# Построение метаязыковых абстракций с использованием макросов

Артамонов Ю.Н.

Университет "Дубна"
филиал Котельники

12 мая 2017 г.

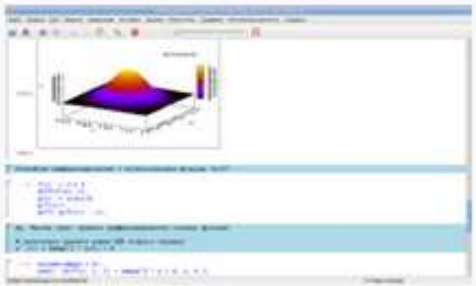
# Пример метаязыковых абстракций - система Maxima

Maxima — свободная система компьютерной алгебры, написанная на языке Common Lisp. Maxima произошла от системы Macsyma, разрабатывавшейся в MIT с 1968 по 1982 годы в рамках проекта Project MAC, финансируемого Министерством энергетики США (DOE) и другими государственными организациями. Профессор Уильям Шелтер (англ. William F. Schelter) из Техасского университета в Остине поддерживал один из вариантов системы, известный как DOE Macsyma, с 1982 года до самой своей смерти в 2001 году.
В 1998 году Шелтер получил от Министерства энергетики разрешение опубликовать исходный код DOE Macsyma под лицензией GPL, и в 2000 году он создал проект на SourceForge.net для поддержания и дальнейшего развития DOE Macsyma под именем Maxima. Maxima имеет широчайший набор средств для проведения аналитических вычислений, численных вычислений и построения графиков. По набору возможностей система близка к таким коммерческим системам, как Maple и Mathematica.

# Полезные ссылки о Maxima

Официальный сайт
Git репозиторий
Исходный код
Maxima online

# Общая информация о Maxima в картинке



Графический интерфейс wxMaxima 0.8.5 на русском языке

| | |
|---|---|
| **Тип** | Система компьютерной алгебры (CAS) |
| **Автор** | *Проект MAC* Массачусетского технологического института |
| **Разработчик** | Уильям Шелтер, сообщество добровольцев |
| **Написана на** | Common Lisp[1] |
| **Интерфейс** | GTK+ и WxWidgets |
| **Операционная система** | Linux, Windows, Mac OS X, FreeBSD, Android[2] |
| **Первый выпуск** | 1982 |
| **Последняя версия** | 5.39.0 (исходный код)<br>5.39.0 (для Windows)<br>5.38.0 (для MacOS)<br>5.39.0 (для Linux)<br>(12.12.2016) |
| **Состояние** | активно |
| **Лицензия** | GNU GPL |
| **Сайт** | maxima.sourceforge.net |

**Maxima** на Викискладе

# Под капотом Maxima

Итак, Maxima полностью написана на Common Lisp. Фактически это означает - работая в системе Maxima, Вы работаете в Common Lisp с его изменённым синтаксисом и массой вспомогательных подгруженных модулей. Например, чтобы локально вычислить выражение в Common Lisp в Maxima можно выполнить команду :lisp.

```
Maxima 5.39.0 http://maxima.sourceforge.net
using Lisp SBCL 1.2.4.debian
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1) :lisp (+ 1 2 3 4)
10
(%i1)
```

# Переход в Lisp

Для глобального перехода в Common Lisp можно использовать команду to_lisp();
После её выполнения, строка приглашения меняется и Вы попадаете в обычный Common Lisp, на котором собрана Maxima. Внимание, она может быть собрана на разных дистрибутивах Common Lisp. В большинстве случаев это не имеет видимой разницы, но не всегда - до тех пор, пока Вы не захотите использовать встроенную функцию конкретного дистрибутива (в лекции в качестве такого дистрибутива используется SBCL). Чтобы вернуться обратно в Maxima, нужно использовать команду (to-maxima). В примере ниже мы перешли в Common Lisp, определили там свою функцию $f(x) = x + 1$, посмотрели, как она работает в Common Lisp, вернулись в Maxima и вызвали ее там. Из примера видно, чтобы использовать функцию Common Lisp в Maxima, перед ее именем нужно ставить знак ?

# Работа в lisp и переход обратно в maxima

```
(%i1) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
MAXIMA> (defun f (x) (+ x 1))
F
MAXIMA> (f 5)
6
MAXIMA> (to-maxima)
Returning to Maxima
(%o1) true
(%i2) ?f(9);
(%o2) 10
(%i3)
```

# Налаживание взаимодействия

Можно также наоборот вызывать полезные функции Maxima из Common Lisp. Для этого перед именем функции нужно ставить знак $. Внимание! При работе в lisp из maxima нельзя использовать в именах переменных заглавные буквы. Посмотрим, например, как использовать в Common Lisp встроенную функцию Maxima разложения числа на простые множители ifactors

```
(%i4) ifactors(9);
(%o4) [[3, 2]]
(%i5) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
MAXIMA> ($ifactors 9)
((MLIST) ((MLIST SIMP) 3 2))
MAXIMA>
```

# Под капотом Maxima (продолжение)

Последний пример показывает, что обычные списки Maxima в Common Lisp представляются с добавлением вспомогательной информации. Это замечание касается любого объекта Maxima.

```
(%i1) h:[1,2,3];
(%o1) [1, 2, 3]
(%i2) :lisp $h
((MLIST SIMP) 1 2 3)
(%i2)
```

# Под капотом Maxima (продолжение)

Попробуем определить для maxima функцию, которая удаляет из списка каждый второй элемент. Вначале посмотрим, как бы эта функция выглядела в Common Lisp:

```
(defun each_two (L)
  (cond
    ((null L) nil)
    ((null (cdr L)) L)
    (t (cons (car L) (each_two (cddr L))))))
```

# Под капотом Maxima (продолжение)

В таком виде в maxima она будет работать неправильно (наоборот удаляет все нечетные). Это связано с тем, что список [1,2,3,4,5] во внутреннем представлении Common Lisp выглядит ((MLIST SIMP) 1 2 3 4 5). А значит все четные: это 1, 3, 5 ))). Кстати, чтобы из Common Lisp использовать выражение maxima, это выражение нужно завернуть в макрос #$Выражение$

```
(%i5) ?each_two([1,2,3,4,5]);
(%o5) [2, 4]
(%i6) :lisp #$[1,2,3,4,5]$
((MLIST SIMP) 1 2 3 4 5)
```

# Под капотом Maxima (продолжение)

Исправим эту ситуацию:

```
(%i6) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
MAXIMA> (defun $each_two (L)
(cons '(MLIST SIMP) (each_two (cdr L))))
$EACH_TWO
MAXIMA> (to-maxima)
Returning to Maxima
(%o6) true
(%i7) each_two([1,2,3,4,5]);
(%o7) [1, 3, 5]
(%i8)
```

# Задания

- Реализовать в Maxima функцию реверса списка.
- Реализовать в Maxima функцию удаления из списка всех отрицательных элементов.
- Реализовать в Maxima функции мемоизации.

# Еще ряд примеров использования

Кроме макроса #$...$, в Common Lisp можно использовать функцию displa, которая выражение из лиспа представляет как в maxima. Например, попробуем использовать встроенную функцию дифференцирования:

```
(%i8) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
MAXIMA> ($diff #$1/3*x ^3 + 1/2*x ^2 + 1$ #$x$)
((MPLUS SIMP) $X ((MEXPT SIMP) $X 2))
MAXIMA> (displa ($diff #$1/3*x^3 + 1/2*x^2 + 1$ #$x$))
```

$$x^2 + x$$

```
NIL
MAXIMA>
```

# Создание образа

Определив в maxima свои полезные функции, можно записать образ всей системы на диск с последующим использованием (предложенный пример работает только с дистрибутивом SBCL):

```
:lisp (sb-ext:save-lisp-and-die "my_core":compression 9 :toplevel #'run
:executable t)
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into my_core:
writing 5680 bytes from the read-only space at 0x20000000
compressed 32768 bytes into 1958 at level 9
writing 3120 bytes from the static space at 0x20100000
compressed 32768 bytes into 868 at level 9
writing 111411200 bytes from the dynamic space at 0x1000000000
compressed 111411200 bytes into 21787208 at level 9
done]
juna@debian: $
```

# Пробуем использовать созданный образ

```
juna@debian: $ ./my_core
Maxima restarted.
(%i9) each_two([1,2,3,4,5,6,7,8]);
(%o9) [1, 3, 5, 7]
(%i10)
```

# Еще одна полезная вещь - удаленный сервер maxima

Можно запустить Maxima на удаленном компьютере и получать доступ к нему через telnet-клиента. В Linux сервер располагается по пути /usr/share/maxima/5.34.1/share/contrib/maxima-server.lisp, там же инструкция по его использованию.
Чтобы его запустить, выполняем следующие команды:
load("maxima-server.lisp");
:lisp (server-run)
Теперь пробуем подключиться:
juna@debian: $ telnet localhost 1042
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Maxima restarted.
(%i10) 9+8;
(%o10) 17
(%i11) quit();
Connection closed by foreign host.

# Метапрограммирование на Common Lisp

Метапрограммирование — вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы (в частности, на стадии компиляции их исходного кода), либо программ, которые меняют себя во время выполнения (самомодифицирующийся код).
Многие идеи, появившиеся в Лиспе, со временем перешли или переходят в другие языки программирования. Но есть ряд идей, оставляющие Лисп в некотором смысле уникальным. Речь идет о его системе макросов. Вы привыкли к этому термину, но в Лиспе под этим понимают совсем другое. Многие считают, что вся мощь Лиспа заключается именно в его макросах (к которой ни один другой язык даже не пытается приблизиться), а если смотреть более широко, то в расширяемости компилятора. По сути макросы позволяют изменять интерпретатор Лиспа, добавляя новые синтаксические конструкции. Таким образом, они делают этот язык перепрограммируемым.

# Макросы Common Lisp

Синтаксис определения макроса выглядит почти так же, как синтаксис используемой при определении функций формы Defun:
(defmacro имя_макроса (аргументы) (тело_ макроса))
На первый взгляд отличия состоят в том, что вместо служебного слова defun стоит defmacro. Но на самом деле отличия глубже — вычисление макроса состоит из двух этапов:

- раскрытие макроса — на этом этапе формируется выражение, которое потом будет вычисляться;
- вычисляется полученное выражение.

В чем преимущества такого подхода? Дело в том, что поскольку вычисление отложено, вы можете заставить программу писать программу саму для себя. Здесь стирается грань между данными для программы и самой программой.

# Макросы Common Lisp (первый пример)

Рассмотрим простой пример. Нужно заданной переменной присвоить 0. Это сразу приводит к функции:
(defun fnil (var) (setq var 0))
или к макросу
(defmacro mnil (var) (list 'setq var 0)).
Таким образом, если для функции мы сразу имеем выражение (setq var 0), то для макроса мы просто формируем это выражение по правилам Лиспа, т.е. на первом этапе макрос mnil получит тоже самое выражение, а потом его вычислит. В чем преимущества? Посмотрим, как это будет работать:
>(fnil r)
переменной R не присвоено значение
>(fnil 'r)
0
>(mnil d)
0
Поскольку в функцию должен передаваться символ, для переменной нужна блокировка, несмотря на наличи в теле setq, блокирующей вычисление первого аргумента. Это происходит потому, что перед тем как вычислить функцию fnil интерпретатор пытается найти значение переменной r. В макросе mnil этого не происходит по той простой причине, что вначале ничего не вычисляется, а просто формируется выражение и осуществляются подстановки символов в него.
Даже в этом простом примере видно, что мы нарушили устоявшийся синтаксис Лисп. С помощью макросов мы можем изменять этот синтаксис по своему усмотрению.

# Макросы Common Lisp (второй пример)

Рассмотрим более сложный пример. Пусть требуется списку переменных присвоить 0. Это приводит к следующей функции:

```
(defun fnil_list (var)
  (if (null var)
    nil
    (let () (set (car var) 0) (fnil_list (cdr var)))))
```

Проверяем

```
>(fnil_list '(a s d))
nil
>a
0
>d
0
```

Все работает, но мы вынуждены использовать блокировку. Построим макрос, который исключает этот недостаток:

```
(defmacro mnil_list (var)
  (list 'fnil_list (list 'quote var)))
```

Теперь мы формируем список из символов fnil_list и списка из символа quote (наша блокировка ') и фактического списка переменных. В результате, например, при обращении

```
>(mnil_list (d f g))
```

строится тело (fnil_list '(d f g)), которое затем и вычисляется.

# Макросы Common Lisp (второй пример)

Можно, конечно, обойтись без использования функции fnil_list и само определение этой функции построить в теле макроса:

```
(defmacro mnil_list (&rest var)
  (list 'cond
    (list (list 'null (list 'quote var)) 'nil)
  (list 't (list 'progn
    (list 'set (list 'car  (list 'quote var)) 0)
  (cons 'mnil_list (cdr var))))))
```

Этот макрос фактически строит тело функции fnil_list, но в отличии от предыдущих, позволяет даже перечислять переменные вне списка (mnil_list a s d)

# Ключ - переменное количество параметров

Это достигается использованием специального ключа &rest — переменное количество параметров. Фактически, все, что стоит после &rest объединяется в список — это и обеспечивает переменное количество. Например,

```
(defun car_arg (&rest arg) (car arg))
(defun cdr_arg (&rest arg) (cdr arg))
>(car_arg 1 2 3 4)
1
>(cdr_arg 1 2 3 4)
(2 3 4)
```

# Использование обратной блокировки при построении макросов

Пример последнего макроса mnil_list показывает, что формировать тело макроса не так то просто, как правило, при таком подходе тело макроса содержит большое количество list, cons и других функций, формирующих списки. Это неудобно. Поэтому был предложен несколько другой подход к формированию тела макроса — использование символа ‘ (на клавиатуре набирается там, где русская буква ё или тильда ∼). Это так называемый символ обратной блокировки. Выражение, заблокированное символом обратной блокировки воспринимается просто как последовательность символом, поэтому тело макроса мы можем не строить, а просто написать, что должно получиться. Может возникнуть вопрос, а почему для этого не подходит обычная блокировка quote (или ее синтаксический сахар ’). Дело в том, что обратная блокировка позволяет, где нужно разблокировать вычисление выражения. Для этого перед этим выражением ставится запятая. Рассмотрим как это работает на ряде примеров.

# Использование обратной блокировки при построении макросов

```
>(setq a 1 b 2 c 3) ;присваиваем a=1, b=2, c=3
3
>'(a b c)
(a b c)
>`(a ,b ,c)
(a 2 3)
```

Теперь возникает вопрос, а где же в макросе может потребоваться разблокировать вычисление выражения. Опять рассмотрим пример макроса mnil. Ранее мы определили его (defmacro mnil (var) (list 'setq var 0)) А теперь делаем так:

```
(defmacro !nil (var)
   `(setq ,var 0))
```

```
> (!nil x) 0
>x
0
```

Как видим, для того чтобы взять сам аргумент, который передается в макрос, и построить выражение (setq x 0) , мы должны разблокировать аргумент var, иначе мы получим форму (setq var nil).

# Более сложные примеры - модификация if

Теперь рассмотрим более сложные примеры. Мы знаем конструкцию if в Лиспе, она обладает одним недостатком — в ветке then и ветке else может быть вычислена только одна форма. Исправим эту ситуацию, введя дополнительный макрос if1

;;Макрос if1 позволяет выполнять последовательность форм в ветке then

;;и последовательность форм в ветке else

```
(defmacro if1
  (conditions then-action &optional else-action )
  `(if ,conditions
    (let ()
      (mapcar 'eval ',then-action))
    (let ()
      (mapcar 'eval ',else-action))))
```

# Более сложные примеры - модификация if

Данный макрос строит выражение
(if condition (вычисляются формы ветки then и формируется список из значений)
иначе ( вычисляются формы ветки else и формируется список из значений))
В макросе есть некоторые новые синтаксические конструкции:
&optional - ключевое слово, после которого указываются необязательные параметры (в нашем случае макрос if1 должен корректно работать и в случае отсутствия ветки else)
mapcar — это так называемый отображающий функционал, позволяет применить функцию к списку. Например, (mapcar 'atom '(2 3 a)) даст (T T T)
eval — функция, снимающая блокировку вычислений '
>(if1 t ((setq x 4) (+ x 4)) ((log 5) (sin 5)))
(4 8)
> (if1 nil ((setq x 4) (+ x 4)) ((log 5) (sin 5)))
(1.609438 -0.9589243)

# Более сложные примеры - реализация progn

В следующем примере расширим синтаксис Лиспа конструкцией, позволяющей последовательно вычислять формы и возвращать в качестве результата значение заданной формы. Например, нам хочется в программе вычислить последовательность форм (setq x 3) (setq y (+ x 1)) и вернуть значение первой. В этом есть смысл — обе формы дают побочный эффект. Наша реализация будет такой:

```
(defmacro prg (k &rest body)
`(nth (- ,k 1) (list ,@body)))
```

k — номер формы, значение которой следует возвращать;
&rest — ключевое слово, после которого описывается переменное количество параметров (в нашем случае количество форм которые нужно вычислить может быть любым);
,@ - обратную блокировку можно использовать совместно с символом @, тогда полученное выражение присоединяется к конечному выражению без скобок.

# Более сложные примеры - реализация progn

Например,

```
>(setq x '(новый))
(новый)
>`(,x элемент)
((новый) элемент)
>`(,@x элемент)
(новый элемент)
```

Рассмотрим пример использования этого макроса:

```
>(prg 1 (setq x 3) (setq y (+ x 1)))
3
>y
4
>(prg 2 (setq x 3) (setq y (+ x 1)))
4
>x
3
```

# Более сложные примеры - реализация progn

В данном случае аргумент &rest это список ((setq x 3) (setq y (+ x 1))), чтобы последовательно вычислить эти формы, нужно освободиться от внешних скобок, поэтому и используется ,@. На самом деле в Лиспе уже есть аналогичные управляющие структуры:

- progn — вычисляет последовательность форм и возвращает значение последней формы;
- prog1 — вычисляет последовательность форм и возвращает значение первой;
- progk — вычисляет значение форм и возвращает значение k-й, где k -некоторое число.

# Более сложные примеры - реализация условного цикла

Следующий макрос реализует условный цикл

```
;;;Макрос реализует условный цикл loop
(defmacro loop1 (&rest body)
`(if (equal (progn ,@body) 'ret) 'ok (loop1 ,@body)))
```

Здесь вычисление переменного количества форм body производится до тех пор, пока значение последней формы не станет равным слову ret. После этого происходит выход из цикла:

```
>(setq x 1)
1
> (loop1 (setq x (+ x 1)) (setq y (expt x 3)) (if (> x 4) 'ret) )
ok
>x
5
>y
125
```

Аналогичный цикл уже реализован в Лисп - это цикл Loop, там выход происходит, если получен return

# Более сложные примеры - безусловный цикл for

Приведенные примеры хорошо демонстрируют расширяемость синтаксиса Лисп. Не будем останавливаться на достигнутом, реализуем в Лиспе некоторые синтаксические конструкции, характерные для языка Pascal. Во-первых, реализуем оператор безусловного цикла for

```
(setq = '=)
(setq do 'do)
(setq to 'to)
;;;Макрос for .. to .. do
(defmacro for (i = n1 to n2 do &rest body)
  `(if (and (equal '= ,=)
    (equal 'to ,to) (equal 'do ,do))
    (let ((,i ,n1))
      (loop1
        ,@body
        (setq ,i (+ 1 ,i))
        (if (> ,i ,n2) 'ret)))
    (print 'Error)))
```

# Более сложные примеры - безусловный цикл for (пример использования)

Вот пример использования

```
>(for i = 1 to 10 do (if (= (rem i 2) 0) (print i)))
2
4
6
8
10
OK
```

Печатаются все четные числа в диапазоне от 1 то 10.

# Более сложные примеры - реализация множественного выбора case

Наконец реализуем множественный выбор case

```
(defmacro case1 (key &rest body)
  `(cond
    ((null ',body) nil)
    ((member1 ,key (car (car ',body)))
      (eval (cons 'progn (cdr (car ',body)))))
    (t `,(eval `,(cons 'case1
      (cons ,key (cdr ',body)))))))
(defun member1 (a L)
  (cond
    ((null L) nil)
    ((equal a (car L)) T)
    (t (member1 a (cdr L)))))
```