`generate` — generate input to other programs

Table of contents

The functions of the `generate` package provide the ability to generate code for programming languages like C or FORTRAN as well as for the popular typesetting software TEX.

`generate::C` – generate C formatted string

`generate::C(e)` generates C output for the MuPAD expression `e`.

Call(s):

- `generate::C(e)`

Parameters:

`e` — an expression, equation or list of equations

Return Value: `generate::C` returns a `string` containing C code.

Related Functions: `fprint`, `print`, `generate::optimize`

Details:

- A MuPAD expression is converted into a C expression.
- An equation is used to represent an assignment in C. The type of the assignment is assumed to be `double`.
- A list of equations may be used to represent a sequence of assignments in C.
- The output string may be printed to a file using `fprint`. Use the printing option `Unquoted` to remove quotes and to expand special characters like newlines and tabs.
- `generate::optimize` may be used to optimize the input before generating the C code.

Example 1. A list of equations is converted into a sequence of assignments.

```
>> generate::C( [ x[1]=y[2+i]^2*(y[1]+sin(z)), x[2]=tan(x[1]^4) ] ):
   print(Unquoted,%)

            x[1] = (y[i + 2]*y[i + 2])*(sin(z) + y[1]) ;
            x[2] = tan(pow(x[1], 4.0)) ;
```

Example 2. The code produced by `generate::C` is not optimized:

```
>> print(Unquoted,
         generate::C([x = a + b, y = (a + b)^2])):

                        x = a + b ;
                        y = pow(a + b, 2.0) ;
```

`generate::optimize` tries to reduce the number of operations:

```
>> print(Unquoted,
         generate::C(
             generate::optimize([x = a + b, y = (a + b)^2])
         )):

                              x = a + b ;
                              y = x*x ;
```

Changes:

- No changes.

`generate::fortran` – generate FORTRAN formatted string

`generate::fortran(e)` generates FORTRAN output for the MuPAD expression `e`.

Call(s):

- `generate::fortran(e)`

Parameters:

`e` — an expression, equation or list of equations

Return Value: `generate::fortran` returns a `string` containing FORTRAN code.

Related Functions: `fprint`, `print`, `generate::Macrofort::genFor`, `generate::optimize`

Details:

- A MuPAD expression is converted into a FORTRAN expression.
- An equation is used to represent an assignment in FORTRAN. The type of the assignment is assumed to be `double`.
- A list of equations may be used to represent a sequence of assignments in FORTRAN.
- The output string may be printed to a file using `fprint`. Use the printing option `Unquoted` to remove quotes and to expand special characters like newlines and tabs.
- `generate::optimize` may be used to optimize the input before generating the FORTRAN code.
- `generate::Macrofort::genFor` is a more general function for generating FORTRAN code with more options.

Example 1. A list of equations is converted into a sequence of assignments:

```
>> generate::fortran( [ x[1]=y[2+i]^2*(y[1]+sin(z)),
                        x[2]=tan(x[1]^4) ] ):
   print(Unquoted,%)

                    x(1) = y(i+2)**2*(sin(z)+y(1))
                    x(2) = tan(x(1)**4)
```

Example 2. The code produced by `generate::fortran` is not optimized:

```
>> print(Unquoted,
         generate::fortran([x = a + b, y = (a + b)^2])):

                              x = a+b
                              y = (a+b)**2
```

`generate::optimize` tries to reduce the number of operations:

```
>> print(Unquoted,
         generate::fortran(
             generate::optimize([x = a + b, y = (a + b)^2])
         )):

                                  x = a+b
                                  y = x*x
```

Changes:

- No changes.

`generate::Macrofort::genFor` – FORTRAN code generator

`Mac::genFor` (where `Mac:=generate::Macrofort`) is the Macrofort package for generating FORTRAN code.

The Macrofort package allows the user to make complete standard FORTRAN 77 code while remaining in the MuPAD environment. Necessities of FORTRAN code such as label numbering and complicated FORTRAN loops are handled automatically. Macrofort has capabilities for:

- "Tailoring" large amounts of FORTRAN code (infeasible by hand).
- Solving storage problems in FORTRAN code for many recursive algorithms.
- Numerically solving recursive algorithms, and in some cases, within the area of vectorized machine architectures (see last example).

Furthermore, the combination of MuPAD's symbolic manipulation capabilities and, in particular, its optimizer `generate::optimize` can help yield efficient codes for ambitious goals in numerical computation. MuPAD can be also used as a preprocessor for numerical analysis.

Macrofort overlaps with MuPAD's `generate::fortran` but it is more general program for generation FORTRAN code and has many more options.

Call(s):

- `generate::Macrofort::genFor(l)`

Parameters:

`l` — list or list of lists

Return Value: the void object of domain type `DOM_NULL`

Side Effects: writes FORTRAN code into an ascii file

Related Functions: `generate::Macrofort::init`, `generate::Macrofort::setOptimizedOption`, `generate::Macrofort::setIOSettings`, `generate::Macrofort::setPrecisionOption`, `generate::Macrofort::setAutoComment`,

`generate::Macrofort::openOutputFile`,
`generate::Macrofort::closeOutputFile`, `generate::optimize`,
`generate::fortran`

Details:

- The syntax of the MuPAD input to generate a FORTRAN statement or a macro FORTRAN statement is a list where the first element is a keyword describing the statement, the other (optional) elements of that list being relevant arguments. The keyword itself is a string referring to a FORTRAN instruction name (when it exists) and for macro statements, a letter character `m` appears at the end of the keyword. The following input list of keywords and their corresponding FORTRAN output refer to Macrofort single instructions:

```
["call", name, list]           =>       call name (list)
["close", n]                   =>       close (n)
["comment", string]            => c     string
["common", name, list]         =>       common /name/ list
["continue", label]            => label continue
["declare", type, list]        =>       type list
["do", label, index,           =>       do label,index=start,end
       start, end]             =>
["do", label, index,           =>       do label,index=start,end,step
       start, end, step]       =>
["else"]                       =>       else
["end"]                        =>       end
["endif"]                      =>       endif
["equal", var, expression]     =>       var=expression
["format", label, list]        => label format (list)
["function", type,             =>       type function name (list)
            name, list]        =>
["goto", label]                =>       goto label
["if_goto", cond, label]       =>       if (cond) goto label
["if_then", cond]              =>       if (cond) then
["parameter", list]            =>       parameter (list)
["program", name]              =>       program name
["read", file, label, list]    =>       read (file,label) list
["return"]                     =>       return
["subroutine", name, list]     =>       subroutine name (list)
["write", file, label, list]   =>       write (file, label) list
```

and also

```
["open", n, file, st]  =>  open(unit=n,file='file',status='st')
```

where *n* appears in cases such as `"open"` correspond to unit numbers (or channel numbers) for FORTRAN I/O instructions and where `cond`

appears as a condition argument of a Macrofort instruction. Examples of conditions are:

["if_then", a>=b].

- For logical operators _not, _and and _or in a condition, you have to use the names "NOT", "AND" and "OR" within a list notation. For instance:

  ["if_then", ["OR", a=b, ["NOT", c<d)]].

- These names are used to ensure that the resulting code directly performs these logical operations in FORTRAN rather than via MuPAD. Label numbers are automatically generated. When label appears as an argument of a Macrofort instruction, a MuPAD variable must be inserted. The label is retained within Macrofort it can always be avoided using the macro instructions.

  When list appears as an argument of a Macrofort instruction, it corresponds to an argument FORTRAN list which you have to write as a MuPAD list. For instance:

  ["call", foo, [a,b,c]]

  or

  ["format", ['2x, e14.7'], [x,y]]

  .

- When you want to generate a FORTRAN array, you have to write a MuPAD array. Also available are the Macrofort macro instructions but for these, labels are not needed:

["do_m", index, start, end, step, doList]	do label,index=start,end,step doList label continue
["do_m", index, start, end, doList]	do label,index=start,end doList label continue
["functionm", type, name, list, bodyList]	type function name (list) bodyList end

["if_then_else_m", cond,
thenList, elseList]
	if cond then
	thenList
	else
	elseList
	endif

["if_then_m", cond, thenList]	if cond then
	thenList
	endif

["programm", name, bodyList]	program name bodyList end
["readm", file, formatList, varList]	read (file,label) varList label format (formatList)
["subroutinem", name, list, bodyList]	subroutine name (list) bodyList end
["writem", file, formatList, varList]	write (file,label) varList label format (formatList)
["declarem", type, list]	type list
["commonm", name, list]	common / name / list

and also

["openm", n, file, st, bodyList]	open(unit=n,file='file',status='st') bodyList close (n)

- The only difference between "commonm" and "declarem" and their single instruction counterparts, namely "common" and "declare" is that one can put the macro instructions everywhere in the list describing the program and Macrofort puts them at the right place in the generated FORTRAN code. This allow us to declare a variable only when it is used in the body of the program. These macros only work within the "programm", "functionm" or "subroutinem" instructions. Otherwise there are ignored.

- There are other very important macro instructions. First, there are two macro instructions corresponding to WHILE and UNTIL loops. Their

syntax has the form:

["whilem", condition, initList, whileList, whileMax(optional)] :

<initList>
while condition **do**
<whileList>
end.

["untilm", condition, initList, untilList, untilMax(optional)] :

<initList>
do <untilList>
until condition
end.

where `whileMax` and `untilMax` are optional arguments and respectively the maximum number of iterations for the WHILE and UNTIL loops. When the maximum is reached, the loop stops and a message is issued. If this argument is not given, there is no maximum number of iterations. Note that `doList`, `thenList`, `elseList`, `bodyList`, `initList`, `untilList` and `whileList` arguments must be MuPAD lists describing FORTRAN statements with a Macrofort syntax. You can nest as many loops as you want.

- `["matrixm",var,matrix]` is another very useful macro instruction. It is used to make assignations of the elements of a matrix or array. Here, `var` is the name of a FORTRAN matrix and `matrix` is a MuPAD matrix (see first example).

- There are global variables defined within the `macrofort` domain. Please note that before any call to `Mac::genFor` (where `Mac:=generate::Macrofort`), the procedure `Mac::init` must be used! The latter sets these global variables to their default values and it initializes the various counters used internally by Macrofort (e.g.: for label generation). Deviations from these settings are done by calls to `Mac::setIOSettings`, `Mac::setOptimizedOption`, `Mac::setPrecisionOption` and `Mac::setAutoComment` (see the help page for `Mac::init` and these other procedures for details). Furthermore, any call to `Mac::genFor` also needs a call to `Mac::openOutputFile` to name and open the ascii FORTRAN file and a call to `Mac::closeOutputFile` to close that same file.

Example 1. Calculation of a matrix.
Initialize Macrofort and open the file "matrix.f"

```
>> Mac := generate::Macrofort:
   Mac::init():
   Mac::openOutputFile("matrix.f"):
```

Create a 2 x 2 array with symbolic entries called a

```
>> a := array(1..2, 1..2, [[x^2, x - y], [x/y, x^2 - 1]])
```

$$\begin{bmatrix} x^2 & x - y \\ \frac{x}{y} & x^2 - 1 \end{bmatrix}$$

and generate a FORTRAN array v for a using "matrixm" in genFor:

```
>> Mac::genFor(["matrixm", v, a]):
```

Close the file "matrix.f"

```
>> Mac::closeOutputFile():
   delete a:
```

The output file matrix.f is:

```
      v(1,1) = x**2
      v(1,2) = x-y
      v(2,1) = x/y
      v(2,2) = x**2-1
```

Example 2. The calculation of a polynomial in Horner form.
Open the file "demo.f" and define the function using the macro "functionm". p is a list containing all specifications in the form of lists. "declarem" defines the FORTRAN variables and their types and "do_m" creates the do-loop by which the polynomial is summed in Horner form.

```
>> Mac::openOutputFile("demo.f"):

   p := ["functionm", doubleprecision,
         horner,[x,n,a],
         [["declarem", doubleprecision,
           [a(n), x]],
```

```
          ["equal", horner, a[n]],
          ["do_m", i, n - 1, 0, -1,
           ["equal", horner, a[i] + horner*x]]]]:

  Mac::genFor(p):
  Mac::closeOutputFile():
  delete p,a,x,n,i:
```

The output file `demo.f` is:

```
c
c     FUNCTION horner
c
      doubleprecision function horner(x,n,a)
        doubleprecision a(n),x
        horner = a(n)
c
          do 1000, i=n-1,0,-1
            horner = x*horner+a(i)
 1000     continue
c
      end
```

Bear in mind that this demo example is lame as the Macrofort MuPAD code is at least as extensive as the resulting FORTRAN output which could have been readily obtained directly by hand. The dividends of Macrofort become more clear in the following examples.

Example 3. A recursive function on a tree.
Although it is possible to write FORTRAN programs for recursive problems, this can be very tedious for complicated recursions and even if the recursion is not so complicated, other problems can arise as shown here. In this example, we consider a binary tree with nodes labeled by pairs of integers (i, j) where $(1,1)$ is the root of the tree, $(2,1)$ and $(2,2)$ are the children nodes of the root and recursively $(i+1, 2j-1)$ and $(i+1, 2j)$ are the children nodes of node (i, j). Node (i, j) is at level i. Consider now the following sequence:

$$f_{1,1} \text{ given}$$
$$f_{i,j} = \begin{cases} g(f_{i-1,\frac{j}{2}}) & \text{if } j \text{ is even} \\ g(f_{i-1,\frac{j+1}{2}}) & \text{if } j \text{ is odd} \end{cases}$$

where g is a given function. We want to compute the values of the sequence up to a given level N, i.e. the 2^{N-1} values $f(N,1) \ldots f(N,2^{N-1})$. To write the corresponding FORTRAN code, you only have to write two loops:

```
      real f(n,m)
      do 1,i=1,n
        do 2,j=1,2**(n-1)-1,2
```

```
          f(i,j)=g(f(i-1,(j+1)/2))
 2      continue
        do 3,j=2,2**(n-1),2
          f(i,j)=g(f(i-1,j/2))
 3      continue
 1    continue
```

but the dimension m of the array f is 2^{N-1}. In other words, we would have to retain the storage of $N \times 2^{N-1}$ real values instead of $2^N - 1$ (which corresponds to 5 times more storage for N equal to 10).

A way to avoid this waste, is to have an array for each level, i.e. to have FORTRAN arrays $f1(1)$, $f2(2)$, $f3(4)$ and so forth but it then becomes very tricky to write the resulting FORTRAN program by hand. However, this can be readily solved using a MuPAD program within Macrofort.

We suppose that a FORTRAN function g has already been defined. The MuPAD function which generates the FORTRAN program is:

```
>> Mac::openOutputFile("Func.f"):

   pushe := proc(e,l) begin [op(eval(l)),e] end_proc:
   gen_Func := proc(n)
     local i,pg,temp,temp1,temp2,temp3;
   begin
     pg:=[]:
   // declaration of the arrays
     for i from 1 to n do
       temp:=eval(text2expr(
       _concat("f",expr2text(i),"[",expr2text(2^(i-1)),"]"))):
       pg:=pushe(["declare",real,[temp]],pg):
     end_for:
   // loops for each array
     for i from 2 to n do
      temp1:=eval(text2expr(_concat("f",expr2text(i),"[j]"))):
      temp2:=eval(text2expr(_concat("f",expr2text(i-1),"[Hold(j+1)/2]"))):
      temp3:=eval(text2expr(_concat("f",expr2text(i-1),"[j/2]"))):
      pg:=pushe(["do_m",j,1,2^(i-1)-1,2,["equal",temp1,g(temp2)]],pg):
      pg:=pushe(["do_m",j,2,2^(i-1),2,["equal",temp1,g(temp3)]],pg):
     end_for:
     pg:=["programm",Func,pg]:
     Mac::genFor(pg):
   end_proc:
   gen_Func(4):
   Mac::closeOutputFile():
   delete j,pushe,gen_Func:
```

The output file `Func.f` is:

```
c
c     MAIN PROGRAM Func
c
      program Func
        real f1(1)
        real f2(2)
        real f3(4)
        real f4(8)
c
          do 1000, j=1,1,2
            f2(j) = g(f1((j + 1)/2))
 1000     continue
c
c
          do 1001, j=2,2,2
            f2(j) = g(f1(j/2))
 1001     continue
c
c
          do 1002, j=1,3,2
            f3(j) = g(f2((j + 1)/2))
 1002     continue
c
c
          do 1003, j=2,4,2
            f3(j) = g(f2(j/2))
 1003     continue
c
c
          do 1004, j=1,7,2
            f4(j) = g(f3((j + 1)/2))
 1004     continue
c
c
          do 1005, j=2,8,2
            f4(j) = g(f3(j/2))
 1005     continue
c
      end
```

By calling `gen_Func(n)` for larger `n`, you can readily have Macrofort generate the needed code for larger and larger n. Of course, the output is proportionally larger but still "digestible" to modern-day compilers for a wide range of n.

Example 4. Optimized Vectorized FORTRAN code for Molecular Integrals. Here, we present a method for the rapid numerical evaluation of molecular integrals which appear in the areas of Quantum Chemistry and Molecular

Physics. The method is based on the exploitation of common intermediates using MuPAD's optimizer (see `generate::optimize`) and the optimization can be adjusted to both serial and vectorised computations.

Integrals based on atom-centered Gaussian-type functions known as the R-integrals are given by the recurrence relations:

$$R_j[\tau+1,\mu,\nu] = xR_{j+1}[\tau,\mu,\nu] + \tau R_{j+1}[\tau-1,\mu,\nu]$$

$$R_j[\tau,\mu+1,\nu] = yR_{j+1}[\tau,\mu,\nu] + \mu R_{j+1}[\tau,\mu-1,\nu]$$

$$R_j[\tau,\mu,\nu+1] = zR_{j+1}[\tau,\mu,\nu] + \nu R_{j+1}[\tau,\mu,\nu-1]$$

$$R_j[0,0,0] = (-2p)^j F_j(\alpha\overline{QP}^2)$$

where

$$F_m(W) = \int_0^1 \exp[-Wt^2]t^{2m}\,dt,$$

where the vector $\overrightarrow{QP} = -(x,y,z)$ and α are given geometrical quantities (determined on input) for a particular molecular geometry, and F is a known function in quantum chemistry for which there already exist various algorithms for its rapid computation (in this example, we are only interested in the computation of the polynomial part of $R(\tau,\mu,\nu)$. The summation indices are bounded by zero and $\tau+\mu+\nu \leq L$ where L is a total angular quantum number for a given molecular problem, and consequently these induces adhere to a *polyhedral structure*. The total number N of R functions to compute for a given L is given by $N = (L+1)(L+2)(L+3)/6$. In the diatomic molecular case (i.e. a molecule made of only two "atoms" whose centers are set on the z-axis), the R-integrals form a sparse three-dimensional matrix (see the work of the references for a fuller understanding of the framework).

Vectorization involved the introduction of a vectorization index, denoted M, which is the first index of all the arrays involved in the computation. The FORTRAN code obtained from the compiler is then "sandwiched" within do-loops. The code shown here is generated for the R integrals up to $L = 3$ although it can generate the code for all the way up to $L = 16$.

This example makes use of MuPAD's optimizer (see `generate::optimize`) and therefore, one obtains optimized vectorized FORTRAN code. This code had been tested on a CRAY and the FLOP (floating-point operations) rate was about 85 percent of its peak efficiency and has been used in specific relativistic quantum chemistry calculations.

Input code:

First we construct the `S` functions by which the `R` functions are later defined as shown in the work of V. Saunders (see references). This definition essentially exploits a decomposition in odd and even symmetries within these functions and provides an economy in the computations.

```
>> S:=proc(j,p,QP,QP2,t,u,v)
   begin
   if (t<0) or (u<0) or (v<0) then
       0;
   elif (t=0) and (u=0) and (v=0) then
       ((-2*p)^j)*FS[M,j+1];
   elif (t>0) and (t mod 2 =1) then
       S(j+1,p,QP,QP2,t-1,u,v)+(t-1)*S(j+1,p,QP,QP2,t-2,u,v);
   elif (u>0) and (u mod 2 =1) then
       S(j+1,p,QP,QP2,t,u-1,v)+(u-1)*S(j+1,p,QP,QP2,t,u-2,v);
   elif (v>0) and (v mod 2 =1) then
       S(j+1,p,QP,QP2,t,u,v-1)+(v-1)*S(j+1,p,QP,QP2,t,u,v-
2);
   elif (t>0) and (t mod 2 =0) then
       QP2[M,1]*S(j+1,p,QP,QP2,t-1,u,v)+(t-1)*S(j+1,p,QP,QP2,t-
2,u,v);
   elif (u>0) and (u mod 2 =0) then
       QP2[M,2]*S(j+1,p,QP,QP2,t,u-1,v)+(u-1)*S(j+1,p,QP,QP2,t,u-
2,v);
   elif (v>0) and (v mod 2 =0) then
       QP2[M,3]*S(j+1,p,QP,QP2,t,u,v-1)+(v-1)*S(j+1,p,QP,QP2,t,u,v-
2);
   end_if;
   end_proc:

   Xi:=proc(xbar,i)
   begin
   if (QP[M,1] <> 0) then
       (-xbar)^(i mod 2);
   elif
       ((i mod 2) > 0) then 0; else 1;
   end_if;
   end_proc:
```

Finally, we construct the R functions

```
>> R:=proc(j,p,QP,QP2,t,u,v)
   local X,Y,Z;
   begin
       X:=Xi(QP[M,1],t);
       Y:=Xi(QP[M,2],u);
       Z:=Xi(QP[M,3],v);
       S(j,p,QP,QP2,t,u,v)*(X*Y*Z);
   end_proc:
```

We restrict ourselves to the Diatomic case i.e. only the z-axis (the 3rd axis) has non-zero components.

```
>> QP[M, 1] := 0:  QP[M, 2] := 0:  QP[M, 3] := QP[M]:
   QP2[M, 1] := 0:  QP2[M, 2] := 0: QP2[M, 3] := QP2[M]:
```

We ensure plenty of guard digits for L up to $L = 16$

```
>> DIGITS:=60:
   subr:=null():
   for LL from 0 to 4 do
       tuv:=0: ds:=null():
       for mu from 0 to LL do
           t:=LL-mu;
           for nu from 0 to mu do
               u:=mu-nu;
               for tau from 0 to nu do
                   v:=nu-tau;
                   tuv:=tuv+1;
                   Rtuv:=float(R(0,ALPHA[M],QP,QP2,t,u,v));
                   if (Rtuv <>0 and Rtuv <> 0.0) then
                      ds:=ds,[RC[M,tuv],Rtuv]:
                   end_if;
               end_for:
           end_for:
       end_for:
       subr:=subr,["if_then_m",L=LL,
                  ["do_m",M,1,MAXM,["equal",[ds]]]];
   end_for:
```

Construct the input list for the FORTRAN Subroutine

```
>> delete QP2:
   subr := ["subroutinem", RMAKE, [L, FS, ALPHA, QP2, MAXM, RC],
            [["declare", "implicit doubleprecision", ["(a-
h,o-z)"]],
             ["parameter", [LMAX = 12, MAXB = 32,
              MAXB2 = "MAXB*MAXB", MAXR = 455]],
             ["declare", dimension,
              ["FS(MAXB2,LMAX)", "ALPHA(MAXB2)", "QP2(MAXB2)",
               "RC(MAXB2,MAXR)"]], subr]]:
```

Initialization of Macrofort:

```
>> Mac:=generate::Macrofort:
   Mac::init():
```

Open file `"vectorized.f"` and switch optimizer on. The desired precision for the FORTRAN code is double:

```
>> Mac::openOutputFile("vectorized.f"):
   Mac::setOptimizedOption(TRUE):
   Mac::setPrecisionOption("double"):
```

Code Generation:

```
>> subr:
   Mac::genFor(subr):
   Mac::closeOutputFile():
   delete subr,S,R,LL,Xi,t,u,v,Rtuv,tuv,ds,tu,mu,nu,QP,QP2,M:
```

The output file `vectorized.f` is:

```
 c
 c     SUBROUTINE RMAKE
 c
       subroutine RMAKE(L,FS,ALPHA,QP2,MAXM,RC)
         implicit doubleprecision (a-h,o-z)
         parameter (LMAX=12,MAXB=32,MAXB2=MAXB*MAXB,MAXR=455)
         dimension FS(MAXB2,LMAX),ALPHA(MAXB2),QP2(MAXB2),RC(MAXB2,MAXR)
           if (L.eq.0) then
 c
               do 1000, M=1,MAXM
                 RC(M, 1) = FS(M,1)
  1000         continue
 c
           endif
           if (L.eq.1) then
 c
               do 1001, M=1,MAXM
                 RC(M, 4) = FS(M,1)
  1001         continue
 c
           endif
           if (L.eq.2) then
 c
               do 1002, M=1,MAXM
                 t1 = ALPHA(M)
                 t2 = FS(M,2)
                 RC(M, 1) = -0.2D1*t1*t2
                 RC(M, 5) = RC(M,1)
                 RC(M, 8) = RC(M,1)+0.4D1*t1**2*QP2(M)*FS(M,3)
                 RC(M, 10) = FS(M,1)
  1002         continue
 c
           endif
           if (L.eq.3) then
 c
               do 1003, M=1,MAXM
                 t3 = ALPHA(M)
                 t4 = FS(M,2)
                 RC(M, 4) = -0.2D1*t3*t4
```

```
                RC(M, 13) = RC(M,4)
                RC(M, 18) = RC(M,4)+0.4D1*t3**2*QP2(M)*FS(M,3)
                RC(M, 20) = FS(M,1)
 1003         continue
c
          endif
          if (L.eq.4) then
c
              do 1004, M=1,MAXM
                t5 = ALPHA(M)
                t6 = t5**2
                t7 = FS(M,3)
                RC(M, 1) = 0.12D2*t6*t7
                RC(M, 5) = 0.4D1*t6*t7
                t8 = QP2(M)
                t9 = FS(M,4)
                t10 = -0.8D1*t5*t6*t8*t9
                RC(M, 8) = t10+RC(M,5)
                t11 = FS(M,2)
                RC(M, 10) = -0.2D1*t5*t11
                RC(M, 21) = RC(M,1)
                RC(M, 24) = RC(M,8)
                RC(M, 26) = RC(M,10)
                RC(M, 31) = t8*(0.16D2*t6**2*t8*FS(M,5)-0.24D2*t5*t6*t9)
     #+RC(M,1)-0.24D2*t5*t6*t8*t9
                RC(M, 33) = 0.4D1*t6*t7*t8+RC(M,10)
                RC(M, 35) = FS(M,1)
 1004         continue
c
          endif
       end
```

The benefits of MuPAD's optimizer become more evident at higher L but we can already see its effects. Common intermediates are exploited and in a number of cases, only re-assignations and not actual computation were needed.

Background:

- References:
  - C. Gomez and T.C. Scott, Maple Programs for Generating Efficient FORTRAN Code for Serial and Vectorized Machines, *Comput. Phys. Comm.*, **115**, (1998).
  - T.C. Scott, I.P. Grant, M.B. Monagan and V.R. Saunders, Proceedings of the fifth International Workshop on New computing Techniques in Physics Research (software engineering, neural nets, genetic algorithms, expert systems, symbolic algebra, automatic cal-

culations), held in Lausanne (Switzerland), Nuc. Instruments & Methods Phys. Research, 389A (1997) 117-120.

- V.R. Saunders, *Methods in Computational Molecular Physics*, Ed. by GHF Diercksen and S. Wilson, (Riedel, Dordrecht-Holland, 1983) 1-36.

`generate::Macrofort::init` – initialize genFor

`Mac::init` (where `Mac:=generate::Macrofort`) initializes the global variables for every call to `Mac::genFor`, the Macrofort FORTRAN code generator.

This procedure *must* be called before the first call to `Mac::genFor`. It initializes the various counters used internally by Macrofort for label generation within used within e.g.: do-loops and format statements. It also sets the variables for precision, comments, optimization, I/O settings, input and output file definition to their default values although these can be subsequently modified by subsidiary routines (mentioned in the list of "Related Functions").

Call(s):

- `generate::Macrofort::init()`

Return Value: the void object of domain type `DOM_NULL`

Side Effects: Default values for global variables `Mac::genFor` are set.

Related Functions: `generate::Macrofort::genFor`, `generate::Macrofort::setOptimizedOption`, `generate::Macrofort::setIOSettings`, `generate::Macrofort::setPrecisionOption`, `generate::Macrofort::setAutoComment`, `generate::Macrofort::openOutputFile`, `generate::Macrofort::closeOutputFile`, `generate::optimize`, `generate::fortran`

Details:

- With `Mac:=generate::Macrofort`, these are the subsidiary routines to `Mac::init` and `Mac::genFor` within the "macrofort" domain:

  `Mac::setAutoComment` allows for the automatic generation of FORTRAN comments.

  `Mac::setPrecisionOption` allows the choice of tailoring FORTRAN code in single, double or quadruple precision.

`Mac::setIOSettings` allows the user to choose the settings for FORTRAN I/O statements.

`Mac::setOptimizedOption` allows for the choice of optimization.

`Mac::openOutputfile` opens an ascii file for the FORTRAN code.

`Mac::closeOutputfile` closes the file open by `Mac::openOutputfile`.

See the help-files of these individual procedures for more details and information about the default values set by `Mac::init`.

Example 1. This example shows how `Mac::init` (with `Mac:=generate::Macrofort`) automatically takes care of the labeling in the FORTRAN output generated by `Mac::genFor`. The output is the ascii file `"test.f"`

```
>> Mac := generate::Macrofort:
   Mac::init():
   Mac::openOutputFile("test.f"):
   Mac::genFor(["continue",label]):
   Mac::genFor(["do",label,i,1,m,-1]):
   Mac::closeOutputFile():
   delete i, m, label:
```

The output file `"test.f"` is:

```
 1000 continue
      do 1000, i=1,m,-1
```

As we can see, the labeling was done automatically without the user having to worry about it.

See the help-file for `Mac::genFor` for a more comprehensive list of examples.

`generate::Macrofort::closeOutputFile` – close FORTRAN file

`Mac::closeOutputFile` (where `Mac:=generate::Macrofort`) closes the FORTRAN file opened by `Mac::openOutputFile`.

Call(s):

- `generate::Macrofort::closeOutputFile()`

Return Value: the void object of domain type `DOM_NULL`

Side Effects: Closes FORTRAN file opened by `Mac::openOutputFile`.

Related Functions: `generate::Macrofort::genFor`, `generate::Macrofort::openOutputFile`

Details:

- `Mac::closeOutputFile` (where `Mac:=generate::Macrofort`) closes the ascii file on which all FORTRAN code generated by successive calls to `Mac::genFor` (see `Mac::genFor` and `Mac::init` for more details) was written.

  `Mac::closedOutputFile` MUST be called at the end of the last call to `Mac::genFor`.

  The help-pages of `Mac::genFor`, `Mac::init` and related functions provide examples.

`generate::Macrofort::openOutputFile` – **open FORTRAN file**

`Mac::openOutputFile` (where `Mac:=generate::Macrofort`) allows to create the ascii file on which resides the FORTRAN code generated by `Mac::genFor`.

Call(s):

- `generate::Macrofort::openOutputFile(b)`

Parameters:

`b` — the file name: a string.

Return Value: A positive integer specifying a file descriptor or FAIL if the file can't be opened.

Side Effects: Creates the ascii file to which the FORTRAN code generated by all calls to `generate::Macrofort::genFor` is written.

Related Functions: `generate::Macrofort::init`, `generate::Macrofort::genFor`, `generate::Macrofort::closeOutputFile`

Details:

- `Mac::openOutputFile` (where `Mac:=generate::Macrofort`) opens the ascii file on which all FORTRAN code made by subsequent calls to `Mac::genFor` is to be generated (see `Mac::genFor` and `Mac::init` for more details). The input must be a string with a termination like `".f"` to ensure a FORTRAN file acceptable to most FORTRAN compilers.

`Mac::openOutputFile` MUST be called before the first call to `Mac::genFor`. The help-pages of `Mac::genFor`, `Mac::init` and related functions provide examples.

`generate::Macrofort::setAutoComment` – **automatic comments**

`Mac::setAutoComment` (where `Mac:=generate::Macrofort`) is a switch to ensure that FORTRAN code generated by `Mac::genFor` includes FORTRAN comments.

Call(s):

- `generate::Macrofort::setAutoComment(b)`

Parameters:

`b` — TRUE or FALSE.

Return Value: the void object of domain type `DOM_NULL`

Side Effects: Resets the internal macrofort variable for the FORTRAN code generated by `generate::Macrofort::genFor`.

Related Functions: `generate::Macrofort::init`, `generate::Macrofort::genFor`

Details:

- `Mac::setAutoComment` (where `Mac:=generate::Macrofort`) is used with `Mac::genFor` and `Mac::init` (see these programs for more details) and adjusts Macrofort (internal) global variable for generation of FORTRAN comments. The default setting for this variable made by an initial call to `Mac::init` is TRUE for the resulting FORTRAN code.

  When a boolean value of FALSE is injected to `Mac::setAutoComment`, the FORTRAN code of `Mac::genFor` is generated without FORTRAN comments.

Example 1.

```
>> Mac:=generate::Macrofort:
   Mac::init():
```

Note that the default mode for the automatic comments set by `Mac::init` is TRUE (meaning on).

```
>> Mac::openOutputFile("test.f"):
   Mac::genFor(["subroutinem", foo, [a, b, i],
                [["equal", a, 1], ["equal", b, 2]]]):
   Mac::closeOutputFile():
```

Switch auto-comment off and send output to a different file.

```
>> Mac::setAutoComment(FALSE):
   Mac::openOutputFile("test2.f"):
   Mac::genFor(["subroutinem", foo, [a, b, i],
                [["equal", a, 1], ["equal", b, 2]]]):
   Mac::closeOutputFile();
```

The output file with comments `test.f` is:

```
 c
 c     SUBROUTINE foo
 c
       subroutine foo(a,b,i)
         a = 1
         b = 2
       end
```

The output file without comments `test2.f` is:

```
       subroutine foo(a,b,i)
         a = 1
         b = 2
       end
```

`generate::Macrofort::setIOSettings` – **sets I/O settings**

`Mac::setIOSettings` (where `Mac:=generate::Macrofort`) sets the unit or channel numbers for the input and output of the FORTRAN code generated by `Mac::genFor`.

Call(s):

- `generate::Macrofort::setIOSettings(inputf,outputf)`

Parameters:

`inputf,outputf` — integers for I/O specifications.

Return Value: the void object of domain type `DOM_NULL`

Side Effects: Resets the I/O settings for the FORTRAN code generated by `generate::Macrofort::genFor`.

Related Functions: `generate::Macrofort::init`, `generate::Macrofort::genFor`

Details:

- In FORTRAN, `read`, `write`, `open` and `close` statements require unit (or channel) numbers. In this sample of FORTRAN code:

  ```
  read(5,100) x
  write(6,200) y
  ```

  the first entry in the `read` instruction is the unit number of the file (or device) by which the input x is read. Similarly, the first entry in the `write` instruction is the unit number of the file (or device) on which y is to be output.

  `Mac::setIOSettings` (where `Mac:=generate::Macrofort`) is used with `Mac::genFor` and `Mac::init` (see these programs for more details) and adjusts the (internal) global I/O unit settings for FORTRAN. The default setting for these variables is respectively 5 and 6 for FORTRAN read and write statements which are made by an initial call to `Mac::init`.

  These I/O settings are needed internally for Macrofort Macro instructions such as e.g. while-do loops. The user also has the choice of injecting his own choice of unit number for `open` and `close` and `read` and `write` statements quite independently from these global variables. This allows the FORTRAN code to read input from several different ascii files without having to call `Mac::setIOSettings` each time. However, in general practice, it suffices to call this procedure once and define I/O variables common to both the user and Macrofort for all `read`, `write`, `open` and `close` statements.

Example 1. Example of `"openm"` Macrofort Statement.
This example illustrates that one can use the Macrofort global variable for the FORTRAN input unit number or some other choice. The resulting FORTRAN code can read data from different files.
First initialize Macrofort and open the ascii file `"test.f"`:

```
>> Mac:=generate::Macrofort:
   Mac::init():
   Mac::openOutputFile("test.f"):
```

Set global FORTRAN I/O settings at 5 and 6 respectively:

```
>> inputf := 5:  outputf := 6:
   Mac::SetIOSettings(inputf, outputf):
```

Generate Open statement with input setting at 10 and then at `inputf`:

```
>> Mac::genFor(["openm", 10, "toto.data", old,
                ["readm", 10, ["i10"], [j]]]):
   Mac::genFor(["openm", inputf, "toto2.data", old,
                ["readm", inputf, ["i10"], [j]]]):
   Mac::closeOutputFile():
   delete j, old, inputf, outputf:
```

The output file `test.f` is:

```
      open(unit=10,file='toto.data',status='old')
      read(10,2000) j
 2000 format(i10)
      close(10)
      open(unit=5,file='toto2.data',status='old')
      read(5,2001) j
 2001 format(i10)
      close(5)
```

Example 2. Example of Macro `"whilem"` Macrofort Statement.
This example uses the Macrofort internal global variable for the FORTRAN output instructions needed in the while-do loop made by the FORTRAN `write` instruction (via the macro `"writem"`) inside the `"whilem"` Macrofort instruction.

```
>> Mac::openOutputFile("test2.f"):
   Mac::genFor(["whilem", abs(a) > eps, ["equal", a, big],
                [["equal", a, a/2.0], ["equal", b,2 ]], 1000]):
   Mac::closeOutputFile():
   delete a, b, big, eps:
```

The output file `test2.f` is:

```
c
c     WHILE  (eps < abs(a)) DO <WHILE_LIST> (1)
c
c     WHILE LOOP INITIALIZATION
          maxwhile1 = 1000
          nwhile1 = 0
        a = big
c
c     WHILE LOOP BEGINNING
 1000 continue
```

```
c
c     WHILE LOOP TERMINATION TESTS
      if (eps.lt.abs(a)) then
        if (nwhile1.le.maxwhile1) then
c
c         NEW LOOP ITERATION
          nwhile1 = nwhile1+1
c
c         <WHILE_LIST>
          a = 0.5E0*a
          b = 2
        goto 1000
        else
c
c         WHILE LOOP TERMINATION :
c         BYPASSING THE MAXIMUM ITERATION NUMBER
          write(6,2002)
 2002     format(' maxwhile1 ')
        endif
c
c     NORMAL WHILE LOOP TERMINATION
      endif
c     WHILE LOOP END (1)
```

See the help-file for `Mac::genFor` for a more comprehensive list of examples.

`generate::Macrofort::setOptimizedOption` – sets optimization

`Mac::setOptimizedOption` (where `Mac:=generate::Macrofort`) is a switch which allows MuPAD's optimizer `generate::optimize` to be applied to the expressions and arrays of the FORTRAN code generated by `Mac::genFor`.

Call(s):

- `generate::Macrofort::setOptimizedOption(b)`

Parameters:

`b` — TRUE or FALSE.

Return Value: the void object of domain type `DOM_NULL`

Side Effects: Optimized FORTRAN code if the setting is TRUE.

Related Functions: `generate::optimize`, `generate::Macrofort::init`, `generate::Macrofort::genFor`

Details:

- `Mac::setOptimizedOption` (where `Mac:=generate::Macrofort`) is used with `Mac::genFor` and `Mac::init` (see these programs for more details) and adjusts Macrofort (internal) the settings for optimization. The default setting for this variable made by an initial call to `Mac::init` is FALSE i.e. no optimization for the resulting FORTRAN code.

  First, to understand how the optimizer works, consult `generate::optimize`. When a boolean value of TRUE is injected to `Mac::setOptimizedOption`, the `Mac::genFor` procedure tailors its FORTRAN code from the results of the optimizer. In the case of arrays, the input array is converted into a computational sequence in the form of a list before submission to the optimizer to facilitate the generation of readable FORTRAN code.

Example 1.

```
>> Mac := generate::Macrofort:
   Mac::init():
```

Note that the default mode for the optimizer set by generate::Macrofort::init is FALSE (meaning off).

```
>> Mac::openOutputFile("test.f"):
   Mac::genFor(["equal", [[a, 1 + sin(t)],
                          [b, cos(t) + sin(t)], [c, 1 + cos(t)]]]):
   Mac::closeOutputFile();
   delete a,b,c,t:
```

Switch the optimizer and send output to a different file.

```
>> Mac::openOutputFile("test2.f"):
   Mac::setOptimizedOption(TRUE):
   Mac::genFor(["equal", [[a, 1 + sin(t)],
                          [b, cos(t) + sin(t)], [c, 1 + cos(t)]]]):
   Mac::closeOutputFile():
   delete a, b, c, t:
```

The output file `test.f` is:

```
      a = sin(t)+1
      b = cos(t)+sin(t)
      c = cos(t)+1
```

The "optimized" output file `test2.f` is:

```
t1 = sin(t)
a = t1+1
t2 = cos(t)
b = t1+t2
c = t2+1
```

This example only shows how to call `Mac::setOptimizedOption` but does not show the advantages given by optimization. To appreciate these advantages, see the help-files of `generate::optimize` and `Mac::genFor` for a more comprehensive list of examples.

`generate::Macrofort::setPrecisionOption` – **sets precision**

`Mac::setPrecisionOption` (where `Mac:=generate::Macrofort`) sets the precision (single, double or quadruple) for the FORTRAN code generated by `Mac::genFor`.

Call(s):

- `generate::Macrofort::setPrecisionOption(s)`

Parameters:

`s` — one of the strings `"single"`, `"double"` or `"quadruple"`.

Return Value: the void object of domain type `DOM_NULL`

Side Effects: Resets the global macrofort precision variable for the FORTRAN code generated by `generate::Macrofort::genFor`.

Related Functions: `DIGITS`, `generate::Macrofort::init`, `generate::Macrofort::genFor`

Details:

- In FORTRAN, floats or floating-point numbers are implemented as hardware floats and their precision can be any of these three choices: 8 digits (`"single"`), 16 digits (`"double"`) or 32 digits (`"quadruple"`). Intrinsic FORTRAN functions are often written with a prefix `"d"` for double precision or a prefix `"q"` for quadruple precision. E.g. the functions `sin` or `cos` as used in single precision become respectively `dsin` `dcos` in double precision or `qsin` and `qcos` in quadruple precision (although some compilers adhere to a standard which does not need these prefixes). The advantage is that MuPAD readily allows the user to generate constants like `PI` to any accuracy without mistakes.

- `Mac::setPrecisionOption` (where `Mac:=generate::Macrofort`) is used with `Mac::genFor` and `Mac::init` (see these programs for more details) and adjusts the Macrofort (internal) global variable for precision. The default setting for this variable made by an initial call to `Mac::init` is `"single"` for the resulting FORTRAN code.
- Warning: The default setting for DIGITS in MuPAD is 10. Thus, if one desires meaningful and accurate code for double or especially for quadruple precision, the user must ensure sufficient accuracy of his input data (usually by resetting `DIGITS` and making a call to `float`). `Mac::genFor` resets the value of `DIGITS` to 8,16 or 32 depending on the choice of precision.

Example 1. Initialize Macrofort and open file `"test.f"`

```
>> Mac := generate::Macrofort:
   Mac::init():
   Mac::openOutputFile("test.f"):
```

This example generates the same FORTRAN assignment in respectively single, double and quadruple precisions.

```
>> Mac::setPrecisionOption("single"):
   Mac::genFor(["equal", a, 1.0 + PI*sinh(E)]):
   Mac::setPrecisionOption("double"):
   Mac::genFor(["equal", a, 1.0 + PI*sinh(E)]):
   Mac::setPrecisionOption("quadruple"):
   Mac::genFor(["equal", a, 1.0 + PI*sinh(E)]):
   Mac::closeOutputFile():
   delete a:
```

The output file `test.f` is:

```
      a = 0.3141593E1*sinh(0.2718282E1)+0.1E1
      a = 0.3141592653589793D1*dsinh(0.2718281828459045D1)+0.1D1
      a = 0.31415926535897932384626433832795Q1*qsinh(0.27182818284590452
     #353602874713527Q1)+0.1Q1
```

See the help-file for `Mac::genFor` for a more comprehensive list of examples.

`generate::optimize` – **generate optimized code**

`generate::optimize(..)` returns a sequence of equations representing an "optimized computation sequence" for the input expression. Each equation in the sequence corresponds to an assignment of a subexpression of the input

expression to a "temporary variable". Common subexpressions are computed only once, thus reducing the total operation count.

Call(s):

- `generate::optimize(r)`

Parameters:

`r` — an expression, array or list of equations

Return Value: a list of equations.

Related Functions: `generate::Macrofort::setOptimizedOption`, `generate::Macrofort::genFor`

Details:

- The output from `generate::optimize` represents a "computation sequence", i.e., a list of equations representing an optimized version of the input. Common intermediates are identified by the optimizer and "stored" in temporary variables `t1`, `t2` etc. Each equation in the sequence corresponds to an assignment to such a variable.

  The number of operations, namely additions (or subtractions), multiplications (or divisions) and in particular functions calls of the output is usually lower than the number of such operations of the input. This facility is useful for code generation (see `generate::Macrofort::setOptimizedOption`).

Example 1. In this first example, we show the effects of optimization for a simple expression:

```
>> generate::optimize(cos(x^2) + x^2*sin(x^2) + x^4)

                  2                                  2
          [t2 = x , t1 = cos(t2) + t2 sin(t2) + t2 ]
```

The "blind" computation of the input expression requires 7 multiplications, 2 additions and 2 function calls. The optimized version introduces a "temporary variable" `t2` storing the subexpression `x^2` that is used to compute the final result `t1`. This reduces the total cost to 3 multiplications, 2 additions and 2 function calls, albeit using 1 extra assignment to the temporary variable `t2`.

Example 2. Here we repeat the exercise of the first example but with an array of expressions:

```
>> generate::optimize(array(1..2, 1..2, [[x^3, x^2],[x^2, x^4]]))
```

```
                --                +-           -+ --
                |                 |  x t4,  t4  |  |
                |        2        |             |  |
                |  t4 = x , t3 =  |          2  |  |
                |                 |   t4,  t4   |  |
                --                +-           -+ --
```

The original input requires 6 multiplications. The optimized version needs only 3 multiplications and 1 extra assignment.

Example 3. We optimize a list of equations representing a computation sequence for 3 variables `t`, `C[1]`, `C[2]`:

```
>> generate::optimize([t = u, C[1] = t*(u - w)^2, C[2] = 2*(u -
w)^3])
```

```
                                  2
   [t = u, t5 = u - w, t6 = t5 , C[1] = t t6, C[2] = 2 t5 t6]
```

The original computation requires 5 multiplications and 2 subtractions. The optimized version needs 4 multiplications and 1 subtraction.

Note that since these examples involve small expressions, the computational savings are slight. In the case of very large expressions, optimization can yield a considerable dividend (c.f. the last example in the help-pages of `generate::Macrofort::genFor` for a real application of this implementation).

Background:

- A number of FORTRAN compilers provide optimizers. However, they use algorithms of complexity $O(n^2)$ and $O(n^3)$ where n is the size of the input expressions. For large amounts of code, these algorithms may "break". MuPAD provides a reasonably good scalar (as in non-vectorized and non-parallelized) optimizer which is limited to common subexpression optimization and using binary powering for integer powers. It uses hashing of expressions so that given a sub-expression, it can determine in constant time if this subexpression has already occurred. This results in an overall efficiency which is of lower complexity namely, $O(n)$ i.e. linear in the size of the input expressions to be optimized, Hence overall efficiency is not compromised by very large expressions. This does mean that not all possible optimizations are made but nonetheless a number of reductions including the exploitation of some symmetries are possible.

It should be understood that "optimization" is meant in the sense of compiler optimization. The end-result rarely corresponds to the absolute irreducible minimum number of operations - or as in the case of FORTRAN code generation, the absolute minimum of floating-point operations (FLOPS). Achieving this limit can be extremely difficult if not impossible especially for large computational sequences. Nonetheless, in a number of real-life instances, MuPAD's optimizer can yield a very useful result. Additionally, MuPAD provides symbolic manipulation tools such as `factor` which can yield additional reduction in operation costs.

In many cases of optimization, it is most often a matter of how best to pose the problem so as to fully exploit every possible symmetry or useful natural property of the given problem.

- Reference: T.C. Scott, I.P. Grant, M.B. Monagan and V.R. Saunders, *MapleTech*, **4**, no. 2, pp. 15-24, (1997).

`generate::TeX` – generate TEX formatted string from expressions

`generate::TeX(e)` generates TEX output for an expression `e`.

Call(s):

- `generate::TeX(e)`

Parameters:

`e` — an arithmetical expression

Return Value: `generate::TeX` returns a `string` containing TEX code.

Overloadable by: `e`

Related Functions: `fprint`, `print`

Details:

- `generate::TeX(e)` returns a TEX formatted string representing `e`. This string may be printed to a file using `fprint`. Use the printing option `Unquoted` to remove quotes and to expand special characters like newlines and tabs.
- The output string may be used in the math-mode of TEX. Note that `generate::TeX` doesn't break large formulas into smaller ones.

Example 1. `generate::TeX` generates a string containing the TEX code:

```
>> generate::TeX(hold(int)(exp(x^2)/x, x))

      "\\int \\frac{\\mbox{exp}\\left(x^2\\right)}{x} d x"
```

Use `print` with option `Unquoted` to get a more readable output:

```
>> print(Unquoted, generate::TeX(hold(int)(exp(x^2)/x, x)))

          \int \frac{\mbox{exp}\left(x^2\right)}{x} d x
```

Example 2. This example shows how to write a `"TeX"`-method for a domain. The domain elements represent open intervals. The `"TeX"`-method makes recursive use of `generate::TeX` in order to TEX-format its operands and concatenates the resulting strings to a new string containing the TEX output of the interval.

```
>> interval := newDomain("interval"):
   interval::TeX :=
     e -> "\\left]".generate::TeX(extop(e, 1)).
          ", ".generate::TeX(extop(e, 2))."\\right[":
   print(Unquoted,
         generate::TeX(new(interval, 1, x^(a+2)))):

                    \left]1, x^{a + 2}\right[
```

Background:

- A domain overloading `generate::TeX` has to provide a function as its `"TeX"`-slot which translates its elements into a TEX formatted string. This function may use `generate::TeX` recursively. Cf. example 2.

Changes:

- No changes.