Rules and patterns in Maxima

Michel Talon

February 19, 2019

1 Introduction.

Rules and patterns have been introduced in Maxima by prof. R. Fateman (in the program matcom.lisp) and are described in a particularly abstruse and compact chapter of the manual. The present discussion is intended to be a more tutorial like introduction to the subject, but also to the lisp program underlying this rules system. It is based on the manual and comments by R. Fateman and R. Dodier in the Maxima mailing list, but for sure many points in the manual and the mailing list are not discussed here.

The subject deals about extracting patterns in expressions and transforming them according to replacement patterns. This allows an expression oriented way of programmation, quite different from the procedure oriented way. This way of manipulating expressions is emphasized in mathematica. An example of this declarative way of doing algebra is given in the presentation of quaternions at the end. Of course under the hood, procedures do the transformation job.

2 The matcher.

The first step is to define matching variables, sort of wildcards which will select the appropriate parts of the considered expressions. The main way to do that is the matchdeclare function, which declares such wild cards, associated to conditions allowing the variable to match or not, also called predicates. For example i want variable a and b to match atoms except numbers, variable c to match non atoms and variable d to match numbers. If a variable is to match anything then put `all` as predicate, or `true`. The syntax is

```
matchdeclare(a,pred1,b,pred1,c,pred2,d,numberp);
```

pred1 is a logical function taking one variable which will be applied to a, then b, etc. For d one can take simply numberp. However pred2 should be `not atom` and this doesn't work. The simplest is to define a Maxima function

```
pred2(x):= not atom(x);
```

or a lambda function lambda([x],not atom(x)) that one puts in place of pred2. Finally

```
pred1(x):=atom(x) and not numberp(x);
```

Advanced remark: there is a more general form for predicates, which allow them to test several wildcards at once. For example, as in the manual:

```
(%i1) gt (i, j) := integerp(j) and i < j;
(%o1)           gt(i, j) := integerp(j) and i < j
(%i2) matchdeclare (i, integerp, j, gt(i));
(%o2)                         done
```

In this situation, where the predicate for j is gt(i), the system adds j at the end of the arguments of gt, and thus runs gt(i,j) so j matches only when j>i.

With this in place the effect of matchdeclare is to put information in the property list of the wild card variables:

```
(%i1) pred1(x):=atom(x) and not numberp(x);
(%o1)              pred1(x) := atom(x) and (not numberp(x))
(%i2) pred2(x):= not atom(x);
(%o2)                        pred2(x) := not atom(x)
(%i3) matchdeclare(a,pred1,b,pred1,c,pred2,d,numberp);
(%o3)                                 done
(%i4) :lisp(symbol-plist '$a)
(MPROPS (NIL MATCHDECLARE ($PRED1)))
```

A pattern is a Maxima expression involving in general several variables, various so-called kernels (such as sin, cos, exp, etc.) some of these variables being the above wildcards. The aim of a pattern matcher is to determine if a given expression is of the same form as the pattern, with wildcards substituted by actual values while other variables are matched by themselves.

To see how this matching occurs let us use the defmatch function that will be described later on, but in its most simple incarnation which reads:

```
(%i4) defmatch(r1,a*b*d);
(%o4)                        r1
```

One will also use the function defrule which can be used to produce essentially the same effect:

```
(%i5) defrule(r2,a*b*d,['a = a,'b = b,'d = d]);
(%o5)               r2 : a b d -> ['a = a, 'b = b, 'd = d]
Let us try to match 2*x*y with these rules. One would like to get x in a
y in b (or the converse) and 2 in d. Instead one gets:
(%i6) r1(2*x*y);
(%o6)                                 false
(%i7) r2(2*x*y);
(%o7)                                 false
```

To understand this, let us relax the predicate on variable a, note it is sufficient to redeclare matchdeclare and redefine the rule:

```
(%i8) matchdeclare(a,atom,b,pred1,c,pred2,d,numberp);
(%o8)                                done
(%i9)  defmatch(r1,a*b*d);
(%o9)                                 r1
(%i10) r1(2*x*y);
(%o10)                      [d = 2, b = x y, a = 1]
(%i11) r2(2*x*y);
(%o11)                                false
(%i12) defrule(r2,a*b*d,['a = a,'b = b,'d = d]);
(%o12)              r2 : a b d -> ['a = a, 'b = b, 'd = d]
(%i13) r2(2*x*y);
(%o13)                      [a = 1, b = x y, d = 2]
(%i14) r2(2*x*y*z);
(%o14)                     [a = 1, b = x y z, d = 2]
(%i15) disprule(r1);
(%t15)                         r1 : a b d -> []
(%o15)                              [%t15]
(%i16) :lisp(symbol-plist '$r1);
(MPROPS (NIL $RULE ((MLIST) ((MTIMES SIMP) $A $B $D) ((MLIST)))))
```

We can now explain various facets of the matching mechanism. First defmatch (or defrule) creates a matching rule r1, and puts on the property list of r1 an MPROP property with value the thing to be matched and result a Maxima list. the command disprule says the same thing in Maxima language. Note that r1 has in its definition access to the appropriate matching variables and thus their property list, which contain the corresponding predicates. The command disprule gives the same information in Maxima language. Second one can modify the declaration of match variables but one has to redefine the matching rules for the new predicate to take effect (see %o11 and %o12).

In the case of %o10 the matcher has correctly associated d to the number 2. The matcher has shuffled terms around using product commutativity to satisfy the predicates, it is not a purely formal matcher like the one in grep, awk, etc. but it uses knowledge of algebraic properties to do matching.

It is more puzzling that a=1 and b=x*y. Recall that b matches symbols that are not numbers. It is a special case that for the addition and the multiplication which are commutative and nary, the matcher is **greedy** and matches the complete string of symbols appearing in the addition or the multiplication. For example in %o14 it matches the 3 symbols x,y,z. Now what about a? The matcher is able to "invent" that xyz = 1*xyz and then tries to associate a with 1. In the first matchdeclare a is forbidden to be a number. But this matcher has **no backtracking capability**, so having set b=xy it will not come back to set a=x, b=y, hence the matcher fails. This explains the results in %o6 and %o7. When we relax the condition on a, the matcher can set a=1 and then everything works OK.

Remark: on the order of matching predicates. The documentation says: In the case of addition and multiplication, the match variable may be assigned a single expression which satisfies the match predicate, or a sum or product (respectively) of such expressions. Such multiple-term matching is greedy: predicates are evaluated in the order in which their associated variables

appear in the match pattern, and a term which satisfies more than one predicate is taken by the first predicate which it satisfies. Each predicate is tested against all operands of the sum or product before the next predicate is evaluated. In addition, if 0 or 1 (respectively) satisfies a match predicate, and there are no other terms which satisfy the predicate, 0 or 1 is assigned to the match variable associated with the predicate. But the order of terms in the match pattern is altered by simplification, so that one has no control over it, excepting simp:false. Illustration:

```
(%i1) matchdeclare([a,b],atom);
(%o1)                                done
(%i2) defmatch(r,a*b);
(%o2)                                 r
(%i3) r(x*y);
(%o3)                           [b = x y, a = 1]
(%i4) defmatch(r,b*a);
(%o4)                                 r
(%i5) r(x*y);
(%o5)                           [b = x y, a = 1]
(%i6) simp:false$ defmatch(r,b*a); simp:true$
(%o7)                                 r
(%i9) r(x*y);
(%o9)                           [a = x y, b = 1]
(%i10)
```

Apparently predicates are in fact tested in the inverse order of their appearance in the matching pattern:

```
(%i5) defmatch(r,a*b*c);
(%o5)                                 r
(%i6) r(x*y*z);
(%o6)                     [c = x y z, b = 1, a = 1]
```

When the formula to be matched is not a sum or product things work in a more natural way, for example considering a dot product:

```
(%i16) matchdeclare(a,pred1,b,pred1,c,pred2,d,numberp);
(%o16)                               done
(%i18) defmatch(r1,a.b*d);
(%o18)                                r1
(%i19) r1(2*x.y);
(%o19)                       [d = 2, b = y, a = x]
```

Finally, when we have both atoms and non atoms such as sin(x) or sin(y), the matcher is able to discern between the two in a sum or a product:

```
(%i20) matchdeclare(a,pred1,b,pred1,c,pred2,d,numberp);
(%o20)                               done
(%i21) defmatch(r2,b*c*d);
(%o21)                                r2
```

```
(%i22) r2(2*x*y*sin(u)*cos(w));
(%o22)                [d = 2, c = sin(u) cos(w), b = x y]
```

So the matcher has attributed the product of all atoms (not numbers) to b, the product of all non atoms to c and the number to d. In general it will partition the expression in three components corresponding to mutually exclusive predicates, and this partitioning is unique, independent of order of factors. However, one has to be aware that terms with a negative sign in a sum or quotients in a product behave in a perhaps non intuitive way. For example with the same rule as above:

```
(%i24) r2(x*y/z);
                                      1
(%o24)                      [d = 1, c = -, b = x y]
                                      z
```

First the matcher has inserted a factor 1 to match d, but note that 1/z appears in the non atoms. This is because 1/z is not an atom; indeed, op(1/z)="/". Similarly for a subtraction:

```
(%i25) defmatch(r2,b+c);
(%o25)                                r2
(%i26) r2(x-y);
(%o26)                         [c = - y, b = x]
```

Finally let us look at how the matcher works with expressions containing sub-expressions:

```
(%o27) matchdeclare(a,pred1,b,pred2,c,numberp,d,pred1);
(%o28)                               done
(%i29) defmatch(r3,c+a*b+3*d);
defmatch: 3 d
         will be matched uniquely since sub-parts would otherwise be ambigious.
defmatch: b a
         will be matched uniquely since sub-parts would otherwise be ambigious.
(%i31) r3(5+x*sin(y)+3*z);
(%o31)                 [d = z, c = 5, a = x, b = sin(y)]
```

Note the 3 in the pattern which is matched by the same 3 in the expression. Also the pattern compiler emits a message whose aim is not clear, but has a typo. Perhaps it means that the two non atoms 3*d and a*b cannot be grouped in the plus blob as otherwise one could not identify the variables in the products. Anyways the wild card variables are correctly identified.

2.1 If one is interested in the inner workings.

Let us see how the matcher works in more details on a simple case. The following only works OK with Maxima compiled with Clisp.

```
(%i1) matchdeclare(a,atom);
```

```
(%o1)                                done
(%i2) defmatch(r,h*a);
(%o2)                                  r
(%i3) r(h*x);
(%o3)                               [a = x]
(%i4)
```

This is the occasion to state that the pattern may contain undeclared objects which are matched only by themselves. One can see the function r by doing:

```
(%i4) :lisp(symbol-function '$r)
#<FUNCTION LAMBDA (TR-GENSYM1) (DECLARE (SPECIAL TR-GENSYM1))
  (CATCH 'MATCH
   (PROG NIL (DECLARE (SPECIAL))
    (SETQ TR-GENSYM1 (MEVAL '((MQUOTIENT) TR-GENSYM1 $H)))
    (COND ((DEFINITELY-SO '(($ATOM) TR-GENSYM1)) (MSETQ $A TR-GENSYM1))
     ((MATCHERR)))
    (RETURN (RETLIST $A))))>
(%i4) ?trace(?matcherr);
;; Tracing function MATCHERR.
(%o4)                              (matcherr)
(%i5) :lisp(trace $r)
;; Tracing function $R.
($R)
(%i5) r(h*x);
1. Trace: ($R '((MTIMES SIMP) $H $X))
1. Trace: $R ==> ((MLIST SIMP) ((MEQUAL SIMP) $A $X))
(%o5)                                [a = x]
(%i6) r(h*sin(x));
1. Trace: ($R '((MTIMES SIMP) $H ((%SIN SIMP) $X)))
2. Trace: (MATCHERR)
1. Trace: $R ==> NIL
(%o6)                                false
```

We see that TR-GENSYM1 is just in the first case '((MTIMES SIMP) $H $X) that is h*x. The first thing done is to divide by h to get x. Then the first clause of the COND tests if x is an atom (with certainty). If so it sets a=x gets out of the COND and returns the list [a=x]. In the second case the first COND clause is not satisfied, so one tries the second one, which runs (matcherr) and if true does nothing. But (matcherr) throws the 'MATCH tag and has result nil. The 'MATCH is caught by the CATCH, this goes to end of function thus returning nil. The general pattern matching functions are composed by a number of similar pieces of code which are emitted by various functions (getdec, compileatom, compileplus, compiletimes) as bits of text and are then coerced to lisp functions. One can see these various bits of code emitted by tracing said functions. For example:

```
(%i33) :lisp(trace compiletimes)
...
(%i35) defmatch(r,a*b);
  0: (COMPILETIMES TR-GENSYM16 ((MTIMES SIMP) $A $B))
```

```
  0: COMPILETIMES returned
       ((COND
         ((PART* TR-GENSYM16 '($B $A)
                 '((LAMBDA (#:G494)
                     (DECLARE (SPECIAL #:G494))
                     (COND
                      ((DEFINITELY-SO '(($PRED2) #:G494)) (MSETQ $B #:G494))
                      ((MATCHERR))))
                   (LAMBDA (#:G495)
                     (DECLARE (SPECIAL #:G495))
                     (COND
                      ((DEFINITELY-SO '(($PRED1) #:G495)) (MSETQ $A #:G495))
                      ((MATCHERR)))))))
         (T (MATCHERR))))
(%o35)
```

r where the part:

```
        (COND
                      ((DEFINITELY-SO '(($PRED2) #:G494)) (MSETQ $B #:G494))
                      ((MATCHERR)))
```

Is in fact issued by getdec as seen by tracing it. We see that the symbol-function of $r is built piece by piece by several functions.

3 The functions defining rules, and transformation rules.

We have seen a particular form of the defmatch function. Its general form is the following:

```
defmatch(name, pattern,x_1,...,x_n)
```

that is one may specify extra "pattern arguments" besides the wildcards declared by matchdeclare. However their use is not as flexible as that of the wildcards as illustrated below.

```
(%i1) pred1(x):=atom(x) and not numberp(x);
(%o1)               pred1(x) := atom(x) and (not numberp(x))
(%i2) pred2(x):= not atom(x);
(%o2)                        pred2(x) := not atom(x)
(%i3) matchdeclare(a,pred1,b,pred2);
(%o3)                                done
(%i4) defmatch(r3,a+x+b*u,x);
(%o4)                                 r3
(%i5) r3(y+z+sin(t)*u,z);
(%o5)                     [a = y, b = sin(t), x = z]
```

Here we have two wildcards and two extra variables x and u in the pattern. The variable x is declared in defmatch and the variable u is undeclared, and will match only itself. The pattern argument x is given value z, but in fact z has to be given as argument to r3, and matches itself, so there is no new information. Indeed:

```
(%i6) r3(y+z+sin(t)*u,x);
(%o6)                                false
```

So x doesn't play a role of wildcard. On the other hand these pattern arguments allow to dissect for example products (this doesn't work with wildcards, as we have seen):

```
(%i7) defmatch(r4,x*y,x,y);
(%o7)                                 r4
(%i8) r4(u*v,u,v);
(%o8)                           [x = u, y = v]
```

Presumably canonical ordering of the product x*y and order of the pattern variables are involved in that specific identification. Note that pattern variables can be kernels: With the wildcard a being an atom:

```
(%i9) defmatch(r4,a*y,y);
(%o9)                                 r4
(%i10) r4(u*v,u);
(%o10)                          [a = v, y = u]
(%i11) r4(u*sin(v),sin(v));
(%o11)                        [a = u, y = sin(v)]
```

Anyways what can we do with the result of applying a rule defined by defmatch? One can substitute the sequence of equalities in any expression of the pattern variables or even use them directly. Beware there is a difference:

```
(%i12) a^2+y^2;
                                      2    2
(%o12)                               y  + u
(%i13) subst(%o11,a^2+y^2);
                                     2       2
(%o13)                            sin (v) + u
```

In %i12 the a=u has been applied but not the y=sin(v). Only the wildcards have been effectively substituted. Using subst gives a complete result.

The defrule function allows to specify a replacement directly. Its syntax is:

```
defrule(name,pattern,replacement)
```

Note that no extra pattern variables as in defmatch are allowed, but one is free to use undeclared variables that will be matched by themselves. The replacement is any expression depending on these variables and wild cards. It may be some mathematical function or some more textual function as we have shown at the beginning

```
defrule(r2,a*b*d,['a = a,'b = b,'d = d]);
```

The replacement pattern is here chosen so that this rule has the same effect than defmatch(r2,a*b*d). Replacement involving mathematical functions is for example:

```
(%i1) pred2(x):= not atom(x);
(%o1)                         pred2(x) := not atom(x)
(%i2) matchdeclare(a,atom,b,integerp,c,pred2);
(%o2)                                 done
(%i3) defrule(r5,a+b*c+x,exp(a)/b*c/x);
defmatch: c b
         will be matched uniquely since sub-parts would otherwise be ambigious.
                                                  a
                                                %e  c
(%o3)                      r5 : x + b c + a -> -----
                                                 b x
(%i4) r5(u+3*sin(v)+x);
                                    u
                                  %e  sin(v)
(%o4)                             ----------
                                     3 x
```

Note that x is undeclared in the pattern and matched by itself in the expression. It is very important to identify precisely b and c in bc since they are transformed into c/b which could easily be erroneously b/c. This is achieved by declaring b integerp and c non atom.

We end this discussion of defrule by an interesting textual replacement function appearing in the manual:

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)                                 done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" = bb]);
(%o2)          r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" = bb]);
(%o3)           r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i4) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
                                                              n
(%o4)     [all atoms = %pi + 8, all nonatoms = sin(x) + 2  - c + a b]
(%i5) r2 (8 * (a + b) * %pi * sin(x) / c * 2^n);
                                                        n + 3
                                              (b + a) 2      sin(x)
(%o5)       [all atoms = %pi, all nonatoms = ----------------------]
                                                         c
```

The results are here a partition of atoms and non atoms in a sum and a product respectively. However while c is certainly an atom it appears in the expressions in the form -c or $1/c$ which both are not atoms. The predicates being logically disjoint aa and bb are indeed partitions of the considered expressions determined without ordering ambiguity. Note also that the simplifier has written $8 * 2^n$ as $2^(n + 3)$.

With this example one can indeed show the greediness problem: let us define instead of two mutually exclusive predicates two overlapping predicates:

```
(%i1) matchdeclare(aa, atom, bb, lambda ([x], not atom(x) or numberp(x)));
(%o1)                                done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" = bb]);
(%o2)         r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
                                                              n
(%o3)     [all atoms = %pi, all nonatoms = sin(x) + 2  - c + a b + 8]
```

Note that a number satisfies both predicates for aa and bb, so that 8 is taken greedily by the first variable bb, and thus does not appear in "all atoms".

At this point we may have a lot of rules defined, a lot of wildcards declared by matchdeclare, etc. There are convenience functions to list this stuff (disprule, propvars, printprops).

One may get for example:

```
(%i27) disprule(all);
(%t27)                     rr1 : 3 d + c + a b -> []

(%t28)                     rr2 : x + b u + a -> [x]

(%t29)                        rr3 : x y -> [x, y]

(%t30)                         rr4 : a y -> [y]

                                                 a
                                               %e  c
(%t31)                   rr5 : x + b c + a -> -----
                                                b x

(%o31)                 [%t27, %t28, %t29, %t30, %t31]

(%i32) propvars (matchdeclare);
(%o32)                           [a, b, c, d]
(%i33) printprops (all, matchdeclare);
(%o33)            [pred1(d), numberp(c), pred2(b), pred1(a)]
```

We see that the rules are labelled by disprule, wildcards are listed by propvars(matchdeclare) and the associated predicates are listed by printprops. In a similar vein clear_rules() kills the existing rules.

Once rules have been defined there is a mechanism to apply them automatically and recursively in an expression. One does that using the functions apply1, apply2, applyb1. apply1(exp,rule_1,...,rule_n) apply rule_1 to exp repeatedly until it fails. The the same rule on all subexpressions, left to right (recall that an expression is a tree, usually with operators at nodes and leafs are atoms). Then one does the same with rule_2 etc. The difference with apply2 is that apply2 applies in order rule_1,...,rule_n to each subexpression before passing to the next one. Finally applyb1 does the same but starting from bottom. The recusrsions are limited by maxapplydepth from top to bottom and maxapplyheight from bottom towards top.

A trivial example: going from dot product to * product.

```
(%i1) matchdeclare(a,true,b,true);
(%o1)                                done
(%i2) defrule(r1,a.b,a*b);
(%o2)                          r1 : a . b -> a b
(%i3) apply1((x+y.(z+t.u)).(v+w.g),r1);
(%o3)                     (g w + v) (y (z + t u) + x)
```

Going the other way round.

```
(%i1)  matchdeclare(a,lambda([e],not atom(e) and (op(e)="*")));
(%o1)                                done
(%i2) defmatch(r,a);
defmatch: evaluation of atomic pattern yields: a
(%o2)                                  r
(%i3) defrule(r2,a,subst(".","*",a));
(%o3)                       r6 : a -> subst(., *, a)
(%i5) apply1((g*w + v)*(y*(z + t*u) + x),r2);
(%o5)                 (v + g . w) . (y . (z + t . u) + x)
```

Transforming only the leaf products in the expression tree:

```
(%i7) maxapplyheight:1$

(%i8) applyb1((g*w + v)*(y*(z + t*u) + x),r6);
(%o8)                    (v + g . w) (y (z + t . u) + x)
```

Suppose one wants to get the arguments of the leaf dot products:

```
(%i1) matchdeclare(a,lambda([e],not atom(e) and (op(e)=".")));
(%o1)                                done
(%i2) defrule(r3,a,print(args(a)));
(%o2)                       r3 : a -> print(args(a))
(%i3) maxapplyheight:1$
(%i5) applyb1((v + g . w)*(y*(z + t . u) + x),r3)$
[g, w]
[t, u]
```

Using recursive functions like apply1 may lead to infinite recursion if the replacement patterns still matches:

```
(%i6) matchdeclare(a,atom);
(%o6)                                done
(%i7) defrule(r,a,[a]);
(%o7)                            r : a -> [a]
(%i8) apply1(x+y,r);
INFO: Binding stack guard page unprotected
Binding stack guard page temporarily disabled: proceed with caution
(%i9) :lisp(trace $r)
($R)
(%i9) apply1(x+y,r);
  0: ($R ((MPLUS SIMP) $X $Y))
  0: $R returned NIL
  0: ($R $X)
  0: $R returned ((MLIST SIMP) $X) T
  0: ($R ((MLIST SIMP) $X))
  0: $R returned NIL
  0: ($R $X) ..............
```

We see that r applies to x, giving [x], then doesn't apply to [x]. It then descends recursively to x inside [x] and infinite recursion begins. Note that applyb1 doesn't suffer from the same problem:

```
(%i5) applyb1(x+y,r);
  0: ($R $X)
  0: $R returned ((MLIST SIMP) $X) T
  0: ($R $Y)
  0: $R returned ((MLIST SIMP) $Y) T
  0: ($R ((MLIST SIMP) ((MPLUS SIMP) $X $Y)))
  0: $R returned NIL
(%o5)                                [y + x]
```

Here r applies on x and gives [x] but is then blocked, similar for y, so it goes upper to x+y and returns [x+y].

Remark: In case the replacement pattern is a lambda function one has to apply the lambda function to the matching variable to get the result, such as:

```
(%i1) matchdeclare(a,atom);
(%o1)                                done
(%i2) defrule(r,a,lambda([e],print("a =", e))(a));
(%o2)               r : a -> lambda([e], print(a =, e))(a)
(%i3) r(x);
a = x
(%o3)                                  x
```

Note the a inside the lambda is not captured automatically, and stays in the printout a=x . This is a remark of R. Dodier.

4 The let rules.

These rules are simplification rules specifically for use with products, namely products of atoms to positive or negative powers and kernels such as sin(x), n!, f(x,y). They are defined in nisimp.lisp.

Some of the arguments of prod may be declared in a matchdeclare statement, but as in the case of defmatch, other arguments mays be specified in the let rule. Moreover one can specify a predicate which specifies when these let arguments match. The replacement is any rational function, and the matched arguments are substituted in it. A typical let rule takes the form:

```
let(product,replacement,predicate,arg_1,...arg_n);
```

To be more precise positive powers are grouped in a numerator and negatives one in a denominator. Powers of an atom in an expression match only when the power is greater or equal than in the pattern, either in the numerator or denominator.

In fact let rules may be grouped in packages, so there is an extended form:

```
let([product,replacement,predicate,arg_1,...arg_n],package_name);
```

The let rules are applied to an expression by the commands:

```
letsimp(expression); or
letsimp(expression,package_name); or
letsimp(expression,pack_1,pack_2,...);
```

The command letsimp simplifies separately the numaerator and the denominator. If one wants that cancellations occur between numerator and denominator, one needs to set

```
letrat:true;
```

Letsimp acts by applying the let rule repeatedly until the expression doesn't change any more. However this needs some comments.

```
(%i1) t(x,y):=true;
(%o1)                          t(x, y) := true
(%i2) let(x*y,x+y,t,x,y);
(%o2)                    x y --> y + x where t(x, y)
(%i3) letsimp(x*y+y*z);
(%o3)                            y z + y + x
```

We note that the letsimp has been distributed across the addition. The part x*y matches and becomes x+y while y*z doesn't match and is left as is. Similarly

```
(%i4) letsimp(x*y*z);
(%o4)                                y z + x z
```

so z which is not declared is left in identical way in the replacement. Things are more tricky when adding powers. Looking at nisimp.lisp one sees that the program to be traced is nisletsimp. We get:

```
(%i5) trace(letsimp);
(%o5)                                [letsimp]
(%i6) :lisp(trace nisletsimp)

(NISLETSIMP)
(%i6) letsimp(x^2*y);
                           2
1 Enter letsimp [letsimp(x  y)]
  0: (NISLETSIMP
      ((MRAT SIMP ($X $Y) (#:X468 #:Y469)) (#:Y469 1 (#:X468 2 1)) . 1))
    1: (NISLETSIMP ((MTIMES RATSIMP) ((MEXPT RATSIMP) $X 2) $Y))
      2: (NISLETSIMP ((MPLUS SIMP) ((MEXPT SIMP) $X 2) ((MTIMES SIMP) $X $Y)))
        3: (NISLETSIMP ((MEXPT SIMP) $X 2))
        3: NISLETSIMP returned ((MEXPT SIMP) $X 2)
        3: (NISLETSIMP ((MTIMES SIMP) $X $Y))
          4: (NISLETSIMP ((MPLUS SIMP) $X $Y))
            5: (NISLETSIMP $X)
            5: NISLETSIMP returned $X
            5: (NISLETSIMP $Y)
            5: NISLETSIMP returned $Y
          4: NISLETSIMP returned ((MPLUS) $X $Y)
        3: NISLETSIMP returned ((MPLUS) $X $Y)
      2: NISLETSIMP returned ((MPLUS) ((MEXPT SIMP) $X 2) ((MPLUS) $X $Y))
    1: NISLETSIMP returned ((MPLUS) ((MEXPT SIMP) $X 2) ((MPLUS) $X $Y))
    1: (NISLETSIMP 1)
    1: NISLETSIMP returned 1
  0: NISLETSIMP returned
       ((MQUOTIENT) ((MPLUS) ((MEXPT SIMP) $X 2) ((MPLUS) $X $Y)) 1)
                2
               x  + (x + y)
1 Exit  letsimp ------------
                     1
                                       2
(%o6)                               y + x  + x
```

At position 1 nisletsimp sees the product of x^2 and y. This product matches the rule thus at position 2 nisletsimp sees x^2+xy. Then at position 4 xy -> x+y and one gets the result x^2+x+y, the quotient by 1 being due to rational representation. Hence the question: how is it that one application of the rule to $x^2 * y$ gives $x^2 + xy$ and not $x^2 + y$? This is understood if one writes $x^2 * y = x * (x * y) -> x * (x + y)$. Let us try to see if this works for higher powers:

```
(%i8) letsimp(x^3*y);
```

```
                                    3    2
(%o8)                          y + x  + x  + x
```

This is explained by the chain:

$x^3 * y = x^2 * (x * y)$ -> $x^2 * (x + y) = x^3 + x * (x * y)$ -> $x^3 + x^2 + x * y$ -> $x^3 + x^2 + x + y$

Finally let us look at:

```
(%i9) letsimp(x^3*y^2);
                           2          3      2
(%o9)                     y  + 3 y + x  + 2 x  + 3 x
```

This is explained by : $x^3 * y^2$ -> $x^2 * y * (x + y) = x^3 * y + x^2 * y^2$. The first one is $x^3 + x^2 + x + y$, the second $x * y * (x + y)$ -> $x * (x + y) + y * (x + y)$ -> $x^2 + y^2 + 2 * (x + y)$, yielding the above result.

This explains the nature of the recursion occurring in let rules. As is the case with defmatch it is frequently more convenient to use variables in patterns defined by matchdeclare. For example in below, variables m and n are matched to a1 and a2 and substituted in the replacement. An example is in the manual:

```
(%i1) matchdeclare ([a, a1, a2], true)$
(%i2) oneless (x, y) := is (x = y-1)$
(%i3) let (a1*a2!, a1!, oneless, a2, a1);
(%o3)         a1 a2! --> a1! where oneless(a2, a1)  /* that is a2=a1-1 */
(%i4) letrat: true$
(%i5) let (a1!/a1, (a1-1)!);
                         a1!
(%o5)                    --- --> (a1 - 1)!
                         a1
(%i6) letsimp (n*m!*(n-1)!/m);
(%o6)                       (m - 1)! n!
```

Finally there are some convenience functions to manipulate let rules.

```
letrules(); display rules in current package, by default default_let_rule_package
letrules(pkg) ; display rules in pkg
let_rule_packages;  is a list of all packages, including default_let_rule_package
remlet(product,package_name); removes rule for product in package_name
remlet();  removes all rules from current package
remlet(all); idem
remlet(all,package_name); removes all rules in package_name, and the name itself.
current_let_rule_package can be assigned to any package name.
```

We see that the let rules allow to simplify differently expressions. However they have to be called explicitely by letsimp(). We are now going to discuss similar simplifying functions which are directly hooked in the Maxima simplifier, and are thus called automatically. They are called letsimp (called before the simplifier) and letsimpafter (called after the simplifier).

5 Tellsimp and tellsimpafter.

Both have the form:

```
tellsimp (pattern, replacement);
tellsimpafter (pattern, replacement);
```

The manual says: tellsimp is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier "knows" something about the expression, but what it returns is not to your liking. If the simplifier "knows" something about the main operator of the expression, but is simply not doing enough for you, you probably want to use tellsimpafter.

The pattern is comprised of wild cards declared with defmatch and other variables or kernels and operators which are taken literally as above. For tellsimp the pattern may not be a sum, product, single variable, or number. For tellsimpafter pattern may be any nonatomic expression in which the main operator is not a pattern variable.

The simplification rule is associated to the main operator. If the name of the rule is foorule, one can trace it by :lisp(trace foorule) and see it (with maxima -l clisp) with

```
:lisp (symbol-function '$foorule).
```

A simple example:

```
(%i1) matchdeclare([a,b,c]:all);
(%o1)                                done
(%i2) tellsimpafter(a+b*c,a-b*c);
defmatch: c b
         will be matched uniquely since sub-parts would otherwise be ambigious.
(%o2)                          [+rule1, simplus]
```

Note the rule is associated to main operator "+" and named +rule1. It is also associated to the simplus generic simplification functions for sums. Tracing it gives:

```
(%i3) :lisp(trace +rule1)
(+RULE1)
(%i3) x+y*z;
  0: (+RULE1 ((MPLUS) $X ((MTIMES SIMP) $Y $Z)) 1 NIL)
    1: (+RULE1 ((MPLUS) $X ((MTIMES SIMP) -1 $Y $Z)) 1 NIL)
    1: +RULE1 returned ((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y $Z))
  0: +RULE1 returned ((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y $Z))
(%o3)                                x - y z
```

The following is trickier:

```
(%i1) matchdeclare([a,b],all);
(%o1)                                done
(%i2) tellsimpafter(a+b,a-b);
(%o2)                          [+rule1, simplus]
(%i3) :lisp(trace +rule1)
(+RULE1)
(%i3) x+y;
  0: (+RULE1 ((MPLUS) $X $Y) 1 NIL)
    1: (+RULE1 ((MPLUS) $X $Y) 1 NIL)
    1: +RULE1 returned ((MPLUS SIMP) $X $Y)
    1: (+RULE1 ((MPLUS) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y)) 1 T)
    1: +RULE1 returned
         ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y))
    1: (+RULE1
        ((MPLUS) 0 ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y)))
        1 NIL)
    1: +RULE1 returned
         ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y))
  0: +RULE1 returned ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y))
(%o3)                               (- y) - x
```

Here we see that a is matched to 0, and b to (x+y) so the result is probably unexpectedly -x-y. We see once more the problems associated with the fact that for "+" and "*" the matcher is greedy and constants like 0 or 1 are added freely in the game. For tellsimp, main operator "+" is forbidden, and indeed if one tries it:

```
(%i1) matchdeclare([a,b],all);
(%o1)                                done
(%i2) tellsimp(a+b,a-b);
tellsimp: warning: rule will treat '+' as noncommutative and nonassociative.
(%o2)                          [+rule1, simplus]
(%i3) x+y;
INFO: Binding stack guard page unprotected
Binding stack guard page temporarily disabled: proceed with caution
Maxima encountered a Lisp error:
Binding stack exhausted.
```

Once more interaction between the tellsimprule and the simplifier causes infinite loop. These function being very automatic can cause many unexpected results. But they can also produce nice results.

As an example let us present quaternions in this way. Here z0 plays the role of unit quaternion, z1,z2,z3 are usually denoted i,j,k. The dot product plays the role of quaternionic product, but we need to introduce scalars as coefficients and impose distributivity with respect to addition and product with a scalar.

```
(%i1) dotdistrib:true$
(%i2) dotscrules:true$
(%i3) tsa(i,j,k,s)::=buildq([i,j,k,s],apply(          /* Utility macro */
        tellsimpafter,[concat(z,i).concat(z,j),s*concat(z,k)]))$
(%i4) for kk in [[1,2,3,1],[2,1,3,-1],[3,1,2,1],[1,3,2,-1],
         [2,3,1,1],[3,2,1,-1]] do ([i,j,k,s]:kk,tsa(i,j,k,s))$
(%i5) for kk in [[0,1],[1,0],[0,2],[2,0],[0,3],[3,0],[0,0]] do
        ([i,j]:kk,tsa(i,j,i+j,1))$
(%i6) for k from 1 thru 3 do tsa(k,k,0,-1)$
(%i7) for i from 0 thru 3 do
      apply(declare,[[concat(a,i),concat(b,i)],scalar])$
(%i8) A: sum(concat(a,i)*concat(z,i),i,0,3)$
(%i9) B: sum(concat(b,i)*concat(z,i),i,0,3)$
(%i10) rat(expand(A.B),z0,z1,z2,z3);
(%o10)/R/ (a0 b3 + a1 b2 - a2 b1 + a3 b0) z3
 + ((- a1 b3) + a0 b2 + a3 b1 + a2 b0) z2
 + (a2 b3 - a3 b2 + a0 b1 + a1 b0) z1
 + ((- a3 b3) - a2 b2 - a1 b1 + a0 b0) z0
(%i11) cA:-a3*z3 - a2*z2 - a1*z1 + a0*z0$             /* Conjugate of A */
(%i12) nA:a3^2  + a2^2  + a1^2  + a0^2$               /* Norm of A = A.cA */
(%i13) rat(expand(A.cA/nA),z0,z1,z2,z3);
(%o13)/R/                            z0               /* The unit quaternion */
```

One can check the proper declaration of scalars and of rules with propvars(scalar); and disprule(all); In %o10 we see the product of two quaternions A and B. If a0=b0=0 only the vector part appears and we see that the quaternionic product reduces to the vector product. In %o13 we see that every non nul quaternion is invertible, i.e. quaternions form a field.

The gamma algebra of quantum field theory can be treated in a similar way, we refer to

`https://github.com/dprodanov/clifford/blob/master/clifford.mac`

by D. Prodanov. This formalism is also amenable to treat the case of groups defined by generators and relations

`https://en.wikipedia.org/wiki/Presentation_of_a_group`