Manual do Maxima

Manual do Maxima

Maxima é um sistema de álgebra computacional, implementado em Lisp.

Maxima é derivado do sistema Macsyma, desenvolvido no MIT nos anos de 1968 a 1982 como parte do Projeto MAC. MIT remanejou uma cópia do código fonte do Macsyma para o Departamento de Energia em 1982; aquela versão é agora conhecida como Macsyma DOE. Uma cópia do Macsyma DOE foi mantida pelo Professor William F. Schelter da Universidade do Texas de 1982 até sua morte em 2001. Em 1998, Schelter obteve permissão do Departamento de Energia para liberar o código fonte do Macsyma DOE sob a Licença Pública GNU, e em 2000 ele iniciou o projeto Maxima no SourceForge para manter e desenvolver o Macsyma DOE, agora chamado Maxima.

O código fonte deste documento encontra-se no formato texinfo. Para contribuir com a equipe do Maxima na tarefa de manter a tradução para o português sempre atualizada envie um e-mail para <maxima at math dot utexas dot edu>.

1 Introdução ao Maxima

Inicie o Maxima com o comando "maxima". Maxima mostrará a informação de versão e uma linha de comando. Termine cada comando Maxima com um ponto e vírgula. Termine uma sessão com o comando "quit();". Aqui está um exemplo de sessão:

```
[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 pode procurar as páginas info. Use o comando *describe* para mostrar todos os comandos e variáveis contendo uma dada seqüência de caracteres, e opcionalmente sua documentação. O ponto de interrogação ? é uma abreviação para 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
```

Para usar um resultado em cálculos posteriores, você pode atribuir isso a uma variável ou referir-se a isso através de seu rótulo gerado automaticamente. Adicionalmente, `%` refere-se ao mais recente resultado calculado:

```
(%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 tem conhecimento sobre números complexos e constantes numéricas:

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

Maxima pode fazer cálculos diferenciais e integrais:

```
(%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 pode resolver sistemas lineares e equações cúbicas:

```
(%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 pode resolver sistemas de equações não lineares. Note que se você não quer um resultado impresso, você pode encerrar seu comando com *$* em lugar de de *;*.

```
(%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 pode gerar gráficos de uma ou mais funções:

```
(%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 Detecção e Relato de Erros

2.1 Introdução à Detecção e Relato de Erros

Como todos os grandes programas, Maxima contém erros conhecidos e erros desconhecidos. Esse capítulo descreve as facilidades internas para executar o conjunto de testes do Maxima bem como informar novos erros.

2.2 Definições para Detecção e Relato de Erros

run_testsuite () Função
run_testsuite (*boolean*) Função
run_testsuite (*boolean, boolean*) Função
run_testsuite (*boolean, boolean, list*) Função

Executa o conjunto de testes do Maxima. Testes que produzem a resposta desejada são considerados "passes," e testes que não produzem a resposta desejada, são marcados como erros conhecidos.

`run_testsuite ()` mostra somente testes que não são aprovados.

`run_testsuite (true)` mostra somente testes que são marcados como bugs conhecidos, bem como as falhas.

`run_testsuite (true, true)` mostra todos os testes.

Se o terceiro argumento opcional for dado, um subconjunto de testes é executado. O subconjunto de testes para executar é dado como uma lista de nomes dos testes. O conjunto completo de testes é especificado por `testsuite_files`.

`run_testsuite` altera a variável de ambiente Maxima. Tipicamente um script de teste executa `kill` para estabelecer uma variável de ambiente (uma a saber sem funções definidas pelo usuário e variáveis) e então define funções e variáveis apropriadamente para o teste.

`run_testsuite` retorna `done`.

testsuite_files Variável de opção

`testsuite_files` é o conjunto de testes a ser executado por `run_testsuite`. Isso é uma lista de nomes de arquivos contendo os testes a executar. Se alguns dos testes em um arquivo é sabido falhar, então em lugar de listar o nome do arquivo, uma lista contendo o nome do arquivo e o número dos testes que falharam é usada.

por exemplo, isso é uma parte do conjunto de testes padrão:

```
["rtest13s", ["rtest14", 57, 63]]
```

Isso especifica a suite de testes que consiste dos arquivos "rtest13s" e "rtest14", mas "rtest14" contém dois testes que são sabidos falhar: 57 e 63.

bug_report () Função

Imprime os números de versão do Maxima e do Lisp, e chama o link para a página web de informação de erros do projeto Maxima. A informação da versão é a mesma reportada por `build_info`.

Quando um erro é informado, é muito útil copiar a versão do Maxima e do Lisp dentro da informação do erro.

`bug_report` retorna uma seqüência de caracteres vazia `""`.

build_info () Função

Imprime um sumário de parâmetros da compilação do Maxima.

`build_info` retorna uma seqüência de caracteres vazia `""`.

3 Ajuda

3.1 Introdução a Ajuda

A função primária de ajuda on-line é `describe`, que é tipicamente invocada através do ponto de interrogação `?` na linha de comando interativa. `? foo` (com um espaço entre `?` e `foo`) é equivalente a `describe ("foo")`, onde `foo` é o nome ou parte do nome de uma função ou tópico; `describe` então acha todos os ítens documentados que possuem a seqüência de caracteres `foo` em seus títulos. Se existe mais que um tal ítem, Maxima solicita ao usuário selecionar um ítem ou ítens para mostrar.

```
(%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, [...]
```

Nesse exemplo, ítens 7 e 8 foram selecionados. Todos ou nenhum dos ítens poderia ter sido selecionados através da inserção de `all` ou `none`, que podem ser abreviados para `a` ou `n`, respectivamente.

3.2 Lisp e Maxima

Maxima é escrito na liguagem de programação Lisp, e é fácil acessar funções Lisp e variáveis a partir do Maxima e vice-versa. Símbolos Lisp e Maxima são distingüidos através de uma convenção de nome. Um símbolo Lisp que começa com um sinal de dólar `$` corresponde a um símbolo Maxima sem o sinal de dólar. Um símbolo Maxima que começa com um ponto de interrogação `?` corresponde a um símbolo Lisp sem o ponto de interrogação. Por exemplo, o símbolo Maxima `foo` corresponde ao símbolo Lisp `$foo`, enquanto o símbolo Maxima `?foo` corresponde ao símbolo Lisp `foo`, Note que `?foo` é escrito sem um espaço entre `?` e `foo`; de outra forma pode ser uma chamada errônea para `describe ("foo")`.

Hífem `-`, asterisco `*`, ou outro caractere especial em símbolos Lisp deve ser precedido por uma barra invertida `\` onde ele aparecer no código Maxima. Por exemplo, o identificador Lisp `*foo-bar*` é escrito `?\*foo\-bar\*` no Maxima.

Código Lisp pode ser executado dentro de uma sessão Maxima. Uma linha simples de Lisp (contendo uma ou mais formas) pode ser executada através do comando especial `:lisp`. Por exemplo,

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

chama a função Lisp `foo` com variáveis Maxima `x` e `y` como argumentos. A constução `:lisp` pode aparecer na linha de comando interativa ou em um arquivo processado por `batch` ou `demo`, mas não em um arquivo processado por `load`, `batchload`, `translate_file`, ou `compile_file`.

A função `to_lisp()` abre uma sessão interativa Lisp. Digitando `(to-maxima)` fecha a sessão Lisp e retorna para o Maxima.

Funções Lisp e variáveis que são para serem visíveis no Maxima como funções e variáveis com nomes comuns (sem pontuação especial) devem ter nomes Lisp começando como sinal de dólar `$`.

Maxima é sensível à caixa, distingue entre letras em caixa alta (maiúsculas) e letras em caixa baixa (minúsculas) em identificadores, enquanto Lisp não é sensível à caixa. Existem algumas regras governando a tradução de nomes de nomes entre o Lisp e o Maxima.

1. Um identificador Lisp não contido entre barras verticais corresponde a um identificador Maxima em caixa baixa. Se o identificador Lisp estiver em caixa alta, caixa baixa, ou caixa mista, é ignorado. E.g., Lisp `$foo`, `$FOO`, e `$Foo` todos correspondem a Maxima `foo`.
2. Um identificador Lisp que está todo em caixa alta ou todo em caixa baixa e contido em barras verticais corresponde a um identificador Maxima com caixa invertida. Isto é, caixa alta é alterada para caixa baixa e caixa baixa para caixa alta. E.g., Lisp `|$FOO|` e `|$foo|` corresponde a Maxima `foo` e `FOO`, respectivamente.
3. Um identificador Lisp que é misto de caixa alta e caixa baixa e contido entre barras verticais corresponde a um identificador Maxima com o mesma caixa. E.g., Lisp `|$Foo|` corresponde a Maxima `Foo`.

A macro Lisp `#$` permite o uso de expressões Maxima em código Lisp. `#$`*expr*`$` expande para uma expressão Lisp equivalente à expressão Maxima *expr*.

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

Isso tem o mesmo efeito que digitar

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

A função Lisp `displa` imprime uma expressão em formato Maxima.

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

Funções definidas em Maxima não são funções comuns em Lisp. A função Lisp `mfuncall` chama uma função Maxima. Por exemplo:

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

Algumas funções Lisp possuem o mesmo nome que no pacote Maxima, a saber as seguintes.

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

3.3 Descartando

Computação simbólica tende a criar um bom volume de arquivos temporários, e o efetivo manuseio disso pode ser crucial para sucesso completo de alguns programas.

Sob GCL, nos sistemas UNIX onde a chamada de sistema mprotect (controle de acessso autorizado a uma região de memória) está disponível (incluindo SUN OS 4.0 e algumas variantes de BSD) uma organnização de arquivos temporários estratificada está disponível. Isso limita a organização para páginas que tenham sido recentemente escritas. Veja a documentação da GCL sob ALLOCATE e GBC. No nível do Lisp fazendo (setq si::*notify-gbc* t) irá ajudar você a determinar quais áreas podem precisar de mais espaço.

3.4 Documentação

O manual on-line de usuário do Maxima pode ser visto em diferentes formas. A partir da linha de comando interativa do Maxima, o manual de usuário é visto em texto plano através do comando `?` (i.e., a função `describe`). O manual de usuário é visto como hipertexto `info` através do programa visualizador `info` e como uma web page através de qualquer navegador web comum.

`example` mostra exemplos de muitas funções do Maxima. Por exemplo,

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

retorna

```
(%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
```

e saída adicional.

3.5 Definições para Ajuda

demo (*nomedearquivo*) Função

Avalia expressões Maxima em *nomedearquivo* e mostra os resultados. `demo` faz uma pausa após avaliar cada expressão e continua após a conclusão com um enter das entradas de usuário. (Se executando em Xmaxima, `demo` pode precisar ver um ponto e vírgula `;` seguido por um enter.)

`demo` procura na lista de diretórios `file_search_demo` para achar `nomedearquivo`. Se o arquivo tiver o sufixo `dem`, o sufixo pode ser omitido. Veja também `file_search`.

`demo` avalia seus argumento. `demo` retorna o nome do arquivo de demonstração.

Exemplo:

```
(%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*) Função

Encontra todos os ítens docmentados que possuem *string* em seus títulos. Se existe mais de um de tal ítem, Maxima solicita ao usuário selecionar um ítem ou ítens para mostrar. Na linha de comando interativa, `? foo` (com um espaço entre `?` e `foo`) é equivalente a `describe ("foo")`.

`describe ("")` retorna uma lista de todos os tópicos documentados no manual online.

`describe` não avalia seu argumento. `describe` sempre retorna `false`.

Exemplo:

```
(%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, [...]
```

Nesse , ítens 7 e 8 foram selecionados. Todos ou nenhum dos ítens poderia ter sido selecionado através da inserção de `all` ou `none`, que podem ser abreviado para `a` ou para `n`, respectivamente.

veja ção a Ajuda-snt [Introdução a Ajuda], página ção a Ajuda-pg

example (*topic*) Função
example () Função

`example (topic)` mostra alguns exemplos de *topic*, que é um símbolo (não uma seqüência de caracteres). A maioria dos tópicos são nomes de função. `example ()` retorna a lista de todos os tópicos reconhecidos.

O nome do arquivo contendo os exemplos é dado pela variável global `manual_demo`, cujo valor padrão é `"manual.demo"`.

`example` não avalia seu argumento. `example` retorna `done` a menos que ocorra um erro ou não exista argumento, nesse caso `example` retorna uma lista de todos os tópicos reconhecidos.

Exemplos:

```
(%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 Linha de Comando

4.1 Introdução a Linha de Comando

"'" operador

O operador apóstrofo ' evita avaliação.

Aplicado a um símbolo, o apóstrofo evita avaliação do símbolo.

Aplicado a uma chamada de função, o apóstrofo evita avaliação da chamada de função, embora os argumentos da função sejam ainda avaliados (se a avaliação nào for de outra forma evitada). O resultado é a forma substantiva da chamada de função.

Aplicada a uma espressão com parêntesis, o apóstrofo evita avaliação de todos os símbolos e chamadas de função na expressão. E.g., `'(f(x))` significa não avalie a expressão `f(x)`. `'f(x)` (com apóstrofo aplicado a `f` em lugar de `f(x)`) retorna a forma substantiva de `f` aplicada a `[x]`.

O apóstrofo nao evita simplificação.

Quando o sinalizador global `noundisp` for `true`, substantivos são mostrados com um apóstrofo. Ess comutador é sempre `true` quando mostrando definições de funções.

Veja também operador apóstrofo-apóstrofo '' e `nouns`.

Exemplos:

Aplicado a um símbolo, o apóstrofo evita avaliação do símbolo.

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

Aplicado a uma chamada de função, o apóstrofo evita avaliação da chamada de função. O resultado é a forma substantiva da chamada de função.

```
(%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
```

Aplicado a uma expressão com parêntesis, o apóstrofo evita avaliação de todos os símbolos e chamadas de função na expressão.

```
(%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
```

O apóstrof não evita simplificação.

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

"'" Operador

O operador '' (aspas simples) faz com que ocorra uma avaliação extra. E.g., `''%i4` irá reavaliar a linha de entrada `%i4`. `''(f(x))` significa avalie a expressão `f(x)` novamente. `''f(x)` (com as aspas simples aplicadas a `f` em lugar de `f(x)`) significa retorne a forma verbal de `f` aplicada a `[x]`.

4.2 Definições para Linha de Comando

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

provê um nome alternativo para uma função (de usuário ou de sistema), variável, array, etc. Qualquer número de argumentos pode ser usado.

debugmode Variável de opção

Valor padrão: `false`

Quando um erro do Maxima ocorre, Maxima iniciará o depurador se `debugmode` é `true`. O usuário pode informar comandos para examinar o histórico de chamadas, marcar pontos de parada, percorrer uma linha por vez o código do Maxima, e assim por diante. Veja `debugging` para uma lista de opções do depurador.

Habilitando `debugmode` não serão capturados erros do Lisp.

ev (*expr, arg_1, ..., arg_n*) Função

Avalia a expressão *expr* no ambiente especificado pelos argumentos *arg_1*, ..., *arg_n*. Os argumentos são comutadores (sinalizadores Booleanos), atribuições, equações, e funções. `ev` retorna o resultado (outra expressão) da avaliação.

A avaliação é realizada em passos, como segue.

1. Primeiro o ambiente é preparado examinando os argumentos que podem ser quaisquer ou todos os seguintes.
   - `simp` faz com que *expr* seja simplificado independentemente da posição do comutador `simp` que inibe simplificação se `false`.
   - `noeval` suprime a fase de avaliação de `ev` (veja passo (4) adiante). Isso é útil juntamente com outros comutadores e faz com que *expr* seja simplificado novamente sem ser reavaliado.
   - `nouns` causa a avaliação de formas substantivas (tipicamente funções não avaliadas tais como `'integrate` ou `'diff`) em *expr*.
   - `expand` causa expansão.
   - `expand (`*m*`,` *n*`)` causa expansão, alterando os valores de `maxposex` e `maxnegex` para *m* e *n* respectivamente.
   - `detout` faz com que qualquer matriz inversa calculada em *expr* tenha seu determinante mantido fora da inversa ao invés de dividindo a cada elementos.
   - `diff` faz com que todas as diferenciações indicadas em *expr* sejam executadas.
   - `derivlist (`*x*`,` *y*`,` *z*`, ...)` causa somente diferenciações referentes às variáveis indicadas.
   - `float` faz com que números racionais não integrais sejam convertidos para ponto flutuante.
   - `numer` faz com que algumas funções matemáticas (incluindo a exponenciação) com argumentos sejam valiadas em ponto flutuante. Isso faz com que variávels em *expr* que tenham sido dados numervals sejam substituídas por seus valores. Isso também modifica o comutador `float` para ativado.
   - `pred` faz com que predicados (expressões que podem ser avaliados em `true` ou `false`) sejam avaliadas.
   - `eval` faz com que uma avaliação posterior de *expr* ocorra. (Veja passo (5) adiante.)
   - `A` onde `A` é um átomo declarado seja um sinalizador de avaliação (veja `evflag`) faz com que `A` seja associado a `true` durante a avaliação de *expr*.
   - `V: expresão` (ou alternativamente `V=expressão`) faz com que `V` seja associado ao valor de `expressão` durante a avaliação de *expr*. Note que se `V` é uma opção do Maxima, então `expression` é usada para seu valor durante a avaliação de *expr*. Se mais que um argumento para `ev` é desse tipo então a associaão termina em paralelo. Se `V` é uma expressão não atômica então a substituição, ao invés de uma associação, é executada.
   - `F` onde `F`, um nome de função, tenha sido declarado para ser uma função de avaliação (veja `evfun`) faz com que `F` seja aplicado a *expr*.

- Qualquer outro nome de função (e.g., `sum`) causa a avaliação de ocorrências desses nomes em *expr* mesmo que eles tenham sido verbos.
- De forma adicional uma função ocorrendo em *expr* (digamos `F(x)`) pode ser definida localmente para o propósito dessa avaliação de *expr* dando `F(x) := expressão` como um argumento para `ev`.
- Se um átomo não mensionado adiante ou uma variável subscrita ou expressão subscrita for dada como um argumento, isso é avaliado e se o resultado é uma equação ou uma atribuição então a associação indicada ou substituição é executada. Se o resultado é uma lista então os membros da lista serão tratados como se eles fossem argumentos adicionais dados para `ev`. Isso permite uma lista de equações seja dada (e.g. `[X=1, Y=A**2]`) ou uma lista de nomes de equações (e.g., `[%t1, %t2]` onde `%t1` e `%t2` são equações) tais como aquelas retornadas por `solve`.

Os argumentos de `ev` podem ser dados em qualquer ordem com exceção de substituições de equações que são manuseadas em seqüência, da esquerda para a direita, e funções de avaliação que são compostas, e.g., `ev (`*expr*`, ratsimp, realpart)` é manuseada como `realpart (ratsimp (`*expr*`))`.

Os comutadores `simp`, `numer`, `float`, e `pred` podem também ser alterados localmente em um bloco, ou globalmente no Maxima dessa forma eles irã permanecer em efeito até serem resetados.

Se *expr* for uma expressão racional canônica (CRE), então a expressão retornada por `ev` é também uma CRE, provida a comutadores `numer` e `float` não sejam ambos `true`.

2. Durante o passo (1), uma lista é feita de variáveis não subscritas aparecendo do lado esquerdo das equações nos argumentos ou nos valores de alguns argumentos se o valor é uma equação. As variáveis (variáveis subscritas que não possuem funções array associadas bem como variáveis não subscritas) na expressão *expr* são substituídas por seus valores globais, exceto para esse aparecendo nessa lista. Usualmente, *expr* é apenas um rótulo ou `%` (como em `%i2` no exemplo adiante), então esse passo simplesmente repete a expressão nomeada pelo rótulo, de modo que `ev` possa trabalhar sobre isso.
3. Se quaisquer substituições são indicadas pelos argumentos, elas serão realizadas agora.
4. A expressão resultante é então reavaliada (a menos que um dos argumentos seja `noeval`) e simplificada conforme os argumentos. Note que qualquer chamada de função em *expr* será completada depois das variáveis nela serem avalidas e que `ev(F(x))` dessa forma possa comportar-se como `F(ev(x))`.
5. Se um dos argumentos for `eval`, passos (3) e (4) serão repetidos.

Exemplos

```
(%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
```

Uma sintaxe alternativa de alto nível tem sido provida por `ev`, por meio da qual se pode apenas digitar seus argumentos, sem o `ev()`. Isto é, se pode escrever simplesmente

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

Isso não é permitido como parte de outra expressão, e.g., em funções, blocos, etc.

Observe o processo de associação paralela no seguinte exemplo.

```
(%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 Propriedade

Alguns sinalizadores Booleanos possuem a propriedae `evflag`. `ev` trata tais sinalizadores especialmente. Um sinalizador com a propriedade `evflag` não será associado a `true` durante a execução de `ev` se isso é mensionado na chamada para `ev`. Por exemplo, `demoivre` e `ratfac` são associados a `true` durante a chamada `ev (%, demoivre, ratfac)`.

Os sinalizadores qu possuem a propriedade `evflag` são: `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`.

A construção `:lisp (putprop '|$foo| t 'evflag)` fornece a propriedade `evflag` para a variável `foo`, então `foo` ássociada a `true` durante a chamada `ev (%, foo)`. Equivalentemente, `ev (%, foo:true)` tem o mesmo efeito.

evfun Propriedade

Agumas funções possuem a proriedade `evfun`. `ev` trata cada função especialmente. Uma função com a propriedade `evfun` será aplicada durante a execução de `ev` se isso for mecionado na chamada a `ev`. Por exemplo, `ratsimp` e `radcan` será aplicada durante a chamada `ev (%, ratsimp, radcan)`.

As funções que possuem a propriedade `evfun` são: `bfloat`, `factor`, `fullratsimp`, `logcontract`, `polarform`, `radcan`, `ratexpand`, `ratsimp`, `rectform`, `rootscontract`, `trigexpand`, and `trigreduce`.

A construção `:lisp (putprop '|$foo| t 'evfun)` fornece a propriedade `evfun` para a função `foo`, de modo que `foo` é aplicada durante a chamada `ev (%, foo)`. Equivalentemente, `foo (ev (%))` tem o mesmo efeito.

infeval Variável de opção

Habilita o modo "avaliação infinita". `ev` repetidamente avalia uma expressão até que ela permaneça invariante. Para prevenir uma variável, digamos `X`, seja demoradamente avaliada nesso modo, simplesmente inclua `X='X` como um argumento para `ev`. Certamente expressões tais como `ev (X, X=X+1, infeval)` irão gerar um ciclo infinito.

kill (*symbol_1, ..., symbol_n*) Função
kill (*labels*) Função
kill (*inlabels, outlabels, linelabels*) Função
kill (*n*) Função
kill ([*m, n*]) Função
kill (*values, functions, arrays, ...*) Função
kill (*all*) Função
kill (*allbut* (*symbol_1, ..., symbol_n*)) Função

Remove todas as associações (valor, funções, array, ou regra) dos argumentos *symbol_1, ..., symbol_n*. Um argumento pode ser um elemento simples de um array ou uma função subscrita.

Muitos argumentos especiais são reconhecidos. Diferentes famílias de argumentos podem ser combinadas, e.g., `kill (inlabels, functions, allbut (foo, bar))`

todos os rótulos de entrada, de saída, e de expressões intermediárias criados até então. `kill (inlabels)` libera somente rótudos de entrada que começam com o valor corrente de `inchar`. De forma semelhante, `kill (outlabels)` libera somente rótulos de saída que começam com o valor corrente de `outchar`, e `kill (linelabels)` libera somente rótulos de expressões intermediárias que começam com o valor corrente de `linechar`.

`kill (n)`, onde *n* é um inteiro, libera os *n* mais recentes rótulos de entrada e saída.

`kill ([`*m*`, `*n*`])` libera rótulos de entrada e saída de *m* até *n*.

`kill (`*infolist*`)`, onde *infolist* é um ítem em `infolists` (tais como `values`, `functions`, ou `arrays`) libera todos os ítens em *infolist*. Veja também `infolists`.

`kill (all)` liberar todos os ítens em todas as infolists. `kill (all)` não retorna variáveis globais para seus valores padrões; Veja `reset` sobre esse ponto.

`kill (allbut (`*symbol_1*`, ..., `*symbol_n*`))` libera todos os ítens em todas as infolists exceto para *symbol_1*, ..., *symbol_n*. `kill (allbut (`*infolist*`))` libera todos os ítens exceto para si próprio em *infolist*, onde *infolist* é `values`, `functions`, `arrays`, etc.

A memória usada por uma propriedade de associação não será liberada até que todos os símbolos sejam liberados disso. Em particular, para liberar a memória usada pelo valor de um símbolo, deve-se liberar o rótulo de saída que mosta o valor associado, bem como liberando o próprio símbolo.

`kill` não avalia seus argumentos. O operador aspas simples, `''`, faz com que ocorra avaliação.

`kill (`*symbol*`)` libera todas as propriedades de *symbol*. Em oposição, `remvalue`, `remfunction`, `remarray`, e `remrule` liberam uma propriedade específica.

`kill` sempre retorna `done`, igualmente se um argumento não tem associações.

labels (*symbol*) Função
labels Variável de sistema

Retorna a lista de rótulos de entradas, de saída, de expressões intermediárias que começam com *symbol*. Tipicamente *symbol* é o valor de `inchar`, `outchar`, ou `linechar`. O caracter rótulo pode ser dado com ou sem o sinal de porcentagem, então, por exemplo, `i` e `%i` retornam o mesmo resultado.

Se nenhum rótulo começa com *symbol*, `labels` retorna uma lista vazia.

A função `labels` não avalia seu argumento. O operador aspas simples `''` faz com que ocorra avaliação. Por exemplo, `labels (''inchar)` retorna os rótulos de entrada que começam com o caractere corrente do rótulo de entrada.

A variável `labels` é uma lista de rótulos de entrada, saída, e de expressòes intermediárias, incluindo todos os rótulos anteriores se `inchar`, `outchar`, ou `linechar` foram redefinidos.

Por padrão, Maxima mostra o resultado de cada expressão de entrada do usuário, dando ao resultado um rótulo de saída. A exibição da saída é suprimida pelo encerramento da entrada com `$` (sinal de dolar) em lugar de `;` (ponto e vírgula). Um rótulo de saída é gerado, mas não é mostrado, e o rótulo pode ser referenciado da mesma forma que rótulos de saída mostrados. Veja também `%`, `%%`, e `%th`.

Rótulos de expressões intermediárias podem ser gerados por algumas funções. O sinalizador `programmode` controla se `solve` e algumas outras funções geram rótulos de expressões intermediárias em lugar de retornar uma lista de expressões. Algumas outras funções, tais como `ldisplay`, sempre gera rótulos de expressões intermediárias.

`first (rest (labels (''inchar)))` retorna o rótulo de entrada mais recente.

Veja também `inchar`, `outchar`, `linechar`, e `infolists`.

linenum Variável de sistema

Retorna o número da linha do par corrente de expressões de entrada e saída.

myoptions Variável de sistema

Valor padrão: `[]`

`myoptions` é a lista de todas as opções alguma vez alteradas pelo usuário, tenha ou não ele retornado a alteração para o seu valor padrão.

nolabels Variável de opção

Valor padrão: `false`

Quando `nolabels` é `true`, rótulos de entrada e saída são gerados mas não adicionados ao final de `labels`, a lista de todos os rótulos de entrada e saída. `kill (labels)` elimina os rótulos na lista `labels`, mas não elimina qualquer rótulo gerado desde `nolabels` foi atribuído `true`. Isso dá a entender que provavelmente esse comportamento está simplesmente quebrado.

Veja também `batch`, `batchload`, e `labels`.

optionset Variável de opção

Valor padrão: `false`

Quando `optionset` é `true`, Maxima mostra uma mensagem sempre que uma opção do Maxima é alterada. Isso é útil se o usuário está incerto sobre a ortografia de alguma opção e quer ter certeza que a variável por ele atribuído um valor foi realmente uma variável de opção.

playback () Função
playback (*n*) Função
playback ([*m*, *n*]) Função
playback ([*m*]) Função
playback (*input*) Função
playback (*slow*) Função
playback (*time*) Função
playback (*grind*) Função

Mostra expressões de entrada, de saída, e expressões intermediárias, sem refazer os cálculos. `playback` somente mostra as expressões associadas a rótulos; qualquer outra saída (tais como textos impressos por `print` ou `describe`, ou messagens de erro) não é mostrada. Veja também `labels`.

`playback` não avalia seus argumentos. O operador aspas simples, `''`, sobrepõe-se às aspas. `playback` sempre retorna `done`.

`playback ()` (sem argumentos) mostra todas as entradas, saídas e expressões intermediárias geradas até então. Uma expressão de saída é mostrada mesmo se for suprimida pelo terminador `$` quando ela tiver sido originalmente calculada.

`playback (`*n*`)` mostra as mais recentes *n* expressões. Cada entrada, saída e expressão intermediária conta como um.

`playback ([`*m*`, `*n*`])` mostra entradas, saídas e expressões intermediárias com os números de *m* até *n*, inclusive.

`playback ([m])` é equivalente a `playback ([m, m])`; isso usualmente imprime um par de expressões de entrada e saída.

`playback (input)` mostra todas as expressões de entrada geradas até então.

`playback (slow)` insere pausas entre expressões e espera que o usuário pressione `enter`. Esse comportamento é similar a `demo`. `playback (slow)` é útil juntamente com `save` ou `stringout` quando criamos um arquivo secundário de armazenagem com a finalidade de capturar expressões úteis.

`playback (time)` mostra o tempo de computação de cada expressão.

`playback (grind)` mostra expressões de entrada no mesmo formato da função `grind`. Expressões de saída não são afetadas pela opção `grind`. Veja `grind`.

Argumentos podem ser combinados, e.g., `playback ([5, 10], grind, time, slow)`.

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

Mostra a propriedade como o indicador *i* associada com o átomo *a*. *a* pode também ser uma lista de átomos ou o átomo `all` nesse caso todos os átomos com a propriedade dada serão usados. Por exemplo, `printprops ([f, g], atvalue)`. `printprops` é para propriedades que não podem ser mostradas de outra forma, i.e. para `atvalue`, `atomgrad`, `gradef`, e `matchdeclare`.

prompt Variável de opção

Valor padrão: `_`

`prompt` é o símbolo de linha de comando da função `demo`, modo `playback (slow)`, e o Maxima interrompe o ciclo (como invocado por `break`).

quit () Função

Encerra a sessão do Maxima. Note que a função pode ser invocada como `quit();` ou `quit()$`, não por si mesma `quit`.

Para parar um cálculo muito longo, digite `control-C`. A ação padrão é retornar à linha de comando do Maxima. Se `*debugger-hook*` é `nil`, `control-C` abre o depurador Lisp. Veja também `debugging`.

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

Desassocia as definições de função dos símbolos *f_1*, ..., *f_n*. Os argumentos podem ser os nomes de funções comuns (criadas por meio de `:=` ou `define`) ou funções macro (criadas por meio de `::=`).

`remfunction (all)` desassocia todas as definições de funcção.

`remfunction` coloca um ap'ostrofo em seus argumentos (não os avalia).

`remfunction` retorna uma lista de símbolos para a qual a definição de função foi desassociada. `false` é retornado em lugar de qualquer símbolo para o qual não exista definição de função.

reset () Função

Retorna muitas variáveis globais e opções, e algumas outras variáveis, para seus valores padrões.

`reset` processa as variáveis na lista Lisp `*variável-initial-values*`. A macro Lisp `defmvar` coloca variáveis nessa lista (entre outras ações). Muitas, mas não todas, variáveis globais e opções são definidas por `defmvar`, e algumas variáveis definidas por `defmvar` não são variáveis globais ou opções.

showtime Variável de opção

Valor padrão: `false`

Quando `showtime` é `true`, o tempo de computação e o tempo decorrido são impressos na tela com cada expressão de saída.

O tempo de cálgulo é sempre gravado, então `time` e `playback` podem mostrar o tempo de cálculo mesmo quando `showtime` for `false`.

Veja também `timer`.

sstatus (*feature*, *package*) Função

Altera o status de *feature* em *package*. Após `sstatus (`*feature*`,` *package*`)` é executado, `status (`*feature*`,` *package*`)` retorna `true`. Isso pode ser útil para quem escreve pacotes, para manter um rastro de quais recursos os pacotes usam.

to_lisp () Função

Insere o sistema Lisp dentro do Maxima. `(to-maxima)` retorna para o Maxima.

values Variável de sistema

Valor inicial: `[]`

`values` é uma lista de todas as varáveis de usuário associadas (não opções Maxima ou comutadores). A lista compreende símbolos associados por `:` , `::`, ou `:=`.

5 Operadores

5.1 "N" Argumentos

Um operador `nary` é usado para denotar uma função com qualquer número de argumentos, cada um dos quais é separado por uma ocorrência do operador, e.g. A+B ou A+B+C. A função `nary("x")` é uma função de extensão sintática para declarar x como sendo um operador `nary`. Funções podem ser declaradas para serem `nary`. Se `declare(j,nary);` é concluída, diz ao simplicador para simplificar, e.g. `j(j(a,b),j(c,d))` para `j(a, b, c, d)`.

Veja também `syntax`.

5.2 Sem Argumentos

Operadores `nofix` são usados para denotar funções sem argumentos. A mera presença de tal operador em um comando fará com que a função correspondente seja avaliada. Por exemplo, quando se digita "exit;" para sair de uma parada do Maxima, "exit" tem comportamento similar a um operador `nofix`. A função `nofix("x")` é uma função de extensão sintática que declara x como sendo um operador `nofix`.

Veja também `syntax`.

5.3 Operador

Veja `operators`.

5.4 Operador Pósfixado

Operadores `postfix` como a variedade `prefix` denotam funções de um argumento simples, mas nesse caso o argumento sucede imediatamente uma ocorrência do operador na seqüência de caracteresde entrada, e.g. 3! . Uma função `postfix("x")` é uma função de extensão sintática que declara x como sendo um operador `postfix`.

Veja também `syntax`.

5.5 Operador Préfixado

Um operador `prefix` é um que significa uma função de um argumento, o qual argumento imediatamente segue uma ocorrência do operador. `prefix("x")` é uma função de extensão sintática que declara x como sendo um operador `prefix`.

Veja também `syntax`.

5.6 Definições para Operadores

"!" Operador

O operador fatorial. Para qualquer número complexo `x` (incluíndo números inteiros, racionais, e reais) exceto para inteiros negativos, `x!` é definido como `gamma(x+1)`.

Para um inteiro `x`, `x!` simplifica para o produto de inteiros de 1 a `x` inclusive. `0!` simplifica para 1. Para um número em ponto flutuante `x`, `x!` simplifica para o valor de `gamma (x+1)`. Para `x` igual a `n/2` onde `n` é um inteiro ímpar, `x!` simplifica para um fator racional vezes `sqrt (%pi)` (uma vez que `gamma (1/2)` é igual a `sqrt (%pi)`). Se `x` for qualquer outra coisa, `x!` não é simplificado.

As variáveis `factlim`, `minfactorial`, e `factcomb` controlam a simplificação de expressões contendo fatoriais.

As funções `gamma`, `bffac`, e `cbffac` são variedades da função `gamma`. `makegamma` substitui `gamma` para funções relacionadas a fatoriais.

Veja também `binomial`.

- O fatorial de um inteiro, inteiro dividido por dois, ou argumento em ponto flutuante é simplificado a menos que o operando seja maior que `factlim`.

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

- O fatorial de um número complexo, constante conhecida, ou expressão geral não é simplificado. Ainda assim pode ser possível simplificar o fatorial após avaliar o operando.

```
(%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]
```

- O fatorial de um símbolo não associado não é simplificado.

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

- Fatoriais são simplificados, não avaliados. Dessa forma `x!` pode ser substituído mesmo em uma expressão com apóstrofo.

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

"!!" Operador

O operador de duplo fatorial.

Para um número inteiro, número em ponto flutuante, ou número racional `n`, `n!!` avalia para o produto `n (n-2) (n-4) (n-6) ... (n - 2 (k-1))` onde `k` é igual a `entier (n/2)`, que é, o maior inteiro menor que ou igual a `n/2`. Note que essa definição não coincide com outras definições publicadas para argumentos que não são inteiros.

Para um mesmo inteiro (ou ímpar) `n`, `n!!` avalia para o produto de todos os consecutivos mesmo inteiros (ou ímpares) de 2 (ou 1) até `n` inclusive.

Para um argumento `n` que não é um número inteiro, um número em ponto flutuante, ou um número racional, `n!!` retorna uma forma substantiva `genfact (n, n/2, 2)`.

"#" Operador

Representa a negação da igualdade sintática `=`.

Note que pelo fato de as regras de avaliação de expressões predicadas (em particular pelo fato de `not` *expr* fazer com que a avaliação de *expr*), a forma `not` *a* `=` *b* não é equivalente à forma *a* `#` *b* em alguns casos.

Exemplos:

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

"." Operador

O operador ponto, para multiplicação (não comutativa) de matrizes. Quando "." é usado com essa finalidade, espaços devem ser colocados em ambos os lados desse operador, e.g. A . B. Isso distingüe o operador ponto plenamente de um ponto decimal em um número em ponto flutuante.

Veja também `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, e `dotscrules`.

":" Operador

O operador de atribuição. E.g. A:3 escolhe a variável A para 3.

"::" Operador

Operador de atribuição. :: atribui o valor da expressão em seu lado direito para o valor da quantidade na sua esquerda, que pode avaliar para uma variável atômica ou variável subscrita.

"::=" Operador

Operador de definio de funo de macro. `::=` define uma funo (chamada uma "macro" por razes histricas) que coloca um apstrofo em seus argumentos (evitando avaliao), e a expresso que retornada (chamada a "expanso de macro") avaliada no contexto a partir do qual a macro foi chamada. Uma funo de macro de outra forma o mesmo que uma funo comum.

`macroexpand` retorna uma expanso de macro (sem avaliar a expanso). `macroexpand (foo (x))` seguida por `''%` equivalente a `foo (x)` quando `foo` for uma funo de macro.

`::=` coloca o nome da nova funo de macro dentro da lista global `macros`. `kill`, `remove`, e `remfunction` desassocia definies de funo de macro e remove nomes de `macros`.

`fundef` or `dispfun` retorna uma definio de funo de macro ou atribui isso a um rtulo, respectivamente.

Funes de macro comumente possuem expresses `buildq` e `splice` para construir uma expresso, que ento avaliada.

Exemplos

Uma funo de macro coloca um apstrofo em seus argumentos evitando ento a avaliao, ento mensagem (1) mostra `y - z`, no o valor de `y - z`. A expanso de macro (a expresso com apstrofo `'(print ("(2) x is equal to", x))` avaliada no contexto a partir do qual a macro for chamada, mostrando a mesnagem (2).

```
(%i1) x: %pi;
(%o1)                           %pi
(%i2) y: 1234;
(%o2)                          1234
(%i3) z: 1729 * w;
(%o3)                        1729 w
(%i4) printq1 (x) ::= block (print ("(1) x  igual a", x), '(print ("(2) x  igu
(%o4) printq1(x) ::= block(print("(1) x  igual a", x),
                                '(print("(2) x  igual a", x)))
(%i5) printq1 (y - z);
(1) x  igual a y - z
(2) x  igual a %pi
(%o5)                           %pi
```

Uma funo comum avalia seus argumentos, ento message (1) mostra o valor de `y - z`. O valor de rtorno no avaliado, ento mensagem (2) no mostrada at a avaliao explcita `''%`.

```
(%i1) x: %pi;
(%o1)                           %pi
(%i2) y: 1234;
(%o2)                          1234
(%i3) z: 1729 * w;
(%o3)                        1729 w
(%i4) printe1 (x) := block (print ("(1) x  igual a", x), '(print ("(2) x  igual
(%o4) printe1(x) := block(print("(1) x  igual a", x),
                                '(print("(2) x  igual a", x)))
(%i5) printe1 (y - z);
(1) x  igual a 1234 - 1729 w
```

```
(%o5)              print((2) x  igual a, x)
(%i6) ''%;
(2) x  igual a %pi
(%o6)                           %pi
```

`macroexpand` retorna uma expanso de macro. `macroexpand (foo (x))` seguido por `''%` equivalente a `foo (x)` quando `foo` for uma funo de macro.

```
(%i1) x: %pi;
(%o1)                           %pi
(%i2) y: 1234;
(%o2)                          1234
(%i3) z: 1729 * w;
(%o3)                        1729 w
(%i4) g (x) ::= buildq ([x], print ("x  igual a", x));
(%o4)    g(x) ::= buildq([x], print("x  igual a", x))
(%i5) macroexpand (g (y - z));
(%o5)               print(x  igual a, y - z)
(%i6) ''%;
x  igual a 1234 - 1729 w
(%o6)                     1234 - 1729 w
(%i7) g (y - z);
x  igual a 1234 - 1729 w
(%o7)                     1234 - 1729 w
```

":=" Operador

O operador de definição de função. E.g. `f(x):=sin(x)` define uma função `f`.

"=" Operador

denota uma equação para o Maxima. Para o verificador de modelos no Maxima isso denota uma relação total que prende duas expressões se e somente se as expressões são sintaticamente idênticas.

A negação de `=` é representada por `#`.

Note que pelo fato de as regras de avaliação de expressões predicadas (em particular pelo fato de `not` *expr* fazer com que a avaliação de *expr*), a forma `not` *a* `=` *b* não é equivalente à forma *a* `#` *b* em alguns casos.

and Operador

O operador lógico de conjunção. `and` é um operador n-ário infixo; seus operandos são expressões Booleanas, e seu resultado é um valor Booleano.

`and` força avaliação (como `is`) de um ou mais operandos, e pode forçar a avaliação de todos os operandos.

Operandos são avaliados na ordem em que aparecem. `and` avalia somente quantos de seus operandos forem necessários para determinar o resultado. Se qualquer operando for `false`, o resultado é `false` e os operandos restantes não são avaliados.

O sinalizador global `prederror` governa o comportamento de `and` quando um operando avaliado não pode ser determinado como sendo `true` ou `false`. `and`

imprime uma mensagem de erro quando `prederror` for `true`. De outra forma, `and` retorna `unknown` (desconhecido).

`and` não é comutativo: `a and b` pode não ser igual a `b and a` devido ao tratamento de operandos indeterminados.

or Operador

O operador lógico de disjunção. `or` é um operador n-ário infixo; seus operandos são expressões Booleanas, e seu resultado é um valor Booleano.

`or` força avaliação (como `is`) de um ou mais operandos, e pode forçar a avaliação de todos os operandos.

Operandos são avaliados na ordem em que aparecem. `or` avalia somente quantos de seus operandos forem necessários para determinar o resultado. Se qualquer operando for `true`, o resultado é `true` e os operandos restantes não são avaliados.

O sinalizador global `prederror` governa o comportamento de `or` quando um operando avaliado não puder ser determinado como sendo `true` ou `false`. `or` imprime uma mensagem de erro quando `prederror` for `true`. De outra forma, `or` retorna `unknown`.

`or` não é comutativo: `a or b` pode não ser igual a `b or a` devido ao tratamento de operando indeterminados.

not Operador

O operador lógico de negação. `not` é operador prefixo; Seu operando é uma expressão Booleana, e seu resultado é um valor Booleano.

`not` força a avaliação (como `is`) de seu operando.

O sinalizador global `prederror` governa o comportamento de `not` quando seu operando não pode ser determinado em termos de `true` ou `false`. `not` imprime uma mensagem de erro quando `prederror` for `true`. De outra forma, `not` retorna `unknown`.

abs (*expr*) Função

Retorna o valor absoluto de *expr*. Se *expr* for um número complexo, retorna o módulo complexo de *expr*.

additive Palavra chave

Se `declare(f,additive)` tiver sido executado, então:

(1) Se `f` for uma função de uma única variável, sempre que o simplificador encontrar `f` aplicada a uma adição, `f` será distribuído sobre aquela adição. I.e. `f(y+x)` irá simplificar para `f(y)+f(x)`.

(2) Se `f` for uma função de 2 ou mais argumentos, a adição é definida como adição no primeiro argumento para `f`, como no caso de `sum` ou `integrate`, i.e. `f(h(x)+g(x),x)` irá simplificar para `f(h(x),x)+f(g(x),x)`. Essa simplificação não ocorre quando `f` é aplicada para expressões da forma `sum(x[i],i,lower-limit,upper-limit)`.

allbut Palavra chave

trabalha com os comandos `part` (i.e. `part`, `inpart`, `substpart`, `substinpart`, `dpart`, e `lpart`). Por exemplo,

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

enquanto

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

Também trabalha com o comando `kill`,

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

executará um `kill (all)` deixando fora do `kill` os nomes especificados. Nota: `name_i` significa um nome tal como nome de função `u`, `f`, `foo`, ou `g`, não um `infolist` tal como `functions`.

antisymmetric Declaração

Se `declare(h,antisymmetric)` é concluída, diz ao simplicador que `h` é uma função antisimétrica. E.g. `h(x,z,y)` simplificará para `- h(x, y, z)`. Isto é, dará (-1)^n vezes o resultado dado por `symmetric` ou `commutative`, quando n for o número de interescolhas de dois argumentos necessários para converter isso naquela forma.

cabs (*expr*) Função

Retorna o valor absoluto complexo (o módulo complexo) de *expr*.

ceiling (x) Função

Quando x for um número real, retorna o último inteiro que é maior que ou igual a x.

Se x for uma expressão constante (`10 * %pi`, por exemplo), `ceiling` avalia x usando grandes números em ponto flutuante, e aplica `ceiling` para o grande número em ponto flutuante resultante. Porque `ceiling` usa avaliação de ponto flutuante, é possível, embora improvável, que `ceiling` possa retornar uma valor errôneo para entradas constantes. Para prevenir erros, a avaliação de ponto flutuante é concluída usando três valores para `fpprec`.

Para entradas não constantes, `ceiling` tenta retornar um valor simplificado. Aqui está um exemplo de simplificações que `ceiling` conhece:

```
(%i1) ceiling(ceiling(x));
(%o1) ceiling(x)
(%i2) ceiling(floor(x));
(%o2) floor(x)
(%i3) declare(n,integer)$
(%i4) [ceiling(n), ceiling(abs(n)), ceiling(max(n,6))];
(%o4) [n, abs(n), max(n,6)]
(%i5) assume(x > 0, x < 1)$
(%i6) ceiling(x);
(%o6) 1
(%i7) tex(ceiling(a));
   $$\left \lceil a \right \rceil$$
```

A função `ceiling` não mapeia automaticamente sobre listas ou matrizes. Finalmente, para todas as entradas que forem manifestamente complexas, `ceiling` retorna uma forma substantiva.

Se o intervalo de uma função é um subconjunto dos inteiros, o intervalo pode ser declarado `integervalued`. Ambas as funções `ceiling` e `floor` podem usar essa informação; por exemplo:

```
(%i1) declare(f,integervalued)$
(%i2) floor(f(x));
(%o2) f(x)
(%i3) ceiling(f(x) -1);
(%o3) f(x)-1
```

charfun (*p*) Função

Retorna 0 quando o predicado *p* avaliar para `false`; retorna 1 quando o predicado avaliar para `true`. Quando o predicado avaliar para alguma coisa que não `true` ou `false` (`unknown`), retorna uma forma substantiva.

Exemplos:

```
(%i1) charfun(x<1);
(%o1) charfun(x<1)
(%i2) subst(x=-1,%);
(%o2) 1
(%i3) e : charfun('"and"(-1 < x, x < 1))$
(%i4) [subst(x=-1,e), subst(x=0,e), subst(x=1,e)];
(%o4) [0,1,0]
```

commutative Declaração

Se `declare(h,commutative)` é concluída, diz ao simplicador que `h` é uma função comutativa. E.g. `h(x,z,y)` irá simplificar para `h(x, y, z)`. Isto é o mesmo que `symmetric`.

compare (*x*, *y*) Função

Retorna um operador de comparação *op* (`<`, `<=`, `>`, `>=`, `=`, ou `#`) tal que `is` (*x op y*) avalia para `true`; quando ou *x* ou *y* dependendo de `%i` e *x* `#` *y*, retorna `notcomparable`; Quando não existir tal operador ou Maxima não estiver apto a determinar o operador, retorna `unknown`.

Exemplos:

```
(%i1) compare(1,2);
(%o1) <
(%i2) compare(1,x);
(%o2) unknown
(%i3) compare(%i,%i);
(%o3) =
(%i4) compare(%i,%i+1);
(%o4) notcomparable
(%i5) compare(1/x,0);
(%o5) #
(%i6) compare(x,abs(x));
(%o6) <=
```

A função `compare` não tenta de terminar se o domínio real de seus argumentos é não vazio; dessa forma

```
(%i1) compare(acos(x^2+1), acos(x^2+1) + 1);
(%o1) <
```

O domínio real de `acos (x^2 + 1)` é vazio.

entier (*x*) Função

Retorna o último inteiro menor que ou igual a *x* onde *x* é numerico. `fix` (como em `fixnum`) é um sinônimo disso, então `fix(x)` é precisamente o mesmo.

equal (*expr_1*, *expr_2*) Função

Usado com um `is`, retorna `true` (ou `false`) se e somente se *expr_1* e *expr_2* forem iguais (ou não iguais) para todos os possíveis valores de suas variáveis (como determinado por `ratsimp`). Dessa forma `is (equal ((x + 1)^2, x^2 + 2*x + 1))` retorna `true` ao passo que se `x` for não associado `is ((x + 1)^2 = x^2 + 2*x + 1)` retorna `false`. Note também que `is(rat(0)=0)` retorna `false` mas `is (equal (rat(0), 0))` retorna `true`.

Se uma determinação não pode ser feita, então `is (equal (a, b))` retorna uma expressão simplificada mas equivalente, ao passo que `is (a=b)` sempre retorna ou `true` ou `false`.

Todas as variáveis que ocorrem em *expr_1* e *expr_2* são presumidas serem valores reais.

A negação de `equal` é `notequal`. Note que devido às regras de avaliação de expressões predicadas (em particular pelo fato de `not` *expr* causar a avaliação de *expr*), `notequal` não seja equivalente a `not equal` em alguns casos.

`ev (`*expr*`, pred)` é equivalente a `is (`*expr*`)`.

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

floor (*x*) Função

Quando *x* for um número real, retorna o maior inteiro que é menor que ou igual a *x*.

Se *x* for uma expressão constante (`10 * %pi`, for exemplo), `floor` avalia *x* usando grandes números em ponto flutuante, e aplica `floor` ao grande número em ponto flutuante resultante. Porque `floor` usa avaliação em ponto flutuante, é possível, embora improvável, que `floor` não possa retornar um valor errôneo para entradas constantes. Para prevenir erros, a avaliação de ponto flutuante é concluída usando três valores para `fpprec`.

Para entradas não constantes, `floor` tenta retornar um valor simplificado. Aqui está exemplos de simplificações que `floor` conhece:

```
(%i1) floor(ceiling(x));
(%o1) ceiling(x)
(%i2) floor(floor(x));
(%o2) floor(x)
```

```
(%i3) declare(n,integer)$
(%i3) [floor(n), floor(abs(n)), floor(min(n,6))];
(%o4) [n,abs(n),min(n,6)]
(%i4) assume(x > 0, x < 1)$
(%i5) floor(x);
(%o5) 0
(%i6) tex(floor(a);
    $$\left \lfloor a \right \rfloor$$
```

A função `floor` não mapeia automaticamente sobre listas ou matrizes. Finalmente, para todas as entradas que forem manifestamente complexas, `floor` retorna uma forma substantiva.

Se o intervalo de uma função for um subconjunto dos inteiros, o intervalo pode ser declarado `integervalued`. Ambas as funções `ceiling` e `floor` podem usar essa informação; por exemplo:

```
(%i1) declare(f,integervalued)$
(%i2) floor(f(x));
(%o2) f(x)
(%i3) ceiling(f(x) -1);
(%o3) f(x)-1
```

notequal (*expr_1*, *expr_2*) Função

Representa a negação de `equal` (*expr_1*, *expr_2*).

Note que pelo fato de as regras de avaliação de expressões predicadas (em particular pelo fato de `not` *expr* causar a avaliação de *expr*), `notequal` não é equivalente a `not equal` em alguns casos.

Exemplos:

```
(%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 Operador

Como um argumento em uma chamada a `ev (`*expr*`)`, `eval` causa uma avaliação extra de *expr*. Veja `ev`.

evenp (*expr*) Função

Retorna `true` se *expr* for um inteiro sempre. `false` é retornado em todos os outros casos.

fix (*x*) Função

Um sinônimo para `entier (`*x*`)`.

fullmap (*f, expr_1, ...*) Função

Similar a `map`, mas `fullmap` mantém mapeadas para baixo todas as subexpressões até que os operadores principais não mais sejam os mesmos.

`fullmap` é usada pelo simplificador do Maxima para certas manipulações de matrizes; dessa forma, Maxima algumas vezes gera uma mensagem de erro concernente a `fullmap` mesmo apesar de `fullmap` não ter sido explicitamente chamada pelo usuário.

```
(%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, ...*) Função

Similar a `fullmap`, mas `fullmapl` somente mapeia sobre listas e matrizes.

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

is (*expr*) Função

Tenta determinar se a *expr* predicada (expressões que avaliam para `true` ou `false`) é dedutível de fatos localizados na base de dados de `assume`.

Se a dedutibilidade do predicado for `true` ou `false`, `is` retorna `true` ou `false`, respectivamente. De outra forma, o valor e retorno é controlado pelo sinalizador global `prederror`. Quando `prederror` for `false`, `is` retorna `unknown` para um predicado que não pode ser provado ou refutado, e reporta um erro de outra forma.

Veja também `assume`, `facts`, and `maybe`.

Exemplos:

`is` causa avaliação de predicados.

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

`is` tenta derivar predicados da base de dados do assume `assume`.

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

Se `is` não puder nem prover nem refutar uma forma predicada a partir da base de dados de `assume`, o sinalizador global `prederror` governa o comportamento de `is`.

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

maybe (*expr*) Função

Tenta determinar se a *expr* predicada é dedutível dos fatos na base de dados de `assume`.

Se a dedutibilidade do predicado for `true` ou `false`, `maybe` retorna `true` ou `false`, respectivamente. De outra forma, `maybe` retorna `unknown`.

`maybe` é funcinalmente equivalente a `is` com `prederror: false`, mas o resultado é computado sem atualmente atribuir um valor a `prederror`.

Veja também `assume`, `facts`, and `is`.

Exemplos:

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

isqrt (*x*) Função

Retorna o "inteiro raíz quadrada" do valor absoluto de *x*, que é um inteiro.

lmax (*L*) Função

Quando *L* for uma lista ou um conjunto, retorna `apply ('max, args (L))`. Quando *L* não for uma lista ou um conjunto, sinaliza um erro.

lmin (*L*) Função

Quando *L* for uma lista ou um conjunto, retorna `apply ('min, args (L))`. Quando *L* não for uma lista ou um conjunto, sinaliza um erro.

max (*x_1, ..., x_n*) Função

Retorna um valor simplificado para o máximo entre as expressões *x_1* a *x_n*. Quando `get (trylevel, maxmin)`, for dois ou mais, `max` usa a simplificação `max (e, -e) --> |e|`. Quando `get (trylevel, maxmin)` for 3 ou mais, *max* tenta eliminar expressões que estiverem entre dois outros argumentos; por exemplo, `max (x, 2*x, 3*x) --> max (x, 3*x)`. Para escolher o valor de `trylevel` para 2, use `put (trylevel, 2, maxmin)`.

min (*x_1, ..., x_n*) Função

Retorna um valor simplificado para o mínimo entre as expressões `x_1` through `x_n`. Quando `get (trylevel, maxmin)`, for 2 ou mais, `min` usa a simplificação `min (e, -e) --> -|e|`. Quando `get (trylevel, maxmin)` for 3 ou mais, `min` tenta eliminar expressões que estiverem entre dois outros argumentos; por exemplo, `min (x, 2*x, 3*x) --> min (x, 3*x)`. Para escolher o valor de `trylevel` para 2, use `put (trylevel, 2, maxmin)`.

polymod (*p*) Função
polymod (*p, m*) Função

Converte o polinômio *p* para uma representação modular com relação ao módulo corrente que é o valor da variável `modulus`.

`polymod (p, m)` especifica um módulo *m* para ser usado em lugar do valor corrente de `modulus`.

Veja `modulus`.

mod (*x, y*) Função

Se *x* e *y* forem números reais e *y* for não nulo, retorna `x - y * floor(x / y)`. Adicionalmente para todo real *x*, nós temos `mod (x, 0) = x`. Para uma discursão da definição `mod (x, 0) = x`, veja a Seção 3.4, de "Concrete Mathematics," por Graham, Knuth, e Patashnik. A função `mod (x, 1)` é uma função dente de serra com peródo 1 e com `mod (1, 1) = 0` e `mod (0, 1) = 0`.

Para encontrar o argumento principal (um número no intervalo `(-%pi, %pi]`) de um número complexo, use a função `x |-> %pi - mod (%pi - x, 2*%pi)`, onde *x* é um argumento.

Quando *x* e *y* forem expressões constantes (`10 * %pi`, por exemplo), `mod` usa o mesmo esquema de avaliação em ponto flutuante que `floor` e `ceiling` usam. Novamente, é possível, embora improvável, que `mod` possa retornar um valor errôneo nesses casos.

Para argumentos não numéricos *x* ou *y*, `mod` conhece muitas regras de simplificação:

```
(%i1) mod(x,0);
(%o1) x
(%i2) mod(a*x,a*y);
(%o2) a*mod(x,y)
(%i3) mod(0,x);
(%o3) 0
```

oddp (*expr*) Função

é `true` se *expr* for um inteiro ímpar. `false` é retornado em todos os outros casos.

pred Operador

Como um argumento em uma chamada a `ev (`*expr*`)`, `pred` faz com que predicados (expressões que avaliam para `true` ou `false`) sejam avaliados. Veja `ev`.

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

Um objeto de estado randômico representa o estado do gerador de números randômicos (aleatórios). O estado compreende 627 palavras de 32 bits.

`make_random_state (`*n*`)` retorna um novo objeto de estado randômico criado de um valor inteiro semente igual a *n* modulo 2^32. *n* pode ser negativo.

`make_random_state (`*s*`)` retorna uma copia do estado randômico *s*.

`make_random_state (true)` retorna um novo objeto de estado randômico, usando a hora corrente do relógio do computador como semente.

`make_random_state (false)` retorna uma cópia do estado corrente do gerador de números randômicos.

set_random_state (*s*) Função

Copia *s* para o estado do gerador de números randômicos.

`set_random_state` sempre retorna `done`.

random (*x*) Função

Retorna um número pseudorandômico. Se *x* é um inteiro, `random (x)` retorna um inteiro de 0 a *x* `-` `1` inclusive. Se *x* for um número em ponto flutuante, `random (x)` retorna um número não negativo em ponto flutuante menor que *x*. `random` reclama com um erro se *x* não for nem um inteiro nem um número em ponto flutuante, ou se *x* não for positivo.

As funções `make_random_state` e `set_random_state` mantém o estado do gerador de números randômicos.

O gerador de números randômicos do Maxima é uma implementação do algorítmo de Mersenne twister MT 19937.

Exemplos:

```
(%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
```

rationalize (*expr*) Função

Converte todos os números em ponto flutuante de precisão dupla e grandes números em ponto flutuante na expressão do Maxima *expr* para seus exatos equivalentes racionais. Se você não estiver faminilarizado com a representação binária de números em ponto flutuante, você pode se surpreender que `rationalize (0.1)` não seja igual a 1/10. Esse comportamento não é especial para o Maxima – o número 1/10 tem uma representação binária repetitiva e não terminada.

```
(%i1) rationalize(0.5);
(%o1) 1/2
(%i2) rationalize(0.1);
(%o2) 3602879701896397/36028797018963968
(%i3) fpprec : 5$
(%i4) rationalize(0.1b0);
(%o4) 209715/2097152
(%i5) fpprec : 20$
(%i6) rationalize(0.1b0);
(%o6) 236118324143482260685/2361183241434822606848
(%i7) rationalize(sin(0.1 * x + 5.6));
(%o7) sin((3602879701896397*x)/36028797018963968+3152519739159347/562949953421
```

Exemplo de utilização:

```
unitfrac(r) := block([uf : [], q],
   if not(ratnump(r)) then error("The input to 'unitfrac' must be a rational n
   while r # 0 do (
```

```
        uf : cons(q : 1/ceiling(1/r), uf),
        r : r - q),
   reverse(uf));

(%i2) unitfrac(9/10);
(%o2) [1/2,1/3,1/15]
(%i3) apply("+",%);
(%o3) 9/10
(%i4) unitfrac(-9/10);
(%o4) [-1,1/10]
(%i5) apply("+",%);
(%o5) -9/10
(%i6) unitfrac(36/37);
(%o6) [1/2,1/3,1/8,1/69,1/6808]
(%i7) apply("+",%);
(%o7) 36/37
```

sign (*expr*) Função

Tenta determinar o sinal de *expr* a partir dos fatos na base de dados corrente. Retorna uma das seguintes respostar: `pos` (positivo), `neg` (negativo), `zero`, `pz` (positivo ou zero), `nz` (negativo ou zero), `pn` (positivo ou negativo), ou `pnz` (positivo, negativo, ou zero, i.e. nada se sabe sobre o sinal da epressão).

signum (*x*) Função

Para um *x* numérico retorna 0 se *x* for 0, de outra forma retorna -1 ou `+1` à medida que *x* seja menor ou maior que 0, respectivamente.

Se *x* não for numérico então uma forma simplificada mas equivalente é retornada. Por exemplo, `signum(-x)` fornece `-signum(x)`.

sort (*list*, *p*) Função

sort (*list*) Função

Ordena a *list* conforme o predicado `p` de dois argumentos, tais como `"<"` ou `orderlessp`.

`sort` (*list*) ordena a *list* conforme a ordem interna do Maxima.

list pode conter ítens numéricos ou não numéricos, ou ambos.

sqrt (*x*) Função

A raíz quadrada de *x*. É representada internamente por `x^(1/2)`. Veja também `rootscontract`.

`radexpand` se `true` fará com que n-ésimas raízes de fatores de um produto que forem potências de n sejam colocados fora do radical, e.g. `sqrt(16*x^2)` retonará `4*x` somente se `radexpand` for `true`.

sqrtdispflag Variável de opção

Valor padrão: `true`

Quando `sqrtdispflag` for `false`, faz com que `sqrt` seja mostrado como expoente 1/2.

sublis (*list*, *expr*) Função

Faz multiplas substituições paralelas dentro de uma expressão.

A variável `sublis_apply_lambda` controla a simplificação após `sublis`.

Exemplo:

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

sublist (*list*, *p*) Função

Retorna a lista de elementos da *list* da qual o predicado `p` retornar `true`.

Exemplo:

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

sublis_apply_lambda Variável de opção

Valor padrão: `true` - controla se os substitutos de `lambda` são aplicados na simplificação após as `sublis` serem usadas ou se você tem que fazer um `ev` para pegar coisas para aplicar. `true` significa faça a aplicação.

subst (*a*, *b*, *c*) Função

Substitue *a* por *b* em *c*. *b* deve ser um átomo ou uma subexpressão completa de *c*. Por exemplo, `x+y+z` é uma subexpressão completa de `2*(x+y+z)/w` enquanto `x+y` não é. Quando *b* não tem essas características, pode-se algumas vezes usar `substpart` ou `ratsubst` (veja abaixo). Alternativamente, se *b* for da forma de `e/f` então se poderá usar `subst (a*f, e, c)` enquanto se *b* for da forma `e^(1/f)` então se poderá usar `subst (a^f, e, c)`. O comando `subst` também discerne o `x^y` de `x^-y` de modo que `subst (a, sqrt(x), 1/sqrt(x))` retorna `1/a`. *a* e *b* podem também ser operadores de uma expressão contida entre aspas duplas `"` ou eles podem ser nomes de função. Se se desejar substituir por uma variável independente em formas derivadas então a função `at` (veja abaixo) poderá ser usada.

`subst` é um álias para `substitute`.

`subst (`*eq_1*`,` *expr*`)` ou `subst ([`*eq_1*`, ..., `*eq_k*`],` *expr*`)` são outras formas permitidas. As *eq_i* são equações indicando substituições a serem feitas. Para cada equação, o lado direito será substituído pelo lado esquerdo na expressão *expr*.

`exptsubst` se `true` permite que substituições como `y` por `%e^x` em `%e^(a*x)` ocorram.

Quando `opsubst` for `false`, `subst` tentará substituir dentro do operador de uma expressão. E.g. `(opsubst: false, subst (x^2, r, r+r[0]))` trabalhará.

Exemplos:

```
(%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
```

Para exemplos adicionais, faça `example (subst)`.

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

Similar a `substpart`, mas `substinpart` trabalha sobre a representação interna de *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)
```

Se o último argumento para a função `part` for uma lista de índices então muitas subexpressões são escolhidas, cada uma correspondendo a um índice da lista. Dessa forma

```
(%i1) part (x+y+z, [1, 3]);
(%o1)                              z + x
```

`piece` recebe o valor da última expressão selecionada quando usando as funções `part`. É escolhida durante a execução da função e dessa forma pode ser referenciada para a própria função como mostrado abaixo. Se `partswitch` é escolhido para `true` então `end` é retornado quando uma parte selecionada de uma expressão não existir, de outra forma uma mensagem de erro é fornecida.

```
(%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
```

Também, escolhendo a opção `inflag` para `true` e chamando `part` ou `substpart` é o mesmo que chamando `inpart` ou `substinpart`.

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

Substitue *x* para a subexpressão selecionada pelo resto dos argumentos como em `part`. Isso retorna o novo valor de *expr*. *x* pode ser algum operador a ser substituído por um operador de *expr*. Em alguns casos *x* precisa ser contido em aspas duplas `"` (e.g. `substpart ("+", a*b, 0)` retorna `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
```

Também, escolhendo a opção `inflag` para `true` e chamando `part` ou `substpart` é o mesmo que chamando `inpart` ou `substinpart`.

subvarp (*expr*) Função

Retorna `true` se *expr* for uma variável subscrita, por exemplo `a[i]`.

symbolp (*expr*) Função

Retorna `true` se *expr* for um símbolo, de outra forma retorna `false`. com efeito, `symbolp(x)` é equivalente ao predicado `atom(x) and not numberp(x)`.

Veja também `Identifiers`

unorder () Função

Disabilita a ação de alias criada pelo último uso dos comandos de ordenação `ordergreat` e `orderless`. `ordergreat` e `orderless` não podem ser usados mais que uma vez cada sem chamar `unorder`. Veja também `ordergreat` e `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*) Função

Retorna o potencial do vetor de um dado vetor de torção, no sistema de coordenadas corrente. `potentialzeroloc` tem um papel similar ao de `potential`, mas a ordem dos lados esquerdos das equações deve ser uma permutação cíclica das variáveis de coordenadas.

xthru (*expr*) Função

Combina todos os termos de *expr* (o qual pode ser uma adição) sobre um denominador comum sem produtos e somas exponenciadas como `ratsimp` faz. `xthru` cancela fatores comuns no numerador e denominador de expressões racionais mas somente se os fatores são explícitos.

Algumas vezes é melhor usar `xthru` antes de `ratsimp` em uma expressão com o objetivo de fazer com que fatores explicitos do máximo divisor comum entre o numerador e o denominador seja cancelado simplificando dessa forma a expressão a ser aplicado o `ratsimp`.

```
(%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*) Função

Testa se a expressão *expr* na variável *v* é equivalente a zero, retornando `true`, `false`, ou `dontknow` (não sei).

`zeroequiv` Tem essas restrições:

1. Não use funções que o Maxima não sabe como diferenciar e avaliar.
2. Se a expressão tem postes sobre o eixo real, podem existir erros no resultado (mas isso é improvável ocorrer).
3. Se a expressão contem funções que não são soluções para equações diferenciais de primeira ordem (e.g. funções de Bessel) pode ocorrer resultados incorretos.
4. O algoritmo usa avaliação em pontos aleatóriamente escolhidos para subexpressões selecionadas cuidadosamente. Isso é sempre negócio um tanto quanto perigoso, embora o algoritmo tente minimizar o potencial de erro.

Por exemplo `zeroequiv (sin(2*x) - 2*sin(x)*cos(x), x)` retorna `true` e `zeroequiv (%e^x + x, x)` retorna `false`. Por outro lado `zeroequiv (log(a*b) - log(a) - log(b), a)` retorna `dontknow` devido à presença de um parâmetro extra `b`.

6 Expressões

6.1 Introdução a Expressões

Existe um conjunto de palavras reservadas que não pode ser usado como nome de variável. Seu uso pode causar um possível erro crítico de sintaxe.

```
integrate           next          from              diff
in                  at            limit             sum
for                 and           elseif            then
else                do            or                if
unless              product       while             thru
step
```

Muitas coisas em Maxima são expressões. Uma seqüência de expressões pode ser feita dentro de uma expressão maior através da separação dessas através de vírgulas e colocando parêntesis em torno dela. Isso é similar ao **C** *expressão com vírgula*.

```
(%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
```

Mesmo ciclos em Maxima são expressões, embora o valor de retorno desses ciclos não seja muito útil (eles retornam sempre `done`).

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

contanto que o que você realmente queira é provavelmente incluir um terceiro termo na *expressão com vírgula* que atualmente fornece de volta o valor.

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

6.2 Atribuição

Existem dois operadores de atribuição no Maxima, `":"` e `"::"`. E.g., `a: 3` escolhe a variável `a` para 3. `::` atribui o valor da expressão sobre seu lado direito para o valor da quantidade sobre seu lado esquerdo, que deve avaliar para uma variável atômica ou para uma variável subscrita.

6.3 Complexo

Uma expressão complexa é especificada no Maxima através da adição da parte real da expressão a `%i` vezes a parte imaginária. Dessa forma as raízes da equação `x^2 - 4*x + 13 = 0` são `2 + 3*%i` e `2 - 3*%i`. Note que produtos de simplificação de expressões complexas

podem ter efeito através da expansão do produto. Simplificação de quocientes, raízes, e outras funções de expressões complexas podem usualmente serem realizadas através do uso das funções `realpart`, `imagpart`, `rectform`, `polarform`, `abs`, `carg`.

6.4 Substantivos e Verbos

Maxima distingue entre operadores que são "substantivos" e operadores que são "verbos". Um verbo é um operador que pode ser executado. Um substantivo é um operador que aparece como um símbolo em uma expressão, sem ser executado. Por padrão, nomes de função são verbos. Um verbo pode ser mudado em um substantivo através da adição de um apóstrofo no início do nome da função ou aplicando a função `nounify`. Um substantivo pode ser mudado em um verbo através da aplicação da função `verbify`. O sinalizador de avaliação `nouns` faz com que `ev` avalie substantivos em uma expressão.

A forma verbal é distinguida através de um sinal de dólar `$` no início do símbolo Lisp correspondente. De forma oposta, a forma substantiva é distinguida através de um sinal de `%` no início do símbolo Lisp correspondente. Alguns substantivos possuem propriedades especiais de exibição, tais como `'integrate` e `'derivative` (retornado por `diff`), mas muitos não. Por padrão, as formas substantiva e verbal de uma função são idênticas quando mostradas. O sinalizador global `noundisp` faz com que Maxima mostre substantivos com um apóstrofo no início `'`.

Veja também `noun`, `nouns`, `nounify`, e `verbify`.

Exemplos:

```
(%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 Identificadores

Identificadores do Maxima podem compreender caracteres alfabéticos, mais os numerais de 0 a 9, mais qualquer caractere especial precedido por um caractere contra-barra \.

Um numeral pode ser o primeiro caractere de um identificador se esse numeral for precedido por uma contra barra. Numerais que forem o segundo ou o último caractere não precisam ser precedidos por uma contra barra.

Um caractere especial pode ser declarado alfabético através da função `declare`. Se isso ocorrer, esse caractere não precisa ser precedido por uma contra barra em um identificador. Os caracteres alfabéticos vão inicialmente de `A` a `Z`, de `a` a `z`, `%`, e `_`.

Maxima é sensível à caixa . Os identificadores `algumacoisa`, `ALGUMACOISA`, e `Algumacoisa` são distintos. Veja Section 3.2 [Lisp e Maxima], página 9 para mais sobre esse ponto.

Um identificador Maxima é um símbolo Lisp que começa com um sianl de dólar `$`. Qualquer outro símbolo Lisp é precedido por um ponto de interrogação `?` quando aparecer no Maxima. Veja Section 3.2 [Lisp e Maxima], página 9 para maiores detalhes sobre esse ponto.

Exemplos:

```
(%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 Desigualdade

Maxima tem os operadores de desigualdade `<`, `<=`, `>=`, `>`, `#`, e `notequal`. Veja `if` para uma descrição de expressões condicionais.

6.7 Sintaxe

É possível definir novos operadores com precedência especificada, remover a definição de operadores existentes, ou redefinir a precedência de operadores existentes. Um operador pode ser unário prefixado ou unário pósfixado, binario infixado, n-ário infixado, matchfix, ou nofix. "Matchfix" significa um par de símbolos que abraçam seu argumento ou seus argumentos, e "nofix" significa um operador que não precisa de argumentos. Como exemplos dos diferentes tipos de operadores, existe o seguinte.

unário prefixado
: negação `- a`

unário posfixado
: fatorial `a!`

binário infixado
: exponenciação `a^b`

n-ário infixado
: adição `a + b`

matchfix
: construção de lista `[a, b]`

(Não existe operadores internos nofix; para um exemplo de tal operador, veja `nofix`.)

O mecanismo para definir um novo operador é direto. Somente é necessário declarar uma função como um operador; a função operador pode ou não estar definida previamente.

Um exemplo de operadores definidos pelo usuário é o seguinte. Note que a chamada explícita de função `"dd" (a)` é equivalente a `dd a`, da mesma forma `"<-" (a, b)` é equivalente a `a <- b`. Note também que as funções `"dd"` e `"<-"` são indefinidas nesse exemplo.

```
(%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
```

As funções máxima que definem novos operadores estão sumarizadas nessa tabela, equilibrando expoente associado esquerdo (padrão) e o expoente associado direito ("eae" e "ead", respectivamente). (Associação de expoentes determina a precedência do operador. todavia, uma vez que os expoentes esquerdo e diretiro podem ser diferentes, associação de expoentes é até certo ponto mais complicado que precedência.) Alguma das funções de definição de operações tomam argumentos adicionais; veja as descrições de função para maiores detalhes.

`prefixado` ead=180

`posfixado` eae=180

`infixado` eae=180, ead=180

`nário` eae=180, ead=180

`matchfix` (associação de expoentes não é aplicável)

`nofix` (associação de expoentes não é aplicável)

Para comparação, aqui está alguns operadores internos e seus expoentes associados esquerdo e direito.

Operador	eae	ead
:	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` e `kill` removem propriedades de operador de um átomo. `remove ("`*a*`", op)` remove somente as propriedades de operador de *a*. `kill ("`*a*`")` remove todas as propriedades de *a*, incluindo as propriedades de operador. Note que o nome do operador dever estar abraçado por aspas duplas.

```
(%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 Definições para Expressões

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

Avalia a expressão *expr* com as variáveis assumindo os valores como especificado para elas na lista de equações [*eqn_1*, ..., *eqn_n*] ou a equação simples *eqn*.

Se uma subexpressão depender de qualquer das variáveis para a qual um valor foi especificado mas não existe `atvalue` especificado e isso não pode ser de outra forma avaliado, então uma forma substantiva de `at` é retornada que mostra em uma forma bidimensional.

`at` realiza múltiplas substituições em série, não em paralelo.

Veja também `atvalue`. Para outras funções que realizam substituições, veja também `subst` e `ev`.

Exemplos:

```
(%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*) Função
box (*expr*, *a*) Função

Retorna *expr* dentro de uma caixa. O valor de retorno é uma expressão com `box` como o operador e *expr* como o argumento. Uma caixa é desenhada sobre a tela quando `display2d` for `true`.

`box` (*expr*, *a*) Empacota *expr* em uma caixa rotulada pelo símbolo a. O rótulo é truncado se for maior que a largura da caixa.

Uma expressão dentro de uma caixa não avalia para seu conteúdo, então expressões dentro de caixas são efetivamente excluídas de cálculos.

`boxchar` é o caractere usado para desenhar a caixa em `box` e nas funções `dpart` e `lpart`.

Exemplos:

```
(%i1) box (a^2 + b^2);
                 """""""""
                 " 2   2 "
(%o1)            "b  + a "
                 """""""""
(%i2) box (a^2 + b^2, term_1);
                 term_1"""
                 " 2   2 "
```

```
(%o2)         "b  + a "
              """""""""""
(%i3) 1729 - box (1729);
                    """""""
(%o3)      1729 - "1729"
                    """""""
(%i4) boxchar: "-";
(%o4)             -
(%i5) box (sin(x) + cos(y));
                  -----------------
(%o5)             -COS(y) + SIN(x)-
                  -----------------
(%i6)
```

boxchar Variável de opção

Valor padrão: "

`boxchar` é o caractere usado para desenhar a caixa por `box` e nas funções `dpart` e `lpart`.

Todas as caixas em uma expressão são desenhadas com o valor atual de `boxchar`; o caractere de desenho não é armazenado com a expressão de caixa. Isso quer dizer que se você desenhar uma caixa e em seguida mudar o caractere de desenho a caixa anteriormente desenhada será redesenhada com o caractere mudado caso isso seja solicitado.

carg (z) Função

Retorna o argumento complexo de z. O argumento complexo é um ngulo `theta` no intervalo de `(-%pi, %pi]` tal que `r exp (theta %i) =` z onde `r` é o módulo de z.

`carg` é uma função computacional, não uma função de simplificação.

`carg` ignora a declaração `declare (`x`, complex)`, e trata x como uma variável real. Isso é um erro.

Veja também `abs` (módulo de número complexo), `polarform`, `rectform`, `realpart`, e `imagpart`.

Exemplos:

```
(%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)` declara *a* para ser uma constante. Veja `declare`.

constantp (*expr*) Função

Retorna `true` se *expr* for uma expressão constante, de outra forma retorna `false`.

Uma expressão é considerada uma expressão constante se seus argumentos forem números (incluindo números racionais, como mostrado com `/R/`), constantes simbólicas como `%pi`, `%e`, e `%i`, variáveis associadas a uma constante ou constante declarada através de `declare`, ou funções cujos argumentos forem constante.

`constantp` avalia seus argumentos.

Exemplos:

```
(%i1) constantp (7 * sin(2));
(%o1)       TRUE
(%i2) constantp (rat (17/29));
(%o2)       TRUE
(%i3) constantp (%pi * sin(%e));
(%o3)       TRUE
(%i4) constantp (exp (x));
(%o4)       FALSE
(%i5) declare (x, constant);
(%o5)       DONE
(%i6) constantp (exp (x));
(%o6)       TRUE
(%i7) constantp (foo (x) + bar (%e) + baz (2));
(%o7)       FALSE
(%i8)
```

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

Atribui ao átomo *a_i* o sinalizador *f_i*. Os *a_i*'s e *f_i*'s podem também serem listas de átomos e sinalizadores respectivamente nesse caso cada um dos átomos tomam todas as propriedades.

`declare` não avalia seus argumentos. `declare` sempre retorna `done`.

Os possíveis sinalizadores e seus significados são:

`constant` torna *a_i* uma constate como é `%pi`.

`mainvar` torna *a_i* uma `mainvar` (variável principal). A escala de ordenação para átomos: numeros < constantes (e.g. `%e`, `%pi`) < escalares < outras variáveis < variáveis principais.

`scalar` torna *a_i* um escalar.

`nonscalar` faz *a_i* comportar-se como como comporta-se uma lista ou matriz com relação ao operador ponto.

`noun` torna a função *a_i* um substantivo de forma que não possa ser avaliada automaticamente.

`evfun` torna *a_i* conhecido para a função `ev` de forma que `ev` possa vir a ser aplicada se seu nome for mencionado. Veja `evfun`.

`evflag` torna *a_i* conhecido para a função `ev` de forma que `ev` possa vir a ser associado a `true` durante a execução de `ev` se isso for mencionado. Veja `evflag`.

`bindtest` faz com que *a_i* sinalize um erro se isso mesmo for usado em um cálculo não associado.

Maxima atualmente reconhe os seguintes recursos de objetos:

```
even, odd, integer, rational, irrational, real, imaginary,
e complex
```

Que pode ser traduzido para o português como:

```
par, ímpar, inteiro, racional, irracional, real, imaginário,
e complexo
```

Os recursos úteis de funções incluem:

```
increasing,
decreasing, oddfun (odd function), evenfun (even function),
commutative (or symmetric), antisymmetric, lassociative and
rassociative
```

Que pode ser traduzido para o português como:

```
incremento,
decremento, oddfun (função ímpar), evenfun (função par),
comutativa (ou simetrica), antisimetrica, lassociative (associativa à esquerda
rassociative (associativa à direita).
```

Os *a_i* e *f_i* podem também serem listas de objetos ou recursos.

`featurep (`*objeto*`, `*recurso*`)` determina se *objeto* foi declarado para ter *recurso*.

Veja também `features`.

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

é similar a `isolate (`*expr*`, `*x*`)` exceto que isso habilita ao usuário isolar mais que uma variável simultneamente. Isso pode ser útil, por exemplo, se se for tentado mudar variáveis em uma integração múltipla, e em mudança de variável envolvendo duas ou mais das variáveis de integração. Essa função é chamada automaticamente de '`simplification/disol.mac`'. Uma demostração está disponível através de `demo("disol")$`.

dispform (*expr*) Função

Retorna a representação externa de *expr* com relação a seu principal operador. Isso pode ser útil em conjunção com `part` que também lida com a representação externa. Suponha que *expr* seja -A . Então a representação interna de *expr* é `"*"`(-1,A), enquanto que a representaçào externa é `"-"`(A). `dispform (`*expr*`, all)` converte a expressão inteira (não apenas o nível mais alto) para o formato externo. Por exemplo, se `expr: sin (sqrt (x))`, então `freeof (sqrt, expr)` e `freeof (sqrt, dispform (expr))` fornece `true`, enquanto `freeof (sqrt, dispform (expr, all))` fornece `false`.

distrib (*expr*) Função

Distribue adições sobre produtos. `distrib` difere de `expand` no fato de que `distrib` trabalha em somente no nível mais alto de uma expressão, i.e., `distrib` não é recursiva e `distrib` é mais rápida que `expand`. `distrib` difere de `multthru` no que `distrib` expande todas as adições naquele nível.

Exemplos:

```
(%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*) Função

Seleciona a mesma subexpressão que `part`, mas em lugar de apenas retornar aquela subexpressão como seu valor, isso retorna a expressão completa com a subexpressão selecionada mostrada dentro de uma caixa. A caixa é atualmente parte da expressão.

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

exp (*x*) Função

Representa função exponencial. Instncias de `exp (x)` em uma entrada são simplificadas para `%e^x`; `exp` não aparece em expressões simplificadas.

`demoivre` se `true` faz com que `%e^(a + b %i)` simplificar para `%e^(a (cos(b) + %i sin(b)))` se `b` for livre de `%i`. veja `demoivre`.

`%emode`, quando `true`, faz com que `%e^(%pi %i x)` seja simplificado. Veja `%emode`.

`%enumer`, quando `true` faz com que `%e` seja substituído por 2.718... quando `numer` for `true`. Veja `%enumer`.

%emode Variável de opção

Valor padrão: `true`

Quando `%emode` for `true`, `%e^(%pi %i x)` é simplificado como segue.

`%e^(%pi %i x)` simplifica para `cos (%pi x) + %i sin (%pi x)` se `x` for um inteiro ou um multiplo de 1/2, 1/3, 1/4, ou 1/6, e então é adicionalmente simplificado.

Para outro `x` numérico, `%e^(%pi %i x)` simplifica para `%e^(%pi %i y)` onde `y` é `x - 2 k` para algum inteiro `k` tal que `abs(y) < 1`.

Quando `%emode` for `false`, nenhuma simplificação adicional de `%e^(%pi %i x)` é realizada.

%enumer Variável de opção

Valor padrão: `false`

Quando `%enumer` for `true`, `%e` é substituido por seu valor numérico 2.718... mesmo que `numer` seja `true`.

Quando `%enumer` for `false`, essa substituição é realizada somente se o expoente em `%e^x` avaliar para um número.

Veja também `ev` e `numer`.

exptisolate Variável de opção

Valor padrão: `false`

`exptisolate`, quando `true`, faz com que `isolate (expr, var)` examine expoentes de átomos (tais como `%e`) que contenham `var`.

exptsubst Variável de opção

Valor padrão: `false`

`exptsubst`, quando `true`, permite substituições tais como `y` para `%e^x` em `%e^(a x)`.

freeof (*x_1, ..., x_n, expr*) Função

`freeof (`*x_1*`, `*expr*`)` Retorna `true` se nenhuma subexpressão de *expr* for igual a *x_1* ou se *x_1* ocorrer somente uma variável que não tenha associação fora da expressão *expr*, e retorna `false` de outra forma.

`freeof (`*x_1*`, ..., `*x_n*`, `*expr*`)` é equivalente a `freeof (`*x_1*`, `*expr*`) and ... and freeof (`*x_n*`, `*expr*`)`.

Os argumentos *x_1*, ..., *x_n* podem ser nomes de funções e variáveis, nomes subscritos, operadores (empacotados em aspas duplas), ou expressões gerais. `freeof` avalia seus argumentos.

`freeof` opera somente sobre *expr* como isso representa (após simplificação e avaliação) e não tenta determinar se alguma expressão equivalente pode fornecer um resultado diferente. Em particular, simplificação pode retornar uma expressão equivalente mas diferente que compreende alguns diferentes elementos da forma original de *expr*.

Uma variável é uma variável dummy em uma expressão se não tiver associação fora da expressão. Variáveis dummy recoreconhecidas através de `freeof` são o índice de um somatório ou produtório, o limite da variável em `limit`, a variável de integração na forma de integral definida de `integrate`, a variável original em `laplace`, variáveis formais em expressoes `at`, e argumentos em expressões `lambda`. Variáveis locais em `block` não são reconhecidas por `freeof` como variáveis dummy; isso é um bug.

A forma indefinida de `integrate` *not* é livre de suas variáveis de integração.

- Argumentos são nomes de funções, variáveis, nomes subscritos, operadores, e expressões. `freeof (a, b, expr)` é equivalente a `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` avalia seus argumentos.

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

- `freeof` não considera expressões equivalentes. Simplificação pode retornar uma expressão equivalente mas diferente.

```
(%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
```

- Um somatório ou uma integral definida está livre de uma variável dummy. Uma integral indefinida não é livre de suas variáveis de integração.

```
(%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*) Função

Retorna o fatorial generalizado, definido como `x (x-z) (x - 2 z) ... (x - (y - 1) z)`. Dessa forma, para integral *x*, `genfact (x, x, 1) = x!` e `genfact (x, x/2, 2) = x!!`.

imagpart (*expr*) Função

Retorna a parte imaginária da expressão *expr*.

`imagpart` é uma função computacional, não uma função de simplificação.

Veja também `abs`, `carg`, `polarform`, `rectform`, e `realpart`.

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

Declara *op* para ser um operador infixo. Um operador infixo é uma função de dois argumentos, com o nome da função escrito entre os argumentos. Por exemplo, o operador de subtração `-` é um operador infixo.

`infix (`*op*`)` declara *op* para ser um operador infixo com expoentes associados padrão (esquerdo e direito ambos iguais a 180) e podendo ser qualquer entre prefixado, infixado, posfixado, nário, matchfix e nofix (esquerdo e direito ambos iguais a `any`).

`infix (`*op*`, `*lbp*`, `*rbp*`)` declara *op* para ser um operador infixo com expoentes associados esquerdo e diretio equilibrados e podendo ser qualquer entre prefixado, infixado, posfixado, nário, matchfix e nofix (esquerdo e direito ambos iguais a `any`).

`infix (`*op*`, `*lbp*`, `*rbp*`, `*lpos*`, `*rpos*`, `*pos*`)` declara *op* para ser um operdor infixo com expoentes associados padrão e podendo ser um entre prefixado, infixado, posfixado, nário, matchfix e nofix.

A precedência de *op* com relação a outros operadores derivam dos expoentes associados diretiro e esquerdo dos operadores em questão. Se os expoentes associados esquerdo e direito de *op* forem ambos maiores que o expoente associado esquerdo e o direito de algum outro operador, então *op* tem prededência sobre o outro operador. Se os expoentes associados não forem ambos maior ou menor, alguma relação mais complicada ocorre.

A associatividade de *op* depende de seus expoentes associados. Maior expoente associado esquerdo (*eae*) implica uma instncia de *op* é avaliadas antes de outros operadores para sua esquerda em uma expressão, enquanto maior expoente associado direito (*ead*) implica uma instncia de *op* é avaliada antes de outros operadores para sua direita em uma expressão. Dessa forma maior *eae* torna *op* associativo à direita, enquanto maior *ead* torna *op* associativa à esquerda. Se *eae* for igual a *ead*, *op* é associativa à esquerda.

Veja também `Syntax`.

Exemplos:

- Se os expoentes associados esquerdo e direito de *op* forem ambos maiores que os expoentes associados à direita e à esquerda de algum outro operador, então *op* tem precedência sobre o outro operador.

```
(%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)
```

- grande *eae* torna *op* associativa à direita, enquanto grande *ead* torna *op* associativa à esquerda.

```
(%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 Variável de opção

Velor padrão: `false`

Quando `inflag` for `true`, funções para extração de partes inspecionam a forma interna de `expr`.

Note que o simplificador re-organiza expressões. Dessa forma `first (x + y)` retorna `x` se `inflag` for `true` e `y` se `inflag` for `false`. (`first (y + x)` fornece os mesmos resultados.)

Também, escolhendo `inflag` para `true` e chamando `part` ou `substpart` é o mesmo que chamar `inpart` ou `substinpart`.

As funções afetadas pela posição do sinalizador `inflag` são: `part`, `substpart`, `first`, `rest`, `last`, `length`, a estrutura `for ... in`, `map`, `fullmap`, `maplist`, `reveal` e `pickapart`.

inpart (*expr, n_1, ..., n_k*) Função

É similar a `part` mas trabalha sobre a representação interna da expressão em lugar da forma de exibição e dessa forma pode ser mais rápida uma vez que nenhuma formatação é realizada. Cuidado deve ser tomado com relação à ordem de subexpressões em adições e produtos (uma vez que a ordem das variáveis na forma interna é muitas vezes diferente daquela na forma mostrada) e no manuseio com menos unário, subtração, e divisão (uma vez que esses operadores são removidos da expressão). `part (x+y, 0)` ou `inpart (x+y, 0)` retorna `+`, embora com o objetivo de referirse ao operador isso deva ser abraçado por aspas duplas. Por exemplo `... if inpart (%o9,0) = "+" then ...`.

Exemplos:

```
(%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*) Função

Retorna *expr* com subexpressões que são adições e que não possuem *x* substituido por rótulos de expressão intermediária (esses sendo símbolos atômicos como `%t1`, `%t2`, ...). Isso é muitas vezes útil para evitar expansões desnecessárias de subexpressões que não possuam a variável de interesse. Uma vez que os rótulos intermediários são associados às subexpressões eles podem todos ser substituídos de volta por avaliação da expressão em que ocorrerem.

`exptisolate` (valor padrão: `false`) se `true` fará com que `isolate` examine expoentes de átomos (como `%e`) que contenham *x*.

`isolate_wrt_times` se `true`, então `isolate` irá também isolar produtos wrt. Veja `isolate_wrt_times`.

Faça `example (isolate)` para exemplos.

isolate_wrt_times Variável de opção

Valor padrão: `false`

Quando `isolate_wrt_times` for `true`, `isolate` irá também isolar produtos wrt. E.g. compare ambas as escolhas do comutador em

```
(%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 Variável de opção

Valor padrão: `false`

Quando `listconstvars` for `true`, isso fará com que `listofvars` inclua `%e`, `%pi`, `%i`, e quaisquer variáveis declaradas contantes na lista seja retornado se aparecer na expressão que chamar `listofvars`. O comportamento padrão é omitir isso.

listdummyvars Variável de opção

Valor padrão: `true`

Quando `listdummyvars` for `false`, "variáveis dummy" na expressão não serão incluídasna lista retornada por `listofvars`. (O significado de "variável dummy" é o mesmo que em `freeof`. "Variáveis dummy" são coisas matemáticas como o índice de um somatório ou produtório, a variável limite, e a variável da integral definida.) Exemplo:

```
(%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*) Função

Retorna uma lista de variáveis em *expr*.

`listconstvars` se `true` faz com que `listofvars` inclua `%e`, `%pi`, `%i`, e quaisquer variáveis declaradas constantes na lista é retornada se aparecer em *expr*. O comportamento padrão é omitir isso.

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

lfreeof (*lista, expr*) Função

Para cada um dos membros *m* de lista, chama `freeof (m, expr)`. Retorna `false` se qualquer chamada a `freeof` for feita e `true` de outra forma.

lopow (*expr, x*) Função

Retorna o menor expoente de *x* que explicitamente aparecer em *expr*. Dessa forma

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

lpart (*rótulo, expr, n_1, ..., n_k*) Função

é similar a `dpart` mas usa uma caixa rotulada. Uma moldura rotulada é similar à que é produzida por `dpart` mas a produzida por `lpart` tem o nome na linha do topo.

multthru (*expr*) Função
multthru (*expr_1, expr_2*) Função

Multiplica um fator (que pode ser uma adição) de *expr* pelos outros fatores de *expr*. Isto é, *expr* é *f_1 f_2 ... f_n* onde ao menos um fator, digamos *f_i*, é uma soma de

termos. Cada termo naquela soma é multiplicado por outros fatores no produto. (A saber todos os fatores exceto *f_i*). `multthru` não expande somas exponenciais. Essa função é o caminho mais rápido para distribuir produtos (comutativos ou não) sobre adições. Uma vez que quocientes são representados como produtos `multthru` podem ser usados para dividir adições por produtos também.

`multthru` (*expr_1*, *expr_2*) multiplica cada termo em *expr_2* (que pode ser uma adição ou uma equação) por *expr_1*. Se *expr_1* não for por si mesmo uma adição então essa forma é equivalente a `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*) Função

Retorna a forma substantiva do nome da função *f*. Isso é necessário se se quer referir ao nome de uma função verbo como se esse nome fosse um substantivo. Note que algumas funções verbos irão retornar sua forma substantiva senão puderem ser avaliadas para certos argumentos. A forma substantiva é também a forma retornada se uma chamada de função é precedida por um apóstrofo.

nterms (*expr*) Função

Retorna o número de termos que *expr* pode ter se for completamente expandida e nenhum cancelamento ou combinação de termos acontecer. Note expressões como `sin` (*expr*), `sqrt` (*expr*), `exp` (*expr*), etc. contam como apenas um termo independentemente de quantos termos *expr* tenha (se *expr* for uma adição).

op (*expr*) Função

Retorna o operador principal da expressão *expr*. `op (expr)` é equivalente a `part (expr, 0)`.

`op` retorna uma seqüência de caracteres se o operador principal for uma operador interno ou definido pelo usuário como prefixado, binário ou n-ário infixo, posfixado, matchfix ou nofix. De outra forma `op` retorna um símbolo.

`op` observa o valor do sinalizador global `inflag`.

`op` avalia seus argumentos.

Veja também `args`.

Exemplos:

```
(%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*) Função
operatorp (*expr, [op_1, ..., op_n]*) Função

`operatorp (expr, op)` retorna `true` se *op* for igual ao operador de *expr*.

`operatorp (expr, [op_1, ..., op_n])` retorna `true` se algum elementos de *op_1*, ..., *op_n* for igual ao operador de *expr*.

optimize (*expr*) Função

Retorna uma expressão que produz o mesmo valor e efeito que *expr* mas faz de forma mais eficientemente por evitar a recomputação de subexpressões comuns. `optimize` também tem o mesmo efeito de "colapsar" seus argumentos de forma que todas as subexpressões comuns são compartilhadas. Faça `example (optimize)` para exemplos.

optimprefix Variável de opção

Valor padrão: `%`

`optimprefix` é o prefixo usado para símbolos gerados pelo comando `optimize`.

ordergreat (*v_1, ..., v_n*) Função

Escolhe aliases para as variáveis *v_1*, ..., *v_n* tais que *v_1* > *v_2* > ... > *v_n*, e *v_n* > qualquer outra variável não mencionada como um argumento.

Veja também `orderless`.

ordergreatp (*expr_1, expr_2*) Função

Retorna `true` se *expr_2* precede *expr_1* na ordenação escolhida com a função `ordergreat`.

orderless (*v_1, ..., v_n*) Função

Escolhe aliases para as variáveis *v_1*, ..., *v_n* tais que *v_1* < *v_2* < ... < *v_n*, and *v_n* < qualquer outra variável não mencionada como um argumento.

Dessa forma a escala de ordenação completa é: constantes numéricas < constantes declaradas < escalares declarados < primeiro argumento para `orderless` < ... < último argumento para `orderless` < variáveis que começam com A < ... < variáveis que começam com Z < último argumento para `ordergreat` < ... < primeiro argumento para `ordergreat` < `mainvar`s - variáveis principais declaradas.

Veja também `ordergreat` e `mainvar`.

orderlessp (*expr_1, expr_2*) Função

Retorna `true` se *expr_1* precede *expr_2* na ordenação escolhida pelo comando `orderless`.

part (*expr, n_1, ..., n_k*) Função

Retorna partes da forma exibida de `expr`. Essa função obtém a parte de `expr` como especificado pelos índices *n_1*, ..., *n_k*. A primeira parte *n_1* de `expr` é obtida, então a parte *n_2* daquela é obtida, etc. O resultado é parte *n_k* de ... parte *n_2* da parte *n_1* da `expr`.

`part` pode ser usada para obter um elemento de uma lista, uma linha de uma matriz, etc.

Se o último argumento para uma função `part` for uma lista de índices então muitas subexpressões serão pinçadas, cada uma correspondendo a um índice da lista. Dessa forma `part (x + y + z, [1, 3])` é `z+x`.

`piece` mantém a última expressão selecionada quando usando as funções `part`. Isso é escolhido durante a execução da função e dessa forma pode referir-se à função em si mesma como mostrado abaixo.

Se `partswitch` for escolhido para `true` então `end` é retornado quando uma parte selecionada de uma expressão não existir, de outra forma uma mensagem de erro é forncecida.

Exemplo: `part (z+2*y, 2, 1)` retorna 2.

`example (part)` mostra exemplos adicionais.

partition (*expr, x*) Função

Retorna uma lista de duas expressões. Elas são (1) os fatores de *expr* (se essa expressão for um produto), os termos de *expr* (se isso for uma adição), ou a lista (se isso for uma lsita) que não contiver `var` e, (2) os fatores, termos, ou lista que faz.

```
(%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 Variável de opção

Valor padrão: `false`

Quando `partswitch` for `true`, `end` é retornado quando uma parte selecionada de uma expressão não existir, de outra forma uma mensagem de erro é fornecida.

pickapart (*expr*, *n*) Função

Atribui rótulos de expressão intermediária a subexpressões de *expr* de comprimento *n*, um inteiro. A subexpressões maiores ou menores não são atribuidos rótulos. `pickapart` retorna uma expressão em termos de expressões intermediárias equivalentes à expressão original *expr*.

Veja também `part`, `dpart`, `lpart`, `inpart`, e `reveal`.

Exemplos:

```
(%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

Mantém a ultima expressão selecionada quando usando funções `part`. Isso é escolhido durante a execução da função e dessa forma pode referir-se à função em si mesma.

polarform (*expr*) Função

Retorna uma expressão `r %e^(%i theta)` equivalente a *expr*, tal que `r` e `theta` sejam puramente reais.

powers (*expr*, *x*) Função

Fornece os expoentes de *x* que ocorrem em expressão *expr*.

`load (powers)` chama essa função.

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

Retorna o produto dos velores de `expr` com o índice *i* variando de *i_0* a *i_1*. A avaliação é similar à que ocorre em `sum`.

Se *i_1* for um menor que *i_0*, o produto é um "produto vazio" e `product` retorna 1 em lugar de reporta um erro. Veja também `prodhack`.

Maxima não simplifica produtos.

Exemplo:

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

realpart (*expr*) Função

Retorna a parte real de *expr*. `realpart` e `imagpart` irão trabalhar sobre expressões envolvendo funções trigonométricas e hiperbólicas, bem como raízes quadradas, logarítmos, e exponenciação.

rectform (*expr*) Função

Retorna uma expressão `a + b %i` equivalente a *expr*, tal que *a* e *b* sejam puramente reais.

rembox (*expr*, *unlabelled*) Função
rembox (*expr*, *rótulo*) Função
rembox (*expr*) Função

Remove caixas de *expr*.

`rembox (`*expr*`, unlabelled)` remove todas as caixas sem rótulos de *expr*.

`rembox (`*expr*`, `*rótulo*`)` remove somente caixas contendo *rótulo*.

`rembox (`*expr*`)` remove todas as caixas, rotuladas e nã rotuladas.

Caixas são desenhadas pelas funções `box`, `dpart`, e `lpart`.

Exemplos:

```
(%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*) Função

Representa um somatório dos valores de *expr* com o índice *i* variando de *i_0* a *i_1*. Somatórios podem ser diferenciados, adicionados, subtraídos, ou multiplicados com alguma simplificação automática sendo executada. A forma substantiva `'sum` é mostrada em notação de sigma.

Se os limites superiores e inferiores diferirem de um número inteiro, o somatoriando *expr* é avaliado para cada valor do índice do somatório *i*, e os resultados são adicionados juntos.

De outra forma, se a variável `simpsum` for `true` o somatório é simplificado. Simplificações podem algumas vezes retornar uma forma fechada. Se o sinalizador de avaliação `simpsum` for `false` ou a simplificação falhar, o resultado é uma forma substantiva `'sum`.

`sum` avalia *i_0* e *i_1* e coloca um apóstrofo em *i*. O somatoriando *expr* recebe um apóstrofo sob algumas circunstncias, ou é avaliado para maior ou menor grau em outras.

Se *i_1* é um menor que *i_0*, a adição é considerada um "somatório vazio" e `sum` retorna 0 em lugar de reportar um erro. Veja também `sumhack`.

Quando o sinalizador de avaliação `cauchysum` for `true`, o produto de somatórios é expresso como um produto de Cauchy, no qual o índice do somatório mais interno é uma função de índice de um nível acima, em lugar de variar independentemente.

A variável global `genindex` é o prefixo alfabético usado para gerar o próximo índice do somatório, quando um índice automaticamente gerado for necessário.

`gensumnum` é o sufixo numérico usando para gerar o próximo índice do somatório, quando um índice gerado automaticamente for necessário. Quando `gensumnum` for `false`, um índice gerado automaticamente é somente `genindex` sem sufixo numérico.

Veja também `sumcontract`, `intosum`, `bashindices`, `niceindices`, `nouns`, e `evflag`.

Exemplos:

```
(%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*) Função

Representas a adição de *expr* a cada elemento *x* em *L*.

Uma forma substantiva `'lsum` é retornada se o argumento *L* não avaliar para uma lista.

Exemplos:

```
(%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*) Função

Retorna a forma verbal da função chamada *f*.

Veja também `verb`, `noun`, e `nounify`.

Exemplos:

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

7 Simplificação

7.1 Definições para Simplificação

askexp Variável de sistema

Quando `asksign` é chamada, `askexp` é a expressão que `asksign` está testando.

Antigamente, isso era possível para um usuário inspecionar `askexp` entrando em uma parada do Maxima com control-A.

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

`askinteger (expr, integer)` tenta determinar a partir da base de dados do `assume` se *expr* é um inteiro. `askinteger` pergunta ao usuário pela linha de comando se isso não pode ser dito de outra forma, e tenta instalar a informação na base de dados do `assume` se for possível. `askinteger (expr)` é equivalente a `askinteger (expr, integer)`.

`askinteger (expr, even)` e `askinteger (expr, odd)` da mesma forma tentam determinar se *expr* é um inteiro par ou inteiro ímpar, respectivamente.

asksign (*expr*) Função

Primeiro tenta determinar se a expressão especificada é positiva, negativa, ou zero. Se isso não for possível, `asksign` pergunta ao usuário com as questões necessárias para completar a sua dedução. As respostas do usuário são guardadas na base de dados pelo tempo que durar a computação corrente. O valor de retorno de `asksign` é um entre `pos`, `neg`, ou `zero`.

demoivre (*expr*) Função
demoivre Variável de opção

A função `demoivre (expr)` converte uma expressão sem escolher a variável global `demoivre`.

Quando a variável `demoivre` for `true`, exponenciais complexas são convertidas em expressões equivalentes em termos de funções circulares: `exp (a + b*%i)` simplifica para `%e^a * (cos(b) + %i*sin(b))` se `b` for livre de `%i`. `a` e `b` não são expandidos.

O valor padrão de `demoivre` é `false`.

`exponentialize` converte funções circulares e hiperbólicas para a forma exponencial. `demoivre` e `exponentialize` não podem ambas serem `true` ao mesmo tempo.

domain Variável de opção

Valor padrão: `real`

Quando `domain` é escolhida para `complex`, `sqrt (x^2)` permanecerá `sqrt (x^2)` em lugar de retornar `abs(x)`.

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

Expande a expressão *expr*. Produtos de somas e somas exponenciadas são multiplicadas para fora, numeradores de expressões racionais que são adições são quebradas em suas respectivas parcelas, e multiplicação (comutativa e não comutativa) é distribuída sobre a adição em todos os níveis de *expr*.

Para polinômios se pode ter usulmente `ratexpand` que usa um algorítmo mais eficiente.

`maxnegex` e `maxposex` controlam o máximo expoente negativo e o máximo expoente positivo, respectivamente, que irão expandir.

`expand (`*expr*`, `*p*`, `*n*`)` expande *expr*, usando *p* para `maxposex` e *n* para `maxnegex`. Isso é útil com o objetivo de expandir partes mas não tudo de uma expressão.

`expon` - o expoente da maior potência negativa que é automaticamente expandida (independente de chamadas a `expand`). Por Exemplo se `expon` for 4 então `(x+1)^(-5)` não será automaticamente expandido.

`expop` - o maior expoente positivo que é automaticamente expandido. Dessa forma `(x+1)^3`, quando digitado, será automaticamente expandido somente se `expop` for maior que ou igual a 3. Se for desejado ter `(x+1)^n` expandido onde `n` é maior que `expop` então executando `expand ((x+1)^n)` trabalhará somente se `maxposex` não for menor que `n`.

O sinalizador `expand` usado com `ev` causa expansão.

O arquivo '`simplification/facexp.mac`' contém muitas funções relacionadas (em particular `facsum`, `factorfacsum` e `collectterms`, que são chamadas automaticamente) e variáveis (`nextlayerfactor` e `facsum_combine`) que fornecem ao usuário com a habilidade para estruturar expressões por expansão controlada. Descrições breves de função estão disponível em '`simplification/facexp.usg`'. Um arquivo demonstrativo está disponível fazendo `demo("facexp")`.

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

Expande expressão `expr` com relação às variáveis *x_1*, ..., *x_n*. Todos os produtos envolvendo as variáveis aparecem explicitamente. A forma retornada será livre de produtos de somas de expressões que não estão livres das variáveis. *x_1*, ..., *x_n* podem ser variáveis, operadores, ou expressões.

Por padrão, denominadores não são expandidos, mas isso pode ser controlado através do comutador `expandwrt_denom`.

Essa função, `expandwrt`, não é automaticamente chamada a partir de '`simplification/stopex.mac`'.

expandwrt_denom Variável de opção

Valor padrão: `false`

`expandwrt_denom` controla o tratamento de expressões racionais por `expandwrt`. Se `true`, então ambos o numerador e o denominador da expressão serão expandidos conforme os argumentos de `expandwrt`, mas se `expandwrt_denom` for `false`, então somente o numerador será expandido por aquele caminho.

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

é similar a `expandwrt`, mas trata expressões que são produtos um tanto quanto diferentemente. `expandwrt_factored` expande somente sobre esses fatores de `expr` que contiverem as variáveis *x_1*, ..., *x_n*.

Essa função é automaticamente chamada a aprtir de '`simplification/stopex.mac`'.

expon Variável de opção

Valor padrão: 0

`expon` é o expoente da maior potência negativa que é automaticamente expandido (independente de chamadas a `expand`). Por exemplo, se `expon` for 4 então `(x+1)^(-5)` não será automaticamente expandido.

exponentialize (*expr*) Função
exponentialize Variável de opção

A função `exponentialize (expr)` converte funções circulares e hiperbólicas em *expr* para exponenciais, sem escolher a variável global `exponentialize`.

Quando a variável `exponentialize` for `true`, todas as funções circulares e hiperbólicas são convertidas para a forma exponencial. O valor padrão é `false`.

`demoivre` converte exponenciais complexas em funções circulares. `exponentialize` e `demoivre` não podem ambas serem `true` ao mesmo tempo.

expop Variável de opção

Valor padrão: 0

`expop` - o maior expoente positivo que é automaticamente expandido. Dessa forma `(x+1)^3`, quando digitado, será automaticamente expandido somente se `expop` for maior que ou igual a 3. Se for desejado ter `(x+1)^n` expandido onde `n` é maior que `expop` então executando `expand ((x+1)^n)` trabalhará somente se `maxposex` não for menor que `n`.

factlim Variável de opção

Valor padrão: -1

`factlim` especifica o maior fatorial que é automaticamente expandido. Se for -1 então todos os inteiros são expandidos.

intosum (*expr*) Função

Move fatores multiplicativos fora de um somatório para dentro. Se o índice for usado na expressão de fora, então a função tentará achar um índice razoável, o mesmo que é feito para `sumcontract`. Isso é essencialmente a idéia reversa da propriedade `outative` de somatórios, mas note que isso não remove essa propriedade, somente pula sua verificação.

Em alguns casos, um `scanmap (multthru,` *expr*`)` pode ser necessário antes de `intosum`.

lassociative Declaration

`declare (g, lassociative)` diz ao simplificador do Maxima que `g` é associativa à esquerda. E.g., `g (g (a, b), g (c, d))` irá simplificar para `g (g (g (a, b), c), d)`.

linear Declaration

Uma das propriedades operativas do Maxima. Para funções de uma única variável `f` então declarada, a "expansão" `f(x + y)` retorna `f(x) + f(y)`, `f(a*x)` retorna `a*f(x)` ocorrem onde `a` é uma "constante". Para funções de dois ou mais argumentos, "linearidade" é definida para ser como no caso de `sum` ou `integrate`, i.e., `f (a*x + b, x)` retorna `a*f(x,x) + b*f(1,x)` para `a` e `b` livres de `x`.

`linear` é equivalente a `additive` e `outative`. Veja também `opproperties`.

mainvar Declaration

Você pode declarar variáveis para serem `mainvar` (variável principal). A escala de ordenação para átomos é essencialmente: números `<` constantes (e.g., `%e`, `%pi`) `<` escalares `<` outras variáveis `<` mainvars. E.g., compare `expand ((X+Y)^4)` com `(declare (x, mainvar), expand ((x+y)^4))`. (Nota: Cuidado deve ser tomado se você eleger o uso desse recurso acima. E.g., se você subtrair uma expressão na qual `x` for uma `mainvar` de uma na qual `x` não seja uma `mainvar`, resimplificação e.g. com `ev (expr, simp)` pode ser necessária se for para ocorrer um cancelamento. Também, se você grava uma expressão na qual `x` é uma `mainvar`, você provavelmente pode também gravar `x`.)

maxapplydepth Variável de opção

Valor padrão: 10000

`maxapplydepth` é a máxima definição para a qual `apply1` e `apply2` irão pesquisar.

maxapplyheight Variável de opção

Valor padrão: 10000

`maxapplyheight` é a elevação máxima a qual `applyb1` irá alcançar antes de abandonar.

maxnegex Variável de opção

Valor padrão: 1000

`maxnegex` é o maior expoente negativo que será expandido pelo comando `expand` (veja também `maxposex`).

maxposex Variável de opção

Valor padrão: 1000

`maxposex` é o maior expoente que irá ser expandido com o comando `expand` (veja também `maxnegex`).

multiplicative Declaration

`declare (f, multiplicative)` diz ao simplificador do Maxima que `f` é multiplicativa.

1. Se `f` for uma função de uma única variável, sempre que o simplificador encontrar `f` aplicada a um produto, `f` distribue sobre aquele procuto. E.g., `f(x*y)` simplifica para `f(x)*f(y)`.

2. Se `f` é uma função de 2 ou mais argumentos, multiplicatividade é definida como multiplicatividade no primeiro argumento para `f`, e.g., `f (g(x) * h(x), x)` simplifica para `f (g(x) ,x) * f (h(x), x)`.

Essa simplificação não ocorre quando `f` é aplicada a expressões da forma `product (x[i], i, m, n)`.

negdistrib Variável de opção

Valor padrão: `true`

Quando `negdistrib` for `true`, -1 distribue sobre uma expressão. E.g., `-(x + y)` transforma-se em `- y - x`. Setting it to `false` Permitirá `- (x + y)` seja mostrado como foi escrito. Isso algumas vezes é útil mas seja muito cuidadoso: como o sinalizador `simp`, isso um sinalizador que você não que escolher para `false` como algo natural ou necessário com excessão de usar localmente no seu Maxima.

negsumdispflag Variável de opção

Valor padrão: `true`

Quando `negsumdispflag` for `true`, `x - y` é mostrado como `x - y` em lugar de como `- y + x`. Escolhendo isso para `false` faz com que a verificação especial em visualização para a diferença das duas expressões não seja concluída. Uma aplicação é que dessa forma `a + %i*b` e `a - %i*b` podem ambos serem mostrados pelo mesmo caminho.

noeval Símbolo especial

`noeval` suprime a fase de avaliação de `ev`. Isso é útil em conjunção com outros comutadores e em fazer com que expressões sejam resimplificadas sem serem reavaliadas.

noun Declaration

`noun` é uma das opções do comando `declare`. Isso faz um função então declarada como `"noun"` (substantivo), significando que ela não deve ser avaliada automaticamente.

noundisp Variável de opção

Valor padrão: `false`

Quando `noundisp` for `true`, substantivos (nouns) são mostrados com um apóstrofo. Esse comutador é sempre `true` quando mostrando definições de função.

nouns Símbolo especial

`nouns` é um `evflag` (sinalizador de avaliação). Quando usado como uma opção para o comando `ev`, `nouns` converte todas as formas substantivas (`"noun"`) que ocorrem na expressão que está sendo avaliada para verbos (`"verbs"`), i.e., avalia essas expressões. Veja também `noun`, `nounify`, `verb`, e `verbify`.

numer Símbolo especial

`numer` faz com que algumas funções matemáticas (incluindo exponenciação) com argumentos numéricos sejam avaliados em ponto flutuante. Isso faz com que variáveis em `expr` que tenham sido dados valores numéricos a elas (`numerval`) sejam substituídas pelos seus valores. Isso também escolhe o sinalizador `float` para `on`.

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

Declara as variáveis `x_1`, ..., *x_n* para terem valores numéricos iguais a `expr_1`, ..., `expr_n`. O valor numérico é avaliado e substituido para a variável em quaisquer expressões na qual a variável ocorra se o sinalizador `numer` for `true`. Veja também `ev`.

As expressões `expr_1`, ..., `expr_n` podem ser quaisquer expressões, não necessariamente numéricas.

oppropertie Variável de sistema

`oppropertie` é a lista de propriedades de operadores especiais reconhecidas pelo simplificador do Maxima: `linear`, `additive`, `multiplicative`, `outative`, `evenfun`, `oddfun`, `commutative`, `symmetric`, `antisymmetric`, `nary`, `lassociative`, `rassociative`.

opsubst Variável de opção

Valor padrão: `true`

Quando `opsubst` for `false`, `subst` não tenta substituir dentro de um operador de uma expressão. E.g., `(opsubst: false, subst (x^2, r, r+r[0]))` irá trabalhar.

outative Declaração

`declare (f, outative)` diz ao simplificador do Maxima que fatores constantes no argumento de `f` podem ser puxados para fora.

1. Se `f` for uma função de uma única variável, sempre que o simplificador encontrar `f` aplicada a um produto, aquele produto será particionado em fatores que são constantes e fatores que não são e os fatores constantes serão puxados para fora. E.g., `f(a*x)` simplificará para `a*f(x)` onde `a` é uma constante. Fatores de constantes não atômicas não serão puxados para fora.
2. Se `f` for uma função de 2 ou mais argumentos, a colocação para fora é definida como no caso de `sum` ou `integrate`, i.e., `f (a*g(x), x)` irá simplificar para `a * f(g(x), x)` sendo `a` livre de `x`.

`sum`, `integrate`, e `limit` são todas `outative`.

posfun Declaração

`declare (f, posfun)` declara `f` para ser uma função positiva. `is (f(x) > 0)` retorna `true`.

radcan (*expr*) Função

Simplifica *expr*, que pode conter logarítmos, exponenciais, e radicais, convertendo essa expressão em uma forma que é canônica sobre uma larga classe de expressões e uma dada ordenação de variáveis; isto é, todas formas funcionalmente equivalentes são mapeadas em uma única forma. Para uma classe um tanto quanto larga de expressões, `radcan` produz uma forma regular. Duas expressões equivalentes nessa classe não possuem necessáriamente a mesma aparência, mas suas diferenças podem ser simplificadas por `radcan` para zero.

Para algumas expressões `radcan` é que consome inteiramente o tempo. Esse é o custo de explorar certos relacionamentos entre os componentes da expressão para simplificações baseadas sobre fatoração e expansões de fração-parcial de expoentes.

Quando `%e_to_numlog` for `true`, `%e^(r*log(expr))` simplifica para `expr^r` se `r` for um número racional.

Quando `radexpand` for `false`, certas transformações são inibidas. `radcan (sqrt (1-x))` permanece `sqrt (1-x)` e não é simplificada para `%i sqrt (x-1)`. `radcan (sqrt (x^2 - 2*x + 11))` permanece `sqrt (x^2 - 2*x + 1)` e não é simplificada para `x - 1`.

`example (radcan)` mostra alguns exemplos.

radexpand Variável de opção

Valor padrão: `true`

`radexpand` controla algumas simplificações de radicais.

Quando `radexpand` for `all`, faz com que nésimas raízes de fatores de um produto que são potências de n sejam puxados para fora do radical. E.g. Se `radexpand` for `all`, `sqrt (16*x^2)` simplifica para `4*x`.

Mais particularmente, considere `sqrt (x^2)`.

- Se `radexpand` for `all` or `assume (x > 0)` tiver sido executado, `sqrt(x^2)` simplifica para `x`.
- Se `radexpand` for `true` e `domain` for `real` (isso é o padrão), `sqrt(x^2)` simplifica para `abs(x)`.
- Se `radexpand` for `false`, ou `radexpand` for `true` e `domain` for `complex`, `sqrt(x^2)` não é simplificado.

Note que `domain` somente interessa quando `radexpand` for `true`.

radsubstflag Variável de opção

Valor padrão: `false`

`radsubstflag`, se `true`, permite a `ratsubst` fazer substituições tais como `u` por `sqrt (x)` em `x`.

rassociative Declaração

`declare (g, rassociative)` diz ao simplificador do Maxima que `g` é associativa à direita. E.g., `g(g(a, b), g(c, d))` simplifica para `g(a, g(b, g(c, d)))`.

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

Simplificação Seqüêncial Comparativa (método devido a Stoute). `scsimp` tenta simplificar *expr* conforme as regras *rule_1*, ..., *rule_n*. Se uma expressão pequena for obtida, o processo repete-se. De outra forma após todas as simplificações serem tentadas, isso retorna a resposta original.

`example (scsimp)` mostra alguns exemplos.

simpsum Variável de opção

Valor padrão: `false`

Quando `simpsum` for `true`, o resultado de uma `sum` é simplificado. Essa simplificação pode algumas vezes estar apta a produzir uma forma fechada. Se `simpsum` for `false` ou se a forma com apóstrofo `'sum` for usada, o valor é uma forma substantiva aditiva que é uma representação da notação sigma usada em matemática.

sumcontract (*expr*) Função

Combina todas as parcelas de uma adição que tem maiores e menores associações que diferem por constantes. O resultado é uma expressão contendo um somatório para cada escolha de cada tais somatórios adicionados a todos os termos extras apropriados que tiveram de ser extraídos para a forma dessa adição. `sumcontract` combina todas as somas compatíveis e usa-se os indices de uma as somas se puder, e então tenta formar um índice razoável se não for usar qualquer dos fornecidos.

Isso pode ser necessário fazer um `intosum (`*expr*`)` antes de `sumcontract`.

sumexpand Variável de opção

Valor padrão: `false`

Quando `sumexpand` for `true`, produtos de somas e somas exponeciadas simplificam para somas aninhadas.

Veja também `cauchysum`.

Exemplos:

```
(%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
```

sumsplitfact Variável de opção

Valor padrão: `true`

When `sumsplitfact` for `false`, `minfactorial` é aplicado após um `factcomb`.

symmetric Declaração

`declare (h, symmetric)` diz ao simplificador do Maxima que `h` é uma função simétrica. E.g., `h (x, z, y)` simplifica para `h (x, y, z)`.

`commutative` é sinônimo de `symmetric`.

unknown (*expr*) Função

Retorna `true` se e somente se *expr* contém um operador ou função não reconhecida pelo simplificador do Maxima.

8 Montando Gráficos

8.1 Definições para Montagem de Gráficos

in_netmath Variável

Valor padrão: `false`

Quando `in_netmath` é `true`, `plot3d` imprime uma saída OpenMath para o console se `plot_format` é `openmath`; caso contrário `in_netmath` (mesmo se `true`) não tem efeito.

`in_netmath` não tem efeito sobre `plot2d`.

openplot_curves (*list*, *rest_options*) Função

Pega uma lista de curvas tais como

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

ou

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

e monta seus gráficos. Isso é similar a xgraph_curves, mas não usa as rotinas open plot. Argumentos adicionais de símbolos podem ser dados tais como `"{xrange -3 4}"`. O exemplo adiante monta o gráfico de duas curvas, usando pontos grandes, rotulando-se o primeiro `jim` e o segundo rotulando-se `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]]);
```

Algumas outras palavras chave especiais são `xfun`, `color`, `plotpoints`, `linecolors`, `pointsize`, `nolines`, `bargraph`, `labelposition`, `xaxislabel`, e `yaxislabel`.

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

Mostra a montagem de uma ou mais expressões como uma função de uma variável.

Em todos os casos, *expr* é uma expressão a ser montado o gráfico no eixo vertical como uma função de uma variável. *x_range*, a amplitude do eixo horizontal, é uma lista da forma `[`*variável*`,` *min*`,` *max*`]`, onde *variável* é uma variável que aparece em *expr*. *y_range*, e a amplitude do eixo vertical, é uma lista da forma `[y,` *min*`,` *max*`]`.

`plot2d (`*expr*`,` *x_range*`)` monta o gráfico *expr* como uma função da variável nomeada em *x_range*, sobre a amplitude especificada em *x_range*. Se a amplitude vertical não for alternativamente especificada por `set_plot_option`, essa é escolhida automaticamente. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

`plot2d (`*expr*`,` *x_range*`,` *y_range*`)` monta o gráfico de *expr* como uma função de uma variável nomeada em *x_range*, sobre a amplitude especificada em *x_range*. O alcance vertical é escolhido para *y_range*. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

`plot2d ([`*expr_1*`, ..., `*expr_n*`],` *x_range*`)` monta o gráfico *expr_1*, ..., *expr_n* como uma função da variável nomeada em *x_range*, sobre a amplitude especificada em *x_range*. Se a amplitude vertical não for alternativamente especificada por `set_plot_option`, essa é escolhida automaticamente. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

`plot2d ([`*expr_1*`, ..., `*expr_n*`],` *x_range*`,` *y_range*`)` monta o gráfico *expr_1*, ..., *expr_n* como uma função de uma variável nomedada em *x_range*, sobre a amplitude especificada em *x_range*. O alcance vertical é escolhido para *y_range*. Todas as opções são assumidas terem valores padrão a menos que sejam alternativamente especificadas por `set_plot_option`.

Exemplos:

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

Em qualquer lugar onde pode existir uma expressão comum, pode existir uma expressão paramétrica: *parametric_expr* é uma lista da forma `[parametric,` *x_expr*`,` *y_expr*`,` *t_range*`,` *options*`]`. Aqui *x_expr* e *y_expr* são expressões de 1 variável *var* que é o primeiro elemento da amplitude *trange*. A montagem do gráfico mostra o caminho descrito pelo par `[`*x_expr*`,` *y_expr*`]` como *var* varia em *trange*.

No exemplo seguinte, montaremos o gráfico de um círculo, então faremos a montagem do gráfico com somente poucos pontos usados, desse modo vamos pegar uma estrela, e inicialmente montaremos o gráfico juntamente com uma função comum de X.

Exemplos:

- Montar o gráfico de um círculo de forma paramétrica.

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

- Monta o gráfico de uma estrela: liga oito pontos sobre a circunferência de um círculo.

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

- Monta o gráfico de um polinômio cúbico da forma comum e de um círculo da forma paramétrica.

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

Expressões discretas podem também serem usadas ou em lugar de expressões comuns ou em lugar de expressões paramétricas: *discrete_expr* é uma lista da forma `[discrete,` *x_list*`,` *y_list*`]` ou da forma `[discrete,` *xy_list*`]`, onde *xy_list* é uma lista de pares `[`*x*`,`*y*`]`.

Exemplos:

- Cria algumas listas.

```
(%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)$
```

- Monta um gráfico com segmentos de reta.

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

- Monta um gráfico com segmentos de reta, usando uma lista de pares.

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

- Monta um gráfico com pontos.

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

- Monta o gráfico da curva `cos(`x`)` usando linhas e (xx,yy) usando pontos.

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

Veja também `plot_options`, que descreve opções de montagem de gráfico e tem mais exemplos.

xgraph_curves (*list*) Função

transforma em gráfico a lista de 'grupos de pontos' dados em lista usando xgraph.

Uma lista de grupos de pontos pode ser da forma

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

ou

```
[[x0, y0], [x1, y1], ...]
```

Um grupo de pontos pode também conter símbolos que fornecem rótulos ou outra informação.

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

transforma em gráfico os três grupos de pontos com três curvas.

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

fizemos com que os grupos de pontos [e os próprios subseqüêntes], não possuam linhas entre os pontos, e para usar pixels largos. Veja a página de manual sobre o xgraph para especificar mais opções.

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

fizemos aí aparecer um "rótulo" de "x^2+y" para esse grupo de pontos em particular. As aspas, ", no início é que dizem ao xgraph isso é um rótulo.

```
pt_set: append ([concat ("TitleText: Dados da Amostra")], [x0, ...])$
```

fizemos o título principal do gráfico ser "Dados da Amostra" ao invés de "Maxima Plot".

Para fazer um gráfico em barras com largura de 0.2 unidades, e para montar o gráfico com duas possibilidades diferentes dos tais gráficos em barras:

```
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))]);
```

Um arquivo temporário '`xgraph-out`' é usado.

plot_options Variável de sistema

Elementos dessa lista estabelecem as opções padrão para a montagem do gráfico. Se uma opção está presente em uma chamada a `plot2d` ou `plot3d`, esse valor tem precedência sobre a opção padrão. De outra forma, o valor em `plot_options` é usado. Opções padrão são atribuídas por `set_plot_option`.

Cada elemento de `plot_options` é uma lista de dois ou mais ítens. O primeiro ítem é o nome de uma opção, e os restantes compreendem o valor ou valores atribuídos à opção. Em alguns casos, o valor atribuído é uma lista, que pode compreender muitos ítens.

As opções de montagem de gráfico que são reconhecidas por `plot2d` e `plot3d` são as seguintes:

- Opção: `plot_format` determina qual pacote de montagem de gráfico é usado por `plot2d` e `plot3d`.
  - Valor padrão: `gnuplot` Gnuplot é o padrão, e mais avançado, pacote de montagem de gráfico. Esse requer uma instalação externa do gnuplot.
  - Valor: `mgnuplot` Mgnuplot é um invólucro em torno do gnuplot baseado no Tk. Isso está incluído na distribuição do Maxima. Mgnuplot oferece uma GUI rudimentar para o gnuplot, mas tem menos recursos em geral que a interface texto. Mgnuplot requer uma instalação externa do gnuplot e Tcl/Tk.
  - Valor: `openmath` Openmath é um programa-GUI para montagem de gráfico baseado no Tcl/Tk. Isso está incluído na distribuição do Maxima.
  - Valor: `ps` Gera arquivos Postscript simples diretamente do Maxima. Saídas Postscript mais sofisticadas podem ser geradas pelo gnuplot, deixando a opção `plot_format` não especificada (para aceitar o padrão), e posicionando a opção `gnuplot_term` para `ps`.
- Opção: `run_viewer` controla se o visualizador apropriado para o formato da montagem do gráfico pode ou não poderá ser executado.
  - Valor padrão: `true` Executa o programa visualizador.
  - Valor: `false` Não executa o programa visualizador.
- `gnuplot_term` Prepara a saída tipo terminal para gnuplot.
  - Valor padrão: `default` A saída do Gnuplot é mostrada em uma janela gráfica separada.
  - Valor: `dumb` A saída do Gnuplot é mostrada no console do Maxima como uma aproximação "arte ASCII" para gráficos.
  - Valor: `ps` Gnuplot gera comandos na linguagem PostScript de descrição de páginas. Se a opção `gnuplot_out_file` está escolhida para *filename*, gnuplot escreve os comandos PostScript para *filename*. De outra forma, os comandos são mostrados no console Maxima.
- Opção: `gnuplot_out_file` Escreve a saída gnuplot para um arquivo.
  - Valor padrão: `false` Nenhum arquivo de saída especificado.
  - Valor: *filename* Exemplo: `[gnuplot_out_file, "myplot.ps"]` Esse exemplo envia uma saída PostScript para o arquivo `myplot.ps` quando usada em conjunto com o terminal PostScript do gnuplot.

- Opção: `x` A amplitude horizontal padrão.

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

  Especifica a amplitude horizontal para [-3, 3].
- Opção: `y` A amplitude vertical padrão.

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

  Especifica a amplitude vertical para [-3, 3].
- Opção: `t` A amplitude padrão para o parâmetro em montagem de gráficos paramétricos.

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

  Especifica a amplitude da variável paramétrica para [0, 10].
- Opção: `nticks` Número de pontos iniciais usado pela rotina adaptativa de montagem do gráfico.

  ```
  [nticks, 20]
  ```

  O padrão para `nticks` é 10.
- Opção: `adapt_depth` O número maximo de quebras usada pela rotina adaptativa de montagem do gráfico.

  ```
  [adapt_depth, 5]
  ```

  O padrão para `adapt_depth` é 10.
- Opção: `grid` Escolhe o número de pontos da grade para usar nas direções x e y para montagem de gráficos tridimensionais.

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

  Escolhe a grade para 50 por 50 pontos. A grade padrão é 30 por 30.
- Opção: `transform_xy` Permite que transformações sejam aplicadas à montagem de gráficos tridimensionais.

  ```
  [transform_xy, false]
  ```

  O padrão `transform_xy` é `false`. Se isso não é `false`, pode ser a saída de

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

  A transformação `polar_xy` é previamente definida no Maxima. Isso fornece a mesma transformação que

  ```
  make_transform ([r, th, z], r*cos(th), r*sin(th), z)$
  ```
- Opção: `colour_z` é específica para o formato `ps` de montagem de gráfico.

  ```
  [colour_z, true]
  ```

  O valor padrão para `colour_z` é `false`.
- Opção: `view_direction` Específico para o formato `ps` de montagem de gráfico.

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

  O padrão `view_direction` é [1, 1, 1].

Existem muitas opções de montagem de gráficos específicas para gnuplot. Todas essas opções (exceto `gnuplot_pm3d`) são comandos gnuplot em estado natural, especificados como seqüências de caracteres. Consulte a documentação do gnuplot para maiores detalhes.

- Opção: `gnuplot_pm3d` Controla o uso do modo PM3D, que possui recursos avançados em 3D. PM3D está somente disponível no gnuplot em versões após a 3.7. O valor padrão para `gnuplot_pm3d` é `false`.

  Exemplo:

  ```
  [gnuplot_pm3d, true]
  ```

- Opção: `gnuplot_preamble` Insere comandos antes que o gráfico seja desenhado. Quaisquer comandos válidos para o gnuplot podem ser usados. Multiplos comandos podem ser separados com um ponto e vírgula. O exemplo mostrado produz uma escala numérica na montagem do gráfico. O valor padrão para `gnuplot_preamble` é uma seqüência de caracteres vazia `""`.

  Exemplo:

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

- Opção: `gnuplot_curve_titles` Controla os títulos dados na chave da montagem do gráfico. O valor padrão é `[default]`, que automaticamente escolhe o título de cada curva para a função cujo gráfico está sendo construído. Se não contiver `[default]`, `gnuplot_curve_titles` pode conter uma lista de seqüências de caracteres, cada uma das quais é `"title '`*title_string*`'"`. (Para disabilitar a chave de impressão de gráfico, adicione `"set nokey"` a `gnuplot_preamble`.)

  Exemplo:

  ```
  [gnuplot_curve_titles, ["title 'Minha primeira função'", "title 'Minha segu
  ```

- Opção: `gnuplot_curve_styles` Uma lista de seqüências de caracteres controlando a aparência das curvas, i.e., cor, largura, brilho, etc., para serem enviadas para o comando de montagem do gráfico do gnuplot. O valor padrão é `["with lines 3", "with lines 1", "with lines 2", "with lines 5", "with lines 4", "with lines 6", "with lines 7"]`, que circula através de diferentes cores. Veja a documentação do gnuplot de `plot` para maiores informações.

  Exemplo:

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

- Opção: `gnuplot_default_term_command` O comando gnuplot para escolher o tipo de terminal para o terminal padrão. O valor padrão é a seqüência de caracteres vazia `""`, i.e., usa os padrões do gnuplot.

  Exemplo:

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

- Opção: `gnuplot_dumb_term_command` O comando gnuplot para escolher o tipo de terminal para o terminal dumb. O valor padrão é `"set term dumb 79 22"`, que faz a saída texto com 79 caracteres por 22 caracteres.

  Exemplo:

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

- Opção: `gnuplot_ps_term_command` O comando gnuplot para escolher o tipo de terminal para o terminal PostScript. O valor padrão é `"set size 1.5, 1.5;set term postscript eps enhanced color solid 24"`, que escolhe o tamanho para 1.5 vezes o padrão do gnuplot, e o tamanho da fonte para 24, além de outras coisas. Veja a documentação do gnuplot de `set term postscript` para mais informação.

Exemplo:

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

Exemplos:

- Grava um gráfico de `sin(x)` para o arquivo `sin.eps`.

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

- Usa a opção do y para arrancar singularidades e a opção gnuplot_preamble para colocar a chave na parte inferior do gráfico em lugar de no topo.

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

- Usa um muito complicado `gnuplot_preamble` para produzir elegantes rótulos para o eixo x. (Note que a seqüência de caracteres `gnuplot_preamble` deve ser fornecida inteiramente sem qualquer quebra de linha.)

```
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]);
```

- Usa uma muito complicada `gnuplot_preamble` para produzir elegantes rótulos para o eixo x, e produzir saídas PostScript que pegam vantagens do formato de texto avançado disponível no gnuplot. (Note que a seqüência de caracteres `gnuplot_preamble` deve ser fornecida inteiramente sem qualquer quebra de linha.)

```
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
```

- Uma montagem de gráfico tridimensional usando o terminal gnuplot pm3d.

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

- Uma montagem de gráfico tridimensional sem a malha e com contornos projetados no plano inferior.

```
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])$
```

- Uma montagem de gráfico onde o eixo z é representado apenas por cores. (Note que a seqüência de caracteres `gnuplot_preamble` deve ser fornecida inteiramente sem qualquer quebra de linha.)

```
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, ...*) Função
plot3d ([*expr_1, expr_2, expr_3*], *x_range, y_range, ..., options, ...*) Função

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

monta o gráfico `z = 2^(-u^2+v^2)` com `u` e `v` variando em [-5,5] e [-7,7] respectivamente, e com *u* sobre o eixo x, e `v` sobre o eixo y.

Um segundo exemplo de modelo de argumento é

```
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]);
```

que monta o gráfico da banda de Moebius, parametrizada por três expressões fornecidas como o primeiro argumento para `plot3d`. Um adicional e opcional argumento `['grid, 50, 15]` fornece o número de retâgulos da grade na direção x e na direção y.

Esse exemplo mostra uma montagem de gráfico da parte real de `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]);
```

Aqui a opção `view_direction` indica a direção da qual nós pegamos a projeção. Nós atualmente fazemos isso de infinitamente distante, mas paralelo à linha de `view_direction` para a origem. Isso é correntemente usado somente em plot_format `ps`, uma vez que outros visualizadores permitem rotação interativa do objeto.

Outro exemplo é uma superfície de Klein:

```
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, 4
```

ou um toro

```
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]
```

Podemos também enviar saídas para o gnuplot:

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

Algumas vezes você precisa definir uma função para mortar o gráfico de uma expressão. Todos os argumentos para plot3d são avaliados antes de serem passados para plot3d, e então tentando fazer um expressão que faz apenas o que você que pode ser difícil, e é apenas mais fácil fazer uma função.

```
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])$
```

Veja `plot_options` para mais exemplos.

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

Retornam uma função adequada para a função transformação em plot3d. Use com a opção de montagem de gráfico `transform_xy`.

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

é uma transformação para coordenadas polares.

plot2d_ps (*expr*, *range*) Função

Escreve para pstream uma seqüência de comandos PostScript que montam o graáfico de *expr* sobre *range*.

expr é uma expressão. *range* é uma lista da forma `[`*x*`,` *min*`,` *max*`]` na qual *x* é uma variável que aparece em *expr*.

Veja também `closeps`.

closeps () Função

Essa poderá usualmente ser chamada no final de um seqüência comandos de montagem de gráfico. Isso fecha o fluxo corrente de saída *pstream*, e altera esse para nil. Isso também pode ser chamado ao iniciar uma montagem de gráfico, para garantir que pstream será fechado se estiver aberto. Todos os comandos que escrevem para pstream, abrem isso se necessário. `closeps` é separada de outros comandos de montagem de gráfico, posteriormente podemos querer montar um gráfico com 2 amplitudes ou sobrepor muitas montagens de gráficos, e então devemos manter esse fluxo aberto.

set_plot_option (*opção*) Função

Atribui uma das varáveis globais para impressão. *option* é especificada como uma lista de dois ou mais elementos, na qual o primeiro elemeto é uma das palavras chave dentro da lista `plot_options`.

`set_plot_option` avalia seu argumento. `set_plot_option` retorna `plot_options` (após modificar um desses elementos).

Veja também `plot_options`, `plot2d`, e `plot3d`.

Exemplos:

Modifica a malha (`grid`) e valores de `x`. Quando uma palavra chave em `plot_options` tem um valor atribuído, colocar um apóstrofo evita avaliação.

```
(%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*) Função

Desenha uma curva conectando os pontos em *ptlist*. O último pode ser da forma `[x0, y0, x1, y1, ...]` ou da forma `[[x0, y0], [x1, y1], ...]`

A função `join` é útil fazendo uma lista dos x's e uma lista dos y's e colocando-os juntos.

psdraw_curve simplesmente invoca a mais primitiva função *pscurve*. Aqui está a definiçã:

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

pscom (*cmd*) Função

cmd é inserida no arquivo PostScript. Exemplo:

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

9 Entrada e Saída

9.1 Introdução a Entrada e Saída

9.2 Arquivos

Um arquivo é simplesmente uma área sobre um dispositivo particular de armazenagem que contém dados ou texto. Arquivos em disco são figurativamente agrupados dentro de "diretórios". Um diretório é apenas uma lista de arquivos. Comandos que lidam com arquivos são: `save`, `load`, `loadfile`, `stringout`, `batch`, `demo`, `writefile`, `closefile`, and `appendfile`.

9.3 Definições para Entrada e Saída de Dados

_ Variável de sistema

`_` é a mais recente expressão de entrada (e.g., `%i1`, `%i2`, `%i3`, ...).

A `_` é atribuída a entrada antes dela ser simplificada ou avaliada. Todavia, o valor de `_` é simplificado (mas não avaliado) quando for mostrado.

`_` é reconhecido por `batch`, mas não por `load`.

Veja também `%`.

Exemplos:

```
(%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
```

% Variável de sistema

`%` é a expressão de saída (e.g., `%o1`, `%o2`, `%o3`, ...) mais recentemente calculada pelo Maxima, pode ou não ser mostrada. `%` é reconhecido por `batch`, mas não por `load`.

Veja também `_`, `%%`, e `%th`

%% Variável de sistema

Em declaração composta, a saber `block`, `lambda`, ou (*s_1*, ..., *s_n*), `%%` é os valor da declaração anterior. Por exemplo,

```
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));
```

retornam o mesmo resultado, a saber `21/2`.

Uma declaração composta pode compreender outras declarações compostas. Pode uma declaração ser simples ou composta, `%%` é o valor da declaração anterior. Por exemplo,

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

retorna `7*a^n`.

Dentro da declaração composta, o valor de `%%` pode ser inspecionado em uma parada de linha de comando, que é aberta pela execução da função `break`. Por exemplo, na parada de linha de comando aberta por

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

digitando `%%;` retorna `42`.

Na primeira declaração em uma declaração composta, ou fora de uma declaração composta, `%%` é indefinido.

`%%` reconhecido por ambos `batch` e `load`.

Veja também `%`.

%edispflag Variável de opção

Valor padrão: `false`

Quando `%edispflag` é `true`, Maxima mostra `%e` para um expoente negativo como um quociente. Por exemplo, `%e^-x` é mostrado como `1/%e^x`.

%th (*i*) Função

O valor da *i*'ésima expressão prévia de saída. Isto é, se a próxima expressão a ser calculada for a *n*'ésima saída, `%th (m)` será a (*n* - *m*)'ésima saída.

`%th` é útil em arquivos `batch` ou para referir-se a um grupo de expressões de saída. Por exemplo,

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

escolhe `s` para a soma das últimas dez expressões de saída.

`%th` é reconhecido por `batch`, mas não por `load`.

Veja também `%`.

"?" Símbolo especial

Como prefixo para uma função ou nome de variável, `?` significa que o nome é um nome Lisp, não um nome Maxima. Por exemplo, `?round` significa a função Lisp `ROUND`. Veja Section 3.2 [Lisp e Maxima], página 9 para mais sobre esse ponto.

A notação `? word` (um ponto de interrogação seguido de uma palavra e separado desta por um espaço em branco) é equivalente a `describe ("word")`.

absboxchar Variável de opção

Valor padrão: `!`

`absboxchar` é o caracter usado para para desenhar o sinal de valor absoluto em torno de expressões que são maiores que uma linha de altura.

file_output_append Variável de opção

Valor padrão: `false`

`file_output_append` governa se funções de saída de arquivo anexam ao final ou truncam seu arquivo de saída. Quando `file_output_append` for `true`, tais funções anexam ao final de seu arquivo de saída. De outra forma, o arquivo de saída é truncado.

`save`, `stringout`, e `with_stdout` respeitam `file_output_append`. Outras funções que escrevem arquivos de saída não respeitam `file_output_append`. Em partivular, montagem de gráficos e traduções de funções sempre truncam seu arquivo de saída, e `tex` e `appendfile` sempre anexam ao final.

appendfile (*filename*) Função

Adiciona ao final de *filename* uma transcrição do console. `appendfile` é o mesmo que `writefile`, exceto que o arquivo transcrito, se já existe, terá sempre alguma coisa adicionada ao seu final.

`closefile` fecha o arquivo transcrito que foi aberto anteriormente por `appendfile` ou por `writefile`.

batch (*filename*) Função

Lê expressões Maxima do arquivo *filename* e as avalia. `batch` procura pelo arquivo *filename* na lista `file_search_maxima`. Veja `file_search`.

filename compreende uma seqüência de expressões Maxima, cada uma terminada com `;` ou `$`. A varável especial `%` e a função `%th` referem-se a resultados prévios dentro do arquivo. O arquivo pode incluir construções `:lisp`. Espaços, tabulações, e o caracter de nova linha no arquivo serão ignorados. um arquivo de entrada conveniente pode ser criado por um editor de texto ou pela função `stringout`.

`batch` lê cada expressão de entrada de *filename*, mostra a entrada para o console, calcula a correspondente expressão de saída, e mostra a expressão de saída. Rótulos de entrada são atribuídos para expressões de entrada e rótulos de saída são atribuídos para expressões de saída. `batch` avalia toda expressão de entrada no arquivo a menos que exista um erro. Se uma entrada de usuário for requisitada (by `asksign` ou `askinteger`, por exemplo) `batch` interrompe para coletar a entrada requisitada e então continua.

Isso possibilita interromper `batch` pela digitação de `control-C` no console. O efeito de `control-C` depende da subjacente implementação do Lisp.

`batch` tem muitos usos, tais como fornecer um reservatório para trabalhar linhas de comando, para fornecer demonstrações livres de erros, ou para ajudar a organizar alguma coisa na solução de problemas complexos.

`batch` avalia seu argumento. `batch` não possui valor de retorno.

Veja também `load`, `batchload`, e `demo`.

batchload (*filename*) Função

Lê expressões Maxima de *filename* e as avalia, sem mostrar a entrada ou expressões de saída e sem atribuir rótulos para expressões de saída. Saídas impressas (tais como produzidas por `print` ou `describe`) são mostradas, todavia.

A variável especial `%` e a função `%th` referem-se a resultados anteriores do interpretador interativo, não a resultados dentro do arquivo. O arquivo não pode incluir construções `:lisp`.

`batchload` retorna o caminho de *filename*, como uma seqüência de caracteres. `batchload` avalia seu argumento.

Veja também `batch` e `load`.

closefile () Função

Fecha o arquivo transcrito aberto por `writefile` ou `appendfile`.

collapse (*expr*) Função

Reduz *expr* fazendo com que todas as suas subexpressões comuns (i.e., iguais) serem compartilhadas (i.e., usam a mesma células), dessa forma exonomizando espaço. (`collapse` é uma subrotina usada pelo comando `optimize`.) Dessa forma, chamar `collapse` pode ser útil após um `save` arquivo. Você pode diminuir muitas expressões juntas pelo uso de `collapse ([`*expr_1*`, ..., `*expr_n*`])`. Similarmente, você pode diminuir os elementos de um array `A` fazendo `collapse (listarray ('A))`.

concat (*arg_1, arg_2, ...*) Função

Concatena seus argumentos. Os argumentos devem obrigatóriamente serem avaliados para atomos. O valor de retorno é um símbolo se o primeiro argumento for um símbolo e uma seqüência de caracteres no formato do Maxima em caso contrário.

`concat` avalia seus argumentos. O apóstrofo `'` evita avaliação.

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

Um símbolo construído por `concat` pode ser atribuído a um valor e aparecer em expressões. O operador de atribuição `::` (duplo dois pontos) avalia seu lado esquerdo.

```
(%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 que embora `concat (1, 2)` seja visto como um números, isso é uma seqüência de caracteres no formato do Maxima.

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

sconcat (*arg_1, arg_2, ...*) Função

Concatena seus argumentos em uma seqüência de caracteres. Ao contrário de `concat`, os argumentos arrumados *não* precisam ser atômicos.

O resultado é uma seqüência de caracteres no format do Lisp.

```
(%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, ...*) Função

é como `display` mas somente os valores dos argumentos são mostrados em lugar de equações. Isso é útil para argumentos complicados que não possuem nomes ou onde somente o valor do argumento é de interesse e não o nome.

dispcon (*tensor_1, tensor_2, ...*) Função

dispcon (*all*) Função

Mostram as propriedades de contração de seus argumentos como foram dados para `defcon`. `dispcon (all)` mostra todas as propriedades de contração que foram definidas.

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

Mostra equações cujo lado esquerdo é *expr_i* não avaliado, e cujo lado direito é o valor da expressão centrada na linha. Essa função é útil em blocos e em `for` declarações com o objetivo de ter resultados intermediários mostrados. The Os argumentos para `display` são usualmente átomos, variáveis subscritas, ou chamadas de função. Veja também `disp`.

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

display2d Variável de opção

Valor padrão: `true`

Quando `display2d` é `false`, O console visualizador é unidimensional ao invés de bidimensional.

display_format_internal Variável de opção

Valor padrão: `false`

Quando `display_format_internal` é `true`, expressões são mostradas sem ser por caminhos que escondam a representação matemática interna. O visualizador então corresponde ao que `inpart` retorna em lugar de `part`.

Exemplos:

```
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*) Função

Mostra *expr* em partes uma abaixo da outra. Isto é, primeiro o operador de *expr* é mostrado, então cada parcela em uma adição, ou fatores em um produto, ou parte de uma expressão mais geral é mostrado separadamente. Isso é útil se *expr* é muito larga para ser mostrada de outra forma. Por exemplo se `P1`, `P2`, ... são expressões muito largas então o programa visualizador pode sair fora do espaço de armazenamento na tentativa de mostrar `P1 + P2 + ...` tudo de uma vez. Todavia, `dispterms (P1 + P2 + ...)` mostra `P1`, então abaixo disso `P2`, etc. Quando não usando `dispterms`, se uma expressão exponencial é muito alta para ser mostrada como `A^B` isso aparece como `expt (A, B)` (ou como `ncexpt (A, B)` no caso de `A^^B`).

error_size Variável de opção

Valor padrão: 10

`error_size` modifica mensagens de erro conforme o tamanho das expressões que aparecem nelas. Se o tamanho de uma expressão (como determinado pela função Lisp `ERROR-SIZE`) é maior que `error_size`, a expressão é substituída na mensagem por um símbolo, e o o símbolo é atribuído à expressão. Os símbolos são obtidos da lista `error_syms`.

De outra forma, a expressão é menor que `error_size`, e a expressão é mostrada na mensagem.

Veja também `error` e `error_syms`.

Exemplo:

O tamanho de `U`, como determinado por `ERROR-SIZE`, é 24.

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

(%i2) error_size: 20$

(%i3) error ("Expressão exemplo é", U);

Expressão exemplo é 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 ("Expressão exemplo é", U);

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

error_syms Variável de opção

Valor padrão: `[errexp1, errexp2, errexp3]`

Em mensagens de erro, expressões mais largas que `error_size` são substituídas por símbolos, e os símbolos são escolhidos para as expressões. Os símbolos são obtidos da lista `error_syms`. A primeira expressão muito larga é substituída por `error_syms[1]`, a segunda por `error_syms[2]`, e assim por diante.

Se houverem mais expressões muito largas que há elementos em `error_syms`, símbolos são construídos automaticamente, com o n-ésimo símbolo equivalente a `concat ('errexp,` n`)`.

Veja também `error` e `error_size`.

expt (a, b) Função

Se uma expressão exponencial é muito alta para ser mostrada cmo `a^b` isso aparece como `expt (`a`, `b`)` (ou como `ncexpt (`a`, `b`)` no caso de `a^^b`).

`expt` e `ncexpt` não são reconhecidas em entradas.

exptdispflag Variável de opção

Valor padrão: `true`

Quando `exptdispflag` é `true`, Maxima mostra expressões com expoente negativo usando quocientes, e.g., `X^(-1)` como `1/X`.

filename_merge (*path, filename*) Função

Constroem um caminho modificado de *path* e *filename*. Se o componente final de *path* é da forma `###.`*algumacoisa*, o componente é substituído com *filename*`.`*algumacoisa*. De outra forma, o componente final é simplesmente substituído por *filename*.

file_search (*filename*) Função
file_search (*filename, pathlist*) Função

`file_search` procura pelo arquivo *filename* e retorna o caminho para o arquivo (como uma seqüência de caracteres) se ele for achado; de outra forma `file_search` retorna `false`. `file_search` (*filename*) procura nos diretórios padrões de busca, que são especificados pelas variáveis `file_search_maxima`, `file_search_lisp`, e `file_search_demo`.

`file_search` primeiro verifica se o nome atual passado existe, antes de tentar coincidir esse nome atual com o modelo "coringa" de busca do arquivo. Veja `file_search_maxima` concernente a modelos de busca de arquivos.

O argumento *filename* pode ser um caminho e nome de arquivo, ou apenas um nome de arquivo, ou, se um diretório de busca de arquivo inclui um modelo de busca de arquivo, apenas a base do nome de arquivo (sem uma extensão). Por exemplo,

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

todos acham o mesmo arquivo, assumindo que o arquivo exista e `/home/wfs/special/###.mac` está em `file_search_maxima`.

`file_search` (*filename*`,` *pathlist*) procura somente nesses diretórios especificados por *pathlist*, que é uma lista de seqüências de caracteres. O argumento *pathlist* substitui os diretórios de busca padrão, então se a lista do caminho é dada, `file_search` procura somente nesses especificados, e não qualquer dos diretórios padrão de busca. Mesmo se existe somente um diretório em *pathlist*, esse deve ainda ser dado como uma lista de um único elemento.

O usuário pode modificar o diretório de busca padrão. Veja `file_search_maxima`.

`file_search` é invocado por `load` com `file_search_maxima` e `file_search_lisp` como diretórios de busca.

file_search_maxima Variável de opção
file_search_lisp Variável de opção
file_search_demo Variável de opção

Essas variáveis especificam listas de diretórios a serem procurados por `load`, `demo`, e algumas outras funções do Maxima. O valor padrão dessas variáveis nomeia vários diretórios na instalaçã padrão do Maxima.

O usuáro pode modificar essas variáveis, quer substituindo os valores padrão ou colocando no final diretórios adicionais. Por exemplo,

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

substitui o valor padrão de `file_search_maxima`, enquanto

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

adiciona no final da lista dois diretórios adicionais. Isso pode ser conveniente para colocar assim uma expressão no arquivo `maxima-init.mac` de forma que o caminho de busca de arquivo é atribuído automaticamente quando o Maxima inicia.

Multiplas extensões de arquivo e e multiplos caminhos podem ser especificados por construções "coringa" especiais. A seqüência de caracteres `###` expande a busca para além do nome básico, enquanto uma lista separada por vírgulas e entre chaves `{foo,bar,baz}` expande em multiplas seqüências de caracteres. Por exemplo, supondo que o nome básico a ser procurado seja `neumann`,

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

expande em `/home/wfs/neumann.lisp`, `/home/gcj/neumann.lisp`, `/home/wfs/neumann.mac`, e `/home/gcj/neumann.mac`.

file_type (*filename*) Função

Retorna uma suposta informação sobre o conteúdo de *filename*, baseada na extensão do arquivo. *filename* não precisa referir-se a um arquivo atual; nenhuma tentativa é feita para abrir o arquivo e inspecionar seu conteúdo.

O valor de retorno é um símbolo, qualquer um entre `object`, `lisp`, ou `maxima`. Se a extensão começa com `m` ou `d`, `file_type` retorna `maxima`. Se a extensão começa om `l`, `file_type` retorna `lisp`. Se nenhum dos acima, `file_type` retorna `object`.

grind (*expr*) Função

grind Variável de opção

A função `grind` imprime *expr* para o console em uma forma adequada de entrada para Maxima. `grind` sempre retorna `done`.

Veja também `string`, que retorna uma seqüência de caracteres em lugar de imprimir sua saída. `grind` tenta imprimir a expressão de uma maneira que a faz levemente mais fácil para ler que a saída de `string`.

Quando a variável `grind` é `true`, a saída de `string` e `stringout` tem o mesmo formato que `grind`; de outra forma nenhuma tentativa é feita para formatar especialmente a saída dessas funções. O valor padrão da variável `grind` é `false`.

`grind` pode também ser especificado como um argumento de `playback`. Quando `grind` está presente, `playback` imprime expressões de entrada no mesmo formato que a função `grind`. De outra forma, nenhuma tentativa é feita para formatar especialmente as expressões de entrada.

ibase Variável de opção

Valor padrão: 10

Inteiros fornecidos dentro do Maxima são interpretados com respeito à base `ibase`.

A `ibase` pode ser atribuído qualquer inteiro entre 2 e 35 (decimal), inclusive. Quando `ibase` é maior que 10, os numerais compreendem aos numerais decimais de 0 até 9 mais as letras maiúsculas do alfabeto A, B, C, ..., como necessário. Os numerais para a base 35, a maior base aceitável, compreendem de 0 até 9 e de A até Y.

Veja também `obase`.

inchar Variável de opção

Valor padrão: `%i`

`inchar` é o prefixo dos rótulos de expressões fornecidas pelo usuário. Maxima automaticamente constrói um rótulo para cada expressão de entrada por concatenação de `inchar` e `linenum`. A `inchar` pode ser atribuído qualquer seqüência de caracteres ou símbolo, não necessariamente um caracter simples.

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

Veja também `labels`.

ldisp (*expr_1, ..., expr_n*) Função

Mostra expressões *expr_1*, ..., *expr_n* para o console como saída impressa na tela. `ldisp` atribue um rótulo de expressão intermediária a cada argumento e retorna a lista de rótulos.

Veja também `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*) Função

Mostra expressões *expr_1*, ..., *expr_n* para o console como saída impressa na tela. Cada expressão é impressa como uma equação da forma `lhs = rhs` na qual `lhs` é um dos argumentos de `ldisplay` e `rhs` é seu valor. Tipicamente cada argumento é uma variável. `ldisp` atribui um rótulo de expressão intermediáia a cada equação e retorna a lista de rótulos.

Veja também `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 Variável de opção

Valor padrão: `%t`

`linechar` é o refixo de rótulos de expressões intermediárias gerados pelo Maxima. Maxima constrói um rótulo para cada expressão intermediária (se for mostrada) pela concatenação de `linechar` e `linenum`. A `linechar` pode ser atribuído qualquer seqüência de caracteres ou símbolo, não necessáriamente um caractere simples.

Expressões intermediárias podem ou não serem mostradas. See `programmode` e `labels`.

linel Variável de opção

Valor padrão: 79

`linel` é a largura assumida (em caracteres) do console para o propósito de mostrar expressões. A `linel` pode ser atribuído qualquer valor pelo usuário, embora valores muio pequenos ou muito grandes possam ser impraticáveis. Textos impressos por funções internas do Maxima, tais como mensagens de erro e a saída de `describe`, não são afetadas por `linel`.

lispdisp Option variable

Valor padrão: `false`

Quando `lispdisp` for `true`, símbolos Lisp são mostrados com um ponto de interrogação `?` na frente. De outra forma, símbolos Lisp serão mostrados sem o ponto de interrogaçõ na frente.

Exemplos:

```
(%i1) lispdisp: false$
(%i2) ?foo + ?bar;
```

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

load (*filename*) Função

Avalia expressões em *filename*, dessa forma conduzindo variáveis, funções, e outros objetos dentro do Maxima. A associação de qualquer objeto existente é substituída pela associação recuperada de *filename*. Para achar o arquivo, `load` chama `file_search` com `file_search_maxima` e `file_search_lisp` como diretórios de busca. Se `load` obtém sucesso, isso retorna o nome do arquivo. De outra forma `load` imprime uma mensagem e erro.

`load` trabalha igualmente bem para códigos Lisp e códigos Maxima. Arquivos criados por `save`, `translate_file`, e `compile_file`, que criam códigos Lisp, e `stringout`, que criam códigos Maxima, podem ser processadas por `load`. `load` chama `loadfile` para carregar arquivos Lisp e `batchload` para carregar arquivos Maxima.

Veja também `loadfile`, `batch`, `batchload`, e `demo`. `loadfile` processa arquivos Lisp; `batch`, `batchload`, e `demo` processam arquivos Maxima.

Veja `file_search` para mais detalhes sobre o mecanismo de busca de arquivos.

`load` avalia seu argumento.

loadfile (*filename*) Função

Avalia expressões Lisp em *filename*. `loadfile` não invoca `file_search`, então `filename` deve obrigatóriamente incluir a extensão do arquivo e tanto quanto o caminho como necessário para achar o arquivo.

`loadfile` pode processar arquivos criados por `save`, `translate_file`, e `compile_file`. O usuário pode achar isso mais conveniente para usar `load` em lugar de `loadfile`.

`loadfile` avalia seu argumento, então `filename` deve obrigatóriamente ser uma seqüência de caracteres literal, não uma variável do tipo seqüência de caracteres. O operador aspas simples não aceita avaliação.

loadprint Variável de opção

Valor padrão: `true`

`loadprint` diz se deve imprimir uma mensagem quando um arquivo é chamado.

- Quando `loadprint` é `true`, sempre imprime uma mensagem.
- Quando `loadprint` é `'loadfile`, imprime uma mensagem somente se um arquivo é chamado pela função `loadfile`.
- Quando `loadprint` é `'autoload`, imprime uma mensagem somente se um arquivo é automaticamente carregado. Veja `setup_autoload`.
- Quando `loadprint` é `false`, nunca imprime uma mensagem.

obase Variável de opção

Valor padrão: 10

`obase` é a base para inteiros mostrados pelo Maxima.

A `obase` poode ser atribuído qualquer inteiro entre 2 e 35 (decimal), inclusive. Quando `obase` é maior que 10, os numerais compreendem os numerais decimais de 0 até 9 e letras maiúsulas do alfabeto A, B, C, ..., quando necessário. Os numerais para a base 35, a maior base aceitável, compreendem de 0 até 9, e de A até Y.

Veja também `ibase`.

outchar Variável de opção

Valor padrão: `%o`

`outchar` é o prefixo dos rótulos de expressões calculadas pelo Maxima. Maxima automaticamente constrói um rótulo para cada expressão calculada pela concatenação de `outchar` e `linenum`. A `outchar` pode ser atribuído qualquer seqüência de caracteres ou símbolo, não necessáriamente um caractere simples.

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

Veja também `labels`.

packagefile Variável de opção

Valor padrão: `false`

Projetistas de pacotes que usam `save` ou `translate` para criar pacotes (arquivos) para outros usarem podem querer escolher `packagefile: true` para prevenir qu informações sejam acrescentadas à lista de informações do Maxima (e.g. `values`, `funções`) exceto onde necessário quando o arquivo é carregado. Nesse caminho, o conteúdo do pacote não pegará no caminho do usuário quando ele adicionar seus próprios dados. Note que isso não resolve o problema de possíveis conflitos de nome. Também note que o sinalizador simplesmente afeta o que é saída para o arquivo pacote. Escolhendo o sinalizador para `true` é também útil para criar arquivos de init do Maxima.

pfeformat Variável de opção

Valor padrão: `false`

Quando `pfeformat` é `true`, uma razão de inteiros é mostrada com o caractere sólido (barra normal), e um denominador inteiro `n` é mostrado como um termo multiplicativo em primeiro lugar `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*) Função

Avalia e mostra *expr_1*, ..., *expr_n* uma após a outra, da esquerda para a direita, iniciando no lado esquerdo do console.

O valor retornado por `print` é o valor de seu último argumento. `print` não gera rótulos de expressão intermediária.

Veja também `display`, `disp`, `ldisplay`, e `ldisp`. Essas funções mostram uma expressão por linha, enquanto `print` tenta mostrar duas ou mais expressões por linha.

Para mostrar o conteúdo de um arquivo, veja `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*) Função

tcl_output (*list*, *i0*) Função

tcl_output ([*list_1*, ..., *list_n*], *i*) Função

Imprime os elementos de uma lista entre chaves `{ }`, conveniente como parte de um programa na linguagem Tcl/Tk.

`tcl_output (`*list*`,` *i0*`,` *skip*`)` imprime *list*, começando com o elemento *i0* e imprimindo elementos *i0* + *skip*, *i0* + 2 *skip*, etc.

`tcl_output (`*list*`,` *i0*`)` é equivalente a `tcl_output (`*list*`,` *i0*`, 2)`.

`tcl_output ([`*list_1*`, ...,` *list_n*`],` *i*`)` imprime os *i*'ésimos elementos de *list_1*, ..., *list_n*.

Exemplos:

```
(%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*) Função

Imprime *expr_1*, ..., *expr_n*, então lê uma expressão do console e retorna a expressão avaliada. A expressão é terminada com um ponto e vírgula `;` ou o sinal de dólar `$`.

Veja também `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*) Função

Imprime *expr_1*, ..., *expr_n*, então lê uma expressão do console e retorna a expressão (sem avaliação). A expressão é terminada com um `;` (ponto e vírgula) ou `$` (sinal de dólar).

```
(%i1) aa: 7$
(%i2) foo: readonly ("Forneça uma expressão:");
Enter an expressão:
2^aa;
                                    aa
(%o2)                              2
(%i3) foo: read ("Forneça uma expressão:");
Enter an expressão:
2^aa;
(%o3)                              128
```

Veja também `read`.

reveal (*expr*, *depth*) Função

Substitue partes de *expr* no inteiro especificado *depth* com sumário descritivo.

- Somas e diferenças são substituídas por `sum(`*n*`)` onde *n* é o número de operandos do produto.
- Produtos são substituídos por `product(`*n*`)` onde *n* é o número de operandos da multiplicação.
- Exponenciais são substituídos por `expt`.
- Quocientes são substituídos por `quotient`.
- Negação unária é substituída por `negterm`.

Quando *depth* é maior que ou igual à máxima intensidade de *expr*, `reveal (`*expr*`,` *depth*) retornam *expr* sem modificações.

`reveal` avalia seus argumentos. `reveal` retorna expressão sumarizada.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `]`

`rmxchar` é the caractere desenhado lado direito de uma matriz.

Veja também `lmxchar`.

save (*filename*, *name_1*, *name_2*, *name_3*, ...) Função
save (*filename*, *values*, *functions*, *labels*, ...) Função
save (*filename*, [*m*, *n*]) Função
save (*filename*, *name_1*=*expr_1*, ...) Função
save (*filename*, *all*) Função

Armazena os valores correntes de *name_1*, *name_2*, *name_3*, ..., em *filename*. Os argumentos são os nomes das variáveis, funções, ou outros objetos. Se um nome não possui valore ou função associada a ele, esse nome sem nenhum valor ou função associado será ignorado. `save` retorna *filename*.

`save` armazena dados na forma de expressões Lisp. Os dados armazenados por `save` podem ser recuperados por `load (`*filename*`)`.

O sinalizador global `file_output_append` governa se `save` anexa ao final ou trunca o arquivo de saída. Quando `file_output_append` for `true`, `save` anexa ao final doarquivo de saída. De outra forma, `save` trunca o arquivo de saída. Nesse caso, `save` cria o arquivo se ele não existir ainda.

A forma especial `save (`*filename*`, values, functions, labels, ...)` armazena os ítens nomeados por `values`, `funções`, `labels`, etc. Os nomes podem ser quaisquer especificados pela variável `infolists`. `values` compreende todas as variáveis definidas pelo usuário.

A forma especial `save (`*filename*`, [`*m*`, `*n*`])` armazena os valores de rótulos de entrada e saída de *m* até *n*. Note que *m* e *n* devem obrigatóriamente ser inteiros literais ou símbolos envolvidos por aspas duplas. Rótulos de entrada e saída podem também ser armazenados um a um, e.g., `save ("foo.1", %i42, %o42)`. `save (`*filename*`, labels)` armazena todos os rótulos de entrada e saída. Quando rótulos armazenados são recuperados, eles substituem rótulos existentes.

A forma especial `save (`*filename*`, `*name_1*`=`*expr_1*`, `*name_2*`=`*expr_2*`, ...)` armazena os valores de *expr_1*, *expr_2*, ..., com nomes *name_1*, *name_2*, ... Isso é útil para aplicar essa forma para rótulos de entrada e saída, e.g., `save ("foo.1", aa=%o88)`. O lado direito dessa igualdade nessa forma pode ser qualquer expressão, que é avaliada. Essa forma não introduz os novos nomes no ambiente corrente do Maxima, mas somente armazena-os em *filename*.

Essa forma especial e a forma geral de `save` podem ser misturados. Por exemplo, `save (`*filename*`, aa, bb, cc=42, funções, [11, 17])`.

A forma especial `save (`*filename*`, all)` armazena o estado corrente do Maxima. Isso inclui todas as variáveis definidas pelo usuário, funções, arrays, etc., bem como alguns ítens definidos automaticamente. Os ítes salvos incluem variáveis de sistema, tais como `file_search_maxima` ou `showtime`, se a elas tiverem sido atribuídos novos valores pelo usuário; veja `myoptions`.

`save` avalia seus argumentos. *filename* deve obrigatóriamente ser uma seqüência de caracteres, não uma variável tipo seqüência de caracteres. O primeiro e o último

rótulos a salvar, se especificado, devem obrigatóriamente serem inteiros. O operador aspas duplas avalia uma variável tipo seqüência de caracteres para seu valor seqüência de caracteres, e.g., `s: "foo.1"$ save (''s, all)$`, e variáveis inteiras para seus valores inteiros, e.g., `m: 5$ n: 12$ save ("foo.1", [''m, ''n])$`.

savedef Variável de opção

Valor padrão: `true`

Quando `savedef` é `true`, a vesão Maxima de uma função de usuário é preservada quando a função é traduzida. Isso permite que a definição seja mostrada por `dispfun` e autoriza a função a ser editada.

Quando `savedef` é `false`, os nomes de funções traduzidas são removidos da lista de `funções`.

show (*expr*) Função

Mostra `expr` com os objetos indexados tendo índices covariantes como subscritos, índices contravariantes como sobrescritos. Os índices derivativos são mostrados como subscritos, separados dos índices covariantes por uma vírgula.

showratvars (*expr*) Função

Retorna uma lista de variáveis expressão racional canônica (CRE) na expressão `expr`.

Veja também `ratvars`.

stardisp Variável de opção

Valor padrão: `false`

Quando `stardisp` é `true`, multiplicação é mostrada com um asterisco `*` entre os operandos.

string (*expr*) Função

Converte `expr` para a notação linear do Maxima apenas como se tivesse sido digitada.

O valor de retorno de `string` é uma seqüência de caracteres, e dessa forma não pode ser usada em um cálculo.

stringdisp Variável Lisp

Valor padrão: `false`

Quando `?stringdisp` for `true`, seqüências de caracteres serão mostradas contidas em aspas duplas. De outra forma, aspas não são mostradas.

`?stringdisp` é sempre `true` quando mostrando uma definição de função.

`?stringdisp` é uma variável Lisp, então deve ser escrita com um ponto de interrogação `?` na frente.

Exemplos:

```
(%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, ...*) Função
stringout (*filename,* [*m, n*]) Função
stringout (*filename, input*) Função
stringout (*filename, functions*) Função
stringout (*filename, values*) Função

`stringout` escreve expressões para um arquivo na mesma forma de expressões que foram digitadas para entrada. O arquivo pode então ser usado como entrada para comandos `batch` ou `demo`, e isso pode ser editado para qualquer propósito. `stringout` pode ser executado enquanto `writefile` está em progresso.

O sinalizador global `file_output_append` governa se `stringout` anexa ao final ou trunca o arquivo de saída. Quando `file_output_append` for `true`, `stringout` anexa ao final do arquivo de saída. De outra forma, `stringout` trunca o arquivo de saída. Nesse caso, `stringout` cria o arquivo de saída se ele não existir ainda.

A forma geral de `stringout` escreve os valores de um ou mais expressões para o arquivo de saída. Note que se uma expressão é uma variável, somente o valor da variável é escrito e não o nome da variável. Como um útil caso especial, as expressões podem ser rótulos de entrada (`%i1`, `%i2`, `%i3`, ...) ou rótulos de saída (`%o1`, `%o2`, `%o3`, ...).

Se `grind` é `true`, `stringout` formata a saída usando o formato `grind`. De outra forma o formato `string` é usado. Veja `grind` e `string`.

A forma especial `stringout (`*filename*`, [`*m*`,` *n*`])` escreve os valores dos rótulos de entrada de m até n, inclusive.

A forma especial `stringout (`*filename*`, input)` escreve todos os rótulos de entrada para o arquivo.

A forma especial `stringout (`*filename*`, functions)` escreve todas as funções definidas pelo usuário (nomeadas pela lista global `functions`) para o arquivo.

A forma especial `stringout (`*filename*`, values)` escreve todas as variáveis atribuídas pelo usuário (nomeadas pela lista global `values`) para o arquivo. Cada variável é impressa como uma declaração de atribuição, com o nome da variável seguida de dois pontos, e seu valor. Note que a forma geral de `stringout` não imprime variáveis como declarações de atribuição.

tex (*expr*) Função
tex (*rótulo*) Função
tex (*expr, momearquivo*) Função
tex (*label, nomearquivo*) Função

Imprime uma representação de uma expressão adequada para o sistema TeX de preparação de documento. O resultado é um fragmento de um documento, que pode ser copiado dentro de um documento maior. Esse fragmento não pode ser processado de forma direta e isolada.

`tex` (*expr*) imprime uma representação TeX da *expr* no console.

`tex` (*rótulo*) imprime uma representação TeX de uma expressão chamada *rótulo* e atribui a essa um rótulo de equação (a ser mostrado à esquerda da expressão). O rótulo de equação TeX é o mesmo que o rótulo da equação no Maxima.

`tex` (*expr*, *nomearquivo*) anexa ao final uma representaação TeX de *expr* no arquivo *nomearquivo*.

`tex` (*rótulo*, *nomearquivo*) anexa ao final uma representação TeX da expressão chamada de *rótulo*, com um rótulo de equação, ao arquivo *nomearquivo*.

`tex` avalia seus argumentos após testar esse argumento para ver se é um rótulo. duplo apóstrofo `''` força a avaliação do argumento, desse modo frustrando o teste e prevenindo o rótulo. Apóstrofo simples `'` previne avaliação.

Veja também `texput`.

Exemplos:

```
(%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)
```

texput (*a*, *s*) Função
texput (*a*, *s*, *operator_type*) Função
texput (*a*, [*s_1*, *s_2*], *matchfix*) Função
texput (*a*, [*s_1*, *s_2*, *s_3*], *matchfix*) Função

Escolhe a saída TeX para o átomo *a*, que pode ser um símbolo ou o nome de um operador.

`texput` (*a*, *s*) faz com que a função `tex` interpole a seqüência de caracteres *s* dentro da saída TeX em lugar de *a*.

`texput` (*a*, *s*, *operator_type*), onde *operator_type* é `prefix`, `infix`, ou `postfix` faz com que a função `tex` interpole *s* dentro da saída TeX em lugar de *a*, e coloca o texto interpolado na posição apropriada.

`texput` (*a*, [*s_1*, *s_2*], `matchfix`) faz com que a função `tex` interpole *s_1* e *s_2* dentro da saída TeX sobre qualquer lado dos argumentos de *a*. Os argumentos (se mais de um) são separados por vírgulas.

`texput (`*a*`, [`*s_1*`,` *s_2*`,` *s_3*`], matchfix)` faz com que a função `tex` interpole *s_1* e *s_2* dentro da saída TeX sobre qualquer lado dos argumentos de *a*, com *s_3* separando os argumentos.

Exemplos:

```
(%i1) texput (me,"\\mu_e");
(%o1)                          \mu_e
(%i2) tex (me);
$$\mu_e$$
(%o2)                          false
(%i3) texput (lcm, "\\mathrm{lcm}");
(%o3)                       \mathrm{lcm}
(%i4) tex (lcm (a, b));
$$\mathrm{lcm}\left(a , b\right)$$
(%o4)                          false
(%i5) prefix ("grad");
(%o5)                          grad
(%i6) texput ("grad", " \\nabla ", prefix);
(%o6)                           180
(%i7) tex (grad f);
$$ \nabla f$$
(%o7)                          false
(%i8) infix ("~");
(%o8)                            ~
(%i9) texput ("~", " \\times ", infix);
(%o9)                           180
(%i10) tex (a ~ b);
$$a \times b$$
(%o10)                         false
(%i11) postfix ("@");
(%o11)                           @
(%i12) texput ("@", "!!", postfix);
(%o12)                          160
(%i13) tex (x @);
$$x!!$$
(%o13)                         false
(%i14) matchfix ("<<", ">>");
(%o14)                           <<
(%i15) texput ("<<", [" \\langle ", " \\rangle "], matchfix);
(%o15)              \langle ( \rangle , false)
(%i16) tex (<<a>>);
$$ \langle a \rangle $$
(%o16)                         false
(%i17) tex (<<a, b>>);
$$ \langle a , b \rangle $$
(%o17)                         false
(%i18) texput ("<<", [" \\langle ", " \\rangle ", " \\, | \\,"], matchfix);
(%o18)            \langle ( \rangle ,  \, | \,)
(%i19) tex (<<a>>);
$$ \langle a \rangle $$
```

```
(%o19)                          false
(%i20) tex (<<a, b>>);
$$ \langle a \, | \,b \rangle $$
(%o20)                          false
```

system (*comando*) Função

Executa *comando* como um processo separado. O comando é passado ao shell padraõ para execução. `system` não é suportado por todos os sistemas operacionais, mas geralmente existe em ambientes Unix e Unix-like.

Supondo que `_hist.out` é uma lista de freqüência que você deseja imprimir como um gráfico em barras usando `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"));
```

Com o objetivo de fazer com que a impressão do gráfico seja concluída em segundo plano (retornando o controle para o Maxima) e remover o arquivo temporário após isso ter sido concluído faça:

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

ttyoff Variável de opção

Valor padrão: `false`

Quando `ttyoff` é `true`, expressões de saída não são mostradas. Expressões de saída são ainda calculadas e atribuídas rótulos. Veja `labels`.

Textos impresso por funções internas do Maxima, tais como mensagens de erro e a saída de `describe`, não são afetadas por `ttyoff`.

with_stdout (*filename, expr_1, expr_2, expr_3, ...*) Função

Abre *filename* e então avalia *expr_1*, *expr_2*, *expr_3*, ... Os valores dos argumentos não são armazenados em *filename*, mas qualquer saída impressa gerada pela avaliação dos argumentos (de `print`, `display`, `disp`, ou `grind`, por exemplo) vai para *filename* em lugar do console.

O sinalizador global `file_output_append` governa se `with_stdout` anexa ao final ou trunca o arquivo de saída. Quando `file_output_append` for `true`, `with_stdout` anexa ao final do arquivo de saída. De outra forma, `with_stdout` trunca o arquivo de saída. Nesse caso, `with_stdout` cria o arquivo se ele não existir ainda.

`with_stdout` retorna o valor do seu argumento final.

Veja também `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*) Função

Começa escrevendo uma transcrição da sessão Maxima para *filename*. Toda interação entre o usuário e Maxima é então gravada nesse arquivo, da mesma forma que aparece no console.

Como a transcrição é impressa no formato de saída do console, isso não pode ser reaproveitado pelo Maxima. Para fazer um arquivo contendo expressões que podem ser reaproveitadas, veja `save` e `stringout`. `save` armazena expressões no formato Lisp, enquanto `stringout` armazena expressões no formato Maxima.

O efeito de executar `writefile` quando *filename* ainda existe depende da implementação Lisp subjacente; o arquivo transcrito pode ser substituído, ou o arquivo pode receber um anexo. `appendfile` sempre anexa para o arquivo transcrito.

Isso pode ser conveniente para executar `playback` após `writefile` para salvar a visualização de interações prévias. Como `playback` mostra somente as variáveis de entrada e saída (`%i1`, `%o1`, etc.), qualquer saída gerada por uma declaração de impressão em uma função (como oposição a um valor de retorno) não é mostrada por `playback`.

`closefile` fecha o arquivo transcrito aberto por `writefile` ou `appendfile`.

10 Ponto Flutuante

10.1 Definições para ponto Flutuante

bffac (*expr, n*) Função

Versão para grandes números em ponto flutuante da função `factorial` (usa o artifício gamma). O segundo argumento informa quantos dígitos reter e retornar, isso é uma boa idéia para requisitar precisão adicional.

`load ("bffac")` chama essa função.

algepsilon Variável de Opção

Valor padrão: 10^8

`algepsilon` é usada por `algsys`.

bfloat (*expr*) Função

Converte todos os números e funções de números em *expr* para grandes números em ponto flutuante (bigfloat). O número de algarismos significativos no grande número em ponto flutuante resultante é especificado através da variável global `fpprec`.

Quando `float2bf` for `false` uma mensagem de alerta é mostrada quando uma número em ponto flutuante (float) é convertido em um grande número em ponto flutuante (bigfloat - uma vez que isso pode resultar em perda de precisão).

bfloatp (*expr*) Função

Retorna `true` se a avaliação da *expr* resultar em um grande número em ponto flutuante, de outra forma retorna `false`.

bfpsi (*n, z, fpprec*) Função

bfpsi0 (*z, fpprec*) Função

`bfpsi` é a função `polygamma` de argumentos reais *z* e ordem de inteiro *n*. `bfpsi0` é a função `digamma`. `bfpsi0 (`*z*`,` *fpprec*`)` é equivalente a `bfpsi (0,` *z*`,` *fpprec*`)`.

Essas funções retornam valores em grandes números em ponto flutuante. *fpprec* é a precisão do valor de retorno dos grandes números em ponto flutuante.

`load ("bffac")` chama essas funções.

bftorat Variável de Opção

Valor padrão: `false`

`bftorat` controla a conversão de `bfloats` para números racionais. Quando `bftorat` for `false`, `ratepsilon` será usada para controlar a conversão (isso resulta em números racionais relativametne pequenos). Quando `bftorat` for `true`, o número racional gerado irá representar precisamente o `bfloat`.

bftrunc Variável de Opção

Valor padrão: `true`

`bftrunc` faz com que tilhas de zeros em grandes números em ponto flutuante diferentes de zero sejam ocultadas. Desse modo, se `bftrunc` for `false`, `bfloat (1)` será mostrado como `1.000000000000000B0`. De outra forma, será mostrado como `1.0B0`.

cbffac (*z*, *fpprec*) Função

Fatorial complexo de grandes números em ponto flutuante.

`load ("bffac")` chama essa função.

float (*expr*) Função

Converte inteiros, números racionais e grandes números em ponto flutuante em *expr* para números em ponto flutuante. Da mesma forma um `evflag`, `float` faz com que números racionais não-inteiros e grandes números em ponto flutuante sejam convertidos para ponto flutuante.

float2bf Variável de Opção

Valor padrão: `false`

Quando `float2bf` for `false`, ma mensagem de alerta é mostrada quando um número em ponto flutuante é convertido em um grande número em ponto flutuante (uma vez que isso pode resultar em perda de precisão).

floatnump (*expr*) Função

Retorna `true` se *expr* for um número em ponto flutuante, de outra forma retorna `false`.

fpprec Variável de Opção

Valor padrão: 16

`fpprec` é o número de algarismos significativos para aritmética sobre grandes números em ponto flutuante `fpprec` não afeta cálculos sobre números em ponto flutuante comuns.

Veja também `bfloat` e `fpprintprec`.

fpprintprec Variável de Opção

Valor padrão: 0

`fpprintprec` é o número de dígitos para impressão quando imprimindo um grande número em ponto flutuante, tornando possível calcular com grande número de dígitos de precisão, mas ter na resposta impressa com um pequeno número de dígitos.

Quando `fpprintprec` for 0, ou maior que ou igual a `fpprec`, então o valor de `fpprec` controla o número de dígitos usado para imprimir.

Quando `fpprintprec` tem um valor entre 2 e `fpprec - 1`, então `fpprintprec` controla o número de dígitos usado. (O menor número de dígitos usado é 2, um do lado esquerdo do ponto e um do lado direito.

O valor 1 para `fpprintprec` é ilegal.

?round (*x*) Função Lisp
?round (*x*, *divisor*) Função Lisp

Arredonda o ponto flutuante *x* para o inteiro mais próximo. O argumento obrigatoriamente deve ser um ponto flutuante comum, não um grandes números em ponto flutuante. A `?` começando o nome indica que isso é uma função Lisp.

```
(%i1) ?round (-2.8);
(%o1)                          - 3
```

?truncate (*x*) Função Lisp
?truncate (*x*, *divisor*) Função Lisp

Trunca o ponto flutuante *x* na direção do 0, para transormar-se em um inteiro. O argumento deve ser um número em ponto flutuante comum, não um grandes números em ponto flutuante. A `?` começando o nome indica que isso é uma função Lisp.

```
(%i1) ?truncate (-2.8);
(%o1)                          - 2
(%i2) ?truncate (2.4);
(%o2)                           2
(%i3) ?truncate (2.8);
(%o3)                           2
```

11 Contextos

11.1 Definições para Contextos

activate (*context_1, ..., context_n*) Função

Ativa os contextos *context_1*, ..., *context_n*. Os fatos nesses contextos estão então disponíveis para fazer deduções e recuperar informação. Os fatos nesses contextos não são listadas através de `facts ()`.

A variável `activecontexts` é a lista de contextos que estão ativos pelo caminho da função `activate`.

activecontexts System variable

Valor padrão: `[]`

`activecontexts` é a lista de contextos que estão ativos pelo caminho da função `activate`, em oposição a sendo ativo porque eles são subcontextos do contexto corrente.

assume (*pred_1, ..., pred_n*) Função

Adiciona predicados *pred_1*, ..., *pred_n* para a base de dados corrente, após verificar se existe redundncia e inconsistência. Se os predicados forem inconsistentes e não redundntes, eles são adicionados à base de dados; se inconsistente ou redundnte, nenhuma ação é tomada.

`assume` retorna uma lista cujos elementos são predicados adicionados à base de dados e os átomos `redundant` ou `inconsistent` onde for aplicável.

assumescalar Option variable

Valor padrão: `true`

`assumescalar` ajuda a governar se expressões `expr` para as quais `nonscalarp (expr)` for `false` são assumidas comportar-se como escalares para certas transformações.

Tomemos `expr` representando qualquer expressão outra que não uma lista ou uma matriz, e tomemos `[1, 2, 3]` representando qualquer lista ou matriz. Então `expr . [1, 2, 3]` retorna `[expr, 2 expr, 3 expr]` se `assumescalar` for `true`, ou `scalarp (expr)` for `true`, ou `constantp (expr)` for `true`.

Se `assumescalar` for `true`, tais expressões irão comportar-se como escalares somente para operadores comutativos, mas não para multiplicação não comutativa `..`

Quando `assumescalar` for `false`, tais expressões irão comportar-se como não escalares.

Quando `assumescalar` for `all`, tais expressões irão comportar-se como escalares para todos os operadores listados acima.

assume_pos Option variable

Valor padrão: `false`

Quando `assume_pos` for `true` e o sinal de um parmetro x não pode ser determinado a partir da base de dados do `assume` ou outras considerações, `sign` e `asksign` (`x`) retornam `true`. Isso pode impedir algum questionamento de `asksign` gerado automaticamente, tal como pode surgir de `integrate` ou de outros cálculos.

Por padrão, um parmetro é x tal como `symbolp` (`x`) or `subvarp` (`x`). A classe de expressões consideradas parmetros pode ser modificada para alguma abrangência através da variável `assume_pos_pred`.

`sign` e `asksign` tentam deduzir o sinal de expressões a partir de sinais de operandos dentro da expressão. Por exemplo, se `a` e `b` são ambos positivos, então `a + b` é também positivo.

Todavia, não existe caminho para desviar todos os questionamentos de `asksign`. Particularmente, quando o argumento de `asksign` for uma diferença x `-` y ou um logarítmo `log`(x), `asksign` sempre solicita uma entrada ao usuário, mesmo quando `assume_pos` for `true` e `assume_pos_pred` for uma função que retorna `true` para todos os argumentos.

assume_pos_pred Variável de opção

Valor padrão: `false`

Quando `assume_pos_pred` for atribuído o nome de uma função ou uma expressão lambda de um argumento x, aquela função é chamada para determinar se x é considerado um parmetro para o propósito de `assume_pos`. `assume_pos_pred` é ignorado quando `assume_pos` for `false`.

A função `assume_pos_pred` é chamada através de `sign` e de `asksign` com um argumento x que é ou um átomo, uma variável subscrita, ou uma expressão de chamada de função. Se a função `assume_pos_pred` retorna `true`, x é considerado um parmetro para o propósito de `assume_pos`.

Por padrão, um parmetro é x tal que `symbolp (x)` ou `subvarp (x)`.

Veja também `assume` e `assume_pos`.

Exemplos:

```
(%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 Variável de opção

Valor padrão: `initial`

`context` nomeia a coleção de fatos mantida através de `assume` e `forget`. `assume` adiciona fatos à coleção nomeada através de `context`, enquanto `forget` remove fatos.

Associando `context` para um nome *foo* altera o contexto corrente para *foo*. Se o contexto especificado *foo* não existe ainda, ele é criado automaticamente através de uma chamada a `newcontext`. O contexto especificado é ativado automaticamente.

Veja `context` para uma descrição geral do mecanismo de contexto.

contexts Variável de opção

Valor padrão: `[initial, global]`

`contexts` é uma lista dos contextos que existem atualmente, incluindo o contexto ativo atualmente.

O mecanismo de contexto torna possível para um usuário associar junto e nomear uma porção selecionada de sua base de dados, chamada um contexto. Assim que isso for concluído, o usuário pode ter o Maxima assumindo ou esquecendo grande quantidade de fatos meramente através da ativação ou desativação seu contexto.

Qualquer átomo simbólico pode ser um contexto, e os fatos contidos naquele contexto irão ser retidos em armazenamento até que sejam destruídos um por um através de chamadas a `forget` ou destruídos com um conjunto através de uma chamada a `kill` para destruir o contexto que eles pertencem.

Contextos existem em uma hierarquía, com o raíz sempre sendo o contexto `global`, que contém informações sobre Maxima que alguma função precisa. Quando em um contexto dado, todos os fatos naquele contexto estão "ativos" (significando que eles são usados em deduções e recuperados) como estão também todos os fatos em qualquer contexto que for um subcontexto do contexto ativo.

Quando um novo Maxima for iniciado, o usuário está em um contexto chamado `initial`, que tem `global` como um subcontexto.

Veja também `facts`, `newcontext`, `supcontext`, `killcontext`, `activate`, `deactivate`, `assume`, e `forget`.

deactivate (*context_1, ..., context_n*) Função

Desativa os contextos especificados *context_1*, ..., *context_n*.

facts (*item*) Função

facts () Função

Se *item* for o nome de um contexto, `facts (`*item*`)` retorna uma lista de fatos no contexto especificado.

Se *item* não for o nome de um contexto, `facts (`*item*`)` retorna uma lista de fatos conhecidos sobre *item* no contexto atual. Fatos que estão atuvos, mas em um diferente contexto, não são listados.

`facts ()` (i.e., sem argumento) lista o contexto atual.

features Declaration

Maxima recnhece ceertas propriedades matemáticas de funções e variáveis. Essas são chamadas "recursos".

`declare (`*x*`,` *foo*`)` fornece a propriedade *foo* para a função ou variável *x*.

`declare (`*foo*`, recurso)` declara um novo recurso *foo*. Por exemplo, `declare ([red, green, blue], feature)` declara três novos recursos, `red`, `green`, e `blue`.

O predicado `featurep (`*x*`,` *foo*`)` retorna `true` se *x* possui a propriedade *foo*, e `false` de outra forma.

A infolista `features` é uma lista de recursos conhecidos. São esses `integer`, `noninteger`, `even`, `odd`, `rational`, `irrational`, `real`, `imaginary`, `complex`, `analytic`, `increasing`, `decreasing`, `oddfun`, `evenfun`, `posfun`, `commutative`, `lassociative`, `rassociative`, `symmetric`, e `antisymmetric`, mais quaisquer recursos definidos pelo usuário.

`features` é uma lista de recursos matemáticos. Existe também uma lista de recursos não matemáticos, recursos dependentes do sistema. Veja `status`.

forget (*pred_1, ..., pred_n*) Função
forget (*L*) Função

Remove predicados estabelecidos através de `assume`. Os predicados podem ser expressões equivalentes a (mas não necessáriamente idênticas a) esses prevamentes assumidos.

`forget (`*L*`)`, onde *L* é uma lista de predicados, esquece cada item da lista.

killcontext (*context_1, ..., context_n*) Função

Mata os contextos *context_1*, ..., *context_n*.

Se um dos contextos estiver for o contexto atual, o novo contexto atual irá tornar-se o primeiro subcontexto disponível do contexto atual que não tiver sido morto. Se o primeiro contexto disponível não morto for `global` então `initial` é usado em seu lugar. Se o contexto `initial` for morto, um novo, porém vazio contexto `initial` é criado.

`killcontext` recusa-se a matar um contexto que estiver ativo atualmente, ou porque ele é um subcontexto do contexto atual, ou através do uso da função `activate`.

`killcontext` avalia seus argumentos. `killcontext` retorna `done`.

newcontext (*nome*) Função

Cria um novo contexto, porém vazio, chamado *nome*, que tem `global` como seu único subcontexto. O contexto recentemente criado torna-se o contexto ativo atualmente.

`newcontext` avalia seu argumento. `newcontext` retorna *nome*.

supcontext (*nome, context*) Função
supcontext (*nome*) Função

Cria um novo contexto, chamado *nome*, que tem *context* como um subcontexto. *context* deve existir.

Se *context* não for especificado, o contexto atual é assumido.

12 Polinômios

12.1 Introdução a Polinômios

Polinômios são armazenados no Maxima ou na forma geral ou na forma de Expressões Racionais Canônicas (CRE). Essa última é uma forma padrão, e é usada internamente por operações tais como `factor`, `ratsimp`, e assim por diante.

Expressões Racionais Canônicas constituem um tipo de representação que é especialmente adequado para polinômios expandidos e funções racionais (também para polinômios parcialmente fatorados e funções racionais quando RATFAC for escolhida para `true`). Nessa forma CRE uma ordenação de variáveis (da mais para a menos importante) é assumida para cada expressão. Polinômios são representados recursivamente por uma lista consistindo da variável principal seguida por uma série de pares de expressões, uma para cada termo do polinômio. O primeiro membro de cada par é o expoente da variável principal naquele termo e o segundo membro é o coeficiente daquele termo que pode ser um número ou um polinômio em outra variável novamente respresentado nessa forma. Sendo assim a parte principal da forma CRE de 3*X^2-1 é (X 2 3 0 -1) e que a parte principal da forma CRE de 2*X*Y+X-3 é (Y 1 (X 1 2) 0 (X 1 1 0 -3)) assumindo Y como sendo a variável principal, e é (X 1 (Y 1 2 0 1) 0 -3) assumindo X como sendo a variável principal. A variável principal é usualmente determineda pela ordem alfabética reversa. As "variáveis" de uma expressão CRE não necessariamente devem ser atômicas. De fato qualquer subexpressão cujo principal operador não for + - * / or ^ com expoente inteiro será considerado uma "variável" da expressão (na forma CRE) na qual essa ocorrer. Por exemplo as variáveis CRE da expressão X+SIN(X+1)+2*SQRT(X)+1 são X, SQRT(X), e SIN(X+1). Se o usuário não especifica uma ordem de variáveis pelo uso da função RATVARS Maxima escolherá a alfabética por conta própria. Em geral, CREs representam expressões racionais, isto é, razões de polinômios, onde o numerador e o denominador não possuem fatores comuns, e o denominador for positivo. A forma interna é essencialmente um par de polinômios (o numerador e o denominador) precedidos pela lista de ordenação de variável. Se uma expressão a ser mostrada estiver na forma CRE ou se contiver quaisquer subexpressões na forma CRE, o símbolo /R/ seguirá o rótulo da linha. Veja a função RAT para saber como converter uma expressão para a forma CRE. Uma forma CRE extendida é usada para a representação de séries de Taylor. A noção de uma expressão racional é extendida de modo que os expoentes das variáveis podem ser números racionais positivos ou negativos em lugar de apenas inteiros positivos e os coeficientes podem eles mesmos serem expressões racionais como descrito acima em lugar de apenas polinômios. Estes são representados internamente por uma forma polinomial recursiva que é similar à forma CRE e é a generalização dessa mesma forma CRE, mas carrega informação adicional tal com o grau de truncação. Do mesmo modo que na forma CRE, o símbolo /T/ segue o rótulo de linha que contém as tais expressões.

12.2 Definições para Polinômios

algebraic Variável de opção

Valor Padrão: `false`

`algebraic` deve ser escolhida para `true` com o objetivo de que a simplificação de inteiros algébricos tenha efeito.

berlefact Variável de opção

Valor Padrão: `true`

Quando `berlefact` for `false` então o algorítmo de fatoração de Kronecker será usado. De outra forma o algorítmo de Berlekamp, que é o padrão, será usado.

bezout (*p1, p2, x*) Função

uma alternativa para o comando `resultant`. Isso retorna uma matriz. `determinant` dessa matriz é o resultante desejado.

bothcoef (*expr, x*) Função

Retorna uma lista da qual o primeiro membro é o coeficiente de *x* em *expr* (como achado por `ratcoef` se *expr* está na forma CRE de outro modo por `coeff`) e cujo segundo membro é a parte restante de *expr*. Isto é, `[A, B]` onde *expr* `= A*`*x* `+ B`.

Exemplo:

```
(%i1) islinear (expr, x) := block ([c],
        c: bothcoef (rat (expr, x), x),
        é (freeof (x, c) and c[1] # 0))$
(%i2) islinear ((r^2 - (x - r)^2)/x, x);
(%o2)                          true
```

coeff (*expr, x, n*) Função

Retorna o coeficiente de `x^`*n* em *expr*. *n* pode ser omitido se for 1. *x* pode ser um átomo, ou subexpressão completa de *expr* e.g., `sin(x)`, `a[i+1]`, `x + y`, etc. (No último caso a expressão `(x + y)` pode ocorrer em *expr*). Algumas vezes isso pode ser necessário para expandir ou fatorar *expr* com o objetivo de fazer `x^`*n* explicito. Isso não é realizado por `coeff`.

Exemplos:

```
(%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*) Função

Simplifica a adição *expr* por termos combinados com o mesmo denominador dentro de um termo simples.

content (*p_1, x_1, ..., x_n*) Função

Retorna uma lista cujo primeiro elemento é o máximo divisor comum dos coeficientes dos termos do polinômio *p_1* na variável *x_n* (isso é o conteúdo) e cujo segundo elemento é o polinômio *p_1* dividido pelo conteúdo.

Exemplos:

```
(%i1) content (2*x*y + 4*x^2*y^2, y);
                                     2
(%o1)                    [2 x, 2 x y  + y]
```

denom (*expr*) Função

Retorna o denominador da expressão racional *expr*.

divide (*p_1*, *p_2*, *x_1*, ..., *x_n*) Função

calcula o quocietne e o resto do polinômio *p_1* dividido pelo polinômio *p_2*, na variável principal do polinômio, *x_n*. As outras variáveis são como na função `ratvars`. O resultado é uma lista cujo primeiro elemento é o quociente e cujo segundo elemento é o resto.

Exemplos:

```
(%i1) divide (x + y, x - y, x);
(%o1)                         [1, 2 y]
(%i2) divide (x + y, x - y);
(%o2)                        [- 1, 2 x]
```

Note que `y` é a variável principal no segundo exemplo.

eliminate ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_k*]) Função

Elimina variáveis de equações (ou expressões assumidas iguais a zero) pegando resultantes sucessivos. Isso retorna uma lista de *n* - *k* expressões com *k* variáveis *x_1*, ..., *x_k* eliminadas. Primeiro *x_1* é eliminado retornando *n* - 1 expressões, então `x_2` é eliminado, etc. Se *k* = *n* então uma expressão simples em uma lista é retornada livre das variáveis *x_1*, ..., *x_k*. Nesse caso `solve` é chamado para resolver a última resultante para a última variável.

Exemplo:

```
(%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*, ...) Função

Retorna uma lista cujo primeiro elemento é o m.d.c. dos polinômios *p_1*, *p_2*, *p_3*, ... e cujos restantes elementos são os polinômios divididos pelo mdc. Isso sempre usa o algoritmo `ezgcd`.

facexpand Variável de opção

Valor Padrão: `true`

`facexpand` controla se os fatores irredutíveis retornados por `factor` estão na forma expandida (o padrão) ou na forma recursiva (CRE normal).

factcomb (*expr*) Função

Tenta combinar os coeficientes de fatoriais em *expr* com os próprios fatoriais convertendo, por exemplo, `(n + 1)*n!` em `(n + 1)!`.

`sumsplitfact` se escolhida para `false` fará com que `minfactorial` seja aplicado após um `factcomb`.

factor (*expr*) Função

Fatora a expressão *expr*, contendo qualquer número de variáveis ou funções, em fatores irredutíveis sobre os inteiros. `factor (`*expr*`, p)` fatora *expr* sobre o campo dos inteiros com um elemento adjunto cujo menor polinômio é p.

`factorflag` se `false` suprime a fatoração de fatores inteiros de expressões racionais.

`dontfactor` pode ser escolhida para uma lista de variáveis com relação à qual fatoração não é para ocorrer. (Essa é inicialmente vazia). Fatoração também não acontece com relação a quaisquer variáveis que são menos importantes (usando a ordenação de variável assumida pela forma CRE) como essas na lista `dontfactor`.

`savefactors` se `true` faz com que os fatores de uma expressão que é um produto de fatores seja guardada por certas funções com o objetivo de aumentar a velocidade de futuras fatorações de expressões contendo alguns dos mesmos fatores.

`berlefact` se `false` então o algoritmo de fatoração de Kronecker será usado de outra forma o algoritmo de Berlekamp, que é o padrão, será usado.

`intfaclim` é o maior divisor que será tentado quando fatorando um grande número inteiro. Se escolhida para `false` (esse é o caso quando o usuário chama explicitamente `factor`), ou se o inteiro for um `fixnum` (i.e. ajustado à paravra da máquina), a fatoração completa do inteiro será tentada. A escolha do usuário para `intfaclim` é usada para chamadas internas para `factor`. Dessa forma, `intfaclim` pode ser colocado em zero para prevenir Maxima de gastar um tempo excessivamente fatorando grandes inteiros.

Exemplos:

```
(%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 Variável de opção

Valor Padrão: `false`

Quando `factorflag` for `false`, suprime a fatoração de fatores inteiros em expressões racionais.

factorout (*expr, x_1, x_2, ...*) Função

Rearranja a adição *expr* em uma adição de parcelas da forma `f (x_1, x_2, ...)*g` onde `g` é um produto de expressões que não possuem qualquer *x_i* e `f` é fatorado.

factorsum (*expr*) Função

Tenta agrupar parcelas em fatores de *expr* que são adições em grupos de parcelas tais que sua adição é fatorável. `factorsum` pode recuperar o resultado de `expand ((x + y)^2 + (z + w)^2)` mas não pode recuperar `expand ((x + 1)^2 + (x + y)^2)` porque os termos possuem variáveis em comum.

Exemplo:

```
(%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*) Função

Retorna o produto dos polinômios *p_1* e *p_2* usando um algorítmo especial para a multiplicação de polinômios. `p_1` e `p_2` podem ser de várias variáveis, densos, e aproximadamente do mesmo tamanho. A multiplicação clássica é de ordem `n_1 n_2` onde `n_1` é o grau de `p_1` and `n_2` é o grau de `p_2`. `fasttimes` é da ordem `max (n_1, n_2)^1.585`.

fullratsimp (*expr*) Função

`fullratsimp` aplica repetidamente `ratsimp` seguido por simplificação não racional a uma expressão até que nenhuma mudança adicional ocorra, e retorna o resultado.

Quando expressões não racionais estão envolvidas, uma chamada a `ratsimp` seguida como é usual por uma simplificação não racional ("geral") pode não ser suficiente para retornar um resultado simplificado. Algumas vezes, mais que uma tal chamada pode ser necessária. `fullratsimp` faz esse processo convenientemente.

`fullratsimp` (*expr*, *x_1*, ..., *x_n*) pega um ou mais argumentos similar a `ratsimp` e `rat`.

Exemplo:

```
(%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*) Função

é o mesmo que `ratsubst` exceto que essa chama a si mesma recursivamente sobre esse resultado até que o resultado para de mudar. Essa função é útil quando a expressão de substituição e a expressão substituída tenham uma ou mais variáveis em comum.

`fullratsubst` irá também aceitar seus argumentos no formato de `lratsubst`. Isto é, o primeiro argumento pode ser uma substituição simples de equação ou uma lista de tais equações, enquanto o segundo argumento é a expressão sendo processada.

`load ("lrats")` chama `fullratsubst` e `lratsubst`.

Exemplos:

```
(%i1) load ("lrats")$
```

- `subst` pode realizar multiplas substituições. `lratsubst` é analogo a `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
```

- Se somente uma substituição é desejada, então uma equação simples pode ser dada como primeiro argumento.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

- `fullratsubst` é equivalente a `ratsubst` exceto que essa executa recursivamente até que seu resultado para de mudar.

```
(%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` também aceita uma lista de equações ou uma equação simples como primeiro argumento.

```
(%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` pode causar uma recursão infinita.

```
(%i9) errcatch (fullratsubst (b*a^2, a^2, a^3));

*** - Lisp stack overflow. RESET
```

gcd (*p_1*, *p_2*, *x_1*, ...) Função

Retorna o máximo divisor comum entre *p_1* e *p_2*. O sinalizador `gcd` determina qual algorítmo é empregado. Escolhendo `gcd` para `ez`, `eez`, `subres`, `red`, ou `spmod` seleciona o algorítmo `ezgcd`, Novo `eez gcd`, `prs` subresultante, reduzido, ou modular, respectivamente. Se `gcd` for `false` então GCD(p1,p2,var) irá sempre retornar 1 para todas as `variáveis`. Muitas funções (e.g. `ratsimp`, `factor`, etc.) fazem com que mdc's sejam feitos implicitamente. Para polinômios homogêneos é recomendado que `gcd` igual a `subres` seja usado. Para pegar o mdc quando uma expressão algébrica está presente, e.g. GCD(X^2-2*SQRT(2)*X+2,X-SQRT(2)); , `algebraic` deve ser

`true` e `gcd` deve não ser `ez`. `subres` é um novo algorítmo, e pessoas que tenham estado usando a opção `red` podem provavelmente alterar isso para `subres`.

O sinalizador `gcd`, padrão: `subres`, se `false` irá também evitar o máximo divisor comum de ser usado quando expressões são convertidas para a forma de expressão racional canônica (CRE). Isso irá algumas vezes aumentar a velocidade dos cálculos se mdc's não são requeridos.

gcdex (*f*, *g*) Função
gcdex (*f*, *g*, *x*) Função

Retornam uma lista `[a, b, u]` onde *u* é o máximo divisor comum (mdc) entre *f* e *g*, e *u* é igual a *a f* `+` *b g*. Os argumentos *f* e *g* podem ser polinômios de uma variável, ou de outra forma polinômios em *x* uma **main**(principal) variável suprida desde que nós precisamos estar em um domínio de ideal principal para isso trabalhar. O mdc significa o mdc considerando *f* e *g* como polinômios de uma única variável com coeficientes sendo funções racionais em outras variáveis.

`gcdex` implementa o algorítmo Euclideano, onde temos a seqüência of `L[i]: [a[i], b[i], r[i]]` que são todos perpendiculares a `[f, g, -1]` e o próximo se é construído como se `q = quotient(r[i]/r[i+1])` então `L[i+2]: L[i] - q L[i+1]`, e isso encerra em `L[i+1]` quando o resto `r[i+2]` for 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 que o mdc adiante é `1` uma vez que trabalhamos em `k(y)[x]`, o `y+1` não pode ser esperado em `k[y, x]`.

```
(%i1) gcdex (x*(y + 1), y^2 - 1, x);
                                  1
(%o1)/R/                 [0, ------, 1]
                                2
                               y  - 1
```

gcfactor (*n*) Função

Fatora o inteiro Gaussiano *n* sobre os inteiros Gaussianos, i.e., números da forma *a* `+` *b* `%i` onde *a* e *b* são inteiros raconais (i.e., inteiros comuns). Fatorações são normalizadas fazendo *a* e *b* não negativos.

gfactor (*expr*) Função

Fatora o polinômio *expr* sobre os inteiros de Gauss (isto é, os inteiros com a unidade imaginária `%i` adjunta). Isso é como `factor (`*expr*`, a^2+1)` trocando *a* por `%i`.

Exemplo:

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

gfactorsum (*expr*) Função

é similar a `factorsum` mas aplica `gfactor` em lugar de `factor`.

hipow (*expr*, *x*) Função

Retorna o maior expoente explícito de *x* em *expr*. *x* pode ser uma variável ou uma expressão geral. Se *x* não aparece em *expr*, `hipow` retorna `0`.

`hipow` não considera expressões equivalentes a `expr`. Em particular, `hipow` não expande `expr`, então `hipow (`*expr*`,` *x*`)` e `hipow (expand (`*expr*`,` *x*`))` podem retornar diferentes resultados.

Exemplos:

```
(%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 Variável de opção

Valor Padrão: 1000

`intfaclim` é o maior divisor que será tentado quando fatorando um grande número inteiro.

Quando `intfaclim` for `false` (esse é o caso quando o usuáro chama `factor` explicitamente), ou se o inteiro é um `fixnum` (i.e., ajustado em uma palavra de máquina), fatores de qualquer tamanho são considerados. `intfaclim` é escolhida para `false` fatores são calculados em `divsum`, `totient`, e `primep`.

Chamadas internas a `factor` respeitam o valor especificado pelo usuário para `intfaclim`. Escolhendo `intfaclim` para um pequeno valor podemos reduzir o tempo gasto fatorando grandes inteiros.

keepfloat Variável de opção

Valor Padrão: `false`

Quando `keepfloat` for `true`, evitamos que números em ponto flutuante sejam racionalizados quando expressões que os possuem são então convertidas para a forma de expressão racional canônica (CRE).

lratsubst (*L*, *expr*) Função

é análogo a `subst (`*L*`,` *expr*`)` exceto que esse usa `ratsubst` em lugar de `subst`.

O primeiro argumento de `lratsubst` é uma equação ou uma lista de equações idênticas em formato para que sejam aceitas por `subst`. As substituições são feitas na ordem dada pela lista de equações, isto é, da esquerda para a direita.

`load ("lrats")` chama `fullratsubst` e `lratsubst`.

Exemplos:

```
(%i1) load ("lrats")$
```

- `subst` pode realizar multiplas substituições. `lratsubst` é analoga a `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
```

- Se somente uma substituição for desejada, então uma equação simples pode ser dada como primeiro argumento.

```
(%i4) lratsubst (a^2 = b, a^3);
(%o4)                           a b
```

modulus Variável de opção

Valor Padrão: `false`

Quando `modulus` for um número positivo *p*, operações sobre os números racionais (como retornado por `rat` e funções relacionadas) são realizadas módulo *p*, usando o então chamado sistema de módulo "balanceado" no qual *n* `módulo` *p* é definido como um inteiro *k* em `[-(`*p*`-1)/2, ..., 0, ..., (`*p*`-1)/2]` quando *p* for ímpar, ou `[-(`*p*`/2 - 1), ..., 0, ...., `*p*`/2]` quando *p* for par, tal que *a* *p* `+` *k* seja igual a *n* para algum inteiro *a*.

Se *expr* já estiver na forma de expressão racional canônica (CRE) quando `modulus` for colocado em seu valor original, então você pode precisar repetir o rat *expr*, e.g., `expr: rat (ratdisrep (expr))`, com o objetivo de pegar resultados corretos.

Tipicamente `modulus` é escolhido para um número primo. Se `modulus` for escolhido para um inteiro não primo positivo, essa escolha é aceita, mas uma mensagem de alerta é mostrada. Maxima permitirá que zero ou um inteiro negativo seja atribuído a `modulus`, embora isso não seja limpo se aquele tiver quaisquer conseqüências úteis.

num (*expr*) Função

Retorna o numerador de *expr* se isso for uma razão. Se *expr* não for uma razão, *expr* é retornado.

`num` avalia seu argumento.

polydecomp (*p*, *x*) Função

Decompões o polinômio *p* na variável *x* em uma composição funcional de polinômios em *x*. `polydecomp` retorna uma lista `[`*p_1*`, ..., `*p_n*`]` tal que

```
lambda ([x], p_1) (lambda ([x], p_2) (... (lambda ([x], p_n) (x)) ...))
```

seja igual a *p*. O grau de *p_i* é maior que 1 para *i* menor que *n*.

Tal decomposição não é única.

Exemplos:

```
(%i1) polydecomp (x^210, x);
                              7   5   3   2
(%o1)                        [x , x , x , x ]
(%i2) p : expand (subst (x^3 - x - 1, x, x^2 - a));
                  6      4      3    2
```

```
(%o2)          x  - 2 x  - 2 x  + x  + 2 x - a + 1
(%i3) polydecomp (p, x);
                           2       3
(%o3)                 [x  - a, x  - x - 1]
```

As seguintes funções compõem `L = [e_1, ..., e_n]` como funções em `x`; essa funçào é a inversa de `polydecomp`:

```
compose (L, x) :=
  block ([r : x], for e in L do r : subst (e, x, r), r) $
```

Re-exprimindo o exemplo acima usando `compose`:

```
(%i3) polydecomp (compose ([x^2 - a, x^3 - x - 1], x), x);
                           2       3
(%o3)                 [x  - a, x  - x - 1]
```

Note que apesar de `compose (polydecomp (`*p*`, x), x)` sempre retornar *p* (não expandido), `polydecomp (compose ([`*p_1*`, ..., `*p_n*`], x), x)` *não* necessáriamente retorna `[`*p_1*`, ..., `*p_n*`]`:

```
(%i4) polydecomp (compose ([x^2 + 2*x + 3, x^2], x), x);
                             2       2
(%o4)                      [x  + 2, x  + 1]
(%i5) polydecomp (compose ([x^2 + x + 1, x^2 + x + 1], x), x);
                         2       2
                        x  + 3  x  + 5
(%o5)                  [------, ------, 2 x + 1]
                          4       2
```

quotient (*p_1, p_2*) Função
quotient (*p_1, p_2, x_1, ..., x_n*) Função

Retorna o polinômio *p_1* dividido pelo polinômio *p_2*. Os argumentos *x_1*, ..., *x_n* são interpretados como em `ratvars`.

`quotient` retorna o primeiro elemento de uma lista de dois elementos retornada por `divide`.

rat (*expr*) Função
rat (*expr, x_1, ..., x_n*) Função

Converte *expr* para a forma de expressão racional canônica (CRE) expandindo e combinando todos os termos sobre um denominador comum e cancelando para fora o máximo divisor comum entre o numerador e o denominador, também convertendo números em ponto flutuante para números racionais dentro da tolerância de `ratepsilon`. As variáveis são ordenadas de acordo com *x_1*, ..., *x_n*, se especificado, como em `ratvars`.

`rat` geralmente não simplifica funções outras que não sejam adição `+`, subtração `-`, multiplicação `*`, divisão `/`, e exponenciação com expoente inteiro, uma vez que `ratsimp` não manuseia esses casos. Note que átomos (números e variáveis) na forma CRE não são os mesmos que eles são na forma geral. Por exemplo, `rat(x)- x` retorna `rat(0)` que tem uma representação interna diferente de 0.

Quando `ratfac` for `true`, `rat` retorna uma forma parcialmente fatorada para CRE. Durante operações racionais a expressão é mantida como totalmente fatorada como

possível sem uma chamada ao pacote de fatoração (`factor`). Isso pode sempre economizar espaço de memória e algum tempo em algumas computações. O numerador e o denominador são ainda tidos como relativamente primos (e.g. `rat ((x^2 - 1)^4/(x + 1)^2)` retorna `(x - 1)^4 (x + 1)^2)`, mas os fatores dentro de cada parte podem não ser relativamente primos.

`ratprint` se `false` suprime a impressão de mensagens informando o usuário de conversões de números em ponto flutuante para números racionais.

`keepfloat` se `true` evita que números em ponto flutuante sejam convertidos para números racionais.

Veja também `ratexpand` e `ratsimp`.

Exemplos:

```
(%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 Variável de opção

Valor Padrão: `true`

Quando `ratalgdenom` for `true`, permite racionalização de denominadores com respeito a radicais tenham efeito. `ratalgdenom` tem efeito somente quando expressões racionais canônicas (CRE) forem usadas no modo algébrico.

ratcoef (*expr, x, n*) Função

ratcoef (*expr, x*) Função

Retorna o coeficiente da expressão *x^n* dentro da expressão *expr*. Se omitido, *n* é assumido ser 1.

O valor de retorno está livre (exceto possivelmente em um senso não racional) das variáveis em *x*. Se nenhum coeficiente desse tipo existe, 0 é retornado.

`ratcoef` expande e simplifica racionalmente seu primeiro argumento e dessa forma pode produzir respostas diferentes das de `coeff` que é puramente sintática. Dessa forma RATCOEF((X+1)/Y+X,X) retorna (Y+1)/Y ao passo que `coeff` retorna 1.

`ratcoef (`*expr*`, `*x*`, 0)`, visualiza *expr* como uma adição, retornando uma soma desses termos que não possuem *x*. portanto se *x* ocorre para quaisquer expoentes negativos, `ratcoef` pode não ser usado.

Uma vez que *expr* é racionalmente simplificada antes de ser examinada, coeficientes podem não aparecer inteiramente no caminho que eles foram pensados.

Exemplo:

```
(%i1) s: a*x + b*x + 5$
(%i2) ratcoef (s, a + b);
(%o2)                           x
```

ratdenom (*expr*) Função

Retorna o denominador de *expr*, após forçar a conversão de *expr* para expressão racional canônica (CRE). O valor de retorno é a CRE.

expr é forçada para uma CRE por `rat` se não for já uma CRE. Essa conversão pode mudar a forma de *expr* colocando todos os termos sobre um denominador comum.

`denom` é similar, mas retorna uma expressão comum em lugar de uma CRE. Também, `denom` não tenta colocar todos os termos sobre um denominador comum, e dessa forma algumas expressões que são consideradas razões por `ratdenom` não são consideradas razões por `denom`.

ratdenomdivide Variável de opção

Valor Padrão: `true`

Quando `ratdenomdivide` for `true`, `ratexpand` expande uma razão cujo o numerador for uma adição dentro de uma soma de razões, tendo todos um denominador comum. De outra forma, `ratexpand` colapsa uma adição de razões dentro de uma razão simples, cujo numerador seja a adição dos numeradores de cada razão.

Exemplos:

```
(%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*) Função

Realiza a derivação da expressão racional *expr* com relação a *x*. *expr* deve ser uma razão de polinômios ou um polinômio em *x*. O argumento *x* pode ser uma variável ou uma subexpressão de *expr*.

O resultado é equivalente a `diff`, embora talvez em uma forma diferente. `ratdiff` pode ser mais rápida que `diff`, para expressões racionais.

`ratdiff` retorna uma expressão racional canônica (CRE) se `expr` for uma CRE. De outra forma, `ratdiff` retorna uma expressão geral.

`ratdiff` considera somente as dependências de *expr* sobre *x*, e ignora quaisquer dependências estabelecidas por `depends`.

Exemplo:

```
(%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*) Função

Retorna seu argumento como uma expressão geral. Se *expr* for uma expressão geral, é retornada inalterada.

Tipicamente `ratdisrep` é chamada para converter uma expressão racional canônica (CRE) em uma expressão geral. Isso é algumas vezes conveniente se deseja-se parar o "contágio", ou caso se esteja usando funções racionais em contextos não racionais.

Veja também `totaldisrep`.

ratepsilon Variável de opção

Valor Padrão: 2.0e-8

`ratepsilon` é a tolerância usada em conversões de números em ponto flutuante para números racionais.

ratexpand (*expr*) Função

ratexpand Variável de opção

Expande *expr* multiplicando para fora produtos de somas e somas exponenciadas, combinando frações sobre um denominador comum, cancelando o máximo divisor comum entre entre o numerador e o denominador, então quebrando o numerador (se for uma soma) dentro de suas respectivas parcelas divididas pelo denominador.

O valor de retorno de `ratexpand` é uma expressão geral, mesmo se *expr* for uma expressão racional canônica (CRE).

O comutador `ratexpand` se `true` fará com que expressões CRE sejam completamente expandidas quando forem convertidas de volta para a forma geral ou mostradas, enquanto se for `false` então elas serão colocadas na forma recursiva. Veja também `ratsimp`.

Quando `ratdenomdivide` for `true`, `ratexpand` expande uma razão na qual o numerador é uma adição dentro de uma adição de razões, todas tendo um denominador comum. De outra forma, `ratexpand` contrai uma soma de razões em uma razão simples, cujo numerador é a soma dos numeradores de cada razão.

Quando `keepfloat` for `true`, evita que números em ponto flutuante sejam racionalizados quando expressões que contenham números em ponto flutuante forem convertidas para a forma de expressão racional canônica (CRE).

Exemplos:

```
(%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 Variável de opção

Valor Padrão: `false`

Quando `ratfac` for `true`, expressões racionais canônicas (CRE) são manipuladas na forma parcialmente fatorada.

Durante operações racionais a expressão é mantida como completamente fatorada como foi possível sem chamadas a `factor`. Isso pode sempre economizar espaço e pode economizar tempo em algumas computações. O numerador e o denominador são feitos relativamente primos, por exemplo `rat ((x^2 - 1)^4/(x + 1)^2)` retorna `(x - 1)^4 (x + 1)^2)`, mas o fator dentro de cada parte pode não ser relativamente primo.

No pacote `ctensr` (Manipulação de componentes de tensores), tensores de Ricci, Einstein, Riemann, e de Weyl e a curvatura escalar são fatorados automaticamente quando `ratfac` for `true`. `ratfac` *pode somente ser escolhido para casos onde as componentes tensoriais sejam sabidametne consistidas de poucos termos.*

Os esquemas de `ratfac` e de `ratweight` são incompatíveis e não podem ambos serem usados ao mesmo tempo.

ratnumer (*expr*) Função

Retorna o numerador de *expr*, após forçar *expr* para uma expressão racional canônica (CRE). O valor de retorno é uma CRE.

expr é forçada para uma CRE por `rat` se isso não for já uma CRE. Essa conversão pode alterar a forma de *expr* pela colocação de todos os termos sobre um denominador comum.

`num` é similar, mas retorna uma expressão comum em lugar de uma CRE. Também, `num` não tenta colocar todos os termos sobre um denominador comum, e dessa forma algumas expressões que são consideradas razões por `ratnumer` não são consideradas razões por `num`.

ratnump (*expr*) Função

Retorna `true` se *expr* for um inteiro literal ou razão de inteiros literais, de outra forma retorna `false`.

ratp (*expr*) Função

Retorna `true` se *expr* for uma expressão racional canônica (CRE) ou CRE extendida, de outra forma retorna `false`.

CRE são criadas por `rat` e funções relacionadas. CRE extendidas são criadas por `taylor` e funções relacionadas.

ratprint Variável de opção

Valor Padrão: `true`

Quando `ratprint` for `true`, uma mensagem informando ao usuário da conversão de números em ponto flutuante para números racionais é mostrada.

ratsimp (*expr*) Função
ratsimp (*expr, x_1, ..., x_n*) Função

Simplifica a expressão *expr* e todas as suas subexpressões, incluindo os argumentos para funções não racionais. O resultado é retornado como o quociente de dois polinômios na forma recursiva, isto é, os coeficientes de variável principal são polinômios em outras variáveis. Variáveis podem incluir funções não racionais (e.g., `sin (x^2 + 1)`) e os argumentos para quaisquer tais funções são também simplificados racionalmente.

`ratsimp` (*expr*, *x_1*, ..., *x_n*) habilita simplificação racional com a especiicação de variável ordenando como em `ratvars`.

Quando `ratsimpexpons` for `true`, `ratsimp` é aplicado para os expoentes de expressões durante a simplificação.

Veja também `ratexpand`. Note que `ratsimp` é afetado por algum dos sinalizadores que afetam `ratexpand`.

Exemplos:

```
(%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 Variável de opção

Valor Padrão: `false`

Quando `ratsimpexpons` for `true`, `ratsimp` é aplicado para os expoentes de expressões durante uma simplificação.

ratsubst (*a*, *b*, *c*) Função

Substitue *a* por *b* em *c* e retorna a expressão resultante. *b* pode também ser uma adição, produto, expoente, etc.

`ratsubst` sabe alguma coisa do significado de expressões uma vez que `subst` não é uma substituição puramente sintática. Dessa forma `subst (a, x + y, x + y + z)` retorna `x + y + z` ao passo que `ratsubst` retorna `z + a`.

Quando `radsubstflag` for `true`, `ratsubst` faz substituição de radicais em expressões que explicitamente não possuem esses radicais.

Exemplos:

```
(%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);
                                x
(%o6)
(%i7) radsubstflag: true$
(%i8) ratsubst (u, sqrt(x), x);
                                2
(%o8)                           u
```

ratvars (*x_1*, ..., *x_n*) Função

ratvars () Função

ratvars Variável de sistema

Declara variáveis principais *x_1*, ..., *x_n* para expressões racionais. *x_n*, se presente em uma expressão racional, é considerada a variável principal. De outra forma, *x_[n-1]* é considerada a variável principal se presente, e assim por diante até as variáveis precedentes para *x_1*, que é considerada a variável principal somente se nenhuma das variáveis que a sucedem estiver presente.

Se uma variável em uma expressão racional não está presente na lista `ratvars`, a ela é dada uma prioridade menor que *x_1*.

Os argumentos para `ratvars` podem ser ou variáveis ou funções não racionais tais como `sin(x)`.

A variável `ratvars` é uma lista de argumentos da função `ratvars` quando ela foi chamada mais recentemente. Cada chamada para a função `ratvars` sobre-grava a lista apagando seu conteúdo anterior. `ratvars ()` limpa a lista.

ratweight (*x_1, w_1, ..., x_n, w_n*) Função
ratweight () Função

Atribui um peso *w_i* para a variável *x_i*. Isso faz com que um termo seja substituído por 0 se seu peso exceder o valor da variável `ratwtlvl` (o padrão retorna sem truncação). O peso de um termo é a soma dos produtos dos pesos de uma variável no termo vezes seu expoente. Por exemplo, o peso de `3 x_1^2 x_2` é `2 w_1 + w_2`. A truncação de acordo com `ratwtlvl` é realizada somente quando multiplicando ou exponencializando expressões racionais canônicas (CRE).

`ratweight ()` retorna a lista cumulativa de atribuições de pesos.

Nota: Os esquemas de `ratfac` e `ratweight` são incompatíveis e não podem ambo serem usados ao mesmo tempo.

Exemplos:

```
(%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 Variável de sistema

Valor Padrão: `[]`

`ratweights` é a lista de pesos atribuídos por `ratweight`. A lista é cumulativa: cada chamada a `ratweight` coloca ítens adicionais na lista.

`kill (ratweights)` e `save (ratweights)` ambos trabalham como esperado.

ratwtlvl Variável de opção

Valor Padrão: `false`

`ratwtlvl` é usada em combinação com a função `ratweight` para controlar a truncação de expressão racionais canônicas (CRE). Para o valor padrão `false`, nenhuma truncação ocorre.

remainder (*p_1, p_2*) Função
remainder (*p_1, p_2, x_1, ..., x_n*) Função

Retorna o resto do polinômio *p_1* dividido pelo polinômio *p_2*. Os argumentos *x_1*, ..., *x_n* são interpretados como em `ratvars`.

`remainder` retorna o segundo elemento de uma lista de dois elementos retornada por `divide`.

resultant (*p_1, p_2, x*) Função
resultant Variável

Calcula o resultante de dois polinômios *p_1* e *p_2*, eliminando a variável *x*. O resultante é um determinante dos coeficientes de *x* em *p_1* e *p_2*, que é igual a zero se e somente se *p_1* e *p_2* tiverem um fator em comum não constante.

Se *p_1* ou *p_2* puderem ser fatorados, pode ser desejável chamar `factor` antes de chamar `resultant`.

A variável `resultant` controla que algorítmo será usado para calcular o resultante. `subres` para o prs subresultante, `mod` para o algorítmo resultante modular, e `red` para prs reduzido. Para muitos problemas `subres` pode ser melhor. Para alguns problemas com valores grandes de grau de uma única variável ou de duas variáveis `mod` pode ser melhor.

A função `bezout` pega os mesmos argumentos que `resultant` e retorna uma matriz. O determinante do valor de retorno é o resultante desejado.

savefactors Variável de opção

Valor Padrão: `false`

Quando `savefactors` for `true`, faz com que os fatores de uma expressão que é um produto de fatores sejam gravados por certas funções com o objetivo de aumentar a velocidade em posteriores fatorações de expressões contendo algum desses mesmos fatores.

sqfr (*expr*) Função

é similar a `factor` exceto que os fatores do polinômio são "livres de raízes". Isto é, eles possuem fatores somente de grau um. Esse algorítmo, que é também usado no primeiro estágio de `factor`, utiliza o fato que um polinômio tem em comum com sua n'ésima derivada todos os seus fatores de grau maior que n. Dessa forma pegando o maior divisor comum com o polinômio das derivadas com relação a cada variável no polinômio, todos os fatores de grau maior que 1 podem ser achados.

Exemplo:

```
(%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*) Função

tellrat () Função

Adiciona ao anel dos inteiros algébricos conhecidos do Maxima os elementos que são as soluções dos polinômios *p_1*, ..., *p_n*. Cada argumento *p_i* é um polinômio concoeficientes inteiros.

`tellrat (x)` efetivamente significa substituir 0 por *x* em funções racionais.

`tellrat ()` retorna uma lista das substituições correntes.

`algebraic` deve ser escolhida para `true` com o objetivo de que a simplificação de inteiros algébricos tenha efeito.

Maxima inicialmente sabe sobre a unidade imaginária `%i` e todas as raízes de inteiros.

Existe um comando `untellrat` que pega kernels (núcleos) e remove propriedades `tellrat`.

Quando fazemos `tellrat` em um polinômio de várias variáveis, e.g., `tellrat (x^2 - y^2)`, pode existir uma ambigüidade como para ou substituir *y*`^2` por *x*`^2` ou vice-versa. Maxima seleciona uma ordenação particular, mas se o usuário desejar especificar qual e.g. `tellrat (y^2 = x^2)` forneçe uma sintaxe que diga para substituir *y*`^2` por *x*`^2`.

Exemplos:

```
(%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*) Função

Converte toda subexpressão de *expr* da forma de expressão racionais canônicas (CRE) para a forma geral e retorna o resultado. Se *expr* é em sí mesma na forma CRE então `totaldisrep` é identica a `ratdisrep`.

`totaldisrep` pode ser usada para fazer um `ratdisrep` em expressões tais como equações, listas, matrizes, etc., que tiverem algumas subexpressões na forma CRE.

untellrat (*x_1*, ..., *x_n*) Função

Remove propriedades `tellrat` de *x_1*, ..., *x_n*.

13 Constantes

13.1 Definições para Constantes

%e Constante
- A base dos logarítmos naturais, e, é representada no Maxima como `%e`.

false Constante
- a contante Booleana, falso. (NIL em Lisp)

inf Constante
- infinito positivo real.

infinity Constante
- infinito complexo.

minf Constante
- menos infinito real.

%pi Constante
- "pi" é representado no Maxima como `%pi`.

true Constante
- a constante Booleana, verdadeiro. (T em Lisp)

14 Logarítmos

14.1 Definições para Logarítmos

%e_to_numlog Variável de opção

Valor padrão: `false`

Quando `true`, sendo `r` algum número racional, e `x` alguma expressão, `%e^(r*log(x))` irá ser simplificado em `x^r` . Note-se que o comando `radcan` também faz essa transformação, e transformações mais complicadas desse tipo também. O comando `logcontract` "contrai" expressões contendo `log`.

log (x) Função

Representa o logarítmo natural de x.

Simplificação e avaliação de logarítmos são governadas por muitos sinalizadores globais:

`logexpand` - faz com que `log(a^b)` torne-se `b*log(a)`. Se `logexpand` for escolhida para `all`, `log(a*b)` irá também simplificar para `log(a)+log(b)`. Se `logexpand` for escolhida para `super`, então `log(a/b)` irá também simplificar para `log(a)-log(b)` para números racionais `a/b`, `a#1`. (`log(1/b)`, para `b` inteiro, sempre simplifica). Se `logexpand` for escolhida para `false`, todas essas simplificações irão ser desabilitadas.

`logsimp` - se `false` então nenhuma simplificação de `%e` para um expoente contendo `log`'s é concluída.

`lognumer` - se `true` então argumentos negativos em ponto flutuante para `log` irá sempre ser convertido para seu valor absoluto antes que `log` seja tomado. Se `numer` for também `true`, então argumentos negativos inteiros para `log` irão também ser convertidos para seu valor absoluto.

`lognegint` - se `true` implementa a regra `log(-n) -> log(n)+%i*%pi` para `n` um inteiro positivo.

`%e_to_numlog` - quando `true`, `r` sendo algum número racional, e `x` alguma expressão, `%e^(r*log(x))` irá ser simplificado em `x^r` . Note-se que o comando `radcan` também faz essa transformação, e transformações mais complicadas desse tipo também. O comando `logcontract` "contrai" expressões contendo `log`.

logabs Variável de opção

Valor padrão: `false`

Quando fazendo integração indefinida onde logs são gerados, e.g. `integrate(1/x,x)`, a resposta é dada em termos de `log(abs(...))` se `logabs` for `true`, mas em termos de `log(...)` se `logabs` for `false`. Para integração definida, a escolha `logabs:true` é usada, porque aqui "avaliação" de integral indefinida nos extremos é muitas vezes necessária.

logarc Variável de opção

Valor padrão: `false`

Se `true` irá fazer com que as funções circularee inversas e hiperbólicas sejam convertidas em formas logarítimicas. `logarc(`*exp*`)` irá fazer com que essa conversão para uma expressão particular *exp* sem escolher o comutador ou tendo que re-avaliar a expressão com `ev`.

logconcoeffp Variável de opção

Valor padrão: `false`

Controla quais coeficientes são contraídos quando usando `logcontract`. Pode ser escolhida para o nome de uma função predicado de um argumento. E.g. se você gosta de gerar raízes quadradas, você pode fazer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) ou ratnump(m)$` . Então `logcontract(1/2*log(x));` irá fornecer `log(sqrt(x))`.

logcontract (*expr*) Função

Recursivamente examina a expressão *expr*, transformando subexpressões da forma `a1*log(b1) + a2*log(b2) + c` em `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 )
```

Se você faz `declare(n,integer);` então `logcontract(2*a*n*log(x));` fornece `a*log(x^(2*n))`. Os coeficientes que "contraem" dessa maneira são aqueles tais que 2 e `n` que satisfazem `featurep(coeff,integer)`. O usuário pode controlar quais coeficientes são contraídos escolhendo a opção `logconcoeffp` para o nome de uma função predicado de um argumento. E.g. se você gosta de gerara raízes quadradas, você pode fazer `logconcoeffp:'logconfun$ logconfun(m):=featurep(m,integer) ou ratnump(m)$` . então `logcontract(1/2*log(x));` irá fornecer `log(sqrt(x))`.

logexpand Variável de opção

Valor padrão: `true`

Faz com que `log(a^b)` torne-se `b*log(a)`. Se for escolhida para `all`, `log(a*b)` irá também simplificar para `log(a)+log(b)`. Se for escolhida para `super`, então `log(a/b)` irá também simplificar para `log(a)-log(b)` para números racionais `a/b`, `a#1`. (`log(1/b)`, para `b` inteiro, sempre simplifica). Se for escolhida para `false`, todas essas simplificações irão ser desabilitadas.

lognegint Variável de opção

Valor padrão: `false`

Se `true` implementa a regra `log(-n) -> log(n)+%i*%pi` para `n` um inteiro positivo.

lognumer Variável de opção

Valor padrão: `false`

Se `true` então argumentos negativos em ponto flutuante para `log` irão sempre ser convertidos para seus valores absolutos antes que o `log` seja tomado. Se `numer` for também `true`, então argumentos inteiros negativos para `log` irão também ser convertidos para seus valores absolutos.

logsimp Variável de opção

Valor padrão: `true`

Se `false` então nenhuma simplificação de `%e` para um expoente contendo `log`'s é concluída.

plog (x) Função

Representa o principal ramo logarítmos naturais avaliados para complexos com `-%pi < carg(`x`) <= +%pi` .

15 Trigonometria

15.1 Introdução ao Pacote Trigonométrico

Maxima tem muitas funções trigonométricas definidas. Não todas as identidades trigonometricas estão programadas, mas isso é possível para o usuário adicionar muitas delas usando a compatibilidade de correspondência de modelos do sistema. As funções trigonométricas definidas no Maxima são: `acos`, `acosh`, `acot`, `acoth`, `acsc`, `acsch`, `asec`, `asech`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `cot`, `coth`, `csc`, `csch`, `sec`, `sech`, `sin`, `sinh`, `tan`, e `tanh`. Existe uma coleção de comandos especialmente para manusear funções trigonométricas, veja `trigexpand`, `trigreduce`, e o comutador `trigsign`. Dois pacotes compartilhados extendem as regras de simplificação construídas no Maxima, `ntrig` e `atrig1`. Faça `describe(`*comando*`)` para detalhes.

15.2 Definições para Trigonometria

acos (x) Função
- Arco Cosseno.

acosh (x) Função
- Arco Cosseno Hiperbólico.

acot (x) Função
- Arco Cotangente.

acoth (x) Função
- Arco Cotangente Hiperbólico.

acsc (x) Função
- Arco Cossecante.

acsch (x) Função
- Arco Cossecante Hiperbólico.

asec (x) Função
- Arco Secante.

asech (x) Função
- Arco Secante Hiperbólico.

asin (x) Função
- Arco Seno.

asinh (x) Função
- Arco Seno Hiperbólico.

atan (x) Função
- Arco Tangente.

atan2 (y, x) Função
- retorna o valor de `atan(`y/x`)` no intervalo de `-%pi` a `%pi`.

atanh (x) Função
- Arco tangente Hiperbólico.

atrig1 Pacote
O pacote `atrig1` contém muitas regras adicionais de simplificação para funções trigonométricas inversas. Junto com regras já conhecidas para Maxima, os seguintes ângulos estão completamente implementados: `0`, `%pi/6`, `%pi/4`, `%pi/3`, and `%pi/2`. Os ângulos correspondentes nos outros três quadrantes estão também disponíveis. Faça `load(atrig1);` para usá-lo.

cos (x) Função
- Cosseno.

cosh (x) Função
- Cosseno hiperbólico.

cot (x) Função
- Cotangente.

coth (x) Função
- Cotangente Hyperbólica.

csc (x) Função
- Cossecante.

csch (x) Função
- Cossecante Hyperbólica.

halfangles Option variable
Default value: `false`

Quando `halfangles` for `true`, meios-ângulos são simplificados imediatamente.

ntrig Pacote
O pacote `ntrig` contém um conjunto de regras de simplificação que são usadas para simplificar função trigonométrica cujos argumentos estão na forma f`(`n `%pi/10)` onde f é qualquer das funções `sin`, `cos`, `tan`, `csc`, `sec` e `cot`.

sec (x) Função
- Secante.

sech (*x*) Função
- Secante Hyperbólica.

sin (*x*) Função
- Seno.

sinh (*x*) Função
- Seno Hyperbólico.

tan (*x*) Função
- Tangente.

tanh (*x*) Função
- Tangente Hyperbólica.

trigexpand (*expr*) Função

Expande funções trigonometricas e hyperbólicas de adições de ângulos e de ângulos multiplos que ocorram em *expr*. Para melhores resultados, *expr* deve ser expandida. Para intensificar o controle do usuário na simplificação, essa função expande somente um nível de cada vez, expandindo adições de ângulos ou ângulos multiplos. Para obter expansão completa dentro de senos e cossenos imediatamente, escolha o comutador `trigexpand: true`.

`trigexpand` é governada pelos seguintes sinalizadores globais:

`trigexpand`
Se `true` causa expansão de todas as expressões contendo senos e cossenos ocorrendo subseqüêntemente.

`halfangles`
Se `true` faz com que meios-ângulos sejam simplificados imediatamente.

`trigexpandplus`
Controla a regra "soma" para `trigexpand`, expansão de adições (e.g. `sin(x + y)`) terão lugar somente se `trigexpandplus` for `true`.

`trigexpandtimes`
Controla a regra "produto" para `trigexpand`, expansão de produtos (e.g. `sin(2 x)`) terão lugar somente se `trigexpandtimes` for `true`.

Exemplos:

```
(%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 Variável de opção

Valor padrão: `true`

`trigexpandplus` controla a regra da "soma" para `trigexpand`. Dessa forma, quando o comando `trigexpand` for usado ou o comutador `trigexpand` escolhido para `true`, expansão de adições (e.g. `sin(x+y)`) terão lugar somente se `trigexpandplus` for `true`.

trigexpandtimes Variável de opção

Valor padrão: `true`

`trigexpandtimes` controla a regra "produto" para `trigexpand`. Dessa forma, quando o comando `trigexpand` for usado ou o comutador `trigexpand` escolhido para `true`, expansão de produtos (e.g. `sin(2*x)`) terão lugar somente se `trigexpandtimes` for `true`.

triginverses Variável de opção

Valor padrão: `all`

`triginverses` controla a simplificação de composições de funções trigonométricas e hiperbólicas com suas funções inversas.

Se `all`, ambas e.g. `atan(tan(`*x*`))` e `tan(atan(`*x*`))` simplificarão para *x*.

Se `true`, a simplificação de *arcfun*`(`*fun*`(`*x*`))` é desabilitada.

Se `false`, ambas as simplificações *arcfun*`(`*fun*`(`*x*`))` e *fun*`(`*arcfun*`(`*x*`))` são desabilitadas.

trigreduce (*expr, x*) Função
trigreduce (*expr*) Função

Combina produtos e expoentes de senos e cossenso trigonométricos e hiperbólicos de *x* dentro daqueles de múltiplos de *x*. Também tenta eliminar essas funções quando elas ocorrerem em denominadores. Se *x* for omitido então todas as variáveis em *expr* são usadas.

Veja também `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
```

As rotinas de simplificação trigonométrica irão usar informações declaradas em alguns casos simples. Declarações sobre variáveis são usadas como segue, 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 Variável de opção

Valor padrão: `true`

Quando `trigsign` for `true`, permite simplificação de argumentos negativos para funções trigonométricas. E.g., `sin(-x)` transformar-se-á em `-sin(x)` somente se `trigsign` for `true`.

trigsimp (*expr*) Função

Utiliza as identidades $sin(x)^2 + cos(x)^2 = 1$ and $cosh(x)^2 - sinh(x)^2 = 1$ para simplificar expressões contendo `tan`, `sec`, etc., para `sin`, `cos`, `sinh`, `cosh`.

`trigreduce`, `ratsimp`, e `radcan` podem estar habilitadas a adicionar simplificações ao resultado.

`demo ("trgsmp.dem")` mostra alguns exemplos de `trigsimp`.

trigrat (*expr*) Função

Fornece uma forma quase-linear simplificada canônica de uma expressão trigonométrica; *expr* é uma fração racional de muitos `sin`, `cos` ou `tan`, os argumentos delas são formas lineares em algumas variáveis (ou kernels-núcleos) e `%pi`/n (n inteiro) com coeficientes inteiros. O resultado é uma fração simplificada com numerador e denominador ambos lineares em `sin` e `cos`. Dessa forma `trigrat` lineariza sempre quando isso for passível.

```
(%i1) trigrat(sin(3*a)/sin(a+%pi/3));
(%o1)            sqrt(3) sin(2 a) + cos(2 a) - 1
```

O seguinte exemplo encontra-se em Davenport, Siret, and Tournier, *Calcul Formel*, Masson (ou em inglês, Addison-Wesley), seção 1.5.5, teorema de Morley.

```
(%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 Funções Especiais

16.1 Introdução a Funções Especiais

16.2 specint

`hypgeo` é um pacote para manusear transformações de Laplace de funções especiais. `hyp` é um pacote para manusear funções Hipergeométricas generalizadas.

`specint` tenta calcular a integral definida (sobre o intervalo de zero a infinito) de uma expressão contendo funções especiais. Quando o integrando contém um fator `exp (-s t)`, o resultado é uma transformação de Laplace.

A sintaxe é como segue:

`specint (exp (-s*`*t*`) * `*expr*`, `*t*`);`

onde *t* é a variável de integração e *expr* é uma expressão contendo funções especiais.

Se `specint` não puder calcular a integral, o valor de retorno pode conter vários símbolos Lisp, incluindo `other-defint-to-follow-negtest`, `other-lt-exponential-to-follow`, `product-of-y-with-nofract-indices`, etc.; isso é um bug.

A notação de função especial segue adiante:

```
bessel_j (index, expr)          Função de Bessel, primeiro tipo
bessel_y (index, expr)          Função de Bessel, segundo tipo
bessel_i (index, expr)          Função de Bessel modificada, primeiro tipo
bessel_k (index, expr)          Função de Bessel modificada, segundo tipo
%he[n] (z)                      Polinmio de Hermite (Note bem: he, não h. Veja A&S 2
%p[u,v] (z)                     Função de Legendre
%q[u,v] (z)                     Função de Legendre, segundo tipo
hstruve[n] (z)                  Função H de Struve H
lstruve[n] (z)                  Função de L Struve
%f[p,q] ([], [], expr)          Função Hipergeométrica Generalizada
gamma()                         Função Gamma
gammagreek(a,z)                 Função gama incompleta
gammaincomplete(a,z)            Final da função gama incompleta
slommel
%m[u,k] (z)                     Função de Whittaker, primeiro tipo
%w[u,k] (z)                     Função de Whittaker, segundo tipo
erfc (z)                        Complemento da função erf (função de er-
ros - integral da distribuição  normal)
ei (z)                          Integral de exponencial (?)
kelliptic (z)                   integral eliptica completa de primeiro tipo (K)
%d [n] (z)                      Função cilíndrica parabólica
```

`demo ("hypgeo")` mostra muitos exemplos de transformações de Laplace calculadas através de `specint`.

Esse é um trabalho em andamento. Alguns nomes de funções podem mudar.

16.3 Definições para Funções Especiais

airy (x) Função

A função Aérea Ai. Se o argumento x for um número, o valor numérico de `airy (x)` é retornado. de outra forma, uma expressão não avaliada `airy (x)` é retornada.

A equação aérea `diff (y(x), x, 2) - x y(x) = 0` tem duas soluções linearmente independentes, chamadas `ai` e `bi`. Essa equação é muito popular como uma aproximação para problemas mais complicados em muitos ambientes de física matemática.

`load ("airy")` chama as funções `ai`, `bi`, `dai`, e `dbi`.

O pacote `airy` contém rotinas para calcular `ai` e `bi` e suas derivadas `dai` e `dbi`. O resultado é um número em ponto flutuante se o argumento for um número, e uma expressão não avaliada de outra forma.

Um erro ocorre se o argumento for maior que o esperado causando um estouro nas exponenciais, ou uma perda de precisão no `sin` ou no `cos`. Isso faz o intervalo de validade sobre -2800 a 10^38 para `ai` e `dai`, e de -2800 a 25 para `bi` e `dbi`.

Essas regras de derivação são conhecidas para Maxima:

- `diff (ai(x), x)` retorna `dai(x)`,
- `diff (dai(x), x)` retorna `x ai(x)`,
- `diff (bi(x), x)` retorna `dbi(x)`,
- `diff (dbi(x), x)` retorna `x bi(x)`.

Valores de função são calculados a partir das séries de Taylor convergentes para `abs(x) < 3`, e a partir de expansões assintóticas para x `< -3` ou x `> 3` como necessário. Esses resultados somente apresentam discrepncias numéricas muito pequenas em x `= 3` e x `= -3`. Para detalhes, veja Abramowitz e Stegun, *Handbook of Mathematical Functions*, Sessão 10.4 e Tabela 10.11.

`ev (taylor (ai(x), x, 0, 9), infeval)` retorna uma expansão de Taylor em ponto flutuante da função `ai`. Uma expressão similar pode ser construída para `bi`.

airy_ai (x) Função

A função Aérea Ai, como definida em Abramowitz e Stegun, *Handbook of Mathematical Functions*, Sessão 10.4.

A equação Aérea `diff (y(x), x, 2) - x y(x) = 0` tem duas soluções linearmente independentes, `y = Ai(x)` e `y = Bi(x)`. A derivada de `diff (airy_ai(x), x)` é `airy_dai(x)`.

Se o argumento `x` for um número real ou um número complexo qualquer deles em ponto flutuante , o valor numérico de `airy_ai` é retornado quando possível.

Veja também `airy_bi`, `airy_dai`, `airy_dbi`.

airy_dai (x) Função

A derivada da função Aérea Ai `airy_ai(x)`.

Veja `airy_ai`.

airy_bi (*x*) Função

A função Aérea Bi, como definida em Abramowitz e Stegun, *Handbook of Mathematical Functions*, Sessão 10.4, é a segunda solução da equação Aérea `diff (y(x), x, 2) - x y(x) = 0`.

Se o argumento `x` for um número real ou um número complexo qualquer deles em ponto flutuante, o valor numérico de `airy_bi` é retornado quando possível. Em outros casos a expressão não avaliada é retornada.

A derivada de `diff (airy_bi(x), x)` é `airy_dbi(x)`.

Veja `airy_ai`, `airy_dbi`.

airy_dbi (*x*) Função

A derivada de função Aérea Bi `airy_bi(x)`.

Veja `airy_ai` e `airy_bi`.

asympa Função

`asympa` é um pacote para análise assintótica. O pacote contém funções de simplificação para análise assintótica, incluindo as funções "grande O" e "pequeno o" que são largamente usadas em análises de complexidade e análise numérica.

`load ("asympa")` chama esse pacote.

bessel (*z*, *a*) Função

A função de Bessel de primeiro tipo.

Essa função está desatualizada. Escreva `bessel_j (z, a)` em lugar dessa.

bessel_j (*v*, *z*) Função

A função de Bessel do primeiro tipo de ordem v e argumento z.

`bessel_j` calcula o array `besselarray` tal que `besselarray [i] = bessel_j [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_j` é definida como

$$\sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{v+2\,k}}{k!\,\Gamma\left(v+k+1\right)}$$

todavia séries infinitas não são usadas nos cálculos.

bessel_y (*v*, *z*) Função

A função de Bessel do segundo tipo de ordem v e argumento z.

`bessel_y` calcula o array `besselarray` tal que `besselarray [i] = bessel_y [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_y` é definida como

$$\frac{\cos\left(\pi\,v\right)\,J_v(z) - J_{-v}(z)}{\sin\left(\pi\,v\right)}$$

quando v não for um inteiro. Quando v for um inteiro n, o limite com v aprocimando-se de n é tomado.

bessel_i (*v*, *z*) Função

A função de Bessel modificada de primeiro tipo de ordem v e argumento z.

`bessel_i` calcula o array `besselarray` tal que `besselarray [i] = bessel_i [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_i` é definida como

$$\sum_{k=0}^{\infty} \frac{1}{k!\,\Gamma\left(v+k+1\right)} \left(\frac{z}{2}\right)^{v+2\,k}$$

todavia séries infinitas não são usadas nos cálculos.

bessel_k (*v*, *z*) Função

A função de Bessel modificada de segundo tipo de ordem v e argumento z.

`bessel_k` calcula o array `besselarray` tal que `besselarray [i] = bessel_k [i + v - int(v)] (z)` para `i` de zero a `int(v)`.

`bessel_k` é definida como

$$\frac{\pi\,\csc\left(\pi\,v\right)\,\left(I_{-v}(z)-I_{v}(z)\right)}{2}$$

quando v não for inteiro. Se v for um inteiro n, então o limite com v aproximando-se de n é tomado.

besselexpand Variável de opção

Valor padrão: `false`

Expansões de controle de funções de Bessel quando a ordem for a metade de um inteiro ímpar. Nesse caso, as funções de Bessel podem ser expandidas em termos de outras funções elementares. Quando `besselexpand` for `true`, a função de Bessel é expandida.

```
(%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*) Função

A função de Bessel de primeiro tipo de ordem 0.

Essa função está desatualizada. Escreva `bessel_j (0, x)` em lugar dessa função.

j1 (*x*) Função

A função de Bessel de primeiro tipo de ordem 1.

Essa função está desatualizada. Escreva `bessel_j (1, x)` em lugar dessa função.

jn (*x*, *n*) Função

A função de Bessel de primeiro tipo de ordem *n*.

Essa função éstá desatualizada. Escreva `bessel_j (n, x)` em lugar dessa função.

i0 (*x*) Função

A função de Bessel modificada de primeiro tipo de ordem 0.

Essa função éstá desatualizada. Escreva `bessel_i (0, x)` em lugar dessa função.

i1 (*x*) Função

A função de Bessel modificada de primeiro tipo de ordem 1.

Essa função está desatualizada. Escreva `bessel_i (1, x)` em lugar dessa função.

beta (*x*, *y*) Função

A função beta, definida como `gamma(x) gamma(y)/gamma(x + y)`.

gamma (*x*) Função

A função gama.

Veja também `makegamma`.

A variável `gammalim` controla a simplificação da função gama.

A constante de Euler-Mascheroni é `%gamma`.

gammalim Variável de opção

Valor padrão: 1000000

`gammalim` controla a simplificação da função gama para integral e argumentos na forma de números racionais. Se o valor absoluto do argumento não for maior que `gammalim`, então a simplificação ocorrerá. Note que `factlim` comuta controle de simplificaçcão do resultado de `gamma` de um argumento inteiro também.

intopois (*a*) Função

Converte *a* em um código de Poisson.

makefact (*expr*) Função

Transforma instncias de funções binomiais, gama, e beta em *expr* para fatoriais.

Veja também `makegamma`.

makegamma (*expr*) Função

Transforma instncias de funções binomiais, fatorial, e beta em *expr* para funções gama.

Veja também `makefact`.

numfactor (*expr*) Função

Retorna o fator numérico multiplicando a expressão *expr*, que pode ser um termo simples.

`content` retorna o máximo divisor comum (mdc) de todos os termos em uma adição.

```
(%i1) gamma (7/2);
                                15 sqrt(%pi)
(%o1)                           ------------
                                     8
(%i2) numfactor (%);
                                     15
(%o2)                                --
                                     8
```

outofpois (*a*) Função

Converte *a* de um código de Poisson para uma representação geral. Se *a* não for uma forma de Poisson, `outofpois` realiza a conversão, i.e., o valor de retorno é `outofpois (intopois (a))`. Essa função é desse modo um simplificador cannico para adições e potências de termos de seno e cosseno de um tipo particular.

poisdiff (*a*, *b*) Função

Deriva *a* com relação a *b*. *b* deve ocorrer somente nos argumentos trigonométricos ou somente nos coeficientes.

poisexpt (*a*, *b*) Função

Funcionalmente identica a `intopois (a^b)`. *b* deve ser um inteiro positico.

poisint (*a*, *b*) Função

Integra em um senso restrito similarmente (para `poisdiff`). Termos não periódicos em *b* são diminuídos se *b* estiver em argumentos trigonométricos.

poislim Option variable

Valor padrão: 5

`poislim` determina o domínio dos coeficientes nos argumentos de funções trigonométricas. O valor inicial de 5 corresponde ao intervalo [-2^(5-1)+1,2^(5-1)], ou [-15,16], mas isso pode ser alterado para [-2^(n-1)+1, 2^(n-1)].

poismap (*series*, *sinfn*, *cosfn*) Função

mapeará as funções *sinfn* sobre os termos de seno e *cosfn* ssobre os termos de cosseno das séries de Poisson dadas. *sinfn* e *cosfn* são funções de dois argumentos que são um coeficiente e uma parte trigonométrica de um termo em séries respectivamente.

poisplus (*a*, *b*) Função

É funcionalmente identica a `intopois (a + b)`.

poissimp (*a*) Função

Converte *a* em séries de Poisson para *a* em representação geral.

poisson Símbolo especial
O símbolo `/P/` segue o rótulo de linha de uma expressão contendo séries de Poisson.

poissubst (*a*, *b*, *c*) Função
Substitue *a* por *b* em *c*. *c* é uma série de Poisson.

(1) Quando *B* é uma variável *u*, *v*, *w*, *x*, *y*, ou *z*, então *a* deve ser uma expressão linear nessas variáveis (e.g., `6*u + 4*v`).

(2) Quando *b* for outra que não essas variáveis, então *a* deve também ser livre dessas variáveis, e alé disso, livre de senos ou cossenos.

`poissubst (a, b, c, d, n)` é um tipo especial d substituição que opera sobre *a* e *b* como no tipo (1) acima, mas onde *d* é uma série de Poisson, expande `cos(d)` e `sin(d)` para a ordem *n* como provendo o resultado da substituição *a* `+` *d* por *b* em *c*. A idéia é que *d* é uma expansão em termos de um pequeno parmetro. Por exemplo, `poissubst (u, v, cos(v), %e, 3)` retorna `cos(u)*(1 - %e^2/2) - sin(u)*(%e - %e^3/6)`.

poistimes (*a*, *b*) Função
É funcionalmente idêntica a `intopois (a*b)`.

poistrim () Função
é um nome de função reservado que (se o usuário tiver definido uma função com esse nome) é aplicada durante multiplicação de Poisson. Isso é uma função predicada de 6 argumentos que são os coeficientes de *u*, *v*, ..., *z* em um termo. Termos para os quais `poistrim` for `true` (para os coeficientes daquele termo) são eliminados durante a multiplicação.

printpois (*a*) Função
Mostra uma série de Poisson em um formato legível. Em comum com `outofpois`, essa função converterá *a* em um código de Poisson primeiro, se necessário.

psi (*x*) Função
psi [*n*](*x*) Função
A derivada de `log (gamma (x))`.

Maxima não sabe como calcular um valor numérico de `psi`. Todavia, a função `bfpsi` no pacote `bffac` pode calcular valores numéricos.

17 Polinômios Ortogonais

17.1 Introdução a Polinômios Ortogonais

O pacote `specfun` contém Código Maxima para a avaliação de todos os polinômios ortogonais listados no Capítulo 22 de Abramowitz e Stegun. Esses incluem polinômios de Chebyshev, Laguerre, Hermite, Jacobi, Legendre, e ultraesférico (Gegenbauer). Adicionalmente, `specfun` contém códigos para funções de Bessel esféricas, funções de Hankel esféricas, e funções harmônicas esféricas. O pacote `specfun` não é parte do Maxima propriamente; ele é chamado por meio de requisição de usuário via `load` ou automaticamente via o sistema `autoload`.

A seguinte tabela lista cada função em `specfun`. `specfun` é um nome Maxima, restrições sobre seus argumentos, e uma referência para o algorítmo que `specfun` usa para avaliar isso. Com poucas excessões, `specfun` segue as convenções de Abramowitz e Stegun. Em todos os casos, m e n devem ser inteiros.

A&S refere-se a Abramowitz e Stegun, *Handbook of Mathematical Functions* (10th edição, Dezembro de 1972), G&R refere-se a Gradshteyn e Ryzhik, *Table of Integrals, Series, and Products* (1980 edição corrigida e ampliada), e Merzbacher refere-se a *Quantum Mechanics* (segunda edição, 1970).

Função	*Nome Maxima*	*Restrições*	*Referência(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 página 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 página 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)	$n > -1, \lvert m\rvert <= n$	Merzbacher 9.64
ultraspherical (Gegenbauer)	ultraspherical(n,a,x)	$n > -1$	A&S 22.5.27

O pacote `specfun` é primáriamente planejado para computação simbólica. A computação simbólica espera que `specfun` forneça resultados acurados em ponto flutuante também; todavia, nenhuma reinvidicação foi feita para que os algorítmos sejam também adequados para avaliação numérica. Algum esforço, todavia, tem sido feito para fornecer boa performace

numérica. Quando todos os argumentos, exceto pela ordem, forem números em ponto flutuante (mas não grandes números em ponto flutuante), muitas funções em `specfun` chamam uma versão mododeclarada da função de Jacobi. Isso aumenta grandemente a velocidade de avaliação de números em ponto flutuante de polinômios ortogonais.

`specfun` manuseia muitos domínios de erro através do retorno de uma função não avaliada. Nenhuma regra de simplificação (baseada em relações recursivas) é definida para funções não avaliadas. Isso é possível para uma expressão envolvendo adições de funções especiais não avaliadas para tender para zero, ainda Maxima é incapaz de reduzir essas funções a zero.

`load ("specfun")` chama o pacote `specfun`. Alternativamente, `setup_autoload` faz com que o pacote seja chamado quando uma das funções `specfun` aparecer em uma expressão. `setup_autoload` pode aparecer na linha de comando ou no arquivo `maxima-init.mac`. Veja `setup_autoload`.

Um exemplo de uso de `specfun` é

```
(%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)
```

Geralmente, código compilado executa mais rápidamente que o código traduzido; todavia, código traduzido pode ser melhor para desenvolvimento de programas.

Algumas funções (a saber `jacobi_p`, `ultraspherical`, `chebyshev_t`, `chebyshev_u` e `legendre_p`), retornam um representação de série quando a ordem for um inteiro simbólico. A representação de série não é usada por `specfun` para qualquer computação, mas essa representação de série pode ser simplificada pelo Maxima automaticamente, ou a simplificação pode ser possível para usar as séries para avaliar a função através de manipulações adicionais. Por exemplo:

```
(%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 )
```

O primeiro e o último termos da adição são adicionados fora do somatório. Removendo esses dois termos evita-se erros do Maxima associados com termos 0^0 em uma adição que pode avaliar para 1, mas avaliada para 0 em um somatório no Maxima. Porque a soma de índices vai de 1 a $n-1$, o menor índice de soma excederá o maior índice de soma quando $n = 0$; escolhendo `sumhack` para `true` temos uma correção. Por exemplo:

```
(%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
```

Muitas funções em `specfun` possuem uma propriedade `gradef`; derivadas com relação a ordem ou outros parmetros de funções são indefinidas, e uma tentativa de calcular tal uma derivada retorna como resiltado uma mensagem de erro.

O pacote `specfun` e sua documentação foram escritos por Barton Willis da Universidade de Nebraska em Kearney. Foi liberado sob os termos da Licença Pública Geral (GPL). Envie relatórios de erro e comentários sobre esse pacote para willisb@unk.edu. Em seu relatório, por favor inclua a versão do Maxima, como reportado por `build_info()`, e a versão de `specfun`, como reportado por `get ('specfun, 'version)`.

17.2 Definições para Polinômios Ortogonais

assoc_legendre_p (*n*, *m*, *x*) Função

Retorna a função associada de Legendre de primeiro tipo para inteiros $n > -1$ e $m > -1$. Quando $|m| > n$ e $n >= 0$, teremos $assoc_legendre_p(n, m, x) = 0$. Referência: A&S 22.5.37 página 779, A&S 8.6.6 (segunda equação) página 334, e A&S 8.2.5 página 333.

`load ("specfun")` chama essa função.

Veja [assoc_legendre_q], página 170, [legendre_p], página 171, e [legendre_q], página 171.

assoc_legendre_q (*n*, *m*, *x*) Função

Retorna a função associada de Legendre de segundo tipo para inteiros $n > -1$ e $m > -1$.

Referência: Gradshteyn e Ryzhik 8.706 página 1000.

`load ("specfun")` chama essa função.

Veja também [assoc_legendre_p], página 169, [legendre_p], página 171, e [legendre_q], página 171.

chebyshev_t (*n*, *x*) Função

Retorna a função de Chebyshev de primeiro tipo para inteiros $n > -1$.

Referência: A&S 22.5.31 página 778 e A&S 6.1.22 página 256.

`load ("specfun")` chama essa função.

Veja também [chebyshev_u], página 170.

chebyshev_u (*n*, *x*) Função

Retorna a função de Chebyshev de segundo tipo para inteiros $n > -1$.

Referência A&S, 22.8.3 página 783 e A&S 6.1.22 página 256.

`load ("specfun")` chama essa função.

Veja também [chebyshev_t], página 170.

gen_laguerre (*n*, *a*, *x*) Função

Retorna o polinômio generalizado de Laguerre para inteiros $n > -1$.

`load ("specfun")` chama essa função.

Referência tabela na página 789 em A&S.

hermite (*n*, *x*) Função

Retorna o polinômio de Hermite para inteiros $n > -1$.

`load ("specfun")` chama essa função.

Referência A&S 22.5.40 e 22.5.41, página 779.

jacobi_p (*n*, *a*, *b*, *x*) Função

Retorna o polinômio de Jacobi para inteiros $n > -1$ e a e b simbólicos ou $a > -1$ e $b > -1$. (Os polinômios de Jacobi são atualmente definidos para todos a e b; todavia, o polinômio de Jacobi peso $(1-x)^a(1+x)^b$ não é integrável para $a <= -1$ ou para $b <= -1$.)

Quando a, b, e x forem números em ponto flutuante (mas não grandes números em ponto flutuante) `specfun` chama uma versão especial modo declarada de $jacobi_p$. Para valore numéricos, a versão modo declarada é mais rápida que a outra versão.

Muitas funções em `specfun` são calculados como um caso especial dos polinômios de Jacobi; Eles também desfrutam do impulso de velocidade da versão modo declarada de *jacobi*.

Se n tiver sido declarado para ser um inteiro, $jacobi_p(n, a, b, x)$ retorna uma representação de somatório para a função de Jacobi. Porque Maxima simplifica 0^0 para 0 em uma adição, dois termos da adição são adicionados fora do somatório.

`load ("specfun")` chama essa função.

Referência tabela na página 789 em A&S.

laguerre (*n*, *x*) Função

Retorna o polinômio de Laguerre para inteiros $n > -1$.

Referência A&S 22.5.16, página 778 e A&S página 789.

`load ("specfun")` chama essa função.

Veja também [gen_laguerre], página 170.

legendre_p (*n*, *x*) Função

Retorna o polinômio de Legendre de primeiro tipo para inteiros $n > -1$.

Referência A&S 22.5.35 página 779.

`load ("specfun")` chama essa função.

Veja [legendre_q], página 171.

legendre_q (*n*, *x*) Função

Retorna o polinômio de Legendre de primeiro tipo para inteiros $n > -1$.

Referência A&S 8.6.19 página 334.

`load ("specfun")` chama essa função.

Veja também [legendre_p], página 171.

spherical_bessel_j (*n*, *x*) Função

Retorna a função de Bessel esférica do primeiro tipo para inteiros $n > -1$.

Referência A&S 10.1.8 página 437 e A&S 10.1.15 página 439.

`load ("specfun")` chama essa função.

Veja também [spherical_hankel1], página 172, [spherical_hankel2], página 172, e [spherical_bessel_y], página 171.

spherical_bessel_y (*n*, *x*) Função

Retorna a função de Bessel esférica do segundo tipo para inteiros $n > -1$.

Referência A&S 10.1.9 página 437 e 10.1.15 página 439.

`load ("specfun")` chama essa função.

Veja também [spherical_hankel1], página 172, [spherical_hankel2], página 172, e [spherical_bessel_y], página 171.

spherical_hankel1 (*n*, *x*) Função

Retorna a função de Hankel esférica do primeiro tipo para inteiros $n > -1$.

Referência A&S 10.1.36 página 439.

`load ("specfun")` chama essa função.

Veja também [spherical_hankel2], página 172, [spherical_bessel_j], página 171, e [spherical_bessel_y], página 171.

spherical_hankel2 (*n*, *x*) Função

Retorna a função de Hankel esférica do segundo tipo para inteiros $n > -1$.

Referência A&S 10.1.17 página 439.

`load ("specfun")` chama essa função.

Veja também [spherical_hankel1], página 172, [spherical_bessel_j], página 171, e [spherical_bessel_y], página 171.

spherical_harmonic (*n*, *m*, *x*, *y*) Função

Retorna a função harmônica esférica para inteiros $n > -1$ e $|m| <= n$.

Referência Merzbacher 9.64.

`load ("specfun")` chama essa função.

Veja também [assoc_legendre_p], página 169.

ultraspherical (*n*, *a*, *x*) Função

Retorna os polinômios ultraesféricos para inteiros $n > -1$. Os polinômios ultraesféricos são também conhecidos com polinômios de Gegenbauer.

Referência A&S 22.5.27

`load ("specfun")` chama essa função.

Veja também [jacobi_p], página 170.

18 Funções Elípticas

18.1 Introdução a Funções Elípticas e Integrais

Maxima inclui suporte a funções elípticas Jacobianas e a integrais elípticas completas e incompletas. Isso inclui manipulação simbólica dessas funções e avaliação numérica também. Definições dessas funções e muitas de suas propriedades podem ser encontradas em Abramowitz e Stegun, Capítulos 16–17. Tanto quanto possível, usamos as definições e relações dadas aí.

Em particular, todas as funções elípticas e integrais elípticas usam o parâmetro m em lugar de módulo k ou o ângulo modular α. Isso é uma área onde discordamos de Abramowitz e Stegun que usam o ângulo modular para as funções elípticas. As seguintes relações são verdadeiras:

$$m = k^2$$

and

$$k = \sin\alpha$$

As funções elípticas e integrais elípticas estão primariamente tencionando suportar computação simbólica. Portanto, a maiora das derivadas de funções e integrais são conhecidas. Todavia, se valores em ponto flutuante forem dados, um resultado em ponto flutuante é retornado.

Suporte para a maioria de outras propriedades das funções elípticas e integrais elípticas além das derivadas não foram ainda escritas.

Alguns exemplos de funções elípticas:

```
(%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)
```

Alguns exemplos de integrais elípticas:

```
(%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))
```

Suporte a funções elípticas e integrais elípticas foi escrito por Raymond Toy. Foi colocado sob os termos da Licençã Pública Geral (GPL) que governa a distribuição do Maxima.

18.2 Definições para Funções Elípticas

jacobi_sn (u, m) Função

A Função elíptica Jacobiana $sn(u, m)$.

jacobi_cn (u, m) Função

A função elíptica Jacobiana $cn(u, m)$.

jacobi_dn (*u*, *m*) Função
A função elíptica Jacobiana $dn(u, m)$.

jacobi_ns (*u*, *m*) Função
A função elíptica Jacobiana $ns(u, m) = 1/sn(u, m)$.

jacobi_sc (*u*, *m*) Função
A função elíptica Jacobiana $sc(u, m) = sn(u, m)/cn(u, m)$.

jacobi_sd (*u*, *m*) Função
A função elíptica Jacobiana $sd(u, m) = sn(u, m)/dn(u, m)$.

jacobi_nc (*u*, *m*) Função
A função elíptica Jacobiana $nc(u, m) = 1/cn(u, m)$.

jacobi_cs (*u*, *m*) Função
A função elíptica Jacobiana $cs(u, m) = cn(u, m)/sn(u, m)$.

jacobi_cd (*u*, *m*) Função
A função elíptica Jacobiana $cd(u, m) = cn(u, m)/dn(u, m)$.

jacobi_nd (*u*, *m*) Função
A função elíptica Jacobiana $nc(u, m) = 1/cn(u, m)$.

jacobi_ds (*u*, *m*) Função
A função elíptica Jacobiana $ds(u, m) = dn(u, m)/sn(u, m)$.

jacobi_dc (*u*, *m*) Função
A função elíptica Jacobiana $dc(u, m) = dn(u, m)/cn(u, m)$.

inverse_jacobi_sn (*u*, *m*) Função
A inversa da função elíptica Jacobiana $sn(u, m)$.

inverse_jacobi_cn (*u*, *m*) Função
A inversa da função elíptica Jacobiana $cn(u, m)$.

inverse_jacobi_dn (*u*, *m*) Função
A inversa da função elíptica Jacobiana $dn(u, m)$.

inverse_jacobi_ns (*u*, *m*) Função
A inversa da função elíptica Jacobiana $ns(u, m)$.

inverse_jacobi_sc (*u*, *m*) Função
A inversa da função elíptica Jacobiana $sc(u, m)$.

inverse_jacobi_sd (*u*, *m*) Função
A inversa da função elíptica Jacobiana $sd(u, m)$.

inverse_jacobi_nc (*u*, *m*) Função
A inversa da função elíptica Jacobiana $nc(u, m)$.

inverse_jacobi_cs (*u*, *m*) Função
A inversa da função elíptica Jacobiana $cs(u, m)$.

inverse_jacobi_cd (*u*, *m*) Função
A inversa da função elíptica Jacobiana $cd(u, m)$.

inverse_jacobi_nd (*u*, *m*) Função
A inversa da função elíptica Jacobiana $nc(u, m)$.

inverse_jacobi_ds (*u*, *m*) Função
A inversa da função elíptica Jacobiana $ds(u, m)$.

inverse_jacobi_dc (*u*, *m*) Função
A inversa da função elíptica Jacobiana $dc(u, m)$.

18.3 Definições para Integrais Elípticas

elliptic_f (*phi*, *m*) Função
A integral elíptica incompleta de primeiro tipo, definida como

$$\int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Veja também [elliptic_e], página 176 e [elliptic_kc], página 177.

elliptic_e (*phi*, *m*) Função
A integral elíptica incompleta de segundo tipo, definida como

$$\int_0^{\phi} \sqrt{1 - m\sin^2\theta} d\theta$$

Veja também [elliptic_e], página 176 and [elliptic_ec], página 177.

elliptic_eu (*u*, *m*) Função
A integral elíptica incompleta de segundo tipo, definida como

$$\int_0^{u} \mathrm{dn}(v, m) dv = \int_0^{\tau} \sqrt{\frac{1 - mt^2}{1 - t^2}} dt$$

onde $\tau = \mathrm{sn}(u, m)$
Isso é relacionado a $elliptic_e$ através de

$$E(u, m) = E(\phi, m)$$

onde $\phi = \sin^{-1}\mathrm{sn}(u, m)$ Veja também [elliptic_e], página 176.

elliptic_pi (*n*, *phi*, *m*) Função

A integral elíptica incompleta de terceiro tipo, definida como

$$\int_0^{\phi} \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - m\sin^2\theta}}$$

Somente a derivada em relação a *phi* é conhecida pelo Maxima.

elliptic_kc (*m*) Função

A integral elíptica completa de primeiro tipo, definida como

$$\int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Para certos valores de m, o valor da integral é conhecido em termos de funções *Gamma*. Use `makegamma` para avaliar esse valor.

elliptic_ec (*m*) Função

A integral elíptica completa de sgundo tipo, definida como

$$\int_0^{\frac{\pi}{2}} \sqrt{1 - m\sin^2\theta} d\theta$$

Para certos valores de m, o valor da integral é conhecido em termos de funçõesv*Gamma*. Use `makegamma` para avaliar esse valor.

19 Limites

19.1 Definições para Limites

lhospitallim Variável de Opção

Valor padrão: 4

`lhospitallim` é o máximo número de vezes que a regra L'Hospital é usada em `limit`. Isso evita ciclos infinitos em casos como `limit (cot(x)/csc(x), x, 0)`.

limit (*expr, x, val, dir*) Função
limit (*expr, x, val*) Função
limit (*expr*) Função

Calcula o limite de *expr* com a variável real *x* aproximando-se do valor *val* pela direção *dir*. *dir* pode ter o valor `plus` para um limite pela direita, `minus` para um limite pela esquerda, ou pode ser omitido (implicando em um limite em ambos os lados é para ser computado).

`limit` usa os seguintes símbolos especiais: `inf` (infinito positivo) e `minf` (infinito negativo). Em saídas essa função pode também usar `und` (undefined - não definido), `ind` (indefinido mas associado) e `infinity` (infinito complexo).

`lhospitallim` é o máximo número de vezes que a regra L'Hospital é usada em `limit`. Isso evita ciclos infinitos em casos como `limit (cot(x)/csc(x), x, 0)`.

`tlimswitch` quando `true` fará o pacote `limit` usar série de Taylor quando possível.

`limsubst` evita que `limit` tente substituições sobre formas desconhecidas. Isso é para evitar erros como `limit (f(n)/f(n+1), n, inf)` dando igual a 1. Escolhendo `limsubst` para `true` permitirá tais substituições.

`limit` com um argumento é muitas vezes chamado em ocasiões para simplificar expressões de constantes, por exemplo, `limit (inf-1)`.

`example (limit)` mostra alguns exemplos.

Para saber sobre o método utilizado veja Wang, P., "Evaluation of Definite Integrals by Symbolic Manipulation", tese de Ph.D., MAC TR-92, Outubro de 1971.

limsubst Variável de Opção

valor padrão: `false` - evita que `limit` tente substituições sobre formas desconhecidas. Isso é para evitar erros como `limit (f(n)/f(n+1), n, inf)` dando igual a 1. Escolhendo `limsubst` para `true` permitirá tais substituições.

tlimit (*expr, x, val, dir*) Função
tlimit (*expr, x, val*) Função
tlimit (*expr*) Função

Retorna `limit` com `tlimswitch` escolhido para `true`.

tlimswitch Variável de Opção

Valor padrão: `false`

Quando `tlimswitch` for `true`, fará o pacote `limit` usar série de Taylor quando possível.

20 Diferenciação

20.1 Definições para Diferenciação

antid (*expr*, *x*, *u*(*x*)) Função

Retorna uma lista de dois elementos, tais que uma antiderivada de *expr* com relação a *x* pode ser constuída a partir da lista. A expressão *expr* pode conter uma função desconhecida *u* e suas derivadas.

Tome *L*, uma lista de dois elementos, como sendo o valor de retorno de `antid`. Então *L*`[1] + 'integrate (`*L*`[2],` *x*`)` é uma antiderivada de *expr* com relação a *x*.

Quando `antid` obtém sucesso inteiramente, o segundo elemento do valor de retorno é zero. De outra forma, o segundo elemento é não zero, e o primeiro elemento não zero ou zero. Se `antid` não pode fazer nenhum progresso, o primeiro elemento é zero e o segundo não zero.

`load ("antid")` chama essa função. O pacote `antid` também define as funções `nonzeroandfreeof` e `linear`.

`antid` está relacionada a `antidiff` como segue. Tome *L*, uma lista de dois elementos, que é o valor de retorno de `antid`. Então o valor de retorno de `antidiff` é igual a *L*`[1] + 'integrate (`*L*`[2],` *x*`)` onde *x* é a variável de integração.

Exemplos:

```
(%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*)) Função

Retorna uma antiderivada de *expr* com relação a *x*. A expressão *expr* pode conter uma função desconhecida *u* e suas derivadas.

Quando `antidiff` obtém sucesso inteiramente, a expressão resultante é livre do sinal de integral (isto é, livre do substantivo `integrate`). De outra forma, `antidiff` retorna uma expressão que é parcialmente ou inteiramente dentro de um sinal de um sinal de integral. Se `antidiff` não pode fazer qualquer progresso, o valor de retorno é inteiramente dentro de um sinal de integral.

`load ("antid")` chama essa função. O pacote `antid` também define as funções `nonzeroandfreeof` e `linear`.

`antidiff` é relacionada a `antid` como segue. Tome L, uma lista de dos elementos, como sendo o valor de retorno de `antid`. Então o valor de retorno de `antidiff` é igual a L`[1] + 'integrate (`L`[2],` x`)` onde x é a variável de integração.

Exemplos:

```
(%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` é a propriedade do gradiente atômico de uma expressão. Essa propriedade é atribuída por `gradef`.

atvalue (*expr*, [*x_1* = *a_1*, ..., *x_m* = *a_m*], *c*) Função
atvalue (*expr*, *x_1* = *a_1*, *c*) Função

Atribui o valor *c* a *expr* no ponto *x* = *a*. Tipicamente valores de extremidade são estabelecidos por esse mecanismo.

expr é a função de avaliação, *f*(*x_1*, ..., *x_m*), ou uma derivada, `diff` (*f*(*x_1*, ..., *x_m*), *x_1*, *n_1*, ..., *x_n*, *n_m*) na qual os argumentos da função explicitamente aparecem. *n_i* é a ordem de diferenciação com relação a *x_i*.

O ponto no qual o `atvalue` é estabelecido é dado pela lista de equações [*x_1* = *a_1*, ..., *x_m* = *a_m*]. Se existe uma variável simples *x_1*, uma única equação pode ser dada sem ser contida em uma lista.

`printprops` ([*f_1*, *f_2*, ...], `atvalue`) mostra os `atvalues` das funções *f_1*, *f_2*, ... como especificado por chamadas a `atvalue`. `printprops` (*f*, `atvalue`) mostra os `atvalues` de uma função *f*. `printprops (all, atvalue)` mostra os `atvalues` de todas as funções para as quais `atvalue`s são definidos.

Os simbolos `@1`, `@2`, ... representam as variáveis *x_1*, *x_2*, ... quando `atvalue`s são mostrados.

`atvalue` avalia seus argumentos. `atvalue` retorna *c*, o `atvalue`.

Exemplos:

```
(%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 - Função

O cálculo exterior de formas diferenciais é uma ferramenta básica de geometria diferencial desenvolvida por Elie Cartan e tem importantes aplicações na teoria das equações diferenciais parciais. O pacote `cartan` implementa as funções `ext_diff` e `lie_diff`, juntamente com os operadores `~` (produto da cunha) e `|` (contração de uma forma com um vetor.) Digite `demo (tensor)` para ver uma breve descrição desses comandos juntamente com exemplos.

`cartan` foi implementado por F.B. Estabrook e H.D. Wahlquist.

del (*x*) Função

`del (x)` representa a diferencial da variável *x*.

`diff` retorna uma expressão contendo `del` se uma variável independente não for especificada. Nesse caso, o valor de retorno é a então chamada "diferencial total".

Exemplos:

```
(%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*) Função

A função Delta de Dirac.

Correntemente somente `laplace` sabe sobre a função `delta`.

Exemplo:

```
(%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 Variável

Valor padrão: `[]`

`dependencies` é a lista de átomos que possuem dependências funcionais, atribuídas por `depends` ou `gradef`. A lista `dependencies` é cumulativa: cada chamada a `depends` ou a `gradef` anexa ítens adicionais.

Veja `depends` e `gradef`.

depends (*f_1, x_1, ..., f_n, x_n*) Função

Declara dependêcias funcionais entre variáveis para o propósito de calcular derivadas. Na ausência de dependêcias declaradas, `diff (f, x)` retorna zero. Se `depends (f,`

`x)` for declarada, `diff (f, x)` retorna uma derivada simbólica (isto é, um substantivo `diff`).

Cada argumento *f_1*, *x_1*, etc., pode ser o nome de uma variável ou array, ou uma lista de nomes. Todo elemento de *f_i* (talvez apenas um elemento simples) é declarado para depender de todo elemento de *x_i* (talvez apenas um elemento simples). Se algum *f_i* for o nome de um array ou contém o nome de um array, todos os elementos do array dependem de *x_i*.

`diff` reconhece dependências indiretas estabelecidas por `depends` e aplica a regra da cadeia nesses casos.

`remove (`*f*`, dependency)` remove todas as dependências declaradas para *f*.

`depends` retorna uma lista de dependências estabelecidas. As dependências são anexadas à variável global `dependencies`. `depends` avalia seus argumentos.

`diff` é o único comando Maxima que reconhece dependências estabelecidas por `depends`. Outras funções (`integrate`, `laplace`, etc.) somente reconhecem dependências explicitamente representadas por seus argumentos. Por exemplo, `integrate` não reconhece a dependência de `f` sobre `x` a menos que explicitamente representada como `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 Variável de opção

Valor padrão: `false`

Quando `derivabbrev` for `true`, derivadas simbólicas (isto é, substantivos `diff`) são mostradas como subscritos. De outra forma, derivadas são mostradas na notação de Leibniz `dy/dx`.

derivdegree (*expr, y, x*) Função

Retorna o maior grau de uma derivada da variável dependente *y* com relação à variável independente *x* ocorrendo em *expr*.

Exemplo:

```
(%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*) Função

Causa somente diferenciações com relação às variáveis indicadas, dentro do comando `ev`.

derivsubst Variável de opção

Valor padrão: `false`

Quando `derivsubst` for `true`, uma substituíção não sintática tais como `subst (x, 'diff (y, t), 'diff (y, t, 2))` retorna `'diff (x, t)`.

diff (*expr, x_1, n_1, ..., x_m, n_m*) Função
diff (*expr, x, n*) Função
diff (*expr, x*) Função
diff (*expr*) Função

Retorna uma derivada ou diferencial de *expr* com relação a alguma ou todas as variáveis em *expr*.

`diff (`*expr*`, `*x*`, `*n*`)` retorna a *n*'ésima derivada de *expr* com relação a *x*.

`diff (`*expr*`, `*x_1*`, `*n_1*`, ..., `*x_m*`, `*n_m*`)` retorna a derivada parcial mista de *expr* com relação a *x_1*, ..., *x_m*. Isso é equivalente a `diff (... (diff (`*expr*`, `*x_m*`, `*n_m*`) ...), `*x_1*`, `*n_1*`)`.

`diff (`*expr*`, `*x*`)` retorna a primeira derivada de *expr* com relação a uma variável *x*.

`diff (`*expr*`)` retorna a diferencial total de *expr*, isto é, a soma das derivadas de *expr* com relação a cada uma de suas variáveis vezes a diferencial `del` de cada variável. Nenhuma simplificação adicional de `del` é oferecida.

A forma substantiva de `diff` é requerida em alguns contextos, tal como declarando uma equação diferencial. Nesses casos, `diff` pode ser colocado apóstrofo (com `'diff`) para retornar a forma substantiva em lugar da realização da diferenciação.

Quando `derivabbrev` for `true`, derivadas são mostradas como subscritos. De outra forma, derivadas são mostradas na notação de Leibniz, `dy/dx`.

Exemplos:

```
(%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)
```

Para o pacote tensor, as seguintes modificações foram incorporadas:

(1) As derivadas de quaisquer objetos indexados em *expr* terão as variáveis *x_i* anexadas como argumentos adicionais. Então todos os índices de derivada serão ordenados.

(2) As variáveis *x_i* podem ser inteiros de 1 até o valor de uma variável `dimension` [valor padrão: 4]. Isso fará com que a diferenciação seja concluída com relação aos *x_i*'ésimos membros da lista `coordinates` que pode ser escolhida para uma lista de nomes de coordenadas, e.g., `[x, y, z, t]`. Se `coordinates` for associada a uma variável atômica, então aquela variável subscrita por *x_i* será usada para uma variável de diferenciação. Isso permite um array de nomes de coordenadas ou nomes subscritos como `X[1]`, `X[2]`, ... sejam usados. Se `coordinates` não foram atribuídas um valor, então as variáveis seram tratadas como em (1) acima.

diff Símbolo especial

Quando `diff` está presente como um `evflag` em chamadas para `ev`, Todas as diferenciações indicadas em `expr` são realizdas.

dscalar (*f*) Função

Aplica o d'Alembertiano escalar para a função escalar *f*.

`load ("ctensor")` chama essa função.

express (*expr*) Função

Expande o substantivo do operador diferencial em expressões em termos de derivadas parciais. `express` reconhece os operadores `grad`, `div`, `curl`, `laplacian`. `express` também expande o produto do X `~`.

Derivadas simbólicas (isto é, substantivos `diff`) no valor de retorno de `express` podem ser avaliadas incluíndo `diff` na chamada à função `ev` ou na linha de comando. Nesse contexto, `diff` age como uma `evfun`.

`load ("vect")` chama essa função.

Exemplos:

```
(%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*) Função
gradef (*a*, *x*, *expr*) Função

Define as derivadas parciais (i.e., os componentes do gradiente) da função *f* ou variável *a*.

`gradef (`*f*`(`*x_1*`, ..., `*x_n*`), `*g_1*`, ..., `*g_m*`)` define `df/dx_i` como *g_i*, onde *g_i* é uma expressão; *g_i* pode ser uma chamada de função, mas não o nome de uma função. O número de derivadas parciais *m* pode ser menor que o número de argumentos *n*, nesses casos derivadas são definidas com relação a *x_1* até *x_m* somente.

`gradef (`*a*`, `*x*`, `*expr*`)` define uma derivada de variável *a* com relação a *x* como *expr*. Isso também estabelece a dependência de *a* sobre *x* (via `depends (`*a*`, `*x*`)`).

O primeiro argumento *f*`(`*x_1*`, ..., `*x_n*`)` ou *a* é acompanhado de apóstrofo, mas os argumentos restantes *g_1*, ..., *g_m* são avaliados. `gradef` retorna a função ou variável para as quais as derivadas parciais são definidas.

`gradef` pode redefinir as derivadas de funções internas do Maxima. Por exemplo, `gradef (sin(x), sqrt (1 - sin(x)^2))` redefine uma derivada de `sin`.

`gradef` não pode definir derivadas parciais para um função subscrita.

`printprops ([`*f_1*`, ..., `*f_n*`], gradef)` mostra as derivadas parciais das funções *f_1*, ..., *f_n*, como definidas por `gradef`.

`printprops ([`*a_n*`, ..., `*a_n*`], atomgrad)` mostra as derivadas parciais das variáveis *a_n*, ..., *a_n*, como definidas por `gradef`.

`gradefs` é a lista de funções para as quais derivadas parciais foram definidas por `gradef`. `gradefs` não inclui quaisquer variáveis para quais derivadas parciais foram definidas por `gradef`.

Gradientes são necessários quando, por exemplo, uma função não é conhecida explicitamente mas suas derivadas primeiras são e isso é desejado para obter derivadas de ordem superior.

gradefs Variável de sistema

Valor padrão: `[]`

`gradefs` é a lista de funções para as quais derivadas parciais foram definidas por `gradef`. `gradefs` não inclui quaisquer variáveis para as quais derivadas parciais foram deinidas por `gradef`.

laplace (*expr*, *t*, *s*) Função

Tenta calcular a transformada de Laplace de *expr* com relação a uma variável *t* e parâmetro de transformação *s*. Se `laplace` não pode achar uma solução, um substantivo `'laplace` é retornado.

`laplace` reconhece em *expr* as funções `delta`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh`, e `erf`, também `derivative`, `integrate`, `sum`, e `ilt`. Se algumas outras funções estiverem presente, `laplace` pode não ser habilitada a calcular a tranformada.

expr pode também ser uma equação linear, diferencial de coeficiente contante no qual caso o `atvalue` da variável dependente é usado. O requerido `atvalue` pode ser fornecido ou antes ou depois da transformada ser calculada. Uma vez que as condições iniciais devem ser especificadas em zero, se um teve condições de limite impostas em

qualquer outro lugar ele pode impor essas sobre a solução geral e eliminar as constantes resolvendo a solução geral para essas e substituindo seus valores de volta.

`laplace` reconhece integrais de convolução da forma `integrate (f(x) * g(t - x), x, 0, t)`; outros tipos de convoluções não são reconhecidos.

Relações funcionais devem ser explicitamente representadas em *expr*; relações implícitas, estabelecidas por `depends`, não são reconhecidas. Isto é, se f depende de x e y, `f (x, y)` deve aparecer em *expr*.

Veja também `ilt`, a transformada inversa de Laplace.

Exemplos:

```
(%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 Integração

21.1 Introdução a Integração

Maxima tem muitas rotinas para manusear integração. A função `integrate` faz uso de muitas dessas. Exite também o pacote `antid`, que manuseia uma função não especificada (e suas derivadas, certamente). Para usos numéricos, existe a função `romberg`; um integrador adaptativo que usa a regra da quadratura dos currais de Newton, chamada `quanc8`; e uma escolha de integradores adaptativos de Quadpack, a saber `quad_qag`, `quad_qags`, etc. Funções hipergeométricas estão sendo trabalhadas, veja `specint` for details. Geralmente falando, Maxima somente manuseia integrais que são integráveis em termos de "funções elementares" (funções racionais, trigonometricas, logarítmicas, exponenciais, radicais, etc.) e umas poucas extensões (função de erro, dilogarithm). Isso não manuseia integrais em termos de funções desconhecidas tais como `g(x)` e `h(x)`.

21.2 Definições para Integração

changevar (*expr, f*(*x,y*), *y, x*) Função

Faz a mudança de variável dada por *f(x,y)* = `0` em todas as integrais que ocorrem em *expr* com integração em relação a *x*. A nova variável é *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
```

Uma expressão contendo uma forma substantiva, tais como as instâncias de `'integrate` acima, pode ser avaliada por `ev` com o sinalizador `nouns`. Por exemplo, a expressão retornada por `changevar` acima pode ser avaliada por `ev (%o3, nouns)`. `changevar` pode também ser usada para alterações nos índices de uma soma ou de um produto. Todavia, isso deve obrigatóriamente ser realizado de forma que quando uma alteração é feita em uma soma ou produto, essa mudança deve ser um artifício, i.e., `i = j+ ...`, não uma função de grau mais alto. 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*) Função

Uma rotina de integral dupla que foi escrita no alto-nível do Maxima e então traduzida e compilada para linguagem de máquina. Use `load (dblint)` para acessar esse pacote. Isso usa o método da regra de Simpson em ambas as direções x e y para calcular

```
/b /s(x)
|  |
|  |    f(x,y) dy dx
|  |
/a /r(x)
```

A função *f* deve ser uma função traduzida ou compilada de duas variáveis, e *r* e *s* devem cada uma ser uma função traduzida ou compilada de uma variável, enquanto *a* e *b* devem ser números em ponto flutuante. A rotina tem duas variáveis globais que determinam o número de divisões dos intervalos x e y: `dblint_x` e `dblint_y`, ambas as quais são inicialmente 10, e podem ser alteradas independentemente para outros valores inteiros (existem `2*dblint_x+1` pontos calculados na direção x , e `2*dblint_y+1` na direção y). A rotina subdivide o eixo X e então para cada valor de X isso primeiro calcula *r*`(x)` e *s*`(x)`; então o eixo Y entre *r*`(x)` e *s*`(x)` é subdividido e a integral ao longo do eixo Y é executada usando a regra de Simpson; então a integral ao longo do eixo X é concluída usando a regra de Simpson com os valores da função sendo as integrais-Y. Esse procedimento pode ser numericamente instável por uma grande variedade razões, mas razoávelmente rápido: evite usar isso sobre funções altamente oscilatórias e funções com singularidades (postes ou pontos de ramificação na região). As integrais Y dependem de quanto fragmentados *r*`(x)` e *s*`(x)` são, então se a ditância *s*`(x) - r(x)` varia rapidamente com X, nesse ponto pode ter erros substanciais provenientes de truncação com diferentes saltos-tamanhos nas várias integrais Y. Um pode incrementar `dblint_x` e `dblint_y` em uma tentativa para melhorar a convergência da reião, com sacrifício do tempo de computação. Os valores da função não são salvos, então se a função é muito desperdiçadora de tempo,você terá de esperar por re-computação se você mudar qualquer coisa (desculpe). Isso é requerido que as funções *f*, *r*, e *s* sejam ainda traduzidas ou compiladas previamente

chamando `dblint`. Isso resultará em ordens de magnitude de melhoramentos de velocidade sobre o código interpretado em muitos casos!

`demo (dblint)` executa uma demonstração de `dblint` aplicado a um problema exemplo.

defint (*expr, x, a, b*) Função

Tenta calcular uma integral definida. `defint` é chamada por `integrate` quando limites de integração são especificados, i.e., quando `integrate` é chamado como `integrate (`*expr*`, `*x*`, `*a*`, `*b*`)`. Dessa forma do ponto de vista do usuário, isso é suficiente para chamar `integrate`.

`defint` retorna uma expressão simbólica, e executa um dos dois: ou calcula a integral ou a forma substantiva da integral. Veja `quad_qag` e funções rellacionadas para aproximação numérica de integrais definidas.

erf (*x*) Função

Representa a função de erro, cuja derivada é: `2*exp(-x^2)/sqrt(%pi)`.

erfflag Variável de opção

Valor padrão: `true`

Quando `erfflag` é `false`, previne `risch` da introdução da função `erf` na resposta se não houver nenhum no integrando para começar.

ilt (*expr, t, s*) Função

Calcula a transformação inversa de Laplace de *expr* em relação a *t* e parâmetro *s*. *expr* deve ser uma razão de polinômios cujo denominador tem somente fatores lineares e quadráticos. Usando a funções `laplace` e `ilt` juntas com as funções `solve` ou `linsolve` o usuário pode resolver uma diferencial simples ou uma equação integral de convolução ou um conjunto delas.

```
(%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*) Função
integrate (*expr, x, a, b*) Função

Tenta simbolicamente calcular a integral de *expr* em relação a *x*. `integrate (`*expr*`,` *x*`)` é uma integral indefinida, enquanto `integrate (`*expr*`,` *x*`,` *a*`,` *b*`)` é uma integral definida, com limites de integração *a* e *b*. Os limites não poderam conter *x*, embora `integrate` não imponha essa restrição. *a* não precisa ser menor que *b*. Se *b* é igual a *a*, `integrate` retorna zero.

Veja `quad_qag` e funções relacionadas para aproximação numérica de integrais definidas. Veja `residue` para computação de resíduos (integração complexa). Veja `antid` para uma forma alternativa de calcular integrais indefinidas.

A integral (uma expressão livre de `integrate`) é retornada se `integrate` obtém sucesso. De outra forma o valor de retorno é a forma substantiva da integral (o operador com apóstrofo `'integrate`) ou uma expressão contendo uma ou mais formas substantivas. A forma substantiva de `integrate` é mostrada com um sinal de integral.

Em algumas circunstâncias isso é útil para construir uma forma substantiva manualmente, colocando em `integrate` um apóstrofo, e.g., `'integrate (`*expr*`,` *x*`)`. Por exemplo, a integral pode depender de alguns parâmetos que não estão ainda calculados. A forma substantiva pode ser aplicada a seus argumentos por `ev (`*i*`, nouns)` onde *i* é a forma substantiva de interesse.

`integrate` manuseia integrais definidas separadamente das indefinidas, e utiliza uma gama de heurísticas para manusear cada caso. Casos especiais de integrais definidas incluem limites de integração iguais a zero ou infinito (`inf` ou `minf`), funções trigonométricas com limites de integração iguais a zero e `%pi` ou `2 %pi`, funções racionais, integrais relacionadas para as definições de funções `beta` e `psi`, e algumas integrais logarítmicas e trigonométricas. Processando funções racionais pode incluir computação de resíduo. Se um caso especial aplicável não é encontrado, tentativa será feita para calcular a integra indefinida e avaliar isso nos limites de integração. Isso pode incluir pegar um limite como um limite de integração tendendo ao infinito ou a menos infinito; veja também `ldefint`.

Casos especiais de integrais indefinidas incluem funções trigonométricas, exponenciais e funções logarítmicas, e funções racionais. `integrate` pode também fazer uso de uma curta tabela de integais elementares.

`integrate` pode realizar uma mudança de variável se o integrando tem a forma `f(g(x)) * diff(g(x), x)`. `integrate` tenta achar uma subexpressão `g(x)` de forma que a derivada de `g(x)` divida o integrando. Essa busca pode fazer uso de derivadas definidas pela função `gradef`. Veja também `changevar` e `antid`.

Se nenhum dos procedimentos heurísticos acha uma integral indefinida, o algorítmo de Risch é executado. O sinalizador `risch` pode ser escolhido como um `evflag`, na chamada para `ev` ou na linha de comando, e.g., `ev (integrate (`*expr*`,` *x*`), risch)` ou `integrate (`*expr*`,` *x*`), risch`. Se `risch` está presente, `integrate` chama a função `risch` sem tentar heurísticas primeiro. Veja também `risch`.

`integrate` trabalha somente com relações funcionais representadas explicitamente com a notação `f(x)`. `integrate` não respeita dependências implicitas estabelecidas pela função `depends`. `integrate` pode necessitar conhecer alguma propriedade de um parâmetro no integrando. `integrate` irá primeiro consultar a base de dados do `assume`, e , se a variável de interesse não está lá, `integrate` perguntará ao usuário. Dependendo da pergunta, respostas adequadas são `yes;` ou `no;`, ou `pos;`, `zero;`, ou `neg;`.

`integrate` não é, por padrão, declarada ser linear. Veja `declare` e `linear`.

`integrate` tenta integração por partes somente em uns poucos casos especiais.

Exemplos:

- Integrais definidas e indefinidas elementares.

```
(%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
```

- Uso de `assume` e dúvida interativa.

```
(%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
```

- Mudança de variável. Existem duas mudanças de variável nesse exemplo: uma usando a derivada estabelecida por `gradef`, e uma usando a derivação `diff(r(x))` de uma função não especificada `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)))
```

- O valor de retorno contém a forma substantiva `'integrate`. Nesse exemplo, Maxima pode extrair um fator do denominador de uma função racional, mas não pode fatorar o restante ou de outra forma achar sua integral. `grind` mostra a forma substantiva `'integrate` no resultado. Veja também `integrate_use_rootsof` para mais sobre integrais de funções racionais.

```
(%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$
```

- Definindo uma função em termos de uma integral. O corpo de uma função não é avaliado quando a função é definida. Dessa forma o corpo de `f_1` nesse exemplo contém a forma substantiva de `integrate`. O operador aspas simples `''` faz com que a integral seja avaliada, e o resultado transforme-se no corpo de `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 Variável de sistema

Valor padrão: 0

`integração_constant_counter` é um contador que é atualizado a cada vez que uma constante de integração (nomeada pelo Maxima, e.g., `integrationconstant1`) é introduzida em uma expressão pela integração indefinida de uma equação.

integrate_use_rootsof Variável de opção

Valor padrão: `false`

Quando `integrate_use_rootsof` é `true` e o denominador de uma função racional não pode ser fatorado, `integrate` retorna a integral em uma forma que é uma soma sobre as raízes (não conhecidas ainda) do denominador.

Por exemplo, com `integrate_use_rootsof` escolhido para `false`, `integrate` retorna uma integral não resolvida de uma função racional na forma substantiva:

```
(%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)
```

Agora vamos escolher o sinalizador para ser true e a parte não resolvida da integral será expressa como um somatório sobre as raízes do denominador da função racional:

```
(%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)
```

Alternativamente o usuário pode calcular as raízes do denominador separadamente, e então expressar o integrando em termos dessas raízes, e.g., `1/((x - a)*(x - b)*(x - c))` ou `1/((x^2 - (a+b)*x + a*b)*(x - c))` se o denominador for um polinômio cúbico. Algumas vezes isso ajudará Maxima a obter resultados mais úteis.

ldefint (*expr, x, a, b*) Função

Tenta calcular a integral definida de *expr* pelo uso de `limit` para avaliar a integral indefinida *expr* em relação a *x* no limite superior *b* e no limite inferior *a*. Se isso falha para calcular a integral definida, `ldefint` retorna uma expressão contendo limites como formas substantivas.

`ldefint` não é chamada por `integrate`, então executando `ldefint` (*expr*, *x*, *a*, *b*) pode retornar um resultado diferente de `integrate` (*expr*, *x*, *a*, *b*). `ldefint` sempre usa o mesmo método para avaliar a integral definida, enquanto `integrate` pode utilizar várias heurísticas e pode reconhecer alguns casos especiais.

potential (*givengradient*) Função

O cálculo faz uso da variável global `potentialzeroloc[0]` que deve ser `nonlist` ou da forma

```
[indeterminatej=expressãoj, indeterminatek=expressãok, ...]
```

O formador sendo equivalente para a expressão nonlist para todos os lados direitos-manuseados mais tarde. Os lados direitos indicados são usados como o limite inferior de integração. O sucesso das integrações pode depender de seus valores e de sua ordem. `potentialzeroloc` é inicialmente escolhido para 0.

qq Função

O pacote `qq` (que pode ser carregado com `load ("qq")`) contém uma função `quanc8` que pode pegar ou 3 ou 4 arguments. A versão de 3 argumentos calcula a integral da função especificada como primeiro argumento sobre o intervalo de lo a hi como em `quanc8 ('função, lo, hi)`. o nome da função pode receber apóstrofo. A versão de 4 argumentos calculará a integral da função ou expressão (primeiro argumento) em relação à variável (segundo argumento) no intervalo de `lo` a `hi` como em `quanc8(<f(x) or expressão in x>, x, lo, hi)`. O método usado é o da quadratura dos currais de Newton, e a rotina é adaptativa. Isso irá dessa forma gastar tempo dividindo o intervalo somente quando necessário para completar as condições de erro especificadas pelas variáveis `quanc8_relerr` (valor padrão=1.0e-4) e `quanc8_abserr` (valor padrão=1.0e-8) que dão o teste de erro relativo:

```
|integral(função) - valor calculado| < quanc8_relerr*|integral(função)|
```

e o teste de erro absoluto:

```
|integral(função) - valor calculado| < quanc8_abserr
```

`printfile ("qq.usg")` yields additional informação.

quanc8 (*expr*, *a*, *b*) Função

Um integrador adaptativo. Demonstração e arquivos de utilização são fornecidos. O método é para usar a regra da quadratura dos currais de Newton, daí o nome da função `quanc8`, disponível em versões de 3 ou 4 argumentos. Verificação de erro absoluto e erro relativo são usadas. Para usar isso faça `load ("qq")`. Veja também `qq`.

residue (*expr*, *z*, *z_0*) Função

Calcula o resíduo no plano complexo da expressão *expr* quando a variável *z* assumes o valor *z_0*. O resíduo é o coeficiente de `(z - z_0)^(-1)` nas séries de Laurent para *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*) Função

Integra *expr* em relação a *x* usando um caso transcendental do algorítmo de Risch. (O caso algébrico do algorítmo de Risch foi implementado.) Isso atualmente manuseia os casos de exponenciais aninhadas e logarítmos que a parte principal de `integrate` não pode fazer. `integrate` irá aplicar automaticamente `risch` se dados esses casos.

`erfflag`, se `false`, previne `risch` da introdução da função `erf` na resposta se não for achado nenhum no integrando para começar.

```
(%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*) Função

romberg (*expr*, *a*, *b*) Função

Integração de Romberg. Existem dois caminhos para usar essa função. O primeiro é um caminho ineficiente como a versão de integral definida de `integrate`: `romberg (<integrando>, <variável of integração>, <lower limit>, <upper limit>)`.

Exemplos:

```
(%i1) showtime: true$
(%i2) romberg (sin(y), y, 0, %pi);
```

```
Evaluação 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);
Evaluação took 11.97 seconds (12.21 elapsed) using 12.423 MB.
(%o7)                  173.6730736617464
```

O segundo é um caminho eficiente que é usado como segue:

```
romberg (<função name>, <lower limit>, <upper limit>);
```

Continuando o exemplo acima, temos:

```
(%i8) f(x) := (mode_declare ([função(f), x], float), ''(%th(5)))$
(%i9) translate(f);
(%o9)                          [f]
(%i10) romberg (f, -5, 5);
Evaluação took 3.51 seconds (3.86 elapsed) using 6.641 MB.
(%o10)                 173.6730736617464
```

O primeiro argumento deve ser uma função trauzida ou compilada. (Se for compilada isso deve ser declarado para retorno a `flonum`.) Se o primeiro argumento não for já traduzido, `romberg` não tentará traduzí-lo mas resultará um erro.

A precisão da integração é governada pelas variáveis globais `rombergtol` (valor padrão 1.E-4) e `rombergit` (valor padrão 11). `romberg` retornará um resultado se a diferença relativa em sucessivas aproximações for menor que `rombergtol`. Isso tentará dividir ao meio o tamanho do passo `rombergit` vezes antes que isso seja abandonado. O número de iterações e avaliações da função que `romberg` fará é governado por `rombergabs` e `rombergmin`.

`romberg` pode ser chamada recursivamente e dessa forma pode fazer integrais duplas e triplas.

Exemplo:

```
(%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 variável in função 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
```

A vantagem com esse caminho é que a função `f` pode ser usada para outros propósitos, como imprimir gráficos. A desvantagem é que você tem que inventar um nome para ambas a função `f` e sua variável independente `x`. Ou, sem a variável 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
```

A vantagem aqui é que o código é menor.

```
(%i3) q (a, b) := romberg (romberg (x*y/(x+y), y, 0, x/2), x, a, b)$
(%i4) q (1, 3);
(%o4)                    .8193022864324522
```

Isso é sempre o caminho mais curto, e as variáveis não precisam ser declaradas porque elas estão no contexto de `romberg`. O uso de `romberg` para integrais multiplas pode ter grandes desvantagens, apesar disso. O amontoado de cálculos extras necessários por causa da informação geométrica descartada durante o processo pela expressão de integrais multiplas por esse caminho pode ser incrível. O usuário deverá ter certeza de entender e usar os comutadores `rombergtol` e `rombergit`.

rombergabs Variável de opção

Valor padrão: 0.0

Assumindo que estimativas sucessivas produzidas por `romberg` são `y[0]`, `y[1]`, `y[2]`, etc., então `romberg` retornará após `n` iterações se (grasseiramente falando)

```
(abs(y[n]-y[n-1]) <= rombergabs ou
abs(y[n]-y[n-1])/(if y[n]=0.0 then 1.0 else y[n]) <= rombergtol)
```

for `true`. (A condição sobre o número de iterações dadas por `rombergmin` deve também ser satisfeita.) Dessa forma se `rombergabs` é 0.0 (o padrão) você apenas pega o teste de erro relativo. A utilidade de uma variável adicional vem quando você executar uma integral, quando a contribuição dominante vem de uma pequena região. Então você pode fazer a integral sobre uma pequena região dominante primeiro, usando a verificação relativa de precisão, seguida pela integral sobre o restante da região usando a verificação absoluta de erro.

Exemplo: Suponha que você quer calcular

```
'integrate (exp(-x), x, 0, 50)
```

(numericamente) com uma precisão relativa de 1 parte em 10000000. Defina a função. `n` é o contador, então nós podemos ver quantas avaliações de função foram necessárias. Primeiro de tudo tente fazer a integral completa de uma só vez.

```
(%i1) f(x) := (mode_declare (n, integer, x, float), n:n+1, exp(-x))$
(%i2) translate(f)$
Warning-> n é 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
```

Que aproximadamente precisou de 257 avaliações de função . Agora faça a integral inteligentemente, primeiro fazendo `'integrate (exp(-x), x, 0, 10)` e então escolhendo `rombergabs` para 1.E-6 vezes (nessa integral parcial). Isso aproximdamente pega somente 130 avaliações de função.

```
(%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
```

Então se `f(x)` onde a função pegou um longo tempo de computação, o segundo método fez a mesma tarefa 2 vezes mais rápido.

rombergit Variável de opção

Valor padrão: 11

A precisão do comando `romberg` de integração é governada pelas variáveis globais `rombergtol` e `rombergit`. `romberg` retornará um resultado se a diferença relativa em sucessivas approximações é menor que `rombergtol`. Isso tentará dividir ao meio o tamanho do passoe `rombergit` vezes antes disso ser abandonado.

rombergmin Variável de opção

Valor padrão: 0

`rombergmin` governa o número mínimo de avaliações de função que `romberg` fará. `romberg` avaliará seu primeiro argumento pelo menos `2^(rombergmin+2)+1` vezes. Isso é útil para integrar funções oscilatórias, onde o teste normal de convergência pode lgumas vezes inadequadamente passar.

rombergtol Variável de opção

Valor padrão: 1e-4

A precisão do comando de integração de `romberg` é governada pelas variáveis globais `rombergtol` e `rombergit`. `romberg` retornará um resultado se a difereça relativa em sucessivas aproximações é menor que `rombergtol`. Isso tentará dividir ao meio o tamanho do passo `rombergit` vezes antes disso ser abandonado.

tldefint (*expr*, *x*, *a*, *b*) Função

Equivalente a `ldefint` com `tlimswitch` escolhido para `true`.

quad_qag (*f*(*x*), *x*, *a*, *b*, *key*, *epsrel*, *limit*) Função

Numericamente avalia a integral

$$\int_a^b f(x)dx$$

usando um integrador adaptativo simples.

A função a ser integrada é *f(x)*, com variável dependente *x*, e a função é para ser integrada entre os limites *a* e *b*. *key* é o integrador a ser usado e pode ser um inteiro entre 1 e 6, inclusive. O valor de *key* seleciona a ordem da regra de integração de Gauss-Kronrod.

A integração numérica é concluída adaptativamente pela subdivisão a região de integração até que a precisão desejada for completada.

Os argumentos opcionais *epsrel* e *limit* são o erro relativo desejado e o número máximo de subintervalos respectivamente. *epsrel* padrão em 1e-8 e *limit* é 200.

`quad_qag` retorna uma lista de quatro elementos:

uma aproximação para a integral,
o erro absoluto estimado da aproximação,
o número de avaliações do integrando,
um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` se nenhum problema for encontrado;

`1` se muitos subintervalos foram concluídos;

`2` se erro excessivo é detectado;

`3` se ocorre comportamento extremamente ruim do integrando;

`6` se a entrada é inválida.

Exemplos:

```
(%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*) Função

Integra numericamente a função dada usando quadratura adaptativa com extrapolação. A função a ser integrated é *f(x)*, com variável dependente *x*, e a função é para ser integrada entre os limites *a* e *b*.

Os argumentos opcionais *epsrel* e *limit* são o erro relativo desejado e o número máximo de subintervalos, respectivamente. *epsrel* padrão em 1e-8 e *limit* é 200.

`quad_qags` retorna uma lista de quatro elementos:

uma aproximação para a integral,
o erro absoluto estimado da aproximação,
o número de avaliações do integrando,
um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu comportamento excessivamente ruim do integrando;

`4` falhou para convergência

`5` integral é provavelmente divergente ou lentamente convergente

`6` se a entrada é inválida.

Exemplos:

```
(%i1) quad_qags (x^(1/2)*log(1/x), x, 0 ,1);
(%o1)   [.4444444444444448, 1.11022302462516E-15, 315, 0]
```

Note que `quad_qags` é mais preciso e eficiente que `quad_qag` para esse integrando.

quad_qagi (*f(x)*, *x*, *a*, *inftype*, *epsrel*, *limit*) Função

Avalia numericamente uma das seguintes integrais

$$\int_a^\infty f(x)dx$$

$$\int_\infty^a f(x)dx$$

$$\int_{-\infty}^\infty f(x)dx$$

usando a rotina Quadpack QAGI. A função a ser integrada é *f(x)*, com variável dependente *x*, e a função é para ser integrada sobre um intervalo infinito.

O parâmetro *inftype* determina o intervalo de integração como segue:

`inf` O intervalo vai de *a* ao infinito positivo.

`minf` O intervalo vai do infinito negativo até *a*.

`both` O intervalo corresponde a toda reta real.

Os argumentos opcionais *epsrel* e *limit* são o erro relativo desejado e o número maximo de subintervalos, respectivamente. *epsrel* padrão para 1e-8 e *limit* é 200.

`quad_qagi` retorna uma lista de quatro elementos:

uma aproximação para a integral,

o erro absoluto estimado da aproximação,

o número de avaliações do integrando,

um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu comportamento excessivamente ruim do integrando;

`4` falhou para convergência;

`5` integral é provavelmente divergente ou lentamente convergente;

`6` se a entrada for inválida.

Exemplos:

```
(%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*) Função

Calcula numericamente o valor principal de Cauchy de

$$\int_a^b \frac{f\left(x\right)}{x-c}\,dx$$

usando a rotina Quadpack QAWC. A função a ser integrada é *f(x)*/(*x* - *c*), com variável dependente *x*, e a função é para ser integrada sobre o intervalo que vai de *a* até *b*.

Os argumentos opcionais *epsrel* e *limit* são o erro relativo desejado e o máximo número de subintervalos, respectivamente. *epsrel* padrão para 1e-8 e *limit* é 200.

`quad_qawc` retorna uma lista de quatro elementos:

uma aproximação para a integral,

o erro absoluto estimado da aproximação,

o número de avaliações do integrando,

um código de erro.

O código de erro (quarto elemento do valoor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu comportamento excessivamente ruim do integrando;

`6` se a entrada é inválida.

Exemplos:

```
(%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*) Função

Calcula numericamente a integral tipo Fourier usando a rotina Quadpack QAWF. A integral é

$$\int_a^\infty f(x)w(x)dx$$

A função peso w é selecionada por *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

Os argumentos opcionais são:

epsabs Erro absoluto de aproximação desejado. Padrão é 1d-10.

limit Tamanho de array interno de trabalho. (*limit* - *limlst*)/2 é o maximo número de subintervalos para usar. O Padrão é 200.

maxp1 O número máximo dos momentos de Chebyshev. Deve ser maior que 0. O padrão é 100.

limlst Limite superior sobre número de ciclos. Deve ser maior ou igual a 3. O padrão é 10.

epsabs e *limit* são o erro relativo desejado e o número maximo de subintervalos, respectivamente. *epsrel* padrão para 1e-8 e *limit* é 200.

`quad_qawf` retorna uma lista de quatro elementos:

uma aproximação para a integral,

o erro absoluto estimado da aproximação,

o número de avaliações do integrando,

um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu um comportamento excessivamente ruim do integrando;

`6` se a entrada é invalida.

Exemplos:

```
(%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*) Função

Calcula numericamente a integral usando a rotina Quadpack QAWO:

$$\int_a^b f(x)w(x)dx$$

A função peso w é selecionada por *trig*:

`cos` $w(x) = cos(omegax)$

`sin` $w(x) = sin(omegax)$

Os argumentos opcionais são:

epsabs Erro absoluto desejado de aproximação. O Padrão é 1d-10.

limit Tamanho do array interno de trabalho. (*limit* - *limlst*)/2 é o número máximo de subintervalos a serem usados. Default é 200.

maxp1 Número máximo dos momentos de Chebyshev. Deve ser maior que 0. O padrão é 100.

limlst Limite superior sobre o número de ciclos. Deve ser maior que ou igual a 3. O padrão é 10.

epsabs e *limit* são o erro relativo desejado e o número máximo de subintervalos, respectivamente. *epsrel* o padrão é 1e-8 e *limit* é 200.

`quad_qawo` retorna uma lista de quatro elementos:

uma aproximação para a integral,

o erro absoluto estimado da aproximação,

o número de avaliações do integrando,

um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` comportamento extremamente ruim do integrando;

`6` se a entrada é inválida.

Exemplos:

```
(%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*) Função

Numéricamente calcula a integral usando a rotina Quadpack QAWS:

$$\int_a^b f(x)w(x)dx$$

A função peso w é selecionada por *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` $w(x) = (x-a)^a lfa(b-x)^b etalog(x-a)log(b-x)$

O argumentos opcionais são:

epsabs Erro absoluto desejado de aproximação. O padrão é 1d-10.

limit Tamanho do array interno de trabalho. (*limit* - *limlst*)/2 é o número máximo de subintervalos para usar. O padrão é 200.

epsabs e *limit* são o erro relativo desejado e o número máximo de subintervalos, respectivamente. *epsrel* o padrão é 1e-8 e *limit* é 200.

`quad_qaws` retorna uma lista de quatro elementos:

uma aproximação para a integral,

o erro absoluto estimado da aproximação,

o número de avaliações do integrando,

um código de erro.

O código de erro (quarto elemento do valor de retorno) pode ter os valores:

`0` nenhum problema foi encontrado;

`1` muitos subintervalos foram concluídos;

`2` erro excessivo é detectado;

`3` ocorreu um comportamento excessivamente ruim do integrando;

6 se a entrada é invalida.

Exemplos:

```
(%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 Equações

22.1 Definições para Equações

%rnum_list Variável

Valor padrão: `[]`

`%rnum_list` é a lista de variáveis introduzidas em soluções por `algsys`. `%r` variáveis São adicionadas a `%rnum_list` na ordem em que forem criadas. Isso é conveniente para fazer substituições dentro da solução mais tarde. É recomendado usar essa lista em lugar de fazer `concat ('%r, j)`.

algexact Variável

Valor padrão: `false`

`algexact` afeta o comportamento de `algsys` como segue:

Se `algexact` é `true`, `algsys` sempre chama `solve` e então usa `realroots` sobre falhas de `solve`.

Se `algexact` é `false`, `solve` é chamada somente se o eliminante não for de uma variável, ou se for uma quadrática ou uma biquadrada.

Dessa forma `algexact: true` não garante soluções exatas, apenas que `algsys` tentará primeiro pegar soluções exatas, e somente retorna aproximações quando tudo mais falha.

algsys ([*expr_1, ..., expr_m*], [*x_1, ..., x_n*]) Função
algsys ([*eqn_1, ..., eqn_m*], [*x_1, ..., x_n*]) Função

Resolve polinômios simultâneos *expr_1*, ..., *expr_m* ou equações polinômiais *eqn_1*, ..., *eqn_m* para as variáveis *x_1*, ..., *x_n*. Uma expressão *expr* é equivalente a uma equação *expr* `= 0`. Pode existir mais equações que variáveis ou vice-versa.

`algsys` retorna uma lista de soluções, com cada solução dada com uma lista de valores de estado das equações das variáveis *x_1*, ..., *x_n* que satisfazem o sistema de equações. Se `algsys` não pode achar uma solução, uma lista vazia `[]` é retornada.

Os símbolos `%r1`, `%r2`, ..., são introduzidos tantos quantos forem necessários para representar parâmetros arbitrários na solução; essas variáveis são também anexadas à lista `%rnum_list`.

O método usado é o seguinte:

(1) Primeiro as equações são fatoradas e quebradas em subsistemas.

(2) Para cada subsistema *S_i*, uma equação *E* e uma variável *x* são selecionados. A variável é escolhida para ter o menor grau não zero. Então a resultante de *E* e *E_j* em relação a *x* é calculada para cada um das equações restantes *E_j* nos subsistemas *S_i*. Isso retorna um novo subsistema *S_i'* em umas poucas variáveis, como *x* tenha sido eliminada. O processo agora retorna ao passo (1).

(3) Eventualmente, um subsistema consistindo de uma equação simples é obtido. Se a equação é de várias variáveis e aproximações na forma de números em ponto flutuante nã tenham sido introduzidas, então `solve` é chamada para achar uma solução exata.

Em alguns casos, `solve` não está habilitada a achar uma solução, ou se isso é feito a solução pode ser uma expressão expressão muito larga.

Se a equação é de uma única variável e é ou linear, ou quadrática, ou biquadrada, então novamente `solve` é chamada se aproximações não tiverem sido introduzidas. Se aproximações tiverem sido introduzidas ou a equação não é de uma única variável e nem tão pouco linear, quadratica, ou biquadrada, então o comutador `realonly` é `true`, A função `realroots` é chamada para achar o valor real das soluções. Se `realonly` é `false`, então `allroots` é chamada a qual procura por soluções reais e complexas.

Se `algsys` produz uma solução que tem poucos digitos significativos que o requerido, o usuário pode escolher o valor de `algepsilon` para um valor maior.

Se `algexact` é escolhido para `true`, `solve` será sempre chamada.

(4) Finalmente, as soluções obtidas no passo (3) são substituídas dentro dos níveis prévios e o processo de solução retorna para (1).

Quando `algsys` encontrar uma equação de várias variáveis que contém aproximações em ponto flutuante (usualmente devido a suas falhas em achar soluções exatas por um estágio mais fácil), então não tentará aplicar métodos exatos para tais equações e em lugar disso imprime a mensagem: "`algsys` cannot solve - system too complicated."

Interações com `radcan` podem produzir expressões largas ou complicadas. Naquele caso, pode ser possível isolar partes do resultado com `pickapart` ou `reveal`.

Ocasionalmente, `radcan` pode introduzir uma unidade imaginária `%i` dentro de uma solução que é atualmente avaliada como real.

Exemplos:

```
(%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*) Função
allroots (*eqn*) Função

Calcula aproximações numéricas de raízes reais e complexas do polinômio *expr* ou equação polinômial *eqn* de uma variável.

O sinalizador `polyfactor` quando `true` faz com que `allroots` fatore o polinômio sobre os números reais se o polinômio for real, ou sobre os números complexos, se o polinômio for complexo.

`allroots` pode retornar resultados imprecisos no caso de multiplas raízes. Se o polinômio for real, `allroots (%i*`*p*`)` pode retornar aproximações mais precisas que `allroots (`*p*`)`, como `allroots` invoca um algorítmo diferente naquele caso.

`allroots` rejeita não-polinômios. Isso requer que o numerador após a classificação (`rat`'ing) poderá ser um polinômio, e isso requer que o denominador seja quando muito um número complexo. Com um resultado disso `allroots` irá sempre retornar uma expressão equivalente (mas fatorada), se `polyfactor` for `true`.

Para polinômios complexos um algorítmo por Jenkins and Traub é usado (Algorithm 419, *Comm. ACM*, vol. 15, (1972), p. 97). Para polinômios reais o algorítmo usado é devido a Jenkins (Algorithm 493, *ACM TOMS*, vol. 1, (1975), p.178).

Exemplos:

```
(%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 Variável

Valor padrão: `true`

Quando `backsubst` é `false`, evita substituições em expressões anteriores após as equações terem sido triangularizadas. Isso pode ser de grande ajuda em problemas muito grandes onde substituição em expressões anteriores pode vir a causar a geração de expressões extremamente largas.

breakup Variável

Valor padrão: `true`

Quando `breakup` é `true`, `solve` expressa soluções de equações cúbicas e quárticas em termos de subexpressões comuns, que são atribuídas a rótulos de expressões intermediárias (`%t1`, `%t2`, etc.). De outra forma, subexpressões comuns não são identificadas.

`breakup: true` tem efeito somente quando `programmode` é `false`.

Exemplos:

```
(%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*) Função
dimension (*eqn_1*, ..., *eqn_n*) Função

`dimen` é um pacote de análise dimensional. `load ("dimen")` chama esse pacote. `demo ("dimen")` mostra uma cura demostração.

dispflag Variável

Valor padrão: `true`

Se escolhida para `false` dentro de um `block` inibirá a visualização da saída gerada pelas funções solve chamadas de dentro de `block`. Terminando `block` com um sinal de dolar, $, escolhe `dispflag` para `false`.

funcsolve (*eqn*, *g*(*t*)) Função

Retorna `[`*g*`(`*t*`) = ...]` ou `[]`, dependendo de existir ou não uma função racional *g*`(`*t*`)` satisfazendo *eqn*, que deve ser de primeira ordem, polinômio linear em (para esse caso) *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));

Equações dependentes eliminadas:  (4 3)
                                    n
(%o2)                 f(n) = ---------------
                             (n + 1) (n + 2)
```

Atenção: essa é uma implementação muito rudimentar – muitas verificações de segurança e obviamente generalizações estão ausêntes.

globalsolve Variável

Valor padrão: `false`

Quando `globalsolve` é `true`, a variáveis resolvidas são atribuídos os valores da solução achados por `solve`.

Exemplos:

```
(%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*) Função

`inteqn` é um pacote para resolver equações integrais. `load ("inteqn")` carrega esse pacote.

ie é a equação integral; *unk* é a função desconhecida; *tech* é a técnica a ser tentada nesses dados acima (*tech* = `first` significa: tente a primeira técnica que achar uma solução; *tech* = `all` significa: tente todas a técnicas aplicáveis); *n* é o número máximo de termos a serem usados de `taylor`, `neumann`, `firstkindseries`, ou `fredseries` (isso é também o número máximo de ciclos de recurssão para o método de diferenciação); *guess* é o inicial suposto para `neumann` ou `firstkindseries`.

Valores padrão do segundo até o quinto parâmetro são:

unk: `p(x)`, onde *p* é a primeira função encontrada em um integrando que é desconhecida para Maxima e *x* é a variável que ocorre como um argumento para a primeira ocorrência de *p* achada fora de uma integral no caso de equações `secondkind` , ou é somente outra variável ao lado da variável de integração em equações `firstkind`. Se uma tentativa de procurar por *x* falha, o usuário será perguntado para suprir a variável independente.

tech: `first`

n: 1

guess: `none` o que fará com que `neumann` e `firstkindseries` use `f(x)` como uma suposição inicial.

ieqnprint Variável de opção

Valor padrão: `true`

`ieqnprint` governa o comportamento do resultado retornado pelo comando `ieqn`. Quando `ieqnprint` é `false`, as listas retornadas pela função `ieqn` são da forma

[*solução*, *tecnica usada*, *nterms*, *sinalizador*]

onde *sinalizador* é retirado se a solução for exata.

De outra forma, isso é a palavra `approximate` ou `incomplete` correspondendo à forma inexata ou forma aberta de solução, respectivamente. Se um método de série foi usado, *nterms* fornece o número de termos usados (que poderá ser menor que os n dados para `ieqn` se ocorrer um erro evita a geração de termos adicionais).

lhs (*eqn*) Função

Retorna o lado esquerdo da equação *eqn*.

Se o argumento não for uma equação, `lhs` retorna o argumento.

Veja também `rhs`.

Exemplo:

```
(%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*]) Função

Resolve a lista de equações lineares simultâneas para a lista de variáveis. As expressões devem ser cada uma polinômios nas variáveis e podem ser equações.

Quando `globalsolve` é `true` então variáveis que foram resolvidas serão escolhidas para a solução do conjunto de equações simultâneas.

Quando `backsubst` é `false`, `linsolve` não realiza substituição em equações anteriores após as equações terem sido triangularizadas. Isso pode ser necessário em problemas muito grandes onde substituição em equações anteriores poderá causar a geração de expressões extremamente largas.

Quando `linsolve_params` é `true`, `linsolve` também gera símbolos `%r` usados para representar parâmetros arbitrários descritos no manual sob `algsys`. De outra forma, `linsolve` resolve um menor-determinado sistema de equações com algumas variáveis expressas em termos de outras.

```
(%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 Variável

Valor padrão: `true`

Quando `linsolvewarn` é `true`, `linsolve` imprime uma mensagem "Dependent equações eliminated".

linsolve_params Variável

Valor padrão: `true`

Quando `linsolve_params` é `true`, `linsolve` também gera os símbolos `%r` usados para representar parâmetros arbitrários descritos no manual sob `algsys`. De outra forma, `linsolve` resolve um menor-determinado sistema de equações com algumas variáveis expressas em termos e outras.

multiplicities Variável

Valor padrão: `not_set_yet`

`multiplicities` é escolhida para uma lista de multiplicidades das soluções individuais retornadas por `solve` ou `realroots`.

nroots (*p, low, high*) Função

Retorna o número de raízes reais do polinômio real de uma única variável *p* no intervalo semi-aberto `(`*low*`,` *high*`]`. Uma extremidade do intervalo podem ser `minf` ou `inf`. infinito e mais infinito.

`nroots` usa o método das sequüências de Sturm.

```
(%i1) p: x^10 - 2*x^4 + 1/2$
(%i2) nroots (p, -6, 9.1);
(%o2)                          4
```

nthroot (*p, n*) Função

Onde p é um polinômio com coeficientes inteiros e n é um inteiro positivo retorna q, um polinômio sobre os inteiros, tal que q^n=p ou imprime uma mensagem de erro indicando que p não é uma potência n-ésima perfeita. Essa rotina é mais rápida que `factor` ou mesmo `sqfr`.

programmode Variável

Valor padrão: `true`

Quando `programmode` é `true`, `solve`, `realroots`, `allroots`, e `linsolve` retornam soluções como elementos em uma lista. (Exceto quando `backsubst` é escolhido para `false`, nesse caso `programmode: false` é assumido.)

Quando `programmode` é `false`, `solve`, etc. cria rótulos de expressões intermediárias `%t1`, `t2`, etc., e atribui as soluções para eles.

realonly Variável

Valor padrão: `false`

Quando `realonly` é `true`, `algsys` retorna somente aquelas soluções que estão livres de `%i`.

realroots (*poly, bound*) Função

Acha todas as raízes reais de um polinômio também real de uma única variável poly dentro de uma tolerância de limite que, se menor que 1, faz com que todas as raízes da integral sejam achadas exatamente. O parâmetro limite pode ser arbitrariamente pequeno com o objetivo de encontrar qualquer precisão desejada. O primeiro argumento pode também ser uma equação. `realroots` escolhe `multiplicities`, útil em caso de multiplas raízes. `realroots (`*p*`)` é equivalente a `realroots (`*p*`, rootsepsilon)`. `rootsepsilon` é um número real usado para estabelecer um intervalo de confidência para as ra@'izes. Faça `example (realroots)` para um exemplo.

rhs (*eqn*) Função

Retorna o lado direito da equação *eqn*.

Se o argumento não é uma equação, `rhs` retorna `0`.

Veja também `lhs`.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `true`

`rootsconmode` governa o comportamento do comando `rootscontract`. Veja `rootscontract` para detalhes.

rootscontract (*expr*) Função

Converte produtos de raízes em raízes de produtos. Por exemplo, `rootscontract (sqrt(x)*y^(3/2))` retorna `sqrt(x*y^3)`.

Quando `radexpand` é `true` e `domain` é `real`, `rootscontract` converte `abs` em `sqrt`, e.g., `rootscontract (abs(x)*sqrt(y))` retorna `sqrt(x^2*y)`.

Existe uma opção `rootsconmode` afetando `rootscontract` como segue:

Problem	Value of rootsconmode	Result of applying 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)

Quando `rootsconmode` é `false`, `rootscontract` contrai somente como relação a expoentes de número racional cujos denominadores são os mesmos. A chave para os exemplos `rootsconmode: true` é simplesmente que 2 divides 4 mas não divide 3. `rootsconmode: all` envolve pegar o menor multiplo comum dos denominadores dos expoentes.

`rootscontract` usa `ratsimp` em uma maneira similar a `logcontract`.

Exemplos:

```
(%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 Variável de opção

Valor padrão: 1.0e-7

`rootsepsilon` é a tolerância que estabelece o intervalo de confiabilidade para as raízes achadas pela função `realroots`.

solve (*expr*, *x*) Função

solve (*expr*) Função

solve ([*eqn_1*, ..., *eqn_n*], [*x_1*, ..., *x_n*]) Função

Resolve a equação algébrica *expr* para a variável *x* e retorna uma lista de equações solução em *x*. Se *expr* não é uma equação, a equação *expr* `= 0` é assumida em seu lugar. *x* pode ser uma função (e.g. `f(x)`), ou outra expressão não atômica exceto uma adição ou um produto. *x* pode ser omitido se *expr* contém somente uma variável. *expr* pode ser uma expressão racional, e pode conter funções trigonométricas, exponenciais, etc.

O seguinte método é usado:

Tome E sendo a expressão e X sendo a variável. Se E é linear em X então isso é trivialmente resolvido para X. De outra forma se E é da forma `A*X^N + B` então o resultado é `(-B/A)^1/N)` vezes as `N`'ésimas raízes da unidade.

Se E não é linear em X então o máximo divisor comum (mdc) dos expoentes de X em E (digamos N) é dividido dentro dos expoentes e a multiplicidade das raízes é multiplicada por N. Então `solve` é chamada novamente sobre o resultado. Se E for

dada em fatores então `solve` é chamada sobre cada um dos fatores. Finalmente `solve` usará as fórmulas quadráticas, cúbicas, ou quárticas onde necessário.

No caso onde E for um polinômio em alguma função de variável a ser resolvida, digamos `F(X)`, então isso é primeiro resolvida para `F(X)` (chama o resultado C), então a equação `F(X)=C` pode ser resolvida para X fornecendo o inverso da função F que é conhecida.

`breakup` se `false` fará com que `solve` expresse as soluções de equações cúbicas ou quárticas como expressões simples ao invés de como feito em cima de várias subexpressões comuns que é o padrão.

`multiplicities` - será escolhido para uma lista de multiplicidades de soluções individuais retornadas por `solve`, `realroots`, ou `allroots`. Tente `apropos (solve)` para os comutadores que afetam `solve`. `describe` pode então ser usada sobre o nome do comutador individual se seu proprósito não é claro.

`solve ([`*eqn_1*`, ..., `*eqn_n*`], [`*x_1*`, ..., `*x_n*`])` resolve um sistema de equações polinomiais (lineares ou não-lineares) simultâneas por chamada a `linsolve` ou `algsys` e retorna uma lista de listas solução nas variáveis. No caso de `linsolve` essa lista conterá uma lista simples de soluções. Isso pega duas listas como argumentos. A primeira lista representa as equações a serem resolvidas; a segunda lista é a lista de desconhecidos a ser determinada. Se o número total de variáveis nas equações é igual ao número de equações, a segunda lista-argumento pode ser omitida. Para sistemas lineares se as dadas equações não são compatíveis, a mensagem `inconsistent` será mostrada (veja o comutador `solve_inconsistent_error`); se não existe solução única, então `singular` será mostrado.

Exemplos:

```
(%i1) solve (asin (cos (3*x))*(f(x) - 1), x);

SOLVE is using arc-trig functions to get a solution.
Some soluções 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 Variável de opção

Valor padrão: `true`

Quando `solvedecomposes` é `true`, `solve` chama `polydecomp` se perguntado para resolver polinômios.

solveexplicit Variável de opção

Valor padrão: `false`

Quando `solveexplicit` é `true`, inibe `solve` de retornar soluções implícitas, isto é, soluções da forma `F(x) = 0` onde `F` é alguma função.

solvefactors Variável de opção

Valor padrão: `true`

Quando `solvefactors` é `false`, `solve` não tenta fatorar a expressão. O `false` escolhido pode ser desejado em alguns casos onde a fatoração não é necessária.

solvenullwarn Variável de opção

Valor padrão: `true`

Quando `solvenullwarn` é `true`, `solve` imprime uma mensagem de alerta se chamada com ou uma lista equação ou uma variável lista nula. Por exemplo, `solve ([], [])` imprimirá duas mensagens de alerta e retorna `[]`.

solveradcan Variável de opção

Valor padrão: `false`

Quando `solveradcan` é `true`, `solve` chama `radcan` que faz `solve` lento mas permitirá certamente que problemas contendo exponeniais e logarítmos sejam resolvidos.

solvetrigwarn Variável de opção

Valor padrão: `true`

Quando `solvetrigwarn` é `true`, `solve` pode imprimir uma mensagem dizendo que está usando funções trigonométricas inversas para resolver a equação, e desse modo perdendo soluções.

solve_inconsistent_error Variável de opção

Valor padrão: `true`

Quando `solve_inconsistent_error` é `true`, `solve` e `linsolve` resultam em erro se as equações a serem resolvidas são inconsistentes.

Se `false`, `solve` e `linsolve` retornam uma lista vazia `[]` se as equações forem inconsistentes.

Exemplo:

```
(%i1) solve_inconsistent_error: true$
(%i2) solve ([a + b = 1, a + b = 2], [a, b]);
Inconsistent equações:  (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 Equações Diferenciais

23.1 Definições para Equações Diferenciais

bc2 (*solução, xval1, yval1, xval2, yval2*) Função

Resolve problema do valor limite para equações diferenciais de segunda ordem. Aqui: *solução* é uma solução geral para a equação, como encontrado por `ode2`, *xval1* é uma equação para a variável independente na forma *x* = *x0*, e *yval1* é uma equação para a variável dependente na forma *y* = *y0*. A *xval2* e a *yval2* são equações para essas variáveis em outro ponto. Veja `ode2` para exemplo de utilização.

desolve (*eqn, x*) Função

desolve ([*eqn_1, ..., eqn_n*], [*x_1, ..., x_n*]) Função

A função `dsolve` resolve sistemas de equações diferenciais ordinárias lineares usando transformada de Laplace. Aqui as *eqn*'s são equações diferenciais nas variáveis dependentes *x_1*, ..., *x_n*. A relação funcional deve ser explicitamente indicada em ambas as equações e as variáveis. Por Exemplo

```
'diff(f,x,2)=sin(x)+'diff(g,x);
'diff(f,x)+x^2-f=2*'diff(g,x,2);
```

não é o formato apropriado. O caminho correto é:

```
'diff(f(x),x,2)=sin(x)+'diff(g(x),x);
'diff(f(x),x)+x^2-f=2*'diff(g(x),x,2);
```

A chamada é então `desolve([%o3,%o4],[f(x),g(x)]);` .

Se as condições iniciais em 0 são conhecidas, elas podem ser fornecidas antes chamando `desolve` através de `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)]
```

Se `desolve` não pode obter uma solução, retorna `false`.

ic1 (*solução, xval, yval*) Função

Resolve o problema do valor inicial para equação diferencial de primeira ordem. Aqui: *solução* é uma solução geral para a equação, como encontrado por `ode2`, *xval* é uma equação para a variável independente na forma *x* `=` *x0*, e *yval* é uma equação para a variável dependente na forma *y* `=` *y0*. Veja `ode2` para exemplo de utilização.

ic2 (*solução, xval, yval, dval*) Função

Resolve o problema do valor inicial para equação diferencial de segunda ordem. Aqui: *solução* é uma solução geral para a equação, como encontrado por `ode2`, *xval* é uma equação para a variável independente na forma *x* `=` *x0*, *yval* é uma equação para a variável dependente na forma *y* `=` *y0*, e *dval* é uma equação para a derivada da variável dependente com relação à variável independente avaliada no ponto *xval*. Veja `ode2` para exemplo de utilização.

ode2 (*eqn, dvar, ivar*) Função

A função `ode2` resolve equações diferenciais ordinária ou de primeira ou de segunda ordem. Recebe três argumentos: uma EDO *eqn*, a variável dependente *dvar*, e a variável independente*ivar*. Quando obtém sucesso, retorna ou uma solução (explícita ou implícita) para a variável dependente. `%c` é usado para representar a constante no caso de equações de primeira ordem, e `%k1` e `%k2` as constantes para equações de segunda ordem. Se `ode2` não pode obter a solução por alguma razão, retorna `false`, após talvez mostra uma mensagem de erro. O método implementado para equações diferenciais de primeira ordem na seqüência na qual eles são testados são: linear, separável, exato - talvez requerendo um fator de integração, homogêneos, equação de Bernoulli, e um método homogêneo geral. Para segunda ordem: coeficiente constante, exato, linear homogêneo com coeficientes não-constantes os quais podem ser transformados para coeficientes constates, o Euler ou equação equidimensional, o método de variação de parâmetros, e equações as quais são livres ou da variável independente ou da dependente de modo que elas possam ser reduzidas duas equações lineares de primeria ordem para serem resolvidas seqüêncialmente. No curso de resolver EDOs, muitas variáveis são escolhidas puramente para propósitos informativos: `método` denota o método de solução usado e.g. `linear`, `intfactor` denota qualquer fator de integração usado, `odeindex` denota o índice para o método de Bernoulli ou para o método homogêneo generalizado, e `yp` denota a solução particular para a técnica de variação de parâmetros.

Com o objetivo de resolver os problemas dos valores iniciais (PVIs) e problemas dos valores limite (PVLs), a rotina `ic1` está disponível para equações de primeira ordem, e `ic2` e `bc2` para segunda ordem PVIs e PVLs, 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 Numérico

24.1 Introdução a Numérico

24.2 Pacotes de Fourier

O pacote `fft` compreende funções para computação numérica (não simbólica) das transformações rápidas de Fourier. `load ("fft")` chama esse pacote. Veja `fft`.

O pacote `fourie` compreende funções para computação simbólica de séries de Fourier. `load ("fourie")` chama esse pacote. Existem funções no pacote `fourie` para calcular coeficientes da integral de Fourier e algumas funções para manipulação de expressões. Veja `Definições para Séries`.

24.3 Definições para Numérico

polartorect (*magnitude_array, phase_array*) Função

Traduz valores complexos da forma `r %e^(%i t)` para a forma `a + b %i`. `load ("fft")` chama essa função dentro do Maxima. Veja também `fft`.

O módulo e a fase, `r` e `t`, São tomados de *magnitude_array* e *phase_array*, respectivamente. Os valores originais de arrays de entrada são substituídos pelas partes real e emaginária, `a` e `b`, no retorno. As saídas são calculadas como

```
a: r cos (t)
b: r sin (t)
```

Os arrays de entrada devem ter o mesmo tamanho e ser unidimensionais. O tamanho do array não deve ser uma potência de 2.

`polartorect` é a função inversa de `recttopolar`.

recttopolar (*real_array, imaginary_array*) Função

Traduz valores complexos da forma `a + b %i` para a forma `r %e^(%i t)`. `load ("fft")` chama essa função dentro do Maxima. Veja também `fft`.

As partes real e imaginária, `a` e `b`, são tomadas de *real_array* e *imaginary_array*, respectivamente. Os valores originais dos arrays de entrada são substituídos pelo módulo e pelo ângulo, `r` e `t`, no retorno. As saídas são calculadas como

```
r: sqrt (a^2 + b^2)
t: atan2 (b, a)
```

O ângulo calculado encontra-se no intervalo de `-%pi` a `%pi`.

Os arrays de entrada devem ter o mesmo tamanho e ser unidimensionais. O tamanho do array não deve ser uma potência de 2.

`recttopolar` é a função inversa de `polartorect`.

ift (*real_array, imaginary_array*) Função

Transformação rápida inversa discreta de Fourier . `load ("fft")` chama essa função dentro do Maxima.

`ift` realiza a transformação rápida complexa de Fourier sobre arrays em ponto flutuante unidimensionais. A transformação inversa é definida como

```
x[j]: sum (y[j] exp (+2 %i %pi j k / n), k, 0, n-1)
```

Veja `fft` para maiores detalhes.

fft (*real_array, imaginary_array*) Função
ift (*real_array, imaginary_array*) Função
recttopolar (*real_array, imaginary_array*) Função
polartorect (*magnitude_array, phase_array*) Função

Transformação rápidada de Fourier e funções relacionadas. `load ("fft")` chama essas funções dentro do Maxima.

`fft` e `ift` realiza transformação rápida complexa de Fourier e a transformação inversa, respectivamente, sobre arrays em ponto flutuante unidimensionais. O tamanho de *imaginary_array* deve ser igual ao tamanho de *real_array*.

`fft` e `ift` operam in-loco. Isto é, sobre o retorno de `fft` ou de `ift`, O conteúdo original dos arrays de entrada é substituído pela saída. A função `fillarray` pode fazer uma cópia de um array, isso pode ser necessário.

A transformação discreta de Fourier e sua transformação inversa são definidas como segue. Tome `x` sendo os dados originais, com

```
x[i]: real_array[i] + %i imaginary_array[i]
```

Tome `y` sendo os dados transformados. A transformação normal e sua transformação inversa são

```
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)
```

Arrays adequadas podem ser alocadas pela função `array`. Por exemplo:

```
array (my_array, float, n-1)$
```

declara um array unidimensional com n elementos, indexado de 0 a n-1 inclusive. O número de elementos n deve ser igual a 2^m para algum m.

`fft` pode ser aplicada a dados reais (todos os arrays imaginários são iguais a zero) para obter coeficientes seno e cosseno. Após chamar `fft`, os coeficientes seno e cosseno, digamos `a` e `b`, podem ser calculados como

```
a[0]: real_array[0]
b[0]: 0
```

e

```
a[j]: real_array[j] + real_array[n-j]
b[j]: imaginary_array[j] - imaginary_array[n-j]
```

para j variando de 1 a n/2-1, e

```
a[n/2]: real_array[n/2]
b[n/2]: 0
```

`recttopolar` traduz valores complexos da forma `a + b %i` para a forma `r %e^(%i t)`. Veja `recttopolar`.

`polartorect` traduz valores complexos da forma `r %e^(%i t)` para a forma `a + b %i`. Veja `polartorect`.

`demo ("fft")` exibe uma demonstração do pacote `fft`.

fortindent Option variable

Valor padrão: 0

`fortindent` controla a margem esquerda de indentação de expressões mostradas pelo comando `fortran`. 0 fornece indentação normal (i.e., 6 espaços), e valores positivos farão com que expressões sejam mostrados mais além para a direita.

fortran (*expr*) Função

Mostra *expr* como uma declaração Fortran. A linha de saída é indentada com espaços. Se a linha for muito longa, `fortran` imprime linhas de continuação. `fortran` mostra o operador de exponenciação `^` como `**`, e mostra um número complexo `a + b %i` na forma `(a,b)`.

expr pode ser uma equação. Nesse caso, `fortran` mostra uma declaração de atribuição, atribuindo o primeiro membro (esquerda) da equação ao segundo membro (direita). Em particular, se o primeiro membro *expr* é um nome de uma matriz, então `fortran` mostra uma declaração de atribuição para cada elemento da matriz.

Se *expr* não for alguma coisa reconhecida por `fortran`, a expressão é mostrada no formato `grind` sem reclamação. `fortran` não conhece listas, arrays ou funções.

`fortindent` controla o margem esquerda das linhas mostradas. 0 é a margem normal (i.e., indentada 6 espaços). Incrementando `fortindent` faz com que expressões sejam mostradas adiante para a direita.

quando `fortspaces` for `true`, `fortran` preenche cada linha mostrada com espaços em branco até completar 80 columas.

`fortran` avalia seus argumentos; colocando um apóstrofo em um argumento evita avaliação. `fortran` sempre retorna `done`.

Exemplos:

```
(%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 Variável de opção

Valor padrão: `false`

Quando `fortspaces` for `true`, `fortran` preenche cada linha mostrada com espaços em branco até completar 80 columas.

horner (*expr*, *x*) Função
horner (*expr*) Função

Retorna uma representação rearranjada de *expr* como na regra de Horner, usando *x* como variável principal se isso for especificado. `x` pode ser omitido e nesse caso a variável principal da forma de expressão racional canônica de *expr* é usada.

`horner` algumas vezes melhora a estabilidade se `expr` for ser numericamente avaliada. Isso também é útil se Maxima é usado para gerar programas para rodar em Fortran. Veja também `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*) Função
interpolate (*f*, *a*, *b*) Função

Encontra a raíz da função *f* com a variável *x* percorrendo o intervalo `[`*a*`,` *b*`]`. A função deve ter um sinal diferente em cada ponto final. Se essa condição não for alcançada, a ação da função é governada por `intpolerror`. Se `intpolerror` for `true` então ocorre um erro, de outra forma o valor de `intpolerror` é retornado (dessa forma para montagem de gráficos `intpolerror` possivelmente pode ser escolhido para 0.0). De outra forma (dado que Maxima pode avaliar o primeiro argumento no intervalo especificado, e que o intervalo é contínuo) `interpolate` é garantido vir para cima com a raíz (ou um deles se existir mais que uma raíz). A precisão de `interpolate`

é governada por `intpolabs` e `intpolrel` os quais devem ser números em ponto flutuante não negativos. `interpolate` encerrará quando o primeiro argumento avaliar para alguma coisa menor que ou igual a `intpolabs` ou se sucessivas aproximações da raíz diferirem por não mais que `intpolrel * <um dos aproximandos>`. O valor padrão de `intpolabs` e `intpolrel` são 0.0 de forma que `interpolate` pega como boa uma resposta como for possível com a precisão aritmética simples que tivermos. O primeiro argumento pode ser uma equação. A ordem dos dois últimos argumentos é irrelevante. Dessa forma

```
interpolate (sin(x) = x/2, x, %pi, 0.1);
```

é equivalente a

```
interpolate (sin(x) = x/2, x, 0.1, %pi);
```

O método usado é uma busca binária no intervalo especificado pelos últimos dois argumentos. Quando o resultado da busca for encontrado a função é fechada o suficiente para ser linear, isso inicia usando interpolação linear.

```
(%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
```

intpolabs Variável de opção

Valor padrão: 0.0

`intpolabs` é a precisão do comando `interpolate` e é governada por `intpolabs` e `intpolrel` que devem ser números não negatvos em ponto flutuante. `interpolate` terminará quando o primeiro argumento avaliar alguma coisa menor que ou igual a `intpolabs` ou se successivas aproximações para a raíz diferirem de não mais que `intpolrel * <um dos aproximandos>`. Os valores padrões de `intpolabs` e de `intpolrel` é 0.0 de forma que `interpolate` toma como bom uma resposta como é possível com a precisão aritmética simples que tivermos.

intpolerror Variável de opção

Valor padrão: `true`

`intpolerror` governa o comportamento de `interpolate`. Quando `interpolate` for chamada, ela determina se a função a ser interpolada satisfaz ou não a condição que os valores da função nos pontos finais do intervalo de interpolação são opostos em sinal. Se eles forem de sinais opostos, a interpolação prossegue. Se eles forem de mesmo sinal, e `intpolerror` for `true`, então um erro é sinalizado. Se eles forem de mesmo sinal e `intpolerror` não for `true`, o valor de `intpolerror` é retornado. Dessa forma para montagem de gráfico, `intpolerror` pode ser escolhida para 0.0.

intpolrel Variável de opção

Valor padrão: 0.0

`intpolrel` é a precisão do comando `interpolate` e é governada por `intpolabs` e `intpolrel` que devem ser números não negativos em ponto flutuante. `interpolate` terminará quando o primeiro argumento avaliar para alguma coisa menor que ou igual a `intpolabs` ou se sucessivas aproximações para a raíz diferirem de não mais que `intpolrel * <um dos aproximandos>`. Os valores padrão de `intpolabs` e `intpolrel` é 0.0 de forma que `interpolate` toma como bom uma resposta como é possível com a precisão aritmética simples que tivermos.

24.4 Definições para Séries de Fourier

equalp (*x*, *y*) Função

Retorna `true` se `equal (`*x*`, `*y*`)` de outra forma `false` (não fornece uma mensagem de erro como `equal (x, y)` poderia fazer nesse caso).

remfun (*f*, *expr*) Função
remfun (*f*, *expr*, *x*) Função

`remfun (`*f*`, `*expr*`)` substitue todas as ocorrências de *f* `(`*arg*`)` por *arg* em *expr*.

`remfun (`*f*`, `*expr*`, `*x*`)` substitue todas as ocorrências de *f* `(`*arg*`)` por *arg* em *expr* somente se *arg* contiver a variável *x*.

funp (*f*, *expr*) Função
funp (*f*, *expr*, *x*) Função

`funp (`*f*`, `*expr*`)` retorna `true` se *expr* contém a função *f*.

`funp (`*f*`, `*expr*`, `*x*`)` retorna `true` se *expr* contém a função *f* e a variável *x* em algum lugar no argumento de uma das instâncias de *f*.

absint (*f*, *x*, *halfplane*) Função
absint (*f*, *x*) Função
absint (*f*, *x*, *a*, *b*) Função

`absint (`*f*`, `*x*`, `*halfplane*`)` retorna a integral indefinida de *f* com relação a *x* no dado semi-plano (`pos`, `neg`, ou `both`). *f* pode conter expressões da forma `abs (x)`, `abs (sin (x))`, `abs (a) * exp (-abs (b) * abs (x))`.

`absint (`*f*`, `*x*`)` é equivalente a `absint (`*f*`, `*x*`, pos)`.

`absint (`*f*`, `*x*`, `*a*`, `*b*`)` retorna a integral definida de *f* com relação a *x* de *a* até *b*. *f* pode incluir valores absolutos.

fourier (*f*, *x*, *p*) Função

Retorna uma lista de coeficientes de Fourier de *f*`(x)` definidos sobre o intervalo `[-%pi, %pi]`.

foursimp (*l*) Função

Simplifica `sin (n %pi)` para 0 se `sinnpiflag` for `true` e `cos (n %pi)` para `(-1)^n` se `cosnpiflag` for `true`.

sinnpiflag Variável de opção

Valor padrão: `true`

Veja `foursimp`.

cosnpiflag Variável de opção

Valor padrão: `true`

Veja `foursimp`.

fourexpand (*l*, *x*, *p*, *limit*) Função

Constrói e retorna a série de Fourier partindo da lista de coeficientes de Fourier *l* até (up through) *limit* termos (*limit* pode ser `inf`). *x* e *p* possuem o mesmo significado que em `fourier`.

fourcos (*f*, *x*, *p*) Função

Retorna os coeficientes do cosseno de Fourier para `f(x)` definida sobre `[0, %pi]`.

foursin (*f*, *x*, *p*) Função

Retorna os coeficientes do seno de Fourier para `f(x)` definida sobre `[0, %pi]`.

totalfourier (*f*, *x*, *p*) Função

Retorna `fourexpand (foursimp (fourier (`*f*`, `*x*`, `*p*`)), `*x*`, `*p*`, 'inf)`.

fourint (*f*, *x*) Função

Constrói e retorna uma lista de coeficientes de integral de Fourier de `f(x)` definida sobre `[minf, inf]`.

fourintcos (*f*, *x*) Função

Retorna os coeficientes da integral do cosseno de Fourier para `f(x)` on `[0, inf]`.

fourintsin (*f*, *x*) Função

Retorna os coeficientes da integral do seno de Fourier para `f(x)` on `[0, inf]`.

25 Estatística

25.1 Definições para Estatística

gauss (*mean*, *sd*) Função

Retorna um número em ponto flutuante randômico de uma distribuição normal com usando *mean* e desvio padrão *sd*.

26 Arrays e Tabelas

26.1 Definições para Arrays e Tabelas

array (*name, dim_1, ..., dim_n*) Função
array (*name, type, dim_1, ..., dim_n*) Função
array ([*name_1, ..., name_m*], *dim_1, ..., dim_n*) Função

Cria um array n-dimensional. n pode ser menor ou igual a 5. Os subscritos para a i'ésima dimensão são inteiros no intervalo de 0 a *dim_i*.

`array (`*name*`, `*dim_1*`, ..., `*dim_n*`)` cria um array genérico.

`array (`*name*`, `*type*`, `*dim_1*`, ..., `*dim_n*`)` cria um array, com elementos de um tipo especificado. *type* pode ser `fixnum` para inteiros de tamanho limitado ou `flonum` para números em ponto flutuante.

`array ([`*name_1*`, ..., `*name_m*`], `*dim_1*`, ..., `*dim_n*`)` cria m arrays, todos da mesma dimensão.

Se o usuário atribui a uma variável subscrita antes de declarar o array correspondente, um array não declarado é criado. Arrays não declarados, também conhecidos como array desordenado (porque o codigo desordenado termina nos subscritos), são mais gerais que arrays declarados. O usuário não declara seu tamanho máximo, e ele cresce dinamicamente e desordenadamente à medida que são atribuídos valores a mais elementos. Os subscritos de um array não declarado não precisam sempre ser números. Todavia, exceto para um array um tanto quanto esparso, é provavelmente mais eficiente declarar isso quando possível que deixar não declarado. A função `array` pode ser usada para transformar um array não declarado em um array declarado.

arrayapply (*A*, [*i_1, ..., i_n*]) Função

Avalia *A* `[`*i_1*`, ..., `*i_n*`]`, quando *A* for um array e *i_1*, ..., *i_n* são inteiros.

Ela é remanescente de `apply`, exceto o primeiro argumento que é um array ao invés de uma função.

arrayinfo (*A*) Função

Retorna uma lista de informações sobre o array *A*. Para arrays desordenados ela retorna uma lista de `hashed`, o números de subscritos, e os subscritos de cada elemento que tem um valor. Para arrays declarados ela retorna uma lista de `declared`, o número de subscritos, e os limites que foram dados à função `array` quando ela foi chamada sobre *A*. Fazer `example(arrayinfo);` por exemplo.

arraymake (*name*, [*i_1, ..., i_n*]) Função

Retorna a expressão *name* `[`*i_1*`, ..., `*i_n*`]`.

Isso é um código remanescente de `funmake`, exceto o valor retornado é um array de referência não avaliado ao invés de uma chamada de função não avaliada.

arrays Variável de sistema

Valor padrão: `[]`

`arrays` é uma lista de todas os arrays que foram alocadas, tanto declarados como não declarados.

Veja também `array`, `arrayapply`, `arrayinfo`, `arraymake`, `fillarray`, `listarray`, and `rearray`.

bashindices (*expr*) Função

Transforma a expressão *expr* dando a cada somatório e a cada produto um único índice. Isso dá a `changevar` grande precisão quando se está trabalhando com somatórios e produtos. A forma do único índice é `j`*number*. A quantidade *number* é determindad por referência a `gensumnum`, que pode ser alterada pelo usuário. Por exemplo, `gensumnum:0$` reseta isso.

fillarray (*A*, *B*) Função

Preenche o array *A* com *B*, que é uma lista ou um array.

Se *A* for um array de ponto flutuante (inteiro) então *B* poderá ser ou uma lista de números (inteiros) em ponto flutuante ou outro array em ponto flutuante (inteiro).

Se as dimensões do array forem diferentes *A* é preenchida na ordem da maior linha. Se não existem elementos livres em *B* o último elemento é usado para preencher todo o resto de *A*. Se existirem muitos os restantes serão descartados.

`fillarray` retorna esse primeiro argumento.

listarray (*A*) Função

Retorna uma lista dos elementos de um array declarado ou desordenado *A*. A ordem é da maior-linha. Elementos que não estão ainda definidos são representados por `#####`.

make_array (*type, dim_1, ..., dim_n*) Função

Cria e retorna um array de Lisp. *type* pode ser `any`, `flonum`, `fixnum`, `hashed` ou `functional`. Existem *n* indices, e o *i*'enésimo indice está no intervalo de 0 a *dim_i* − 1.

A vantagem de `make_array` sobre `array` é que o valor de retorno não tem um nome, e uma vez que um ponteiro a ele vai, ele irá também. Por exemplo, se `y: make_array (...)` então `y` aponta para um objeto que ocupa espaço, mas depois de `y: false`, `y` não mais aponta para aquele objeto, então o objeto pode ser descartado.

`y: make_array ('functional, 'f, 'hashed, 1)` - o segundo argumento para `make_array` nesse caso é a função que chama o cálculo dos elementos do array, e os argumentos restantes são passados recursivamente a `make_array` para gerar a "memoria" para a função array objeto.

rearray (*A*, *dim_1*, ..., *dim_n*) Função

Altera as dimenções de um array. O novo array será preenchido com os elementos do antigo em ordem da maior linha. Se o array antigo era muito pequeno, os elementos restantes serão preenchidos com `false`, `0.0` ou `0`, dependendo do tipo do array. O tipo do array não pode ser alterado.

remarray (*A_1, ..., A_n*) Função
remarray (*all*) Função

Remove arrays e funções associadas a arrays e libera o espaço ocupado.

`remarray (all)` remove todos os ítens na lista global `arrays`.

Isso pode ser necessário para usar essa função se isso é desejado para redefinir os valores em um array desordenado.

`remarray` retorna a lista dos arrays removidos.

use_fast_arrays Variável de pção

- Se `true` somente dois tipos de arrays são reconhecidos.

1) O array art-q (t no Lisp Comum) que pode ter muitas dimensões indexadas por inteiros, e pode aceitar qualquer objeto do Lisp ou do Maxima como uma entrada. Para construir assim um array, insira `a:make_array(any,3,4);` então `a` terá como valor, um array com doze posições, e o índice é baseado em zero.

2) O array Hash_table que é o tipo padrão de array criado se um faz `b[x+1]:y^2` (e `b` não é ainda um array, uma lista, ou uma matriz – se isso ou um desses ocorrer um erro pode ser causado desde `x+1` não poderá ser um subscrito válido para um array art-q, uma lista ou uma matriz). Esses índices (também conhecidos como chaves) podem ser quaisquer objetos. Isso somente pega uma chave por vez a cada vez (`b[x+1,u]:y` ignorará o `u`). A referência termina em `b[x+1] ==> y^2`. Certamente a chave poe ser uma lista , e.g. `b[[x+1,u]]:y` poderá ser válido. Isso é incompatível com os arrays antigos do Maxima, mas poupa recursos.

Uma vantagem de armazenar os arrays como valores de símbolos é que as convenções usuais sobre variáveis locais de uma função aplicam-se a arrays também. O tipo Hash_table também usa menos recursos e é mais eficiente que o velho tipo hashar do Maxima. Para obter comportamento consistente em códigos traduzidos e compilados posicione `translate_fast_arrays` para ser `true`.

27 Matrizes e Álgebra Linear

27.1 Introdução a Matrizes e Álgebra Linear

27.1.1 Ponto

O operador `.` representa multiplicação não comutativa e produto escalar. Quando os operandos são matrizes 1-coluna ou 1-linha `a` e `b`, a expresão `a.b` é equivalente a `sum (a[i]*b[i], i, 1, length(a))`. Se `a` e `b` não são complexos, isso é o produto escalar, também chamado produto interno ou produto do ponto, de `a` e `b`. O produto escalar é definido como `conjugate(a).b` quando `a` e `b` são complexos; `innerproduct` no pacote `eigen` fornece o produto escalar complexo.

Quando os operandos são matrizes mais gerais, o produto é a matriz produto `a` e `b`. O número de linhas de `b` deve ser igual ao número de colunas de `a`, e o resultado tem número de linhas igual ao número de linhas de `a` e número de colunas igual ao número de colunas de `b`.

Para distingüir `.` como um operador aritmético do ponto decimal em um número em ponto flutuante, pode ser necessário deixar espaços em cada lado. Por exemplo, `5.e3` é `5000.0` mas `5 . e3` é `5` vezes `e3`.

Existem muitos sinalizadores que governam a simplificação de expresões envolvendo `.`, a saber `dot`, `dot0nscsimp`, `dot0simp`, `dot1simp`, `dotassoc`, `dotconstrules`, `dotdistrib`, `dotexptsimp`, `dotident`, e `dotscrules`.

27.1.2 Vetores

`vect` é um pacote de funções para análise vetorial. `load ("vect")` chama esse pacote, e `demo ("vect")` permite visualizar uma demonstração.

O pacote de análise vetorial pode combinar e simplificar expresões simbólicas incluindo produtos dos pontos e productos dos x, juntamente com o gradiente, divergencia, torção, e operadores Laplacianos. A distribuição desses operadores sobre adições ou produtos é governada por muitos sinalizadores, como são várias outras expansões, incluindo expansão dentro de componentes em qualquer sistema de coordenadas ortogonais. Existem também funções para derivar o escalar ou vetor potencial de um campo.

O pacote `vect` contém essas funções: `vectorsimp`, `scalefactors`, `express`, `potential`, e `vectorpotential`.

Atenção: o pacote `vect` declara o operador ponto `.` como sendo um operador comutativo.

27.1.3 auto

O pacote `eigen` contém muitas funções devotadas para a computação simbólica de autovalores e autovetores. Maxima chama o pacote automaticamente se uma das funções `eigenvalues` ou `eigenvectors` é invocada. O pacote pode ser chamado explicitamente com `load ("eigen")`.

`demo ("eigen")` mostra uma demonstração das compatibilidades desse pacote. `batch ("eigen")` executa a mesma demonstração, mas sem lembretes de usuário entre sucessivas computações.

As funções no pacote `eigen` são `innerproduct`, `unitvector`, `columnvector`, `gramschmidt`, `eigenvalues`, `eigenvectors`, `uniteigenvectors`, e `similaritytransform`.

27.2 Definições para Matrizes e Álgebra Linear

addcol (*M, list_1, ..., list_n*) Função

Anexa a(s) coluna(s) dadas por uma ou mais listas (ou matrizes) sobre a matriz *M*.

addrow (*M, list_1, ..., list_n*) Função

Anexa a(s) linha(s) dadas por uma ou mais listas (ou matrizes) sobre a matriz *M*.

adjoint (*M*) Função

Retorna a matriz adjunta da matriz *M*.

augcoefmatrix ([*eqn_1, ..., eqn_m*], [*x_1, ..., x_n*]) Função

Retorna a matriz dos coeficientes aumentada para as variáveis *x_1*, ..., *x_n* do sistema de equações lineares *eqn_1*, ..., *eqn_m*. Essa é a matriz dos coeficientes com uma coluna anexada para os termos independentes em cada equação (i.e., esses termos não dependem de *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*) Função

Retorna um polinômio característico para a matriz *M* em relação à variável *x*. Que é, `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*]) Função

Retorna a matriz dos coeficientes para as variáveis *eqn_1*, ..., *eqn_m* do sistema de equações lineares *x_1*, ..., *x_n*.

col (*M*, *i*) Função

Reorna a *i*'ésima coluna da matriz *M*. O valor de retorno é uma matriz.

columnvector (*L*) Função

covect (*L*) Função

Retorna uma matriz de uma coluna e `length` (*L*) linhas, contendo os elementos da lista *L*.

`covect` é um sinônimo para `columnvector`.

`load ("eigen")` chama essa função.

Isso é útil se você quer usar partes das saídas das funções nesse pacote em cálculos matriciais.

Exemplo:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function autovalores
Warning - you are redefining the Macsyma function autovetores
(%i2) columnvector ([aa, bb, cc, dd]);
                               [ aa ]
                               [    ]
                               [ bb ]
(%o2)                          [    ]
                               [ cc ]
                               [    ]
                               [ dd ]
```

conjugate (*x*) Função

Retorna o conjugado complexo de *x*.

```
(%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) Função

Retorna uma cópia da matriz M. Esse é o único para fazer uma copia separada copiando M elemento a elemento.

Note que uma atribuição de uma matriz para outra, como em `m2: m1`, não copia `m1`. Uma atribuição `m2 [i,j]: x` ou `setelmx (x, i, j, m2` também modifica `m1 [i,j]`. criando uma cópia com `copymatrix` e então usando atribução cria uma separada e modificada cópia.

determinant (M) Função

Calcula o determinante de M por um método similar à eliminação de Gauss.

A forma do resultado depende da escolha do comutador `ratmx`.

Existe uma rotina especial para calcular determinantes esparsos que é chamada quando os comutadores `ratmx` e `sparse` são ambos `true`.

detout Variável

Valor padrão: `false`

Quando `detout` é `true`, o determinante de uma matriz cuja inversa é calculada é fatorado fora da inversa.

Para esse comutador ter efeito `doallmxops` e `doscmxops` deveram ambos serem `false` (veja suas transcrições). Alternativamente esses comutadores podem ser dados para `ev` o que faz com que os outros dois sejam escolhidos corretamente.

Exemplo:

```
(%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*) Função

Retorna uma matriz diagonal de tamanho *n* por *n* com os elementos da diagonal todos iguais a *x*. `diagmatrix (n, 1)` retorna uma matriz identidade (o mesmo que `ident (n)`).

n deve avaliar para um inteiro, de outra forma `diagmatrix` reclama com uma mensagem de erro.

x pode ser qualquer tipo de expresão, incluindo outra matriz. Se *x* é uma matriz, isso não é copiado; todos os elementos da diagonal referem-se à mesma instância, *x*.

doallmxops Variável

Valor padrão: `true`

Quando `doallmxops` é `true`, todas as operações relacionadas a matrizes são realizadas. Quando isso é `false` então a escolha de comutadores individuais `dot` governam quais operações são executadas.

domxexpt Variável

Valor padrão: `true`

Quando `domxexpt` é `true`, uma matriz exponencial, `exp (M)` onde *M* é a matriz, é interpretada como uma matriz com elementos `[i,j` iguais a `exp (m[i,j])`. de outra forma `exp (M)` avalia para `exp (`*ev(M)*.

`domxexpt` afeta todas as expresões da forma *base*`^`*expoente* onde *base* é uma expresão assumida escalar ou constante, e *expoente* é uma lista ou matriz.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `true`

Quando `domxmxops` é `true`, todas as operações matriz-matriz ou matriz-lista são realizadas (mas não operações escalar-matriz); se esse comutador é `false` tais operações não são.

domxnctimes Variável de opção

Valor padrão: `false`

Quando `domxnctimes` é `true`, produtos não comutativos de matrizes são realizados.

dontfactor Variável de opção

Valor padrão: `[]`

`dontfactor` pode ser escolhido para uma lista de variáveis em relação a qual fatoração não é para ocorrer. (A lista é inicialmente vazia.) Fatoração também não pegará lugares com relação a quaisquer variáveis que são menos importantes, conforme a hierarquía de variável assumida para a forma expresão racional canônica (CRE), que essas na lista `dontfactor`.

doscmxops Variável de opção

Valor padrão: `false`

Quando `doscmxops` é `true`, operações escalar-matriz são realizadas.

doscmxplus Variável de opção

Valor padrão: `false`

Quando `doscmxplus` é `true`, operações escalar-matriz retornam uma matriz resultado. Esse comutador não é subsomado sob `doallmxops`.

dot0nscsimp Variável de opção

Valor padrão: `true`

Quando `dot0nscsimp` é `true`, um produto não comutativo de zero e um termo não escalar é simplificado para um produto comutativo.

dot0simp Variável de opção

Valor padrão: `true`

Quando `dot0simp` é `true`, um produto não comutativo de zero e um termo escalar é simplificado para um produto não comutativo.

dot1simp Variável de opção

Valor padrão: `true`

Quando `dot1simp` é `true`, um produto não comutativo de um e outro termo é simplificado para um produto comutativo.

dotassoc Variável de opção

Valor padrão: `true`

Quando `dotassoc` é `true`, uma expresão `(A.B).C` simplifica para `A.(B.C)`.

dotconstrules Variável de opção

Valor padrão: `true`

Quando `dotconstrules` é `true`, um produto não comutativo de uma constante e outro termo é simplificado para um produto comutativo. Ativando esse sinalizador efetivamente ativamos `dot0simp`, `dot0nscsimp`, e `dot1simp` também.

dotdistrib Variável de opção

Valor padrão: `false`

Quando `dotdistrib` é `true`, uma expresão `A.(B + C)` simplifica para `A.B + A.C`.

dotexptsimp Variável de opção

Valor padrão: `true`

Quando `dotexptsimp` é `true`, uma expresão `A.A` simplifica para `A^^2`.

dotident Variável de opção

Valor padrão: 1

`dotident` é o valor retornado por `X^^0`.

dotscrules Variável de opção

Valor padrão: `false`

Quando `dotscrules` é `true`, uma expresão `A.SC` ou `SC.A` simplifica para `SC*A` e `A.(SC*B)` simplifica para `SC*(A.B)`.

echelon (M) Função

Retorna a forma escalonada da matriz M, como produzido através da eliminação de Gauss. A forma escalonada calculada de M por operações elementares de linha tais que o primeiro elemento não zero em cada linha na matriz resultante é um número um e os elementos da coluna abaixo do primeiro número um em cada linha são todos zero.

```
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);
                        [  3   7  aa  bb ]
                        [                ]
(%o1)                   [ - 1  8  5   2  ]
                        [                ]
                        [  9   2  11  4  ]
(%i2) echelon (M);
                   [ 1  - 8  - 5      - 2     ]
                   [                          ]
                   [         28       11      ]
                   [ 0   1   --       --      ]
(%o2)              [         37       37      ]
                   [                          ]
                   [              37 bb - 119 ]
                   [ 0   0    1   ----------- ]
                   [              37 aa - 313 ]
```

eigenvalues (M) Função
eivals (M) Função

Retorna uma lista de duas listas contendo os autovalores da matriz M. A primeira sublista do valor de retorno é a lista de autovalores da matriz, e a segunda sublista é a lista de multiplicidade dos autovalores na ordem correspondente.

`eivals` é um sinônimo de `eigenvalues`.

`eigenvalues` chama a função `solve` para achar as raízes do polinômio característico da matriz. Algumas vezes `solve` pode não estar habilitado a achar as raízes do polinômio; nesse caso algumas outras funções nesse pacote (except `innerproduct`, `unitvector`, `columnvector` e `gramschmidt`) não irão trabalhar.

Em alguns casos os autovalores achados por `solve` podem ser expresões complicadas. (Isso pode acontecer quando `solve` retorna uma expresão real não trivial para um autovalor que é sabidamente real.) Isso pode ser possível para simplificar os autovalores usando algumas outras funções.

O pacote `eigen.mac` é chamado automaticamente quando `eigenvalues` ou `eigenvectors` é referenciado. Se `eigen.mac` não tiver sido ainda chamado, `load ("eigen")` chama-o. Após ser chamado, todas as funções e variáveis no pacote estarão disponíveis.

eigenvectors (M) Função
eivects (M) Função

pegam uma matriz M como seu argumento e retorna uma lista de listas cuja primeira sublista é a saída de `eigenvalues` e as outras sublistas são os autovetores da matriz correspondente para esses autovalores respectivamente. Os autovetores e os autovetores unitários da matriz são os autovetores direitos e os autovetores unitários direitos.

`eivects` é um sinônimo para `eigenvectors`.

O pacote `eigen.mac` é chamado automaticamente quando `eigenvalues` ou `eigenvectors` é referenciado. Se `eigen.mac` não tiver sido ainda chamado, `load ("eigen")` chama-o. Após ser chamado, todas as funções e variáveis no pacote estarão disponíveis.

Os sinalizadores que afetam essa função são:

`nondiagonalizable` é escolhido para `true` ou `false` dependendo de se a matriz é não diagonalizável ou diagonalizável após o retorno de `eigenvectors`.

`hermitianmatrix` quando `true`, faz com que os autovetores degenerados da matriz Hermitiana sejam ortogonalizados usando o algorítmo de Gram-Schmidt.

`knowneigvals` quando `true` faz com que o pacote `eigen` assumir que os autovalores da matriz são conhecidos para o usuário e armazenados sob o nome global `listeigvals`. `listeigvals` poderá ser escolhido para uma lista similar à saída de `eigenvalues`.

A função `algsys` é usada aqui para resolver em relação aos autovetores. Algumas vezes se os autovalores estão ausêntes, `algsys` pode não estar habilitado a achar uma solução. Em alguns casos, isso pode ser possível para simplificar os autovalores por primeiro achando e então usando o comando `eigenvalues` e então usando outras funções para reduzir os autovalores a alguma coisa mais simples. Continuando a simplificação, `eigenvectors` pode ser chamada novamente com o sinalizador `knowneigvals` escolhido para `true`.

ematrix (*m, n, x, i, j*) Função

Retorna uma matriz *m* por *n*, todos os elementos da qual são zero exceto para o elemento `[`*i*`,` *j*`]` que é *x*.

entermatrix (*m, n*) Função

Retorna uma matriz *m* por *n*, lendo os elementos interativamente.

Se *n* é igual a *m*, Maxima pergunta pelo tipo de matriz (diagonal, simétrica, anti-simétrica, ou genérica) e por cada elemento. Cada resposta é terminada por um ponto e vírgula `;` ou sinal de dólar `$`.

Se *n* não é igual a *m*, Maxima pergunta por cada elemento.

Os elementos podem ser quaisquer expressões, que são avaliadas. `entermatrix` avalia seus argumentos.

```
(%i1) n: 3$
(%i2) m: entermatrix (n, n)$

Is the matriz  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)$

Matriz 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*) Função
genmatrix (*a, i_2, j_2, i_1*) Função
genmatrix (*a, i_2, j_2*) Função

Retorna uma matriz gerada de *a*, pegando o elemento *a*`[`*i_1*`,`*j_1*`]` como o elemento do canto superior esquerdo e *a*`[`*i_2*`,`*j_2*`]` como o elemento do canto inferior direto da matriz. Aqui *a* é um array (criado por `array` mas não por `make_array`) ou por uma função array.** (Uma funçãO array é criado como outras funções com `:=` ou `define`, mas os argumentos são colocados entre colchêtes em lugar de parêntesis.)

Se *j_1* é omitido, isso é assumido ser igual a *i_1*. Se ambos *j_1* e *i_1* são omitidos, ambos são assumidos iguais a 1.

Se um elemento selecionado `i,j` de um array é indefinido, a matriz conterá um elemento simbólico *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*) Função
gschmit (*x*) Função

Realiza o algorítmo de ortonalização de Gram-Schmidt sobre *x*, seja ela uma matriz ou uma lista de listas. *x* não é modificado por `gramschmidt`.

Se *x* é uma matriz, o algorítmo é aplicado para as linhas de *x*. Se *x* é uma lista de listas, o algorítmo é aplicado às sublistas, que devem ter igual números de elementos. Nos dois casos, o valor de retorno é uma lista de listas, as sublistas das listas são ortogonais e alcançam o mesmo spaço que *x*. Se a dimensão do alcance de *x* é menor que o número de linhas ou sublistas, algumas sublistas do valor de retorno são zero.

`factor` é chamada a cada estágio do algorítmo para simplificar resultados intermediários. Como uma conseqüência, o valor de retorno pode conter inteiros fatorados.

`gschmit` (nota ortográfica) é um sinônimo para `gramschmidt`.

`load ("eigen")` chama essa função.

Exemplo:

```
(%i1) load ("eigen")$
Warning - you are redefining the Macsyma function autovalores
Warning - you are redefining the Macsyma function autovetores
(%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*) Função

`hach` é um implementação algorítmo de programação linear de Hacijan.

`load ("kach")` chama essa função. `demo ("kach")` executa uma demonstração dessa função.

ident (*n*) Função

Retorna uma matriz identidade *n* por *n*.

innerproduct (*x*, *y*) Função

inprod (*x*, *y*) Função

Retorna o produto interno (também chamado produto escalar ou produto do ponto) de *x* e *y*, que são listas de igual comprimento, ou ambas matrizes 1-coluna ou 1-linha de igual comprimento. O valor de retorno é `conjugate (x) . y`, onde `.` é o operador de multiplicação não comutativa.

`load ("eigen")` chama essa função.

`inprod` é um sinônimo para `innerproduct`.

invert (*M*) Função

Retorna a inversa da matriz *M*. A inversa é calculada pelo método adjunto.

Isso permite a um usuário calcular a inversa de uma matriz com entradas bfloat ou polinômios com coeficientes em ponto flutuante sem converter para a forma CRE.

Cofatores são calculados pela função `determinant`, então se `ratmx` é `false` a inversa é calculada sem mudar a representação dos elementos.

A implementação corrente é ineficiente para matrizes de alta ordem.

Quando `detout` é `true`, o determinante é fatorado fora da inversa.

Os elementos da inversa não são automaticamente expandidos. Se *M* tem elementos polinomiais, melhor aparência de saída pode ser gerada por `expand (invert (m)), detout`. Se isso é desejável para ela divisão até pelo determinante pode ser excelente por `xthru (%)` ou alternativamente na unha por

```
expe (adjoint (m)) / expand (determinant (m))
invert (m) := adjoint (m) / determinant (m)
```

Veja `^^` (expoente não comutativo) para outro método de inverter uma matriz.

lmxchar Variável de opção

Valor padrão: `[`

`lmxchar` é o caractere mostrado como o delimitador esquerdo de uma matriz. Veja também `rmxchar`.

Exemplo:

```
(%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*) Função

Retorna uma matriz retangular que tem as linhas *row_1*, ..., *row_n*. Cada linha é uma lista de expressões. Todas as linhas devem ter o mesmo comprimento.

As operações `+` (adição), `-` (subtração), `*` (multiplicação), e `/` (divisão), são realizadas elemento por elemento quando os operandos são duas matrizes, um escalar e uma matriz, ou uma matriz e um escalar. A operação `^` (exponenciação, equivalentemente `**`) é realizada elemento por elemento se os operandos são um escalar e uma matriz ou uma matriz e um escalar, mas não se os operandos forem duas matrizes. Todos as operações são normalmente realizadas de forma completa, incluindo `.` (multiplicação não comutativa).

Multiplicação de matrizes é representada pelo operador de multiplicação não comutativa `..` O correspondente operador de exponenciação não comutativa é `^^`. Para uma matriz A, A`.`A = A`^^2` e A`^^-1` é a inversa de A, se existir.

Existem comutadores para controlar a simplificação de expresões envolvendo operações escalar e matriz-lista. São eles `doallmxops`, `domxexpt` `domxmxops`, `doscmxops`, e `doscmxplus`.

Existem opções adicionais que são relacionadas a matrizes. São elas: `lmxchar`, `rmxchar`, `ratmx`, `listarith`, `detout`, `scalarmatrix`, e `sparse`.

Existe um número de funções que pegam matrizes como argumentos ou devolvem matrizes como valor de retorno. Veja `eigenvalues`, `eigenvectors`, `determinant`, `charpoly`, `genmatrix`, `addcol`, `addrow`, `copymatrix`, `transpose`, `echelon`, e `rank`.

Exemplos:

- Construção de matrizes de listas.

```
(%i1) x: matrix ([17, 3], [-8, 11]);
                              [ 17   3  ]
(%o1)                         [         ]
                              [ - 8  11 ]
(%i2) y: matrix ([%pi, %e], [a, b]);
                              [ %pi  %e ]
(%o2)                         [         ]
                              [  a   b  ]
```

- Adição, elemento por elemento.

```
(%i3) x + y;
                        [ %pi + 17  %e + 3 ]
(%o3)                   [                  ]
                        [  a - 8    b + 11 ]
```

- Subtração, elemento por elemento.

```
(%i4) x - y;
                        [ 17 - %pi  3 - %e ]
(%o4)                   [                  ]
                        [ - a - 8   11 - b ]
```

- Multiplicação, elemento por elemento.

```
(%i5) x * y;
                          [ 17 %pi  3 %e ]
(%o5)                     [              ]
                          [ - 8 a   11 b ]
```

- Divisão, elemento por elemento.

```
(%i6) x / y;
                          [ 17       - 1 ]
                          [ ---  3 %e    ]
                          [ %pi          ]
(%o6)                     [              ]
                          [   8    11    ]
                          [ - -    --    ]
                          [   a    b     ]
```

- Matriz para um expoente escalar, elemento por elemento.

```
(%i7) x ^ 3;
                           [ 4913    27  ]
(%o7)                      [             ]
                           [ - 512  1331 ]
```

- Base escalar para um expoente matriz, elemento por elemento.

```
(%i8) exp(y);
                           [   %pi    %e ]
                           [ %e     %e   ]
(%o8)                      [             ]
                           [    a     b  ]
                           [  %e    %e   ]
```

- Base matriz para um expoente matriz. Essa não é realizada elemento por elemento.

```
(%i9) x ^ y;
                                    [ %pi  %e ]
                                    [         ]
                                    [  a   b  ]
                         [ 17   3  ]
(%o9)                    [         ]
                         [ - 8  11 ]
```

- Multiplicação não comutativa de matrizes.

```
(%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   ]
```

- Exponenciação não comutativa de matrizes. Uma base escalar b para uma potência matriz M é realizada elemento por elemento e então `b^^m` é o mesmo que `b^m`.

```
(%i12) x ^^ 3;
                           [  3833   1719 ]
(%o12)                     [              ]
                           [ - 4584  395  ]
(%i13) %e ^^ y;
                            [   %pi    %e ]
                            [ %e     %e   ]
(%o13)                      [             ]
                            [    a     b  ]
                            [  %e    %e   ]
```

- A matriz elevada a um expoente -1 com exponenciação não comutativa é a matriz inversa, se existir.

```
(%i14) x ^^ -1;
                            [ 11      3  ]
                            [ ---  - --- ]
                            [ 211    211 ]
(%o14)                      [            ]
                            [  8    17   ]
                            [ ---   ---  ]
                            [ 211   211  ]
(%i15) x . (x ^^ -1);
                               [ 1  0 ]
(%o15)                         [      ]
                               [ 0  1 ]
```

matrixmap (*f, M*) Função

Retorna uma matriz com elemento `i,j` igual a *f*(*M*`[i,j]`).

Veja também `map`, `fullmap`, `fullmapl`, e `apply`.

matrixp (*expr*) Função

Retorna `true` se *expr* é uma matriz, de outra forma retorna `false`.

matrix_element_add Variável de opção

Valor padrão: `+`

`matrix_element_add` é a operação invocada em lugar da adição em uma multiplicação de matrizes. A `matrix_element_add` pode ser atribuído qualquer operador n-ário

(que é, uma função que manuseia qualquer número de argumentos). Os valores atribuídos podem ser o nome de um operador entre aspas duplas, o nome da função, ou uma expressão lambda.

Veja também `matrix_element_mult` e `matrix_element_transpose`.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `*`

`matrix_element_mult` é a operação invocada em lugar da multiplicação em uma multiplicação de matrizes. A `matrix_element_mult` pode ser atribuído qualquer operador binário. O valor atribuído pode ser o nome de um operador entre aspas duplas, o nome de uma função, ou uma expressão lambda.

O operador do ponto `.` é uma escolha útil em alguns contextos.

Veja também `matrix_element_add` e `matrix_element_transpose`.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `false`

`matrix_element_transpose` é a operação aplicada a cada elemento de uma matriz quando for uma transposta. A `matrix_element_mult` pode ser atribuído qualquer operador unário. O valor atribuído pode ser nome de um operador entre aspas duplas, o nome de uma função, ou uma expressão lambda.

Quando `matrix_element_transpose` for igual a `transpose`, a função `transpose` é aplicada a todo elemento. Quando `matrix_element_transpose` for igual a `nonscalars`, a função `transpose` é aplicada a todo elemento não escalar. Se algum elemento é um átomo, a opção `nonscalars` aplica `transpose` somente se o átomo for declarado não escalar, enquanto a opção `transpose` sempre aplica `transpose`.

O valor padrão, `false`, significa nenhuma operação é aplicada.

Veja também `matrix_element_add` e `matrix_element_mult`.

Exemplos:

```
(%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*) Função

Retorna o traço (que é, a soma dos elementos sobre a diagonal principal) da matriz quadrada *M*.

`mattrace` é chamada por `ncharpoly`, uma alternativa para `charpoly` do Maxima.

`load ("nchrpl")` chama essa função.

minor (*M*, *i*, *j*) Função

Retorna o *i*, *j* menor do elemento localizado na linha *i* coluna *j* da matriz *M*. Que é *M* com linha *i* e coluna *j* ambas removidas.

ncexpt (*a*, *b*) Função

Se uma expressão exponencial não comutativa é muito alta para ser mostrada como *a*`^^`*b* aparecerá como `ncexpt (`*a*`,`*b*`)`.

`ncexpt` não é o nome de uma função ou operador; o nome somente aparece em saídas, e não é reconhecido em entradas.

ncharpoly (*M*, *x*) Função

Retorna o polinômio característico da matriz *M* com relação a *x*. Essa é uma alternativa para `charpoly` do Maxima.

`ncharpoly` trabalha pelo cálculo dos traços das potências na dada matriz, que são sabidos serem iguais a somas de potências das raízes do polinômio característico. Para essas quantidade a função simétrica das raízes pode ser calculada, que nada mais são que os coeficientes do polinômio característico. `charpoly` trabalha formatando o determinante de *x* `* ident [n] - a`. Dessa forma `ncharpoly` é vencedor, por exemplo, no caso de largas e densas matrizes preencidas com inteiros, desde que isso evite inteiramente a aritmética polinomial.

`load ("nchrpl")` loads this file.

newdet (*M*, *n*) Função

Calcula o determinante de uma matriz ou array *M* pelo algorítmo da árvore menor de Johnson-Gentleman. O argumento *n* é a ordem; isso é optional se *M* for uma matriz.

nonscalar Declaration

Faz átomos ser comportarem da mesma forma que uma lista ou matriz em relação ao operador do ponto.

nonscalarp (*expr*) Função

Retorna `true` se *expr* é um não escalar, i.e., isso contém átomos declarados como não escalares, listas, ou matrizes.

permanent (*M*, *n*) Função

Calcula o permanente da matriz *M*. Um permanente é como um determinante mas sem mudança de sinal.

rank (*M*) Função

Calcula o posto da matriz *M*. Que é, a ordem do mais largo determinante não singular de *M*.

rank pode retornar uma resposta ruim se não puder determinar que um elemento da matriz que é equivalente a zero é realmente isso.

ratmx Variável de opção

Valor padrão: `false`

Quando `ratmx` é `false`, adição, subtração, e multiplicação para determinantes e matrizes são executados na representação dos elementos da matriz e fazem com que o resultado da inversão de matrizes seja esquerdo na representação geral.

Quando `ratmx` é `true`, as 4 operações mencionadas acima são executadas na forma CRE e o resultado da matriz inversa é dado na forma CRE. Note isso pode fazer com que os elementos sejam expandidos (dependendo da escolha de `ratfac`) o que pode não ser desejado sempre.

row (*M*, *i*) Função

retorna a *i*'ésima linha da matriz *M*. O valor de retorno é uma matriz.

scalarmatrixp Variável de opção

Valor padrão: `true`

Quando `scalarmatrixp` é `true`, então sempre que uma matriz 1 x 1 é produzida como um resultado de cálculos o produto do ponto de matrizes é simplificado para um escalar, a saber o elemento solitário da matriz.

Quando `scalarmatrixp` é `all`, então todas as matrizes 1 x 1 serão simplificadas para escalares.

Quando `scalarmatrixp` é `false`, matrizes 1 x 1 não são simplificadas para escalares.

scalefactors (*coordinatetransform*) Função

Aqui coordinatetransform avalia para a forma [[expresão1, expresão2, ...], indeterminação1, indeterminação2, ...], onde indeterminação1, indeterminação2, etc. são as variáveis de coordenadas curvilíneas e onde a escolha de componentes cartesianas retangulares é dada em termos das coordenadas curvilíneas por [expresão1, expresão2, ...]. `coordinates` é escolhida para o vetor [indeterminação1, indeterminação2,...], e `dimension` é escolhida para o comprimento desse vetor. SF[1], SF[2], ..., SF[DIMENSION] são escohidos para fatores de escala de coordenada, e `sfprod` é escohido para o produto desse fatores de escala. Inicialmente, `coordinates` é [X, Y, Z], `dimension` é 3, e SF[1]=SF[2]=SF[3]=SFPROD=1, correspondendo a coordenadas Cartesianas retangulares 3-dimensional. Para expandir uma expresão dentro de componentes físicos no sistema de coordenadas corrente , existe uma função com uso da forma

setelmx (*x*, *i*, *j*, *M*) Função

Atribue *x* para o (*i*, *j*)'ésimo elemento da matriz *M*, e retorna a matriz alterada.

M `[i, j]` : *x* tem o mesmo efeito, mas retorna *x* em lugar de *M*.

similaritytransform (*M*) Função
simtran (*M*) Função

`similaritytransform` calcula uma transformação homotética da matriz `M`. Isso retorna uma lista que é a saída do comando `uniteigenvectors`. Em adição se o sinalizador `nondiagonalizable` é `false` duas matrizes globais `leftmatrix` e `rightmatrix` são calculadas. Essas matrizes possuem a propriedade de `leftmatrix` . *M* . `rightmatrix` é uma matriz diagonal com os autovalores de *M* sobre a diagonal. Se `nondiagonalizable` é `true` as matrizes esquerda e direita não são computadas.

Se o sinalizador `hermitianmatrix` é `true` então `leftmatrix` é o conjugado complexo da transposta de `rightmatrix`. De outra forma `leftmatrix` é a inversa de `rightmatrix`.

`rightmatrix` é a matriz cujas colunas são os autovetores unitários de *M*. Os outros sinalizadores (veja `eigenvalues` e `eigenvectors`) possuem o mesmo efeito desde que `similaritytransform` chama as outras funções no pacote com o objetivo de estar habilitado para a forma `rightmatrix`.

`load ("eigen")` chama essa função.

`simtran` é um sinônimo para `similaritytransform`.

sparse Variável de opção

Valor padrão: `false`

Quando `sparse` é `true`, e se `ratmx` é `true`, então `determinant` usará rotinas especiais para calcular determinantes esparsos.

submatrix (*i_1*, ..., *i_m*, *M*, *j_1*, ..., *j_n*) Função
submatrix (*i_1*, ..., *i_m*, *M*) Função
submatrix (*M*, *j_1*, ..., *j_n*) Função

Retorna uma nova matriz formada pela matrix *M* com linhas *i_1*, ..., *i_m* excluídas, e colunas *j_1*, ..., *j_n* excluídas.

transpose (*M*) Função

Retorna a transposta de *M*.

Se *M* é uma matriz, o valor de retorno é outra matriz *N* tal que `N[i,j] = M[j,i]`.

De outra forma *M* é uma lista, e o valor de retorno é uma matriz *N* de `length (m)` linhas e 1 coluna, tal que `N[i,1] = M[i]`.

triangularize (*M*) Função

Retorna a maior forma triangular da matriz `M`, como produzido através da eliminação de Gauss. O valor de retorno é o mesmo que `echelon`, exceto que o o coeficiente lider não nulo em cada linha não é normalizado para 1.

`lu_factor` e `cholesky` são outras funções que retornam matrizes triangularizadas.

```
(%i1) M: matrix ([3, 7, aa, bb], [-1, 8, 5, 2], [9, 2, 11, 4]);
                          [  3   7  aa  bb ]
                          [                ]
(%o1)                     [ - 1  8  5   2  ]
```

```
                          [                 ]
                          [  9   2  11  4   ]
(%i2) triangularize (M);
                [ - 1   8         5            2      ]
                [                                     ]
(%o2)           [  0   - 74     - 56         - 22     ]
                [                                     ]
                [  0    0    626 - 74 aa  238 - 74 bb ]
```

uniteigenvectors (M) Função
ueivects (M) Função

Calcula autovetores unitários da matriz M. O valor de retorno é uma lista de listas, a primeiro sublista é a saída do comando `eigenvalues`, e as outras sublistas são os autovetores unitários da matriz correspondente a esses autovalores respectivamente.

Os sinalizadores mencionados na descrição do comando `eigenvectors` possuem o mesmo efeito aqui também.

Quando `knowneigvects` é `true`, o pacote `eigen` assume que os autovetores da matriz são conhecidos para o usuário são armazenados sob o nome global `listeigvects`. `listeigvects` pode ser ecolhido para uma lista similar à saída do comando `eigenvectors`.

Se `knowneigvects` é escolhido para `true` e a lista de autovetores é dada a escolha do sinalizador `nondiagonalizable` pode não estar correta. Se esse é o caso por favor ecolha isso para o valor correto. O autor assume que o usuário sabe o que está fazendo e que não tentará diagonalizar uma matriz cujos autovetores não alcançam o mesmo espaço vetorial de dimensão apropriada.

`load ("eigen")` chama essa função.

`ueivects` é um sinônimo para `uniteigenvectors`.

unitvector (x) Função
uvect (x) Função

Retorna $x/norm(x)$; isso é um vetor unitário na mesma direção que x.

`load ("eigen")` chama essa função.

`uvect` é um sinônimo para `unitvector`.

vectorsimp (*expr*) Função

Aplica simplificações e expansões conforme os seguintes sinalizadores globais:

`expandall`, `expanddot`, `expanddotplus`, `expandcross`, `expandcrossplus`, `expandcrosscross`, `expandgrad`, `expandgradplus`, `expandgradprod`, `expanddiv`, `expanddivplus`, `expanddivprod`, `expandcurl`, `expandcurlplus`, `expandcurlcurl`, `expandlaplacian`, `expandlaplacianplus`, e `expandlaplacianprod`.

Todos esses sinalizadores possuem valor padrão `false`. O sufixo `plus` refere-se a utilização aditivamente ou distribuitivamente. O sufixo `prod` refere-se a expansão para um operando que é qualquer tipo de produto.

`expandcrosscross`

Simplifica $p\ (q\ r)$ para $(p.r) * q - (p.q) * r$.

`expandcurlcurl`
: Simplifica *curlcurlp* para *graddivp* + *divgradp*.

`expandlaplaciantodivgrad`
: Simplifica *laplacianp* para *divgradp*.

`expandcross`
: Habilita `expandcrossplus` e `expandcrosscross`.

`expandplus`
: Habilita `expanddotplus`, `expandcrossplus`, `expandgradplus`, `expanddivplus`, `expandcurlplus`, e `expandlaplacianplus`.

`expandprod`
: Habilita `expandgradprod`, `expanddivprod`, e `expandlaplacianprod`.

Esses sinalizadores foram todos declarados `evflag`.

vect_cross Variável de opção

Valor padrão: `false`

Quando `vect_cross` é `true`, isso permite DIFF(X~Y,T) trabalhar onde ~ é definido em SHARE;VECT (onde VECT_CROSS é escolhido para `true`, de qualqeur modo.)

zeromatrix (*m*, *n*) Função

Retorna um matriz *m* por *n*, com todos os elementos sendo zero.

"[" Símbolo especial

"]" Símbolo especial

`[` e `]` marcam o omeço e o fim, respectivamente, de uma lista.

`[` e `]` também envolvem os subscritos de uma lista, array, array desordenado, ou função array.

Exemplos:

```
(%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  := ------
                                 2
                                k  + 1
```

```
                                  k      2
                                       k  + 1
(%i9) g[10];
                                       1
(%o9)                                 ---
                                      101
```

28 Funções Afins

28.1 Definições para Funções Afins

fast_linsolve ([*expr_1, ..., expr_m*], [*x_1, ..., x_n*]) Função

Resolve equações lineares simultâneas *expr_1*, ..., *expr_m* para as variáveis *x_1*, ..., *x_n*. Cada *expr_i* pode ser uma equação ou uma expressão geral; se dada como uma expressão geral, ela tratada como uma equação na forma *expr_i* `= 0`.

O valor de retorno é uma lista de equações da forma `[`*x_1* `=` *a_1*`, ..., `*x_n* `=` *a_n*`]` onde *a_1*, ..., *a_n* são todas livres de *x_1*, ..., *x_n*.

`fast_linsolve` é mais rápido que `linsolve` para sistemas de equações que são esparsas.

grobner_basis ([*expr_1, ..., expr_m*]) Função

Retorna uma base de Groebner para as equações *expr_1*, ..., *expr_m*. A funçã `polysimp` pode então ser usada para simplificar outras funções relativas às equações.

```
grobner_basis ([3*x^2+1, y*x])$

polysimp (y^2*x + x^3*9 + 2) ==> -3*x + 2
```

`polysimp(f)` produz 0 se e somente se *f* está no ideal gerado por *expr_1*, ..., *expr_m*, isto é, se e somente se *f* for uma combinação polinomial dos elementos de *expr_1*, ..., *expr_m*.

set_up_dot_simplifications (*eqns, check_through_degree*) Função

set_up_dot_simplifications (*eqns*) Função

As *eqns* são equações polinomiais em variáveis não comutativas. O valor de `current_variables` é uma lista de variáveis usadas para calcular graus. As equações podem ser homogêneas, em ordem para o procedimento terminar.

Se você checou simplificações de envoltório em `dot_simplifications` acima do grau de *f*, então o seguinte é verdadeiro: `dotsimp (`*f*`)` retorna 0 se e somente se *f* está no ideal gerado pelas equações, i.e., se e somente se *f* for uma combinação polinomial dos elementos das equações.

acima do grau de f, então o seguinte é verdadeiro: `dotsimp (f)` retorna 0 se e somente se *f* está no ideal gerado pelas equações, i.e., se e somente se *f* for uma combinação polinomial dos elementos das equações.

O grau é aquele retornado por `nc_degree`. Isso por sua vez é nfluenciado pelos pesos das variáveis individuais.

declare_weight (*x_1, w_1, ..., x_n, w_n*) Função

Atribui pesos *w_1*, ..., *w_n* to *x_1*, ..., *x_n*, respectivamente. Esses são pesos usados em cálculos `nc_degree`.

nc_degree (*p*) Função

Retorna o grau de um polinômio não comutativo *p*. Veja `declare_weights`.

dotsimp (*f*) Função

Retorna 0 se e somente se *f* for um ideal gerado pelas equações, i.e., se e somente se *f* for uma combinação polinomial dos elementos das equações.

fast_central_elements ([*x_1*, ..., *x_n*], *n*) Função

Se `set_up_dot_simplifications` tiver sido feito previamente, ache o polinômio central nas variáveis *x_1*, ..., *x_n* no grau dado, *n*.

Por exemplo:

```
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*) Função

Verifica as sobreposies através do grau *n*, tendo certeza que você tem regras de simplificaçõ suficiente em cada grau, para `dotsimp` trabalhar corretamente. Esse processo pode ter sua velocidade aumentada se você souber antes de começar souber de qual dimensão do espaço de monômios é. Se ele for de dimensão global finita, então `hilbert` pode ser usada. Se você não conhece as dimensões monomiais, não especifique um `rank_function`. Um opcional terceiro argumento `reset`, `false` diz para não se incomodar em perguntar sobre resetar coisas.

mono ([*x_1*, ..., *x_n*], *n*) Função

Retorna a lista de monômios independentes relativamente à simplificação atual do grau *n* nas variáveis *x_1*, ..., *x_n*.

monomial_dimensions (*n*) Função

Calcula a série de Hilbert através do grau *n* para a algebra corrente.

extract_linear_equations ([*p_1*, ..., *p_n*], [*m_1*, ..., *m_n*]) Função

Faz uma lista dos coeficientes dos polinômios não comutativos *p_1*, ..., *p_n* dos monomios não comutatvos *m_1*, ..., *m_n*. Os coeficientes podem ser escalares. Use `list_nc_monomials` para construir a lista dos monômios.

list_nc_monomials ([*p_1*, ..., *p_n*]) Função

list_nc_monomials (*p*) Função

Retorna uma lista de monômios não comutativos que ocorrem em um polinômio *p* ou em uma lista de polinômios *p_1*, ..., *p_n*.

all_dotsimp_denoms Variável de opção

Valor padrão: `false`

Quando `all_dotsimp_denoms` é uma lista, os denominadores encontrados por `dotsimp` são adicionados ao final da lista. `all_dotsimp_denoms` pode ser iniciado como uma lista vazia `[]` antes chamando `dotsimp`.

Por padrão, denominadores não são coletados por `dotsimp`.

29 itensor

29.1 Introdução a itensor

Maxima implementa a manipulação de tensores simbólicos d dois tipos distintos: manipulação de componentes de tensores (pacote `ctensor`) e manipulação de tensores indiciais (pacote `itensor`).

Note bem: Por favor veja a nota sobre 'nova notação de tensor' abaixo.

Manipulação de componentes de tensores significa que objetos do tipo tensor geométrico são representados como arrays ou matrizes. Operações com tensores tais com contração ou diferenciação covariante são realizadas sobre índices (que ocorrem exatamente duas vezes) repetidos com declarações `do`. Isto é, se executa explicitamente operações sobre as componentes apropriadas do tensor armazenadas em um array ou uma matriz.

Manipulação tensorial de índice é implementada através da representação de tensores como funções e suas covariantes, contravariantes e índices de derivação. Operações com tensores como contração ou diferenciação covariante são executadas através de manipulação dos índices em si mesmos em lugar das componentes para as quais eles correspondem.

Esses dois métodos aproximam-se do tratamento de processos diferenciais, algébricos e analíticos no contexto da geometria de Riemannian possuem várias vantagens e desvantagens as quais se revelam por si mesmas somente apesar da natureza particular e dificuldade dos problemas de usuário. Todavia, se pode ter em mente as seguintes características das duas implementações:

As representações de tensores e de operações com tensores explicitamente em termos de seus componntes tornam o pacote `ctensor` fácil de usar. Especificação da métrica e o cálculo de tensores induzidos e invariantes é direto. Embora todas a capacidade de simplificação poderosa do Maxima está em manusear, uma métrica complexa com intrincada dependência funcional e de coordenadas pode facilmente conduzir a expressões cujo tamanho é excessivo e cuja estrutura está escondida. Adicionalmente, muitos cálculos envolvem expressões intermediárias cujo crescimento fazem com que os programas terminem antes de serem completados. Através da experiência, um usuário pode evitar muitas dessas dificuldade.

O motivo de caminhos especiais através dos quais tensores e operações de tensores são representados em termos de operações simbólicas sobre seus índices, expressões cujas representação de componentes podem ser não gerenciaveis da forma comum podem algumas vezes serem grandemente simplificadas através do uso das rotinas especiais para objetos simétricos em `itensor`. Nesse caminho a estrutura de uma expressão grande pode ser mais transparente. Por outro lado, o motivo da representação indicial especial em `itensor`, faz com que em alguns casos o usuário possa encontrar dificuldade com a especificação da métrica, definição de função, e a avaliação de objetos "indexados" diferenciados.

29.1.1 Nova notação d tensores

Até agora, o pacote `itensor` no Maxima tinha usado uma notação que algumas vezes conduzia a ordenação incorreta de índices. Considere o seguinte, por exemplo:

```
(%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
```

O resultado está incorreto a menos que ocorra ser `a` um tensor simétrico. A razão para isso é que embora `itensor` mantenha corretamente a ordem dentro do conjunto de índices covariantes e contravariantes, assim que um índice é incrementado ou decrementado, sua posição relativa para o outro conjunto de índices é perdida.

Para evitar esse problema, uma nova notação tem sido desenvolvida que mantém total compatibilidade com a notação existente e pode ser usada intercambiavelmente. Nessa notação, índices contravariantes são inseridos na posição apropriada na lista de índices covariantes, mas com um sinal de menos colocado antes. Funções como `contract` e `ishow` estão agora consciente dessa nova notação de índice e podem processar tensores apropriadamente.

Nessa nova notação, o exemplo anterior retorna um resultado correto:

```
(%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
```

Presentemente, o único código que faz uso dessa notação é a função `lc2kdt`. Através dessa notação, a função `lc2kdt` encontra com êxito resultados consistentes como a aplicação do tensor métrico para resolver os símbolos de Levi-Civita sem reordenar para índices numéricos.

Uma vez que esse código é um tipo novo, provavelmente contém erros. Enquanto esse tipo novo não tiver sido testado para garantir que ele não interrompe nada usando a "antiga" notação de tensor, existe uma considerável chance que "novos" tensores irão falhar em interoperar com certas funções ou recursos. Essas falhas serão corrigidas à medida que forem encontradas... até então, seja cuidadoso!

29.1.2 Manipulação de tensores indiciais

o pacote de manipulação de tensores indiciais pode ser chamado através de `load(itensor)`. Demonstações estão também disponíveis: tente `demo(tensor)`. Em `itensor` um tensor é representado como um "objeto indexado" . Um "objeto indexado" é uma função de 3 grupos de índices os quais representam o covariante, o contravariante e o índice de derivação. Os índices covariantes são especificados através de uma lista com o primeiro argumento para o objeto indexado, e os índices contravariantes através de uma lista como segundo argumento. Se o objeto indexado carece de algum desses grupos de índices então a lista vazia [] é fornecida como o argumento correspondente. Dessa forma, `g([a,b],[c])` representa um objeto indexado chamado `g` o qual tem dois índices covariantes `(a,b)`, um índice contravariante (`c`) e não possui índices de derivação.

Os índices de derivação, se estiverem presente, são anexados ao final como argumentos adicionais para a função numérica representando o tensor. Eles podem ser explicitamente especificado pelo usuário ou serem criados no processo de diferenciação com relação a alguma variável coordenada. Uma vez que diferenciação ordinária é comutativa, os índices de derivação são ordenados alfanumericamente, a menos que `iframe_flag` seja escolhida para `true`, indicando que uma moldura métrica está sendo usada. Essa ordenação canônica torna possível para Maxima reconhecer que, por exemplo, `t([a],[b],i,j)` é o mesmo que `t([a],[b],j,i)`. Diferenciação de um objeto indexado com relação a alguma coordenada cujos índices não aparecem como um argumento para o objeto indexado podem normalmente retornar zero. Isso é porque Maxima pode não saber que o tensor representado através do objeto indexado possívelmente depende implicitamente da respectiva coordenada. Pela modificação da função existente no Maxima, `diff`, em `itensor`, Maxima sabe assumir que todos os objetos indexados dependem de qualquer variável de diferenciação a menos que seja declarado de outra forma. Isso torna possível para a convençào de somatório ser extendida para índices derivativos. Pode ser verificado que `itensor` não possui a compatibilidade de incrementar índices derivativos, e então eles são sempre tratados como covariantes.

As seguintes funções estão disponíveis no pacote tensor para manipulação de objetos. Atualmente, com relação às rotinas de simplificação, é assumido que objetos indexados não possuem por padrão propriedades simétricas. Isso pode ser modificado através da escolha da variável `allsym[false]` para `true`, o que irá resultar no tratamento de todos os objetos indexados completamente simétricos em suas listas de índices covariantes e simétricos em suas listas de índices contravariantes.

O pacote `itensor` geralmente trata tensores como objetos opacos. Equações tensoriais são manipuladas baseadas em regras algébricas, especificamente simetria e regras de contração. Adicionalmente, o pacote `itensor` não entende diferenciação covariante, curvatura, e torsão. Cálculos podem ser executados relativamente a um métrica de molduras de movimento, dependendo da escolha para a variável `iframe_flag`.

Uma sessão demonstrativa abaixo mostra como chamar o pacote `itensor`, especificando o nome da métrica, e executando alguns cálculos simples.

```
(%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 Definições para itensor

29.2.1 Gerenciando objetos indexados

entertensor (*nome*) Função

É uma função que, através da linha de comando, permite criar um objeto indexado chamado *nome* com qualquer número de índices de tensores e derivativos. Ou um índice simples ou uma lista de índices (às quais podem ser nulas) são entradas aceitáveis (veja o exemplo sob `covdiff`).

changename (*antigo, novo, expr*) Função

Irá mudar o nome de todos os objetos indexados chamados *antigo* para *novo* em *expr*. *antigo* pode ser ou um símbolo ou uma lista da forma [*nome*, *m*, *n*] nesse caso somente esses objetos indexados chamados *nome* com índice covariante *m* e índice contravariante *n* serão renomeados para *novo*.

listoftens Função

Lista todos os tensores em uma expressão tensorial, incluindo seus índices. 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*) Função

Mostra *expr* com os objetos indexados tendo seus índices covariantes como subscritos e índices contravariantes como sobrescritos. Os índices derivativos são mostrados como subscritos, separados dos índices covariantes por uma vírgula (veja os exemplos através desse documento).

indices (*expr*) Função

Retorna uma lista de dois elementos. O primeiro é uma lista de índices livres em *expr* (aqueles que ocorrem somente uma vez). O segundo é uma lista de indices que ocorrem exatamente duas vezes em *expr* (dummy) como demonstra o seguinte exemplo.

```
(%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]]
```

Um produto de tensores contendo o mesmo índice mais que duas vezes é sintaticamente ilegal. `indices` tenta lidar com essas expressões de uma forma razoável; todavia, quando `indices` é chamada para operar sobre tal uma expressão ilegal, seu comportamento pode ser considerado indefinido.

rename (*expr*) Função
rename (*expr, contador*) Função

Retorna uma expressão equivalente para *expr* mas com índices que ocorrem exatamente duas vezes em cada termo alterado do conjunto `[%1, %2,...]`, se o segundo argumento opcional for omitido. De outra forma, os índices que ocorrem exatamente duas vezes são indexados começando no valor de *contador*. Cada índice que ocorre exatamente duas vezes em um produto será diferente. Para uma adição, `rename` irá operar sobre cada termo na a adição zerando o contador com cada termo. Nesse caminho `rename` pode servir como um simplificador tensorial. Adicionalmente, os índices serão ordenados alfanumericamente (se `allsym` for `true`) com relação a índices covariantes ou contravariantes dependendo do valor de `flipflag`. Se `flipflag` for `false` então os índices serão renomeados conforme a ordem dos índices contravariantes. Se `flipflag` for `true` a renomeação ocorrerá conforme a ordem dos índices contravariantes. Isso muitas vezes ajuda que o efeito combinado dos dois restantes sejam reduzidos a uma expressão de valor um ou mais que um por si mesma.

```
(%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 Variável de Opção

Valor padrão: `false`. Se `false` então os índices irão ser renomeados conforme a ordem dos índices contravariantes, de outra forma serão ordenados conforme a ordem dos índices covariantes.

Se `flipflag` for `false` então `rename` forma uma lista de índices contravariantes na ordem em que forem encontrados da esquerda para a direita (se `true` então de índices contravariantes). O primeiro índice que ocorre exatamente duas vezes na lista é renomeado para `%1`, o seguinte para `%2`, etc. Então a ordenação ocorre após a ocorrência do `rename` (veja o exemplo sob `rename`).

defcon (*tensor_1*) Função
defcon (*tensor_1, tensor_2, tensor_3*) Função

Dado *tensor_1* a propriedade que a contração de um produto do *tensor_1* e do *tensor_2* resulta em *tensor_3* com os índices apropriados. Se somente um argumento, *tensor_1*, for dado, então a contração do produto de *tensor_1* com qualquer objeto indexado tendo os índices apropriados (digamos `my_tensor`) irá retornar como resultado um objeto indexado com aquele nome, i.e. `my_tensor`, e com uma nova escolha de índices refletindo as contrações executadas. Por exemplo, se `imetric:g`, então `defcon(g)` irá implementar o incremento e decremento de índices através da contração com o tensor métrico. Mais de uma `defcon` pode ser dada para o mesmo objeto indexado; o último fornecido que for aplicado a uma contração particular irá ser usado. `contractions` é uma lista de objetos indexados que tenham fornecido propriedades de contrações com `defcon`.

remcon (*tensor_1, ..., tensor_n*) Função
remcon (*all*) Função

Remove todas as propriedades de contração de *tensor_1*, ..., *tensor_n*). `remcon(all)` remove todas as propriedades de contração de todos os objetos indexados.

contract (*expr*) Função

Realiza contrações tensoriais em *expr* a qual pode ser qualquer combinação de adições e produtos. Essa função usa a informação dada para a função `defcon`. Para melhores resultados, `expr` pode ser completamente expandida. `ratexpand` é o meio mais rápido para expandir produtos e expoentes de adições se não existirem variáveis nos denominadores dos termos. O comutador `gcd` pode ser `false` se cancelamentos de máximo divisor comum forem desnecessários.

indexed_tensor (*tensor*) Função

Deve ser executada antes de atribuir componentes para um *tensor* para o qual um valor interno já existe como com `ichr1`, `ichr2`, `icurvature`. Veja o exemplo sob `icurvature`.

components (*tensor, expr*) Função

Permite que se atribua um valor indicial a uma expressão *expr* dando os valores das componentes do *tensor*. Esses são automaticamente substituídos para o tensor mesmo que isso ocorra com todos os seus índices. O tensor deve ser da forma `t([...],[...])` onde qualquer lista pode ser vazia. *expr* pode ser qualquer expressão indexada envolvendo outros objetos com os mesmos índices livres que *tensor*. Quando usada para atribuir valores a um tensor métrico no qual as componentes possuem índices que ocorrem exatamente duas vezes se deve ser cuidadoso para definir esses índices de forma a evitar a geração de índices que ocorrem exatamente duas vezes e que são multiplos. a remoção dessas atribuições é dada para a função `remcomps`.

É importante ter em mente que `components` cuida somente da valência de um tensor, e que ignora completamente qualquer ordenação particular de índices. Dessa forma atribuindo componentes a, digamos, `x([i,-j],[])`, `x([-j,i],[])`, ou `x([i],[j])` todas essas atribuições produzem o mesmo resultado, a saber componentes sendo atribuidas a um tensor chamado `x` com valência `(1,1)`.

Componentes podem ser atribuidas a uma expressão indexada por quatro caminhos, dois dos quais envolvem o uso do comando `components`:

1) Como uma expressão indexada. Por exemplo:

```
(%i2) components(g([],[i,j]),e([],[i])*p([],[j]))$
(%i3) ishow(g([],[i,j]))$
                                      i  j
(%t3)                                 e  p
```

2) Como uma matriz:

```
(%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) Como uma função. Você pode usar uma função Maxima para especificar as componentes de um tensor baseado nesses índices. Por exemplo, os seguintes códigos atribuem `kdelta` a `h` se `h` tiver o mesmo número de índices covariantes e índices contravariantes e nenhum índice derivativo, e atribui `kdelta` a `g` caso as condições anteriores não sejam atendidas:

```
(%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) Usando a compatibilidade dos modelos de coincidência do Maxima, especificamente os comandos `defrule` e `applyb1`:

```
(%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*) Função

Desassocia todos os valores de *tensor* que foram atribuídos com a função `components`.

showcomps Função

Mostra atribuições de componentes de um tensor, feitas usando o comando `components`. Essa função pode ser particularmente útil quando uma matriz é atribuída a um tensor indicial usando `components`, como demonstrado através do seguinte exemplo:

```
(%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
```

O comando `showcomps` pode também mostrar componentes de um tensor de categoria maior que 2.

idummy () Função

Incrementos `icounter` e retorno como seu valor um índice da forma `%n` onde n é um inteiro positivo. Isso garante que índices que ocorrem exatamente duas vezes e

que são necessários na formação de expressões não irão conflitar com índices que já estiverem sendo usados (veja o exemplo sob `indices`).

idummyx Variável

É o prefixo para índices que ocorrem exatamente duas vezes (veja o exemplo sob índices `indices`).

icounter Variável de Opção

valor padrão: [1] determina o sufixo numérico para ser usado na geração do próximo índice que ocorre exatamente duas vezes no pacote tensor. O prefixo é determinado através da opção `idummy` (caractere padrão: %).

kdelta (*L1, L2*) Função

é a função delta generalizada de Kronecker definida no pacote `itensor` com *L1* a lista de índices covariantes e *L2* a lista de índices contravariantes. `kdelta([i],[j])` retorna o delta de Kronecker comum. O comando `ev(expr,kdelta)` faz com que a avaliação de uma expressão contendo `kdelta([],[])` se dê para a dimensão de multiplicação.

No que conduzir a um abuso dessa notação, `itensor` também permite `kdelta` ter 2 covariantes e nenhum contravariante, ou 2 contravariantes e nenhum índice covariante, com efeito fornecendo uma compatibilidade para "matriz unitária" covariante ou contravariante. Isso é estritamente considerado um recurso de programação e não significa implicar que `kdelta([i,j],[])` seja um objeto tensorial válido.

kdels (*L1, L2*) Função

Delta de Kronecker simetrizado, usado em alguns cálculos. Por exemplo:

```
(%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*) Função

é o tensor de permutação (ou de Levi-Civita) que retorna 1 se a lista L consistir de uma permutação par de inteiros, -1 se isso consistir de uma permutação ímpar, e 0 se alguns índices em L forem repetidos.

lc2kdt (*expr*) Função

Simplifica expressões contendo os símbolos de Levi-Civita, convertendo esses para expressões delta de Kronecker quando possível. A principal diferença entre essa função e simplesmente avaliar os simbolos de Levi-Civita é que a avaliação direta muitas vezes resulta em expressões Kronecker contendo índices numéricos. Isso é muitas vezes indesejável como na prevenção de simplificação adicional. A função `lc2kdt` evita esse problema, retornando expressões que são mais facilmente simplificadas com `rename` ou `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
```

A função `lc2kdt` algumas vezes faz uso de tensores métricos. Se o tensor métrico não tiver sido definido previamente com `imetric`, isso resulta em um erro.

```
(%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 Função

Regra de simplificação usada para expressões contendo símbolos não avaliados de Levi-Civita (`levi_civita`). Juntamente com `lc_u`, pode ser usada para simplificar muitas expressões mais eficientemente que a avaliação de `levi_civita`. Por exemplo:

```
(%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 Função

Regra de simplificação usada para expressões contendo símbolos não avaliados de Levi-Civita (`levi_civita`). Juntamente com `lc_u`, pode ser usada para simplificar muitas expressões mais eficientemente que a avaliação de `levi_civita`. Para detalhes, veja `lc_l`.

canten (*expr*) Função

Simplifica *expr* por renomeação (veja `rename`) e permutando índices que ocorrem exatamente duas vezes. `rename` é restrito a adições de produto de tensores nos quais nenhum índice derivativo estiver presente. Como tal isso é limitado e pode somente ser usado se `canform` não for capaz de realizar a simplificação requerida.

A função `canten` retorna um resultado matematicamente correto somente se seu argumento for uma expressão que é completamente simétrica em seus índices. Por essa razão, `canten` retorna um erro se `allsym` não for posicionada em `true`.

concan (*expr*) Função

Similar a `canten` mas também executa contração de índices.

29.2.2 Simetrias de tensores

allsym Variável de Opção

Valor padrão: `false`. Se `true` então todos os objetos indexados são assumidos simétricos em todos os seus índices covariantes e contravariantes. Se `false` então nenhum simétrico de qualquer tipo é assumidos nesses índices. Índices derivativos são sempre tomados para serem simétricos a menos que `iframe_flag` seja escolhida para `true`.

decsym (*tensor, m, n,* [*cov_1, cov_2, ...*], [*contr_1, contr_2, ...*]) Função

Declara propriedades de simetria para *tensor* de covariante *m* e *n* índices contravariantes. As *cov_i* e *contr_i* são pseudofunções expressando relações de simetrias em meio a índices covariante e índices contravariantes respectivamente. Esses são da forma `symoper(`*index_1*`,` *index_2*`,...)` onde `symoper` é um entre `sym`, `anti` ou `cyc` e os *index_i* são inteiros indicando a posição do índice no *tensor*. Isso irá declarar *tensor* para ser simétrico, antisimétrico ou cíclico respectivamente nos *index_i*. `symoper(all)` é também forma permitida que indica todos os índices obedecem à condição de simetria. Por exemplo, dado um objeto `b` com 5 índices covariantes, `decsym(b,5,3,[sym(1,2),anti(3,4)],[cyc(all)])` declara `b` simétrico no seu primeiro e no seu segundo índices e antisimétrico no seu terceiro e quarto índices covariantes, e cíclico em todos de seus índices contravariantes. Qualquer lista de declarações de simetria pode ser nula. A função que executa as simplificações é `canform` como o exemplo abaixo ilustra.

```
(%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*) Função

Remove todas as propriedades de simetria de *tensor* que tem *m* índices covariantes e *n* índices contravariantes.

canform (*expr*) Função

Simplifica *expr* através de mudança de nome de índices que ocorrem exatamente duas vezes e reordenação de todos os índices como ditados pelas condições de simetria impostas sobre eles. Se `allsym` for `true` então todos os índices são assumidos simétricos, de outra forma a informação de simetria fornecida pelas declarações `decsym` irão ser usadas. Os índices que ocorrem exatamente duas vezes são renomeados da mesma maneira que na função `rename`. Quando `canform` é aplicada a uma expressão larga o cálculo pode tomar um considerável montante de tempo. Esse tempo pode ser diminuído através do uso de `rename` sobre a expressão em primeiro lugar. Também veja o exemplo sob `decsym`. Nota: `canform` pode não estar apta a reduzir um expressão completamente para sua forma mais simples embora retorne sempre um resultado matemáticamente correto.

29.2.3 Cálculo de tensores indiciais

diff (*expr*, *v_1*, [*n_1*, [*v_2*, *n_2*] ...]) Função

É a função usual de diferenciação do Maxima que tem sido expandida nessas habilidades para `itensor`. `diff` toma a derivada de *expr* *n_1* vezes com relação a *v_1*, *n_2* vezes com relação a *v_2* , etc. Para o pacote `tensor`, a função tem sido modificada de forma que os *v_i* possam ser inteiros de 1 até o valor da variável `dim`. Isso causará a conclusão da diferenciação com relação ao *v_i*ésimo membro da lista `vect_coords`. Se `vect_coords` for associado a uma variável atômica, então aquela variável subscrita através de *v_i* irá ser usada para a variável de diferenciação. Isso permite que um array de nomes de coordenadas ou nomes subscritos como `x[1]`, `x[2]`, ... sejam usados.

idiff (*expr*, *v_1*, [*n_1*, [*v_2*, *n_2*] ...]) Função

Diferenciação indicial. A menos que `diff`, que diferencia com relação a uma variável independente, `idiff` possa ser usada para diferenciar com relação a uma coordenada. Para um objeto indexado, isso equivale a anexar ao final os *v_i* como índices

derivativos. Subseqüêntemente, índices derivativos irão ser ordenados, a menos que `iframe_flag` seja escolhida para `true`.

`idiff` pode também ser o determinante de um tensor métrico. Dessa forma, se `imetric` tiver sido associada a `G` então `idiff(determinant(g),k)` irá retornar `2*determinant(g)*ichr2([%i,k],[%i])` onde o índice que ocorre exatamente duas vezes `%i` é escolhido apropriadamente.

liediff (*v*, *ten*) Função

Calcula a derivada de Lie da expressão tensorial *ten* com relação ao campo vetorial *v*. *ten* pode ser qualquer expressão tensorial indexada; *v* pode ser o nome (sem índices) de um campo vetorial. Por exemplo:

```
(%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*) Função

Avalia todas as ocorrências do comando `idiff` na expressão tensorial *ten*.

undiff (*expr*) Função

Retorna uma expressão equivalente a *expr* mas com todas as derivadas de objetos indexados substituídas pela forma substantiva da função `idiff`. Seu argumento pode retornar aquele objeto indexado se a diferenciação for concluída. Isso é útil quando for desejado substituir um objeto indexado que sofreu diferenciação com alguma definição de função resultando em *expr* e então concluir a diferenciação através de digamos `ev(`*expr*`, idiff)`.

evundiff Função

Equivalente à execução de `undiff`, seguida por `ev` e `rediff`.

O ponto dessa operação é facilmente avaliar expressões que não possam ser diretamente avaliadas na forma derivada. Por exemplo, o seguinte causa um erro:

```
(%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.
```

Todavia, se `icurvature` é informado em sua forma substantiva, pode ser avaliado usando `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
```

Nota: Em versões anteriores do Maxima, formas derivadas dos símbolos de Christoffel também não podiam ser avaliadas. Isso foi corrigido atualmente, de forma que `evundiff` não mais é necessária para expressões como essa:

```
(%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, ...*) Função

Escolhe para zero, em *expr*, todas as ocorrências de *tensor_i* que não tiverem índices derivativos.

flushd (*expr, tensor_1, tensor_2, ...*) Função

Escolhe para zero, em *expr*, todas as ocorrências de *tensor_i* que tiverem índices derivativos.

flushnd (*expr, tensor, n*) Função

Escolhe para zero, em *expr*, todas as ocorrências do objeto diferenciado *tensor* que tem *n* ou mais índices derivativos como demonstra o seguinte exemplo.

```
(%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, ...*) Função

Dados os *tensor_i* a propriedade de diferenciação da coordenada que a derivada do vetor contravariante cujo nome é um dos *tensor_i* retorna um delta de Kronecker. Por exemplo, se `coord(x)` tiver sido concluída então `idiff(x([],[i]),j)` fornece `kdelta([i],[j])`. `coord` que é uma lista de todos os objetos indexados tendo essa propriedade.

remcoord (*tensor_1, tensor_2, ...*) Função

remcoord (*all*) Função

Remove a propriedade de coordenada de diferenciação dos `tensor_i` que foram estabelecidos através da função `coord`. `remcoord(all)` remove essa propriedade de todos os objetos indexados.

makebox (*expr*) Função

Mostra *expr* da mesma maneira que `show`; todavia, qualquer tensor d'Alembertiano ocorrendo em *expr* irá ser indicado usando o símbolo `[]`. Por exemplo, `[]p([m],[n])` representa `g([],[i,j])*p([m],[n],i,j)`.

conmetderiv (*expr, tensor*) Função

Simplifica expressões contendo derivadas comuns de ambas as formas covariantes e contravariantes do tensor métrico (a restrição corrente). Por exemplo, `conmetderiv` pode relatar a derivada do tensor contravariante métrico com símbolos de Christoffel como visto adiante:

```
(%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*]) Função

Simplifica expressões contendo produtos de derivadas de tensores métricos. Especificamente, `simpmetderiv` reconhece duas identidades:

```
  ab         ab           ab                  a
 g   g   + g   g     = (g   g  )   = (kdelta )   = 0
  ,d  bc        bc,d         bc ,d          c ,d
```

conseqüêntemente

```
  ab          ab
 g   g   = - g   g
  ,d  bc          bc,d
```

e

```
  ab          ab
 g   g     = g   g
  ,j  ab,i    ,i  ab,j
```

que seguem de simetrias de símbolos de Christoffel.

A função `simpmetderiv` toma um parmetro opcional que, quando presente, faz com que a função pare após a primeira substituição feita com sucesso em uma expressão produto. A função `simpmetderiv` também faz uso da variável global *flipflag* que determina como aplicar uma ordenação "canonica" para os índices de produto.

Colocados juntos, essas compatibilidades podem ser usadas poderosamente para encontrar simplificações que são difíceis ou impossíveis de realizar de outra forma. Isso é demonstrado através do seguinte exemplo que explicitamente usa o recurso de simplificação parcial de `simpmetderiv` para obter uma expressão contractível:

```
(%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
```

Veja também `weyl.dem` para um exemplo que usa `simpmetderiv` e `conmetderiv` juntos para simplificar contrações do tensor de Weyl.

flush1deriv (*expr*, *tensor*) Função

Escolhe para zero, em `expr`, todas as ocorrências de `tensor` que possuem exatamente um índice derivativo.

29.2.4 Tensores em espaços curvos

imetric (g) Função

Especifica a métrica através de atribuição à variável `imetric`:g adicionalmente, as propriedades de contração da métrica g são escolhidas através da execução dos comandos `defcon(`g`)`,`defcon(`g`,`g`,kdelta)`. A variável `imetric`, valor padrão: [], está associada à métrica, atribuida pelo comando `imetric(`g`)`.

idim (n) Função

Escolhe as dimensões da métrica. Também inicializa as propriedades de antisimetria dos símbolos de Levi-Civita para as dimensões dadas.

ichr1 ([i, j, k]) Função

Retorna o símbolo de Christoffel de primeiro tipo via definição

```
(g      + g      - g     )/2 .
  ik,j     jk,i     ij,k
```

Para avaliar os símbolos de Christoffel para uma métrica particular, à variável `imetric` deve ser atribuída um nome como no exemplo sob `chr2`.

ichr2 ([i, j], [k]) Função

Retorna o símbolo de Christoffel de segundo tipo definido pela relação

```
                          ks
ichr2([i,j],[k]) = g    (g      + g      - g     )/2
                           is,j     js,i     ij,s
```

icurvature ([*i*, *j*, *k*], [*h*]) Função

Retorna o tensor da curvatura de Riemann em termos de símbolos de Christoffel de segundo tipo (`ichr2`). A seguinte notação é usada:

```
                 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*, ...) Função

Retorna a derivada da covariante de *expr* com relação às variáveis *v_i* em termos de símbolos de Christoffel de segundo tipo (`ichr2`). Com o objetivo de avaliar esses, se pode usar `ev(`*expr*`,ichr2)`.

```
(%i1) load(itensor);
(%o1)      /share/tensor/itensor.lisp
(%i2) entertensor()$
Enter tensor name: a;
Enter a list of the índices covariantes: [i,j];
Enter a list of the índices contravariantes: [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
(%i4) imetric:g;
(%o4)         g
(%i5) ishow(ev(%th(2),ichr2))$
 %1 %4  k
g      a     (g       - g       + g  )
i %1   s %4,j  j s,%4    j %4,s
(%t5) - ----------------------------------------
    2
    %1 %3  k
   g   a     (g       - g    + g     )
   %1 j   s %3,i    i s,%3    i %3,s
 - ----------------------------------------
       2
    k %2  %1
   g     a    (g        - g    + g      )
  i j   s %2,%1    %1 s,%2    %1 %2,s   k
 + ---------------------------------------- + a
2   i j,s
(%i6)
```

lorentz_gauge (*expr*) Função

Impõe a condição de Lorentz através da substituição de 0 para todos os objetos indexados em *expr* que possui um índice de derivada idêntico ao índice contravariante.

igeodesic_coords (*expr*, *nome*) Função

Faz com que símbolos de Christoffel não diferenciados e a primeira derivada do tensor métrico tendam para zero em *expr*. O *nome* na função `igeodesic_coords` refere-se à métrica *nome* (se isso aparecer em *expr*) enquando os coeficientes de conecção devem ser chamados com os nomes `ichr1` e/ou `ichr2`. O seguinte exemplo demonstra a verificação da identidade cíclica satisfeita através do tensor da curvatura de Riemann usando a função `igeodesic_coords`.

```
(%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 Molduras móveis

Maxima atualmente tem a habilidade de executar cálculos usando molduras móveis. Essas podem ser molduras ortonormais (tetrads, vielbeins) ou uma moldura arbitrária.

Para usar molduras, você primeiro escolhe `iframe_flag` para `true`. Isso faz com que os símbolos de Christoffel, `ichr1` e `ichr2`, sejam substituídos pelas molduras mais gerais de coeficientes de conecção `icc1` e `icc2` em cálculos. Especialmente, o comportamento de `covdiff` e `icurvature` são alterados.

A moldura é definida através de dois tensores: o campo de moldura inversa (`ifri`), a base tetrad dual), e a métrica da moldura `ifg`. A métrica da moldura é a matriz identidade para molduras ortonormais, ou a métrica de Lorentz para molduras ortonormais no espaço-tempo de Minkowski. O campo de moldura inversa define a base da moldura (vetores

unitários). Propriedades de contração são definidas para o campo de moldura e para a métrica da moldura.

Quando `iframe_flag` for `true`, muitas expressões `itensor` usam a métrica da moldura `ifg` em lugar da métrica definida através de `imetric` para o decremento e para o incremento de índices.

IMPORTANTE: Escolhendo a variável `iframe_flag` para `true` NÃO remove a definição das propriedades de contração de uma métrica definida através de uma chamada a `defcon` ou `imetric`. Se um campo de moldura for usado, ele é melhor para definir a métrica através de atribuição desse nome para a variável `imetric` e NÃO invoque a função `imetric`.

Maxima usa esses dois tensores para definir os coeficientes de moldura (`ifc1` e `ifc2`) cuja forma parte dos coeficientes de conecção (`icc1` e `icc2`), como demonstra o seguinte exemplo:

```
(%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
```

Um método alternativo é usado para calcular o suporte da moldura (`ifb`) se o sinalizador `iframe_bracket_form` é escolhido para `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 () Função

Uma vez que nessa versão do Maxima, identidades de contração para `ifr` e `ifri` são sempre definidas, como é o suporte da moldura (`ifb`), essa função não faz nada.

ifb Variável

O suporte da moldura. A contribuição da métrica da moldura para os coeficientes de conecção é expressa usando o suporte da moldura:

```
                - ifb      + ifb      + ifb
                     c a b      b c a      a b c
ifc1    = ---------------------------------
    abc                   2
```

O suporte da moldura por si mesmo é definido em termos de campo de moldura e métrica da moldura. Dois métodos alternativos de cálculo são usados dependendo do valor de `frame_bracket_form`. Se `true` (o padrão) ou se o sinalizador `itorsion_flag` for `true`:

```
          d      e                                        f
ifb =  ifr    ifr   (ifri      - ifri      - ifri    itr   )
   abc    b      c       a d,e       a e,d       a f    d e
```

Otherwise:

```
             e      d        d      e
ifb    = (ifr    ifr    - ifr    ifr   ) ifri
   abc       b      c,e      b,e    c        a d
```

icc1 Variável

Coeficientes de conecção de primeiro tipo. Em `itensor`, definido como

```
icc1    = ichr1    - ikt1    - inmc1
    abc        abc       abc        abc
```

Nessa expressão, se `iframe_flag` for `true`, o símbolo de Christoffel `ichr1` é substituído com o coeficiente de conecção da moldura `ifc1`. Se `itorsion_flag` for `false`, `ikt1` será omitido. `ikt1` é também omitido se uma base de moldura for usada, como a torsão está já calculada como parte do suporte da moldura. Ultimamente, como `inonmet_flag` é `false`, `inmc1` não estará presente.

icc2 Variável

Coeficientes de conecção de segundo tipo. Em `itensor`, definido como

```
         c         c        c         c
```

```
icc2    = ichr2    - ikt2    - inmc2
    ab         ab        ab         ab
```

Nessa expressão, se `iframe_flag` for `true`, o símbolo de Christoffel `ichr2` é substituído com o coeficiente de conecção `ifc2`. Se `itorsion_flag` for `false`, `ikt2` será omitido. `ikt2` também será omitido se uma base de moldura for usada, uma vez que a torsão já está calculada como parte do suporte da moldura. Ultimamente, como `inonmet_flag` é `false`, `inmc2` não estará presente.

ifc1 Variável

Coeficiente de moldura de primeiro tipo (também conhecido como coeficientes de rotação de Ricci). Esse tensor representa a contribuição da métrica da moldura para o coeficiente de conecção de primeiro tipo. Definido como:

```
             - ifb      + ifb      + ifb
                  c a b      b c a      a b c
ifc1    = ---------------------------------
    abc                  2
```

ifc2 Variável

Coeficiente de moldura de primeiro tipo. Esse tensor representa a contribuição da métrica da moldura para o coeficiente de conecção de primeiro tipo. Definido como uma permutação de suporte de moldura (`ifb`) com os índices apropriados incrementados e decrementados como necessário:

```
    c       cd
ifc2   = ifg   ifc1
    ab             abd
```

ifr Variável

O campo da moldura. Contrai (`ifri`) para e com a forma do campo inverso da moldura para formar a métrica da moldura (`ifg`).

ifri Variável

O campo inverso da moldura. Especifica a base da moldura (vetores base duais). Juntamente com a métrica da moldura, forma a base de todos os cálculos baseados em molduras.

ifg Variável

A métrica da moldura. O valor padrão é `kdelta`, mas pode ser mudada usando `components`.

ifgi Variável

O inverso da métrica da moldura. Contrai com a métrica da moldura (`ifg`) para `kdelta`.

iframe_bracket_form Variável de Opção

Especifica como o suporte da moldura (`ifb`) é calculado. O valor padrão é `true`.

29.2.6 Torsão e não metricidade

Maxima pode trabalhar com torsão e não metricidade. Quando o sinalizador `itorsion_flag` for escolhido para `true`, a contribuição de torsão é adicionada aos coeficientes de conecção. Similarmente, quando o sinalizador `inonmet_flag` for `true`, componentes de não metricidades são incluídos.

inm Variável

O vetor de não metricidade. Conforme a não metricidade está definida através da derivada covariante do tensor métrico. Normalmente zero, o tensor da métrica derivada covariante irá avaliar para o seguinte quando `inonmet_flag` for escolhido para `true`:

```
g     =- g  nm
 ij;k     ij  k
```

inmc1 Variável

Permutação covariante de componentes do vetor de não metricidade. Definida como

```
                g   inm  - inm  g   - g   inm
                 ab    c      a  bc    ac    b
inmc1     = ------------------------------
     abc                 2
```

(Substitue `ifg` em lugar de `g` se uma moldura métrica for usada.)

inmc2 Variável

Permutação covariante de componentes do vetor de não metricidade. Usada nos coeficicientes de conecção se `inonmet_flag` for `true`. Definida como:

```
                        c         c         cd
             -inm  kdelta  - kdelta  inm  + g   inm  g
     c         a       b         a    b          d  ab
inmc2   = ----------------------------------------------
     ab                       2
```

(Substitue `ifg` em lugar de `g` se uma moldura métrica for usada.)

ikt1 Variável

Permutação covariante do tensor de torsão (também conhecido como contorsão). Definido como:

```
                  d           d       d
           -g   itr  - g    itr   - itr   g
             ad    cb    bd    ca      ab  cd
ikt1    = ---------------------------------
    abc                  2
```

(Substitue `ifg` em lugar de `g` se uma moldura métrica for usada.)

ikt2 Variável

Permutação contravariante do tensor de torsão (também conhecida como contorsão). Definida como:

```
    c     cd
ikt2   = g   ikt1
    ab           abd
```

(Substitue `ifg` em lugar de `g` se uma moldura métrica for usada.)

itr Variável

O tensor de torsão. Para uma métrica com torsão, diferenciação covariante repetida sobre uma funçào escalar não irá comutar,como demonstrado através do seguinte exemplo:

```
(%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 Álgebra exterior

O pacote `itensor` pode executar operações sobre campos tensores covariantes totalmente antisimétricos. Um campo tensor totalmente antisimétrico de classe (0,L) corresponde a uma forma diferencial L. Sobre esses objetos, uma operação de multiplicação funciona como um produto exterior, ou produto cunha, é definido.

Desafortunadamente, nem todos os autores concordam sobre a definição de produto cunha. Alguns autores preferem uma definição que corresponde à noção de antisimetrização: nessas palavras, o produto cunha de dois campos vetoriais, por exemplo, pode ser definido como

```
                a a  - a a
                 i j    j i
    a  /\ a  = -----------
     i     j        2
```

Mais geralmente, o produto de uma forma p e uma forma q pode ser definido como

```
                          1     k1..kp l1..lq
  A       /\ B       = ------ D              A       B
   i1..ip     j1..jq   (p+q)!  i1..ip j1..jq  k1..kp  l1..lq
```

onde `D` simboliza o delta de Kronecker.

Outros autores, todavia, preferem uma definição "geométrica" que corresponde à notação de elemento volume:

```
    a  /\ a  = a a  - a a
     i     j    i j    j i
```

e, no caso geral

```
                        1    k1..kp l1..lq
A       /\ B       = ----- D              A       B
 i1..ip     j1..jq   p! q!  i1..ip j1..jq  k1..kp  l1..lq
```

Uma vez que `itensor` é um pacote de algebra de tensores, a primeira dessas duas definições aparenta ser a mais natural por si mesma. Muitas aplicações, todavia, usam a segunda definição. Para resolver esse dilema, um sinalizador tem sido implementado que controla o comportamento do produto cunha: se `igeowedge_flag` for `false` (o padrão), a primeira, definição "tensorial" é usada, de outra forma a segunda, definição "geométrica" irá ser aplicada.

"~" Operator

O operador do produto cunha é definido como sendo o acento til `~`. O til é um operador binário. Seus argumentos podem ser expressões envolvendo escalares, tensores covariantes de categoria 1, ou tensores covariantes de categoria `l` que tiverem sido declarados antisimétricos em todos os índices covariantes.

O comportamento do operador do produto cunha é controlado através do sinalizador `igeowedge_flag`, como no seguinte exemplo:

```
(%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

A barra vertical `|` denota a operação binária "contração com um vetor". Quando um tensor covariante totalmente antisimétrico é contraído com um vetor contravariante, o resultado é o mesmo independente de qual índice foi usado para a contração. Dessa forma, é possível definir a operação de contração de uma forma livre de índices.

No pacote `itensor`, contração com um vetor é sempre realizada com relação ao primeiro índice na ordem literal de ordenação. Isso garante uma melhor simplificação de expressões envolvendo o operador `|`. Por exemplo:

```
(%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 que isso é essencial que os tensores usado como o operador `|` seja declarado totalmente antisimétrico em seus índices covariantes. De outra forma, os resultados serão incorretos.

extdiff (*expr, i*) Função

Calcula a derivada exterior de *expr* com relação ao índice *i*. A derivada exterior é formalmente definida como o produto cunha do operador de derivada parcial e uma forma diferencial. Como tal, essa operação é também controlada através da escolha de `igeowedge_flag`. Por exemplo:

```
(%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*) Função

Calcula o Hodge dual de *expr*. Por exemplo:

```
(%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 Variável de Opção

Controla o comportamento de produto cunha e derivada exterior. Quando for esconhida para `false` (o padrão), a noção de formas diferenciais irá corresponder àquela de um campo tensor covariante totalmente antisimétrico. Quando escolhida para `true`, formas diferenciais irão concordar com a noção do elemento volume.

29.2.8 Exportando expressões TeX

O pacote `itensor` fornece suporte limitado à exportação de expressões de tensores para o TeX. Uma vez que expressões `itensor` aparecem como chamada a funções, o comando regular `tex` do Maxima não produzirá a saída esperada. Você pode tentar em seu lugar o comando `tentex`, o qual tenta traduzir expressões de tensores dentro de objetos TeX indexados apropriadamente.

tentex (*expr*) Função

Para usar a função `tentex`, você deve primeiro chamar `tentex`, como no seguinte exemplo:

```
(%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 o uso da declaração `idummyx`, para evitar o aparecimento do sinal de porcentagem na expressão TeX, o qual pode induzir a erros de compilação.

Note Bem: Essa vesão da função `tentex` é um tanto quanto experimental.

29.2.9 Interagindo com o pacote `ctensor`

O pacote `itensor` possui a habilidade de gerar código Maxima que pode então ser executado no contexto do pacote `ctensor`. A função que executa essa tarefa é `ic_convert`.

ic_convert (*eqn*) Função

Converte a equação *eqn* na sintaxe `itensor` para uma declaração de atribuição `ctensor`. Adições implícitas sobre índices que ocorrem exatamente duas vezes são tornadas explícitas enquanto objetos indexados são transformados em arrays (os arrays subscritos estão na ordem de covarincia seguidos de índices contravariantes dos objetos indexados). A derivada de um objeto indexado irá ser substituída pela forma substantiva de `diff` tomada com relação a `ct_coords` subscrita pelo índice de derivação. Os símbolos de Christoffel `ichr1` e `ichr2` irão ser traduzidos para `lcs` e `mcs`, respectivamente e se `metricconvert` for `true` então todas as ocorrências da métrica com dois índices covariantes (ou contravariantes) irão ser renomeadas para `lg` (ou `ug`). Adicionalmente, ciclos `do` irão ser introduzidos adicionando sobre todos os índices livres de forma que a declaração de atribuição transformada pode ser avaliada através de apenas fazendo `ev`. Os seguintes exemplos demonstam os recursos dessa função.

```
(%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 Palavras reservadas

As palavras seguintes do Maxima são usadas internamente pelo pacote `itensor` e não podem ser redefinidas:

Keyword	Comments
indices2()	versão interna de indices()
conti	Lista de índices contravariantes
covi	Lista de índices covariantes de um objeto indexado
deri	Lista de índices de derivada de um objeto indexado
name	Retorna o nome de um objeto indexado
concan	
irpmon	
lc0	
_lc2kdt0	
_lcprod	
_extlc	

30 ctensor

30.1 Introdução a ctensor

`ctensor` é um pacote de manipulação de componentes. Para usar o pacote `ctensor`, digite `load(ctensor)`. Para começar uma sessão iterativa com `ctensor`, digite `csetup()`. Você é primeiramente solicitado a especificar a dimensão a ser manipulada. Se a dimensão for 2, 3 ou 4 então a lista de coordenadas padrão é `[x,y]`, `[x,y,z]` ou `[x,y,z,t]` respectivamente. Esses nomes podem ser mudados através da atribuição de uma nova lista de coordenadas para a variável `ct_coords` (descrita abaixo) e o usuário é perguntado sobre isso.Cuidado deve ser tomado para evitar o conflito de nomes de coordenadas com outras definições de objetos.

No próximo passo, o usuário informa a métrica ou diretamente ou de um arquivo especificando sua posição ordinal. Como um exemplo de um arquivo de métrica comum, veja `share/tensor/metrics.mac`. A métrica está armazenada na matriz LG. Finalmente, o inverso da métrica é calculado e armazenado na matriz UG. Se tem a opção de realizar todos os cálculos em séries de potência.

Um protocolo amostra é iniciado abaixo para a métrica estática, esfericamente simétrica (coordenadas padrão) que será aplicadas ao problema de derivação das equações de vácuo de Einstein (que levam à solução de Schwarzschild) como um exemplo. Muitas das funções em `ctensor` irão ser mostradas para a métrica padrão como exemplos.

```
(%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 Definições para ctensor

30.2.1 Inicialização e configuração

csetup () Função

É uma função no pacote `ctensor` (component tensor) que inicializa o pacote e permite ao usuário inserir uma métrica interativamente. Veja `ctensor` para mais detalhes.

cmetric (*dis*) Função
cmetric () Função

É uma função no pacote `ctensor` que calcula o inverso da métrica e prepara o pacote para cálculos adiante.

Se `cframe_flag` for `false`, a função calcula a métrica inversa `ug` a partir da matriz `lg` (definida pelo usuário). O determinante da métrica é também calculado e armazenado na variável `gdet`. Mais adiante, o pacote determina se a métrica é diagonal e escolhe o valor de `diagmetric` conforme a determinação. Se o argumento opcional *dis* estiver presente e não for `false`, a saída é mostrada ao usuário pela linha de comando para que ele possa ver o inverso da métrica.

Se `cframe_flag` for `true`, a função espera que o valor de `fri` (a matriz moldura inversa) e `lfg` (a métrica da moldura) sejam definidas. A partir dessas, a matriz da moldura `fr` e a métrica da moldura inversa `ufg` são calculadas.

ct_coordsys (*sistema_de_coordenadas*, *extra_arg*) Função
ct_coordsys (*sistema_de_coordenadas*) Função

Escolhe um sistema de coordenadas predefinido e uma métrica. O argumento *sistema_de_coordenadas* pode ser um dos seguintes símbolos:

```
SYMBOL               Dim Coordenadas       Descrição/comentários
-------------------------------------------------------------------------
cartesian2d           2  [x,y]             Sist. de coord. cartesianas 2D
polar                 2  [r,phi]           Sist. de coord. Polare
elliptic              2  [u,v]
confocalelliptic      2  [u,v]
bipolar               2  [u,v]
parabolic             2  [u,v]
cartesian3d           3  [x,y,z]           Sist. de coord. cartesianas 3D
```

```
polarcylindrical      3  [r,theta,z]
ellipticcylindrical   3  [u,v,z]          Elíptica 2D com Z cilíndrico
confocalellipsoidal   3  [u,v,w]
bipolarcylindrical    3  [u,v,z]          Bipolar 2D com Z cilíndrico
paraboliccylindrical  3  [u,v,z]          Parabólico 2D com Z cilíndrico
paraboloidal          3  [u,v,phi]
conical               3  [u,v,w]
toroidal              3  [u,v,phi]
spherical             3  [r,theta,phi]    Sist. de coord. Esféricas
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]        Sist. de coord. 4D
spherical4d           4  [r,theta,eta,phi]
exteriorschwarzschild 4  [t,r,theta,phi]  Métrica de Schwarzschild
interiorschwarzschild 4  [t,z,u,v]       Métrica de Schwarzschild Interior
kerr_newman           4  [t,r,theta,phi]  Métrica simétrica axialmente alte
```

`sistema_de_coordenadas` pode também ser uma lista de funções de transformação, seguida por uma lista contendo as varáveis coordenadas. Por exemplo, você pode especificar uma métrica esférica como segue:

```
(%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
```

Funções de transformação podem também serem usadas quando `cframe_flag` for `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) ]
```

O argumento opcional *extra_arg* pode ser qualquer um dos seguintes:

`cylindrical` diz a `ct_coordsys` para anexar uma coordenada adicional cilíndrica.

`minkowski` diz a `ct_coordsys` para anexar uma coordenada com assinatura métrica negativa.

`all` diz a `ct_coordsys` para chamar `cmetric` e `christof(false)` após escolher a métrica.

Se a variável global `verbose` for escolhida para `true`, `ct_coordsys` mostra os valores de `dim`, `ct_coords`, e ou `lg` ou `lfg` e `fri`, dependendo do valor de `cframe_flag`.

init_ctensor () Função

Inicializa o pacote `ctensor`.

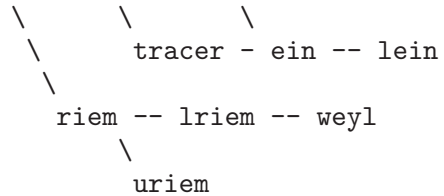
A função `init_ctensor` reinicializa o pacote `ctensor`. Essa função remove todos os arrays e matrizes usados por `ctensor`, coloca todos os sinalizadores de volta a seus valores padrão, retorna `dim` para 4, e retorna a métrica da moldura para a métrica da moldura de Lorentz.

30.2.2 Os tensores do espaço curvo

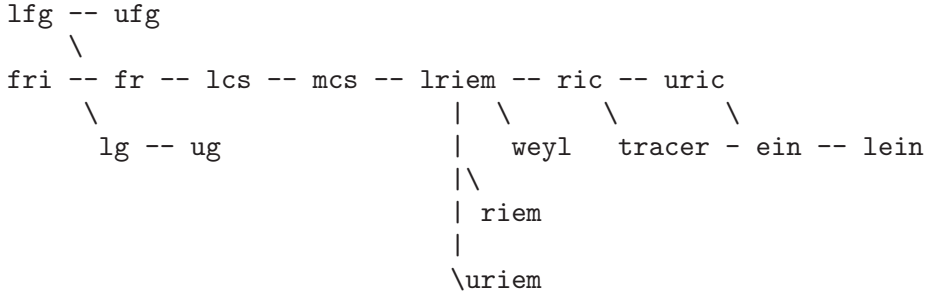
O principal propósito do pacote `ctensor` é calcular os tensores do espaç(tempo) curvo, mais notavelmente os tensores usados na relatividade geral.

Quando uma base métrica é usada, `ctensor` pode calcular os seguintes tensores:

```
lg  -- ug
  \      \
   lcs -- mcs -- ric -- uric
```



`ctensor` pode também usar molduras móveis. Quando `cframe_flag` for escolhida para `true`, os seguintes tensores podem ser calculados:



christof (*dis*) Função

Uma função no pacote `ctensor`. Essa função calcula os símbolos de Christoffel de ambos os tipos. O argumento *dis* determina quais resultados são para serem imediatamente mostrados. Os símbolos de Christoffel de primeiro e de segundo tipo são armazenados nos arrays `lcs[i,j,k]` e `mcs[i,j,k]` respectivamente e definidos para serem simétricos nos primeiros dois índices. Se o argumento para `christof` for `lcs` ou for `mcs` então o único valor não nulo de `lcs[i,j,k]` ou de `mcs[i,j,k]`, respectivamente, será mostrado. Se o argumento for `all` então o único valor não nulo de `lcs[i,j,k]` e o único valor não nulo de `mcs[i,j,k]` serão mostrados. Se o argumento for `false` então a exibição dos elementos não acontecerá. Os elementos do array `mcs[i,j,k]` são definidos de uma tal maneira que o índice final é contravariante.

ricci (*dis*) Função

Uma função no pacote `ctensor`. `ricci` calcula as componentes contravariantes (simétricas) `ric[i,j]` do tensor de Ricci. Se o argumento *dis* for `true`, então as componentes não nulas são mostradas.

uricci (*dis*) Função

Essa função primeiro calcula as componentes contravariantes `ric[i,j]` do tensor de Ricci. Então o tensor misto de Ricci é calculado usando o tensor métrico contravariante. Se o valor do argumento *dis* for `true`, então essas componentes mistas, `uric[i,j]` (o índice `"i"` é covariante e o índice `"j"` é contravariante), serão mostradas diretamente. De outra forma, `ricci(false)` irá simplesmente calcular as entradas do array `uric[i,j]` sem mostrar os resultados.

scurvature () Função

Retorna a curvatura escalar (obtida através da contração do tensor de Ricci) do Riemaniano multiplicado com a métrica dada.

einstein (*dis*) Função

Uma função no pacote `ctensor`. `einstein` calcula o tensor misto de Einstein após os símbolos de Christoffel e o tensor de Ricci terem sido obtidos (com as funções `christof` e `ricci`). Se o argumento *dis* for `true`, então os valores não nulos do tensor misto de Einstein `ein[i,j]` serão mostrados quando `j` for o índice contravariante. A variável `rateinstein` fará com que a simplificação racional ocorra sobre esses componentes. Se `ratfac` for `true` então as componentes irão também ser fatoradas.

leinstein (*dis*) Função

Tensor covariante de Einstein. `leinstein` armazena o valor do tensor covariante de Einstein no array `lein`. O tensor covariante de Einstein é calculado a partir tensor misto de Einstein `ein` através da multiplicação desse pelo tensor métrico. Se o argumento *dis* for `true`, então os valores não nulos do tensor covariante de Einstein são mostrados.

riemann (*dis*) Função

Uma função no pacote `ctensor`. `riemann` calcula o tensor de curvatura de Riemann a partir da métrica dada e correspondendo aos símbolos de Christoffel. As seguintes convenções de índice são usadas:

```
                  l       _l        _l        _l    _m     _l    _m
R[i,j,k,l] =  R     = |       - |       + |     |    - |     |
                  ijk      ij,k      ik,j      mk    ij     mj    ik
```

Essa notação é consistente com a notação usada por no pacote `itensor` e sua função `icurvature`. Se o argumento opcional *dis* for `true`, as componentes não nulas `riem[i,j,k,l]` serão mostradas. Como com o tensor de Einstein, vários comutadores escolhidos pelo usuário controlam a simplificação de componentes do tensor de Riemann. Se `ratriemann` for `true`, então simplificação racional será feita. Se `ratfac` for `true` então cada uma das componentes irá também ser fatorada.

Se a variável `cframe_flag` for `false`, o tensor de Riemann é calculado diretamente dos símbolos de Christoffel. Se `cframe_flag` for `false`, o tensor covariante de Riemann é calculado primeiro dos coeficientes de campo da moldura.

lriemann (*dis*) Função

Tensor covariante de Riemann (`lriem[]`).

Calcula o tensor covariante de Riemann como o array `lriem`. Se o argumento *dis* for `true`, únicos valores não nulos são mostrados.

Se a variável `cframe_flag` for `true`, o tensor covariante de Riemann é calculado diretamente dos coeficientes de campo da moldura. De outra forma, o tensor (3,1) de Riemann é calculado primeiro.

Para informação sobre a ordenação de índice, veja `riemann`.

uriemann (*dis*) Função

Calcula as componentes contravariantes do tensor de curvatura de Riemann como elementos do array `uriem[i,j,k,l]`. Esses são mostrados se *dis* for `true`.

rinvariant () Função

Compõe o invariante de Kretchmann (`kinvariant`) obtido através da contração dos tensores

```
lriem[i,j,k,l]*uriem[i,j,k,l].
```

Esse objeto não é automaticamente simplificado devido ao fato de poder ser muito largo.

weyl (*dis*) Função

Calcula o tensor conformal de Weyl. Se o argumento *dis* for `true`, as componentes não nulas `weyl[i,j,k,l]` irão ser mostradas para o usuário. De outra forma, essas componentes irão simplesmente serem calculadas e armazenadas. Se o comutador `ratweyl` é escolhido para `true`, então as componentes irão ser racionalmente simplificadas; se `ratfac` for `true` então os resultados irão ser fatorados também.

30.2.3 Expansão das séries de Taylor

O pacote `ctensor` possui a habilidade para truncar resultados assumindo que eles são aproximações das séries de Taylor. Esse comportamenteo é controlado através da variável `ctayswitch`; quando escolhida para `true`, `ctensor` faz uso internamente da função `ctaylor` quando simplifica resultados.

A função `ctaylor` é invocada pelas seguintes funções de `ctensor`:

```
Function     Comments
--------------------------------
christof()   só para mcs
ricci()
uricci()
einstein()
riemann()
weyl()
checkdiv()
```

ctaylor () Função

A função `ctaylor` trunca seus argumentos através da conversão destes para uma série de Taylor usando `taylor`, e então chamando `ratdisrep`. Isso tem efeito combinado de abandonar termos de ordem mais alta na variável de expansão `ctayvar`. A ordem dos termos que podem ser abandonados é definida através de `ctaypov`; o ponto em torno do qual a expansão da série é realizada está especificado em `ctaypt`.

Como um exemplo, considere uma métrica simples que é uma perturbação da métrica de Minkowski. Sem restrições adicionais, mesmo uma métrica diagonal produz expressões para o tensor de Einstein que são de longe muito complexas:

```
(%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
```

Todavia, se nós recalcularmos esse exemplo como uma aproximação que é linear na variável `l`, pegamos expressões muito simples:

```
(%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
```

Essa compatibilidade pode ser útil, por exemplo, quando trabalhamos no limite do campo fraco longe de uma fonte gravitacional.

30.2.4 Campos de moldura

Quando a variável `cframe_flag` for escolhida para `true`, o pacote `ctensor` executa seus cálculos usando uma moldura móvel.

frame_bracket (*fr*, *fri*, *diagframe*) Função

O delimitador da moldura (`fb[]`).

Calcula o delimitador da moldura conforme a seguinte definição:

```
   c          c          c         d     e
ifb   = ( ifri    - ifri    ) ifr   ifr
   ab         d,e        e,d       a     b
```

30.2.5 Classificação Algébrica

Um novo recurso (a partir de November de 2004) de `ctensor` é sua habilidade para calcular a classificação de Petrov de uma métrica espaço tempo tetradimensional. Para uma demonstração dessa compatibilidade, veja o arquivo `share/tensor/petrov.dem`.

nptetrad () Função

Calcula um tetrad nulo de Newman-Penrose (`np`) e seus índices ascendentes em contrapartida (`npi`). Veja `petrov` para um exemplo.

O tetrad nulo é construído assumindo que uma moldura métrica ortonormal tetradimensional com assinatura métrica (-,+,+,+) está sendo usada. As componentes do tetrad nulo são relacionadas para a matriz moldura inversa como segue:

```
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*) Função

Calcula os cinco coeficientes de Newman-Penrose `psi[0]`...`psi[4]`. Se `psi` for escolhida para `true`, os coeficientes são mostrados. Veja `petrov` para um exemplo.

Esses coeficientes são calculados a partir do tensor de Weyl em uma base de coordenada. Se uma base de moldura for usada,o tensor de Weyl é primeiro convertido para a base de coordenada, que pode ser um procedimento computacional expansível. Por essa razão, em alguns casos pode ser mais vantajoso usar uma base de coordenada em primeiro lugar antes que o tensor de Weyl seja calculado. Note todavia, que para a construção de um tetrad nulo de Newman-Penrose é necessário uma base de moldura. Portanto, uma seqüência de cálculo expressiva pode começar com uma base de moldura, que é então usada para calcular `lg` (calculada automaticamente através de `cmetric`) e em seguida calcula `ug`. Nesse ponto, você pode comutar de volta para uma base de coordenada escolhendo `cframe_flag` para `false` antes de começar a calcular os símbolos de Christoffel. Mudando para uma base de moldura em um estágio posterior pode retornar resultados inconsistentes, já que você pode terminar com um grande mistura de tensores, alguns calculados em uma base de moldura, alguns em uma base de coordenada, sem nenhum modo para distingüir entre os dois tipos.

petrov () Função

Calcula a classificação de petrov da métrica caracterizada através de `psi[0]`...`psi[4]`.

Por exemplo, o seguinte demonstra como obter a classificação de Petrov da métrica de Kerr:

```
(%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
```

A função de classificação Petrov é baseada no algorítmo publicado em "Classifying geometries in general relativity: III Classification in practice" por Pollney, Skea, e d'Inverno, Class. Quant. Grav. 17 2885-2902 (2000). Exceto para alguns casos de

teste simples, a implementação não está testada até 19 de Dezembro de 2004, e é provável que contenha erros.

30.2.6 Torsão e não metricidade

`ctensor` possui a habilidade de calcular e incluir coeficientes de torsão e não metricidade nos coeficientes de conecção.

Os coeficientes de torsão são calculados a partir de um tensor fornecido pelo usuário `tr`, que pode ser um tensor de categoria (2,1). A partir disso, os coeficientes de torsão `kt` são calculados de acordo com a seguinte fórmula:

```
                m           m       m
        - g  tr    - g   tr    - tr   g
           im  kj     jm   ki      ij  km
kt    = -------------------------------
  ijk                 2


  k     km
kt   = g   kt
  ij         ijm
```

Note que somente o tensor de índice misto é calculao e armazenado no array `kt`.

Os coeficientes de não metricidade são calculados a partir do vetor de não metricidade fornecido pelo usuário `nm`. A partir disso, os coeficientes de não metricidade `nmc` são calculados como segue:

```
              k    k         km
        -nm  D  - D  nm  + g   nm  g
   k       i  j    i   j         m  ij
nmc  = -----------------------------
   ij               2
```

onde D simboliza o delta de Kronecker.

Quando `ctorsion_flag` for escolhida para `true`, os valores de `kt` são subtraídos dos coeficientes de conecção indexados mistos calculados através de `christof` e armazenados em `mcs`. Similarmente, se `cnonmet_flag` for escolhida para `true`, os valores de `nmc` são subtraídos dos coeficientes de conecção indexados mistos.

Se necessário, `christof` chama as funções `contortion` e `nonmetricity` com o objetivo de calcular `kt` e `nm`.

contortion (*tr*) Função

Calcula os coeficientes de contorsão de categoria (2,1) a partir do tensor de torsão *tr*.

nonmetricity (*nm*) Função

Calcula o coeficiente de não metricidade de categoria (2,1) a partir do vetor de não metricidade *nm*.

30.2.7 Recursos diversos

ctransform (M) Função

Uma função no pacote `ctensor` que irá executar uma transformação de coordenadas sobre uma matriz simétrica quadrada arbitrária M. O usuário deve informar as funçãoes que definem a transformação. (Formalmente chamada `transform`.)

findde (A, n) Função

Retorna uma lista de equações diferenciais únicas (expressões) correspondendo aos elementos do array quadrado n dimensional A. Atualmente, n pode ser 2 ou 3. `deindex` é uma lista global contendo os índices de A correspondendo a essas únicas equações diferenciais. Para o tensor de Einstein (`ein`), que é um array dimensional, se calculado para a métrica no exemplo abaixo, `findde` fornece as seguintes equações diferenciais independentes:

```
(%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 () Função

Calcula o gradiente covariante de uma função escalar permitindo ao usuário escolher o nome do vetor correspondente como o exemplo sob `contragrad` ilustra.

contragrad () Função

Calcula o gradiente contravariante de uma função escalar permitindo ao usuário escolher o nome do vetor correspondente como o exemplo abaixo como ilustra a métrica de Schwarzschild:

```
(%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 () Função

Calcula o tensor d'Alembertiano da função escalar assim que as dependências tiverem sido declaradas sobre a função. Po exemplo:

```
(%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 () Função

Calcula a divergência covariante do tensor de segunda categoria misto (cujo primeiro índice deve ser covariante) imprimindo as correspondentes n componentes do campo do vetor (a divergência) onde n = `dim`. Se o argumento para a função for `g` então a divergência do tensor de Einstein irá ser formada e pode ser zero. Adicionalmente, a divergência (vetor) é dada no array chamado `div`.

cgeodesic (*dis*) Função

Uma função no pacote `ctensor`. `cgeodesic` calcula as equações geodésicas de movimento para uma dada métrica. Elas são armazenadas no array `geod[i]`. Se o argumento *dis* for `true` então essas equações são mostradas.

bdvac (*f*) Função

Gera as componentes covariantes das equações de campo de vácuo da teoria de gravitação de Brans-Dicke. O campo escalar é especificado através do argumento *f*, que pode ser um nome de função (com apóstrofo) com dependências funcionais, e.g., `'p(x)`.

As componentes de segunda categoria do tensor campo covariante são as componentes de segunda categoria representadas pelo array `bd`.

invariant1 () Função

Gera o tensor misto de Euler-Lagrange (equações de campo) para a densidade invariante de R^2. As equações de campo são componentes de um array chamado `inv1`.

invariant2 () Função

*** NOT YET IMPLEMENTED ***

Gera o tensor misto de Euler-Lagrange (equações de campo) para a densidade invariante de `ric[i,j]*uriem[i,j]`. As equações de campo são as componentes de um array chamado `inv2`.

bimetric () Função

*** NOT YET IMPLEMENTED ***

Gera as euauações de campo da teoria bimétrica de Rosen. As equações de campo são as componentes de um array chamado `rosen`.

30.2.8 Funções utilitárias

diagmatrixp (*M*) Função

Retorna `true` se *M* for uma matriz diagonal ou um array (2D).

symmetricp (*M*) Função

Retorna `true` se *M* for uma matriz simétrica ou um array (2D).

ntermst (*f*) Função

Fornece ao usuário um rápido quadro do "tamanho" do tensor duplamente subscrito (array) *f*. Imprime uma lista de dois elementos onde o segundo elemento corresponde a N-TERMOS de componentes especificadas através dos primeiros elementos. Nesse caminho, é possível rapidamente encontrar as expressões não nulas e tentar simplificação.

cdisplay (*ten*) Função

Mostra todos os elementos do tensor *ten*, como representados por um array multidimensional. Tensores de categoria 0 e 1, assim como outros tipos de variáveis, são mostrados com `ldisplay`. Tensores de categoria 2 são mostrados como matrizes bidimensionais, enquanto tensores de alta categoria são mostrados como uma lista de matrizes bidimensionais. Por exemplo, o tensor de Riemann da métrica de Schwarzschild pode ser visto como:

```
(%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*) Função

Retorna uma nova lista consistindo de *L* com o *n*'ésimo elemento apagado.

30.2.9 Variáveis usadas por `ctensor`

dim Variável de opção

Valor padrão: 4

Uma opção no pacote `ctensor`. `dim` é a dimensão de multiplicação com o padrão 4. O comando `dim: n` irá escolher a dimensão para qualquer outro valor `n`.

diagmetric Variável de opção

Valor padrão: `false`

Uma opção no pacote `ctensor`. Se `diagmetric` for `true` rotinas especiais calculam todos os objetos geométricos (que possuem o tensor métrico explicitamente) levando em consideração a diagonalidade da métrica. Tempo de execução reduzido irá, com certeza, resultar dessa escolha. Nota: essa opção é escolhida automaticamente por `csetup` se uma métrica diagonal for especificada.

ctrgsimp Variável de opção

Faz com que simplificações trigonométricas sejam usadas quando tensores forem calculados. Atualmente, `ctrgsimp` afeta somente cálculos envolvendo uma moldura móvel.

cframe_flag Variável de opção

Faz com que cálculos sejam executados relativamente a uma moldura móvel em oposição a uma métrica holonômica. A moldura é definida através do array da moldura inversa `fri` e da métrica da moldura `lfg`. Para cálculos usando uma moldura Cartesiana, `lfg` pode ser a matriz unitária de dimensão apropriada; para cálculos em uma moldura de Lorentz, `lfg` pode ter a assinatura apropriada.

ctorsion_flag Variável de opção

Faz com que o tensor de contorsão seja incluído no cálculo dos coeficientes de conecção. O tensor de contorsão por si mesmo é calculado através de `contortion` a partir do tensor `tr` fornecido pelo usuário.

cnonmet_flag Variável de opção

Faz com que os coeficientes de não metricidade sejam incluídos no cálculo dos coeficientes de conecção. Os coeficientes de não metricidade são calculados a partir do vetor de não metricidade `nm` fornecido pelo usuário através da função `nonmetricity`.

ctayswitch Variável de opção

Se escolhida para `true`, faz com que alguns cálculos de `ctensor` sejam realizados usando expansões das séries de Taylor. atualmente, `christof`, `ricci`, `uricci`, `einstein`, e `weyl` levam em conta essa escolha.

ctayvar Variável de opção

Variável usada pela expansão de séries de Taylor se `ctayswitch` é escolhida para `true`.

ctaypov Variável de opção

Maximo expoente usado em expansões de séries de Taylor quando `ctayswitch` for escolhida para `true`.

ctaypt Variável de opção

Ponto em torno do qual expansões de séries de Taylor sao realizadas quando `ctayswitch` for escolhida para `true`.

gdet Variável de sistema

O determinante do tensor métrico `lg`. Calculado através de `cmetric` quando `cframe_flag` for escolhido para `false`.

ratchristof Variável de opção

Faz com que simplificações racionais sejam aplicadas através de `christof`.

rateinstein Variável de opção

Valor padrão: `true`

Se `true` simplificação racional irá ser executada sobre as componentes não nulas de tensores de Einstein; se `ratfac` for `true` então as componentes irão também ser fatoradas.

ratriemann Variável de opção

Valor padrão: `true`

Um dos comutadores que controlam simplificações dos tensores de Riemann; se `true`, então simplificações racionais irão ser concluídas; se `ratfac` for `true` então cada uma das componentes irá também ser fatorada.

ratweyl Variável de opção

Valor padrão: `true`

Se `true`, esse comutador faz com que a função de `weyl` aplique simplificações racionais aos valores do tensor de Weyl. Se `ratfac` for `true`, então as componentes irão também ser fatoradas.

lfg Variável

A moldura métrica covariante. Por padrão, é inicializada para a moldura tetradimensional de Lorentz com assinatura (+,+,+,-). Usada quando `cframe_flag` for `true`.

ufg Variável

A métrica da moldura inversa. Calculada de `lfg` quando `cmetric` for chamada enquanto `cframe_flag` for escolhida para `true`.

riem Variável

O tensor de categoria (3,1) de Riemann. Calculado quando a função `riemann` é invocada. Para informação sobre ordenação de índices, veja a descrição de `riemann`.

Se `cframe_flag` for `true`, `riem` é calculado a partir do tensor covariante de Riemann `lriem`.

lriem Variável

O tensor covariante de Riemann. Calculado através de `lriemann`.

uriem Variável

O tensor contravariante de Riemann. Calculado através de `uriemann`.

ric Variável

O tensor misto de Ricci. Calculado através de `ricci`.

uric Variável

O tensor contravariante de Ricci. Calculado através de `uricci`.

lg Variável

O tensor métrico. Esse tensor deve ser especificado (como uma `dim` através da matriz `dim`) antes que outro cálculo possa ser executado.

ug Variável

O inverso do tensor métrico. Calculado através de `cmetric`.

weyl Variável

O tensor de Weyl. Calculado através de `weyl`.

fb Variável

Coeficientes delimitadores da moldura, como calculado através de `frame_bracket`.

kinvariant Variável

O invariante de Kretchmann. Calculado através de `rinvariant`.

np Variável

Um tetrad nulo de Newman-Penrose. Calculado através de `nptetrad`.

npi Variável

O índice ascendente do tetrad nulo de Newman-Penrose. Calculado através de `nptetrad`. Definido como `ug.np`. O produto `np.transpose(npi)` é constante:

```
(%i39) trigsimp(np.transpose(npi));
                                [  0   - 1  0  0 ]
                                [                ]
                                [ - 1   0   0  0 ]
(%o39)                          [                ]
                                [  0    0   0  1 ]
                                [                ]
                                [  0    0   1  0 ]
```

tr Variável

Tensor de categoria 3 fornecido pelo usuário representando torsão. Usado por `contortion`.

kt Variável

O tensor de contorsão, calculado a partir de `tr` através de `contortion`.

nm Variável

Vetor de não metrcidade fornecido pelo usuário. Usado por `nonmetricity`.

nmc Variável

Os coeficientes de não metricidade, calculados a partir de `nm` por `nonmetricity`.

tensorkill Variável de sistema

Variável indicando se o pacote tensor foi inicializado. Escolhida e usada por `csetup`, retornada ao seu valor original através de `init_ctensor`.

ct_coords Variável de opção

Valor padrão: `[]`

Uma opção no pacote `ctensor`. `ct_coords` contém uma lista de coordenadas. Enquanto normalmente definida quando a função `csetup` for chamada, se pode redefinir as coordenadas com a atribuição `ct_coords: [j1, j2, ..., jn]` onde os j's são os novos nomes de coordenadas. Veja também `csetup`.

30.2.10 Nomes reservados

Os seguintes nomes são usados internamente pelo pacote `ctensor` e não devem ser redefinidos:

```
Name         Description
--------------------------------------
_lg()        Avalia para lfg se a moldura métrica for usada,
  para lg de outra forma
_ug()        Avalia para ufg se a moldura métrica for usada,
  para ug de outra forma
cleanup()    Remove ítens da lista deindex
contract4()  Usado por psi()
filemet()    Usado por csetup() quando lendo a métrica de um arquivo
findde1()    Usado por findde()
findde2()    Usado por findde()
findde3()    Usado por findde()
kdelt()      Delta de Kronecker (não generalizado)
newmet()     Usado por csetup() para escolher uma métrica
  interativamente
setflags()   Usado por init_ctensor()
readvalue()
resimp()
sermet()     Usado por csetup() para informar uma métricacom série
  de Taylor
txyzsum()
tmetric()    Moldura métrica, usado por cmetric() quando
  cframe_flag:true
triemann()   Tensor de Riemann em base de moldura, usado quando
  cframe_flag:true
tricci()     Tensor de Ricci em base de moldura, usada quando
  cframe_flag:true
trrc()       Coeficientes de rotação de Ricci, usado por
  christof()
yesp()
```

30.2.11 Changes

Em Novembro de 2004, o pacote `ctensor` foi extensivamente reescrito. Muitas funções e variáveis foram renomeadas com o objetivo de tornar o pacote com a versão comercial do Macsyma.

```
Novo Nome    Nome Antigo     Descrição
-------------------------------------------------------------------
ctaylor()    DLGTAYLOR()     Expansão da série de Taylor de uma
-----------------------------expressão
lgeod[]      EM              Equações geodésicas
ein[]        G[]             Tensor misto de Einstein
ric[]        LR[]            Tensor misto de Ricci
ricci()      LRICCICOM()     Calcula o tensor misto de Ricci
ctaypov      MINP            Maximo expoente em expansões de séries de
```

```
-----------------------------Taylor
cgeodesic()  MOTION          Calcula as equações geodésicas
ct_coords    OMEGA           Coordenadas métricas
ctayvar      PARAM           Variável de expansão de séries de
-----------------------------Taylor
lriem[]      R[]             Tensor covariante de Riemann
uriemann()   RAISERIEMANN()  Calcula o tensor contravariante de
-----------------------------Riemann
ratriemann   RATRIEMAN       Simplificação racional do tensor de
-----------------------------Riemann
uric[]       RICCI[]         Tensor de Ricci contravariante
uricci()     RICCICOM()      Calcula o tensor de Ricci contravariante
cmetric()    SETMETRIC()     Escolhe a métrica
ctaypt       TAYPT           Ponto para expansões de séries de Taylor
ctayswitch   TAYSWITCH       Escolhe o comutador de séries de Taylor
csetup()     TSETUP()        Inicia sessão interativa de configuração
ctransform() TTRANSFORM()    Transformação de coordenadas interativa
uriem[]      UR[]            Tensor contravariante de Riemann
weyl[]       W[]             Tensor (3,1) de Weyl
```

31 Pacote atensor

31.1 Introdução ao Pacote atensor

`atensor` é um pacote de manipulção de tensores algébricos. Para usar `atensor`, digite `load(atensor)`, seguido por uma chamada à função `init_atensor`.

A essência de `atensor` é um conjunto de regras de simplificação para o operador de produto (ponto) não comutativo (`"."`). `atensor` reconhece muitos tipos de álgebra; as regras de simplificação correspondentes são ativadas quando a função `init_atensor` é chamada.

A compatibilidade de `atensor` pode ser demonstrada pela definição da álgebra de quatérnios como uma álgera-Clifford Cl(0,2) com dois vetores fundamentais. As três unidades quaterniônicas imaginárias fundamentais são então os dois vetores base e seu produto, i.e.:

```
i = v     j = v     k = v  .  v
     1         2         1    2
```

Embora o pacote `atensor` tenha uma definição interna para a álgebra dos quatérnios, isso não foi usado nesse exemplo, no qual nós nos esforçamos para construir a tabela de multiplicação dos quatérnios como uma matriz:

```
(%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` reconhece como bases vetoriais símbolos indexados, onde o símbolo é aquele armazenado em `asymbol` e o índice está entre 1 e `adim`. Para símbolos indexado, e somente para símbolos indexados, as formas bilineares `sf`, `af`, e `av` são avaliadas. A avaliação substitui os valores de `aform[i,j]` em lugar de `fun(v[i],v[j])` onde `v` representa o valor de `asymbol` e `fun` é ainda `af` ou `sf`; ou, isso substitui `v[aform[i,j]]` em lugar de `av(v[i],v[j])`.

Desnecessário dizer, as funções `sf`, `af` e `av` podem ser redefinidas.

Quando o pacote `atensor` é chamado, os seguintes sinalizadores são configurados:

```
dotscrules:true;
dotdistrib:true;
dotexptsimp:false;
```

Se você deseja experimentar com uma álgebra não associativa, você pode também considerar a configuração de `dotassoc` para `false`. Nesse caso, todavia, `atensimp` não stará sempre habilitado a obter as simplificações desejadas.

31.2 Definições para o Pacote atensor

init_atensor (*alg_type*, *opt_dims*) Função
init_atensor (*alg_type*) Função

Inicializa o pacote `atensor` com o tipo especificado de álgebra. *alg_type* pode ser um dos seguintes:

`universal`: A álgebra universal tendo regras não comutativas.

`grassmann`: A álgebra de Grassman é definida pela relação de comutação `u.v+v.u=0`.

`clifford`: A álgebra de Clifford é definida pela relação de comutação `u.v+v.u=-2*sf(u,v)` onde `sf` é a função valor-escalar simétrico. Para essa álgebra, *opt_dims* pode ser acima de três inteiros não negativos, representando o número de dimensões positivas, dimensões degeneradas, e dimensões negativas da álgebra, respectivamente. Se quaisquer valores *opt_dims* são fornecidos, `atensor` irá configurar os valores de `adim` e `aform` apropriadamente. Caso contrário, `adim` irá por padrão para 0 e `aform` não será definida.

`symmetric`: A álgebra simétrica é definida pela relação de comutação `u.v-v.u=0`.

`symplectic`: A álgebra simplética é definida pela relação de comutação `u.v-v.u=2*af(u,v)` onde `af` é uma função valor-escalar antisimétrica. Para a

álgebra simplética, *opt_dims* pode mais de dois inteiros não negativos, representando a dimensão não degenerada e e a dimensão degenerada, respectivamente. Se quaisquer valores *opt_dims* são fornecidos, `atensor` irá configurar os valores de `adim` e `aform` apropriadamente. Caso contrário, `adim` irá por padrão para 0 e `aform` não será definida.

`lie_envelop`: O invólucro da álgebra de Lie é definido pela relação de comutação `u.v-v.u=2*av(u,v)` onde `av` é uma função antisimétrica.

A função `init_atensor` também reconhece muitos tipos pré-definidos de álgebra:

`complex` implementa a álgebra de números complexos como a álgebra de Clifford Cl(0,1). A chamada `init_atensor(complex)` é equivalente a `init_atensor(clifford,0,0,1)`.

`quaternion` implementa a álgebra de quatérnios. A chamada `init_atensor(quaternion)` é equivalente a `init_atensor(clifford,0,0,2)`.

`pauli` implementa a álgebra de Pauli-spinors como a Clifford-álgebra Cl(3,0). Uma chamada a `init_atensor(pauli)` é equivalente a `init_atensor(clifford,3)`.

`dirac` implementa a álgebra de Dirac-spinors como a Clifford-álgebra Cl(3,1). Uma chamada a `init_atensor(dirac)` é equivalente a `init_atensor(clifford,3,0,1)`.

atensimp (*expr*) Função

Simplifica a expressão algébrica de tensores *expr* conforme as regras configuradas por uma chamada a `init_atensor`. Simplificações incluem aplicação recursiva de relações comutativas e resoluções de chamadas a `sf`, `af`, e `av` onde for aplicável. Uma salvaguarda é usada para garantir que a função sempre termine, mesmo para expressões complexas.

alg_type Função

O tipo de álgebra. Valores válidos sáo `universal`, `grassmann`, `clifford`, `symmetric`, `symplectic` and `lie_envelop`.

adim Variável

A dimensionalidade da álgebra. `atensor` usa o valor de `adim` para determinar se um objeto indexado é uma base vetorial válida. Veja `abasep`.

aform Variável

Valor padrão para as formas bilineares `sf`, `af`, e `av`. O padrão é a matriz identidade `ident(3)`.

asymbol Variável

O símbolo para bases vetoriais.

sf (*u*, *v*) Função

É uma função escalar simétrica que é usada em relações comutativas. A implementação padrão verifica se ambos os argumentos são bases vetoriais usando `abasep` e se esse for o caso, substitui o valor correspondente da matriz `aform`.

af (*u*, *v*) Função

É uma função escalar antisimétrica que é usada em relações comutativas. A implementação padrão verifica se ambos os argumentos são bases vetoriais usando `abasep` e se esse for o caso, substitui o valor correspondente da matriz `aform`.

av (*u*, *v*) Função

É uma função antisimétrica que é usada em relações comutativas. A implementação padrão verifica se ambos os argumentos são bases vetoriais usando `abasep` e se esse for o caso, substitui o valor correspondente da matriz `aform`.

Por exemplo:

```
(%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*) Função

Verifica se esse argumento é uma base vetorial `atensor` .

E será, se ele for um símbolo indexado, com o símbolo sendo o mesmo que o valor de `asymbol`, e o índice tiver o mesmo valor numérico entre 1 e `adim`.

32 Séries

32.1 Introdução a Séries

Maxima contém funções `taylor` e `powerseries` (séries de potência) para encontrar as séries de funções diferenciáveis. Maxima também tem ferramentas tais como `nusum` capazes de encontrar a forma fechada de algumas séries. Operações tais como adição e multiplicação travalham da forma usual sobre séries. Essa seção apresenta as variáveis globais que controlam a expansão.

32.2 Definições para Séries

cauchysum Variável de opção

Valor padrão: `false`

Quando multiplicando adições jutas com `inf` como seus limites superiores, se `sumexpand` for `true` e `cauchysum` for `true` então o produto de Cauchy será usado em lugar do produto usual. No produto de Cauchy o índice do somatório interno é uma função do índice do externo em lugar de variar independentemente.

Exemplo:

```
(%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*) Função

Para cada função *f_i* de uma variável *x_i*, `deftaylor` define *expr_i* como a séries de Taylor sobre zero. *expr_i* é tipicamente um polinômio em *x_i* ou um somatório; expressões mais gerais são aceitas por `deftaylor` sem reclamações.

`powerseries` (*f_i*(*x_i*), *x_i*, 0) retorna as séries definidas por `deftaylor`.

`deftaylor` retorna uma lista das funções *f_1*, ..., *f_n*. `deftaylor` avalia seus argumentos.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `true`

Quando `maxtayorder` for `true`, durante a manipulação algébrica de séries (truncadas) de Taylor, `taylor` tenta reter tantos termos quantos forem conhecidos serem corretos.

niceindices (*expr*) Função

Renomeia os índices de adições e produtos em *expr*. `niceindices` tenta renomear cada índice para o valor de `niceindicespref[1]`, a menos que o nome apareça nas parcelas do somatório ou produtório, nesses casos `niceindices` tenta os elementos seguintes de `niceindicespref` por sua vez, até que uma varável não usada unused variable seja encontrada. Se a lista inteira for exaurida, índices adicionais são constrídos através da anexaao de inteiros ao valor de `niceindicespref[1]`, e.g., `i0`, `i1`, `i2`, ...

`niceindices` retorna uma expressão. `niceindices` avalia seu argumento.

Exemplo:

```
(%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 Variável de opção

Valor padrão: `[i, j, k, l, m, n]`

`niceindicespref` é a lista da qual `niceindices` pega os nomes dos índices de adições e produtos products.

Os elementos de `niceindicespref` são tipicamente nomes de variáveis, embora que não seja imposto por `niceindices`.

Exemplo:

```
(%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*) Função

Realiza o somatório hipergeométrico indefinido de *expr* com relação a *x* usando um procedimento de decisão devido a R.W. Gosper. *expr* e o resultado deve ser expressável como produtos de expoentes inteiros, fatoriais, binomios, e funções recionais.

Os termos "definido" and "e somatório indefinido" são usados analogamente a "definida" and "integração indefinida". Adicionar indefinidamente significa dar um resultado simólico para a adição sobre intervalos de comprimentos de variáveis, não apenas e.g. 0 a infinito. Dessa forma, uma vez que não existe fórmula para a adição parcial geral de séries binomiais, `nusum` não pode fazer isso.

`nusum` e `unsum` conhecem um porco sobre adições e subtrações de produtos finitos. Veja também `unsum`.

Exemplos:

```
(%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*) Função

Retorna uma lista de todas as funções racionais que possuem a dada expansão da séries de Taylor onde a adição dos graus do numerador e do denominador é menor que ou igual ao nível de truncação das séries de potência, i.e. são "melhores" aproximações, e que adicionalmente satisfazem o grau especificado associado.

taylor_series é uma séries de Taylor de uma variável. *numer_deg_bound* e *denom_deg_bound* são inteiros positivos especificando o grau associado sobre o numerador e o denominador.

taylor_series podem também ser séries de Laurent, e o grau associado pode ser `inf` que acarreta todas funções racionais cujo grau total for menor que ou igual ao comprimento das séries de potências a serem retornadas. O grau total é definido como *numer_deg_bound* + *denom_deg_bound*. O comprimento de séries de potência é definido como `"nível de trncação" + 1 - min(0, "ordem das séries")`.

```
(%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)                           []
```

Não existe função racional de grau 4 numerador/denominador, com essa expansão de série de potência. Você obrigatoriamente em geral tem grau do numerador e grau do denominador adicionando para cima ao menor grau das séries de potência, com o objetivo de ter disponível coeficientes desconhecidos para resolver.

```
(%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 Variável de opção

Valor padrão: `false`

Quando `powerdisp` for `true`, uma adição é mostrada com seus termos em ordem do crescimento do expoente. Dessa forma um polinômio é mostrado como séries de potências truncadas, com o termo constante primeiro e o maior expoente por último.

Por padão, termos de uma adição são mostrados em ordem do expoente decrescente.

powerseries (*expr, x, a*) Função

Retorna a forma geral expansão de séries de potência para *expr* na variável *x* sobre o ponto *a* (o qual pode ser `inf` para infinito).

Se `powerseries` incapaz de expandir *expr*, `taylor` pode dar os primeiros muitos termos de séries.

Quando `verbose` for `true`, `powerseries` mostra mensagens de progresso.

```
(%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 Variável de opção

Valor padrão: `false`

Quando `psexpand` for `true`, uma expressão função racional extendida é mostrada completamente expandida. O comutador `ratexpand` tem o mesmo efeito.

Quando `psexpand` for `false`, uma expressão de várias variáveis é mostrada apenas como no pacote de função racional.

Quando `psexpand` for `multi`, então termos com o mesmo grau total nas variáveis são agrupados juntos.

revert (*expr, x*) Função

revert2 (*expr, x, n*) Função

Essas funções retornam a reversão de *expr*, uma série de Taylor sobre zero na variável *x*. `revert` retorna um polinômio de grau igual ao maior expoente em *expr*. `revert2` retorna um polinômio de grau *n*, o qual pode ser maior que, igual a, ou menor que o grau de *expr*.

`load ("revert")` chama essas funções.

Exemplos:

```
(%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*) Função
taylor (*expr*, [*x_1*, *x_2*, ...], *a*, *n*) Função
taylor (*expr*, [*x*, *a*, *n*, *'asymp*]) Função
taylor (*expr*, [*x_1*, *x_2*, ...], [*a_1*, *a_2*, ...], [*n_1*, *n_2*, ...]) Função

`taylor` (*expr*, *x*, *a*, *n*) expande a expressão *expr* em uma série truncada de Taylor ou de Laurent na variável *x* em torno do ponto *a*, contendo termos até `(`*x* `-` *a*`)^`*n*.

Se *expr* é da forma *f*`(`*x*`)/`*g*`(`*x*`)` e *g*`(`*x*`)` não possui de grau acima do grau *n* então `taylor` tenta expandir *g*`(`*x*`)` acima do gau `2` *n*. Se existe ainda termos não zero, `taylor` dobra o grau de expansão de *g*`(`*x*`)` contanto que o grau da expansão o grau da expansão seja menor que ou igual a *n* `2^taylordepth`.

`taylor` (*expr*, [*x_1*, *x_2*, ...], *a*, *n*) retorna uma série de potência truncada de grau *n* em todas as variáveis *x_1*, *x_2*, ... sobre o ponto `(`*a*, *a*, `...)`.

`taylor` (*expr*, [*x_1*, *a_1*, *n_1*], [*x_2*, *a_2*, *n_2*], ...) retorna uma série de potência truncada nas variáveis *x_1*, *x_2*, ... sobre o ponto `(`*a_1*, *a_2*, `...)`, truncada em *n_1*, *n_2*, ...

`taylor` (*expr*, [*x_1*, *x_2*, ...], [*a_1*, *a_2*, ...], [*n_1*, *n_2*, ...]) retorna uma série de potência truncada nas variáveis *x_1*, *x_2*, ... sobre o ponto `(`*a_1*, *a_2*, `...)`, truncada em *n_1*, *n_2*, ...

`taylor` (*expr*, [*x*, *a*, *n*, `'asymp]`) retorna uma expansão de *expr* em expoentes negativos de *x* `-` *a*. O termo de maior ordem é `(`*x* `-` *a*`)^-`*n*.

Quando `maxtayorder` for `true`, então durante maniplulação algébrica da séries de Taylor (truncada), `taylor` tenta reter tantos termos quantos forem conhecidos serem corretos.

Quando `psexpand` for `true`, uma expressão de função racional extendida é mostrada completamente expandida. O comutador `ratexpand` tem o mesmo efeito. Quando `psexpand` for `false`, uma expressão de várias variáveis é mostrada apenas como no pacote de função racional. Quando `psexpand` for `multi`, então os termos com o mesmo grau total nas variáveis são agrupados juntos.

Veja também o comutador `taylor_logexpand` para controlar a expansão.

Exemplos:

```
(%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 Variável de opção

Valor padrão: 3

Se existem ainda termos não zero, `taylor` dobra o grau da expansão de $g(x)$ contanto que o grau da expansão seja menor que ou igual a n `2^taylordepth`.

taylorinfo (*expr*) Função

Retorna information about the séries de Taylor *expr*. O valor de retorno é uma lista de listas. Cada lista compreende o nome de uma variável, o ponto de expansão, e o grau da expansão.

`taylorinfo` retorna `false` se *expr* não for uma séries de Taylor.

Exemplo:

```
(%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*) Função

Retorna `true` se *expr* for uma séries de Taylor, e `false` de outra forma.

taylor_logexpand Variável de opção

Valor padrão: `true`

`taylor_logexpand` controla expansão de logarítmos em séries de `taylor`.

Quando `taylor_logexpand` for `true`, todos logarítmos são expandidos completamente dessa forma problemas de reconhecimento de zero envolvendo envolvendo identidades logarítmicas não atrapalham o processo de expansão. Todavia, esse esquema não é sempre maematicamente correto uma vez que isso ignora informações de ramo.

Quando `taylor_logexpand` for escolhida para `false`, então a expansão logarítmica que ocorre é somente aquela que for necessária para obter uma séries de potência formal.

taylor_order_coefficients Variável de opção

Valor padrão: `true`

`taylor_order_coefficients` controla a ordenação dos coeficientes em uma série de Taylor.

Quando `taylor_order_coefficients` for `true`, coeficientes da séries de Taylor são ordenados canonicamente.

taylor_simplifier (*expr*) Função

Simplifica coeficientes da séries de potência *expr*. `taylor` chama essa função.

taylor_truncate_polynomials Variável de opção

Valor padrão: `true`

Quando `taylor_truncate_polynomials` for `true`, polinômios são truncados baseados sobre a entrada de níveis de truncação.

De outra forma, entrada de polinômios para `taylor` são consideradas terem precisão infinita.

taytorat (*expr*) Função

Converte *expr* da forma `taylor` para a forma de expressão racional canônica (CRE). O efeito é o mesmo que `rat (ratdisrep (`*expr*`))`, mas mais rápido.

trunc (*expr*) Função

Coloca notas na representação interna da expressão geral *expr* de modo que isso é mostrado como se suas adições forem séries de Taylor truncadas. *expr* is not otherwise modified.

Exemplo:

```
(%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*) Função

Retorna a primeira diferençã de trás para frente `f(n) - f(n - 1)`. Dessa forma `unsum` logicamente é a inversa de `sum`.

Veja também `nusum`.

Exemplos:

```
(%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 Variável de opção

Valor padrão: `false`

Quando `verbose` for `true`, `powerseries` mostra mensagens de progresso.

33 Teoria dos Números

33.1 Definições para Teoria dos Números

bern (*n*) Função

Retorna o *n*'ésimo número de Bernoulli para o inteiro *n*. Números de Bernoulli iguais a zero são suprimidos se `zerobern` for `false`.

Veja também `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*) Função

Retorna o *n*'ésimo polinômio de Bernoulli na variável *x*.

bfzeta (*s*, *n*) Função

Retorna a função zeta de Riemann para o argumento *s*. O valor de retorno é um grande inteiro em ponto flutuante (bfloat); *n* é o número de dígitos no valor de retorno.

`load ("bffac")` chama essa função.

bfhzeta (*s*, *h*, *n*) Função

Retorna a função zeta de Hurwitz para os argumentos *s* e *h*. O valor de retorno é um grande inteiro em ponto flutuante (bfloat); *n* é o números de dígitos no valor de retorno.

A função zeta de Hurwitz é definida como

```
sum ((k+h)^-s, k, 0, inf)
```

`load ("bffac")` chama essa função.

binomial (*x*, *y*) Função

O coeficiente binomial `(x + y)!/(x! y!)`. Se *x* e *y* forem inteiros, então o valor numérico do coeficiente binomial é calculado. Se *y*, ou *x* - *y*, for um inteiro, o the coeficiente binomial é expresso como um polinômio.

burn (*n*) Função

Retorna o *n*'ésimo número de Bernoulli para o inteiro *n*. `burn` pode ser mais eficitente que `bern` para valores grandes e isolados de *n* (talvez *n* maior que 105 ou algo

parecido), como `bern` calcula todos os números de Bernoulli até o índice *n* antes de retornar.

`burn` explora a observação que números de Bernoulli (racionais) podem ser aproximados através de zetas (transcendentes) com eficiência tolerável.

`load ("bffac")` chama essa função.

cf (*expr*) Função

Converte *expr* em uma fração contínua. *expr* é uma expressão compreendendo frações contínuas e raízes quadradas de inteiros. Operandos na expressão podem ser combinados com operadores aritméticos. Com excessão de frações contínuas e raízes quadradas, fatores na expressão devem ser números inteiros ou racionais. Maxima não conhece operações sobre frações contínuas fora de `cf`.

`cf` avalia seus argumentos após associar `listarith` a `false`. `cf` retorna uma fração contínua, representada como uma lista.

Uma fração contínua `a + 1/(b + 1/(c + ...))` é representada através da lista `[a, b, c, ...]`. Os elementos da lista `a`, `b`, `c`, ... devem avaliar para inteiros. *expr* pode também conter `sqrt (n)` onde `n` é um inteiro. Nesse caso `cf` fornecerá tantos termos de fração contínua quantos forem o valor da variável `cflength` vezes o período.

Uma fração contínua pode ser avaliada para um número através de avaliação da representação aritmética retornada por `cfdisrep`. Veja também `cfexpand` para outro caminho para avaliar uma fração contínua.

Veja também `cfdisrep`, `cfexpand`, e `cflength`.

Exemplos:

- *expr* é uma expressão compreendendo frações contínuas e raízes quadradas de inteiros.

```
(%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` controla quantos períodos de fração contínua são computados para números algébricos, números irracionais.

```
(%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]
```

- Um fração contínua pode ser avaliado através da avaliação da representação aritmética retornada por `cfdisrep`.

```
(%i1) cflength: 3$
(%i2) cfdisrep (cf (sqrt (3)))$
(%i3) ev (%, numer);
```

```
(%o3)                    1.731707317073171
```

- Maxima não conhece operações sobre frações contínuas fora de `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*) Função

Constrói e retorna uma expressão aritmética comum da forma `a + 1/(b + 1/(c + ...))` a partir da representação lista de uma fração contínua `[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*) Função

Retorna uma matriz de numeradores e denominadores dos último (columa 1) e penúltimo (columa 2) convergentes da fração contínua *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 Variável de opção

Valor padrão: 1

`cflength` controla o número de termos da fração contínua que a função `cf` fornecerá, como o valor de `cflength` vezes o período. Dessa forma o padrão é fornecer um período.

```
(%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*) Função
divsum (*n*) Função

`divsum (n, k)` retorna a adição dos divisores de *n* elevados à *k*'ésima potência.

`divsum (n)` retorna a adição dos divisores de *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*) Função

Retorna o *n*'ésimo número de Euler para o inteiro *n* não negativo.

Para a constante de Euler-Mascheroni, veja `%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 Constante

A constante de Euler-Mascheroni, 0.5772156649015329 ....

factorial (*x*) Função

Representa a função fatorial. Maxima trata `factorial (x)` da mesma forma que *x*`!`. Veja `!`.

fib (*n*) Função

Retorna o *n*'ésimo número de Fibonacci. `fib(0)` igual a 0 e `fib(1)` igual a 1, e `fib (-n)` igual a `(-1)^(n + 1) * fib(n)`.

Após chamar `fib`, `prevfib` é iguala `fib (x - 1)`, o número de Fibonacci anterior ao último calculado.

```
(%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*) Função

Expressa números de Fibonacci em termos da constante `%phi`, que é `(1 + sqrt(5))/2`, aproximadamente 1.61803399.

Por padrão, Maxima não conhece `%phi`. Após executar `tellrat (%phi^2 - %phi - 1)` e `algebraic: true`, `ratsimp` pode simplificar algumas expressões contendo `%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*) Função

Retorna a parte inteira da *n*'ésima raíz do valor absoluto de *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*) Função

Retorna símbolo de Jacobi de *p* e *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*) Função

Retorna o menor múltiplo comum entre seus argumentos. Os argumentos podem ser expressões gerais também inteiras.

`load ("functs")` chama essa função.

minfactorial (*expr*) Função

Examina *expr* procurando por ocorrências de dois fatoriais que diferem por um inteiro. `minfactorial` então converte um em um polinômio vezes o outro.

```
(%i1) n!/(n+2)!;
                                 n!
(%o1)                          --------
                               (n + 2)!
(%i2) minfactorial (%);
                                  1
(%o2)                     ---------------
                          (n + 1) (n + 2)
```

partfrac (*expr*, *var*) Função

Expande a expressão *expr* em frações parciais com relação à variável principal *var*. `partfrac` faz uma decomposição completa de fração parcial. O algorítmo utilizado é baseado no fato que os denominadores de uma expansão de fração parcial (os fatores do denominador original) são relativamente primos. Os numeradores podem ser escritos como combinação linear dos denominadores, e a expansão acontece.

```
(%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*) Função

Retorna `true` se `n` for um primo, `false` se não.

qunit (*n*) Função

Retorna a principal unidade do campo dos números quadráticos reais `sqrt (`*n*`)` onde *n* é um inteiro, i.e., o elemento cuja norma é unidade. Isso é importante para resolver a equação de Pell `a^2 -` *n* `b^2 = 1`.

```
(%i1) qunit (17);
(%o1)                        sqrt(17) + 4
(%i2) expand (% * (sqrt(17) - 4));
(%o2)                             1
```

totient (*n*) Função

Retorna o número de inteiros menores que ou iguais a *n* que são relativamente primos com *n*.

zerobern Variável de opção

Valor padrão: `true`

Quando `zerobern` for `false`, `bern` exclui os números de Bernoulli que forem iguais a zero. Veja `bern`.

zeta (*n*) Função

Retorna a função zeta de Riemann se *x* for um inteiro negativo, 0, 1, ou número par positivo, e retorna uma forma substantiva `zeta (`*n*`)` para todos os outros argumentos, incluindo não inteiros racionais, ponto flutuante, e argumentos complexos.

Veja também `bfzeta` e `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 Variável de opção

Valor padrão: `true`

Quando `zeta%pi` for `true`, `zeta` retorna uma expressão proporcional a `%pi^n` para inteiro par `n`. De outra forma, `zeta` retorna uma forma substantiva `zeta (n)` para inteiro par `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

THIS SECTION NEEDS TO BE TRANSLATED.

35 Grupos

35.1 Definições para Grupos

todd_coxeter (*relação, subgroupo*) Função
todd_coxeter (*relação*) Função

Acha a ordem de G/H onde G é o módulo do Grupo Livre *relação*, e H é o subgroupo de G gerado por *subgroupo*. *subgroupo* é um argumento opcional, cujo valor padrão é []. Em fazendo isso a função produz uma tabela de multiplicação à direita de G sobre G/H, onde os co-conjuntos são enumerados [H,Hg2,Hg3,...]. Isso pode ser visto internamente no `$todd_coxeter_state`.

As tabelas de multiplicação para as variáveis estão em `table:todd_coxeter_state[2]`. Então `table[i]` fornece a tabela para a i'ésima variável. `multiplos_co_conjuntos(co_conjunto,i) := table[varnum][co_conjunto];`

Exemplo:

```
(%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)}
```

Observe que somente os elementos de 1 a 20 desse array `%o8` são significativos. `table[1][4] = 7` indica coset4.var1 = coset7

36 Tempo de Execução

36.1 Introdução a Tempo de Execução

`maxima-init.mac` é um arquivo que é chamado automaticamente quando o Maxima inicia. Você pode usar `maxima-init.mac` para personalizar seu ambiente Maxima. `maxima-init.mac`, se existir, é tipicamente colocado no diretório chamado por `:lisp (default-userdir)`, embora possa estar em qualquer outro diretório procurado pela função `file_search`.

Aqui está um exemplo do arquivo `maxima-init.mac`:

```
setup_autoload ("specfun.mac", ultraspherical, assoc_legendre_p);
showtime:all;
```

Nesse Exemplo, `setup_autoload` diz ao Maxima para chamar o arquivo especificado (`specfun.mac`) se qualquer das funções (`ultraspherical`, `assoc_legendre_p`) forem chamadas sem estarem definidas. Dessa forma você não precisa lembrar de chamar o arquivo antes das funções.

A declaração `showtime: all` diz ao Maxima escolher a variável `showtime`. O arquivo `maxima-init.mac` pode conter qualquer outras atribuições ou outras declarações do Maxima.

36.2 Interrupções

O usuário pode parar uma computação que consome muito tempo com o caractere ^C (control-C). A ação padrão é parar a computação e mostrar outra linha de comando do usuário. Nesse caso, não é possível continuar a computação interrompida.

Se a variável `*debugger-hook*` é escolhida para `nil`, através do comando

```
:lisp (setq *debugger-hook* nil)
```

então na ocasião do recebimento do ^C, Maxima iniciará o depurador Lisp, e o usuário pode usar o depurador para inspecionar o ambiente Lisp. A computação interrompida pode ser retomada através do comando `continue` no depurador Lisp. O método de retorno para ao Maxima partindo do depurador Lisp (outro como executando a computação para complementação) é diferente para cada versão do Lisp.

Em sistemas Unix, o caratere ^Z (control-Z) faz com que Maxima pare tudo e aguarde em segundo plano, e o controle é retornado para a linha de comando do shell. O comando `fg` faz com que o Maxima retorne ao primeiro plano e continue a partir do ponto no qual foi interrompido.

36.3 Definições para Tempo de Execução

feature Declaration

Maxima compreende dois tipos distintos de recurso, recursos do sistema e recursos aplicados a expressões matemáticas. Veja Também `status` para informações sobre recursos do sistema. Veja Também `features` e `featurep` para informações sobre recursos matemáticos.

`feature` por si mesmo não é o nome de uma função ou variável.

featurep (*a*, *f*) Função

Tenta determinar se o objeto *a* tem o recurso *f* na base dos fatos dentro base de dados corrente. Se possue, é retornado `true`, de outra forma é retornado `false`.

Note que `featurep` retorna `false` quando nem *f* nem a negação de *f* puderem ser estabelecidas.

`featurep` avalia seus argumentos.

Veja também `declare` e `features`.

```
(%i1) declare (j, even)$
(%i2) featurep (j, integer);
(%o2)                          true
```

room () Função
room (*true*) Função
room (*false*) Função

Mostra uma descrição do estado de armazenamento e gerenciamento de pilha no Maxima. `room` chama a função Lisp de mesmo nome.

- `room ()` mostra uma descrição moderada.
- `room (true)` mostra uma descrição detalhada.
- `room (false)` mostra uma descrição resumida.

status (*feature*) Função
status (*feature, recurso_ativo*) Função
status (*status*) Função

Retorna informações sobre a presença ou ausência de certos recursos dependentes do sistema operacional.

- `status (feature)` retorna uma lista dos recursos do sistema. Inclui a versão do Lisp, tipo de sistema operacional, etc. A lista pode variar de um tipo de Lisp para outro.
- `status (feature,` *recurso_ativo*`)` retorna `true` se *recurso_ativo* está na lista de ítens retornada através de `status (feature)` e `false` de outra forma. `status` não avalia o argumento *recurso_ativo*. O operador aspas simples, `''`, evita a avaliação. Um recurso cujo nome contém um caractere especial, tal como um hífem, deve ser fornecido como um argumento em forma de seqüência de caracteres. Por Exemplo, `status (feature, "ansi-cl")`.
- `status (status)` retorna uma lista de dois elementos `[feature, status]`. `feature` e `status` são dois argumentos aceitos pela função `status`; Não está claro se essa lista tem significância adicional.

A variável `features` contém uma lista de recursos que se aplicam a expressões matemáticas. Veja `features` e `featurep` para maiores informações.

time (*%o1, %o2, %o3, ...*) Função

Retorna uma lista de tempos, em segundos, usados para calcular as linhas de saída `%o1`, `%o2`, `%o3`, ... O tempo retornado é uma estimativa do Maxima do tempo interno de computação, não do tempo decorrido. `time` pode somente ser aplicado a

variáveis(rótulos) de saída de linha; para quaisquer outras variáveis, `time` retorna `unknown` (tempo desconhecido).

Escolha `showtime: true` para fazer com que Maxima moste o tempo de computação e o tempo decorrido a cada linha de saída.

timedate () Função

Retorna uma seqüência de caracteres representando a data e hora atuais. A seqüência de caracteres tem o formato `HH:MM:SS Dia, mm/dd/aaaa (GMT-n)`, Onde os campos são horas, minutos, segundos, dia da semana, mês, dia do mês, ano, e horas que diferem da hora GMT.

O valor de retorno é uma seqüência de caracteres Lisp.

Exemplo:

```
(%i1) d: timedate ();
(%o1) 08:05:09 Wed, 11/02/2005 (GMT-7)
(%i2) print ("timedate mostra o tempo atual", d)$
timedate reports current time 08:05:09 Wed, 11/02/2005 (GMT-7)
```

37 Opções Diversas

37.1 Introdução a Opções Diversas

Nessa seção várias opções são tratadas pelo fato de possuirem um efeito global sobre a operação do Maxima. Também várias listas tais como a lista de todas as funções definidas pelo usuário, são discutidas.

37.2 Compartilhado

O diretório "share" do Maxima contém programas e outros arquivos de interesse para os usuários do Maxima, mas que não são parte da implementação do núcleo do Maxima. Esses programas são tipicamente chamados via `load` ou `setup_autoload`.

`:lisp *maxima-sharedir*` mostra a localização do diretório compartilhado dentro do sistema de arquivos do usuário.

`printfile ("share.usg")` imprime uma lista de pacotes desatualizados dos pacotes compartilhados. Usuários podem encontrar isso de forma mais detalhada navegando no diretório compartilhado usando um navegador de sistema de arquivo.

37.3 Definições para Opções Diversas

aliases Variável de sistema

Valor padrão: `[]`

`aliases` é a lista de átomos que possuem um alias definido pelo usuário (escolhido através das funções `alias`, `ordergreat`, `orderless` ou através da declaração do átomo como sendo um `noun` (substantivo) com `declare`).

alphabetic Declaration

`declare (`*char*`, alphabetic)` adiciona `char` (caracteres) ao alfabeto do Maxima, que inicialmente contém as letras de `A` até `Z`, de `a` até `z`, `%` e `_`. *char* é especificado como uma seqüência de caracteres de comprimento 1, e.g., `"~"`.

```
(%i1) declare ("~", alphabetic);
(%o1)                          done
(%i2) foo~bar;
(%o2)                         foo~bar
(%i3) atom (foo~bar);
(%o3)                          true
```

apropos (*string*) Função

Procura por nomes Maxima que possuem *string* aparecendo em qualquer lugar dentro de seu nome. Dessa forma, `apropos (exp)` retorna uma lista de todos os sinalizadores e funções que possuem `exp` como parte de seus nomes, tais como `expand`, `exp`, e `exponentialize`. Dessa forma você pode somente lembra parte do nome de alguma coisa você pode usar esse comando para achar o restante do nome. Similarmente, você pode dizer `apropos (tr_)` para achar uma lista de muitos dos comutadores relatando para o tradutor, muitos dos quais começam com `tr_`.

args (*expr*) Função

Retorna a lista de argumentos de `expr`, que pode ser de qualquer tipo de expressão outra como um átomo. Somente os argumentos do operador de nível mais alto são extraídos; subexpressões de `expr` aparecem como elementos ou subexpressões de elementos da lista de argumentos.

A ordem dos ítens na lista pode depender do sinalizador global `inflag`.

`args` (*expr*) é equivalente a `substpart ("[", expr, 0)`. Veja também `substpart`.

Veja também `op`.

genindex Variável de opção

Valor padrão: `i`

`genindex` é o prefixo usado para gerar a próxima variável do somatório quando necessário.

gensumnum Variável de opção

Valor padrão: 0

`gensumnum` é o sufixo numérico usado para gerar variável seguinte do somatório. Se isso for escolhido para `false` então o índice consistirá somente de `genindex` com um sufixo numérico.

inf Constante

Infinito positivo real.

infinity Constante

Infinito complexo, uma magnitude infinita de ângulo de fase arbitrária. Veja também `inf` e `minf`.

infolists Variável de sistema

Valor padrão: `[]`

`infolists` é uma lista dos nomes de todas as listas de informação no Maxima. São elas:

`labels` Todos associam `%i`, `%o`, e rótulos `%t`.

`values` Todos associam átomos que são variáveis de usuário, não opções do Maxima ou comutadores, criados através de `:` ou `::` ou associando funcionalmente.

`functions`
Todas as funções definidas pelo usuário, criadas através de `:=` ou `define`.

`arrays` Todos os arrays declarados e não declarados, criados através de `:`, `::`, ou `:=`.

`macros` Todas as macros definidas pelo usuário.

`myoptions`
Todas as opções alguma vez alteradas pelo usuário (mesmo que tenham ou não elas tenham mais tarde retornadas para seus valores padrão).

`rules` Todos os modelos definidos pelo usuário que coincidirem e regras de simplificação, criadas através de `tellsimp`, `tellsimpafter`, `defmatch`, ou `defrule`.

`aliases` Todos os átomos que possuem um alias definido pelo usuário, criado através das funções `alias`, `ordergreat`, `orderless` ou declarando os átomos como um `noun` com `declare`.

`dependencies`
Todos os átomos que possuem dependências funcionais, criadas através das funções `depends` ou `gradef`.

`gradefs` Todas as funções que possuem derivadas definidas pelo usuário, cridas através da função `gradef`.

`props` Todos os átomos que possuem quaisquer propriedades outras que não essas mencionadas acima, tais como propriedades estabelecidas por `atvalue` , `matchdeclare`, etc., também propriedades estabelecidas na função `declare`.

`let_rule_packages`
Todos os pacote de régras em uso definidos pelo usuário mais o pacote especial `default_let_rule_package`. (`default_let_rule_package` é o nome do pacote de régras usado quando um não está explicitamente escolhido pelo usuário.)

integerp (*expr*) Função

Retorna `true` se *expr* é um inteiro numérico literal, de outra forma retorna `false`.

`integerp` retorna falso se seu argumento for um símbolo, mesmo se o argumento for declarado inteiro.

Exemplos:

```
(%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 Variável de opção

Valor padrão: `false`

`m1pbranch` é principal descendente de `-1` a um expoente. Quantidades tais como `(-1)^(1/3)` (isto é, um expoente racional "ímpar") e `(-1)^(1/4)` (isto é, um expoente racional "par") são manuseados como segue:

```
                  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*) Função

Retorna `true` se *expr* for um inteiro literal, número racional, número em ponto flutuante, ou um grande número em ponto flutuante, de outra forma retorna `false`.

`numberp` retorna falso se seu argumento for um símbolo, mesmo se o argumento for um número simbólico tal como `%pi` ou `%i`, ou declarado ser par, ímpar, inteiro, racional, irracional, real, imaginário, ou complexo.

Exemplos:

```
(%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*) Função

Retorna uma lista de nomes de todas as propriedades associadas com o átomo *a*.

props Símbolo especial

`props` são átomos que possuem qualquer propriedade outra como essas explicitamente mencionadas em `infolists`, tais como atvalues, matchdeclares, etc., também propriedades especificadas na função `declare`.

propvars (*prop*) Função

Retorna uma lista desses átomos sobre a lista `props` que possui a propriedade indicada através de *prop*. Dessa forma `propvars (atvalue)` retorna uma lista de átomos que possuem atvalues.

put (*átomo*, *valor*, *indicador*) Função

Atribui *valor* para a propriedade (especificada através de *indicador*) do *átomo*. *indicador* pode ser o nome de qualquer propriedade, não apenas uma propriedade definida pelo sistema.

`put` avalia seus argumentos. `put` retorna *valor*.

Exemplos:

```
(%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 (*átomo*, *valor*, *indicador*) Função

Atribui *valor* para a propriedade (especificada através de *indicador*) do *átomo*. Isso é o mesmo que `put`, exceto que os argumentos nã são avaliados.

Exemplo:

```
(%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 (*átomo*, *indicador*) Função

Remove a propriedade indicada através de *indicador* do *átomo*.

remove (*a_1, p_1, ..., a_n, p_n*) Função
remove ([*a_1, ..., a_m*], [*p_1, ..., p_n*], ...) Função
remove ("*a*", *operator*) Função
remove (*a, transfun*) Função
remove (*all, p*) Função

Remove propriedades associadas a átomos.

`remove (a_1, p_1, ..., a_n, p_n)` remove a propriedade `p_k` do átomo `a_k`.

`remove ([a_1, ..., a_m], [p_1, ..., p_n], ...)` remove as propriedades *p_1*, ..., *p_n* dos átomos *a_1*, ..., *a_m*. Pode existir mais que um par de listas.

`remove (all, p)` remove a propriedade *p* de todos os átomos que a possuem.

A propriedade removida pode ser definida pelo sistema tal como `function`, `macro` ou `mode_declare`, ou propriedades definidas pelo usuário.

uma propriedade pode ser `transfun` para remover a versão traduzida Lisp de uma função. Após executar isso, a versão Maxima da função é executada em lugar da versão traduzida.

`remove ("a", operator)` ou, equivalentemente, `remove ("a", op)` remove de *a* as propriedades `operator` declaradas através de `prefix`, `infix`, `nary`, `postfix`, `matchfix`, ou `nofix`. Note que o nome do operador deve ser escrito como uma seqüência de caracteres com apóstofo.

`remove` sempre retorna `done` se um átomo possui ou não uma propriedade especificada. Esse comportamento é diferente das funções remove mais específicas `remvalue`, `remarray`, `remfunction`, e `remrule`.

remvalue (*nome_1, ..., nome_n*) Função
remvalue (*all*) Função

Remove os valores de Variáveis de usuário *nome_1*, ..., *nome_n* (que podem ser subscritas) do sistema.

`remvalue (all)` remove os valores de todas as variáveis em `values`, a lista de todas as variáveis nomeadas através do usuário (em oposição a essas que são automaticamente atribuídas através do Maxima).

Veja também `values`.

rncombine (*expr*) Função

Transforma *expr* combinando todos os termos de *expr* que possuem denominadores idênticos ou denominadores que diferem de cada um dos outros apenas por fatores numéricos somente. Isso é ligeiramente diferente do comportamento de de `combine`, que coleta termos que possuem denominadores idênticos.

Escolhendo `pfeformat: true` e usando `combine` retorna resultados similares a esses que podem ser obtidos com `rncombine`, mas `rncombine` pega o passo adicional de multiplicar cruzado fatores numérios do denominador. Esses resultados em forma ideal, e a possibilidade de reconhecer alguns cancelamentos.

scalarp (*expr*) Função

Retorna `true` se *expr* for um número, constante, ou variável declarada `scalar` com `declare`, ou composta inteiramente de números, constantes, e tais Variáveis, bmas não contendo matrizes ou listas.

setup_autoload (*nomedearquivo, função_1, ..., função_n*) Função

Especifica que se qualquer entre *função_1*, ..., *função_n* for referenciado e não ainda definido, *nomedeqrquivo* é chamado via `load`. *nomedearquivo* usualmente contém definições para as funções especificadas, embora isso não seja obrigatório.

`setup_autoload` não trabalha para funções array.

`setup_autoload` não avalia seus argumentos.

Exemplo:

```
(%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 função ultraspherical
Warning - you are redefining the Macsyma função 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 Regras e Modelos

38.1 Introdução a Regras e Modelos

Essa seção descreve coincidências de modelos definidos pelo usuário e regras de simplificação. Existem dois grupos de funções que implementam até certo ponto diferentes esquemas de coincidência de modelo. Em um grupo estão `tellsimp`, `tellsimpafter`, `defmatch`, `defrule`, `apply1`, `applyb1`, e `apply2`. Em outro grupo estão `let` e `letsimp`. Ambos os esquemas definem modelos em termos de variáveis de modelo declaradas por `matchdeclare`.

Regras de coincidência de modelos definidas por `tellsimp` e `tellsimpafter` são aplicadas automaticamente através do simplificador do Maxima. Regras definidas através de `defmatch`, `defrule`, e `let` são aplicadas através de uma chamada explícita de função.

Existe mecanismos adicionais para regras aplicadas a polinômios através de `tellrat`, e para álgebra comutativa e não comutativa no pacote `affine`.

38.2 Definições para Regras e Modelos

apply1 (*expr, rule_1, ..., rule_n*) Função

Repetidamente aplica *rule_1* a *expr* até que isso falhe, então repetidamente aplica a mesma regra a todas as subexpressões de *expr*, da esquerda para a direita, até que *rule_1* tenha falhado sobre todas as subexpressões. Chama o resultado da transformação de *expr* dessa maneira de *expr_2*. Então *rule_2* é aplicada no mesmo estilo iniciando no topo de *expr_2*. Quando *rule_n* falhar na subexpressão final, o resultado é retornado.

`maxapplydepth` é a intensidade de nível mais distante de subexpressões processadas por `apply1` e `apply2`.

Veja também `applyb1`, `apply2`, e `let`.

apply2 (*expr, rule_1, ..., rule_n*) Função

Se *rule_1* falhar sobre uma dada subexpressão, então *rule_2* é repetidamente aplicada, etc. Somente se todas as regras falharem sobre uma dada subexpressão é que o conjunto completo de regras é repetidamente aplicada à próxima subexpressão. Se uma das regras obtém sucesso, então a mesma subexpressão é reprocessada, iniciando com a primeira regra.

`maxapplydepth` é a intensidade do nível mais distante de subexpressões processadas através de `apply1` e `apply2`.

Veja também `apply1` e `let`.

applyb1 (*expr, rule_1, ..., rule_n*) Função

Repetidamente aplica *rule_1* para a subexpressão mais distante de *expr* até falhar, então repetidamente aplica a mesma regra um nível mais acima (i.e., subexpressãos mais larga), até que *rule_1* tenha falhado sobre a expressão de nível mais alto. Então *rule_2* é aplicada com o mesmo estilo para o resultado de *rule_1*. após *rule_n* ter sido aplicada à expressão de nível mais elevado, o resultado é retornado.

`applyb1` é similar a `apply1` mas trabalha da base para cima em lugar de do topo para baixo.

`maxapplyheight` é o ápice que `applyb1` encontra antes de interromper.

Veja também `apply1`, `apply2`, e `let`.

current_let_rule_package Variável de opção

Valor padrão: `default_let_rule_package`

`current_let_rule_package` é o nome do pacote de regras que está sendo usado por funções no pacote `let` (`letsimp`, etc.) se nenhum outro pacote de regras for especificado. A essa variável pode ser atribuído o nome de qualquer pacote de regras definido via comando `let`.

Se uma chamada tal como `letsimp (expr, nome_pct_regras)` for feita, o pacote de regras `nome_pct_regras` é usado para aquela chamada de função somente, e o valor de `current_let_rule_package` não é alterado.

default_let_rule_package Variável de opção

Valor padrão: `default_let_rule_package`

`default_let_rule_package` é o nome do pacote de regras usado quando um não for explicitamente escolhido pelo usuário com `let` ou através de alteração do valor de `current_let_rule_package`.

defmatch (*prognome, modelo, x_1, ..., x_n*) Função

Cria uma função *prognome* (*expr*, *y_1*, ..., *y_n*) que testa *expr* para ver se essa expressão coincide com *modelo*.

modelo é uma expresão contendo as variáveis de modelo *x_1*, ..., *x_n* e parmetros de modelo, se quaisquer. As variáveis de modelo são dadas explicitamente como argumentos para `defmatch` enquanto os parmetros de modelo são declarados através da função `matchdeclare`.

O primeiro argumento para a função criada *prognome* é uma expressão a ser comparada contra o modelo e os outros argumentos são as variáveis atuais *y_1*, ..., *y_n* ne expressão que corresponde às variáveis correspondentes *x_1*, ..., *x_n* no modelo.

Se a tentativa de coincidência obtiver sucesso, *progname* retorna uma lista de equações cujos lados esquerdos são as variáveis de modelo e os parmetros de modelo, e cujos lados direitos são expressões cujas variáveis de modelo e modelos coincidirão. Os parmetros de modelo, mas não as variáveis de modelo, são atribuídos às subexpressões que elas coincidem. Se a coincidência falhar, *prognome* retorna `false`.

Quaisquer variáveis não declaradas como parmetros de modelo em `matchdeclare` ou como variáveis em `defmatch` coincidem somente consigo mesmas.

Um modelo que não contiver nenhuma variável de modelo ou parmetros retorna `true` se a coincidência ocorre.

Veja também `matchdeclare`, `defrule`, `tellsimp`, e `tellsimpafter`.

Exemplos:

Esse `defmatch` define a função `linearp (expr, y)`, que testa `expr` para ver se essa expressão é da forma `a*y + b` tal que `a` e `b` não contenham `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
```

Se o terceiro argumento para `defmatch` na linha (%i2) tiver sido omitido, então `linear` pode somente coincidir com expressões lineares em x, não em qualquer outra variável.

```
(%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 (*nomeregra, modelo, substituição*) Função

Define e nomeia uma regra de substituição para o modelo dado. Se a regra nomeada *nomeregra* for aplicada a uma expressão (através de `apply1`, `applyb1`, ou `apply2`), toda subexpressão coincidindo com o modelo irá ser substituída por `substituição`. Todas as variáveis em `substituição` que tiverem sido atribuidos valores pela coincidência com o modelo são atribuidas esses valores na `substituição` que é então simplificado.

As regras por si mesmas podem ser tratadas como funções que transforma uma expressão através de uma operação de coincidência de modelo e substituição. Se a coincidência falhar, a expressão original é retornada.

disprule (*nomeregra_1, ..., nomeregra_2*) Função
disprule (*all*) Função

Mostra regras com os nomes *nomeregra_1*, ..., *nomeregra_n*, como retornado por `defrule`, `tellsimp`, ou `tellsimpafter`, ou um modelo definido por meio de `defmatch`.

Por exemplo, a primeira regra modificando `sin` é nomeada `sinrule1`.

`disprule (all)` mostra todas as regras.

Veja também `letrules`, que mostra regras definidas através de `let`.

let (*prod, repl, prednome, arg_1, ..., arg_n*) Função
let ([*prod, repl, prednome, arg_1, ..., arg_n*], *nome_pacote*) Função

Define uma regra de substituição para `letsimp` tal que *prod* é substituído por *repl*. *prod* é um produto de expoentes positivos ou negativos dos seguintes termos:

- Atomos que `letsimp` irá procurar literalmente a menos que previamente chamando `letsimp` a função `matchdeclare` é usada para associar um predicado com o átomo. Nesse caso `letsimp` irá coincidir com o átomo para qualquer termo de um produto satisfazendo o predicado.
- Núcleos tais como `sin(x)`, `n!`, `f(x,y)`, etc. Como com átomos acima `letsimp` irá olhar um literal coincidente a menos que `matchdeclare` seja usada para associar um predicado com o argumento do núcleo.

Um termo para um expoente positivo irá somente coincidir com um termo tendo ao menos aquele expoente. Um termo para um expoente negativo por outro lado irá somente coincidir com um termo com um expoente ao menos já negativo. o caso de expentes negativos em *prod* o comutador `letrat` deve ser escolhido para `true`. Veja também `letrat`.

Se um predicado for incluído na função `let` seguido por uma lista de argumentos, uma tentativa de coincidência (i.e. uma que pode ser aceita se o predicado fosse omitido) é aceita somente se `prednome (arg_1', ..., arg_n')` avaliar para `true` onde *arg_i'* é o valor coincidente com *arg_i*. O *arg_i* pode ser o nome de qualquer átomo ou o argumento de qualquer núcleo aparecendo em *prod*. *repl* pode ser qualquer expressão racional. Se quaisquer dos átomos ou argumentos de *prod* aparecer em *repl* a substituição é feita.

O sinalizador global `letrat` controla a simplificação dos quocientes através de `letsimp`. Quando `letrat` for `false`, `letsimp` simplifica o numerador e o denominador de *expr* separadamente, e não simplifica o quociente. Substituições tais como `n!/n` vão para `(n-1)!` então falham quando `letrat` for `false`. Quando `letrat` for `true`, então o numerador, o denominador, e o quociente são simplificados nessa ordem.

Essas funções de substituição permitem a você trabalhar com muitos pacotes de regras. Cada pacote de regras pode conter qualquer número de regras `let` e é referenciado através de um nome definido pelo usuário. `let ([`*prod*`,` *repl*`,` *prednome*`,` *arg_1*`, ...,` *arg_n*`],` *nome_pacote*`)` adiciona a regra *prednome* ao pacote de regras *nome_pacote*. `letsimp (`*expr*`,` *nome_pacote*`)` aplica as regras em *nome_pacote*. `letsimp (`*expr*`,` *nome_pacote1*`,` *nome_pacote2*`, ...)` é equivalente a `letsimp (`*expr*`,` *nome_pacote1*`)` seguido por `letsimp (%,` *nome_pacote2*`)`, ...

`current_let_rule_package` é o nome do pacote de regras que está atualmente sendo usando. Essa variável pode receber o nome de qualquer pacote de regras definidos via o comando `let`. Quando qualquer das funções compreendidas no pacote `let` são chamadas sem o nome do pacote, o pacote nomeado por `current_let_rule_package` é usado. Se uma chamada tal como `letsimp (`*expr*`,` *nome_pct_regras*`)` é feita, o pacote de regras *nome_pct_regras* é usado somente para aquele comando `letsimp`, e `current_let_rule_package` não é alterada. Se não especificado de outra forma, `current_let_rule_package` avalia de forma padronizada para `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);
                             (m - 1)! n!
(%o6)                        -----------
                                 m
(%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 Variável de opção

Valor padrão: `false`

Quando `letrat` for `false`, `letsimp` simplifica o numerador e o denominador de uma razão separadamente, e não simplifica o quociente.

Quando `letrat` for `true`, o numerador, o denominador, e seu quocienten são simplificados nessa ordem.

```
(%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 () Função
letrules (*nome_pacote*) Função

Mostra as regras em um pacote de regras. `letrules ()` mostra as regras no pacote de regras corrente. `letrules (`*nome_pacote*`)` mostra as regras em `nome_pacote`.

O pacote de regras corrente é nomeado por `current_let_rule_package`. Se não especificado de outra forma, `current_let_rule_package` avalia de forma padrão para `default_let_rule_package`.

Veja também `disprule`, que mostra regras definidas por `tellsimp` e `tellsimpafter`.

letsimp (*expr*) Função
letsimp (*expr, nome_pacote*) Função
letsimp (*expr, nome_pacote_1, ..., nome_pacote_n*) Função

Repetidamente aplica a substituição definida por `let` até que nenhuma mudança adicional seja feita para *expr*.

`letsimp (`*expr*`)` usa as regras de `current_let_rule_package`.

`letsimp (`*expr*`, `*nome_pacote*`)` usa as regras de *nome_pacote* sem alterar `current_let_rule_package`.

`letsimp (`*expr*`, `*nome_pacote_1*`, ..., `*nome_pacote_n*`)` é equivalente a `letsimp (`*expr*`, `*nome_pacote_1*`)` seguido por `letsimp (%, `*nome_pacote_2*`)`, e assim sucessivamente.

let_rule_packages Variável de opção

Valor padrão: `[default_let_rule_package]`

`let_rule_packages` é uma lista de todos os pacotes de regras `let` definidos pelo usuário mais o pacote padrão `default_let_rule_package`.

matchdeclare (*a_1, pred_1, ..., a_n, pred_n*) Função

Associa um predicado *pred_k* com uma variável ou lista de variáveis *a_k* de forma que *a_k* coincida com expressões para as quais o predicado retorne qualquer coisa que não `false`.

O predicado é o nome de uma função, uma chamada de função omitindo o último argumento, ou `true`. Qualquer expressão coincide com `true`. Se o predicado for especificado como uma chamada de função, a expressão a ser testada é anexada ao final da lista de argumentos; os argumentos são avaliados ao mesmo tempo que a coincidência é avaliada. De outra forma, o predicado é especificado como um nome de função, e a expressão a ser testada é o argumento sozinho. Uma função predicado não precisa ser definida quando `matchdeclare` for chamada; o predicado não é avaliado até que uma coincidência seja tentada.

Um predicado `matchdeclare` não pode ser qualquer tipo de expressão outra que não um nome de função ou chamada de função. Em particular, um predicado não pode ser um `lambda` ou `block`.

Se uma expressão satisfaz uma coincidência de predicado, a variável de coincidência é atribuída à expressão, exceto para variáveis de coincidência que são operandos de adição `+` ou multiplicação `*`. Somente adição e multiplicação são manuseadas de forma

especial; outros operadores enários (ambos os definidos internamente e os definidos pelo usuário) são tratados como funções comuns.

No caso de adição e multiplicação, a variável de coincidência pode ser atribuida a uma expressão simples que satisfaz o predicado de coincidência, ou uma adição ou um produto (respectivamente) de tais expressões. Tal coincidência de termo multiplo é gulosa: predicados são avaliados na ordem em que suas variáveis associadas aparecem no modelo de coincidência, e o termo que satisfizer mais que um predicado é tomado pelo primeiro predicado que satisfizer. Cada predicado é testado contra todos os operandos de adição ou produto antes que o próximo predicado seja avaliado. Adicionalmente, se 0 ou 1 (respectivamente) satisfazem um predicado de coincidência, e não existe outros termos que satisfaçam o predicado, 0 ou 1 é atribuído para a variável de coincidência associada com o predicado.

O algorítmo para processar modelos contendo adição e multiplicação faz alguns resultados de coincidência (por exemplo, um modelo no qual uma variável "coincida com qualquer coisa" aparecer) dependerem da ordem dos termos no modelo de coincidência e na expressão a ser testada a coincidência. Todavia, se todos os predicados de coincidência são mutuamente exclusivos, o resultado de coincidência é insensível a ordenação, como um predicado de coincidência não pode aceitar termos de coincidência de outro.

Chamado `matchdeclare` com uma variável *a* como um argumento muda a propriedade `matchdeclare` para *a*, se a variável *a* tiver sido declarada anteriormente; somente o `matchdeclare` mais recente está em efeito quando uma regra é definida, mudanças posteriores para a propriedade `matchdeclare` (via `matchdeclare` ou `remove`) não afetam regras existentes.

`propvars (matchdeclare)` retorna a lista de todas as variáveis para as quais exista uma propriedade `matchdeclare`. `printprops (`*a*`, matchdeclare)` retorna o predicado para a variável `a`. `printprops (all, matchdeclare)` retorna a lista de predicados para todas as variáveis `matchdeclare`. `remove (`*a*`, matchdeclare)` remove a propriedade `matchdeclare` da variável *a*.

As funções `defmatch`, `defrule`, `tellsimp`, `tellsimpafter`, e `let` constroem regras que testam expressões contra modelos.

`matchdeclare` coloca apóstrofo em seus argumentos. `matchdeclare` sempre retorna `done`.

Exemplos:

- `q` coincide com uma expressão que não contém `x` ou `%e`.

```
(%i1) matchdeclare (q, freeof (x, %e))$
```

matchfix (*delimitador_e, delimitador_d*) Função
matchfix (*delimitador_e, delimitador_d, arg_pos, pos*) Função

Declara um operador `matchfix` com delimitadores esquerdo e direito *delimitador_e* and *delimitador_d*. Os delimitadores são especificados como seqüências de caracteres.

Um operador "matchfix" é uma função que aceita qualquer número de argumentos, tal que os argumentos ocorram entre os delimitadores correspondentes esquerdo e direito. Os delimitadores podem ser quaisquer seqüências de caracteres, contanto que o analisador de expressões do Maxima possa distingüir os delimitadores dos operandos

e de outras expressões e operadores. Na prática essas regras excluem delimitadores não analisáveis tais como `%`, `,`, `$` e `;`, e pode ser necessário isolar os delimitadores com espaços em branco. O delimitador da direita pode ser o mesmo ou diferente do delimitador da esquerda.

Um delimitador esquerdo pode ser associado com somente um delimitador direito; dois diferentes operadores `matchfix` não podem ter o mesmo delimitador esquerdo.

Um operador existente pode ser redeclarado com um operador `matchfix` sem alterar suas outras propriedades. Particularmente, operadores internos tais como adição `+` podem ser declarados `matchfix`, mas funções operadores não podem ser definidas para operadores internos.

`matchfix` (*delimitador_e*, *delimitador_d*, *arg_pos*, *pos*) declara o argumento *arg_pos* como sendo um entre: expressão lógica, expressão comum do Maxima mas que não seja do tipo anterior, e qualquer outro tipo de expressão que não esteja incluída nos dois primeiros tipos. Essa declaração resulta em *pos* sendo um entre: expressão lógica, expressão comum do Maxima mas que não seja do tipo anterior, e qualquer outro tipo de expressão que não esteja incluída nos dois primeiros tipos e os delimitadores *delimitador_e* e *delimitador_d*.

A função para realizar uma operação `matchfix` é uma função comum definida pelo usuário. A função operador é definida da forma usual com o operador de definição de função `:=` ou `define`. Os argumentos podem ser escritos entre os delimitadores, ou com o delimitador esquerdo com uma seqüência de caracteres com apóstrofo e os argumentos seguindo entre parêntesis. `dispfun` (*delimitador_e*) mostra a definição da função operador.

O único operador interno `matchfix` é o construtor de listas `[ ]`. Parêntesis `( )` e aspas duplas `" "` atuam como operadores `matchfix`, mas não são tratados como tal pelo analisador do Maxima.

`matchfix` avalia seus argumentos. `matchfix` retorna seu primeiro argumento, *delimitador_e*.

Exemplos:

- Delimitadores podem ser quase quaisquer seqüência de caracteres.

```
(%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~
```

- Operadores `matchfix` são funções comuns definidas pelo usuário.

```
(%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, nome*) Função
remlet () Função
remlet (*all*) Função
remlet (*all, nome*) Função

Apaga a regra de substituiçao, prod –> repl, mais recentemente definida através dea função `let`. Se `nome` for fornecido a regra é apagada do pacote de regras chamado `nome`.

`remlet()` e `remlet(all)` apagam todas as regras de substituição do pacote de regras corrente. Se o nome de um pacote de regras for fornecido, e.g. `remlet (all,` *nome*`)`, o pacote de regras *nome* é também apagado.

Se uma substituição é para ser mudada usando o mesmo produto, `remlet` não precisa ser chamada, apenas redefina a substituição usando o mesmo produto (literalmente) com a função `let` e a nova substituição e/ou nome de predicado. Pode agora `remlet` (*prod*) ser chamada e a regra de substituição original é ressuscitada.

Veja também `remrule`, que remove uma regra definida através de `tellsimp` ou de `tellsimpafter`.

remrule (*op, nomeregra*) Função
remrule (*op, all*) Função

Remove regras definidas por `tellsimp`, ou `tellsimpafter`.

`remrule (`*op*`, `*nomeregra*`)` remove a regra com o nome `nomeregra` do operador *op*. Quando *op* for um operador interno ou um operador definido pelo usuário (como definido por `infix`, `prefix`, etc.), *op* e *rulename* devem ser colocados entre aspas duplas.

`remrule (`*op*`, all)` remove todas as regras para o operador *op*.

Veja também `remlet`, que remove uma regra definida através de `let`.

Examples:

```
(%i1) tellsimp (foo (aa, bb), bb - aa);
(%o1)                    [foorule1, false]
(%i2) tellsimpafter (aa + bb, special_add (aa, bb));
(%o2)                    [+rule1, simplus]
(%i3) infix ("@");
(%o3)                              @
(%i4) tellsimp (aa @ bb, bb/aa);
(%o4)                    [@rule1, false]
(%i5) tellsimpafter (quux (%pi, %e), %pi - %e);
(%o5)                   [quuxrule1, false]
(%i6) tellsimpafter (quux (%e, %pi), %pi + %e);
(%o6)              [quuxrule2, quuxrule1, false]
(%i7) [foo (aa, bb), aa + bb, aa @ bb, quux (%pi, %e), quux (%e, %pi)];
                                          bb
(%o7) [bb - aa, special_add(aa, bb), --, %pi - %e, %pi + %e]
                                          aa
(%i8) remrule (foo, foorule1);
(%o8)                             foo
(%i9) remrule ("+", "+rule1");
(%o9)                              +
(%i10) remrule ("@", "@rule1");
(%o10)                             @
(%i11) remrule (quux, all);
(%o11)                           quux
(%i12) [foo (aa, bb), aa + bb, aa @ bb, quux (%pi, %e), quux (%e, %pi)];
(%o12) [foo(aa, bb), bb + aa, aa @ bb, quux(%pi, %e),
                                                  quux(%e, %pi)]
```

tellsimp (*pattern, replacement*) Função

é similar a `tellsimpafter` mas coloca nova informação antes da antiga de forma que essa nova regra seja aplicada antes das regras de simplificação internas.

`tellsimp` é usada quando for importante modificar a expressão antes que o simplificador trabalhe sobre ela, por exemplo se o simplificador "sabe" alguma coisa sobre a expressão, mas o que ele retorna não é para sua apreciação. Se o simplificador "sabe" alguma coisa sobre o principal operador da expressão, mas está simplesmente escondendo de você, você provavelmente quer usar `tellsimpafter`.

O modelo pode não ser uma adição, um produto, variável simples, ou número.

`rules` é a lista de regras definidas por `defrule`, `defmatch`, `tellsimp`, e `tellsimpafter`.

Exemplos:

```
(%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 (*modelo, substituição*) Função

Define a uma regra de simplificação que o simplificador do Maxima aplica após as regras de simplificação internas. *modelo* é uma expressão, compreendendo variáveis de modelo (declaradas através de `matchdeclare`) e outros átomos e operaçes, considerados literais para o propósito de coincidência de modelos. *substituição* é substituída

para uma expressão atual que coincide com *modelo*; variáveis de modelo em *substituição* são atribuidas a valores coincidentes na expressão atual.

modelo pode ser qualquer expressão não atômica na qual o principal operador não é uma variável de modelo; a regra de simplificação está associada com o operador principal. Os nomes de funções (com uma excessão, descrita abaixo), listas, e arrays podem aparecer em *modelo* como o principal operador somente como literais (não variáveis de modelo); essas regras fornecem expressões tais como `aa(x)` e `bb[y]` como modelos, se `aa` e `bb` forem variáveis de modelo. Nomes de funções, listas, e arrays que são variáveis de modelo podem aparecer como operadores outros que não o operador principal em *modelo*.

Existe uma excessão para o que foi dito acima com relação a regras e nomes de funções. O nome de uma função subscrita em uma expressão tal como `aa[x](y)` pode ser uma variável de modelo, porque o operador principal não é `aa` mas ao contrário o átomo Lisp `mqapply`. Isso é uma conseqüência da representação de expressões envolvendo funções subscritas.

Regras de simplificação são aplicadas após avaliação (se não suprimida através de colocação de apóstrofo ou do sinalizador `noeval`). Regras estabelecidas por `tellsimpafter` são aplicadas na ordem em que forem definidas, e após quaisquer regras internas. Regras são aplicadas de baixo para cima, isto é, aplicadas primeiro a subexpressões antes de ser aplicada à expressão completa. Isso pode ser necessário para repetidamente simplificar um resultado (por exemplo, via o operador apóstrofo-apóstrofo `''` ou o sinalizador `infeval`) para garantir que todas as regras são aplicadas.

Variáveis de modelo são tratadas como variáveis locais em regras de simplificação. Assim que uma regra é definida, o valor de uma variável de modelo não afeta a regra, e não é afetado pela regra. Uma atribuição para uma variável de modelo que resulta em uma coincidência de regra com sucesso não afeta a atribuição corrente (ou necessita disso) da variável de modelo. Todavia, como com todos os átomos no Maxima, as propriedades de variáveis de modelo (como declarado por `put` e funções relacionadas) são globais.

A regra construída por `tellsimpafter` é nomeada após o operador principal de `modelo`. Regras para operadores internos, e operadores definidos pelo usuário definidos por meio de `infix`, `prefix`, `postfix`, `matchfix`, e `nofix`, possuem nomes que são seqüências de caracteres do Maxima. Regras para outras funções possuem nomes que são identificadores comuns do Maxima.

O tratamento de substantivos e formas verbais é desprezívelmente confuso. Se uma regra é definida para uma forma substantiva (ou verbal) e uma regra para o verbo correspondente (ou substantivo) já existe, então a nova regra definida aplica-se a ambas as formas (substantiva e verbal). Se uma regra para a correspondente forma verbal (ou substantiva) não existe, a nova regra definida aplicar-se-á somente para a forma substantiva (ou verbal).

A regra construída através de `tellsimpafter` é uma função Lisp comum. Se o nome da regra for `$foorule1`, a construção `:lisp (trace $foorule1)` rastreia a função, e `:lisp (symbol-function '$foorule1` mostra sua definição.

`tellsimpafter` não avalia seus argumentos. `tellsimpafter` retorna a lista de regras para o operador principal de *modelo*, incluindo a mais recente regra estabelecia.

Veja também `matchdeclare`, `defmatch`, `defrule`, `tellsimp`, `let`, `kill`, `remrule`, e `clear_rules`.

Exemplos:

modelo pode ser qualquer expressão não atômica na qual o principal operador não é uma variável de modelo.

```
(%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))
```

Regras são aplicadas na ordem em que forem definidas. Se duas regras podem coincidir com uma expressão, a regra que foi primeiro definida é a que será aplicada.

```
(%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)
```

variáveis de modelo são tratadas como variáveis locais em regras de simplificação. (Compare a `defmatch`, que trata variáveis de modelo como variáveis globais.)

```
(%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
```

Como com todos os átomos, propriedades de variáveis de modelo são globais embora valores sejam locais. Nesse exemplo, uma propriedade de atribuição é declarada via `define_variable`. Essa é a propriedade do átomo `bb` através de todo o 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);
```

Regras são nomeadas após operadores principais. Nomes de regras para operadores internos e operadores definidos pelo usuário são seqüências de caracteres, enquanto nomes para outras funções são identificadores comuns.

```
(%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 () Função

Executa `kill (rules)` e então re-escolhe o próximo número de regra para 1 para adição `+`, multiplicação `*`, e exponenciação `^`.

39 Listas

39.1 Introdução a Listas

Listas são o bloco básico de construção para Maxima e Lisp.**Todos os outros tipos de dado como arrays, tabelas desordenadas, números são representados como listas Lisp. Essas listas Lisp possuem a forma

```
((MPLUS) $A 2)
```

para indicar a expressão `a+2`.**No nível um do Maxima poderemos ver a notação infixa `a+2`.**Maxima também tem listas que foram impressas como

```
[1, 2, 7, x+y]
```

para uma lista com 4 elementos.**Internamente isso corresponde a uma lista Lisp da forma

```
((MLIST) 1  2  7  ((MPLUS)  $X $Y ))
```

O sinalizador que denota o tipo campo de uma expressão Maxima é uma lista em si mesmo, após ter sido adicionado o simplificador a lista poderá transforma-se

```
((MLIST SIMP) 1 2 7 ((MPLUS SIMP) $X $Y))
```

39.2 Definições para Listas

append (*list_1, ..., list_n*) Função

Retorna uma lista simples dos elementos de *list_1* seguidos pelos elementos de *list_2*, ... `append` também trabalha sobre expressões gerais, e.g. `append (f(a,b), f(c,d,e));` retorna `f(a,b,c,d,e)`.

Faça `example(append);` para um exemplo.

assoc (*key, list, default*) Função

assoc (*key, list*) Função

Essa função procura pela chave *key* do lado esquerdo da entrada *list* que é da forma `[x,y,z,...]` onde cada elemento de *list* é uma expressão de um operando binário e 2 elementos. Por exemplo `x=1`, `2^3`, `[a,b]` etc. A chave *key* é verificada contra o primeiro operando. `assoc` retorna o segundo operando se `key` for achada. Se a chave `key` não for achada isso retorna o valor padrão *default*. *default* é opcional e o padrão é `false`.

atom (*expr*) Função

Retorna `true` se *expr* for atomica (i.e. um número, nome ou seqüência de caracteres) de outra forma retorna `false`. Desse modo `atom(5)` é `true` enquanto `atom(a[1])` e `atom(sin(x))` São `false` (assumindo `a[1]` e `x` não estão associados).

cons (*expr, list*) Função

Retorna uma nova lista construída do elemento *expr* como seu primeiro elemento, seguido por elementos de *list*. `cons` também trabalha sobre outras expressões, e.g. `cons(x, f(a,b,c));` -> `f(x,a,b,c)`.

copylist (*list*) Função

Retorna uma cópia da lista *list*.

create_list (*form, x_1, list_1, ..., x_n, list_n*) Função

Cria uma lista por avaliação de *form* com *x_1* associando a cada elemento *list_1*, e para cada tal associação anexa *x_2* para cada elemento de *list_2*, ... O número de elementos no resultado será o produto do número de elementos de cada lista. Cada variável *x_i* pode atualmente ser um símbolo –o qual não pode ser avaliado. A lista de argumentos será avaliada uma única vez no início do bloco de repetição.

```
(%i82) create_list1(x^i,i,[1,3,7]);
(%o82) [x,x^3,x^7]
```

Com um bloco de repetição duplo:

```
(%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]]
```

Em lugar de *list_i* dois argumentos podem ser fornecidos cada um dos quais será avaliado como um número. Esses podem vir a ser inclusive o limite inferior e superior do bloco de repetição.

```
(%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 que os limites ou lista para a variável *j* podem depender do valor corrente de *i*.

delete (*expr_1, expr_2*) Função
delete (*expr_1, expr_2, n*) Função

Remove todas as ocorrências de *expr_1* em *expr_2*. *expr_1* pode ser uma parcela de *expr_2* (se isso for uma adição) ou um fator de *expr_2* (se isso for um produto).

```
(%i1) delete(sin(x), x+sin(x)+y);
(%o1)                          y + x
```

`delete`(*expr_1*, *expr_2*, *n*) remove as primeiras *n* ocorrências de *expr_1* em *expr_2*. Se houver menos que *n* ocorrências de *expr_1* em *expr_2* então todas as corrências seram excluídas.

```
(%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*) Função

Retorna o oitavo ítem de uma expressão ou lista *expr*. Veja `first` para maiores detalhes.

endcons (*expr, list*) Função

Retorna uma nova lista consistindo de elementos de `list` seguidos por *expr*. `endcons` também trabalha sobre expressões gerais, e.g. `endcons(x, f(a,b,c));` -> `f(a,b,c,x)`.

fifth (*expr*) Função

Retorna o quinto ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

first (*expr*) Função

Retorna a primeira parte de *expr* que pode resultar no primeiro elemento de uma lista, a primeira linha de uma matriz, a primeira parcela de uma adição, etc. Note que `first` e suas funções relacionadas, `rest` e `last`, trabalham sobre a forma de *expr* que é mostrada não da forma que é digitada na entrada. Se a variável `inflag` é escolhida para `true` todavia, essa funções olharão na forma interna de *expr*. Note que o simplificador re-ordena expressões. Desse modo `first(x+y)` será `x` se `inflag` for `true` e `y` se `inflag` for `false` (`first(y+x)` fornece os mesmos resultados). As funções `second` .. `tenth` retornam da segunda até a décima parte do seu argumento.

fourth (*expr*) Função

Retorna o quarto ítem da expressõ ou lista *expr*. Veja `first` para maiores detalhes.

get (*a*, *i*) Função

Recupera a propriedade de usuário indicada por *i* associada com o átomo *a* ou retorna `false` se "a" não tem a propriedade *i*.

`get` avalia seus argumentos.

```
(%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]]
```

join (*l*, *m*) Function

Cria uma nova lista contendo os elementos das lista *l* and *m*, intercaladas. O resultado tem os elementos `[`*l*`[1], `*m*`[1], `*l*`[2], `*m*`[2], ...]`. As listas *l* e *m* podem conter qualquer tipo de elementos.

Se as listas forem de diferentes comprimentos, `join` ignora elementos da lista mais longa.

Maxima reclama se *L_1* ou *L_2* não for uma lista.

Exemplos:

```
(%i1) L1: [a, sin(b), c!, d - 1];
(%o1)                [a, sin(b), c!, d - 1]
(%i2) join (L1, [1, 2, 3, 4]);
(%o2)          [a, 1, sin(b), 2, c!, 3, d - 1, 4]
(%i3) join (L1, [aa, bb, cc, dd, ee, ff]);
(%o3)        [a, aa, sin(b), bb, c!, cc, d - 1, dd]
```

last (*expr*) Função

Retorna a última parte (parcela, linha, elemento, etc.) de *expr*.

length (*expr*) Função

Retorna (por padrão) o número de partes na forma externa (mostrada) de *expr*. Para listas isso é o número de elementos, para matrizes isso é o número de linhas, e para adições isso é o número de parcelas (veja `dispform`).

O comando `length` é afetado pelo comutador `inflag`. Então, e.g. `length(a/(b*c));` retorna 2 se `inflag` for `false` (Assumindo `exptdispflag` sendo `true`), mas 3 se `inflag` for `true` (A representação interna é essencialmente `a*b^-1*c^-1`).

listarith Variável de opção

Valor padrão: `true` - se `false` faz com que quaisquer operações aritméticas com listas sejam suprimidas; quando `true`, operações lista-matriz são contagiosas fazendo com que listas sejam convertidas para matrizes retornando um resultado que é sempre uma matriz. Todavia, operações lista-lista podem retornar listas.

listp (*expr*) Função

Retorna `true` se *expr* for uma lista de outra forma retorna `false`.

makelist (*expr, i, i_0, i_1*) Função
makelist (*expr, x, list*) Função

Constrói e retorna uma lista, cada elemento dessa lista é gerado usando *expr*.

`makelist (`*expr*`, `*i*`, `*i_0*`, `*i_1*`)` retorna uma lista, o `j`'ésimo elemento dessa lista é igual a `ev (`*expr*`, `*i*`=j)` para `j` variando de *i_0* até *i_1*.

`makelist (`*expr*`, `*x*`, `*list*`)` retorna uma lista, o `j`'ésimo elemento é igual a `ev (`*expr*`, `*x*`=`*list*`[j])` para `j` variando de 1 até `length (`*list*`)`.

Exemplos:

```
(%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*) Função

Retorna `true` se *expr* ocorre como um membro de *list* (não dentro de um membro). De outra forma `false` é retornado. `member` também trabalha sobre expressões não-lista, e.g. `member(b,f(a,b,c)); -> true`.

ninth (*expr*) Função
Retorna o nono ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

rest (*expr*, *n*) Função
rest (*expr*) Função
Retorna *expr* com seus primeiros *n* elementos removidos se *n* for positivo e seus últimos - *n* elementos removidos se *n* for negativo. Se *n* for 1 isso pode ser omitido. *expr* pode ser uma lista, matriz, ou outra expressão.

reverse (*list*) Função
Ordem reversa para os membros de *list* (não os membros em si mesmos). `reverse` também trabalha sobre expressões gerais, e.g. `reverse(a=b);` fornece `b=a`.

second (*expr*) Função
Retorna o segundo ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

seventh (*expr*) Função
Retorna o sétimo ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

sixth (*expr*) Função
Retorna o sexto ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

tenth (*expr*) Função
Retorna o décimo ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

third (*expr*) Função
Retorna o terceiro ítem da expressão ou lista *expr*. Veja `first` para maiores detalhes.

40 Conjuntos

40.1 Introdução a Conjuntos

Maxima fornece funções de conjunto, tais como intersecção e união , para conjuntos finitos que são definidos através de enumeração explícita. Maxima trata listas e conjuntos como objetos distintos. Esse recurso torna possível trabalhar com conjuntos que possuem elementos que são ou listas ou conjuntos.

Adicionalmente a funções para conjuntos finitos, Maxima fornece algumas funções relacionadas à combinatória; essas incluem números de Stirling, números de Bell, e muitas outras.

40.1.1 Uso

Para construir um conjunto como elementos `a_1, ..., a_n`, use o comando `set(a_1, ..., a_n)`; para construir o conjunto vazio, use `set()`. Se um elemento for listado mais que uma vez, o processo de simplificação elimina o número redundante.

```
(%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]}
```

Conjuntos são sempre mostrados na *saída* como uma lista delimitada por chaves; Se você quiser que seja possível *inserir* um conjunto usando chaves, veja [Definindo conjuntos com chaves], página 396.

Para construir um conjunto a partir dos elementos de uma lista, use `setify`.

```
(%i1) setify([b, a]);
(%o1)                         {a, b}
```

Membros de conjuntos `x` e `y` são iguais fazendo com que `is(x = y)` avalie para `true`. Dessa forma `rat(x)` e `x` são iguais com conjuntos membros; conseqüentemente,

```
(%i1) set(x, rat(x));
(%o1)                           {x}
```

Adicionalmente, uma vez que `is((x-1)*(x+1) = x^2 - 1)` avalia para `false`, `(x-1)*(x+1)` e `x^2-1` são distintos membros de conjuntos; dessa forma

```
(%i1) set((x - 1)*(x + 1), x^2 - 1);
                                          2
(%o1)               {(x - 1) (x + 1), x  - 1}
```

Para reduzir esses conjuntos a um único conjunto, aplique `rat` a cada membro do conjunto:

```
(%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}
```

Para remover redundncias de outros conjuntos, você pode precisar usar outras funções de simplificação. Aqui está um exemplo que usa `trigsimp`:

```
(%i1) set(1, cos(x)^2 + sin(x)^2);
                              2         2
(%o1)                 {1, sin (x) + cos (x)}
(%i2) map(trigsimp, %);
(%o2)                           {1}
```

Um conjunto é simplificado quando seus elementos não são redundantes e ordenados. a versão atual das funções de conjunto usam a função Maxima `orderlessp` para ordenar conjuntos; todavia, *versões futuras das funções de conjunto podem usar uma diferente função de ordenação.*

Algumas operações sobre conjuntos, tais como substituição de elementos, forçam automaticamente uma re-simplificação; por exemplo,

```
(%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 trata listas e conjuntos como objetos distintos; funções tais como `union` e `intersection` sinalizarão um erro se qualquer argumento for uma lista. Se você precisar aplicar uma função de conjunto a uma lista, use a função `setify` para coverter a lista para um conjunto. Dessa forma

```
(%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}
```

Para extrair todos os elementos de um conjunto `s` que atendem a uma condição `f`, use `subset(s,f)`. (Uma *condição* é uma função booleana que avalia para verdadeiro ou falso.) Por exemplo, para achar as equações em um dado conjunto que não depende de uma variável `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}
```

A seção ções para Conjuntos-snt [Definições para Conjuntos], página ções para Conjuntos-pg tem uma lista completa das funções de conjunto no Maxima.

40.1.2 Iteração entre Membros de Conjuntos

Existem dois caminhos para interagir sobre membros de conjuntos. Um caminho é usar `map`; por exemplo:

```
(%i1) map (f, set (a, b, c));
(%o1)                  {f(a), f(b), f(c)}
```

O outro caminho é usar `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
```

As funções Maxima `first` e `rest` trabalham corretamente sobre conjuntos. Aplicada a um conjunto, `first` retorna o primeiro elemento mostrado de um conjunto; elemento esse que pode ser dependente da implementação. Se `s` for um conjunto, então `rest(s)` é equivalente a `disjoin (first(s), s)`. Atualmente, existe outras funções Maxima que trabalham corretamente sobre conjuntos. Em futuras versões de funções de conjunto, `first` e `rest` podem funcionar diferentemente ou não funcionar em algumas situações.

40.1.3 Falhas

As funções de conjunto usam a função Maxima `orderlessp` para ordenar membros de conjuntos e a função (a nível de Lisp) `like` para testar a igualdade entre membros de conjuntos. Ambas (`orderlessp` e `like`) possuem falhas conhecidas (versões 5.9.2 e seguintes) que podem se manifestar se você tentar usar conjuntos com membros que são listas ou matrizes que contenham expressões na forma CRE (expressão racional canônica). Um exemplo é

```
(%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.
```

Esse comando faz com que Maxima feche com um erro (a mensagem de erro depende de qual versão de Lisp seu Maxima estiver utilizando). Outro exemplo é

```
(%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.
```

Essas falhas são causadas por erros em `orderlessp` e `like`; elas não são causadas através de falhas em funções de conjunto. Para ilustrar, tente os comandos

```
(%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
```

Até essas falhas serem corrigidas, não construa conjuntos com membros que sejam listas ou matrizes contendo expressões na forma de expressão racional canônica; um conjunto com um membro na forma de expressão racional canônica, todavia, pode não ser um problema:

```
(%i1) set (x, rat (x));
(%o1)                           {x}
```

A função `orderlessp` do Maxima tem outra falha que pode trazer problemas com funções de conjunto, a saber que o predicado de ordenação `orderlessp` não é transitivo. O exemplo mais simples conhecido que mostra isso é

```
(%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
```

Essa falha poderá causar problemas com todas as funções de conjunto bem como com funções Maxima em geral. \'E possivel, mas não certoe, que se todos membros de conjunto estiverem ou na forma de expressão racional canônica ou tiverem sido simplificados usando `ratsimp`, essa falha não se manifeste.

Os mecanismos `orderless` e `ordergreat` do Maxima são incompatíveis com funções de conjuntos. Se você precisar usar ou `orderless` ou `ordergreat`, descarregue esses comandos antes de construir quaisquer conjuntos e não use o comando `unorder`.

Você pode encontrar duas outras falhas menores. Na versão Maxima 5.5 e seguintes tinha uma falha na função `tex` que fazia o conjunto vazio ser incorretamente traduzido para TeX; essa falha foi corrigida no Maxima 5.9.0. Adicionalmente, a função `setup_autoload` no Maxima 5.9.0 não funciona adequadamente; uma correção está no arquivo `nset-init.lisp` localizado no diretório `maxima/share/contrib/nset`.

A função de sinal do Maxima tem uma falha que pode causar comportamento inadequado na função de Kronecker; por exemplo:

```
(%i1) kron_delta (1/sqrt(2), sqrt(2)/2);
(%o1)                            0
```

O valor correto é 1; a falha é relatada para `sign`

```
(%i1) sign (1/sqrt(2) - sqrt(2)/2);
(%o1)                          pos
```

Se você encontrar alguma coisa que você pense ser uma falha de função de conjunto, por favor emita um relatório para a base de dados de falhas do Maxima. Veja `bug_report`.

40.1.4 Definindo conjuntos com chaves

Se você quiser entrar conjuntos usando chaves, você pode fazer então através de declaração para que a chave esquerda seja um operador `matchfix`; isso é feito usando os comandos

```
(%i1) matchfix("{","}")$
(%i2) "{" ([a]) := apply (set, a)$
```

Agora podemos definir conjuntos usando chaves; dessa forma

```
(%i1) matchfix("{","}")$
(%i2) "{" ([a]) := apply (set, a)$
(%i3) {};
(%o3)                            {}
(%i4) {a, {a, b}};
(%o4)                       {a, {a, b}}
```

Para sempre permitir essa forma de entrada de conjunto, coloque os dois comandos nas linhas (%i1) e (%i2) em seu arquivo `maxima-init.mac`.

40.1.5 Combinatória e Funções diversas

Adicionalmente para funções de conjuntos finitos, Maxima fornece algumas funções relacionada a combinatória; essas incluem números de Stirling de primeiro e de segundo tipo, os número de Bell, coeficientes multinomiais, particionamento de inteiros não negativos, e umas poucas outras. Maxima também define uma função delta de Kronecker.

40.1.6 Autores

Stavros Macrakis de Cambridge, Massachusetts e Barton Willis da Universidade de Nebraska e Kearney (UNK) escreveram as funções de conjunto do Maxima e sua documentação.

40.2 Definições para Conjuntos

adjoin (*x*, *a*) Função

Anexa *x* ao conjunto *a* e retorna um conjunto. Dessa forma `adjoin(x, a)` e `union(set(x),a)` são equivalentes; todavia, usando `adjoin` pode ser algo mais rápido que usando `union`. Se *a* não for um conjunto, sinaliza um erro.

```
(%i1) adjoin (c, set (a, b));
(%o1)                         {a, b, c}
(%i2) adjoin (a, set (a, b));
(%o2)                          {a, b}
```

See also `disjoin`.

belln (*n*) Função

Para inteiros não negativos *n*, retorna o n-ésimo número de Bell. Se `s` for um conjunto com `n` membros, `belln(n)` é o número de partições de `s`. Por exemplo:

```
(%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
```

Quando *n* não for um inteiro não negativo, `belln(n)` não simplifica.

```
(%i1) [belln (x), belln (sqrt(3)), belln (-9)];
(%o1)        [belln(x), belln(sqrt(3)), belln(- 9)]
```

A função `belln` trabalha sobre igualdades, listas, matrizes, e conjuntos.

cardinality (*a*) Função

Retorna o números de elementos distintos do conjunto *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
```

Na linha (%o3), vemos que `cardinality` trabalha correetamente mesmo quandoa simplificação tiver sido desabilitada.

cartesian_product (*b_1, ... , b_n*) Função

Retorna um conjunto de listas da forma `[`*x_1*`, ..., `*x_n*`]`, onde *x_1* `está em` *b_1*, ..., *x_n* `in` *b_n*. Sinaliza um erro quando qualquer *b_k* não for um conjunto.

```
(%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*) Função

Remove *x* do conjunto *a* e retorna um conjunto. Se *x* não for um membro de *a*, retorna *a*. Cada um dos seguintes faz a mesma coisa: `disjoin(`*x*`, `*a*`)`, `delete(`*x*`, `*a*`)`, e `setdifference(`*a*`,set(`*x*`))`; todavia, `disjoin` é geralmente o caminho mais rápido para remover um membro de um conjunto. Sinaliza um erro se *a* não for um conjunto.

disjointp (*a*, *b*) Função

Retorna `true` se os conjuntos *a* e *b* forem disjuntos. sinaliza um erro se ou *a* ou *b* não for um conjunto.

divisors (*n*) Função

Quando *n* for um inteiro não nulo, retorna o conjunto de seus divisores. O conjunto de divisores inclue os membros 1 e *n*. Os divisores de um inteiro negativo são os divisores de seus valores absolutos.

Podemos verificar que 28 é um número perfeito.

```
(%i1) s: divisors(28);
(%o1)                {1, 2, 4, 7, 14, 28}
(%i2) lreduce ("+", args(s)) - 28;
(%o2)                          28
```

A função `divisors` trabalha através de simplificação; você pode não precisar reavaliar manualmente apóes uma substituição. Por exemplo:

```
(%i1) divisors (a);
(%o1)                       divisors(a)
(%i2) subst (8, a, %);
(%o2)                      {1, 2, 4, 8}
```

A função `divisors` trabalha sobre igualdades, listas, matrizes, e conjuntos. Aqui está um exemplo de trabalho sobre uma lista e uma igualdade.

```
(%i1) divisors ([a, b, c=d]);
(%o1) [divisors(a), divisors(b), divisors(c) = divisors(d)]
```

elementp (*x*, *a*) Função

Retorna `true` se e somente se *x* for um membro do conjunto *a*. Sinaliza um erro se *a* não for um conjunto.

emptyp (*a*) Função

Retorna `true` se e somente se *a* for um conjunto vazio ou a lista vazia.

```
(%i1) map (emptyp, [set (), []]);
(%o1)                       [true, true]
(%i2) map (emptyp, [a + b, set (set ()), %pi]);
(%o2)                  [false, false, false]
```

equiv_classes (*s*, *f*) Função

Retorna um conjunto de classes de equivalência de *s* com relação à relação de equivalência *f*. A função *f* pode ser uma função de valor booleano definida sobre o produto cartesiano de *s* com *s*. Adicionalmente, a função *f* pode ser uma relação de equivalência; `equiv_classes`, todavia, não verifica se *f* é uma relação de equivalência.

```
(%i1) equiv_classes (set (a, b, c), lambda ([x, y], is (x=y)));
(%o1)                    {{a}, {b}, {c}}
```

Atualmente, `equiv_classes` (*s*, *f*) aplica automaticamente a função do Maxima `is` após aplicar a função *f*; portanto, podemos reescrever o exemplo anterior mais resumidamente.

```
(%i1) equiv_classes (set (a, b, c), "=");
(%o1)                    {{a}, {b}, {c}}
```

Aqui está outro exemplo.

```
(%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*) Função
every (*f*, *L_1*, ..., *L_n*) Função

O primeiro argumento *f* pode ser um predicado (uma função que avalia para verdadeiro, falso, ou indeterminado).

Fornecendo um conjunto como segundo argumento, `every (f, a)` retorna `true` se `f(a_i)` retornar `true` para todos *a_i* em *a*. Uma vez que conjuntos são desordenados, `every` está livre para avaliar `f(a_i)` em qualquer ordem. `every` pode ou não avaliar *f* para todo *a_i* em *a*. Porque a ordem de avaliação não é especificada, o predicado *f* pode não ter efeito de lado ou erro de sinal para qualquer entrada.

Fornecendo uma ou mais listas como argumentos, `every (f, L_1, ..., L_n)` retorna `true` se `f(x_1, ..., x_n)` retornar `true` para todos *x_1*, ..., *x_n* em *L_1*, ..., *L_n*, respectivamente. `every` pode ou não avaliar *f* para toda conbinação *x_1*, ..., *x_n*. Uma vez que listas são ordenadas, `every` avalia na ordem de incremento do índice.

Para usar `every` sobre multiplos conjuntos argumentos, os conjuntos devem primeiro serem convertidos para uma seqüência ordenada de forma que seu alinhamento relativo comece bem definido.

Se o sinalizador global `maperror` for `true` (o padrão), todas as listas *L_1*, ..., *L_n* devem ter comprimentos iguais – de outra forma, `every` sinalizará um erro. Quando `maperror` for `false`, os argumentos lista são efetivamente truncados para o comprimento da lista mais curta.

A função Maxima `is` é automaticamente aplicada após a avaliação do predicado *f*.

```
(%i1) every ("=", [a, b], [a, b]);
(%o1)                         true
(%i2) every ("#", [a, b], [a, b]);
(%o2)                         false
```

extremal_subset (*s*, *f*, *max*) Função
extremal_subset (*s*, *f*, *min*) Função

Quando o terceiro argumento for `max`, retorna o subconjunto do conjunto ou a lista *s* para a qual a função real avaliada *f* toma sobre seu maior valor; quando o terceiro argumento for `min`, retorna o subconjunto para o qual *f* toma sobre seu menor valor.

```
(%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*) Função

`flatten` essencialmente avalia uma expressão como se seu principal operador tivesse sido declarado n-ário; existe, todavia, uma diferença – `flatten` não não age recursivamente dentro de outros argumentos de função. Por exemplo:

```
(%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)))
```

Aplicada a um conjunto, `flatten` reune todos os membros de elementos de conjuntos que são conjuntos; por exemplo:

```
(%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` trabalha corretamente quando o operador principal for uma função subscrita

```
(%i1) flatten (f[5] (f[5] (x)));
(%o1)                             f (x)
                                   5
```

Para aplicar `flatten` a uma expressão, o principal operador deve ser definido para zero ou mais argumentos; se esse não for o caso, Maxima sairá com um erro. Expressões com representações especiais, por exemplo expressões racionais canônicas, não podem ser tratadas por `flatten`; nesse caso, `flatten` retorna seu argumento inalterado.

full_listify (*a*) Função

Se *a* for um conjunto, converte *a* para uma lista e aplica `full_listify` para cada elemento lista.

Para converter apenas o operador de nível mais alto de um conjunto para uma lista, veja [listify], página 403.

fullsetify (*a*) Função

Se *a* for uma lista, converte *a* para um conjunto e aplica `fullsetify` para cada membro do conjunto.

```
(%i1) fullsetify ([a, [a]]);
(%o1)                          {a, {a}}
(%i2) fullsetify ([a, f([b])]);
(%o2)                         {a, f([b])}
```

Na linha (%o2), o argumento de `f` não é convertido para um conjunto porque o principal operador de `f([b])` não é uma lista.

Para converter apenas o operador de nível mais alto para um conjunto, veja [setify], página 407.

identity (*x*) Função

A função identidade avalia para seu argumento em todas as entradas. Para determinar se todo membro de um conjunto é `true`, você pode usar

```
(%i1) every (identity, [true, true]);
(%o1)                           true
```

integer_partitions (*n*) Função
integer_partitions (*n*, *len*) Função

Se o segundo argumento opcional *len* não for especificado, retorna o conjunto de todas as partições do inteiro *n*. Quando *len* for especificado, retorna todas as partições de comprimento *len* ou menor; nesse caso, zeros são anexados a cada partição de comprimento menor que *len* termos para fazer com que cada partição tenha exatamente *len* termos. No outro caso, cada partição é uma lista ordenada do maior para o menor.

Dizemos que uma lista $[a_1, ..., a_m]$ é uma partição de um inteiro não negativo n fazendo com que (1) cada a_i é um inteiro não nulo e (2) $a_1 + ... + a_m = n$. Dessa forma 0 não tem partições.

```
(%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]}
```

Para achar todas as partições que satisfazem uma condição, use a função `subset`; aqui está um exemplo que encontra todas as partições de 10 que consistem em números primos.

```
(%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]}
```

(Note que `primep(1)` é verdadeiro em Maxima. Isso discorda de muitas definições de número primo.)

intersect (*a_1*, ..., *a_n*) Função

Retorna um conjunto contendo os elementos que são comuns aos conjuntos de *a_1* até *a_n*. A função `intersect` deve receber um ou mais argumentos. Sinaliza um erro se qualquer dos *a_1* até *a_n* não for um conjunto. Veja também [intersection], página 402.

intersection (*a_1*, ..., *a_n*) Função

Retorna um conjunto contendo os elementos que são comuns aos conjuntos de *a_1* até *a_n*. A função `intersection` must receive one or more arguments. Sinaliza um erro se quaisquer dos *a_1* até *a_n* não for um conjunto. Veja também [intersect], página 402.

kron_delta (*x*, *y*) Função

A função delta de Kronecker; `kron_delta` (*x*, *y*) simplifica para 1 quando `is(x = y)` for verdadeiro e para zero quando `sign` (`|`*x* `-` *y*`|`) for `pos`. Quando `sign` (`|`*x* `-`

y`|)` for zero e *x* `-` *y* não for um número em ponto flutuante (nem do tipo `double` e nem do tipo `bfloat`), retorna 0. De outra forma, retorna uma forma substantiva.

A função, `kron_delta` é declarada para ser simétrica; dessa forma, por exemplo, `kron_delta(x, y) - kron_delta(y, x)` simplifica para zero.

Aqui alguns exemplos.

```
(%i1) [kron_delta (a, a), kron_delta (a + 1, a)];
(%o1)                          [1, 0]
(%i2) kron_delta (a, b);
(%o2)                    kron_delta(a, b)
```

Assumindo que `a > b` faz `sign (|a - b|)` avaliar para `pos`; dessa forma

```
(%i1) assume (a > b)$
(%i2) kron_delta (a, b);
(%o2)                              0
```

Se de outra maneira assumirmos que `x >= y`, então `sign (|x - y|)` avalia para `pz`; nesse caso, `kron_delta (x, y)` não simplifica

```
(%i1) assume(x >= y)$
(%i2) kron_delta (x, y);
(%o2)                    kron_delta(x, y)
```

Finalmente, uma vez que `1/10 - 0.1` avalia para um número em ponto flutuante, teremos

```
(%i1) kron_delta (1/10, 0.1);
                                    1
(%o1)                   kron_delta(--, 0.1)
                                    10
```

Se você quiser que `kron_delta (1/10, 0.1)` avalie para 1, aplique `float`.

```
(%i1) float (kron_delta (1/10, 0.1));
(%o1)                              1
```

listify (*a*) Função

Se *a* for um conjunto, retorna uma lista contendo os membros de *a*; quando *a* não for um conjunto, retorna *a*. Para converter um conjunto e todos os seus membros para listas, veja [full_listify], página 401.

lreduce (*f*, *s*) Função
lreduce (*f*, *s*, *init*) Função

A função `lreduce` (reduzir à esquerda) extende a função 2-aridade para uma n-aridade através de composição; um exemplo pode tornar isso mais claro. Quando o argumento opcional não for definido, teremos

```
(%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)
```

Note que a função *f* é primeiro aplicada aos `leftmost` (mais à esquerda) elementos da lista (dessa forma o nome lreduce). Quando *init* for definido, o segundo argumento para a avaliação de função mais interna é *init*; por exemplo:

```
(%i1) lreduce (f, [1, 2, 3], 4);
(%o1)                  f(f(f(4, 1), 2), 3)
```

A função `lreduce` torna isso fácil para encontrar o produto ou adição dos elementos de uma lista.

```
(%i1) lreduce ("+", args (set (a, b)));
(%o1)                          b + a
(%i2) lreduce ("*", args (set (1, 2, 3, 4, 5)));
(%o2)                           120
```

Veja também Veja [rreduce], página 406, Veja [xreduce], página 410, e Veja [tree_reduce], página 410.

makeset (*e*, *v*, *s*) Função

Essa função é similar a `makelist`, mas `makeset` permite substituições multiplas. o primeiro argumento *e* é uma espressão; o segundo argumento *v* é uma lista de variáveis; e *s* é uma lista ou conjunto de valores para as variáveis *v*. Cada membro de *s* deve ter o mesmo comprimento que *v*. Temos `makeset (e, v, s)` é o conjunto `{z | z = substitute(v -> s_i) e s_i em 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*) Função

A função de Moebius; quando *n* for produto de *k* primos distintos, `moebius(n)` avalia para $(-1)^k$; isso avalia para 1 quando $n = 1$; e isso avalia para 0 para todos os outros inteiros positivos. A função de Moebius trabalha sobre igualdades, listas, matrizes, e conjuntos.

multinomial_coeff (*a_1*, ..., *a_n*) Função
multinomial_coeff () Função

Retorna o coeficiente multinomial. Quando cada *a_k* for um inteiro não negativo, o coeficiente multinomial fornece o número de caminhos de substituição *a_1* + ... + *a_n* objetos distintos dentro de *n* caixas com *a_k* elementos na *k*'ésima caixa. Em geral, `multinomial (a_1, ..., a_n)` avalia para `(a_1 + ... + a_n)!/(a_1! ... a_n!)`. Sem nenhum argumento, `multinomial()` avalia para 1. É possível usar `minfactorial` para simplificar o valor retornado por `multinomial_coeff`; por exemplo:

```
(%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*) Função
num_distinct_partitions (*n*, *a*) Função

Quando *n* for um inteiro não negativo, retorna o número de partições inteiras distintas de *n*.

Se o parmetro opcional *a* tiver o valor `list`, retorna uma lista do número de partições distintas de 1,2,3, ... , n. Se *n* não for um inteiro não negativo, retorna uma forma substantiva.

Definição: Se $n = k_1 + ... + k_m$, onde k_1 até k_m são distintos inteiros positivos, chamamos $k_1 + ... + k_m$ uma partição distinta de *n*.

num_partitions (*n*) Função
num_partitions (*n*, *a*) Função

Quando *n* for um intero não negativo, retorna o número de partições de *n*. Se o parmetro opcional *a* tem o valor `list`, retorna uma lista do número de partições de 1,2,3, ... , n. Se *n* não for um inteiro não negativo, retorna uma forma substantiva.

```
(%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)
```

Para um inteiro não negativo *n*, `num_partitions` (*n*) é igual a `cardinality (integer_partitions` (*n*)); todavia, chamando `num_partitions` torna-se mais rápido.

partition_set (*a*, *f*) Função

Retorna uma lista de dois conjuntos; o primeiro conjunto é o subconjunto de *a* para o qual o predicado *f* avalia pra falso e o segundo é o subconjunto de *a* *a* para o qual *f* avalia para verdadeiro. Se *a* não for um conjunto, sinaliza um erro. Veja também [subset], página 409.

```
(%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*) Função

Retorna um conjunto de todas as permutações distintas *distintas* dos membros da lista ou conjunto *a*. (Cada permutacão é uma lista, não um conjunto.) Quando *a* for uma lista, membros duplicados de *a não* são apagados antes de encontradas todas as pemutações. Dessa forma

```
(%i1) permutations ([a, a]);
(%o1)                       {[a, a]}
(%i2) permutations ([a, a, b]);
(%o2)           {[a, a, b], [a, b, a], [b, a, a]}
```

Se *a* não for uma lista ou conjunto, sinaliza um erro.

powerset (*a*) Função

powerset (*a*, *n*) Função

Quando o segundo argumento opcional *n* não for definido, retorna o conjunto de todos os subconjuntos de conjunto *a*. `powerset(a)` tem `2^cardinality(a)` membros. Fornecido um segundo argumento, `powerset(a,n)` retorna o conjunto de todos os subconjunto de *a* que possuem cardinalidade *n*. Sinaliza um erro se *a* não for um conjunto; adicionalmente sinaliza um erro se *n* não for um inteiro positivo.

rreduce (*f*, *s*) Função

rreduce (*f*, *s*, *init*) Função

A função `rreduce` (right reduce) extende a 2-aridade da função pra uma n-aridade por composição; um exemplo pode tornar isso claro. Quando o argumento opcional *init* não for definido, temos

```
(%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)))
```

Note que a função *f* é primeiro aplicada aos elementos da lista mais à direita (dessa forma o nome rreduce). Quando *init* for definido, o segundo argumento para a avaliação da função mais interna é *init*; por exemplo:

```
(%i1) rreduce (f, [1, 2, 3], 4);
(%o1)                  f(1, f(2, f(3, 4)))
```

A função `rreduce` torna isso fácil para encontrar o produto ou adição de elementos de uma lista.

```
(%i1) rreduce ("+", args (set (a, b)));
(%o1)                          b + a
(%i2) rreduce ("*", args (set (1, 2, 3, 4, 5)));
(%o2)                           120
```

Veja também Veja [lreduce], página 403, Veja [tree_reduce], página 410, e Veja [xreduce], página 410.

setdifference (*a*, *b*) Função

Retorna um conjunto contendo os elementos no conjunto *a* que não estão no conjunto *b*. Sinaliza um erro se *a* ou *b* não for um conjunto.

setify (*a*) Função

Constrói um conjunto a partir dos elementos da lista *a*. Elementos duplicados da lista *a* são apagados e os elementos são organizados conforme o predicado `orderlessp`. Sinaliza um erro se `a` não for uma lista.

setp (*a*) Função

Retorna verdadeiro se e somente se *a* for um conjunto Maxima. A função `setp` verifica se o operador de seu argumento é conjunto; a função `setp` não verifica se seu argumento é um conjunto *simplificado*. Dessa forma

```
(%i1) setp (set (a, a)), simp: false;
(%o1)                          true
```

A função `setp` pode ser codificada no Maxima como `setp(a) := is (inpart (a, 0) = set)`.

set_partitions (*a*) Função
set_partitions (*a*, *n*) Função

Quando o argumento opcional *n* for definido, retorna um conjunto de todas as decomposições de *a* dentro de *n não vazios* subconjuntos disjuntos. Quando *n* não for definido, retorna o conjunto de todas as partições.

Dizemos que um conjunto P é uma partição de um conjunto S dado quando

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.

O conjunto vazio é uma partição de si mesmo (as condições 1 e 2 sendo vacuosamente verdadeiras); dessa forma

```
(%i1) set_partitions (set ());
(%o1)                          {{}}
```

A cardinalidade do conjunto de partições de um conjunto pode ser encontrada usando `stirling2`; dessa forma

```
(%i1) s: set (0, 1, 2, 3, 4, 5)$
(%i2) p: set_partitions (s, 3)$
(%o3)                        90 = 90
(%i4) cardinality(p) = stirling2 (6, 3);
```

Cada membro de `p` pode ter 3 membros; Vamos verificar.

```
(%i1) s: set (0, 1, 2, 3, 4, 5)$
(%i2) p: set_partitions (s, 3)$
(%o3)                           {3}
(%i4) map (cardinality, p);
```

Finalmente, para cada membro de `p`, a união de seus membros será igual a `s`; novamente vamos verificar.

```
(%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*) Função
some (*f*, *L_1*, ..., *L_n*) Função

O primeiro argumento de *f* pode ser um predicado (uma função que avalia para verdadeiro, falso ou indeterminado).

Fornecendo um conjunto como o segundo argumento, `some (f, a)` retorna `true` se `f(a_i)` retornar `true` para ao menos um *a_i* em *a*. Uma vez que conjuntos são não ordenados, `some` está livre para avaliar `f(a_i)` em qualquer ordem. `some` pode ou não avaliar *f* para todos *a_i* em *a*. Porque a ordem de avaliação não é especificada, o predicado *f* poderá não ter efeitos de lado ou erros de sinal para qualquer entrada. Para usar `some` sobre multiplos argumentos de conjunto, eles devem primeiro ser convertidos para uma seqüência ordenada de forma que seu alinhamento relativo torne-se bem definido.

Fornecendo uma ou mais listas como argumentos, `some (f, L_1, ..., L_n)` retorna `true` se `f(x_1, ..., x_n)` retornar `true` para ao menos um *x_1*, ..., *x_n* in *L_1*, ..., *L_n*, respectivamente. `some` pode ou não avaliar *f* para toda combinação *x_1*, ..., *x_n*. Uma vez que listar são ordenadas, `some` avalia na ordem de incremento dos índices.

Se o sinalizador global `maperror` for `true` (o padrão), todas as listas *L_1*, ..., *L_n* devem ter igual comprimento – de outra forma, `some` sinaliza um erro. Quando `maperror` for falso, os argumentos lista são efetivamente truncados cada um para ter o comprimento da menor lista.

A função Maxima `is` é automaticamente aplicada após o predicado *f*.

```
(%i1) some ("<", [a, b, 5], [1, 2, 8]);
(%o1)                          true
(%i2) some ("=", [2, 3], [2, 7]);
(%o2)                          true
```

stirling1 (*n*, *m*) Função

O número de Stirling de primeiro tipo. Quando *n* e *m* forem inteiros não negativos, a magnitude (módulo) de `stirling1 (n, m)` é o número de permutações de um conjunto com *n* membros que possuem *m* ciclos. Para detalhes, veja Graham, Knuth e Patashnik *Concrete Mathematics*. Usamos uma relação recursiva para definir `stirling1 (n, m)` para *m* menor que 0; nós não extendemos isso para *n* menor que 0 ou para argumentos não inteiros.

A função `stirling1` trabalha através de simplificação; A função `stirling1` conhece os valores especiais básicos (veja Donald Knuth, *The Art of Computer Programming,* terceira edição, Volume 1, Seção 1.2.6, Equações 48, 49, e 50). Para Maxima aplicar essas regras, os argumentos devem ser declarados para serem inteiros e o primeiro argumento deve ser não negativo. Por exemplo:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling1 (n, n);
(%o3)                           1
```

`stirling1` não simplifica para argumentos não inteiros.

```
(%i1) stirling1 (sqrt(2), sqrt(2));
(%o1)              stirling1(sqrt(2), sqrt(2))
```

Maxima conhece uns poucos outros valores especiais; por exemplo:

```
(%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*) Função

O número de Stirling de segundo tipo. Quando *n* e *m* são inteiros não negativos, `stirling2 (`*n*`, `*m*`)` é o número de possibilidades que um conjunto com cardinalidade *n* pode ser particionado em *m* subconjuntos disjuntos. Usamos uma relação recursiva para definir `stirling2 (`*n*`, `*m*`)` para *m* menor que 0; não extendemos isso para *n* menor que 0 ou para argumentos não inteiros.

A função `stirling2` trabalha através de simplificação; A função `stirling2` conhece os valores especiais básicos (veja Donald Knuth, *The Art of Computer Programming,* terceira edição, Volume 1, Seção 1.2.6, Equações 48, 49, e 50). Para Maxima aplicar essas regras, os argumentos devem ser declarados para serem inteiros e o primeiro argumento deve ser não negativo. Por exemplo:

```
(%i1) declare (n, integer)$
(%i2) assume (n >= 0)$
(%i3) stirling2 (n, n);
(%o3)                            1
```

`stirling2` não simplifica para argumentos não inteiros.

```
(%i1) stirling2 (%pi, %pi);
(%o1)                   stirling2(%pi, %pi)
```

Maxima conhece uns poucos outros valores especiais.

```
(%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*) Função

Retorna um subconjunto do conjunto *a* que satisfaz o predicado *f*. Por exemplo:

```
(%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}
```

O segundo argumento para `subset` deve ser um predicado (uma função que avalia para valores booleanos de um argumento) se o primeiro argumento para `subset` não for um conjunto, sinaliza com um erro. Veja também [partition_set], página 405.

subsetp (*a*, *b*) Função

Retorna verdadeiro se e somente se o conjunto *a* for um subconjunto de *b*. Sinaliza um erro se *a* ou *b* não for um conjunto.

symmdifference (*a_1*, ..., *a_n*) Função

Retorna o conjunto dos membros que ocorrem em exatamente um conjunto *a_k*. Sinaliza um erro se qualquer argumento *a_k* não for um conjunto. Fornecidos dois argumentos, `symmdifference (a, b)` é o mesmo que `union (setdifference (a, b), setdifference (b, a))`.

tree_reduce (*f*, *s*) Função
tree_reduce (*f*, *s*, *init*) Função

A função `tree_reduce` extende um operador binário associativo $f : SxS- > S$ de dois argumentos para qualquer número de argumentos usando uma árvore de comprimento mínimo. Um exemplo pode tornar isso claro.

```
(%i1) tree_reduce (f, [a, b, c, d]);
(%o1)                  f(f(a, b), f(c, d))
```

Fornecido um número ímpar de argumentos, `tree_reduce` favorece o lado esquerdo da árvore; por exemplo:

```
(%i1) tree_reduce (f, [a, b, c, d, e]);
(%o1)               f(f(f(a, b), f(c, d)), e)
```

Para adição de número em ponto flutuantes, usando `tree_reduce` pode fornecer uma adição que tem a menor perda de algarismos significativos que usando ou `rreduce` ou `lreduce`.

union (*a_1*, ..., *a_n*) Função

Retorna a união dos conjuntos *a_1* até *a_n*. Quando `union` não recebe argumentos, retorna o conjunto vazio. Sinaliza um erro quando um ou mais argumentos para `union` não for um conjunto.

xreduce (*f*, *s*) Função
xreduce (*f*, *s*, *init*) Função

Essa função é similar a ambas `lreduce` e `rreduce` exceto que `xreduce` está livre para usar ou a associatividade à esquerda ou a associatividade à direita; em particular quando *f* for uma função associativa e Maxima tem um avaliador interno para isso, `xreduce` pode usar a função n-ária; essas funções n-árias incluem adição `+`, multiplicação `*`, `and`, `or`, `max`, `min`, e `append`. Para esses operadores, nós geralmente esperamos usar `xreduce` para ser mais rápida que usando ou `rreduce` ou `lreduce`. Quando *f* não for n-ária, `xreduce` usa associatividade à esquerda. Adição em ponto flutuante não é associativa; todavia, `xreduce` usa a adição n-ária do Maxima quando o conjunto ou lista *s* contiver números em ponto flutuante.

41 Definição de Função

41.1 Introdução a Definição de Função

41.2 Função

Para definir uma função no Maxima você usa o operador :=. E.g.

```
f(x) := sin(x)
```

define uma função `f`. Funções anônimas podem também serem criadas usando `lambda`. Por exemplo

```
lambda ([i, j], ...)
```

pode ser usada em lugar de `f` onde

```
f(i,j) := block ([], ...);
map (lambda ([i], i+1), l)
```

retornará uma lista com 1 adicionado a cada termo.

Você pode também definir uma função com um número variável de argumentos, teno um argumento final que é atribuído para uma lista de argumentos extras:

```
(%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]]
```

O lado direito de uma função é uma expressão. Desse modo Se você quer uma seqüência de expressões, você faz

```
f(x) := (expr1, expr2, ...., exprn);
```

e o valor de *exprn* é que é retornado pela função.

Se você deseja fazer um `return` de alguma expressão dentro da função então você deve usar `block` e `return`.

```
block ([], expr1, ..., if (a > 10) then return(a), ..., exprn)
```

é em si mesma uma expressão, e então poderá ocupar o lugar do lado direito de uma definição de função. Aqui pode acontecer que o retorno aconteça mais facilmente que no exemplo anterior a essa última expressão.

O primeiro `[]` no bloco, pode conter uma lista de variáveis e atribuições de variáveis, tais como `[a: 3, b, c: []]`, que farão com que as três variáveis `a`,`b`,e `c` não se refiram a seus valores globais, mas ao contrário tenham esses valores especiais enquanto o código estiver executando a parte dentro do bloco `block`, ou dentro da funções chamadas de dentro do bloco `block`. Isso é chamado associação *dynamic*, uma vez que as variáveis permanecem do início do bloco pelo tempo que ele existir. Uma vez que você retorna do `block`, ou descarta-o, os valores antigos (quaisquer que sejam) das variáveis serão restaurados. É certamente

uma boa idéia para proteger suas variáveis nesse caminho. Note que as atribuições em variáveis do bloco, são concluídas em paralelo. Isso significa, que se tiver usado `c: a` acima, o valor de `c` será o valor de `a` a partir do momento em que vocêntrou no bloco, mas antes `a` foi associado. Dessa forma fazendo alguma coisa como

```
block ([a: a], expr1, ...  a: a+3, ..., exprn)
```

protegerá o valor externo de `a` de ser alterado, mas impedirá você acessar o valor antigo. Dessa forma o lado direito de atribuições, é avaliado no contexto inserido, antes que qualquer avaliação ocorra. Usando apenas `block ([x], ...` faremos com que o `x` ter a si mesmo como valor, apenas como tivesse você entrar numa breve sessão **Maxima**.

Os atuais argumentos para uma função são tratados exatamente da mesma que as variáveis em um bloco. Dessa forma em

```
f(x) := (expr1, ..., exprn);
```

e

```
f(1);
```

teremos um contexto similar para avaliação de expressões como se tivéssemos concluído

```
block ([x: 1], expr1, ..., exprn)
```

Dentro de funções, quando o lado direito de uma definição, pode ser calculado em tempo de execução, isso é úti para usar `define` e possivelmente `buildq`.

41.3 Macros

buildq (*L, expr*) Função

Substitue variáveis nomeadas pela lista *L* dentro da expressão *expr*, paralelamente, sem avaliar *expr*. A expressão resultante é simplificada, mas não avaliada, após `buildq` realizar a substituição.

Os elementos de *L* são símbolos ou expressões de atribuição *símbolo*`:` *valor*, avaliadas paralelamente. Isto é, a associação de uma variável sobre o lado direito de uma atribuição é a associação daquela variável no contexto do qual `buildq` for chamada, não a associação daquela variável na lista *L* de variáveis. Se alguma variável em *L* não dada como uma atribuição explícita, sua associação em `buildq` é a mesma que no contexto no qual `buildq` for chamada.

Então as variáveis nomeadas em *L* são substituidas em *expr* paralelamente. Isto é, a substituição para cada variável é determinada antes que qualquer substituição seja feita, então a substituição para uma variável não tem efeito sobre qualquer outra.

Se qualquer variável *x* aparecer como `splice (x)` em *expr*, então *x* deve estar associada para uma lista, e a lista recebe uma aplicação da função `splice` (é interpolada) na *expr* em lugar de substituída.

Quaisquer variáveis em *expr* não aparecendo em *L* são levados no resultado tal como foram escritos, mesmo se elas tiverem associações no contexto do qual `buildq` tiver sido chamada.

Exemplos

`a` é explicitamente associada a `x`, enquanto `b` tem a mesma associação (nomeadamente 29) como no contexto chamado, e `c` é levada do começo ao fim da forma como foi

escrita. A expressão resultante não é avaliada até a avaliação explícita (com duplo apóstrofo - não com aspas - ’’%.

```
(%i1) (a: 17, b: 29, c: 1729)$
(%i2) buildq ([a: x, b], a + b + c);
(%o2)                      x + c + 29
(%i3) ’’%;
(%o3)                       x + 1758
```

`e` está associado a uma lista, a qual aparece também como tal nos argumentos de `foo`, e interpolada nos argumentos de `bar`.

```
(%i1) buildq ([e: [a, b, c]], foo (x, e, y));
(%o1)                 foo(x, [a, b, c], y)
(%i2) buildq ([e: [a, b, c]], bar (x, splice (e), y));
(%o2)                  bar(x, a, b, c, y)
```

O resultado é simplificado após substituição. Se a simplificação for aplicada antes da substituição, esses dois resultados podem ser iguais.

```
(%i1) buildq ([e: [a, b, c]], splice (e) + splice (e));
(%o1)                     2 c + 2 b + 2 a
(%i2) buildq ([e: [a, b, c]], 2 * splice (e));
(%o2)                       2 a b c
```

As variáveis em L são associadas em paralelo; se associadas seqüêncialmente, o primeiro resultado pode ser `foo (b, b)`. Substituições são realizadas em paralelo; compare o segundo resultado com o resultado de `subst`, que realiza substituições seqüêncialmente.

```
(%i1) buildq ([a: b, b: a], foo (a, b));
(%o1)                       foo(b, a)
(%i2) buildq ([u: v, v: w, w: x, x: y, y: z, z: u], bar (u, v, w, x, y, z));
(%o2)                 bar(v, w, x, y, z, u)
(%i3) subst ([u=v, v=w, w=x, x=y, y=z, z=u], bar (u, v, w, x, y, z));
(%o3)                 bar(u, u, u, u, u, u)
```

Constrói uma lista de euqções com algumas variáveis ou expressões sobre o lado esquerdo e seus valores sobre o lado direito. `macroexpand` mostra a expressão retornada por `show_values`.

```
(%i1) show_values ([L]) ::= buildq ([L], map ("=", ’L, L));
(%o1)   show_values([L]) ::= buildq([L], map("=", ’L, L))
(%i2) (a: 17, b: 29, c: 1729)$
(%i3) show_values (a, b, c - a - b);
(%o3)              [a = 17, b = 29, c = 1729]
```

macroexpand (*expr*) Função

Retorna a expansão da macro de *expr* sem avaliar a expressão, quando `expr` for uma chamada de função de macro. De outra forma, `macroexpand` retorna *expr*.

Se a expansão de *expr* retorna outra chamada de função de macro, aquela chamada de função de macro é também expandida.

`macroexpand` coloca apóstrofo em seus argumentos, isto é, não os avalia. Todavia, se a expansão de uma chamada de função de macro tiver algum efeito, esse efeito colateral é executado.

Veja também `::=`, `macros`, e `macroexpand1`.

Exemplos

```
(%i1) g (x) ::= x / 99;
                                      x
(%o1)                       g(x) ::= --
                                      99
(%i2) h (x) ::= buildq ([x], g (x - a));
(%o2)            h(x) ::= buildq([x], g(x - a))
(%i3) a: 1234;
(%o3)                           1234
(%i4) macroexpand (h (y));
                                y - a
(%o4)                           -----
                                 99
(%i5) h (y);
                               y - 1234
(%o5)                          --------
                                  99
```

macroexpand1 (*expr*) Função

Retorna a expansão de macro de *expr* sem avaliar a expressão, quando `expr` for uma chamada de função de macro. De outra forma, `macroexpand1` retorna *expr*.

`macroexpand1` não avalia seus argumentos. Todavia, se a expansão de uma chamada de função de macro tiver algum efeito, esse efeito colateral é executado.

Se a expansão de *expr* retornar outra chamada de função de macro, aquela chamada de função de macro não é expandida.

Veja também `::=`, `macros`, e `macroexpand`.

Examples

```
(%i1) g (x) ::= x / 99;
                                      x
(%o1)                       g(x) ::= --
                                      99
(%i2) h (x) ::= buildq ([x], g (x - a));
(%o2)            h(x) ::= buildq([x], g(x - a))
(%i3) a: 1234;
(%o3)                           1234
(%i4) macroexpand1 (h (y));
(%o4)                          g(y - a)
(%i5) h (y);
                               y - 1234
(%o5)                          --------
                                  99
```

macros Global variable

Default value: `[]`

`macros` é a lista de funções de macro definidas pelo usuário. O operador de definição de função de macro `::=` coloca uma nova função de macro nessa lista, e `kill`, `remove`, e `remfunction` removem funções de macro da lista.

Veja também `infolists`.

splice (*a*) Função

Une como se fosse um elo de ligação (interpola) a lista nomeada através do átomo *a* em uma expressão, mas somente se `splice` aparecer dentro de `buildq`; de outra forma, `splice` é tratada como uma função indefinida. Se aparecer dentro de `buildq` com *a* sozinho (sem `splice`), *a* é substituido (não interpolado) como uma lista no resultado. O argumento de `splice` pode somente ser um átomo; não pode ser uma lista lateral ou uma expressão que retorna uma lista.

Tipicamente `splice` fornece os argumentos para uma função ou operador. Para uma função `f`, a expressão `f (splice (`*a*`))` dentro de `buildq` expande para `f (`*a*`[1], `*a*`[2], `*a*`[3], ...)`. Para um operador `o`, a expressão `"o" (splice (`*a*`)` dentro de `buildq` expande para `"o" (`*a*`[1], `*a*`[2], `*a*`[3], ...)`, onde `o` pode ser qualquer tipo de operador (tipicamente um que toma multiplos argumentos). Note que o operador deve ser contido dentro de aspas duplas `"`.

Exemplos

```
(%i1) buildq ([x: [1, %pi, z - y]], foo (splice (x)) / length (x));
                       foo(1, %pi, z - y)
(%o1)                 ----------------------
                      length([1, %pi, z - y])
(%i2) buildq ([x: [1, %pi]], "/" (splice (x)));
                                1
(%o2)                           ---
                                %pi
(%i3) matchfix ("<>", "<>");
(%o3)                           <>
(%i4) buildq ([x: [1, %pi, z - y]], "<>" (splice (x)));
(%o4)                   <>1, %pi, z - y<>
```

41.4 Definições para Definição de Função

apply (*f*, [*x_1*, ..., *x_n*]) Função

Retorna o resultado da aplicação da função *f* para a lista de argumentos *x_1*, ..., *x_n*. *f* é o nome de uma função ou uma expressão lambda.

Isso é útil quando é desejado calcular os argumentos para uma função antes de aplicar aquela função. Por exemplo, se `l` é a lista `[1, 5, -10.2, 4, 3]`, então `apply (min, l)` resulta -10.2. `apply` é também útil quando chama funções que não possuem seus argumentos avaliados e é desejável fazer a avaliação deles. Por exemplo, se `filespec` é uma variável associada à lista `[test, case]` então `apply (closefile, filespec)` é equivalente a `closefile (test, case)`. Em geral o primeiro argumento para `apply` será processado por um ' (apóstrofo) para fazer isso avaliar para si mesmo. Uma vez que algumas variáveis atômicas possuem o mesmo nome que certas funções os valores

da variável poderão ser usados em lugar da função porque `apply` teve seu primeiro argumento avaliado assim como seu segundo.

block ([*v_1*, ..., *v_m*], *expr_1*, ..., *expr_n*) Função
block (*expr_1*, ..., *expr_n*) Função

`block` avalia *expr_1*, ..., *expr_n* em seqüência e retorna o valor da última expressão avaliada. A seqüência pode ser modificada pelas funções `go`, `throw`, e `return`. A última expressão é *expr_n* a menos que `return` ou uma expressão contendo `throw` seja avaliada. Algumas variáveis *v_1*, ..., *v_m* podem ser declaradas locais para o bloco; essas são distinguidas das variáveis globais dos mesmos nomes. Se variáveis não forem declaradas locais então a lista pode ser omitida. Dentro do bloco, qualquer variável que não *v_1*, ..., *v_m* é uma variável global.

`block` salva os valores correntes das variáveis *v_1*, ..., *v_m* (quaisquer valores) na hora da entrada para o bloco, então libera as variáveis dessa forma eles avaliam para si mesmos. As variáveis locais podem ser associadas a valores arbitrários dentro do bloco mas quando o bloco é encerrado o valores salvos são restaurados, e os valores atribuídos dentro do bloco são perdidos.

`block` pode aparecer dentro de outro `block`. Variáveis locais são estabelecidas cada vez que um novo `block` é avaliado. Variáveis locais parecem ser globais para quaisquer blocos fechados. Se uma variável é não local em um bloco, seu valor é o valor mais recentemente atribuído por um bloco fechado, quaisquer que sejam, de outra forma, seu valor é o valor da variável no ambiente global. Essa política pode coincidir com o entendimento usual de "escopo dinâmico".

Se isso for desejado para salvar e restaurar outras propriedades locais ao lado de `value`, por exemplo `array` (exceto para arrays completos), `function`, `dependencies`, `atvalue`, `matchdeclare`, `atomgrad`, `constant`, e `nonscalar` então a função `local` pode ser usada dentro do bloco com argumentos sendo o nome das variáveis.

O valor do bloco é o valor da última declaração ou o valor do argumento para a função `return` que pode ser usada para sair explicitamente do bloco. A função `go` pode ser usada para transferir o controle para a declaração do bloco que é identificada com o argumento para `go`. Para identificar uma declaração, coloca-se antes dela um argumento atômico como outra declaração no bloco. Por exemplo: `block ([x], x:1, loop, x: x+1, ..., go(loop), ...)`. O argumento para `go` deve ser o nome de um identificador que aparece dentro do bloco. Não se deve usar `go` para transferir para um identificador em um outro bloco a não ser esse que contém o `go`.

Blocos tipicamente aparecem do lado direito de uma definição de função mas podem ser usados em outros lugares também.

break (*expr_1*, ..., *expr_n*) Função

Avalia e imprime *expr_1*, ..., *expr_n* e então causa uma parada do Maxima nesse ponto e o usuário pode examinar e alterar seu ambiente. Nessa situação digite `exit;` para que o cálculo seja retomado.

catch (*expr_1*, ..., *expr_n*) Função

Avalia *expr_1*, ..., *expr_n* uma por uma; se qualquer avaliação levar a uma avaliação de uma expressão da forma `throw (arg)`, então o valor de `catch` é o valor de `throw`

`(arg)`, e expressões adicionais não são avaliadas. Esse "retorno não local" atravessa assim qualquer profundidade de aninhar para o mais próximo contendo `catch`. Se não existe nenhum `catch` contendo um `throw`, uma mensagem de erro é impressa.

Se a avaliação de argumentos não leva para a avaliação de qualquer `throw` então o valor de `catch` é o valor de *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
```

A função `g` retorna uma lista de `f` de cada elemento de `l` se `l` consiste somente de números não negativos; de outra forma, `g` "captura" o primeiro elemento negativo de `l` e "arremessa-o".

compfile (*filename, f_1, ..., f_n*) Função

Traduz funções Maxima *f_1*, ..., *f_n* para Lisp e escreve o código traduzido no arquivo *filename*.

As traduções Lisp não são avaliadas, nem é o arquivo de saída processado pelo compilador Lisp. `translate` cria e avalia traduções Lisp. `compile_file` traduz Maxima para Lisp, e então executa o compilador Lisp.

Veja também `translate`, `translate_file`, e `compile_file`.

compile (*f_1, ..., f_n*) Função
compile (*functions*) Função
compile (*all*) Função

Traduz funções Maxima *f_1*, ..., *f_n* para Lisp, avalia a tradução Lisp, e chama a função Lisp `COMPILE` sobre cada função traduzida. `compile` retorna uma lista de nomes de funções compiladas.

`compile (all)` ou `compile (functions)` compila todas as funções definidas pelo usuário.

`compile` não avalia seus argumentos; o operador aspas simples `''` faz com que ocorra avaliação.

define (*f*(*x_1, ..., x_n*), *expr*) Função

Define uma função chamada *f* com argumentos *x_1*, ..., *x_n* e corpo da função *expr*.

`define` não avalia seu primeiro argumento na maioria dos casos, e avalia seu segundo argumento a menos que explicitamente seja pedido o contrário. Todavia, se o primeiro argumento for uma expressão da forma `ev (`*expr*`)`, `funmake (`*expr*`)`, ou `arraymake (`*expr*`)`, o primeiro argumento será avaliado; isso permite para o nome da função seja calculado, também como o corpo.

`define` é similar ao operador de definição de função `:=`, mas quando `define` aparece dentro da função, a definição é criada usando o valor de `expr` em tempo de execução em lugar de em tempo de definição da função que a contém.

Todas as definições de função aparecem no mesmo nível de escopo e visibilidade; definindo uma função `f` dentro de outra função `g` não limita o escopo de `f` a `g`.

Exemplos:

```
(%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*) Função

Introduz uma variável global dentro do ambiente Maxima. `define_variable` é útil em pacotes escritos pelo usuário, que são muitas vezes traduzidos ou compilados.

`define_variable` realiza os seguintes passos:

1. `mode_declare (`*name*`, `*mode*`)` declara o modo de *name* para o tradutor. Veja `mode_declare` para uma lista dos modos possíveis.
2. Se a variável é não associada, *default_value* é atribuído para *name*.
3. `declare (`*name*`, special)` declara essa variável especial.
4. Associa *name* com uma função de teste para garantir que a *name* seja somente atribuído valores do modo declarado.

A propriedade `value_check` pode ser atribuída a qualquer variável que tenha sido definida via `define_variable` com um outro modo que não `any`. A propriedade

`value_check` é uma expressão lambda ou o nome de uma função de uma variável, que é chamada quando uma tentativa é feita para atribuir um valor a uma variável. O argumento da função `value_check` é o valor que será atribuído.

`define_variable` avalia `default_value`, e não avalia `name` e `mode`. `define_variable` retorna o valor corrente de `name`, que é `default_value` se `name` não tiver sido associada antes, e de outra forma isso é o valor prévio de `name`.

Exemplos:

`foo` é uma variável Booleana, com o valor inicial `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` é uma variável inteira, que deve ser um número primo.

```
(%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 é not prime.
#0: prime_test(y=1440)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) bar;
(%o6)                         1439
```

`baz_quux` é uma variável que não pode receber a atribuição de um valor. O modo `any_check` é como `any`, mas `any_check` habilita o mecanismo `value_check`, e `any` não habilita.

```
(%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*) Função

dispfun (*all*) Função

Mostra a definição de funções definidas pelo usuário *f_1*, ..., *f_n*. Cada argumento pode ser o nome de uma macro (definida com `::=`), uma função comum (definida com `:=` ou `define`), uma função array (definida com `:=` ou com `define`, mas contendo argumentos entre colchêtes `[ ]`), uma função subscrita, (definida com `:=` ou `define`, mas contendo alguns argumentos entre colchêtes e outros entre parêntesis `( )`) uma da família de funções subscritas selecionadas por um valor subscrito particular, ou uma função subscrita definida com uma constante subscrita.

`dispfun (all)` mostra todas as funções definidas pelo usuário como dadas pelas `functions`, `arrays`, e listas de `macros`, omitindo funções subscritas definidas com constantes subscritas.

`dispfun` cria um Rótulo de expressão intermediária (`%t1`, `%t2`, etc.) para cada função mostrada, e atribui a definição de função para o rótulo. Em contraste, `fundef` retorna a definição de função.

`dispfun` não avalia seus argumentos; O operador aspas simples `’’` faz com que ocorra avaliação. `dispfun` sempre retorna `done`.

Exemplos:

```
(%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 Variável de sistema

Valor padrão: `[]`

`functions` é uma lista de todas as funções Maxima definidas pelo usuário na sessão corrente. Um função definida pelo usuário é uma função construída por `define` or `:=`. Uma função pode ser definida pela linha de comando do Maxima de forma interativa com o usuário ou em um arquivo Maxima chamado por `load` ou `batch`. Funções Lisp, todavia, não são adicinadas à lista `functions`.

fundef (*f*) Função

Retorna a definição da função *f*.

O argumento pode ser o nome de uma macro (definida com `::=`), uma função comum (definida com `:=` ou `define`), uma função array (definida com `:=` ou `define`, mas contendo argumentos entre colchêtes `[ ]`), Uma função subscrita, (definida com `:=` ou `define`, mas contendo alguns argumentos entre colchêtes e parêntesis `( )`) uma da família de funções subscritas selecionada por um valor particular subscrito, ou uma função subscrita definida com uma constante subscrita.

`fundef` não avalia seu argumento; o operador aspas simples `''` faz com que ocorra avaliação.

`fundef (`*f*`)` retorna a definição de *f*. Em contraste, `dispfun (`*f*`)` cria um rótulo de expressão intermediária e atribui a definição para o rótulo.

funmake (*name*, [*arg_1*, ..., *arg_n*]) Função

Retorna uma expressão *name* `(`*arg_1*`, ..., `*arg_n*`)`. O valor de retorno é simplificado, mas não avaliado, então a função não é chamada.

`funmake` avalia seus argumentos.

Exemplos:

- `funmake` avalia seus argumentos, mas não o valor de retorno.

```
(%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 simplifica o valor de retorno de `funmake`.

```
(%i1) funmake (sin, [%pi/2]);
(%o1)                            1
```

lambda ([*x_1, ..., x_m*], *expr_1, ..., expr_n*) Função

Define e retorna uma expressão lambda (que é, uma função anônima) com argumentos *x_1*, ..., *x_m* e valor de retorno *expr_n*. Uma expressão lambda pode ser atribuída para uma variável e avaliada como uma função comum. Uma expressão lambda pode aparecer em contextos nos quais uma avaliação de função (mas não um nome de função) é esperado como resposta.

Quando a função é avaliada, variáveis locais não associadas *x_1*, ..., *x_m* são criadas. `lambda` pode aparecer dentro de `block` ou outra função `lambda`; variáveis locais são estabelecidas cada vez que outro `block` ou função `lambda` é avaliada. Variáveis locais parecem ser globais para qualquer coisa contendo `block` ou `lambda`. Se uma variável é não local, seu valor é o valor mais recentemente atribuído em alguma coisa contendo `block` ou `lambda`, qualquer que seja, de outra forma, seu valor é o valor da variável no ambiente global. Essa política pode coincidir com o entendimento usual de "escopo dinâmico".

Após variáveis locais serem estabelecidas, *expr_1* até *expr_n* são avaliadas novamente. a variável especial `%%`, representando o valor da expressão precedente, é reconhecida. `throw` e `catch` pode também aparecer na lista de expressões.

`return` não pode aparecer em uma expressão lambda a menos que contendo `block`, nesse caso `return` define o valor de retorno do bloco e não da expressão lambda, a menos que o bloco seja *expr_n*. Da mesma forma, `go` não pode aparecer em uma expressão lambda a menos que contendo `block`.

`lambda` não avalia seus argumentos; o operador aspas simples `''` faz com que ocorra avaliação.

Exemplos:

- A expressão lambda pode ser atribuída para uma variável e avaliada como uma função comum.

```
(%i1) f: lambda ([x], x^2);
                                        2
(%o1)                       lambda([x], x )
(%i2) f(a);
                                      2
(%o2)                                a
```

- Uma expressão lambda pode aparecer em contextos nos quais uma avaliação de função é esperada como resposta.

```
(%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 ]
```

- Variáveis argumento são variáveis locais. Outras variáveis aparecem para serem variáveis globais. Variáveis globais são avaliadas ao mesmo tempo em que a expressão lambda é avaliada, a menos que alguma avaliação especial seja forçada por alguns meios, tais como ''.

```
(%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
```

- Expressões lambda podem ser aninhadas. Variáveis locais dentro de outra expressão lambda parece ser global para a expressão interna a menos que mascarada por variáveis locais de mesmos nomes.

```
(%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
```

- Uma vez que `lambda` não avalia seus argumentos, a expressão lambda `i` abaixo não define uma função "multiplicação por `a`". Tanto uma função pode ser definida via `buildq`, como na expressão lambda `i2` abaixo.

```
(%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*) Função

Declara as variáveis *v_1*, ..., *v_n* para serem locais com relação a todas as propriedades na declaração na qual essa função é usada.

`local` não avalia seus argumentos. `local` retorna `done`.

`local` pode somente ser usada em `block`, no corpo de definições de função ou expressões `lambda`, ou na função `ev`, e somente uma ocorrêcia é permitida em cada.

`local` é independente de `context`.

macroexpansion Variável de opção

Valor padrão: `false`

`macroexpansion` controla recursos avançados que afetam a eficiência de macros. Escolhas possíveis:

- `false` – Macros expandem normalmente cada vez que são chamadas.
- `expand` – A primeira vez de uma chamada particular é avaliada, a expansão é lembrada internamente, dessa forma não tem como ser recalculada em chamadas subseqüênte rapidamente. A macro chama ainda chamadas `grind` e `display` normalmente. Todavia, memória extra é requerida para lembrar todas as expansões.
- `displace` – A primeira vez de uma chamada particular é avaliada, a expansão é substituída pela chamada. Isso requer levemente menos armazenagem que quando `macroexpansion` é escolhida para `expand` e é razoávelmente rápido, mas tem a desvantagem de a macro original ser lentamente lembrada e daí a expansão será vista se `display` ou `grind` for chamada. Veja a documentação para `translate` e `macros` para maiores detalhes.

mode_checkp Variável de opção

Valor padrão: `true`

Quando `mode_checkp` é `true`, `mode_declare` verifica os modos de associação de variáveis.

mode_check_errorp Variável de opção

Valor padrão: `false`

Quando `mode_check_errorp` é `true`, `mode_declare` chama a função "error".

mode_check_warnp Variável de opção

Valor padrão: `true`

Quando `mode_check_warnp` é `true`, modo "errors" são descritos.

mode_declare (*y_1, mode_1, ..., y_n, mode_n*) Função

`mode_declare` é usado para declarar os modos de variáveis e funções para subseqüênte tradução ou compilação das funções. `mode_declare` é tipicamente colocada no início de uma definição de função, no início de um script Maxima, ou executado através da linha de comando de forma interativa.

Os argumentos de `mode_declare` são pares consistindo de uma variável e o modo que é um de `boolean`, `fixnum`, `number`, `rational`, ou `float`. Cada variável pode também ser uma lista de variáveis todas as quais são declaradas para ter o mesmo modo.

Se uma variável é um array, e se todo elemento do array que é referenciado tiver um valor então `array (yi, complete, dim1, dim2, ...)` em lugar de

```
array(yi, dim1, dim2, ...)
```

deverá ser usado primeiro declarando as associações do array. Se todos os elementos do array estão no modo `fixnum` (`float`), use `fixnum` (`float`) em lugar de `complete`. Também se todo elemento do array está no mesmo modo, digamos `m`, então

```
mode_declare (completearray (yi), m))
```

deverá ser usado para uma tradução eficiente.

Código numéricos usando arrays podem rodar mais rápidamente se for decladado o tamanho esperado do array, como em:

```
mode_declare (completearray (a [10, 10]), float)
```

para um array numérico em ponto flutuante que é 10 x 10.

Pode-se declarar o modo do resultado de uma função usando `function (f_1, f_2, ...)` como um argumento; aqui `f_1`, `f_2`, ... são nomes de funções. Por exemplo a expressão,

```
mode_declare ([function (f_1, f_2, ...)], fixnum)
```

declara que os valores retornados por `f_1`, `f_2`, ... são inteiros palavra simples.

`modedeclare` é um sinônimo para `mode_declare`.

mode_identity (*arg_1, arg_2*) Função

Uma forma especial usada com `mode_declare` e `macros` para declarar, e.g., uma lista de listas de números em ponto flutuante ou outros objetos de dados. O primeiro argumento para `mode_identity` é um valor primitivo nome de modo como dado para `mode_declare` (i.e., um de `float`, `fixnum`, `number`, `list`, ou `any`), e o segundo argumento é uma expressão que é avaliada e retornada com o valor de `mode_identity`. Todavia, se o valor de retorno não é permitido pelo modo declarado no primeiro argumento, um erro ou alerta é sinalizado. Um ponto importante é que o modo da expressão como determinado pelo Maxima para o tradutor Lisp, será aquele dado como o primeiro argumento, independente de qualquer coisa que vá no segundo argumento. E.g., `x: 3.3; mode_identity (fixnum, x);` retorna um erro. `mode_identity (flonum, x)` returns 3.3 . Isso tem númerosas utilidades, e.g., se você soube que `first (l)` retornou um número então você pode escrever `mode_identity`

`(number, first (l))`. Todavia, um mais eficiente caminho para fazer isso é definir uma nova primitiva,

```
firstnumb (x) ::= buildq ([x], mode_identity (number, x));
```

e usar `firstnumb` toda vez que você pegar o primeiro de uma lista de números.

transcompile Variável de opção

Valor padrão: `true`

Quando `transcompile` é `true`, `translate` e `translate_file` geram declarações para fazer o código traduzido mais adequado para compilação.

`compfile` escolhe `transcompile: true` para a duração.

translate (*f_1, ..., f_n*) Função
translate (*functions*) Função
translate (*all*) Função

Traduz funções definidas pelo usuário *f_1*, ..., *f_n* da linguagem de Maxima para Lisp e avalia a tradução Lisp. Tipicamente as funções traduzidas executam mais rápido que as originais.

`translate (all)` ou `translate (functions)` traduz todas as funções definidas pelo usuário.

Funções a serem traduzidas incluir˜ao uma chamada para `mode_declare` no início quando possível com o objetivo de produzir um código mais eficiente. Por exemplo:

```
f (x_1, x_2, ...) := block ([v_1, v_2, ...],
    mode_declare (v_1, mode_1, v_2, mode_2, ...), ...)
```

quando *x_1*, *x_2*, ... são parâmetros para a função e *v_1*, *v_2*, ... são variáveis locais.

Os nomes de funções traduzidas são removidos da lista `functions` se `savedef` é `false` (veja abaixo) e são adicionados nas listas `props`.

Funções não poderão ser traduzidas a menos que elas sejam totalmente depuradas.

Expressões são assumidas simplificadas; se não forem, um código correto será gerado mas não será um código ótimo. Dessa forma, o usuário não poderá escolher o comutador `simp` para `false` o qual inibe simplificação de expressões a serem traduzidas.

O comutador `translate`, se `true`, causa tradução automatica de uma função de usuário para Lisp.

Note que funções traduzidas podem não executar identicamente para o caminho que elas faziam antes da tradução como certas incompatabilidades podem existir entre o Lisp e versões do Maxima. Principalmente, a função `rat` com mais de um argumento e a função `ratvars` não poderá ser usada se quaisquer variáveis são declaradas com `mode_declare` como sendo expressões rotacionais canônicas(CRE). Também a escolha `prederror: false` não traduzirá.

`savedef` - se `true` fará com que a versão Maxima de uma função usuário permaneça quando a função é traduzida com `translate`. Isso permite a que definição seja mostrada por `dispfun` e autoriza a função a ser editada.

`transrun` - se `false` fará com que a versão interpretada de todas as funções sejam executadas (desde que estejam ainda disponíveis) em lugar da versão traduzida.

O resultado retornado por `translate` é uma lista de nomes de funções traduzidas.

translate_file (*maxima_filename*) Função
translate_file (*maxima_filename*, *lisp_filename*) Função

Traduz um arquivo com código Maxima para um arquivo com código Lisp. `translate_file` retorna uma lista de três nomes de arquivo: O nome do arquivo Maxima, o nome do arquivo Lisp, e o nome do arquivo contendo informações adicionais sobre a tradução. `translate_file` avalia seus argumentos.

`translate_file ("foo.mac"); load("foo.LISP")` é o mesmo que `batch ("foo.mac")` exceto por certas restrições, o uso de `''` e `%`, por exemplo.

`translate_file (`*maxima_filename*`)` traduz um arquivo Maxima *maxima_filename* para um similarmente chamado arquivo Lisp. Por exemplo, `foo.mac` é traduzido em `foo.LISP`. O nome de arquivo Maxima pod incluir nome ou nomes de diretório(s), nesse caso o arquivo de saída Lisp é escrito para o mesmo diretório que a entrada Maxima.

`translate_file (`*maxima_filename*`,` *lisp_filename*`)` traduz um arquivo Maxima *maxima_filename* em um arquivo Lisp *lisp_filename*. `translate_file` ignora a extensão do nome do arquivo, se qualquer, de `lisp_filename`; a extensão do arquivo de saída Lisp é sempre `LISP`. O nome de arquivo Lisp pode incluir um nome ou nomes de diretórios), nesse caso o arquivo de saída Lisp é escrito para o diretório especificado.

`translate_file` também escreve um arquivo de mensagens de alerta do tradutor em vários graus de severidade. A extensão do nome de arquivo desse arquivo é `UNLISP`. Esse arquivo pode conter informação valiosa, apesar de possivelmente obscura, para rastrear erros no código traduzido. O arquivo `UNLISP` é sempre escrito para o mesmo diretório que a entrada Maxima.

`translate_file` emite código Lisp o qual faz com que algumas definições tenham efeito tão logo o código Lisp é compilado. Veja `compile_file` para mais sobre esse tópico.

Veja também `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 Variável de opção

Valor padrão: `true`

Quando `transrun` é `false` fará com que a versão interpretada de todas as funções sejam executadas (desde que estejam ainda disponíveis) em lugar de versão traduzidas.

tr_array_as_ref Variável de opção

Valor padrão: `true`

Se `translate_fast_arrays` for `false`, referências a arrays no Código Lisp emitidas por `translate_file` são afetadas por `tr_array_as_ref`. Quando `tr_array_as_ref` é `true`, nomes de arrays são avaliados, de outra forma nomes de arrays aparecem como símbolos literais no código traduzido.

`tr_array_as_ref` não terão efeito se `translate_fast_arrays` for `true`.

tr_bound_function_applyp Variável de opção

Valor padrão: `true`

Quando `tr_bound_function_applyp` for `true`, Maxima emite um alerta se uma associação de variável (tal como um argumento de função) é achada sendo usada como uma função. `tr_bound_function_applyp` não afeta o código gerado em tais casos.

Por exemplo, uma expressão tal como `g (f, x) := f (x+1)` irá disparar a mensagem de alerta.

tr_file_tty_messagesp Variável de opção

Valor padrão: `false`

Quando `tr_file_tty_messagesp` é `true`, messagens geradas por `translate_file` durante a tradução de um arquivo são mostradas sobre o console e inseridas dentro do arquivo UNLISP. Quando `false`, messagens sobre traduções de arquivos são somente inseridas dentro do arquivo UNLISP.

tr_float_can_branch_complex Variável de opção

Valor padrão: `true`

Diz ao tradutor Maxima-para-Lisp assumir que as funções `acos`, `asin`, `asec`, e `acsc` podem retornar resultados complexos.

O efeito ostensivo de `tr_float_can_branch_complex` é mostrado adiante. Todavia, parece que esse sinalizador não tem efeito sobre a saída do tradutor.

Quando isso for `true` então `acos(x)` será do modo `any` sempre que `x` for do modo `float` (como escolhido por `mode_declare`). Quando `false` então `acos(x)` será do modo `float` se e somente se `x` for do modo `float`.

tr_function_call_default Variável de opção

Valor padrão: `general`

`false` significa abandonando e chamando `meval`, `expr` significa que Lisp assume função de argumento fixado. `general`, o código padrão dado como sendo bom para `mexprs` e `mlexprs` mas não `macros`. `general` garante que associações de variável são corretas em códigos compilados. No modo `general`, quando traduzindo F(X), se F for uma variável associada, então isso assumirá que `apply (f, [x])` é significativo, e traduz como tal, com o alerta apropriado. Não é necessário desabilitar isso. Com as escolhas padrão, sem mensagens de alerta implica compatibilidade total do código traduzido e compilado com o interpretador Maxima.

tr_numer Variável de opção

Valor padrão: `false`

Quando `tr_numer` for `true` propriedades `numer` são usadas para átomos que possuem essa propriedade, e.g. `%pi`.

tr_optimize_max_loop Variável de opção

Valor padrão: 100

`tr_optimize_max_loop` é número máximo de vezes do passo de macro-expansão e otimização que o tradutor irá executar considerando uma forma. Isso é para capturar erros de expansão de macro, e propriedades de otimização não terminadas.

tr_semicompile Variável de opção

Valor padrão: `false`

Quando `tr_semicompile` for `true`, as formas de saída de `translate_file` e `compfile` serão macroexpandidas mas não compiladas em código de máquina pelo compilador Lisp.

tr_state_vars Variável de sistema

Valor padrão:

```
[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]
```

A lista de comutadores que afetam a forma de saída da tradução. Essa informação é útil para sistemas populares quando tentam depurar o tradutor. Comparando o produto traduzido para o qual pode ter sido produzido por um dado estado, isso é possível para rastrear erros.

tr_warnings_get () Função

Imprime uma lista de alertas que podem ter sido dadas pelo tradutor durante a tradução corrente.

tr_warn_bad_function_calls Variável de opção

Valor padrão: `true`

- Emite um alerta quando chamadas de função estão sendo feitas por um caminho que pode não ser correto devido a declarações impróprias que foram feitas em tempo de tradução.

tr_warn_fexpr Variável de opção

Valor padrão: `compfile`

- Emite um alerta se quaisquer FEXPRs forem encontradas. FEXPRs não poderão normalmente ser saída em código traduzido, todas as formas de programa especial legítimo são traduzidas.

tr_warn_meval Variável

Valor padrão: `compfile`

- Emite um alerta se a função `meval` recebe chamadas. Se `meval` é chamada isso indica problemas na tradução.

tr_warn_mode Variável

Valor padrão: `all`

- Emite um alerta quando a variáveis forem atribuídos valores inapropriados para seu modo.

tr_warn_undeclared Variável de opção

Valor padrão: `compile`

- Determina quando enviar alertas sobre variáveis não declaradas para o TTY.

tr_warn_undefined_variable Variável de opção

Valor padrão: `all`

- Emite um alerta quando variáveis globais indefinidas forem vistas.

tr_windy Variável de opção

Valor padrão: `true`

- Gera comentários "de grande ajuda" e dicas de programação.

compile_file (*filename*) Função
compile_file (*filename*, *compiled_filename*) Função
compile_file (*filename*, *compiled_filename*, *lisp_filename*) Função

Traduz o arquivo Maxima *filename* para Lisp, executa o compilador Lisp, e, se a tradução e a compilação obtiverem sucesso, chama o código compilado dentro do Maxima.

`compile_file` retorna uma lista dos nomes de quatro arquivos: o arquivo original do Maxima, o nome da tradução Lisp, uma arquivo de notas sobre a tradução, e o nome do arquivo que contém o código compilado. Se a compilação falhar, o quarto ítem é `false`.

Algumas declarações e definições passam a ter efeito tão logo o código Lisp seja compilado (sem que seja necessário chamar o código compilado). Isso inclui funções definidas com o operador `:=`, macros definidas com o operador `::=`, `alias`, `declare`, `define_variable`, `mode_declare`, e `infix`, `matchfix`, `nofix`, `postfix`, `prefix`, e `compfile`.

Atribuições e chamadas de função não serão avaliadas até que o código compilado seja carregado. Em particular, dentro do arquivo Maxima, atribuições para sinalizadores traduzidos (`tr_numer`, etc.) não têm efeito sobre a tradução.

filename pode não conter declarações `:lisp`.

`compile_file` avalia seus argumentos.

declare_translated (*f_1*, *f_2*, ...) Função

Quando traduzindo um arquivo do código Maxima para Lisp, é importante para o programa tradutor saber quais funções no arquivo são para serem chamadas como funções traduzidas ou compiladas, e quais outras são apenas funções Maxima ou indefinidas. Colocando essa declaração no topo do arquivo, faremos conhecido que embora um símbolo diga que não temos ainda um valor de função Lisp, teremos uma em tempo de chamada. `(MFUNCTION-CALL fn arg1 arg2 ...)` é gerado quando o tradutor n˜ao sabe que `fn` está sendo compilada para ser uma função Lisp.

42 Fluxo de Programa

42.1 Introdução a Fluxo de Programa

Maxima fornece um `do` para ciclos iterativos, também contruções mais primitivas tais como `go`.

42.2 Definições para Fluxo de Programa

backtrace () Função
backtrace (*n*) Função

Imprime a pilha de chamadas, que é, a lista de funções que foram chamadas pela função correntemente ativa.

`backtrace()` imprime toda a pilha de chamadas.

`backtrace (`*n*`)` imprime as *n* mais recentes chamadas a funções, incluindo a função correntemente ativa.

`backtrace` pode ser chamada por um script, uma função, ou a partir da linha de comando interativa (não somente em um contexto de depuração).

Exemplos:

- `backtrace()` imprime toda a pilha de chamadas.

```
(%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*`)` imprime as *n* mais recentes chamadas a funções, incluindo a função correntemente ativa.

```
(%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 Operador especial

A declaração `do` é usada para executar iteração. Devido à sua grande generalidade a declaração `do` será descrita em duas partes. Primeiro a forma usual será dada que é análoga à forma que é usada em muitas outras linguagens de programação (Fortran, Algol, PL/I, etc.); em segundo lugar os outros recursos serão mencionados.

Existem três variantes do operador especial `do` que diferem somente por suas condições de encerramento. São elas:

- `for` *Variável*`:` *valor_inicial* `step` *incremento* `thru` *limite* `do` *corpo*
- `for` *Variável*`:` *valor_inicial* `step` *incremento* `while` *condition* `do` *corpo*
- `for` *Variável*`:` *valor_inicial* `step` *incremento* `unless` *condition* `do` *corpo*

(Alternativamente, o `step` pode ser dado após a condição de encerramento ou limite.)

valor_inicial, *incremento*, *limite*, e *corpo* podem ser quaisquer expressões. Se o incremento for 1 então `"step 1"` pode ser omitido.

A execução da declaração `do` processa-se primeiro atribuindo o valor_inicial para a variável (daqui em diante chamada a variável de controle). Então: (1) Se a variável de controle excede o limite de uma especificação `thru`, ou se a condição de `unless` for `true`, ou se a condição de `while` for `false` então o `do` será encerrado. (2) O corpo é avaliado. (3) O incremento é adicionado à variável de controle. O processo de (1) a (3) é executado repetidamente até que a condição de encerramento seja satisfeita. Pode-se também dar muitas condições de encerramento e nesse caso o `do` termina quando qualquer delas for satisfeita.

Em geral o teste `thru` é satisfeito quando a variável de controle for maior que o limite se o incremento for não negativo, ou quando a variável de controle for menor que o limite se o incremento for negativo. O incremento e o limite podem ser expressões não numéricas enquanto essa desigualdade puder ser determinada. Todavia, a menos que o incremento seja sintaticamente negativo (e.g. for um número negativo) na hora em que a declaração `do` for iniciada, Maxima assume que o incremento e o limite serão positivos quando o `do` for executado. Se o limite e o incremento não forem positivos, então o `do` pode não terminar propriamente.

Note que o limite, incremento, e condição de encerramento são avaliados cada vez que ocorre um ciclo. Dessa forma se qualquer desses for responsável por muitos cálculos, e retornar um resultado que não muda durante todas as execuções do corpo, então é mais eficiente escolher uma variável para seu valor anterior para o `do` e usar essa variável na forma `do`.

O valor normalmente retornado por uma declaração `do` é o átomo `done`. Todavia, a função `return` pode ser usada dentro do corpo para sair da delcaração `do` prematuramente e dar a isso qualquer valor desejado. Note todavia que um `return` dentro de um `do` que ocorre em um `block` encerrará somente o `do` e não o `block`. Note também que a função `go` não pode ser usada para sair de dentro de um `do` dentro de um `block` que o envolve.

A variável de controle é sempre local para o `do` e dessa forma qualquer variável pode ser usada sem afetar o valor de uma variável com o mesmo nome fora da declaração `do`. A variável de controle é liberada após o encerramento da declaração `do`.

```
(%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 que a condição `while i <= 10` é equivalente a `unless i > 10` e também `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
```

que fornece 8 termos da série de Taylor para `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
```

Esse exemplo calcula a raíz quadrada negativa de 10 usando a iteração de Newton-Raphson um maximum de 10 vezes. Caso o critério de convergêcia não tenha sido encontrado o valor retornado pode ser `done`. Em lugar de sempre adicionar uma quantidade à variável de controle pode-se algumas vezes desejar alterar isso de alguma outra forma para cada iteração. Nesse caso pode-se usar `next` *expressão* em lugar de `step` *incremento*. Isso fará com que a variável de controle seja escolhida para o resultado da expressão de avaliação cada vez que o ciclo de repetição for executado.

```
(%i6) for count: 2 next 3*count thru 20 do display (count)$
                              count = 2
```

```
                                count = 6

                               count = 18
```

Como uma alternativa para `for` *Variável*`:` *valor* `...do...` a sintaxe `for` *Variável* `from` *valor* `...do...` pode ser usada. Isso permite o `from` *valor* ser colocado após o `step` ou proximo valor ou após a condição de encerramento. Se `from` *valor* for omitido então 1 é usado como o valor inicial.

Algumas vezes se pode estar interessado em executar uma iteração onde a variável de controle nunca seja usada. Isso é permissível para dar somente as condições de encerramento omitindo a inicialização e a informação de atualização como no exemplo seguinte para para calcular a raíz quadrada de 5 usando uma fraca suposição inicial.

```
(%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
```

Se isso for desejado pode-se sempre omitir as condições de encerramento inteiramente e apenas dar o corpo `do` *corpo* que continuará a ser avaliado indefinidamente. Nesse caso a função `return` será usada para encerrar a execução da declaração `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 que `return`, quando executado, faz com que o valor corrente de `x` seja retornado como o valor da declaração `do`. O `block` é encerrado e esse valor da declaração `do` é retornado como o valor do `block` porque o `do` é a última declaração do `block`.)

Uma outra forma de `do` é disponível no Maxima. A sintaxe é:

`for` *Variável* `in` *list* *end_tests* `do` *corpo*

Os elementos de *list* são quaisquer expressões que irão sucessivamente ser atribuídas para a variável a cada iteração do corpo. O teste opcional *end_tests* pode ser usado para encerrar a execução da declaração `do`; de outra forma o `do` terminará quando a lista for exaurida ou quando um `return` for executado no corpo. (De fato, a lista pode ser qualquer expressão não atômica, e partes sucessivas são usadas.)

```
(%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*) Função

Avalia *expr_1*, ..., *expr_n* uma por uma e retorna `[expr_n]` (uma lista) se nenhum erro ocorrer. Se um erro ocorrer na avaliação de qualquer argumento, `errcatch` evita que o erro se propague e retorna a lista vazia `[]` sem avaliar quaisquer mais argumentos.

`errcatch` é útil em arquivos `batch` onde se suspeita que um erro possa estar ocorrendo o `errcatch` terminará o `batch` se o erro não for detectado.

error (*expr_1, ..., expr_n*) Função
error Variável de sistema

Avalia e imprime *expr_1*, ..., *expr_n*, e então causa um retorno de erro para o nível mais alto do Maxima ou para o mais próximo contendo `errcatch`.

A variável `error` é escolhida para uma lista descrevendo o erro. O primeiro elemento de `error` é uma seqüência de caracteres de formato, que junta todas as seqüências de caracteres entre os argumentos *expr_1*, ..., *expr_n*, e os elementos restantes são os valores de quaisquer argumentos que não são seqüências de caracteres.

`errormsg()` formata e imprime `error`. Isso efetivamente reimprime a mais recente mensagem de erro.

errormsg () Função

Reimprime a mais recente mensagem de erro. A variável `error` recebe a mensagem, e `errormsg` formata e imprime essa mensagem.

for Operador especial

Usado em iterações. Veja `do` para uma descrição das facilidades de iteração do Maxima.

go (*tag*) Função

é usada dentro de um `block` para transferir o controle para a declaração do bloco que for identificada com o argumento para `go`. Para identificar uma declaração, coloque antes dessa declaração um argumento atômico como outra declaração no `block`. Por exemplo:

```
block ([x], x:1, loop, x+1, ..., go(loop), ...)
```

O argumento para `go` deve ser o nome de um identificardor aparecendo no mesmo `block`. Não se pode usar `go` para transferir para um identificador em um outro `block` que não seja o próprio contendo o `go`.

if Operador especial

A declaração `if` é usada para execução condicional. A sintaxe é:

```
if <condição> then <expr_1> else <expr_2>
```

O resultado de uma declaração `if` será *expr_1* se condição for `true` e *expr_2* de outra forma. *expr_1* e *expr_2* são quaisquer expressões Maxima (incluindo declarações `if` aninhadas), e *condição* é uma expressão que avalia para `true` ou `false` e é composto de operadores relacionais e lógicos que são os seguintes:

Operação	Símbolo	Tipo
menor que	<	infixo relacional
menor que ou igual a	<=	infixo relacional
igualdade (sintática)	=	infixo relacional
negação de =	#	infixo relacional
igualdade (valor)	equal	função relacional
negação de igualdade	notequal	função relacional
maior que ou igual a	>=	infixo relacional
maior que	>	infixo relacional
e	and	infixo lógico
ou	or	infixo lógico
não	not	prefixo lógico

map (*f*, *expr_1*, ..., *expr_n*) Função

Retorna uma expressão cujo operador principal é o mesmo que o das expressões *expr_1*, ..., *expr_n* mas cujas subpartes são os resultados da aplicação de *f* nas correspondentes subpartes das expressões. *f* é ainda o nome de uma função de *n* argumentos ou é uma forma `lambda` de *n* argumentos.

`maperror` - se `false` fará com que todas as funções mapeadas (1) parem quando elas terminarem retornando a menor expi se não forem todas as expi do mesmo comprimento e (2) aplique fn a [exp1, exp2,...] se expi não forem todas do mesmo tipo de objeto. Se `maperror` for `true` então uma mensagem de erro será dada nas duas instâncias acima.

Um dos usos dessa função é para mapear (`map`) uma função (e.g. `partfrac`) sobre cada termo de uma expressão muito larga onde isso comumente não poderia ser possível usar a função sobre a expressão inteira devido a uma exaustão de espaço da lista de armazenamento no decorrer da computação.

```
(%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*) Função

Retorna `true` se e somente se *expr* for tratada pelas rotinas de mapeamento como um átomo. "Mapatoms" são átomos, números (incluíndo números racioanais), e variáveis subscritas.

maperror Variável de opção

Valor padrão: `true`

Quando `maperror` é `false`, faz com que todas as funções mapeadas, por exemplo

```
map (f, expr_1, expr_2, ...))
```

(1) parem quando elas terminarem retornando a menor expi se não forem todas as expi do mesmo comprimento e (2) aplique `f` a `[expr_1, expr_2, ...]` se `expr_i` não forem todas do mesmo tipo de objeto.

Se `maperror` for `true` então uma ,mensagem de erro é mostrada nas duas instâncias acima.

maplist (*f, expr_1, ..., expr_n*) Função

Retorna uma lista de aplicações de *f* em todas as partes das expressões *expr_1*, ..., *expr_n*. *f* é o nome de uma função, ou uma expressão lambda.

`maplist` difere de `map (`*f*`, `*expr_1*`, ..., `*expr_n*`)` que retorna uma expressão com o mesmo operador principal que *expr_i* tem (exceto para simplificações e o caso onde `map` faz um `apply`).

prederror Variável de opção

Valor padrão: `true`

Quando `prederror` for `true`, uma mensagem de erro é mostrada sempre que o predicado de uma declaração `if` ou uma função `is` falha em avaliar ou para `true` ou para `false`.

Se `false`, `unknown` é retornado no lugar nesse caso. O modo `prederror: false` não é suportado no código traduzido; todavia, `maybe` é suportado no código traduzido.

Veja também `is` e `maybe`.

return (*valor*) Função

Pode ser usada para sair explicitamente de um bloco, levando seu argumento. Veja `block` para mais informação.

scanmap (*f, expr*) Função
scanmap (*f, expr, bottomup*) Função

Recursivamente aplica *f* a *expr*, de cima para baixo. Isso é muito útil quando uma fatoração completa é desejada, por exemplo:

```
(%i1) exp:(a^2+2*a+1)*y + x^2$
(%i2) scanmap(factor,exp);
                                      2      2
(%o2)                          (a + 1)  y + x
```

Note o caminho através do qual `scanmap` aplica a dada função `factor` para as subexpressões constituintes de *expr*; se outra forma de *expr* é apresentada para `scanmap`

então o resultado pode ser diferente. Dessa forma, `%o2` não é recuperada quando `scanmap` é aplicada para a forma expandida de exp:

```
(%i3) scanmap(factor,expand(exp));
                              2                    2
(%o3)                         a  y + 2 a y + y + x
```

Aqui está um outro exemplo do caminho no qual `scanmap` aplica recursivamente uma função dada para todas as subexpressões, incluindo expoentes:

```
(%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`) aplica *f* a *expr* de baixo para cima. E.g., para `f` indefinida,

```
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))
```

Nesse caso, você pega a mesma resposta em ambos os caminhos.

throw (*expr*) Função

Avalia *expr* e descarta o valor retornado para o mais recente `catch`. `throw` é usada com `catch` como um mecanismo de retorno não local.

outermap (*f*, *a_1*, ..., *a_n*) Função

Aplica a função *f* para cada um dos elementos do produto externo *a_1* vezes *a_2* ... vezes *a_n*.

f é para ser o nome de uma função de *n* argumentos ou uma expressão lambda de *n* argumentos. Os argumentos *a_1*, ..., *a_n* podem ser listas ou não listas. Argumentos listas podem ter diferentes comprimentos. Argumentos outros que não listas são tratados como listas de comprimento 1 para o propósito de construção do produto externo.

O resultado da aplicação de *f* para o produto externo é organizado como uma lista aninhada. A intensidade do aninhamento é igual ao número de argumentos listas (argumentos outros que não listas não contribuem com um nível de aninhamento). Uma lista de intensidade de aninhamento *k* tem o mesmo comprimento que o *k*'ésimo argumento da lista.

`outermap` avalia seus argumentos.

Veja também `map`, `maplist`, e `apply`.

Exemplos:

```
(%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 Depurando

43.1 Depurando o Código Fonte

Maxima tem um depurador interno de código fonte. O usuário pode escolher um ponto de parada em uma função, e então caminhar linha por linha a partir daí. A pilha de chamadas po ser examinada, juntamente com as variáveis associadas àquele nível.

O comando `:help` ou `:h` mostra a lista de comando de depuração. (Em geral, comandos podem ser abreviados se a abreviação for única. Se não for única, as alternativas podem ser listadas.) Dentro do depurador, o usuário pode também usar qualquer funções comuns do Maxima para examinar, definir, e manipular variáveis e expressões.

Um ponto de parada é escolhido através do comando `:br` na linha de comando do Maxima. Dentro do depurador, o usuário pode avançar uma linha de cada vez usando o comando `:n` ("next"). o comando `:bt` ("backtrace") mostra uma lista da pilha de frames. O comando `:r` ("resume") sai do depurador e continua com a execução. Esses comandos são demostrados no exemplo abaixo.

```
(%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 digitado aqui lista os frames
#0: foo(y=5)(foobar.mac line 1)
#1: bar(x=2,y=3)(foobar.mac line 9)

(dbm:1) :n                           <-- Aqui digite :n para avançar linha
(foobar.mac 2)
/tmp/foobar.mac:2::

(dbm:1) :n                           <-- Aqui digite :n para avançar linha
(foobar.mac 3)
/tmp/foobar.mac:3::

(dbm:1) u;                           <-- Investiga o valor de u
28

(dbm:1) u: 33;                       <-- Altera u para ser 33
33

(dbm:1) :r                           <-- Digite :r para retomar a computação
```

```
(%o2)                                1094
```

O arquivo `/tmp/foobar.mac` é o seguinte:

```
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);
```

USO DO DEPURADOR ATRAVÉS DO EMACS

Se o usuário estiver rodando o código sob o GNU emacs em uma janela shell (shel dbl), ou está rodando a versão de interface gráfica, `xmaxima`, então se ele para em um ponto de parada, ele verá sua posição corrente no arquivo fonte a qua será mostrada na outra metade da janela, ou em vermelho brilhante, ou com um pequeno seta apontando na direita da linha. Ele pode avançar uma linha por vez digitando M-n (Alt-n).

Sob Emacs você pode executar em um shell `dbl`, o qual requer o arquivo `dbl.el` no diretório elisp. Tenha certeza que instalou os arquivos elisp ou adicionou o diretório elisp do Macima ao seu caminho: e.g., adicione o seguinte ao seu arquivo '`.emacs`' ou ao seu arquivo `site-init.el`

```
(setq load-path (cons "/usr/share/maxima/5.9.1/emacs" load-path))
(autoload 'dbl "dbl")
```

então no emacs

```
M-x dbl
```

pode iniciar uma janela shell na qual você pode executar programas, por exemplo Maxima, gcl, gdb etc. Essa janela de shell também reconhece informações sobre depuração de código fonte, e mostra o código fonte em outra janela.

O usuário pode escolher um ponto de parada em certa linha do arquivo digitando `C-x space`. Isso encontra qual a função que o cursor está posicionado, e então mostra qual a linha daquela função que o cursor está habilitado. Se o cursor estiver habilitado, digamos, na linha 2 de `foo`, então isso irá inserir na outra janela o comando, "`:br foo 2`", para parar `foo` nessa segunda linha. Para ter isso habilitado, o usuário deve ter maxima-mode.el habilitado na janela na qual o arquivo `foobar.mac` estiver interagindo. Existe comandos adicional disponíveis naquela janela de arquivo, tais como avaliando a função dentro do Maxima, através da digitação de `Alt-Control-x`.

43.2 Comandos Palavra Chave

Comandos palavra chave são palavras chaves especiais que não são interpretadas como expressões do Maxima. Um comando palavra chave pode ser inserido na linha de comando do Maxima ou na linha de comando do depurador, embora não possa ser inserido na linha de comando de parada. Comandos palavra chave iniciam com um dois pontos Keyword

commands start with a colon, ':'. Por exemplo, para avaliar uma forma Lisp você pode digitar `:lisp` seguido pela forma a ser avaliada.

```
(%i1) :lisp (+ 2 3)
5
```

O número de argumentos tomados depende do comando em particular. Também, você não precisa digitar o comando completo, apenas o suficiente para ser único no meio das palavras chave de parada. Dessa forma `:br` será suficiente para `:break`.

Os comandos de palavra chave são listados abaixo.

`:break F n`
: Escolhe um ponto de parada em uma função `F` na linha `n` a partir do início da função. Se `F` for dado como uma seqüência de caracteres, então essa seqüência de caracteres é assumida referir-se a um arquivo, e `n` é o deslocamente a partir do início do arquivo. O deslocamento é opcional. Se for omitido, é assumido ser zero (primeira linha da função ou do arquivo).

`:bt` Imprime na tela uma lista da pilha de frames

`:continue`
: Continua a computao

`:delete` Remove o ponto de parada selecionado, ou todos se nenum for especificado

`:disable` Desabilita os pontos de parada selecionados, ou todos se nenhum for especificado

`:enable` Habilita os pontos de de parada especificados, ou todos se nenhum for especificado

`:frame n` Imprime na tela a pilha de frame `n`, ou o corrente frame se nenhum for especificado

`:help` Imprime na tela a ajuda sobre um comando do depurador, ou todos os comandos se nenhum for especificado

`:info` Imprime na tela informaes sobre um tem

`:lisp alguma-forma`
: Avalia `alguma-forma` como uma forma Lisp

`:lisp-quiet alguma-forma`
: Avalia a forma Lisp `alguma-forma` sem qualquer sada

`:next` Como `:step`, exceto `:next` passos sobre chamadas de fuo

`:quit` Sai do nvel corrente do depurador sem concluir a computao

`:resume` Continua a computao

`:step` Continua a computao at encontraruma nova linha de cdico

`:top` Retorne para a linha de comando do Maxima (saindo de qualquer nvel do depurador) sem completar a computao

43.3 Definições para Depuração

refcheck Varivel de opção

Valor padro: `false`

Quando `refcheck` for `true`, Maxima imprime uma mensagem cada vez que uma varivel associada for usada pela primeira vez em uma computao.

setcheck Varivel de opção

Valor padro: `false`

Se `setcheck` for escolhido para uma lista de variveis (as quais podem ser subscritas), Maxima mostra uma mensagem quando as variveis, ou ocorrncias subscritas delas, forem associadas com o operador comum de atribuio `:`, o operador `::` de atribuio, ou associando argumentos de funo, mas no com o operador de atribuio de funo `:=` nem o operador de atribuio `::=` de macro. A mensagem compreende o nome das variveis e o valor associado a ela.

`setcheck` pode ser escolhida para `all` ou `true` incluindo desse modo todas as variveis.

Cada nova atribuio de `setcheck` estabelece uma nova lista de variveis para verificar, e quaisquer variveis previamente atribudas a `setcheck` so esquecidas.

Os nomes atribudos a `setcheck` devem ter um apstrofo no incio se eles forem de outra forma avaliam para alguma outra coisa que no eles mesmo. Por exemplo, se `x`, `y`, e `z` estiverem atualmente associados, ento digite

```
setcheck: [’x, ’y, ’z]$
```

para coloc-los na lista de variveis monitoradas.

Nenhuma sada gerada quando uma varivel na lista `setcheck` for atribuda a s mesma, e.g., `X: ’X`.

setcheckbreak Varivel de opção

Valor padro: `false`

Quando `setcheckbreak` for `true`, Maxima mostrar um ponto de parada quando uma varivel sob a lista `setcheck` for atribuda a um novo valor. A parada ocorre antes que a atribuo seja concluda. Nesse ponto, `setval` retm o valor para o qual a varivel est para ser atribuda. Conseqentemente, se pode atribuir um valor diferente atravs da atribuio a `setval`.

Veja tambm `setcheck` e `setval`.

setval Varivel de sistema

Mantm o valor para o qual a varivel est para ser escolhida quando um `setcheckbreak` ocorrer. Conseqentemente, se pode atribuir um valor diferente atravs da atribuio a `setval`.

Veja tambm `setcheck` e `setcheckbreak`.

timer (*f_1*, ..., *f_n*) Função
timer () Função

Dadas as funes *f_1*, ..., *f_n*, `timer` coloca cada uma na lista de funes para as quais cronometragens estatsticas so coletadas. `timer(f)$ timer(g)$` coloca `f` e ento `g` sobre a lista; a lista acumula de uma chamada para a chamada seguinte.

Sem argumentos, `timer` retorna a lista das funes tempo estatisticamente monitoradas.

Maxima armazena quanto tempo empregado executando cada funo na lista de funes tempo estatisticamente monitoradas. `timer_info` retorna a coronometragem estatstica, incluindo o tempo mdio decorrido por chamada de funo, o nmero de chamadas, e o tempo total decorrido. `untimer` remove funes da lista de funes tempo estatisticamente monitoradas.

`timer` no avalia seus argumentos. `f(x) := x^2$ g:f$ timer(g)$` no coloca `f` na lista de funes estatisticamente monitoradas.

Se `trace(f)` est vigorando, ento `timer(f)` no tem efeito; `trace` e `timer` no podem ambas atuarem ao mesmo tempo.

Veja tambm `timer_devalue`.

untimer (*f_1*, ..., *f_n*) Função
untimer () Função

Dadas as funes *f_1*, ..., *f_n*, `untimer` remove cada uma das funes listadas da lista de funes estatisticamente monitoradas.

Sem argumentos, `untimer` remove todas as funes atualmente na lista de funes estatisticamente monitoradas.

Aps `untimer (f)` ser executada, `timer_info (f)` ainda retorna estatisticas de tempo previamente coletadas, embora `timer_info()` (sem argumentos) no retorna informaes sobre qualquer funo que no estiver atualmente na lista de funes tempo estatisticamente monitoradas. `timer (f)` reposiciona todas as estatisticas de tempo para zero e coloca `f` na lista de funes estatisticamente monitoradas novamente.

timer_devalue Varivel de opção

Valor Padro: `false`

Quando `timer_devalue` for `true`, Maxima subtrai de cada funo estatisticamente monitorada o tempo empregado em ou funes estatisticamente monitoradas. De outra forma, o tempo reportado para cada funo inclui o tempo empregado em outras funes. Note que tempo empregado em funes no estatisticamente monitoradas no subtrado do tempo total.

Veja tambm `timer` e `timer_info`.

timer_info (*f_1*, ..., *f_n*) Função
timer_info () Função

Dadas as funes *f_1*, ..., *f_n*, `timer_info` retorna uma matriz contendo informaes de cronometragem para cada funo. Sem argumentos, `timer_info` retorna informaes de cronometragem para todas as funes atualmente na lista de funes estatisticamente monitoradas.

A matriz retornada atravs de `timer_info` contm o nome da funo, tempo por chamda de funo, nmero de chamadas a funes,tempo total, e `gctime`, cujja forma "tempo de descarte" no Macsyma original mas agora sempre zero.

Os dados sobre os quais `timer_info` constri seu valor de retorno podem tambm serem obtidos atravs da funo `get`:

```
get(f, 'calls);  get(f, 'runtime);  get(f, 'gctime);
```

Veja tambm `timer`.

trace (*f_1, ..., f_n*) Função
trace () Função

Dadas as funes *f_1*, ..., *f_n*, `trace` instrui Maxima para mostrar informaes de depurao quando essas funes forem chamadas. `trace(f)$ trace(g)$` coloca `f` e ento `g` na lista de funes para serem colocadas sob a ao de `trace`; a lista acumula de uma chamada para a seguinte.

Sem argumentos, `trace` retorna uma lista de todas as funes atualmente sob a ao de `trace`.

A funo `untrace` desabilita a ao de `trace`. Veja tambm `trace_options`.

`trace` no avalia seus argumentos. Dessa forma, `f(x) := x^2$ g:f$ trace(g)$` no coloca `f` sobre a lista de funes monitoradas por `trace`.

Quando uma funo for redefinida, ela removida da lista de `timer`. Dessa forma aps `timer(f)$ f(x) := x^2$`, a funo `f` no mais est na lista de `timer`.

Se `timer (f)` estiver em efeito, ento `trace (f)` no est agindo; `trace` e `timer` no podem ambas estar agindo para a mesma funo.

trace_options (*f, option_1, ..., option_n*) Função
trace_options (*f*) Função

Escolhe as opes de `trace` para a funo *f*. Quaisquer opes anteriores so substitudas. `trace_options (f, ...)` no tem efeito a menos que `trace (f)` tenha sido tambm chamada (ou antes ou aps `trace_options`).

`trace_options (f)` reposiciona todas as opes para seus valores padro.

As opes de palavra chave so:

- `noprint` No mostre uma mensagem na entrada da funo e saia.
- `break` Coloque um ponto de parada antes da funo ser inserida, e aps a funos er retirada. Veja `break`.
- `lisp_print` Mostre argumentos e valores de retorno com objetos Lisp.
- `info` Mostre `-> true` na entrada da funo e saia.
- `errorcatch` Capture os erros, fornecendo a opo para sinalizar um erro, tentar novamente a chamada de funo, ou especificar um valor de retorno.

Opes para `trace` so especificadas em duas formas. A presena da palavra chave de opo sozinha coloca a opo para ter efeito incondicionalmente. (Note que opo *foo* no coloca para ter efeito especificando *foo*`: true` ou uma forma similar; note tambm que palavras chave no precisam estar com apstrofo.) Especificando a opo palavra chave com uma funo predicado torna a opo condicional sobre o predicado.

A lista de argumentos para a função predicado sempre `[level, direction, function, item]` onde `level` é o nível recursão para a função, `direction` é ou `enter` ou `exit`, `function` é o nome da função, e `item` é a lista de argumentos (sobre entrada) ou o valor de retorno (sobre a saída).

Aqui está um exemplo de opções incondicionais de `trace`:

```
(%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);
```

Aqui está a mesma função, com a opção `break` condicional sobre um predicado:

```
(%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*) Função

untrace () Função

Dadas as funções *f_1*, ..., *f_n*, `untrace` desabilita a a monitoração habilitada pela função `trace`. Sem argumentos, `untrace` desabilita a atuação da função `trade` para todas as funções.

`untrace` retorne uma lista das funções para as quais `untrace` desabilita a atuação de `trace`.

44 Índice de Função e Variável

Apêndice A Índice de Função e Variável

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Índice Resumido

Sumário