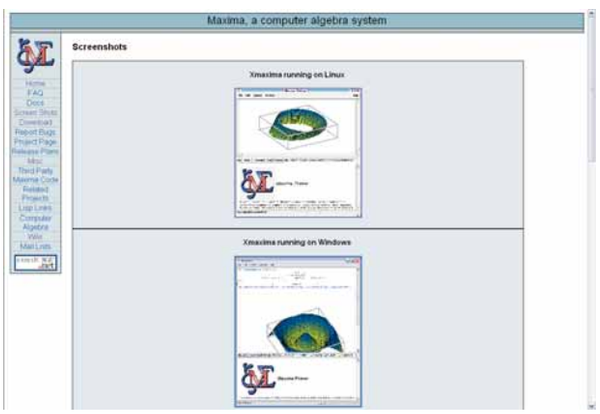
Работа с файлами

ЧАСТЬ 6 Завершая этот длинный цикл статей, **Тихон Тарнавский** коснется вопросов работы с файлами, базой данных фактов и напишет собственную функцию символьного дифференцирования!



В прошлый раз мы остановились на возможностях программирования, предназначенных для написания собственных функций и модулей к *Maxima* – и теперь для их полноценного использования рассмотрим несколько инструментов работы с файлами, позволяющих сохранять и загружать эти функции и модули на диск и с диска. Далее речь пойдет о наложении определенных условий на неизвестные и значения функций. Напоследок познакомимся с функциями по работе... с функциями: это один из очень мощных инструментов, позаимствованных из функционального программирования; а также разберем несколько более крупных учебных примеров, использующих многое из изученного нами во всех статьях цикла.

Учимся читать и писать

Среди средств для операций с файлами функции с наиболее очевидными именами – save и load – имеют, вопреки привычной для *Maxima* логичности всех названий, различный контекст. Первая предназначена для выгрузки *Maxima*-выражений в виде исходных кодов на Lisp, так что если вы не знаток Lisp (да и реализации внутренних механизмов *Maxima*), то эта функция представляет лишь чисто академический интерес. Посему подробнее мы займемся другими функциями – для обработки так называемых пакетных (batch) файлов, хранящих выражения уже в синтаксисе самой *Maxima*. А поскольку в виде таких файлов поставляется немалое количество функционала *Maxima*, то начнем с загрузки. И вот о второй из очевидно-именуемых функций здесь уже будет рассказано.

Функции чтения файлов с выражениями *Maxima* существует три: demo(имя-файла), batch(имя-файла) и batchload(имя-файла). Первая предназначена для загрузки так называемых демо-файлов, задуманных, как и явствует из названия, для демонстрационных примеров. Она загружает демо-файл и выполняет его в пошаговом режиме, ожидая нажатия Enter после выполнения каждой строки. В составе *Maxima* поставляется значительное количество демо-файлов; упоминания о них можно найти в документации, а сами файлы несложно обнаружить среди содержимого пакета *maxima-share* (либо, в случае отсутствия такового в вашем дистрибутиве, просто *maxima*) по их расширению – **.dem**.

Функция batch() загружает *Maxima*-файл с расширением **.mac** или **.mc** (от первоначального названия программы – *Macsyma*) и выполняет содержащиеся в нем выражения так, как если бы они вводились прямо в текущей сессии, то есть с отображением результата каждого выражения и назначением меток %iN, %oN. Функция batchload(), напротив, подгружает пакетный файл «молча»: все назначенные в нем функции и переменные становятся доступны, но результаты не видны, и весь хранимый ввод-вывод, включая значения символов % и _ и результаты, возвращаемые функцией %th(), остается тем же, что и до вызова.

Функции batch() и batchload() используют при поиске файлов для загрузки путь (точнее сказать, шаблон, потому как в нем содержатся не только имена каталогов, но и допустимые расширения файлов), который хранится в переменной file_search_maxima. По умолчанию эта переменная содержит все каталоги, в которые устанавливаются **.mac**-файлы из пакетов *Maxima*, а также **~/.maxima**, предназначенный для пользовательских файлов. Для других функций загрузки существуют отдельные переменные: file_search_lisp и file_search_demo, смысл которых понятен из их названий.

Ну и под конец здесь нужно вспомнить о вышеназванной функции load. Она, фактически, является оберткой над двумя функциями: уже описанной выше batchload() и loadfile(), вторая, совершенно аналогично первой, загружает файл, но уже не с выражениями *Maxima*,

Месяц назад Мы научились строить графики и писать программы.

и фактами

а с исходным кодом Lisp, то есть является парной к функции save(). Функцию load() можно, в принципе, использовать вместо batchload(): путь file_search_maxima задан в ней раньше, чем file_search_lisp, так что в случае неоднозначности она будет загружать файлы *Maxima*; а кроме того, так короче.

Некоторый функционал *Maxima* содержится в неподгружаемых автоматически внешних файлах, которые, соответственно, нужно принудительно загрузить перед использованием:

```
(%i1) A: matrix([a11, a12, a13], [a21, a22, a23])
(%o1) ( a11  a12  a13 )
      ( a21  a22  a23 )
(%i2) matrix_size(A)
(%o2) matrix_size( ( a11  a12  a13 ) )
                   ( a21  a22  a23 )
(%i3) load(linearalgebra)
(%o3) /usr/share/maxima/5.10.0/share/linearalgebra/linearalgebra.mac
(%i4) '%i2
(%o4) [2, 3]
```

Помимо ручной загрузки нужного файла, можно также настроить *Maxima* на автоматическую подгрузку в случае вызова заданной функции. Делается это так: setup_autoload(имя-файла, имена-функций); нужные функции здесь перечисляются через запятую прямо после имени файла. Удобнее, конечно, будет не вызывать функцию setup_autoload() вручную (так в ней и толку немного), а настроить *Maxima* на автоматический ее запуск при старте программы. Файл, который, при его наличии, вызывается при каждом запуске *Maxima*, называется **maxima-init.mac** и самое логичное для него местоположение – все тот же каталог **~/.maxima**. Конечно, он может содержать не только вызовы функции setup_autoload(), а любые выражения *Maxima*, которые вы хотите выполнять при каждом ее запуске. Использование этой функции может сделать вашу работу с *Maxima* намного более удобной в том случае, если вы часто используете некоторые из внешних функций *Maxima* или функции, вами же и написанные.

Для полноценного чтения файлов всего сказанного уже вполне достаточно, теперь перейдем к записи в них. Тут нас в первую очередь интересует функция stringout(), которая позволяет выгружать в файл любые выражения и функции *Maxima* в точно таком виде, в каком их загружают функции demo(), batch() и batchload(). С ее помощью можно писать выражения, которые вы хотите иметь во внешнем модуле, находясь непосредственно в интерфейсе *Maxima*, с последующей записью в этот самый модуль. Для выгрузки функций в один из стандартных каталогов *Maxima* (самым логичным вариантом будет, пожалуй, упомянутый выше **~/.maxima**) имя файла во всех вариантах вызова функции stringout() нужно задавать с полным путем; в случае задания имени без пути файл будет создан в текущем каталоге, то есть в том, откуда производился запуск *Maxima*.

Здесь, чтобы было интереснее и не приходилось писать в файлы всякую ерунду, немного прервемся и создадим пару небольших функций.

```
(%i1) primes(n) :=
        if integerp(n) then
         if n <= 2 then [] else
          append(primes(prev_prime(n)), [prev_prime(n)])$
```

Эта функция возвращает список всех простых чисел, меньших чем заданное целое число. Сначала мы проверяем, является ли аргумент целым числом и делаем это простейшим образом: в случае невыполнения условия оператор if, напомню, вернет *false*. Генерируется список тоже самым простым и коротким в реализации способом – рекурсией. (примечание для людей, далеких от программирования: рекурсивная функция – это функция, вызывающая саму себя; чаще всего такие функции строятся по принципу индукции). Здесь используется функция *Maxima* по имени prev_prime(), которая возвращает простое число, предшествующее заданному целому.

У рекурсии, при всей ее простоте реализации, есть неоспоримый минус – только один, но весьма существенный: чрезвычайная требовательность к объему памяти. Поэтому, для обеспечения возможности получать последовательности из больших простых чисел, добавим в наш учебный пример еще одну функцию:

```
(%i2) primesbetween(n, m) :=
        if integerp(n) and integerp(m) then
         if m <= 2 or prev_prime(m) <= n then [] else
          append(primesbetween(n, prev_prime(m)),
                 [prev_prime(m)])$
```

Смысл, думаю, понятен по аналогии с предыдущей: теперь мы еще и ограничили возвращаемый список снизу.

Теперь, когда у нас уже есть primesbetween(), первую функцию можно написать по «принципу чайника» – сведя задачу к предыдущей:

```
(%i3) primes1(n) := primesbetween(1, n)$
```

Теперь вернемся к stringout(). Эта функция, как и многие другие, может принимать несколько различных вариантов аргументов, первым из которых всегда выступает имя файла для записи, а остальные отвечают за то, что же именно будет туда записано. В варианте stringout(имя-файла, [начало, конец]) записаны будут ячейки ввода с номерами от «начала» до «конца» включительно:

»

```
(%i4) stringout(".maxima/primes.mac", [1,2])
(%o4) /home/t/.maxima/primes.mac
```

```
$ cat .maxima/primes.mac
primes(n):=if integerp(n) then (if n <= 2 then [] else append(primes(prev_
prime(n)),[prev_prime(n)]));
primesbetween(n,m):=if integerp(n) and integerp(m) then (if m <= 2 or
prev_prime(m) <= n then [] else append(primesbetween(n,prev_
prime(m)),[prev_prime(m)]));
```

Как видите, по умолчанию вывод получается не слишком красивым, поэтому сразу рассмотрим один ключ, влияющий на его формат. Долго рассказывать о нем смысла нет, лучше показать на примере:

```
(%i5) grind: true$ stringout(".maxima/primes.mac", [1,2])
(%o6) /home/t/.maxima/primes.mac
```

```
$ cat .maxima/primes.mac

primes(x):=if integerp(x)
      then (if x <= 2 then []
            else append(primes(prev_prime(x)),[prev_prime(x)]));
primesbetween(n,m):=if integerp(n) and integerp(m)
          then (if m <= 2 or prev_prime(m) <= n then []
                else append(primesbetween(n,prev_prime(m)),
                     [prev_prime(m)]));
```

Представления о правилах отступов у создателей этой опции несколько специфичные, но тем не менее, результат стал намного читабельнее. Так что, если вы планируете сохранять выражения *Maxima* не только для того, чтобы потом загружать их обратно, а желаете редактировать созданные файлы, я рекомендую вам прописать grind:true глобально в файле **~/.maxima/maxima-init.mac**.

Идем дальше. С помощью ключевого слова input можно выгрузить в файл все ячейки ввода разом:

```
(%i7) (N: [random(100000)],
       for i thru 9 do
        N: append(N, [N[i] + random(100000)]),
       N)
(%o7) [49900, 61971, 153219, 244360, 290427, 347723, 396481, 465378,
522906, 568462]
(%i8) (P: [],
       for i thru 10 do
        P: append(P, primesbetween(N[i] - 50, N[i])),
       P)
(%o8) [49853, 49871, 49877, 49891, 61927, 61933, 61949, 61961, 61967,
153191, 244313, 244333, 244339, 244351, 244357, 290383, 290393,
290399, 290419, 347707, 347717, 396437, 396443, 396449, 396479,
465331, 465337, 465373, 522857, 522871, 522881, 522883, 522887,
568433, 568439, 568441, 568453]
(%i9) stringout("primes-sample.mac", input)
(%o9) /home/t/primes-sample.mac
```

```
$ cat primes-sample.mac

primes(n):=if integerp(n)
      then (if n <= 2 then []
            else append(primes(prev_prime(n)),[prev_prime(n)]));
primesbetween(n,m):=if integerp(n) and integerp(m)
          then (if m <= 2 or prev_prime(m) <= n then []
                else append(primesbetween(n,prev_prime(m)),
                     [prev_prime(m)]));
primes1(n):=primesbetween(1,n);
stringout(".maxima/primes.mac",[1,2]);
grind:true;
stringout(".maxima/primes.mac",[1,2]);
(N:[random(100000)],for i thru 9 do N:append(N,[N[i]+random(100000)]
),N);
(P:[],for i thru 10 do P:append(P,primesbetween(N[i]-50,N[i])),P);
```

Кроме input, есть еще два ключевых слова: functions и values. Первое позволяет записать определения всех функций, второе – присвоение всем символам выражений их текущих значений:

```
(%i10) stringout(".maxima/primes.mac", functions)
(%o10) /home/t/.maxima/primes.mac
(%i11) stringout("primes-sample.mac", functions, values)
(%o11) /home/t/primes-sample.mac
```

```
$ cat .maxima/primes.mac

primes(n):=if integerp(n)
      then (if n <= 2 then []
            else append(primes(prev_prime(n)),[prev_prime(n)]));
primesbetween(n,m):=if integerp(n) and integerp(m)
          then (if m <= 2 or prev_prime(m) <= n then []
                else append(primesbetween(n,prev_prime(m)),
                     [prev_prime(m)]));
primes1(n):=primesbetween(1,n);


$ cat primes-sample.mac

primes(n):=if integerp(n)
      then (if n <= 2 then []
            else append(primes(prev_prime(n)),[prev_prime(n)]));
primesbetween(n,m):=if integerp(n) and integerp(m)
          then (if m <= 2 or prev_prime(m) <= n then []
                else append(primesbetween(n,prev_prime(m)),
                     [prev_prime(m)]));
primes1(n):=primesbetween(1,n);
N:[49900,61971,153219,244360,290427,347723,396481,465378,522906
,568462];
P:[49853,49871,49877,49891,61927,61933,61949,61961,61967,153191,
244313,244333,
 244339,244351,244357,290383,290393,290399,290419,347707,34771
7,396437,
 396443,396449,396479,465331,465337,465373,522857,522871,52288
1,522883,
 522887,568433,568439,568441,568453];
```

И кроме всего этого, функцию stringout() можно вызвать с непосредственным перечислением в аргументах конкретных выражений. В этом случае, надо заметить, будут сохраняться не ячейки, содержащие заданные выражения, а именно сами выражения. То есть, если перечислить символ, для которого задано значение, то в файл будет записано только это значение. С именами функций, заданными непосредственно, дело обстоит не лучше: функцию таким образом задать, по сути, вообще нельзя: если просто написать ее имя, то вместо функции будет подставлен одноименный символ (или его значение, если оно задано). Но из обеих ситуаций есть выход. Для функций – штатный: функция fundef, которая принимает имя любой пользовательской функции и возвращает ее определение в точности в таком же виде, в каком оно было введено (или могло бы быть введено) в «командной строке» *Maxima*, с точностью до пробелов:

```
(%i12) stringout(".maxima/primesbetween.mac",
        fundef(primesbetween))
(%o12) /home/t/.maxima/primesbetween.mac
(%i13) stringout(".maxima/primes1.mac",
        fundef(primes), fundef(primes1))
(%o13) /home/t/.maxima/primes1.mac
```

```
$ cat .maxima/primesbetween.mac

primesbetween(n,m):=if integerp(n) and integerp(m)
            then (if m <= 2 or prev_prime(m) <= n then []
                        else append(primesbetween(n,prev_prime(m)),
                                    [prev_prime(m)]));


$ cat .maxima/primes1.mac

primes(n):=if integerp(n)
      then (if n <= 2 then []
                else append(primes(prev_prime(n)),[prev_prime(n)]));
primes1(n):=primesbetween(1,n);
```

А для символов можно использовать небольшую хитрость: блокировать вычисление переданного выражения, а в нем написать сначала сам символ, а потом через двоеточие – его же, предварив знаком принудительного вычисления (два апострофа):

```
(%i14) stringout("random-primes.mac", '(P:''P))

(%o14) /home/t/random-primes.mac
```

```
t:~$ cat random-primes.mac

P:[49853,49871,49877,49891,61927,61933,61949,61961,61967,153191,
244313,244333,
 244339,244351,244357,290383,290393,290399,290419,347707,34771
7,396437,
 396443,396449,396479,465331,465337,465373,522857,522871,52288
1,522883,
 522887,568433,568439,568441,568453];
```

В довершение темы работы с файлами стоит обратить внимание еще на один момент: при загрузке файлы в текущем каталоге не ищутся – и как раз для него надо задавать путь, причем полный, а не через **./имя-файла**:

```
(%i1) load("primes-sample")

Could not find 'primes-sample' using paths in
file_search_maxima,file_search_lisp (combined values:
[/home/t/.maxima/###.mac,mc,
/usr/share/maxima/5.10.0/share/###.mac,mc,
/usr/share/maxima/5.10.0/share/affine,algebra,algebra/charsets
/home/t/.maxima/###.o,lisp,lsp,
/usr/share/maxima/5.10.0/share/###.o,lisp,lsp,
/usr/share/maxima/5.10.0/share/affine,algebra,algebra/charsets
/usr/share/maxima/5.10.0/src/###.o,lisp,lsp] )
 -- an error.  Quitting.  To debug this try
debugmode(true);

(%i2) load(primes)

(%o2) /home/t/.maxima/primes.mac

(%i3) load("/home/t/primes-sample")

(%o3) /home/t/primes-sample.mac

(%i4) load("./primes-sample")

Could not find './primes-sample' using paths in
file_search_maxima,file_search_lisp (combined values:
```

«Прослушайте объявление»

Теперь поговорим о функциях, позволяющих налагать определенные условия на выражения, которыми оперирует *Maxima*. Таких функций существует две, и достаточно разноплановых; но определенная связь между ними есть, так как все условия, заданные ими на данный момент, хранятся в общей «базе». Первая из этих функций называется declare (объявлять). С ее помощью можно объявлять весьма разнообразные факты о произвольных символах или выражениях; синтаксис ее весьма прост: declare(имя, факт) или declare(имя1, факт1, имя2, факт2, ...); факты задаются с помощью ключевых слов. Сами факты я бы разделил на три группы: «технические» факты *Maxima*, позволяющие использовать наделенный ими символ некоторым специальным образом при вводе выражений; факты о символах (атомарных выражениях); и факты о значениях функций. К первым относятся, к примеру, свойства evflag и evfun, о которых шла речь в описании функции ev; некоторые штатные функции обладают ими по умолчанию, а с помощью функции declare мы можем присвоить эти свойства любым другим, в том числе и пользовательским, функциям. Вторая группа фактов несет информацию о неизвестных; например, мы можем указать, что некоторая неизвестная является константой, или что ее значение – целое. И третья группа – примерно то же самое, но о функциях; примеры: четная функция (f(–x)=f(x)), аддитивная (f(x+y)=f(x)+f(y)) или целочисленная. Для краткости просто перечислим наиболее интересные из возможных фактов, сгруппировав соответственно трем упомянутым группам.

Технические факты

evfun

Позволяет применять функцию или переменную как опцию, то есть «выражение, имя-функции» вместо «имя-функции(выражение)» или «выражение, имя-переменной» вместо «имя-переменной:true; выражение». Подробнее см. в LXF82.

bindtest

Запрещает использовать символ в выражениях до присвоения ему значения. При таком использовании *Maxima* выдаст ошибку. Пример см. в документации.

feature

Делает заданное имя именем свойства (факта), что дает возможность использовать его точно так же, как все перечисленные здесь имена.

Факты о символах

constant

Имя трактуется как константа.

scalar

Имя трактуется как скалярная величина. На это также влияет флаг assumescalar: если он равен *true*, то все неопределенные символы воспринимаются как скаляры. Тут есть небольшая коллизия: если верить документации, то по умолчанию assumescalar равен *false*, реально же в *Maxima 5.10.0* он равен *true*.

```
(%i1) v: [v1, v2, v3]$ assumescalar: false$
(%i3) Nv
(%o3) N [v1, v2, v3]
(%i4) declare(N, scalar); Nv
(%o4) done
(%o5) [N v1, N v2, N v3]
```

nonscalar

Имя трактуется как не-скалярная величина, то есть матрица или вектор.

integer, noninteger

Целое и нецелое число.

even, odd

Четное и нечетное целое число.

```
(%i1) declare(n, even)$
(%i2) askinteger(n)
(%o2) yes
```

Факты о функциях

rassociative

Объявляет функцию как «ассоциативную» по правому аргументу.

lassociative

Аналогично – по левому аргументу.

```
(%i1) declare(f, rassociative)$
(%i2) f(a, b, c, d); f(f(a, b), f(c, d))
(%o2) f(a, f(b, f(c, d)))
(%o3) f(a, f(b, f(c, d)))
(%i4) declare(f, lassociative)$
(%i5) f(a, b, c, d); f(f(a, b), f(c, d))
(%o5) f(f(f(a, b), c), d)
(%o6) f(f(f(a, b), c), d)
(%i7) f(a, f(b, f(c, d)))
(%o7) f(f(f(a, b), c), d)
```

nary

Объявляет «n-арную» функцию. Это и два предыдущих названия не совсем точны: n-арной правильно называть функцию от n аргументов, а лево- и право- ассоциативной – функции именно с односторонней ассоциативностью, то есть, для «лево-» f(f(a,b),c)≠f(a,b,c)≠f(a,f(b,c)). А в *Maxima* все три факта объявляют на самом деле полно-ассоциативную функцию от произвольного числа аргументов, а различаются только тем, как будут расставлены скобки по умолчанию.

```
(%i8) kill(f)$ declare(f, nary)$
(%i10) ''%i7
(%o10) f(a, b, c, d)
```

symmetric/commutative

Оба ключевых слова объявляют функцию как симметричную (коммутативную).

```
(%i1) declare(f, symmetric)$
(%i2) f(a, b) + f(b, a)
(%o2) 2 f(a, b)
```

antisymmetric

Объявляет функцию как антисимметричную.

```
(%i1) declare(f, antisymmetric)$
(%i2) f(a, b) + f(b, a)
(%o2) 0
```

outative

Константа выносится за знак функции.

```
(%i1) declare(f, outative, N, constant)$
(%i2) f(N x)
(%o2) N f(x)
```

Многие из фактов, которые можно устанавливать с помощью функции declare, сохраняются в «базе данных» фактов. Узнать текущее состояние этой базы можно с помощью функции facts(). Ее можно вызывать, либо передав в качестве единственного аргумента имя, список фактов по которому мы хотим получить, либо вообще без аргументов – тогда будут выданы все известные факты обо всех пользовательских именах. Удалить свойства позволяет функция remove(). Она, как и многие другие, имеет несколько вариантов вызова. Будучи вызвана как remove(имя, свойство) или remove(имя1, свойство1, имя2, свойство2, ...), она лишает каждое переданное имя одного соответствующего ему свойства. Можно также передавать ей списки имен и свойств: remove([имя1, имя2, ...], [свойство1, свойство2, ...]); тогда каждое имя из списка будет лишено всех перечисленных свойств. Пар списков тоже может быть более одной: remove(список-имен1, список-свойств1, список-имен2, список-свойств2, ...) – этот вызов аналогичен последовательным remove(список-имен1, список-свойств1); remove(список-имен2, список-свойств2); ... И последний интересующий нас вариант – remove(all, свойство) удаляет «свойство» со всех имен, у которых оно есть.

Вторая «условная» функция – это функция assume() (допускать, принимать). Здесь все проще: в качестве аргументов ей можно передавать в любом количестве самые обыкновенные равенства и неравенства. Напомню только, что задавать их нужно не в синтаксической, а в логической форме, то есть не «a=b», «a#b», а «equal(a,b)», «not equal(a,b)». Из логических операторов допускается также использование and (по сути assume(x>0 and x<1) это то же самое, что и assume(x>0, x<1)), но не or – база фактов не поддерживает информацию вида «или»; и речь не о синтаксисе, а именно о конструкциях, то есть выражения типа not(a>b and a<c) тоже недопустимы. Факты, добавленные assume(), также видны функции facts():

```
(%i1) declare(n, integer)$ assume(n > 10)$
(%i3) facts(n)
(%o3) [kind(n, integer), n > 10]
```

Ключевое слово kind используется только для отображения тех фактов из базы, которые добавлены с помощью declare().

Если факты, заданные функцией declare(), удаляются вызовом remove(), то для assume() есть своя «обратная» функция – forget(), которая также принимает любое количество условий (точно таких же как и assume()), либо в качестве отдельных аргументов, либо списком.

Общая база фактов используется этими двумя не очень похожими функциями неспроста: все, кому эти факты могут пригодиться, используют обе их разновидности, причем одновременно. Например, уже известный нам предикат is:

```
(%i1) declare(f,increasing)$
(%i2) assume(x>y)$
(%i3) is(f(x)>f(y))
(%o3) true
```

Еще один пример использования assume()/declare() – возможность избежать неопределенностей. Вы, возможно, помните, как в одном из примеров LXF84 в ответ на попытку посчитать некий интеграл Maxima задала нам вопрос о знаке входящего в него символа. Вот в таких ситуациях тоже может пригодиться assume, дабы предвосхитить расспросы:

```
(%i1) integrate(x^2*sqrt(a^2-x^2),x,0,a)
Is a positive, negative, or zero?p
                          4
(%o1)                    %pi a
                         ------
                           16
(%i2) assume(a<0)$
(%i3) "%i1
                          4
(%o3)                   - %pi a
                         ------
                           16
(%i1) limit(x^inf)
Is |x| - 1 positive, negative, or zero?n
Is x positive, negative, or zero?n
(%o1) 0
(%i2) assume(x>0,x<1)$
(%i3) limit(x^inf)
(%o3) 0
```

Вот мы и подошли к концу «теоретической» части. Надеюсь, функционала, рассмотренного на протяжении шести статей, будет достаточно для решения многих задач, а также для того, чтобы черпать дальнейшие сведения из документации – ведь мы уже изучили такие вещи, благодаря которым *Maxima* становится не просто «вычислялкой» отдельных небольших примеров, а настоящей «средой программирования с математическим уклоном», позволяющей создавать свои собственные математические «типы данных» – числовые системы, функционалы и прочая и прочая – и полноценные программные модули, которые могут использовать весь встроенный (или также собственноручно достроенный) функционал *Maxima*. Рассмотрим, напоследок, более серьезный учебный пример, в котором эти возможности можно будет лучше прочувствовать. Одна заявленная тема у нас пока осталась нераскрытой – функции для работы с функциями и «глубокой» обработки выражений. Но это настолько серьезный инструмент, что на маленьких примерах его рассматривать было бы бессмысленно, а потому мы поговорим о нем в приложении-практикуме. Удачи! LXF

Практикум Maxima

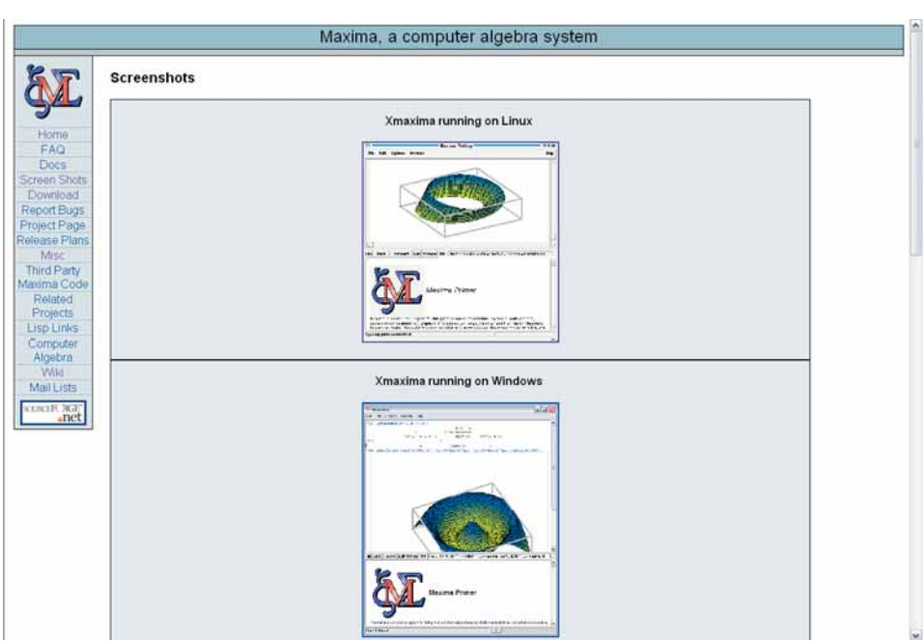
Сначала я хотел рассмотреть несколько отдельных практических примеров: и маленьких, и чуть побольше. Но потом мне подумалось, что один, но более серьезный пример будет значительно лучше: с одной стороны, его можно строить понемногу, отрабатывая отдельные приемы точно так же, как это было бы сделано и с меньшими примерами, а с другой – в результате все эти приемы переплетутся между собой во что-то объемное, и на этих переплетениях возникнет более цельное ощущение возможностей программы, чем на несвязанных маленьких кусочках. К тому же по ходу дела мы соорудим несколько небольших вспомогательных функций, а заодно, для дополнительной практики, и более расширенную версию одной из них, которая, вполне возможно, пригодится вам и в дальнейшем.

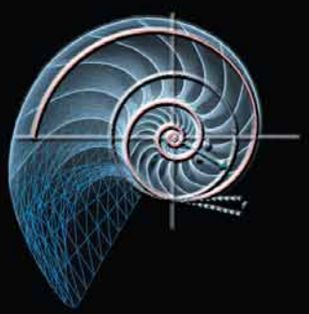
А писать мы будем настоящую функцию дифференцирования. Практически такую же, как встроенная diff(), только без вычисления полного дифференциала – чтобы не слишком сложно было «охватить» пониманием сразу весь пример. Ну а если будет интерес, то дописать вычисление полного дифференциала к этой же функции вы можете попробовать самостоятельно – после освоения возможностей, которые сейчас будут продемонстрированы, это будет уже несложно. Примеров применения по ходу создания функции я давать не буду. Если вы хотите смотреть на практические результаты, по мере добавления кода можно сохранять его в файле, скажем, **~/.maxima/deriv.mac** и выполнять в *Maxima* строку load(deriv)$ deriv(какое-нибудь-выражение);.

Я буду писать код постепенно и по ходу написания давать комментарии к последнему написанному участку. Комментировать буду, просто вставляя куски кода в текст. К слову: *Maxima* поддерживает комментарии в коде «в стиле Си», то есть комментарий начинается символами /*, а заканчивается */. Причем, в отличие от Си, допускаются вложенные комментарии: /* вот /* такие */ */.

Чтобы не повторять каждый раз весь код от самого начала, я буду сокращать его с помощью многоточия. Если вы будете проверять код по мере чтения, не забывайте о разделяющих запятых после последних строк предыдущих участков.

Полный текст практикума вы найдете на LXFDVD в разделе «Журнал».





Пишем свой diff()

БОНУС В этом приложении-практикуме Тихон Тарнавский покажет, как использовать Maxima для решения «настоящих» задач.

Сначала я хотел рассмотреть несколько отдельных практических примеров: и маленьких, и чуть побольше. Но потом мне подумалось, что один, но более серьезный пример будет значительно лучше: с одной стороны, его можно строить понемногу, отрабатывая отдельные приемы точно так же, как это было бы сделано и с меньшими примерами, а с другой – в результате все эти приемы переплетутся между собой во что-то объемное, и на этих переплетениях возникнет более цельное ощущение возможностей программы, чем на несвязанных маленьких кусочках. К тому же по ходу дела мы соорудим несколько небольших вспомогательных функций, а заодно, для дополнительной практики, и более расширенную версию одной из них, которая, вполне возможно, пригодится вам и в дальнейшем.

А писать мы будем настоящую функцию дифференцирования. Практически такую же, как встроенная diff(), только без вычисления полного дифференциала – чтобы не слишком сложно было «охватить» пониманием сразу весь пример. Ну а если будет интерес, то дописать вычисление полного дифференциала к этой же функции вы можете попробовать самостоятельно – после освоения возможностей, которые сейчас будут продемонстрированы, это будет уже несложно. Примеров применения по ходу создания функции я давать не буду. Если вы хотите смотреть на практические результаты, по мере добавления кода можно сохранять его в файле, скажем, **~/.maxima/deriv.mac** и выполнять в *Maxima* строку load(deriv)$ deriv(какое-нибудь-выражение);.

Я буду писать код постепенно и по ходу написания давать комментарии к последнему написанному участку. Комментировать буду, просто вставляя куски кода в текст. К слову: *Maxima* поддерживает комментарии в коде «в стиле Си», то есть комментарий начинается символами /*, а заканчивается */. Причем, в отличие от Си, допускаются вложенные комментарии: /* вот /* такие */ */.

Чтобы не повторять каждый раз весь код от самого начала, я буду сокращать его с помощью многоточия. Если вы будете проверять код по мере чтения, не забывайте о разделяющих запятых после последних строк предыдущих участков.

Начнем с «подготовительных работ»: проверки определенных условий и сохранения нужных значений в локальных переменных.

```
deriv([l]):=block([f,len,x],
        len:length(l),
        if len=0 then
        error("deriv can't be used without arguments"),
        f:l[1],
        x:listofvars(f)
)$
```

Итак, по порядку. Символ в квадратных скобках означает, что ему будет присвоен список из всех аргументов, с которыми вызвана функция. Эта конструкция предназначена для создания функций с переменным числом аргументов.

Функция block() – это расширенный аналог составного оператора. Отличается она двумя вещами. Во-первых, поддерживается возврат значений через return(), точно так же как из цикла, то есть по return(выражение) будет осуществлен выход из блока и результатом вычисления блока станет «выражение». А во-вторых, в блоке можно использовать локальные переменные – то есть такие, которые не повлияют на значения символов вне блока, даже если будут иметь совпадающие с ними имена. Такие локальные символы перечисляются в виде списка в самом начале блока.

Далее мы сохраняем в одной из таких локальных переменных длину списка аргументов (функция length) и в случае, если она равна нулю (то есть аргументов нет), генерируем ошибку функцией error, которая может принимать произвольное число аргументов, которые она вычисляет и выводит прежде чем создать ошибку.

Функция listofvars возвращает список переменных переданного ей выражения. Этот список понадобится нам для небольшого расширения возможностей: так как мы не будем вычислять полный дифференциал, то вызов с одним аргументом у нас освобождается, и мы будем его использовать аналогично функции solve: если переданное выражение включает в себя только одну неизвестную, будем дифференцировать его по ней. Продолжаем:

```
deriv([l]):=block([f,len,x],
...
    x:listofvars(f),
    if len=1 then (
        if length(x)=0 then
            return(0),
        if length(x)>1 then
            error("Expression has more than one unknowns and none was
specified.","Unknowns given:", x),
        x:x[1] )
    else
        x:l[2]
)$
```

Если параметр дифференцирования в списке аргументов не задан, то проверяем длину списка неизвестных. Если она равна нулю – то это

константа и следовательно возвращаем ноль. Если больше единицы, то неизвестно, по чему дифференцировать, следовательно, снова генерируем ошибку. Ну а в случае единицы, просто превращаем список из одного элемента в сам этот элемент. Если же список аргументов длиннее, то берем параметр оттуда.

```
deriv([l]):=block([f,len,x],
...
    else
        x:l[2]
    if len>=3 then
    error("More than 2 arguments not implemented yet.")
)$
```

Пока ограничимся производной первого порядка по одной переменной. Когда этот этап будет пройден, остальное будет уже нетрудно написать на основе имеющегося кода. Теперь, когда проверки закончены, приступаем непосредственно к реализации. Строить эту функцию мы будем поэтапно. Для начала научим ее дифференцировать просто переменную и константу:

```
deriv([l]):=block([f,len,x],
...
    error("More than 2 arguments not implemented yet.")
    if atom(f) or subvarp(f) then
        if f=x then
            return(1)
        else
            return(0),
    else
        return 'diff(f,x)
)$
```

Предикат atom() проверяет, является ли его аргумент атомарным выражением, то есть константой (целой либо с плавающей точкой) или одиночным символом. Второй предикат – subvarp() – расшифровывается как subscripted variable (predicate), где первые два слова означают «индексированная переменная», то есть что-то вида **a[1]**. Добавлен этот предикат в эту же проверку в связи с тем, что *Maxima* такие выражения атомарными не считает, а с точки зрения дифференцирования они как раз являются атомами. Дальше в этом варианте все просто: если атомарное выражение является параметром дифференцирования, то результат будет равен единице, иначе – нулю: в полном соответствии с правилами дифференцирования.

В самом конце функции добавляем строку, которая в нештатном случае (таком, который мы еще не посчитали) просто вернет несовершенную форму производной от оставшегося выражения. Эта строка у нас вплоть до самой полной реализации будет оставаться последней, а все остальное мы будем вписывать до нее, сокращая тем самым этому некрасивому умолчательному случаю шансы на выживание. А двигаться дальше мы будем достаточно интересным способом, с помощью уже упомянутой в статье рекурсии. Мы будем постепенно обучать нашу функцию все новым и новым трюкам (точнее, правилам дифференцирования), разбивая неизвестные выражения некоторыми способами на более простые, уже обработанные варианты; то есть действуя снова по известному «принципу чайника». И вы увидите, что математики не зря так любят этот принцип: с его помощью такая, на первый взгляд, сложная задача будет разбита на множество простых подзадачек и таким образом упростится сама. Например, первым пойдет вычитание. Точнее, унарный минус или попросту отрицательные величины: бинарного минуса в *Maxima* по сути не существует, а любое выражение вида **a–b** имеет внутреннюю форму a+(–b), то есть сводится по все тому же принципу к плюсу. Итак, приступим:

```
setup_autoload(stringproc,sequal)$
deriv([l]):=block([f,len,x,o],
...
        else
            return(0),
    o:op(f),
    return (
        if sequal(o,"-") then
            -deriv(-f,x)
        else
            'diff(f,x)
    )
)$
```

Тут мы уже начинаем использовать те самые функции по «глубокой» обработке выражений. Функция op() возвращает основной оператор заданного выражения. Основным считается самый внешний; например op(a+b/c) будет равен "+", op((a+b)*2) – "*", а op(sin(x^2+y^2)) – sin. Дальше включается «принцип чайника»: для отрицательного выражения мы просто выносим минус за скобки, а для остального, теперь уже положительного, вызываем саму же функцию deriv.

Здесь для сверки значения оператора с минусом используется не equal(), а ее строковый аналог – sequal(), проверяющий на равенство две строки. Связано это с тем, что разные операторы *Maxima* хранит в разном виде, и при сверке, скажем, того же минуса, который хранится как текстовый знак с синусом, хранящимся как символ (идентификатор) *Maxima*, обычный equal() просто выдаст ошибку.

Функция sequal() – внешняя, она хранится в файле **stringproc** (от фразы «string processing» – обработка строк), который и нужно подгрузить до использования этой функции. А для того чтобы, файл не приходилось загружать вручную, но при этом он и не загружался бы при каждом вызове функции (как было бы в случае вызова load() внутри функции deriv()), есть, с одной стороны, традиционный способ: определить внутри файла некую константу или свойство, а перед его загрузкой проверять их наличие: если нету – тогда и подгружать. Мы же используем не общепринятый, но в чем-то более простой метод: рассмотренную в статье функцию setup_autoload. Благодаря ей, нам с одной стороны, не надо лезть в исходники библиотек (которые, кстати говоря, часто бывают не на языке *Maxima*, а на Lisp) и искать там флаги; а с другой – мы все же уверены, что файл будет загружаться не больше одного раза: именно это и гарантируется функцией setup_autoload.

И последний момент в этом кусочке: обратите внимание на оператор if, сместившийся внутрь функции return(). Напомню, что if в *Maxima* является полноценным оператором, то есть всегда возвращает последнее вычисленное значение. А раз так, нет никакого смысла вызывать return() много раз. По большому счету, здесь и один вызов return() не нужен: результатом block(), как и примитивного составного оператора, будет последнее вычисленное выражение. Так что для еще большей краткости напишем даже так:

```
...
    o:op(f),
    if sequal(o,"-") then
        -deriv(-f,x)
    else
        'diff(f,x)
)$
```

После минуса логично было бы заняться плюсом; но поскольку сумма при дифференцировании переходит в сумму, то проще будет реализовать ее сразу для произвольного числа слагаемых, а это уже немного сложнее. Потому начнем с более простых в реализации арифметических действий: умножения и деления.

```
...
    if sequal(o,"-") then
        -deriv(-f,x)
    else if sequal(o,"*") then
        deriv(first(f),x)*rest(f)+first(f)*deriv(rest(f),x)
    else if sequal(o,"/") then
    (deriv(first(f),x)*last(f)-first(f)*deriv(last(f),x))/last(f)^2
    else
        'diff(f,x)
)$
```

Здесь мы сталкиваемся с одним очень интересным и весьма полезным свойством: многие из функций работы со списками, которых в *Maxima* немало, воспринимают как списки также и любые выражения. Так, «списковая» функция first(), возвращающая первый элемент заданного списка, вызванная как first(a*b*c), вернет **a**; а у функции rest() («остаток»), отдающей (в варианте вызова с одним аргументом), наоборот, весь список кроме первого элемента, на том же выражении результатом будет **b*c**. Этим мы и воспользовались, вызывая при этом снова для каждого слагаемого саму функцию deriv(). Если сомножителей будет больше чем два, то вызов deriv(rest(f),x) пройдет по этой же ветке и отсечет еще один.

Так же мы поступаем и с делением. Здесь, так как аргумента всегда два, вместо rest() используется функция last() – последний элемент списка (rest() в этом же случае вернула бы список из одного элемента, а потому last() более удобна). Только одно «но»: деление почему-то обозначается во внутреннем представлении *Maxima* не одиночной, а двойной косой чертой.

Точно таким же образом можно обработать последний бинарный оператор (кроме оставленного на закуску сложения) – возведение в степень. Здесь тоже нет никаких сложностей, и даже нечего дополнительно объяснять по сравнению с делением:

```
...
    else if sequal(o,"^") then
        first(f)^last(f)*log(first(f))*deriv(last(f),x)+
        first(f)^(last(f)-1)*last(f)*deriv(first(f),x)
    else
        'diff(f,x)
)$
```

Теперь вернемся к сложению. Тут нам уже пригодятся упомянутые в статье функции по работе с функциями, а конкретно – функция map(). Она принимает в качестве первого аргумента имя функции и как бы вкладывает эту функцию внутрь выражений – последующих аргументов. Проще всего будет пояснить на примере: map(f,[a,b,c]) даст результат [f(a),f(b),f(c)]. И, что самое замечательное, она, точно так же, как и «списковые» функции, работает не только со списками, но и с любыми выражениями; например, map(f,a+b+c) –> f(a)+f(b)+f(c). Как хорошо подходит для нашей задачи, не правда ли? Именно так и должна действовать на сумму функция дифференцирования. Все было бы совсем хорошо, если бы deriv() принимала, кроме выражения, только один аргумент. С двумя выражениями map тоже умеет работать, но только если у них одинаковый основной оператор; то есть сумму можно «отобразить» только на сумму: map(f,a+b+c,x+y+z) –> f(c,z)+f(b,y)+f(a,x). Проблема здесь в том, что у нас второй аргумент во всех вызовах deriv(), которые должны попасть внутрь суммы, одинаков, а выражение вида x+x+x передать невозможно: оно автоматически упростится в 3*x. Но, как известно, из любой безвыходной ситуации всегда есть как минимум два выхода. И в данном случае один из этих выходов достаточно прост: написать небольшую функцию-«обертку» вокруг map:

```
map1st(f,expr,x):=block([o],
    o:op(expr),
    subst(o,"[",map(f,subst("[",o,expr),makelist(x,i,1,length(expr))))
)$
```

Еще одна новая функция «глубинной» работы с выражениями: subst(). Она способна заменять в выражении... да почти что угодно и почти на что угодно. Вызывается так: subst(стало, было, выражение), заменяя в «выражении» все, что «было», на «стало». Опять же, в качестве подвыражений могут использоваться операторы, то есть subst("*","+",x+y+z) –> x*y*z. Мы используем ее для временной подмены основного оператора выражения оператором списка (который обозначается как "[", то есть [a,b,c] – это, по сути, "["(a,b,c)). Затем генерируем список такой же, как выражение, длины, заполненный заданной переменной, – и применяем к двум полученным спискам функцию map(), а затем возвращаем назад вместо списка первоначальный базовый оператор. То есть теперь, к примеру, map1st(f,a+b+c,x) будет равно как раз f(c,x)+f(b,x)+f(a,x). Et voila, как говорят французы! И теперь внутри deriv() можно применить к сложению именно эту новую функцию. Заодно применим ее и к списку – (deriv([f,g],x) будет равно [deriv(f,x),deriv(g,x)]) и, чего уж там мелочиться, и к множеству:

```
...
    o:op(f),
    if sequal(o,"+") or sequal(o,"[") or sequal(o,set) then
        map1st(deriv,f,x)
    else if sequal(o,"-") then
        -deriv(-f,x)
...
```

Множества, к слову, в *Maxima* реализованы в самом что ни на есть математическом смысле: множество может включать в себя каждый элемент только один раз; и это учитывается и встроенными операциями по работе с множествами: пересечением, объединением и т.д. Есть еще некоторые ошибки, но они документированы и потому не неожиданны.

Движемся дальше. У нас уже реализована производная от всех бинарных операторов, а дальше мы нарисуем «таблицу производных» и будем работать с нею:

```
deriv([l]):=block([f,len,o,x,func,fdrv],
...
    o:op(f),
    func:[sqrt, sin, cos, abs, exp, log, tan, cot, sec, csc, asin, acos, atan,
 acot, asec, acsc, sinh, cosh, tanh, coth, asinh, acosh, atanh, acoth,
 asech, acsch],
    fdrv:[1/2/arg, cos(arg), -sin(arg), arg/abs(arg), exp, 1/arg, sec(arg)^2,
-csc(arg)^2, tan(arg)*sec(arg), -cot(arg)*csc(arg), 1/sqrt(1-arg^2), -1/
 sqrt(1-arg^2), 1/(1+arg^2), -1/(1+arg^2), 1/arg^2/sqrt(1-1/arg^2), -1/
 arg^2/sqrt(1-1/arg^2), cosh(arg), sinh(arg), sech(arg), -csch(arg), 1/
 sqrt(arg^2+1), 1/sqrt(arg^2-1), 1/(1-arg^2), 1/(1-arg^2), -1/arg^2/sqrt(1/
 arg^2-1), -1/arg^2/sqrt(1/arg^2+1)],
    if sequal(o,"+") or sequal(o,"[") or sequal(o,set) then
...
```

Для упрощения работы с «таблицей» напишем еще две небольших вспомогательных функции: одна будет проверять, входит ли заданный элемент в заданный список, а вторая – возвращать номер, соответствующий заданному элементу в заданном списке, при условии что он там есть.

```
smember(expr,list):=
    if sequal(true,
        for i in list do
            if sequal(expr,i) then
                return(true) )
    then true$
sindex(expr,list):=block([num],
```

```
    num:for i:1 thru length(list) do
            if sequal(expr,list[i]) then
                return(i),
    if integerp(num) then num
)$
```

Здесь есть только одна тонкость, связанная с небольшой проблемой. Заключается эта проблема в том, что для возвращения значения из блока и из цикла в *Maxima* используется одна и та же функция return(). Это приводит к тому, что выйти из блока, находясь внутри цикла в нем, невозможно – приходится выдумывать некоторые несложные ухищрения. Теперь с использованием двух новых функций заменяем элементы «таблицы» их производными; с помощью уже знакомой нам subst, которая подставит нужное выражение внутрь табличной функции вместо ключевого слова arg.

```
...
    else if smember(o,func) then
    deriv(first(f),x)*subst(first(f),arg,fdrv[sindex(o,func)])
    else
        'diff(f,x)
)$
```

Вот так, начиная с самых простых элементов, а затем, подобно Мюнхгаузену, вытаскивая самих себя сантиметр за сантиметром, мы и получили полноценную функцию дифференцирования. Правда, пока только первого порядка и только по одному аргументу. Но имея то, что имеем, двигаться дальше, следуя известному принципу, уже совсем не сложно: просто заменим строку «if len>=3 then error...» следующим куском:

```
...
    if len=3 then (
        integerp(l[3]) or
            return('diff(x,l[3])),
        if l[3]=0 then
            return(f),
        if l[3]<0 then
            error("Improper count to deriv:",l[3]),
        if l[3]>1 then
            return(deriv(deriv(f,x,l[3]-1),x))
    ),
    if len>3 then (
        if(evenp(len)) then
            l:endcons(1,l),
        return(deriv(apply(deriv,rest(l,-2)),l[len],l[len+1]))
    ),
...
```

Пройдемся по нескольким неосвещенным моментам. В силу способов вычисления в *Maxima* (которые сродны таковым во многих языках программирования) конструкция вида «условие or выражение» равносильно «if not условие then выражение» – и использована здесь исключительно для разнообразия, в учебных целях. Здесь мы в случае нецелого порядка дифференцирования просто возвращаем несовершенную форму – точно так же, как это делает и штатная функция diff().

Производная нулевого порядка от любой функции – это сама функция. А производные отрицательных порядков некорректны, о чем мы и генерируем сообщение. Для порядков, больших единицы, понижаем порядок как и раньше – за счет самовызова.

Далее я немного усовершенствовал поведение функции по сравнению со встроенной: если та не умеет принимать четное количество аргументов больше двух (то есть с неуказанным порядком дифференцирования по последней неизвестной когда неизвестных больше одной), то у нас в данном случае, так же как и для одной неизвестной, будет подразумеваться единица. Здесь предикат evenp() проверяет число на четность (even – четный), а функция endcons() добавляет заданный элемент в конец заданного списка. Ее имя носит исторический характер: парная к ней функция cons(), добавляющая элемент в начало списка, свое имя позаимствовала из Lisp, а слово end здесь добавлено «по смыслу».

Далее мы, снова самовызовом, укорачиваем список параметров дифференцирования. При этом используется еще одна функция, работающая с функциями, – apply() (применять). Она принимает два аргумента, первый из которых – имя функции, а второй – список, и применяет заданную функцию к списку как к списку аргументов. Также здесь использован более широкий вариант вызова rest(): он может принимать второй аргумент – целое число, не равное нулю. Если число положительно, то такое количество элементов выбрасывается из начала списка, а если отрицательно – то с конца; в данном случае мы теряем последние два элемента.

Вот и все. Мы уже имеем полную функцию дифференцирования, берущую производные с произвольным количеством параметров и любых порядков. Полный текст всех созданных функций вы можете найти в файле deriv.mac на прилагаемом к журналу диске.

Дополнительно хочется остановиться на одной незамысловатой функции, которая, тем не менее, может неплохо помочь в отладке собственных модулей. Это функция display(), которая принимает имена и отображает их значения в виде «имя=значение». В качестве эксперимента можете добавить ее где-нибудь внутри функции deriv() и отследить процесс самовызова (в файле на диске вызов display() достаточно раскомментировать).

И в качестве финального аккорда сделаем еще и более универсальную версию вспомогательной функции map1st() – возможно, тогда она вам пригодится и еще где-нибудь.

```
mapany(f,[lst]):=block([o,l],l:lst,
    if length(setify(map(length,l)))>1 then
    error("Arguments to mapany are not of the same length"),
    o:op(l[1]),
    for i:1 thru length(l) do
        l[i]:subst("[",op(l[i]),l[i]),
    subst(o,"[",apply(map,cons(f,l)))
)$
```

Здесь я уже воздержусь от столь подробных комментариев, так как практически все, что используется в этой функции, уже было в той или иной степени разъяснено в процессе описания deriv(). Остановлюсь только на одной строчке:

```
            if length(setify(map(length,l)))>1 then
```

Здесь используется не совсем простой прием для проверки длин списков на одинаковость. Так как l – это список из списков, то сначала получаем список длин «вкручиванием» внутрь внешнего списка функции length(). Дальше – интереснее. Функция setify (дословно – что-то вроде «множествицировать») превращает список в множество. Так как множество не может содержать несколько равных между собой элементов, то такие элементы при этом «склеиваются»: из них остается один. Таким образом если «длина» (количество элементов) множества больше единицы, то как минимум два элемента в первоначальном списке были неравны между собой.

И вернувшись к рассмотренной функции дифференцирования, хочется еще раз обратить ваше внимание на использованный прием: конструировать большие и сложные функции из более маленьких и простых кусочков с помощью рекурсии. Этот метод очень часто и продуктивно используется в функциональном программировании, к которому *Maxima*, в силу своих Lisp-овских корней, очень близка. LXF