Коротко о Maxima

Роберт Додиер

1 Что такое Maxima?

Maxima — система для работы с выражениями, такими как $x + y$, $\sin(a + b\pi)$ и $u \cdot v - v \cdot u$.

Maxima не особо заботится о смысле выражения. Только пользователь решает, какой смысл несет выражение.

Иногда требуется задать значения неизвестным и вычислить выражение — Maxima с радостью сделает это. Но система с той же радостью отложит присваивание конкретных значений, так что вы можете провести с выражением некоторые преобразования, после чего уже определить неизвестные (или не определять их вовсе).

Рассмотрим несколько примеров.

1. Нужно найти объем шара:

```
(%i1) V: 4/3 * %pi * r^3;
                                    3
                             4 %pi r
(%o1)                        --------
                                3
```

2. Радиус равен 10:

```
(%i2) r: 10;
(%o2)                           10
```

3. V — то же, что и было; Maxima не поменяет V, если это не указать:

```
(%i3) V;
                                    3
                             4 %pi r
(%o3)                        --------
                                3
```

4. «Maxima, пересчитай, пожалуйста, V»:

```
(%i4) ''V;
                             4000 %pi
(%o4)                        --------
                                3
```

5. Вместо выражения хотелось бы видеть численное значение:

```
(%i5) ''V, numer;
(%o5)                    4188.79020478639
```

2 Выражения

Всё в Maxima является выражениями, в том числе математические выражения, объекты и программные блоки. Выражение — атом либо оператор с аргументами.

Атом — символ (имя), строка в кавычках, либо число (целое или с плавающей точкой).

Все выражения не-атомы представляются в виде $op(a_1, \ldots, a_n)$, где op — имя оператора, а $a_1, \ldots, a_n$ — его аргументы. Выражения могут отображаться по-разному, но внутреннее представление всегда одинаково. Аргументы выражения могут быть атомами либо выражениями не-атомами.

Математические выражения включают математические операторы, такие как

$$+ \quad - \quad * \quad / \quad < \quad = \quad >$$

либо вычисление функции вроде $\mathbf{sin}(x), \mathbf{bessel\_j}(n, x)$. В таких случаях оператором является функция.

Объекты в Maxima — тоже выражения. Список $[a_1, \ldots, a_n]$ — выражение $\mathbf{list}(a_1, \ldots, a_n)$. Матрица — выражение

$$\mathbf{matrix}(\mathbf{list}(a_{1,1}, \ldots, a_{1,n}), \ldots, \mathbf{list}(a_{m,1}, \ldots, a_{m,n}))$$

Программными блоками являются выражения. Блок кода $\mathbf{block}(a_1, \ldots, a_n)$ — выражение с оператором **block** и аргументами $a_1, \ldots, a_n$. Условная конструкция **if** a **then** b **elseif** c **then** d — выражение $\mathbf{if}(a, b, c, d)$. Цикл **for** a **in** L **do** S — выражение, соответствующее $\mathbf{do}(a, L, S)$.

Функция Maxima **op** возвращает оператор выражения-не-атома. Функция **args** врзвращает аргументы выражения-не-атома. Функция **atom** указывает, является ли выражение атомом.

Рассмотрим другие примеры.

1. Атомы — символы, строки и числа. Вот список с элементами-атомами:

```
(%i2) [a, foo, foo_bar, "Hello, world!", 42, 17.29];
(%o2)      [a, foo, foo_bar, Hello, world!, 42, 17.29]
```

2. Математические выражения:

```
(%i1) [a + b + c, a * b * c, foo = bar, a*b < c*d];
(%o1)       [c + b + a, a b c, foo = bar, a b < c d]
```

3. Списки и матрицы. Элементами списка или матрицы могут быть любые выражения, в том числе списки или матрицы:

```
(%i1) L: [a, b, c, %pi, %e, 1729, 1/(a*d - b*c)];
                                                1
(%o1)           [a, b, c, %pi, %e, 1729, ---------]
                                         a d - b c
```

```
(%i2) L2: [a, b, [c, %pi, [%e, 1729], 1/(a*d - b*c)]];
                                                1
(%o2)        [a, b, [c, %pi, [%e, 1729], ---------]]
                                         a d - b c
(%i3) L [7];
                                1
(%o3)                       ---------
                            a d - b c
(%i4) L2 [3];
                                          1
(%o4)            [c, %pi, [%e, 1729], ---------]
                                       a d - b c
(%i5) M: matrix ([%pi, 17], [29, %e]);
                            [ %pi  17 ]
(%o5)                       [         ]
                            [ 29   %e ]
(%i6) M2: matrix ([[%pi, 17], a*d - b*c], [matrix ([1, a], [b, 7]), %e]);
                     [ [%pi, 17]  a d - b c ]
                     [                      ]
(%o6)                [ [ 1  a ]             ]
                     [ [      ]      %e     ]
                     [ [ b  7 ]             ]
(%i7) M [2][1];
(%o7)                           29
(%i8) M2 [2][1];
                              [ 1  a ]
(%o8)                         [      ]
                              [ b  7 ]
```

4. Программные блоки — выражения. $x : y$ означает присваивание y к x; значение выражения присваивания — y. **block** объединяет несколько выражений и последовательно их вычисляет; значение блока соответствует значению его последнего выражения.

```
(%i1) (a: 42) - (b: 17);
(%o1)                           25
(%i2) [a, b];
(%o2)                        [42, 17]
(%i3) block ([a], a: 42, a^2 - 1600) + block ([b], b: 5, %pi^b);
                                5
(%o3)                        %pi  + 164
(%i4) (if a > 1 then %pi else %e) + (if b < 0 then 1/2 else 1/7);
                                     1
(%o4)                          %pi + -
                                     7
```

5. **op** возвращает оператор, **args** возвращает аргументы, **atom** определяет, является ли выражение атомом:

```
(%i1) op (p + q);
(%o1)                           +
(%i2) op (p + q > p*q);
(%o2)                           >
(%i3) op (sin (p + q));
(%o3)                         sin
(%i4) op (foo (p, q));
(%o4)                         foo
(%i5) op (foo (p, q) := p - q);
(%o5)                          :=
(%i6) args (p + q);
(%o6)                        [q, p]
(%i7) args (p + q > p*q);
(%o7)                    [q + p, p q]
(%i8) args (sin (p + q));
(%o8)                        [q + p]
(%i9) args (foo (p, q));
(%o9)                       [p, - q]
(%i10) args (foo (p, q) := p - q);
(%o10)                [foo(p, q), p - q]
(%i11) atom (p);
(%o11)                        true
(%i12) atom (p + q);
(%o12)                        false
(%i13) atom (sin (p + q));
(%o13)                        false
```

6. Операторы и аргументы программных блоков. Одинарная кавычка указывает Maxima создать выражение, но отложить его вычисление. Мы еще рассмотрим это позже.

```
(%i1) op ('(block ([a], a: 42, a^2 - 1600)));
(%o1)                         block
(%i2) op ('(if p > q then p else q));
(%o2)                          if
(%i3) op ('(for x in L do print (x)));
(%o3)                         mdoin
(%i4) args ('(block ([a], a: 42, a^2 - 1600)));
                                  2
(%o4)              [[a], a : 42, a  - 1600]
(%i5) args ('(if p > q then p else q));
(%o5)                  [p > q, p, true, q]
(%i6) args ('(for x in L do print (x)));
(%o6)     [x, L, false, false, false, false, print(x)]
```

3 Вычисление

Значение символа — выражение, связанное с этим символом. Каждый символ имеет значение; если значение не задавалось, символ вычисляется сам в себя. Например, x имеет значение x, если символу не присваивалось значение.

Числа и строки совпадают со своими значениями.
Выражение-не-атом вычисляется приблизительно таким образом:

1. Вычисляется каждый аргумент оператора.
2. Если оператор связан с вызовом функции, то значение, возвращенное функцией, является значением выражения.

Вычисление может проходить по-разному. Некоторые изменения уменьшают объем вычислений:

1. Некоторые функции не вычисляют свои аргументы или часть из них, либо меняют ход вычисления аргументов.
2. Одиночная кавычка $'$ предотвращает вычисление:
   — $'a$ вычисляется как a. Все другие значения a игнорируются.
   — $'f(a_1, \ldots, a_n)$ вычисляется в $f(\mathbf{ev}(a_1), \ldots, \mathbf{ev}(a_n))$. Таким образом, вычисляются аргументы, но f не вызывается.
   — $'(\ldots)$ предотвращает вычисление любых выражений внутри $(\ldots)$.

Некоторые изменения увеличивают объем вычислений:

1. Две одиночные кавычки $''a$ вызывают дополнительное вычисление в момент обработки a.
2. $\mathbf{ev}(a)$ вызывает дополнительное вычисление a при каждом вычислении $\mathbf{ev}(a)$.
3. Запись $\mathbf{apply}(f, [a_1, \ldots, a_n])$ вызывает вычисление аргументов $a_1, \ldots, a_n$, даже если f ставит перед ними одиночные кавычки.
4. **define** соответствует определению функции вроде :=, но вычисляет тело функции, в то время как := откладывает вычисление.

Рассмотрим, как вычисляются некоторые выражения.

1. Символ вычисляется сам в себя, если ему не присваивалось значение:

```
(%i1) block (a: 1, b: 2, e: 5);
(%o1)                          5
(%i2) [a, b, c, d, e];
(%o2)                  [1, 2, c, d, 5]
```

2. Аргументы операторов вычисляются обычным путем (если вычисление не отложено тем или иным образом):

```
(%i1) block (x: %pi, y: %e);
(%o1)                         %e
(%i2) sin (x + y);
(%o2)                      - sin(%e)
(%i3) x > y;
(%o3)                      %pi > %e
(%i4) x!;
(%o4)                        %pi!
```

3. Если оператор связан с вызовом функции, значение, возвращенное функцией, является значением выражения (если вычисление не отложено); иначе вычисление дает другое выражение с тем же оператором:

```
(%i1) foo (p, q) := p - q;
(%o1)                  foo(p, q) := p - q
(%i2) p: %phi;
(%o2)                          %phi
(%i3) foo (p, q);
(%o3)                        %phi - q
(%i4) bar (p, q);
(%o4)                      bar(%phi, q)
```

4. Некоторые функции откладывают вычисление аргументов, например, **save**, :=, **kill**:

```
(%i1) block (a: 1, b: %pi, c: x + y);
(%o1)                          y + x
(%i2) [a, b, c];
(%o2)                    [1, %pi, y + x]
(%i3) save ("tmp.save", a, b, c);
(%o3)                        tmp.save
(%i4) f (a) := a^b;
                                    b
(%o4)                      f(a) := a
(%i5) f (7);
                             %pi
(%o5)                        7
(%i6) kill (a, b, c);
(%o6)                          done
(%i7) [a, b, c];
(%o7)                       [a, b, c]
```

5. Одиночная кавычка предотвращает вычисление, даже если оно должно производиться:

```
(%i1) foo (x, y) := y - x;
(%o1)                  foo(x, y) := y - x
(%i2) block (a: %e, b: 17);
(%o2)                           17
(%i3) foo (a, b);
(%o3)                        17 - %e
(%i4) foo ('a, 'b);
(%o4)                          b - a
(%i5) 'foo (a, b);
(%o5)                      foo(%e, 17)
(%i6) '(foo (a, b));
(%o6)                       foo(a, b)
```

6. Две одиночные кавычки вызывают дополнительное вычисление во время обработки выражения:

```
(%i1) diff (sin (x), x);
(%o1)                          cos(x)
(%i2) foo (x) := diff (sin (x), x);
(%o2)                foo(x) := diff(sin(x), x)
(%i3) foo (x) := ''(diff (sin (x), x));
(%o3)                    foo(x) := cos(x)
```

7. **ev** всякий раз вызывает дополнительное вычисление (сравните это с поведением для двух кавычек):

```
(%i1) block (xx: yy, yy: zz);
(%o1)                            zz
(%i2) [xx, yy];
(%o2)                          [yy, zz]
(%i3) foo (x) := ''x;
(%o3)                         foo(x) := x
(%i4) foo (xx);
(%o4)                            yy
(%i5) bar (x) := ev (x);
(%o5)                      bar(x) := ev(x)
(%i6) bar (xx);
(%o6)                            zz
```

8. **apply** вызывает вычисление аргументов, даже если перед ними стоят кавычки:

```
(%i1) block (a: aa, b: bb, c: cc);
(%o1)                            cc
(%i2) block (aa: 11, bb: 22, cc: 33);
(%o2)                            33
(%i3) [a, b, c, aa, bb, cc];
(%o3)                 [aa, bb, cc, 11, 22, 33]
(%i4) apply (kill, [a, b, c]);
(%o4)                           done
(%i5) [a, b, c, aa, bb, cc];
(%o5)                 [aa, bb, cc, aa, bb, cc]
(%i6) kill (a, b, c);
(%o6)                           done
(%i7) [a, b, c, aa, bb, cc];
(%o7)                   [a, b, c, aa, bb, cc]
```

9. **define** вычисляет тело определения функции:

```
(%i1) integrate (sin (a*x), x, 0, %pi);
                          1   cos(%pi a)
(%o1)                     - - ----------
                          a       a
(%i2) foo (x) := integrate (sin (a*x), x, 0, %pi);
```

```
(%o2)       foo(x) := integrate(sin(a x), x, 0, %pi)
(%i3) define (foo (x), integrate (sin (a*x), x, 0, %pi));
                               1   cos(%pi a)
(%o3)               foo(x) := - - ----------
                               a       a
```

4 Упрощение

После вычисления выражения, Maxima пытается найти эквивалентное ему «более простое», для чего применяется ряд правил, связанных с условным понятием простоты. Так, $1+1$ упрощается до 2, $x+x$ — до $2x$, а **sin**(%**pi**) — до 0.

Однако многие известные тождества не применяются автоматически. Например, не используются формулы двойного угла для тригонометрических функций и не производится приведение дробей вида $a/b + c/b \rightarrow (a + c)/b$. Для применения тождеств существуют отдельные функции.

Упрощение всегда применяется, если не было явно отложено, даже в том случае, когда выражение не вычисляется.

tellsimpafter вводит пользовательские правила упрощения.

Рассмотрим несколько примеров.

1. Знак кавычки откладывает вычисление, но не упрощение; Если для глобального флага **simp** установлено **false**, то упрощение не производится, но производится вычисление:

```
(%i1) '[1 + 1, x + x, x * x, sin (%pi)];
                                    2
(%o1)                    [2, 2 x, x , 0]
(%i2) simp: false$
(%i3) block ([x: 1], x + x);
(%o3)                           1 + 1
```

2. Некоторые тождества не применяются автоматически. **expand**, **ratsimp**, **trigexpand**, **demoivre** — примеры функций, применяющих тождества:

```
(%i1) (a + b)^2;
                                    2
(%o1)                         (b + a)
(%i2) expand (%);
                            2             2
(%o2)                       b  + 2 a b + a
(%i3) a/b + c/b;
                              c   a
(%o3)                         - + -
                              b   b
(%i4) ratsimp (%);
                              c + a
(%o4)                         -----
                                b
(%i5) sin (2*x);
(%o5)                        sin(2 x)
```

```
(%i6) trigexpand (%);
(%o6)                     2 cos(x) sin(x)
(%i7) a * exp (b * %i);
                                 %i b
(%o7)                         a %e
(%i8) demoivre (%);
(%o8)                a (%i sin(b) + cos(b))
```

5 apply, map и lambda

1. **apply** создает и вычисляет выражение. Аргументы выражения всегда вычисляются (даже, если бы они не вычислялись при других обстоятельствах):

```
(%i1) apply (sin, [x * %pi]);
(%o1)                        sin(%pi x)
(%i2) L: [a, b, c, x, y, z];
(%o2)                    [a, b, c, x, y, z]
(%i3) apply ("+", L);
(%o3)                  z + y + x + c + b + a
```

2. **map** создает и вычисляет выражение для каждого элемента списка аргументов. Аргументы выражения всегда вычисляются (даже, если бы они не вычислялись при других обстоятельствах). В качестве результата возвращается список:

```
(%i1) map (foo, [x, y, z]);
(%o1)                [foo(x), foo(y), foo(z)]
(%i2) map ("+", [1, 2, 3], [a, b, c]);
(%o2)                  [a + 1, b + 2, c + 3]
(%i3) map (atom, [a, b, c, a + b, a + b + c]);
(%o3)           [true, true, true, false, false]
```

3. **lambda** создает лямбда-выражение (безымянную функцию). Лямбда-выражение может использоваться в некоторых случаях как обычная функция. **lambda** не вычисляет тело функции:

```
(%i1) f: lambda ([x, y], (x + y)*(x - y));
(%o1)            lambda([x, y], (x + y) (x - y))
(%i2) f (a, b);
(%o2)                      (a - b) (b + a)
(%i3) apply (f, [p, q]);
(%o3)                      (p - q) (q + p)
(%i4) map (f, [1, 2, 3], [a, b, c]);
(%o4)  [(1 - a) (a + 1), (2 - b) (b + 2), (3 - c) (c + 3)]
```

6 Встроенные типы объектов

Объект представляется в виде выражения. Как и другие выражения, объект содержит оператор и его аргументы.

Основные встроенные типы объектов — списки, матрицы и множества.

6.1 Списки

1. Список задается в виде $[a, b, c]$.
2. В списке L $L[i]$ — i-й элемент. $L[1]$ — первый элемент.
3. **map**(f, L) применяет f к каждому элементу L.
4. **apply**$(" + ", L)$ — сумма всех элементов L.
5. **for** x **in** L **do** *expr* вычисляет *expr* для каждого элемента L.
6. **length**(L) — число элементов L.

6.2 Матрицы

1. Матрица задается в виде **matrix**$(L_1, \ldots, L_n)$, где $L_1, \ldots, L_n$ — списки элементов строк.
2. Если M — матрица, то $M[i, j]$ или $M[i][j]$ — ее (i, j)-й элемент. $M[1, 1]$ — элемент в верхнем левом углу.
3. Оператор . представляет некоммутативное умножение. $M.L$, $L.M$ и $M.N$ — некоммутативные произведения, где L — список, а M и N — матрицы.
4. **transpose**(M) — транспонированная матрица M^T.
5. **eigenvalues**(M) возвращает собственные значения M.
6. **eigenvectors**(M) возвращает собственные векторы M.
7. **length**(M) возвращает число строк M.
8. **length**(**transpose**(M)) возвращает число столбцов M.

6.3 Множества

1. Maxima работает с явно заданными конечными множествами. Множества — не то же самое, что и списки, и преобразование множества в список и наоборот должно производиться в явном виде.
2. Множество задается в виде **set**$(a, b, c, \ldots)$, где $a, b, c, \ldots$ — его элементы.
3. **union**(A, B) — объединение множеств A и B.
4. **intersection**(A, B) — пересечение множеств A и B.
5. **cardinality**(A) — число элементов множества A.

7 Типичные задачи

7.1 Определение функции

1. Функция определяется оператором :=, при этом вычисление тела функции откладывается.

   В примере ниже **diff** пересчитывается при каждом вызове функции. Аргумент подставляется вместо x, и вычисляется результирующее выражение. Когда аргумент представляет собой нечто отличное от символа, происходит ошибка: для **foo**(1) Maxima пытается вычислить **diff**$(\mathbf{sin}(1)^2, 1)$.

```
(%i1) foo (x) := diff (sin(x)^2, x);
                                        2
(%o1)              foo(x) := diff(sin (x), x)
(%i2) foo (u);
(%o2)                      2 cos(u) sin(u)
(%i3) foo (1);
Non-variable 2nd argument to diff:
1
#0: foo(x=1)
 -- an error.
```

2. **define** определяет функцию и вычисляет ее тело.

   В следующем примере **diff** вычисляется единожды (при определении), поэтому **foo**(1) не вызывает ошибки:

```
(%i1) define (foo (x), diff (sin(x)^2, x));
(%o1)                foo(x) := 2 cos(x) sin(x)
(%i2) foo (u);
(%o2)                      2 cos(u) sin(u)
(%i3) foo (1);
(%o3)                      2 cos(1) sin(1)
```

7.2 Решение уравнений

```
(%i1) eq_1: a * x + b * y + z = %pi;
(%o1)                   z + b y + a x = %pi
(%i2) eq_2: z - 5*y + x = 0;
(%o2)                     z - 5 y + x = 0
(%i3) s: solve ([eq_1, eq_2], [x, z]);
                (b + 5) y - %pi      (b + 5 a) y - %pi
(%o3)   [[x = - ---------------, z = -----------------]]
                    a - 1                 a - 1
(%i4) length (s);
(%o4)                           1
(%i5) [subst (s[1], eq_1), subst (s[1], eq_2)];
       (b + 5 a) y - %pi   a ((b + 5) y - %pi)
(%o5) [----------------- - ------------------- + b y = %pi,
             a - 1                a - 1
                   (b + 5 a) y - %pi   (b + 5) y - %pi
                   ----------------- - --------------- - 5 y = 0]
                         a - 1               a - 1
(%i6) ratsimp (%);
(%o6)                   [%pi = %pi, 0 = 0]
```

7.3 Интегрирование и дифференцирование

integrate вычисляет определенные и неопределенные интегралы:

```
(%i1) integrate (1/(1 + x), x, 0, 1);
(%o1)                          log(2)
(%i2) integrate (exp(-u) * sin(u), u, 0, inf);
                                1
(%o2)                           -
                                2
(%i3) assume (a > 0);
(%o3)                         [a > 0]
(%i4) integrate (1/(1 + x), x, 0, a);
(%o4)                       log(a + 1)
(%i5) integrate (exp(-a*u) * sin(a*u), u, 0, inf);
                                1
(%o5)                           ---
                                2 a
(%i6) integrate (exp (sin (t)), t, 0, %pi);
                         %pi
                        /
                        [      sin(t)
(%o6)                   I    %e       dt
                        ]
                        /
                         0
(%i7) 'integrate (exp(-u) * sin(u), u, 0, inf);
                       inf
                      /
                      [      - u
(%o7)                 I    %e    sin(u) du
                      ]
                      /
                       0
```

diff вычисляет производные и дифференциалы:

```
(%i1) diff (sin (y*x));
(%o1)         x cos(x y) del(y) + y cos(x y) del(x)
(%i2) diff (sin (y*x), x);
(%o2)                        y cos(x y)
(%i3) diff (sin (y*x), y);
(%o3)                        x cos(x y)
(%i4) diff (sin (y*x), x, 2);
                               2
(%o4)                       - y  sin(x y)
(%i5) 'diff (sin (y*x), x, 2);
                         2
                        d
(%o5)                   --- (sin(x y))
                          2
                        dx
```

7.4 Построение графиков

plot2d строит двумерные графики:

```
(%i1) plot2d (exp(-u) * sin(u), [u, 0, 2*%pi]);
(%o1)
(%i2) plot2d ([exp(-u), exp(-u) * sin(u)], [u, 0, 2*%pi]);
(%o2)
(%i3) xx: makelist (i/2.5, i, 1, 10);
(%o3)  [0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6, 4.0]
(%i4) yy: map (lambda ([x], exp(-x) * sin(x)), xx);
(%o4) [0.261034921143457, 0.322328869227062, .2807247779692679,
.2018104299334517, .1230600248057767, .0612766372619573,
.0203706503896865, - .0023794587414574, - .0120913057698414,
- 0.013861321214153]
(%i5) plot2d ([discrete, xx, yy]);
(%o5)
(%i6) plot2d ([discrete, xx, yy], [gnuplot_curve_styles, ["with points"]]);
(%o6)
```

См. также **plot3d**.

7.5 Сохранение и загрузка файлов

save записывает выражения в файл:

```
(%i1) a: foo - bar;
(%o1)                         foo - bar
(%i2) b: foo^2 * bar;
                                     2
(%o2)                         bar foo
(%i3) save ("my.session", a, b);
(%o3)                        my.session
(%i4) save ("my.session", all);
(%o4)                        my.session
```

load читает выражения из файла.

```
(%i1) load ("my.session");
(%o4)                        my.session
(%i5) a;
(%o5)                         foo - bar
(%i6) b;
                                     2
(%o6)                         bar foo
```

См. также **stringout** и **batch**.

8 Программирование под Maxima

Существует одно пространство имен, содержащее все символы Maxima. Другое пространство имен создать нельзя.

Все переменные глобальны, если не определены локально — в функциях, лямбда-выражениях и блоках.

Значением переменной считается то, что было присвоено в последний раз, в явном виде, либо через присваивание значения локальной переменной в блоке, функции или лямбда-выражении. Эта концепция известна как *динамическая область видимости*.

Если переменная является локальной внутри функции, лямбда-выражения или блока, ее значение локально, но остальные свойства (заданные **declare**) глобальны. Функция **local** делает переменную локальной в отношении всех свойств.

По умолчанию, определение функции глобально, даже если оно содержится внутри функции, лямбда-выражения или блока. **local**$(f), f(x) := \ldots$ создает локальное определение функции.

trace(*foo*) указывает Maxima печатать сообщение при входе в функцию *foo* и выходе из нее.

Рассмотрим некоторые примеры программирования под Maxima.

1. Все переменные глобальны, если не определены локально — в функциях, лямбда-выражениях и блоках:

```
(%i1) (x: 42, y: 1729, z: foo*bar);
(%o1)                          bar foo
(%i2) f (x, y) := x*y*z;
(%o2)                    f(x, y) := x y z
(%i3) f (aa, bb);
(%o3)                       aa bar bb foo
(%i4) lambda ([x, z], (x - z)/y);
                                       x - z
(%o4)                 lambda([x, z], -----)
                                         y
(%i5) apply (%, [uu, vv]);
                               uu - vv
(%o5)                          -------
                                1729
(%i6) block ([y, z], y: 65536, [x, y, z]);
(%o6)                      [42, 65536, z]
```

2. Значением переменной считается то, что было присвоено в последний раз, в явном виде, либо через присваивание значения локальной переменной:

```
(%i1) foo (y) := x - y;
(%o1)                     foo(y) := x - y
(%i2) x: 1729;
(%o2)                           1729
(%i3) foo (%pi);
(%o3)                         1729 - %pi
(%i4) bar (x) := foo (%e);
(%o4)                     bar(x) := foo(%e)
```

```
(%i5) bar (42);
(%o5)                          42 - %e
```

9 Lisp и Maxima

Запись **:lisp** *expr* вычисляет *expr* в интерпретаторе Lisp. Эта запись распознается в строке ввода и файлах, обрабатываемых **batch**, но не **load**.

Символ **foo** в Maxima соответствует символу $foo в Lisp, а символ Lisp foo соответсвует символу Maxima ?**foo**.

:lisp (**defun** $foo (a) (...)) задает функцию Lisp foo, вычисляющую свои аргументы. Из Maxima функция вызывается записью **foo**(a).

:lisp (**defmspec** $foo (e) (...)) задает функцию Lisp **foo**, откладывающую вычисление аргументов. Из Maxima функция вызывается записью **foo**(a). Аргументами $foo являются (**cdr** e), а (**caar** e) всегда совпадает с $foo.

Запись (**mfuncall** '$foo $a_1 \ldots a_n$) вызывает из Lisp функцию **foo**, определенную в Maxima.

Обратимся к Lisp из Maxima и наоборот.

1. Запись **:lisp** *expr* вычисляет *expr* в интерпретаторе Lisp:

```
(%i1) (aa + bb)^2;
                                      2
(%o1)                        (bb + aa)
(%i2) :lisp $%
((MEXPT SIMP) ((MPLUS SIMP) $AA $BB) 2)
```

2. **:lisp** (**defun** $foo (a) (...)) задает функцию Lisp foo, вычисляющую свои аргументы:

```
(%i1) :lisp (defun $foo (a b) `((mplus) ((mtimes) ,a ,b) $%pi))
$FOO
(%i1) (p: x + y, q: x - y);
(%o1)                            x - y
(%i2) foo (p, q);
(%o2)                  (x - y) (y + x) + %pi
```

3. **:lisp** (**defmspec** $foo (e) (...)) задает функцию Lisp **foo**, откладывающую вычисление аргументов:

```
(%i1) :lisp (defmspec $bar (e) (let ((a (cdr e))) `((mplus) ((mtimes) ,@a) $%pi)))
#<CLOSURE LAMBDA (E) (LET ((A (CDR E))) `((MPLUS) ((MTIMES) ,@A) $%PI))>
(%i1) bar (p, q);
(%o1)                          p q + %pi
(%i2) bar (''p, ''q);
(%o2)                          p q + %pi
```

4. Запись (**mfuncall** '$foo $a_1 \ldots a_n$) вызывает из Lisp функцию **foo**, определенную в Maxima:

```
(%i1) blurf (x) := x^2;
                                        2
(%o1)                     blurf(x) := x
(%i2) :lisp (displa (mfuncall '$blurf '((mplus) $grotz $mumble)))
                 2
(mumble + grotz)
NIL
```