SLUD 2006

# Programación de Máxima

In memoriam William Schelter

Robert Dodier
Proyecto Máxima

## Toda cosa es expresión

Toda cosa en Máxima (casi todas) es un objeto de forma **foo**$(a, b, c)$, es decir, una expresión que comprende un operador como **foo** y sus argumentos $a, b, c$

Programación en Máxima es más o menos la creación y modificación de expresiones

(Se puede modificar la aparencia de una expresión por medida de los códigos mostrantes. Sin eso, toda cosa aparecería como **foo**$(a, b, c)$.)

## Estructura de expresiones: op y args

Funciones normales

```
(%i1) expr : Foo (x, y);
(%o1)                          Foo(x, y)
(%i2) op (expr);
(%o2)                             Foo
(%i3) args (expr);
(%o3)                           [x, y]
(%i4) apply (op (expr), args (expr));
(%o4)                          Foo(x, y)
```

## Estructura ...

Expresiones con operadores

```
(%i5) expr : 1 + %pi + %e + x + a;
(%o5)                      x + a + %pi + %e + 1
(%i6) [op (expr), args (expr)];
(%o6)                    [+, [x, a, %pi, %e, 1]]
```

## Estructura ...

```
(%i1) [expr_1 : 'sum (F(i), i, 1, n), "    ",
       expr_2 : 'integrate (sin(x), x, 0, %pi)];
                 n                    %pi
                ====                 /
                \                    [
(%o1)           [ >    F(i),      , I    sin(x) dx]
                /                    ]
                ====                 /
                i = 1                 0
(%i2) [op (expr_1), args (expr_1)];
(%o2)                  [sum, [F(i), i, 1, n]]
(%i3) [op (expr_2), args (expr_2)];
(%o3)             [integrate, [sin(x), x, 0, %pi]]
```

## Estructura ...

Objetos: lista, conjunto, matriz son expresiones

```
(%i11) L : [1, 2, u, v];
(%o11)                          [1, 2, u, v]
(%i12) [op (L), args (L)];
(%o12)                        [[, [1, 2, u, v]]
(%i13) S : {a, b, c, x, y};
(%o13)                         {a, b, c, x, y}
(%i14) [op (S), args (S)];
(%o14)                      [set, [a, b, c, x, y]]
(%i15) M : matrix ([1, 2], [a, b]);
                                [ 1  2 ]
(%o15)                          [      ]
                                [ a  b ]
```

```
(%i16) [op (M), args (M)];
(%o16)                [matrix, [[1, 2], [a, b]]]
```

## Estructura ...

Condicional **if** – **then**, bucle **for** son expresiones

```
(%i2) expr_1 : '(if x > a then F(x) else G(x));
(%o2)   if x > a then F(x) else G(x)
(%i3) expr_2 : '(for i:10 thru 1 step -1
       do suma : suma + F(i));
(%o3) for i from 10 step - 1 thru 1
                          do suma : suma + F(i)
(%i4) [op (expr_1), args (expr_1)];
(%o4)  [if, [x > a, F(x), true, G(x)]]
(%i5) [op (expr_2), args (expr_2)];
(%o5) [mdo, [i, 10, - 1, false, 1, false,
                             suma : suma + F(i)]]
```

## Evaluación y simplificación

Evaluación = sustitución de valores por símbolos y invocación de funciones

Simplificación = sustitución de expresiones por equivalentes

Evaluación cambia el valor de una expresión, mientras que simplificación cambia la forma

## Más detalles de evaluación

Se puede aumentar o evitar evaluación por medida de varias opciones

Operador comilla ' evita evaluación de un símbolo o expresión. Ejemplos: $a : 17;\ 'a;\ \Rightarrow a$, $a : 17; b : 29;\ '(a + b);\ \Rightarrow a + b$

Operador comilla trasforma una función hasta "expresión nombre" (como opuesto a "expresión verba"). Ejemplo: $f(x) := 1 - x;\ 'f(a);\ \Rightarrow f(a)$

Operador comilla-comilla ' ' (dos comillas simples) causa una más evaluación cuando se encuentra en una expresión entregada

Función **ev** causa una más evaluación cada vez que se evalua la expresión

## Más detalles de simplificación

Identidades matemáticas son expresadas como simplificación en Máxima. Ejemplo: $x + x \Rightarrow 2x$

La distinción entre evaluación y simplificación es un poco nublado. Ejemplos: En Máxima, $1 + 1 \Rightarrow 2$ es una simplificación. También $\sin(1.0) \Rightarrow 0.84147$

Simplificación se hace con funciones asociadas al nombre de una función o operador. Se puede asociar nuevamente una función (que se llama una *régula de simplificación*) con cualquier función o operador por **tellsimp** y **tellsimpafter**.

## Más detalles . . .

Se puede aplicar simplificación a un operador o una función que no existe. Simplificación maneja una expresión sin que invoque funciones que aparezcan en ella; es sólo una manipulación formal.

Hay muchas identidades que no son aplicadas automaticamente; el usuario tiene que pedirlas especificamente.

Ejemplos: **ratsimp**$(a/x + b/x); \Rightarrow (b + a)/x$
**trigreduce**$(2 \sin x \cos x); \Rightarrow \sin(2x)$

## Funciones cotidianas

Definición de una función normal. A la derecha se pone sólo una expresión; se puede agrupar multiples expresiónes con **block**.

```
(%i2) foo_bar (x, y) := y - x;
(%o2)      foo_bar(x, y) := y - x
(%i3) baz_quux (a, b, c) :=
        block ([u, v], u : b - c, v : a - b, u/v);
(%o3) baz_quux(a, b, c) :=
                                                u
       block([u, v], u : b - c, v : a - b, -)
                                                v
```

## Funciones cotidianas ...

```
(%i4) foo_bar (1729, %pi);
(%o4)                 %pi - 1729
(%i5) baz_quux (%pi, %e, %i);
                      %e - %i
(%o5)                 --------
                      %pi - %e
```

No es necesario poner **return** (devolver) en una función. Se devuelve el valor computado por la expresión a la derecha.

## Funciones cotidianas ...

No se evalua la expresión a la derecha en el momento en que se define la función. El operador comilla-comilla (dos comillas simples) causa evaluación.

```
(%i6) integrate (2 * sin(x) * cos(x), x);
                              2
(%o6)                  - cos (x)
(%i7) F (x) := integrate (2 * sin(x) * cos(x), x);
(%o7) F(x) := integrate(2 sin(x) cos(x), x)
(%i8) F (42);
Attempt to integrate wrt a number: 42
#0: F(x=42)
 -- an error.
```

## Funciones cotidianas ...

¡Qué lástima! Esperábamos el resultado de la integración. Obtenemos el resultado por el operador comilla-comilla.

```
(%i9) F (x) :=
       ''(integrate (2 * sin(x) * cos(x), x));
                              2
(%o9)           F(x) := - cos (x)
(%i10) F (42);
                          2
(%o10)                - cos (42)
```

Para indicar que el número de argumentos no es fijo, se escribe un argumento como lista en la definición. Tal argumento tiene que ser final o el único argumento. Cuando está invocada la función, el argumento aparece como lista.

```
(%i2) G (x, y, [z]) := if x > y then first (z)
        else last (z);
(%o2) G(x, y, [z]) :=
          if x > y then first(z) else last(z)
(%i3) H ([a]) := map (sin, a);
(%o3)       H([a]) := map(sin, a)
(%i4) G (17, 29, aa, bb, cc);
(%o4)                   cc
(%i5) G (29, 17, aa, bb, cc);
(%o5)                   aa
(%i6) H (1, 2, 3, a, b, c);
(%o6) [sin(1), sin(2), sin(3), sin(a),
                                sin(b), sin(c)]
```

## Programación funcional

Es decir, programación con enfásis en funciones

**apply**: aplicar una función a sus argumentos

```
(%i1) F : Foo;
(%o1)                                  Foo
(%i2) a : [1, 2, z];
(%o2)                                [1, 2, z]
(%i3) apply (F, a);
(%o3)                             Foo(1, 2, z)
```

## Programación funcional ...

**map**: aplicar una función a una lista de argumentos. (1) Es muy común que reemplace bucles **for** con **map**. (2) No es necesario definir una función prestando atención en objetos compuestos — usemos **map**.

```
(%i2) L : [a, b, c, x, y, z];
(%o2)          [a, b, c, x, y, z]
(%i3) sin (L);
(%o3)       sin([a, b, c, x, y, z])
(%i4) map (sin, L);
(%o4) [sin(a), sin(b), sin(c), sin(x),
                                sin(y), sin(z)]
```

## Programación funcional ...

**lambda**: crear una función sin nombre

```
(%i2) map (lambda ([x], is (x > 0)),
        [-2, -1, 0, 1, 2, 3]);
(%o2) [false, false, false, true, true, true]
(%i3) subset ({x, y, %pi, 1.729, 42, 47},
        lambda ([a], integerp (a)));
(%o3)                  {42, 47}
```

## Funciones de array

(1) $F[x] := \ldots$ indica una función "de memoria", es decir, que recuerda resultados anteriores

```
(%i1) F[x] := block (print ("Saludos!"), x^2 + 1);
                                                  2
(%o1)          F  := block(print("Saludos!"), x  + 1)
                x
(%i2) [F[10], F[20], F[30]];
Saludos!
Saludos!
Saludos!
(%o2)                       [101, 401, 901]
(%i3) [F[10], F[20], F[30]];
(%o3)                       [101, 401, 901]
```

(2) $G[x](y) := \ldots$ indica una función "de array". Es una diferente función de $y$ por cada valor de $x$.

```
(%i6) G[x](y) := x^y;
                                    y
(%o6)                          G (y) := x
                                x
(%i7) G[1];
(%o7)                        lambda([y], 1)
(%i8) G[2];
                                     y
(%o8)                        lambda([y], 2 )
(%i9) G[3];
                                     y
(%o9)                        lambda([y], 3 )
```

```
(%i10) G[3](u + v);
                                    v + u
(%o10)                             3
(%i11) map (G[3], [2, 3, 5, 7]);
(%o11)                      [9, 27, 243, 2187]
```

## Operadores

Es muy fácil declarar nuevos operadores

```
(%i1) prefix ("FOO");
(%o1)                                FOO
(%i2) infix ("##");
(%o2)                                ##
(%i3) nary ("@@");
(%o3)                                @@
(%i4) postfix ("%!");
(%o4)                                %!
(%i5) matchfix ("!<", ">!");
(%o5)                                !<
(%i6) FOO a;
(%o6)                              FOO a
```

```
(%i7) a ## b;
(%o7)                                a ## b
(%i8) a @@ b @@ c @@ d;
(%o8)                          a @@ b @@ c @@ d
(%i9) 42 %!;
(%o9)                                 42 %!
(%i10) !< 17, 29 >!;
(%o10)                              !<17, 29>!
```

## Operadores ...

Se puede definir (o no) una función correspondente al operador.

```
(%i11) "##" (x, y) := y/x;
                                          y
(%o11)                          x ## y := -
                                          x
(%i12) "@@"([L]) := apply("+", L) / apply("*", L);
                                     apply("+", L)
(%o12)                 "@@"([L]) := -------------
                                     apply("*", L)
```

## Operadores ...

```
(%i13) (a + b) ## c;
                                    c
(%o13)                             -----
                                   b + a
(%i14) 17 @@ 29 @@ x @@ y @@ z;
                                z + y + x + 46
(%o14)                          --------------
                                  493 x y z
```

## Lisp y Máxima

Referencia a variables en Lisp desde Máxima. Variables en Máxima normalmente tiene símbolo dolar inicial en Lisp. Si no, tiene que poner signo de interrogación inicial en Máxima.

```
(%i1) :lisp (defvar foo-bar 1729)
FOO-BAR
(%i1) :lisp (defvar $foo_bar '$z)
$FOO_BAR
(%i1) ?foo\-bar + foo_bar;
(%o1)                          z + 1729
```

## Lisp y Máxima ...

Referencia a variables en Máxima desde Lisp. Como siempre tenemos que prestar atención en el primer carácter de los símbolos.

```
(%i1) x : 1729;
(%o1)                                1729
(%i2) y : a * b;
(%o2)                                 a b
(%i3) :lisp (m+ $x $y)
((MPLUS SIMP) 1729 ((MTIMES SIMP) $A $B))
```

## Lisp y Máxima ...

Toda expresión (casi todas) en Maxima se representa en Lisp como ((FOO) A B C) donde FOO es nombre de función o operador y A B C son los argumentos

```
(%i1) expr_1 : foo (1, 2, 3)$
(%i2) expr_2 : a * b * c$
(%i3) expr_3 : (a + b) * (17 + c)$
(%i4) :lisp $expr_1
(($FOO SIMP) 1 2 3)
(%i4) :lisp $expr_2
((MTIMES SIMP) $A $B $C)
(%i4) :lisp $expr_3
((MTIMES SIMP) ((MPLUS SIMP) $A $B)
    ((MPLUS SIMP) 17 $C))
```

## Lisp y Máxima ...

Invocación de una función en Lisp desde Máxima. Una función normal en Lisp es una función normal en Máxima también.

```
(%i1) :lisp (defun $baz_quux (x y) (m+ x y))
$BAZ_QUUX
(%i1) baz_quux (42, a * b);
(%o1)                          a b + 42
```

## Lisp y Máxima ...

Invocación de una función en Máxima desde Lisp. Una función de Máxima no es una función normal en Lisp. Tiene que invocarla por **mfuncall**.

```
(%i1) f (x, y) := x * y;
(%o1)                          f(x, y) := x y
(%i2) :lisp (mfuncall '$f 42 '$z)
((MTIMES SIMP) 42 $Z)
```

## Funciones que no evaluan sus argumentos

La mayor parte de funciones en Máxima evaluan sus argumentos antes de procesarlos

Pero hay algunos (como **save** (almacenar), **kill** ("matar", borrar)) que no los evaluan

E.g. en **save**("foo.datos", $a, b, c$) queremos indicar a **save** almacenar $a, b, c$ (símbolos), no sus valores

Tales funciones se definen con **defmspec** en Lisp

Argumentos se suplican a tal función ni evaluados ni simplificados

Se puede causar evaluación y simplificación por comilla-comilla o **apply**($foo, [a, b, c]$)

## Ejemplo: cantidades con unidades

```
(%i24) densidad : 52 ‘ (kg/m^3);
                                    kg
(%o24)                         52 ‘ (--)
                                     3
                                    m
(%i25) volumen : V ‘ (m^3);
                                    3
(%o25)                         V ‘ (m )
(%i26) masa : densidad * volumen;
(%o26)                        (52 V) ‘ kg
(%i27) [cantidad (masa), unidad (masa)];
(%o27)                        [52 V, kg]
```

## Ejemplo: cantidades ...

Declaración de ‘ (comilla al revés) como operador

```
infix ("‘");
```

Régulas de simplificación: reconocer cantidad con unidad

```
matchdeclare (uu, unidadp, nn,
    lambda ([e], not unidadp (e)));
tellsimpafter (uu^xx,
    (cantidad(uu)^xx) ‘ (unidad(uu)^xx));
tellsimpafter (nn * uu,
    multiplicar_unidades (nn, asegurar_lista (uu)));
```

## Ejemplo: cantidades ...

Funciones: hacer las operaciones

```
unidadp (e) := not atom(e) and op(e) = "‘";
no_unidad_no_1p (e) := not unidadp (e) and e # 1;
cantidad (e) := first (e);
unidad (e) := second (e);
multiplicar_unidades (nn, uu) :=
    nn * (apply ("*", map (first, uu))
            ‘ apply ("*", map (second, uu)));
asegurar_lista (e) :=
    if atom(e) or not op(e) = "*"
        then [e] else args(e);
```

## Otros sujetos: Depuración

**trace**: rastreo de funciones en Máxima

**:lisp trace**: rastreo de funciones en Lisp

Hay otros funciones de depuración, pero no los uso mucho

## Otros sujetos: Errores y excepciones

**errcatch**: coger a un error

**throw** – **catch**: causar un retorno no local. **throw** echa una expresión hasta el **catch** (coger) más cerca

Se puede usar **throw** – **catch** para recuperarse de errores (en vez de simplemente pararse) pero no se usa mucho en los códigos existentes

## Otros sujetos: Funciones "macros"

Una función que no evalua sus argumentos y que reevalua su valor resultado se llama un "macro" (por razones no muy claros)

Un macro es apto cuando quiere controlar la evaluacion muy cuidadosamente los argumentos

E.g. definición de una nueva estructura de control — tiene que evitar la evaluación de los argumentos hasta el momento apropriado

# Recursos para el programador

Resumen de organización interna:
http://maxima.sourceforge.net/wiki/index.php/outline%20of%20Maxima%20internals

Más detalles de organización interna:
http://maxima.sourceforge.net/wiki/index.php/Maxima%20internals

Otro sumario: http://maxima.sourceforge.net/docs/tutorial/en/minimal-maxima.pdf

Mirador de CVS (códigos fuentes): http://maxima.cvs.sourceforge.net/maxima/maxima

Portada del proyecto (versión castellano): http://maxima.sourceforge.net/es

Página del proyecto en SourceForge: http://sourceforge.net/projects/maxima

Documentación (versión castellano): http://maxima.sourceforge.net/es/docs.shtml

Errores: http://sourceforge.net/tracker/?group_id=4933&atid=104933

Correos electronicos (versión castellano):
http://lists.sourceforge.net/lists/listinfo/maxima-lang-es