Московский Государственный Университет им. М.В.Ломоносова
физический факультет
кафедра квантовой теории и ФВЭ

В.А.Ильина, П.К.Силаев

Система аналитических вычислений

MAXIMA

для физиков-теоретиков

Москва 2007

1. Общие сведения о MAXIM'е

Полезность систем аналитических вычислений для физиков-теоретиков (по мнению авторов) не вызывает никаких сомнений. Во-первых, это способ получить правильный ответ при аналитических вычислениях, которые одновременно являются относительно простыми (в том смысле, что очевидна процедура вычислений) и исключительно громоздкими (в смысле длины формул). Вероятность получения правильной окончательной формулы при ручном счете в этом случае обыкновенно близка к нулю. Во-вторых, нередко их можно использовать как справочник. В них зашито довольно большое количество сведений о, скажем, высших трансцендентных функциях, и если Вы смутно помните общий вид некоторого соотношения, но не помните точный его вид, то нередко быстрее воспользоваться системой аналитических вычислений, чем листать соответствующий DjVu файл математического справочника. В-третьих, они позволяют быстренько что-либо прикинуть, провести простенькую выкладку, нарисовать график — нередко ручкой на бумаге то же самое делается медленнее.

С другой стороны, следует отчетливо понимать, что пользу эти системы могут принести только в том случае, когда Вы хорошо знаете их синтаксис и общие принципы их работы. Кроме того, Вы должны отчетливо представлять себе процедуру вычислений, которые хотите провести. Хотя все системы снабжены некоторым набором универсальных функций для "упрощения" и преобразования выражений, очень редко бывает так, что система заметит что-то такое, чего не заметили Вы.

MAXIMA — это одна из первых систем аналитических вычислений. К настоящему времени существует множество таких систем — это, прежде всего, язык аналитических вычислений REDUCE; системы аналитических вычислений MAXIMA, Mathematica и Maple/MathLab[(1)]; программа MathCad (которая скорее являются игрушкой, чем рабочим инструментом); вымершие к настоящему времени программы Derive и Eureca; системы, ориентированные на специальные задачи (например, на вычисления в физике высоких энергий) типа ScoonShip или FORM; и многие другие системы.

Имеет смысл различать язык аналитических вычислений, т.е. минимальный набор синтаксических конструкций, который расширяется библиотечными функциями (типичный пример — REDUCE), и систему аналитических вычислений, в которой изначально определено очень много функций, причем эти

(1) Система MathLab использует в качестве "двигателя" (т.е. вычислительного модуля) именно Maple

функции (как правило) избыточны. Система аналитических вычислений старается "уметь все" и на каждую задачу в ней найдется своя функция (типичные примеры — MAXIMA и Mathematica).
Кроме того, следует различать работу с системой аналитических вычислений в интерактивном и в пакетном режиме. В реальной работе чаще встречается пакетный режим. Вы отчетливо представляете себе последовательность выкладок, которую хотите провести, в текстовом редакторе набираете соответствующую программу, запускаете ее на счет и смотрите на результаты ее работы, содержащиеся в выходном файле. Интерактивный режим встречается тогда, когда Вы хотите что-либо прикинуть. Вы вводите одну команду, смотрите что получилось, в зависимости от этого вводите следующую команду и т.д.
При выборе системы аналитических вычислений следует учитывать:

Во-первых, насколько удачен замысел системы: естественность синтаксиса, простота реализации тех или иных действий, автоматическое или принудительное упрощение, полнота набора функций и т.п.

Во-вторых, насколько удачна ее конкретная реализация: так, старенький REDUCE 3.0 компактен, но крайне ограничен и по памяти и по скорости по сравнению с более "новым" REDUCE 3.4, а оформление REDUCE 5.0 заметно лучше, чем у всех предыдущих; Mathematica 1.0 — это программа, которая почти не работает, Mathematica 2.0 и 2.2 работают, но с "причудами", Mathematica 3.0 для LINUX (один из вариантов UNIX) работает в 3 раза быстрее, чем Mathematica 3.0 для Windows 95, если они инсталлированы на идентичных компьютерах, оформление Mathematica 5.0 гораздо лучше, чем у предыдущих версий и быстренько прикинуть что-либо в интерактивной моде удобнее всего именно на ней (Mathematica 5.1 почти не отличается от Mathematic'и 5.0), и т.д.

В-третьих, легальность ее использования: стоимость REDUCE 4.0 — (была) 99$, стоимость Mathematic'и 3.0 под Solaris (один из вариантов UNIX) — (была) 500$, стоимость Mathematic'и 5.1 — от 2000$ до 5000$, MAXIMA 5.13 под UNIX и WinXXX — программа общего пользования (более того, open source), ее исходники и уже откомпилированные версии для разных операционных систем легко найти с помощью любого интернетовского поисковика. Ее также легко найти в проекте GNU, но там она существует в режиме "not supported", что довольно печально.[(2)] В настоящее время ее можно скачать с

[(2)] Если Вы захотите сами скомпилировать ее под UNIX (вероятно, это будет

"`http://sourceforge.net`", но, как показывает опыт, это утверждение останется верным не слишком долго.

В-четвертых, привычкой пользователя. Этот фактор, как ни странно, на наш взгляд, является основным. С одной стороны, освоив любую из систем, Вы с легкостью овладеете и всеми остальными, но все равно самая первая система так и останется для Вас самой удобной, ее синтаксис будет казаться самым естественным, и Вы не будете "замечать его" при реальной научной работе.
Авторы имеют опыт реальной работы с REDUCE, MAXIMA и Mathematica, и опыт игры с Maple, MathLab и MathCad.
Среди перечисленных программ самым удобным сервисом (оформлением) обладает, на наш взгляд, Mathematica. Если Вам нужно что-либо быстренько прикинуть (проверить формулу, нарисовать график, оценить порядок величины) в интерактивном режиме, пожалуй, следует воспользоваться Mathematic'ой. (3)
Однако Вы должны помнить, что среди всех перечисленных систем Mathematica содержит наибольшее количество ошибок. Хотя основные ее идеи и синтаксические конструкции были позаимствованы у MAXIM'ы,(4) но их конкретная реализация оставляет желать лучшего. История взаимоотношений MAXIM'ы и Mathematic'и практически повторяет историю взаимоотношений UNIX и WinXXX. К сожалению, Mathematica изначально развивалась как коммерческая программа и была ориентирована на массового пользователя. Так что конструировалась система, которая "умеет все", но делает это "все" не слишком хорошо ("чтобы хоть как-нибудь работало"). Мало того, если в не слишком сложных случаях она обычно выдает правильные ответы, то в нетривиальных случаях она нередко (и без каких-либо предупреждений) выдает

Linux), имейте в виду, что последние версии MAXIMA не всегда хорошо понимают последние версии GNU Common Lisp (GCL), так что проще собирать ее на базе CLISP. В принципе в большинстве версий Linux обе версии Common Lisp (и GCL, и CLISP) входят в устанавливаемый комплект программ, но (как правило) не целиком, а в виде огрызков. Этих огрызков вполне достаточно, чтобы получить работающую в текстовом режиме MAXIM'у. Как правило, для реальной научной работы большего и не требуется, но если Вы склонны к программистским экспериментам (тревожный признак — может быть Вы не физик, а программист?), то поставьте CLISP в полном объеме.

(3) Авторы не настаивают на этом утверждении. Если Maple или MathLab Вам кажутся более удобными, используйте их.

(4) Значительно улучшен только аппарат подстановок по шаблону, который в MAXIM'е крайне неудобен.

нечто совершенно неправильное. Опять-таки, при усовершенствовании системы основное внимание уделялось оформлению, а не оптимизации работы. Реализация именно аналитических вычислений в Mathematic'е не очень удачна (медленно и не экономно по памяти), и при большом объеме выкладок они не могут быть выполнены за конечное время.

Несмотря на все сказанное, для работы в интерактивном режиме авторы все же рекомендуют Mathematic'у.

Напротив, для работы в пакетном режиме мы рекомендуем MAXIM'у.[5] Как нам кажется, она обладает большей гибкостью при работе с выражениями, чем REDUCE, хотя иногда реализация некоторого алгоритма вычислений на MAXIM'е требует больших усилий, чем на REDUCE. Тем не менее, при работе с MAXIM'ой можно достичь гораздо большего, чем на REDUCE. Дело не столько в том, что в MAXIM'е определено множество полезных функций, которых нет в REDUCE. Главным преимуществом MAXIM'ы при работе с аналитическими выражениями является то, что она не старается "упростить" выражение до некоторой канонической формы, если ее об этом не просят. Для очень многих выражений такое "упрощение" приводит к невероятному усложнению (удлинению). REDUCE, напротив, всегда сводит выражение к каноническому виду, и проводит упрощения до тех пор, пока выражение не перестает меняться. Единственное, на что может повлиять пользователь REDUCE — это поменять сам канонический вид с помощью флагов. Гибридная операция (например, факторизация отдельных слагаемых в сумме) в REDUCE требует очень больших программистских усилий. Напротив, MAXIMA делает в точности то, о чем ее просит пользователь с помощью тех или иных функций или флаговых переменных. При этом она не проводит упрощения или преобразования "до конца" (т.е. до тех пор, пока выражение не перестает меняться), так что часто повторный вызов той же самой функции меняет выражение (опыт показывает, что это большое преимущество, а не недостаток, как может показаться на первый взгляд).

Что касается интерфейсов MAXIM'ы, то их в настоящее время насчитывается четыре.

Во-первых, работа с командной строки ("пакетный режим"). В любом текстовом редакторе Вы набираете программу, запускаете ее на счет и потом (с

---

[5] Авторы не настаивают на этом утверждении. Если REDUCE Вам кажется более удобным, используйте его.

изумлением) рассматриваете выходной файл. Для этого достаточно уметь запускать MAXIM'у с командной строки. При работе под UNIX (вероятно, это будет Linux) легко найти скрипт (обычно он называется просто "`maxima`"), запускающий "голую" текстовую моду MAXIM'ы. По умолчанию она запустится в интерактивном режиме (Вы можете последовательно вводить команды и немедленно получать на них ответ). Но, пользуясь обычным перенаправлением ввода с помощью "`<`", можно сразу запустить на счет Ваш файл. Под WinXXX (старайтесь не пользоваться WinXXX!) все совершенно аналогично, только надо искать запускающий файл "`maxima.bat`". Как показывает опыт, при реальной научной работе этот режим самый эффективный, и авторы горячо рекомендуют именно его. Кроме того, этот режим выглядит (почти) одинаково на разных операционных системах, работает в текстовой моде, и, следовательно, может быть использован при работе с удаленного терминала.

Во-вторых, работа с помощью редактора `emacs`. Для эффективной работы в этом режиме надо освоить `emacs`, что полезно и само по себе, поскольку это исключительный по мощности редактор (это часть проекта GNU, так что он существует и под UNIX, и под WinXXX). Затем следует познакомить `emacs` с MAXIM'ой. Для этого достаточно скопировать содержимое директории "`emacs`", которая присутствует в установленной Вами MAXIM'е, в любое место, из которого `emacs` согласен читать файлы "`Emacs Lisp`" (они имеют имена с расширением "`.el`"). Для этого проще всего найти внутри самого `emacs` директорию, где лежат файлы с расширением "`.el`". Наконец, следует создать конфигурационный файл `emacs`, который должен иметь имя "`.emacs`" (начинается с точки!) и содержать следующие строки

```
(autoload 'maxima "maxima" "Maxima interaction" t)
(autoload 'maxima-mode "maxima" "Maxima mode" t)
(setq auto-mode-alist (cons '
        ("\\.mxm" .  maxima-mode) auto-mode-alist))
```

(это следует воспринимать аксиоматически). Под UNIX этот файл должен лежать в Вашей `home`-директории, а под WinXXX — в корне диска "`C:`"
После этого Вы можете набирать свои программы, пользуясь редактором `emacs`. Если расширение файла будет "`.mxm`", то `emacs` автоматически перейдет в режим "`maxima-mode`". При этом текст будет снабжен "боевой раскраской" — функции MAXIM'ы будут рисоваться одним цветом, переменные другим, комментарии — третьим.(6) Кроме того, Вы сможете немедленно посылать на исполнение как отдельные строки (`Ctrl-C Ctrl-C`) или маркированные блоки (`Ctrl-C Ctrl-R`, буква "R" из-за того, что в `emacs` блоки называются

(6) На взгляд авторов, это скорее недостаток, чем достоинство.

"`regions`"), так и весь набранный файл (`Ctrl-C Ctrl-B`, буква "B" из-за того, что в `emacs` файлы называются "`buffers`"). Окно редактора будете поделено пополам, и в одной половине Вы увидите исходный текст, а в другой — результаты работы MAXIM'ы. Кстати, результаты работы — это тоже "`buffer`", так что его тоже можно сохранить в файл.

В принципе на быстрой машине это довольно удобный способ работы. Более того, под UNIX `emacs` существует как в графической, так и в текстовой версии, так что возможна работа с удаленного терминала. Но следует помнить, что все архитектурные украшения замедляют работу. Кроме того, необходимо основательно изучить `emacs`.

В-третьих, это интерфейс "`xMaxima`". Это строго графический интерфейс. Окошко в нем поделено пополам, в верхней части живет интерактивная MAXIMA, а в нижней присутствует псевдобраузер. В этом псевдобраузере могут быть определены ссылки, которые рассматриваются как скрипты. Их текст пересылается MAXIM'е, а результат работы можно отправить обратно в псевдобраузер. Забавно, что текст в псевдобраузере можно редактировать. Этот интерфейс реализован как под UNIX, так и под WinXXX. Как интерактивный help по MAXIM'е это выглядит неплохо (немедленная иллюстрация работы, причем исходную команду можно поменять и посмотреть, что при этом выйдет). Однако (на наш взгляд!) для собственно научной работы это не очень удобно.

В-четвертых, это интерфейс "`wxMaxima`". Это строго графический интерфейс, выполненный в стиле WinXXX, что уже настораживает. В нем реализованы подсказки, упрощенный ввод команд (с помощью кнопок) и прочий сервис, который обыкновенно сильно замедляет работу. (Любой человек, набиравший *настоящие* формулы в TeX и WinWord, знает, насколько просто и быстро набирать команды TeX в любом текстовом редакторе и насколько мучительное занятие пользоваться "сервисом" с кнопочками в WinWord. Это уже не говоря о переносимости файлов хотя бы с машины на машину, даже в рамках одной операционной системы. И не говоря о непредсказуемом поведении, если где-то затесался невидимый символ форматирования. И не говоря о качестве итоговых формул — на формулы WinWord обыкновенно без слез смотреть невозможно).

На наш взгляд, это (пока) неудачное подражание оформлению Mathematic'и. Сервис получился не слишком удобный. Так что для собственно научной работы этот интерфейс (пока) не очень подходит.

Следует подчеркнуть, что в этом руководстве описаны отнюдь не все возможности MAXIM'ы и далеко не все определенные в ней функции. Разумеется, критерии отбора были в значительной мере субъективными. На наш взгляд, мы описали практически все синтаксические конструкции, позволяющие контролировать исполнение программ, практически все универсальные функции для работы с аналитическими выражениями и большинство общеупотребительных прикладных функций, ориентированных на частные случаи.

Из изложения сознательно исключено описание "низкоуровневых" возможностей MAXIM'ы (взаимодействие с LISP'ом, и т.п.). Опыт показывает, что написание подпрограмм на LISP'е, которые затем загружаются в MAXIM'у, позволяет значительно ускорить процесс счета и расширить возможности MAXIM'ы (совершенно так же, как программирование на ассемблере расширяет возможности языка "C"). Однако, на наш взгляд, физик-теоретик не должен программировать ни на ассемблере, ни на LISP'е. Ну а если читатель умеет программировать на LISP'е, он (несомненно) сможет освоить "низкоуровневые" возможности MAXIM'ы самостоятельно.

Кроме того, из изложения исключено описание рисовательных функций MAXIM'ы. Эти (довольно разнообразные) функции ориентированы на работу в интерактивном режиме, а (как нам кажется) MAXIM'у лучше использовать в пакетом режиме.

Кроме того, из изложения исключено описание тензорных пакетов MAXIM'ы. Таких пакетов существует два с половиною, и (на первый взгляд) они могли бы быть чрезвычайно полезны для физиков-теоретиков, имеющих дело с гравитацией. К сожалению, определенные там универсальные функции великолепно работают только в "игрушечных" тестовых задачах. При попытке использовать их в реальной научной работе оказывается, что совершенно необходимо самостоятельно, с учетом особенностей Вашей конкретной задачи, писать процедуру вычисления даже таких несложных конструкций, как символы Кристоффеля.

В любом случае пополнить недостающие сведения легко с помощью описания MAXIM'ы (MAXIMA manual) и с помощью функции "`describe`", которая дает краткое (хотя и не всегда достаточно понятное) описание команд и функций MAXIM'ы. При работе в интерактивном режиме Вы можете ввести в командной строке команду "`describe(describe);`". Ответ MAXIM'ы будет довольно поучительным.

2. Первоначальные сведения о работе с MAXIM'ой

Идентификаторы в MAXIM'е составляются из 26×2 латинских букв (**теперь** она **различает** строчные и прописные буквы), 10 цифр, символа подчеркивания "_", процента "%". Как правило с "%" начинаются специальные имена, например "%i" — это мнимая единица, "%pi" — это π, а "%e" — основание натурального логарифма.

При реальной работе MAXIMA дублирует ввод и печатает его вперемежку с выводом. Для удобства чтения мы будем в примерах выделять вывод другим шрифтом и сдвигать его вправо. Это позволяет сделать наши примеры не слишком похожими на то, что Вы увидите, непосредственно работая с MAXIM'ой, но зато они будут более читабельными.

Ввод в MAXIM'е завершается одним из двух терминаторов — ";" или "$". В первом случае результат вычислений печатается, во втором — нет. Каждый ввод нумеруется с помощью меток "label" — "%i1", "%i2", "%i3", и т.д. Каждая из них является переменной, которой присвоено значение, равное введенной команде. Соответственно, каждый вывод также нумеруется с помощью меток — "%o1", "%o2", "%o3", и т.д. Опять-таки, каждая из них является переменной, которой присвоено значение, равное результату выкладок. Существуют метки третьего типа "%t1", "%t2", и т.д., о которых будет сказано ниже.

MAXIMA (в точности как язык "C") игнорирует разбиение текста на строки. Можно вводить несколько команд в одной строке, можно разбивать одну команду на несколько строк. Комментарии в MAXIM'е тоже реализованы совершенно так же, как в "C" — два символа "/*" открывают комментарий, а два символа "*/" закрывают его.

Переменной "%" по определению присваивается результат последней выкладки.

Основные математические операции в MAXIM'е пишутся обычным образом — "+", "-", "*", "/"; возведение в степень — это "^" (крышечка), а присвоение (пожалуй, это довольно неудачная идея) записывается как ":" (двоеточие).

Попытка записать присвоение в виде "=" — постоянная ошибка при работе с MAXIM'ой.

Переменные могут принимать числовые значения — целые, рациональные, с плавающей точкой фиксированной (машинной) точности и с плавающей точкой неограниченной точности:

```
x:-7;
x:-13/5;
```

```
x:-0.012345;
x:77.7777e-5;
x:77.77777777777777777777777777777777b-5;
```

Кроме того, переменным могут быть присвоены значения в виде аналитических выражений

```
x:a^2+b;
```

Собственно, именно это и делает MAXIM'у системой аналитических вычислений. Заметим, что здесь определена **функция** (в математическом смысле этого слова) $x(a,b)$. В MAXIM'е есть функции и в программистском смысле, но они нужны совсем для других целей.

Если Вам нужна функция $f(x,y,z)$, то не определяйте функцию, просто заведите переменную `f`.

Кроме того, переменным могут быть присвоены значения в виде строковых переменных

```
x:"abcdefgh";
```

Вставить в строку одиночную кавычку "'", двойную кавычку "\"" и обратный слэш "\" можно совершенно таким же образом, как в "C", т.е. "\'", "\"" и "\\" соответственно.

Наконец, переменным могут быть присвоены логические значения — "`true`" или "`false`" ("истина" или "ложь"):

```
x:true;    y:false;
```

Знак "::" (два двоеточия подряд) — это присвоение с вычислением левой части. Значением левой части должен быть объект, которому можно что-либо присваивать. Так что в результате исполнения

```
s:2*a+b$
t:-b-a$
(s+t)::777$
```

переменной "a" будет присвоено значение "777". Круглые скобки здесь необходимы, т.к.

```
s+t::777$
```

вызовет сообщение об ошибке.

Опыт показывает, что в реальной научной работе проку от этой возможности немного.

Знак равенства резервирован для "уравнений" ("equation"). Уравнение — это левая и правая части равенства, соединенные знаком "=". К уравнению можно что-либо прибавить, из него можно что-либо вычесть, его можно умножить или поделить на то или иное выражение. Те же операции можно проделать с двумя уравнениями:

```
eq1:a=b$
eq1-c;
```

```
                                a-c = b-c
```

```
eq2:x=y$
eq1*eq2;
```

```
                                a x = b y
```

Знак ":=" применяется для определения функций, именно, записи

```
f(x):=x^2+5$
g(a,b):=a^2+3/b$
```

определяют функции одного и двух аргументов соответственно. Заметим еще раз, что оба определения функций совершенно нелепы, если Вы собираетесь, скажем, вычислять $\int dx\, f(x)$ или $\frac{\partial}{\partial a}\, g(a,b)$. Для этих манипуляций с функциями (в математическом смысле) следовало бы написать

```
f:x^2+5$  integrate(f,x);
```

или

```
g:a^2+3/b$  diff(g,a);
```

Кроме функций, в MAXIM'е можно определять макросы. Для этого используется знак "::=".

```
f(x)::=x^2+5$
g(a,b)::=a^2+3/b$
```

Можно считать, что макросы работают (почти) так же, как функции. Разницу между ними мы обсуждать не будем. Так что особой пользы от них нет.

Разумеется, определены стандартные математические функции

```
exp(x)      log(x)      sqrt(x)
sin(x)      cos(x)      tan(x)      cot(x)      csc(x)
sinh(x)     cosh(x)     tanh(x)     coth(x)     csch(x)
asin(x)     acos(x)     atan(x)     acot(x)     acsc(x)
asinh(x)    acosh(x)    atanh(x)    acoth(x)    acsch(x)
atan2(x,y)
```

При этом MAXIMA знает, что $\sin(-x) = -\sin(x)$, $\cos(0) = 1$, $\log(1) = 0$, и т.д.

Определены операции факториала и двойного факториала:

```
5!;
```

```
                                120
```

```
6!!;
```

```
                                48
```

```
5!!;
```

```
                                15
```

- **Функция** `max`
  перебирает свои аргументы и находит максимальное число

```
max(33,-22,11);
```

```
33
```

- **Функция** `min`
  перебирает свои аргументы и находит минимальное число

```
min(33,-22,11,44);
```

```
-22
```

3. Функции вывода на экран

При использовании терминатора ";" MAXIMA печатает результат вычислений, т.е. значение соответствующего выражения. Однако, для сложных синтаксических конструкций типа циклов значение будет "done", т.е. вполне бесполезное. Чтобы можно было реализовывать осмысленный вывод из сложных синтаксических конструкций типа блоков или циклов, предусмотрены специальные функции вывода.

- **Функция** print

печатает значения всех своих аргументов в одну строку

```
(%i31) print("D=",a+A,
       ", x is equal to",77,
       "or",88," or ",99);
                    D=a+A, x is equal to 77 or 88   or   99
                    (%o31)                          99
```

Эта функция, пожалуй, является основной и самой удобной для вывода на печать.

- **Функция** disp

печатает значения своих аргументов, причем каждое значение печатается в отдельной строке

```
(%i22) x:77$ y:44$ z:11$
(%i25) disp(x,y,z);
                                77
                                44
                                11
                    (%o25)      done
```

- **Функция** display

печатает значения своих аргументов вместе с их именем, каждое в отдельной строке

```
(%i12) display(x,y);
                                x=77
                                y=44
                    (%o12)      done
```

- **Функция** `ldisp`
печатает значения своих аргументов вместе с метками "%t". Эта функция "сбивает" нумерацию меток "%i" и "%o".

```
(%i4) ldisp(x,y,z);
                        (%t4)    77
                        (%t5)    44
                        (%t6)    11
                        (%o6)    [%t4, %t5, %t6]
(%i7) x;
                        (%o7)    77
```

- **Функция** `ldisplay`
печатает значения своих аргументов вместе с их именем и метками "%t". Эта функция также "сбивает" нумерацию меток "%i" и "%o".

```
(%i18) ldisplay(x,y);
                        (%t18)    x=77
                        (%t19)    y=44
                        (%o19)    [%t18, %t19]
```

Для иллюстрации приведем коротенький пример, иллюстрирующий ввод и вывод в MAXIM'е.

```
(%i1) 2+3;
                        (%o1)    5
(%i2) %;
                        (%o2)    5
(%i3) exp(x);
                                 x
                        (%o3)    %e
(%i4) (a+b)^2/(c+d);
                                 2
                        (%o4)  (b+a)
                               ------
                               d+c
(%i5) diff(f(x),x);
                        (%o5)    d
                                 -- (f(x))
                                 dx
(%i6) %o3;
                                 x
                        (%o6)    %e
(%i7) 3+4$
```

```
(%i8) %;
                                (%o8)    7
```

В дальнейшем мы (как правило) будем опускать метки в примерах.

Как видно из приведенного примера, MAXIMA старается нарисовать свою выдачу "красиво". Способ рисования определяется несколькими переменными, перечислим некоторые из них.

- **Переменная** `linel`

  определяет длину строки, в которую должна вписываться выдача. Изначально установлена величина "79". Если желательно получить более узкую страницу (например, 60 позиций для двухколоночной печати), следует присвоить переменной "linel" значение "60":

```
linel:60;
                                60
```

- **Переменная** `display2d`

  включает или выключает "двумерное" рисование дробей, степеней, и т.п. Изначально установлено значение "`true`". При этом дроби рисуются красиво, но выдача не может быть использована для ввода в MAXIM'у. Если установить значение "`false`", то вывод может быть впоследствии использован как ввод:

```
(x^2+a)/(y^2+b);
                                 2
                                x +a
                                ----
                                 2
                                y +b
display2d:false$
(x^2+a)/(y^2+b);
                                (x^2+a)/(y^2+b)
```

- **Переменная** `showtime`

  включает или выключает печать времени, затраченного на каждое действие. Изначально установлено значение "`false`". Если установить значение "`true`", то будет печататься, во-первых, "идеальное" процессорное время, в течение которого выполнялась выкладка, и, во-вторых, "физическое" время, затраченное на выкладку (в системах с делением времени второе всегда превышает первое).

```
showtime:true$
                  Evaluation took 0.00 seconds (0.00 elapsed)
fun1(a,b,c)$
                  Evaluation took 4.08 seconds (4.20 elapsed)
```

- **Функция** `fortran`
  позволяет получить выдачу, которую можно использовать как фрагмент фортрановской программы

```
fortran(sum(x^i,i,0,15));
                                  x**15+x**14+x**13+x**12+x**11+
                               1     x**10+x**9+x**8+x**7+x**6+x**5+
                               2     x**4+x**3+x**2+x+1
```

  Имейте в виду, что эта функция игнорирует параметр `linel` — длина строк соответствует старому фортрановскому стандарту — 72 позиции.

4. Работа с файлами

- **Функция** `batch`

  запускает файл с программой. Операторы выполняются один за другим либо до конца файла, либо до синтаксической ошибки, либо до некорректной операции.

  ```
  batch("myfile.mxm");
  ```

  Имейте в виду, что если Вы работаете с командной строки, то вложенный "`batch`" правильно работать не будет. Это значит, что если Вы запустили на исполнение некоторый файл (первый) с помощью "`batch`", а внутри этого первого файла есть команда "`batch`", запускающая на исполнение еще один файл (второй), то этот второй файл благополучно отработает, но, когда его исполнение прекратится, заодно прекратится и исполнение первого файла. Так что ни один оператор, стоящий после "`batch`" в первом файле, не будет исполнен. При использовании других интерфейсов это может быть незаметно, потому что, например, `emacs` при запуске файла на счет просто пересылает MAXIM'е одну строку за другой, а не использует команду "`batch`".

- **Функция** `load`

  загружает тот или иной файл.

  ```
  load(somefile);
  ```

  Тип загрузки зависит от типа файла. Именно, можно загружать файл с макросами, т.е. фактически файл с программой на MAXIMA (типичные расширения имен таких файлов "`.max`", "`.mxm`", "`.mc`" или "`.mac`"), можно загружать файл с программой на LISP (типичные расширения имен таких файлов "`.lisp`" или "`.lsp`"), и можно загружать двоичный файл с уже оттранслированными кодами (типичное расширение имен таких файлов "`.o`").

  Как правило эта функция необходима для загрузки того или иного пакета, который не загружается автоматически. Интересно, что разные пакеты хранятся в разных форматах, так что может грузиться и файл в формате MAXIMA, и LISP-файл и двоичный файл. Стандартные функции в MAXIMA либо входят в ядро системы и поэтому доступны изначально, либо являются автозагрузочными, т.е. при их первом вызове происходит автоматическая загрузка необходимого файла, либо требуют явной загрузки того или иного файла — в этом случае до их вызова необходимо написать "`load(packetname);`".

  При этом для загрузки, например, пакета интегрирования по частям можно написать как

  ```
  load(bypart)$
  ```

  так и

```
load("bypart");
```
и в том и в другом случае загрузится файл "`bypart.mac`".

- **Функция** `writefile`
  начинает писать всю выдачу MAXIM'ы в указанный файл. При этом (в отличие от REDUCE) выдача на терминал не прекращается:
  ```
  writefile("myoutput.mxm")$
  ```

- **Функция** `closefile`
  прекращает вывод в файл:
  ```
  closefile()$
  ```

Существуют и другие функции, позволяющие писать в файлы и читать из них, но (как нам кажется) при реальной научной работе они не очень полезны.

5. Преобразования аналитических выражений общего вида

- **Функция** `ev`

  является основной функцией, обрабатывающей выражения. Ее синтаксис довольно разнообразен.

```
ev(expr);
ev(expr,flag1,flag2,...);
ev(expr,x=a+b,y:c/d,...);
ev(expr,flag1,x=a,y:b,flag2,...);
```

  Можно даже опускать имя функции "`ev`"

```
expr,flag1,flag2,...;
expr,x=val1,y=val2,...;
expr,flag1,x=val1,y=val2,flag2,...;
```

  Следует, однако, иметь в виду, что в то время как записи "`ev(expr,flag);`" и "`expr,flag;`" являются синонимами, записи "`expr;`" и "`ev(expr);`" не идентичны, а именно:

```
v:a+b$ a:7$
v;
                          b+a
ev(v);
                          b+7
v,expand;
                          b+7
```

  На выражение "`expr`" по умолчанию действует функция упрощения. Если указаны флаги (их имена как правило совпадают с именами других функций, преобразующих выражения), то с выражением производятся действия в соответствии с этими флагами. Вот некоторые из флагов:

  `expand` `factor` `trigexpand` `trigreduce`

  (на выражение действуют одноименные функции, их описание см. далее),

  `pred` `diff` `simp`

  ("`pred`" вызывает вычисление значения логического выражения, "`diff`" вызывает выполнение "замороженного" дифференцирования, "`simp`" вызывает выполнение функции упрощения даже в том случае, когда переменная "`simp`" равна "`false`").

  Если указаны подстановки (в виде "`x=val1`" или "`x:val2`"), то они выполняются.

При этом повторный вызов функции "ev" вполне способен еще раз изменить выражение, т.е. обработка выражения не идет до конца при однократном вызове функции "ev".

```
ev((a+b)^2,expand);
```

$$b^2 + 2\,a\,b + a^2$$

```
ev((a+b)^2,a=x);
```

$$(x+b)^2$$

```
ev((a+b)^2,a:x,expand,b=7);
```

$$x^2 + 14\,x + 49$$

```
(a+b)^2,a=x,expand,b=7;
```

$$x^2 + 14\,x + 49$$

- **Переменная** `simp`

разрешает либо запрещает упрощение выражений. Изначально она равна "`true`", если установить ее равной "`false`", то упрощения производиться не будут:

```
simp:false$
x+y+x;
```

$$x + y + x$$

```
simp:true$
x+y+x;
```

$$y + 2\,x$$

- **Функция** `factor`

факторизует выражение.

```
factor( a*c+b*c+a*d+b*d );
```

$$(b+a)(d+c)$$

```
factor( (x^3+2*x^2*y+y^3)/
    (x^2+2*x*y+y^2)+
    x^2*y/(x^2+2*x*y+y^2)+
    3*x*y^2/(x+y)^2 );
```

$$y+x$$

- **Функция** gfactor

  отличается от функции "factor" тем, что умеет работать с мнимой единицей, т.е. может факторизовать выражения типа $x^2 + a^2$ и $x^2 + 2ixa - a^2$

```
factor(x^2+a^2);
```

$$x^2+a^2$$

```
factor(x^2+2*%i*x*a-a^2);
```

$$x^2 + 2\%i\ a\ x - a^2$$

```
gfactor(x^2+a^2);
```

$$(x-\%ia)(x+\%ia)$$

```
gfactor(x^2+2*%i*x*a-a^2);
```

$$(x+\%ia)^2$$

- **Функция** factorsum

  факторизует отдельные слагаемые в выражении.

```
factorsum( a^3+3*a^2*b+3*a*b^2+b^3
    +x^2+2*x*y+y^2 );
```

$$(y+x)^2+(b+a)^3$$

Не стоит особенно надеяться на эту функцию, поскольку многого она не замечает:

```
factorsum(a+x^2+2*x*y+y^2);
```

$$(y+x)^2+a$$

```
factorsum(a+x^2-y^2);
```

$$-y^2+x^2+a$$

- **Функция** gfactorsum

  отличается от "factorsum" тем же, чем "gfactor" отличается от "factor":

```
gfactorsum( a^3+3*a^2*b+3*a*b^2+b^3
    +x^2+2*%i*x*y-y^2 );
```

$$(b + a)^3 - (y - \%i\ x)^2$$

На эту функцию тоже не стоит надеяться, поскольку она многого не замечает:

```
gfactorsum(a+x^2+2*%i*x*y-y^2);
```

$$a - (y - \%i\ x)^2$$

```
gfactorsum(a+x^2+y^2);
```

$$y^2 + x^2 + a$$

- **Функция** expand

  раскрывает скобки.

```
expand( (a+b)*(c+d) );
```

$$b\,d + a\,d + b\,c + a\,c$$

```
expand( (x^3+2*x^2*y+y^3)/
    (x^2+2*x*y+y^2)+
    x^2*y/(x^2+2*x*y+y^2)+
    3*x*y^2/(x+y)^2 );
```

$$\frac{y^3}{y^2+2xy+x^2} + \frac{3xy^2}{y^2+2xy+x^2} + \frac{3x^2y}{y^2+2xy+x^2} + \frac{x^3}{y^2+2xy+x^2}$$

- **Функция** combine

  объединяет слагаемые с идентичным знаменателем

```
combine( x^3/(x^2+2*x*y+y^2)+
    3*x^2*y/(x^2+2*x*y+y^2)+
    3*x*y^2/(x^2+2*x*y+y^2)+
    y^3/(x^2+2*x*y+y^2)+
    a/(c+d)+b/(c+d) );
```

$$\frac{y^3+3xy^2+3x^2y+x^3}{y^2+2xy+x^2} + \frac{b+a}{d+c}$$

- **Функция** xthru

  приводит выражение к общему знаменателю, не раскрывая скобок и не пытаясь факторизовать слагаемые

```
xthru( 1/(x+y)^10+1/(x+y)^12 );
```

$$\frac{(y+x)^2 + 1}{(y+x)^{12}}$$

```
xthru( m/(x^2+2*x*y+y^2)+
    n/(x+y)^4 );
```

$$\frac{n\,(y^2 + 2\,x\,y + x^2) + m\,(y + x)^4}{(y + x)^4\,(y^2 + 2\,x\,y + x^2)}$$

Разумеется, в последнем случае разумнее сначала факторизовать каждое слагаемое, а уж потом применять "xthru". Применить функцию "factor" к отдельным слагаемым выражения можно с помощью функции "map":

```
f:map(factor, m/(x^2+2*x*y+y^2)+
    n/(x+y)^4 );
```

$$\frac{m}{(y + x)^2} + \frac{n}{(y + x)^4}$$

```
xthru(f);
```

$$\frac{m\ (y + x)^2 + n}{(y + x)^4}$$

- **Функция** multthru

умножает каждое слагаемое в сумме на множитель, причем при умножении скобки в выражении не раскрываются. Она допускает два варианта синтаксиса

```
multthru(mult,sum);
multthru(mult*sum);
```

(порядок сомножителей в последнем варианте не существенен).

```
multthru( (x+y)^2,(x+y)^5
    +1/(x+y)^7+(x+y)^2 );
```

$$(y + x)^7 + (y + x)^4 + \frac{1}{(y + x)^5}$$

```
multthru(   ( (x+y)^5+1/(x+y)^7
    +(x+y)^2 ) * (x+y)^2   );
```

$$(y + x)^7 + (y + x)^4 + \frac{1}{(y + x)^5}$$

```
multthru(    (  (x^3+2*x^2*y+y^3)/
    (x^2+2*x*y+y^2)+
    x^2*y/(x^2+2*x*y+y^2)+
    3*x*y^2/(x+y)^2  ) *
    (x^2+2*x*y+y^2)*(m+n)/(x+y)   );
```

$$\frac{(n + m)\ (y^3 + 2\ x^2\ y + x^3)}{y + x} + \frac{3\ (n + m)\ x\ y^2\ (y^2 + 2\ x\ y + x^2\ )}{(y + x)^3}$$

$$+ \frac{(n + m)\ x^2\ y}{y + x}$$

6. Преобразование рациональных выражений

Хотя функции, преобразующие выражения, не приводят их к канонической форме (в отличие от REDUCE), каноническое представление (CRE) существует — это каноническая форма для дробно-рациональных выражений. Выражение, приведенное к CRE, снабжается меткой /R/ сразу после метки "%o". Дальнейшая работа с ним идет быстрее, а вероятность его упрощения выше, чем для выражения общего вида.

- **Функция** `rat`

приводит выражение к каноническому представлению и снабжает его меткой "/R/". Она упрощает любое выражение, рассматривая его как дробно-рациональную функцию, т.е. работает с операциями "+", "-", "*", "/" и с возведением в целую степень. Нецелые степени она не упрощает, т.е. она не знает, что $(x^{a/2})^2 = x^a$. При этом вид ответа зависит от того, какие переменные она считает более главными, а какие менее главными. Упорядочение сначала идет по степеням самой главной переменной, внутри коэффициентов при этих степенях — по степеням менее главной переменной, и т.д. Изначально переменные упорядочены в алфавитном порядке и от начала к концу "главность" возрастает. Этот порядок можно откорректировать, добавив в аргументы функции имена переменных в порядке возрастания главности.

```
(%i11) rat( (x^3+2*x^2*y+y^3)/
    (x^2+2*x*y+y^2)+
    x^2*y/(x^2+2*x*y+y^2)+
    3*x*y^2/(x+y)^2 );
                        (%o11)/R/      y + x
(%i12) v1:m/(a+b)+n/(x+y)$

(%i13) rat(v1);
                                          my+mx+(b+a)n
                        (%o13)/R/     ------------------
                                      (b+a)y+(b+a)x
(%i14) rat(v1,y,x,n,m,b,a);
                                          na+nb+(x+y)m
                        (%o14)/R/     ------------------
                                      (x+y)a+(x+y)b
(%i15) rat(v1,m,n,a,b,x,y);
                                          my+mx+nb+na
                        (%o15)/R/     ------------------
                                      (b+a)y+(b+a)x
(%i16) rat(v1,m,n);
```

```
                                (%o16)/R/     (b+a)n+(y+x)m
                                            ---------------
                                            (b+a)y+(b+a)x
(%i17) rat(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  );

                                                  a/2 4      a/2 2
                                (%o17)/R/     (x   )  - 2(x   )  + 1
                                            ------------------------
                                                    a
                                                   x  - 1
```

- **Функция** `ratvars`

позволяет изменить алфавитный порядок "главности" переменных, принятый по умолчанию.

```
ratvars(z,y,x,w,v,u,t,s,r,q,p,o,n,m,l,k,j,i,h,g,f,e,d,c,b,a)$
```

меняет порядок главности в точности на обратный, а

```
ratvars(m,n,a,b)$
```

упорядочивает переменные "`m, n, a, b`" в порядке возрастания "главности", и делает их более главными, чем все остальные переменные. После этой команды получим

```
rat(v1);
```

$$\frac{\mathrm{nb+na+(y+x)m}}{\mathrm{(y+x)b+(y+x)a}}$$

- **Переменная** `ratfac`

включает или выключает частичную факторизацию выражений при сведении их к CRE. Изначально установлено значение "`false`". Если установить значение "`true`", то будет производиться частичная факторизация.

```
(%i4) v2:m/(a+b)^2+n/(x+y)^2$

(%i5) rat(v2);

                                (%o5)/R/     (my^2+2mxy+mx^2
                                           +(b^2+2ab+a^2)n) / ((b^2+2ab+a^2)y^2
                                           +(2b^2+4ab+2a^2)xy + (b^2+2ab+a^2)x^2 )
(%i6) ratfac:true$

(%i7) rat(v1);

                                (%o7)/R/     my+mx+(b+a)n
                                           ---------------
                                           (b+a)(y+x)
(%i8) rat(v2);
```

$$(\%o8)/R/ \qquad \frac{my^2+2mxy+mx^2 + (b^2+2ab+a^2)n}{(y^2+2xy+x^2)\ (b^2+2ab+a^2)}$$

- **Функция** `ratsimp`

приводит все куски (в том числе аргументы функций) выражения, которое не является дробно-рациональной функцией, к каноническому представлению, производя упрощения, которые не делает функция "`rat`". Не снабжает выражение меткой "`/R/`". Повторный вызов функции может изменить результат, т.е. упрощение не идет до конца.

```
(%i77) ratsimp(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  );
```

$$(\%o77) \ \frac{x^{2a} - 2x^a + 1}{x^a - 1}$$

```
(%i78) ratsimp(%);
```

$$(\%o78) \ x^a - 1$$

- **Функция** `fullratsimp`

вызывает функцию "`ratsimp`" до тех пор, пока выражение не перестанет меняться.

```
fullratsimp(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  );
```

$$x^a - 1$$

- **Переменная** `ratsimpexpons`

управляет упрощением показателей степени в выражениях. Изначально установлено значение "`false`". Забавно, что при этом аргумент любой функции упрощается:

```
fullratsimp(   sin(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  )   );
```

$$\sin(x^a - 1)$$

а показатель степени (в том числе показатель экспоненты) — нет:

```
fullratsimp(   exp(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  )   );
```

$$\%e^{\frac{x^{2a}}{x^a-1} - \frac{2x^a}{x^a-1} + \frac{1}{x^a-1}}$$

Если установить значение "`true`", то показатели степени начнут упрощаться:

```
ratsimpexpons:true$
fullratsimp(   exp(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  )   );
                                 a
                          %e^(x  - 1)
```

- **Функция** `ratexpand`
  раскрывает скобки в выражении. Не снабжает выражение меткой "`/R/`". Отличается от функции "`expand`" тем, что приводит выражение к канонической форме, поэтому ответ может оказаться короче, чем при применении "`expand`":

```
ratexpand( (x^3+2*x^2*y+y^3)/
    (x^2+2*x*y+y^2)+
    x^2*y/(x^2+2*x*y+y^2)+
    3*x*y^2/(x+y)^2 );
                              y + x
```

  (см. выше аналогичный пример с "`expand`").

7. Преобразование тригонометрических выражений

- **Функция** `trigexpand`
  раскладывает все тригонометрические функции от сумм в суммы произведений тригонометрических функций

```
trigexpand(sin(x+y));
                        cos(x) sin(y) + sin(x) cos(y)
```

- **Переменная** `trigexpand`
  управляет работой функции "`trigexpand`". Изначально переменная "`trigexpand`" равняется "`false`", это приводит к тому, что функция "`trigexpand`" не работает до конца, т.е. ее повторный вызов может изменить выражение. Если переменная "`trigexpand`" будет равна "`true`", то функция "`trigexpand`" будет работать до тех пор, пока выражение не перестанет меняться.

```
trigexpand(sin(2*x+y));
                        cos(2 x) sin(y) + sin(2 x) cos(y)
trigexpand(%);
                        (cos^2(x) - sin^2(x)) sin(y)
                              + 2 cos(x) sin(x) cos(y)
trigexpand:true;
                        true
trigexpand(sin(2*x+y));
                        (cos^2(x) - sin^2(x)) sin(y) +
                              + 2 cos(x) sin(x) cos(y)
```

- **Функция** `trigreduce`
  свертывает все произведения тригонометрических функций в тригонометрические функции от сумм. Функция работает не до конца, так что повторный вызов может изменить выражение

```
trigreduce( (cos(x)^2-
    sin(x)^2)*sin(y) +
    2*cos(x)*sin(x)*cos(y) );
                        sin(y+2x)/2 - sin(y-2x)/2 + cos(2x)sin(y)
trigreduce(%);
                        sin(y+2x)
```

- **Функция** `trigsimp`
  вовсе не упрощает выражение, а только применяет к нему правило $\sin^2(x) + \cos^2(x) = 1$:

```
trigsimp( (cos(x)^2-
    sin(x)^2)*sin(y) +
    2*cos(x)*sin(x)*cos(y) );
                    (2 cos^2(x) - 1) sin(y)
                          + 2 cos(x) sin(x) cos(y)
```

- **Функция** `trirat`
  пытается свести выражение с тригонометрическими функциями к некому универсальному каноническому виду (в общем, пытается упростить выражение). Как правило существенно упрощает выражение, но иногда работает очень долго (иногда бесконечно долго).

```
trigrat( (cos(x)^2-
    sin(x)^2)*sin(y) +
    2*cos(x)*sin(x)*cos(y) );
                    sin(y+2x)
```

8. Преобразование выражений со степенями и логарифмами

- **Функция** radcan
  упрощает выражения со вложенными степенями и логарифмами:

```
radcan(  log( x^3*exp(4*y)*
    exp(5*log(w))/z^6 )  );
                    - 6 log(z) + 4 y + 3 log(x) + 5 log(w)
radcan(  (x^(a/2)-1)^2 *
    (x^(a/2)+1)^2 / (x^a-1)  );
                    x^a - 1
```

- **Функция** rootscontract
  компактифицирует возведения в степень в данном выражении

```
rootscontract( x^(1/6)*
    y^(1/12)*z^(1/30) );
                    (x sqrt(y) z^(1/5) )^(1/6)
rootscontract( x^(1/2)*y^(1/2)*
    z^(1/4) );
                    sqrt(x y sqrt(z))
```

- **Функция** logcontract
  компактифицирует логарифмы в данном выражении

```
logcontract( a*log(x)+
    3*log(y)-4*log(x) );
                                3
                               y
                    log(------) + a log(x)
                               4
                              x
```

9. Логические выражения и база данных

Логические выражения образуются из операций сравнения

```
>   <   >=   <=   =   #
```

(символ # означает "не равно", а запись "a=b" имеет синоним "equal(a,b)"). Странноватой особенностью операций сравнения является то, что если их поставить в качестве условий в циклах и условных выражениях, то они будут вычислены, но взятые сами по себе, они не вычисляются:

```
3>2;
                        3>2
equal(3,2);
                        equal(3,2)
3#2;
                        3#2
```

Флаг "pred" в функции "ev" вызывает вычисление логических выражений:

```
ev(3#2,pred);
                        true
3#2,pred;
                        true
```

- **Функция** `is`

инициирует вычисление логического выражения

```
is(3>2);
                        true
is(3=2);
                        false
is(equal(3,2));
                        false
is(3#2);
                        true
```

Кроме того, определены встроенные логические функции, перечислим некоторые из них.

- **Функция** `atom`

  возвращает "`true`", если аргумент не имеет структуры, т.е. составных частей (например, число или переменная не имеют структуры).

```
atom(x);
                          true
atom(f(x));
                          false
```

- **Функция** `zeroequiv`

  проверяет, является ли некоторая функция одного аргумента нулем. Она возвращает "`true`", если функция равна нулю и "`false`" в противном случае.

```
zeroequiv(exp(2*x) - exp(x)^2, x)
                          true
```

- **Функция** `freeof`

  возвращает "`true`", если второй ее аргумент не содержит ("свободен от") первого

```
freeof(x,f(x+g(y)));
                          false
freeof(g,f(x+g(y)));
                          false
freeof(z,f(x+g(y)));
                          true
```

- **Функция** `symbolp`

  возвращает "`true`", если ее аргумент является символом:

```
symbolp(f(x));
                          false
symbolp(3);
                          false
symbolp(f);
                          true
```

- **Функция** scalarp
  возвращает "true", если ее аргумент является константой:

```
scalarp(f);
                          false
scalarp(sin(1/3));
                          true
scalarp(f(1/3));
                          false
scalarp(1/3);
                          true
```

- **Функция** listp
  возвращает "true", если ее аргумент является списком.

```
listp(x);
                          false
listp([x,y]);
                          true
```

- **Функция** matrixp
  возвращает "true", если ее аргумент является матрицей.

```
m:ident(2);
                          [ 1  0 ]
                          [      ]
                          [ 0  1 ]
matrixp(x);
                          false
matrixp(m);
                          true
```

- **Функция** numberp
  возвращает "true", если ее аргумент является числом:

```
numberp(1/3);
                          true
numberp(sin(1/3));
                          false
numberp(exp(1.0));
                          true
numberp(exp(1));
                          false
numberp(%pi);
                          false
numberp(1.3b22);
```

```
                        true
```

- **Функция** integerp
  возвращает "true", если ее аргумент является целым числом.

```
integerp(-3);
                        true
integerp(1/5);
                        false
```

- **Функция** oddp
  возвращает "true", если ее аргумент является целым нечетным числом.

```
oddp(-3);
                        true
oddp(4);
                        false
```

- **Функция** evenp
  возвращает "true", если ее аргумент является целым четным числом.

```
evenp(4);
                        true
evenp(-3);
                        false
```

- **Функция** primep
  возвращает "true", если ее аргумент является целым простым числом.

```
primep(11);
                        true
primep(9);
                        false
```

- **Функция** floatnump
  возвращает "true", если ее аргумент является действительным числом машинной точности.

```
floatnump(1.0);
                        true
floatnump(1);
                        false
floatnump(2.3e-4);
                        true
floatnump(2.3b-4);
                        false
```

- **Функция** `bfloatp`
возвращает "`true`", если ее аргумент является действительным числом неограниченной точности.

```
bfloatp(1.0);
                        false
bfloatp(1);
                        false
bfloatp(2.3b-4);
                        true
bfloatp(2.3e-4);
                        false
```

Кроме того, логическими выражениями являются запросы из базы данных:

```
is(a>3);
```

Следует подчеркнуть, что речь идет не о значении переменной "`a`" (которое не присвоено и, следовательно, неизвестно), а об информации, в какой области эта переменная может меняться.

- **Функция** `assume`
вводит информацию о переменной в базу данных.

```
assume(n>4);
                        [n>4]
```

После этого можно вводить запрос типа

```
is(n>1);
                        true
```

При этом запросы на информацию, которой в базе данных нет, вызовут сообщение "`unknown`":

```
is(n<7);
                        unknown
is(n>7);
                        unknown
```

Точно так же вызовет сообщение "`unknown`" более сложный запрос (который в принципе должен был бы дать значение "`true`"):

```
is(n^2+n>19);
                        unknown
```

Забавно, что результат других запросов довольно загадочен:

```
is(n^2+n>1);
                        unknown
```

```
is(n^2+n>0);
                                true
```

Повторное применение функции "assume" проверяется на противоречивость и на избыточность. В случае, если новая информация не противоречит предыдущим данным и не вытекает из них, она аддитивно добавляется к базе данных. К сожалению, предыдущие условия не проверяются на избыточность при появлении новых условий:

```
assume(n>3);
                                [redundant]
assume(n<3);
                                [inconsistent]
assume(n>10);
                                [n>10]
assume(n<30);
                                [n<30]
is(n<9);
                                false
is(n>31);
                                false
```

- **Функция** properties

печатает свойства переменной и, тем самым, позволяет выяснить, какая именно информация содержится в базе данных о данной переменной

```
properties(n);
                                [database info, n > 4, n > 10, 30 > n]
```

(новое условие $n > 10$ не отменило избыточное теперь условие $n > 4$).

Из приведенных примеров видно, что поменять свойство переменной на противоположное с помощью функции "assume" невозможно:

```
assume(x>0)$
assume(x<0);
                                [inconsistent]
is(x>0);
                                true
```

- **Функция** `forget`

  отменяет сведение, введенное в базу данных. Это позволяет поменять свойство переменной на противоположное.

```
forget(n<30);
                              [n<30]
properties(n);
                              [database info, n > 4, n > 10]
```

  Забавно, что после всех этих манипуляций можно присвоить переменной "n" значение, которое будет противоречить информации из базы данных:

```
n:-77;
                              -77
is(n>0);
                              false
properties(n);
                              [value, database info, n > 4, n > 10]
```

- **Функция** `kill`

  уничтожает всю информацию (как свойства, так и присвоенное значение) об объекте или нескольких объектах:

```
kill(x,y,z);
                              done
```

  Эта функция позволяет за один раз ликвидировать всю ранее введенную информацию о переменной "n".

```
kill(n);
                              done
properties(n);
                              [ ]
```

  Интересно, что эту функцию можно вызвать с аргументом "all". При этом будут "убиты" все определенные к настоящему времени переменные. Однако при этом MAXIMA не возвращается в стартовое состояние, поскольку параметрам и флагам не присваиваются первоначальные значения. Если переменной "linel" было присвоено значение "40", то после "kill(all);" оно так и останется "40", а не вернется к исходному значению "79".

Составные логические выражения формируются с помощью логических операций "and", "or", "not".

```
is( 3>1 and -1>=-3 and
    2#1 and not(equal(2,1)) );
                              true
```

```
is( 3<1 or -1<=-3 or
    not 2#1 or 2=1 );
                      false
```

10. Условные выражения и циклы

Синтаксис условного выражения может быть проиллюстрирован примером

```
a:1$
if a>3 then x:1 else x:-1;
                        -1
x;
                        -1
if a<3 then x:x+1 else x:x-1;
                        0
x;
                        0
```

Как обычно, часть с "`else`" можно опустить

```
if a>3 then x:1;
                        false
if a<3 then x:1;
                        1
```

Синтаксис цикла допускает три варианта

```
(%i5) for i:1 thru 3 step 2 do disp(i);
                                  1
                                  3
                        (%o5)     done
(%i6) for i:1 step 2 while i<6 do ldisplay(i);
                        (%t6)     i=1
                        (%t7)     i=3
                        (%t8)     i=5
                        (%o8)     done
(%i9) for i:1 step 2 unless i>4 do display(i);
                                  i=1
                                  i=3
                        (%o9)     done
```

(заодно мы еще раз проиллюстрировали работу функций "`disp`", "`display`" и "`ldisplay`").

Как обычно, если шаг равен единице, его можно опустить:

```
x:0$
for i:1 thru 5 do x:x+1$
x;
                        5
```

Кроме того, возможны циклы, в которых переменная цикла меняется не на фиксированную величину, а по произвольному закону:

```
for i:1 next 2*i thru 5 do disp(i)$
                        1
                        2
                        4
```

Разумеется, допустимы вложенные циклы и вложенные условные выражения.

Существуют также циклы суммирования и умножения.

- **Функция** `sum`
  реализует цикл суммирования

```
sum(x^i,i,3,5);
```

$$x^5+x^4+x^3$$

```
sum(x^i,i,a+3,a+5);
```

$$x^{a+5}+x^{a+4}+x^{a+3}$$

- **Функция** `product`
  реализует цикл умножения

```
product(x+i,i,3,5);
```

$$(x+3)(x+4)(x+5)$$

11. Блоки

Как в условных выражениях, так и в циклах вместо простых операторов можно писать составные операторы, т.е. блоки.

Стандартный блок имеет вид:

```
block([r,s,t],r:1,s:r+1,t:s+1,x:t,t*t);
```

Сначала идет список локальных переменных блока (глобальные переменные с теми же именами никак не связаны с этими локальными переменными). Список локальных переменных может быть пустым. Далее идет набор операторов.

Упрощенный блок имеет вид:

```
(x:1,x:x+2,a:x);
```

Обычно в циклах и в условных выражениях применяют именно эту форму блока.

Значением блока является значение последнего из его операторов.

Приведем несколько вполне бессмысленных примеров применения блоков в циклах и в условных выражениях

```
a:1$
if a>3 then (r:y,r:(r+1)*2) else
       block([s],s:x,s:s+1,r:s^2);
                                  (x + 1)^2
a:4$
if a>3 then (r:y,r:(r+1)*2) else
       block([s],s:x,s:s+1,r:s^2);
                                  2 (y + 1)
for i:1 thru 4 do  (s:0,x:z^i,t:1,
     for j:i thru i+3 do
          block([],s:s+j*t,t:t*x),
     print("s(",i,")=",s) )$
                        s( 1 )= 4 z^3 + 3 z^2 + 2 z + 1
                        s( 2 )= 5 z^6 + 4 z^4 + 3 z^2 + 2
                        s( 3 )= 6 z^9 + 5 z^6 + 4 z^3 + 3
                        s( 4 )= 7 z^12 + 6 z^8 + 5 z^4 + 4
```

Внутри *данного* блока допускаются оператор перехода на метку и оператор "`return`".

Оператор "return" прекращает выполнение *текущего* блока и возвращает в качестве значения блока свой аргумент

```
block([],x:2,x:x*x,
    return(x), x:x*x);
                        4
x;
                        4
```

В отсутствие оператора перехода на метку операторы в блоке выполняются последовательно. (В данном случае слово "метка" означает отнюдь не метку типа "%i5" или "%o7"). Оператор "go" выполняет переход на метку, расположенную в *этом же* блоке:

```
block([a],a:1,metka, a:a+1, if a=1001
    then return(-a), go(metka) );
                                      -1001
```

В этом блоке реализован цикл, который завершается по достижении "переменной цикла" значения 1001. Меткой может быть произвольный идентификатор. Следует иметь в виду, что цикл сам по себе является блоком, так что (в отличие от языка "C") прервать выполнение циклов (особенно вложенных циклов) с помощью оператора "go" невозможно — оператор "go" и метка окажутся в разных блоках.

То же самое относится к оператору "return". Если цикл, расположенный внутри блока, содержит оператор "return", то при исполнении оператора "return" произойдет выход из цикла, но не выход из блока:

```
block([],x:for i:1 thru 15 do if i=2 then
    return(555),display(x),777);
                                x=555
                                777
block([],x:for i:1 thru 15 do if i=52 then
    return(555),display(x),777);
                                x=done
                                777
```

Если необходимо выйти из нескольких вложенных блоков сразу (или нескольких блоков и циклов сразу) и при этом возвратить некоторое значение, то следует применять блок "catch"

```
catch( block([],a:1,a:a+1,
    throw(a),a:a+7),a:a+9 );
                                2
a;
                                2
catch(block([],for i:1 thru 15 do
```

```
    if i=2 then throw(555)),777);
                                    555
catch(block([],for i:1 thru 15 do
    if i=52 then throw(555)),777);
                                    777
```

Оператор "`throw`" — это аналог оператора "`return`", но он обрывает не текущий блок, а все вложенные блоки вплоть до первого встретившегося блока "`catch`".

Наконец, блок "`errcatch`" позволяет перехватывать некоторые (к сожалению, не все!) из ошибок, которые в нормальной ситуации привели бы к завершению счета. Например, если переменным "`y`" и "`z`" присвоены значения "`b+c`" и "`b-c`" соответственно, то оператор

```
(y+z)::3;
```

вызовет сообщение об ошибке и прервет исполнение файла. Однако, если поместить этот запрос в блок "`errcatch`", то произойдет только выход из этого блока. Значением блока в этом случае является пустой список:

```
(%i7) errcatch(a:2,y:b+c,z:b-c,(y+z)::3, a:-77);
                                Improper value assignment:
                                2 b
                                (%o7)          []
(%i8) a;
                                (%o8)          2
```

12. Списки

Список в MAXIMA — это объект, вполне аналогичный списку в LISP, т.е. упорядоченная совокупность произвольных (возможно разнородных) объектов (что-то вроде одномерного массива). Чтобы задать список, достаточно записать его элементы через запятую и ограничить запись квадратными скобками

```
li1:[a,b,11];
```

```
[a,b,11]
```

Элементом списка может быть любой объект, в том числе и другой список

```
li2:[a,b,[c,d],e,f];
```

```
[a,b,[c,d],e,f]
```

Список может быть пустым

```
li3:[];
```

```
[ ]
```

или состоять из одного элемента

```
li4:[77];
```

```
[77]
```

Ссылка на элемент списка производится так:

```
li1[2];
```

```
b
```

```
li1[2]:c;
```

```
c
```

```
li1;
```

```
[a,c,11]
```

```
li2[3];
```

```
[c,d]
```

```
li2[3][2];
```

```
d
```

- **Функция** `length`

возвращает длину списка

```
length([a,b,[c,d],e,f]);
```

```
5
```

- **Функция** part

позволяет выделить тот или иной элемент часть списка.

```
part([a,b,c],2);
                                b
part([a,[b,c],d],2);
                                [b,c]
```

Если список вложенный, то необязательно писать

```
part(part([a,[b,c],d],2),2);
                                c
```

можно просто написать

```
part([a,[b,c],d],2,2);
                                c
```

Следует подчеркнуть, что при присвоении списков

```
li1:[a,b,c]$
li2:li1;
                                [a,b,c]
```

отнюдь не создается копия списка "li1", просто переменная "li2" становится еще одним указателем на *тот же самый* список. Поэтому

```
li1[3]:d$
li2;
                                [a,b,d]
```

- **Функция** copylist

изготовляет "настоящую" копию списка

```
li1:[a,b,c]$
li3:copylist(li1);
                                [a,b,c]
li1[3]:d$
li3;
                                [a,b,c]
```

- **Функция** makelist

позволяет создавать списки и допускает 2 варианта синтаксиса

```
makelist(i^3,i,1,3);
                                [1, 8, 27]
```

(здесь реализуется цикл по переменной "i" в пределах от 1 до 3), либо

```
makelist(x=i^2,i,[c,d,e]);
                                [x=c^2, x=d^2, x=e^2]
```

(здесь переменная "i" пробегает все элементы заданного списка).

- **Функция** append
  позволяет склеивать два списка

```
append([a,b],[c,d]);
                        [a, b, c, d]
li1:[a]$ li2:[b,c]$
append(li1,li2);
                        [a, b, c]
```

- **Функция** cons
  позволяет добавлять элемент в начало списка

```
cons(x,[a,b]);
                        [x, a, b]
```

- **Функция** endcons
  позволяет добавлять элемент в конец списка

```
endcons(x,[a,b]);
                        [a, b, x]
```

- **Функция** reverse
  меняет порядок элементов в списке на обратный

```
reverse([a,b,[c,d],e,f]);
                        [f,e,[c,d],b,a]
```

- **Функция** sort
  упорядочивает элементы списка

```
sort([3,a,-1,x,b,0]);
                        [-1,0,3,a,b,x]
```

  сначала идут числа (в порядке возрастания), затем идентификаторы (в алфавитном порядке).

- **Функция** sublist
  составляет список из тех элементов исходного списка, для которых заданная логическая функция возвращает значение "true"

```
sublist([3,a,-1,x,b,0],numberp);
                        [3,-1,0]
```

  (функция "numberp" выдает "true" для чисел и "false" во всех остальных случаях). При этом может использоваться логическая функция, определенная пользователем

```
f(x):=is(x>5)$
sublist([7,3,6,4,5],f);
                        [7,6]
```

- **Функция** `member`

  возвращает "`true`", если ее первый аргумент является элементом заданного списка, и "`false`" в противном случае

```
li1:[a,b,[c,d],e,f]$
member(b,li1);
                        true
member(c,li1);
                        false
member([c,d],li1);
                        true
```

- **Функция** `first`

  выделяет первый элемент списка

```
first([a,b,c]);
                        a
```

- **Функция** `rest`

  выделяет остаток после удаления первого элемента списка

```
rest(a,b,c);
                        [b,c]
```

- **Функция** `last`

  выделяет последний элемент списка

```
last([a,b,c]);
                        c
```

- **Функция** `map`

  применяет заданную функцию к каждому элементу списка

```
map(sin,[-1,0,1]);
                        [-sin(1),0,sin(1)]
```

  при этом может использоваться как стандартная функция, так и функция, определенная пользователем

```
f(x):=2*x$ map(f,[1,2,3]);
                        [2,4,6]
```

- **Функция** `apply`
  применяет заданную функцию ко всему списку (список становится списком аргументов функции). Например, если Вы соорудили список, состоящий из чисел:

```
li1:[33,-22,11]$
```

  то, чтобы найти максимальное или минимальное число, надо вызвать функцию "`max`" или "`min`". Однако, обе функции в качестве аргумента ожидают несколько чисел, а не список, составленный из чисел. Применять подобные функции к спискам и позволяет функция "`apply`":

```
apply(max,li1);
                        33
apply(min,li1);
                        -22
```

13. Массивы

Массивы — это объекты с индексами. Ссылка на элемент массива производится обычным образом: команда

```
ar[2,0,1];
```

выведет значение элемента массива, а команда

```
ar[55]:77+x$
```

присвоит элементу массива указанное значение.

В MAXIM'е определены массивы 3 видов.

Во-первых, это массивы неопределенного размера ("hashed" массив). Значения элементов такого массива задается либо "индивидуально"

```
ar1[23]:7777$
ar1[-4]:5555$
ar2[2,3]:777$
ar2[-22,-33]:555;$
```

либо как функция индексов

```
ar3[i,j]:=i+j;
```

$$ar3_{i\ ,\ j}:=i+j$$

Индексы этого массива могут принимать любые целочисленные (в том числе и отрицательные) значения. Если элементы заданы как функция индексов, то их значения будут вычисляться по приведенной формуле. При повторной ссылке на этот же элемент выкладки не производятся — используется вычисленное предыдущий раз значение. Кроме того, даже если задана функция индексов, отдельные элементы массива можно переопределять индивидуально

```
ar3[1,2];
                    3
ar3[-2,-3];
                   -5
ar3[-2,-3]:7;
                    7
ar3[3,4]:5;
                    5
```

Во-вторых, это массивы заданного размера

- **Функция** `array`

определяет массив с данным именем, определенным количеством индексов и заданным размером. Можно также указать его тип

```
array(ar4,2,1);
                        ar4
array(ar5,2);
                        ar5
array(ar6,float,2,1);
                        ar6
array(ar7,float,1,1,1);
                        ar7
```

Индексы этих массивов пробегают значения от 0 до указанного числа, так что фактический размер массивов "`ar4`" и "`ar6`" — 3×2, массива "`ar5`" — 3, а массива "`ar7`" — $2 \times 2 \times 2$.

Первоначальные (сразу после определения) значения элементов для массива с определенным типом — это нули, а для массива с неопределенным типом первоначальные значения не определены, и при попытке их вывода печатаются 5 диезов (`#####`).

В-третьих, это "самопечатающиеся" массивы

- **Функция** `make_array`

создает массивы третьего вида, содержимое которых печатается автоматически. Целесообразность использования массивов этого вида сомнительна.

```
ar8:make_array('any,3,2);
                        {Array:  #2A((NIL NIL)(NIL NIL)(NIL NIL)) }
ar9:make_array('float,3);
                        {Array:  #(0.0 0.0 0.0) }
```

Индексы здесь пробегают значения от 0 до указанного числа *минус 1* , т.е. размерность "`ar8`" — 3×2, а "`ar9`" — 3.

Первоначальные (сразу после создания) значения элементов для массива с определенным типом — это нули, а для массива с неопределенным типом (т.е. с типом "`any`") — это значение "`NIL`".

Для массивов первого и второго видов идентификатор — это именно имя массива и его использование без квадратных скобок не вызывает никаких действий:

```
ar4
                        ar4
```

Для массивов третьего вида идентификатор — это не имя массива, а обычная переменная, значение которой есть массив третьего вида. Но содержимое массива третьего вида печатается автоматически, поэтому

ar9;

```
{Array:  #(0.0 0.0 0.0) }
```

- **Переменная** `arrays`

  содержит список имен массивов первого и второго видов, определенных на данный момент

  arrays;

  ```
  [ar1,ar2,ar3,ar4,ar5,ar6,ar7]
  ```

- **Функция** `arrayinfo`

  печатает информацию о массиве — его вид (для массивов первого вида — "`hashed`", для массивов второго вида с неопределенным типом — "`declared`", для второго вида с определенным типом — "`complete`", для массивов третьего вида — "`declared`"). Далее печатается число индексов массива. Далее для массивов первого вида печатаются индексы элементов, значения которых известны (т.е. либо были присвоены, либо были вычислены по формуле), а для массивов второго и третьего видов — размер (в виде списка максимальных значений каждого из индексов, при этом для массива третьего вида из заданного при определении размера автоматически вычитается единица).

  arrayinfo(ar1);

  ```
  [hashed, 1, [-4], [23]]
  ```

  arrayinfo(ar2);

  ```
  [hashed, 2, [-22, -33], [2, 3]]
  ```

  arrayinfo(ar3);

  ```
  [hashed, 2, [-2, -3], [1, 2], [3, 4]]
  ```

  arrayinfo(ar4);

  ```
  [declared, 2, [2, 1]]
  ```

  arrayinfo(ar5);

  ```
  [declared, 1, [2]]
  ```

  arrayinfo(ar6);

  ```
  [complete, 2, [2, 1]]
  ```

  arrayinfo(ar7);

  ```
  [complete, 3, [1, 1, 1]]
  ```

  arrayinfo(ar8);

  ```
  [declared, 2, [2, 1]]
  ```

  arrayinfo(ar9);

  ```
  [declared, 1, [2]]
  ```

- **Функция** `listarray`

  печатает содержимое массивов первого и второго видов. При этом содержимое печатается в таком порядке: сначала все допустимые значения пробегает последний индекс, потом предпоследний и.т.д. Например, для массива с двумя индексами порядок индексов будет такой: (0,0) (0,1) (0,2) (0,3) (1,0) (1,1) (1,2) (1,3) (2,0) (2,1) (2,2) (2,3).

  Вызов этой функции для массива третьего вида приводит к сообщению об ошибке. Как уже было сказано, содержимым массива первого вида считаются те элементы, которые вычислялись либо присваивались явно. Поэтому

```
listarray(ar3);
                        [7, 3, 5]
ar3[6,6];
                        12
listarray(ar3);
                        [7, 3, 5, 12]
listarray(ar4);
                        [#####, #####, #####, #####, #####, #####]
ar4[2,0]:3$ ar4[0,1]:a+b$
listarray(ar4);
                        [#####, a+b, #####, #####, 3, #####]
listarray(ar6);
                        [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
ar8[1,0]:77$
ar8;
                        {Array:  #2A((NIL NIL)(77 NIL)(NIL NIL)) }
```

- **Функция** `fillarray`

  позволяет заполнять *одноиндексные* массивы третьего вида из списка (при этом длина списка может не совпадать с размерностью массива)

```
ar9:make_array('float,3);
                        {Array:  #(0.0 0.0 0.0) }
fillarray(ar9,[4.0,5.0])$
ar9;
                        {Array:  #(4.0 5.0 0.0) }
fillarray(ar9,[6.0,7.0,8.0,9.0])$
ar9;
                        {Array:  #(6.0 7.0 8.0) }
```

- **Функция** kill
  уничтожает указанный объект или объекты, в том числе и массив.

```
kill(ar1,ar4);
                        done
arrays;
                        [ar2,ar3,ar5,ar6,ar7]
```

- **Функция** remarray
  уничтожает массив или массивы

```
remarray(ar2,ar3,ar5,ar7);
                        [ar2,ar3,ar5,ar7]
arrays;
                        [ar6]
remarray(ar6)$
arrays;
                        [ ]
```

14. Матрицы

В MAXIM'е определены прямоугольные матрицы.

- **Функция** `matrix`

возвращает матрицу, заданную поэлементно

```
ma1:matrix([a,b,c],[d,e,f]);
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

Значение элемента матрицы извлекается так

```
ma1[1,2];
```

$$b$$

Точно так же можно поменять отдельный элемент матрицы

```
ma1[2,3]:77$
ma1;
```

$$\begin{bmatrix} a & b & c \\ d & e & 77 \end{bmatrix}$$

Существуют матрицы, состоящие из одной строки

```
ma2:matrix([a,b])
```

$$\begin{bmatrix} a & b \end{bmatrix}$$

и из одного столбца

```
ma3:matrix([a],[b])
```

$$\begin{bmatrix} a \\ b \end{bmatrix}$$

- **Функция** `genmatrix`

возвращает матрицу заданной размерности, составленную из элементов двухиндексного массива

```
ma4:genmatrix(ar1,2,2)
```

$$\begin{bmatrix} ar1_{1,1} & ar1_{1,2} \\ ar1_{2,1} & ar1_{2,2} \end{bmatrix}$$

при этом можно задать элемент массива в общем виде

```
ar2[i,j]:=10*i+2*j$
ma5:genmatrix(ar2,2,2);
```

$$\begin{bmatrix} 12 & 14 \\ 22 & 24 \end{bmatrix}$$

- **Функция** `zeromatrix`
  возвращает матрицу заданной размерности, составленную из нулей

```
zeromatrix(2,3);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- **Функция** `ident`
  возвращает единичную матрицу заданной размерности

```
ident(2);
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Присвоение матриц устроено в MAXIM'е так же, как и присвоение списков, именно, набор операторов

```
ma1:matrix([a,b],[c,d])$ ma2:ma1$
```

отнюдь не создает копию матрицы "`ma1`" с именем "`ma2`", а делает переменную "`ma2`" еще одним указателем на ту же самую матрицу. В результате

```
ma1[2,2]:77$
```

приведет к

```
ma2;
```

$$\begin{bmatrix} a & b \\ c & 77 \end{bmatrix}$$

- **Функция** `copymatrix`
  изготовляет "настоящую" копию матрицы

```
ma1:matrix([a,b],[c,d])$
ma2:copymatrix(ma1)$
ma1[2,2]:77$ ma2;
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

- **Функция** `row`
  выделяет заданную строку матрицы

```
ma1:matrix([a,b],[c,d])$
row(ma1,2);
```

$$\begin{bmatrix} c & d \end{bmatrix}$$

- **Функция** `col`
  выделяет заданный столбец матрицы

```
col(ma1,1);
```

$$\begin{bmatrix} a \\ c \end{bmatrix}$$

- **Функция** addrow

  добавляет строку к матрице

```
addrow(ma1,[e,f]);
```

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

- **Функция** addcol

  добавляет столбец к матрице

```
addcol(ma1,[e,f]);
```

$$\begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix}$$

- **Функция** submatrix

  выделяет из матрицы подматрицу. Аргументы функции имеют следующий вид: сначала через запятую идут номера вычеркиваемых строк, затем сама матрица, а затем номера вычеркиваемых столбцов

```
ar2[i,j]:=10*i+j$
ma2:genmatrix(ar2,4,5);
```

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \\ 41 & 42 & 43 & 44 & 45 \end{bmatrix}$$

```
submatrix(1,3,ma2,2,4,5);
```

$$\begin{bmatrix} 21 & 23 \\ 41 & 43 \end{bmatrix}$$

На матрицах определены обычные операции умножения на число, сложения и матричного умножения. Последнее реализуется с помощью бинарной операции "." (точка). Разумеется, размерности матриц должны обеспечивать математическую корректность операций

```
ma1:matrix([a,b],[c,d])$
ma1+3*ma1+ident(2);
```

$$\begin{bmatrix} 4a+1 & 4b \\ 4c & 4d+1 \end{bmatrix}$$

```
ma1.ma1;
```

$$\begin{bmatrix} bc+a^2 & bd+ab \\ cd+ac & d^2+bc \end{bmatrix}$$

По не очень ясной причине определена загадочная операция возведения матрицы в "обычную" степень

```
ma1^3;
```

$$\begin{bmatrix} a^3 & b^3 \\ c^3 & d^3 \end{bmatrix}$$

Однако, определена и операция матричного возведения в *целую* степень

```
ma1^^2;
```

$$\begin{bmatrix} bc+a^2 & bd+ab \\ cd+ac & d^2+bc \end{bmatrix}$$

```
ma1^^(-1);
```

$$\begin{bmatrix} \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

Детерминант можно вынести за пределы матрицы, если вызвать функцию "ev" с флагом "detout":

```
ev(ma1^^(-1),detout);
```

$$\frac{\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}}{ad-bc}$$

Кроме того, отдельно определена функция обращения матрицы, которая является синонимом операции возведения матрицы в степень "-1".

```
invert(ma1);
```

$$\begin{bmatrix} \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$$

Следует отдельно обсудить свойства умножения матриц на строки и столбцы. Как это ни странно, если матрица стоит слева, то правым сомножителем может быть не только столбец, но и строка и даже список.

```
li1:[e,f]$
str1:matrix([e,f])$
```

```
stlb1:matrix([e],[f])$
```

После этого операторы

```
ma1.li1;
ma1.str1;
ma1.stlb1;
```

приведут к одной и той же выдаче

$$\begin{bmatrix} bf+ae \\ df+ce \end{bmatrix}$$

Если матрица стоит справа, то в качестве левого сомножителя допустимы список и строка, но не столбец — в случае столбца появится сообщение об ошибке. Так что операторы

```
li1.ma1;
str1.ma1;
```

приведут к одной и той же выдаче

$$\begin{bmatrix} cf+ae & df+be \end{bmatrix}$$

- **Функция** `transpose`

  транспонирует матрицу

  ```
  transpose(ma1);
  ```

  $$\begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

- **Функция** `determinant`

  вычисляет детерминант матрицы

  ```
  determinant(ma1);
  ```

  ```
  ad-bc
  ```

- **Функция** `mattrace`

  вычисляет след матрицы (сумму ее диагональных элементов). Перед тем, как вызывать ее первый раз, необходимо загрузить пакет "`nchrpl`".

  ```
  load(nchrpl)$
  mattrace(ma1);
  ```

  ```
  a+d
  ```

- **Функция** charpoly
является до некоторой степени избыточной — она вычисляет характеристический полином матрицы, т.е. $det(\hat{m} - x)$ (корни этого полинома — собственные значения матрицы). Перед тем, как вызывать ее первый раз, необходимо загрузить пакет "nchrpl".

```
load(nchrpl)$
charpoly(ma1,t);
                              (a-t)(d-t)-bc
```

- **Функция** ncharpoly
есть улучшенная версия функции "charpoly". Перед тем, как вызывать ее первый раз, необходимо загрузить пакет "nchrpl".

```
load(nchrpl)$
ncharpoly(ma1,t);
                                 2
                              t  + (-a -d) t + a d - b c
```

- **Функция** eigenvalues
вычисляет собственные значения матрицы аналитически, если это возможно. Выдача этой функции достаточно прихотлива — она возвращает список, состоящий из двух списков. Первый содержит собственные значения, а второй — их кратности

```
ma2:matrix([0,1,0],[1,0,0],
           [0,0,1]);
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
eigenvalues(ma2);
                              [[-1, 1], [1, 2]]
```

- **Функция** eigenvectors
аналитически вычисляет собственные значения и собственные вектора матрицы, если это возможно. Выдача этой функции чрезвычайно прихотлива — она возвращает список, первый элемент которого — это в точности выдача функции "eigenvalues", а далее идут собственные вектора, каждый из которых представлен как список своих компонент (т.е. как строка)

```
eigenvectors(ma2);
                              [[[-1, 1], [1, 2]],
                                    [1, -1, 0], [1, 1, 0], [0, 0, 1]]
```

- **Функция** `uniteigenvectors`
  отличается от функции "`eigenvectors`" тем, что возвращает нормированные на единицу собственные вектора

```
uniteigenvectors(ma2);
```

$$[[[-1,1],[1,2]],\ [\frac{1}{\mathrm{sqrt}(2)},-\frac{1}{\mathrm{sqrt}(2)},\ 0],$$

$$[\frac{1}{\mathrm{sqrt}(2)},\ \frac{1}{\mathrm{sqrt}(2)},\ 0],\ [0,0,1]]$$

15. Внутреннее представление в MAXIM'е

Как и в большинстве других систем аналитических вычислений, аналитические выражения в MAXIM'е представляются в виде вложенных списков.
Поэтому те функции, которые действуют на списки, вполне можно применять и к аналитическим выражениям. Для определения количества частей в выражении следует применять функцию "`length`", для выделения отдельной части — функцию "`part`". При этом надо помнить, что применение функции "`part`" к объекту без структуры вызовет ошибку. Поэтому если природа выражения заранее неизвестна, прежде чем применять к выражению функцию "`part`", надо применить к нему функцию "`atom`". Только если "`atom`" выдаст "`false`", можно применять "`part`".
Чтобы проиллюстрировать особенности внутреннего представления, приведем несколько примеров

```
length(a+b)
                    2
length(a+b+c)
                    3
length(-a)
                    1
length(f(a))
                    1
part([a,b],0);
                    [
part([a,b],1);
                    a
part([a,b],2);
                    b
part(a=b,0);
                    =
part(a=b,1);
                    a
part(a=b,2);
                    b
part(a+b,0);
                    +
part(a+b,1);
                    b
part(a+b,2);
                    a
```

```
part(a-b,0);
                        +
part(a-b,1);
                        a
part(a-b,2);
                        -b
part(-b,0);
                        -
part(-b,1);
                        b
part(a*b,0);
                        *
part(a*b,1);
                        a
part(a*b,2);
                        b
part(a/b,0);
                        //
```
(именно так — два знака деления подряд!)
```
part(a/b,1);
                        a
part(a/b,2);
                        b
part(f(x),0);
                        f
part(f(x),1);
                        x
part(a+b+c,0);
                        +
part(a+b+c,1);
                        c
part(a+b+c,2);
                        b
part(a+b+c,3);
                        a
part(a*b*c,0);
                        *
part(a*b*c,1);
                        a
part(a*b*c,2);
```

```
                              b
part(a*b*c,3);
                              c
part((a+b)/(c+d),0);
                              //
part((a+b)/(c+d),1);
                              b+a
part((a+b)/(c+d),2);
                              d+c
part((a+b)/(c+d),2,2);
                              c
```

Очень полезны при выделении частей выражения функции "`first`", "`rest`" и "`last`".

```
first(x+y+z);
                              z
rest(x+y+z);
                              y+x
last(x+y+z);
                              x
```

Существуют и специальные функции для выделения частей выражения.

- **Функция** `lhs`

  выделяет левую часть уравнения

```
lhs(a+b=c+d);
                              b+a
```

- **Функция** `rhs`

  выделяет правую часть уравнения

```
rhs(a+b=c+d);
                              d+c
```

- **Функция** `num`

  выделяет числитель

```
num((a+b)/(c+d));
                              b+a
num(a+b);
                              b+a
```

- **Функция** `denom`

выделяет знаменатель

```
denom((a+b)/(c+d));
                              d+c
denom(a+b);
                              1
```

Для модификации выражений можно применять функции "`map`", "`apply`" и "`subst`".

Сумма является списком своих слагаемых, так что

```
map(factor, m/(x^2+2*x*y+y^2)+
    n/(x+y)^4);
```

$$\frac{\mathtt{m}}{(\mathtt{y}+\mathtt{x})^2}+\frac{\mathtt{n}}{(\mathtt{y}+\mathtt{x})^4}$$

Одним из имен функции сложения является символ "+". Так что

```
apply("+",[a,b]);
                              b+a
```

Функция "`subst`" видит все части выражения, в том числе и с номером "0". Поэтому

```
subst("+","[",[a,b]);
                              b+a
```

- **Функция** `pickapart`

позволяет разложить выражение на части вплоть до указанного уровня.

```
(%i23)    v1:pickapart( (a+b)/2 +
          sin(c)-d^2,1);
```

$$\begin{array}{ll}(\%\mathtt{t23}) & -\mathtt{d}^2\\ (\%\mathtt{t24}) & \sin(\mathtt{c})\\ (\%\mathtt{t25}) & \frac{\mathtt{b}+\mathtt{a}}{2}\\ (\%\mathtt{o25}) & \%\mathtt{t23}+\%\mathtt{t24}+\%\mathtt{t25}\end{array}$$

После этого можно упрощать не "v1", а работать с его частями — с переменными "%t23", "%t24" и "%t25", поскольку переменная "v1" все равно выражается через них:

```
%t25:77$
ev(v1);
                              -d^2 + sin(c) + 77
```

- **Функция** isolate
  позволяет отделить часть выражения, которая содержит указанную переменную, от части, которая ее не содержит

```
(%i6)  isolate(x*x+a*x+b*x+a+b+1,x);
                          (%t6)    b + a + 1
                                    2
                          (%o6)    x  + b x + a x + %t6
```

16. Синтаксические подстановки

- **Функция** `subst`
  реализует "чисто синтаксическую" подстановку. Подстановка — это замена некоторой переменной или более сложной конструкции в аналитическом выражении на нечто другое. Например, вместо переменной x можно "подставить" $a + b$. Аргументы функции "`subst`" идут в таком порядке: новое (то, что мы подставляем вместо старого), старое (то, вместо чего мы подставляем), и, наконец, выражение, в котором производится подстановка. Эта функция не понимает, что x^4 — это $x^2 * x^2$, поэтому в выражении

```
subst(y,x,x^2+x^4+x^5);
```

$$y^5 + y^4 + y^2$$

подстановка выполняется полностью, в выражении

```
subst(y^2,x^2,x^2+x^4+x^5);
```

$$y^2 + x^5 + x^4$$

выполняется частично, а в выражении

```
subst(m,x+y,x+y+z);
```

$$z + y + x$$

не выполняется вовсе.
Подстановка возможна не только для имен переменных, но и для имен функций

```
subst(gg,ff,ff(7+ff(5));
```

$$gg(gg(5) + 7)$$

- **Переменная** `exptsubst`
  запрещает или разрешает "степенные" подстановки. Изначально установлено значение "`false`", в результате

```
subst(y,%e^x,%e^x+%e^(a*x)+
    %e^(a*x+b*x) );
```

$$y + \%e^{(b + a)\ x} + \%e^{a\ x}$$

Если установить значение "`true`", то

```
exptsubst:true$
subst(y^2,x^2,x^2+x^4+x^5);
```

$$y^4 + y^2 + x^5$$

```
subst(y,%e^x,%e^x+%e^(a*x)+
    %e^(a*x+b*x) );
```

$$y^{b + a} + y^a + y$$

- **Функция** `ratsubst`
  работает так же, как "`subst`", но понимает, что x^4 — это $x^2 * x^2$, так что

```
ratsubst(y^2,x^2,x^2+x^4+x^5);
                                 4    4    2
                            x y  + y  + y
ratsubst(m,x+y,x+y+z);
                            z + m
ratsubst( m,3*x*y,
    expand((x+y)^3) );
                             3          3
                            y  + m y + x  + m x
```

17. Алгебраические подстановки

В рациональных выражениях существует особый класс подстановок, выполнение которых инициируется флагом "algebraic" в функции "ev".
Сначала задается одна или несколько подстановок.

- **Функция** tellrat
аддитивно добавляет алгебраическую подстановку к уже определенным подстановкам и печатает весь их список.

```
tellrat(z=y^3);
                              [z - y^3]
tellrat(w^3=x);
                              [w^3 - x, z - y^3]
tellrat();
                              [w^3 - x, z - y^3]
```

Для реализации подстановок выражение должно быть рациональным (должно быть снабжено меткой "/R/"), и к нему надо применить функцию "ev" с флагом "algebraic":

```
ev(z^2+z^3,algebraic);
                              z^3 + z^2
rat(%);
                              /R/    z^3 + z^2
ev(%,algebraic);
                              /R/    y^9 + y^6
ev(rat(w^3+w^6),algebraic);
                              /R/    x^2 + x
```

18. Подстановки по шаблону

Аппарат подстановок по шаблону в MAXIM'е заметно уступает по удобству работы и продуманности синтаксиса аналогичному аппарату в REDUCE. Более того, по сути единственным реальным усовершенствованием Mathematic'и по сравнению с MAXIM'ой является удобный аппарат подстановок по шаблону. Если в REDUCE достаточно написать

```
for all x,y let sin(x+y)=sin(x)*cos(y)+cos(x)*sin(y);
for all x,n such that numberp(n) and n>1 let sin(n*x) =
    sin((n-1)*x)*cos(x)+cos((n-1)*x)*sin(x);
```

чтобы задать правила преобразования выражений типа "sin(a+b+3*c)", то в MAXIM'е это потребует значительно больших усилий.

Зато вариантов функций, реализующих подстановки по шаблону в MAXIM'е гораздо больше, и все они работают по-разному.

Прежде всего, следует определить шаблон.

- **Функция** `matchdeclare`

определяет шаблон, удовлетворяющий тому или иному условию.

```
matchdeclare(a,true)$
matchdeclare(b,true)$
matchdeclare(n,numberp)$
matchdeclare(m,matrixp)$
```

после этого шаблоны "`a`" и "`b`" означают "что угодно", шаблон "`n`" означает "любое число" (т.е. любой объект, который при подстановке в функцию "`numberp`" дает "`true`"), шаблон "`m`" означает "любая матрица".

К сожалению, в отличие от REDUCE и Mathematic'и, запись "`a+b`" не означает теперь "любая сумма". Чтобы определить шаблон "любая сумма", придется проделать следующее

```
summap(x):=block([],if atom(x) then
    return(false), if part(x,0)="+" then
    return(true) else return(false))$
matchdeclare(anys,summap)$
```

Аналогично, чтобы определить шаблон "число, большее единицы", следует написать

```
num_g_1(x):=block([],if not numberp(x) then
    return(false), if x>1 then
    return(true) else return(false))$
matchdeclare(nnn,num_g_1)$
```

Теперь существует три возможности.

Во-первых, можно определять правила и применять их.

- **Функция** defrule

определяет "правило". Ее аргументы — имя правила, старое выражение, новое выражение.

```
defrule( ru1, fff(m),
    m^^3 + m );
```

```
                    ru1 :  fff(m) -> m^3 + m
```

```
defrule( ru2, ff(n), n+x^n );
```

```
                    ru2 :  ff(n) -> x^n + n
```

```
defrule( ru3, f(a), a+a^2 );
```

```
                    ru3 :  f(a) -> a^2 + a
```

```
defrule( ru4, sin(anys),
    sin(first(anys))*cos(rest(anys)) +
    cos(first(anys))*sin(rest(anys)) )$
defrule( ru5, cos(anys),
    cos(first(anys))*cos(rest(anys)) -
    sin(first(anys))*sin(rest(anys)) )$
```

- **Функция** disprule

печатает правила.

```
(%i5) disprule(ru2,ru3);
```

```
                    (%t5)    ru2 :  ff(n) -> x^n + n
                    (%t6)    ru3 :  f(a) -> a^2 + a
                    (%o6)    [%t5, %t6]
```

- **Функция** apply1

применяет указанные правила к выражению. Правила применяются "сверху вниз" (от более высокого уровня в выражении к более мелким его частям), притом на каждом уровне выражения сначала целиком (пока выражение не перестанет меняться) отрабатывается первое правило, потом второе, и т.д.

- **Функция** applyb1

отличается от "apply1" тем, что тот же самый алгоритм подстановок применяется "снизу вверх".

- **Функция** apply2

  отличается от "apply1" тем, что на каждом уровне выражения целиком отрабатывается весь указанный список правил.

```
ma1:matrix([2,0],[0,2])$
apply1(fff(ma1)*fff(7),ru1);
                          [ 10 fff(7)      0     ]
                          [                      ]
                          [    0       10 fff(7) ]
apply1(ff(5)+ff(y),ru2);
                                    5
                        ff(y) + x  + 5
apply1(f(sin(z)^2),ru3);
                                4       2
                        sin (z) + sin (z)
apply1(cos(x+y+z),ru4,ru5);
                        (cos(x) cos(y) - sin(x) sin(y)) cos(z) -
                         sin(y + x) sin(z)
apply1(%,ru4,ru5);
                        (cos(x) cos(y) - sin(x) sin(y)) cos(z) -
                         (cos(x) sin(y) + sin(x) cos(y)) sin(z)
apply1(cos(x+y+z),ru5,ru4,ru5);
                        (cos(x) cos(y) - sin(x) sin(y)) cos(z) -
                         (cos(x) sin(y) + sin(x) cos(y)) sin(z)
apply2(cos(x+y+z),ru4,ru5);
                        (cos(x) cos(y) - sin(x) sin(y)) cos(z) -
                         (cos(x) sin(y) + sin(x) cos(y)) sin(z)
applyb1(cos(x+y+z),ru4,ru5);
                        cos(y+x) cos(z) - sin(y+x) sin(z)
```

Приведенные здесь примеры иллюстрируют разницу между тремя вариантами функции "`apply`". Их вызов для одного и того же выражения "`cos(x+y+z)`" для одного и того же списка правил "`ru4, ru5`" дает разные ответы. Причины этого довольно очевидны.

Функция "`applyb1`" идет "снизу вверх", так что правило "`ru5`" срабатывает только на самом последнем ("верхнем") этапе, оно применяется ко всему выражению "`cos(x+y+z)`" в целом. После этого к отдельным слагаемым полученного выражения никаких правил уже не применяется, т.к. эти слагаемые "лежат ниже".

Напротив, функция "`apply2`" применяет оба правила сначала ко всему выражению "`cos(x+y+z)`" в целом, а потом применяет оба правила к отдельным слагаемым полученного выражения, в результате никаких сумм в аргументах тригонометрических функций не остается.

Наконец, функция "apply1" сначала отрабатывает правило "ru4", которое в выражении "cos(x+y+z)" применять не к чему. Только после этого она начинает применять правило "ru5". Сначала оно применяется ко всему выражению "cos(x+y+z)" в целом, а потом (более "низкий" уровень) к отдельному слагаемому "cos(x+y)". Правило "ru4" уже отработано, так что к слагаемому "sin(x+y)") оно не применяется (в отличие от функции "apply2").
Кроме того, здесь показано, как можно заставить "доработать до конца" функцию "apply1". Этого можно добиться или повторным вызовом функции (правило "ru4" применится к "недоработанному" слагаемому "sin(x+y)"), или удлинением списка правил за счет повторения одних и тех же правил (правило "ru5" применится к исходному выражению, после чего правила "ru4" и "ru5" применятся к отдельным слагаемым).

Как уже было сказано, попытка работать с шаблоном "a+b" ("любая сумма"), или с шаблоном "a+nnn" ("что угодно плюс число, большее единицы"), или с шаблоном "a*n" ("что угодно умножить на число") некорректны и вызывают предупреждения об ошибке:

```
defrule(ru6,f(a+b), g(a)+g(b) );
                        b + a partitions 'sum'
                        ru6 :  f(b + a) -> g(b) + g(a)
defrule(ru7,f(a+nnn), g(a)+g(nnn) );
                        nnn + a partitions 'sum'
                        ru7 :  f(nnn + a) -> g(nnn) + g(a)
defrule(ru8,f(n*a), n*g(a) );
                        n a partitions 'product'
                        ru8 :  f(a n) -> g(a) n
```

В строгом смысле эти записи действительно некорректны: "a" — это уже "что угодно", так что "a+b" это просто "a", точно так же "a+nnn" и "a*n" — это "a". Если Вы проигнорируете эти предупреждения, то начнутся чудеса:

```
apply2(f(5),ru7);
                        g(5) + g(0)
```

Этот результат вполне логичен, хотя и нежелателен. Впрочем, иногда правило "ru7" работает так, как задумано:

```
apply2(f(x+5),ru7);
                        g(x) + g(5)
apply2(f(x+1),ru7);
                        f(x + 1)
```

Для правила "ru6" ситуация еще хуже:

```
apply2(f(x+y),ru6);
```

```
                        g(y + x) + g(0)
```

```
apply2(f(x),ru6);
```

```
                        g(x) + g(0)
```

```
apply2(f(0),ru6);
```

```
                        2 g(0)
```

Корректный способ определить шаблон "что угодно умножить на число, большее единицы", выглядит так:

```
pronum(x):=block([aaa],
    if atom(x) then return(false),
    if not part(x,0)="*" then return(false),
    aaa:first(x),
    if not numberp(aaa) then return(false),
    if aaa>1 then return(true) else return(false) )$
matchdeclare(ppp,pronum)$
```

После этого можно определять правило преобразования выражений типа "sin(7x)":

```
defrule(ru9a,sin(ppp),
    sin(rest(ppp))*cos((first(ppp)-1)*rest(ppp))+
    cos(rest(ppp))*sin((first(ppp)-1)*rest(ppp)) )$
defrule(ru9b,cos(ppp),
    cos(rest(ppp))*cos((first(ppp)-1)*rest(ppp))-
    sin(rest(ppp))*sin((first(ppp)-1)*rest(ppp)) )$
apply2(sin(4x),ru9a,ru9b);
```

```
                        sin(x) (cos(x) (cos^2 (x) - sin^2 (x))
                              - 2 cos(x) sin^2 (x))
                              + cos(x) (sin(x) (cos^2 (x)
                              - sin^2 (x)) + 2 cos^2 (x) sin(x))
```

Разумеется, это крайне неудобно по сравнению с REDUC'ом или с Mathmatic'ой.

Во-вторых, можно определить безымянные let-правила, которые накапливаются аддитивно

- **Функция** `let`

  определяет `let`-правило.

```
let( fff(m), m^^3 + m );
                        fff(m) --> m^<3> + m
let( ff(n), n+x^n );
                        ff(n) --> x^n + n
let( f(a), a+a^2 );
                        f(a) --> a^2 + a
```

- **Функция** `remlet`

  отменяет определенные ранее правила. При этом

```
remlet(ff(n))$
```

  отменит только указанное правило, а

```
remlet(all)$
```

  отменит все определенные к настоящему моменту правила.

- **Функция** `letsimp`

  применяет к своему аргументу все известные на данный момент let-правила. Правила отрабатываются целиком, т.е. до тех пор, пока выражение не перестанет меняться.

```
letsimp(ff(5)+ff(y)+f(sin(z)^2);
                        sin^4(z) + sin^2(z) + ff(y) + x^5 + 5
remlet(ff(n))$
letsimp(ff(5)+ff(y)+f(sin(z)^2);
                        sin^4(z) + sin^2(z) + ff(y) + ff(5)
remlet(all)$
letsimp(ff(5)+ff(y)+f(sin(z)^2);
                        f(sin^2(z)) + ff(y) + ff(5)
```

В-третьих, можно довести то или иное правило до сведения основной функции, упрощающей выражения. В этом случае все замены будут выполняться автоматически, без каких либо действий со стороны пользователя.

- **Функция** `tellsimp`

  вводит правило в базу данных основной функции, упрощающей выражения.

```
tellsimp( sin(anys),
    sin(first(anys))*cos(rest(anys)) +
    cos(first(anys))*
    sin(rest(anys)) );
                        [sinrule1, simp-%sin]
tellsimp( cos(anys),
    cos(first(anys))*cos(rest(anys)) -
    sin(first(anys))*
    sin(rest(anys)) );
                        [cosrule1, simp-%cos]
cos(x+y+z);
                        (cos(x) cos(y) - sin(x) sin(y)) cos(z) -
                         - (cos(x) sin(y) + sin(x) cos(y)) sin(z)
```

19. Работа с float-числами

Обыкновенно MAXIMA старается работать с бесконечной точностью:

```
exp(1);
                                %e
```

Однако ее можно заставить работать и с действительными числами. Действительные числа машинной точности (как правило 16 знаков) записываются обычным образом:

```
2.5    -1.0e20    5.768e-34
```

Действительные числа неограниченной точности обязаны иметь показатель степени "b":

```
2.5b0    -1.0b20    5.768b-34
```

- **Переменная** `fpprec`

  определяет количество значащих цифр для чисел неограниченной точности. Изначально она равна 16.

```
exp(1.0);
                          2.7182818284590451
exp(1.0b0);
                          2.7182818284590452b0
fpprec:21;
                          21
exp(1.0b0);
                          2.7182818284590452353602b0
```

- **Функция** `float`

  конвертирует любые числа в выражениях в числа машинной точности.

```
float(1/3);
                          0.3333333333333333
float(f(1/3));
                          f(0.3333333333333333)
float(%e);
                          %e
float(%pi);
                          %pi
float(1.0b0);
                          1.0e0
float(sin(%pi/6);
                          0.5
```

- **Функция** `bfloat`
  конвертирует любые числа в выражениях в числа неограниченной точности. При этом, если встречается действительное число машинной точности, она печатает предупреждение о изменении точности:

```
bfloat(1/3);
                        0.3333333333333333b0
bfloat(%e);
                        2.7182818284590 45b0
bfloat(%pi);
                        3.1415926535897 93b0
bfloat(exp(1.0));
                        Warning:   float to bigfloat conversion
                                        of 2.7182818284590451
                        2.7182818284590 45b0
bfloat(sin(%pi/6);
                        0.5b0
```

  Существует и обратное преобразование. Каноническая форма рационального выражения не должна содержать действительных чисел. Поэтому функция "`ratsimp`" преобразует любое действительное число в рациональное и печатает при этом предупреждение.

- **Переменная** `ratepsilon`
  задает точность преобразования действительного числа в рациональное. Изначально установлено значение $2.0 \cdot 10^{-8}$.

```
ratsimp(x+1.23456789);
                        'rat' replaced 1.2345678900000001 by
                                       100/81=1.2345679012345678
                        81 x + 100
                        ----------
                            81
ratepsilon:0.001$
ratsimp(x+1.23456789);
                        'rat' replaced 1.2345678900000001 by
                                       21/17=1.2352941176470589
                        17 x + 21
                        ---------
                            17
```

Существует целый набор функций, который реализует не аналитические, а численные алгоритмы, и выдает в качестве ответов действительные числа.

- **Функция** `allroots`
  функция, которая находит и печатает все (в том числе и комплексные) корни *полиномиального* уравнения с действительными либо комплексными коэффициентами.

- **Переменная** `polyfactor`
  определяет форму выдачи функции "`allroots`". Изначально она равна "`false`", при этом корни выводятся в виде списка. Если установить ее равной "`true`", то вместо списка корней будет в общем случае выводиться разложение полинома на линейные сомножители. Для полинома с действительными коэффициентами будет выводиться разложение на линейные сомножители для действительных корней и на квадратичные для каждой из пар сопряженных друг другу комплексных корней.

```
allroots(x^2-3*%i*x-2=0);
                        [x = 1.0 %i + 9.4182784545287731E-21,
                            x = 2.0*%i - 9.4182784545287731E-21]
allroots(x^6+1=x^2+x^4);
                        [x = 1.0 %i - 1.1323771713605928E-19,
                          x = -1.0 %i - 1.1323771713605928E-19,
                          x = 4.7121609153868797E-8 %i
                                                - 0.99999999999992,
                          x = - 4.7121609153868797E-8 %i
                                                - 0.99999999999992,
                          x = 0.99999944304722,
                                 x = 1.000000556952626]
polyfactor:true$
allroots(x^2-3*%i*x-2=0);
                    1.0 (x - 2.0 %i + 9.4182784545287731E-21)
                           (x - 1.0 %i - 9.4182784545287731E-21)
allroots(x^6+1=x^2+x^4);
                    1.0 (x - 1.000000556952626)
                                          (x - 0.99999944304722)
                        (x^2 + 2.2647543427211855E-19 x + 1.0)
                           (x^2 + 1.999999999999846 x
                                                + 0.99999999999985)
```

Здесь хорошо заметно одно неприятное свойство функции "`allroots`": если корни не являются кратными, то точность ответа порядка машинного нуля, а вот при поиске кратных корней точность существенно снижается (в данном случае точность порядка $5 \cdot 10^{-7}$)

- **Функция** `find_root`

  находит корень уравнения на заданном интервале методом деления отрезка пополам.

```
6*find_root(sin(x)=0.5,x,0,1);
```

```
3.141592653589795
```

- **Функция** `newton`

  находит корень указанной функции методом Ньютона.

  По совершенно необъяснимым причинам у этой функции есть две версии с разным синтаксисом.

  Первая из версий содержится в пакете "`newton`" и имеет два аргумента — функцию, корень которой мы ищем, и начальную точку

```
load(newton)$
6*newton(sin(x)-1/2,0);
```

```
3.1415926123623486b0
```

  Вторая из версий содержится в пакете "`newton1`" и имеет четыре аргумента — функцию, корень которой мы ищем, переменную, начальную точку и заказанную точность поиска корня

```
load(newton1)$
2*newton (cos (u), u, 1, 1/100);
```

```
3.141350554322501
```

- **Функция** `mnewton`

  находит корень системы уравнений многомерным методом Ньютона. Для использования функции необходимо сначала загрузить пакет "`mnewton`". Функция имеет три аргумента — список уравнений (в виде списка функций, нули которых мы ищем), список переменных, и список начальных значений переменных (начальная точка рекурсивной процедуры).

- **Переменная** `newtonepsilon`

  задает точность поиска корня для функции "`mnewton`". По умолчанию установлено значение `10^(-fpprec/2)`, что далеко не всегда является оптимальным выбором.

- **Переменная** `newtonmaxiter`

  задает максимальное количество итераций в многомерном методе Ньютона. По умолчанию установлено значение "50". Обыкновенно этого количества действительно бывает достаточно для сходимости.

```
load(mnewton)$
mnewton([x^2-y^2, x^2+y^2],[x,y],[2,1]);
                          [[x = 7.4505805969238281E-9,
                                y = 3.7252902984619141E-9]]
newtonepsilon:1.0e-12$
mnewton([x^2-y^2, x^2+y^2],[x,y],[2,1]);
                         [[x = 9.0949470177292824E-13,
                                y = 4.5474735088646412E-13]]
```

- **Функция** `lsquares_estimates`

  фитирует параметры уравнения (уравнение пишется для "измеряемых величин") в соответствии с "экспериментальными данными" методом наименьших квадратов. Для использования функции необходимо сначала загрузить пакет "`lsquares`". Функция имеет четыре аргумента — список "экспериментальных данных" в виде матрицы (каждая строка матрицы — это список значений "измеряемых величин", т.е. "одна экспериментальная точка"); список имен "измеряемых величин"; гипотетическое уравнение (в это уравнение кроме "измеряемых величин" должны входить и искомые параметры); и список имен параметров.

```
load(lsquares)$
lsquares_estimates(matrix([1,0.9],[2,2.1],[3,2.9]),
        [x,y], y=a*x+b, [a,b]);
```

$$[[a = 1,\ b = -\frac{1}{30}]]$$

- **Функция** `romberg`

  численно находит определенный интеграл функции на заданном отрезке. При этом используется алгоритм Ромберга, т.е. экстраполяция интегральных сумм, полученных при убывающем шаге решетки h_0, $h_1 = h_0/2$, $h_2 = h_1/2$, $h_3 = h_2/2$, ... по переменной $h^{(7)}$ в точку $h_\infty = 0$, соответствующую точному ответу.

---

(7) На самом деле по переменной h^2.

- **Переменная** `rombergtol`
  задает относительную точность, которая должна быть достигнута при интегрировании. Изначально установлено значение `1.0e-4`. Это разумный выбор, т.к. фактическая точность алгоритма Ромберга обычно заметно больше заказанной

```
romberg(cos(x)^2,x,0,2*%pi)-
    4.0*atan(1.0);
                        -4.2479656877873199E-9
rombergtol:1.0e-11$
romberg(cos(x)^2,x,0,2*%pi)-
    4.0*atan(1.0);
                        -7.8179366199075434E-17
```

- **Переменная** `rombergit`
  задает максимальное количество итераций. Изначально устанавливается значение 11. Если требуемая точность за это количество итераций не достигается, выдается сообщение об ошибке и происходит выход из системы.

```
rombergtol:1.0e-11$
errcatch(romberg(sin(x)/
    (x+0.01)^2,x,0,20));
                        'romberg' failed to converge
                        []
rombergit:20$
errcatch(romberg(sin(x)/
    (x+0.01)^2,x,0,20));
                        [4.042159703041129]
```

Если Вы считаете нужным вычислять определенный интеграл, пользуясь именно MAXIM'ой (в действительности это лучше сделать на языке "C"), то подынтегральную функцию следует откомпилировать, причем непременно с декларацией типа функции и аргумента (см. раздел "Транслятор и компилятор в MAXIM'е").

20. Комплексные числа и выражения

Мнимая единица в MAXIM'е записывается как "%i". С ее помощью можно конструировать комплексные выражения:

```
a+%i*b;
```

```
%i b + a
```

- **Функция** realpart

возвращает действительную часть выражения

```
realpart(a+%i*b);
```

```
a
```

- **Функция** imagpart

возвращает действительную часть выражения

```
imagpart(a+%i*b);
```

```
b
```

- **Функция** cabs

возвращает модуль комплексного выражения

```
cabs(a+%i*b);
```

```
sqrt(b^2 + a^2)
```

- **Функция** carg

возвращает фазу комплексного выражения

```
carg(a+%i*b);
```

```
atan2(b, a)
```

Как видно из приведенных примеров, по умолчанию все переменные считаются действительными. Можно декларировать, что та или иная переменная является комплексной:

```
declare(z,complex);
```

```
done
```

Эта информация записывается в базу данных и является свойством переменной.

- **Функция** properties

печатает свойства переменной

```
properties(z);
```

```
[database info, kind(z, complex)]
```

- **Функция** remove
  удаляет свойство переменной

```
remove(z,complex);
                                done
properties(z);
                                []
```

  Если переменная "z" декларирована как комплексная переменная, то функции типа "realpart" это учитывают, например:

```
realpart(z^2);
                                realpart^2(z) - imagpart^2(z)
```

- **Функция** rectform
  приводит выражение к виду $\mathrm{Re}(z) + i * \mathrm{Im}(z)$

```
rectform(r*exp(%i*fi));
                                %i sin(fi) r + cos(fi) r
```

- **Функция** polarform
  приводит выражение к виду $\mathrm{mod}(z) * \exp(i * \arg(z))$

```
polarform(x+%i*y);
                                sqrt(y^2 + x^2) %e^%i atan2(y,vx)
```

- **Функция** exponentialize
  заменяет все тригонометрические функции на соответствующие комбинации экспонент

```
exponentialize(cos(a+%i*b));
```

$$\frac{\%e^{\%i\ (\%i\ b\ +\ a)} + \%e^{-\%i\ (\%i\ b\ +\ a)}}{2}$$

- **Функция** demoivre
  заменяет все экспоненты с мнимыми показателями на соответствующие тригонометрические функции

```
demoivre(exp(a+%i*b)
                                %e^a (%i sin(b) + cos(b))
```

21. Дифференцирование

- **Функция** `diff`

выполняет дифференцирование. Ее синтаксис довольно разнообразен

```
diff(x^2,x)
```

$$2\ x$$

```
diff(x^3,x,2)
```

$$6\ x$$

```
diff(y^3 * x^3,x,2,y,1)
```

$$18\ x\ y^{2}$$

При этом первоначально все переменные считаются независимыми

```
diff(y,x)
```

$$0$$

Работать с производной одной переменной по другой можно тремя способами: "заморозить" операцию дифференцирования, указать на зависимость явно и декларировать зависимость неявно. Запретить выполнение функции "`diff`" и получить "замороженную" производную можно, если перед именем функции "`diff`" поставить одиночную кавычку:

```
'diff(y,x)
```

$$\frac{dy}{dx}$$

Можно явно указать на зависимость, т.е. работать с функцией:

```
diff(y(x),x)
```

$$\frac{d}{dx}\ (y(x))$$

```
diff(v(x,y),x,2,y,1)
```

$$\frac{d^{3}}{dx^{2}dy}\ (v(x,y))$$

(здесь предполагается, что функции "`y`" и "`v`" не были ранее определены с помощью оператора определения функции "`:=`").

- **Функция** `depends`

позволяет декларировать, что переменная зависит от одной или нескольких других переменных

```
depends(y,x);
```

$$[y(x)]$$

```
depends(u,[x,y]);
```

$$[u(x,\ y)]$$

- **Переменная** `dependencies`

содержит список "зависимостей", определенных на данный момент

dependecies;

```
[y(x), u(x, y)]
```

Зависимость "u" от "x" и "y" является "свойством" переменной "u".

- **Функция** `properties`

печатает список свойств заданной переменной

properties(u);

```
[dependency]
```

- **Функция** `remove`

удаляется указанное свойство данной переменной

remove(u,dependency);

```
done
```

dependencies;

```
[y(x)]
```

Интересно, что MAXIMA правильно учитывает зависимости переменных для случая вложенных функций:

diff(v(x,y),x,1,y,1)

$$\frac{d^2}{dx\ dy}\ (v(x,y))$$

diff(u,x,1,y,1)

$$\frac{d^2\ u}{dy^2}\ \frac{dy}{dx} + \frac{d^2\ u}{dx\ dy}$$

(как нетрудно понять, обе записи абсолютно корректны математически).

- **Функция** `gradef`
  определяет результат дифференцирования функции по своим аргументам

```
gradef(f(a,b,c),g(a,b,c),
       77,c*f(a,b,c));
                                    f(a,b,c)
```

  тем самым определено, что

$$\frac{\partial}{\partial a} f(a,b,c) = g(a,b,c), \qquad \frac{\partial}{\partial b} f(a,b,c) = 77,$$

$$\frac{\partial}{\partial c} f(a,b,c) = c * f(a,b,c)$$

  (здесь предполагается, что функция "`f`" не была ранее определена с помощью "`:=`").
  Градиент функции, как и зависимость переменных, есть свойство:

```
properties(f);
                                    [gradef]
```

- **Функция** `prinprops`
  печатает указанное свойство данной переменной

```
printprops(f,gradef);
                                    d
                                   --- (f(a,b,c)) = g(a,b,c)
                                    da
                                    d
                                   --- (f(a,b,c)) = 77
                                    db
                                    d
                                   --- (f(a,b,c)) = c f(a,b,c)
                                    dc
```

  Имейте в виду, что информация о зависимости переменных может быть распечатана только с помощью переменной "`dependencies`". По не очень понятным причинам функция "`prinprops`" отказывается печатать эту информацию, хотя "`dependency`" — это свойство переменной. Так что команда

```
prinprops(y,dependency);
```

  вызовет сообщение об ошибке.

22. Пределы

- **Функция** `limit`
  вычисляет предел заданного выражения при стремлении переменной к указанному значению. В тех случаях, когда левый и правый предел не совпадают, можно уточнить, с какой стороны берется предел. Существует четыре специальные значения — "`inf`" ($+\infty$), "`minf`" ($-\infty$), "`und`" ($\pm\infty$), "`ind`" (неопределенность):

```
limit(sin(x)/x,x,0);
                        1
limit(1/x,x,0);
                        und
limit(1/x,x,0,plus);
                        inf
limit(1/x,x,0,minus);
                        minf
limit(sin(1/x),x,0);
                        ind
limit((x+1)/(x+2),x,inf);
                        1
```

  Функция применяет правило Лопиталя.

- **Переменная** `lhospitallim`
  ограничивает число дифференцирований при вычислении предела, причем если этого количества дифференцирований не хватает, то функция возвращает саму себя. Изначально установлено значение "4", поэтому

```
limit((sin(x)-x)^2/
    (x^4*(cos(x)-1)),x,0);
                        limit  (sin(x)-x)^2
                         x->0  ------------
                                4  cos(x)-x^4
                               x
lhospitallim:16$
limit((sin(x)-x)^2/
    (x^4*(cos(x)-1)),x,0);
                        - 1/18
```

  Вообще функция реализована не очень аккуратно, примером чему может служить такая выкладка

```
lhospitallim:16$
```

```
limit((sin(x)-x)^2/x^6,x,0);
                          1
                         --
                         36
limit((cos(x)-1)^3/x^6,x,0);
                         - 1
                           -
                           8
limit((cos(x)-1)^3/
    (sin(x)-x)^2,x,0);
                         - 9
                           -
                           2
limit((sin(x)-x)^2/
    (cos(x)-1)^3,x,0);
                         inf
```

Последний ответ, очевидно, неверен.

- **Функция** `tlimit`
  отличается от функции "`limit`" только алгоритмом — она раскладывает выражение в ряд Тейлора. Благодаря этому она (в отличие от функция "`limit`") выдает верный ответ и для случая

```
limit((sin(x)-x)^2/
    (cos(x)-1)^3,x,0);
                         - 2
                           -
                           9
```

23. Интегрирование

- **Функция** integrate

  выполняет интегрирование заданного выражения по указанной переменной (неопределенная константа не добавляется). Можно также указать пределы интегрирования — в этом случае вычисляется определенный интеграл.

```
integrate(1/(x+a),x);
```

$$\log(x+a)$$

```
integrate(x^3,x,a,b);
```

$$\frac{b^4}{4} - \frac{a^4}{4}$$

Определенный интеграл, зависящий от параметра, может быть по нему продифференцирован

```
w:integrate(f(x,y),x,a(y),b(y));
```

$$\int_{a(y)}^{b(y)} f(x,y)\,dx$$

```
diff(w,y);
```

$$f(b(y),y)\,\frac{d}{dy}\,(b(y)) - f(a(y),y)\,\frac{d}{dy}\,(a(y)) + \int_{a(y)}^{b(y)} \frac{d}{dy}\,(f(x,y))\,dx$$

- **Функция** changevar

  реализует замену переменных в интеграле. Ее аргументы должны иметь вид: сам интеграл, связь старой и новой переменной, новая переменная, старая переменная. Связь переменных задается либо в явной форме ("x=g(t)"), либо в виде выражения, один из корней которого и дает связь переменных ("x-g(t)").

```
w:integrate(f(x),x)$
changevar(w,x=g(t),t,x);
```

$$\int f(g(t))\,(\frac{d}{dt}\,(g(t)))\,dt$$

```
changevar(w,x^2-g(t),t,x);
```

$$-\frac{\int \frac{f(-sqrt(g(t)))}{sqrt(g(t))}\,(\frac{d}{dt}\,(g(t)))\,dt}{2}$$

Для определенных интегралов выполняется подстановка в пределах интегрирования, так что функция, связывающая старую и новую переменную, должна быть обратима. Поэтому попытка написать

```
w:integrate(f(x),x,a,b)$
changevar(w,x=g(t),t,x);
```

```
                        Unable to solve for t
                                    []
```

вызовет сообщение об ошибке ("$g^{-1}(a)$" и "$g^{-1}(b)$") не могут быть вычислены). Однако, вполне допустимо

```
changevar(w,x=t+c,t,x);
```

$$\int_{a-c}^{b-c} f(t+c)\,dt$$

и

```
changevar(w,x^2-t, t, x);
```

$$-\frac{\int_{a^2}^{b^2} \frac{f(-\sqrt{t})}{\sqrt{t}}\,dt}{2}$$

- **Функция** byparts

выполняет интегрирование по частям. Перед первым вызовом функции необходимо загрузить пакет "bypart", в котором она определена:

```
load(bypart)$
```

Аргументы должны иметь вид: интеграл, переменная интегрирования, "u" и "v'" (используется формула "$\int uv' = uv - \int u'v$"). Функция не слишком удачно реализована — она не отличает определенный интеграл от неопределенного и всегда возвращает неопределенный, поэтому наборы операторов

```
w:integrate(f(x)*cos(x),x)$
byparts(w,x,f(x),cos(x));
```

и

```
w:integrate(f(x)*cos(x),x,a,b)$
byparts(w,x,f(x),cos(x));
```

приведут к одной и той же выдаче

$$f(x)\,\sin(x) - \int \sin(x)\,\left(\frac{d}{dx}\,(f(x))\right)\,dx$$

Функция "integrate" может сообразить, что интеграл от производной произвольной функции есть сама функция

```
integrate(diff(u(x),x,2),x);
```

$$\frac{d}{dx}\ (u(x))$$

Но в более сложных случаях он не замечает полной производной

```
integrate(diff(u(x),x,2)*sin(u(x))
  +diff(u(x),x)^2*cos(u(x)),x);
```

$$\int\ (\sin(u(x))\ \frac{d^2}{dx^2}\ (u(x))$$

$$+\ (\cos(u(x))\ (\frac{d}{dx}\ (u(x)))^2\ )\ dx$$

- **Функция** antidiff

выполняет интегрирование выражений с произвольными функциями, перед ее первым вызовом следует загрузить пакет "antid"

```
load(antid)$
```

Разумеется, интегрирование выполняется только для случая полной производной

```
antidiff(diff(u(x),x,2)*sin(u(x))
  +diff(u(x),x)^2*cos(u(x)),x);
```

$$\sin(u(x))\ (\frac{d}{dx}\ (u(x)))$$

24. Ряды, паде-аппроксимация и цепные дроби

- **Функция** `taylor`

раскладывает функцию в ряд Тейлора. Результат вызова функции является особым выражением — "рядом", это выражение снабжается меткой "/T/" сразу после метки "%o". Аргументы функции таковы: выражение, которое будет разложено; переменная, по которой идет разложение; точка, в которой мы раскладываем; и порядок, до которого идет разложение:

```
(%i7) ta1:taylor(sin(x),x,0,3)
```

$$(\%o7)\ /T/\ x - \frac{x^3}{6} + \ .\ \ .\ \ .$$

```
(%i8) ta2:taylor(cos(x),x,0,6)
```

$$(\%o8)\ /T/\ 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \ .\ \ .\ \ .$$

Ряды можно складывать, вычитать, умножать и делить друг на друга, при этом точность разложения учитывается автоматически, например

```
(%i9) ta1/ta2;
```

$$(\%o9)\ /T/\ x + \frac{x^3}{3} + \ .\ \ .\ \ .$$

т.е. разложение идет до третьего порядка (точность первого ряда).

При этом фактически речь идет о ряде Лорана, т.е. допускается

```
taylor(sin(x)/x^3,x,0,5);
```

$$\frac{1}{x^2} - \frac{1}{6} + \frac{x^2}{120} - \frac{x^4}{5040} + \ .\ \ .\ \ .$$

Более того, существует экзотическая возможность

```
taylor(exp(1/x),[x,0,3,'asymp]);
```

$$1 + \frac{1}{x} + \frac{1}{2\,x^2} + \frac{1}{6\,x^3} + \ .\ \ .\ \ .$$

- **Функция** `pade`
аппроксимирует отрезок ряда Тейлора, содержащий слагаемые до N-го порядка включительно, дробно-рациональной функцией. Ее аргументы — ряд Тейлора, порядок числителя n, порядок знаменателя m. Разумеется, количество известных коэффициентов ряда Тейлора должно совпадать с общим количеством коэффициентов в дробно-рациональной функции минус один (поскольку числитель и знаменатель определены с точностью до общего множителя). Иными словами, $N + 1 = (n + 1) + (m + 1) - 1$.

```
ta1:taylor(log(1+x),x,0,6);
```

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \ldots$$

```
pade(ta1,3,3)
```

$$\left[\frac{11\,x^3 + 60\,x^2 + 60\,x}{3\,x^3 + 36\,x^2 + 90\,x + 60}\right]$$

```
pade(ta1,4,2)
```

$$\left[-\frac{x^4 - 12\,x^3 - 150\,x^2 - 180\,x}{72\,x^2 + 240\,x + 180}\right]$$

- **Функция** `cf`
Создает цепную дробь, аппроксимирующую данное выражение. Выражение должно состоять из целых чисел, квадратных корней целых чисел и знаков арифметических операций. Выдача имеет вид списка.

- **Переменная** `cflength`
определяет количество периодов цепной дроби. Изначально установлено значение 1.

- **Функция** `cfdisrep`
преобразует список (как правило выдачу функции "cf") в собственно цепную дробь.

```
cf(sqrt(3));
```

```
[1,1,2]
```

```
cfdisrep(%);
```

$$1+\frac{1}{1+\frac{1}{2}}$$

```
float(%-sqrt(3.0));
```

```
-0.065384140902211
```

```
cflength:2$
cf(sqrt(3));
```

```
[1,1,2,1,2]
```

```
cfdisrep(%);
```

$$1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{2}}}}$$

```
cflength:5$
cf(sqrt(3));
```

```
[1,1,2,1,2,1,2,1,2,1,2]
```

```
cfdisrep(%)$
float(%-sqrt(3.0));
```

```
-1.7707912940423398E-6
```

25. Преобразование Лапласа и вычеты

- **Функция** `laplace`
  реализует прямое преобразование Лапласа

```
laplace(exp(t),t,s);
```

$$\frac{1}{s - 1}$$

```
laplace(sin(t),t,s);
```

$$\frac{1}{s^2 + 1}$$

```
laplace(diff(x(t),t),t,s);
```

```
s laplace(x(t), t, s) - x(0)
```

  (последнее соотношение позволяет решать линейные дифференциальные уравнения).

- **Функция** `ilt`
  реализует обратное преобразование Лапласа

```
ilt(1/(s-1),s,t);
```

```
%e^t
```

```
ilt(laplace(x(t),t,s),s,t);
```

```
x(t)
```

  (последнее свойство позволяет получать явные ответы при решении линейных дифференциальных уравнений).

- **Функция** `residue`
  позволяет вычислять вычеты

```
residue(1/(s^2+a^2),s,%i*a);
```

$$-\frac{\%i}{2\ a}$$

26. Уравнения

Часть функций, которые позволяют решать уравнения, перечислена в разделе "Работа с float-числами". Это функции, которые выдают ответ в терминах `float`-чисел (`allroots`, `find_root`, `newton`, `mnewton`).

- **Функция** `realroots`

  функция, которая выдает действительные корни *полиномиального* уравнения с *действительными* коэффициентами. При этом функция работает вовсе не с `float`-числами, она выдает ответ в терминах рациональных чисел. Если коэффициенты в исходном уравнении являются `float`-числами, то они конвертируются в рациональные. Точность этой операции задается параметром "`ratepsilon`" (см. раздел "Работа с float-числами").

- **Переменная** `rootsepsilon`

  задает заказанную точность поиска корней для функции "`realroots`". По умолчанию задано довольно странное значение `1.0e-7`. Имейте в виду, что это точность поиска корней не исходного уравнения, а того уравнения, которое получается после перехода к рациональным коэффициентам. Так что точность решения исходного уравнения с `float`-коэффициентами будет зависеть еще и от параметра "`ratepsilon`".

- **Переменная** `multiplicities`

  после выполнения функции "realroots" содержит список кратностей корней.

```
realroots(x^6+1=x^2+x^4);
                          [x = -1, x = 1]
multiplicities;
                          [2, 2]
```

- **Функция** `nroots`

  функция, которая выдает количество действительных корней полиномиального уравнения с действительными коэффициентами, которые локализованы в указанном интервале

```
nroots(x^6+1=x^2+x^4,minf,inf);
                           4
nroots(x^6+1=x^2+x^4,0,inf);
                           2
```

- **Функция** `algsys`

  решает *полиномиальные* системы уравнений. Допускаются системы из одного уравнения с одной неизвестной. Кроме того, допускаются недоопределенные системы. Аргументы функции "`algsys`" — это список уравнений и список переменных, а ее выдача — это список решений. Поскольку каждое решение есть список значений каждой из переменных, то список решений — это двойной вложенный список.

  Для уравнений с нулевой правой частью эту нулевую правую часть можно опускать.

  Логика функции "`algsys`" довольно разветвленная, в зависимости от вида конкретной системы уравнений она может вызывать функции "`allroots`", "`realroots`", "`solve`". (Кстати, в некоторых ситуациях функция "`solve`", логика которой также весьма разветвленная, может, в свою очередь, вызывать функцию "`algsys`").

  Функция "`algsys`" не конвертирует float-числа, входящие в систему, в рациональные.

- **Переменная** `algepsilon`

  Должна задавать точность решения системы для функции "`algsys`". По умолчанию установлено значение 10^8, что означает 8 верных знаков. Повышение точности соответствует увеличению показателя степени.

  К сожалению, этот параметр не работает.

  Функция "`algsys`" игнорирует переменную "`algepsilon`".

- **Переменная** `%rnum_list`

  после вызова функции "`algsys`" содержит список неопределенных параметров, входящих в решение для недоопределенной системы. Имена этих параметров конструируются из префикса "`%r`" и целого числа, например "`%r7`".

```
(%i) algsys([x^2-3*x+2=0],[x]);
(%o)                    [[x = 1], [x = 2]]
(%i) algsys([x^2-3*y+2,x=y],[x,y]);
(%o)                    [[x = 2, y = 2], [x = 1, y = 1]]
(%i) algsys([x^2-y],[x,y]);
                                 2
(%o)                     [[x = %r13, y = %r13  ]]
(%i) %rnum_list;
(%o)                    [%r13]
```

- **Функция** `solve`

  решает уравнения и системы уравнений. Ее аргументы — список уравнений и список переменных, а выдача — список решений. При этом для уравнений с нулевой правой частью эту правую часть можно опускать.
  В отличие от функции "`algsys`", функция "`solve`" конвертирует float-числа, входящие в систему, в рациональные. Поэтому ее поведение может зависеть от параметра "`ratepsilon`" (см. раздел "Работа с float-числами").

```
solve([x^2-3*x+2=0],[x]);
                        [x = 1, x = 2]
solve([sin(x)-1/2],[x]);
                        'solve' is using arc-trig functions to
                                        get a solution.
                        Some solutions will be lost.
                                        %pi
                                  [x = ----]
                                         6
```

- **Переменная** `multiplicities`

  содержит список кратностей корней, найденных функцией "`solve`"

```
solve([x^4-3*x^3+2*x^2=0],[x]);
                        [x = 1, x = 2, x = 0]
multiplicities;
                        [1, 1, 2]
```

  Для систем уравнений решение — это двойной вложенный список (см. функцию "`algsys`"):

```
solve([x+y=4,x-y=2],[x,y]);
                        [[x = 3, y = 1]]
```

  (в данном случае решение одно).
  Для недоопределенных систем в решение входят неопределенные параметры вида "`%r4`", а переменная "`%rnum_list`" после вызова функции "`solve`" содержит их список (см. функцию "`algsys`"):

```
solve([x^2-y],[x,y]);
                                    2
                        [[x = %r5, y = %r5 ]]
%rnum_list;
                        [%r5]
```

Функция "`solve`" имеет довольно разветвленную логику. В зависимости от конкретного вида уравнения или системы она ведет себя очень по-разному и

может вызывать другие функции ("linsolve", "algsys", и.т.п.), которые предназначены для поиска решений в тех или иных частных случаях.
Следует также иметь в виду, что функция "solve" управляется довольно большим количеством переменных (флагов), которые меняют ее поведение. К сожалению, при реальной работе они почти бесполезны. Дело в том, что функция "solve" надежно работает главным образом для тех уравнений или систем, которые с очевидностью имеют решение. В этом случае флаги не нужны. В более нетривиальных случаях решение обычно получить невозможно, вне зависимости от значения флаговых переменных.
Чтобы продемонстрировать, насколько причудливым может быть поведение функции "solve", приведем несколько примеров:

```
solve([x^5+y=7,x=y],[x,y]);
                    [[x = 1.36861648832 %i + 0.5084694089,
                      y = 1.36861648832 %i + 0.5084694089],
                     [x = 0.5084694089 - 1.36861648832 %i,
                      y = 0.5084694089 - 1.36861648832 %i],
                     [x = 0.9241881109 %i - 1.21387633450,
                      y = 0.9241881109 %i - 1.21387633450],
                     [x = - 0.9241881109 %i - 1.21387633450,
                      y = - 0.9241881109 %i - 1.21387633450],
                     [x = 1.41081382385, y = 1.41081382385]]
solve(x^5+x-7);
                                 5
                     [0 = x  + x - 7]
solve(x^6+x-7.0000000123456789);
                    'rat' replaced -7.00000001234568
                                           by -7/1 = -7.0
                                 6
                        [0 = x  + x - 7]
solve([sin(x)-y,sin(x)+y=1],[x,y]);
                     []
```

- **Функция** `eliminate`

  исключает из системы уравнений указанные переменные. Оставшиеся уравнения приводятся к виду с нулевой правой частью, которая опускается. Функция "eliminate" конвертирует float-числа, входящие в систему, в рациональные.

```
eliminate([x+y+z=1,
           x-y+z=2,x+y-z=3],[z]);
                     [2 (y + x - 2), - 2 y - 1]
eliminate([x+y+z=1,
           x-y+z=2,x+y-z=3],[x,y]);
```

```
                              [- 2 (z + 1)]
eliminate([sin(x)-y=0,sin(x)+y=0.5],[y]);
                              'rat' replaced -0.5 by -1/2 = -0.5
                                        [1 - 4 sin(x)]
```

Однако не следует ждать от функции "eliminate" слишком многого:

```
eliminate([sin(x)-sin(y)=0,sin(x)+sin(y)=1/2],[y]);
                               [1]
```

27. Дифференциальные уравнения

- **Функция** `ode2`

  решает дифференциальные уравнения первого и второго порядков. Ее аргументы — само дифференциальное уравнение в форме с "замороженной" производной (т.е. с производной, вычисление которой запрещено с помощью одиночной кавычки: "`'diff(y,x)`"), функция и переменная. Неопределенные константы для уравнений первого порядка пишутся как "`%c`", а для уравнений второго порядка — как "`%k1`", "`%k2`".

- **Переменная** `method`

  после работы функции "`ode2`" содержит указание на тип уравнения.

  Функция распознает линейные уравнения первого порядка

  ```
  ode2('diff(y,x)=2*y+exp(x),y,x);
  ```

  $$y = (\%c - \%e^{-x})\ \%e^{2\,x}$$

  ```
  method;
  ```

  ```
  linear
  ```

  Функция распознает уравнения первого порядка с разделяющимися переменными

  ```
  ode2('diff(y,x)=(x^2+1)*y^4,y,x);
  ```

  $$-\frac{1}{3\,y^3} = \frac{x^3 + 3\,x}{3} + \%c$$

  ```
  method;
  ```

  ```
  separable
  ```

  Функция распознает точно интегрируемые уравнения первого порядка

  ```
  ode2('diff(y,x)=(x^2+3*y^2)/(2*x*y),y,x);
  ```

  $$\frac{y^2 + x^2}{x^3} = \%c$$

  ```
  method;
  ```

  ```
  exact
  ```

  Для уравнений этого типа вводится еще и

- **Переменная intfactor**

  после работы функции "ode2" для уравнений типа "exact" содержит интегрирующий множитель

```
intfactor;
```

$$\frac{1}{x^4}$$

  Функция распознает линейные неоднородные уравнения второго порядка

```
ode2('diff(y,x,2)-3*'diff(y,x)+
        2*y=4*exp(3*x),y,x);
```

$$y = 2\%e^{3\,x} + \%k1\ \%e^{2x} + \%k2\ \%e^{x}$$

```
method;
```

```
variationofparameters
```

  Для уравнений этого типа вводится еще и

- **Переменная yp**

  после работы функции "ode2" для уравнений типа "variationofparameters" содержит частное решение

```
yp;
```

$$2\ \%e^{3\,x}$$

- **Функция ic1**

  позволяет учесть начальное условие в решениях дифференциальных уравнений первого порядка, ее аргументы — решение, значение "x" в виде уравнения и соответствующее значение "y" тоже в виде уравнения.

```
ode2('diff(y,x)=2*y+exp(x),y,x);
```

$$y = (\%c - \%e^{-x})\ \%e^{2\,x}$$

```
ic1(%,x=0,y=1);
```

$$y = 2\ \%e^{2\,x} - \%e^{x}$$

- **Функция ic2**

  позволяет учесть начальные условия в решениях дифференциальных уравнений второго порядка, ее аргументы — решение, значение "x" в виде уравнения и соответствующие значения "y" и "замороженной" производной "dy/dx" (" 'diff(y,x) ") тоже в виде уравнения.

- **Функция** bc2

  позволяет учесть краевые условия в решениях дифференциальных уравнений второго порядка, ее аргументы — решение, значение "x" в первой точке в виде уравнения и соответствующее значение "y", значение "x" во второй точке и соответствующее значение "y" тоже в виде уравнений.

```
re1:ode2('diff(y,x,2)-3*'diff(y,x)+
            2*y=4*exp(3*x),y,x);
                        3 x      2 x      x
                  y = 2 %e    + %k1 %e    + %k2 %e
ic2(re1,x=0,y=4,'diff(y,x)=9);
                        3 x      2 x      x
                  y = 2 %e    + %e    + %e
bc2(re1,x=0,y=4,x=1,
            y=exp(1)+exp(2)+2*exp(3) );
                        3 x      2 x      x
                  y = 2 %e    + %e    + %e
```

Совершенно по-другому организована альтернативная функция, которая также умеет решать дифференциальные уравнения, и, кроме того, системы дифференциальных уравнений.

Она существенным образом использует свойство функций, которое называется "atvalue".

- **Функция** atvalue

  позволяет задать значение функции и ее производных при некоторых значениях аргументов.

```
atvalue(x(t),t=0,5);
                          5
atvalue(diff(x(t),t),t=0,55);
                          55
atvalue(diff(x(t),t),t=1,77);
                          77
atvalue(f(a,b),[a=0,b=1],555);
                          555
atvalue(diff(f(a,b),b),
        [a=1,b=0],777);
                          777
```

Эта информация является свойством функций "x(t)" и "f(a,b)".

- **Функция** properties

  печатает свойства переменной

  properties(x);

  [atvalue]

- **Функция** printprops

  печатает информацию о заданном свойстве переменной

  printprops(x,atvalue);

  $$\frac{d}{d@1}\,(x(@1))\,\Big|_{@1=1} = 77$$

  $$\frac{d}{d@1}\,(x(@1))\,\Big|_{@1=0} = 55$$

  x(0)=5

  printprops(f,atvalue);

  $$\frac{d}{d@2}\,(f(@1,@2))\,\Big|_{@1=1,\ @2=0} = 777$$

  f(0,1)=555

- **Функция** remove

  отменяет указанное свойство переменной

  remove(x,atvalue);

  done

  properties(x);

  [ ]

- **Функция** at

  вычисляет значение выражения в заданной точке с учетом свойства "atvalue".

  *at(x(t)+10*diff(x(t),t), t=0);*

  555

  at(f(a,b) + diff(f(a,b),b) ,
  [a=1,b=0]);

  f(1, 0) + 777

- **Функция** desolve
  решает дифференциальные уравнения и системы дифференциальных уравнений. Ее аргументы — список уравнений, в которых функции записаны явно ("y(t)") и список неизвестных функций (также в виде "y(t)").

```
re2:desolve( [diff(y(t),t)=
          2*y(t)+%e^t], [y(t)]);
                        y(t) = (y(0) + 1) %e^(2 t) - %e^t
atvalue(y(t),t=0,1)$
ev(re2,at);
                        y(t) = 2 %e^(2 t) - %e^t
desolve( [diff(y(t),t)=
          2*y(t)+%e^t], [y(t)] );
                        y(t) = 2 %e^(2 t) - %e^t
atvalue(x(t),t=0,2)$
atvalue(y(t),t=0,0)$
desolve( [diff(x(t),t)=y(t),
    diff(y(t),t)=x(t)], [x(t),y(t)]);
                        [x(t) = %e^t + %e^-t, y(t) = %e^t - %e^-t]
desolve( [diff(x(t),t,2)
    -3*diff(x(t),t)+2*x(t)],[x(t)]);
                        x(t) = %e^(2 t) ((d/dt (x(t)))|t = 0 - x(0))

                                + %e^t (2 x(0) - (d/dt (x(t)))|t = 0)
```

28. Специальные функции

В MAXIM'е определены следующие специальные функции:

- Функции Эйри
  `airy_ai(x)` `airy_bi(x)`
  и их производные
  `airy_dai(x)` `airy_dbi(x)`

- Цилиндрические функции Бесселя, Неймана, Инфельда и Макдональда индекса m
  `bessel_j(m,x)` `bessel_y(m,x)`
  `bessel_i(m,x)` `bessel_k(m,x)`

- **Переменная** `besselexpand`
  определяет, будут ли цилиндрические функции полуцелого индекса заменяться на соответствующие выражения, составленные из "элементарных" функций. По умолчанию установлено значение "`false`":

```
bessel_j(1/2,x);
```

$$\texttt{bessel\_j}\left(\frac{1}{2},\ \texttt{x}\right)$$

```
besselexpand:true$
bessel_j(1/2,x);
```

$$\frac{\texttt{sqrt(2) sin(x)}}{\texttt{sqrt(\%pi) sqrt(x)}}$$

- Гамма-функция, бета-функция и пси-функция (логарифмическая производная гамма-функции)
  `gamma(x)` `beta(x,y)` `psi[m](x)`
  Здесь "`psi[m](x)`" — это m-ая производная функции $\psi(x) = \Gamma'(x)/\Gamma(x)$. При этом сама функция $\psi(x)$ — это "`psi[0](x)`".

- Большинство стандартных семейств ортогональных полиномов определены в пакете "`orthopoly`".
  Так что после вызова

```
load(orthopoly)$
```

  станут известными следующие функции:
  функции Лежандра 1-го и 2-го рода индекса "`n`"
  `legendre_p(n,x)` `legendre_q (n, x)`
  присоединенные функции Лежандра 1-го и 2-го рода индексов "`n,m`"
  `assoc_legendre_p(n,m,x)` `assoc_legendre_q(n,m,x)`

полиномы Чебышева 1-го и 2-го рода индекса "n"

```
chebyshev_t(n,x)        chebyshev_u(n,x)
```

полиномы Лягерра индекса "n" и обобщенные полиномы Лягерра индексов "n,a"

```
laguerre(n,x)        gen_laguerre(n,a,x)
```

полиномы Эрмита индекса "n"

```
hermite(n,x)
```

полиномы Якоби индексов "n,a,b"

```
jacobi_p(n,a,b,x)
```

полиномы Гегенбауэра индексов "n,a"

```
ultraspherical(n,a,x);
```

Кроме того, определены две "справочные" функции.

- **Функция** `orthopoly_recur`
  печатает рекурсивную формулу для полиномов. Ее первый аргумент — имя функции, вычисляющей полиномы, второй — список, составленный из имен индексов и имени переменной. Порядок имен такой же, как при вызове функции.

```
orthopoly_recur(chebyshev_t,[k,z]);
                        T    (z) = 2 T (z) z - T    (z)
                         k+1        k       k-1
```

- **Функция** `orthopoly_weight`
  печатает список, составленный из веса, нижней границы интервала и верхней границы интервала, на котором определены полиномы. Ее первый аргумент — имя функции, вычисляющей полиномы, второй — список, составленный из имен индексов и имени переменной. Порядок имен такой же, как при вызове функции.

```
orthopoly_weight(chebyshev_t,[n,x]);
                                    1
                          [-------------------, - 1, 1]
                                     2
                           sqrt(1 - x )
```

■ Семейство эллиптических функций.
Определены обычные эллиптические функции Якоби:

```
jacobi_sn(x,m)        jacobi_cn(x,m)        jacobi_dn(x,m)
```

Кроме того, определены явно избыточные функции:

```
jacobi_ns(x,m)        jacobi_nc(x,m)        jacobi_nd(x,m)
```

$ns(x,m) = 1/sn(x,m)$, $nc(x,m) = 1/cn(x,m)$, $nd(x,m) = 1/dn(x,m)$

```
jacobi_sc(x,m)        jacobi_sd(x,m)
```

$sc(x,m) = sn(x,m)/cn(x,m)$, $sd(x,m) = sn(x,m)/dn(x,m)$

`jacobi_cs(x,m)` `jacobi_cd(x,m)`

$cs(x,m) = cn(x,m)/sn(x,m)$, $cd(x,m) = cn(x,m)/dn(x,m)$

`jacobi_ds(x,m)` `jacobi_dc(x,m)`

$ds(x,m) = dn(x,m)/sn(x,m)$, $dc(x,m) = dn(x,m)/cn(x,m)$

Кроме того, определены функции, обратные ко всем перечисленным

`inverse_jacobi_sn(x,m)` `inverse_jacobi_cn(x,m)`
`inverse_jacobi_dn(x,m)`
`inverse_jacobi_ns(x,m)` `inverse_jacobi_nc(x,m)`
`inverse_jacobi_nd(x,m)`
`inverse_jacobi_sc(x,m)` `inverse_jacobi_sd(x,m)`
`inverse_jacobi_cs(x,m)` `inverse_jacobi_cd(x,m)`
`inverse_jacobi_ds(x,m)` `inverse_jacobi_dc(x,m)`

Кроме того, определены эллиптические интегралы

`elliptic_kc(m)` `elliptic_ec(m)`

$$kc(m) = \int_0^{\pi/2} \frac{1}{\sqrt{1 - m\sin^2 x}}\, dx \qquad ec(m) = \int_0^{\pi/2} \sqrt{1 - m\sin^2 x}\, dx$$

Наконец, определены неполные эллиптические функции

`elliptic_f(x,m)` `elliptic_e(x,m)`

$$f(\varphi,m) = \int_0^{\varphi} \frac{1}{\sqrt{1 - m\sin^2 x}}\, dx \qquad e(\varphi,m) = \int_0^{\varphi} \sqrt{1 - m\sin^2 x}\, dx$$

`elliptic_eu(x,m)` `elliptic_pi(n,x,m)`

$$eu(x,m) = \int_0^{sn(x,m)} \frac{\sqrt{1-mt^2}}{\sqrt{1-t^2}}\, dt$$

$$pi(n,\varphi,m) = \int_0^{\varphi} \frac{1}{\sqrt{1 - n\sin^2 x}\sqrt{1 - m\sin^2 x}}\, dx$$

29. Транслятор и компилятор в MAXIM'е

Определив ту или иную функцию, можно заметно ускорить ее выполнение, если ее оттранслировать или откомпилировать. Это происходит потому, что если Вы не оттранслировали и не откомпилировали определенную Вами функцию, то при каждом очередном ее вызове MAXIMA каждый раз заново выполняет те действия, которые входят в определение функции, т.е. фактически разбирает соответствующее выражение на уровне синтаксиса MAXIM'ы.

- **Функция** `translate`

транслирует функцию MAXIM'ы на язык LISP.

```
f(x):=1+x+x^2+x^3+x^4+x^5+x^6+x^7$
translate(f);
                                [f]
```

После этого функция (как правило) начинает считаться быстрее.

- **Функция** `compile`

сначала транслирует функцию MAXIM'ы на язык LISP, а затем компилирует эту функцию LISP'а до двоичных кодов и загружает их в память.

```
(%i9)      compile(f);
                      Compiling /tmp/gazonk_1636_0.lsp.
                      End of Pass 1.
                      End of Pass 2.
                      OPTIMIZE levels:  Safety=2,
                                           Space=3, Speed=3
                      Finished compiling /tmp/gazonk_1636_0.lsp.
                      (%o92)         [f]
```

После этого функция (как правило) начинает считаться еще быстрее, чем после трансляции.

Следует иметь в виду, что как при трансляции, так и при компиляции MAXIMA старается оптимизировать функцию по скорости, не заботясь об аккуратности. Поэтому при работе с большими по объему функциями могут возникнуть чудеса. В этом случае следует отказаться от трансляции или компиляции, либо переписать функцию.

Выигрыш во времени существенным образом зависит от типа машины, от вида функции, от того, декларирован ли тип функции и ее аргумента при определении функции, от типа аргумента, с которым вызывается функция.

Если предполагается использовать функцию только для работы с действительными числами (например для вычисления определенного интеграла с помощью

функции "romberg" или поиска корня с помощью функций "find_root" и "newton"), то обязательно следует декларировать тип аргумента и самой функции как "float". Это во много раз усилит эффект от трансляции или компиляции.

Для того, чтобы дать общее представление о влиянии трансляции и компиляции на скорость счета разных типов функций и разных типов аргумента, приведем табличку с временами исполнения функций на одной конкретной машине.

Были определены четыре разные функции, вычисляющие одно и то же выражение

```
f1(x):=1+x+x^2+x^3+x^4+x^5+x^6+x^7+x^8+x^9$
f2(x):=block([s],s:1,for i:1 thru 9 do s:s+x^i,s)$
f3(x):=block([],mode_declare([function(f),x],float),
    1+x+x^2+x^3+x^4+x^5+x^6+x^7+x^8+x^9)$
f4(x):=block([s],mode_declare([function(f),x,s],float),
    s:1, for i:1 thru 9 do s:s+x^i,s)$
```

Форму записи двух последних функций ("f3" и "f4") следует воспринимать аксиоматически.

Далее каждая из этих функций вызывалась с аналитическим аргументом "s" (кроме "f3" и "f4", для которых аналитический аргумент невозможен), целочисленным аргументом "7" и вещественным аргументом "0.3".

После этого все четыре функции были оттранслированы ([**t**]) и эксперимент был повторен.

Наконец, все четыре функции были откомпилированы ([**c**]) и эксперимент снова был повторен.

Ниже приведены соответствующие (условные) времена исполнения функций.

	s	7	0.3	[**t**] s	[**t**] 7	[**t**] 0.3	[**c**] s	[**c**] 7	[**c**] 0.3
f1	8.75	4.75	4.22	7.32	3.52	3.04	6.98	3.25	2.75
f2	17.73	12.44	12.03	9.11	4.80	4.09	7.83	3.20	2.67
f3		5.39	4.87		2.72	2.17		1.67	1.23
f4		13.30	12.75		3.45	2.91		2.47	1.94

Приведенные цифры дают весьма богатый материал для анализа.

Во-первых, хорошо заметно, насколько трудоемкой оказывается процедура упрощения. Функция "f1" задана в виде явной формулы, и ее вычисление сводится к однократной подстановке, в то время как функция "f2" требует упрощения на каждом обороте цикла, что фатально сказывается на скорости

ее вычисления. Зато и эффект от трансляции или компиляции для неявных функций типа "`f2`" оказывается гораздо заметнее.
Во-вторых, заметно, что экономия времени для "более простых" числовых аргументов (по сравнению с символьными аргументами) оказывается не такой уж радикальной, хотя и довольно существенной. Это связано с тем, что числа все равно рассматриваются как элемент аналитического выражения, хотя упрощение аналитического выражения, составленного исключительно из чисел, идет быстрее.
В-третьих, заметно, что для функций, которые декларированы как вещественные функции от вещественных аргументов, вычисление от целочисленного аргумента идет медленнее, чем от вещественного аргумента.
В-четвертых, очень хорошо видно, что если Вы не выполняете компиляцию для вещественной функции, то Вы рискуете замедлить ее вычисление более чем в 6 раз.

Содержание