this matrix is the desired resultant.

bothcoef (*expr*, *x*) Function

Returns a list whose first member is the coefficient of *x* in *expr* (as found by `ratcoef` if *expr* is in CRE form otherwise by `coeff`) and whose second member is the remaining part of *expr*. That is, `[A, B]` where *expr* `= A*`*x* `+ B`.

Example:

```
(%i1) islinear (expr, x) := block ([c],
        c: bothcoef (rat (expr, x), x),
        is (freeof (x, c) and c[1] # 0))$
(%i2) islinear ((r^2 - (x - r)^2)/x, x);
(%o2)                          true
```

coeff (*expr*, *x*, *n*) Function

Returns the coefficient of *x*`^`*n* in *expr*. *n* may be omitted if it is 1. *x* may be an atom, or complete subexpression of *expr* e.g., `sin(x)`, `a[i+1]`, `x + y`, etc. (In the last case the expression `(x + y)` should occur in *expr*). Sometimes it may be necessary to expand or factor *expr* in order to make *x*`^`*n* explicit. This is not done automatically by `coeff`.

Examples:

```
(%i1) coeff (2*a*tan(x) + tan(x) + b = 5*tan(x) + 3, tan(x));
(%o1)                      2 a + 1 = 5
(%i2) coeff (y + x*%e^x + 1, x, 0);
(%o2)                         y + 1
```

combine (*expr*) Function

Simplifies the sum *expr* by combining terms with the same denominator into a single term.

content (*p_1*, *x_1*, ..., *x_n*) Function

Returns a list whose first element is the greatest common divisor of the coefficients of the terms of the polynomial *p_1* in the variable *x_n* (this is the content) and whose second element is the polynomial *p_1* divided by the content.

Examples:

```
(%i1) content (2*x*y + 4*x^2*y^2, y);
                                    2
(%o1)                   [2 x, 2 x y  + y]
```

denom (*expr*) Function

Returns the denominator of the rational expression *expr*.

divide (*p_1, p_2, x_1, ..., x_n*) Function

computes the quotient and remainder of the polynomial *p_1* divided by the polynomial *p_2*, in a main polynomial variable, *x_n*. 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.

Examples:

```
(%i1) divide (x + y, x - y, x);
(%o1)                        [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)                      [- 1, 2 x]
```

Note that `y` is the main variable in the second example.

eliminate ([*eqn_1, ..., eqn_n*], [*x_1, ..., x_k*]) 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 *x_1, ..., x_k* eliminated. First *x_1* is eliminated yielding $n - 1$ expressions, then `x_2` is eliminated, etc. If $k = n$ then a single expression in a list is returned free of the variables *x_1, ..., x_k*. In this case `solve` is called to solve the last resultant for the last variable.

Example:

```
(%i1) expr1: 2*x^2 + y*x + z;
                                      2
(%o1)                       z + x y + 2 x
(%i2) expr2: 3*x + 5*y - z - 1;
(%o2)                     - z + 5 y + 3 x - 1
(%i3) expr3: z^2 + x - y^2 + 5;
                             2    2
(%o3)                       z  - y  + x + 5
(%i4) eliminate ([expr3, expr2, expr1], [y, z]);
             8         7         6          5          4
(%o4) [7425 x  - 1170 x  + 1299 x  + 12076 x  + 22887 x

                                    3         2
                            - 5154 x  - 1291 x  + 7688 x + 15376]
```

ezgcd (*p_1, p_2, p_3, ...*) Function

Returns a list whose first element is the g.c.d of the polynomials *p_1*, *p_2*, *p_3*, ... and whose remaining elements are the polynomials divided by the g.c.d. This always uses the `ezgcd` algorithm.

facexpand Option variable

Default value: `true`

`facexpand` controls whether the irreducible factors returned by `factor` are in expanded (the default) or recursive (normal CRE) form.

factcomb (*expr*) Function

Tries to combine the coefficients of factorials in *expr* with the factorials themselves by converting, for example, `(n + 1)*n!` into `(n + 1)!`.

`sumsplitfact` if set to `false` will cause `minfactorial` to be applied after a `factcomb`.

factor (*expr*) Function

Factors the expression *expr*, containing any number of variables or functions, into factors irreducible over the integers. `factor (`*expr*`, p)` factors *expr* over the field of integers with an element adjoined whose minimum polynomial is p.

`factorflag` if `false` suppresses the factoring of integer factors of rational expressions.

`dontfactor` may be set to a list of variables with respect to which factoring is not to occur. (It is initially empty). Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the `dontfactor` list.

`savefactors` 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` if `false` then the Kronecker factoring algorithm will be used otherwise the Berlekamp algorithm, which is the default, will be used.

`intfaclim` 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 Maxima from taking an inordinately long time factoring large integers.

Examples:

```
(%i1) factor (2^63 - 1);
                    2
(%o1)              7  73 127 337 92737 649657
(%i2) factor (-8*y - 4*x + z^2*(2*y + x));
(%o2)               (2 y + x) (z - 2) (z + 2)
(%i3) -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2;
                 2  2        2    2    2
(%o3)           x  y  + 2 x y  + y  - x  - 2 x - 1
(%i4) block ([dontfactor: [x]], factor (%/36/(1 + 2*y + y^2)));
                       2
                     (x  + 2 x + 1) (y - 1)
(%o4)                ----------------------
                           36 (y + 1)
(%i5) factor (1 + %e^(3*x));
                      x         2 x     x
(%o5)              (%e  + 1) (%e    - %e  + 1)
(%i6) factor (1 + x^4, a^2 - 2);
                    2              2
(%o6)             (x  - a x + 1) (x  + a x + 1)
(%i7) factor (-y^2*z^2 - x*z^2 + x^2*y^2 + x^3);
```

```
                                 2
(%o7)                     - (y  + x) (z - x) (z + x)
(%i8) (2 + x)/(3 + x)/(b + x)/(c + x)^2;
                                    x + 2
(%o8)                     -----------------------
                                                 2
                          (x + 3) (x + b) (x + c)
(%i9) ratsimp (%);
               4                  3
(%o9) (x + 2)/(x  + (2 c + b + 3) x

     2                     2            2                   2
 + (c  + (2 b + 6) c + 3 b) x  + ((b + 3) c  + 6 b c) x + 3 b c )
(%i10) partfrac (%, x);
           2                   4                3
(%o10) - (c  - 4 c - b + 6)/((c  + (- 2 b - 6) c

     2              2         2               2
 + (b  + 12 b + 9) c  + (- 6 b  - 18 b) c + 9 b ) (x + c))

                 c - 2
 - ---------------------------------
     2                             2
   (c  + (- b - 3) c + 3 b) (x + c)

                         b - 2
 + -------------------------------------------------
             2             2      3      2
   ((b - 3) c  + (6 b - 2 b ) c + b  - 3 b ) (x + b)

                         1
 - ----------------------------------------------
             2
   ((b - 3) c  + (18 - 6 b) c + 9 b - 27) (x + 3)
(%i11) map ('factor, %);
              2
             c  - 4 c - b + 6                 c - 2
(%o11) - ------------------------- - ------------------------
                2        2                                  2
         (c - 3)  (c - b)  (x + c)   (c - 3) (c - b) (x + c)

                     b - 2                        1
           + ------------------------ - ------------------------
                             2                          2
             (b - 3) (c - b)  (x + b)   (b - 3) (c - 3)  (x + 3)
(%i12) ratsimp ((x^5 - 1)/(x - 1));
                          4    3    2
(%o12)                   x  + x  + x  + x + 1
(%i13) subst (a, x, %);
```

```
                         4    3    2
(%o13)                  a  + a  + a  + a + 1
(%i14) factor (%th(2), %);
                         2        3        3    2
(%o14)   (x - a) (x - a ) (x - a ) (x + a  + a  + a + 1)
(%i15) factor (1 + x^12);
                         4        8    4
(%o15)                 (x  + 1) (x  - x  + 1)
(%i16) factor (1 + x^99);
                    2            6    3
(%o16) (x + 1) (x  - x + 1) (x  - x  + 1)

   10    9    8    7    6    5    4    3    2
 (x   - x  + x  - x  + x  - x  + x  - x  + x  - x + 1)

   20    19    17    16    14    13    11    10    9    7    6
 (x   + x   - x   - x   + x   + x   - x   - x   - x  + x  + x

    4    3            60    57    51    48    42    39    33
 - x  - x  + x + 1) (x   + x   - x   - x   + x   + x   - x

    30    27    21    18    12    9    3
 - x   - x   + x   + x   - x   - x  + x  + 1)
```

factorflag Option variable

Default value: `false`

When `factorflag` is `false`, suppresses the factoring of integer factors of rational expressions.

factorout (*expr, x_1, x_2, ...*) Function

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

factorsum (*expr*) Function

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

Example:

```
(%i1) expand ((x + 1)*((u + v)^2 + a*(w + z)^2));
           2      2                            2      2
(%o1) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x

                                     2        2    2            2
                        + 2 u v x + u  x + a w  + v  + 2 u v + u
(%i2) factorsum (%);
                                   2          2
(%o2)            (x + 1) (a (z + w)  + (v + u) )
```

fasttimes (*p_1*, *p_2*) Function

Returns the product of the polynomials *p_1* and *p_2* by using a special algorithm for multiplication of polynomials. `p_1` and `p_2` should be multivariate, dense, and nearly the same size. Classical multiplication is of order `n_1 n_2` where `n_1` is the degree of `p_1` and `n_2` is the degree of `p_2`. `fasttimes` is of order `max (n_1, n_2)^1.585`.

fullratsimp (*expr*) Function

`fullratsimp` repeatedly applies `ratsimp` followed by non-rational simplification to an expression until no further change occurs, and returns the result.

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. `fullratsimp` makes this process convenient.

`fullratsimp (`*expr*`, `*x_1*`, ..., `*x_n*`)` takes one or more arguments similar to `ratsimp` and `rat`.

Example:

```
(%i1) expr: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
                       a/2     2   a/2     2
                     (x    - 1)  (x    + 1)
(%o1)                -----------------------
                              a
                             x  - 1
(%i2) ratsimp (expr);
                          2 a      a
                         x    - 2 x  + 1
(%o2)                    ---------------
                              a
                             x  - 1
(%i3) fullratsimp (expr);
                              a
(%o3)                        x  - 1
(%i4) rat (expr);
                       a/2 4       a/2 2
                     (x   )  - 2 (x   )  + 1
(%o4)/R/             -----------------------
                              a
                             x  - 1
```

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.

`load ("lrats")` loads `fullratsubst` and `lratsubst`.

Examples:

```
(%i1) load ("lrats")$
```

- `subst` can carry out multiple substitutions. `lratsubst` is analogous to `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                           d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                            a b
```

- `fullratsubst` is equivalent to `ratsubst` except that it recurses until its result stops changing.

```
(%i5) ratsubst (b*a, a^2, a^3);
                                   2
(%o5)                           a  b
(%i6) fullratsubst (b*a, a^2, a^3);
                                     2
(%o6)                           a b
```

- `fullratsubst` also accepts a list of equations or a single equation as first argument.

```
(%i7) fullratsubst ([a^2 = b, b^2 = c, c^2 = a], a^3*b*c);
(%o7)                               b
(%i8) fullratsubst (a^2 = b*a, a^3);
                                     2
(%o8)                           a b
```

- `fullratsubst` may cause an indefinite recursion.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));

*** - Lisp stack overflow. RESET
```

gcd (*p_1, p_2, x_1, ...*) Function

Returns the greatest common divisor of *p_1* and *p_2*. The flag `gcd` 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` equal to `subres` be used. To take the gcd when an algebraic is present, e.g. GCD(X^2-2*SQRT(2)*X+2,X-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: `subres`, if `false` will also prevent the greatest common divisor from being taken when expressions are converted to canonical rational expression (CRE) form. This will sometimes speed the calculation if gcds are not required.

gcdex (*f*, *g*) Function

gcdex (*f*, *g*, *x*) Function

Returns a list `[a, b, u]` where *u* is the greatest common divisor (gcd) of *f* and *g*, and *u* is equal to *a f* + *b g*. The arguments *f* and *g* should be univariate polynomials, or else polynomials in *x* a supplied **main** variable since we need to be in a principal ideal domain for this to work. The gcd means the gcd regarding *f* and *g* as univariate polynomials with coefficients being rational functions in the other variables.

`gcdex` implements the Euclidean algorithm, where we have a sequence of `L[i]: [a[i], b[i], r[i]]` which are all perpendicular to `[f, g, -1]` and the next one is built as if `q = quotient(r[i]/r[i+1])` then `L[i+2]: L[i] - q L[i+1]`, and it terminates at `L[i+1]` when the remainder `r[i+2]` is zero.

```
(%i1) gcdex (x^2 + 1, x^3 + 4);
                                  2
                                 x  + 4 x - 1  x + 4
(%o1)/R/                      [- ------------, -----, 1]
                                      17        17
(%i2) % . [x^2 + 1, x^3 + 4, -1];
(%o2)/R/                               0
```

Note that the gcd in the following is `1` since we work in `k(y)[x]`, not the `y+1` we would expect in `k[y, x]`.

```
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
                                       1
(%o1)/R/                        [0, ------, 1]
                                     2
                                    y  - 1
```

gcfactor (*n*) Function

Factors the Gaussian integer *n* over the Gaussian integers, 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 (*expr*) Function

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

Example:

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

gfactorsum (*expr*) Function

is similar to `factorsum` but applies `gfactor` instead of `factor`.

hipow (*expr*, *x*) Function

Returns the highest explicit exponent of *x* in *expr*. *x* may be a variable or a general expression. If *x* does not appear in *expr*, `hipow` returns `0`.

`hipow` does not consider expressions equivalent to `expr`. In particular, `hipow` does not expand `expr`, so `hipow (`*expr*`, `*x*`)` and `hipow (expand (`*expr*`, `*x*`))` may yield different results.

Examples:

```
(%i1) hipow (y^3 * x^2 + x * y^4, x);
(%o1)                           2
(%i2) hipow ((x + y)^5, x);
(%o2)                           1
(%i3) hipow (expand ((x + y)^5), x);
(%o3)                           5
(%i4) hipow ((x + y)^5, x + y);
(%o4)                           5
(%i5) hipow (expand ((x + y)^5), x + y);
(%o5)                           0
```

intfaclim Option variable

Default value: 1000

`intfaclim` is the largest divisor which will be tried when factoring a bignum integer.

When `intfaclim` is `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), factors of any size are considered. `intfaclim` is set to `false` when factors are computed in `divsum`, `totient`, and `primep`.

Internal calls to `factor` respect the user-specified value of `intfaclim`. Setting `intfaclim` to a smaller value may reduce the time spent factoring large integers.

keepfloat Option variable

Default value: `false`

When `keepfloat` is `true`, prevents floating point numbers from being rationalized when expressions which contain them are converted to canonical rational expression (CRE) form.

lratsubst (*L*, *expr*) Function

is analogous to `subst (`*L*`, `*expr*`)` except that it uses `ratsubst` instead of `subst`.

The first argument of `lratsubst` is an equation or a list of equations identical in format to that accepted by `subst`. The substitutions are made in the order given by the list of equations, that is, from left to right.

`load ("lrats")` loads `fullratsubst` and `lratsubst`.

Examples:

```
(%i1) load ("lrats")$
```

- `subst` can carry out multiple substitutions. `lratsubst` is analogous to `subst`.

```
(%i2) subst ([a = b, c = d], a + c);
(%o2)                         d + b
(%i3) lratsubst ([a^2 = b, c^2 = d], (a + e)*c*(a + c));
(%o3)                (d + a c) e + a d + b c
```

- If only one substitution is desired, then a single equation may be given as first argument.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                          a b
```

modulus Option variable

Default value: `false`

When `modulus` is a positive number *p*, operations on rational numbers (as returned by `rat` and related functions) are carried out modulo *p*, using the so-called "balanced" modulus system in which *n* `modulo` *p* is defined as an integer *k* in `[-(`*p*`-1)/2, ..., 0, ..., (`*p*`-1)/2]` when *p* is odd, or `[-(`*p*`/2 - 1), ..., 0, ...., `*p*`/2]` when *p* is even, such that *a p* + *k* equals *n* for some integer *a*.

If *expr* is already in canonical rational expression (CRE) form when `modulus` is reset, then you may need to re-rat *expr*, e.g., `expr: rat (ratdisrep (expr))`, in order to get correct results.

Typically `modulus` is set to a prime number. If `modulus` is set to a positive non-prime integer, this setting is accepted, but a warning message is displayed. Maxima will allow zero or a negative integer to be assigned to `modulus`, although it is not clear if that has any useful consequences.

num (*expr*) Function

Returns the numerator of *expr* if it is a ratio. If *expr* is not a ratio, *expr* is returned.

`num` evaluates its argument.

quotient (*p_1, p_2*) Function
quotient (*p_1, p_2, x_1, ..., x_n*) Function

Returns the polynomial *p_1* divided by the polynomial *p_2*. The arguments *x_1*, ..., *x_n* are interpreted as in `ratvars`.

`quotient` returns the first element of the two-element list returned by `divide`.

rat (*expr*) Function
rat (*expr, x_1, ..., x_n*) Function

Converts *expr* to canonical rational expression (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`. The variables are ordered according to the *x_1*, ..., *x_n*, if specified, as in `ratvars`.

`rat` does not generally simplify functions other than addition `+`, subtraction `-`, multiplication `*`, division `/`, and exponentiation to an integer power, whereas `ratsimp` does handle those cases. Note that atoms (numbers and variables) in CRE form are not the same as they are in the general form. For example, `rat(x)- x` yields `rat(0)` which has a different internal representation than 0.

When `ratfac` is `true`, `rat` yields a partially factored form for CRE. 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` if `false` suppresses the printout of the message informing the user of the conversion of floating point numbers to rational numbers.

`keepfloat` if `true` prevents floating point numbers from being converted to rational numbers.

See also `ratexpand` and `ratsimp`.

Examples:

```
(%i1) ((x - 2*y)^4/(x^2 - 4*y^2)^2 + 1)*(y + a)*(2*y + x) /(4*y^2 + x^2);
                                           4
                                  (x - 2 y)
              (y + a) (2 y + x) (------------ + 1)
                                   2      2 2
                                 (x  - 4 y )
(%o1)         ------------------------------------
                                2    2
                             4 y  + x
(%i2) rat (%, y, a, x);
                            2 a + 2 y
(%o2)/R/                    ---------
                             x + 2 y
```

ratalgdenom Option variable

Default value: `true`

When `ratalgdenom` is `true`, allows rationalization of denominators with respect to radicals to take effect. `ratalgdenom` has an effect only when canonical rational expressions (CRE) are used in algebraic mode.

ratcoef (*expr*, *x*, *n*) Function
ratcoef (*expr*, *x*) Function

Returns the coefficient of the expression *x*`^`*n* in the expression *expr*. If omitted, *n* is assumed to be 1.

The return value is free (except possibly in a non-rational sense) of the variables in *x*. If no coefficient of this type exists, 0 is 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 (`*expr*`,` *x*`, 0)`, viewing *expr* as a sum, returns a sum of those terms which do not contain *x*. Therefore if *x* occurs to any negative powers, `ratcoef` should not be used.

Since *expr* is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned.

Example:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                           x
```

ratdenom (*expr*) Function

Returns the denominator of *expr*, after coercing *expr* to a canonical rational expression (CRE). The return value is a CRE.

expr is coerced to a CRE by `rat` if it is not already a CRE. This conversion may change the form of *expr* by putting all terms over a common denominator.

`denom` is similar, but returns an ordinary expression instead of a CRE. Also, `denom` does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by `ratdenom` are not considered ratios by `denom`.

ratdenomdivide Option variable

Default value: `true`

When `ratdenomdivide` is `true`, `ratexpand` expands a ratio in which the numerator is a sum into a sum of ratios, all having a common denominator. Otherwise, `ratexpand` collapses a sum of ratios into a single ratio, the numerator of which is the sum of the numerators of each ratio.

Examples:

```
(%i1) expr: (x^2 + x + 1)/(y^2 + 7);
                              2
                             x  + x + 1
(%o1)                        ----------
                                2
                               y  + 7
(%i2) ratdenomdivide: true$
(%i3) ratexpand (expr);
                         2
                        x        x        1
(%o3)                 ------ + ------ + ------
                       2        2        2
                      y  + 7   y  + 7   y  + 7
(%i4) ratdenomdivide: false$
(%i5) ratexpand (expr);
                              2
                             x  + x + 1
(%o5)                        ----------
                                2
                               y  + 7
(%i6) expr2: a^2/(b^2 + 3) + b/(b^2 + 3);
                                        2
                             b        a
(%o6)                      ------ + ------
                            2        2
                           b  + 3   b  + 3
```

```
(%i7) ratexpand (expr2);
                                   2
                              b + a
(%o7)                         ------
                               2
                              b  + 3
```

ratdiff (*expr, x*) Function

Differentiates the rational expression *expr* with respect to *x*. *expr* must be a ratio of polynomials or a polynomial in *x*. The argument *x* may be a variable or a subexpression of *expr*.

The result is equivalent to `diff`, although perhaps in a different form. `ratdiff` may be faster than `diff`, for rational expressions.

`ratdiff` returns a canonical rational expression (CRE) if `expr` is a CRE. Otherwise, `ratdiff` returns a general expression.

`ratdiff` considers only the dependence of *expr* on *x*, and ignores any dependencies established by `depends`.

Example:

```
(%i1) expr: (4*x^3 + 10*x - 11)/(x^5 + 5);
                         3
                      4 x  + 10 x - 11
(%o1)                 ----------------
                           5
                          x  + 5
(%i2) ratdiff (expr, x);
                    7       5       4       2
                 8 x  + 40 x  - 55 x  - 60 x  - 50
(%o2)          - ---------------------------------
                          10       5
                         x   + 10 x  + 25
(%i3) expr: f(x)^3 - f(x)^2 + 7;
                         3       2
(%o3)                 f (x) - f (x) + 7
(%i4) ratdiff (expr, f(x));
                           2
(%o4)                 3 f (x) - 2 f(x)
(%i5) expr: (a + b)^3 + (a + b)^2;
                              3          2
(%o5)                (b + a)  + (b + a)
(%i6) ratdiff (expr, a + b);
                    2                    2
(%o6)            3 b  + (6 a + 2) b + 3 a  + 2 a
```

ratdisrep (*expr*) Function

Returns its argument as a general expression. If *expr* is a general expression, it is returned unchanged.

Typically `ratdisrep` is called to convert a canonical rational expression (CRE) into a general expression. This is sometimes convenient if one wishes to stop the "contagion", or use rational functions in non-rational contexts.

See also `totaldisrep`.

ratepsilon Option variable

Default value: 2.0e-8

`ratepsilon` is the tolerance used in the conversion of floating point numbers to rational numbers.

ratexpand (*expr*) Function
ratexpand Option variable

Expands *expr* 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.

The return value of `ratexpand` is a general expression, even if *expr* is a canonical rational expression (CRE).

The switch `ratexpand` 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 also `ratsimp`.

When `ratdenomdivide` is `true`, `ratexpand` expands a ratio in which the numerator is a sum into a sum of ratios, all having a common denominator. Otherwise, `ratexpand` collapses a sum of ratios into a single ratio, the numerator of which is the sum of the numerators of each ratio.

When `keepfloat` is `true`, prevents floating point numbers from being rationalized when expressions which contain them are converted to canonical rational expression (CRE) form.

Examples:

```
(%i1) ratexpand ((2*x - 3*y)^3);
                     3         2       2        3
(%o1)          - 27 y  + 54 x y  - 36 x  y + 8 x
(%i2) expr: (x - 1)/(x + 1)^2 + 1/(x - 1);
                         x - 1       1
(%o2)                   -------- + -----
                               2   x - 1
                        (x + 1)
(%i3) expand (expr);
                    x              1           1
(%o3)          ------------ - ------------ + -----
                2              2             x - 1
               x  + 2 x + 1   x  + 2 x + 1
(%i4) ratexpand (expr);
                        2
                     2 x                 2
(%o4)           --------------- + ---------------
```

```
  3    2            3    2
 x  + x  - x - 1   x  + x  - x - 1
```

ratfac Option variable

Default value: `false`

When `ratfac` is `true`, canonical rational expressions (CRE) are manipulated in a partially factored form.

During rational operations the expression is maintained as fully factored as possible without calling `factor`. This should always save space and may save time in some computations. The numerator and denominator are made relatively prime, for example `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, Ricci, Einstein, Riemann, and Weyl tensors and the scalar curvature are factored automatically when `ratfac` is `true`. `ratfac` *should only be set for cases where the tensorial components are known to consist of few terms.*

The `ratfac` and `ratweight` schemes are incompatible and may not both be used at the same time.

ratnumer (*expr*) Function

Returns the numerator of *expr*, after coercing *expr* to a canonical rational expression (CRE). The return value is a CRE.

expr is coerced to a CRE by `rat` if it is not already a CRE. This conversion may change the form of *expr* by putting all terms over a common denominator.

`num` is similar, but returns an ordinary expression instead of a CRE. Also, `num` does not attempt to place all terms over a common denominator, and thus some expressions which are considered ratios by `ratnumer` are not considered ratios by `num`.

ratnump (*expr*) Function

Returns `true` if *expr* is a literal integer or ratio of literal integers, otherwise `false`.

ratp (*expr*) Function

Returns `true` if *expr* is a canonical rational expression (CRE) or extended CRE, otherwise `false`.

CRE are created by `rat` and related functions. Extended CRE are created by `taylor` and related functions.

ratprint Option variable

Default value: `true`

When `ratprint` is `true`, a message informing the user of the conversion of floating point numbers to rational numbers is displayed.

ratsimp (*expr*) Function
ratsimp (*expr, x_1, ..., x_n*) Function

Simplifies the expression *expr* 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, that is, the coefficients of the main variable are polynomials in the other variables. Variables may include non-rational functions (e.g., `sin (x^2 + 1)`) and the arguments to any such functions are also rationally simplified.

`ratsimp (`*expr*`, `*x_1*`, ..., `*x_n*`)` enables rational simplification with the specification of variable ordering as in `ratvars`.

When `ratsimpexpons` is `true`, `ratsimp` is applied to the exponents of expressions during simplification.

See also `ratexpand`. Note that `ratsimp` is affected by some of the flags which affect `ratexpand`.

Examples:

```
(%i1) sin (x/(x^2 + x)) = exp ((log(x) + 1)^2 - log(x)^2);
                                      2      2
                   x         (log(x) + 1)  - log (x)
(%o1)        sin(------) = %e
                  2
                 x  + x
(%i2) ratsimp (%);
                                1          2
(%o2)                   sin(-----) = %e x
                            x + 1
(%i3) ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1));
                        3/2
                 (x - 1)    - sqrt(x - 1) (x + 1)
(%o3)            --------------------------------
                      sqrt((x - 1) (x + 1))
(%i4) ratsimp (%);
                            2 sqrt(x - 1)
(%o4)                     - -------------
                                  2
                            sqrt(x  - 1)
(%i5) x^(a + 1/a), ratsimpexpons: true;
                               2
                              a  + 1
                              ------
                                a
(%o5)                        x
```

ratsimpexpons Option variable

Default value: `false`

When `ratsimpexpons` is `true`, `ratsimp` is applied to the exponents of expressions during simplification.

ratsubst (*a*, *b*, *c*) Function

Substitutes *a* for *b* in *c* and returns the resulting expression. *b* may be a sum, product, power, etc.

`ratsubst` knows something of the meaning of expressions whereas `subst` does a purely syntactic substitution. Thus `subst (a, x + y, x + y + z)` returns `x + y + z` whereas `ratsubst` returns `z + a`.

When `radsubstflag` is `true`, `ratsubst` makes substitutions for radicals in expressions which don't explicitly contain them.

Examples:

```
(%i1) ratsubst (a, x*y^2, x^4*y^3 + x^4*y^8);
                                3      4
(%o1)                        a x  y + a
(%i2) cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1;
                  4         3         2
(%o2)          cos (x) + cos (x) + cos (x) + cos(x) + 1
(%i3) ratsubst (1 - sin(x)^2, cos(x)^2, %);
                4           2                     2
(%o3)       sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
(%i4) ratsubst (1 - cos(x)^2, sin(x)^2, sin(x)^4);
                            4           2
(%o4)                    cos (x) - 2 cos (x) + 1
(%i5) radsubstflag: false$
(%i6) ratsubst (u, sqrt(x), x);
(%o6)                              x
(%i7) radsubstflag: true$
(%i8) ratsubst (u, sqrt(x), x);
                                   2
(%o8)                             u
```

ratvars (*x_1*, ..., *x_n*) Function
ratvars () Function
ratvars System variable

Declares main variables *x_1*, ..., *x_n* for rational expressions. *x_n*, if present in a rational expression, is considered the main variable. Otherwise, *x_[n-1]* is considered the main variable if present, and so on through the preceding variables to *x_1*, which is considered the main variable only if none of the succeeding variables are present.

If a variable in a rational expression is not present in the `ratvars` list, it is given a lower priority than *x_1*.

The arguments to `ratvars` can be either variables or non-rational functions such as `sin(x)`.

The variable `ratvars` is a list of the arguments of the function `ratvars` when it was called most recently. Each call to the function `ratvars` resets the list. `ratvars ()` clears the list.

ratweight (*x_1, w_1, ..., x_n, w_n*) Function

ratweight () Function

Assigns a weight *w_i* to the variable *x_i*. This causes a term to be replaced by 0 if its weight exceeds the value of the variable `ratwtlvl` (default yields 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. For example, the weight of `3 x_1^2 x_2` is `2 w_1 + w_2`. Truncation according to `ratwtlvl` is carried out only when multiplying or exponentiating canonical rational expressions (CRE).

`ratweight ()` returns the cumulative list of weight assignments.

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

Examples:

```
(%i1) ratweight (a, 1, b, 1);
(%o1)                     [a, 1, b, 1]
(%i2) expr1: rat(a + b + 1)$
(%i3) expr1^2;
                  2                  2
(%o3)/R/         b  + (2 a + 2) b + a  + 2 a + 1
(%i4) ratwtlvl: 1$
(%i5) expr1^2;
(%o5)/R/                  2 b + 2 a + 1
```

ratweights System variable

Default value: `[]`

`ratweights` is the list of weights assigned by `ratweight`. The list is cumulative: each call to `ratweight` places additional items in the list.

`kill (ratweights)` and `save (ratweights)` both work as expected.

ratwtlvl Option variable

Default value: `false`

`ratwtlvl` is used in combination with the `ratweight` function to control the truncation of canonical rational expressions (CRE). For the default value of `false`, no truncation occurs.

remainder (*p_1, p_2*) Function

remainder (*p_1, p_2, x_1, ..., x_n*) Function

Returns the remainder of the polynomial *p_1* divided by the polynomial *p_2*. The arguments *x_1, ..., x_n* are interpreted as in `ratvars`.

`remainder` returns the second element of the two-element list returned by `divide`.

resultant (*p_1, p_2, x*) Function

resultant Variable

Computes the resultant of the two polynomials *p_1* and *p_2*, eliminating the variable *x*. The resultant is a determinant of the coefficients of *x* in *p_1* and *p_2*, which equals zero if and only if *p_1* and *p_2* have a non-constant factor in common.

If *p_1* or *p_2* can be factored, it may be desirable to call `factor` before calling `resultant`.

The variable `resultant` controls which algorithm will be used to compute the resultant. `subres` for subresultant prs, `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.

The function `bezout` takes the same arguments as `resultant` and returns a matrix. The determinant of the return value is the desired resultant.

savefactors Option variable

Default value: `false`

When `savefactors` is `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 (*expr*) 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 greater than n. Thus by taking greatest common divisors with the polynomial of the derivatives with respect to each variable in the polynomial, all factors of degree greater than 1 can be found.

Example:

```
(%i1) sqfr (4*x^4 + 4*x^3 - 3*x^2 - 4*x - 1);
                                2   2
(%o1)                  (2 x + 1)  (x  - 1)
```

tellrat (*p_1, ..., p_n*) Function
tellrat () Function

Adds to the ring of algebraic integers known to Maxima the elements which are the solutions of the polynomials *p_1*, ..., *p_n*. Each argument *p_i* is a polynomial with integer coefficients.

`tellrat (x)` effectively means substitute 0 for *x* in rational functions.

`tellrat ()` returns a list of the current substitutions.

`algebraic` must be set to `true` in order for the simplification of algebraic integers to take effect.

Maxima initially knows about the imaginary unit `%i` and all roots of integers.

There is a command `untellrat` which takes kernels and removes `tellrat` properties.

When `tellrat`'ing a multivariate polynomial, e.g., `tellrat (x^2 - y^2)`, there would be an ambiguity as to whether to substitute *y*`^2` for *x*`^2` or vice versa. Maxima picks a particular ordering, but if the user wants to specify which, e.g. `tellrat (y^2 = x^2)` provides a syntax which says replace *y*`^2` by *x*`^2`.

Examples:

```
(%i1) 10*(%i + 1)/(%i + 3^(1/3));
                           10 (%i + 1)
(%o1)                      -----------
                                  1/3
                            %i + 3
(%i2) ev (ratdisrep (rat(%)), algebraic);
             2/3      1/3              2/3      1/3
(%o2)    (4 3    - 2 3    - 4) %i + 2 3    + 4 3    - 2
(%i3) tellrat (1 + a + a^2);
                            2
(%o3)                     [a  + a + 1]
(%i4) 1/(a*sqrt(2) - 1) + a/(sqrt(3) + sqrt(2));
                      1                 a
(%o4)           ------------- + -----------------
                sqrt(2) a - 1   sqrt(3) + sqrt(2)
(%i5) ev (ratdisrep (rat(%)), algebraic);
         (7 sqrt(3) - 10 sqrt(2) + 2) a - 2 sqrt(2) - 1
(%o5)    ----------------------------------------------
                               7
(%i6) tellrat (y^2 = x^2);
                        2    2   2
(%o6)                 [y  - x , a  + a + 1]
```

totaldisrep (*expr*) Function

Converts every subexpression of *expr* from canonical rational expressions (CRE) to general form and returns the result. If *expr* is itself in CRE form then `totaldisrep` is identical to `ratdisrep`.

`totaldisrep` may be useful for ratdisrepping expressions such as equations, lists, matrices, etc., which have some subexpressions in CRE form.

untellrat (x_1, ..., x_n) Function

Removes `tellrat` properties from x_1, ..., x_n.

13 Constants

13.1 Definitions for Constants

%e Constant

- The base of natural logarithms, e, is represented in Maxima as `%e`.

false Constant

- the Boolean constant, false. (NIL in Lisp)

inf Constant

- real positive infinity.

infinity Constant

- complex infinity.

minf Constant

- real minus infinity.

%pi Constant

- `"pi"` is represented in Maxima as `%pi`.

true Constant

- the Boolean constant, true. (T in Lisp)

14 Logarithms

14.1 Definitions for Logarithms

%e_to_numlog Option variable

Default value: `false`

When `true`, `r` some rational number, and `x` some expression, `%e^(r*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`.

log (*x*) Function

Represents the natural logarithm of *x*.

Simplification and evaluation of logarithms is governed by several global flags:

`logexpand` - causes `log(a^b)` to become `b*log(a)`. If it is set to `all`, `log(a*b)` will also simplify to `log(a)+log(b)`. If it is set to `super`, then `log(a/b)` will also simplify to `log(a)-log(b)` for rational numbers `a/b`, `a#1`. (`log(1/b)`, for `b` integer, always simplifies.) If it is set to `false`, all of these simplifications will be turned off.

`logsimp` - if `false` then no simplification of `%e` to a power containing `log`'s is done.

`lognumer` - 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` - if `true` implements the rule `log(-n) -> log(n)+%i*%pi` for `n` a positive integer.

`%e_to_numlog` - when `true`, `r` some rational number, and `x` some expression, `%e^(r*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 Option variable

Default value: `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 Option variable

Default value: `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 *exp* without setting the switch or having to re-evaluate the expression with `ev`.

logconcoeffp Option variable

Default value: `false`

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

logcontract (*expr*) Function

Recursively scans the expression *expr*, transforming subexpressions of the form `a1*log(b1) + a2*log(b2) + c` into `log(ratsimp(b1^a1 * b2^a2)) + c`

```
(%i1) 2*(a*log(x) + 2*a*log(y))$
(%i2) logcontract(%);
                                 2  4
(%o2)                     a log(x  y )
```

If you do `declare(n,integer);` then `logcontract(2*a*n*log(x));` gives `a*log(x^(2*n))`. The coefficients that "contract" in this manner are those such as the 2 and the `n` here which satisfy `featurep(coeff,integer)`. The user can control which coefficients are contracted by setting the option `logconcoeffp` to the name of a predicate function of one argument. E.g. if you like to generate SQRTs, you can do `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) or ratnump(m)$` . Then `logcontract(1/2*log(x));` will give `log(sqrt(x))`.

logexpand Option variable

Default value: `true`

Causes `log(a^b)` to become `b*log(a)`. If it is set to `all`, `log(a*b)` will also simplify to `log(a)+log(b)`. If it is set to `super`, then `log(a/b)` will also simplify to `log(a)-log(b)` for rational numbers `a/b`, `a#1`. (`log(1/b)`, for integer `b`, always simplifies.) If it is set to `false`, all of these simplifications will be turned off.

lognegint Option variable

Default value: `false`

If `true` implements the rule `log(-n) -> log(n)+%i*%pi` for `n` a positive integer.

lognumer Option variable

Default value: `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 Option variable

Default value: `true`

If `false` then no simplification of `%e` to a power containing `log`'s is done.

plog (*x*) Function

Represents the principal branch of the complex-valued natural logarithm with `-%pi < carg(x) <= +%pi` .

15 Trigonometric

15.1 Introduction to Trigonometric

Maxima has many trigonometric functions defined. Not all trigonometric identities are programmed, but it is possible for the user to add many of them using the pattern matching capabilities of the system. The trigonometric functions defined in Maxima 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 trigonometric functions, see `trigexpand`, `trigreduce`, and the switch `trigsign`. Two share packages extend the simplification rules built into Maxima, `ntrig` and `atrig1`. Do `describe(`*command*`)` for details.

15.2 Definitions for Trigonometric

acos (*x*) Function

- Arc Cosine.

acosh (*x*) Function

- Hyperbolic Arc Cosine.

acot (*x*) Function

- Arc Cotangent.

acoth (*x*) Function

- Hyperbolic Arc Cotangent.

acsc (*x*) Function

- Arc Cosecant.

acsch (*x*) Function

- Hyperbolic Arc Cosecant.

asec (*x*) Function

- Arc Secant.

asech (*x*) Function

- Hyperbolic Arc Secant.

asin (*x*) Function

- Arc Sine.

asinh (*x*) Function

- Hyperbolic Arc Sine.

atan (x) Function

- Arc Tangent.

atan2 (y, x) Function

- yields the value of `atan(y/x)` in the interval `-%pi` to `%pi`.

atanh (x) Function

- Hyperbolic Arc Tangent.

atrig1 Package

The `atrig1` package contains several additional simplification rules for inverse trigonometric functions. Together with rules already known to Maxima, 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.

cos (x) Function

- Cosine.

cosh (x) Function

- Hyperbolic Cosine.

cot (x) Function

- Cotangent.

coth (x) Function

- Hyperbolic Cotangent.

csc (x) Function

- Cosecant.

csch (x) Function

- Hyperbolic Cosecant.

halfangles Option variable

Default value: `false`

When `halfangles` is `true`, half-angles are simplified away.

ntrig Package

The `ntrig` package contains a set of simplification rules that are used to simplify trigonometric function whose arguments are of the form `f(n %pi/10)` where f is any of the functions `sin`, `cos`, `tan`, `csc`, `sec` and `cot`.

sec (x) Function

- Secant.

sech (*x*) Function

- Hyperbolic Secant.

sin (*x*) Function

- Sine.

sinh (*x*) Function

- Hyperbolic Sine.

tan (*x*) Function

- Tangent.

tanh (*x*) Function

- Hyperbolic Tangent.

trigexpand (*expr*) Function

Expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in *expr*. For best results, *expr* 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` is governed by the following global flags:

`trigexpand`
: If `true` causes expansion of all expressions containing sin's and cos's occurring subsequently.

`halfangles`
: If `true` causes half-angles to be simplified away.

`trigexpandplus`
: Controls the "sum" rule for `trigexpand`, expansion of sums (e.g. `sin(x + y)`) will take place only if `trigexpandplus` is `true`.

`trigexpandtimes`
: Controls the "product" rule for `trigexpand`, expansion of products (e.g. `sin(2 x)`) will take place only if `trigexpandtimes` is `true`.

Examples:

```
(%i1) x+sin(3*x)/sin(x),trigexpand=true,expand;
                           2           2
(%o1)                - sin (x) + 3 cos (x) + x
(%i2) trigexpand(sin(10*x+y));
(%o2)           cos(10 x) sin(y) + sin(10 x) cos(y)
```

trigexpandplus Option variable

Default value: `true`

`trigexpandplus` 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 Option variable

Default value: `true`

`trigexpandtimes` 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 Option variable

Default value: `all`

`triginverses` controls the simplification of the composition of trigonometric 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 *arcfun*`(`*fun*`(`*x*`))` simplification is turned off.

If `false`, both the *arcfun*`(`*fun*`(`*x*`))` and *fun*`(`*arcfun*`(`*x*`))` simplifications are turned off.

trigreduce (*expr*, *x*) Function

trigreduce (*expr*) Function

Combines products and powers of trigonometric and hyperbolic sin's and cos's of *x* into those of multiples of *x*. It also tries to eliminate these functions when they occur in denominators. If *x* is omitted then all variables in *expr* are used.

See also `poissimp`.

```
(%i1) trigreduce(-sin(x)^2+3*cos(x)^2+x);
               cos(2 x)      cos(2 x)   1        1
(%o1)          -------- + 3 (-------- + -) + x - -
                  2             2       2        2
```

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

```
(%i1) declare(j, integer, e, even, o, odd)$
(%i2) sin(x + (e + 1/2)*%pi);
(%o2)                        cos(x)
(%i3) sin(x + (o + 1/2)*%pi);
(%o3)                       - cos(x)
```

trigsign Option variable

Default value: `true`

When `trigsign` is `true`, it 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`.

`trigreduce`, `ratsimp`, and `radcan` may be able to further simplify the result.

`demo ("trgsmp.dem")` displays some examples of `trigsimp`.

trigrat (*expr*) Function

Gives a canonical simplifyed quasilinear form of a trigonometrical expression; *expr* 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.

```
(%i1) trigrat(sin(3*a)/sin(a+%pi/3));
(%o1)            sqrt(3) sin(2 a) + cos(2 a) - 1
```

The following example is taken from Davenport, Siret, and Tournier, *Calcul Formel*, Masson (or in English, Addison-Wesley), section 1.5.5, Morley theorem.

```
(%i1) c: %pi/3 - a - b;
                                           %pi
(%o1)                          - b - a + ---
                                            3
(%i2) bc: sin(a)*sin(3*c)/sin(a+b);
                           sin(a) sin(3 b + 3 a)
(%o2)                      ---------------------
                                sin(b + a)
(%i3) ba: bc, c=a, a=c$
(%i4) ac2: ba^2 + bc^2 - 2*bc*ba*cos(b);
         2       2
      sin (a) sin (3 b + 3 a)
(%o4) -----------------------
                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
(%i5) trigrat (ac2);
(%o5) - (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
```

16 Special Functions

16.1 Introduction to Special Functions

16.2 specint

`hypgeo` is a package for handling Laplace transforms of special functions. `hyp` is a package for handling generalized Hypergeometric functions.

`specint` attempts to compute the definite integral (over the range from zero to infinity) of an expression containing special functions. When the integrand contains a factor `exp (-s t)`, the result is a Laplace transform.

The syntax is as follows:

```
specint (exp (-s*t) * expr, t);
```

where *t* is the variable of integration and *expr* is an expression containing special functions.

If `specint` cannot compute the integral, the return value may contain various Lisp symbols, including `other-defint-to-follow-negtest`, `other-lt-exponential-to-follow`, `product-of-y-with-nofract-indices`, etc.; this is a bug.

Special function notation follows:

```
bessel_j (index, expr)         Bessel function, 1st kind
bessel_y (index, expr)         Bessel function, 2nd kind
bessel_i (index, expr)         Modified Bessel function, 1st kind
bessel_k (index, expr)         Modified Bessel function, 2nd kind
%he[n] (z)                     Hermite polynomial (Nota bene: he, not h. See A&S 22
%p[u,v] (z)                    Legendre function
%q[u,v] (z)                    Legendre function, 2nd kind
hstruve[n] (z)                 Struve H function
lstruve[n] (z)                 Struve L function
%f[p,q] ([], [], expr)         Generalized Hypergeometric function
gamma()                        Gamma function
gammagreek(a,z)                Incomplete gamma function
gammaincomplete(a,z)           Tail of incomplete gamma function
slommel
%m[u,k] (z)                    Whittaker function, 1st kind
%w[u,k] (z)                    Whittaker function, 2nd kind
erfc (z)                       Complement of the erf function
ei (z)                         Exponential integral (?)
kelliptic (z)                  Complete elliptic integral of the first kind (K)
%d [n] (z)                     Parabolic cylinder function
```

`demo ("hypgeo")` displays several examples of Laplace transforms computed by `specint`.

This is a work in progress. Some of the function names may change.

16.3 Definitions for Special Functions

airy (*x*) Function

The Airy function Ai. If the argument *x* is a number, the numerical value of `airy (x)` is returned. Otherwise, an unevaluated expression `airy (x)` is returned.

The Airy equation `diff (y(x), x, 2) - x y(x) = 0` has two linearly independent solutions, named `ai` and `bi`. This equation is very popular as an approximation to more complicated problems in many mathematical physics settings.

`load ("airy")` loads the functions `ai`, `bi`, `dai`, and `dbi`.

The `airy` package contains routines to compute `ai` and `bi` and their derivatives `dai` and `dbi`. The result is a floating point number if the argument is a number, and an unevaluated expression otherwise.

An error occurs if the argument is large enough to cause an overflow in the exponentials, or a loss of accuracy in `sin` or `cos`. This makes the range of validity about -2800 to 10^38 for `ai` and `dai`, and -2800 to 25 for `bi` and `dbi`.

These derivative rules are known to Maxima:

- `diff (ai(x), x)` yields `dai(x)`,
- `diff (dai(x), x)` yields `x ai(x)`,
- `diff (bi(x), x)` yields `dbi(x)`,
- `diff (dbi(x), x)` yields `x bi(x)`.

Function values are computed from the convergent Taylor series for `abs(x) < 3`, and from the asymptotic expansions for `x < -3` or `x > 3` as needed. This results in only very minor numerical discrepancies at `x = 3` and `x = -3`. For details, see Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 10.4 and Table 10.11.

`ev (taylor (ai(x), x, 0, 9), infeval)` yields a floating point Taylor expansions of the function `ai`. A similar expression can be constructed for `bi`.

airy_ai (*x*) Function

The Airy function Ai, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 10.4.

The Airy equation `diff (y(x), x, 2) - x y(x) = 0` has two linearly independent solutions, `y = Ai(x)` and `y = Bi(x)`. The derivative `diff (airy_ai(x), x)` is `airy_dai(x)`.

If the argument `x` is a real or complex floating point number, the numerical value of `airy_ai` is returned when possible.

See also `airy_bi`, `airy_dai`, `airy_dbi`.

airy_dai (*x*) Function

The derivative of the Airy function Ai `airy_ai(x)`.

See `airy_ai`.

airy_bi *(x)* Function

The Airy function Bi, as defined in Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 10.4, is the second solution of the Airy equation `diff (y(x), x, 2) - x y(x) = 0`.

If the argument `x` is a real or complex floating point number, the numerical value of `airy_bi` is returned when possible. In other cases the unevaluated expression is returned.

The derivative `diff (airy_bi(x), x)` is `airy_dbi(x)`.

See `airy_ai`, `airy_dbi`.

airy_dbi *(x)* Function

The derivative of the Airy Bi function `airy_bi(x)`.

See `airy_ai` and `airy_bi`.

asympa Function

`asympa` is a package for asymptotic analysis. The package contains simplification functions for asymptotic analysis, including the "big O" and "little o" functions that are widely used in complexity analysis and numerical analysis.

`load ("asympa")` loads this package.

bessel *(z, a)* Function

The Bessel function of the first kind.

This function is deprecated. Write `bessel_j (`*z*`,` *a*`)` instead.

bessel_j *(v, z)* Function

The Bessel function of the first kind of order v and argument z.

`bessel_j` computes the array `besselarray` such that `besselarray [i] = bessel_j [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_j` is defined as

$$\sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{v+2\,k}}{k!\,\Gamma(v+k+1)}$$

although the infinite series is not used for computations.

bessel_y *(v, z)* Function

The Bessel function of the second kind of order v and argument z.

`bessel_y` computes the array `besselarray` such that `besselarray [i] = bessel_y [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_y` is defined as

$$\frac{\cos(\pi\,v)\;J_v(z) - J_{-v}(z)}{\sin(\pi\,v)}$$

when v is not an integer. When v is an integer n, the limit as v approaches n is taken.

bessel_i (*v*, *z*) Function

The modified Bessel function of the first kind of order v and argument z.

`bessel_i` computes the array `besselarray` such that `besselarray [i] = bessel_i [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_i` is defined as

$$\sum_{k=0}^{\infty} \frac{1}{k!\,\Gamma\left(v+k+1\right)} \left(\frac{z}{2}\right)^{v+2\,k}$$

although the infinite series is not used for computations.

bessel_k (*v*, *z*) Function

The modified Bessel function of the second kind of order v and argument z.

`bessel_k` computes the array `besselarray` such that `besselarray [i] = bessel_k [i + v - int(v)] (z)` for `i` from zero to `int(v)`.

`bessel_k` is defined as

$$\frac{\pi\ \csc\left(\pi\,v\right)\ \left(I_{-v}(z)-I_v(z)\right)}{2}$$

when v is not an integer. If v is an integer n, then the limit as v approaches n is taken.

besselexpand Option variable

Default value: `false`

Controls expansion of the Bessel functions when the order is half of an odd integer. In this case, the Bessel functions can be expanded in terms of other elementary functions. When `besselexpand` is `true`, the Bessel function is expanded.

```
(%i1) besselexpand: false$
(%i2) bessel_j (3/2, z);
                                    3
(%o2)                      bessel_j(-, z)
                                    2
(%i3) besselexpand: true$
(%i4) bessel_j (3/2, z);
                               2 z   sin(z)   cos(z)
(%o4)                     sqrt(---) (------ - ------)
                               %pi      2       z
                                       z
```

j0 (*x*) Function

The Bessel function of the first kind of order 0.

This function is deprecated. Write `bessel_j (0, x)` instead.

j1 (*x*) Function

The Bessel function of the first kind of order 1.

This function is deprecated. Write `bessel_j (1, x)` instead.

jn (*x*, *n*) Function

The Bessel function of the first kind of order *n*.

This function is deprecated. Write `bessel_j (`*n*`, `*x*`)` instead.

i0 (*x*) Function

The modified Bessel function of the first kind of order 0.

This function is deprecated. Write `bessel_i (0, `*x*`)` instead.

i1 (*x*) Function

The modified Bessel function of the first kind of order 1.

This function is deprecated. Write `bessel_i (1, x)` instead.

beta (*x*, *y*) Function

The beta function, defined as `gamma(x) gamma(y)/gamma(x + y)`.

gamma (*x*) Function

The gamma function.

See also `makegamma`.

The variable `gammalim` controls simplification of the gamma function.

The Euler-Mascheroni constant is `%gamma`.

gammalim Option variable

Default value: 1000000

`gammalim` 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 (*expr*) Function

Transforms instances of binomial, gamma, and beta functions in *expr* into factorials.

See also `makegamma`.

makegamma (*expr*) Function

Transforms instances of binomial, factorial, and beta functions in *expr* into gamma functions.

See also `makefact`.

numfactor (*expr*) Function

Returns the numerical factor multiplying the expression *expr*, which should be a single term.

`content` returns the greatest common divisor (gcd) of all terms in a sum.

```
(%i1) gamma (7/2);
                              15 sqrt(%pi)
(%o1)                         ------------
                                   8
(%i2) numfactor (%);
                                   15
(%o2)                              --
                                   8
```

outofpois (*a*) Function

Converts *a* from Poisson encoding to general representation. If *a* is not in Poisson form, `outofpois` carries out the conversion, i.e., the return value is `outofpois (intopois (a))`. This function is thus a canonical simplifier for sums of powers of sine and cosine terms 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

Functionally identical to `intopois (a^b)`. *b* must be a positive integer.

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 Option variable

Default value: 5

`poislim` 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)` yields `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 `outofpois`, it will convert *a* into a Poisson encoding first, if necessary.

psi (*x*) Function

psi [*n*](*x*) Function

The derivative of `log (gamma (x))`.

Maxima does not know how to compute a numerical value of `psi`. However, the function `bfpsi` in the `bffac` package can compute numerical values.

17 Orthogonal Polynomials

17.1 Introduction to Orthogonal Polynomials

The `specfun` package 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 `specfun` package is not part of Maxima proper; it is loaded at request of the user via `load` or automatically via the `autoload` system.

The following table lists each function in `specfun`, its Maxima name, restrictions on its arguments, and a reference to the algorithm `specfun` uses to evaluate it. With few exceptions, `specfun` follows the conventions of Abramowitz and Stegun. In all cases, m and n must be integers.

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* (second edition, 1970).

Function	*Maxima Name*	*Restrictions*	*Reference(s)*
Chebyshev T	chebyshev_t(n, x)	$n > -1$	A&S 22.5.31
Chebyshev U	chebyshev_u(n, x)	$n > -1$	A&S 22.5.32
generalized Laguerre	gen_laguerre(n,a,x)	$n > -1$	A&S page 789
Laguerre	laguerre(n,x)	$n > -1$	A&S 22.5.67
Hermite	hermite(n,x)	$n > -1$	A&S 22.4.40, 22.5.41
Jacobi	jacobi_p(n,a,b,x)	$n > -1, a, b > -1$	A&S page 789
associated Legendre P	assoc_legendre_p(n,m,x)	$n > -1$	A&S 22.5.37, 8.6.6, 8.2.5
associated Legendre Q	assoc_legendre_q(n,m,x)	$n > -1, m > -1$	G & R 8.706
Legendre P	legendre_p(n,m,x)	$n > -1$	A&S 22.5.35
Legendre Q	legendre_q(n,m,x)	$n > -1$	A&S 8.6.19
spherical Hankel 1st	spherical_hankel1(n, x)	$n > -1$	A&S 10.1.36
spherical Hankel 2nd	spherical_hankel2(n, x)	$n > -1$	A&S 10.1.17
spherical Bessel J	spherical_bessel_j(n,x)	$n > -1$	A&S 10.1.8, 10.1.15
spherical Bessel Y	spherical_bessel_y(n,x)	$n > -1$	A&S 10.1.9, 10.1.15
spherical harmonic	spherical_harmonic(n,m,x,y)	$n > -1, \|m\| <= n$	Merzbacher 9.64
ultraspherical (Gegenbauer)	ultraspherical(n,a,x)	$n > -1$	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 bigfloats), many functions in specfun call a float modedeclared 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 simplification rules (based on recursion relations) are defined for unevaluated functions. It is possible for an expression involving sums of unevaluated special functions to vanish, yet Maxima is unable to reduce it to zero.

load ("specfun") loads the specfun package. Alternatively, setup_autoload causes the package to be loaded when one of the specfun functions appears in an expression. setup_autoload may appear at the command line or in the maxima-init.mac file. See setup_autoload.

An example use of specfun is

```
(%i1) load ("specfun")$
(%i2) [hermite (0, x), hermite (1, x), hermite (2, x)];
                                                  2
(%o2)              [1, 2 x, - 2 (1 - 2 x )]
(%i3) diff (hermite (n, x), x);
(%o3)                2 n hermite(n - 1, x)
```

Generally, compiled code runs faster than translated code; however, translated code may be better for program development.

Some functions (namely jacobi_p, ultraspherical, chebyshev_t, chebyshev_u, and legendre_p), return a series representation when the order is a symbolic integer. The series representation is not used by specfun for any computations, but it may be simplified by Maxima automatically, or it may be possible to use the series to evaluate the function through further manipulations. For example:

```
(%i1) load ("specfun")$
(%i2) legendre_p (n, x);
(%o2)                  legendre_p(n, x)
(%i3) ultraspherical (n, 3/2, 2);
              genfact(3, n, - 1) jacobi_p(n, 1, 1, 2)
(%o3)         ---------------------------------------
                         genfact(2, n, - 1)
(%i4) declare (n, integer)$
(%i5) legendre_p (n, x);
       n - 1
       ====
       \                                               n - i%
(%o5) ( >     binomial(n, i%) binomial(n, n - i%) (x - 1)
       /
       ====
       i% = 1

                                i%          n          n   n
                         (x + 1)   + (x + 1)  + (x - 1) )/2
(%i6) ultraspherical (n, 3/2, 2);
                         n - 1
                         ====
                         \       i%
```

```
(%o6) genfact(3, n, - 1) ( >      3   binomial(n + 1, i%)
                            /
                            ====
                            i% = 1

                                                 n
 binomial(n + 1, n - i%) + (n + 1) 3  + n + 1)

                              n
/(genfact(2, n, - 1) 2 )
```

The first and last terms of the sum are added outside the summation. Removing these two terms avoids Maxima bugs 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:

```
(%i1) load ("specfun")$
(%i2) declare (n, integer)$
(%i3) e: legendre_p(n,x)$
(%i4) 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);
(%i5) ev (e, sum, n=0, sumhack=true);
(%o5)                          1
```

Most functions in `specfun` have a `gradef` property; derivatives with respect to the order or other function parameters are undefined, and an attempt to compute such a derivative yields an error message.

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 the Maxima version, as reported by `build_info()`, and the `specfun` version, as reported by `get ('specfun, 'version)`.

17.2 Definitions for Orthogonal Polynomials

assoc_legendre_p (n, m, x) Function

Returns the associated Legendre function of the first kind for integers $n > -1$ and $m > -1$. When $|m| > n$ and $n >= 0$, 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.

`load ("specfun")` loads this function.

See [assoc_legendre_q], page 157, [legendre_p], page 159, and [legendre_q], page 159.

assoc_legendre_q (n, m, x) Function

Returns the associated Legendre function of the second kind for integers $n > -1$ and $m > -1$.

Reference: Gradshteyn and Ryzhik 8.706 page 1000.

`load ("specfun")` loads this function.

See also [assoc_legendre_p], page 157, [legendre_p], page 159, and [legendre_q], page 159.

chebyshev_t (*n*, *x*) Function

Returns 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.

`load ("specfun")` loads this function.

See also [chebyshev_u], page 158.

chebyshev_u (*n*, *x*) Function

Returns 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.

`load ("specfun")` loads this function.

See also [chebyshev_t], page 158.

gen_laguerre (*n*, *a*, *x*) Function

Returns the generalized Laguerre polynomial for integers $n > -1$.

`load ("specfun")` loads this function.

Reference: table on page 789 in A&S.

hermite (*n*, *x*) Function

Returns the Hermite polynomial for integers $n > -1$.

`load ("specfun")` loads this function.

Reference: A&S 22.5.40 and 22.5.41, page 779.

jacobi_p (*n*, *a*, *b*, *x*) Function

Returns 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.

`load ("specfun")` loads this function.

Reference: table on page 789 in A&S.

laguerre (n, x) Function

Returns the Laguerre polynomial for integers $n > -1$.

Reference: A&S 22.5.16, page 778 and A&S page 789.

`load ("specfun")` loads this function.

See also [gen_laguerre], page 158.

legendre_p (n, x) Function

Returns the Legendre polynomial of the first kind for integers $n > -1$.

Reference: A&S 22.5.35 page 779.

`load ("specfun")` loads this function.

See [legendre_q], page 159.

legendre_q (n, x) Function

Returns the Legendre polynomial of the first kind for integers $n > -1$.

Reference: A&S 8.6.19 page 334.

`load ("specfun")` loads this function.

See also [legendre_p], page 159.

spherical_bessel_j (n, x) Function

Returns 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.

`load ("specfun")` loads this function.

See also [spherical_hankel1], page 159, [spherical_hankel2], page 159, and [spherical_bessel_y], page 159.

spherical_bessel_y (n, x) Function

Returns 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.

`load ("specfun")` loads this function.

See also [spherical_hankel1], page 159, [spherical_hankel2], page 159, and [spherical_bessel_y], page 159.

spherical_hankel1 (n, x) Function

Returns the spherical hankel function of the first kind for integers $n > -1$.

Reference: A&S 10.1.36 page 439.

`load ("specfun")` loads this function.

See also [spherical_hankel2], page 159, [spherical_bessel_j], page 159, and [spherical_bessel_y], page 159.

spherical_hankel2 (n, x) Function

Returns the spherical hankel function of the second kind for integers $n > -1$.

Reference: A&S 10.1.17 page 439.

`load ("specfun")` loads this function.

See also [spherical_hankel1], page 159, [spherical_bessel_j], page 159, and [spherical_bessel_y], page 159.

spherical_harmonic (n, m, x, y) Function

Returns the spherical harmonic function for integers $n > -1$ and $|m| <= n$.

Reference: Merzbacher 9.64.

`load ("specfun")` loads this function.

See also [assoc_legendre_p], page 157.

ultraspherical (n, a, x) Function

Returns the ultraspherical polynomials for integers $n > -1$. The ultraspherical polynomials are also known as Gegenbauer polynomials.

Reference: A&S 22.5.27

`load ("specfun")` loads this function.

See also [jacobi_p], page 158.

18 Elliptic Functions

18.1 Introduction to Elliptic Functions and Integrals

Maxima includes support for Jacobian elliptic functions and for complete and incomplete elliptic integrals. This includes symbolic manipulation of these functions and numerical evaluation as well. Definitions of these functions and many of their properties can by found in Abramowitz and Stegun, Chapter 16–17. As much as possible, we use the definitions and relationships given there.

In particular, all elliptic functions and integrals use the parameter m instead of the modulus k or the modular angle α. This is one area where we differ from Abramowitz and Stegun who use the modular angle for the elliptic functions. The following relationships are true:

$$m = k^2$$

and

$$k = \sin\alpha$$

The elliptic functions and integrals are primarily intended to support symbolic computation. Therefore, most of derivatives of the functions and integrals are known. However, if floating-point values are given, a floating-point result is returned.

Support for most of the other properties of elliptic functions and integrals other than derivatives has not yet been written.

Some examples of elliptic functions:

```
(%i1) jacobi_sn (u, m);
(%o1)                    jacobi_sn(u, m)
(%i2) jacobi_sn (u, 1);
(%o2)                        tanh(u)
(%i3) jacobi_sn (u, 0);
(%o3)                        sin(u)
(%i4) diff (jacobi_sn (u, m), u);
(%o4)            jacobi_cn(u, m) jacobi_dn(u, m)
(%i5) diff (jacobi_sn (u, m), m);
(%o5) jacobi_cn(u, m) jacobi_dn(u, m)

      elliptic_e(asin(jacobi_sn(u, m)), m)
 (u - ------------------------------------)/(2 m)
                     1 - m

            2
   jacobi_cn (u, m) jacobi_sn(u, m)
 + --------------------------------
               2 (1 - m)
```

Some examples of elliptic integrals:

```
(%i1) elliptic_f (phi, m);
(%o1)                  elliptic_f(phi, m)
```

```
(%i2) elliptic_f (phi, 0);
(%o2)                         phi
(%i3) elliptic_f (phi, 1);
                              phi   %pi
(%o3)                  log(tan(--- + ---))
                               2     4
(%i4) elliptic_e (phi, 1);
(%o4)                      sin(phi)
(%i5) elliptic_e (phi, 0);
(%o5)                         phi
(%i6) elliptic_kc (1/2);
                                    1
(%o6)                    elliptic_kc(-)
                                    2
(%i7) makegamma (%);
                                2 1
                           gamma (-)
                                  4
(%o7)                     -----------
                          4 sqrt(%pi)
(%i8) diff (elliptic_f (phi, m), phi);
                                1
(%o8)                 ---------------------
                                    2
                      sqrt(1 - m sin (phi))
(%i9) diff (elliptic_f (phi, m), m);
       elliptic_e(phi, m) - (1 - m) elliptic_f(phi, m)
(%o9) (-----------------------------------------------
                               m

                                   cos(phi) sin(phi)
                             - ---------------------)/(2 (1 - m))
                                               2
                               sqrt(1 - m sin (phi))
```

Support for elliptic functions and integrals was written by Raymond Toy. It is placed under the terms of the General Public License (GPL) that governs the distribution of Maxima.

18.2 Definitions for Elliptic Functions

jacobi_sn (*u*, *m*) Function

The Jacobian elliptic function $sn(u, m)$.

jacobi_cn (*u*, *m*) Function

The Jacobian elliptic function $cn(u, m)$.

jacobi_dn (*u*, *m*) Function

The Jacobian elliptic function $dn(u, m)$.

jacobi_ns (*u*, *m*) Function

The Jacobian elliptic function $ns(u,m) = 1/sn(u,m)$.

jacobi_sc (*u*, *m*) Function

The Jacobian elliptic function $sc(u,m) = sn(u,m)/cn(u,m)$.

jacobi_sd (*u*, *m*) Function

The Jacobian elliptic function $sd(u,m) = sn(u,m)/dn(u,m)$.

jacobi_nc (*u*, *m*) Function

The Jacobian elliptic function $nc(u,m) = 1/cn(u,m)$.

jacobi_cs (*u*, *m*) Function

The Jacobian elliptic function $cs(u,m) = cn(u,m)/sn(u,m)$.

jacobi_cd (*u*, *m*) Function

The Jacobian elliptic function $cd(u,m) = cn(u,m)/dn(u,m)$.

jacobi_nd (*u*, *m*) Function

The Jacobian elliptic function $nc(u,m) = 1/cn(u,m)$.

jacobi_ds (*u*, *m*) Function

The Jacobian elliptic function $ds(u,m) = dn(u,m)/sn(u,m)$.

jacobi_dc (*u*, *m*) Function

The Jacobian elliptic function $dc(u,m) = dn(u,m)/cn(u,m)$.

inverse_jacobi_sn (*u*, *m*) Function

The inverse of the Jacobian elliptic function $sn(u,m)$.

inverse_jacobi_cn (*u*, *m*) Function

The inverse of the Jacobian elliptic function $cn(u,m)$.

inverse_jacobi_dn (*u*, *m*) Function

The inverse of the Jacobian elliptic function $dn(u,m)$.

inverse_jacobi_ns (*u*, *m*) Function

The inverse of the Jacobian elliptic function $ns(u,m)$.

inverse_jacobi_sc (*u*, *m*) Function

The inverse of the Jacobian elliptic function $sc(u,m)$.

inverse_jacobi_sd (*u*, *m*) Function

The inverse of the Jacobian elliptic function $sd(u,m)$.

inverse_jacobi_nc (*u*, *m*) Function

The inverse of the Jacobian elliptic function $nc(u,m)$.

inverse_jacobi_cs (*u*, *m*) Function

The inverse of the Jacobian elliptic function $cs(u,m)$.

inverse_jacobi_cd (*u*, *m*) Function

The inverse of the Jacobian elliptic function $cd(u,m)$.

inverse_jacobi_nd (*u*, *m*) Function

The inverse of the Jacobian elliptic function $nc(u,m)$.

inverse_jacobi_ds (*u*, *m*) Function

The inverse of the Jacobian elliptic function $ds(u,m)$.

inverse_jacobi_dc (*u*, *m*) Function

The inverse of the Jacobian elliptic function $dc(u,m)$.

18.3 Definitions for Elliptic Integrals

elliptic_f (*phi*, *m*) Function

The incomplete elliptic integral of the first kind, defined as

$$\int_0^\phi \frac{d\theta}{\sqrt{1-m\sin^2\theta}}$$

See also [elliptic_e], page 164 and [elliptic_kc], page 165.

elliptic_e (*phi*, *m*) Function

The incomplete elliptic integral of the second kind, defined as

$$\int_0^\phi \sqrt{1-m\sin^2\theta}d\theta$$

See also [elliptic_e], page 164 and [elliptic_ec], page 165.

elliptic_eu (*u*, *m*) Function

The incomplete elliptic integral of the second kind, defined as

$$\int_0^u \mathrm{dn}(v,m)dv = \int_0^\tau \sqrt{\frac{1-mt^2}{1-t^2}}dt$$

where $\tau = \mathrm{sn}(u,m)$

This is related to $elliptic_e$ by

$$E(u,m) = E(\phi,m)$$

where $\phi = \sin^{-1}\mathrm{sn}(u,m)$ See also [elliptic_e], page 164.

elliptic_pi (*n*, *phi*, *m*) Function

The incomplete elliptic integral of the third kind, defined as

$$\int_0^\phi \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - m\sin^2\theta}}$$

Only the derivative with respect to *phi* is known by Maxima.

elliptic_kc (*m*) Function

The complete elliptic integral of the first kind, defined as

$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

For certain values of m, the value of the integral is known in terms of *Gamma* functions. Use `makegamma` to evaluate them.

elliptic_ec (*m*) Function

The complete elliptic integral of the second kind, defined as

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m\sin^2\theta}d\theta$$

For certain values of m, the value of the integral is known in terms of *Gamma* functions. Use `makegamma` to evaluate them.

19 Limits

19.1 Definitions for Limits

lhospitallim Option variable

Default: 4

`lhospitallim` is the maximum number of times L'Hospital's rule is used in `limit`. This prevents infinite looping in cases like `limit (cot(x)/csc(x), x, 0)`.

limit (*expr, x, val, dir*) Function
limit (*expr, x, val*) Function
limit (*expr*) Function

Computes the limit of *expr* as the real variable *x* 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).

`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).

`lhospitallim` is the maximum number of times L'Hospital's rule is used in `limit`. This prevents infinite looping in cases like `limit (cot(x)/csc(x), x, 0)`.

`tlimswitch` when true will cause the limit package to use Taylor series when possible.

`limsubst` prevents `limit` from attempting substitutions on unknown forms. This is to avoid bugs like `limit (f(n)/f(n+1), n, inf)` giving 1. Setting `limsubst` to `true` will allow such substitutions.

`limit` with one argument is often called upon to simplify constant expressions, for example, `limit (inf-1)`.

`example (limit)` displays some examples.

For the method see Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", Ph.D. thesis, MAC TR-92, October 1971.

limsubst Option variable

default value: `false` - prevents `limit` from attempting substitutions on unknown forms. This is to avoid bugs like `limit (f(n)/f(n+1), n, inf)` giving 1. Setting `limsubst` to `true` will allow such substitutions.

tlimit (*expr, x, val, dir*) Function
tlimit (*expr, x, val*) Function
tlimit (*expr*) Function

Returns `limit` with `tlimswitch` set to `true`.

tlimswitch Option variable

Default value: `false`

When `tlimswitch` is `true`, it causes the limit package to use Taylor series when possible.

20 Differentiation

20.1 Definitions for Differentiation

antid (*expr, x, u(x)*) Function

Returns a two-element list, such that an antiderivative of *expr* with respect to *x* can be constructed from the list. The expression *expr* may contain an unknown function *u* and its derivatives.

Let *L*, a list of two elements, be the return value of `antid`. Then *L*`[1] + 'integrate (`*L*`[2], `*x*`)` is an antiderivative of *expr* with respect to *x*.

When `antid` succeeds entirely, the second element of the return value is zero. Otherwise, the second element is nonzero, and the first element is nonzero or zero. If `antid` cannot make any progress, the first element is zero and the second nonzero.

`load ("antid")` loads this function. The `antid` package also defines the functions `nonzeroandfreeof` and `linear`.

`antid` is related to `antidiff` as follows. Let *L*, a list of two elements, be the return value of `antid`. Then the return value of `antidiff` is equal to *L*`[1] + 'integrate (`*L*`[2], `*x*`)` where *x* is the variable of integration.

Examples:

```
(%i1) load ("antid")$
(%i2) expr: exp (z(x)) * diff (z(x), x) * y(x);
                                z(x)  d
(%o2)                     y(x) %e     (-- (z(x)))
                                       dx
(%i3) a1: antid (expr, x, z(x));
                       z(x)      z(x)  d
(%o3)          [y(x) %e    , - %e     (-- (y(x)))]
                                       dx
(%i4) a2: antidiff (expr, x, z(x));
                            /
                     z(x)   [   z(x)  d
(%o4)         y(x) %e     - I %e     (-- (y(x))) dx
                            ]         dx
                            /
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5)                           0
(%i6) antid (expr, x, y(x));
                             z(x)  d
(%o6)             [0, y(x) %e     (-- (z(x)))]
                                   dx
(%i7) antidiff (expr, x, y(x));
                  /
                  [        z(x)  d
(%o7)             I y(x) %e     (-- (z(x))) dx
                  ]              dx
                  /
```

antidiff (*expr, x, u(x)*) Function

Returns an antiderivative of *expr* with respect to *x*. The expression *expr* may contain an unknown function *u* and its derivatives.

When `antidiff` succeeds entirely, the resulting expression is free of integral signs (that is, free of the `integrate` noun). Otherwise, `antidiff` returns an expression which is partly or entirely within an integral sign. If `antidiff` cannot make any progress, the return value is entirely within an integral sign.

`load ("antid")` loads this function. The `antid` package also defines the functions `nonzeroandfreeof` and `linear`.

`antidiff` is related to `antid` as follows. Let *L*, a list of two elements, be the return value of `antid`. Then the return value of `antidiff` is equal to *L*`[1] + 'integrate (`*L*`[2], `*x*`)` where *x* is the variable of integration.

Examples:

```
(%i1) load ("antid")$
(%i2) expr: exp (z(x)) * diff (z(x), x) * y(x);
                                z(x)  d
(%o2)                     y(x) %e     (-- (z(x)))
                                       dx
(%i3) a1: antid (expr, x, z(x));
                           z(x)      z(x)  d
(%o3)                [y(x) %e    , - %e     (-- (y(x)))]
                                            dx
(%i4) a2: antidiff (expr, x, z(x));
                                /
                         z(x)   [   z(x)  d
(%o4)              y(x) %e     - I %e     (-- (y(x))) dx
                                ]         dx
                                /
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5)                               0
(%i6) antid (expr, x, y(x));
                                 z(x)  d
(%o6)                  [0, y(x) %e     (-- (z(x)))]
                                        dx
(%i7) antidiff (expr, x, y(x));
                      /
                      [        z(x)  d
(%o7)                 I y(x) %e     (-- (z(x))) dx
                      ]              dx
                      /
```

atomgrad Property

`atomgrad` is the atomic gradient property of an expression. This property is assigned by `gradef`.

atvalue (*expr*, [*x_1* = *a_1*, ..., *x_m* = *a_m*], *c*) Function

atvalue (*expr*, *x_1* = *a_1*, *c*) Function

Assigns the value *c* to *expr* at the point `x = a`. Typically boundary values are established by this mechanism.

expr is a function evaluation, `f (x_1, ..., x_m)`, or a derivative, `diff (f (x_1, ..., x_m), x_1, n_1, ..., x_n, n_m)` in which the function arguments explicitly appear. *n_i* is the order of differentiation with respect to *x_i*.

The point at which the atvalue is established is given by the list of equations `[x_1 = a_1, ..., x_m = a_m]`. If there is a single variable *x_1*, the sole equation may be given without enclosing it in a list.

`printprops ([f_1, f_2, ...], atvalue)` displays the atvalues of the functions `f_1, f_2, ...` as specified by calls to `atvalue`. `printprops (f, atvalue)` displays the atvalues of one function *f*. `printprops (all, atvalue)` displays the atvalues of all functions for which atvalues are defined.

The symbols `@1`, `@2`, ... represent the variables *x_1*, *x_2*, ... when atvalues are displayed.

`atvalue` evaluates its arguments. `atvalue` returns *c*, the atvalue.

Examples:

```
(%i1) atvalue (f(x,y), [x = 0, y = 1], a^2);
                                    2
(%o1)                              a
(%i2) atvalue ('diff (f(x,y), x), x = 0, 1 + y);
(%o2)                            @2 + 1
(%i3) printprops (all, atvalue);
                                    !
                  d                 !
                 --- (f(@1, @2))!       = @2 + 1
                 d@1                !
                                    !@1 = 0

                                        2
                           f(0, 1) = a

(%o3)                             done
(%i4) diff (4*f(x,y)^2 - u(x,y)^2, x);
                  d                           d
(%o4)  8 f(x, y) (-- (f(x, y))) - 2 u(x, y) (-- (u(x, y)))
                  dx                          dx
(%i5) at (%, [x = 0, y = 1]);
                                                  !
              2               d                   !
(%o5)     16 a  - 2 u(0, 1) (-- (u(x, y))!              )
                              dx                  !
                                                  !x = 0, y = 1
```

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 `cartan` package implements the functions `ext_diff` and `lie_diff`, along with the operators `~` (wedge product) and `|` (contraction of a form with a vector.) Type `demo (tensor)` to see a brief description of these commands along with examples.

`cartan` was implemented by F.B. Estabrook and H.D. Wahlquist.

del (*x*) Function

`del (x)` represents the differential of the variable x.

`diff` returns an expression containing `del` if an independent variable is not specified. In this case, the return value is the so-called "total differential".

Examples:

```
(%i1) diff (log (x));
                                 del(x)
(%o1)                            ------
                                   x
(%i2) diff (exp (x*y));
                     x y              x y
(%o2)            x %e    del(y) + y %e    del(x)
(%i3) diff (x*y*z);
(%o3)         x y del(z) + x z del(y) + y z del(x)
```

delta (*t*) Function

The Dirac Delta function.

Currently only `laplace` knows about the `delta` function.

Example:

```
(%i1) laplace (delta (t - a) * sin(b*t), t, s);
Is  a  positive, negative, or zero?

p;
                                       - a s
(%o1)                     sin(a b) %e
```

dependencies System variable

Default value: `[]`

`dependencies` is the list of atoms which have functional dependencies, assigned by `depends` or `gradef`. The `dependencies` list is cumulative: each call to `depends` or `gradef` appends additional items.

See `depends` and `gradef`.

depends (*f_1, x_1, ..., f_n, x_n*) Function

Declares functional dependencies among variables for the purpose of computing derivatives. In the absence of declared dependence, `diff (f, x)` yields zero. If `depends (f, x)` is declared, `diff (f, x)` yields a symbolic derivative (that is, a `diff` noun).

Each argument *f_1*, *x_1*, etc., can be the name of a variable or array, or a list of names. Every element of *f_i* (perhaps just a single element) is declared to depend on every element of *x_i* (perhaps just a single element). If some *f_i* is the name of an array or contains the name of an array, all elements of the array depend on *x_i*.

`diff` recognizes indirect dependencies established by `depends` and applies the chain rule in these cases.

`remove (`*f*`, dependency)` removes all dependencies declared for *f*.

`depends` returns a list of the dependencies established. The dependencies are appended to the global variable `dependencies`. `depends` evaluates its arguments.

`diff` is the only Maxima command which recognizes dependencies established by `depends`. Other functions (`integrate`, `laplace`, etc.) only recognize dependencies explicitly represented by their arguments. For example, `integrate` does not recognize the dependence of `f` on `x` unless explicitly represented as `integrate (f(x), x)`.

```
(%i1) depends ([f, g], x);
(%o1)                       [f(x), g(x)]
(%i2) depends ([r, s], [u, v, w]);
(%o2)               [r(u, v, w), s(u, v, w)]
(%i3) depends (u, t);
(%o3)                          [u(t)]
(%i4) dependencies;
(%o4)      [f(x), g(x), r(u, v, w), s(u, v, w), u(t)]
(%i5) diff (r.s, u);
                         dr           ds
(%o5)                    -- . s + r . --
                         du           du
(%i6) diff (r.s, t);
                      dr du           ds du
(%o6)                 -- -- . s + r . -- --
                      du dt           du dt
(%i7) remove (r, dependency);
(%o7)                           done
(%i8) diff (r.s, t);
                                ds du
(%o8)                       r . -- --
                                du dt
```

derivabbrev Option variable

Default value: `false`

When `derivabbrev` is `true`, symbolic derivatives (that is, `diff` nouns) are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation `dy/dx`.

derivdegree (*expr*, *y*, *x*) Function

Returns the highest degree of the derivative of the dependent variable *y* with respect to the independent variable *x* occuring in *expr*.

Example:

```
(%i1) 'diff (y, x, 2) + 'diff (y, z, 3) + 'diff (y, x) * x^2;
                         3     2
                        d y   d y    2 dy
(%o1)                   --- + --- + x  --
                          3     2      dx
                        dz    dx
(%i2) derivdegree (%, y, x);
(%o2)                         2
```

derivlist (*var_1, ..., var_k*) Function

Causes only differentiations with respect to the indicated variables, within the `ev` command.

derivsubst Option variable

Default value: `false`

When `derivsubst` is `true`, a non-syntactic substitution such as `subst (x, 'diff (y, t), 'diff (y, t, 2))` yields `'diff (x, t)`.

diff (*expr, x_1, n_1, ..., x_m, n_m*) Function
diff (*expr, x, n*) Function
diff (*expr, x*) Function
diff (*expr*) Function

Returns the derivative or differential of *expr* with respect to some or all variables in *expr*.

`diff (`*expr*`, `*x*`, `*n*`)` returns the *n*'th derivative of *expr* with respect to *x*.

`diff (`*expr*`, `*x_1*`, `*n_1*`, ..., `*x_m*`, `*n_m*`)` returns the mixed partial derivative of *expr* with respect to *x_1*, ..., *x_m*. It is equivalent to `diff (... (diff (`*expr*`, `*x_m*`, `*n_m*`) ...), `*x_1*`, `*n_1*`)`.

`diff (`*expr*`, `*x*`)` returns the first derivative of *expr* with respect to the variable *x*.

`diff (`*expr*`)` returns the total differential of *expr*, that is, the sum of the derivatives of *expr* with respect to each its variables times the differential `del` of each variable. No further simplification of `del` is offered.

The noun form of `diff` is required in some contexts, such as stating a differential equation. In these cases, `diff` may be quoted (as `'diff`) to yield the noun form instead of carrying out the differentiation.

When `derivabbrev` is `true`, derivatives are displayed as subscripts. Otherwise, derivatives are displayed in the Leibniz notation, `dy/dx`.

Examples:

```
(%i1) diff (exp (f(x)), x, 2);
                     2
              f(x)  d               f(x)  d         2
(%o1)       %e     (--- (f(x))) + %e     (-- (f(x)))
                      2                   dx
                    dx
```

```
(%i2) derivabbrev: true$
(%i3) 'integrate (f(x, y), y, g(x), h(x));
                         h(x)
                        /
                        [
(%o3)                   I     f(x, y) dy
                        ]
                        /
                         g(x)
(%i4) diff (%, x);
       h(x)
      /
      [
(%o4) I     f(x, y)  dy + f(x, h(x)) h(x)  - f(x, g(x)) g(x)
      ]            x                     x                  x
      /
       g(x)
```

For the tensor package, the following modifications have been incorporated:

(1) The derivatives of any indexed objects in *expr* will have the variables *x_i* appended as additional arguments. Then all the derivative indices will be sorted.

(2) The *x_i* 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 with respect to the *x_i*'th 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 *x_i* 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.

diff Special symbol

When `diff` is present as an `evflag` in call to `ev`, all differentiations indicated in `expr` are carried out.

dscalar (*f*) Function

Applies the scalar d'Alembertian to the scalar function *f*.

`load ("ctensor")` loads this function.

express (*expr*) Function

Expands differential operator nouns into expressions in terms of partial derivatives. `express` recognizes the operators `grad`, `div`, `curl`, `laplacian`. `express` also expands the cross product `~`.

Symbolic derivatives (that is, `diff` nouns) in the return value of express may be evaluated by including `diff` in the `ev` function call or command line. In this context, `diff` acts as an `evfun`.

`load ("vect")` loads this function.

Examples:

```
(%i1) load ("vect")$
(%i2) grad (x^2 + y^2 + z^2);
                                   2    2    2
(%o2)                  grad (z  + y  + x )
(%i3) express (%);
       d    2    2    2   d    2    2    2   d    2    2    2
(%o3) [-- (z  + y  + x ), -- (z  + y  + x ), -- (z  + y  + x )]
       dx                 dy                 dz
(%i4) ev (%, diff);
(%o4)                    [2 x, 2 y, 2 z]
(%i5) div ([x^2, y^2, z^2]);
                              2   2   2
(%o5)                   div [x , y , z ]
(%i6) express (%);
                   d    2    d    2    d    2
(%o6)              -- (z ) + -- (y ) + -- (x )
                   dz        dy        dx
(%i7) ev (%, diff);
(%o7)                    2 z + 2 y + 2 x
(%i8) curl ([x^2, y^2, z^2]);
                               2   2   2
(%o8)                   curl [x , y , z ]
(%i9) express (%);
       d    2    d    2   d    2    d    2   d    2    d    2
(%o9) [-- (z ) - -- (y ), -- (x ) - -- (z ), -- (y ) - -- (x )]
       dy        dz       dz        dx       dx        dy
(%i10) ev (%, diff);
(%o10)                      [0, 0, 0]
(%i11) laplacian (x^2 * y^2 * z^2);
                                  2  2  2
(%o11)                laplacian (x  y  z )
(%i12) express (%);
         2                 2                 2
        d     2  2  2     d     2  2  2     d     2  2  2
(%o12)  --- (x  y  z ) + --- (x  y  z ) + --- (x  y  z )
          2                 2                 2
        dz                dy                dx
(%i13) ev (%, diff);
                      2  2      2  2      2  2
(%o13)             2 y  z  + 2 x  z  + 2 x  y
(%i14) [a, b, c] ~ [x, y, z];
(%o14)                [a, b, c] ~ [x, y, z]
(%i15) express (%);
(%o15)          [b z - c y, c x - a z, a y - b x]
```

gradef (*f*(*x_1*, ..., *x_n*), *g_1*, ..., *g_m*) Function

gradef (*a*, *x*, *expr*) Function

Defines the partial derivatives (i.e., the components of the gradient) of the function *f* or variable *a*.

`gradef (`*f*`(`*x_1*`, ..., `*x_n*`), `*g_1*`, ..., `*g_m*`)` defines `d`*f*`/dx`*_i* as *g_i*, where *g_i* is an expression; *g_i* may be a function call, but not the name of a function. The number of partial derivatives *m* may be less than the number of arguments *n*, in which case derivatives are defined with respect to *x_1* through *x_m* only.

`gradef (`*a*`, `*x*`, `*expr*`)` defines the derivative of variable *a* with respect to *x* as *expr*. This also establishes the dependence of *a* on *x* (via `depends (`*a*`, `*x*`)`).

The first argument *f*`(`*x_1*`, ..., `*x_n*`)` or *a* is quoted, but the remaining arguments *g_1*, ..., *g_m* are evaluated. `gradef` returns the function or variable for which the partial derivatives are defined.

`gradef` can redefine the derivatives of Maxima's built-in functions. For example, `gradef (sin(x), sqrt (1 - sin(x)^2))` redefines the derivative of `sin`.

`gradef` cannot define partial derivatives for a subscripted function.

`printprops ([`*f_1*`, ..., `*f_n*`], gradef)` displays the partial derivatives of the functions *f_1*, ..., *f_n*, as defined by `gradef`.

`printprops ([`*a_n*`, ..., `*a_n*`], atomgrad)` displays the partial derivatives of the variables *a_n*, ..., *a_n*, as defined by `gradef`.

`gradefs` is the list of the functions for which partial derivatives have been defined by `gradef`. `gradefs` does not include any variables for which partial derivatives have been defined by `gradef`.

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.

gradefs System variable

Default value: `[]`

`gradefs` is the list of the functions for which partial derivatives have been defined by `gradef`. `gradefs` does not include any variables for which partial derivatives have been defined by `gradef`.

laplace (*expr*, *t*, *s*) Function

Attempts to compute the Laplace transform of *expr* with respect to the variable *t* and transform parameter *s*. If `laplace` cannot find a solution, a noun `'laplace` is returned.

`laplace` recognizes in *expr* the functions `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh`, and `erf`, as well as `derivative`, `integrate`, `sum`, and `ilt`. If some other functions are present, `laplace` may not be able to compute the transform.

expr may also be a linear, constant coefficient differential equation in which case `atvalue` of the dependent variable is used. The required atvalue may be supplied either before or after the transform is computed. 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.

`laplace` recognizes convolution integrals of the form `integrate (f(x) * g(t - x), x, 0, t)`; other kinds of convolutions are not recognized.

Functional relations must be explicitly represented in *expr*; implicit relations, established by `depends`, are not recognized. That is, if *f* depends on *x* and *y*, `f (x, y)` must appear in *expr*.

See also `ilt`, the inverse Laplace transform.

Examples:

```
(%i1) laplace (exp (2*t + a) * sin(t) * t, t, s);
                            a
                          %e  (2 s - 4)
(%o1)                    ---------------
                           2           2
                         (s  - 4 s + 5)
(%i2) laplace ('diff (f (x), x), x, s);
(%o2)             s laplace(f(x), x, s) - f(0)
(%i3) diff (diff (delta (t), t), t);
                          2
                         d
(%o3)                    --- (delta(t))
                           2
                         dt
(%i4) laplace (%, t, s);
                            !
               d            !         2
(%o4)        - -- (delta(t))!      + s  - delta(0) s
               dt           !
                            !t = 0
```

21 Integration

21.1 Introduction to Integration

Maxima has several routines for handling integration. The `integrate` function 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; an adaptave integrator which uses the Newton-Cotes 8 panel quadrature rule, called `quanc8`; and a set of adaptive integrators from Quadpack, named `quad_qag`, `quad_qags`, etc. Hypergeometric functions are being worked on, see `specint` for details. Generally speaking, Maxima 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)`.

21.2 Definitions for Integration

changevar (*expr, f(x,y), y, x*) Function

Makes the change of variable given by *f(x,y)* `= 0` in all integrals occurring in *expr* with integration with respect to *x*. The new variable is *y*.

```
(%i1) assume(a > 0)$
(%i2) 'integrate (%e**sqrt(a*y), y, 0, 4);
                      4
                     /
                     [    sqrt(a) sqrt(y)
(%o2)                I  %e                dy
                     ]
                     /
                      0
(%i3) changevar (%, y-z^2/a, z, y);
                      0
                     /
                     [                abs(z)
                   2 I            z %e       dz
                     ]
                     /
                      - 2 sqrt(a)
(%o3)            - ----------------------------
                                  a
```

An expression containing a noun form, such as the instances of `'integrate` above, may be evaluated by `ev` with the `nouns` flag. For example, the expression returned by `changevar` above may be evaluated by `ev (%o3, nouns)`.

`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+ ...`, not a higher degree function. E.g.,

```
(%i4) sum (a[i]*x^(i-2), i, 0, inf);
                         inf
                         ====
                         \         i - 2
(%o4)                     >    a  x
                         /      i
                         ====
                         i = 0
(%i5) changevar (%, i-2-n, n, i);
                        inf
                        ====
                        \               n
(%o5)                    >      a      x
                        /        n + 2
                        ====
                        n = - 2
```

dblint (*f*, *r*, *s*, *a*, *b*) Function

A double-integral routine which was written in top-level Maxima 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

```
/b /s(x)
|  |
|  |    f(x,y) dy dx
|  |
/a /r(x)
```

The function *f* must be a translated or compiled function of two variables, and *r* and *s* 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!

`demo (dblint)` executes a demonstration of `dblint` applied to an example problem.

defint (*expr, x, a, b*) Function

Attempts to compute a definite integral. `defint` is called by `integrate` when limits of integration are specified, i.e., when `integrate` is called as `integrate (`*expr*`,` *x*`,` *a*`,` *b*`)`. Thus from the user's point of view, it is sufficient to call `integrate`.

`defint` returns a symbolic expression, either the computed integral or the noun form of the integral. See `quad_qag` and related functions for numerical approximation of definite integrals.

erf (*x*) Function

Represents the error function, whose derivative is: `2*exp(-x^2)/sqrt(%pi)`.

erfflag Option variable

Default value: `true`

When `erfflag` is `false`, prevents `risch` from introducing the `erf` function in the answer if there were none in the integrand to begin with.

ilt (*expr, t, s*) Function

Computes the inverse Laplace transform of *expr* with respect to *t* and parameter *s*. *expr* 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.

```
(%i1) 'integrate (sinh(a*x)*f(t-x), x, 0, t) + b*f(t) = t**2;
              t
             /
             [                                    2
(%o1)        I  f(t - x) sinh(a x) dx + b f(t) = t
             ]
             /
              0
(%i2) laplace (%, t, s);
                               a laplace(f(t), t, s)   2
(%o2)  b laplace(f(t), t, s) + --------------------- = --
                                       2    2           3
                                      s  - a           s
(%i3) linsolve ([%], ['laplace(f(t), t, s)]);
                                        2      2
                                     2 s  - 2 a
(%o3)     [laplace(f(t), t, s) = --------------------]
                                    5         2     3
                                 b s  + (a - a  b) s
(%i4) ilt (rhs (first (%)), s, t);
Is  a b (a b - 1)  positive, negative, or zero?

pos;
```

```
                   sqrt(a b (a b - 1)) t
            2 cosh(---------------------)         2
                             b                  a t
(%o4) - ----------------------------- + -------
                 3  2      2                a b - 1
                a  b  - 2 a  b + a

                                                      2
                                         + ------------------
                                              3  2      2
                                             a  b  - 2 a  b + a
```

integrate (*expr, x*) Function

integrate (*expr, x, a, b*) Function

Attempts to symbolically compute the integral of *expr* with respect to *x*. `integrate (expr, x)` is an indefinite integral, while `integrate (expr, x, a, b)` is a definite integral, with limits of integration *a* and *b*. The limits should not contain *x*, although `integrate` does not enforce this restriction. *a* need not be less than *b*. If *b* is equal to *a*, `integrate` returns zero.

See `quad_qag` and related functions for numerical approximation of definite integrals. See `residue` for computation of residues (complex integration). See `antid` for an alternative means of computing indefinite integrals.

The integral (an expression free of `integrate`) is returned if `integrate` succeeds. Otherwise the return value is the noun form of the integral (the quoted operator `'integrate`) or an expression containing one or more noun forms. The noun form of `integrate` is displayed with an integral sign.

In some circumstances it is useful to construct a noun form by hand, by quoting `integrate` with a single quote, e.g., `'integrate (expr, x)`. For example, the integral may depend on some parameters which are not yet computed. The noun may be applied to its arguments by `ev (i, nouns)` where *i* is the noun form of interest.

`integrate` handles definite integrals separately from indefinite, and employs a range of heuristics to handle each case. Special cases of definite integrals include limits of integration equal to zero or infinity (`inf` or `minf`), trigonometric functions with limits of integration equal to zero and `%pi` or `2 %pi`, rational functions, integrals related to the definitions of the `beta` and `psi` functions, and some logarithmic and trigonometric integrals. Processing rational functions may include computation of residues. If an applicable special case is not found, an attempt will be made to compute the indefinite integral and evaluate it at the limits of integration. This may include taking a limit as a limit of integration goes to infinity or negative infinity; see also `ldefint`.

Special cases of indefinite integrals include trigonometric functions, exponential and logarithmic functions, and rational functions. `integrate` may also make use of a short table of elementary integrals.

`integrate` may carry out a change of variable if the integrand has the form `f(g(x)) * diff(g(x), x)`. `integrate` attempts to find a subexpression `g(x)` such that the derivative of `g(x)` divides the integrand. This search may make use of derivatives defined by the `gradef` function. See also `changevar` and `antid`.

If none of the preceding heuristics find the indefinite integral, the Risch algorithm is executed. The flag `risch` may be set as an `evflag`, in a call to `ev` or on the command line, e.g., `ev (integrate (`*expr*`, `*x*`), risch)` or `integrate (`*expr*`, `*x*`), risch`. If `risch` is present, `integrate` calls the `risch` function without attempting heuristics first. See also `risch`.

`integrate` works only with functional relations represented explicitly with the `f(x)` notation. `integrate` does not respect implicit dependencies established by the `depends` function.

`integrate` may need to know some property of a parameter in the integrand. `integrate` will first consult the `assume` database, and, if the variable of interest is not there, `integrate` will ask the user. Depending on the question, suitable responses are `yes;` or `no;`, or `pos;`, `zero;`, or `neg;`.

`integrate` is not, by default, declared to be linear. See `declare` and `linear`.

`integrate` attempts integration by parts only in a few special cases.

Examples:

- Elementary indefinite and definite integrals.

```
(%i1) integrate (sin(x)^3, x);
                           3
                        cos (x)
(%o1)                   ------- - cos(x)
                           3
(%i2) integrate (x/ sqrt (b^2 - x^2), x);
                                 2    2
(%o2)                    - sqrt(b  - x )
(%i3) integrate (cos(x)^2 * exp(x), x, 0, %pi);
                               %pi
                           3 %e      3
(%o3)                      ------- - -
                              5      5
(%i4) integrate (x^2 * exp(-x^2), x, minf, inf);
                            sqrt(%pi)
(%o4)                       ---------
                                2
```

- Use of `assume` and interactive query.

```
(%i1) assume (a > 1)$
(%i2) integrate (x**a/(x+1)**(5/2), x, 0, inf);
    2 a + 2
Is  -------  an integer?
       5

no;
Is  2 a - 3  positive, negative, or zero?

neg;
                                   3
(%o2)                  beta(a + 1, - - a)
                                   2
```

- Change of variable. There are two changes of variable in this example: one using a derivative established by `gradef`, and one using the derivation `diff(r(x))` of an unspecified function `r(x)`.

```
(%i3) gradef (q(x), sin(x**2));
(%o3)                          q(x)
(%i4) diff (log (q (r (x))), x);
                      d               2
                     (-- (r(x))) sin(r (x))
                      dx
(%o4)                ----------------------
                            q(r(x))
(%i5) integrate (%, x);
(%o5)                      log(q(r(x)))
```

- Return value contains the `'integrate` noun form. In this example, Maxima can extract one factor of the denominator of a rational function, but cannot factor the remainder or otherwise find its integral. `grind` shows the noun form `'integrate` in the result. See also `integrate_use_rootsof` for more on integrals of rational functions.

```
(%i1) expand ((x-4) * (x^3+2*x+1));
                    4      3      2
(%o1)              x  - 4 x  + 2 x  - 7 x - 4
(%i2) integrate (1/%, x);
                              /  2
                              [ x  + 4 x + 18
                              I ------------- dx
                              ]  3
                 log(x - 4)   / x  + 2 x + 1
(%o2)            ---------- - ------------------
                     73               73
(%i3) grind (%);
log(x-4)/73-('integrate((x^2+4*x+18)/(x^3+2*x+1),x))/73$
```

- Defining a function in terms of an integral. The body of a function is not evaluated when the function is defined. Thus the body of `f_1` in this example contains the noun form of `integrate`. The double-single-quotes operator `''` causes the integral to be evaluated, and the result becomes the body of `f_2`.

```
(%i1) f_1 (a) := integrate (x^3, x, 1, a);
                                     3
(%o1)           f_1(a) := integrate(x , x, 1, a)
(%i2) ev (f_1 (7), nouns);
(%o2)                          600
(%i3) /* Note parentheses around integrate(...) here */
      f_2 (a) := ''(integrate (x^3, x, 1, a));
                                   4
                                  a    1
(%o3)                   f_2(a) := -- - -
                                  4    4
(%i4) f_2 (7);
(%o4)                          600
```

integration_constant_counter System variable

Default value: 0

`integration_constant_counter` is a counter which is updated each time a constant of integration (named by Maxima, e.g., `integrationconstant1`) is introduced into an expression by indefinite integration of an equation.

integrate_use_rootsof Option variable

Default value: `false`

When `integrate_use_rootsof` is `true` and the denominator of a rational function cannot be factored, `integrate` returns the integral in a form which is a sum over the roots (not yet known) of the denominator.

For example, with `integrate_use_rootsof` set to `false`, `integrate` returns an unsolved integral of a rational function in noun form:

```
(%i1) integrate_use_rootsof: false$
(%i2) integrate (1/(1+x+x^5), x);
        /  2
        [ x  - 4 x + 5
        I ------------ dx                            2 x + 1
        ]  3    2                2              5 atan(-------)
        / x  - x  + 1       log(x  + x + 1)          sqrt(3)
(%o2)   ----------------- - --------------- + ---------------
                7                 14             7 sqrt(3)
```

Now we set the flag to be true and the unsolved part of the integral will be expressed as a summation over the roots of the denominator of the rational function:

```
(%i3) integrate_use_rootsof: true$
(%i4) integrate (1/(1+x+x^5), x);
      ====        2
      \       (%r4  - 4 %r4 + 5) log(x - %r4)
       >      -------------------------------
      /                    2
      ====            3 %r4  - 2 %r4
                      3    2
      %r4 in rootsof(x  - x  + 1)
(%o4) ----------------------------------------------------------
               7

                                                             2 x + 1
                                         2              5 atan(-------)
                                    log(x  + x + 1)          sqrt(3)
                                  - --------------- + ---------------
                                          14             7 sqrt(3)
```

Alternatively the user may compute the roots of the denominator separately, and then express the integrand in terms of these roots, e.g., `1/((x - a)*(x - b)*(x - c))` or `1/((x^2 - (a+b)*x + a*b)*(x - c))` if the denominator is a cubic polynomial. Sometimes this will help Maxima obtain a more useful result.

ldefint (*expr, x, a, b*) Function

Attempts to compute the definite integral of *expr* by using `limit` to evaluate the indefinite integral of *expr* with respect to *x* at the upper limit *b* and at the lower limit *a*. If it fails to compute the definite integral, `ldefint` returns an expression containing limits as noun forms.

`ldefint` is not called from `integrate`, so executing `ldefint (`*expr*`, `*x*`, `*a*`, `*b*`)` may yield a different result than `integrate (`*expr*`, `*x*`, `*a*`, `*b*`)`. `ldefint` always uses the same method to evaluate the definite integral, while `integrate` may employ various heuristics and may recognize some special cases.

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 package `qq` (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, 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_relerr` (default value=1.0e-4) and `quanc8_abserr` (default value=1.0e-8) which give the relative error test:

```
|integral(function) - computed value| < quanc8_relerr*|integral(function)|
```

and the absolute error test:

```
|integral(function) - computed value| < quanc8_abserr
```

`printfile ("qq.usg")` yields additional information.

quanc8 (*expr, a, b*) Function

An adaptive integrator. Demonstration 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")`. See also `qq`.

residue (*expr, z, z_0*) Function

Computes the residue in the complex plane of the expression *expr* when the variable *z* assumes the value *z_0*. The residue is the coefficient of `(`*z*` - `*z_0*`)^(-1)` in the Laurent series for *expr*.

```
(%i1) residue (s/(s**2+a**2), s, a*%i);
                                1
(%o1)                           -
                                2
(%i2) residue (sin(a*x)/x**4, x, 0);
                                 3
                                a
(%o2)                         - --
                                6
```

risch (*expr, x*) Function

Integrates *expr* with respect to *x* 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`, if `false`, prevents `risch` from introducing the `erf` function in the answer if there were none in the integrand to begin with.

```
(%i1) risch (x^2*erf(x), x);
                                                        2
             3                      2                - x
        %pi x  erf(x) + (sqrt(%pi) x  + sqrt(%pi)) %e
(%o1)   -------------------------------------------------
                              3 %pi
(%i2) diff(%, x), ratsimp;
                             2
(%o2)                       x  erf(x)
```

romberg (*expr, x, a, b*) Function

romberg (*expr, a, b*) Function

Romberg integration. 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:

```
(%i1) showtime: true$
(%i2) romberg (sin(y), y, 0, %pi);
Evaluation took 0.00 seconds (0.01 elapsed) using 25.293 KB.
(%o2)                   2.000000016288042
(%i3) 1/((x-1)^2+1/100) + 1/((x-2)^2+1/1000) + 1/((x-3)^2+1/200)$
(%i4) f(x) := ''%$
(%i5) rombergtol: 1e-6$
(%i6) rombergit: 15$
(%i7) romberg (f(x), x, -5, 5);
Evaluation took 11.97 seconds (12.21 elapsed) using 12.423 MB.
(%o7)                   173.6730736617464
```

The second is an efficient way that is used as follows:

```
romberg (<function name>, <lower limit>, <upper limit>);
```

Continuing the above example, we have:

```
(%i8) f(x) := (mode_declare ([function(f), x], float), ''(%th(5)))$
(%i9) translate(f);
(%o9)                              [f]
(%i10) romberg (f, -5, 5);
Evaluation took 3.51 seconds (3.86 elapsed) using 6.641 MB.
(%o10)                     173.6730736617464
```

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`.

`romberg` may be called recursively and thus can do double and triple integrals.

Example:

```
(%i1) assume (x > 0)$
(%i2) integrate (integrate (x*y/(x+y), y, 0, x/2), x, 1, 3)$
(%i3) radcan (%);
                    26 log(3) - 26 log(2) - 13
(%o3)             - --------------------------
                                3
(%i4) %,numer;
(%o4)                      .8193023963959073
(%i5) define_variable (x, 0.0, float, "Global variable in function F")$
(%i6) f(y) := (mode_declare (y, float), x*y/(x+y))$
(%i7) g(x) := romberg ('f, 0, x/2)$
(%i8) romberg (g, 1, 3);
(%o8)                      .8193022864324522
```

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:

```
(%i1) g_1(x) := (mode_declare (x, float), romberg (x*y/(x+y), y, 0, x/2))$
(%i2) romberg (g_1, 1, 3);
(%o2)                      .8193022864324522
```

The advantage here is shortness.

```
(%i3) q (a, b) := romberg (romberg (x*y/(x+y), y, 0, x/2), x, a, b)$
(%i4) q (1, 3);
(%o4)                      .8193022864324522
```

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.

rombergabs Option variable

Default value: 0.0

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)

```
(abs(y[n]-y[n-1]) <= rombergabs or
abs(y[n]-y[n-1])/(if y[n]=0.0 then 1.0 else y[n]) <= 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

```
'integrate (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. First of all try doing the whole integral at once.

```
(%i1) f(x) := (mode_declare (n, integer, x, float), n:n+1, exp(-x))$
(%i2) translate(f)$
Warning-> n is an undefined global variable.
(%i3) block ([rombergtol: 1.e-6, romberabs: 0.0], n:0, romberg (f, 0, 50));
(%o3)                   1.000000000488271
(%i4) n;
(%o4)                           257
```

That approach required 257 function evaluations. Now do the integral intelligently, by first doing `'integrate (exp(-x), x, 0, 10)` and then setting `rombergabs` to 1.E-6 times (this partial integral). This approach takes only 130 function evaluations.

```
(%i5) block ([rombergtol: 1.e-6, rombergabs:0.0, sum:0.0],
  n: 0, sum: romberg (f, 0, 10), rombergabs: sum*rombergtol, rombergtol:0.0,
      sum + romberg (f, 10, 50));
(%o5)                   1.000000001234793
(%i6) n;
(%o6)                           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 Option variable

Default value: 11

The accuracy of the `romberg` integration command is governed by the global variables `rombergtol` and `rombergit`. `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 Option variable

Default value: 0

`rombergmin` 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 Option variable

Default value: 1e-4

The accuracy of the `romberg` integration command is governed by the global variables `rombergtol` and `rombergit`. `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 (*expr, x, a, b*) Function

Equivalent to `ldefint` with `tlimswitch` set to `true`.

quad_qag (*f(x), x, a, b, key, epsrel, limit*) Function

Numerically evaluate the integral

$$\int_a^b f(x)dx$$

using a simple adaptive integrator.

The function to be integrated is *f(x)*, with dependent variable *x*, and the function is to be integrated between the limits *a* and *b*. *key* is the integrator to be used and should be an integer between 1 and 6, inclusive. The value of *key* selects the order of the Gauss-Kronrod integration rule.

The numerical integration is done adaptively by subdividing the integration region into sub-intervals until the desired accuracy is achieved.

The optional arguments *epsrel* and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qag` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` if no problems were encountered;

`1` if too many sub-intervals were done;

`2` if excessive roundoff error is detected;

`3` if extremely bad integrand behavior occurs;

6 if the input is invalid.

Examples:

```
(%i1) quad_qag (x^(1/2)*log(1/x), x, 0, 1, 3);
(%o1)    [.4444444444492108, 3.1700968502883E-9, 961, 0]
(%i2) integrate (x^(1/2)*log(1/x), x, 0, 1);
                                        4
(%o2)                                   -
                                        9
```

quad_qags (*f(x), x, a, b, epsrel, limit*) Function

Numerically integrate the given function using adaptive quadrature with extrapolation. The function to be integrated is *f(x)*, with dependent variable *x*, and the function is to be integrated between the limits *a* and *b*.

The optional arguments *epsrel* and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qags` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

0 no problems were encountered;

1 too many sub-intervals were done;

2 excessive roundoff error is detected;

3 extremely bad integrand behavior occurs;

4 failed to converge

5 integral is probably divergent or slowly convergent

6 if the input is invalid.

Examples:

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0 ,1);
(%o1)   [.4444444444444448, 1.11022302462516E-15, 315, 0]
```

Note that `quad_qags` is more accurate and efficient than `quad_qag` for this integrand.

quad_qagi (*f(x), x, a, inftype, epsrel, limit*) Function

Numerically evaluate one of the following integrals

$$\int_a^\infty f(x)dx$$

$$\int_\infty^a f(x)dx$$

$$\int_{-\infty}^{\infty} f(x)dx$$

using the Quadpack QAGI routine. The function to be integrated is *f(x)*, with dependent variable *x*, and the function is to be integrated over an infinite range.

The parameter *inftype* determines the integration interval as follows:

`inf` The interval is from *a* to positive infinity.

`minf` The interval is from negative infinity to *a*.

`both` The interval is the entire real line.

The optional arguments *epsrel* and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qagi` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`4` failed to converge

`5` integral is probably divergent or slowly convergent

`6` if the input is invalid.

Examples:

```
(%i1) quad_qagi (x^2*exp(-4*x), x, 0, inf);
(%o1)        [0.03125, 2.95916102995002E-11, 105, 0]
(%i2) integrate (x^2*exp(-4*x), x, 0, inf);
                                  1
(%o2)                             --
                                  32
```

quad_qawc (*f*(*x*), *x*, *c*, *a*, *b*, *epsrel*, *limit*) Function

Numerically compute the Cauchy principal value of

$$\int_{a}^{b} \frac{f(x)}{x-c}\, dx$$

using the Quadpack QAWC routine. The function to be integrated is *f(x)*`/(x - `*c*`)`, with dependent variable *x*, and the function is to be integrated over the interval *a* to *b*.

The optional arguments *epsrel* and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qawc` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qawc (2^(-5)*((x-1)^2+4^(-5))^(-1), x, 2, 0, 5);
(%o1)    [- 3.130120337415925, 1.306830140249558E-8, 495, 0]
(%i2) integrate (2^(-alpha)*(((x-1)^2 + 4^(-alpha))*(x-2))^(-1), x, 0, 5);
Principal Value
                       alpha
        alpha       9 4                 9
       4      log(------------- + -------------)
                      alpha           alpha
                  64 4      + 4   64 4      + 4
(%o2) (-----------------------------------------
                        alpha
                     2 4      + 2

         3 alpha                       3 alpha
         -------                       -------
            2             alpha/2         2          alpha/2
      2 4        atan(4 4       )   2 4        atan(4       )   alpha
    - -------------------------- - -------------------------)/2
                alpha                        alpha
             2 4      + 2                 2 4      + 2
(%i3) ev (%, alpha=5, numer);
(%o3)                    - 3.130120337415917
```

quad_qawf (*f(x), x, a, omega, trig, epsabs, limit, maxp1, limlst*) Function

Numerically compute the a Fourier-type integral using the Quadpack QAWF routine. The integral is

$$\int_a^\infty f(x)w(x)dx$$

The weight function w is selected by *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

The optional arguments are:

epsabs Desired absolute error of approximation. Default is 1d-10.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

maxp1 Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.

limlst Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

epsabs and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qawf` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qawf (exp(-x^2), x, 0, 1, 'cos);
(%o1)   [.6901942235215714, 2.84846300257552E-11, 215, 0]
(%i2) integrate (exp(-x^2)*cos(x), x, 0, inf);
                              - 1/4
                            %e      sqrt(%pi)
(%o2)                       -----------------
                                    2
(%i3) ev (%, numer);
(%o3)                       .6901942235215714
```

quad_qawo (*f(x), x, a, b, omega, trig, epsabs, limit, maxp1, limlst*) Function

Numerically compute the integral using the Quadpack QAWO routine:

$$\int_a^b f(x)w(x)dx$$

The weight function w is selected by *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

The optional arguments are:

epsabs Desired absolute error of approximation. Default is 1d-10.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

maxp1 Maximum number of Chebyshev moments. Must be greater than 0. Default is 100.

limlst Upper bound on the number of cycles. Must be greater than or equal to 3. Default is 10.

epsabs and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qawo` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

`0` no problems were encountered;

`1` too many sub-intervals were done;

`2` excessive roundoff error is detected;

`3` extremely bad integrand behavior occurs;

`6` if the input is invalid.

Examples:

```
(%i1) quad_qawo (x^(-1/2)*exp(-2^(-2)*x), x, 1d-8, 20*2^2, 1, cos);
(%o1)     [1.376043389877692, 4.72710759424899E-11, 765, 0]
(%i2) rectform (integrate (x^(-1/2)*exp(-2^(-alpha)*x) * cos(x), x, 0, inf));
                   alpha/2 - 1/2            2 alpha
        sqrt(%pi) 2              sqrt(sqrt(2        + 1) + 1)
(%o2)   -----------------------------------------------------
                                    2 alpha
                              sqrt(2        + 1)
(%i3) ev (%, alpha=2, numer);
(%o3)                         1.376043390090716
```

quad_qaws (*f(x)*, *x*, *a*, *b*, *alfa*, *beta*, *wfun*, *epsabs*, *limit*) Function

Numerically compute the integral using the Quadpack QAWS routine:

$$\int_a^b f(x)w(x)dx$$

The weight function w is selected by *wfun*:

1 $w(x) = (x-a)^a lfa(b-x)^b eta$

2 $w(x) = (x-a)^a lfa(b-x)^b etalog(x-a)$

3 $w(x) = (x-a)^a lfa(b-x)^b etalog(b-x)$

2 $w(x) = (x-a)^a lfa(b-x)^b etalog(x-a)log(b-x)$

The optional arguments are:

epsabs Desired absolute error of approximation. Default is 1d-10.

limit Size of internal work array. (*limit* - *limlst*)/2 is the maximum number of subintervals to use. Default is 200.

epsabs and *limit* are the desired relative error and the maximum number of subintervals, respectively. *epsrel* defaults to 1e-8 and *limit* is 200.

`quad_qaws` returns a list of four elements:

an approximation to the integral,

the estimated absolute error of the approximation,

the number integrand evaluations,

an error code.

The error code (fourth element of the return value) can have the values:

0 no problems were encountered;

1 too many sub-intervals were done;

2 excessive roundoff error is detected;

3 extremely bad integrand behavior occurs;

6 if the input is invalid.

Examples:

```
(%i1) quad_qaws (1/(x+1+2^(-4)), x, -1, 1, -0.5, -0.5, 1);
(%o1)     [8.750097361672832, 1.24321522715422E-10, 170, 0]
(%i2) integrate ((1-x*x)^(-1/2)/(x+1+2^(-alpha)), x, -1, 1);
        alpha
Is  4 2      - 1  positive, negative, or zero?

pos;
                             alpha         alpha
                      2 %pi 2      sqrt(2 2      + 1)
(%o2)                 -------------------------------
                                    alpha
                                 4 2      + 2
(%i3) ev (%, alpha=4, numer);
(%o3)                        8.750097361672829
```

22 Equations

22.1 Definitions for Equations

%rnum_list System variable

Default value: `[]`

`%rnum_list` is the list of variables introduced in solutions by `algsys`. `%r` variables 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 Option variable

Default value: `false`

`algexact` 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 ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) Function
algsys ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) Function

Solves the simultaneous polynomials *expr_1*, ..., *expr_m* or polynomial equations *eqn_1*, ..., *eqn_m* for the variables *x_1*, ..., *x_n*. An expression *expr* is equivalent to an equation *expr* `= 0`. There may be more equations than variables or vice versa.

`algsys` returns a list of solutions, with each solution given as a list of equations stating values of the variables *x_1*, ..., *x_n* which satisfy the system of equations. If `algsys` cannot find a solution, an empty list `[]` is returned.

The symbols `%r1`, `%r2`, ..., are introduced as needed to represent arbitrary parameters in the solution; these variables are also appended to the list `%rnum_list`.

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 *x* are selected. The variable is chosen to have lowest nonzero degree. Then the resultant of *E* and *E_j* with respect to *x* 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, as *x* 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.

In some cases, `solve` is not be able to find 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 biquadratic, 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 biquadratic, then if the switch `realonly` is `true`, the function `realroots` is called to find the real-valued solutions. If `realonly` is `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 `algepsilon` to a higher value.

If `algexact` is set to `true`, `solve` will always be called.

(4) Finally, the solutions obtained in step (3) are substituted into previous levels and the solution process returns to (1).

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, it may be possible to isolate parts of the result with `pickapart` or `reveal`.

Occasionally, `radcan` may introduce an imaginary unit `%i` into a solution which is actually real-valued.

Examples:

```
(%i1) e1: 2*x*(1 - a1) - 2*(x - 1)*a2;
(%o1)             2 (1 - a1) x - 2 a2 (x - 1)
(%i2) e2: a2 - a1;
(%o2)                        a2 - a1
(%i3) e3: a1*(-y - x^2 + 1);
                                   2
(%o3)                  a1 (- y - x  + 1)
(%i4) e4: a2*(y - (x - 1)^2);
                                       2
(%o4)                  a2 (y - (x - 1) )
(%i5) algsys ([e1, e2, e3, e4], [x, y, a1, a2]);
(%o5) [[x = 0, y = %r1, a1 = 0, a2 = 0],

                           [x = 1, y = 0, a1 = 1, a2 = 1]]
(%i6) e1: x^2 - y^2;
                              2    2
(%o6)                        x  - y
(%i7) e2: -1 - y + 2*y^2 - x + x^2;
                         2        2
(%o7)                 2 y  - y + x  - x - 1
(%i8) algsys ([e1, e2], [x, y]);
                 1            1
(%o8) [[x = - -------, y = -------],
              sqrt(3)      sqrt(3)
```

```
           1              1             1        1
[x = -------, y = - -------], [x = - -, y = - -], [x = 1, y = 1]]
     sqrt(3)        sqrt(3)             3        3
```

allroots (*expr*) Function

allroots (*eqn*) Function

Computes numerical approximations of the real and complex roots of the polynomial *expr* or polynomial equation *eqn* of one variable.

The flag `polyfactor` 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 the polynomial is real, `allroots (%i*`*p*`)` may yield more accurate approximations than `allroots (`*p*`)`, as `allroots` invokes a different algorithm in that case.

`allroots` rejects non-polynomials. It requires that the numerator after `rat`'ing 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`.

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, *ACM TOMS*, vol. 1, (1975), p.178).

Examples:

```
(%i1) eqn: (1 + 2*x)^3 = 13.5*(1 + x^5);
                                  3          5
(%o1)                 (2 x + 1)  = 13.5 (x  + 1)
(%i2) soln: allroots (eqn);
(%o2) [x = .8296749902129361, x = - 1.015755543828121,

x = .9659625152196369 %i - .4069597231924075,

x = - .9659625152196369 %i - .4069597231924075, x = 1.0]
(%i3) for e in soln
        do (e2: subst (e, eqn), disp (expand (lhs(e2) - rhs(e2))));
                      - 3.5527136788005E-15

                     - 5.32907051820075E-15

         4.44089209850063E-15 %i - 4.88498130835069E-15

       - 4.44089209850063E-15 %i - 4.88498130835069E-15

                       3.5527136788005E-15

(%o3)                         done
(%i4) polyfactor: true$
(%i5) allroots (eqn);
```

```
(%o5) - 13.5 (x - 1.0) (x - .8296749902129361)

                                 2
 (x + 1.015755543828121) (x  + .8139194463848151 x

 + 1.098699797110288)
```

backsubst Option variable

Default value: `true`

When `backsubst` is `false`, prevents back substitution after the equations have been triangularized. This may be helpful in very big problems where back substitution would cause the generation of extremely large expressions.

breakup Option variable

Default value: `true`

When `breakup` is `true`, `solve` expresses solutions of cubic and quartic equations in terms of common subexpressions, which are assigned to intermediate expression labels (`%t1`, `%t2`, etc.). Otherwise, common subexpressions are not identified.

`breakup: true` has an effect only when `programmode` is `false`.

Examples:

```
(%i1) programmode: false$
(%i2) breakup: true$
(%i3) solve (x^3 + x^2 - 1);

                        sqrt(23)    25 1/3
(%t3)                  (--------- + --)
                        6 sqrt(3)   54
Solution:

                                          sqrt(3) %i   1
                                          ---------- - -
                sqrt(3) %i   1                2        2   1
(%t4)    x = (- ---------- - -) %t3 + -------------- - -
                    2        2            9 %t3            3

                                          sqrt(3) %i   1
                                        - ---------- - -
              sqrt(3) %i   1                  2        2   1
(%t5)    x = (---------- - -) %t3 + ---------------- - -
                  2        2            9 %t3              3

                                   1     1
(%t6)                  x = %t3 + ----- - -
                                 9 %t3   3
(%o6)                    [%t4, %t5, %t6]
(%i6) breakup: false$
(%i7) solve (x^3 + x^2 - 1);
```

```
Solution:

                 sqrt(3) %i   1
                 ---------- - -
                     2        2        sqrt(23)    25 1/3
(%t7) x = --------------------- + (--------- + --)
                 sqrt(23)    25 1/3    6 sqrt(3)   54
              9 (--------- + --)
                 6 sqrt(3)   54

                                                          sqrt(3) %i   1    1
                                                       (- ---------- - -) - -
                                                              2        2    3

               sqrt(23)    25 1/3  sqrt(3) %i   1
(%t8) x = (--------- + --)    (---------- - -)
               6 sqrt(3)   54          2        2

                                                        sqrt(3) %i   1
                                                      - ---------- - -
                                                            2        2      1
                                                  + --------------------- - -
                                                       sqrt(23)    25 1/3   3
                                                     9 (--------- + --)
                                                       6 sqrt(3)   54

                sqrt(23)    25 1/3             1             1
(%t9)  x = (--------- + --)    + --------------------- - -
                6 sqrt(3)   54          sqrt(23)    25 1/3   3
                                     9 (--------- + --)
                                        6 sqrt(3)   54
(%o9)                     [%t7, %t8, %t9]
```

dimension (*eqn*) Function

dimension (*eqn_1*, ..., *eqn_n*) Function

`dimen` is a package for dimensional analysis. `load ("dimen")` loads this package. `demo ("dimen")` displays a short demonstration.

dispflag Option variable

Default value: `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

Returns `[g(t) = ...]` or `[]`, depending on whether or not there exists a rational function `g(t)` satisfying *eqn*, which must be a first order, linear polynomial in (for this case) `g(t)` and `g(t+1)`

```
(%i1) eqn: (n + 1)*f(n) - (n + 3)*f(n + 1)/(n + 1) = (n - 1)/(n + 2);
                            (n + 3) f(n + 1)   n - 1
(%o1)        (n + 1) f(n) - ---------------- = -----
                                 n + 1         n + 2
(%i2) funcsolve (eqn, f(n));

Dependent equations eliminated:  (4 3)
                                   n
(%o2)                f(n) = ---------------
                            (n + 1) (n + 2)
```

Warning: this is a very rudimentary implementation – many safety checks and obvious generalizations are missing.

globalsolve Option variable

Default value: `false`

When `globalsolve` is `true`, solved-for variables are assigned the solution values found by `solve`.

Examples:

```
(%i1) globalsolve: true$
(%i2) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);
Solution

                                 17
(%t2)                        x : --
                                 7

                                   1
(%t3)                        y : - -
                                   7
(%o3)                     [[%t2, %t3]]
(%i3) x;
                               17
(%o3)                          --
                               7
(%i4) y;
                                 1
(%o4)                          - -
                                 7
(%i5) globalsolve: false$
(%i6) kill (x, y)$
(%i7) solve ([x + 3*y = 2, 2*x - y = 5], [x, y]);
Solution

                                 17
(%t7)                        x = --
                                 7

                                   1
```

```
(%t8)                          y = - -
                                     7
(%o8)                        [[%t7, %t8]]
(%i8) x;
(%o8)                             x
(%i9) y;
(%o9)                             y
```

ieqn (*ie, unk, tech, n, guess*) Function

`inteqn` is a package for solving integral equations. `load ("inteqn")` loads this package.

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 Maxima 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 Option variable

Default value: `true`

`ieqnprint` governs the behavior of the result returned by the `ieqn` command. When `ieqnprint` is `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

Returns the left side of the equation *eqn*.

If the argument is not an equation, `lhs` returns the argument.

See also `rhs`.

Example:

```
(%i1) e: x^2 + y^2 = z^2;
                               2    2    2
(%o1)                         y  + x  = z
(%i2) lhs (e);
                                 2    2
(%o2)                           y  + x
(%i3) rhs (e);
                                   2
(%o3)                             z
```

linsolve ([*expr_1, ..., expr_m*], [*x_1, ..., x_n*]) Function

Solves the list of simultaneous linear equations for the list of variables. The expressions must each be polynomials in the variables and may be equations.

When `globalsolve` is `true` then variables which are solved for will be set to the solution of the set of simultaneous equations.

When `backsubst` is `false`, `linsolve` does not carry out 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.

When `linsolve_params` is `true`, `linsolve` also generates the `%r` symbols used to represent arbitrary parameters described in the manual under `algsys`. Otherwise, `linsolve` solves an under-determined system of equations with some variables expressed in terms of others.

```
(%i1) e1: x + z = y$
(%i2) e2: 2*a*x - y = 2*a^2$
(%i3) e3: y - 2*z = 2$
(%i4) linsolve ([e1, e2, e3], [x, y, z]);
(%o4)            [x = a + 1, y = 2 a, z = a - 1]
```

linsolvewarn Option variable

Default value: `true`

When `linsolvewarn` is `true`, `linsolve` prints a message "Dependent equations eliminated".

linsolve_params Option variable

Default value: `true`

When `linsolve_params` is `true`, `linsolve` also generates the `%r` symbols used to represent arbitrary parameters described in the manual under `algsys`. Otherwise, `linsolve` solves an under-determined system of equations with some variables expressed in terms of others.

multiplicities System variable

Default value: `not_set_yet`

`multiplicities` is set to a list of the multiplicities of the individual solutions returned by `solve` or `realroots`.

nroots (*p, low, high*) Function

Returns the number of real roots of the real univariate polynomial *p* in the half-open interval (*low*, *high*]. The endpoints of the interval may be `minf` or `inf`. infinity and plus infinity.

`nroots` uses the method of Sturm sequences.

```
(%i1) p: x^10 - 2*x^4 + 1/2$
(%i2) nroots (p, -6, 9.1);
(%o2)                           4
```

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 nth power. This routine is much faster than `factor` or even `sqfr`.

programmode Option variable

Default value: `true`

When `programmode` is `true`, `solve`, `realroots`, `allroots`, and `linsolve` return solutions as elements in a list. (Except when `backsubst` is set to `false`, in which case `programmode: false` is assumed.)

When `programmode` is `false`, `solve`, etc. create intermediate expression labels `%t1`, `t2`, etc., and assign the solutions to them.

realonly Option variable

Default value: `false`

When `realonly` is `true`, `algsys` returns 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 (p)` is equivalent to `realroots (p, rootsepsilon)`. `rootsepsilon` is a real number used to establish the confidence interval for the roots. Do `example (realroots)` for an example.

rhs (*eqn*) Function

Returns the right side of the equation *eqn*.

If the argument is not an equation, `rhs` returns `0`.

See also `lhs`.

Example:

```
(%i1) e: x^2 + y^2 = z^2;
                                2    2    2
(%o1)                          y  + x  = z
(%i2) lhs (e);
```

```
                                 2    2
(%o2)                           y  + x
(%i3) rhs (e);
                                   2
(%o3)                             z
```

rootsconmode Option variable

Default value: `true`

`rootsconmode` governs the behavior of the `rootscontract` command. See `rootscontract` for details.

rootscontract (*expr*) Function

Converts products of roots into roots of products. For example, `rootscontract (sqrt(x)*y^(3/2))` yields `sqrt(x*y^3)`.

When `radexpand` is `true` and `domain` is `real`, `rootscontract` converts `abs` into `sqrt`, e.g., `rootscontract (abs(x)*sqrt(y))` yields `sqrt(x^2*y)`.

There is an option `rootsconmode` affecting `rootscontract` as follows:

```
Problem            Value of         Result of applying
                  rootsconmode        rootscontract

x^(1/2)*y^(3/2)      false          (x*y^3)^(1/2)
x^(1/2)*y^(1/4)      false          x^(1/2)*y^(1/4)
x^(1/2)*y^(1/4)      true           (x*y^(1/2))^(1/2)
x^(1/2)*y^(1/3)      true           x^(1/2)*y^(1/3)
x^(1/2)*y^(1/4)      all            (x^2*y)^(1/4)
x^(1/2)*y^(1/3)      all            (x^3*y^2)^(1/6)
```

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

`rootscontract` uses `ratsimp` in a manner similar to `logcontract`.

Examples:

```
(%i1) rootsconmode: false$
(%i2) rootscontract (x^(1/2)*y^(3/2));
                                   3
(%o2)                      sqrt(x y )
(%i3) rootscontract (x^(1/2)*y^(1/4));
                                   1/4
(%o3)                     sqrt(x) y
(%i4) rootsconmode: true$
(%i5) rootscontract (x^(1/2)*y^(1/4));
(%o5)                    sqrt(x sqrt(y))
(%i6) rootscontract (x^(1/2)*y^(1/3));
                                   1/3
(%o6)                     sqrt(x) y
(%i7) rootsconmode: all$
```

```
(%i8) rootscontract (x^(1/2)*y^(1/4));
                               2   1/4
(%o8)                        (x   y)
(%i9) rootscontract (x^(1/2)*y^(1/3));
                               3  2 1/6
(%o9)                        (x   y )
(%i10) rootsconmode: false$
(%i11) rootscontract (sqrt(sqrt(x) + sqrt(1 + x))
                  *sqrt(sqrt(1 + x) - sqrt(x)));
(%o11)                          1
(%i12) rootsconmode: true$
(%i13) rootscontract (sqrt(5 + sqrt(5)) - 5^(1/4)*sqrt(1 + sqrt(5)));
(%o13)                          0
```

rootsepsilon Option variable

Default value: 1.0e-7

`rootsepsilon` is the tolerance which establishes the confidence interval for the roots found by the `realroots` function.

solve (*expr, x*) Function
solve (*expr*) Function
solve ([*eqn_1, ..., eqn_n*], [*x_1, ..., x_n*]) Function

Solves the algebraic equation *expr* for the variable *x* and returns a list of solution equations in *x*. If *expr* is not an equation, the equation *expr* `= 0` is assumed in its place. *x* may be a function (e.g. `f(x)`), or other non-atomic expression except a sum or product. *x* may be omitted if *expr* contains only one variable. *expr* 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` 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` - 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 by used on the individual switch names if their purpose is not clear.

`solve ([eqn_1, ..., eqn_n], [x_1, ..., x_n])` 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 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.

Examples:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);

SOLVE is using arc-trig functions to get a solution.
Some solutions will be lost.
                                  %pi
(%o1)                  [x = ---, f(x) = 1]
                                   6
(%i2) ev (solve (5^f(x) = 125, f(x)), solveradcan);
                                log(125)
(%o2)                   [f(x) = --------]
                                 log(5)
(%i3) [4*x^2 - y^2 = 12, x*y - x = 2];
                              2    2
(%o3)              [4 x  - y  = 12, x y - x = 2]
(%i4) solve (%, [x, y]);
(%o4) [[x = 2, y = 2], [x = .5202594388652008 %i

 - .1331240357358706, y = .0767837852378778

 - 3.608003221870287 %i], [x = - .5202594388652008 %i

 - .1331240357358706, y = 3.608003221870287 %i

 + .0767837852378778], [x = - 1.733751846381093,

y = - .1535675710019696]]
(%i5) solve (1 + a*x + x^3, x);
                                                3
              sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(%o5) [x = (- ---------- - -) (--------------- - -)
                  2        2      6 sqrt(3)      2

        sqrt(3) %i   1
       (---------- - -) a
            2        2
 - --------------------------, x =
              3
```

```
      sqrt(4 a  + 27)   1 1/3
   3 (--------------- - -)
         6 sqrt(3)      2

                            3
 sqrt(3) %i   1   sqrt(4 a  + 27)   1 1/3
(---------- - -) (--------------- - -)
     2        2      6 sqrt(3)      2

         sqrt(3) %i   1
      (- ---------- - -) a
             2        2
 - --------------------------, x =
             3
      sqrt(4 a  + 27)   1 1/3
   3 (--------------- - -)
         6 sqrt(3)      2

         3
 sqrt(4 a  + 27)   1 1/3               a
(--------------- - -)    - --------------------------]
    6 sqrt(3)      2                    3
                                 sqrt(4 a  + 27)   1 1/3
                              3 (--------------- - -)
                                    6 sqrt(3)      2
(%i6) solve (x^3 - 1);
              sqrt(3) %i - 1        sqrt(3) %i + 1
(%o6)   [x = --------------, x = - --------------, x = 1]
                    2                     2
(%i7) solve (x^6 - 1);
           sqrt(3) %i + 1      sqrt(3) %i - 1
(%o7) [x = --------------, x = --------------, x = - 1,
                 2                   2

                     sqrt(3) %i + 1        sqrt(3) %i - 1
               x = - --------------, x = - --------------, x = 1]
                           2                     2
(%i8) ev (x^6 - 1, %[1]);
                                      6
                      (sqrt(3) %i + 1)
(%o8)                 ----------------- - 1
                             64
(%i9) expand (%);
(%o9)                           0
(%i10) x^2 - 1;
                                2
(%o10)                         x  - 1
(%i11) solve (%, x);
(%o11)                  [x = - 1, x = 1]
```

```
(%i12) ev (%th(2), %[1]);
(%o12)                         0
```

solvedecomposes Option variable

Default value: `true`

When `solvedecomposes` is `true`, `solve` calls `polydecomp` if asked to solve polynomials.

solveexplicit Option variable

Default value: `false`

When `solveexplicit` is `true`, inhibits `solve` from returning implicit solutions, that is, solutions of the form `F(x) = 0` where `F` is some function.

solvefactors Option variable

Default value: `true`

When `solvefactors` is `false`, `solve` does not try to factor the expression. The `false` setting may be desired in some cases where factoring is not necessary.

solvenullwarn Option variable

Default value: `true`

When `solvenullwarn` is `true`, `solve` prints a warning message if called with either a null equation list or a null variable list. For example, `solve ([], [])` would print two warning messages and return `[]`.

solveradcan Option variable

Default value: `false`

When `solveradcan` is `true`, `solve` calls `radcan` which makes `solve` slower but will allow certain problems containing exponentials and logarithms to be solved.

solvetrigwarn Option variable

Default value: `true`

When `solvetrigwarn` is `true`, `solve` may print a message saying that it is using inverse trigonometric functions to solve the equation, and thereby losing solutions.

solve_inconsistent_error Option variable

Default value: `true`

When `solve_inconsistent_error` is `true`, `solve` and `linsolve` give an error if the equations to be solved are inconsistent.

If `false`, `solve` and `linsolve` return an empty list `[]` if the equations are inconsistent.

Example:

```
(%i1) solve_inconsistent_error: true$
(%i2) solve ([a + b = 1, a + b = 2], [a, b]);
Inconsistent equations:  (2)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i3) solve_inconsistent_error: false$
(%i4) solve ([a + b = 1, a + b = 2], [a, b]);
(%o4)                          []
```

23 Differential Equations

23.1 Definitions for Differential Equations

bc2 (*solution, xval1, yval1, xval2, yval2*) Function

Solves boundary value problem for second order differential equation. Here: *solution* is a general solution to the equation, as found by `ode2`, *xval1* is an equation for the independent variable in the form `x = `*x0*, and *yval1* is an equation for the dependent variable in the form `y = `*y0*. The *xval2* and *yval2* are equations for these variables at another point. See `ode2` for example of usage.

desolve (*eqn, x*) Function

desolve ([*eqn_1, ..., eqn_n*], [*x_1, ..., x_n*]) Function

The function `dsolve` solves systems of linear ordinary differential equations using Laplace transform. Here the *eqn*'s are differential equations in the dependent variables *x_1*, ..., *x_n*. The functional relationships must be explicitly indicated in both the equations and the variables. For example

```
'diff(f,x,2)=sin(x)+'diff(g,x);
'diff(f,x)+x^2-f=2*'diff(g,x,2);
```

is not the proper format. The correct way is:

```
'diff(f(x),x,2)=sin(x)+'diff(g(x),x);
'diff(f(x),x)+x^2-f=2*'diff(g(x),x,2);
```

The call is then `desolve([%o3,%o4],[f(x),g(x)]);` .

If initial conditions at 0 are known, they should be supplied before calling `desolve` by using `atvalue`.

```
(%i1) 'diff(f(x),x)='diff(g(x),x)+sin(x);
                 d           d
(%o1)            -- (f(x)) = -- (g(x)) + sin(x)
                 dx          dx
(%i2) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
                  2
                 d            d
(%o2)            --- (g(x)) = -- (f(x)) - cos(x)
                   2          dx
                 dx
(%i3) atvalue('diff(g(x),x),x=0,a);
(%o3)                          a
(%i4) atvalue(f(x),x=0,1);
(%o4)                          1
(%i5) desolve([%o1,%o2],[f(x),g(x)]);
                  x
(%o5) [f(x) = a %e  - a + 1, g(x) =

                                                x
                                   cos(x) + a %e  - a + g(0) - 1]
```

```
(%i6) [%o1,%o2],%o5,diff;
                 x        x       x                   x
(%o6)    [a %e  = a %e , a %e  - cos(x) = a %e  - cos(x)]
```

If `desolve` cannot obtain a solution, it returns `false`.

ic1 (*solution, xval, yval*) Function

Solves initial value problem for first order differential equation. Here: *solution* is a general solution to the equation, as found by `ode2`, *xval* is an equation for the independent variable in the form *x* = *x0*, and *yval* is an equation for the dependent variable in the form *y* = *y0*. See `ode2` for example of usage.

ic2 (*solution, xval, yval, dval*) Function

Solves initial value problem for second order differential equation. Here: *solution* is a general solution to the equation, as found by `ode2`, *xval* is an equation for the independent variable in the form *x* = *x0*, *yval* is an equation for the dependent variable in the form *y* = *y0*, and *dval* is an equation for the derivative of the dependent variable with respect to independent variable evaluated at the point *xval*. See `ode2` for example of usage.

ode2 (*eqn, dvar, ivar*) Function

The function `ode2` solves ordinary differential equations of first or second order. It takes three arguments: an ODE *eqn*, the dependent variable *dvar*, and the independent variable *ivar*. 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.

In order to solve initial value problems (IVPs) and boundary value problems (BVPs), the routine `ic1` is available for first order equations, and `ic2` and `bc2` for second order IVPs and BVPs, respectively.

Example:

```
(%i1) x^2*'diff(y,x) + 3*y*x = sin(x)/x;
                          2 dy           sin(x)
```

```
(%o1)                x  -- + 3 x y = ------
                        dx               x
(%i2) ode2(%,y,x);
                             %c - cos(x)
(%o2)                     y = -----------
                                   3
                                  x
(%i3) ic1(%o2,x=%pi,y=0);
                              cos(x) + 1
(%o3)                   y = - ----------
                                   3
                                  x
(%i4) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                         2
                        d y      dy 3
(%o4)                   --- + y (--)  = 0
                          2      dx
                        dx
(%i5) ode2(%,y,x);
                      3
                     y  + 6 %k1 y
(%o5)                ------------ = x + %k2
                          6
(%i6) ratsimp(ic2(%o5,x=0,y=0,'diff(y,x)=2));
                             3
                          2 y  - 3 y
(%o6)                   - ---------- = x
                              6
(%i7) bc2(%o5,x=0,y=1,x=1,y=3);
                         3
                        y  - 10 y       3
(%o7)                   --------- = x - -
                            6           2
```

24 Numerical

24.1 Introduction to Numerical

24.2 Fourier packages

The `fft` package comprises functions for the numerical (not symbolic) computation of the fast Fourier transform. `load ("fft")` loads this package. See `fft`.

The `fourie` package comprises functions for the symbolic computation of Fourier series. `load ("fourie")` loads this package. There are functions in the `fourie` package to calculate Fourier integral coefficients and some functions for manipulation of expressions. See `Definitions for Fourier Series`.

24.3 Definitions for Numerical

polartorect (*magnitude_array, phase_array*) Function

Translates complex values of the form `r %e^(%i t)` to the form `a + b %i`. `load ("fft")` loads this function into Maxima. See also `fft`.

The magnitude and phase, `r` and `t`, are taken from *magnitude_array* and *phase_array*, respectively. The original values of the input arrays are replaced by the real and imaginary parts, `a` and `b`, on return. The outputs are calculated as

```
a: r cos (t)
b: r sin (t)
```

The input arrays must be the same size and 1-dimensional. The array size need not be a power of 2.

`polartorect` is the inverse function of `recttopolar`.

recttopolar (*real_array, imaginary_array*) Function

Translates complex values of the form `a + b %i` to the form `r %e^(%i t)`. `load ("fft")` loads this function into Maxima. See also `fft`.

The real and imaginary parts, `a` and `b`, are taken from *real_array* and *imaginary_array*, respectively. The original values of the input arrays are replaced by the magnitude and angle, `r` and `t`, on return. The outputs are calculated as

```
r: sqrt (a^2 + b^2)
t: atan2 (b, a)
```

The computed angle is in the range `-%pi` to `%pi`.

The input arrays must be the same size and 1-dimensional. The array size need not be a power of 2.

`recttopolar` is the inverse function of `polartorect`.

ift (*real_array, imaginary_array*) Function

Fast inverse discrete Fourier transform. `load ("fft")` loads this function into Maxima.

`ift` carries out the inverse complex fast Fourier transform on 1-dimensional floating point arrays. The inverse transform is defined as

```
x[j]: sum (y[j] exp (+2 %i %pi j k / n), k, 0, n-1)
```

See `fft` for more details.

fft (*real_array, imaginary_array*) Function
ift (*real_array, imaginary_array*) Function
recttopolar (*real_array, imaginary_array*) Function
polartorect (*magnitude_array, phase_array*) Function

Fast Fourier transform and related functions. `load ("fft")` loads these functions into Maxima.

`fft` and `ift` carry out the complex fast Fourier transform and inverse transform, respectively, on 1-dimensional floating point arrays. The size of *imaginary_array* must equal the size of *real_array*.

`fft` and `ift` operate in-place. That is, on return from `fft` or `ift`, the original content of the input arrays is replaced by the output. The `fillarray` function can make a copy of an array, should it be necessary.

The discrete Fourier transform and inverse transform are defined as follows. Let `x` be the original data, with

```
x[i]: real_array[i] + %i imaginary_array[i]
```

Let `y` be the transformed data. The forward and inverse transforms are

```
y[k]: (1/n) sum (x[j] exp (-2 %i %pi j k / n), j, 0, n-1)

x[j]:       sum (y[j] exp (+2 %i %pi j k / n), k, 0, n-1)
```

Suitable arrays can be allocated by the `array` function. For example:

```
array (my_array, float, n-1)$
```

declares a 1-dimensional array with n elements, indexed from 0 through n-1 inclusive. The number of elements n must be equal to 2^m for some m.

`fft` can be applied to real data (imaginary array all zeros) to obtain sine and cosine coefficients. After calling `fft`, the sine and cosine coefficients, say `a` and `b`, can be calculated as

```
a[0]: real_array[0]
b[0]: 0
```

and

```
a[j]: real_array[j] + real_array[n-j]
b[j]: imaginary_array[j] - imaginary_array[n-j]
```

for j equal to 1 through n/2-1, and

```
a[n/2]: real_array[n/2]
b[n/2]: 0
```

`recttopolar` translates complex values of the form `a + b %i` to the form `r %e^(%i t)`. See `recttopolar`.

`polartorect` translates complex values of the form `r %e^(%i t)` to the form `a + b %i`. See `polartorect`.

`demo ("fft")` displays a demonstration of the `fft` package.

fortindent Option variable

Default value: 0

`fortindent` 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.

fortran (*expr*) Function

Prints *expr* as a Fortran statement. The output line is indented with spaces. If the line is too long, `fortran` prints continuation lines. `fortran` prints the exponentiation operator `^` as `**`, and prints a complex number `a + b %i` in the form `(a,b)`.

expr may be an equation. If so, `fortran` prints an assignment statement, assigning the right-hand side of the equation to the left-hand side. In particular, if the right-hand side of *expr* is the name of a matrix, then `fortran` prints an assignment statement for each element of the matrix.

If *expr* is not something recognized by `fortran`, the expression is printed in `grind` format without complaint. `fortran` does not know about lists, arrays, or functions.

`fortindent` controls the left margin of the printed lines. 0 is the normal margin (i.e., indented 6 spaces). Increasing `fortindent` causes expressions to be printed further to the right.

When `fortspaces` is `true`, `fortran` fills out each printed line with spaces to 80 columns.

`fortran` evaluates its arguments; quoting an argument defeats evaluation. `fortran` always returns `done`.

Examples:

```
(%i1) expr: (a + b)^12$
(%i2) fortran (expr);
      (b+a)**12
(%o2)                         done
(%i3) fortran ('x=expr);
      x = (b+a)**12
(%o3)                         done
(%i4) fortran ('x=expand (expr));
      x = b**12+12*a*b**11+66*a**2*b**10+220*a**3*b**9+495*a**4*b**8+792
     1   *a**5*b**7+924*a**6*b**6+792*a**7*b**5+495*a**8*b**4+220*a**9*b
     2   **3+66*a**10*b**2+12*a**11*b+a**12
(%o4)                         done
(%i5) fortran ('x=7+5*%i);
      x = (7,5)
(%o5)                         done
(%i6) fortran ('x=[1,2,3,4]);
      x = [1,2,3,4]
(%o6)                         done
```

```
(%i7) f(x) := x^2$
(%i8) fortran (f);
      f
(%o8)                            done
```

fortspaces Option variable

Default value: `false`

When `fortspaces` is `true`, `fortran` fills out each printed line with spaces to 80 columns.

horner (*expr*, *x*) Function
horner (*expr*) Function

Returns a rearranged representation of *expr* as in Horner's rule, using *x* as the main variable if it is specified. `x` may be omitted in which case the main variable of the canonical rational expression form of *expr* is used.

`horner` sometimes improves stability if `expr` is to be numerically evaluated. It is also useful if Maxima is used to generate programs to be run in Fortran. See also `stringout`.

```
(%i1) expr: 1e-155*x^2 - 5.5*x + 5.2e155;
                                2
(%o1)             1.0E-155 x  - 5.5 x + 5.2E+155
(%i2) expr2: horner (%, x), keepfloat: true;
(%o2)             (1.0E-155 x - 5.5) x + 5.2E+155
(%i3) ev (expr, x=1e155);
Maxima encountered a Lisp error:

 floating point overflow

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i4) ev (expr2, x=1e155);
(%o4)                       7.0E+154
```

interpolate ($f(x)$, *x*, *a*, *b*) Function
interpolate (*f*, *a*, *b*) Function

Finds the zero of function *f* as variable *x* varies over the range `[`*a*`,` *b*`]`. The function must have a different sign at each endpoint. If this condition is not met, the action of the function is governed by `intpolerror`. 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 Maxima 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` and `intpolrel` 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, 0.1);
```

is equivalent to

```
interpolate (sin(x) = x/2, x, 0.1, %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.

```
(%i1) f(x) := sin(x) - x/2;
                                        x
(%o1)                 f(x) := sin(x) - -
                                        2
(%i2) interpolate (sin(x) - x/2, x, 0.1, %pi);
(%o2)                  1.895494267033981
(%i3) interpolate (sin(x) = x/2, x, 0.1, %pi);
(%o3)                  1.895494267033981
(%i4) interpolate (f(x), x, 0.1, %pi);
(%o4)                  1.895494267033981
(%i5) interpolate (f, 0.1, %pi);
(%o5)                  1.895494267033981
```

There is also a Newton method interpolation routine. See `newton`.

intpolabs — Option variable

Default value: 0.0

`intpolabs` is the accuracy of the `interpolate` command is governed by `intpolabs` and `intpolrel` 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 — Option variable

Default value: `true`

`intpolerror` 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 — Option variable

Default value: 0.0

`intpolrel` is the accuracy of the `interpolate` command is governed by `intpolabs` and `intpolrel` 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 (*expr, x, x_0, eps*) Function

Interpolation by Newton's method. `load ("newton1")` loads this function.

`newton` can handle some expressions that `interpolate` refuses to handle, since `interpolate` requires that everything evaluate to a floating point number. Thus `newton (x^2 - a^2, x, a/2, a^2/100)` complains that it can't tell if `6.098490481853958E-4 a^2 < a^2/100`. After `assume (a>0)`, the same function call succeeds, yielding a symbolic result, `1.00030487804878 a`.

On the other hand, `interpolate (x^2 - a^2, x, a/2, 2*a)` complains that `0.5 a` is not a floating point number.

An adaptive integrator which uses the Newton-Cotes 8 panel quadrature rule is available. See `qq`.

24.4 Definitions for Fourier Series

equalp (*x, y*) Function

Returns `true` if `equal (`*x*`, `*y*`)` otherwise `false` (doesn't give an error message like `equal (x, y)` would do in this case).

remfun (*f, expr*) Function

remfun (*f, expr, x*) Function

`remfun (`*f*`, `*expr*`)` replaces all occurrences of *f* `(`*arg*`)` by *arg* in *expr*.

`remfun (`*f*`, `*expr*`, `*x*`)` replaces all occurrences of *f* `(`*arg*`)` by *arg* in *expr* only if *arg* contains the variable *x*.

funp (*f, expr*) Function

funp (*f, expr, x*) Function

`funp (`*f*`, `*expr*`)` returns `true` if *expr* contains the function *f*.

`funp (`*f*`, `*expr*`, `*x*`)` returns `true` if *expr* contains the function *f* and the variable *x* is somewhere in the argument of one of the instances of *f*.

absint (*f, x, halfplane*) Function

absint (*f, x*) Function

absint (*f, x, a, b*) Function

`absint (`*f*`, `*x*`, `*halfplane*`)` returns the indefinite integral of *f* with respect to *x* in the given halfplane (`pos`, `neg`, or `both`). *f* may contain expressions of the form `abs (x)`, `abs (sin (x))`, `abs (a) * exp (-abs (b) * abs (x))`.

`absint (`*f*`, `*x*`)` is equivalent to `absint (`*f*`, `*x*`, pos)`.

`absint (`*f*`, `*x*`, `*a*`, `*b*`)` returns the definite integral of *f* with respect to *x* from *a* to *b*. *f* may include absolute values.

fourier (*f*, *x*, *p*) Function

Returns a list of the Fourier coefficients of `f(x)` defined on the interval `[-%pi, %pi]`.

foursimp (*l*) Function

Simplifies `sin (n %pi)` to 0 if `sinnpiflag` is `true` and `cos (n %pi)` to `(-1)^n` if `cosnpiflag` is `true`.

sinnpiflag Option variable

Default value: `true`

See `foursimp`.

cosnpiflag Option variable

Default value: `true`

See `foursimp`.

fourexpand (*l*, *x*, *p*, *limit*) Function

Constructs and returns the Fourier series from the list of Fourier coefficients *l* up through *limit* terms (*limit* may be `inf`). *x* and *p* have same meaning as in `fourier`.

fourcos (*f*, *x*, *p*) Function

Returns the Fourier cosine coefficients for `f(x)` defined on `[0, %pi]`.

foursin (*f*, *x*, *p*) Function

Returns the Fourier sine coefficients for `f(x)` defined on `[0, %pi]`.

totalfourier (*f*, *x*, *p*) Function

Returns `fourexpand (foursimp (fourier (f, x, p)), x, p, 'inf)`.

fourint (*f*, *x*) Function

Constructs and returns a list of the Fourier integral coefficients of `f(x)` defined on `[minf, inf]`.

fourintcos (*f*, *x*) Function

Returns the Fourier cosine integral coefficients for `f(x)` on `[0, inf]`.

fourintsin (*f*, *x*) Function

Returns the Fourier sine integral coefficients for `f(x)` on `[0, inf]`.

25 Statistics

25.1 Definitions for Statistics

gauss (*mean, sd*) Function

Returns a random floating point number from a normal distribution with mean *mean* and standard deviation *sd*.

26 Arrays and Tables

26.1 Definitions for Arrays and Tables

array (*name, dim_1, ..., dim_n*) Function
array (*name, type, dim_1, ..., dim_n*) Function
array ([*name_1, ..., name_m*], *dim_1, ..., dim_n*) Function

Creates an n-dimensional array. n may be less than or equal to 5. The subscripts for the i'th dimension are the integers running from 0 to *dim_i*.

`array (`*name*`, `*dim_1*`, ..., `*dim_n*`)` creates a general array.

`array (`*name*`, `*type*`, `*dim_1*`, ..., `*dim_n*`)` creates an array, with elements of a specified type. *type* can be `fixnum` for integers of limited size or `flonum` for floating-point numbers.

`array ([`*name_1*`, ..., `*name_m*`], `*dim_1*`, ..., `*dim_n*`)` creates m arrays, all of the same dimensions.

If the user assigns to a subscripted variable before declaring the corresponding array, an undeclared array is created. 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 (*A*, [*i_1, ..., i_n*]) Function

Evaluates *A* `[`*i_1*`, ..., `*i_n*`]`, where *A* is an array and *i_1, ..., i_n* are integers.

This is reminiscent of `apply`, except the first argument is an array instead of a function.

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*, [*i_1, ..., i_n*]) Function

Returns the expression *name* `[`*i_1*`, ..., `*i_n*`]`.

This is reminiscent of `funmake`, except the return value is an unevaluated array reference instead of an unevaluated function call.

arrays System variable

Default value: `[]`

`arrays` is a list of all the arrays that have been allocated, both declared and undeclared.

See also `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 (*A*, *B*) Function

Fills array *A* from *B*, which is a list or an array.

If *A* is a floating-point (integer) array then *B* should be either a list of floating-point (integer) numbers or another floating-point (integer) array.

If the dimensions of the arrays are different *A* is filled in row-major order. If there are not enough elements in *B* the last element is used to fill out the rest of *A*. 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 (*A*) Function

Returns a list of the elements of a declared or hashed array *A*. The order is row-major. Elements which are not yet defined are represented by `#####`.

make_array (*type*, *dim_1*, ..., *dim_n*) Function

Creates and returns a Lisp array. *type* may be `any`, `flonum`, `fixnum`, `hashed` or `functional`. There are n indices, and the i'th index runs from 0 to *dim_i* $- 1$.

The advantage of `make_array` over `array` is that the return value doesn't have a name, and once a pointer to it goes away, it will also go away. For example, if `y: make_array (...)` then `y` points to an object which takes up space, but after `y: false`, `y` no longer points to that object, so the object can be garbage collected.

`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 (*A, dim_1, ..., dim_n*) Function

Changes the 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, the remaining elements are filled with `false`, `0.0` or `0`, depending on the type of the array. The type of the array cannot be changed.

remarray (*A_1, ..., A_n*) Function

remarray (*all*) Function

Removes arrays and array associated functions and frees the storage occupied.

`remarray (all)` removes all items in the global list `arrays`.

It may be necessary to use this function if it is desired to redefine the values in a hashed array.

`remarray` returns the list of arrays removed.

use_fast_arrays Option variable

- 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 Maxima 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 art-q 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, e.g. `b[[x+1,u]]:y` would be valid. This is incompatible with the old Maxima 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 Maxima hashar. To obtain consistent behaviour in translated and compiled code set `translate_fast_arrays` to be `true`.

27 Matrices and Linear Algebra

27.1 Introduction to Matrices and Linear Algebra

27.1.1 Dot

The operator `.` represents noncommutative multiplication and scalar product. When the operands are 1-column or 1-row matrices `a` and `b`, the expression `a.b` is equivalent to `sum (a[i]*b[i], i, 1, length(a))`. If `a` and `b` are not complex, this is the scalar product, also called the inner product or dot product, of `a` and `b`. The scalar product is defined as `conjugate(a).b` when `a` and `b` are complex; `innerproduct` in the `eigen` package provides the complex scalar product.

When the operands are more general matrices, the product is the matrix product `a` and `b`. The number of rows of `b` must equal the number of columns of `a`, and the result has number of rows equal to the number of rows of `a` and number of columns equal to the number of columns of `b`.

To distinguish `.` as an arithmetic operator from the decimal point in a floating point number, it may be necessary to leave spaces on either side. For example, `5.e3` is `5000.0` but `5 . e3` is `5` times `e3`.

There are several flags which govern the simplification of expressions involving `.`, namely `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, and `dotscrules`.

27.1.2 Vectors

`vect` is a package of functions for vector analysis. `load ("vect")` loads this package, and `demo ("vect")` displays a demonstration.

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 governed by several flags, as are various other expansions, including expansion into components in any specific orthogonal coordinate systems. There are also functions for deriving the scalar or vector potential of a field.

The `vect` package contains these functions: `vectorsimp`, `scalefactors`, `express`, `potential`, and `vectorpotential`.

Warning: the `vect` package declares the dot operator `.` to be a commutative operator.

27.1.3 eigen

The package `eigen` contains several functions devoted to the symbolic computation of eigenvalues and eigenvectors. Maxima loads the package automatically if one of the functions `eigenvalues` or `eigenvectors` is invoked. The package may be loaded explicitly as `load ("eigen")`.

`demo ("eigen")` displays a demonstration of the capabilities of this package. `batch ("eigen")` executes the same demonstration, but without the user prompt between successive computations.

The functions in the `eigen` package are `innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors`, and `similaritytransform`.

27.2 Definitions for Matrices and Linear Algebra

addcol (*M, list_1, ..., list_n*) Function

Appends the column(s) given by the one or more lists (or matrices) onto the matrix *M*.

addrow (*M, list_1, ..., list_n*) Function

Appends the row(s) given by the one or more lists (or matrices) onto the matrix *M*.

adjoint (*M*) Function

Returns the adjoint of the matrix *M*.

augcoefmatrix ([*eqn_1, ..., eqn_m*], [*x_1, ..., x_n*]) Function

Returns the augmented coefficient matrix for the variables *x_1, ..., x_n* of the system of linear equations *eqn_1, ..., eqn_m*. This is the coefficient matrix with a column adjoined for the constant terms in each equation (i.e., those terms not dependent upon *x_1, ..., x_n*).

```
(%i1) m: [2*x - (a - 1)*y = 5*b, c + b*y + a*x = 0]$
(%i2) augcoefmatrix (m, [x, y]);
                       [ 2  1 - a  - 5 b ]
(%o2)                  [                 ]
                       [ a    b      c   ]
```

charpoly (*M, x*) Function

Returns the characteristic polynomial for the matrix *M* with respect to variable *x*. That is, `determinant (M - diagmatrix (length (M), x))`.

```
(%i1) a: matrix ([3, 1], [2, 4]);
                            [ 3  1 ]
(%o1)                       [      ]
                            [ 2  4 ]
(%i2) expand (charpoly (a, lambda));
                           2
(%o2)                lambda  - 7 lambda + 10
(%i3) (programmode: true, solve (%));
(%o3)               [lambda = 5, lambda = 2]
(%i4) matrix ([x1], [x2]);
                             [ x1 ]
(%o4)                        [    ]
                             [ x2 ]
```

```
(%i5) ev (a . % - lambda*%, %th(2)[1]);
                              [ x2 - 2 x1 ]
(%o5)                         [           ]
                              [ 2 x1 - x2 ]
(%i6) %[1, 1] = 0;
(%o6)                         x2 - 2 x1 = 0
(%i7) x2^2 + x1^2 = 1;
                                2     2
(%o7)                         x2  + x1  = 1
(%i8) solve ([%th(2), %], [x1, x2]);
                  1                 2
(%o8) [[x1 = - -------, x2 = - -------],
               sqrt(5)          sqrt(5)

                                                  1              2
                                           [x1 = -------, x2 = -------]]
                                                 sqrt(5)        sqrt(5)
```

coefmatrix ([*eqn_1*, ..., *eqn_m*], [*x_1*, ..., *x_n*]) Function

Returns the coefficient matrix for the variables *eqn_1*, ..., *eqn_m* of the system of linear equations *x_1*, ..., *x_n*.

col (*M*, *i*) Function

Returns the *i*'th column of the matrix *M*. The return value is a matrix.

columnvector (*L*) Function

covect (*L*) Function

Returns a matrix of one column and `length (`*L*`)` rows, containing the elements of the list *L*.

`covect` is a synonym for `columnvector`.

`load ("eigen")` loads this function.

This is useful if you want to use parts of the outputs of the functions in this package in matrix calculations.

Example:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function eigenvalues
Warning - you are redefining the Macsyma function eigenvectors
(%i2) columnvector ([aa, bb, cc, dd]);
                                 [ aa ]
                                 [    ]
                                 [ bb ]
(%o2)                            [    ]
                                 [ cc ]
                                 [    ]
                                 [ dd ]
```

conjugate (*x*) Function

Returns the complex conjugate of *x*.

`load (conjugate)` loads this function.

```
(%i1) declare ([aa, bb], real, cc, complex, ii, imaginary);

(%o1)                           done
(%i2) conjugate (aa + bb*%i);

(%o2)                        aa - %i bb
(%i3) conjugate (cc);

(%o3)                       conjugate(cc)
(%i4) conjugate (ii);

(%o4)                           - ii
(%i5) conjugate (xx + yy);

(%o5)               conjugate(yy) + conjugate(xx)
```

copymatrix (*M*) Function

Returns a copy of the matrix *M*. This is the only way to make a copy aside from copying *M* element by element.

Note that an assignment of one matrix to another, as in `m2: m1`, does not copy `m1`. An assignment `m2 [i,j]: x` or `setelmx (x, i, j, m2` also modifies `m1 [i,j]`. Creating a copy with `copymatrix` and then using assignment creates a separate, modified copy.

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 computing sparse determinants which is called when the switches `ratmx` and `sparse` are both `true`.

detout Option variable

Default value: `false`

When `detout` is `true`, the determinant of a matrix whose inverse is computed is factored out 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.

Example:

```
(%i1) m: matrix ([a, b], [c, d]);
                                [ a  b ]
(%o1)                           [      ]
                                [ c  d ]
(%i2) detout: true$
(%i3) doallmxops: false$
```

```
(%i4) doscmxops: false$
(%i5) invert (m);
                              [  d   - b ]
                              [          ]
                              [ - c   a  ]
(%o5)                         ------------
                               a d - b c
```

diagmatrix (*n*, *x*) Function

Returns a diagonal matrix of size *n* by *n* with the diagonal elements all equal to *x*. `diagmatrix (`*n*`, 1)` returns an identity matrix (same as `ident (`*n*`)`).

n must evaluate to an integer, otherwise `diagmatrix` complains with an error message.

x can be any kind of expression, including another matrix. If *x* is a matrix, it is not copied; all diagonal elements refer to the same instance, *x*.

doallmxops Option variable

Default value: `true`

When `doallmxops` is `true`, all operations relating to matrices are carried out. When it is `false` then the setting of the individual `dot` switches govern which operations are performed.

domxexpt Option variable

Default value: `true`

When `domxexpt` is `true`, a matrix exponential, `exp (`*M*`)` where *M* is a matrix, is interpreted as a matrix with element `[i,j` equal to `exp (m[i,j])`. Otherwise `exp (`*M*`)` evaluates to `exp (`*ev(M)*.

`domxexpt` affects all expressions of the form *base^power* where *base* is an expression assumed scalar or constant, and *power* is a list or matrix.

Example:

```
(%i1) m: matrix ([1, %i], [a+b, %pi]);
                         [   1    %i  ]
(%o1)                    [            ]
                         [ b + a  %pi ]
(%i2) domxexpt: false$
(%i3) (1 - c)^m;
                             [   1    %i  ]
                             [            ]
                             [ b + a  %pi ]
(%o3)                 (1 - c)
(%i4) domxexpt: true$
(%i5) (1 - c)^m;
                  [                      %i  ]
                  [    1 - c      (1 - c)    ]
(%o5)             [                          ]
                  [        b + a         %pi ]
                  [ (1 - c)       (1 - c)    ]
```

domxmxops Option variable

Default value: `true`

When `domxmxops` is `true`, all matrix-matrix or matrix-list operations are carried out (but not scalar-matrix operations); if this switch is `false` such operations are not carried out.

domxnctimes Option variable

Default value: `false`

When `domxnctimes` is `true`, non-commutative products of matrices are carried out.

dontfactor Option variable

Default value: `[]`

`dontfactor` may be set to a list of variables with respect to which factoring is not to occur. (The list is initially empty.) Factoring also will not take place with respect to any variables which are less important, according the variable ordering assumed for canonical rational expression (CRE) form, than those on the `dontfactor` list.

doscmxops Option variable

Default value: `false`

When `doscmxops` is `true`, scalar-matrix operations are carried out.

doscmxplus Option variable

Default value: `false`

When `doscmxplus` is `true`, scalar-matrix operations yield a matrix result. This switch is not subsumed under `doallmxops`.

dot0nscsimp Option variable

Default value: `true`

When `dot0nscsimp` is `true`, a non-commutative product of zero and a nonscalar term is simplified to a commutative product.

dot0simp Option variable

Default value: `true`

When `dot0simp` is `true`, a non-commutative product of zero and a scalar term is simplified to a commutative product.

dot1simp Option variable

Default value: `true`

When `dot1simp` is `true`, a non-commutative product of one and another term is simplified to a commutative product.

dotassoc Option variable

Default value: `true`

When `dotassoc` is `true`, an expression `(A.B).C` simplifies to `A.(B.C)`.

dotconstrules Option variable

Default value: `true`

When `dotconstrules` is `true`, a non-commutative product of a constant and another term is simplified to a commutative product. Turning on this flag effectively turns on `dot0simp`, `dot0nscsimp`, and `dot1simp` as well.

dotdistrib Option variable

Default value: `false`

When `dotdistrib` is `true`, an expression `A.(B + C)` simplifies to `A.B + A.C`.

dotexptsimp Option variable

Default value: `true`

When `dotexptsimp` is `true`, an expression `A.A` simplifies to `A^^2`.

dotident Option variable

Default value: 1

`dotident` is the value returned by `X^^0`.

dotscrules Option variable

Default value: `false`

When `dotscrules` is `true`, an expression `A.SC` or `SC.A` simplifies to `SC*A` and `A.(SC*B)` simplifies to `SC*(A.B)`.

echelon (*M*) Function

Returns the echelon form of the matrix *M*. The echelon form is computed from *M* by elementary row operations 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.

```
(%i1) m: matrix ([2, 1-a, -5*b], [a, b, c]);
                         [ 2  1 - a  - 5 b ]
(%o1)                    [                 ]
                         [ a    b      c   ]
(%i2) echelon (m);
                   [      a - 1      5 b     ]
                   [ 1  - -----    - ---     ]
                   [        2         2      ]
(%o2)/R/           [                         ]
                   [             2 c + 5 a b ]
                   [ 0     1     ------------ ]
                   [                   2     ]
                   [             2 b + a  - a ]
```

eigenvalues (*M*) Function
eivals (*M*) Function

Returns a list of two lists containing the eigenvalues of the matrix *M*. The first sublist of the return value is the list of eigenvalues of the matrix, and the second sublist is the list of the multiplicities of the eigenvalues in the corresponding order.

`eivals` is a synonym for `eigenvalues`.

`eigenvalues` calls the function `solve` 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 some other functions in this package (except `conjugate`, `innerproduct`, `unitvector`, `columnvector` and `gramschmidt`) will not work.

In some cases the eigenvalues found by `solve` may be complicated expressions. (This may happen when `solve` returns a not-so-obviously real expression for an eigenvalue which is known to be real.) It may be possible to simplify the eigenvalues using some other functions.

The package `eigen.mac` is loaded automatically when `eigenvalues` or `eigenvectors` is referenced. If `eigen.mac` is not already loaded, `load ("eigen")` loads it. After loading, all functions and variables in the package are available.

eigenvectors (*M*) Function
eivects (*M*) Function

takes a matrix *M* as its argument and returns a list of lists the first sublist of which is the output of `eigenvalues` and the other sublists of which are the eigenvectors of the matrix corresponding to those eigenvalues respectively. The calculated eigenvectors and the unit eigenvectors of the matrix are the right eigenvectors and the right unit eigenvectors respectively.

`eivects` is a synonym for `eigenvectors`.

The package `eigen.mac` is loaded automatically when `eigenvalues` or `eigenvectors` is referenced. If `eigen.mac` is not already loaded, `load ("eigen")` loads it. After loading, all functions and variables in the package are available.

The flags that affect this function are:

`nondiagonalizable` is set to `true` or `false` depending on whether the matrix is nondiagonalizable or diagonalizable after `eigenvectors` returns.

`hermitianmatrix` when `true`, causes the degenerate eigenvectors of the Hermitian matrix to be orthogonalized using the Gram-Schmidt algorithm.

`knowneigvals` when `true` causes the `eigen` package to 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 `eigenvalues`.

The function `algsys` is used here to solve for the eigenvectors. Sometimes if the eigenvalues are messy, `algsys` may not be able to find a solution. In some cases, it may be possible to simplify the eigenvalues by first finding them using `eigenvalues` command and then using other functions to reduce them to something simpler. Following simplification, `eigenvectors` can be called again with the `knowneigvals` flag set to `true`.

ematrix (*m, n, x, i, j*) Function

Returns an *m* by *n* matrix, all elements of which are zero except for the `[`*i*`,` *j*`]` element which is *x*.

entermatrix (*m, n*) Function

Returns an *m* by *n* matrix, reading the elements interactively.

If *n* is equal to *m*, Maxima prompts for the type of the matrix (diagonal, symmetric, antisymmetric, or general) and for each element. Each response is terminated by a semicolon `;` or dollar sign `$`.

If *n* is not equal to *m*, Maxima prompts for each element.

The elements may be any expressions, which are evaluated. `entermatrix` evaluates its arguments.

```
(%i1) n: 3$
(%i2) m: entermatrix (n, n)$

Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4 :
1$
Row 1 Column 1:
(a+b)^n$
Row 2 Column 2:
(a+b)^(n+1)$
Row 3 Column 3:
(a+b)^(n+2)$

Matrix entered.
(%i3) m;
                     [        3                     ]
                     [ (b + a)      0          0    ]
                     [                              ]
(%o3)                [                  4           ]
                     [    0      (b + a)       0    ]
                     [                              ]
                     [                            5 ]
                     [    0         0      (b + a)  ]
```

genmatrix (*a, i_2, j_2, i_1, j_1*) Function
genmatrix (*a, i_2, j_2, i_1*) Function
genmatrix (*a, i_2, j_2*) Function

Returns a matrix generated from *a*, taking element *a*`[`*i_1*`,`*j_1*`]` as the upper-left element and *a*`[`*i_2*`,`*j_2*`]` as the lower-right element of the matrix. Here *a* is an array (created by `array` but not by `make_array`) or an array function. (An array function is created like other functions with `:=` or `define`, but arguments are enclosed in square brackets instead of parentheses.)

If *j_1* is omitted, it is assumed equal to *i_1*. If both *j_1* and *i_1* are omitted, both are assumed equal to 1.

If a selected element `i,j` of the array is undefined, the matrix will contain a symbolic element *a*`[i,j]`.

```
(%i1) h[i,j] := 1/(i+j-1)$
(%i2) genmatrix (h, 3, 3);
                                    [    1  1 ]
```

```
                               [ 1  -  - ]
                               [    2  3 ]
                               [         ]
                               [ 1  1  1 ]
(%o2)                          [ -  -  - ]
                               [ 2  3  4 ]
                               [         ]
                               [ 1  1  1 ]
                               [ -  -  - ]
                               [ 3  4  5 ]
(%i3) array (a, fixnum, 2, 2)$
(%i4) a[1,1]: %e$
(%i5) a[2,2]: %pi$
(%i6) kill (a[1,2], a[2,1])$
(%i7) genmatrix (a, 2, 2);
                            [  %e    a     ]
                            [         1, 2 ]
(%o7)                       [              ]
                            [ a       %pi  ]
                            [  2, 1        ]
```

gramschmidt (*x*) Function

gschmit (*x*) Function

Carries out the Gram-Schmidt orthogonalization algorithm on *x*, which is either a matrix or a list of lists. *x* is not modified by `gramschmidt`.

If *x* is a matrix, the algorithm is applied to the rows of *x*. If *x* is a list of lists, the algorithm is applied to the sublists, which must have equal numbers of elements. In either case, the return value is a list of lists, the sublists of which are orthogonal and span the same space as *x*. If the dimension of the span of *x* is less than the number of rows or sublists, some sublists of the return value are zero.

`factor` is called at each stage of the algorithm to simplify intermediate results. As a consequence, the return value may contain factored integers.

`gschmit` (note spelling) is a synonym for `gramschmidt`.

`load ("eigen")` loads this function.

Example:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function eigenvalues
Warning - you are redefining the Macsyma function eigenvectors
(%i2) x: matrix ([1, 2, 3], [9, 18, 30], [12, 48, 60]);
                             [ 1   2   3  ]
                             [            ]
(%o2)                        [ 9   18  30 ]
                             [            ]
                             [ 12  48  60 ]
(%i3) y: gramschmidt (x);
                        2      2             4     3
                       3      3   3 5      2  3  2  3
```

```
(%o3)  [[1, 2, 3], [- ---, - --, ---], [- ----, ----, 0]]
                       2 7    7   2 7       5     5
(%i4) i: innerproduct$
(%i5) [i (y[1], y[2]), i (y[2], y[3]), i (y[3], y[1])];
(%o5)                         [0, 0, 0]
```

hach (*a, b, m, n, l*) Function

`hach` is an implementation of Hacijan's linear programming algorithm.

`load ("kach")` loads this function. `demo ("kach")` executes a demonstration of this function.

ident (*n*) Function

Returns an *n* by *n* identity matrix.

innerproduct (*x, y*) Function
inprod (*x, y*) Function

Returns the inner product (also called the scalar product or dot product) of *x* and *y*, which are lists of equal length, or both 1-column or 1-row matrices of equal length. The return value is `conjugate (x) . y`, where `.` is the noncommutative multiplication operator.

`load ("eigen")` loads this function.

`inprod` is a synonym for `innerproduct`.

invert (*M*) Function

Returns the inverse of the matrix *M*. The inverse is computed by 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.

Cofactors are computed by the `determinant` function, so if `ratmx` is `false` the inverse is computed without changing the representation of the elements.

The current implementation is inefficient for matrices of high order.

When `detout` is `true`, the determinant is factored out of the inverse.

The elements of the inverse are not automatically expanded. If *M* has polynomial elements, better appearing output can be generated by `expand (invert (m)), 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 (m)) / expand (determinant (m))
invert (m) := adjoint (m) / determinant (m)
```

See `^^` (noncommutative exponent) for another method of inverting a matrix.

lmxchar Option variable

Default value: `[`

`lmxchar` is the character displayed as the left delimiter of a matrix. See also `rmxchar`.

Example:

```
(%i1) lmxchar: "|"$
(%i2) matrix ([a, b, c], [d, e, f], [g, h, i]);
                                 | a  b  c ]
                                 |         ]
(%o2)                            | d  e  f ]
                                 |         ]
                                 | g  h  i ]
```

matrix (*row_1, ..., row_n*) Function

Returns a rectangular matrix which has the rows *row_1*, ..., *row_n*. Each row is a list of expressions. All rows must be the same length.

The operations `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), are carried out element by element when the operands are two matrices, a scalar and a matrix, or a matrix and a scalar. The operation `^` (exponentiation, equivalently `**`) is carried out element by element if the operands are a scalar and a matrix or a matrix and a scalar, but not if the operands are two matrices. All operations are normally carried out in full, including `.` (noncommutative multiplication).

Matrix multiplication is represented by the noncommutative multiplication operator `.`. The corresponding noncommutative exponentiation operator is `^^`. For a matrix *A*, *A*`.`*A* `=` *A*`^^2` and *A*`^^-1` is the inverse of *A*, if it exists.

There are switches for controlling simplification of expressions involving dot and matrix-list operations. These are `doallmxops`, `domxexpt domxmxops`, `doscmxops`, and `doscmxplus`.

There are additional options which are related to matrices. These are: `lmxchar`, `rmxchar`, `ratmx`, `listarith`, `detout`, `scalarmatrix`, and `sparse`.

There are a number of functions which take matrices as arguments or yield matrices as return values. See `eigenvalues`, `eigenvectors`, `determinant`, `charpoly`, `genmatrix`, `addcol`, `addrow`, `copymatrix`, `transpose`, `echelon`, and `rank`.

Examples:

- Construction of matrices from lists.

```
(%i1) x: matrix ([17, 3], [-8, 11]);
                                 [ 17   3  ]
(%o1)                            [         ]
                                 [ - 8  11 ]
(%i2) y: matrix ([%pi, %e], [a, b]);
                                 [ %pi  %e ]
(%o2)                            [         ]
                                 [  a   b  ]
```

- Addition, element by element.

```
(%i3) x + y;
                           [ %pi + 17  %e + 3 ]
(%o3)                      [                  ]
                           [  a - 8    b + 11 ]
```

- Subtraction, element by element.

```
(%i4) x - y;
                           [ 17 - %pi  3 - %e ]
(%o4)                      [                  ]
                           [ - a - 8   11 - b ]
```

- Multiplication, element by element.

```
(%i5) x * y;
                             [ 17 %pi  3 %e ]
(%o5)                        [              ]
                             [ - 8 a   11 b ]
```

- Division, element by element.

```
(%i6) x / y;
                             [ 17         - 1 ]
                             [ ---   3 %e     ]
                             [ %pi            ]
(%o6)                        [                ]
                             [   8    11      ]
                             [ - -    --      ]
                             [   a    b       ]
```

- Matrix to a scalar exponent, element by element.

```
(%i7) x ^ 3;
                              [ 4913    27  ]
(%o7)                         [             ]
                              [ - 512  1331 ]
```

- Scalar base to a matrix exponent, element by element.

```
(%i8) exp(y);
                              [   %pi    %e ]
                              [ %e     %e   ]
(%o8)                         [             ]
                              [    a     b  ]
                              [  %e    %e   ]
```

- Matrix base to a matrix exponent. This is not carried out element by element.

```
(%i9) x ^ y;
                                     [ %pi  %e ]
                                     [         ]
                                     [  a   b  ]
                          [ 17   3  ]
(%o9)                     [         ]
                          [ - 8  11 ]
```

- Noncommutative matrix multiplication.

```
(%i10) x . y;
                        [ 3 a + 17 %pi  3 b + 17 %e ]
(%o10)                  [                           ]
                        [ 11 a - 8 %pi  11 b - 8 %e ]
(%i11) y . x;
                      [ 17 %pi - 8 %e  3 %pi + 11 %e ]
(%o11)                [                              ]
                      [  17 a - 8 b     11 b + 3 a   ]
```

- Noncommutative matrix exponentiation. A scalar base b to a matrix power M is carried out element by element and so `b^^m` is the same as `b^m`.

```
(%i12) x ^^ 3;
                              [  3833   1719 ]
(%o12)                        [              ]
                              [ - 4584  395  ]
(%i13) %e ^^ y;
                               [   %pi    %e ]
                               [ %e     %e   ]
(%o13)                         [             ]
                               [    a     b  ]
                               [  %e    %e   ]
```

- A matrix raised to a -1 exponent with noncommutative exponentiation is the matrix inverse, if it exists.

```
(%i14) x ^^ -1;
                               [ 11      3  ]
                               [ ---  - --- ]
                               [ 211    211 ]
(%o14)                         [            ]
                               [  8    17   ]
                               [ ---   ---  ]
                               [ 211   211  ]
(%i15) x . (x ^^ -1);
                                  [ 1  0 ]
(%o15)                            [      ]
                                  [ 0  1 ]
```

matrixmap (*f*, *M*) Function

Returns a matrix with element `i,j` equal to *f*(*M*`[i,j]`).

See also `map`, `fullmap`, `fullmapl`, and `apply`.

matrixp (*expr*) Function

Returns `true` if *expr* is a matrix, otherwise `false`.

matrix_element_add Option variable

Default value: `+`

`matrix_element_add` is the operation invoked in place of addition in a matrix multiplication. `matrix_element_add` can be assigned any n-ary operator (that is, a function which handles any number of arguments). The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

See also `matrix_element_mult` and `matrix_element_transpose`.

Example:

```
(%i1) matrix_element_add: "*"$
(%i2) matrix_element_mult: "^"$
(%i3) aa: matrix ([a, b, c], [d, e, f]);
```

```
                              [ a  b  c ]
(%o3)                         [         ]
                              [ d  e  f ]
(%i4) bb: matrix ([u, v, w], [x, y, z]);
                              [ u  v  w ]
(%o4)                         [         ]
                              [ x  y  z ]
(%i5) aa . transpose (bb);
                        [  u  v  w   x  y  z ]
                        [ a  b  c   a  b  c  ]
(%o5)                   [                    ]
                        [  u  v  w   x  y  z ]
                        [ d  e  f   d  e  f  ]
```

matrix_element_mult Option variable

Default value: `*`

`matrix_element_mult` is the operation invoked in place of multiplication in a matrix multiplication. `matrix_element_mult` can be assigned any binary operator. The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

The dot operator `.` is a useful choice in some contexts.

See also `matrix_element_add` and `matrix_element_transpose`.

Example:

```
(%i1) matrix_element_add: lambda ([[x]], sqrt (apply ("+", x)))$
(%i2) matrix_element_mult: lambda ([x, y], (x - y)^2)$
(%i3) [a, b, c] . [x, y, z];
                          2          2          2
(%o3)         sqrt((c - z)  + (b - y)  + (a - x) )
(%i4) aa: matrix ([a, b, c], [d, e, f]);
                              [ a  b  c ]
(%o4)                         [         ]
                              [ d  e  f ]
(%i5) bb: matrix ([u, v, w], [x, y, z]);
                              [ u  v  w ]
(%o5)                         [         ]
                              [ x  y  z ]
(%i6) aa . transpose (bb);
                [             2          2          2  ]
                [ sqrt((c - w)  + (b - v)  + (a - u) ) ]
(%o6)  Col 1 =  [                                      ]
                [             2          2          2  ]
                [ sqrt((f - w)  + (e - v)  + (d - u) ) ]

                         [             2          2          2  ]
                         [ sqrt((c - z)  + (b - y)  + (a - x) ) ]
                 Col 2 = [                                      ]
                         [             2          2          2  ]
                         [ sqrt((f - z)  + (e - y)  + (d - x) ) ]
```

matrix_element_transpose Option variable

Default value: `false`

`matrix_element_transpose` is the operation applied to each element of a matrix when it is transposed. `matrix_element_mult` can be assigned any unary operator. The assigned value may be the name of an operator enclosed in quote marks, the name of a function, or a lambda expression.

When `matrix_element_transpose` equals `transpose`, the `transpose` function is applied to every element. When `matrix_element_transpose` equals `nonscalars`, the `transpose` function is applied to every nonscalar element. If some element is an atom, the `nonscalars` option applies `transpose` only if the atom is declared nonscalar, while the `transpose` option always applies `transpose`.

The default value, `false`, means no operation is applied.

See also `matrix_element_add` and `matrix_element_mult`.

Examples:

```
(%i1) declare (a, nonscalar)$
(%i2) transpose ([a, b]);
                        [ transpose(a) ]
(%o2)                   [              ]
                        [      b       ]
(%i3) matrix_element_transpose: nonscalars$
(%i4) transpose ([a, b]);
                        [ transpose(a) ]
(%o4)                   [              ]
                        [      b       ]
(%i5) matrix_element_transpose: transpose$
(%i6) transpose ([a, b]);
                        [ transpose(a) ]
(%o6)                   [              ]
                        [ transpose(b) ]
(%i7) matrix_element_transpose: lambda ([x], realpart(x) - %i*imagpart(x))$
(%i8) m: matrix ([1 + 5*%i, 3 - 2*%i], [7*%i, 11]);
                    [ 5 %i + 1  3 - 2 %i ]
(%o8)               [                    ]
                    [   7 %i       11    ]
(%i9) transpose (m);
                     [ 1 - 5 %i  - 7 %i ]
(%o9)                [                  ]
                     [ 2 %i + 3    11   ]
```

mattrace (*M*) Function

Returns the trace (that is, the sum of the elements on the main diagonal) of the square matrix *M*.

`mattrace` is called by `ncharpoly`, an alternative to Maxima's `charpoly`.

`load ("nchrpl")` loads this function.

minor (*M, i, j*) Function

Returns the *i*, *j* minor of the matrix *M*. That is, *M* with row *i* and column *j* removed.

ncexpt (*a*, *b*) Function

If a non-commutative exponential expression is too wide to be displayed as a^^b it appears as `ncexpt (a,b)`.

`ncexpt` is not the name of a function or operator; the name only appears in output, and is not recognized in input.

ncharpoly (*M*, *x*) Function

Returns the characteristic polynomial of the matrix M with respect to x. This is an alternative to Maxima'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 x `* ident [n] - a`. Thus `ncharpoly` wins, for example, in the case of large dense matrices filled with integers, since it avoids polynomial arithmetic altogether.

`load ("nchrpl")` loads this file.

newdet (*M*, *n*) Function

Computes the determinant of the matrix or array M by the Johnson-Gentleman tree minor algorithm. The argument n is the order; it is optional if M is a matrix.

nonscalar Declaration

Makes atoms behave as does a list or matrix with respect to the dot operator.

nonscalarp (*expr*) Function

Returns `true` if *expr* 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.

rank may return the wrong answer if it cannot determine that a matrix element that is equivalent to zero is indeed so.

ratmx Option variable

Default value: `false`

When `ratmx` is `false`, determinant and matrix addition, subtraction, and multiplication are performed in the representation of the matrix elements and cause the result of matrix inversion to be left in general representation.

When `ratmx` is `true`, the 4 operations mentioned above are performed in CRE form and the result of matrix inverse is 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

Returns the *i*'th row of the matrix *M*. The return value is a matrix.

scalarmatrixp Option variable

Default value: `true`

When `scalarmatrixp` is `true`, then whenever a 1 x 1 matrix is produced as a result of computing the dot product of matrices it is simplified to a scalar, namely the sole element of the matrix.

When `scalarmatrixp` is `all`, then all 1 x 1 matrices are simplified to scalars.

When `scalarmatrixp` is `false`, 1 x 1 matrices are not simplified to scalars.

scalefactors (*coordinatetransform*) Function

Here coordinatetransform evaluates to the form [[expression1, expression2, ...], indeterminate1, indeterminat2, ...], 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

setelmx (*x*, *i*, *j*, *M*) Function

Assigns *x* to the (*i*, *j*)'th element of the matrix *M*, and returns the altered matrix.

M `[`*i*`,` *j*`]:` *x* has the same effect, but returns *x* instead of *M*.

similaritytransform (*M*) Function
simtran (*M*) Function

`similaritytransform` computes a similarity transform of the matrix `M`. It returns a list which is the output of the `uniteigenvectors` command. In addition if the flag `nondiagonalizable` is `false` two global matrices `leftmatrix` and `rightmatrix` are computed. These matrices have the property that `leftmatrix .` *M* `. rightmatrix` is a diagonal matrix with the eigenvalues of *M* on the diagonal. If `nondiagonalizable` is `true` the left and right matrices are not computed.

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 *M*. The other flags (see `eigenvalues` and `eigenvectors`) have the same effects since `similaritytransform` calls the other functions in the package in order to be able to form `rightmatrix`.

`load ("eigen")` loads this function.

`simtran` is a synonym for `similaritytransform`.

sparse Option variable

Default value: `false`

When `sparse` is `true`, and if `ratmx` is `true`, then `determinant` will use special routines for computing sparse determinants.

submatrix (*i_1, ..., i_m, M, j_1, ..., j_n*) Function
submatrix (*i_1, ..., i_m, M*) Function
submatrix (*M, j_1, ..., j_n*) Function

Returns a new matrix composed of the matrix *M* with rows *i_1*, ..., *i_m* deleted, and columns *j_1*, ..., *j_n* deleted.

transpose (*M*) Function

Returns the transpose of *M*.

If *M* is a matrix, the return value is another matrix *N* such that `N[i,j] = M[j,i]`.

Otherwise *M* is a list, and the return value is a matrix *N* of `length (m)` rows and 1 column, such that `N[i,1] = M[i]`.

triangularize (*M*) Function

Returns the upper triangular form of the matrix `M`.

M need not be square.

uniteigenvectors (*M*) Function
ueivects (*M*) Function

Computes unit eigenvectors of the matrix *M*. The return value is 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.

When `knowneigvects` is `true`, the `eigen` package assumes 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.

`load ("eigen")` loads this function.

`ueivects` is a synonym for `uniteigenvectors`.

unitvector (x) Function
uvect (x) Function

Returns $x/norm(x)$; this is a unit vector in the same direction as x.

`load ("eigen")` loads this function.

`uvect` is a synonym for `unitvector`.

vectorsimp (*expr*) Function

Applies simplifications and expansions according to the following global flags:

`expandall`, `expanddot`, `expanddotplus`, `expandcross`, `expandcrossplus`, `expandcrosscross`, `expandgrad`, `expandgradplus`, `expandgradprod`, `expanddiv`, `expanddivplus`, `expanddivprod`, `expandcurl`, `expandcurlplus`, `expandcurlcurl`, `expandlaplacian`, `expandlaplacianplus`, and `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`
: Simplifies $p\ (q\ r)$ to $(p.r) * q - (p.q) * r$.

`expandcurlcurl`
: Simplifies $curlcurlp$ to $graddivp + divgradp$.

`expandlaplaciantodivgrad`
: Simplifies $laplacianp$ to $divgradp$.

`expandcross`
: Enables `expandcrossplus` and `expandcrosscross`.

`expandplus`
: Enables `expanddotplus`, `expandcrossplus`, `expandgradplus`, `expanddivplus`, `expandcurlplus`, and `expandlaplacianplus`.

`expandprod`
: Enables `expandgradprod`, `expanddivprod`, and `expandlaplacianprod`.

These flags have all been declared `evflag`.

vect_cross Option variable

Default value: `false`

When `vect_cross` is `true`, it allows DIFF(X~Y,T) to work where ~ is defined in SHARE;VECT (where VECT_CROSS is set to `true`, anyway.)

zeromatrix (m, n) Function

Returns an m by n matrix, all elements of which are zero.

`"["` Special symbol
`"]"` Special symbol

`[` and `]` mark the beginning and end, respectively, of a list.

`[` and `]` also enclose the subscripts of a list, array, hash array, or array function.

Examples:

```
(%i1) x: [a, b, c];
(%o1)                         [a, b, c]
(%i2) x[3];
(%o2)                             c
(%i3) array (y, fixnum, 3);
(%o3)                             y
(%i4) y[2]: %pi;
(%o4)                            %pi
(%i5) y[2];
(%o5)                            %pi
(%i6) z['foo]: 'bar;
(%o6)                            bar
(%i7) z['foo];
(%o7)                            bar
(%i8) g[k] := 1/(k^2+1);
                                  1
(%o8)                     g  := ------
                           k      2
                                 k  + 1
(%i9) g[10];
                                 1
(%o9)                           ---
                                101
```

28 Affine

28.1 Definitions for Affine

fast_linsolve ([*expr_1*, ..., *expr_m*], [*x_1*, ..., *x_n*]) Function

Solves the simultaneous linear equations *expr_1*, ..., *expr_m* for the variables *x_1*, ..., *x_n*. Each *expr_i* may be an equation or a general expression; if given as a general expression, it is treated as an equation of the form *expr_i* `= 0`.

The return value is a list of equations of the form `[`*x_1* `=` *a_1*`, ...,` *x_n* `=` *a_n*`]` where *a_1*, ..., *a_n* are all free of *x_1*, ..., *x_n*.

`fast_linsolve` is faster than `linsolve` for system of equations which are sparse.

grobner_basis ([*expr_1*, ..., *expr_m*]) Function

Returns a Groebner basis for the equations *expr_1*, ..., *expr_m*. The function `polysimp` can then 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)` yields 0 if and only if *f* is in the ideal generated by *expr_1*, ..., *expr_m*, that is, if and only if *f* is a polynomial combination of the elements of *expr_1*, ..., *expr_m*.

set_up_dot_simplifications (*eqns*, *check_through_degree*) Function
set_up_dot_simplifications (*eqns*) 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)` yields 0 if and only if *f* is in the ideal generated by the equations, i.e., if and only if *f* is a polynomial combination of the elements of the equations.

The degree is that returned by `nc_degree`. This in turn is influenced by the weights of individual variables.

declare_weight (*x_1*, *w_1*, ..., *x_n*, *w_n*) Function

Assigns weights *w_1*, ..., *w_n* to *x_1*, ..., *x_n*, respectively. These are the weights used in computing `nc_degree`.

nc_degree (*p*) Function

Returns the degree of a noncommutative polynomial *p*. See `declare_weights`.

dotsimp (*f*) Function

Returns 0 if and only if *f* is in the ideal generated by the equations, i.e., if and only if *f* is a polynomial combination of the elements of the equations.

fast_central_elements ([*x_1, ..., x_n*], *n*) Function

If `set_up_dot_simplifications` has been previously done, finds the central polynomials in the variables *x_1*, ..., *x_n* in the given degree, *n*.

For example:

```
set_up_dot_simplifications ([y.x + x.y], 3);
fast_central_elements ([x, y], 2);
[y.y, x.x];
```

check_overlaps (*n, add_to_simps*) Function

Checks the overlaps thru degree *n*, 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_functiion`. An optional third argument `reset`, `false` says don't bother to query about resetting things.

mono ([*x_1, ..., x_n*], *n*) Function

Returns the list of independent monomials relative to the current dot simplifications of degree *n* in the variables *x_1*, ..., *x_n*.

monomial_dimensions (*n*) Function

Compute the Hilbert series through degree *n* for the current algebra.

extract_linear_equations ([*p_1, ..., p_n*], [*m_1, ..., m_n*]) Function

Makes a list of the coefficients of the noncommutative polynomials *p_1*, ..., *p_n* of the noncommutative monomials *m_1*, ..., *m_n*. The coefficients should be scalars. Use `list_nc_monomials` to build the list of monomials.

list_nc_monomials ([*p_1, ..., p_n*]) Function
list_nc_monomials (*p*) Function

Returns a list of the non commutative monomials occurring in a polynomial *p* or a list of polynomials *p_1*, ..., *p_n*.

create_list (*form, x_1, list_1, ..., x_n, list_n*) Function

Create a list by evaluating *form* with *x_1* bound to each element of *list_1*, and for each such binding bind *x_2* to each element of *list_2*, ... The number of elements in the result will be the product of the number of elements in each list. Each variable *x_i* must actually be a symbol–it will not be evaluated. The list arguments will be evaluated once at the beginning of the iteration.

```
(%i82) create_list1(x^i,i,[1,3,7]);
(%o82) [x,x^3,x^7]
```

With a double iteration:

```
(%i79) create_list([i,j],i,[a,b],j,[e,f,h]);
(%o79) [[a,e],[a,f],[a,h],[b,e],[b,f],[b,h]]
```

Instead of *list_i* two args may be supplied each of which should evaluate to a number. These will be the inclusive lower and upper bounds for the iteration.

```
(%i81) create_list([i,j],i,[1,2,3],j,1,i);
(%o81) [[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 Option variable

Default value: `false`

When `all_dotsimp_denoms` is a list, the denominators encountered by `dotsimp` are appended to the list. `all_dotsimp_denoms` may be initialized to an empty list `[]` before calling `dotsimp`.

By default, denominators are not collected by `dotsimp`.

29 itensor

29.1 Introduction to itensor

Maxima implements symbolic tensor manipulation of two distinct types: component tensor manipulation (`ctensor` package) and indicial tensor manipulation (`itensor` package).

Nota bene: Please see the note on 'new tensor notation' below.

Component tensor manipulation means that geometrical tensor objects are represented as arrays or matrices. Tensor operations such as contraction or covariant differentiation are carried out by actually summing over repeated (dummy) indices with `do` statements. That is, one explicitly performs operations on the appropriate tensor components stored in an array or matrix.

Indicial tensor manipulation is implemented by representing tensors as functions of their covariant, contravariant and derivative indices. Tensor operations such as contraction or covariant differentiation are performed by manipulating the indices themselves rather than the components to which they correspond.

These two approaches to the treatment of differential, algebraic and analytic processes in the context of Riemannian geometry have various advantages and disadvantages which reveal themselves only through the particular nature and difficulty of the user's problem. However, one should keep in mind the following characteristics of the two implementations:

The representation of tensors and tensor operations explicitly in terms of their components makes `ctensor` easy to use. Specification of the metric and the computation of the induced tensors and invariants is straightforward. Although all of Maxima's powerful simplification capacity is at hand, a complex metric with intricate functional and coordinate dependencies can easily lead to expressions whose size is excessive and whose structure is hidden. In addition, many calculations involve intermediate expressions which swell causing programs to terminate before completion. Through experience, a user can avoid avoid many of these difficulties.

Because of the special way in which tensors and tensor operations are represented in terms of symbolic operations on their indices, expressions which in the component representation would be unmanageable can sometimes be greatly simplified by using the special routines for symmetrical objects in `itensor`. In this way the structure of a large expression may be more transparent. On the other hand, because of the the special indicial representation in `itensor`, in some cases the user may find difficulty with the specification of the metric, function definition, and the evaluation of differentiated "indexed" objects.

29.1.1 New tensor notation

Until now, the `itensor` package in Maxima has used a notation that sometimes led to incorrect index ordering. Consider the following, for instance:

```
(%i2) imetric(g);
(%o2)                                done
(%i3) ishow(g([],[j,k])*g([],[i,l])*a([i,j],[]))$
                                 i l  j k
(%t3)                           g    g    a
```

```
                                                i j
(%i4) ishow(contract(%))$
                                           k l
(%t4)                                      a
```

This result is incorrect unless `a` happens to be a symmetric tensor. The reason why this happens is that although `itensor` correctly maintains the order within the set of covariant and contravariant indices, once an index is raised or lowered, its position relative to the other set of indices is lost.

To avoid this problem, a new notation has been developed that remains fully compatible with the existing notation and can be used interchangeably. In this notation, contravariant indices are inserted in the appropriate positions in the covariant index list, but with a minus sign prepended. Functions like `contract` and `ishow` are now aware of this new index notation and can process tensors appropriately.

In this new notation, the previous example yields a correct result:

```
(%i5) ishow(g([-j,-k],[])*g([-i,-l],[])*a([i,j],[]))$
                                       i l       j k
(%t5)                                 g    a    g
                                           i j
(%i6) ishow(contract(%))$
                                           l k
(%t6)                                      a
```

Presently, the only code that makes use of this notation is the `lc2kdt` function. Through this notation, it achieves consistent results as it applies the metric tensor to resolve Levi-Civita symbols without resorting to numeric indices.

Since this code is brand new, it probably contains bugs. While it has been tested to make sure that it doesn't break anything using the "old" tensor notation, there is a considerable chance that "new" tensors will fail to interoperate with certain functions or features. These bugs will be fixed as they are encountered... until then, caveat emptor!

29.1.2 Indicial tensor manipulation

The indicial tensor manipulation package may be loaded by `load(itensor)`. Demos are also available: try `demo(tensor)`.

In `itensor` a tensor is represented as an "indexed object" . This is a function of 3 groups of indices which represent the covariant, contravariant and derivative indices. The covariant indices are specified by a list as the first argument to the indexed object, and the contravariant indices by a list as the second argument. If the indexed object lacks either of these groups of indices then the empty list [] is given as the corresponding argument. Thus, `g([a,b],[c])` represents an indexed object called `g` which has two covariant indices `(a,b)`, one contravariant index (`c`) and no derivative indices.

The derivative indices, if they are present, are appended as additional arguments to the symbolic function representing the tensor. They can be explicitly specified by the user or be created in the process of differentiation with respect to some coordinate variable. Since ordinary differentiation is commutative, the derivative indices are sorted alphanumerically, unless `iframe_flag` is set to `true`, indicating that a frame metric is being used.. This canonical ordering makes it possible for Maxima to recognize that, for example, `t([a],[b],i,j)` is the same as `t([a],[b],j,i)`. Differentiation of an indexed object with respect to some coordinate whose index does not appear as an argument to the indexed object would normally yield zero. This is because Maxima would not know that the tensor represented by the indexed object might depend implicitly on the corresponding coordinate. By modifying the existing Maxima function `diff` in `itensor`, Maxima now assumes that all indexed objects depend on any variable of differentiation unless otherwise stated. This makes it possible for the summation convention to be extended to derivative indices. It should be noted that `itensor` does not possess the capabilities of raising derivative indices, and so they are always treated as covariant.

The following functions are available in the tensor package for manipulating indexed objects. At present, with respect to the simplification routines, it is assumed that indexed objects do not by default possess symmetry properties. This can be overridden by setting the variable `allsym[false]` to `true`, which will result in treating all indexed objects completely symmetric in their lists of covariant indices and symmetric in their lists of contravariant indices.

The `itensor` package generally treats tensors as opaque objects. Tensorial equations are manipulated based on algebraic rules, specifically symmetry and contraction rules. In addition, the `itensor` package understands covariant differentiation, curvature, and torsion. Calculations can be performed relative to a metric of moving frame, depending on the setting of the `iframe_flag` variable.

A sample session below demonstrates how to load the `itensor` package, specify the name of the metric, and perform some simple calculations.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                                done
(%i3) components(g([i,j],[]),p([i,j],[])*e([],[]))$
(%i4) ishow(g([k,l],[]))$
(%t4)                               e p
                                       k l
(%i5) ishow(diff(v([i],[]),t))$
(%t5)                                  0
(%i6) depends(v,t);
(%o6)                               [v(t)]
(%i7) ishow(diff(v([i],[]),t))$
                                    d
(%t7)                               -- (v )
                                    dt   i
(%i8) ishow(idiff(v([i],[]),j))$
(%t8)                                v
                                      i,j
(%i9) ishow(extdiff(v([i],[]),j))$
(%t9)                             v    - v
                                   j,i    i,j
                                  -----------
                                       2
(%i10) ishow(liediff(v,w([i],[])))$
```

```
                               %3          %3
(%t10)                        v   w     + v   w
                                   i,%3    ,i  %3
(%i11) ishow(covdiff(v([i],[]),j))$
                                              %4
(%t11)                        v    - v   ichr2
                               i,j    %4      i j
(%i12) ishow(ev(%,ichr2))$
               %4 %5
(%t12) v    - g      v   (e p       + e   p     - e p       - e    p
        i,j           %4     j %5,i    ,i  j %5      i j,%5    ,%5  i j

                                                + e p       + e   p    )/2
                                                     i %5,j    ,j  i %5
(%i13) iframe_flag:true;
(%o13)                               true
(%i14) ishow(covdiff(v([i],[]),j))$
                                             %6
(%t14)                        v    - v   icc2
                               i,j    %6     i j
(%i15) ishow(ev(%,icc2))$
                                             %6
(%t15)                        v    - v   ifc2
                               i,j    %6     i j
(%i16) ishow(radcan(ev(%,ifc2,ifc1)))$
             %6 %8                    %6 %8
(%t16) - (ifg      v   ifb       + ifg      v   ifb       - 2 v
                    %6    j %8 i             %6    i j %8      i,j

                                                    %6 %8
                                               - ifg      v   ifb      )/2
                                                           %6    %8 i j
(%i17) ishow(canform(s([i,j],[])-s([j,i])))$
(%t17)                            s    - s
                                   i j    j i
(%i18) decsym(s,2,0,[sym(all)],[]);
(%o18)                               done
(%i19) ishow(canform(s([i,j],[])-s([j,i])))$
(%t19)                                 0
(%i20) ishow(canform(a([i,j],[])+a([j,i])))$
(%t20)                            a    + a
                                   j i    i j
(%i21) decsym(a,2,0,[anti(all)],[]);
(%o21)                               done
(%i22) ishow(canform(a([i,j],[])+a([j,i])))$
(%t22)                                 0
```

29.2 Definitions for itensor

29.2.1 Managing indexed objects

entertensor (*name*) Function

is a function which, by prompting, allows one to create an indexed object called *name* with any number of tensorial and derivative indices. Either a single index or a list of indices (which may be null) is acceptable input (see the example under `covdiff`).

changename (*old, new, expr*) Function

will change the name of all indexed objects called *old* to *new* in *expr*. *old* may be either a symbol or a list of the form `[`*name*`,` *m*`,` *n*`]` in which case only those indexed objects called *name* with *m* covariant and *n* contravariant indices will be renamed to *new*.

listoftens Function

Lists all tensors in a tensorial expression, complete with their indices. E.g.,

```
(%i6) ishow(a([i,j],[k])*b([u],[],v)+c([x,y],[])*d([],[])*e)$
                                         k
(%t6)                        d e c    + a    b
                                  x y    i j  u,v
(%i7) ishow(listoftens(%))$
                               k
(%t7)                        [a    , b    , c    , d]
                               i j    u,v    x y
```

ishow (*expr*) Function

displays *expr* with the indexed objects in it shown having their covariant indices as subscripts and contravariant indices as superscripts. The derivative indices are displayed as subscripts, separated from the covariant indices by a comma (see the examples throughout this document).

indices (*expr*) Function

Returns a list of two elements. The first is a list of the free indices in *expr* (those that occur only once). The second is the list of the dummy indices in *expr* (those that occur exactly twice) as the following example demonstrates.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i,j],[k,l],m,n)*b([k,o],[j,m,p],q,r))$
                                k l      j m p
(%t2)                          a        b
                                i j,m n  k o,q r
(%i3) indices(%);
(%o3)                  [[l, p, i, n, o, q, r], [k, j, m]]
```

A tensor product containing the same index more than twice is syntactically illegal. `indices` attempts to deal with these expressions in a reasonable manner; however, when it is called to operate upon such an illegal expression, its behavior should be considered undefined.

rename (*expr*) Function
rename (*expr, count*) Function

Returns an expression equivalent to *expr* but with the dummy indices in each term chosen from the set `[%1, %2,...]`, if the optional second argument is omitted. Otherwise, the dummy indices are indexed beginning at the value of *count*. Each dummy index in a product will be different. For a sum, `rename` will operate upon each term in the sum resetting the counter with each term. In this way `rename` can serve as a tensorial simplifier. In addition, the indices will be sorted alphanumerically (if `allsym` is `true`) with respect to covariant or contravariant indices depending upon the value of `flipflag`. If `flipflag` is `false` then the indices will be renamed according to the order of the contravariant indices. If `flipflag` is `true` the renaming will occur according to the to the order of the contravariant indices. It often happens that the combined effect of the two renamings will reduce an expression more than either one by itself.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) allsym:true;
(%o2)                                true
(%i3) g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%4],[%3])*
ichr2([%2,%3],[u])*ichr2([%5,%6],[%1])*ichr2([%7,r],[%2])-
g([],[%4,%5])*g([],[%6,%7])*ichr2([%1,%2],[u])*
ichr2([%3,%5],[%1])*ichr2([%4,%6],[%3])*ichr2([%7,r],[%2]),noeval$
(%i4) expr:ishow(%)$

       %4 %5  %6 %7      %3         u          %1         %2
(%t4) g      g      ichr2      ichr2      ichr2      ichr2
                         %1 %4      %2 %3      %5 %6      %7 r

              %4 %5  %6 %7      u          %1         %3         %2
           - g      g      ichr2      ichr2      ichr2      ichr2
                                %1 %2      %3 %5      %4 %6      %7 r
(%i5) flipflag:true;
(%o5)                                true
(%i6) ishow(rename(expr))$
       %2 %5  %6 %7      %4         u          %1         %3
(%t6) g      g      ichr2      ichr2      ichr2      ichr2
                         %1 %2      %3 %4      %5 %6      %7 r

              %4 %5  %6 %7      u          %1         %3         %2
           - g      g      ichr2      ichr2      ichr2      ichr2
                                %1 %2      %3 %4      %5 %6      %7 r
(%i7) flipflag:false;
```

```
(%o7)                                false
(%i8) rename(%th(2));
(%o8)                                  0
(%i9) ishow(rename(expr))$
       %1 %2  %3 %4      %5          %6          %7        u
(%t9) g      g      ichr2       ichr2       ichr2     ichr2
                         %1 %6       %2 %3       %4 r      %5 %7

              %1 %2  %3 %4      %6          %5          %7        u
           - g      g      ichr2       ichr2       ichr2     ichr2
                                %1 %3       %2 %6       %4 r      %5 %7
```

flipflag Option variable

Default: `false`. if `false` then the indices will be renamed according to the order of the contravariant indices, otherwise according to the order of the covariant indices. The function influences `rename` in the following way: If `flipflag` is `false` then `rename` forms a list of the contravariant indices as they are encountered from left to right (if `true` then of the covariant indices). The first dummy index in the list is renamed to `%1`, the next to `%2`, etc. Then sorting occurs after the `rename`-ing (see the example under `rename`).

defcon (*tensor_1*) Function

defcon (*tensor_1*, *tensor_2*, *tensor_3*) Function

gives *tensor_1* the property that the contraction of a product of *tensor_1* and *tensor_2* results in *tensor_3* with the appropriate indices. If only one argument, *tensor_1*, is given, then the contraction of the product of *tensor_1* with any indexed object having the appropriate indices (say `my_tensor`) will yield an indexed object with that name, i.e. `my_tensor`, and with a new set of indices reflecting the contractions performed. For example, if `imetric: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`.

remcon (*tensor_1*, ..., *tensor_n*) Function

remcon (*all*) Function

removes all the contraction properties from the *tensor_1*, ..., *tensor_n*). `remcon(all)` removes all contraction properties from all indexed objects.

contract (*expr*) Function

Carries out the tensorial contractions in *expr* which may be any combination of sums and products. This function uses the information given to the `defcon` function. For best results, `expr` 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.

indexed_tensor (*tensor*) Function

Must be executed before assigning components to a *tensor* for which a built in value already exists as with `ichr1`, `ichr2`, `icurvature`. See the example under `icurvature`.

components (*tensor*, *expr*) Function

permits one to assign an indicial value to an expression *expr* giving the values of the components of *tensor*. These are automatically substituted for the tensor whenever it occurs with all of its indices. The tensor must be of the form `t([...],[...])` where either list may be empty. *expr* can be any indexed expression involving other objects with the same free indices as *tensor*. When used to assign values to the metric tensor wherein the components contain dummy indices one must be careful to define these indices to avoid the generation of multiple dummy indices. Removal of this assignment is given to the function `remcomps`.

It is important to keep in mind that `components` cares only about the valence of a tensor, not about any particular index ordering. Thus assigning components to, say, `x([i,-j],[])`, `x([-j,i],[])`, or `x([i],[j])` all produce the same result, namely components being assigned to a tensor named `x` with valence `(1,1)`.

Components can be assigned to an indexed expression in four ways, two of which involve the use of the `components` command:

1) As an indexed expression. For instance:

```
(%i2) components(g([],[i,j]),e([],[i])*p([],[j]))$
(%i3) ishow(g([],[i,j]))$
                                          i  j
(%t3)                                    e  p
```

2) As a matrix:

```
(%i6) components(g([i,j],[]),lg);
(%o6)                                    done
(%i7) ishow(g([i,j],[]))$
(%t7)                                    g
                                          i j
(%i8) g([3,3],[]);
(%o8)                                      1
(%i9) g([4,4],[]);
(%o9)                                     - 1
```

3) As a function. You can use a Maxima function to specify the components of a tensor based on its indices. For instance, the following code assigns `kdelta` to `h` if `h` has the same number of covariant and contravariant indices and no derivative indices, and `g` otherwise:

```
(%i4) h(l1,l2,[l3]):=if length(l1)=length(l2) and length(l3)=0
  then kdelta(l1,l2) else apply(g,append([l1,l2], l3))$
(%i5) ishow(h([i],[j]))$
```

```
                                           j
(%t5)                                kdelta
                                           i

(%i6) ishow(h([i,j],[k],l))$
                                       k
(%t6)                                 g
                                       i j,l
```

4) Using Maxima's pattern matching capabilities, specifically the `defrule` and `applyb1` commands:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) matchdeclare(l1,listp);
(%o2)                                  done
(%i3) defrule(r1,m(l1,[]),(i1:idummy(),
      g([l1[1],l1[2]],[])*q([i1],[])*e([],[i1])))$

(%i4) defrule(r2,m([],l1),(i1:idummy(),
      w([],[l1[1],l1[2]])*e([i1],[])*q([],[i1])))$

(%i5) ishow(m([i,n],[])*m([],[i,m]))$
                                    i m
(%t5)                              m    m
                                         i n
(%i6) ishow(rename(applyb1(%,r1,r2)))$
                           %1  %2  %3 m
(%t6)                     e   q   w     q   e   g
                                         %1  %2  %3 n
```

remcomps (*tensor*) Function

Unbinds all values from *tensor* which were assigned with the `components` function.

showcomps Function

Shows component assignments of a tensor, as made using the `components` command. This function can be particularly useful when a matrix is assigned to an indicial tensor using `components`, as demonstrated by the following example:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) load(itensor);
(%o2)      /share/tensor/itensor.lisp
(%i3) lg:matrix([sqrt(r/(r-2*m)),0,0,0],[0,r,0,0],
                [0,0,sin(theta)*r,0],[0,0,0,sqrt((r-2*m)/r)]);
               [         r                                     ]
               [ sqrt(-------)  0       0              0       ]
```

```
                [      r - 2 m                                   ]
                [                                                ]
                [       0        r       0              0        ]
(%o3)           [                                                ]
                [       0        0  r sin(theta)        0        ]
                [                                                ]
                [                                          r - 2 m  ]
                [       0        0       0        sqrt(-------) ]
                [                                              r     ]
(%i4) components(g([i,j],[]),lg);
(%o4)                                done
(%i5) showcomps(g([i,j],[]));
                   [         r                                    ]
                   [ sqrt(-------)  0       0              0       ]
                   [      r - 2 m                                  ]
                   [                                               ]
                   [       0        r       0              0       ]
(%t5)      g     = [                                               ]
            i j    [       0        0  r sin(theta)        0       ]
                   [                                               ]
                   [                                     r - 2 m  ]
                   [       0        0       0        sqrt(-------) ]
                   [                                        r      ]
(%o5)                                false
```

The `showcomps` command can also display components of a tensor of rank higher than 2.

idummy () Function

Increments `icounter` and returns as its value an index 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 (see the example under `indices`).

idummyx Variable

Is the prefix for dummy indices (see the example under `indices`).

icounter Option 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 `idummy` (default: %).

kdelta (*L1*, *L2*) Function

is the generalized Kronecker delta function defined in the `itensor` 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(expr,kdelta)` causes the evaluation of an expression containing `kdelta([],[])` to the dimension of the manifold.

In what amounts to an abuse of this notation, `itensor` also allows `kdelta` to have 2 covariant and no contravariant, or 2 contravariant and no covariant indices, in effect providing a co(ntra)variant "unit matrix" capability. This is strictly considered a programming aid and not meant to imply that `kdelta([i,j],[])` is a valid tensorial object.

kdels (*L1*, *L2*) Function

Symmetricized Kronecker delta, used in some calculations. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) kdelta([1,2],[2,1]);
(%o2)                                    - 1
(%i3) kdels([1,2],[2,1]);
(%o3)                                     1
(%i4) ishow(kdelta([a,b],[c,d]))$
                                 c       d         d       c
(%t4)                      kdelta  kdelta  - kdelta  kdelta
                                 a       b         a       b
(%i4) ishow(kdels([a,b],[c,d]))$
                                 c       d         d       c
(%t4)                      kdelta  kdelta  + kdelta  kdelta
                                 a       b         a       b
```

levi_civita (*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.

lc2kdt (*expr*) Function

Simplifies expressions containing the Levi-Civita symbol, converting these to Kronecker-delta expressions when possible. The main difference between this function and simply evaluating the Levi-Civita symbol is that direct evaluation often results in Kronecker expressions containing numerical indices. This is often undesirable as it prevents further simplification. The `lc2kdt` function avoids this problem, yielding expressions that are more easily simplified with `rename` or `contract`.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) expr:ishow('levi_civita([],[i,j])*'levi_civita([k,l],[])*a([j],[k]))$
                                      i j  k
(%t2)                      levi_civita    a  levi_civita
                                           j            k l
(%i3) ishow(ev(expr,levi_civita))$
                                      i j  k       1 2
```

```
(%t3)                      kdelta    a  kdelta
                                 1 2  j       k l
(%i4) ishow(ev(%,kdelta))$
             i       j         j       i   k
(%t4) (kdelta  kdelta  - kdelta  kdelta ) a
             1       2         1       2   j

                               1       2         2       1
                        (kdelta  kdelta  - kdelta  kdelta )
                               k       l         k       l
(%i5) ishow(lc2kdt(expr))$
                     k       i       j    k       j       i
(%t5)               a  kdelta  kdelta  - a  kdelta  kdelta
                     j       k       l    j       k       l
(%i6) ishow(contract(expand(%)))$
                                 i           i
(%t6)                           a  - a kdelta
                                 l           l
```

The `lc2kdt` function sometimes makes use of the metric tensor. If the metric tensor was not defined previously with `imetric`, this results in an error.

```
(%i7) expr:ishow('levi_civita([],[i,j])*'levi_civita([],[k,l])*a([j,k],[]))$
                                 i j            k l
(%t7)                  levi_civita    levi_civita    a
                                                      j k
(%i8) ishow(lc2kdt(expr))$
Maxima encountered a Lisp error:

 Error in $IMETRIC [or a callee]:
 $IMETRIC [or a callee] requires less than two arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i9) imetric(g);
(%o9)                                done
(%i10) ishow(lc2kdt(expr))$
        %3 i       k   %4 j       l     %3 i       l   %4 j       k
(%t10) (g     kdelta   g     kdelta   - g     kdelta   g     kdelta  ) a
                   %3              %4               %3              %4   j k
(%i11) ishow(contract(expand(%)))$
                                  l i      l i
(%t11)                           a    - a g
```

lc_l Function

Simplification rule used for expressions containing the unevaluated Levi-Civita symbol (`levi_civita`). Along with `lc_u`, it can be used to simplify many expressions more efficiently than the evaluation of `levi_civita`. For example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2)  el1:ishow('levi_civita([i,j,k],[])*a([],[i])*a([],[j]))$
                                  i   j
(%t2)                            a   a  levi_civita
                                                 i j k
(%i3) el2:ishow('levi_civita([],[i,j,k])*a([i])*a([j]))$
                                        i j k
(%t3)                            levi_civita      a  a
                                                   i   j
(%i4) ishow(canform(contract(expand(applyb1(el1,lc_l,lc_u)))))$
(%t4)                                   0
(%i5) ishow(canform(contract(expand(applyb1(el2,lc_l,lc_u)))))$
(%t5)                                   0
```

lc_u Function

Simplification rule used for expressions containing the unevaluated Levi-Civita symbol (`levi_civita`). Along with `lc_u`, it can be used to simplify many expressions more efficiently than the evaluation of `levi_civita`. For details, see `lc_l`.

canten (*expr*) Function

Simplifies *expr* by renaming (see `rename`) and permuting dummy indices. `rename` 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.

The `canten` function returns a mathematically correct result only if its argument is an expression that is fully symmetric in its indices. For this reason, `canten` returns an error if `allsym` is not set to `true`.

concan (*expr*) Function

Similar to `canten` but also performs index contraction.

29.2.2 Tensor symmetries

allsym Option variable

Default: `false`. 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 unless `iframe_flag` is set to `true`.

decsym (*tensor, m, n,* [*cov_1, cov_2, ...*], [*contr_1, contr_2, ...*]) Function

Declares symmetry properties for *tensor* of *m* covariant and *n* contravariant indices. The *cov_i* and *contr_i* are pseudofunctions expressing symmetry relations among the covariant and contravariant indices respectively. These are of the form `symoper(`*index_1*`, `*index_2*`,...)` where `symoper` is one of `sym`, `anti` or `cyc` and the *index_i* are integers indicating the position of the index in the *tensor*. This will declare *tensor* to be symmetric, antisymmetric or cyclic respectively in the *index_i*. `symoper(all)` is also an allowable form which indicates all indices obey the symmetry condition. For example, given an object `b` with 5 covariant indices, `decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)])` declares `b` symmetric in its first and second and antisymmetric in its third and fourth covariant indices, and cyclic in all of its contravariant indices. Either list of symmetry declarations may be null. The function which performs the simplifications is `canform` as the example below illustrates.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) expr:contract(expand(a([i1,j1,k1],[])*kdels([i,j,k],[i1,j1,k1])))$
(%i3) ishow(expr)$
(%t3)        a      + a      + a      + a      + a      + a
              k j i    k i j    j k i    j i k    i k j    i j k
(%i4) decsym(a,3,0,[sym(all)],[]);
(%o4)                                done
(%i5) ishow(canform(expr))$
(%t5)                              6 a
                                      i j k
(%i6) remsym(a,3,0);
(%o6)                                done
(%i7) decsym(a,3,0,[anti(all)],[]);
(%o7)                                done
(%i8) ishow(canform(expr))$
(%t8)                                  0
(%i9) remsym(a,3,0);
(%o9)                                done
(%i10) decsym(a,3,0,[cyc(all)],[]);
(%o10)                               done
(%i11) ishow(canform(expr))$
(%t11)                        3 a      + 3 a
                                 i k j      i j k
(%i12) dispsym(a,3,0);
(%o12)                     [[cyc, [[1, 2, 3]], []]]
```

remsym (*tensor*, *m*, *n*) Function

Removes all symmetry properties from *tensor* which has *m* covariant indices and *n* contravariant indices.

canform (*expr*) Function

Simplifies *expr* 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.

29.2.3 Indicial tensor calculus

diff (*expr, v_1,* [*n_1,* [*v_2, n_2*] ...]) Function

is the usual Maxima differentiation function which has been expanded in its abilities for `itensor`. It takes the derivative of *expr* with respect to *v_1 n_1* times, with respect to *v_2 n_2* times, etc. For the tensor package, the function has been modified so that the *v_i* may be integers from 1 up to the value of the variable `dim`. This will cause the differentiation to be carried out with respect to the *v_i*th member of the list `vect_coords`. If `vect_coords` is bound to an atomic variable, then that variable subscripted by *v_i* 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.

idiff (*expr, v_1,* [*n_1,* [*v_2, n_2*] ...]) Function

Indicial differentiation. Unlike `diff`, which differentiates with respect to an independent variable, `idiff)` can be used to differentiate with respect to a coordinate. For an indexed object, this amounts to appending the *v_i* as derivative indices. Subsequently, derivative indices will be sorted, unless `iframe_flag` is set to `true`.

`idiff` can also differentiate the determinant of the metric tensor. Thus, if `imetric` has been bound to `G` then `idiff(determinant(g),k)` will return `2*determinant(g)*ichr2([%i,k],[%i])` where the dummy index `%i` is chosen appropriately.

liediff (*v, ten*) Function

Computes the Lie-derivative of the tensorial expression *ten* with respect to the vector field *v*. *ten* should be any indexed tensor expression; *v* should be the name (without indices) of a vector field. For example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(liediff(v,a([i,j],[])*b([],[k],l)))$
       k    %2            %2          %2
(%t2) b   (v   a       + v   a     + v   a    )
       ,l       i j,%2    ,j  i %2    ,i  %2 j

                                %1  k         %1  k       %1  k
                             + (v   b      - b   v     + v   b   ) a
                                     ,%1 l     ,l  ,%1     ,l  ,%1   i j
```

rediff (*ten*) Function

Evaluates all occurrences of the `idiff` command in the tensorial expression *ten*.

undiff (*expr*) Function

Returns an expression equivalent to *expr* but with all derivatives of indexed objects replaced by the noun form of the `idiff` function. Its arguments 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 resulting in *expr* and then carry out the differentiation by saying `ev(`*expr*`, idiff)`.

evundiff Function

Equivalent to the execution of `undiff`, followed by `ev` and `rediff`.

The point of this operation is to easily evalute expressions that cannot be directly evaluated in derivative form. For instance, the following causes an error:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) icurvature([i,j,k],[l],m);
Maxima encountered a Lisp error:

 Error in $ICURVATURE [or a callee]:
 $ICURVATURE [or a callee] requires less than three arguments.

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
```

However, if `icurvature` is entered in noun form, it can be evaluated using `evundiff`:

```
(%i3) ishow('icurvature([i,j,k],[l],m))$
                                         l
(%t3)                          icurvature
                                         i j k,m
(%i4) ishow(evundiff(%))$
             l              l         %1           l           %1
(%t4) - ichr2        - ichr2     ichr2      - ichr2       ichr2
             i k,j m        %1 j      i k,m        %1 j,m      i k

                l              l         %1           l           %1
         + ichr2        + ichr2     ichr2      + ichr2       ichr2
                i j,k m        %1 k      i j,m        %1 k,m      i j
```

Note: In earlier versions of Maxima, derivative forms of the Christoffel-symbols also could not be evaluated. This has been fixed now, so `evundiff` is no longer necessary for expressions like this:

```
(%i5) imetric(g);
(%o5)                               done
(%i6) ishow(ichr2([i,j],[k],l))$
       k %3
      g     (g         - g         + g        )
              j %3,i l    i j,%3 l    i %3,j l
(%t6) -----------------------------------------
                          2

                         k %3
```

```
                                     g     (g       - g       + g       )
                                      ,l     j %3,i    i j,%3    i %3,j
                                   + ----------------------------------
                                                     2
```

flush (*expr, tensor_1, tensor_2, ...*) Function

Set to zero, in *expr*, all occurrences of the *tensor_i* that have no derivative indices.

flushd (*expr, tensor_1, tensor_2, ...*) Function

Set to zero, in *expr*, all occurrences of the *tensor_i* that have derivative indices.

flushnd (*expr, tensor, n*) Function

Set to zero, in *expr*, all occurrences of the differentiated object *tensor* that have *n* or more derivative indices as the following example demonstrates.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i],[J,r],k,r)+a([i],[j,r,s],k,r,s))$
                                J r      j r s
(%t2)                          a      + a
                                i,k r    i,k r s
(%i3) ishow(flushnd(%,a,3))$
                                     J r
(%t3)                               a
                                     i,k r
```

coord (*tensor_1, tensor_2, ...*) Function

Gives *tensor_i* the coordinate differentiation property that the derivative of contravariant vector whose name is one of the *tensor_i* yields a Kronecker delta. For example, if `coord(x)` has been done then `idiff(x([],[i]),j)` gives `kdelta([i],[j])`. `coord` is a list of all indexed objects having this property.

remcoord (*tensor_1, tensor_2, ...*) Function
remcoord (*all*) Function

Removes the coordinate differentiation property from the `tensor_i` that was established by the function `coord`. `remcoord(all)` removes this property from all indexed objects.

makebox (*expr*) Function

Display *expr* in the same manner as `show`; however, any tensor d'Alembertian occurring in *expr* will be indicated using the symbol `[]`. For example, `[]p([m],[n])` represents `g([],[i,j])*p([m],[n],i,j)`.

conmetderiv (*expr, tensor*) Function

Simplifies expressions containing ordinary derivatives of both covariant and contravariant forms of the metric tensor (the current restriction). For example, `conmetderiv` can relate the derivative of the contravariant metric tensor with the Christoffel symbols as seen from the following:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(g([],[a,b],c))$
                                        a b
(%t2)                                  g
                                        ,c
(%i3) ishow(conmetderiv(%,g))$
                         %1 b      a       %1 a      b
(%t3)                 - g     ichr2     - g     ichr2
                                   %1 c              %1 c
```

simpmetderiv (*expr*[, *stop*]) Function

Simplifies expressions containing products of the derivatives of the metric tensor. Specifically, `simpmetderiv` recognizes two identities:

```
   ab        ab           ab                 a
  g   g   + g   g     = (g   g  )   = (kdelta )   = 0
   ,d  bc        bc,d         bc ,d          c ,d
```

hence

```
   ab          ab
  g   g   = - g   g
   ,d  bc          bc,d
```

and

```
   ab          ab
  g   g     = g   g
   ,j  ab,i    ,i  ab,j
```

which follows from the symmetries of the Christoffel symbols.

The `simpmetderiv` function takes one optional parameter which, when present, causes the function to stop after the first successful substitution in a product expression. The `simpmetderiv` function also makes use of the global variable *flipflag* which determines how to apply a "canonical" ordering to the product indices.

Put together, these capabilities can be used to achieve powerful simplifications that are difficult or impossible to accomplish otherwise. This is demonstrated through the following example that explicitly uses the partial simplification features of `simpmetderiv` to obtain a contractible expression:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                                  done
(%i3) ishow(g([],[a,b])*g([],[b,c])*g([a,b],[],d)*g([b,c],[],e))$
```

```
                                 a b  b c
(%t3)                          g    g    g      g
                                             a b,d  b c,e
(%i4) ishow(canform(%))$

errexp1 has improper indices
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) ishow(simpmetderiv(%))$
                                 a b  b c
(%t5)                          g    g    g      g
                                             a b,d  b c,e
(%i6) flipflag:not flipflag;
(%o6)                                   true
(%i7) ishow(simpmetderiv(%th(2)))$
                                   a b  b c
(%t7)                            g    g    g    g
                                   ,d   ,e   a b  b c
(%i8) flipflag:not flipflag;
(%o8)                                   false
(%i9) ishow(simpmetderiv(%th(2),stop))$
                                   a b  b c
(%t9)                          - g    g    g      g
                                        ,e   a b,d  b c
(%i10) ishow(contract(%))$
                                        b c
(%t10)                             - g    g
                                        ,e   c b,d
```

See also `weyl.dem` for an example that uses `simpmetderiv` and `conmetderiv` together to simplify contractions of the Weyl tensor.

flush1deriv (*expr, tensor*) Function

Set to zero, in `expr`, all occurrences of `tensor` that have exactly one derivative index.

29.2.4 Tensors in curved spaces

imetric (*g*) Function

Specifies the metric by assigning the variable `imetric:`*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 `imetric`, default: [], is bound to the metric, assigned by the `imetric(`*g*`)` command.

idim (*n*) Function

Sets the dimensions of the metric. Also initializes the antisymmetry properties of the Levi-Civita symbols for the given dimension.

ichr1 ([*i, j, k*]) Function

Yields the Christoffel symbol of the first kind via the definition

```
(g      + g      - g     )/2 .
  ik,j     jk,i     ij,k
```

To evaluate the Christoffel symbols for a particular metric, the variable `imetric` must be assigned a name as in the example under `chr2`.

ichr2 ([*i*, *j*], [*k*]) Function

Yields the Christoffel symbol of the second kind defined by the relation

```
                       ks
    ichr2([i,j],[k]) = g    (g      + g      - g     )/2
                              is,j     js,i     ij,s
```

icurvature ([*i*, *j*, *k*], [*h*]) Function

Yields the Riemann curvature tensor in terms of the Christoffel symbols of the second kind (`ichr2`). The following notation is used:

```
            h             h            h         %1         h
  icurvature     = - ichr2      - ichr2     ichr2    + ichr2
            i j k         i k,j        %1 j      i k        i j,k
                            h          %1
                   + ichr2      ichr2
                          %1 k       i j
```

covdiff (*expr*, *v_1*, *v_2*, ...) Function

Yields the covariant derivative of *expr* with respect to the variables *v_i* in terms of the Christoffel symbols of the second kind (`ichr2`). In order to evaluate these, one should use `ev(`*expr*`,ichr2)`.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) entertensor()$
Enter tensor name: a;
Enter a list of the covariant indices: [i,j];
Enter a list of the contravariant indices: [k];
Enter a list of the derivative indices: [];
                                      k
(%t2)                                a
                                      i j
(%i3) ishow(covdiff(%,s))$
             k         %1     k         %1     k            k     %1
(%t3)     - a     ichr2    - a     ichr2    + a      + ichr2     a
             i %1      j s    %1 j      i s    i j,s        %1 s  i j
```

lorentz_gauge (*expr*) Function

Imposes the Lorentz condition by substituting 0 for all indexed objects in *expr* that have a derivative index identical to a contravariant index.

igeodesic_coords (*expr*, *name*) Function

Causes undifferentiated Christoffel symbols and first derivatives of the metric tensor vanish in *expr*. The *name* in the `igeodesic_coords` function refers to the metric *name* (if it appears in *expr*) while the connection coefficients must be called with the names `ichr1` and/or `ichr2`. The following example demonstrates the verification of the cyclic identity satisfied by the Riemann curvature tensor using the `igeodesic_coords` function.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(icurvature([r,s,t],[u]))$
              u            u         %1         u            u         %1
(%t2) - ichr2      - ichr2     ichr2    + ichr2      + ichr2     ichr2
              r t,s        %1 s      r t        r s,t        %1 t      r s
(%i3) ishow(igeodesic_coords(%,ichr2))$
                                  u            u
(%t3)                        ichr2      - ichr2
                                  r s,t        r t,s
(%i4) ishow(igeodesic_coords(icurvature([r,s,t],[u]),ichr2)+
            igeodesic_coords(icurvature([s,t,r],[u]),ichr2)+
            igeodesic_coords(icurvature([t,r,s],[u]),ichr2))$
              u            u            u            u            u
(%t4) - ichr2      + ichr2      + ichr2      - ichr2      - ichr2
              t s,r        t r,s        s t,r        s r,t        r t,s

                                                                  u
                                                           + ichr2
                                                                  r s,t
(%i5) canform(%);
(%o5)                              0
```

29.2.5 Moving frames

Maxima now has the ability to perform calculations using moving frames. These can be orthonormal frames (tetrads, vielbeins) or an arbitrary frame.

To use frames, you must first set `iframe_flag` to `true`. This causes the Christoffel-symbols, `ichr1` and `ichr2`, to be replaced by the more general frame connection coefficients `icc1` and `icc2` in calculations. Speficially, the behavior of `covdiff` and `icurvature` is changed.

The frame is defined by two tensors: the inverse frame field (`ifri`), and the frame metric `ifg`. The frame metric is the identity matrix for orthonormal frames, or the Lorentz metric for orthonormal frames in Minkowski spacetime. The inverse frame field defines the frame base (unit vectors). Contraction properties are defined for the frame field and the frame metric.

When `iframe_flag` is true, many `itensor` expressions use the frame metric `ifg` instead of the metric defined by `imetric` for raising and lowerind indices.

IMPORTANT: Setting the variable `iframe_flag` to `true` does NOT undefine the contraction properties of a metric defined by a call to `defcon` or `imetric`. If a frame field is used, it is best to define the metric by assigning its name to the variable `imetric` and NOT invoke the `imetric` function.

Maxima uses these two tensors to define the frame coefficients (`ifc1` and `ifc2`) which form part of the connection coefficients (`icc1` and `icc2`), as the following example demonstrates:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) iframe_flag:true;
(%o2)                                true
(%i3) ishow(covdiff(v([],[i]),j))$
                                i        i     %1
(%t3)                          v   + icc2     v
                                ,j       %1 j
(%i4) ishow(ev(%,icc2))$
                        %1      i           i        i
(%t4)                  v   (ifc2     + ichr2    ) + v
                                %1 j        %1 j     ,j
(%i5) ishow(ev(%,ifc2))$
            %1    i %2
           v   ifg     (ifb         - ifb         + ifb        )
                           j %2 %1      %2 %1 j      %1 j %2     i
(%t5)      -------------------------------------------------- + v
                                   2                             ,j
(%i6) ishow(ifb([a,b,c]))$
                       %5    %4
(%t6)               ifr   ifr   (ifri        - ifri        )
                       a     b       c %4,%5       c %5,%4
```

An alternate method is used to compute the frame bracket (`ifb`) if the `iframe_bracket_form` flag is set to `false`:

```
(%i8) block([iframe_bracket_form:false],ishow(ifb([a,b,c])))$
                       %7    %6        %6      %7
(%t8)              (ifr   ifr     - ifr     ifr  ) ifri
                       a     b,%7      a,%7    b       c %6
```

iframes () Function

Since in this version of Maxima, contraction identities for `ifr` and `ifri` are always defined, as is the frame bracket (`ifb`), this function does nothing.

ifb Variable

The frame bracket. The contribution of the frame metric to the connection coefficients is expressed using the frame bracket:

$$ifc1_{abc} = \frac{- ifb_{c a b} + ifb_{b c a} + ifb_{a b c}}{2}$$

The frame bracket itself is defined in terms of the frame field and frame metric. Two alternate methods of computation are used depending on the value of `frame_bracket_form`. If true (the default) or if the `itorsion_flag` is `true`:

$$ifb_{abc} = ifr_{b}^{d} \, ifr_{c}^{e} \, (ifri_{a\ d,e} - ifri_{a\ e,d} - ifri_{a\ f} \, itr_{d\ e}^{f})$$

Otherwise:

$$ifb_{abc} = (ifr_{b}^{e} \, ifr_{c,e}^{d} - ifr_{b,e}^{d} \, ifr_{c}^{e}) \, ifri_{a\ d}$$

icc1 Variable

Connection coefficients of the first kind. In `itensor`, defined as

$$icc1_{abc} = ichr1_{abc} - ikt1_{abc} - inmc1_{abc}$$

In this expression, if `iframe_flag` is true, the Christoffel-symbol `ichr1` is replaced with the frame connection coefficient `ifc1`. If `itorsion_flag` is `false`, `ikt1` will be omitted. It is also omitted if a frame base is used, as the torsion is already calculated as part of the frame bracket. Lastly, of `inonmet_flag` is `false`, `inmc1` will not be present.

icc2 Variable

Connection coefficients of the second kind. In `itensor`, defined as

$$icc2_{ab}^{c} = ichr2_{ab}^{c} - ikt2_{ab}^{c} - inmc2_{ab}^{c}$$

In this expression, if `iframe_flag` is true, the Christoffel-symbol `ichr2` is replaced with the frame connection coefficient `ifc2`. If `itorsion_flag` is `false`, `ikt2` will be omitted. It is also omitted if a frame base is used, as the torsion is already calculated as part of the frame bracket. Lastly, of `inonmet_flag` is `false`, `inmc2` will not be present.

ifc1 Variable

Frame coefficient of the first kind (also known as Ricci-rotation coefficients.) This tensor represents the contribution of the frame metric to the connection coefficient of the first kind. Defined as:

```
          - ifb      + ifb      + ifb
               c a b      b c a      a b c
ifc1    = -------------------------------
    abc                  2
```

ifc2 Variable

Frame coefficient of the first kind. This tensor represents the contribution of the frame metric to the connection coefficient of the first kind. Defined as a permutation of the frame bracket (`ifb`) with the appropriate indices raised and lowered as necessary:

```
    c       cd
ifc2   = ifg   ifc1
    ab             abd
```

ifr Variable

The frame field. Contracts with the inverse frame field (`ifri`) to form the frame metric (`ifg`).

ifri Variable

The inverse frame field. Specifies the frame base (basis vectors). Along with the frame metric, it forms the basis of all calculations based on frames.

ifg Variable

The frame metric. Defaults to `kdelta`, but can be changed using `components`.

ifgi Variable

The inverse frame metric. Contracts with the frame metric (`ifg`) to `kdelta`.

iframe_bracket_form Option variable

Specifies how the frame bracket (`ifb`) is computed. Default is `true`.

29.2.6 Torsion and nonmetricity

Maxima can now take into account torsion and nonmetricity. When the flag `itorsion_flag` is set to `true`, the contribution of torsion is added to the connection coefficients. Similarly, when the flag `inonmet_flag` is true, nonmetricity components are included.

inm Variable

The nonmetricity vector. Conformal nonmetricity is defined through the covariant derivative of the metric tensor. Normally zero, the metric tensor's covariant derivative will evaluate to the following when `inonmet_flag` is set to `true`:

$$g_{ij;k} = -\, g_{ij}\ nm_{k}$$

inmc1 Variable

Covariant permutation of the nonmetricity vector components. Defined as

$$inmc1_{abc} = \frac{g_{ab}\ inm_{c} - inm_{a}\ g_{bc} - g_{ac}\ inm_{b}}{2}$$

(Substitute `ifg` in place of `g` if a frame metric is used.)

inmc2 Variable

Contravariant permutation of the nonmetricity vector components. Used in the connection coefficients if `inonmet_flag` is `true`. Defined as:

$$inmc2^{c}_{ab} = \frac{-inm_{a}\ kdelta^{c}_{b} - kdelta^{c}_{a}\ inm_{b} + g^{cd}\ inm_{d}\ g_{ab}}{2}$$

(Substitute `ifg` in place of `g` if a frame metric is used.)

ikt1 Variable

Covariant permutation of the torsion tensor (also known as contorsion). Defined as:

$$ikt1_{abc} = \frac{-g_{ad}\ itr^{d}_{cb} - g_{bd}\ itr^{d}_{ca} - itr^{d}_{ab}\ g_{cd}}{2}$$

(Substitute `ifg` in place of `g` if a frame metric is used.)

ikt2 Variable

Contravariant permutation of the torsion tensor (also known as contorsion). Defined as:

```
         c     cd
    ikt2   = g   ikt1
        ab          abd
```

(Substitute `ifg` in place of `g` if a frame metric is used.)

itr Variable

The torsion tensor. For a metric with torsion, repeated covariant differentiation on a scalar function will not commute, as demonstrated by the following example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric:g;
(%o2)                                   g
(%i3) covdiff(covdiff(f([],[]),i),j)-covdiff(covdiff(f([],[]),j),i)$
(%i4) ishow(%)$
                                   %4              %2
(%t4)                    f    ichr2    - f    ichr2
                          ,%4      j i    ,%2      i j
(%i5) canform(%);
(%o5)                                   0
(%i6) itorsion_flag:true;
(%o6)                                 true
(%i7) covdiff(covdiff(f([],[]),i),j)-covdiff(covdiff(f([],[]),j),i)$
(%i8) ishow(%)$
                           %8             %6
(%t8)             f    icc2    - f    icc2    - f     + f
                   ,%8     j i    ,%6     i j    ,j i    ,i j
(%i9) ishow(canform(%))$
                                   %1             %1
(%t9)                     f    icc2    - f    icc2
                           ,%1     j i    ,%1     i j
(%i10) ishow(canform(ev(%,icc2)))$
                                   %1             %1
(%t10)                    f    ikt2    - f    ikt2
                           ,%1     i j    ,%1     j i
(%i11) ishow(canform(ev(%,ikt2)))$
                      %2 %1                    %2 %1
(%t11)          f    g      ikt1       - f    g      ikt1
                 ,%2            i j %1    ,%2            j i %1
(%i12) ishow(factor(canform(rename(expand(ev(%,ikt1))))))$
                           %3 %2            %1       %1
                     f    g      g      (itr    - itr   )
                      ,%3         %2 %1     j i      i j
(%t12)               ------------------------------------
                                          2
(%i13) decsym(itr,2,1,[anti(all)],[]);
(%o13)                                done
```

```
(%i14) defcon(g,g,kdelta);
(%o14)                               done
(%i15) subst(g,nounify(g),%th(3))$
(%i16) ishow(canform(contract(%)))$
                                             %1
(%t16)                           - f    itr
                                    ,%1    i j
```

29.2.7 Exterior algebra

The `itensor` package can perform operations on totally antisymmetric covariant tensor fields. A totally antisymmetric tensor field of rank (0,L) corresponds with a differential L-form. On these objects, a multiplication operation known as the exterior product, or wedge product, is defined.

Unfortunately, not all authors agree on the definition of the wedge product. Some authors prefer a definition that corresponds with the notion of antisymmetrization: in these works, the wedge product of two vector fields, for instance, would be defined as

```
             a a  - a a
              i j    j i
 a  /\ a  = -----------
  i     j        2
```

More generally, the product of a p-form and a q-form would be defined as

```
                      1     k1..kp l1..lq
A       /\ B       = ------ D              A       B
 i1..ip     j1..jq   (p+q)!  i1..ip j1..jq  k1..kp  l1..lq
```

where `D` stands for the Kronecker-delta.

Other authors, however, prefer a "geometric" definition that corresponds with the notion of the volume element:

```
a  /\ a  = a a  - a a
 i     j    i j    j i
```

and, in the general case

```
                      1    k1..kp l1..lq
A       /\ B       = ----- D              A       B
 i1..ip     j1..jq   p! q!  i1..ip j1..jq  k1..kp  l1..lq
```

Since `itensor` is a tensor algebra package, the first of these two definitions appears to be the more natural one. Many applications, however, utilize the second definition. To resolve this dilemma, a flag has been implemented that controls the behavior of the wedge product: if `igeowedge_flag` is `false` (the default), the first, `"tensorial"` definition is used, otherwise the second, `"geometric"` definition will be applied.

`"~"` Operator

The wedge product operator is denoted by the tilde `~`. This is a binary operator. Its arguments should be expressions involving scalars, covariant tensors of rank one, or covariant tensors of rank `l` that have been declared antisymmetric in all covariant indices.

The behavior of the wedge product operator is controlled by the `igeowedge_flag` flag, as in the following example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(a([i])~b([j]))$
                                 a  b  - b  a
                                  i  j    i  j
(%t2)                            -------------
                                       2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                done
(%i4) ishow(a([i,j])~b([k]))$
                          a    b  + b  a    - a    b
                           i j  k    i  j k    i k  j
(%t4)                     ---------------------------
                                       3
(%i5) igeowedge_flag:true;
(%o5)                                true
(%i6) ishow(a([i])~b([j]))$
(%t6)                            a  b  - b  a
                                  i  j    i  j
(%i7) ishow(a([i,j])~b([k]))$
(%t7)                     a    b  + b  a    - a    b
                           i j  k    i  j k    i k  j
```

`"|"` Operator

The vertical bar `|` denotes the "contraction with a vector" binary operation. When a totally antisymmetric covariant tensor is contracted with a contravariant vector, the result is the same regardless which index was used for the contraction. Thus, it is possible to define the contraction operation in an index-free manner.

In the `itensor` package, contraction with a vector is always carried out with respect to the first index in the literal sorting order. This ensures better simplification of expressions involving the `|` operator. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) decsym(a,2,0,[anti(all)],[]);
(%o2)                                done
(%i3) ishow(a([i,j],[])|v)$
                                    %1
(%t3)                              v   a
                                        %1 j
(%i4) ishow(a([j,i],[])|v)$
                                     %1
(%t4)                            - v   a
                                         %1 j
```

Note that it is essential that the tensors used with the `|` operator be declared totally antisymmetric in their covariant indices. Otherwise, the results will be incorrect.

extdiff (*expr*, *i*) Function

Computes the exterior derivative of *expr* with respect to the index *i*. The exterior derivative is formally defined as the wedge product of the partial derivative operator and a differential form. As such, this operation is also controlled by the setting of `igeowedge_flag`. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) ishow(extdiff(v([i]),j))$
                                  v    - v
                                   j,i    i,j
(%t2)                             -----------
                                       2
(%i3) decsym(a,2,0,[anti(all)],[]);
(%o3)                                  done
(%i4) ishow(extdiff(a([i,j]),k))$
                           a      - a      + a
                            j k,i    i k,j    i j,k
(%t4)                      ------------------------
                                      3
(%i5) igeowedge_flag:true;
(%o5)                                  true
(%i6) ishow(extdiff(v([i]),j))$
(%t6)                             v    - v
                                   j,i    i,j
(%i7) ishow(extdiff(a([i,j]),k))$
(%t7)                      a      - a      + a
                            j k,i    i k,j    i j,k
```

hodge (*expr*) Function

Compute the Hodge-dual of *expr*. For instance:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) imetric(g);
(%o2)                            done
(%i3) idim(4);
(%o3)                            done
(%i4) icounter:100;
(%o4)                             100
(%i5) decsym(A,3,0,[anti(all)],[])$

(%i6) ishow(A([i,j,k],[]))$
(%t6)                           A
                                 i j k
(%i7) ishow(canform(hodge(%)))$
                          %1 %2 %3 %4
               levi_civita            g        A
                                       %1 %102  %2 %3 %4
```

```
(%t7)          ----------------------------------------
                                   6
(%i8) ishow(canform(hodge(%)))$
                %1 %2 %3 %8            %4 %5 %6 %7
(%t8) levi_civita            levi_civita            g        g
                                                     %1 %106  %2 %107
                                                g        g      A       /6
                                                 %3 %108  %4 %8  %5 %6 %7
(%i9) lc2kdt(%)$

(%i10) %,kdelta$

(%i11) ishow(canform(contract(expand(%))))$
(%t11)                    - A
                             %106 %107 %108
```

igeowedge_flag Option variable

Controls the behavior of the wedge product and exterior derivative. When set to `false` (the default), the notion of differential forms will correspond with that of a totally antisymmetric covariant tensor field. When set to `true`, differential forms will agree with the notion of the volume element.

29.2.8 Exporting TeX expressions

The `itensor` package provides limited support for exporting tensor expressions to TeX. Since `itensor` expressions appear as function calls, the regular Maxima `tex` command will not produce the expected output. You can try instead the `tentex` command, which attempts to translate tensor expressions into appropriately indexed TeX objects.

tentex (*expr*) Function

To use the `tentex` function, you must first load `tentex`, as in the following example:

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) load(tentex);
(%o2)       /share/tensor/tentex.lisp
(%i3) idummyx:m;
(%o3)                                m
(%i4) ishow(icurvature([j,k,l],[i]))$
           m1       i           m1       i           i           i
(%t4)  ichr2    ichr2     - ichr2    ichr2     - ichr2      + ichr2
            j k      m1 l        j l      m1 k        j l,k        j k,l
(%i5) tentex(%)$
$$\Gamma_{j\,k}^{m_1}\,\Gamma_{l\,m_1}^{i}-\Gamma_{j\,l}^{m_1}\,
 \Gamma_{k\,m_1}^{i}-\Gamma_{j\,l,k}^{i}+\Gamma_{j\,k,l}^{i}$$
```

Note the use of the `idummyx` assignment, to avoid the appearance of the percent sign in the TeX expression, which may lead to compile errors.

NB: This version of the `tentex` function is somewhat experimental.

29.2.9 Interfacing with ctensor

The `itensor` package has the ability to generate Maxima code that can then be executed in the context of the `ctensor` package. The function that performs this task is `ic_convert`.

ic_convert (*eqn*) Function

Converts the `itensor` equation *eqn* to a `ctensor` assignment statement. Implied sums over dummy indices are made explicit while indexed objects are transformed into arrays (the array subscripts are in the order of covariant followed by contravariant indices of the indexed objects). The derivative of an indexed object will be replaced by the noun form of `diff` taken with respect to `ct_coords` subscripted by the derivative index. The Christoffel symbols `ichr1` and `ichr2` will be translated to `lcs` and `mcs`, respectively and if `metricconvert` is `true` then all occurrences of the metric with two covariant (contravariant) indices will be renamed to `lg` (`ug`). In addition, `do` loops will be introduced summing over all free indices so that the transformed assignment statement can be evaluated by just doing `ev`. The following examples demonstrate the features of this function.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) eqn:ishow(t([i,j],[k])=f([],[])*g([l,m],[])*a([],[m],j)*b([i],[l,k]))$
                             k        m   l k
(%t2)                       t    = f a   b    g
                             i j      ,j  i    l m
(%i3) ic_convert(eqn);
(%o3) for i thru dim do (for j thru dim

do (for k thru dim do t        : f sum(sum(diff(a , ct_coords ) b
                       i, j, k                   m           j   i, l, k

 g    , l, 1, dim), m, 1, dim)))
  l, m
(%i4) imetric(g);
(%o4)                                done
(%i5) metricconvert:true;
(%o5)                                true
(%i6) ic_convert(eqn);
(%o6) for i thru dim do (for j thru dim

do (for k thru dim do t        : f sum(sum(diff(a , ct_coords ) b
                       i, j, k                   m           j   i, l, k

 lg    , l, 1, dim), m, 1, dim)))
   l, m
```

29.2.10 Reserved words

The following Maxima words are used by the `itensor` package internally and should not be redefined:

```
Keyword    Comments
------------------------------------------
indices2() Internal version of indices()
conti      Lists contravariant indices
covi       Lists covariant indices of a indexed object
deri       Lists derivative indices of an indexed object
name       Returns the name of an indexed object
concan
irpmon
lc0
_lc2kdt0
_lcprod
_extlc
```

30 ctensor

30.1 Introduction to ctensor

ctensor is a component tensor manipulation package. To use the ctensor package, type load(ctensor). To begin an interactive session with ctensor, type csetup(). You are 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 ct_coords (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 share/tensor/metrics.mac. 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 ctensor will be displayed for the standard metric as examples.

```
(%i1) load(ctensor);
(%o1)      /usr/local/lib/maxima/share/tensor/ctensor.mac
(%i2) csetup();
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?
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  ]
                          [                        ]
                          [         2    2         ]
                          [ 0  0   x  sin (y)   0  ]
                          [                        ]
                          [ 0  0       0       - d ]
(%o2)                                done
(%i3) christof(mcs);
                                            a
                                             x
(%t3)                          mcs        = ---
                                  1, 1, 1   2 a

                                             1
(%t4)                           mcs        = -
                                   1, 2, 2   x

                                             1
(%t5)                           mcs        = -
                                   1, 3, 3   x


                                            d
                                             x
(%t6)                          mcs        = ---
                                  1, 4, 4   2 d

                                              x
(%t7)                          mcs        = - -
                                  2, 2, 1     a

                                           cos(y)
(%t8)                         mcs        = ------
                                 2, 3, 3   sin(y)

                                               2
                                           x sin (y)
(%t9)                      mcs        = - ---------
                              3, 3, 1          a

(%t10)                   mcs        = - cos(y) sin(y)
                            3, 3, 2
```

```
                                              d
                                               x
(%t11)                        mcs          = ---
                                 4, 4, 1     2 a
(%o11)                              done
```

30.2 Definitions for ctensor

30.2.1 Initialization and setup

csetup () Function

A function in the `ctensor` (component tensor) package which initializes the package and allows the user to enter a metric interactively. See `ctensor` for more details.

cmetric (*dis*) Function
cmetric () Function

A function in the `ctensor` (component tensor) package that computes the metric inverse and sets up the package for further calculations.

If `cframe_flag` is false, the function computes the inverse metric ug from the (user-defined) matrix `lg`. The metric determinant is also computed and stored in the variable `gdet`. Furthermore, the package determines if the metric is diagonal and sets the value of `diagmetric` accordingly. If the optional argument *dis* is present and not equal to false, the user is prompted to see the metric inverse.

If `cframe_flag` is true, the function expects that the values of `fri` (the inverse frame matrix) and `lfg` (the frame metric) are defined. From these, the frame matrix `fr` and the inverse frame metric `ufg` are computed.

ct_coordsys (*coordinate_system, extra_arg*) Function
ct_coordsys (*coordinate_system*) Function

Sets up a predefined coordinate system and metric. The argument *coordinate_system* can be one of the following symbols:

```
SYMBOL             Dim Coordinates       Description/comments
--------------------------------------------------------------------
cartesian2d         2  [x,y]             Cartesian 2D coordinate system
polar               2  [r,phi]           Polar coordinate system
elliptic            2  [u,v]
confocalelliptic    2  [u,v]
bipolar             2  [u,v]
parabolic           2  [u,v]
cartesian3d         3  [x,y,z]           Cartesian 3D coordinate system
polarcylindrical    3  [r,theta,z]
ellipticcylindrical 3  [u,v,z]           Elliptic 2D with cylindrical Z
confocalellipsoidal 3  [u,v,w]
```

```
bipolarcylindrical    3  [u,v,z]           Bipolar 2D with cylintrical Z
paraboliccylindrical  3  [u,v,z]           Parabolic 2D with cylindrical Z
paraboloidal          3  [u,v,phi]
conical               3  [u,v,w]
toroidal              3  [u,v,phi]
spherical             3  [r,theta,phi]     Spherical coordinate system
oblatespheroidal      3  [u,v,phi]
oblatespheroidalsqrt  3  [u,v,phi]
prolatespheroidal     3  [u,v,phi]
prolatespheroidalsqrt 3  [u,v,phi]
ellipsoidal           3  [r,theta,phi]
cartesian4d           4  [x,y,z,t]         Cartesian 4D coordinate system
spherical4d           4  [r,theta,eta,phi]
exteriorschwarzschild 4  [t,r,theta,phi]   Schwarzschild metric
interiorschwarzschild 4  [t,z,u,v]         Interior Schwarzschild metric
kerr_newman           4  [t,r,theta,phi]   Charged axially symmetric metric
```

`coordinate_system` can also be a list of transformation functions, followed by a list containing the coordinate variables. For instance, you can specify a spherical metric as follows:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) ct_coordsys([r*cos(theta)*cos(phi),r*cos(theta)*sin(phi),
      r*sin(theta),[r,theta,phi]]);
(%o2)                                done
(%i3) lg:trigsimp(lg);
                           [ 1  0         0        ]
                           [                       ]
                           [     2                 ]
(%o3)                      [ 0  r         0        ]
                           [                       ]
                           [         2    2        ]
                           [ 0  0   r  cos (theta) ]
(%i4) ct_coords;
(%o4)                           [r, theta, phi]
(%i5) dim;
(%o5)                                  3
```

Transformation functions can also be used when `cframe_flag` is `true`:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) cframe_flag:true;
(%o2)                                true
(%i3) ct_coordsys([r*cos(theta)*cos(phi),r*cos(theta)*sin(phi),
      r*sin(theta),[r,theta,phi]]);
(%o3)                                done
```

```
(%i4) fri;
      [ cos(phi) cos(theta)  - cos(phi) r sin(theta)  - sin(phi) r cos(theta)
      [
(%o4) [ sin(phi) cos(theta)  - sin(phi) r sin(theta)   cos(phi) r cos(theta)
      [
      [    sin(theta)           r cos(theta)                  0
(%i5) cmetric();
(%o5)                               false
(%i6) lg:trigsimp(lg);
                                  [ 1  0         0        ]
                                  [                       ]
                                  [     2                 ]
(%o6)                             [ 0  r         0        ]
                                  [                       ]
                                  [         2    2        ]
                                  [ 0  0   r  cos (theta) ]
```

The optional argument *extra_arg* can be any one of the following:

`cylindrical` tells `ct_coordsys` to attach an additional cylindrical coordinate.

`minkowski` tells `ct_coordsys` to attach an additional coordinate with negative metric signature.

`all` tells `ct_coordsys` to call `cmetric` and `christof(false)` after setting up the metric.

If the global variable `verbose` is set to `true`, `ct_coordsys` displays the values of `dim`, `ct_coords`, and either `lg` or `lfg` and `fri`, depending on the value of `cframe_flag`.

init_ctensor () Function

Initializes the `ctensor` package.

The `init_ctensor` function reinitializes the `ctensor` package. It removes all arrays and matrices used by `ctensor`, resets all flags, resets `dim` to 4, and resets the frame metric to the Lorentz-frame.

30.2.2 The tensors of curved space

The main purpose of the `ctensor` package is to compute the tensors of curved space(time), most notably the tensors used in general relativity.

When a metric base is used, `ctensor` can compute the following tensors:

```
 lg  -- ug
   \      \
    lcs -- mcs -- ric -- uric
              \      \       \
               \      tracer - ein -- lein
                \
                 riem -- lriem -- weyl
                     \
```

```
                          uriem
```

`ctensor` can also work using moving frames. When `cframe_flag` is set to `true`, the following tensors can be calculated:

```
lfg -- ufg
    \
fri -- fr -- lcs -- mcs -- lriem -- ric -- uric
     \                       |  \      \       \
      lg -- ug               |   weyl   tracer - ein -- lein
                             |\
                             | riem
                             |
                             \uriem
```

christof (*dis*) Function

A function in the `ctensor` (component tensor) package. It computes the Christoffel symbols of both kinds. The argument *dis* determines which results are to be immediately displayed. The Christoffel symbols of the first and second kinds are stored in the arrays `lcs[i,j,k]` and `mcs[i,j,k]` respectively and defined to be symmetric in the first two indices. If the argument to `christof` is `lcs` or `mcs` then the unique non-zero values of `lcs[i,j,k]` or `mcs[i,j,k]`, respectively, will be displayed. If the argument is `all` then the unique non-zero values of `lcs[i,j,k]` and `mcs[i,j,k]` will be displayed. If the argument is `false` then the display of the elements will not occur. The array elements `mcs[i,j,k]` are defined in such a manner that the final index is contravariant.

ricci (*dis*) Function

A function in the `ctensor` (component tensor) package. `ricci` computes the covariant (symmetric) components `ric[i,j]` of the Ricci tensor. If the argument *dis* is `true`, then the non-zero components are displayed.

uricci (*dis*) Function

This function first computes the covariant components `ric[i,j]` of the Ricci tensor. Then the mixed Ricci tensor is computed using the contravariant metric tensor. If the value of the argument *dis* is `true`, then these mixed components, `uric[i,j]` (the index i is covariant and the index j is contravariant), will be displayed directly. Otherwise, `ricci(false)` will simply compute the entries of the array `uric[i,j]` without displaying the results.

scurvature () Function

returns the scalar curvature (obtained by contracting the Ricci tensor) of the Riemannian manifold with the given metric.

einstein (*dis*) Function

A function in the `ctensor` (component tensor) package. `einstein` computes the mixed Einstein tensor after the Christoffel symbols and Ricci tensor have been obtained (with the functions `christof` and `ricci`). If the argument *dis* is `true`, then the non-zero values of the mixed Einstein tensor `ein[i,j]` will be displayed where `j` is the contravariant index. The variable `rateinstein` will cause the rational simplification on these components. If `ratfac` is `true` then the components will also be factored.

leinstein (*dis*) Function

Covariant Einstein-tensor. `leinstein` stores the values of the covariant Einstein tensor in the array `lein`. The covariant Einstein-tensor is computed from the mixed Einstein tensor `ein` by multiplying it with the metric tensor. If the argument *dis* is `true`, then the non-zero values of the covariant Einstein tensor are displayed.

riemann (*dis*) Function

A function in the `ctensor` (component tensor) package. `riemann` computes the Riemann curvature tensor from the given metric and the corresponding Christoffel symbols. The following index conventions are used:

```
                l      _l       _l       _l   _m    _l   _m
 R[i,j,k,l] =  R    = |      - |      + |    |   - |    |
                ijk     ij,k     ik,j     mk   ij    mj   ik
```

This notation is consistent with the notation used by the ITENSOR package and its `icurvature` function. If the optional argument *dis* is `true`, the non-zero components `riem[i,j,k,l]` will be displayed. As with the Einstein tensor, various switches set by the user control the simplification of the components of the Riemann tensor. If `ratriemann` is `true`, then rational simplification will be done. If `ratfac` is `true` then each of the components will also be factored.

If the variable `cframe_flag` is `false`, the Riemann tensor is computed directly from the Christoffel-symbols. If `cframe_flag` is `false`, the covariant Riemann-tensor is computed first from the frame field coefficients.

lriemann (*dis*) Function

Covariant Riemann-tensor (`lriem[]`).

Computes the covariant Riemann-tensor as the array `lriem`. If the argument *dis* is `true`, unique nonzero values are displayed.

If the variable `cframe_flag` is `true`, the covariant Riemann tensor is computed directly from the frame field coefficients. Otherwise, the (3,1) Riemann tensor is computed first.

For information on index ordering, see `riemann`.

uriemann (*dis*) Function

Computes the contravariant components of the Riemann curvature tensor as array elements `uriem[i,j,k,l]`. These are displayed if *dis* is `true`.

rinvariant () Function

Forms the Kretchmann-invariant (`kinvariant`) obtained by contracting the tensors

```
lriem[i,j,k,l]*uriem[i,j,k,l].
```

This object is not automatically simplified since it can be very large.

weyl (*dis*) Function

Computes the Weyl conformal tensor. If the argument *dis* is `true`, the non-zero components `weyl[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 `ratfac` is `true` then the results will be factored as well.

30.2.3 Taylor series expansion

The `ctensor` package has the ability to truncate results by assuming that they are Taylor-series approximations. This behavior is controlled by the `ctayswitch` variable; when set to true, `ctensor` makes use internally of the function `ctaylor` when simplifying results.

The `ctaylor` function is invoked by the following `ctensor` functions:

```
Function     Comments
---------------------------------
christof()   For mcs only
ricci()
uricci()
einstein()
riemann()
weyl()
checkdiv()
```

ctaylor () Function

The `ctaylor` function truncates its argument by converting it to a Taylor-series using `taylor`, and then calling `ratdisrep`. This has the combined effect of dropping terms higher order in the expansion variable `ctayvar`. The order of terms that should be dropped is defined by `ctaypov`; the point around which the series expansion is carried out is specified in `ctaypt`.

As an example, consider a simple metric that is a perturbation of the Minkowski metric. Without further restrictions, even a diagonal metric produces expressions for the Einstein tensor that are far too complex:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                               true
(%i3) derivabbrev:true;
(%o3)                               true
(%i4) ct_coords:[t,r,theta,phi];
(%o4)                        [t, r, theta, phi]
```

```
(%i5) lg:matrix([-1,0,0,0],[0,1,0,0],[0,0,r^2,0],[0,0,0,r^2*sin(theta)^2]);
                        [ - 1  0  0         0        ]
                        [                            ]
                        [  0   1  0         0        ]
                        [                            ]
(%o5)                   [           2                ]
                        [  0   0  r         0        ]
                        [                            ]
                        [               2    2       ]
                        [  0   0  0    r  sin (theta) ]
(%i6) h:matrix([h11,0,0,0],[0,h22,0,0],[0,0,h33,0],[0,0,0,h44]);
                            [ h11   0    0    0  ]
                            [                    ]
                            [  0   h22   0    0  ]
(%o6)                       [                    ]
                            [  0    0   h33   0  ]
                            [                    ]
                            [  0    0    0   h44 ]
(%i7) depends(l,r);
(%o7)                                [l(r)]
(%i8) lg:lg+l*h;
         [ h11 l - 1      0          0                 0            ]
         [                                                          ]
         [     0      h22 l + 1      0                 0            ]
         [                                                          ]
(%o8)    [                        2                                 ]
         [     0          0      r  + h33 l            0            ]
         [                                                          ]
         [                                    2    2                ]
         [     0          0          0       r  sin (theta) + h44 l ]
(%i9) cmetric(false);
(%o9)                                 done
(%i10) einstein(false);
(%o10)                                done
(%i11) ntermst(ein);
[[1, 1], 62]
[[1, 2], 0]
[[1, 3], 0]
[[1, 4], 0]
[[2, 1], 0]
[[2, 2], 24]
[[2, 3], 0]
[[2, 4], 0]
[[3, 1], 0]
[[3, 2], 0]
[[3, 3], 46]
[[3, 4], 0]
[[4, 1], 0]
[[4, 2], 0]
```

```
[[4, 3], 0]
[[4, 4], 46]
(%o12)                               done
```

However, if we recompute this example as an approximation that is linear in the variable `l`, we get much simpler expressions:

```
(%i14) ctayswitch:true;
(%o14)                               true
(%i15) ctayvar:l;
(%o15)                                 l
(%i16) ctaypov:1;
(%o16)                                 1
(%i17) ctaypt:0;
(%o17)                                 0
(%i18) christof(false);
(%o18)                               done
(%i19) ricci(false);
(%o19)                               done
(%i20) einstein(false);
(%o20)                               done
(%i21) ntermst(ein);
[[1, 1], 6]
[[1, 2], 0]
[[1, 3], 0]
[[1, 4], 0]
[[2, 1], 0]
[[2, 2], 13]
[[2, 3], 2]
[[2, 4], 0]
[[3, 1], 0]
[[3, 2], 2]
[[3, 3], 9]
[[3, 4], 0]
[[4, 1], 0]
[[4, 2], 0]
[[4, 3], 0]
[[4, 4], 9]
(%o21)                               done
(%i22) ratsimp(ein[1,1]);
                         2      2  4               2     2
(%o22) - (((h11 h22 - h11 ) (l )  r  - 2 h33 l    r ) sin (theta)
                              r              r r

                              2               2      4    2
                - 2 h44 l    r  - h33 h44 (l ) )/(4 r  sin (theta))
                         r r                r
```

This capability can be useful, for instance, when working in the weak field limit far from a gravitational source.

30.2.4 Frame fields

When the variable `cframe_flag` is set to true, the `ctensor` package performs its calculations using a moving frame.

frame_bracket (*fr, fri, diagframe*) Function

The frame bracket (`fb[]`).

Computes the frame bracket according to the following definition:

```
   c          c          c         d     e
ifb   = ( ifri    - ifri    ) ifr   ifr
   ab         d,e        e,d       a     b
```

30.2.5 Algebraic classification

A new feature (as of November, 2004) of `ctensor` is its ability to compute the Petrov classification of a 4-dimensional spacetime metric. For a demonstration of this capability, see the file `share/tensor/petrov.dem`.

nptetrad () Function

Computes a Newman-Penrose null tetrad (`np`) and its raised-index counterpart (`npi`). See `petrov` for an example.

The null tetrad is constructed on the assumption that a four-diemensional orthonormal frame metric with metric signature (-,+,+,+) is being used. The components of the null tetrad are related to the inverse frame matrix as follows:

```
np  = (fri  + fri ) / sqrt(2)
  1       1      2

np  = (fri  - fri ) / sqrt(2)
  2       1      2

np  = (fri  + %i fri ) / sqrt(2)
  3       3         4

np  = (fri  - %i fri ) / sqrt(2)
  4       3         4
```

psi (*dis*) Function

Computes the five Newman-Penrose coefficients `psi[0]`...`psi[4]`. If `psi` is set to `true`, the coefficients are displayed. See `petrov` for an example.

These coefficients are computed from the Weyl-tensor in a coordinate base. If a frame base is used, the Weyl-tensor is first converted to a coordinate base, which can be a computationally expensive procedure. For this reason, in some cases it may be more advantageous to use a coordinate base in the first place before the Weyl tensor is computed. Note however, that constructing a Newman-Penrose null tetrad requires a frame base. Therefore, a meaningful computation sequence may begin with a frame base, which is then used to compute `lg` (computed automatically by `cmetric` and then `ug`. At this point, you can switch back to a coordinate base by setting `cframe_flag` to false before beginning to compute the Christoffel symbols. Changing to a frame base at a later stage could yield inconsistent results, as you may end up with a mixed bag of tensors, some computed in a frame base, some in a coordinate base, with no means to distinguish between the two.

petrov () Function

Computes the Petrov classification of the metric characterized by `psi[0]`...`psi[4]`.

For example, the following demonstrates how to obtain the Petrov-classification of the Kerr metric:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) (cframe_flag:true,gcd:spmod,ctrgsimp:true,ratfac:true);
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) ug:invert(lg)$
(%i5) weyl(false);
(%o5)                                done
(%i6) nptetrad(true);
(%t6) np =

       [  sqrt(r - 2 m)           sqrt(r)
       [ ---------------   ---------------------      0             0
       [ sqrt(2) sqrt(r)   sqrt(2) sqrt(r - 2 m)
       [
       [  sqrt(r - 2 m)            sqrt(r)
       [ ---------------  - ---------------------     0             0
       [ sqrt(2) sqrt(r)    sqrt(2) sqrt(r - 2 m)
       [
       [                                              r      %i r sin(theta)
       [       0                   0               -------   ---------------
       [                                           sqrt(2)       sqrt(2)
       [
       [                                              r       %i r sin(theta)
       [       0                   0               -------  - ---------------
       [                                           sqrt(2)        sqrt(2)

                              sqrt(r)          sqrt(r - 2 m)
(%t7) npi = matrix([- ---------------------, ---------------, 0, 0],
                      sqrt(2) sqrt(r - 2 m)  sqrt(2) sqrt(r)

       sqrt(r)            sqrt(r - 2 m)
```

```
[- ---------------------, - ---------------, 0, 0],
   sqrt(2) sqrt(r - 2 m)    sqrt(2) sqrt(r)

           1               %i
[0, 0, ---------, --------------------],
       sqrt(2) r  sqrt(2) r sin(theta)

           1                 %i
[0, 0, ---------, - --------------------])
       sqrt(2) r    sqrt(2) r sin(theta)

(%o7)                              done
(%i7) psi(true);
(%t8)                            psi  = 0
                                    0

(%t9)                            psi  = 0
                                    1

                                        m
(%t10)                           psi  = --
                                    2    3
                                        r

(%t11)                           psi  = 0
                                    3

(%t12)                           psi  = 0
                                    4
(%o12)                             done
(%i12) petrov();
(%o12)                               D
```

The Petrov classification function is based on the algorithm published in "Classifying geometries in general relativity: III Classification in practice" by Pollney, Skea, and d'Inverno, Class. Quant. Grav. 17 2885-2902 (2000). Except for some simple test cases, the implementation is untested as of December 19, 2004, and is likely to contain errors.

30.2.6 Torsion and nonmetricity

`ctensor` has the ability to compute and include torsion and nonmetricity coefficients in the connection coefficients.

The torsion coefficients are calculated from a user-supplied tensor `tr`, which should be a rank (2,1) tensor. From this, the torsion coefficients `kt` are computed according to the following formulae:

```
              m           m      m
```

$$kt_{ijk} = \frac{- g_{im} tr_{kj} - g_{jm} tr_{ki} - tr_{ij} g_{km}}{2}$$

$$kt^{k}_{ij} = g^{km} kt_{ijm}$$

Note that only the mixed-index tensor is calculated and stored in the array `kt`.

The nonmetricity coefficients are calculated from the user-supplied nonmetricity vector `nm`. From this, the nonmetricity coefficients `nmc` are computed as follows:

$$nmc^{k}_{ij} = \frac{-nm_{i} D^{k}_{j} - D^{k}_{i} nm_{j} + g^{km} nm_{m} g_{ij}}{2}$$

where D stands for the Kronecker-delta.

When `ctorsion_flag` is set to `true`, the values of `kt` are substracted from the mixed-indexed connection coefficients computed by `christof` and stored in `mcs`. Similarly, if `cnonmet_flag` is set to `true`, the values of `nmc` are substracted from the mixed-indexed connection coefficients.

If necessary, `christof` calls the functions `contortion` and `nonmetricity` in order to compute `kt` and `nm`.

contortion (*tr*) Function

Computes the (2,1) contortion coefficients from the torsion tensor *tr*.

nonmetricity (*nm*) Function

Computes the (2,1) nonmetricity coefficients from the nonmetricity vector *nm*.

30.2.7 Miscellaneous features

ctransform (*M*) Function

A function in the `ctensor` (component tensor) package which will perform a coordinate transformation upon an arbitrary square symmetric matrix *M*. The user must input the functions which define the transformation. (Formerly called `transform`.)

findde (*A*, *n*) Function

returns a list of the unique differential equations (expressions) corresponding to the elements of the *n* dimensional square array *A*. Presently, *n* may be 2 or 3. `deindex` is a global list containing the indices of *A* corresponding to these unique differential equations. For the Einstein tensor (`ein`), which is a two dimensional array, if computed for the metric in the example below, `findde` gives the following independent differential equations:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) dim:4;
(%o3)                                 4
(%i4) lg:matrix([a,0,0,0],[0,x^2,0,0],[0,0,x^2*sin(y)^2,0],[0,0,0,-d]);
                          [ a  0       0        0  ]
                          [                        ]
                          [     2                  ]
                          [ 0  x       0        0  ]
(%o4)                     [                        ]
                          [         2    2         ]
                          [ 0  0   x  sin (y)   0  ]
                          [                        ]
                          [ 0  0       0      - d  ]
(%i5) depends([a,d],x);
(%o5)                            [a(x), d(x)]
(%i6) ct_coords:[x,y,z,t];
(%o6)                            [x, y, z, t]
(%i7) cmetric();
(%o7)                                done
(%i8) einstein(false);
(%o8)                                done
(%i9) findde(ein,2);
                                            2
(%o9) [d  x - a d + d, 2 a d d    x - a (d )  x - a  d d  x + 2 a d d
        x                     x x         x        x    x            x

                                                        2          2
                                                - 2 a  d , a  x + a  - a]
                                                     x      x
(%i10) deindex;
(%o10)                     [[1, 1], [2, 2], [4, 4]]
```

cograd () Function

Computes the covariant gradient of a scalar function allowing the user to choose the corresponding vector name as the example under `contragrad` illustrates.

contragrad () Function

Computes the contravariant gradient of a scalar function allowing the user to choose the corresponding vector name as the example below for the Schwarzschild metric illustrates:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) depends(f,r);
(%o4)                               [f(r)]
(%i5) cograd(f,g1);
(%o5)                                done
(%i6) listarray(g1);
(%o6)                           [0, f , 0, 0]
                                      r
(%i7) contragrad(f,g2);
(%o7)                                done
(%i8) listarray(g2);
                               f  r - 2 f  m
                                r        r
(%o8)                      [0, -------------, 0, 0]
                                     r
```

dscalar () Function

computes the tensor d'Alembertian of the scalar function once dependencies have been declared upon the function. For example:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) derivabbrev:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) depends(p,r);
(%o4)                               [p(r)]
(%i5) factor(dscalar(p));
                          2
                    p    r  - 2 m p    r + 2 p  r - 2 m p
                     r r           r r        r          r
(%o5)               --------------------------------------
                                       2
                                      r
```

checkdiv () Function

computes the covariant divergence of the mixed second rank tensor (whose first index must be covariant) by printing the corresponding n components of the vector field (the divergence) where n = `dim`. If the argument to the function is `g` then the divergence of the Einstein tensor will be formed and must be zero. In addition, the divergence (vector) is given the array name `div`.

cgeodesic (*dis*) Function

A function in the `ctensor` (component tensor) package. `cgeodesic` computes the geodesic equations of motion for a given metric. They are stored in the array `geod[i]`. If the argument *dis* is `true` then these equations are displayed.

bdvac (*f*) Function

generates the covariant components of the vacuum field equations of the Brans- Dicke gravitational theory. The scalar field is specified by the argument *f*, which should be a (quoted) function name with functional dependencies, e.g., `'p(x)`.

The components of the second rank covariant field tensor are represented by the array `bd`.

invariant1 () Function

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density of R^2. The field equations are the components of an array named `inv1`.

invariant2 () Function

*** NOT YET IMPLEMENTED ***

generates the mixed Euler- Lagrange tensor (field equations) for the invariant density of `ric[i,j]*uriem[i,j]`. The field equations are the components of an array named `inv2`.

bimetric () Function

*** NOT YET IMPLEMENTED ***

generates the field equations of Rosen's bimetric theory. The field equations are the components of an array named `rosen`.

30.2.8 Utility functions

diagmatrixp (*M*) Function

Returns `true` if *M* is a diagonal matrix or (2D) array.

symmetricp (*M*) Function

Returns `true` if *M* is a symmetric matrix or (2D) array.

ntermst (*f*) Function

gives the user a quick picture of the "size" of the doubly subscripted tensor (array) *f*. It prints two element lists where the second element corresponds to NTERMS of the components specified by the first elements. In this way, it is possible to quickly find the non-zero expressions and attempt simplification.

cdisplay (*ten*) Function

displays all the elements of the tensor *ten*, as represented by a multidimensional array. Tensors of rank 0 and 1, as well as other types of variables, are displayed as with `ldisplay`. Tensors of rank 2 are displayed as 2-dimensional matrices, while tensors of higher rank are displayed as a list of 2-dimensional matrices. For instance, the Riemann-tensor of the Schwarzschild metric can be viewed as:

```
(%i1) load(ctensor);
(%o1)       /share/tensor/ctensor.mac
(%i2) ratfac:true;
(%o2)                                true
(%i3) ct_coordsys(exteriorschwarzschild,all);
(%o3)                                done
(%i4) riemann(false);
(%o4)                                done
(%i5) cdisplay(riem);
               [ 0               0                   0            0      ]
               [                                                         ]
               [                              2                          ]
               [      3 m (r - 2 m)   m    2 m                           ]
               [ 0  - ------------- + -- - ----      0            0      ]
               [            4          3     4                           ]
               [           r          r     r                            ]
               [                                                         ]
    riem     = [                                m (r - 2 m)              ]
        1, 1   [ 0               0              -----------       0      ]
               [                                     4                   ]
               [                                    r                    ]
               [                                                         ]
               [                                             m (r - 2 m) ]
               [ 0               0                   0       ----------- ]
               [                                                  4      ]
               [                                                 r       ]

                                [    2 m (r - 2 m)       ]
                                [ 0  -------------  0  0 ]
                                [          4             ]
                                [         r              ]
                     riem     = [                        ]
                         1, 2   [ 0        0        0  0 ]
                                [                        ]
                                [ 0        0        0  0 ]
                                [                        ]
                                [ 0        0        0  0 ]

                                [         m (r - 2 m)    ]
                                [ 0  0  - -----------  0 ]
                                [              4         ]
                                [             r          ]
                     riem     = [                        ]
                         1, 3   [ 0  0        0        0 ]
                                [                        ]
                                [ 0  0        0        0 ]
                                [                        ]
                                [ 0  0        0        0 ]
```

```
                  [             m (r - 2 m) ]
                  [ 0  0  0  - ----------- ]
                  [                  4      ]
                  [                 r       ]
       riem     = [                         ]
           1, 4   [ 0  0  0         0       ]
                  [                         ]
                  [ 0  0  0         0       ]
                  [                         ]
                  [ 0  0  0         0       ]

                 [       0         0  0  0 ]
                 [                         ]
                 [       2 m               ]
                 [ - ------------  0  0  0 ]
      riem     = [    2                    ]
          2, 1   [   r  (r - 2 m)          ]
                 [                         ]
                 [       0         0  0  0 ]
                 [                         ]
                 [       0         0  0  0 ]

       [     2 m                                                 ]
       [ ------------  0        0                 0              ]
       [  2                                                      ]
       [ r  (r - 2 m)                                            ]
       [                                                         ]
       [      0        0        0                 0              ]
       [                                                         ]
riem = [                        m                                ]
  2, 2 [      0        0  - ------------          0              ]
       [                     2                                   ]
       [                    r  (r - 2 m)                         ]
       [                                                         ]
       [                                                 m       ]
       [      0        0        0              - ------------    ]
       [                                              2          ]
       [                                             r  (r - 2 m) ]

                  [ 0  0       0        0 ]
                  [                       ]
                  [            m          ]
                  [ 0  0  ------------  0 ]
       riem     = [         2             ]
           2, 3   [        r  (r - 2 m)   ]
                  [                       ]
                  [ 0  0       0        0 ]
                  [                       ]
                  [ 0  0       0        0 ]
```

```
                   [ 0  0  0       0       ]
                   [                       ]
                   [               m       ]
                   [ 0  0  0  ------------ ]
        riem     = [           2           ]
            2, 4   [          r  (r - 2 m) ]
                   [                       ]
                   [ 0  0  0       0       ]
                   [                       ]
                   [ 0  0  0       0       ]

                         [ 0  0  0  0 ]
                         [            ]
                         [ 0  0  0  0 ]
                         [            ]
              riem     = [ m          ]
                  3, 1   [ -  0  0  0 ]
                         [ r          ]
                         [            ]
                         [ 0  0  0  0 ]

                         [ 0  0  0  0 ]
                         [            ]
                         [ 0  0  0  0 ]
                         [            ]
              riem     = [    m       ]
                  3, 2   [ 0  -  0  0 ]
                         [    r       ]
                         [            ]
                         [ 0  0  0  0 ]

                  [   m                          ]
                  [ - -   0   0       0          ]
                  [   r                          ]
                  [                              ]
                  [        m                     ]
                  [  0   - -  0       0          ]
       riem     = [        r                     ]
           3, 3   [                              ]
                  [  0    0   0       0          ]
                  [                              ]
                  [               2 m - r        ]
                  [  0    0   0   ------- + 1    ]
                  [                  r           ]

                       [ 0  0  0    0   ]
                       [                ]
                       [ 0  0  0    0   ]
```

```
                                    [                 ]
                           riem     = [             2 m ]
                               3, 4   [ 0  0  0  - --- ]
                                    [               r  ]
                                    [                 ]
                                    [ 0  0  0    0    ]

                                [       0        0  0  0 ]
                                [                        ]
                                [       0        0  0  0 ]
                                [                        ]
                       riem     = [       0        0  0  0 ]
                           4, 1   [                        ]
                                [      2                 ]
                                [ m sin (theta)          ]
                                [ -------------  0  0  0 ]
                                [       r                ]

                                [ 0        0        0  0 ]
                                [                        ]
                                [ 0        0        0  0 ]
                                [                        ]
                       riem     = [ 0        0        0  0 ]
                           4, 2   [                        ]
                                [         2              ]
                                [    m sin (theta)       ]
                                [ 0  -------------  0  0 ]
                                [          r             ]

                              [ 0  0          0          0 ]
                              [                            ]
                              [ 0  0          0          0 ]
                              [                            ]
                     riem     = [ 0  0          0          0 ]
                         4, 3   [                            ]
                              [                 2          ]
                              [         2 m sin (theta)    ]
                              [ 0  0  - ---------------  0 ]
                              [                 r          ]

                 [        2                                                  ]
                 [   m sin (theta)                                           ]
                 [ - -------------           0                0          0 ]
                 [         r                                                 ]
                 [                                                           ]
                 [                           2                               ]
                 [                    m sin (theta)                          ]
      riem     = [        0         - -------------           0          0 ]
          4, 4   [                           r                               ]
```

```
                    [                                               ]
                    [                                    2          ]
                    [                             2 m sin (theta)   ]
                    [      0             0        ---------------  0 ]
                    [                                    r          ]
                    [                                               ]
                    [      0             0               0         0 ]

(%o5)                              done
```

deleten (*L*, *n*) Function

Returns a new list consisting of *L* with the *n*'th element deleted.

30.2.9 Variables used by `ctensor`

dim Option variable

Default value: 4

An option in the `ctensor` (component tensor) package. `dim` is the dimension of the manifold with the default 4. The command `dim: n` will reset the dimension to any other value `n`.

diagmetric Option variable

Default value: `false`

An option in the `ctensor` (component tensor) 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 `csetup` if a diagonal metric is specified.

ctrgsimp Option variable

Causes trigonometric simplifications to be used when tensors are computed. Presently, `ctrgsimp` affects only computations involving a moving frame.

cframe_flag Option variable

Causes computations to be performed relative to a moving frame as opposed to a holonomic metric. The frame is defined by the inverse frame array `fri` and the frame metric `lfg`. For computations using a Cartesian frame, `lfg` should be the unit matrix of the appropriate dimension; for computations in a Lorentz frame, `lfg` should have the appropriate signature.

ctorsion_flag Option variable

Causes the contortion tensor to be included in the computation of the connection coefficients. The contortion tensor itself is computed by `contortion` from the user-supplied tensor `tr`.

cnonmet_flag Option variable

Causes the nonmetricity coefficients to be included in the computation of the connection coefficients. The nonmetricity coefficients are computed from the user-supplied nonmetricity vector `nm` by the function `nonmetricity`.

ctayswitch Option variable

If set to `true`, causes some `ctensor` computations to be carried out using Taylor-series expansions. Presently, `christof`, `ricci`, `uricci`, `einstein`, and `weyl` take into account this setting.

ctayvar Option variable

Variable used for Taylor-series expansion if `ctayswitch` is set to `true`.

ctaypov Option variable

Maximum power used in Taylor-series expansion when `ctayswitch` is set to `true`.

ctaypt Option variable

Point around which Taylor-series expansion is carried out when `ctayswitch` is set to `true`.

gdet System variable

The determinant of the metric tensor `lg`. Computed by `cmetric` when `cframe_flag` is set to `false`.

ratchristof Option variable

Causes rational simplification to be applied by `christof`.

rateinstein Option variable

Default value: `true`

If `true` rational simplification will be performed on the non-zero components of Einstein tensors; if `ratfac` is `true` then the components will also be factored.

ratriemann Option variable

Default value: `true`

One of the switches which controls simplification of Riemann tensors; if `true`, then rational simplification will be done; if `ratfac` is `true` then each of the components will also be factored.

ratweyl Option variable

Default value: `true`

If `true`, this switch causes the `weyl` function to apply rational simplification to the values of the Weyl tensor. If `ratfac` is `true`, then the components will also be factored.

lfg Variable

The covariant frame metric. By default, it is initialized to the 4-dimensional Lorentz frame with signature (+,+,+,-). Used when `cframe_flag` is `true`.

ufg Variable

The inverse frame metric. Computed from `lfg` when `cmetric` is called while `cframe_flag` is set to `true`.

riem Variable

The (3,1) Riemann tensor. Computed when the function `riemann` is invoked. For information about index ordering, see the description of `riemann`.

if `cframe_flag` is `true`, `riem` is computed from the covariant Riemann-tensor `lriem`.

lriem Variable

The covariant Riemann tensor. Computed by `lriemann`.

uriem Variable

The contravariant Riemann tensor. Computed by `uriemann`.

ric Variable

The mixed Ricci-tensor. Computed by `ricci`.

uric Variable

The contravariant Ricci-tensor. Computed by `uricci`.

lg Variable

The metric tensor. This tensor must be specified (as a `dim` by `dim` matrix) before other computations can be performed.

ug Variable

The inverse of the metric tensor. Computed by `cmetric`.

weyl Variable

The Weyl tensor. Computed by `weyl`.

fb Variable

Frame bracket coefficients, as computed by `frame_bracket`.

kinvariant Variable

The Kretchmann invariant. Computed by `rinvariant`.

np Variable

A Newman-Penrose null tetrad. Computed by `nptetrad`.

npi Variable

The raised-index Newman-Penrose null tetrad. Computed by `nptetrad`. Defined as `ug.np`. The product `np.transpose(npi)` is constant:

```
(%i39) trigsimp(np.transpose(npi));
                              [  0    - 1  0  0 ]
                              [                 ]
                              [ - 1    0   0  0 ]
(%o39)                        [                 ]
                              [  0     0   0  1 ]
                              [                 ]
                              [  0     0   1  0 ]
```

tr Variable

User-supplied rank-3 tensor representing torsion. Used by `contortion`.

kt Variable

The contortion tensor, computed from `tr` by `contortion`.

nm Variable

User-supplied nonmetricity vector. Used by `nonmetricity`.

nmc Variable

The nonmetricity coefficients, computed from `nm` by `nonmetricity`.

tensorkill System variable

Variable indicating if the tensor package has been initialized. Set and used by `csetup`, reset by `init_ctensor`.

ct_coords Option variable

Default value: `[]`

An option in the `ctensor` (component tensor) package. `ct_coords` contains a list of coordinates. While normally defined when the function `csetup` is called, one may redefine the coordinates with the assignment `ct_coords: [j1, j2, ..., jn]` where the j's are the new coordinate names. See also `csetup`.

30.2.10 Reserved names

The following names are used internally by the `ctensor` package and should not be redefined:

```
Name         Description
---------------------------------------
_lg()        Evaluates to lfg if frame metric used, lg otherwise
_ug()        Evaluates to ufg if frame metric used, ug otherwise
cleanup()    Removes items drom the deindex list
contract4()  Used by psi()
filemet()    Used by csetup() when reading the metric from a file
findde1()    Used by findde()
```

```
findde2()    Used by findde()
findde3()    Used by findde()
kdelt()      Kronecker-delta (not generalized)
newmet()     Used by csetup() for setting up a metric interactively
setflags()   Used by init_ctensor()
readvalue()
resimp()
sermet()     Used by csetup() for entering a metric as Taylor-series
txyzsum()
tmetric()    Frame metric, used by cmetric() when cframe_flag:true
triemann()   Riemann-tensor in frame base, used when cframe_flag:true
tricci()     Ricci-tensor in frame base, used when cframe_flag:true
trrc()       Ricci rotation coefficients, used by christof()
yesp()
```

30.2.11 Changes

In November, 2004, the `ctensor` package was extensively rewritten. Many functions and variables have been renamed in order to make the package compatible with the commercial version of Macsyma.

```
New Name     Old Name        Description
-----------------------------------------------------------------------------
ctaylor()    DLGTAYLOR()     Taylor-series expansion of an expression
lgeod[]      EM              Geodesic equations
ein[]        G[]             Mixed Einstein-tensor
ric[]        LR[]            Mixed Ricci-tensor
ricci()      LRICCICOM()     Compute the mixed Ricci-tensor
ctaypov      MINP            Maximum power in Taylor-series expansion
cgeodesic()  MOTION          Compute geodesic equations
ct_coords    OMEGA           Metric coordinates
ctayvar      PARAM           Taylor-series expansion variable
lriem[]      R[]             Covariant Riemann-tensor
uriemann()   RAISERIEMANN()  Compute the contravariant Riemann-tensor
ratriemann   RATRIEMAN       Rational simplification of the Riemann-tensor
uric[]       RICCI[]         Contravariant Ricci-tensor
uricci()     RICCICOM()      Compute the contravariant Ricci-tensor
cmetric()    SETMETRIC()     Set up the metric
ctaypt       TAYPT           Point for Taylor-series expansion
ctayswitch   TAYSWITCH       Taylor-series setting switch
csetup()     TSETUP()        Start interactive setup session
ctransform() TTRANSFORM()    Interactive coordinate transformation
uriem[]      UR[]            Contravariant Riemann-tensor
weyl[]       W[]             (3,1) Weyl-tensor
```

31 atensor

31.1 Introduction to atensor

`atensor` is an algebraic tensor manipulation package. To use `atensor`, type `load(atensor)`, followed by a call to the `init_atensor` function.

The essence of `atensor` is a set of simplification rules for the noncommutative (dot) product operator ("`.`"). `atensor` recognizes several algebra types; the corresponding simplification rules are put into effect when the `init_atensor` function is called.

The capabilities of `atensor` can be demonstrated by defining the algebra of quaternions as a Clifford-algebra Cl(0,2) with two basis vectors. The three quaternionic imaginary units are then the two basis vectors and their product, i.e.:

```
    i = v     j = v     k = v  . v
         1         2         1    2
```

Although the `atensor` package has a built-in definition for the quaternion algebra, it is not used in this example, in which we endeavour to build the quaternion multiplication table as a matrix:

```
(%i1) load(atensor);
(%o1)       /share/tensor/atensor.mac
(%i2) init_atensor(clifford,0,0,2);
(%o2)                                done
(%i3) atensimp(v[1].v[1]);
(%o3)                                 - 1
(%i4) atensimp((v[1].v[2]).(v[1].v[2]));
(%o4)                                 - 1
(%i5) q:zeromatrix(4,4);
                                [ 0  0  0  0 ]
                                [            ]
                                [ 0  0  0  0 ]
(%o5)                           [            ]
                                [ 0  0  0  0 ]
                                [            ]
                                [ 0  0  0  0 ]
(%i6) q[1,1]:1;
(%o6)                                  1
(%i7) for i thru adim do q[1,i+1]:q[i+1,1]:v[i];
(%o7)                                done
(%i8) q[1,4]:q[4,1]:v[1].v[2];
(%o8)                               v  . v
                                     1    2
(%i9) for i from 2 thru 4 do for j from 2 thru 4 do
      q[i,j]:atensimp(q[i,1].q[1,j]);
(%o9)                                done
(%i10) q;
                   [    1        v         v      v  . v  ]
```

```
                 [                 1           2       1     2 ]
                 [                                              ]
                 [    v          - 1      v  . v     - v        ]
                 [     1                   1     2      2       ]
(%o10)           [                                              ]
                 [    v      - v  . v      - 1        v         ]
                 [     2        1     2                1        ]
                 [                                              ]
                 [ v  . v       v          - v        - 1       ]
                 [  1     2      2           1                  ]
```

atensor recognizes as base vectors indexed symbols, where the symbol is that stored in asymbol and the index runs between 1 and adim. For indexed symbols, and indexed symbols only, the bilinear forms sf, af, and av are evaluated. The evaluation substitutes the value of aform[i,j] in place of fun(v[i],v[j]) where v represents the value of asymbol and fun is either af or sf; or, it substitutes v[aform[i,j]] in place of av(v[i],v[j]).

Needless to say, the functions sf, af and av can be redefined.

When the atensor package is loaded, the following flags are set:

```
dotscrules:true;
dotdistrib:true;
dotexptsimp:false;
```

If you wish to experiment with a nonassociative algebra, you may also consider setting dotassoc to false. In this case, however, atensimp will not always be able to obtain the desired simplifications.

31.2 Definitions for atensor

init_atensor (*alg_type, opt_dims*) Function
init_atensor (*alg_type*) Function

Initializes the atensor package with the specified algebra type. *alg_type* can be one of the following:

universal: The universal algebra has no commutation rules.

grassmann: The Grassman algebra is defined by the commutation relation u.v+v.u=0.

clifford: The Clifford algebra is defined by the commutation relation u.v+v.u=-2*sf(u,v) where sf is a symmetric scalar-valued function. For this algebra, *opt_dims* can be up to three nonnegative integers, representing the number of positive, degenerate, and negative dimensions of the algebra, respectively. If any *opt_dims* values are supplied, atensor will configure the values of adim and aform appropriately. Otherwise, adim will default to 0 and aform will not be defined.

symmetric: The symmetric algebra is defined by the commutation relation u.v-v.u=0.

symplectic: The symplectic algebra is defined by the commutation relation u.v-v.u=2*af(u,v) where af is an antisymmetric scalar-valued function. For the symplectic algebra, *opt_dims* can be up to two nonnegative integers, representing the nondegenerate and degenerate dimensions, respectively. If any *opt_dims* values are supplied, `atensor` will configure the values of `adim` and `aform` appropriately. Otherwise, `adim` will default to 0 and `aform` will not be defined.

`lie_envelop`: The algebra of the Lie envelope is defined by the commutation relation `u.v-v.u=2*av(u,v)` where `av` is an antisymmetric function.

The `init_atensor` function also recognizes several predefined algebra types:

`complex` implements the algebra of complex numbers as the Clifford algebra Cl(0,1). The call `init_atensor(complex)` is equivalent to `init_atensor(clifford,0,0,1)`.

`quaternion` implements the algebra of quaternions. The call `init_atensor(quaternion)` is equivalent to `init_atensor(clifford,0,0,2)`.

`pauli` implements the algebra of Pauli-spinors as the Clifford-algebra Cl(3,0). A call to `init_atensor(pauli)` is equivalent to `init_atensor(clifford,3)`.

`dirac` implements the algebra of Dirac-spinors as the Clifford-algebra Cl(3,1). A call to `init_atensor(dirac)` is equivalent to `init_atensor(clifford,3,0,1)`.

atensimp (*expr*) Function

Simplifies an algebraic tensor expression *expr* according to the rules configured by a call to `init_atensor`. Simplification includes recursive application of commutation relations and resolving calls to `sf`, `af`, and `av` where applicable. A safeguard is used to ensure that the function always terminates, even for complex expressions.

alg_type Function

The algebra type. Valid values are `universal`, `grassmann`, `clifford`, `symmetric`, `symplectic` and `lie_envelop`.

adim Variable

The dimensionality of the algebra. `atensor` uses the value of `adim` to determine if an indexed object is a valid base vector. Defaults to 0.

aform Variable

Default values for the bilinear forms `sf`, `af`, and `av`. The default is the identity matrix `ident(3)`.

asymbol Variable

The symbol for base vectors. Defaults to `v`.

sf (*u*, *v*) Function

A symmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using `abasep` and if that is the case, substitutes the corresponding value from the matrix `aform`.

af (*u*, *v*) Function

An antisymmetric scalar function that is used in commutation relations. The default implementation checks if both arguments are base vectors using `abasep` and if that is the case, substitutes the corresponding value from the matrix `aform`.

av (*u*, *v*) Function

An antisymmetric function that is used in commutation relations. The default implementation checks if both arguments are base vectors using `abasep` and if that is the case, substitutes the corresponding value from the matrix `aform`.

For instance:

```
(%i1) load(atensor);
(%o1)       /share/tensor/atensor.mac
(%i2) adim:3;
(%o2)                                  3
(%i3) aform:matrix([0,3,-2],[-3,0,1],[2,-1,0]);
                               [  0    3   - 2 ]
                               [               ]
(%o3)                          [ - 3   0    1  ]
                               [               ]
                               [  2   - 1   0  ]
(%i4) asymbol:x;
(%o4)                                  x
(%i5) av(x[1],x[2]);
(%o5)                                 x
                                       3
```

abasep (*v*) Function

Checks if its argument is an `atensor` base vector. That is, if it is an indexed symbol, with the symbol being the same as the value of `asymbol`, and the index having a numeric value between 1 and `adim`.

32 Series

32.1 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 global variables which control the expansion.

32.2 Definitions for Series

cauchysum Option variable

Default value: `false`

When multiplying together sums with `inf` as their upper limit, if `sumexpand` is `true` and `cauchysum` is `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.

Example:

```
(%i1) sumexpand: false$
(%i2) cauchysum: false$
(%i3) s: sum (f(i), i, 0, inf) * sum (g(j), j, 0, inf);
                          inf         inf
                          ====        ====
                          \           \
(%o3)                    ( >    f(i))  >    g(j)
                          /           /
                          ====        ====
                          i = 0       j = 0
(%i4) sumexpand: true$
(%i5) cauchysum: true$
(%i6) ''s;
                     inf     i1
                     ====   ====
                     \      \
(%o6)                 >      >     g(i1 - i2) f(i2)
                     /      /
                     ====   ====
                     i1 = 0 i2 = 0
```

deftaylor (*f_1*(*x_1*), *expr_1*, ..., *f_n*(*x_n*), *expr_n*) Function

For each function *f_i* of one variable *x_i*, `deftaylor` defines *expr_i* as the Taylor series about zero. *expr_i* is typically a polynomial in *x_i* or a summation; more general expressions are accepted by `deftaylor` without complaint.

`powerseries (`*f_i*`(`*x_i*`), `*x_i*`, 0)` returns the series defined by `deftaylor`.

`deftaylor` returns a list of the functions *f_1*, ..., *f_n*. `deftaylor` evaluates its arguments.

Example:

```
(%i1) deftaylor (f(x), x^2 + sum(x^i/(2^i*i!^2), i, 4, inf));
(%o1)                          [f]
(%i2) powerseries (f(x), x, 0);
                      inf
                      ====      i1
                      \        x         2
(%o2)                  >     -------- + x
                      /       i1    2
                      ====   2   i1!
                      i1 = 4
(%i3) taylor (exp (sqrt (f(x))), x, 0, 4);
                      2         3          4
                     x    3073 x    12817 x
(%o3)/T/     1 + x + -- + ------- + -------- + . . .
                     2     18432     307200
```

maxtayorder Option variable

Default value: `true`

When `maxtayorder` is `true`, then during algebraic manipulation of (truncated) Taylor series, `taylor` tries to retain as many terms as are known to be correct.

niceindices (*expr*) Function

Renames the indices of sums and products in *expr*. `niceindices` attempts to rename each index to the value of `niceindicespref[1]`, unless that name appears in the summand or multiplicand, in which case `niceindices` tries the succeeding elements of `niceindicespref` in turn, until an unused variable is found. If the entire list is exhausted, additional indices are constructed by appending integers to the value of `niceindicespref[1]`, e.g., `i0`, `i1`, `i2`, ...

`niceindices` returns an expression. `niceindices` evaluates its argument.

Example:

```
(%i1) niceindicespref;
(%o1)                  [i, j, k, l, m, n]
(%i2) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
                 inf    inf
                /===\   ====
                 ! !    \
(%o2)            ! !     >      f(bar i j + foo)
                 ! !    /
                bar = 1 ====
                        foo = 1
(%i3) niceindices (%);
                     inf  inf
                    /===\ ====
                     ! !  \
```

```
(%o3)                ! !   >    f(i j l + k)
                     ! !  /
                   l = 1 ====
                          k = 1
```

niceindicespref Option variable

Default value: `[i, j, k, l, m, n]`

`niceindicespref` is the list from which `niceindices` takes the names of indices for sums and products.

The elements of `niceindicespref` are typically names of variables, although that is not enforced by `niceindices`.

Example:

```
(%i1) niceindicespref: [p, q, r, s, t, u]$
(%i2) product (sum (f (foo + i*j*bar), foo, 1, inf), bar, 1, inf);
                  inf    inf
                 /===\   ====
                  ! !    \
(%o2)             ! !     >      f(bar i j + foo)
                  ! !    /
                 bar = 1 ====
                         foo = 1
(%i3) niceindices (%);
                     inf  inf
                    /===\ ====
                     ! !  \
(%o3)                ! !   >    f(i j q + p)
                     ! !  /
                    q = 1 ====
                          p = 1
```

nusum (*expr, x, i_0, i_1*) Function

Carries out indefinite hypergeometric summation of *expr* with respect to *x* using a decision procedure due to R.W. Gosper. *expr* and the result must be expressible as products of integer 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 symbolic result 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.

`nusum` and `unsum` know a little about sums and differences of finite products. See also `unsum`.

Examples:

```
(%i1) nusum (n*n!, n, 0, n);

Dependent equations eliminated:  (1)
(%o1)                      (n + 1)! - 1
(%i2) nusum (n^4*4^n/binomial(2*n,n), n, 0, n);
```

```
                     4         3        2               n
       2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4       2
(%o2) ------------------------------------------------ - ------
                     693 binomial(2 n, n)                 3 11 7
(%i3) unsum (%, n);
                              4  n
                             n  4
(%o3)                   ----------------
                        binomial(2 n, n)
(%i4) unsum (prod (i^2, i, 1, n), n);
                     n - 1
                     /===\
                      ! !   2
(%o4)               ( ! !  i ) (n - 1) (n + 1)
                      ! !
                     i = 1
(%i5) nusum (%, n, 1, n);

Dependent equations eliminated:  (2 3)
                                n
                              /===\
                               ! !   2
(%o5)                          ! !  i  - 1
                               ! !
                              i = 1
```

pade (*taylor_series*, *numer_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.

taylor_series is a univariate Taylor series. *numer_deg_bound* and *denom_deg_bound* are positive integers specifying degree bounds on the numerator and denominator.

taylor_series 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 defined as *numer_deg_bound* + *denom_deg_bound*. Length of a power series is defined as `"truncation level" + 1 - min(0, "order of series")`.

```
(%i1) taylor (1 + x + x^2 + x^3, x, 0, 3);
                              2    3
(%o1)/T/            1 + x + x  + x  + . . .
(%i2) pade (%, 1, 1);
                                 1
(%o2)                       [- -----]
                               x - 1
(%i3) t: 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);
                      2    3       4       5       6        7
              x   3 x    x    15 x    23 x    21 x    189 x
(%o3)/T/ 1 - - + ---- - -- - ----- + ----- - ----- - ------
              2    16    32   1024    2048    32768   65536

                                     8        9          10
                               5853 x    2847 x    83787 x
                             + ------- + ------- - --------- + . . .
                               4194304   8388608   134217728
(%i4) pade (t, 4, 4);
(%o4)                             []
```

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.

```
(%i5) pade (t, 5, 5);
                         5                 4                  3
(%o5) [- (520256329 x  - 96719020632 x  - 489651410240 x

                       2
 - 1619100813312 x  - 2176885157888 x - 2386516803584)

                  5                 4                  3
/(47041365435 x  + 381702613848 x  + 1360678489152 x

                       2
 + 2856700692480 x  + 3370143559680 x + 2386516803584)]
```

powerdisp Option variable

Default value: `false`

When `powerdisp` is `true`, a sum is displayed with its terms in order of increasing power. Thus a polynomial is displayed as a truncated power series, with the constant term first and the highest power last.

By default, terms of a sum are displayed in order of decreasing power.

powerseries (*expr*, *x*, *a*) Function

Returns the general form of the power series expansion for *expr* in the variable *x* about the point *a* (which may be `inf` for infinity).

If `powerseries` is unable to expand *expr*, `taylor` may give the first several terms of the series.

When `verbose` is `true`, `powerseries` prints progress messages.

```
(%i1) verbose: true$
(%i2) powerseries (log(sin(x)/x), x, 0);
can't expand
```

```
                                 log(sin(x))
          so we'll try again after applying the rule:
                                        d
                                      / -- (sin(x))
                                      [ dx
                        log(sin(x)) = i ----------- dx
                                      ]   sin(x)
                                      /
          in the first simplification we have returned:
                                   /
                                   [
                                   i cot(x) dx - log(x)
                                   ]
                                   /
                    inf
                    ====        i1  2 i1             2 i1
                    \      (- 1)   2     bern(2 i1) x
                     >     ------------------------------
                    /                i1 (2 i1)!
                    ====
                    i1 = 1
          (%o2)      -------------------------------------
                                     2
```

psexpand Option variable

Default value: `false`

When `psexpand` is `true`, an extended rational function expression is displayed fully expanded. The switch `ratexpand` has the same effect.

When `psexpand` is `false`, a multivariate expression is displayed just as in the rational function package.

When `psexpand` is `multi`, then terms with the same total degree in the variables are grouped together.

revert (*expr*, *x*) Function

revert2 (*expr*, *x*, *n*) Function

These functions return the reversion of *expr*, a Taylor series about zero in the variable *x*. `revert` returns a polynomial of degree equal to the highest power in *expr*. `revert2` returns a polynomial of degree *n*, which may be greater than, equal to, or less than the degree of *expr*.

`load ("revert")` loads these functions.

Examples:

```
(%i1) load ("revert")$
(%i2) t: taylor (exp(x) - 1, x, 0, 6);
                  2    3    4    5     6
                 x    x    x    x     x
(%o2)/T/     x + -- + -- + -- + --- + --- + . . .
                 2    6    24   120   720
```

```
(%i3) revert (t, x);
               6       5       4       3       2
           10 x  - 12 x  + 15 x  - 20 x  + 30 x  - 60 x
(%o3)/R/ - --------------------------------------------
                                 60
(%i4) ratexpand (%);
                     6    5    4    3    2
                    x    x    x    x    x
(%o4)             - -- + -- - -- + -- - -- + x
                    6    5    4    3    2
(%i5) taylor (log(x+1), x, 0, 6);
                    2    3    4    5    6
                   x    x    x    x    x
(%o5)/T/       x - -- + -- - -- + -- - -- + . . .
                   2    3    4    5    6
(%i6) ratsimp (revert (t, x) - taylor (log(x+1), x, 0, 6));
(%o6)                          0
(%i7) revert2 (t, x, 4);
                          4    3    2
                         x    x    x
(%o7)                  - -- + -- - -- + x
                         4    3    2
```

taylor (*expr*, *x*, *a*, *n*) Function
taylor (*expr*, [*x_1*, *x_2*, ...], *a*, *n*) Function
taylor (*expr*, [*x*, *a*, *n*, *'asymp*]) Function
taylor (*expr*, [*x_1*, *x_2*, ...], [*a_1*, *a_2*, ...], [*n_1*, *n_2*, ...]) Function

`taylor (`*expr*`, `*x*`, `*a*`, `*n*`)` expands the expression *expr* in a truncated Taylor or Laurent series in the variable *x* around the point *a*, containing terms through `(`*x*` - `*a*`)^`*n*.

If *expr* is of the form *f*`(`*x*`)/`*g*`(`*x*`)` and *g*`(`*x*`)` has no terms up to degree *n* then `taylor` attempts to expand *g*`(`*x*`)` up to degree `2` *n*. If there are still no nonzero terms, `taylor` doubles the degree of the expansion of *g*`(`*x*`)` so long as the degree of the expansion is less than or equal to *n* `2^taylordepth`.

`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], `*a*`, `*n*`)` returns a truncated power series of degree *n* in all variables *x_1*, *x_2*, ... about the point `(`*a*`, `*a*`, ...)`.

`taylor (`*expr*`, [`*x_1*`, `*a_1*`, `*n_1*`], [`*x_2*`, `*a_2*`, `*n_2*`], ...)` returns a truncated power series in the variables *x_1*, *x_2*, ... about the point `(`*a_1*`, `*a_2*`, ...)`, truncated at *n_1*, *n_2*, ...

`taylor (`*expr*`, [`*x_1*`, `*x_2*`, ...], [`*a_1*`, `*a_2*`, ...], [`*n_1*`, `*n_2*`, ...])` returns a truncated power series in the variables *x_1*, *x_2*, ... about the point `(`*a_1*`, `*a_2*`, ...)`, truncated at *n_1*, *n_2*, ...

`taylor (`*expr*`, [`*x*`, `*a*`, `*n*`, 'asymp])` returns an expansion of *expr* in negative powers of *x* `-` *a*. The highest order term is `(`*x*` - `*a*`)^-`*n*.

When `maxtayorder` is `true`, then during algebraic manipulation of (truncated) Taylor series, `taylor` tries to retain as many terms as are known to be correct.

When `psexpand` is `true`, an extended rational function expression is displayed fully expanded. The switch `ratexpand` has the same effect. When `psexpand` is `false`, a multivariate expression is displayed just as in the rational function package. When `psexpand` is `multi`, then terms with the same total degree in the variables are grouped together.

See also the `taylor_logexpand` switch for controlling expansion.

Examples:

```
(%i1) taylor (sqrt (sin(x) + a*x + 1), x, 0, 3);
                           2             2
             (a + 1) x   (a  + 2 a + 1) x
(%o1)/T/ 1 + --------- - -----------------
                 2               8

                                   3      2             3
                               (3 a  + 9 a  + 9 a - 1) x
                             + -------------------------- + . . .
                                           48
(%i2) %^2;
                                    3
                                   x
(%o2)/T/           1 + (a + 1) x - -- + . . .
                                   6
(%i3) taylor (sqrt (x + 1), x, 0, 5);
                       2    3      4      5
                  x   x    x    5 x    7 x
(%o3)/T/      1 + - - -- + -- - ---- + ---- + . . .
                  2   8    16   128    256
(%i4) %^2;
(%o4)/T/                  1 + x + . . .
(%i5) product ((1 + x^i)^2.5, i, 1, inf)/(1 + x^2);
                         inf
                        /===\
                         ! !    i     2.5
                         ! !  (x  + 1)
                         ! !
                        i = 1
(%o5)                   -----------------
                              2
                             x  + 1
(%i6) ev (taylor(%, x,  0, 3), keepfloat);
                               2           3
(%o6)/T/    1 + 2.5 x + 3.375 x  + 6.5625 x  + . . .
(%i7) taylor (1/log (x + 1), x, 0, 3);
                               2       3
                 1   1   x    x    19 x
(%o7)/T/         - + - - -- + -- - ----- + . . .
                 x   2   12   24    720
(%i8) taylor (cos(x) - sec(x), x, 0, 5);
                                4
                           2   x
```

```
(%o8)/T/                  - x  - -- + . . .
                                 6
(%i9) taylor ((cos(x) - sec(x))^3, x, 0, 5);
(%o9)/T/                    0 + . . .
(%i10) taylor (1/(cos(x) - sec(x))^3, x, 0, 5);
                                               2         4
           1     1       11      347    6767 x    15377 x
(%o10)/T/ - -- + ---- + ------ - ----- - ------- - --------
            6      4        2   15120   604800    7983360
           x    2 x    120 x

                                                          + . . .
(%i11) taylor (sqrt (1 - k^2*sin(x)^2), x, 0, 6);
               2  2       4      2   4
              k  x    (3 k  - 4 k ) x
(%o11)/T/ 1 - ----- - ----------------
                2           24

                                 6       4       2   6
                            (45 k  - 60 k  + 16 k ) x
                          - -------------------------- + . . .
                                       720
(%i12) taylor ((x + 1)^n, x, 0, 4);
                      2       2     3      2         3
                    (n  - n) x    (n  - 3 n  + 2 n) x
(%o12)/T/ 1 + n x + ----------- + --------------------
                         2                 6

                               4      3       2         4
                             (n  - 6 n  + 11 n  - 6 n) x
                           + ---------------------------- + . . .
                                          24
(%i13) taylor (sin (y + x), x, 0, 3, y, 0, 3);
               3                 2
              y                 y
(%o13)/T/ y - -- + . . . + (1 - -- + . . .) x
              6                 2

                  3                       2
               y   y            2      1   y            3
          + (- - + -- + . . .) x  + (- - + -- + . . .) x  + . . .
               2   12                  6   12
(%i14) taylor (sin (y + x), [x, y], 0, 3);
                     3        2       2      3
                    x  + 3 y x  + 3 y  x + y
(%o14)/T/   y + x - ------------------------- + . . .
                                6
(%i15) taylor (1/sin (y + x), x, 0, 3, y, 0, 3);
          1   y              1    1              1            2
```

```
(%o15)/T/ - + - + . . . + (- -- + - + . . .) x + (-- + . . .) x
          y   6               2   6                3
                             y                    y

                                                 1            3
                                          + (- -- + . . .) x  + . . .
                                                 4
                                                y
(%i16) taylor (1/sin (y + x), [x, y], 0, 3);
                             3         2       2        3
             1     x + y   7 x  + 21 y x  + 21 y  x + 7 y
(%o16)/T/ ----- + ----- + ------------------------------- + . . .
          x + y     6                   360
```

taylordepth Option variable

Default value: 3

If there are still no nonzero terms, `taylor` doubles the degree of the expansion of *g*(*x*) so long as the degree of the expansion is less than or equal to *n* `2^taylordepth`.

taylorinfo (*expr*) Function

Returns information about the Taylor series *expr*. The return value is a list of lists. Each list comprises the name of a variable, the point of expansion, and the degree of the expansion.

`taylorinfo` returns `false` if *expr* is not a Taylor series.

Example:

```
(%i1) taylor ((1 - y^2)/(1 - x), x, 0, 3, [y, a, inf]);
                  2                       2
(%o1)/T/ - (y - a)  - 2 a (y - a) + (1 - a )

         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   3
 + (1 - a  - 2 a (y - a) - (y - a) ) x  + . . .
(%i2) taylorinfo(%);
(%o2)               [[y, a, inf], [x, 0, 3]]
```

taylorp (*expr*) Function

Returns `true` if *expr* is a Taylor series, and `false` otherwise.

taylor_logexpand Option variable

Default value: `true`

`taylor_logexpand` controls expansions of logarithms in `taylor` series.

When `taylor_logexpand` is `true`, all logarithms 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.

When `taylor_logexpand` is set to `false`, then the only expansion of logarithms that occur is that necessary to obtain a formal power series.

taylor_order_coefficients Option variable

Default value: `true`

`taylor_order_coefficients` controls the ordering of coefficients in a Taylor series.

When `taylor_order_coefficients` is `true`, coefficients of taylor series are ordered canonically.

taylor_simplifier (*expr*) Function

Simplifies coefficients of the power series *expr*. `taylor` calls this function.

taylor_truncate_polynomials Option variable

Default value: `true`

When `taylor_truncate_polynomials` is `true`, polynomials are truncated based upon the input truncation levels.

Otherwise, polynomials input to `taylor` are considered to have infinite precison.

taytorat (*expr*) Function

Converts *expr* from `taylor` form to canonical rational expression (CRE) form. The effect is the same as `rat (ratdisrep (`*expr*`))`, but faster.

trunc (*expr*) Function

Annotates the internal representation of the general expression *expr* so that it is displayed as if its sums were truncated Taylor series. *expr* is not otherwise modified.

Example:

```
(%i1) expr: x^2 + x + 1;
                                 2
(%o1)                           x  + x + 1
(%i2) trunc (expr);
                                     2
(%o2)                        1 + x + x  + . . .
(%i3) is (expr = trunc (expr));
(%o3)                              true
```

unsum (*f*, *n*) Function

Returns the first backward difference `f (n) - f (n - 1)`. Thus `unsum` in a sense is the inverse of `sum`.

See also `nusum`.

Examples:

```
(%i1) g(p) := p*4^n/binomial(2*n,n);
                                           n
                                        p 4
(%o1)               g(p) := ----------------
                                binomial(2 n, n)
(%i2) g(n^4);
                              4  n
                             n  4
(%o2)                   ----------------
                        binomial(2 n, n)
(%i3) nusum (%, n, 0, n);
                     4        3       2              n
      2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4      2
(%o3) ------------------------------------------------ - ------
                    693 binomial(2 n, n)                 3 11 7
(%i4) unsum (%, n);
                              4  n
                             n  4
(%o4)                   ----------------
                        binomial(2 n, n)
```

verbose Option variable

Default value: `false`

When `verbose` is `true`, `powerseries` prints progress messages.

33 Number Theory

33.1 Definitions for Number Theory

bern (*n*) Function

Returns the *n*'th Bernoulli number for integer *n*. Bernoulli numbers equal to zero are suppressed if `zerobern` is `false`.

See also `burn`.

```
(%i1) zerobern: true$
(%i2) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
                  1  1       1      1        1
(%o2)       [1, - -, -, 0, - --, 0, --, 0, - --]
                  2  6       30     42       30
(%i3) zerobern: false$
(%i4) map (bern, [0, 1, 2, 3, 4, 5, 6, 7, 8]);
            1  1    1   5     691   7    3617  43867
(%o4) [1, - -, -, - --, --, - ----, -, - ----, -----]
            2  6    30  66    2730  6    510    798
```

bernpoly (*x*, *n*) Function

Returns the *n*'th Bernoulli polynomial in the variable *x*.

bfzeta (*s*, *n*) Function

Returns the Riemann zeta function for the argument *s*. The return value is a big float (bfloat); *n* is the number of digits in the return value.

`load ("bffac")` loads this function.

bfhzeta (*s*, *h*, *n*) Function

Returns the Hurwitz zeta function for the arguments *s* and *h*. The return value is a big float (bfloat); *n* is the number of digits in the return value.

The Hurwitz zeta function is defined as

```
sum ((k+h)^-s, k, 0, inf)
```

`load ("bffac")` loads this function.

binomial (*x*, *y*) Function

The binomial coefficient `(x + y)!/(x! y!)`. If *x* and *y* are integers, then the numerical value of the binomial coefficient is computed. If *y*, or *x - y*, is an integer, the binomial coefficient is expressed as a polynomial.

burn (*n*) Function

Returns the *n*'th Bernoulli number for integer *n*. `burn` may be more efficient than `bern` for large, isolated *n* (perhaps *n* greater than 105 or so), as `bern` computes all the Bernoulli numbers up to index *n* before returning.

`burn` exploits the observation that (rational) Bernoulli numbers can be approximated by (transcendental) zetas with tolerable efficiency.

`load ("bffac")` loads this function.

cf (*expr*) Function

Converts *expr* into a continued fraction. *expr* is an expression comprising continued fractions and square roots of integers. Operands in the expression may be combined with arithmetic operators. Aside from continued fractions and square roots, factors in the expression must be integer or rational numbers. Maxima does not know about operations on continued fractions outside of `cf`.

`cf` evaluates its arguments after binding `listarith` to `false`. `cf` returns a continued fraction, represented as a list.

A continued fraction `a + 1/(b + 1/(c + ...))` is represented by the list `[a, b, c, ...]`. The list elements `a`, `b`, `c`, ... must evaluate to integers. *expr* may also contain `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 `cflength` times the period.

A continued fraction can be evaluated to a number by evaluating the arithmetic representation returned by `cfdisrep`. See also `cfexpand` for another way to evaluate a continued fraction.

See also `cfdisrep`, `cfexpand`, and `cflength`.

Examples:

- *expr* is an expression comprising continued fractions and square roots of integers.

```
(%i1) cf ([5, 3, 1]*[11, 9, 7] + [3, 7]/[4, 3, 2]);
(%o1)               [59, 17, 2, 1, 1, 1, 27]
(%i2) cf ((3/17)*[1, -2, 5]/sqrt(11) + (8/13));
(%o2)        [0, 1, 1, 1, 3, 2, 1, 4, 1, 9, 1, 9, 2]
```

- `cflength` controls how many periods of the continued fraction are computed for algebraic, irrational numbers.

```
(%i1) cflength: 1$
(%i2) cf ((1 + sqrt(5))/2);
(%o2)                    [1, 1, 1, 1, 2]
(%i3) cflength: 2$
(%i4) cf ((1 + sqrt(5))/2);
(%o4)               [1, 1, 1, 1, 1, 1, 1, 2]
(%i5) cflength: 3$
(%i6) cf ((1 + sqrt(5))/2);
(%o6)           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

- A continued fraction can be evaluated by evaluating the arithmetic representation returned by `cfdisrep`.

```
(%i1) cflength: 3$
(%i2) cfdisrep (cf (sqrt (3)))$
(%i3) ev (%, numer);
(%o3)                   1.731707317073171
```

- Maxima does not know about operations on continued fractions outside of `cf`.

```
(%i1) cf ([1,1,1,1,1,2] * 3);
(%o1)                     [4, 1, 5, 2]
(%i2) cf ([1,1,1,1,1,2]) * 3;
(%o2)                  [3, 3, 3, 3, 3, 6]
```

cfdisrep (*list*) Function

Constructs and returns an ordinary arithmetic expression of the form `a + 1/(b + 1/(c + ...))` from the list representation of a continued fraction `[a, b, c, ...]`.

```
(%i1) cf ([1, 2, -3] + [1, -2, 1]);
(%o1)                        [1, 1, 1, 2]
(%i2) cfdisrep (%);
                                  1
(%o2)                   1 + ---------
                                    1
                              1 + -----
                                      1
                                  1 + -
                                      2
```

cfexpand (*x*) Function

Returns a matrix of the numerators and denominators of the last (column 1) and next-to-last (column 2) convergents of the continued fraction *x*.

```
(%i1) cf (rat (ev (%pi, numer)));

`rat' replaced 3.141592653589793 by 103993//33102 = 3.141592653011902
(%o1)                  [3, 7, 15, 1, 292]
(%i2) cfexpand (%);
                         [ 103993  355 ]
(%o2)                    [             ]
                         [ 33102   113 ]
(%i3) %[1,1]/%[2,1], numer;
(%o3)                   3.141592653011902
```

cflength Option variable

Default value: 1

`cflength` controls the number of terms of the continued fraction the function `cf` will give, as the value `cflength` times the period. Thus the default is to give one period.

```
(%i1) cflength: 1$
(%i2) cf ((1 + sqrt(5))/2);
(%o2)                    [1, 1, 1, 1, 2]
(%i3) cflength: 2$
(%i4) cf ((1 + sqrt(5))/2);
(%o4)               [1, 1, 1, 1, 1, 1, 1, 2]
(%i5) cflength: 3$
(%i6) cf ((1 + sqrt(5))/2);
(%o6)           [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

divsum (*n*, *k*) Function
divsum (*n*) Function

`divsum (`*n*`, `*k*`)` returns the sum of the divisors of *n* raised to the *k*'th power.

`divsum (`*n*`)` returns the sum of the divisors of *n*.

```
(%i1) divsum (12);
(%o1)                          28
(%i2) 1 + 2 + 3 + 4 + 6 + 12;
(%o2)                          28
(%i3) divsum (12, 2);
(%o3)                          210
(%i4) 1^2 + 2^2 + 3^2 + 4^2 + 6^2 + 12^2;
(%o4)                          210
```

euler (*n*) Function

Returns the *n*'th Euler number for nonnegative integer *n*.

For the Euler-Mascheroni constant, see `%gamma`.

```
(%i1) map (euler, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1)    [1, 0, - 1, 0, 5, 0, - 61, 0, 1385, 0, - 50521]
```

%gamma Constant

The Euler-Mascheroni constant, 0.5772156649015329 ....

factorial (*x*) Function

Represents the factorial function. Maxima treats `factorial (x)` the same as `x!`. See `!`.

fib (*n*) Function

Returns the *n*'th Fibonacci number. `fib(0)` equal to 0 and `fib(1)` equal to 1, and `fib (-n)` equal to `(-1)^(n + 1) * fib(n)`.

After calling `fib`, `prevfib` is equal to `fib (x - 1)`, the Fibonacci number preceding the last one computed.

```
(%i1) map (fib, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
(%o1)         [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

fibtophi (*expr*) Function

Expresses Fibonacci numbers in terms of the constant `%phi`, which is `(1 + sqrt(5))/2`, approximately 1.61803399.

By default, Maxima does not know about `%phi`. After executing `tellrat (%phi^2 - %phi - 1)` and `algebraic: true`, `ratsimp` can simplify some expressions containing `%phi`.

```
(%i1) fibtophi (fib (n));
                           n             n
                       %phi  - (1 - %phi)
(%o1)                  -------------------
                           2 %phi - 1
(%i2) fib (n-1) + fib (n) - fib (n+1);
(%o2)          - fib(n + 1) + fib(n) + fib(n - 1)
(%i3) ratsimp (fibtophi (%));
(%o3)                           0
```

inrt (*x*, *n*) Function

Returns the integer *n*'th root of the absolute value of *x*.

```
(%i1) l: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], inrt (10^a, 3)), l);
(%o2) [2, 4, 10, 21, 46, 100, 215, 464, 1000, 2154, 4641, 10000]
```

jacobi (*p*, *q*) Function

Returns the Jacobi symbol of *p* and *q*.

```
(%i1) l: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
(%i2) map (lambda ([a], jacobi (a, 9)), l);
(%o2)         [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0]
```

lcm (*expr_1*, ..., *expr_n*) Function

Returns the least common multiple of its arguments. The arguments may be general expressions as well as integers.

`load ("functs")` loads this function.

minfactorial (*expr*) Function

Examines *expr* for occurrences of two factorials which differ by an integer. `minfactorial` then turns one into a polynomial times the other.

```
(%i1) n!/(n+2)!;
                                   n!
(%o1)                           --------
                                (n + 2)!
(%i2) minfactorial (%);
                                    1
(%o2)                        ---------------
                             (n + 1) (n + 2)
```

partfrac (*expr*, *var*) Function

Expands the expression *expr* 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.

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

```
(%o3)                 ----- - ----- + --------
                      x + 2   x + 1          2
                                      (x + 1)
```

primep (*n*) Function

Returns `true` if `n` is a prime, `false` if not.

qunit (*n*) Function

Returns 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`.

```
(%i1) qunit (17);
(%o1)                     sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2)                           1
```

totient (*n*) Function

Returns the number of integers less than or equal to *n* which are relatively prime to *n*.

zerobern Option variable

Default value: `true`

When `zerobern` is `false`, `bern` excludes the Bernoulli numbers which are equal to zero. See `bern`.

zeta (*n*) Function

Returns the Riemann zeta function if *x* is a negative integer, 0, 1, or a positive even number, and returns a noun form `zeta (`*n*`)` for all other arguments, including rational noninteger, floating point, and complex arguments.

See also `bfzeta` and `zeta%pi`.

```
(%i1) map (zeta, [-4, -3, -2, -1, 0, 1, 2, 3, 4, 5]);
                                             2              4
           1         1     1              %pi            %pi
(%o1) [0, ---, 0, - --, - -, inf, ----, zeta(3), ----, zeta(5)]
          120        12    2                6               90
```

zeta%pi Option variable

Default value: `true`

When `zeta%pi` is `true`, `zeta` returns an expression proportional to `%pi^n` for even integer `n`. Otherwise, `zeta` returns a noun form `zeta (n)` for even integer `n`.

```
(%i1) zeta%pi: true$
(%i2) zeta (4);
                                 4
                              %pi
(%o2)                         ----
                               90
```

```
(%i3) zeta%pi: false$
(%i4) zeta (4);
(%o4)                           zeta(4)
```

34 Symmetries

34.1 Definitions for Symmetries

comp2pui (*n, l*) Function

re'alise le passage des fonctions syme'triques comple'tes, donnee's dans la liste *l*, aux fonctions syme'triques e'le'mentaires de 0 a' *n*. Si la liste *l* contient moins de *n*+1 e'le'ments les valeurs formelles viennent la completer. Le premier e'le'ment de la liste *l* donne le cardinal de l'alphabet si il existe, sinon on le met e'gal a *n*.

```
(%i1) comp2pui (3, [4, g]);
                        2                    2
(%o1)    [4, g, 2 h2 - g , 3 h3 - g h2 + g (g  - 2 h2)]
```

cont2part (*pc, lvar*) Function

rend le polyno^me partitionne' associe' a' la forme contracte'e *pc* dont les variables sont dans *lvar*.

```
(%i1) pc: 2*a^3*b*x^4*y + x^5;
                           3    4      5
(%o1)                   2 a  b x  y + x
(%i2) cont2part (pc, [x, y]);
                                   3
(%o2)              [[1, 5, 0], [2 a  b, 4, 1]]
```

Autres fonctions de changements de repre'sentations :

`contract`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

contract (*psym, lvar*) Function

rend une forme contracte'e (i.e. un mono^me par orbite sous l'action du groupe syme'trique) du polyno^me *psym* en les variables contenues dans la liste *lvar*. La fonction `explose` re'alise l'ope'ration inverse. La fonction `tcontract` teste en plus la syme'trie du polyno^me.

```
(%i1) psym: explose (2*a^3*b*x^4*y, [x, y, z]);
         3      4      3      4      3    4        3    4
(%o1) 2 a  b y z  + 2 a  b x z  + 2 a  b y  z + 2 a  b x  z

                                           3      4      3    4
                                    + 2 a  b x y  + 2 a  b x  y
(%i2) contract (psym, [x, y, z]);
                              3    4
(%o2)                      2 a  b x  y
```

Autres fonctions de changements de repre'sentations :

`cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

direct ([*p_1, ..., p_n*], *y, f,* [*lvar_1, ..., lvar_n*]) Function

calcul l'image directe (voir M. GIUSTI, D. LAZARD et A. VALIBOUZE, ISSAC 1988, Rome) associe'e a' la fonction *f*, en les listes de variables *lvar_1*, ..., *lvar_n*, et aux polynoˆmes *p_1*, ..., *p_n* d'une variable *y*. l'arite' de la fonction *f* est importante pour le calcul. Ainsi, si l'expression de *f* ne depend pas d'une variable, non seulement il est inutile de donner cette variable mais cela diminue conside'rablement lees calculs si on ne le fait pas.

```
(%i1) direct ([z^2  - e1* z + e2, z^2  - f1* z + f2],
              z, b*v + a*u, [[u, v], [a, b]]);
       2
(%o1) y  - e1 f1 y

                                 2            2             2   2
               - 4 e2 f2 - (e1  - 2 e2) (f1  - 2 f2) + e1  f1
             + -----------------------------------------------
                                         2
(%i2) ratsimp (%);
              2                2                   2
(%o2)        y  - e1 f1 y + (e1  - 4 e2) f2 + e2 f1
(%i3) ratsimp (direct ([z^3-e1*z^2+e2*z-e3,z^2  - f1* z + f2],
              z, b*v + a*u, [[u, v], [a, b]]));
       6            5         2                        2    2   4
(%o3) y  - 2 e1 f1 y  + ((2 e1  - 6 e2) f2 + (2 e2 + e1 ) f1 ) y

                          3                               3   3
 + ((9 e3 + 5 e1 e2 - 2 e1 ) f1 f2 + (- 2 e3 - 2 e1 e2) f1 ) y

         2       2        4    2
 + ((9 e2  - 6 e1  e2 + e1 ) f2

                    2       2       2                   2    4
 + (- 9 e1 e3 - 6 e2  + 3 e1  e2) f1  f2 + (2 e1 e3 + e2 ) f1 )

  2          2                      2     3          2
 y  + (((9 e1  - 27 e2) e3 + 3 e1 e2  - e1  e2) f1 f2

                 2            2    3              5
 + ((15 e2 - 2 e1 ) e3 - e1 e2 ) f1  f2 - 2 e2 e3 f1 ) y

          2                   3           3     2   2    3
 + (- 27 e3  + (18 e1 e2 - 4 e1 ) e3 - 4 e2  + e1  e2 ) f2

         2      3                   3    2   2
 + (27 e3  + (e1  - 9 e1 e2) e3 + e2 ) f1  f2

                   2    4         2   6
 + (e1 e2 e3 - 9 e3 ) f1  f2 + e3  f1
```

Recherche du polynoˆme dont les racines sont les somme a+u ou a est racine de z^2 - e1* z + e2 et u est racine de z^2 - f1* z + f2

```
(%i1) ratsimp (direct ([z^2  - e1* z + e2, z^2  - f1* z + f2],
                          z, a + u, [[u], [a]]));
```

```
        4                    3             2
(%o1) y  + (- 2 f1 - 2 e1) y  + (2 f2 + f1  + 3 e1 f1 + 2 e2

     2   2                            2               2
 + e1 ) y  + ((- 2 f1 - 2 e1) f2 - e1 f1  + (- 2 e2 - e1 ) f1

                  2                     2            2
 - 2 e1 e2) y + f2  + (e1 f1 - 2 e2 + e1 ) f2 + e2 f1  + e1 e2 f1

     2
 + e2
```

`direct` peut prendre deux drapeaux possibles : `elementaires` et `puissances` (valeur par de'faut) qui permettent de de'composer les polyno^mes syme'triques apparaissant dans ce calcul par les fonctions syme'triques e'le'mentaires 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

passe des fonctions syme'triques e'le'mentaires 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*) Function

donne le polyno^me en *z* dont les fonctions syme'triques e'le'mentaires des racines sont dans la liste *l*. *l* = [*n*, *e_1*, ..., *e_n*] ou' *n* est le degre' du polyno^me et *e_i* la *i*-ie'me fonction syme'trique e'le'mentaire.

```
(%i1) ele2polynome ([2, e1, e2], z);
                          2
(%o1)                    z  - e1 z + e2
(%i2) polynome2ele (x^7 - 14*x^5 + 56*x^3  - 56*x + 22, x);
(%o2)          [7, 0, - 14, 0, 56, 0, - 56, - 22]
(%i3) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
                  7       5       3
(%o3)            x  - 14 x  + 56 x  - 56 x + 22
```

La re'ciproque: `polynome2ele (`*P*`, `*z*`)`

Autres fonctions a' voir :

`polynome2ele`, `pui2polynome`.

ele2pui (*m, l*) Function

passe des fonctions syme'triques e'le'mentaires aux fonctions comple'tes. Similaire a' `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*) Function

de'compose le polyno^me syme'trique *sym*, en les variables contenues de la liste *lvar*, par les fonctions syme'triques e'le'mentaires contenues dans la liste *ele*. Si le premier e'le'ment de *ele* 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 *ele* des valeurs formelles du type "ei" sont rajoute'es. Le polyno^me *sym* peut etre donne' sous 3 formes diffe'rentes : contracte'e (`elem` doit alors valoir 1 sa valeur par de'faut), partitionne'e (`elem` doit alors valoir 3) ou e'tendue (i.e. le polyno^me en entier) (`elem` doit alors valoir 2). L'utilsation de la fonction `pui` se re'alise sur le me^me mode'le.

Sur un alphabet de cardinal 3 avec e1, la premie're fonction syme'trique e'le'mentaire, valant 7, le polyno^me syme'trique en 3 variables dont la forme contracte'e (ne de'pendant ici que de deux de ses variables) est x^4-2*x*y se de'compose ainsi en les fonctions syme'triques e'le'mentaires :

```
(%i1) elem ([3, 7], x^4 - 2*x*y, [x, y]);
(%o1) 7 (e3 - 7 e2 + 7 (49 - e2)) + 21 e3

                                         + (- 2 (49 - e2) - 2) e2
(%i2) ratsimp (%);
                                       2
(%o2)             28 e3 + 2 e2  - 198 e2 + 2401
```

Autres fonctions de changements de bases :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `mon2schur`, `multi_elem`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

explose (*pc, lvar*) Function

rend le polyno^me syme'trique associe' a' la forme contracte'e *pc*. La liste *lvar* contient les variables.

```
(%i1) explose (a*x + 1, [x, y, z]);
(%o1)                    a z + a y + a x + 1
```

Autres fonctions de changements de repre'sentations :

`contract`, `cont2part`, `part2cont`, `partpol`, `tcontract`, `tpartpol`.

kostka (*part_1, part_2*) Function

e'crite par P. ESPERET, calcule le nombre de Kostka associe' aux partition *part_1* et *part_2*.

```
(%i1) kostka ([3, 3, 3], [2, 2, 2, 1, 1, 1]);
(%o1)                              6
```

lgtreillis (*n, m*) Function

rend la liste des partitions de poids *n* et de longueur *m*.

```
(%i1) lgtreillis (4, 2);
(%o1)                   [[3, 1], [2, 2]]
```

Voir e'galement : `ltreillis`, `treillis` et `treinat`.

ltreillis (*n, m*) Function

rend la liste des partitions de poids *n* et de longueur infe'rieure ou e'gale a' *m*.

```
(%i1) ltreillis (4, 2);
(%o1)               [[4, 0], [3, 1], [2, 2]]
```

Voir e'galement : `lgtreillis`, `treillis` et `treinat`.

mon2schur (*l*) Function

la liste *l* repre'sente la fonction de Schur S_*l*: On a *l* = [*i_1*, *i_2*, ..., *i_q*] avec *i_1* <= *i_2* <= ... <= *i_q*. La fonction de Schur est S_[*i_1*, *i_2*, ..., *i_q*] est le mineur de la matrice infinie (h_{i-j}) *i* >= 1, *j* >= 1 compose' des q premie'res lignes et des colonnes *i_1* + 1, *i_2* + 2, ..., *i_q* + *q*.

On e'crit cette fonction de Schur en fonction des formes monomiales en utilisant les fonctions `treinat` et `kostka`. La forme rendue est un polyno^me syme'trique dans une de ses repre'sentations contracte'es avec les variables *x_1*, *x_2*, ...

```
(%i1) mon2schur ([1, 1, 1]);
(%o1)                        x1 x2 x3
(%i2) mon2schur ([3]);
                                  2        3
(%o2)                x1 x2 x3 + x1  x2 + x1
(%i3) mon2schur ([1, 2]);
                                      2
(%o3)                  2 x1 x2 x3 + x1  x2
```

ce qui veut dire que pour 3 variables cela donne :

```
2 x1 x2 x3 + x1^2 x2 + x2^2 x1 + x1^2 x3 + x3^2 x1
 + x2^2 x3 + x3^2 x2
```

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

de'compose un polyno^me multi-syme'trique sous la forme multi-contracte'e *multi_pc* en les groupes de variables contenue dans la liste de listes *l_var* sur les groupes de fonctions syme'triques e'le'mentaires contenues dans *l_elem*.

```
(%i1) multi_elem ([[2, e1, e2], [2, f1, f2]], a*x + a^2 + x^3, [[x, y], [a, b]
                                                    3
(%o1)         - 2 f2 + f1 (f1 + e1) - 3 e1 e2 + e1
(%i2) ratsimp (%);
                         2                       3
(%o2)         - 2 f2 + f1  + e1 f1 - 3 e1 e2 + e1
```

Autres fonctions de changements de bases :

`comp2ele`, `comp2pui`, `ele2comp`, `ele2pui`, `elem`, `mon2schur`, `multi_pui`, `pui`, `pui2comp`, `pui2ele`, `puireduc`, `schur2comp`.

multi_orbit (*P*, [*lvar_1*, *lvar_2*, ..., *lvar_p*]) Function

P est un polyno^me en l'ensemble des variables contenues dans les listes *lvar_1*, *lvar_2*, ..., *lvar_p*. Cette fonction rame'ne l'orbite du polyno^me *P* sous l'action du produit des groupes syme'triques des ensembles de variables repre'sente's par ces *p* listes.

```
(%i1) multi_orbit (a*x + b*y, [[x, y], [a, b]]);
(%o1)                [b y + a x, a y + b x]
(%i2) multi_orbit (x + y + 2*a, [[x, y], [a, b, c]]);
(%o2)         [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 `multi_elem` est a' la fonction `elem`.

```
(%i1) multi_pui ([[2, p1, p2], [2, t1, t2]], a*x + a^2 + x^3, [[x, y], [a, b]]
                                                  3
                                       3 p1 p2   p1
(%o1)               t2 + p1 t1 + ------- - ---
                                          2       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 *i_1*, *i_2*, ..., *i_k*, le re'sultat de `multinomial` est `r!/(i_1! i_2! ... i_k!)`.

multsym (*ppart_1*, *ppart_2*, *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 :

```
(%i1) multsym ([[3, 1], [2, 1, 1]], [[5, 2]], 2);
(%o1)          [[10, 3, 1], [15, 3, 0], [15, 2, 1]]
```

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 polyno^me *P* en les variables de la liste *lvar* sous l'action du groupe syme'trique de l'ensemble des variables contenues dans la liste *lvar*.

```
(%i1) orbit (a*x + b*y, [x, y]);
(%o1)                [a y + b x, b y + a x]
(%i2) orbit (2*x + x^2, [x, y]);
                        2         2
(%o2)                 [y  + 2 y, x  + 2 x]
```

Voir e'galement : `multi_orbit` pour l'action d'un produit de groupes syme'triques sur un polyno^me.

part2cont (*ppart, lvar*) Function

passe de la forme partitionne'e a' la forme contracte'e d'un polyno^me syme'trique. La forme contracte'e est rendue avec les variables contenues dans *lvar*.

```
(%i1) part2cont ([[2*a^3*b, 4, 1]], [x, y]);
                                  3    4
(%o1)                          2 a  b x  y
```

Autres fonctions de changements de repre'sentations :

`contract`, `cont2part`, `explose`, `partpol`, `tcontract`, `tpartpol`.

partpol (*psym, lvar*) Function

psym est un polyno^me syme'trique en les variables de *lvar*. Cette fonction rame'ne sa repre'sentation partitionne'e.

```
(%i1) partpol (-a*(x + y) + 3*x*y, [x, y]);
(%o1)               [[3, 1, 1], [- a, 1, 0]]
```

Autres fonctions de changements de repre'sentations :

`contract`, `cont2part`, `explose`, `part2cont`, `tcontract`, `tpartpol`.

permut (*l*) Function

rame'ne la liste des permutations de la liste *l*.

polynome2ele (*P, x*) Function

donne la liste *l* = `[`*n*`, `*e_1*`, ..., `*e_n*`]` ou' *n* est le degre' du polyno^me *P* en la variable *x* et *e_i* la *i*-ieme fonction syme'trique e'le'mentaire des racines de *P*.

```
(%i1) polynome2ele (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x);
(%o1)          [7, 0, - 14, 0, 56, 0, - 56, - 22]
(%i2) ele2polynome ([7, 0, -14, 0, 56, 0, -56, -22], x);
                   7       5       3
(%o2)             x  - 14 x  + 56 x  - 56 x + 22
```

La re'ciproque : `ele2polynome (`*l*`, `*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 (*l, 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 *l*. Si le premier e'le'ment de *l* 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 *l*, 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.

```
(%i1) pui;
(%o1)                                  1
(%i2) pui ([3, a, b], u*x*y*z, [x, y, z]);
                           2
                       a (a  - b) u   (a b - p3) u
(%o2)                  ------------ - ------------
                            6              3
(%i3) ratsimp (%);
                                           3
                         (2 p3 - 3 a b + a ) u
(%o3)                    ---------------------
                                   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`.

```
(%i1) pui2comp (2, []);
                                           2
                                    p2 + p1
(%o1)                   [2, p1, --------]
                                       2
(%i2) pui2comp (3, [2, a1]);
                                                     2
                                   a1 (p2 + a1 )
                           2  p3 + ------------- + a1 p2
                  p2 + a1                2
(%o2)     [2, a1, --------, --------------------------]
                     2                    3
(%i3) ratsimp (%);
                              2                     3
                     p2 + a1   2 p3 + 3 a1 p2 + a1
(%o3)        [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 *n*-ie'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*.

```
(%i1) pui;
(%o1)                                1
(%i2) kill(labels);
(%o0)                              done
(%i1) polynome2ele (x^3 - 4*x^2 + 5*x - 1, x);
(%o1)                          [3, 4, 5, 1]
(%i2) ele2pui (3, %);
(%o2)                          [3, 4, 6, 7]
(%i3) pui2polynome (x, %);
                             3      2
(%o3)                       x  - 4 x  + 5 x - 1
```

Autres fonctions a' voir : `polynome2ele`, `ele2polynome`.

pui_direct (*orbite, [lvar_1, ..., lvar_n], [d_1, d_2, ..., d_n]*) Function

Soit *f* un polynome en *n* blocs de variables *lvar_1*, ..., *lvar_n*. Soit *c_i* le nombre de variables dans *lvar_i* . Et *SC* le produit des *n* groupes syme'triques de degre' *c_1*, ..., *c_n*. Ce groupe agit naturellement sur *f*. La liste *orbite* est l'orbite, note'e *SC(f)*, de la fonction *f* sous l'action de *SC*. (Cette liste peut e^tre obtenue avec la fonction : `multi_orbit`). Les di sont des entiers tels que *c_1* <= *d_1*, *c_2* <= *d_2*, ..., *c_n* <= *d_n*. Soit *SD* le produit des groupes syme'triques *S_d1* x *S_d2* x ... x *S_dn*.

La fonction `pui_direct` rame'ne les *n* premie'res fonctions puissances de *SD(f)* de'duites des fonctions puissances de *SC(f)* ou' *n* est le cardinal de *SD(f)*.

Le re'sultat est rendue sous forme multi-contracte'e par rapport a *SD*. i.e. on ne conserve qu'un e'le'ment par orbite sous l'action de *SD*).

```
(%i1) l: [[x, y], [a, b]];
(%o1)                         [[x, y], [a, b]]
(%i2) pui_direct (multi_orbit (a*x + b*y, l), l, [2, 2]);
                                               2  2
(%o2)                   [a x, 4 a b x y + a  x ]
(%i3) pui_direct (multi_orbit (a*x + b*y, l), l, [3, 2]);
                                     2  2     2    2        3  3
(%o3) [2 a x, 4 a b x y + 2 a  x , 3 a  b x  y + 2 a  x ,

    2  2  2  2      3    3        4  4
12 a  b  x  y  + 4 a  b x  y + 2 a  x ,

    3  2  3  2      4    4        5  5
10 a  b  x  y  + 5 a  b x  y + 2 a  x ,

    3  3  3  3       4  2  4  2      5    5        6  6
40 a  b  x  y  + 15 a  b  x  y  + 6 a  b x  y + 2 a  x ]
```

```
(%i4) pui_direct ([y + x + 2*c, y + x + 2*b, y + x + 2*a], [[x, y], [a, b, c]]
                                      2             2
(%o4) [3 x + 2 a, 6 x y + 3 x  + 4 a x + 4 a ,

                 2                   3       2       2        3
                9 x  y + 12 a x y + 3 x  + 6 a x  + 12 a  x + 8 a ]
```

puireduc (*n*, *lpui*) Function

lpui est une liste dont le premier e'le'ment est un entier *m*. `puireduc` donne les *n* premie'res fonctions puissances en fonction des *m* premie'res.

```
(%i1) puireduc (3, [2]);
                                         2
                                   p1 (p1  - p2)
(%o1)          [2, p1, p2, p1 p2 - -------------]
                                         2
(%i2) ratsimp (%);
                                           3
                               3 p1 p2 - p1
(%o2)              [2, p1, p2, -------------]
                                     2
```

resolvante (*P*, *x*, *f*, [*x_1*, ..., *x_d*]) Function

calcule la re'solvante du polyno^me *P* de la variable *x* et de degre' *n* `>=` *d* par la fonction *f* exprime'e en les variables *x_1*, ..., *x_d*. Il est important pour l'efficacite' des calculs de ne pas mettre dans la liste `[`*x_1*`, ..., `*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 ade'quates soient utilise's :

Si la fonction *f* est unitaire :

- un polyno^me 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)

  ```
  (x1*x2 + x2*x3 + x3*x4 + x4*x5 + x5*x1 -
       (x1*x3 + x3*x5 + x5*x2 + x2*x4 + x4*x1))^2
  ```

  generale,

le drapeau de `resolvante` pourra e^tre respectivement :

- unitaire,
- lineaire,
- alternee,
- somme,

- produit,
- cayley,
- generale.

```
(%i1) resolvante: unitaire$
(%i2) resolvante (x^7 - 14*x^5 + 56*x^3 - 56*x + 22, x, x^3 - 1, [x]);

" resolvante unitaire " [7, 0, 28, 0, 168, 0, 1120, - 154, 7840, - 2772, 56448

413952, - 352352, 3076668, - 3363360, 23114112, - 30494464,

175230832, - 267412992, 1338886528, - 2292126760]
  3       6      3       9      6      3
[x  - 1, x  - 2 x  + 1, x  - 3 x  + 3 x  - 1,

 12      9      6      3       15      12       9       6      3
x   - 4 x  + 6 x  - 4 x  + 1, x   - 5 x   + 10 x  - 10 x  + 5 x

       18      15       12       9       6      3
 - 1, x   - 6 x   + 15 x   - 20 x  + 15 x  - 6 x  + 1,

 21      18       15       12       9       6      3
x   - 7 x   + 21 x   - 35 x   + 35 x  - 21 x  + 7 x  - 1]
[- 7, 1127, - 6139, 431767, - 5472047, 201692519, - 3603982011]
       7      6        5         4          3           2
(%o2) y  + 7 y  - 539 y  - 1841 y  + 51443 y  + 315133 y

                                              + 376999 y + 125253
(%i3) resolvante: lineaire$
(%i4) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante lineaire "
       24       20         16            12             8
(%o4) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                    4
                                       + 344489984 y  + 655360000
(%i5) resolvante: general$
(%i6) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3]);

" resolvante generale "
       24       20         16            12             8
(%o6) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                    4
                                       + 344489984 y  + 655360000
(%i7) resolvante (x^4 - 1, x, x1 + 2*x2 + 3*x3, [x1, x2, x3, x4]);

" resolvante generale "
```

```
      24       20         16             12              8
(%o7) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                    4
                                        + 344489984 y  + 655360000
(%i8) direct ([x^4 - 1], x, x1 + 2*x2 + 3*x3, [[x1, x2, x3]]);
      24       20         16             12              8
(%o8) y   + 80 y   + 7520 y   + 1107200 y   + 49475840 y

                                                    4
                                        + 344489984 y  + 655360000
(%i9) resolvante :lineaire$
(%i10) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante lineaire "
                             4
(%o10)                      y  - 1
(%i11) resolvante: symetrique$
(%i12) resolvante (x^4 - 1, x, x1 + x2 + x3, [x1, x2, x3]);

" resolvante symetrique "
                             4
(%o12)                      y  - 1
(%i13) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante symetrique "
                          6      2
(%o13)                   y  - 4 y  - 1
(%i14) resolvante: alternee$
(%i15) resolvante (x^4 + x + 1, x, x1 - x2, [x1, x2]);

" resolvante alternee "
           12      8       6        4        2
(%o15)     y   + 8 y  + 26 y  - 112 y  + 216 y  + 229
(%i16) resolvante: produit$
(%i17) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante produit "
       35      33         29        28         27        26
(%o17) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

        24           23           22            21           20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

         19           18             17              15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

            14               12              11             10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y
```

```
               9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

             5            3
 - 3720087 y  + 26040609 y  + 14348907
(%i18) resolvante: symetrique$
(%i19) resolvante (x^7 - 7*x + 3, x, x1*x2*x3, [x1, x2, x3]);

" resolvante symetrique "
        35      33          29         28          27         26
(%o19) y   - 7 y   - 1029 y   + 135 y   + 7203 y   - 756 y

         24            23           22             21            20
 + 1323 y   + 352947 y   - 46305 y   - 2463339 y   + 324135 y

          19           18              17               15
 - 30618 y   - 453789 y   - 40246444 y   + 282225202 y

             14               12              11             10
 - 44274492 y   + 155098503 y   + 12252303 y   + 2893401 y

               9            8            7             6
 - 171532242 y  + 6751269 y  + 2657205 y  - 94517766 y

             5            3
 - 3720087 y  + 26040609 y  + 14348907
(%i20) resolvante: cayley$
(%i21) resolvante (x^5 - 4*x^2 + x + 1, x, a, []);

" resolvante de Cayley "
        6       5         4          3            2
(%o21) 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_produit_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 $x_1x_2\ldots x_{n/2}+x_{n/2+1}\ldotsx_n$

Voir e'galement :

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante` , `resolvante_klein`, `resolvante_klein3`, `resolvante_vierer`, `resolvante_diedrale`, `resolvante_alternee1`.

```
(%i1) resolvante_bipartite (x^6 + 108, x);
               10        8           6             4
(%o1)         y   - 972 y  + 314928 y  - 34012224 y
```

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 $x_1 x_2 + x_3 x_4$.

```
(%i1) resolvante_diedrale (x^5 - 3*x^4 + 1, x);
        15       12       11       10        9         8         7
(%o1) x   - 21 x   - 81 x   - 21 x   + 207 x  + 1134 x  + 2331 x

        6         5          4          3          2
 - 945 x  - 4970 x  - 18333 x  - 29079 x  - 20745 x  - 25326 x

 - 697
```

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 $x_1 x_2 x_4 + x_4$.

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 $x_1 x_2 x_4 + x_4$.

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 polyn\^ome $P(x)$.

```
(%i1) resolvante_produit_sym (x^5 + 3*x^4 + 2*x - 1, x);
        5      4             10      8       7       6       5
(%o1) [y  + 3 y  + 2 y - 1, y   - 2 y  - 21 y  - 31 y  - 14 y

    4       3      2       10      8       7    6       5       4
 - y  + 14 y  + 3 y  + 1, y   + 3 y  + 14 y  - y  - 14 y  - 31 y

       3      2       5      4
 - 21 y  - 2 y  + 1, y  - 2 y  - 3 y - 1, y - 1]
(%i2) resolvante: produit$
(%i3) resolvante (x^5 + 3*x^4 + 2*x - 1, x, a*b*c, [a, b, c]);

" resolvante produit "
       10      8       7    6        5       4       3     2
(%o3) 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_produit_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_1 x_2 - x_3 x_4*.

Voir e'galement :

`resolvante_produit_sym`, `resolvante_unitaire`, `resolvante_alternee1`, `resolvante_klein`, `resolvante_klein3`, `resolvante`, `resolvante_diedrale`.

schur2comp (*P*, *l_var*) Function

P est un polyno^mes 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 ie'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.

```
(%i1) schur2comp (h1*h2 - h3, [h1, h2, h3]);
(%o1)                        s
                              1, 2
(%i2) schur2comp (a*h3, [h3]);
(%o2)                        s  a
                              3
```

somrac (*l, k*) Function

la liste *l* 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 repre'sentations :

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tpartpol`.

tpartpol (*pol, lvar*) Function

teste si le polyno^me 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 repre'sentations :

`contract`, `cont2part`, `explose`, `part2cont`, `partpol`, `tcontract`.

treillis (*n*) Function

rame'ne toutes les partitions de poids *n*.

```
(%i1) treillis (4);
(%o1)    [[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

Voir e'galement : `lgtreillis`, `ltreillis` et `treinat`.

treinat (*part*) Function

rame'ne la liste des partitions infe'rieures a' la partition *part* pour l'ordre naturel.

```
(%i1) treinat ([5]);
(%o1)                         [[5]]
(%i2) treinat ([1, 1, 1, 1, 1]);
(%o2) [[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],

                                                 [1, 1, 1, 1, 1]]
(%i3) treinat ([3, 2]);
(%o3)                 [[5], [4, 1], [3, 2]]
```

Voir e'galement : `lgtreillis`, `ltreillis` et `treillis`.

35 Groups

35.1 Definitions for Groups

todd_coxeter (*relations, subgroup*) Function
todd_coxeter (*relations*) 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,Hg2,Hg3,...]. 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 ith variable. `mulcoset(coset,i) := table[varnum][coset];`

Example:

```
(%i1) 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>
(%o1) 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)
(%i2) p(i,j) := concat(x,i).concat(x,j);
(%o2)        p(i, j) := concat(x, i) . concat(x, j)
(%i3) symet(5);
         <2>           <3>    <2>           <2>           <3>
(%o3) [x1   , (x1 . x2)   , x2   , (x1 . x3)   , (x2 . x3)   ,

            <2>           <2>           <2>           <3>    <2>
          x3   , (x1 . x4)   , (x2 . x4)   , (x3 . x4)   , x4   ]
(%i4) todd_coxeter(%o3);

Rows tried 426
(%o4)                               120
(%i5) todd_coxeter(%o3,[x1]);

Rows tried 213
(%o5)                               60
(%i6) todd_coxeter(%o3,[x1,x2]);

Rows tried 71
(%o6)                               20
```

```
(%i7) table:todd_coxeter_state[2]$
(%i8) table[1];
(%o8) {Array: (SIGNED-BYTE 30) #(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 0

  0 0 0)}
```

Note only the elements 1 thru 20 of this array `%o8` are meaningful. `table[1][4] = 7` indicates coset4.var1 = coset7

36 Runtime Environment

36.1 Introduction for Runtime Environment

`maxima-init.mac` is a file which is loaded automatically when Maxima starts. You can use `maxima-init.mac` to customize your Maxima environment. `maxima-init.mac`, if it exists, is typically placed in the directory named by `:lisp (default-userdir)`, although it can be in any directory searched by the function `file_search`.

Here is an example `maxima-init.mac` file:

```
setup_autoload ("specfun.mac", ultraspherical, assoc_legendre_p);
showtime:all;
```

In this example, `setup_autoload` tells Maxima to load the specified file (`specfun.mac`) if any of the functions (`ultraspherical`, `assoc_legendre_p`) are called but not yet defined. Thus you needn't remember to load the file before calling the functions.

The statement `showtime: all` tells Maxima to set the `showtime` variable. The `maxima-init.mac` file can contain any other assignments or other Maxima statements.

36.2 Interrupts

The user can stop a time-consuming computation with the ^C (control-C) character. The default action is to stop the computation and print another user prompt. In this case, it is not possible to restart a stopped computation.

If the variable `*debugger-hook*` is set to `nil`, by executing

```
:lisp (setq *debugger-hook* nil)
```

then upon receiving ^C, Maxima will enter the Lisp debugger, and the user may use the debugger to inspect the Lisp environment. The stopped computation can be restarted by entering `continue` in the Lisp debugger. The means of returning to Maxima from the Lisp debugger (other than running the computation to completion) is different for each version of Lisp.

On Unix systems, the character ^Z (control-Z) causes Maxima to stop altogether, and control is returned to the shell prompt. The `fg` command causes Maxima to resume from the point at which it was stopped.

36.3 Definitions for Runtime Environment

feature Declaration

Maxima understands two distinct types of features, system features and features which apply to mathematical expressions. See also `status` for information about system features. See also `features` and `featurep` for information about mathematical features.

`feature` itself is not the name of a function or variable.

featurep (*a*, *f*) Function

Attempts to determine whether the object *a* has the feature *f* on the basis of the facts in the current database. If so, it returns `true`, else `false`.

Note that `featurep` returns `false` when neither *f* nor the negation of *f* can be established.

`featurep` evaluates its argument.

See also `declare` and `features`.

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                           true
```

room () Function
room (*true*) Function
room (*false*) Function

Prints out a description of the state of storage and stack management in Maxima. `room` calls the Lisp function of the same name.

- `room ()` prints out a moderate description.
- `room (true)` prints out a verbose description.
- `room (false)` prints out a terse description.

status (*feature*) Function
status (*feature*, *putative_feature*) Function
status (*status*) Function

Returns information about the presence or absence of certain system-dependent features.

- `status (feature)` returns a list of system features. These include Lisp version, operating system type, etc. The list may vary from one Lisp type to another.
- `status (feature,` *putative_feature*`)` returns `true` if *putative_feature* is on the list of items returned by `status (feature)` and `false` otherwise. `status` quotes the argument *putative_feature*. The double single quotes operator, `''`, defeats the quotation. A feature whose name contains a special character, such as a hyphen, must be given as a string argument. For example, `status (feature, "ansi-cl")`.
- `status (status)` returns a two-element list `[feature, status]`. `feature` and `status` are the two arguments accepted by the `status` function; it is unclear if this list has additional significance.

The variable `features` contains a list of features which apply to mathematical expressions. See `features` and `featurep` for more information.

time (*%o1*, *%o2*, *%o3*, *...*) Function

Returns a list of the times, in seconds, taken to compute the output lines `%o1`, `%o2`, `%o3`, ... The time returned is Maxima's estimate of the internal computation time, not the elapsed time. `time` can only be applied to output line variables; for any other variables, `time` returns `unknown`.

Set `showtime: true` to make Maxima print out the computation time and elapsed time with each output line.

37 Miscellaneous Options

37.1 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.

37.2 Share

The Maxima "share" directory contains programs and other files of interest to Maxima users, but not part of the core implementation of Maxima. These programs are typically loaded via `load` or `setup_autoload`.

`:lisp *maxima-sharedir*` displays the location of the share directory within the user's file system.

`printfile ("share.usg")` prints an out-of-date list of share packages. Users may find it more informative to browse the share directory using a file system browser.

37.3 Definitions for Miscellaneous Options

aliases System variable

Default value: `[]`

`aliases` is the list of atoms which have a user defined alias (set up by the `alias`, `ordergreat`, `orderless` functions or by declaring the atom a `noun` with `declare`).

alphabetic Declaration

`declare (`*char*`, alphabetic)` adds `char` to Maxima's alphabet, which initially contains the letters `A` through `Z`, `a` through `z`, `%` and `_`. *char* is specified as a string of length 1, e.g., `"~"`.

```
(%i1) declare ("~", alphabetic);
(%o1)                          done
(%i2) foo~bar;
(%o2)                        foo~bar
(%i3) atom (foo~bar);
(%o3)                          true
```

apropos (*string*) Function

Searches for Maxima names which have *string* appearing anywhere within them. Thus, `apropos (exp)` returns a list of all the flags and functions which have `exp` as part of their names, such as `expand`, `exp`, and `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. Similarily, 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 (*expr*) Function

Returns the list of arguments of `expr`, which may be any kind of expression other than an atom. Only the arguments of the top-level operator are extracted; subexpressions of `expr` appear as elements or subexpressions of elements of the list of arguments.

The order of the items in the list may depend on the global flag `inflag`.

`args (`*expr*`)` is equivalent to `substpart ("[", `*expr*`, 0)`. See also `substpart`.

See also `op`.

genindex Option variable

Default value: `i`

`genindex` is the alphabetic prefix used to generate the next variable of summation when necessary.

gensumnum Option variable

Default value: 0

`gensumnum` 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 Constant

Real positive infinity.

infinity Constant

Complex infinity, an infinite magnitude of arbitrary phase angle. See also `inf` and `minf`.

infolists System variable

Default value: `[]`

`infolists` is a list of the names of all of the information lists in Maxima. These are:

`labels` - all bound `%i`, `%o`, and `%t` labels.

`values` - all bound atoms which are user variables, not Maxima options or switches, created by `:` or `::` or functional binding.

`functions` - all user-defined functions, created by `:=`.

`arrays` - declared and undeclared arrays, created by `:`, `::`, or `:=`.

`macros` - any macros defined by the user.

`myoptions` - all options ever reset by the user (whether or not they are later reset to their default values).

`rules` - user-defined pattern matching and simplification rules, created by `tellsimp`, `tellsimpafter`, `defmatch`, or `defrule`.

`aliases` - atoms which have a user-defined alias, created by the `alias`, `ordergreat`, `orderless` functions or by declaring the atom as a `noun` with `declare`.

`dependencies` - atoms which have functional dependencies, created by the `depends` or `gradef` functions.

`gradefs` - functions which have user-defined derivatives, created 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 (*expr*) Function

Returns `true` if *expr* is a literal numeric integer, otherwise `false`.

`integerp` returns false if its argument is a symbol, even if the argument is declared integer.

Examples:

```
(%i1) integerp (0);
(%o1)                           true
(%i2) integerp (1);
(%o2)                           true
(%i3) integerp (-17);
(%o3)                           true
(%i4) integerp (0.0);
(%o4)                           false
(%i5) integerp (1.0);
(%o5)                           false
(%i6) integerp (%pi);
(%o6)                           false
(%i7) integerp (n);
(%o7)                           false
(%i8) declare (n, integer);
(%o8)                           done
(%i9) integerp (n);
(%o9)                           false
```

m1pbranch Option variable

Default value: `false`

`m1pbranch` is the principal branch for `-1` to a power. Quantities such as `(-1)^(1/3)` (that is, an "odd" rational exponent) and `(-1)^(1/4)` (that is, an "even" rational exponent) are handled as follows:

```
                   domain:real

(-1)^(1/3):      -1
(-1)^(1/4):   (-1)^(1/4)

                  domain:complex
m1pbranch:false          m1pbranch:true
(-1)^(1/3)               1/2+%i*sqrt(3)/2
(-1)^(1/4)              sqrt(2)/2+%i*sqrt(2)/2
```

numberp (*expr*) Function

Returns `true` if *expr* is a literal integer, rational number, floating point number, or bigfloat, otherwise `false`.

`numberp` returns false if its argument is a symbol, even if the argument is a symbolic number such as `%pi` or `%i`, or declared to be even, odd, integer, rational, irrational, real, imaginary, or complex.

Examples:

```
(%i1) numberp (42);
(%o1)                          true
(%i2) numberp (-13/19);
(%o2)                          true
(%i3) numberp (3.14159);
(%o3)                          true
(%i4) numberp (-1729b-4);
(%o4)                          true
(%i5) map (numberp, [%e, %pi, %i, %phi, inf, minf]);
(%o5)      [false, false, false, false, false, false]
(%i6) declare (a, even, b, odd, c, integer, d, rational,
     e, irrational, f, real, g, imaginary, h, complex);
(%o6)                          done
(%i7) map (numberp, [a, b, c, d, e, f, g, h]);
(%o7) [false, false, false, false, false, false, false, false]
```

properties (*a*) Function

Returns a list of the names of all the properties associated with the atom *a*.

props Special symbol

`props` are 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

Returns a list of those atoms on the `props` list which have the property indicated by *prop*. Thus `propvars (atvalue)` returns a list of atoms which have atvalues.

put (*atom, value, indicator*) Function

Assigns *value* to the property (specified by *indicator*) of *atom*. *indicator* may be the name of any property, not just a system-defined property.

`put` evaluates its arguments. `put` returns *value*.

Examples:

```
(%i1) put (foo, (a+b)^5, expr);
                                   5
(%o1)                       (b + a)
(%i2) put (foo, "Hello", str);
(%o2)                          Hello
(%i3) properties (foo);
```

```
(%o3)             [[user properties, str, expr]]
(%i4) get (foo, expr);
                                         5
(%o4)                              (b + a)
(%i5) get (foo, str);
(%o5)                               Hello
```

qput (*atom, value, indicator*) Function

Assigns *value* to the property (specified by *indicator*) of *atom*. This is the same as `put`, except that the arguments are quoted.

Example:

```
(%i1) foo: aa$
(%i2) bar: bb$
(%i3) baz: cc$
(%i4) put (foo, bar, baz);
(%o4)                               bb
(%i5) properties (aa);
(%o5)                 [[user properties, cc]]
(%i6) get (aa, cc);
(%o6)                               bb
(%i7) qput (foo, bar, baz);
(%o7)                               bar
(%i8) properties (foo);
(%o8)             [value, [user properties, baz]]
(%i9) get ('foo, 'baz);
(%o9)                               bar
```

rem (*atom, indicator*) Function

Removes the property indicated by *indicator* from *atom*.

remove (*a_1, p_1, ..., a_n, p_n*) Function
remove ([*a_1, ..., a_m*], [*p_1, ..., p_n*], ...) Function
remove ("a", *operator*) Function
remove (*a, transfun*) Function
remove (*all, p*) Function

Removes properties associated with atoms.

`remove (`*a_1*`, `*p_1*`, ..., `*a_n*`, `*p_n*`)` removes property `p_k` from atom `a_k`.

`remove ([`*a_1*`, ..., `*a_m*`], [`*p_1*`, ..., `*p_n*`], ...)` removes properties *p_1*, ..., *p_n* from atoms *a_1*, ..., *a_m*. There may be more than one pair of lists.

`remove (all, `*p*`)` removes the property *p* from all atoms which have it.

The removed properties may be system-defined properties such as `function` or `mode_declare`, or user-defined properties.

A property may be `transfun` to remove the translated Lisp version of a function. After executing this, the Maxima version of the function is executed rather than the translated version.

`remove ("a", operator)` or, equivalently, `remove ("a", op)` removes from *a* the operator properties declared by `prefix`, `infix`, `nary`, `postfix`, `matchfix`, or `nofix`. Note that the name of the operator must be written as a quoted string.

`remove` always returns `done` whether or not an atom has a specified property. This behavior is unlike the more specific remove functions `remvalue`, `remarray`, `remfunction`, and `remrule`.

remvalue (*name_1, ..., name_n*) Function
remvalue (*all*) Function

Removes the values of user variables *name_1*, ..., *name_n* (which can be subscripted) from the system.

`remvalue (all)` removes the values of all variables in `values`, the list of all variables given names by the user (as opposed to those which are automatically assigned by Maxima).

See also `values`.

rncombine (*expr*) Function

Transforms *expr* by combining all terms of *expr* that have identical denominators or denominators that differ from each other by numerical factors only. This is slightly different from the behavior of `combine`, which collects terms that have identical denominators.

Setting `pfeformat: true` and using `combine` yields results similar to those that can be obtained with `rncombine`, but `rncombine` takes the additional step of cross-multiplying numerical denominator factors. This results in neater forms, and the possiblity of recognizing some cancellations.

scalarp (*expr*) Function

Returns `true` if *expr* is a number, constant, or variable declared `scalar` with `declare`, or composed entirely of numbers, constants, and such variables, but not containing matrices or lists.

setup_autoload (*filename, function_1, ..., function_n*) Function

Specifies that if any of *function_1*, ..., *function_n* are referenced and not yet defined, *filename* is loaded via `load`. *filename* usually contains definitions for the functions specified, although that is not enforced.

`setup_autoload` does not work for array functions.

`setup_autoload` quotes its arguments.

Example:

```
(%i1) legendre_p (1, %pi);
(%o1)                  legendre_p(1, %pi)
(%i2) setup_autoload ("specfun.mac", legendre_p, ultraspherical);
(%o2)                          done
(%i3) ultraspherical (2, 1/2, %pi);
Warning - you are redefining the Macsyma function ultraspherical
Warning - you are redefining the Macsyma function legendre_p
```

```
                               2
                    3 (%pi - 1)
(%o3)               ------------ + 3 (%pi - 1) + 1
                         2
(%i4) legendre_p (1, %pi);
(%o4)                          %pi
(%i5) legendre_q (1, %pi);
                              %pi + 1
                     %pi log(-------)
                              1 - %pi
(%o5)                ---------------- - 1
                            2
```

38 Rules and Patterns

38.1 Introduction to Rules and Patterns

This section describes user-defined pattern matching and simplification rules. There are two groups of functions which implement somewhat different pattern matching schemes. In one group are `tellsimp`, `tellsimpafter`, `defmatch`, `defrule`, `apply1`, `applyb1`, and `apply2`. In the other group are `let` and `letsimp`. Both schemes define patterns in terms of pattern variables declared by `matchdeclare`.

Pattern-matching rules defined by `tellsimp` and `tellsimpafter` are applied automatically by the Maxima simplifier. Rules defined by `defmatch`, `defrule`, and `let` are applied by an explicit function call.

There are additional mechanisms for rules applied to polynomials by `tellrat`, and for commutative and noncommutative algebra in `affine` package.

38.2 Definitions for Rules and Patterns

apply1 (*expr, rule_1, ..., rule_n*) Function

Repeatedly applies *rule_1* to *expr* until it fails, then repeatedly applies the same rule to all subexpressions of *expr*, left to right, until *rule_1* has failed on all subexpressions. Call the result of transforming *expr* in this manner *expr_2*. Then *rule_2* is applied in the same fashion starting at the top of *expr_2*. When *rule_n* fails on the final subexpression, the result is returned.

`maxapplydepth` is the depth of the deepest subexpressions processed by `apply1` and `apply2`.

See also `applyb1`, `apply2`, and `let`.

apply2 (*expr, rule_1, ..., rule_n*) Function

If *rule_1* fails on a given subexpression, then *rule_2* is repeatedly applied, etc. Only if all rules 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` is the depth of the deepest subexpressions processed by `apply1` and `apply2`.

See also `apply1` and `let`.

applyb1 (*expr, rule_1, ..., rule_n*) Function

Repeatedly applies *rule_1* to the deepest subexpression of *expr* until it fails, then repeatedly applies the same rule one level higher (i.e., larger subexpressions), until *rule_1* has failed on the top-level expression. Then *rule_2* is applied in the same fashion to the result of *rule_1*. After *rule_n* has been applied to the top-level expression, the result is returned.

`applyb1` is similar to `apply1` but works from the bottom up instead of from the top down.

maxapplyheight is the maximum height which applyb1 reaches before giving up.

See also apply1, apply2, and let.

current_let_rule_package Option variable

Default value: default_let_rule_package

current_let_rule_package is the name of the rule package that is used by functions in the let package (letsimp, etc.) if no other rule package is specified. This variable may be assigned the name of any rule package defined via the let command.

If a call such as letsimp (expr, rule_pkg_name) is made, the rule package rule_pkg_name is used for that function call only, and the value of current_let_rule_package is not changed.

default_let_rule_package Option variable

Default value: 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 with let or by changing the value of current_let_rule_package.

defmatch (*progname, pattern, x_1, ..., x_n*) Function

Creates a function *progname* (*expr*, *y_1*, ..., *y_n*) which tests *expr* to see if it matches *pattern*.

pattern is an expression containing the pattern variables *x_1*, ..., *x_n* and pattern parameters, if any. The pattern variables are given explicitly as arguments to defmatch while the pattern parameters are declared by the matchdeclare function.

The first argument to the created function *progname* is an expression to be matched against the pattern and the other arguments are the actual variables *y_1*, ..., *y_n* in the expression which correspond to the dummy variables *x_1*, ..., *x_n* in the pattern.

If the match is successful, *progname* returns a list of equations whose left sides are the pattern variables and pattern parameters, and whose right sides are the expressions which the pattern variables and parameters matched. The pattern parameters, but not the variables, are assigned the subexpressions they match. If the match fails, *progname* returns false.

Any variables not declared as pattern parameters in matchdeclare or as variables in defmatch match only themselves.

A pattern which contains no pattern variables or parameters returns true if the match succeeds.

See also matchdeclare, defrule, tellsimp, and tellsimpafter.

Examples:

This defmatch defines the function linearp (expr, y), which tests expr to see if it is of the form a*y + b such that a and b do not contain y.

```
(%i1) matchdeclare (a, freeof(x), b, freeof(x))$
(%i2) defmatch (linearp, a*x + b, x)$
(%i3) linearp (3*z + (y+1)*z + y^2, z);
                                   2
```

```
(%o3)              [b = y , a = y + 4, x = z]
(%i4) a;
(%o4)                        y + 4
(%i5) b;
                               2
(%o5)                         y
```

If the third argument to `defmatch` in line (%i2) had been omitted, then `linear` would only match expressions linear in *x*, not in any other variable.

```
(%i1) matchdeclare ([a, f], true)$
(%i2) constinterval (l, h) := constantp (h - l)$
(%i3) matchdeclare (b, constinterval (a))$
(%i4) matchdeclare (x, atom)$
(%i5) (remove (integrate, outative),
          defmatch (checklimits, 'integrate (f, x, a, b)),
          declare (integrate, outative))$
(%i6) 'integrate (sin(t), t, %pi + x, 2*%pi + x);
                        x + 2 %pi
                       /
                       [
(%o6)                  I          sin(t) dt
                       ]
                       /
                        x + %pi
(%i7) checklimits (%);
(%o7)    [b = x + 2 %pi, a = x + %pi, x = t, f = sin(t)]
(%i8) a;
(%o8)                       x + %pi
(%i9) b;
(%o9)                      x + 2 %pi
(%i10) f;
(%o10)                      sin(t)
(%i11) x;
(%o11)                         t
```

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 `apply1`, `applyb1`, or `apply2`), 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 transform an expression by one operation of the pattern match and replacement. If the match fails, the original expression is returned.

disprule (*rulename_1, ..., rulename_2*) Function
disprule (*all*) Function

Display rules with the names *rulename_1*, ..., *rulename_n*, as returned by `defrule`, `tellsimp`, or `tellsimpafter`, or a pattern defined by `defmatch`.

For example, the first rule modifying `sin` is named `sinrule1`.

`disprule (all)` displays all rules.

See also `letrules`, which displays rules defined by `let`.

let (*prod, repl, predname, arg_1, ..., arg_n*) Function

let ([*prod, repl, predname, arg_1, ..., arg_n*], *package_name*) Function

Defines a substitution rule for `letsimp` such that *prod* is replaced by *repl*. *prod* is a product of positive or negative powers of the following terms:

- 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.
- 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. 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 *prod* the switch `letrat` must be set to `true`. See also `letrat`.

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) is accepted only if `predname (arg_1', ..., arg_n')` evaluates to `true` where *arg_i'* is the value matched to *arg_i*. The *arg_i* 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 are made.

The global flag `letrat` controls the simplification of quotients by `letsimp`. When `letrat` is `false`, `letsimp` simplifies the numerator and denominator of *expr* separately, and does not simplify the quotient. Substitutions such as `n!/n` goes to `(n-1)!` then fail. When `letrat` is `true`, then the numerator, denominator, and the quotient are simplified in that order.

These substitution functions allow you to work with several rule packages at once. Each rule package can contain any number of `let` rules and is referenced by a user-defined name. `let ([`*prod*`, `*repl*`, `*predname*`, `*arg_1*`, ..., `*arg_n*`], `*package_name*`)` adds the rule *predname* to the rule package *package_name*. `letsimp (`*expr*`, `*package_name*`)` applies the rules in *package_name*. `letsimp (`*expr*`, `*package_name1*`, `*package_name2*`, ...)` is equivalent to `letsimp (`*expr*`, `*package_name1*`)` followed by `letsimp (%, `*package_name2*`)`, ...

`current_let_rule_package` is the name of the rule package that is presently being used. This variable may be assigned the name of any rule package defined via the `let` command. Whenever any of the functions comprising the `let` package are called with no package name, the package named by `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, and `current_let_rule_package` is not changed. If not otherwise specified, `current_let_rule_package` defaults to `default_let_rule_package`.

```
(%i1) matchdeclare ([a, a1, a2], true)$
(%i2) oneless (x, y) := is (x = y-1)$
(%i3) let (a1*a2!, a1!, oneless, a2, a1);
(%o3)         a1 a2! --> a1! where oneless(a2, a1)
(%i4) letrat: true$
(%i5) let (a1!/a1, (a1-1)!);
                        a1!
(%o5)                   --- --> (a1 - 1)!
                        a1
(%i6) letsimp (n*m!*(n-1)!/m);
(%o6)                      (m - 1)! n!
(%i7) let (sin(a)^2, 1 - cos(a)^2);
                        2               2
(%o7)                sin (a) --> 1 - cos (a)
(%i8) letsimp (sin(x)^4);
                        4         2
(%o8)                cos (x) - 2 cos (x) + 1
```

letrat Option variable

Default value: `false`

When `letrat` is `false`, `letsimp` simplifies the numerator and denominator of a ratio separately, and does not simplify the quotient.

When `letrat` is `true`, the numerator, denominator, and their quotient are simplified in that order.

```
(%i1) matchdeclare (n, true)$
(%i2) let (n!/n, (n-1)!);
                         n!
(%o2)                    -- --> (n - 1)!
                         n
(%i3) letrat: false$
(%i4) letsimp (a!/a);
                               a!
(%o4)                          --
                               a
(%i5) letrat: true$
(%i6) letsimp (a!/a);
(%o6)                       (a - 1)!
```

letrules () Function

letrules (*package_name*) Function

Displays the rules in a rule package. `letrules ()` displays the rules in the current rule package. `letrules (`*package_name*`)` displays the rules in `package_name`.

The current rule package is named by `current_let_rule_package`. If not otherwise specified, `current_let_rule_package` defaults to `default_let_rule_package`.

See also `disprule`, which displays rules defined by `tellsimp` and `tellsimpafter`.

letsimp (*expr*) Function
letsimp (*expr, package_name*) Function
letsimp (*expr, package_name_1, ..., package_name_n*) Function

Repeatedly applies the substitution rules defined by `let` until no further change is made to *expr*.

`letsimp (`*expr*`)` uses the rules from `current_let_rule_package`.

`letsimp (`*expr*`,` *package_name*`)` uses the rules from *package_name* without changing `current_let_rule_package`.

`letsimp (`*expr*`,` *package_name_1*`, ...,` *package_name_n*`)` is equivalent to `letsimp (`*expr*`,` *package_name_1*, followed by `letsimp (%,` *package_name_2*`)`, and so on.

let_rule_packages Option variable

Default value: `[default_let_rule_package]`

`let_rule_packages` is a list of all user-defined let rule packages plus the default package `default_let_rule_package`.

matchdeclare (*a_1, pred_1, ..., a_n, pred_n*) Function

Associates a predicate *pred_k* with a variable or list of variables *a_k* so that *a_k* matches expressions for which the predicate returns anything other than `false`.

The predicate is the name of a function, a function call missing the last argument, or `true`. Any expression matches `true`. If the predicate is specified as a function call, the expression to be tested is appended to the list of arguments; the arguments are evaluated at the time the match is evaluated. Otherwise, the predicate is specified as a function name, and the expression to be tested is the sole argument. A predicate function need not be defined when `matchdeclare` is called; the predicate is not evaluated until a match is attempted.

A `matchdeclare` predicate cannot be any kind of expression other than a function name or function call. In particular, a predicate cannot be a `lambda` or `block`.

If an expression satisfies a match predicate, the match variable is assigned the expression, except for match variables which are operands of addition `+` or multiplication `*`. Only addition and multiplication are handled specially; other n-ary operators (both built-in and user-defined) are treated like ordinary functions.

In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions. Such multiple-term matching is greedy: predicates are evaluated in the order in which their associated variables appear in the match pattern, and a term which satisfies more than one predicate is taken by the first predicate which it satisfies. Each predicate is tested against all operands of the sum or product before the next predicate is evaluated. In addition, if 0 or 1 (respectively) satisfies a match predicate, and there are no other terms which satisfy the predicate, 0 or 1 is assigned to the match variable associated with the predicate.

The algorithm for processing addition and multiplication patterns makes some match results (for example, a pattern in which a `"match anything"` variable appears) dependent on the ordering of terms in the match pattern and in the expression to be matched. However, if all match predicates are mutually exclusive, the match result is insensitive to ordering, as one match predicate cannot accept terms matched by another.

Calling `matchdeclare` with a variable *a* as an argument changes the `matchdeclare` property for *a*, if one was already declared; only the most recent `matchdeclare` is in effect when a rule is defined, Later changes to the `matchdeclare` property (via `matchdeclare` or `remove`) do not affect existing rules.

`propvars (matchdeclare)` returns the list of all variables for which there is a `matchdeclare` property. `printprops (`*a*`, matchdeclare)` returns the predicate for variable `a`. `printprops (all, matchdeclare)` returns the list of predicates for all `matchdeclare` variables. `remove (`*a*`, matchdeclare)` removes the `matchdeclare` property from *a*.

The functions `defmatch`, `defrule`, `tellsimp`, `tellsimpafter`, and `let` construct rules which test expressions against patterns.

`matchdeclare` quotes its arguments. `matchdeclare` always returns `done`.

Examples:

- `q` matches an expression not containing `x` or `%e`.

```
(%i1) matchdeclare (q, freeof (x, %e))$
```

matchfix (*ldelimiter, rdelimiter*) Function
matchfix (*ldelimiter, rdelimiter, arg_pos, pos*) Function

Declares a matchfix operator with left and right delimiters *ldelimiter* and *rdelimiter*. The delimiters are specified as strings.

A "matchfix" operator is a function of any number of arguments, such that the arguments occur between matching left and right delimiters. The delimiters may be any strings, so long as the parser can distinguish the delimiters from the operands and other expressions and operators. In practice this rules out unparseable delimiters such as `%`, `,`, `$` and `;`, and may require isolating the delimiters with white space. The right delimiter can be the same or different from the left delimiter.

A left delimiter can be associated with only one right delimiter; two different matchfix operators cannot have the same left delimiter.

An existing operator may be redeclared as a matchfix operator without changing its other properties. In particular, built-in operators such as addition `+` can be declared matchfix, but operator functions cannot be defined for built-in operators.

`matchfix (`*ldelimiter*`, `*rdelimiter*`, `*arg_pos*`, `*pos*`)` declares the argument part-of-speech *arg_pos* and result part-of-speech *pos*, and the delimiters *ldelimiter* and *rdelimiter*.

The function to carry out a matchfix operation is an ordinary user-defined function. The operator function is defined in the usual way with the function definition operator `:=` or `define`. The arguments may be written between the delimiters, or with the left delimiter as a quoted string and the arguments following in parentheses. `dispfun (`*ldelimiter*`)` displays the function definition.

The only built-in matchfix operator is the list constructor `[ ]`. Parentheses `( )` and double-quotes `" "` act like matchfix operators, but are not treated as such by the Maxima parser.

`matchfix` evaluates its arguments. `matchfix` returns its first argument, *ldelimiter*.

Examples:

- Delimiters may be almost any strings.

```
(%i1) matchfix ("@", "~");
(%o1)                          "@"
(%i2) @ a, b, c ~;
(%o2)                        @a, b, c~
(%i3) matchfix (">>", "<<");
(%o3)                         ">>"
(%i4) >> a, b, c <<;
(%o4)                       >>a, b, c<<
(%i5) matchfix ("foo", "oof");
(%o5)                         "foo"
(%i6) foo a, b, c oof;
(%o6)                       fooa, b, coof
(%i7) >> w + foo x, y oof + z << / @ p, q ~;
                     >>z + foox, yoof + w<<
(%o7)                ----------------------
                            @p, q~
```

- Matchfix operators are ordinary user-defined functions.

```
(%i1) matchfix ("!-", "-!");
(%o1)                         "!-"
(%i2) !- x, y -! := x/y - y/x;
                                    x   y
(%o2)                  !-x, y-! := - - -
                                    y   x
(%i3) define (!-x, y-!, x/y - y/x);
                                    x   y
(%o3)                  !-x, y-! := - - -
                                    y   x
(%i4) define ("!-" (x, y), x/y - y/x);
                                    x   y
(%o4)                  !-x, y-! := - - -
                                    y   x
(%i5) dispfun ("!-");
                                    x   y
(%t5)                  !-x, y-! := - - -
                                    y   x

(%o5)                         done
(%i6) !-3, 5-!;
                                16
(%o6)                         - --
                                15
(%i7) "!-" (3, 5);
                                16
(%o7)                         - --
                                15
```

remlet (*prod, name*) Function
remlet () Function
remlet (*all*) Function
remlet (*all, 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 rule package. If the name of a rule package is supplied, e.g. `remlet (all, *name*)`, the rule package *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 (*prod*)` now be called the original substitution rule is revived.

See also `remrule`, which removes a rule defined by `tellsimp` or `tellsimpafter`.

remrule (*op, rulename*) Function
remrule (*op, all*) Function

Removes rules defined by `defrule`, `defmatch`, `tellsimp`, or `tellsimpafter`.

`remrule (*op*, *rulename*)` removes the rule with the name `rulename` from the operator *op*.

`remrule (*function*, all)` removes all rules for the operator *op*.

See also `remlet`, which removes a rule defined by `let`.

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.

`tellsimp` 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 the list of rules defined by `defrule`, `defmatch`, `tellsimp`, and `tellsimpafter`.

Examples:

```
(%i1) matchdeclare (x, freeof (%i));
(%o1)                          done
(%i2) %iargs: false$
(%i3) tellsimp (sin(%i*x), %i*sinh(x));
(%o3)                   [sinrule1, simp-%sin]
(%i4) trigexpand (sin (%i*y + x));
(%o4)          sin(x) cos(%i y) + %i cos(x) sinh(y)
(%i5) %iargs:true$
(%i6) errcatch(0^0);
 0
```

```
0  has been generated
(%o6)                          []
(%i7) ev (tellsimp (0^0, 1), simp: false);
(%o7)                  [^rule1, simpexpt]
(%i8) 0^0;
(%o8)                           1
(%i9) remrule ("^", %th(2)[1]);
(%o9)                           ^
(%i10) tellsimp (sin(x)^2, 1 - cos(x)^2);
(%o10)                 [^rule2, simpexpt]
(%i11) (1 + sin(x))^2;
                                     2
(%o11)                    (sin(x) + 1)
(%i12) expand (%);
                                  2
(%o12)               2 sin(x) - cos (x) + 2
(%i13) sin(x)^2;
                                 2
(%o13)                    1 - cos (x)
(%i14) kill (rules);
(%o14)                        done
(%i15) matchdeclare (a, true);
(%o15)                        done
(%i16) tellsimp (sin(a)^2, 1 - cos(a)^2);
(%o16)                 [^rule3, simpexpt]
(%i17) sin(y)^2;
                                 2
(%o17)                    1 - cos (y)
```

tellsimpafter (*pattern, replacement*) Function

Defines a simplification rule which the Maxima simplifier applies after built-in simplification rules. *pattern* is an expression, comprising pattern variables (declared by `matchdeclare`) and other atoms and operators, considered literals for the purpose of pattern matching. *replacement* is substituted for an actual expression which matches *pattern*; pattern variables in *replacement* are assigned the values matched in the actual expression.

pattern may be any nonatomic expression in which the main operator is not a pattern variable; the simplification rule is associated with the main operator. The names of functions (with one exception, described below), lists, and arrays may appear in *pattern* as the main operator only as literals (not pattern variables); this rules out expressions such as `aa(x)` and `bb[y]` as patterns, if `aa` and `bb` are pattern variables. Names of functions, lists, and arrays which are pattern variables may appear as operators other than the main operator in *pattern*.

There is one exception to the above rule concerning names of functions. The name of a subscripted function in an expression such as `aa[x](y)` may be a pattern variable, because the main operator is not `aa` but rather the Lisp atom `mqapply`. This is a consequence of the representation of expressions involving subscripted functions.

Simplification rules are applied after evaluation (if not suppressed through quotation or the flag `noeval`). Rules established by `tellsimpafter` are applied in the order they were defined, and after any built-in rules. Rules are applied bottom-up, that is, applied first to subexpressions before application to the whole expression. It may be necessary to repeatedly simplify a result (for example, via the quote-quote operator `''` or the flag `infeval`) to ensure that all rules are applied.

Pattern variables are treated as local variables in simplification rules. Once a rule is defined, the value of a pattern variable does not affect the rule, and is not affected by the rule. An assignment to a pattern variable which results from a successful rule match does not affect the current assignment (or lack of it) of the pattern variable. However, as with all atoms in Maxima, the properties of pattern variables (as declared by `put` and related functions) are global.

The rule constructed by `tellsimpafter` is named after the main operator of `pattern`. Rules for built-in operators, and user-defined operators defined by `infix`, `prefix`, `postfix`, `matchfix`, and `nofix`, have names which are Maxima strings. Rules for other functions have names which are ordinary Maxima identifiers.

The treatment of noun and verb forms is slightly confused. If a rule is defined for a noun (or verb) form and a rule for the corresponding verb (or noun) form already exists, the newly-defined rule applies to both forms (noun and verb). If a rule for the corresponding verb (or noun) form does not exist, the newly-defined rule applies only to the noun (or verb) form.

The rule constructed by `tellsimpafter` is an ordinary Lisp function. If the name of the rule is `$foorule1`, the construct `:lisp (trace $foorule1)` traces the function, and `:lisp (symbol-function '$foorule1` displays its definition.

`tellsimpafter` quotes its arguments. `tellsimpafter` returns the list of rules for the main operator of *pattern*, including the newly established rule.

See also `matchdeclare`, `defmatch`, `defrule`, `tellsimp`, `let`, `kill`, `remrule`, and `clear_rules`.

Examples:

pattern may be any nonatomic expression in which the main operator is not a pattern variable.

```
(%i1) matchdeclare (aa, atom, [ll, mm], listp, xx, true)$
(%i2) tellsimpafter (sin (ll), map (sin, ll));
(%o2)                 [sinrule1, simp-%sin]
(%i3) sin ([1/6, 1/4, 1/3, 1/2, 1]*%pi);
                    1  sqrt(2)  sqrt(3)
(%o3)              [-, -------, -------, 1, 0]
                    2     2        2
(%i4) tellsimpafter (ll^mm, map ("^", ll, mm));
(%o4)                  [^rule1, simpexpt]
(%i5) [a, b, c]^[1, 2, 3];
                                 2   3
(%o5)                      [a, b , c ]
(%i6) tellsimpafter (foo (aa (xx)), aa (foo (xx)));
(%o6)                   [foorule1, false]
(%i7) foo (bar (u - v));
```

```
(%o7)                        bar(foo(u - v))
```

Rules are applied in the order they were defined. If two rules can match an expression, the rule which was defined first is applied.

```
(%i1) matchdeclare (aa, integerp);
(%o1)                             done
(%i2) tellsimpafter (foo (aa), bar_1 (aa));
(%o2)                       [foorule1, false]
(%i3) tellsimpafter (foo (aa), bar_2 (aa));
(%o3)                  [foorule2, foorule1, false]
(%i4) foo (42);
(%o4)                           bar_1(42)
```

Pattern variables are treated as local variables in simplification rules. (Compare to `defmatch`, which treats pattern variables as global variables.)

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                             done
(%i2) tellsimpafter (foo(aa, bb), bar('aa=aa, 'bb=bb));
(%o2)                       [foorule1, false]
(%i3) bb: 12345;
(%o3)                             12345
(%i4) foo (42, %e);
(%o4)                   bar(aa = 42, bb = %e)
(%i5) bb;
(%o5)                             12345
```

As with all atoms, properties of pattern variables are global even though values are local. In this example, an assignment property is declared via `define_variable`. This is a property of the atom `bb` throughout Maxima.

```
(%i1) matchdeclare (aa, integerp, bb, atom);
(%o1)                             done
(%i2) tellsimpafter (foo(aa, bb), bar('aa=aa, 'bb=bb));
(%o2)                       [foorule1, false]
(%i3) foo (42, %e);
(%o3)                   bar(aa = 42, bb = %e)
(%i4) define_variable (bb, true, boolean);
(%o4)                             true
(%i5) foo (42, %e);
Error: bb was declared mode boolean, has value: %e
 -- an error.  Quitting.  To debug this try debugmode(true);
```

Rules are named after main operators. Names of rules for built-in and user-defined operators are strings, while names for other functions are ordinary identifiers.

```
(%i1) tellsimpafter (foo (%pi + %e), 3*%pi);
(%o1)                       [foorule1, false]
(%i2) tellsimpafter (foo (%pi * %e), 17*%e);
(%o2)                  [foorule2, foorule1, false]
(%i3) tellsimpafter (foo (%i ^ %e), -42*%i);
(%o3)             [foorule3, foorule2, foorule1, false]
(%i4) tellsimpafter (foo (9) + foo (13), quux (22));
(%o4)                       [+rule1, simplus]
```

```
(%i5) tellsimpafter (foo (9) * foo (13), blurf (22));
(%o5)                  [*rule1, simptimes]
(%i6) tellsimpafter (foo (9) ^ foo (13), mumble (22));
(%o6)                  [^rule1, simpexpt]
(%i7) rules;
(%o7) [trigrule0, trigrule1, trigrule2, trigrule3, trigrule4,
htrigrule1, htrigrule2, htrigrule3, htrigrule4, foorule1,
foorule2, foorule3, +rule1, *rule1, ^rule1]
(%i8) foorule_name: first (%o1);
(%o8)                       foorule1
(%i9) plusrule_name: first (%o4);
(%o9)                        +rule1
(%i10) [?mstringp (foorule_name), symbolp (foorule_name)];
(%o10)                    [false, true]
(%i11) [?mstringp (plusrule_name), symbolp (plusrule_name)];
(%o11)                    [true, true]
(%i12) remrule (foo, foorule1);
(%o12)                         foo
(%i13) remrule ("^", "^rule1");
(%o13)                          ^
```

clear_rules () Function

Executes `kill (rules)` and then resets the next rule number to 1 for addition `+`, multiplication `*`, and exponentiation `^`.

39 Lists

39.1 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))
```

39.2 Definitions for Lists

append (*list_1, ..., list_n*) Function

Returns a single list of the elements of *list_1* followed by the elements of *list_2*, ... `append` also works on general expressions, e.g. `append (f(a,b), f(c,d,e));` yields `f(a,b,c,d,e)`.

Do `example(append);` for an example.

assoc (*key, list, default*) Function
assoc (*key, list*) Function

This function searches for the *key* in the left hand side of the input *list* of the form `[x,y,z,...]` where each of the *list* elements is an expression of a binary operand and 2 elements. For example `x=1`, `2^3`, `[a,b]` etc. The *key* is checked againts the first operand. `assoc` returns the second operand if the `key` is found. If the `key` is not found it either returns the *default* value. *default* is optional and defaults to `false`.

atom (*expr*) Function

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

cons (*expr, list*) Function

Returns a new list constructed of the element *expr* 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 (*list*) Function

Returns a copy of the list *list*.

delete (*expr_1*, *expr_2*) Function
delete (*expr_1*, *expr_2*, *n*) Function

Removes all occurrences of *expr_1* from *expr_2*. *expr_1* may be a term of *expr_2* (if it is a sum) or a factor of *expr_2* (if it is a product).

```
(%i1) delete(sin(x), x+sin(x)+y);
(%o1)                              y + x
```

`delete(`*expr_1*`, `*expr_2*`, `*n*`)` removes the first *n* occurrences of *expr_1* from *expr_2*. If there are fewer than *n* occurrences of *expr_1* in *expr_2* then all occurrences will be deleted.

```
(%i1) delete(a, f(a,b,c,d,a));
(%o1)                          f(b, c, d)
(%i2) delete(a, f(a,b,a,c,d,a), 2);
(%o2)                         f(b, c, d, a)
```

eighth (*expr*) Function

Returns the 8'th item of expression or list *expr*. See `first` for more details.

endcons (*expr*, *list*) Function

Returns a new list consisting of the elements of `list` followed by *expr*. `endcons` also works on general expressions, e.g. `endcons(x, f(a,b,c)); -> f(a,b,c,x)`.

fifth (*expr*) Function

Returns the 5'th item of expression or list *expr*. See `first` for more details.

first (*expr*) Function

Returns the first part of *expr* 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 *expr* which is displayed not the form which is typed on input. If the variable `inflag` is set to `true` however, these functions will look at the internal form of *expr*. 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). The functions `second` .. `tenth` yield the second through the tenth part of their input argument.

fourth (*expr*) Function

Returns the 4'th item of expression or list *expr*. See `first` for more details.

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*.

`get` evaluates its arguments.

```
(%i1) put (%e, 'transcendental, 'type);
(%o1)                        transcendental
(%i2) put (%pi, 'transcendental, 'type)$
```

```
(%i3) put (%i, 'algebraic, 'type)$
(%i4) typeof (expr) := block ([q],
        if numberp (expr)
        then return ('algebraic),
        if not atom (expr)
        then return (maplist ('typeof, expr)),
        q: get (expr, 'type),
        if q=false
        then errcatch (error(expr,"is not numeric.")) else q)$
(%i5) typeof (2*%e + x*%pi);
x is not numeric.
(%o5)  [[transcendental, []], [algebraic, transcendental]]
(%i6) typeof (2*%e + %pi);
(%o6)     [transcendental, [algebraic, transcendental]]
```

last (*expr*) Function

Returns the last part (term, row, element, etc.) of the *expr*.

length (*expr*) Function

Returns (by default) the number of parts in the external (displayed) form of *expr*. 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. 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 Option variable

default value: `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 (*expr*) Function

Returns `true` if *expr* is a list else `false`.

makelist (*expr, i, i_0, i_1*) Function
makelist (*expr, x, list*) Function

Constructs and returns a list, each element of which is generated from *expr*.

`makelist (`*expr*`, `*i*`, `*i_0*`, `*i_1*`)` returns a list, the `j`'th element of which is equal to `ev (`*expr*`, `*i*`=j)` for `j` equal to *i_0* through *i_1*.

`makelist (`*expr*`, `*x*`, `*list*`)` returns a list, the `j`'th element of which is equal to `ev (`*expr*`, `*x*`=`*list*`[j])` for `j` equal to 1 through `length (`*list*`)`.

Examples:

```
(%i1) makelist(concat(x,i),i,1,6);
(%o1)               [x1, x2, x3, x4, x5, x6]
(%i2) makelist(x=y,y,[a,b,c]);
(%o2)                 [x = a, x = b, x = c]
```

member (*expr, list*) Function

Returns `true` if *expr* 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`.

ninth (*expr*) Function

Returns the 9'th item of expression or list *expr*. See `first` for more details.

rest (*expr, n*) Function
rest (*expr*) Function

Returns *expr* 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. *expr* 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`.

second (*expr*) Function

Returns the 2'nd item of expression or list *expr*. See `first` for more details.

seventh (*expr*) Function

Returns the 7'th item of expression or list *expr*. See `first` for more details.

sixth (*expr*) Function

Returns the 6'th item of expression or list *expr*. See `first` for more details.

tenth (*expr*) Function

Returns the 10'th item of expression or list *expr*. See `first` for more details.

third (*expr*) Function

Returns the 3'rd item of expression or list *expr*. See `first` for more details.

40 Sets

40.1 Introduction to Sets

Maxima provides set functions, such as intersection and union, for finite sets that are defined by explicit enumeration. Maxima treats lists and sets as distinct objects. This feature makes it possible to work with sets that have members that are either lists or sets.

In addition to functions for finite sets, Maxima provides some functions related to combinatorics; these include the Stirling numbers, the Bell numbers, and several others.

40.1.1 Usage

To construct a set with members `a_1, ..., a_n`, use the command `set(a_1, ..., a_n)`; to construct the empty set, use `set()`. If a member is listed more than once, the simplification process eliminates the redundant member.

```
(%i1) set();
(%o1)                          {}
(%i2) set(a, b, a);
(%o2)                        {a, b}
(%i3) set(a, set(b));
(%o3)                       {a, {b}}
(%i4) set(a, [b]);
(%o4)                       {a, [b]}
```

Sets are always displayed as brace delimited lists; if you would like to be able to *input* a set using braces, see [Defining sets with braces], page 388.

To construct a set from the elements of a list, use `setify`.

```
(%i1) setify([b, a]);
(%o1)                        {a, b}
```

Set members `x` and `y` are equal provided `is(x = y)` evaluates to true. Thus `rat(x)` and `x` are equal as set members; consequently,

```
(%i1) set(x, rat(x));
(%o1)                          {x}
```

Further, since `is((x-1)*(x+1) = x^2 - 1)` evaluates to false, `(x-1)*(x+1)` and `x^2-1` are distinct set members; thus

```
(%i1) set((x - 1)*(x + 1), x^2 - 1);
                                       2
(%o1)                  {(x - 1) (x + 1), x  - 1}
```

To reduce this set to a singleton set, apply `rat` to each set member:

```
(%i1) set((x - 1)*(x + 1), x^2 - 1);
                                       2
(%o1)                  {(x - 1) (x + 1), x  - 1}
(%i2) map(rat, %);
                              2
(%o2)/R/                    {x  - 1}
```

To remove redundancies from other sets, you may need to use other simplification functions. Here is an example that uses `trigsimp`:

```
(%i1) set(1, cos(x)^2 + sin(x)^2);
                            2         2
(%o1)                {1, sin (x) + cos (x)}
(%i2) map(trigsimp, %);
(%o2)                          {1}
```

A set is simplified when its members are non-redundant and sorted. The current version of the set functions uses the Maxima function `orderlessp` to order sets; however, *future versions of the set functions might use a different ordering function.*

Some operations on sets, such as substitution, automatically force a re-simplification; for example,

```
(%i1) s: set (a, b, c)$
(%i2) subst (c=a, s);
(%o2)                        {a, b}
(%i3) subst ([a=x, b=x, c=x], s);
(%o3)                          {x}
(%i4) map (lambda ([x], x^2), set (-1, 0, 1));
(%o4)                        {0, 1}
```

Maxima treats lists and sets as distinct objects; functions such as `union` and `intersection` will signal an error if any argument is a list. If you need to apply a set function to a list, use the `setify` function to convert it to a set. Thus

```
(%i1) union ([1, 2], set (a, b));
Function union expects a set, instead found [1,2]
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i2) union (setify ([1, 2]), set (a, b));
(%o2)                     {1, 2, a, b}
```

To extract all set elements of a set `s` that satisfy a predicate `f`, use `subset(s,f)`. (A *predicate* is a boolean-valued function.) For example, to find the equations in a given set that do not depend on a variable `z`, use

```
(%i1) subset (set (x + y + z, x - y + 4, x + y - 5), lambda ([e], freeof (z, e)));
(%o1)               {- y + x + 4, y + x - 5}
```

The section Section 40.2 [Definitions for Sets], page 389 has a complete list of the set functions in Maxima.

40.1.2 Set Member Iteration

There two ways to to iterate over set members. One way is the use `map`; for example:

```
(%i1) map (f, set (a, b, c));
(%o1)                  {f(a), f(b), f(c)}
```

The other way is to use `for` *x* `in` *s* `do`

```
(%i1) s: set (a, b, c);
(%o1)                       {a, b, c}
(%i2) for si in s do print (concat (si, 1));
a1
b1
c1
(%o2)                          done
```

The Maxima functions `first` and `rest` work correctly on sets. Applied to a set, `first` returns the first displayed element of a set; which element that is may be implementation-dependent. If `s` is a set, then `rest(s)` is equivalent to `disjoin (first(s), s)`. Currently, there are other Maxima functions that work correctly on sets. In future versions of the set functions, `first` and `rest` may function differently or not at all.

40.1.3 Bugs

The set functions use the Maxima function `orderlessp` to order set members and the (Lisp-level) function `like` to test for set member equality. Both of these functions have known bugs (versions 5.9.2 and earlier) that may manifest if you attempt to use sets with members that are lists or matrices that contain expressions in CRE form. An example is

```
(%i1) set ([x], [rat (x)]);
Maxima encountered a Lisp error:

 CAR: #:X13129 is not a LIST

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
```

This command causes Maxima to halt with an error (the error message depends on which version of Lisp your Maxima uses). Another example is

```
(%i1) setify ([[rat(a)], [rat(b)]]);
Maxima encountered a Lisp error:

 CAR: #:A13129 is not a LIST

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
```

These bugs are caused by bugs in `orderlessp` and `like`; they are not caused by bugs in the set functions. To illustrate, try the commands

```
(%i1) orderlessp ([rat(a)], [rat(b)]);
Maxima encountered a Lisp error:

 CAR: #:B13130 is not a LIST

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i2) is ([rat(a)] = [rat(a)]);
(%o2)                              false
```

Until these bugs are fixed, do not construct sets with members that are lists or matrices containing expressions in CRE form; a set with a member in CRE form, however, shouldn't be a problem:

```
(%i1) set (x, rat (x));
(%o1)                               {x}
```

Maxima's `orderlessp` has another bug that can cause problems with set functions, namely that the ordering predicate `orderlessp` is not transitive. The simplest known example that shows this is

```
(%i1) q: x^2$
(%i2) r: (x + 1)^2$
(%i3) s: x*(x + 2)$
(%i4) orderlessp (q, r);
(%o4)                          true
(%i5) orderlessp (r, s);
(%o5)                          true
(%i6) orderlessp (q, s);
(%o6)                          false
```

This bug can cause trouble will all set functions as well as with Maxima functions in general. It's likely, but not certain, that if all set members are either in CRE form or have been simplified using `ratsimp`, this bug will not manifest.

Maxima's `orderless` and `ordergreat` mechanisms are incompatible with the set functions. If you need to use either `orderless` or `ordergreat`, issue these commands before constructing any sets and do not use the `unorder` command.

You may encounter two other minor bugs. Maxima versions 5.5 and earlier had a bug in the `tex` function that makes the empty set incorrectly translate to TeX; this bug is fixed in the Maxima 5.9.0. Additionally, the `setup_autoload` function in Maxima 5.9.0 is broken; a fix is in the `nset-init.lisp` file located in the directory `maxima/share/contrib/nset`.

Maxima's sign function has a bug that may cause the Kronecker delta function to misbehave; for example:

```
(%i1) kron_delta (1/sqrt(2), sqrt(2)/2);
(%o1)                            0
```

The correct value is 1; the bug is related to the `sign` bug

```
(%i1) sign (1/sqrt(2) - sqrt(2)/2);
(%o1)                           pos
```

If you find something that you think might be a set function bug, please report it to the Maxima bug database. See `bug_report`.

40.1.4 Defining sets with braces

If you'd like to be able to input sets using braces, you may do so by declaring the left brace to be a matchfix operator; this is done using the commands

```
(%i1) matchfix("{","}")$
(%i2) "{" ([a]) := apply (set, a)$
```

Now we can define sets using braces; thus

```
(%i1) matchfix("{","}")$
(%i2) "{" ([a]) := apply (set, a)$
(%i3) {};
(%o3)                           {}
(%i4) {a, {a, b}};
(%o4)                       {a, {a, b}}
```

To always allow this form of set input, place the two commands in lines (%i1) and (%i2) in your `maxima-init.mac` file.

40.1.5 Combinatorial and Miscellaneous Functions

In addition to functions for finite sets, Maxima provides some functions related to combinatorics; these include the Stirling numbers of the first and second kind, the Bell numbers, multinomial coefficients, partitions of nonnegative integers, and a few others. Maxima also defines a Kronecker delta function.

40.1.6 Authors

Stavros Macrakis of Cambridge, Massachusetts and Barton Willis of the University of Nebraska at Kearney (UNK) wrote the Maxima set functions and their documentation.

40.2 Definitions for Sets

adjoin (*x*, *a*) Function

Adjoin *x* to the set *a* and return a set. Thus `adjoin(x, a)` and `union(set(x),a)` are equivalent; however, using `adjoin` may be somewhat faster than using `union`. If *a* isn't a set, signal an error.

```
(%i1) adjoin (c, set (a, b));
(%o1)                         {a, b, c}
(%i2) adjoin (a, set (a, b));
(%o2)                          {a, b}
```

See also `disjoin`.

belln (*n*) Function

For nonnegative integers *n*, return the n-th Bell number. If `s` is a set with `n` members, `belln(n)` is the number of partitions of `s`. For example:

```
(%i1) makelist (belln (i), i, 0, 6);
(%o1)               [1, 1, 2, 5, 15, 52, 203]
(%i2) is (cardinality (set_partitions (set ())) = belln (0));
(%o2)                         true
(%i3) is (cardinality (set_partitions (set (1, 2, 3, 4, 5, 6))) = belln (6));
(%o3)                         true
```

When *n* isn't a nonnegative integer, `belln(n)` doesn't simplify.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)        [belln(x), belln(sqrt(3)), belln(- 9)]
```

The function `belln` threads over equalities, lists, matrices, and sets.

cardinality (*a*) Function

Return the number of distinct elements of the set *a*.

```
(%i1) cardinality (set ());
(%o1)                           0
(%i2) cardinality (set (a, a, b, c));
(%o2)                           3
```

```
(%i3) cardinality (set (a, a, b, c)), simp: false;
(%o3)                           3
```

In line (%o3), we see that cardinality works correctly even when simplification has been turned off.

cartesian_product (*b_1, ... , b_n*) Function

Return a set of lists of the form `[`*x_1*`, ..., `*x_n*`]`, where *x_1* `in` *b_1*, ..., *x_n* `in` *b_n*. Signal an error when any *b_k* isn't a set.

```
(%i1) cartesian_product (set (0, 1));
(%o1)                     {[0], [1]}
(%i2) cartesian_product (set (0, 1), set (0, 1));
(%o2)           {[0, 0], [0, 1], [1, 0], [1, 1]}
(%i3) cartesian_product (set (x), set (y), set (z));
(%o3)                     {[x, y, z]}
(%i4) cartesian_product (set (x), set (-1, 0, 1));
(%o4)              {[x, - 1], [x, 0], [x, 1]}
```

disjoin (*x, a*) Function

Remove *x* from the set *a* and return a set. If *x* isn't a member of *a*, return *a*. Each of the following do the same thing: `disjoin(`*x*`, `*a*`)`, `delete(`*x*`, `*a*`)`, and `setdifference(`*a*`,set(`*x*`))`; however, `disjoin` is generally the fastest way to remove a member from a set. Signal an error if *a* isn't a set.

disjointp (*a, b*) Function

Return `true` if the sets *a* and *b* are disjoint. Signal an error if either *a* or *b* isn't a set.

divisors (*n*) Function

When *n* is a nonzero integer, return the set of its divisors. The set of divisors includes the members 1 and *n*. The divisors of a negative integer are the divisors of its absolute value.

We can verify that 28 is a perfect number.

```
(%i1) s: divisors(28);
(%o1)                 {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)                          28
```

The function divisors works by simplification; you shouldn't need to manually re-evaluate after a substitution. For example:

```
(%i1) divisors (a);
(%o1)                      divisors(a)
(%i2) subst (8, a, %);
(%o2)                     {1, 2, 4, 8}
```

The function divisors threads over equalities, lists, matrices, and sets. Here is an example of threading over a list and an equality.

```
(%i1) divisors ([a, b, c=d]);
(%o1) [divisors(a), divisors(b), divisors(c) = divisors(d)]
```

elementp (*x*, *a*) Function

Return `true` if and only if *x* is a member of the set *a*. Signal an error if *a* isn't a set.

emptyp (*a*) Function

Return `true` if and only if *a* is the empty set or the empty list.

```
(%i1) map (emptyp, [set (), []]);
(%o1)                        [true, true]
(%i2) map (emptyp, [a + b, set (set ()), %pi]);
(%o2)                    [false, false, false]
```

equiv_classes (*s*, *f*) Function

Return a set of the equivalence classes of *s* with respect to the equivalence relation *f*. The function *f* should be a boolean-valued function defined on the cartesian product of *s* with *s*. Further, the function *f* should be an equivalence relation; `equiv_classes`, however, doesn't check that it is.

```
(%i1) equiv_classes (set (a, b, c), lambda ([x, y], is (x=y)));
(%o1)                      {{a}, {b}, {c}}
```

Actually, `equiv_classes (s, f)` automatically applies the Maxima function `is` after applying the function *f*; accordingly, we can restate the previous example more briefly.

```
(%i1) equiv_classes (set (a, b, c), "=");
(%o1)                      {{a}, {b}, {c}}
```

Here is another example.

```
(%i1) equiv_classes (set (1, 2, 3, 4, 5, 6, 7), lambda ([x, y], remainder (x -
(%o1)                {{1, 4, 7}, {2, 5}, {3, 6}}
```

every (*f*, *a*) Function

every (*f*, *L_1*, ..., *L_n*) Function

The first argument *f* should be a predicate (a function that evaluates to true, false, or unknown).

Given one set as the second argument, `every (f, a)` returns `true` if `f(a_i)` returns `true` for all *a_i* in *a*. Since sets are unordered, `every` is free to evaluate `f(a_i)` in any order. `every` may or may not evaluate *f* for all *a_i* in *a*. Because the order of evaluation isn't specified, the predicate *f* should not have side-effects or signal errors for any input.

Given one or more lists as arguments, `every (f, L_1, ..., L_n)` returns `true` if `f(x_1, ..., x_n)` returns `true` for all *x_1*, ..., *x_n* in *L_1*, ..., *L_n*, respectively. `every` may or may not evaluate *f* for every combination *x_1*, ..., *x_n*. Since lists are ordered, `every` evaluates in the order of increasing index.

To use `every` on multiple set arguments, they should first be converted to an ordered sequence so that their relative alignment becomes well-defined.

If the global flag `maperror` is `true` (the default), all lists *L_1*, ..., *L_n* must have equal lengths – otherwise, `every` signals an error. When `maperror` is false, the list arguments are effectively truncated each to the length of the shortest list.

The Maxima function `is` automatically applied after evaluating the predicate *f*.

```
(%i1) every ("=", [a, b], [a, b]);
(%o1)                          true
(%i2) every ("#", [a, b], [a, b]);
(%o2)                          false
```

extremal_subset (*s*, *f*, *max*) Function

extremal_subset (*s*, *f*, *min*) Function

When the third argument is max, return the subset of the set or list *s* for which the real-valued function *f* takes on its greatest value; when the third argument is min, return the subset for which *f* takes on its least value.

```
(%i1) extremal_subset (set (-2, -1, 0, 1, 2), abs, max);
(%o1)                        {- 2, 2}
(%i2) extremal_subset (set (sqrt(2), 1.57, %pi/2), sin, min);
(%o2)                        {sqrt(2)}
```

flatten (*e*) Function

Flatten essentially evaluates an expression as if its main operator had been declared n-ary; there is, however, one difference – flatten doesn't recurse into other function arguments. For example:

```
(%i1) expr: flatten (f (g (f (f (x)))));
(%o1)                       f(g(f(f(x))))
(%i2) declare (f, nary);
(%o2)                           done
(%i3) ev (expr);
(%o3)                        f(g(f(x)))
```

Applied to a set, flatten gathers all members of set elements that are sets; for example:

```
(%i1) flatten (set (a, set (b), set (set (c))));
(%o1)                        {a, b, c}
(%i2) flatten (set (a, set ([a], set (a))));
(%o2)                        {a, [a]}
```

Flatten works correctly when the main operator is a subscripted function

```
(%i1) flatten (f[5] (f[5] (x)));
(%o1)                          f (x)
                                5
```

To flatten an expression, the main operator must be defined for zero or more arguments; if this isn't the case, Maxima will halt with an error. Expressions with special representations, for example CRE expressions, can't be flattened; in this case, flatten returns its argument unchanged.

full_listify (*a*) Function

If *a* is a set, convert *a* to a list and apply `full_listify` to each list element.

To convert just the top-level operator of a set to a list, see [listify], page 394.

fullsetify (*a*) Function

If *a* is a list, convert *a* to a set and apply `fullsetify` to each set member.

```
(%i1) fullsetify ([a, [a]]);
(%o1)                       {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2)                      {a, f([b])}
```

In line (%o2), the argument of `f` isn't converted to a set because the main operator of `f([b])` isn't a list.

To convert just the top-level operator of a list to a set, see [setify], page 398.

identity (*x*) Function

The identity function evaluates to its argument for all inputs. To determine if every member of a set is `true`, you can use

```
(%i1) every (identity, [true, true]);
(%o1)                         true
```

integer_partitions (*n*) Function
integer_partitions (*n, len*) Function

If the optional second argument *len* isn't specified, return the set of all partitions of the integer *n*. When *len* is specified, return all partitions that have length *len* or less; in this case, zeros are appended to each partition with fewer than *len* terms to make each partition have exactly *len* terms. In either case, each partition is a list sorted from greatest to least.

We say a list $[a_1, ..., a_m]$ is a partition of a nonnegative integer n provided (1) each a_i is a nonzero integer and (2) $a_1 + ... + a_m = n$. Thus 0 has no partitions.

```
(%i1) integer_partitions (3);
(%o1)               {[1, 1, 1], [2, 1], [3]}
(%i2) s: integer_partitions (25)$
(%i3) cardinality (s);
(%o3)                         1958
(%i4) map (lambda ([x], apply ("+", x)), s);
(%o4)                         {25}
(%i5) integer_partitions (5, 3);
(%o5) {[2, 2, 1], [3, 1, 1], [3, 2, 0], [4, 1, 0], [5, 0, 0]}
(%i6) integer_partitions (5, 2);
(%o6)               {[3, 2], [4, 1], [5, 0]}
```

To find all partitions that satisfy a condition, use the function `subset`; here is an example that finds all partitions of 10 that consist of prime numbers.

```
(%i1) s: integer_partitions (10)$
(%i2) xprimep(x) := integerp(x) and (x > 1) and primep(x)$
(%i3) subset (s, lambda ([x], every (xprimep, x)));
(%o3) {[2, 2, 2, 2, 2], [3, 3, 2, 2], [5, 3, 2], [5, 5], [7, 3]}
```

(Notice that `primep(1)` is true in Maxima. This disagrees with most definitions of prime.)

intersect (*a_1*, ..., *a_n*) Function

Return a set containing the elements that are common to the sets *a_1* through *a_n*. The function `intersect` must receive one or more arguments. Signal an error if any of *a_1* through *a_n* isn't a set. See also [intersection], page 394.

intersection (*a_1*, ..., *a_n*) Function

Return a set containing the elements that are common to the sets *a_1* through *a_n*. The function `intersection` must receive one or more arguments. Signal an error if any of *a_1* through *a_n* isn't a set. See also [intersect], page 393.

kron_delta (*x*, *y*) Function

The Kronecker delta function; `kron_delta (x, y)` simplifies to 1 when `is(x = y)` is true and it simplifies to zero when `sign (|x - y|)` is `pos`. When `sign (|x - y|)` is zero and `x - y` isn't a floating point number (neither a double nor a bfloat), return 0. Otherwise, return a noun form.

The function, `kron_delta` is declared to be symmetric; thus, for example, `kron_delta(x, y) - kron_delta(y, x)` simplifies to zero.

Here are a few examples.

```
(%i1) [kron_delta (a, a), kron_delta (a + 1, a)];
(%o1)                           [1, 0]
(%i2) kron_delta (a, b);
(%o2)                      kron_delta(a, b)
```

Assuming that `a > b` makes `sign (|a - b|)` evaluate to `pos`; thus

```
(%i1) assume (a > b)$
(%i2) kron_delta (a, b);
(%o2)                              0
```

If we instead assume that `x >= y`, then `sign (|x - y|)` evaluates to `pz`; in this case, `kron_delta (x, y)` doesn't simplify

```
(%i1) assume(x >= y)$
(%i2) kron_delta (x, y);
(%o2)                      kron_delta(x, y)
```

Finally, since `1/10 - 0.1` evaluates to a floating point number, we have

```
(%i1) kron_delta (1/10, 0.1);
                                        1
(%o1)                     kron_delta(--, 0.1)
                                        10
```

If you want `kron_delta (1/10, 0.1)` to evaluate to 1, apply `float`.

```
(%i1) float (kron_delta (1/10, 0.1));
(%o1)                              1
```

listify (*a*) Function

If *a* is a set, return a list containing the members of *a*; when *a* isn't a set, return *a*. To convert a set and all of its members to lists, see [full_listify], page 392.

lreduce (*f, s*) Function

lreduce (*f, s, init*) Function

The function `lreduce` (left reduce) extends a 2-arity function to an n-arity function by composition; an example should make this clear. When the optional argument *init* isn't defined, we have

```
(%i1) lreduce (f, [1, 2, 3]);
(%o1)                    f(f(1, 2), 3)
(%i2) lreduce (f, [1, 2, 3, 4]);
(%o2)                 f(f(f(1, 2), 3), 4)
```

Notice that the function *f* is first applied to the `leftmost` list elements (thus the name lreduce). When *init* is defined, the second argument to the inner most function evaluation is *init*; for example:

```
(%i1) lreduce (f, [1, 2, 3], 4);
(%o1)                 f(f(f(4, 1), 2), 3)
```

The function `lreduce` makes it easy to find the product or sum of the elements of a list.

```
(%i1) lreduce ("+", args (set (a, b)));
(%o1)                        b + a
(%i2) lreduce ("*", args (set (1, 2, 3, 4, 5)));
(%o2)                         120
```

See also See [rreduce], page 397, See [xreduce], page 401, and See [tree_reduce], page 401.

makeset (*e, v, s*) Function

This function is similar to `makelist`, but `makeset` allows multiple substitutions. The first argument *e* is an expression; the second argument *v* is a list of variables; and *s* is a list or set of values for the variables *v*. Each member of *s* must have the same length as *v*. We have `makeset (`*e*`, `*v*`, `*s*`)` is the set `{z | z = substitute(v -> s_i) and s_i in s}`.

```
(%i1) makeset (i/j, [i, j], [[a, b], [c, d]]);
                                a  c
(%o1)                          {-, -}
                                b  d
(%i2) ind: set (0, 1, 2, 3)$
(%i3) makeset (i^2 + j^2 + k^2, [i, j, k], cartesian_product (ind, ind, ind));
(%o3) {0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 17, 18,
                                                        19, 22, 27}
```

moebius (*n*) Function

The Moebius function; when *n* is product of *k* distinct primes, `moebius(`*n*`)` evaluates to $(-1)^k$; it evaluates to 1 when $n = 1$; and it evaluates to 0 for all other positive integers. The Moebius function threads over equalities, lists, matrices, and sets.

multinomial_coeff (*a_1, ..., a_n*) Function
multinomial_coeff () Function

Return the multinomial coefficient. When each *a_k* is a nonnegative integer, the multinomial coefficient gives the number of ways of placing *a_1* `+ ... +` *a_n* distinct objects into *n* boxes with *a_k* elements in the *k*'th box. In general, `multinomial (`*a_1*`, ...,` *a_n*`)` evaluates to `(`*a_1* `+ ... +` *a_n*`)!/(`*a_1*`! ...` *a_n*`!)`. Given no arguments, `multinomial()` evaluates to 1. A user may use `minfactorial` to simplify the value returned by `multinomial_coeff`; for example:

```
(%i1) multinomial_coeff (1, 2, x);
                                (x + 3)!
(%o1)                           --------
                                  2 x!
(%i2) minfactorial (%);
                         (x + 1) (x + 2) (x + 3)
(%o2)                    -----------------------
                                   2
(%i3) multinomial_coeff (-6, 2);
                                 (- 4)!
(%o3)                           --------
                                2 (- 6)!
(%i4) minfactorial (%);
(%o4)                              10
```

num_distinct_partitions (*n*) Function
num_distinct_partitions (*n, a*) Function

When *n* is a nonnegative integer, return the number of distinct integer partitions of *n*.

If the optional parameter *a* has the value `list`, return a list of the number of distinct partitions of 1,2,3, ... , n. If *n* isn't a nonnegative integer, return a noun form.

Definition: If $n = k_1 + ... + k_m$, where k_1 through k_m are distinct positive integers, we call $k_1 + ... + k_m$ a distinct partition of *n*.

num_partitions (*n*) Function
num_partitions (*n, a*) Function

When *n* is a nonnegative integer, return the number of partitions of *n*. If the optional parameter *a* has the value `list`, return a list of the number of partitions of 1,2,3, ... , n. If *n* isn't a nonnegative integer, return a noun form.

```
(%i1) num_partitions (5) = cardinality (integer_partitions (5));
(%o1)                         7 = 7
(%i2) num_partitions (8, list);
(%o2)            [1, 1, 2, 3, 5, 7, 11, 15, 22]
(%i3) num_partitions (n);
(%o3)                   num_partitions(n)
```

For a nonnegative integer *n*, `num_partitions (`*n*`)` is equal to `cardinality (integer_partitions (`*n*`))`; however, calling `num_partitions` is much faster.

partition_set (*a*, *f*) Function

Return a list of two sets; the first set is the subset of *a* for which the predicate *f* evaluates to false and the second is the subset of *a* for which *f* evaluates to true. If *a* isn't a set, signal an error. See also [subset], page 400.

```
(%i1) partition_set (set (2, 7, 1, 8, 2, 8), evenp);
(%o1)                    [{1, 7}, {2, 8}]
(%i2) partition_set (set (x, rat(y), rat(y) + z, 1), lambda ([x], ratp(x)));
(%o2)/R/               [{1, x}, {y, y + z}]
```

permutations (*a*) Function

Return a set of all *distinct* permutations of the members of the list or set *a*. (Each permutation is a list, not a set.) When *a* is a list, duplicate members of *a* are *not* deleted before finding the permutations. Thus

```
(%i1) permutations ([a, a]);
(%o1)                         {[a, a]}
(%i2) permutations ([a, a, b]);
(%o2)           {[a, a, b], [a, b, a], [b, a, a]}
```

If *a* isn't a list or set, signal an error.

powerset (*a*) Function

powerset (*a*, *n*) Function

When the optional second argument *n* isn't defined, return the set of all subsets of the set *a*. `powerset(a)` has `2^cardinality(a)` members. Given a second argument, `powerset(a,n)` returns the set of all subsets of *a* that have cardinality *n*. Signal an error if *a* isn't a set; additionally signal an error if *n* isn't a positive integer.

rreduce (*f*, *s*) Function

rreduce (*f*, *s*, *init*) Function

The function `rreduce` (right reduce) extends a 2-arity function to an n-arity function by composition; an example should make this clear. When the optional argument *init* isn't defined, we have

```
(%i1) rreduce (f, [1, 2, 3]);
(%o1)                      f(1, f(2, 3))
(%i2) rreduce (f, [1, 2, 3, 4]);
(%o2)                   f(1, f(2, f(3, 4)))
```

Notice that the function *f* is first applied to the rightmost list elements (thus the name rreduce). When *init* is defined, the second argument to the inner most function evaluation is *init*; for example:

```
(%i1) rreduce (f, [1, 2, 3], 4);
(%o1)                   f(1, f(2, f(3, 4)))
```

The function `rreduce` makes it easy to find the product or sum of the elements of a list.

```
(%i1) rreduce ("+", args (set (a, b)));
(%o1)                          b + a
```

```
(%i2) rreduce ("*", args (set (1, 2, 3, 4, 5)));
(%o2)                            120
```

See also See [lreduce], page 394, See [tree_reduce], page 401, and See [xreduce], page 401.

setdifference (*a*, *b*) Function

Return a set containing the elements in the set *a* that are not in the set *b*. Signal an error if *a* or *b* is not a set.

setify (*a*) Function

Construct a set from the elements of the list *a*. Duplicate elements of the list *a* are deleted and the elements are sorted according to the predicate `orderlessp`. Signal an error if `a` isn't a list.

setp (*a*) Function

Return true if and only if *a* is a Maxima set. The function `setp` checks that the operator of its argument is set; it doesn't check that its argument is a *simplified* set. Thus

```
(%i1) setp (set (a, a)), simp: false;
(%o1)                            true
```

The function `setp` could be coded in Maxima as `setp(a) := is (inpart (a, 0) = set)`.

set_partitions (*a*) Function
set_partitions (*a*, *n*) Function

When the optional argument *n* is defined, return a set of all decompositions of *a* into *n nonempty* disjoint subsets. When *n* isn't defined, return the set of all partitions.

We say a set *P* is a partition of a set *S* provided

1. each member of *P* is a nonempty set,
2. distinct members of *P* are disjoint,
3. the union of the members of *P* equals *S*.

The empty set is a partition of itself (the conditions 1 and 2 being vacuously true); thus

```
(%i1) set_partitions (set ());
(%o1)                            {{}}
```

The cardinality of the set of partitions of a set can be found using `stirling2`; thus

```
(%i1) s: set (0, 1, 2, 3, 4, 5)$
(%i2) p: set_partitions (s, 3)$
(%o3)                          90 = 90
(%i4) cardinality(p) = stirling2 (6, 3);
```

Each member of `p` should have 3 members; let's check.

```
(%i1) s: set (0, 1, 2, 3, 4, 5)$
(%i2) p: set_partitions (s, 3)$
(%o3)                           {3}
(%i4) map (cardinality, p);
```

Finally, for each member of `p`, the union of its members should equal `s`; again let's check.

```
(%i1) s: set (0, 1, 2, 3, 4, 5)$
(%i2) p: set_partitions (s, 3)$
(%o3)                  {{0, 1, 2, 3, 4, 5}}
(%i4) map (lambda ([x], apply (union, listify (x))), p);
```

some (*f*, *a*) Function

some (*f*, *L_1*, ..., *L_n*) Function

The first argument *f* should be a predicate (a function that evaluates to true, false, or unknown).

Given one set as the second argument, `some (`*f*`, `*a*`)` returns `true` if *f*`(`*a_i*`)` returns `true` for at least one *a_i* in *a*. Since sets are unordered, `some` is free to evaluate *f*`(`*a_i*`)` in any order. `some` may or may not evaluate *f* for all *a_i* in *a*. Because the order of evaluation isn't specified, the predicate *f* should not have side-effects or signal errors for any input. To use `some` on multiple set arguments, they should first be converted to an ordered sequence so that their relative alignment becomes well-defined.

Given one or more lists as arguments, `some (`*f*`, `*L_1*`, ..., `*L_n*`)` returns `true` if *f*`(`*x_1*`, ..., `*x_n*`)` returns `true` for at least one *x_1*, ..., *x_n* in *L_1*, ..., *L_n*, respectively. `some` may or may not evaluate *f* for every combination *x_1*, ..., *x_n*. Since lists are ordered, `some` evaluates in the order of increasing index.

If the global flag `maperror` is true (the default), all lists *L_1*, ..., *L_n* must have equal lengths – otherwise, `some` signals an error. When `maperror` is false, the list arguments are effectively truncated each to the length of the shortest list.

The Maxima function `is` is automatically applied after evaluating the predicate *f*.

```
(%i1) some ("<", [a, b, 5], [1, 2, 8]);
(%o1)                          true
(%i2) some ("=", [2, 3], [2, 7]);
(%o2)                          true
```

stirling1 (*n*, *m*) Function

The Stirling number of the first kind. When *n* and *m* are nonnegative integers, the magnitude of `stirling1 (`*n*`, `*m*`)` is the number of permutations of a set with *n* members that have *m* cycles. For details, see Graham, Knuth and Patashnik *Concrete Mathematics*. We use a recursion relation to define `stirling1 (`*n*`, `*m*`)` for *m* less than 0; we do not extend it for *n* less than 0 or for non-integer arguments.

The function `stirling1` works by simplification; it knows the basic special values (see Donald Knuth, *The Art of Computer Programming,* third edition, Volume 1, Section 1.2.6, Equations 48, 49, and 50). For Maxima to apply these rules, the arguments must be declared to be integer and the first argument must nonnegative. For example:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3)                          1
```

`stirling1` does not simplify for non-integer arguments.

```
(%i1) stirling1 (sqrt(2), sqrt(2));
(%o1)              stirling1(sqrt(2), sqrt(2))
```

Maxima knows a few other special values; for example:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n + 1, n);
                            n (n + 1)
(%o3)                       ---------
                                2
(%i4) stirling1 (n + 1, 1);
(%o4)                          n!
```

stirling2 (n, m) Function

The Stirling number of the second kind. When n and m are nonnegative integers, `stirling2 (n, m)` is the number of ways a set with cardinality n can be partitioned into m disjoint subsets. We use a recursion relation to define `stirling2 (n, m)` for m less than 0; we do not extend it for n less than 0 or for non-integer arguments.

The function `stirling2` works by simplification; it knows the basic special values (see Donald Knuth, *The Art of Computer Programming,* third edition, Volume 1, Section 1.2.6, Equations 48, 49, and 50). For Maxima to apply these rules, the arguments must be declared to be integer and the first argument must nonnegative. For example:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3)                          1
```

`stirling2` does not simplify for non-integer arguments.

```
(%i1) stirling2 (%pi, %pi);
(%o1)                  stirling2(%pi, %pi)
```

Maxima knows a few other special values.

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n + 9, n + 8);
                         (n + 8) (n + 9)
(%o3)                    ---------------
                                2
(%i4) stirling2 (n + 1, 2);
                              n
(%o4)                        2  - 1
```

subset (a, f) Function

Return the subset of the set a that satisfies the predicate f. For example:

```
(%i1) subset (set (1, 2, x, x + y, z, x + y + z), atom);
(%o1)                      {1, 2, x, z}
(%i2) subset (set (1, 2, 7, 8, 9, 14), evenp);
(%o2)                       {2, 8, 14}
```

The second argument to `subset` must be a predicate (a boolean-valued function of one argument) if the first argument to `subset` isn't a set, signal an error. See also [partition_set], page 396.

subsetp (*a*, *b*) Function

Return true if and only if the set *a* is a subset of *b*. Signal an error if *a* or *b* is not a set.

symmdifference (*a_1*, ..., *a_n*) Function

Return the set of members that occur in exactly one set *a_k*. Signal an error if any argument *a_k* isn't a set. Given two arguments, `symmdifference (a, b)` is the same as `union (setdifference (a, b), setdifference (b, a))`.

tree_reduce (*f*, *s*) Function

tree_reduce (*f*, *s*, *init*) Function

The function `tree_reduce` extends a associative binary operator $f : SxS- > S$ from two arguments to any number of arguments using a minimum depth tree. An example should make this clear.

```
(%i1) tree_reduce (f, [a, b, c, d]);
(%o1)                  f(f(a, b), f(c, d))
```

Given an odd number of arguments, `tree_reduce` favors the left side of the tree; for example:

```
(%i1) tree_reduce (f, [a, b, c, d, e]);
(%o1)               f(f(f(a, b), f(c, d)), e)
```

For addition of floating point numbers, using `tree_reduce` may give a sum that has a smaller rounding error than using either `rreduce` or `lreduce`.

union (*a_1*, ..., *a_n*) Function

Return the union of the sets *a_1* through *a_n*. When `union` receives no arguments, it returns the empty set. Signal an error when one or more arguments to `union` is not a set.

xreduce (*f*, *s*) Function

xreduce (*f*, *s*, *init*) Function

This function is similar to both `lreduce` and `rreduce` except that `xreduce` is free to use either left or right associativity; in particular when *f* is an associative function and Maxima has a built-in evaluator for it, `xreduce` may use the n-ary function; these n-ary functions include addition `+`, multiplication `*`, `and`, `or`, `max`, `min`, and `append`. For these operators, we generally expect using `xreduce` to be faster than using either `rreduce` or `lreduce`. When *f* isn't n-ary, `xreduce` uses left-associativity.

Floating point addition is not associative; nevertheless, `xreduce` uses Maxima's n-ary addition when the set or list *s* contains floating point numbers.

41 Function Definition

41.1 Introduction to Function Definition

41.2 Function

To define a function in Maxima you use the := operator. E.g.

```
f(x) := sin(x)
```

defines a function `f`. Anonmyous 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:

```
(%i1) f ([u]) := u;
(%o1)                      f([u]) := u
(%i2) f (1, 2, 3, 4);
(%o2)                     [1, 2, 3, 4]
(%i3) f (a, b, [u]) := [a, b, u];
(%o3)               f(a, b, [u]) := [a, b, u]
(%i4) f (1, 2, 3, 4, 5, 6);
(%o4)                 [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`.

41.3 Macros

buildq (*variables, expr*) Function

expr is any single Maxima expression and *variables* is a list of elements of the form `<atom>` or `<atom>: <value>`.

41.3.1 Semantics

The elements of the list *variables* 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 *expr*, then the value associated with that *atom* must be a Maxima list, and it is spliced into *expr* instead of substituted.

41.3.2 Simplification

The arguments to `buildq` need to be protected from simplification until the substitutions have been carried out. This code should effect 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) 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.

```
(%i1) foo (x,y,z) := mshow (x, y, z);
(%i2) 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 ([l]) ::= block ([ans:[], n:length(l)],
    for i:1 thru n do
        (ans: append (ans, buildq ([u: l[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:

```
(%i1) buildq ([a: '[b, c, d]], +splice(a));
(%o1)                           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`.

```
(%i1) buildq ([a: '[b,c,d]], +splice(a));
(%o1)                           d + c + b
(%i2) buildq ([a: '[b,c,d]], splice(a)+splice(a));
(%o2)                       2 d + 2 c + 2 b
```

but

```
(%i3) buildq ([a: '[b,c,d]], 2*splice(a));
(%o3)                            2 b c d
```

Finally `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] := 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.

```
(%i1) val: -((n^2 - 2*n + 1)*f[n-1] + f[n-2] + f[n-3])/(n^2-n)$
(%i2) define (f[n], buildq ([u: val], expand(u)))$
```

does the job. This might be useful, since when you do (with `expand`)

```
(%i3) f[0]: aa0$
(%i4) f[1]: aa1$
(%i5) f[2]: aa2$
(%i6) f[6];
                         3 aa2   aa1   7 aa0
(%o6)                    ----- + --- + -----
                          10     40     90
```

where as without it is kept unsimplified, and even after 6 terms it becomes:

```
(%i7) define (g[n], buildq ([u: val], u))$
(%i8) g[0]: bb0$
(%i9) g[1]: bb1$
(%i10) g[2]: bb2$
(%i11) g[6];
          aa2         7 aa2   aa1   11 aa0    aa1   aa0
          --- - 25 (- ----- - --- - ------) + --- + ---
           4           20     40     120       8    24
(%o11)    ---------------------------------------------
                                 30
(%i12) expand (%);
                         3 aa2   aa1   7 aa0
(%o12)                   ----- + --- + -----
                          10     40     90
```

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.

41.4 Definitions for Function Definition

apply (*f*, [*x_1*, ..., *x_n*]) Function

Returns the result of applying the function *f* to the list of arguments *x_1*, ..., *x_n*. *f* is the name of a function or a lambda expression.

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.

block ([*v_1, ..., v_m*], *expr_1, ..., expr_n*) Function
block (*expr_1, ..., expr_n*) Function

`block` evaluates *expr_1*, ..., *expr_n* in sequence and returns the value of the last expression evaluated. The sequence can be modified by the `go`, `throw`, and `return` functions. The last expression is *expr_n* unless `return` or an expression containing `throw` is evaluated. Some variables *v_1*, ..., *v_m* can be declared local to the block; these are distinguished from global variables of the same names. If no variables are declared local then the list may be omitted. Within the block, any variable other than *v_1*, ..., *v_m* is a global variable.

`block` saves the current values of the variables *v_1*, ..., *v_m* (if any) upon entry to the block, then unbinds the variables so that they evaluate to themselves. The local variables may be bound to arbitrary values within the block but when the block is exited the saved values are restored, and the values assigned within the block are lost.

`block` may appear within another `block`. Local variables are established each time a new `block` is evaluated. Local variables appear to be global to any enclosed blocks. If a variable is non-local in a block, its value is the value most recently assigned by an enclosing block, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of "dynamic scope".

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 (*expr_1, ..., expr_n*) Function

Evaluates and prints *expr_1*, ..., *expr_n* and then causes a Maxima break at which point the user can examine and change his environment. Upon typing `exit;` the computation resumes.

catch (*expr_1, ..., expr_n*) Function

Evaluates *expr_1*, ..., *expr_n* one by one; if any leads to the evaluation of an expression of the form `throw (arg)`, then the value of the `catch` is the value of `throw (arg)`, and no further expressions are evaluated. This "non-local return" thus goes through any depth of nesting to the nearest enclosing `catch`. If there is no `catch` enclosing a `throw`, an error message is printed.

If the evaluation of the arguments does not lead to the evaluation of any `throw` then the value of `catch` is the value of *expr_n*.

```
(%i1) lambda ([x], if x < 0 then throw(x) else f(x))$
(%i2) g(l) := catch (map (''%, l))$
(%i3) g ([1, 2, 3, 7]);
(%o3)              [f(1), f(2), f(3), f(7)]
(%i4) g ([1, 2, -3, 7]);
(%o4)                          - 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.

compfile (*filename, f_1, ..., f_n*) Function

Translates Maxima functions *f_1*, ..., *f_n* into Lisp and writes the translated code into the file *filename*.

The Lisp translations are not evaluated, nor is the output file processed by the Lisp compiler. `translate` creates and evaluates Lisp translations. `compile_file` translates Maxima into Lisp, and then executes the Lisp compiler.

See also `translate`, `translate_file`, and `compile_file`.

compile (*f_1, ..., f_n*) Function
compile (*functions*) Function
compile (*all*) Function

Translates Maxima functions *f_1*, ..., *f_n* into Lisp, evaluates the Lisp translations, and calls the Lisp function `COMPILE` on each translated function. `compile` returns a list of the names of the compiled functions.

`compile (all)` or `compile (functions)` compiles all user-defined functions.

`compile` quotes its arguments; the double-single-quotes operator `''` defeats quotation.

define (*f*(*x_1, ..., x_n*), *expr*) Function

Defines a function named *f* with arguments *x_1*, ..., *x_n* and function body *expr*.

`define` quotes its first argument in most cases, and evaluates its second argument unless explicitly quoted. However, if the first argument is an expression of the form `ev (`*expr*`)`, `funmake (`*expr*`)`, or `arraymake (`*expr*`)`, the first argument is evaluated; this allows for the function name to be computed, as well as the body.

`define` is similar to the function definition operator `:=`, but when `define` appears inside a function, the definition is created using the value of `expr` at execution time rather than at the time of definition of the function which contains it.

All function definitions appear in the same namespace; defining a function `f` within another function `g` does not limit the scope of `f` to `g`.

Examples:

```
(%i1) foo: 2^bar;
                                  bar
(%o1)                            2
(%i2) g(x) := (f_1 (y) :=   foo*x*y,
               f_2 (y) := ''foo*x*y,
       define (f_3 (y),     foo*x*y),
       define (f_4 (y),   ''foo*x*y));
                                                bar
(%o2) g(x) := (f_1(y) := foo x y, f_2(y) := 2    x y,
                                                           bar
               define(f_3(y), foo x y), define(f_4(y), 2    x y))
(%i3) functions;
(%o3)                        [g(x)]
(%i4) g(a);
                                      bar
(%o4)                  f_4(y) := a 2    y
(%i5) functions;
(%o5)        [g(x), f_1(y), f_2(y), f_3(y), f_4(y)]
(%i6) dispfun (f_1, f_2, f_3, f_4);
(%t6)                   f_1(y) := foo x y

                                    bar
(%t7)                  f_2(y) := 2    x y

                                      bar
(%t8)                  f_3(y) := a 2    y

                                      bar
(%t9)                  f_4(y) := a 2    y

(%o9)                            done
```

define_variable (*name*, *default_value*, *mode*) Function

Introduces a global variable into the Maxima environment. `define_variable` is useful in user-written packages, which are often translated or compiled.

`define_variable` carries out the following steps:

1. `mode_declare (`*name*`,` *mode*`)` declares the mode of *name* to the translator. See `mode_declare` for a list of the possible modes.
2. If the variable is unbound, *default_value* is assigned to *name*.
3. `declare (`*name*`, special)` declares it special.
4. Associates *name* with a test function to ensure that *name* is only assigned values of the declared mode.

The `value_check` property can be assigned to any variable which has been defined via `define_variable` with a mode other than `any`. The `value_check` property is a lambda expression or the name of a function of one variable, which is called when an attempt is made to assign a value to the variable. The argument of the `value_check` function is the would-be assigned value.

`define_variable` evaluates `default_value`, and quotes `name` and `mode`. `define_variable` returns the current value of `name`, which is `default_value` if `name` was unbound before, and otherwise it is the previous value of `name`.

Examples:

`foo` is a Boolean variable, with the initial value `true`.

```
(%i1) define_variable (foo, true, boolean);
(%o1)                          true
(%i2) foo;
(%o2)                          true
(%i3) foo: false;
(%o3)                          false
(%i4) foo: %pi;
Error: foo was declared mode boolean, has value: %pi
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i5) foo;
(%o5)                          false
```

`bar` is an integer variable, which must be prime.

```
(%i1) define_variable (bar, 2, integer);
(%o1)                            2
(%i2) qput (bar, prime_test, value_check);
(%o2)                       prime_test
(%i3) prime_test (y) := if not primep(y) then error (y, "is not prime.");
(%o3) prime_test(y) := if not primep(y)

                                   then error(y, "is not prime.")
(%i4) bar: 1439;
(%o4)                          1439
(%i5) bar: 1440;
1440 is not prime.
#0: prime_test(y=1440)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) bar;
(%o6)                          1439
```

`baz_quux` is a variable which cannot be assigned a value. The mode `any_check` is like `any`, but `any_check` enables the `value_check` mechanism, and `any` does not.

```
(%i1) define_variable (baz_quux, 'baz_quux, any_check);
(%o1)                        baz_quux
(%i2) F: lambda ([y], if y # 'baz_quux then error ("Cannot assign to `baz_quux
(%o2) lambda([y], if y # 'baz_quux

                            then error(Cannot assign to `baz_quux'.))
(%i3) qput (baz_quux, ''F, value_check);
(%o3) lambda([y], if y # 'baz_quux

                            then error(Cannot assign to `baz_quux'.))
(%i4) baz_quux: 'baz_quux;
(%o4)                        baz_quux
```

```
(%i5) baz_quux: sqrt(2);
Cannot assign to ‘baz_quux’.
#0: lambda([y],if y # 'baz_quux then error("Cannot assign to ‘baz_quux’."))(y=
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) baz_quux;
(%o6)                          baz_quux
```

dispfun (*f_1, ..., f_n*) Function

dispfun (*all*) Function

Displays the definition of the user-defined functions *f_1, ..., f_n*. Each argument may be the name of a macro (defined with `::=`), an ordinary function (defined with `:=` or `define`), an array function (defined with `:=` or `define`, but enclosing arguments in square brackets `[ ]`), a subscripted function, (defined with `:=` or `define`, but enclosing some arguments in square brackets and others in parentheses `( )`) one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

`dispfun (all)` displays all user-defined functions as given by the `functions`, `arrays`, and `macros` lists, omitting subscripted functions defined with constant subscripts.

`dispfun` creates an intermediate expression label (`%t1`, `%t2`, etc.) for each displayed function, and assigns the function definition to the label. In contrast, `fundef` returns the function definition.

`dispfun` quotes its arguments; the double-single-quote operator `''` defeats quotation. `dispfun` always returns `done`.

Examples:

```
(%i1) m(x, y) ::= x^(-y)$
(%i2) f(x, y) :=  x^(-y)$
(%i3) g[x, y] :=  x^(-y)$
(%i4) h[x](y) :=  x^(-y)$
(%i5) i[8](y) :=  8^(-y)$
(%i6) dispfun (m, f, g, h, h[5], h[10], i[8])$
                                     - y
(%t6)                   m(x, y) ::= x


                                     - y
(%t7)                    f(x, y) := x


                                    - y
(%t8)                     g     := x
                           x, y


                                    - y
(%t9)                     h (y) := x
                           x


                                     1
(%t10)                     h (y) := --
                            5         y
```

```
                                   5

                                    1
(%t11)                 h  (y) := ---
                        10         y
                                   10

                                   - y
(%t12)                 i (y) := 8
                        8
```

functions System variable

Default value: `[]`

`functions` is the list of user-defined Maxima functions in the current session. A user-defined function is a function constructed by `define` or `:=`. A function may be defined at the Maxima prompt or in a Maxima file loaded by `load` or `batch`. Lisp functions, however, are not added to `functions`.

fundef (*f*) Function

Returns the definition of the function *f*.

The argument may be the name of a macro (defined with `::=`), an ordinary function (defined with `:=` or `define`), an array function (defined with `:=` or `define`, but enclosing arguments in square brackets `[ ]`), a subscripted function, (defined with `:=` or `define`, but enclosing some arguments in square brackets and others in parentheses `( )`) one of a family of subscripted functions selected by a particular subscript value, or a subscripted function defined with a constant subscript.

`fundef` quotes its argument; the double-single-quote operator `''` defeats quotation.

`fundef (`*f*`)` returns the definition of *f*. In contrast, `dispfun (`*f*`)` creates an intermediate expression label and assigns the definition to the label.

funmake (*name*, [*arg_1*, ..., *arg_n*]) Function

Returns an expression *name* `(`*arg_1*`, ..., `*arg_n*`)`. The return value is simplified, but not evaluated, so the function is not called.

`funmake` evaluates its arguments.

Examples:

- `funmake` evaluates its arguments, but not the return value.

```
(%i1) det(a,b,c) := b^2 -4*a*c$
(%i2) x: 8$
(%i3) y: 10$
(%i4) z: 12$
(%i5) f: det$
(%i6) funmake (f, [x, y, z]);
(%o6)                      det(8, 10, 12)
(%i7) ''%;
(%o7)                           - 284
```

- Maxima simplifies `funmake`'s return value.

```
(%i1) funmake (sin, [%pi/2]);
(%o1)                              1
```

lambda ([*x_1*, ..., *x_m*], *expr_1*, ..., *expr_n*) Function

Defines and returns a lambda expression (that is, an anonymous function) with arguments *x_1*, ..., *x_m* and return value *expr_n*. A lambda expression can be assigned to a variable and evaluated like an ordinary function. A lambda expression may appear in contexts in which a function evaluation (but not a function name) is expected.

When the function is evaluated, unbound local variables *x_1*, ..., *x_m* are created. `lambda` may appear within `block` or another `lambda`; local variables are established each time another `block` or `lambda` is evaluated. Local variables appear to be global to any enclosed `block` or `lambda`. If a variable is not local, its value is the value most recently assigned in an enclosing `block` or `lambda`, if any, otherwise, it is the value of the variable in the global environment. This policy may coincide with the usual understanding of "dynamic scope".

After local variables are established, *expr_1* through *expr_n* are evaluated in turn. The special variable `%%`, representing the value of the preceding expression, is recognized. `throw` and `catch` may also appear in the list of expressions.

`return` cannot appear in a lambda expression unless enclosed by `block`, in which case `return` defines the return value of the block and not of the lambda expression, unless the block happens to be *expr_n*. Likewise, `go` cannot appear in a lambda expression unless enclosed by `block`.

`lambda` quotes its arguments; the double-single-quote operator `''` defeats quotation.

Examples:

- A lambda expression can be assigned to a variable and evaluated like an ordinary function.

```
(%i1) f: lambda ([x], x^2);
                                      2
(%o1)                    lambda([x], x )
(%i2) f(a);
                                2
(%o2)                          a
```

- A lambda expression may appear in contexts in which a function evaluation is expected.

```
(%i3) lambda ([x], x^2) (a);
                                2
(%o3)                          a
(%i4) apply (lambda ([x], x^2), [a]);
                                2
(%o4)                          a
(%i5) map (lambda ([x], x^2), [a, b, c, d, e]);
                         2   2   2   2   2
(%o5)                  [a , b , c , d , e ]
```

- Argument variables are local variables. Other variables appear to be global variables. Global variables are evaluated at the time the lambda expression is evaluated, unless some special evaluation is forced by some means, such as `''`.

```
(%i6) a: %pi$
(%i7) b: %e$
(%i8) g: lambda ([a], a*b);
(%o8)                   lambda([a], a b)
(%i9) b: %gamma$
(%i10) g(1/2);
                             %gamma
(%o10)                       ------
                               2
(%i11) g2: lambda ([a], a*''b);
(%o11)               lambda([a], a %gamma)
(%i12) b: %e$
(%i13) g2(1/2);
                             %gamma
(%o13)                       ------
                               2
```

- Lambda expressions may be nested. Local variables within the outer lambda expression appear to be global to the inner expression unless masked by local variables of the same names.

```
(%i14) h: lambda ([a, b], h2: lambda ([a], a*b), h2(1/2));
                                                   1
(%o14)   lambda([a, b], h2 : lambda([a], a b), h2(-))
                                                   2
(%i15) h(%pi, %gamma);
                             %gamma
(%o15)                       ------
                               2
```

- Since `lambda` quotes its arguments, lambda expression `i` below does not define a "multiply by `a`" function. Such a function can be defined via `buildq`, as in lambda expression `i2` below.

```
(%i16) i: lambda ([a], lambda ([x], a*x));
(%o16)            lambda([a], lambda([x], a x))
(%i17) i(1/2);
(%o17)                  lambda([x], a x)
(%i18) i2: lambda([a], buildq([a: a], lambda([x], a*x)));
(%o18)   lambda([a], buildq([a : a], lambda([x], a x)))
(%i19) i2(1/2);
                                     x
(%o19)                  lambda([x], -)
                                     2
(%i20) i2(1/2)(%pi);
                               %pi
(%o20)                         ---
                                2
```

local (v_1, ..., v_n) Function

Declares the variables v_1, ..., v_n to be local with respect to all the properties in the statement in which this function is used.

`local` quotes its arguments. `local` returns `done`.

`local` may only be used in `block`, 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 Option variable

Default value: `false`

`macroexpansion` 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 calls `grind` and `display` 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 Option variable

Default value: `true`

When `mode_checkp` is `true`, `mode_declare` checks the modes of bound variables.

mode_check_errorp Option variable

Default value: `false`

When `mode_check_errorp` is `true`, `mode_declare` calls error.

mode_check_warnp Option variable

Default value: `true`

When `mode_check_warnp` is `true`, mode errors are described.

mode_declare (*y_1, mode_1, ..., y_n, mode_n*) Function

`mode_declare` is used to declare the modes of variables and functions for subsequent translation or compilation of functions. `mode_declare` is typically placed at the beginning of a function definition, at the beginning of a Maxima script, or executed at the interactive prompt.

The arguments of `mode_declare` are pairs consisting of a variable and a mode which is one of `boolean`, `fixnum`, `number`, `rational`, or `float`. Each variable may also be a list of variables all of which are declared to have the same mode.

If a variable 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.

Numeric code using arrays might 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.

One may declare the mode of the result of a function by using `function (f_1, f_2, ...)` as an argument; here `f_1`, `f_2`, ... are the names of functions. For example the expression,

```
mode_declare ([function (f_1, f_2, ...)], fixnum)
```

declares that the values returned by `f_1`, `f_2`, ... are single-word integers.

`modedeclare` is a synonym for `mode_declare`.

mode_identity (*arg_1*, *arg_2*) 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., one of `float`, `fixnum`, `number`, `list`, or `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 Maxima 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);` yields 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.

transcompile Option variable

Default value: `true`

When `transcompile` is `true`, `translate` and `translate_file` generate declarations to make the translated code more suitable for compilation.

`compfile` sets `transcompile: true` for the duration.

translate (*f_1, ..., f_n*) Function
translate (*functions*) Function
translate (*all*) Function

Translates the user-defined functions *f_1*, ..., *f_n* from the Maxima language into Lisp and evaluates the Lisp translations. Typically the translated functions run faster than the originals.

`translate (all)` or `translate (functions)` translates all user-defined functions.

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 (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, mode_1, v_2, mode_2, ...), ...)
```

where the *x_1*, *x_2*, ... are the parameters to the function and the *v_1*, *v_2*, ... 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.

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`, 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 incompatabilities may exist between the Lisp and Maxima versions. Principally, the `rat` function with more than one argument and the `ratvars` function should not be used if any variables are `mode_declare`'d canonical rational expressions (CRE). Also the `prederror: false` setting will not translate.

`savedef` - if `true` will cause the Maxima version of a user function to remain when the function is `translate`'d. This permits the definition to be displayed by `dispfun` and allows the function to be edited.

`transrun` - if `false` will cause the interpreted version of all functions to be run (provided they are still around) rather than the translated version.

The result returned by `translate` is a list of the names of the functions translated.

translate_file (*maxima_filename*) Function
translate_file (*maxima_filename, lisp_filename*) Function

Translates a file of Maxima code into a file of Lisp code. `translate_file` returns a list of three filenames: the name of the Maxima file, the name of the Lisp file, and the name of file containing additional information about the translation. `translate_file` evaluates its arguments.

`translate_file ("foo.mac"); load("foo.LISP")` is the same as `batch ("foo.mac")` except for certain restrictions, the use of `''` and `%`, for example.

`translate_file (`*maxima_filename*`)` translates a Maxima file *maxima_filename* into a similarly-named Lisp file. For example, `foo.mac` is translated into `foo.LISP`. The

Maxima filename may include a directory name or names, in which case the Lisp output file is written to the same directory from which the Maxima input comes.

`translate_file (`*maxima_filename*`,` *lisp_filename*`)` translates a Maxima file *maxima_filename* into a Lisp file *lisp_filename*. `translate_file` ignores the filename extension, if any, of `lisp_filename`; the filename extension of the Lisp output file is always `LISP`. The Lisp filename may include a directory name or names, in which case the Lisp output file is written to the specified directory.

`translate_file` also writes a file of translator warning messages of various degrees of severity. The filename extension of this file is `UNLISP`. This file may contain valuable information, though possibly obscure, for tracking down bugs in translated code. The `UNLISP` file is always written to the same directory from which the Maxima input comes.

`translate_file` emits Lisp code which causes some declarations and definitions to take effect as soon as the Lisp code is compiled. See `compile_file` for more on this topic.

See also `tr_array_as_ref`, `tr_bound_function_applyp`, `tr_exponent`, `tr_file_tty_messagesp`, `tr_float_can_branch_complex`, `tr_function_call_default`, `tr_numer`, `tr_optimize_max_loop`, `tr_semicompile`, `tr_state_vars`, `tr_warnings_get`, `tr_warn_bad_function_calls`, `tr_warn_fexpr`, `tr_warn_meval`, `tr_warn_mode`, `tr_warn_undeclared`, `tr_warn_undefined_variable`, and `tr_windy`.

transrun Option variable

Default value: `true`

When `transrun` is `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 Option variable

Default value: `true`

If `translate_fast_arrays` is false, array references in Lisp code emitted by `translate_file` are affected by `tr_array_as_ref`. When `tr_array_as_ref` is `true`, array names are evaluated, otherwise array names appear as literal symbols in translated code.

`tr_array_as_ref` has no effect if `translate_fast_arrays` is `true`.

tr_bound_function_applyp Option variable

Default value: `true`

When `tr_bound_function_applyp` is `true`, Maxima gives a warning if a bound variable (such as a function argument) is found being used as a function. `tr_bound_function_applyp` does not affect the code generated in such cases.

For example, an expression such as `g (f, x) := f (x+1)` will trigger the warning message.

tr_file_tty_messagesp Option variable

Default value: `false`

When `tr_file_tty_messagesp` is `true`, messages generated by `translate_file` during translation of a file are displayed on the console and inserted into the UNLISP file. When `false`, messages about translation of the file are only inserted into the UNLISP file.

tr_float_can_branch_complex Option variable

Default value: `true`

Tells the Maxima-to-Lisp translator to assume that the functions `acos`, `asin`, `asec`, and `acsc` can return complex results.

The ostensible effect of `tr_float_can_branch_complex` is the following. However, it appears that this flag has no effect on the translator output.

When it is `true` then `acos(x)` is of mode `any` even if `x` is of mode `float` (as set by `mode_declare`). When `false` then `acos(x)` is of mode `float` if and only if `x` is of mode `float`.

tr_function_call_default Option variable

Default value: `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 apropriate 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 Maxima interpreter.

tr_numer Option variable

Default value: `false`

When `tr_numer` is `true` numer properties are used for atoms which have them, e.g. `%pi`.

tr_optimize_max_loop Option variable

Default value: 100

`tr_optimize_max_loop` is 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_semicompile Option variable

Default value: `false`

When `tr_semicompile` is `true`, `translate_file` and `compfile` output forms which will be macroexpanded but not compiled into machine code by the Lisp compiler.

tr_state_vars System variable

Default value:

```
[transcompile, tr_semicompile, 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_warnings_get () Function

Prints a list of warnings which have been given by the translator during the current translation.

tr_warn_bad_function_calls Option variable

Default value: `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 Option variable

Default value: `compfile`

- 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 Option variable

Default value: `compfile`

- Gives a warning if the function `meval` gets called. If `meval` is called that indicates problems in the translation.

tr_warn_mode Option variable

Default value: `all`

- Gives a warning when variables are assigned values inappropriate for their mode.

tr_warn_undeclared Option variable

Default value: `compile`

- Determines when to send warnings about undeclared variables to the TTY.

tr_warn_undefined_variable Option variable

Default value: `all`

- Gives a warning when undefined global variables are seen.

tr_windy Option variable

Default value: `true`

- Generate `"helpfull"` comments and programming hints.

compile_file (*filename*) Function
compile_file (*filename, compiled_filename*) Function
compile_file (*filename, compiled_filename, lisp_filename*) Function

Translates the Maxima file *filename* into Lisp, executes the Lisp compiler, and, if the translation and compilation succeed, loads the compiled code into Maxima.

`compile_file` returns a list of the names of four files: the original Maxima file, the Lisp translation, notes on translation, and the compiled code. If the compilation fails, the fourth item is `false`.

Some declarations and definitions take effect as soon as the Lisp code is compiled (without loading the compiled code). These include functions defined with the `:=` operator, macros define with the `::=` operator, `alias`, `declare`, `define_variable`, `mode_declare`, and `infix`, `matchfix`, `nofix`, `postfix`, `prefix`, and `compfile`.

Assignments and function calls are not evaluated until the compiled code is loaded. In particular, within the Maxima file, assignments to the translation flags (`tr_numer`, etc.) have no effect on the translation.

filename may not contain `:lisp` statements.

`compile_file` evaluates its arguments.

declare_translated (*f_1, f_2, ...*) Function

When translating a file of Maxima 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 Maxima 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.

42 Program Flow

42.1 Introduction to Program Flow

Maxima provides a `do` loop for iteration, as well as more primitive constructs such as `go`.

42.2 Definitions for Program Flow

backtrace () Function
backtrace (*n*) Function

Prints the call stack, that is, the list of functions which called the currently active function.

`backtrace()` prints the entire call stack.

`backtrace (n)` prints the *n* most recent functions, including the currently active function.

`backtrace` can be called from a script, a function, or the interactive prompt (not only in a debugging context).

Examples:

- `backtrace()` prints the entire call stack.

```
(%i1) h(x) := g(x/7)$
(%i2) g(x) := f(x-11)$
(%i3) f(x) := e(x^2)$
(%i4) e(x) := (backtrace(), 2*x + 13)$
(%i5) h(10);
#0: e(x=4489/49)
#1: f(x=-67/7)
#2: g(x=10/7)
#3: h(x=10)
                              9615
(%o5)                         ----
                               49
```

- `backtrace (n)` prints the *n* most recent functions, including the currently active function.

```
(%i1) h(x) := (backtrace(1), g(x/7))$
(%i2) g(x) := (backtrace(1), f(x-11))$
(%i3) f(x) := (backtrace(1), e(x^2))$
(%i4) e(x) := (backtrace(1), 2*x + 13)$
(%i5) h(10);
#0: h(x=10)
#0: g(x=10/7)
#0: f(x=-67/7)
#0: e(x=4489/49)
                              9615
(%o5)                         ----
                               49
```

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.

There are three variants of this form that differ only in their terminating conditions. They are:

- `for` *variable*`:` *initial_value* `step` *increment* `thru` *limit* `do` *body*
- `for` *variable*`:` *initial_value* `step` *increment* `while` *condition* `do` *body*
- `for` *variable*`:` *initial_value* `step` *increment* `unless` *condition* `do` *body*

(Alternatively, the `step` may be given after the termination condition or limit.)

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, Maxima 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`. 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.

```
(%i1) for a:-3 thru 26 step 7 do display(a)$
                             a = - 3

                              a = 4
```

```
                                  a = 11

                                  a = 18

                                  a = 25
(%i1) s: 0$
(%i2) for i: 1 while i <= 10 do s: s+i;
(%o2)                              done
(%i3) s;
(%o3)                               55
```

Note that the condition `while i <= 10` is equivalent to `unless i > 10` and also `thru 10`.

```
(%i1) series: 1$
(%i2) term: exp (sin (x))$
(%i3) for p: 1 unless p > 7 do
          (term: diff (term, x)/p,
           series: series + subst (x=0, term)*x^p)$
(%i4) series;
                  7    6     5    4    2
                 x    x     x    x    x
(%o4)            -- - --- - -- - -- + -- + x + 1
                 90   240   15   8    2
```

which gives 8 terms of the Taylor series for `e^sin(x)`.

```
(%i1) poly: 0$
(%i2) for i: 1 thru 5 do
          for j: i step -1 thru 1 do
              poly: poly + i*x^j$
(%i3) poly;
                  5      4       3       2
(%o3)          5 x  + 9 x  + 12 x  + 14 x  + 15 x
(%i4) guess: -3.0$
(%i5) for i: 1 thru 10 do
          (guess: subst (guess, x, 0.5*(x + 10/x)),
           if abs (guess^2 - 10) < 0.00005 then return (guess));
(%o5)                  - 3.162280701754386
```

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`.

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.

```
(%i6) 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.

```
(%i1) x: 1000$
(%i2) thru 20 do x: 0.5*(x + 5.0/x)$
(%i3) x;
(%o3)                   2.23606797749979
(%i4) sqrt(5), numer;
(%o4)                   2.23606797749979
```

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`.

```
(%i1) newton (f, x):= ([y, df, dfx], df: diff (f ('x), 'x),
          do (y: ev(df), x: x - f(x)/y,
              if abs (f (x)) < 5e-6 then return (x)))$
(%i2) sqr (x) := x^2 - 5.0$
(%i3) newton (sqr, 1000);
(%o3)                   2.236068027062195
```

(Note that `return`, when executed, causes the current value of `x` 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 Maxima. The syntax is:

`for` *variable* `in` *list* *end_tests* `do` *body*

The elements of *list* are any expressions which will successively be assigned to the variable on each iteration of the body. The optional termination tests *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.)

```
(%i1)  for f in [log, rho, atan] do ldisp(f(1))$
(%t1)                               0
(%t2)                             rho(1)
                                  %pi
(%t3)                              ---
                                   4
(%i4) ev(%t3,numer);
(%o4)                         0.78539816
```

errcatch (*expr_1, ..., expr_n*) Function

Evaluates *expr_1*, ..., *expr_n* one by one and returns [*expr_n*] (a list) if no error occurs. If an error occurs in the evaluation of any argument, `errcatch` prevents the error from propagating and returns the empty list `[]` without evaluating any more arguments.

`errcatch` is useful in `batch` files where one suspects an error might occur which would terminate the `batch` if the error weren't caught.

error (*expr_1, ..., expr_n*) Function
error System variable

Evaluates and prints *expr_1*, ..., *expr_n*, and then causes an error return to top level Maxima or to the nearest enclosing `errcatch`.

The variable `error` is set to a list describing the error. The first element of `error` is a format string, which merges all the strings among the arguments *expr_1*, ..., *expr_n*, and the remaining elements are the values of any non-string arguments.

`errormsg()` formats and prints `error`. This is effectively reprinting the most recent error message.

errormsg () Function

Reprints the most recent error message. The variable `error` holds the message, and `errormsg` formats and prints it.

for Special operator

Used in iterations. See `do` for a description of Maxima'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 <expr_1> else <expr_2>
```

The result of an `if` statement is *expr_1* if condition is `true` and *expr_2* otherwise. *expr_1* and *expr_2* are any Maxima 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:

```
Operation            Symbol      Type

less than            <           relational infix
less than            <=
  or equal to                    relational infix
```

```
equality (syntactic) =          relational infix
negation of =        #          relational infix
equality (value)     equal      relational function
negation of equal    notequal   relational function
greater than         >=
  or equal to                   relational infix
greater than         >          relational infix
and                  and        logical infix
or                   or         logical infix
not                  not        logical prefix
```

map (*f, expr_1, ..., expr_n*) Function

Returns an expression whose leading operator is the same as that of the expressions *expr_1, ..., expr_n* but whose subparts are the results of applying f to the corresponding subparts of the expressions. f is either the name of a function of n arguments or is a `lambda` form of n arguments.

`maperror` - 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.

```
(%i1) map(f,x+a*y+b*z);
(%o1)                        f(b z) + f(a y) + f(x)
(%i2) map(lambda([u],partfrac(u,x)),x+1/(x^3+4*x^2+5*x+2));
                           1       1        1
(%o2)                    ----- - ----- + -------- + x
                         x + 2   x + 1          2
                                         (x + 1)
(%i3) map(ratsimp, x/(x^2+x)+(y^2+y)/y);
                                      1
(%o3)                           y + ----- + 1
                                    x + 1
(%i4) map("=",[a,b],[-0.5,3]);
(%o4)                          [a = - 0.5, b = 3]
```

mapatom (*expr*) Function

Returns `true` if and only if *expr* is treated by the mapping routines as an atom. "Mapatoms" are atoms, numbers (including rational numbers), and subscripted variables.

maperror Option variable

Default value: `true`

When `maperror` is `false`, causes all of the mapping functions, for example

```
map (f, expr_1, expr_2, ...))
```

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 `f` to `[expr_1, expr_2, ...]` if the `expr_i` are not all the same type of object.

If `maperror` is `true` then an error message is displayed in the above two instances.

maplist (*f, expr_1, ..., expr_n*) Function

Returns a list of the applications of *f* to the parts of the expressions *expr_1*, ..., *expr_n*. *f* is the name of a function, or a lambda expression.

`maplist` differs from `map (`*f*`, `*expr_1*`, ..., `*expr_n*`)` which returns an expression with the same main operator as *expr_i* has (except for simplifications and the case where `map` does an `apply`).

prederror Option variable

Default value: `true`

When `prederror` is `true`, an error message is displayed 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; however, `maybe` is supported in translated code.

See also `is` and `maybe`.

return (*value*) Function

May be used to exit explicitly from a block, bringing its argument. See `block` for more information.

scanmap (*f, expr*) Function
scanmap (*f, expr, bottomup*) Function

Recursively applies *f* to *expr*, in a top down manner. This is most useful when complete factorization is desired, for example:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
                                    2      2
(%o2)                        (a + 1)  y + x
```

Note the way in which `scanmap` applies the given function `factor` to the constituent subexpressions of *expr*; if another form of *expr* is presented to `scanmap` then the result may be different. Thus, `%o2` is not recovered when `scanmap` is applied to the expanded form of exp:

```
(%i3) scanmap(factor,expand(exp));
                           2                  2
(%o3)                      a  y + 2 a y + y + x
```

Here is another example of the way in which `scanmap` recursively applies a given function to all subexpressions, including exponents:

```
(%i4) expr : u*v^(a*x+b) + c$
(%i5) scanmap('f, expr);
                    f(f(f(a) f(x)) + f(b))
(%o5) f(f(f(u) f(f(v)                        )) + f(c))
```

`scanmap` (*f*, *expr*, `bottomup`) applies *f* to *expr* 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 (*expr*) Function

Evaluates *expr* and throws the value back to the most recent `catch`. `throw` is used with `catch` as a nonlocal return mechanism.

outermap (*f, a_1, ..., a_n*) Function

Applies the function *f* to each one of the elements of the outer product *a_1* cross *a_2* ... cross *a_n*.

f is be the name of a function of n arguments or a lambda expression of n arguments. The arguments *a_1*, ..., *a_n* may be lists or nonlists. List arguments may have different lengths. Arguments other than lists are treated as lists of length 1 for the purpose of constructing the outer product.

The result of applying *f* to the outer product is organized as a nested list. The depth of nesting is equal to the number of list arguments (arguments other than lists do not contribute a nesting level). A list at nesting depth k has the same length as the k'th list argument.

`outermap` evaluates its arguments.

See also `map`, `maplist`, and `apply`.

Examples:

```
(%i1) f (x, y) := x - y$
(%i2) outermap (f, [2, 3, 5], [a, b, c, d]);
(%o2) [[2 - a, 2 - b, 2 - c, 2 - d],
      [3 - a, 3 - b, 3 - c, 3 - d], [5 - a, 5 - b, 5 - c, 5 - d]]
(%i3) outermap (lambda ([x, y], y/x), [55, 99], [Z, W]);
                         Z   W     Z   W
(%o3)                  [[--, --], [--, --]]
                         55  55    99  99
(%i4) g: lambda ([x, y, z], x + y*z)$
(%i5) outermap (g, [a, b, c], %pi, [11, 17]);
(%o5) [[a + 11 %pi, a + 17 %pi], [b + 11 %pi, b + 17 %pi],
                                          [c + 11 %pi, c + 17 %pi]]
(%i6) flatten (%);
(%o6) [a + 11 %pi, a + 17 %pi, b + 11 %pi, b + 17 %pi,
                                            c + 11 %pi, c + 17 %pi]
```

43 Debugging

43.1 Source Level Debugging

Maxima has a built-in source level debugger. The user can set a breakpoint at a function, and then step line by line from there. The call stack may be examined, together with the variables bound at that level.

The command `:help` or `:h` shows the list of debugger commands. (In general, commands may be abbreviated if the abbreviation is unique. If not unique, the alternatives will be listed.) Within the debugger, the user can also use any ordinary Maxima functions to examine, define, and manipulate variables and expressions.

A breakpoint is set by the `:br` command at the Maxima prompt. Within the debugger, the user can advance one line at a time using the `:n` ("next") command. The `:bt` ("backtrace") command shows a list of stack frames. The `:r` ("resume") command exits the debugger and continues with execution. These commands are demonstrated in the example below.

```
(%i1) load ("/tmp/foobar.mac");

(%o1)                           /tmp/foobar.mac

(%i2) :br foo
Turning on debugging debugmode(true)
Bkpt 0 for foo (in /tmp/foobar.mac line 1)

(%i2) bar (2,3);
Bkpt 0:(foobar.mac 1)
/tmp/foobar.mac:1::

(dbm:1) :bt                        <-- :bt typed here gives a backtrace
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n                         <-- Here type :n to advance line
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n                         <-- Here type :n to advance line
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u;                         <-- Investigate value of u
28

(dbm:1) u: 33;                     <-- Change u to be 33
33

(dbm:1) :r                         <-- Type :r to resume the computation
```

```
(%o2)                              1094
```

The file /tmp/foobar.mac is the following:

```
foo(y) := block ([u:y^2],
  u: u+3,
  u: u^2,
  u);

bar(x,y) := (
  x: x+2,
  y: y+2,
  x: foo(y),
  x+y);
```

USE OF THE DEBUGGER THROUGH EMACS

If the user is running the code under GNU emacs in a shell window (dbl shell), or is running the graphical interface version, xmaxima, then if he stops at a break point, he will see his current position in the source file which will be displayed in the other half of the window, either highlighted in red, or with a little arrow pointing at the right line. He can advance single lines at a time by typing M-n (Alt-n).

Under Emacs you should run in a dbl shell, which requires the dbl.el file in the elisp directory. Make sure you install the elisp files or add the Maxima elisp directory to your path: e.g., add the following to your '.emacs' file or the site-init.el

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

then in emacs

```
M-x dbl
```

should start a shell window in which you can run programs, for example Maxima, gcl, gdb etc. This shell window also knows about source level debugging, and display of source code in the other window.

The user may set a break point at a certain line of the file by typing C-x space. This figures out which function the cursor is in, and then it sees which line of that function the cursor is on. If the cursor is on, say, line 2 of foo, then it will insert in the other window the command, ":br foo 2", to break foo at its second line. To have this enabled, the user must have maxima-mode.el turned on in the window in which the file foobar.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.

43.2 Keyword Commands

Keyword commands are special keywords which are not interpreted as Maxima expressions. A keyword command can be entered at the Maxima prompt or the debugger prompt, although not at the break prompt. Keyword commands start with a colon, ':'. For example, to evaluate a Lisp form you may type :lisp followed by the form to be evaluated.

```
(%i1) :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 break keywords. Thus `:br` would suffice for `:break`.

The keyword commands are listed below.

`:break F n`
: Set a breakpoint in function `F` at line offset `n` from the beginning of the function. If `F` is given as a string, then it is assumed to be a file, and `n` is the offset from the beginning of the file. The offset is optional. If not given, it is assumed to be zero (first line of the function or file).

`:bt` Print a backtrace of the stack frames

`:continue`
: Continue the computation

`:delete` Delete the specified breakpoints, or all if none are specified

`:disable` Disable the specified breakpoints, or all if none are specified

`:enable` Enable the specified breakpoints, or all if none are specified

`:frame n` Print stack frame `n`, or the current frame if none is specified

`:help` Print help on a debugger command, or all commands if none is specified

`:info` Print information about item

`:lisp some-form`
: Evaluate `some-form` as a Lisp form

`:lisp-quiet some-form`
: Evaluate Lisp form `some-form` without any output

`:next` Like `:step`, except `:next` steps over function calls

`:quit` Quit the current debugger level without completing the computation

`:resume` Continue the computation

`:step` Continue the computation until it reaches a new source line

`:top` Return to the Maxima prompt (from any debugger level) without completing the computation

43.3 Definitions for Debugging

refcheck Option variable

Default value: `false`

When `refcheck` is `true`, Maxima prints a message each time a bound variable is used for the first time in a computation.

setcheck Option variable

Default value: `false`

If `setcheck` is set to a list of variables (which can be subscripted), Maxima prints a message whenever the variables, or subscripted occurrences of them, are bound with the ordinary assignment operator `:`, the `::` assignment operator, or function argument binding, but not the function assignment `:=` nor the macro assignment `::=` operators. The message comprises the name of the variable and the value it is bound to.

`setcheck` may be set to `all` or `true` thereby including all variables.

Each new assignment of `setcheck` establishes a new list of variables to check, and any variables previously assigned to `setcheck` are forgotten.

The names assigned to `setcheck` must be quoted if they would otherwise evaluate to something other than themselves. For example, if `x`, `y`, and `z` are already bound, then enter

```
setcheck: ['x, 'y, 'z]$
```

to put them on the list of variables to check.

No printout is generated when a variable on the `setcheck` list is assigned to itself, e.g., `X: 'X`.

setcheckbreak Option variable

Default value: `false`

When `setcheckbreak` is `true`, Maxima will present a break prompt whenever a variable on the `setcheck` list is assigned a new value. The break occurs before the assignment is carried out. At this point, `setval` holds the value to which the variable is about to be assigned. Hence, one may assign a different value by assigning to `setval`.

See also `setcheck` and `setval`.

setval System variable

Holds the value to which a variable is about to be set when a `setcheckbreak` occurs. Hence, one may assign a different value by assigning to `setval`.

See also `setcheck` and `setcheckbreak`.

timer (*f_1, ..., f_n*) Function

timer () Function

Given functions *f_1, ..., f_n*, `timer` puts each one on the list of functions for which timing statistics are collected. `timer(f)$ timer(g)$` puts `f` and then `g` onto the list; the list accumulates from one call to the next.

With no arguments, `timer` returns the list of timed functions.

Maxima records how much time is spent executing each function on the list of timed functions. `timer_info` returns the timing statistics, including the average time elapsed per function call, the number of calls, and the total time elapsed. `untimer` removes functions from the list of timed functions.

`timer` quotes its arguments. `f(x) := x^2$ g:f$ timer(g)$` does not put `f` on the timer list.

If `trace(f)` is in effect, then `timer(f)` has no effect; `trace` and `timer` cannot both be in effect at the same time.

See also `timer_devalue`.

untimer (*f_1, ..., f_n*) Function
untimer () Function

Given functions *f_1*, ..., *f_n*, `untimer` removes each function from the timer list.

With no arguments, `untimer` removes all functions currently on the timer list.

After `untimer (f)` is executed, `timer_info (f)` still returns previously collected timing statistics, although `timer_info()` (with no arguments) does not return information about any function not currently on the timer list. `timer (f)` resets all timing statistics to zero and puts `f` on the timer list again.

timer_devalue Option variable

Default value: `false`

When `timer_devalue` is `true`, Maxima subtracts from each timed function the time spent in other timed functions. Otherwise, the time reported for each function includes the time spent in other functions. Note that time spent in untimed functions is not subtracted from the total time.

See also `timer` and `timer_info`.

timer_info (*f_1, ..., f_n*) Function
timer_info () Function

Given functions *f_1*, ..., *f_n*, `timer_info` returns a matrix containing timing information for each function. With no arguments, `timer_info` returns timing information for all functions currently on the timer list.

The matrix returned by `timer_info` contains the function name, time per function call, number of function calls, total time, and `gctime`, which meant "garbage collection time" in the original Macsyma but is now always zero.

The data from which `timer_info` constructs its return value can also be obtained by the `get` function:

```
get(f, 'calls);  get(f, 'runtime);  get(f, 'gctime);
```

See also `timer`.

trace (*f_1, ..., f_n*) Function
trace () Function

Given functions *f_1*, ..., *f_n*, `trace` instructs Maxima to print out debugging information whenever those functions are called. `trace(f)$ trace(g)$` puts `f` and then `g` onto the list of functions to be traced; the list accumulates from one call to the next.

With no arguments, `trace` returns a list of all the functions currently being traced.

The `untrace` function disables tracing. See also `trace_options`.

`trace` quotes its arguments. Thus, `f(x) := x^2$ g:f$ trace(g)$` does not put `f` on the trace list.

When a function is redefined, it is removed from the timer list. Thus after `timer(f)$ f(x) := x^2$`, function `f` is no longer on the timer list.

If `timer (f)` is in effect, then `trace (f)` has no effect; `trace` and `timer` can't both be in effect for the same function.

trace_options (*f, option_1, ..., option_n*) Function
trace_options (*f*) Function

Sets the trace options for function *f*. Any previous options are superseded. `trace_options (`*f*`, ...)` has no effect unless `trace (`*f*`)` is also called (either before or after `trace_options`).

`trace_options (`*f*`)` resets all options to their default values.

The option keywords are:

- `noprint` Do not print a message at function entry and exit.
- `break` Put a breakpoint before the function is entered, and after the function is exited. See `break`.
- `lisp_print` Display arguments and return values as Lisp objects.
- `info` Print `-> true` at function entry and exit.
- `errorcatch` Catch errors, giving the option to signal an error, retry the function call, or specify a return value.

Trace options are specified in two forms. The presence of the option keyword alone puts the option into effect unconditionally. (Note that option *foo* is not put into effect by specifying *foo*`: true` or a similar form; note also that keywords need not be quoted.) Specifying the option keyword with a predicate function makes the option conditional on the predicate.

The argument list to the 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, and `item` is the argument list (on entering) or the return value (on exiting).

Here is an example of unconditional trace options:

```
(%i1) ff(n) := if equal(n, 0) then 1 else n * ff(n - 1)$

(%i2) trace (ff)$

(%i3) trace_options (ff, lisp_print, break)$

(%i4) ff(3);
```

Here is the same function, with the `break` option conditional on a predicate:

```
(%i5) trace_options (ff, break(pp))$

(%i6) pp (level, direction, function, item) := block (print (item),
    return (function = 'ff and level = 3 and direction = exit))$

(%i7) ff(6);
```

untrace (*f_1*, ..., *f_n*) Function
untrace () Function

Given functions *f_1*, ..., *f_n*, `untrace` disables tracing enabled by the `trace` function. With no arguments, `untrace` disables tracing for all functions.

`untrace` returns a list of the functions for which it disabled tracing.

44 Indices

Appendix A Function and Variable Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Short Contents

Table of Contents