`Pref` — user preferences

Table of contents

`Pref::alias` – **controls the output of aliased expressions**

`Pref::alias(TRUE)` switches the usage of `alias` abbrevations in outputs on.

`Pref::alias(FALSE)` switches the usage of `alias` abbrevations in outputs off.

`Pref::alias()` returns the current setting.

Call(s):

- `Pref::alias()`
- `Pref::alias(TRUE)`
- `Pref::alias(FALSE)`
- `Pref::alias(NIL)`

Return Value: the last defined value

Side Effects: `Pref::alias` changes the output of aliased expressions.

Related Functions: `alias`, `expr2text`, `fprint`, `print`

Details:

- An `alias` is an abbrevation for a MuPAD expression. If `Pref::alias` is enabled, the `alias` abbrevations will be used for output.
- `Pref::alias()` returns the current value.
- `Pref::alias(TRUE)` switches the usage of `alias` abbrevations in outputs on. This is the default setting.
- `Pref::alias(FALSE)` switches the usage of aliases in outputs off.
- `Pref::alias(NIL)` restores the default value which is `TRUE`.
- `Pref::alias` has no effect on `print` and `fprint`.

Example 1. If an aliased expression occurs in output, it is replaced by the alias abbrevation:

```
>> alias(X = a + b):
   X, a + b

                         X, X
```

This only works if the syntactical structure of expression matches the aliased expression:

```
>> 2*X

                              2 a + 2 b
```

`prog::exprtree` shows that `2*X` does not contain `a + b` any more:

```
>> prog::exprtree(X): prog::exprtree(2*X):

                                _plus
                                |
                                +-- a
                                |
                                `-- b

                              _plus
                              |
                              +-- _mult
                              |   |
                              |   +-- a
                              |   |
                              |   `-- 2
                              |
                              `-- _mult
                                  |
                                  +-- b
                                  |
                                  `-- 2
```

The same holds for `X+c`:

```
>> X + c; prog::exprtree(X + c):

                              a + b + c

                                _plus
                                |
                                +-- a
                                |
                                +-- b
                                |
                                `-- c
```

With `Pref::alias(FALSE)` the back translation of aliases in the output is disabled:

```
>> Pref::alias(FALSE):
   X
```

```
                                a + b
```

`Pref::alias` has no effect on `print` and `fprint` outputs:

```
>> Pref::alias(TRUE):
   print(X):

                                a + b
```

Changes:

- `Pref::alias` is a new function.

`Pref::ansi` – Use ANSI control sequences in terminal help

`Pref::ansi(FALSE)` causes `?` to emit plain text help. Use if your terminal cannot display ANSI/vt100.

Call(s):

- `Pref::ansi(<value>)`

Parameters:

`value` — `TRUE`, `FALSE`, or `NIL`

Return Value: the previously defined value

Related Functions: `help`

Details:

- When using the terminal version of MuPAD, the documentation will usually be displayed using ANSI/vt100 sequences for boldface and underlining. If your terminal or your pager do not support these sequences, you will see them printed directly, which resembles noise. Use `Pref::ansi(FALSE)` to switch these sequences off.
- A call of `Pref::ansi` without arguments will return the current value. The argument `NIL` will reset the default value, which is `TRUE`.

Changes:

- `Pref::ansi` is a new function.

`Pref::callBack` – informations during evaluation

`Pref::callBack(func)` defines a function `func`, that will be called frequently during evaluation.

Call(s):

- `Pref::callBack(<func>)`

Parameters:

`func` — function to display informations

Return Value: the previously defined function

Related Functions: `Pref::report`, `Pref::postInput`, `Pref::postOutput`

Details:

- The function `func` defined by `Pref::callBack(func)` will be called permanently, when the MuPAD kernel works. Therewith informations can be displayed to inform the user.
- A call of `Pref::callBack` without arguments returns the current value. The argument `NIL` resets the default value, which is `NIL`.

Example 1. The following combination of `Pref::postInput` (initialization) and time count with `Pref::callBack` shows the seconds during evaluating.

```
>> Pref::postInput(proc() begin START:= time(); TIME:= START end_proc):
   Pref::callBack(proc()
                  begin
                    if time() - TIME > 1000 then // 1 sec
                      TIME:= TIME+1000;
                      print((time() - START) div 1000)
                    end_if
                  end_proc):
   NOW:= time():
   while time() - NOW <= 10000 do 1 end_while:
```

```
1

2

3

4

5

6

7

8

9
```

Changes:

- No changes.

`Pref::callOnExit` – **defines an exit handler**

`Pref::callOnExit(f)` defines a function `f` which is called on exit of MuPAD.

Call(s):

- `Pref::callOnExit(f)`
- `Pref::callOnExit(list)`
- `Pref::callOnExit(NIL)`
- `Pref::callOnExit()`

Parameters:

`f` — a function
`list` — a list of functions

Return Value: `Pref::callOnExit` returns the previously defined function, list of functions, or `NIL`.

Related Functions: `_quit`, `Pref::postOutput`

Details:

- `Pref::callOnExit(f)` defines a function `f` which is called on exit of MuPAD.
- `Pref::callOnExit(list)` defines a list of functions which are executed in the order of their occurence in `list` on exit of MuPAD.
- `Pref::callOnExit(NIL)` sets the default value, which is `NIL`.
- `Pref::callOnExit()` returns the current value.

Example 1. This example shows how to print some text on exit of MuPAD. It only works in the UNIX terminal versions of MuPAD:

```
>> Pref::callOnExit(
     ()->print(Unquoted, "Good by, thank You for using MuPAD.")
   ):
   quit

                   Good by, thank You for using MuPAD.
```

In the frontends on all platforms the output is the following since

Background:

- `Pref::callOnExit` can be used to send communication modules a disconnect message or to remove temporary user-defined files when leaving MuPAD.

Changes:

- No changes.

`Pref::echo` – suppress displaying of user inputs

`Pref::echo(FALSE)` suppresses the displaying of user inputs.

Call(s):

- `Pref::echo(<value>)`

Parameters:

`value` — `TRUE`, `FALSE` or `NIL`

Return Value: the previously defined value

Details:

- `Pref::echo(FALSE)` suppresses the displaying of user inputs. This is useful when using MuPAD without frontend and inputs should not be printed on screen. `Pref::echo(TRUE)` enables the printing of any input.
- `Pref::echo`works only in the terminal version of MuPAD.
- A call of `Pref::echo` without arguments will return the current value. The argument `NIL` will reset the default value, which is `TRUE`.

Changes:

- No changes.

`Pref::floatFormat` – representation of floating point numbers

`Pref::floatFormat` controls the representation of floating point numbers.

Call(s):

- `Pref::floatFormat(<modus>)`

Parameters:

`modus` — the kind of representation as character `"e"`, `"f"`, `"g"` or `"h"`, or `NIL`

Return Value: the previously defined representation

Related Functions: `DIGITS`, `Pref::trailingZeroes`, `print`

Details:

- The argument of `Pref::floatFormat` can be one of `"e"`, `"f"`, `"g"` and `"h"`. These are the standard C–command `printf` switches.
- The meaning is:

  "e" exponential representation

  "f" decimal representation without exponents

  "g" mix between `"e"` and `"f"`, only numbers less than `2^-32` will be displayed with exponential representation

  "h" hexadecimal representation

- The default value is `"g"` (see examples).
- A call of `Pref::floatFormat` without arguments returns the current value. The argument `NIL` resets the default value, which is `"g"`.

Example 1. The default display:

```
>> Pref::floatFormat(NIL):
   12345.67890, 0.00012345

                  12345.6789, 0.00012345
```

The exponential representation:

```
>> Pref::floatFormat("e"):
   12345.67890, 0.00012345

                  1.23456789e4, 1.2345e-4
```

The mixed representation:

```
>> Pref::floatFormat("g"):
   12345.67890, 0.00000000012345

               12345.6789, 0.00000000012345
```

Hexadecimal display:

```
>> Pref::floatFormat("h"):
   12345.67890, 0.00012345

 0x0.CE6B7318FC50481*2^(0x00000E),

    0x0.81725B672EE3425A*2^(-0x00000C)
```

Changes:

- The character `"h"` can be used instead of `"x"` to switch to hexadecimal representation.

`Pref::ignoreNoDebug` – controls debugging of procedures

`Pref::ignoreNoDebug(TRUE)` allows debugging of `procedures` even if they have the option `noDebug` set.

Call(s):

- `Pref::ignoreNoDebug()`
- `Pref::ignoreNoDebug(TRUE)`
- `Pref::ignoreNoDebug(FALSE)`
- `Pref::ignoreNoDebug(NIL)`

Return Value: `TRUE` or `FALSE`

Related Functions: `debug`, `DOM_PROC`

Details:

- `Pref::ignoreNoDebug()` returns the current value.
- `Pref::ignoreNoDebug(TRUE)` causes the debugger to ignore the option `noDebug` of procedures. This allows to debug even such procedures.
- `Pref::ignoreNoDebug(NIL)` resets the default value, which is `FALSE`.
- `Pref::ignoreNoDebug(FALSE)` resets the default value, which is `FALSE`.

Changes:

- `Pref::ignoreNoDebug` is a new function.

`Pref::keepOrder` – **order of terms in sum outputs**

`Pref::keepOrder` influences the output order of terms in sums.

Call(s):

- `Pref::keepOrder(`*`Always`*`)`
- `Pref::keepOrder(`*`DomainsOnly`*`)`
- `Pref::keepOrder(`*`System`*`)`
- `Pref::keepOrder(NIL)`
- `Pref::keepOrder()`

Options:

`Always`	—	the output always corresponds to the internal order
`DomainsOnly`	—	only polynomials and domain elements are printed in their internal order
`System`	—	the output system always decides the output order

Return Value: the previously defined value: *`Always`*, *`DomainsOnly`*, or *`System`*.

Related Functions: `DOM_POLY`, `Dom::MultivariatePolynomial`, `Dom::Polynomial`, `Dom::UnivariatePolynomial`, `print`

Details:

- Usually, the output system uses its own ordering of the terms in a `sum` to optimize the appearance of the output. This order may be different from the internal ordering of the sum. The output system prefers to re-order the terms such that the first term is positive.
- Sometimes it is desirable to see the terms of a sum in the internal order. This can be achieved with `Pref::keepOrder(`*`Always`*`)`.
- By default, the term order of polynomials and domain elements is left unchanged.
- `Pref::keepOrder(NIL)` restores the default state, which is `DomainsOnly`.
- `Pref::keepOrder()` returns the currently set value.

Option <*`Always`*>:

- The output always corresponds to the internal order.

Option <*`DomainsOnly`*>:

- In polynomials and domain elements, the ordering of terms corresponds to the internal order. Other sums may be re-ordered by the output system.
- This is the default setting of `Pref::keepOrder`.

Option <*`System`*>:

- The output order of terms in sums is determined by the output system and does not necessarily correspond to the internal order.

Example 1. Here we create a domain element `e`, an expression `f`, and a polynomial `p` containing sums. With the default setting *`DomainsOnly`*, only the output of the expression `f` is not in the internal order:

```
>> d := newDomain("d"):  d::print := x -> extop(x):
   e := new(d, b - a):  f := b - a:  p := poly(1 - x):
   e, f, p

                  - a + b, b - a, poly(- x + 1, [x])
```

With the setting *`Always`*, `e`, `f`, and `p` are all printed in the internal order:

```
>> Pref::keepOrder(Always):
   e, f, p

                 - a + b, - a + b, poly(- x + 1, [x])
```

With the setting *`System`*, the output order differs from the internal ordering for `e`, `f`, and `p`:

```
>> Pref::keepOrder(System):
   e, f, p

                    b - a, b - a, poly(1 - x, [x])
```

`Pref::keepOrder(NIL)` restores the default state; `Pref::keepOrder()` returns the current setting:

```
>> Pref::keepOrder(NIL):  Pref::keepOrder()

                              DomainsOnly
```

Changes:

- `Pref::keepOrder` is now taken into account in both `PRETTYPRINT` modes.

`Pref::kernel` – the version number of the presently used MuPAD kernel

`Pref::kernel()` returns the version number of the presently used kernel.

Call(s):

- `Pref::kernel()`

Return Value: the version number: a list of three nonnegative integers.

Related Functions: `patchlevel`, `version`

Details:

- The version numbers of the kernel and the library may differ. `Pref::kernel` refers to the kernel, whereas the call `version()` returns the version number of the installed MuPAD library.

Example 1. Do the version numbers of kernel and library coincide?

```
>> Pref::kernel() = version()

                     [2, 0, 0] = [2, 0, 0]
```

Changes:

- No changes.

`Pref::matrixSeparator` – sets the separator of matrix entries

`Pref::matrixSeparator("")` sets the separator of matrix entries to be a space.

Call(s):

- `Pref::matrixSeparator(<string>)`

Parameters:

`string` — string, that separates matrix entries for printing, or `NIL`

Return Value: the previously defined string

Related Functions: `print`

Details:

- The default value is `", "`. The separator string should not contain a tabulator, and should be at least one character.
- A call of `Pref::matrixSeparator` without arguments returns the current value. The argument `NIL` resets the default value.
- The setting of `Pref::matrixSeparator` only configures the ASCII output with `PRETTYPRINT` set to `TRUE`. It does not influence the typesetting output.

Example 1.

```
>> A:= array(1..2, 1..2, [[1, 2], [2, 3]])

                           +-      -+
                           |  1, 2  |
                           |        |
                           |  2, 3  |
                           +-      -+

>> Pref::matrixSeparator("   "):
   A

                           +-       -+
                           |  1   2  |
                           |         |
                           |  2   3  |
                           +-       -+
```

Changes:

- No changes.

`Pref::maxMem` – memory limit for calculation

`Pref::maxMem(kbyte)` sets a memory limit for calculations in kilobyte.

Call(s):

- `Pref::maxMem(<kbyte>)`

Parameters:

`kbyte` — integer in kilobyte, or `NIL`

Return Value: the last defined memory limit

Related Functions: `Pref::maxTime`, `bytes`, `MAXDEPTH`

Details:

- The value `0` effects no limitation.
- A call of `Pref::maxMem` without arguments returns the current value. The argument `NIL` resets the default value, which is `0`.

Example 1. The memory usage will be limited to a value above the current memory usage, displayed by `bytes`.

```
>> bytes()

                    507780, 717300, 2147483647

>> Pref::maxMem(1):
   [i] $ i = 1 .. 1000000:

 Error: Watchdog reset [watchdog-memory]
```

Changes:

- No changes.

`Pref::maxTime` – time limit for calculation

`Pref::maxTime(seconds)` sets a time limit for calculations in seconds.

Call(s):

- `Pref::maxTime(<seconds>)`

Parameters:

`seconds` — integer in seconds, or `NIL`

Return Value: the previously defined memory

Related Functions: `Pref::maxMem`, `time`

Details:

- The value `0` effects no limitation.
- A call of `Pref::maxTime` without arguments returns the current value. The argument `NIL` resets the default value, which is `0`.

Example 1. No computation can take more than ten seconds.

```
>> Pref::maxTime(10):
   TIME:= time(): while time() - TIME < 11111 do null() end_while

 Error: Watchdog reset [watchdog-time]
```

Changes:

- No changes.

`Pref::noProcRemTab` – disable "remember" tables

`Pref::noProcRemTab(TRUE)` disables the "remember" tables.

Call(s):

- `Pref::noProcRemTab(<value>)`

Parameters:

`value` — `TRUE`, `FALSE`, or `NIL`

Return Value: the last defined value

Side Effects: Without the "remember" tables the computation of any functions will be very much slower. The results are the same.

Related Functions: `proc`

Details:

- With `Pref::noProcRemTab(TRUE)` the "remember" tables of procedures can be disabled. `Pref::noProcRemTab(FALSE)` enables the "remember" tables.
- With the option `remember` of procedures results of calculations will be kept and "recycled": If a function will be called with the same arguments once again the previously calculated result will be returned immediately.
- A call of `Pref::noProcRemTab` without arguments returns the current value. The argument `NIL` resets the default value, which is `FALSE`.

Example 1. Because of the unclever definition, the function `fac` (factorial function) will be called permamently with the same arguments, and thats very often. The option `remember` corrects this, as a previous calculated result will be returned immediately without a new call of the function `fac`.

```
>> reset():
   fac:= proc(n = 1)
           option remember;
         begin
           if n > 2 then
             fac(n - 1)*fac(n - 2)
           else
             n
           end_if
         end_proc:
   time(fac(28))

                              890
```

Without this "remember" mechanism the effect of the unclever definition will be gigantic, even on a very hurry computer. Don't try `fac(32)`.

```
>> reset():
   Pref::noProcRemTab(TRUE):
   time(fac(28))

                             13600
```

Changes:

- No changes.

`Pref::output` – influence output of objects

With `Pref::output` the output of objects can be influenced.

Call(s):

- `Pref::output(function)`

Parameters:

`function` — function, that influence the output

Return Value: the previously defined value

Related Functions: `Pref::postOutput`, `Pref::postInput`, `Pref::keepOrder`

Details:

- With `Pref::output` a function can be defined that manipulates the output of objects.
- The given function will be called before any object will be outputted. The argument is the object, that will be outputted. The result of the function will be outputted instead of the given object.
- A call of `Pref::output` without arguments will return the current value. The argument `NIL` will reset the default value, which is `NIL`.

Example 1. All numbers shall be displayed as floating point numbers, but the input and calculations should not be influenced. Therefor a function, that applies `float` to all numeric objects, will be mapped to all objects of the result.

```
>> Pref::output(
     proc()
     begin
       map(args(), proc(num)
                   begin
                     if testtype(num, Type::Numeric) then
                       float(num)
                     else
                       num
                     end_if
                   end_proc)
     end_proc):
   1, 528/44, 194/8, 2 + 4/5*I
```

```
                    1.0, 12.0, 24.25, 2.0 + 0.8 I
```

In the next example the procedure `generate::TeX` will be applied to every output, before any object will be displayed.

```
>> Pref::output(generate::TeX):
   sqrt(x^2 - 1/x)

                    "\\sqrt{x^2 - \\frac{1}{x}}"
```

Changes:

- No changes.

`Pref::postInput` – actions after input

With `Pref::postInput`, actions directly after the data input can be initiated.

Call(s):

- `Pref::postInput(value)`

Parameters:

`value` — function to be executed after data input

Return Value: the previously defined function

Related Functions: `Pref::postOutput`, `Pref::promptString`

Details:

- With `Pref::postInput` a function can be defined to initiate actions after ending every complete input line with <RETURN>.
- The function will be called with the complete input line as argument.
- After the execution of the defined function the normally execution will be continued.
- `Pref::postInput` in joint with `Pref::promptString` und `Pref::postOutput` can be used to create status informations about evaluation. Possibilities are informations to time, memory usage, types of results etc. (see `Pref::postOutput`)
- A call of `Pref::postInput` without arguments will return the current value. The argument `NIL` will reset the default value, which is `NIL`.

Example 1. `Pref::postInput` will be used to numerate the input lines in joint with `Pref::promptString`. The global variable `NumberOfLine` must be initialized with `0`. This all can be done in the file “`userinit.mu`”.

```
>> NumberOfLine:= 0:
   Prompt:= Pref::promptString():
   Pref::postInput(proc()
                   begin
                     NumberOfLine:= NumberOfLine + 1;
                     Pref::promptString(expr2text(NumberOfLine) . Prompt)
                   end_proc):
```

Example 2. Time mesure in seconds.

```
>> Pref::postInput(() -> (TIME:= time())):
   Pref::postOutput(proc()
                      local Time;
                    begin
                      Time:= trunc((time() - TIME)/1000);
                      stringlib::format("Time: ".expr2text(Time)." s",
                                        TEXTWIDTH, Right)
                    end_proc):
   T:= time(): while time() - T < 1000 do null() end_while

                                                       Time: 2 s
```

The output depends on the value of the variable `TEXTWIDTH`.

Changes:

- No changes.

`Pref::postOutput` – actions after any output

`Pref::postOutput` controls user defined actions directly after any output.

Call(s):

- `Pref::postOutput(<func>)`

Parameters:

`func` — function to be executed after output, or `NIL`

Return Value: the last defined function

Related Functions: `Pref::postInput`, `Pref::promptString`

Details:

- `Pref::postOutput(func)` declares the function `func` to be called after every output.
- The function `func` will be called with the result of any evaluation after the output of the result.
- A call of `Pref::postOutput` without arguments returns the current value. The argument `NIL` resets the default value, which is `NIL`.
- `Pref::postOutput` in joint with `Pref::promptString` und `Pref::postInput` can be used to create status informations about evaluation. Possibilities are informations to time and memory usage, types of results etc. (see `Pref::postInput`)

Example 1.

```
>>
```

Example 2. `Pref::postOutput` will be used to numerate the output lines and show the type of the result. The global variable `NumberOfLine` must be initialized with `0`. This all can be done in the file "`userinit.mu`".

```
>> NumberOfLine:= 0:
   Pref::postOutput(proc()
                    begin
                      NumberOfLine:= NumberOfLine + 1;
                      stringlib::format(NumberOfLine, TEX-
TWIDTH, Right)."\n".
                      stringlib::format("Type: ".expr2text(map([args()], do
type)),
                                        TEXTWIDTH, Right)
                    end_proc):
```

Example 3. Time mesure in seconds.

```
>> Pref::postInput(() -> (TIME:= time())):
   Pref::postOutput(proc()
                      local Time;
                    begin
                      Time:= trunc((time() - TIME)/1000);
                      stringlib::format("Time: ".expr2text(Time)." s",
                                        TEXTWIDTH, Right)
                    end_proc):
   T:= time(): while time() - T < 1000 do null() end_while
```

```
                                                       Time: 2 s
```

The output depends on the value of the variable `TEXTWIDTH`.

Example 4. Show all identifiers of the result, that have properties assumed by the user.The first assignment to `ID` selects all identifiers of the output, that have properties. The second assignment to `ID` collects the properties of all identifiers.

```
>> Pref::postOutput(proc()
                      local ID;
                    begin
                      if args(0) > 0 then
                        ID := select(indets(args()), property::hasprop);
                      else
                        ID := {};
                      end_if;
                      if nops(ID) > 0 then
                        stringlib::format("Props: ".expr2text(op(ID)),
                                          TEXTWIDTH, Right)
                      else
                        null()
                      end_if
                    end_proc):
   assume(a>0): a
```

```
                                a
                                                        Props: a
```

The output depends on the value of the variable `TEXTWIDTH`.

Changes:

- No changes.

`Pref::prompt` – visible “prompt”

`Pref::prompt` determines, whether the string in front of any input line (the “prompt”) will be printed.

Call(s):

- `Pref::prompt(<value>)`

Parameters:

`value` — `TRUE` (to enable the prompt), `FALSE` (to disable the prompt), or `NIL`

Return Value: the last defined value

Related Functions: `Pref::promptString`, `Pref::echo`

Details:

- The “prompt” is a string that will be printed in front of each input line to mark it as input line. With `Pref::prompt(FALSE)` the “prompt” can be disabled (and also be enabled again with `Pref::prompt(TRUE)`).
- A call of `Pref::prompt` without arguments will return the current value. The argument `NIL` will reset the default value, which is `TRUE`.

Example 1. Disabling the “prompt”:

```
>> Pref::prompt(FALSE):
```

Background:

- In combination with `Pref::echo`, MuPAD can be caused to be totally quiet, that only the output can be seen.

Changes:

- No changes.

`Pref::promptString` – user defined “prompt”

`Pref::promptString` determines the string, that will be printed in front of any input line—the “prompt”.

Call(s):

- `Pref::promptString(<prompt>)`

Parameters:

`prompt` — string, that contains the “prompt”, or `NIL`

Return Value: the last defined string

Related Functions: `Pref::prompt`, `Pref::postInput`, `Pref::postOutput`

Details:

- `Pref::promptString` changes the user “prompt”.
- The “prompt” is the string, that will be printed in front of each input line to mark it as input line.
- The prompt can be extended to display additional information. In joint with `Pref::postInput` and `Pref::postOutput` the “prompt” can be formed alterable (see example 2).
- A call of `Pref::promptString` without arguments returns the current value. The argument `NIL` will reset the default value, which is `"» "`.

Example 1. Prints out the current prompt:

```
>> Pref::promptString()

                              ">> "
```

Example 2. `Pref::promptString` will be used to numerate the input lines in joint with `Pref::postInput`. I.e., after each input the variable `NumberOfLine` will be incremented. The variable `NumberOfLine` must be initialized with `0`. (This all could be done in the file “`userinit.mu`”.)

```
>> NumberOfLine:= 0:
   Pref::postInput(proc()
                   begin
                     NumberOfLine:= NumberOfLine + 1;
                     Pref::promptString(expr2text(NumberOfLine) . " >> ")
                   end_proc):
```

Changes:

- No changes.

`Pref::report` – informations during evaluation

`Pref::report` controls the output of informations during evaluation.

Call(s):

- `Pref::report(level)`

Parameters:

`level` — integer level between `0` and `9`, or `NIL`

Return Value: the last defined level

Related Functions: `Pref::callBack`

Details:

- `Pref::report` controls the frequence of report messages of the MuPAD kernel during evaluation.
- A kernel function displayes frequently the three informations *memory used*, *memory reserved* and *evaluation time in seconds*.
- The level `0` disables printing information. If `level` is `1`, about every hour a message will be printed. With `9` as argument the most reports will be printed. The frequency is dependent on the machines speed.
- A call of `Pref::report` without arguments returns the current value. The argument `NIL` resets the default value, which is `0`.

Example 1. Frequently information:

```
>> Pref::report(9):
   limit((1+1/n)^n,n=infinity)
```

```
 [used=1612k, reserved=1738k, seconds=1]
 [used=2716k, reserved=2856k, seconds=2]

                                             exp(1)
```

Reset to no information:

```
>> Pref::report(0):
```

Changes:

- The frequency of message was decreased.

`Pref::timesDot` – determines the output of products

`Pref::timesDot(str)` sets the output of the multiplication symbol in products to the string `str`.

Call(s):

- `Pref::timesDot()`
- `Pref::timesDot(str)`
- `Pref::timesDot(i)`
- `Pref::timesDot(NIL)`

Parameters:

`str` — a `string`
`i` — an `integer` between 1 and 255
`NIL` — the MuPAD object `NIL`

Return Value: the previously defined value

Side Effects: Changes the output of products.

Related Functions: `_mult`, `print`

Details:

- `Pref::timesDot` determines the output of the multiplication symbol between factors of a product in `PRETTYPRINT` output mode.
- `Pref::timesDot()` returns the current multiplication symbol.
- `Pref::timesDot(str)` sets the multiplication symbol to the given string `str`.
- `Pref::timesDot(i)` interprets `i` as an ASCII code and sets the multiplication symbol to the corresponding ASCII character.
- `Pref::timesDot(NIL)` restores the default value: the blank character.

Example 1. By default factors of products are separated by blanks.

```
>> a*b

                              a b
```

This can be changed by calling `Pref::timesDot` with an argument of type `string`.

```
>> Pref::timesDot(" * "):
   a*b

                             a * b
```

183 is the ASCII code for the character '·'.

```
>> Pref::timesDot(183):
   a*b

                              a·b
```

`NIL` restores the default output.

```
>> Pref::timesDot(NIL):
   a*b

                              a b
```

Changes:

- `Pref::timesDot` is a new function.

`Pref::trailingZeroes` – trailing zeroes when printing floating point numbers

`Pref::trailingZeroes` determines, whether trailing zeroes will be appended, when floating point numbers are printed.

Call(s):

- `Pref::trailingZeroes(value)`
- `Pref::trailingZeroes(<NIL>)`

Parameters:

`value` — TRUE, FALSE or `NIL`

Return Value: the last defined value

Related Functions: `DIGITS`, `Pref::floatFormat`, `print`

Details:

- If enabled (with argument `TRUE`), after the significant numbers of a floating point number (behind the point) zeroes will be appended until the number of digits reaches the value of `DIGITS`.
- A call of `Pref::trailingZeroes` without arguments will return the current value. The argument `NIL` will reset the default value, which is `FALSE`.

Example 1. By default trailing zeroes will not be displayed:

```
>> DIGITS:= 10:
   1.4

                                          1.4
```

Display of trailing zeroes will be enabled:

```
>> Pref::trailingZeroes(TRUE):
   1.4

                                       1.400000000
```

Changes:

- No changes.

`Pref::typeCheck` – type checking of formal parameters

`Pref::typeCheck` determines the kind of type checking of procedure parameters.

Call(s):

- `Pref::typeCheck(value)`
- `Pref::typeCheck(<NIL>)`

Parameters:

`value` — one of `Always`, `Interactive`, `None`, or `NIL`

Return Value: the last defined value

Related Functions: `args`, `DOM_PROC`, `domtype`, `hastype`, `proc`, `testargs`, `testtype`, `Type`, `type`

Details:

- The definition of a MuPAD procedure can be contain formal parameters. A type can be determined to every formal parameter with a new syntax. If the type checking will be enabled, the types of given parameters of such a procedure will be checked and results an error if it fails.
- As types MuPAD standard types and objects of the domain `Type` can be used. With `Type`, user defined types can be easily added to the system to extend the type checking mechanism.
- The arguments of `Pref::typeCheck` can be:

  `None` no parameter will be checked

  `Interactive` only when calculate interactively the formal parameters will be checked (default)

  `Always` the formal parameters will always be checked
- The default value `Interactive` means: When the user is calling a procedure `f`, their parameters will be checked, but all procedures, that will be called by the user called procedure `f`, performs no type checking.
- A call of `Pref::typeCheck` without arguments returns the current value. The argument `NIL` resets the default value, which is `Interactive`.

Example 1. The parameters of the procedure `f` must be an identifier followed by an integer:

```
>> f:= proc(a : DOM_IDENT, b : DOM_INT)
       begin
         evalassign(a, b)
       end_proc:
   f(a, 2)

                                             2
```

Now `a` has the value `2`, but an identifier is expected:

```
>> f(a, a + 2)

 Error: Wrong type of 2. argument (type 'DOM_INT' expected,
        got argument 'a + 2');
 during evaluation of 'f'
```

Background:

- The new syntax to test parameters directly (without a test in the procedure body) is the formal parameter followed by a colon and then the type object: `proc(a :  DOM_IDENT, b :  Type::Integer)`. That means: `a` must be of the type `DOM_IDENT` and `b` must be of the type `Type::Integer`.
- The objects of `Type` covers generally more objects as the MuPAD kernel types.

Changes:

- `Pref::typeCheck` is a new function.

`Pref::userOptions` – additionally options when starting MuPAD

`Pref::userOptions()` returns additional options, given by the user when calling MuPAD.

Call(s):

- `Pref::userOptions()`

Return Value: the user defined options as strings

Details:

- When starting the MuPAD kernel with the flag `"-U"` the user can define options, that can be used in the MuPAD session.

Example 1. When enter MuPAD with the command `mupad -U "Hello World"` the current directory will be stored and can be restored with `Pref::userOptions`:

```
>> Pref::userOptions()

                                   "Hello World"
```

Changes:

- No changes.

`Pref::verboseRead` – shows reading of files

With `Pref::verboseRead` the reading of library files can be shown.

Call(s):

- `Pref::verboseRead(value)`

Parameters:

`value` — `0`, `1`, `2` or `NIL`

Return Value: the last defined value

Related Functions: `read`, `fread`, `prog::trace`, `loadproc`

Details:

- With `Pref::verboseRead` the reading of library packages and files can be shown.
- The arguments of `Pref::verboseRead` stays for

  `0` no messages when reading files (default)

  `1` message if a library packages will be read

  `2` messages if a package or any library function will be read
- A call of `Pref::verboseRead` without arguments returns the current value. The argument `NIL` will reset the default value, which is `0`.

Example 1. Show the reading of library packages:

```
>> reset():
   Pref::verboseRead(1):
   sin(x)
```

```
 loading package 'Type' [mupad/share/lib/lib.tar#lib/]

                                0.8414709848
```

Show reading of all library files:

```
>> reset():
   Pref::verboseRead(2):
   sin(1.0)
```

```
 reading file mupad/share/lib/lib.tar#lib/SPECFUNC/sin.mu
 reading file mupad/share/lib/lib.tar#lib/SPECFUNC/sinh.mu
 reading file mupad/share/lib/lib.tar#lib/STDLIB/infinity.mu
 loading package 'Type' [mupad/share/lib/lib.tar#lib/]
 reading file mupad/share/lib/lib.tar#lib/TYPE/Arith.mu

                                0.8414709848
```

Changes:

- No changes.

`Pref::warnChanges` – **warnings about changes wrt. the previous version of MuPAD**

`Pref::warnChanges(TRUE)` switches on parser warnings about the usage of obsolete features from previous MuPAD versions.

Call(s):

- `Pref::warnChanges()`
- `Pref::warnChanges(TRUE)`
- `Pref::warnChanges(FALSE)`
- `Pref::warnChanges(NIL)`

Return Value: the previously defined value

Side Effects: Allows or suppresses warning messages.

Further Documentation: changes

Related Functions: `Pref::warnDeadProcEnv`, `Pref::warnLexProcEnv`

Details:

- `Pref::warnChanges()` returns the current value.
- `Pref::warnChanges(TRUE)` switches warning messages on. Now the parser warns if environment variables are declared as local variables (use *save* instead) or obsolete environment variables are used.
- `Pref::warnChanges(NIL)` or `Pref::warnChanges(FALSE)` will reset the default value, which is `FALSE`.

Example 1. If an environment variable is declared as local variable a warning is given:

```
>> Pref::warnChanges(TRUE):
   p := proc() local DIGITS; begin x end:

 Warning: Former environment variable 'DIGITS' used as lo-
cal [l\
 ine 2, col 25]
```

Use *save* instead of *local* for environment varibles:

```
>> p := proc() save DIGITS; begin x end:
```

```
>> Pref::warnChanges(FALSE):
```

Example 2. `ERRORLEVEL` is obsolete:

```
>> Pref::warnChanges(TRUE):
   p := proc() begin ERRORLEVEL end:
```

```
 Warning: Obselete environment variable 'ERRORLEVEL' used \
 [_check_global]
```

```
>> Pref::warnChanges(FALSE):
```

Changes:

- `Pref::warnChanges` is a new function.

`Pref::warnDeadProcEnv` – warnings about wrong usage of lexical scope

`Pref::warnDeadProcEnv()` returns the current setting.

`Pref::warnDeadProcEnv(TRUE)` switches on warnings about unreachable procedure environments.

`Pref::warnDeadProcEnv(FALSE)` switches warning messages off.

`Pref::warnDeadProcEnv(NIL)` will reset the default value, which is `FALSE`.

Call(s):

- `Pref::warnDeadProcEnv()`
- `Pref::warnDeadProcEnv(TRUE)`
- `Pref::warnDeadProcEnv(FALSE)`
- `Pref::warnDeadProcEnv(NIL)`

Return Value: the previously defined value; `TRUE` or `FALSE`

Side Effects: Allows or suppresses warning messages.

Further Documentation: changes

Related Functions: `Pref::warnChanges`, `Pref::warnLexProcEnv`, `proc`

Details:

- If a procedure is executed a *procedure environment* is created for this procedure. It contains the current values of formal parameters and local variables. On exit of the procedure this procedure environment is normally not needed any more and destroyed.
- If a procedure returns a local procedure as its result, this local *procedure escapes its scope*. Usually this is no problem. Only if the escaping procedure contains references to formal parameters or local variables of the outer procedure these *variables escape their scope*. These variables can not be dereferenced since they reference values of a procedure environment of the outer procedure which does not exist any more.
- Use option *escape* in the outer procedure in order to keep its procedure environment untouched.

Example 1. Here we write procedure `p` which returns a local procedure. The returned procedure adds the value of its argument `y` to the value of the argument `x` of the first procedure. The following naive implementation produces a strange output and, when the resulting procedure is called, a warning message and an error:

```
>> Pref::warnDeadProcEnv(FALSE):
   p := proc(x) begin y -> x + y end:
   f := p(1);  f(2)

                     y -> DOM_VAR(1,2) + y
 Warning: Uninitialized variable 'unknown' used;
 during evaluation of 'f'
 Error: Illegal operand [_plus];
 during evaluation of 'f'
```

If `Pref::warnDeadProcEnv` is set to `TRUE` MuPAD will print a warning message when the local procedure escapes its scope:

```
>> Pref::warnDeadProcEnv(TRUE):
   p := proc(x) begin y -> x + y end:
   f := p(1)

 Warning: Found dead closure of procedure 'p'

                     y -> DOM_VAR(1,2) + y
```

Use option *escape* in the outer procedure to prevent this warning. The returned procedure `f` will then work as expected:

```
>> p := proc(x) option escape; begin y -> x + y end:
   f := p(1);  f(2)

                           y -> x + y

                                3
```

Changes:

- `Pref::warnDeadProcEnv` is a new function.

`Pref::warnLexProcEnv` – warnings about usage of variables from lexical scope

`Pref::warnLexProcEnv()` returns the current setting.

`Pref::warnLexProcEnv(TRUE)` switches on parser warnings about the usage of variables from the lexical scope.

`Pref::warnLexProcEnv(FALSE)` switches warning messages off.

`Pref::warnLexProcEnv(NIL)` will reset the default value, which is `FALSE`.

Call(s):

- `Pref::warnLexProcEnv()`
- `Pref::warnLexProcEnv(TRUE)`
- `Pref::warnLexProcEnv(FALSE)`
- `Pref::warnLexProcEnv(NIL)`

Return Value: the previously defined value; `TRUE` or `FALSE`

Side Effects: Allows or suppresses warning messages of the parser.

Further Documentation: changes

Related Functions: `Pref::warnChanges`, `Pref::warnDeadProcEnv`, `proc`

Details:

- If `Pref::warnLexProcEnv` is enabled the parser warns if a procedure defined in the lexical scope of another procedure uses variables from its lexical scope.
- These warnings are not always critical. The example below shows a procedure which initiates a warning message but works without problems.
- `Pref::warnDeadProcEnv` switches on warnings about the critical usage of the lexical scope.

Example 1. Here you can see a procedure which computes the square of its argument in a very complicated way. The inner procedure `g` makes use of the variable `x` of procedure `f`, thus a warning is given. But this is no problem, since `g` does not escape its scope. See `Pref::warnDeadProcEnv` about problems of procedure leaving its scope.

```
>> Pref::warnLexProcEnv(TRUE):
   f := proc(x) local g; begin g := y -> x*y; g(x) end:
   f(5)

 Warning: Procedure '->' is referring outer lexical closure \
  [col 42]

                                  25
```

Changes:

- `Pref::warnLexProcEnv` is a new function.