`prog` — programmer's toolbox

Table of contents

Library `prog`

This library contains programming utilities.

With this utilities MuPAD functions can be analyzed, traced and errors can be found easier.

`prog::allFunctions` – **overview of all functions**

`prog::allFunctions` reads in all functions and libraries and all names will be printed.

Call(s):

- `prog::allFunctions(<`*`Recursive`* `<, `*`Recursive`*`>>)`

Options:

`Recursive` — also domains and their operands will be examined

Return Value: the void object `null()`

Related Functions: `prog::init`

Details:

- `prog::allFunctions` initializes all functions and libraries and prints a table with all these objects. The functions will be sorted alphabetically.

Option <*`Recursive`*>:

- With *`Recursive`* given once, domains will be examined too and their operands be printed. If *`Recursive`* is given twice, also these operands will be inspected. This may take a while...

Example 1. `prog::allFunctions` reads in and prints out every library and function:

```
>> prog::allFunctions()

 Warning: Reading all MuPAD objects, this may take a while... [\
 prog::allFunctions]
 DOM_EXEC: [189]
 _act_proc_env    _and          _assign
 _break           _case         _concat
 ...

>> prog::allFunctions(Recursive, Recursive)
```

```
Warning: Reading all MuPAD objects, this may take a while... [\
prog::allFunctions]
DOM_EXEC: [189]
_act_proc_env    _and           _assign
_break           _case          _concat
...
```

Changes:

- `prog::allFunctions` is a new function.

`prog::calltree` – visualize the call structure of nested function calls

With `prog::calltree` the call structure of nested function calls can be visualized.

Call(s):

- `prog::calltree(statement <, maxdepth > <, excl_funcs > <, option>)`

Parameters:

`statement`	—	a MuPAD statement to examine
`maxdepth`	—	maximal "call depth" to show calls on screen
`excl_funcs`	—	set of MuPAD objects to exclude from showing on screen
`option`	—	a set or one of *`Plain`* and *`Args`*

Options:

`Args`	—	The arguments of any function call are printed additionally.
`Tree`	—	The call of `prog::calltree` returns an object of type `adt::Tree`.

Return Value: `prog::calltree` returns the result of the execution of `statement`. Additionally, information on the flow of control is printed.

Related Functions: `prog::trace`, `setuserinfo`, `debug`, `prog::profile`

Details:

- `prog::calltree` visualizes the call structure of nested function calls.
- `statement` can be any MuPAD statement. If a function is called, `prog::calltree` prints all functions called while executing `statement` in a tree structure.
- Using this output, dependencies between functions can be observed and analyzed.

Option <*Args*>:

- The arguments of every function call are printed.

Option <*Tree*>:

- The call of `prog::calltree` returns an object of type `adt::Tree`. No output is printed on screen. The return object contains all call structure informations, that would be printed on screen without this option (see `adt::Tree`).

Example 1. `fib` is a self-calling procedure, `prog::calltree` visualizes the calling structure:

```
>> fib:= proc(n)
        begin
          if n < 2 then
            n
          else
            fib(n - 1) + fib(n - 2)
          end_if
        end_proc:
   prog::calltree(fib(4))
```

```
fib
|
+-- fib
|   |
|   +-- fib
|   |   |
|   |   +-- fib
|   |   |
|   |   `-- fib
|   |
```

```
|   '-- fib
|
'-- fib
    |
    +-- fib
    |
    '-- fib

                3
```

With the option *Args*, the arguments of each function call are printed:

```
>> prog::calltree(fib(3), Args)

                      fib(fib(3))
                      |
                      +-- fib(fib(2))
                      |   |
                      |   +-- fib(fib(1))
                      |   |
                      |   '-- fib(fib(0))
                      |
                      '-- fib(fib(1))

                           2
```

Changes:

- `prog::calltree` is a new function.

`prog::changes` – generate obsolete functions of MuPAD version 1.4

`prog::changes()` invokes a "compatibility mode" between the MuPAD versions 1.4 and 2.0. This call generates "dummy" implementations of system functions that existed in version 1.4 but do not exist in 2.0 anymore. Calling one of these obsolete functions produces a warning. The corresponding function of version 2.0 is called automatically by its new name.

`prog::changes(object)` prints information on changes of the object.

Call(s):

- `prog::changes(<option> <, Quiet>)`
- `prog::changes(object)`

Parameters:

`option` — either *`Warning`* or *`Error`* or *`Remove`*. The default is *`Warning`*.

`object` — any MuPAD object

Options:

`Warning` — Calling an obsolete function produces a warning. If a corresponding function exists in version 2.0, it is called automatically by its new name.

`Error` — Calling an obsolete function produces an error.

`Remove` — Switches the "compatibility mode" off. All functions and domains generated by a previous call to `prog::changes` are removed.

`Quiet` — No messages are printed during the call to `prog::changes`.

Return Value: the void object `null()`.

Related Functions: `help`, `info`, `Pref::warnChanges`

Details:

- Many functions and domains that existed in version 1.4 were given new names, or were moved to other places in the library. `prog::changes()` generates "dummy" versions of all obsolete functions and domains. A subsequent call to such a "dummy" function produces a warning informing the user about the nature of the change. Depending on the option *`Warning`* or *`Error`* used in the call to `prog::changes`, the corresponding function of version 2.0 is called automatically, or an error is produced, respectively. Cf. examples 1 and 2.

- `prog::changes(object)` prints information about changes of the object. Such information is available for functions, function environments, domains and environment variables. Cf. example 3.

Option <*`Warning`*>:

- When an obsolete function is called, a warning is printed. The corresponding function of version 2.0 is called automatically with the same arguments. If no corresponding function exists in version 2.0, a warning is printed and `FAIL` is returned.

- The calls `prog::changes()` and `prog::changes(`*`Warning`*`)` are equivalent.

Option <*Error*>:

- When an obsolete function is called, an error is produced. In particular, no corresponding new function is called. Cf. example 2.

Option <*Remove*>:

- With this option, all obsolete functions and domains generated by a previous call to `prog::changes` are deleted. Also a call to `reset` deletes the objects generated by `prog::changes`.

Option <*Quiet*>:

- This option disables the printing of messages while `prog::changes` is executed. The messages produced by calling the obsolete objects are not influenced.

Example 1. The MuPAD version 1.4 provides the function `asin` for the inverse sine function. This function has become obsolete in the current version. The same holds for the function `unassign` of version 1.4:

```
>> a := asin(1)

                              asin(1)

>> unassign(a)

                         unassign(asin(1))
```

A call to `prog::changes` is useful to find out information *and* to execute code written for version 1.4. The following command makes MuPAD produce "dummy" versions of all missing functions:

```
>> prog::changes()

 Info: Obsolete functions of MuPAD version 1.4 are 'restored'.
       Any call to an obsolete function will produce a warning.
       A function from the present library with the same or a
       similar functionality will be called automatically.
```

The "dummy" functions can be called and produce useful hints. They also forward the arguments to an appropriate new function of version 2.0:

```
>> a := asin(1)
```

```
 Warning: 'asin' was changed to 'arcsin' [asin]

                                PI
                                --
                                2
```

```
>> unassign(a):

 Warning: 'unassign' was removed. Use the new keyword 'delete' \
 [unassign]
```

```
>> a

                                 a
```

To remove all obsolete functions, `prog::changes` can be called with the option *`Remove`*:

```
>> prog::changes(Remove)

 Info: All redefined functions and domains are removed.
```

Example 2. `prog::changes` is called with the option *`Error`*. The "dummy" functions generated by this call produce error messages. The option *`Quiet`* suppresses all messages during the execution of `prog::changes`:

```
>> reset():
   prog::changes(Error, Quiet):

>> a := asin(1)

 Error: 'asin' was changed to 'arcsin' [asin]

>> READ_PATH

 Error: system variable 'READ_PATH' was renamed, please use 'RE\
 ADPATH'
```

Example 3. `prog::changes` provides information about specific objects:

```
>> reset():
   prog::changes(fun)

 Info: 'fun' is removed. [prog::changes]

>> prog::changes(asin)
```

```
 Info: 'asin' is a renamed function.
       'asin' is changed to 'arcsin'. [prog::changes]

>> prog::changes(sharelib::trace)

 Info: 'sharelib::trace' is a renamed function.
       'sharelib::trace' is changed to 'prog::trace'. [prog::changes]
```

Changes:

- `prog::changes` is a new function.

`prog::check` – Checking MuPAD objects

`prog::check` checks MuPAD objects and draws attention to errors and possible problems in programming. `prog::check` is very helpful for finding errors in user-defined domains and procedures.

Call(s):

- `prog::check(object <, infolevel <, options>>)`

Parameters:

`object`	—	procedure, function environment or domain to check, the identifier `All`, or a list of objects
`infolevel`	—	positive integer that determines the completeness of messages
`options`	—	set of options, a single option, or `All`

Options:

`Global`	—	report unknown global identifiers
`Local`	—	report unused local variables
`Localf`	—	report unused local variables *and* unused formal parameters
`Assign`	—	report assignments to formal parameters of procedures
`Level`	—	report applications of `level` to local variables
`Domain`	—	report undefined entries of domains (uses the slot `"undefinedEntries"`)
`Environment`	—	report assignments to environment variables
`Protect`	—	report assignments to (global) protected identifiers
`Special`	—	report special statements
`Escape`	—	report possible pointers to procedure environments

Return Value: `prog::check` returns the void object `null()`. Output messages are printed on the screen.

Related Functions: `debug`, `prog::init`, `prog::isGlobal`, `prog::trace`, `prog::getname`

Details:

- The call `prog::check(object)` checks the MuPAD object `object`. `object` may be a procedure, a function environment, or a domain. One may also give a list of such objects.
- If `All` is given as first parameter, all defined procedures, function environments and domains are checked (see `anames`).
- `infolevel` determines the amount of information given while checking. The following values are useful:
  - **1** number of warnings per checked object, if at least one warning occurs (default)
  - **2** as 1, but a short message is printed even if no warning was produced
  - **3** summary of warnings per checked object
  - **5** displays each checked object, followed by individual warnings, followed by a summary and the number of warnings, if any.
  - **10** ... **15** additional outputs (for debugging)
- `options` can be a set containing one or more of the following options, a single option or `All` (see options).
- With option `All`, all are checked. Without options, the set `{Domain, Global, Level, Local, Protect}` is used.
- The arguments of `hold` expressions are not checked.

Option <*Global*>:

- Unknown (global) identifiers

Option <*Local*>:

- Information about unused local variables are printed. These are variables that were declared by `local`, but never used in the procedure.

Option <`Localf`>:

- The same as `Local`, but the same check is additionally performed for formal parameters of a procedure. Those are the argument names as given in the definition of the procedure.

Option <`Assign`>:

- Information about assignments to formal parameters are printed. Because a formal parameter will be overwritten in MuPAD 2.0, unlike in previous versions, those assignments *could* indicate a programming error (however, not imperative).

Option <`Level`>:

- The application of `level` to local variables is reported. Starting with MuPAD 2.0, local variables are simply replaced by their values on evaluation and calling `level` on them does not have any effect.

Option <`Domain`>:

- Information about undefined required entries of domains are printed.

Option <`Environment`>:

- Information about assignments to environment variables of MuPAD are printed. These assignments could change the global behavior of MuPADif the change is not undone (preferably using `save`, to catch error conditions).

Option <`Protect`>:

- Information about assignments to protected variables of MuPAD are printed.

Option <*Special*>:

- Information about some special cases are printed. Currently, the only implemented special case is assignments to `HISTORY`.

Option <*Escape*>:

- `prog::check` prints warnings about procedures which may require the option `escape`.

Example 1. The following function contains a number of mistakes, some of which were actually legal in previous versions of MuPAD.Lines `1` and `2` contains declarations of local variables. In line `4` an undeclared (global) variable `g` is used. Line `7` applies `level` to a local variable (the call simply returns the value of `X` in MuPAD 2.0). Line `10` contains an assignment to a formal parameter. This parameter will be overwritten and its old value lost:

```
>> f:= proc(X, Y)                 //  1  Local
         local a, b;               //  2  Local
       begin                       //  3
         g:= proc(X)               //  4  Global
               option hold;        //  5
             begin                 //  6
               a:= level(X, 2);    //  7  Level
               a:= a + X           //  8
             end_proc;             //  9
         Y:= g(Y);                 // 10  Assign, Global
       end_proc:
   prog::check(f, 3)

 'level' applied to variables: {X} in [f, proc in 'f']
 Global idents: {g} in [f]
 Unused local  var's:  {b} in [f]
 Warnings: 3 [f]
```

Only search for global variables, but give more messages:

```
>> prog::check(f, 5, Global)

 checking f (DOM_PROC)
 Warning: Global variable 'g' in f []
 Warning: Global variable 'g' in f []
 Global idents: {g} in [f]
 Warnings: 1 [f]
```

Now check everything:

```
>> prog::check(f, 5, All)
```

```
 checking f (DOM_PROC)
 Warning: Global variable 'g' in f []
 Warning: critical usage of 'level' on local variable 'X' in proc in 'f' [f
 'level' applied to variables: {X} in [f, proc in 'f']
 Procedure environment of [f] used by
          [f, proc in 'f']
 Warning: assignment to FORMAL parameter 'Y' in f []
 Warning: Global variable 'g' in f []
 Global idents: {g} in [f]
 Unused local  var's:  {b} in [f]
 Unused formal par's:  {X} in [f]
 Assignments to formal parameters: {Y} in [f]
 Warnings: 5 [f]
```

Further Documentation: From MuPAD 1.4 to MuPAD 2.0

Changes:

- `prog::check` is a new function.
- `prog::check` includes the functionality of the obsolete functions `misc::checkLib` and `misc::checkFunction`.

`prog::exprtree` – visualize an expression as tree

`prog::exprtree(ex)` visualizes any MuPAD expression `ex` as tree.

Call(s):

- `prog::exprtree(ex <, Quiet>)`

Parameters:

`ex` — expression to visualize

Options:

`Quiet` — suppress screen output

Return Value: an object of type `adt::Tree`

Related Functions: `prog::calltree`, `adt::Tree`

Details:

- `prog::exprtree` visualizes the tree structure of any MuPAD expression.
- Every expression in MuPAD is internally a tree. The operations are the nodes, and the operands are the leafs.

Option <*Quiet*>:

- With this option no output will be printed on screen. The return value of type `adt::Tree` represents the tree structure of `ex`.

Example 1. The example shows the structure of the expression `a + b*2 - d*(a + c)`:

```
>> prog::exprtree(a + b*2 - d*(a + c))
```

```
                          _plus
                          |
                          +-- a
                          |
                          +-- _mult
                          |   |
                          |   +-- b
                          |   |
                          |   '-- 2
                          |
                          '-- _mult
                              |
                              +-- d
                              |
                              +-- _plus
                              |   |
                              |   +-- a
                              |   |
                              |   '-- c
                              |
                              '-- -1

                              Tree1
```

`Tree1` is the return value of type `adt::Tree`. This object can be exposed or taken for other operations.

The option *`Quiet`* suppresses the output, only the tree is returned:

```
>> prog::exprtree(a + b*2 - d*(a + c), Quiet)
                                Tree2
```

Changes:

- `prog::exprtree` is a new function.

`prog::error` – error message and internal error number

`prog::error` converts an internal error number into the corresponding message and vice versa.

Call(s):

- `prog::error(number)`
- `prog::error(message)`

Parameters:

`number` — an integer internal error number
`message` — an error message as string

Return Value: a string with the error message or an integer internal error number

Related Functions: `error`, `warning`, `traperror`, `lasterror`

Details:

- `prog::error` converts an internal error number into the corresponding message and vice versa.

Example 1. The corresponding message to the internal error number `1010` is:

```
>> prog::error(1010)
                      "Recursive definition"
```

The error message `"Division by zero"` has the internal number:

```
>> prog::error("Division by zero")
                                1025
```

Changes:

- `prog::error` is a new function.

`prog::find` – find operands of expressions

`prog::find(ex, opr)` returns all "paths" to the operand `opr` in the expression `ex`.

Call(s):

- `prog::find(ex, opr)`

Parameters:

`ex` — any MuPAD expression of type `DOM_EXPR`
`opr` — any MuPAD object

Return Value: a list of numbers that determine the position of the given object inside of the given expression, or a sequence of lists, if the expression contains the object several times

Related Functions: `op`, `subsop`, `prog::exprtree`

Details:

- `prog::find(ex, obj)` returns the position of the object `obj` in the expression `ex` as list. The list represents a "path" to the given object. With this list and the functions `op` and `subsop`, the object can directly be accessed.
- A path to an object is a list that contains integers `i1, ..., in`.

  The meaning is that the object is the `in`-th operand of the `(in - 1)`-st operand etc. of the `i1`-st operand of the expression `ex`.

  Stated differently, `op(ex, [i1, ..., in]) = opr`.
- If the expression contains the object several times, a sequence of lists is returned.

Example 1. The identifier `a` is the first operand of the expression:

```
>> prog::find(a + b + c, a)

                                [1]
```

The number `1` occurs several times:

```
>> prog::find(f(1, 1, 1), 1)
```

```
[1], [2], [3]
```

Example 2. The identifier `a` is the first operand of the second operand of the first operand of the expression:

```
>> prog::find(b*(a - 1) + b*(x - 1), a)
```

```
[1, 2, 1]
```

The result of `prog::find` can be used to access the element with `op` or replace it with `subsop`:

```
>> op(b*(a - 1) + b*(x - 1), [1, 2, 1]);
   subsop(b*(a - 1) + b*(x - 1), [1, 2, 1] = A)
```

```
a
```

```
b (A - 1) + b (x - 1)
```

Background:

- `prog::find` can be used to manipulate complex MuPAD objects with `subsop`.

Changes:

- `prog::find` is a new function.

`prog::getname` – the name of an object

`prog::getname(object)` returns the name of the MuPAD object `object`.

Call(s):

- `prog::getname(object)`

Parameters:

`object` — any MuPAD object

Return Value: the name as string

Related Functions: `op`, `print`, `expr2text`, `text2expr`, `info`

Details:

- `prog::getname` returns the name of any MuPAD object. The return value is a string, irrespective of the type of the input.
- Names can be extracted from procedures, identifiers, function environments, domains and their methods (and strings, of course). If no name can be extracted from an object, the string `"(noname)"` is returned.
- For all other MuPAD objects the result of `expr2text(object)` is returned as name.

Example 1. My own name:

```
>> prog::getname(prog::getname)

                          "prog::getname"
```

The name of a Domain:

```
>> prog::getname(Dom::ExpressionField())

                    "Dom::ExpressionField()"
```

The "name" of an arbitrary MuPAD object:

```
>> prog::getname(1)

                                "1"

>> prog::getname(a + 2*b)

                            "a + 2*b"
```

Changes:

- `prog::getname` is a new function.

`prog::init` – loading objects

`prog::init(object)` initializes the MuPAD object `object`.

Call(s):

- `prog::init(object)`

Parameters:

`object` — MuPAD object to initialize or option *`All`*

Options:

`All` — initialising all MuPAD objects

Return Value: `prog::init` returns the void object `null()`.

Related Functions: `loadproc`, `Pref::verboseRead`, `prog::check`

Details:

- `prog::init` can be used to initialize MuPAD objects.
- Almost all MuPAD objects (domains, procedures etc.) are loaded into memory at their first use. This mechanism saves a lot of memory and time while starting MuPAD. Most of the MuPAD objects are not needed in a given session and would only fill up the system.
- This strategy is transparent with respect to the usage of MuPAD objects. On slower computers, you may notice a delay on the first use of a function or domain.
- Using `Pref::verboseRead`, you can make MuPAD print information on files loaded automatically.

Option <*`All`*>:

- With this option (instead of some MuPAD object), all MuPAD objects will be initialized.

Example 1. Initializing all MuPAD objects takes a lot of time and greatly increases the memory requirements:

```
>> bytes()

                  522304, 815604, 2147483647

>> prog::init(All):
```

Check the memory usage again:

```
>> bytes()

                15990660, 16507016, 2147483647
```

Example 2. Using `Pref::verboseRead`, we obtain information on what is loaded by the system:

```
>> reset():
   Pref::verboseRead(2):
   prog::init(prog::trace)

 loading package 'prog' [lib/]
 reading file lib/PROG/checkini.mu
 reading file lib/PROG/trace.mu
```

Changes:

- `prog::init` is a new function.

`prog::isGlobal` – information about reserved identifiers

`prog::isGlobal(ident)` determines whether the identifier `ident` is used by the system.

Call(s):

- `prog::isGlobal(ident)`

Parameters:

`ident` — identifier to check

Return Value: `prog::isGlobal` return `TRUE`, if the given identifier is used by the system, otherwise `FALSE`.

Related Functions: `prog::check`, `anames`, `type`, `domtype`

Details:

- `prog::isGlobal(ident)` checks if the identifier `ident` is "used by the system". Here, "used by the system" means that `ident` is an environment variable (e.g., `PRETTYPRINT`), a system-wide constant (e.g., `PI` and `undefined`), an option (for some function call, e.g., `All`), or a system function (such as `sin`).

Example 1. Assume you would like to use some identifiers as options for a new function you wrote. In this example, we will check the elements of the list `[All, Beta, Circle, D, eval, First]` for suitability. (Note that `eval` would not be a good choice, even if it was not a system function, because options should start with a capital letter.)

We define a test function which is mapped to the list and returns `FAIL`, if the tested object is not an identifier, `TRUE`, if the identifier is used by the system and `FALSE` otherwise:

```
>> reset():
   LIST:= [All, Beta, Circle, D, eval, First]:
   map(LIST, X -> if domtype(X) <> DOM_IDENT then
                    X = FAIL
                  else
                    X = prog::isGlobal(X)
                  end_if)

 [All = TRUE, Beta = FALSE, Circle = FALSE, D = FAIL,

    eval = FAIL, First = TRUE]
```

The identifiers `All` and `First` can be used as options because they have already been protected by the system (actually, they are already used as options, which makes them a good choice), the identifiers `Beta` and `Circle` are free and one must only take care that they have no value if they will be used as options—they should be protected first. `D` and `eval` have values and cannot be used as options.

Changes:

- `prog::isGlobal` is a new function.

`prog::makeBinLib` – binary version of a library

`prog::makeBinLib` creates a binary version of a function or library.

Call(s):

- `prog::makeBinLib()`
- `prog::makeBinLib(path)`
- `prog::makeBinLib(path, fname)`

Parameters:

`path` — string that determines a path
`fname` — string that determines a file

Return Value: the void object `null()`

Side Effects: `prog::makeBinLib` creates a binary version `file.mb` corresponding to a MuPAD source file `file.mu`.

Details:

- This function ist mostly obsolete with MuPAD 2.0, due to the following changes:

  - The parse process is much faster than in earlier versions.
  - The binary code takes considerably more space than the source generating it.

- Using `prog::makeBinLib` does not stop anyone from reading your MuPAD code, it only causes the parsing step to be omitted when reading the file.
- The binary format is platform independent.

Changes:

- `prog::makeBinLib` used to be `misc::makeBinLib`.

`prog::memuse` – memory usage of a computation

`prog::memuse(stmt)` shows the memory usage for computation and loading library functions while evaluating `stmt`.

Call(s):

- `prog::memuse(stmt)`

Parameters:

`stmt` — a MuPAD statement

Return Value: the result of `stmt`

Related Functions: `Pref::verboseRead`, `prog::trace`, `prog::profile`

Details:

- `prog::memuse(stmt)` shows the memory used while evaluating `stmt`.
- `stmt` is evaluated by `prog::memuse`. If any function or library is loaded, `prog::memuse` prints the increment of memory usage.

  In the end, a summary is printed showing the memory usage in two parts: `loadproc` means the memory used by loaded functions and libraries, `executing` means the memory allocated while computing.
- The result of `prog::memuse` is the result of the evaluation of `stmt`.
- `prog::memuse` works only on Unix-like machines. It uses the temporary file `/tmp/mem.tmp`.

Example 1. The example shows the memory usage of a first call of the function `testtype`: The library `Type` and the object `Type::Unknown` are loaded:

```
>> reset():
   prog::memuse(testtype(x, Type::Unknown))

 'LIBFILES/Type'                  : 16.4 kB
 'TYPE/Unknown'                   : 1.4 kB

 loadproc                         = 17.9 kB
 executing                        = 0.4 kB
 All                              = 17.5 kB

                                TRUE
```

The next example shows the memory usage for creating a large MuPAD object. The result is not shown (suppressed by `:`):

```
>> prog::memuse([random()] $ i = 1..1000):


 loadproc                         = 0.0 kB
 executing                        = 112.3 kB
 All                              = 112.3 kB
```

Background:

- MuPAD does not load all its library functions on startup or after a `reset()`. This saves a lot of time and memory. The library functions are loaded on their first use, which may in some cases cause a noticeable delay on the first invocation.

Changes:

- `prog::memuse` is a new function.

`prog::profile` – display timing data of nested function calls

`prog::profile(stmt)` evaluates the MuPAD statement `stmt` and displays timing data of all nested function calls.

Call(s):

- `prog::profile(stmt)`

Parameters:

`stmt` — a MuPAD statement

Return Value: the result of `stmt`

Related Functions: `prog::calltree`, `prog::trace`

Details:

- The time usage of functions can be measured and displayed with `prog::profile`. For every function called during the evaluation of `stmt`, `prog::profile` prints the time spent in this function and the number of calls.
- `prog::profile` can be helpful in finding time critical functions and unnecessary nested function calls.

Example 1. We define three functions `f`, `g` and `h`. `prog::profile` displays the time spent in each function and the number of calls to it:

```
>> f := proc() local i; begin for i from 1 to 20000 do end_for end_proc:
   g := proc() begin f(), f() end_proc:
   h := proc() begin g(), f(), g() end_proc:
   prog::profile(h()):

 Total time: 300 ms
 ------------------
 f:100.0 % 300 ms total 5 call(s) 0 lookup(s) 60.0 ms/call
 g:  0.0 %   0 ms total 2 call(s) 0 lookup(s) 0.0 ms/call
 h:  0.0 %   0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call

 <h> calls
    f : 1 time(s)
```

```
   g : 2 time(s)

<g> calls
   f : 4 time(s)
```

Background:

- The timings displayed by `prog::profile` are generated by the kernel.
- Evaluation of `stmt` inside `prog::profile` takes substantially longer than evaluating `stmt` directly. This extra time does not influence the validity of the result, i.e., if `prog::profile` reports `f` taking three times as long as `g`, then this is also the case when evaluating `stmt` directly.

Changes:

- No changes.

`prog::test` – automatic comparing of calculation results

`prog::test(stmt, res)` evaluates both MuPAD expressions `stmt` and `res` and compares the results.

Call(s):

- `prog::test(stmt, res <, message>)`
- `prog::test(stmt, TrapError = errnr <, message>)`
- `prog::test(stmt <, message>)`

Parameters:

`statement`	—	a MuPAD statement to test
`res`	—	a MuPAD expression or statement that determines the expected result.
`errnr`	—	a positive integer that determines a MuPAD error
`message`	—	a string that will be displayed as message if the test fails

Options:

`TrapError = errnr`	—	With this option it is possible to check the error behavior of a function. The tested call must produce an error, and the internal error number must be equal to `errnr`. In this case no error message is printed. In all other cases the test fails and an error message is printed.

Return Value: `prog::test` returns the void object `null()`. While executing, messages will be printed on screen or into files.

Related Functions: `prog::error`, `prog::testinit`, `prog::testexit`, `prog::testfunc`, `TESTPATH`

Details:

- If the results of the evaluation of both arguments `stmt` and `res` are different, a message is printed on screen that contains several information.
- `prog::test` works in two different modes: interactive and inside of test files.
- In interactive mode a single call of `prog::test` can be used to compare two MuPAD statements.

  If the evaluation of both first arguments leads to the same MuPAD object, nothing is printed and `prog::test` returns the void object `null()`.

  If the results are different, the test fails and an error message is printed.
- The other mode is using `prog::test` inside of test files.

  A test file must contain at least a starting and an exit statement.

  First the function `prog::testinit` initializes the test file. Next `prog::testfunc` determines the name of the tested function.

  Now several tests can be performed (see next paragraph).

  With `prog::testfunc` another function can be initialized in the same file. The number of the tests is reseted by `prog::testfunc`.

  The last statement in a test file must be `prog::testexit`.
- The tests can be arbitrary MuPAD statements and `prog::test` statements. However, most of the functionality should be executed as argument of `prog::test`. Only initialization of variables should be performed outside of `prog::test` statements in a test file, because:

  `prog::test` traps every error (with the function `traperror`) and prints a specific error message.

  If an error occurs outside of `prog::test`, the reading of the test file is interrupted.

  

  If no error occurs (as should be the default case), the results are compared and a message is printed, if they are different.
- If a test fails, i.e., the two first arguments of `prog::test` lead to different MuPAD objects, a message is printed in the format:

  ```
  Error in test 'interactive' 1:  1 + 2 = 4 [<> 3] (first
  test)
  ```

  `prog::test` emits messages with up to six pieces of information:

1. The name of the test as given by `prog::testfunc` or `interactive`, if `prog::test` is called without a prior call to `prog::testinit`.
2. All tests are numbered (in this case we get number `1` for the first interactive test). The numbering is reset by `prog::testfunc` (and also by `reset`).
3. Then the unevaluated test equation is printed (`1 + 2 = 4`), `1 + 2` is the first argument of `prog::test`,
4. `4` is the second argument.
5. Next, we get the result of the first statement (`3`), which is different from the expected test result `4`.
6. The string `first test` was given by the last optional argument of `prog::test` and can be used to identify a test (see next paragraph).

- If the third argument `string` is given, `string` is evaluated and appended to the end of a message, if the test fails (in the last example `"first test"`). With this message a test inside of a test file can be identified easier.
- If only one argument is given, the argument is evaluated and compared with `TRUE`, i.e., `prog::test(ex)` is equivalent to `prog::test(ex, TRUE)`.
- After the statement `setuserinfo(prog::test, 2)`, additional information for every test is printed on screen.
- When a test is initialized with `prog::testinit` and ended by `prog::testexit`, a short message is printed with the format:

  ```
  Info:  2 tests, 1 error, time:  3.10 / 4.52 s
  ```

  The message contains the number of all tests performed (`2`), the number of errors (`1`), and two times: the first time is the number of seconds for only the evaluations of all first statements, the second time is the time between the call of `prog::testinit` and `prog::testexit`.

Option <*`TrapError`*>:

- The evaluation of `stmt` may lead to a MuPAD error on purpose. In this case the result can be the equation *`TrapError`* `= errnr`, whereby `errnr` is the internal MuPAD error number, which is returned by the function `traperror`.

Example 1. `prog::test` can be called interactively:

```
>> prog::test(1 + 1, 2):
   prog::test(is(2 > 1)):
   prog::test(sin(PI), 0, "check sin"):
```

These tests checked all right, therefore nothing was printed. In the next tests wrong results are tested against, to demonstrate the messages given by `prog::test`:

```
>> prog::test(1 + 2, 2):
   prog::test(is(x > 1)):
   prog::test(sin(PI), PI, "check sin"):

 Error in test 'interactive' 3:  1 + 2 = 2 [<> 3]
 Error in test 'interactive' 4:  is(1 < x) = TRUE [<> UNKNOWN]
 Error in test 'interactive' 5:  sin(PI) = PI [<> 0] (check sin)
```

Example 2. This is a short example for a test file with name `"test.tst"`:

```
 // test file "test.tst"
 test:= (a, b) -> a^2 + 2*b^2 - a*b:
 prog::testinit("test"):
 prog::testfunc(test):
 prog::test(test(1, 4), 29, "my first example"):
 prog::test(test(3, -2), 24, "the second example"):
 prog::test(error("test"), TrapError = 1028):
 prog::testexit():
```

The first statement is only a comment. The second line contains an initialization of a test procedure called `test`. Then the test is initialized with `prog::testinit` and the function name is set with `prog::testfunc`.

After that three tests are performed: The first test is right, the second expected result is wrong, and the third test produces an error, but the expected result is this error, the error number returned by `traperror` is `1028` (user call of `error`).

The whole test takes nearly no time:

```
>> read("test.tst")

 Error in test 'test' 2: test(3, -2) = 24 [<> 23] (the sec-
ond example)
 Info: 2 tests, 1 error,  time:  0.00 / 0.01 s

>> setuserinfo(prog::test, 2):
   read("test.tst")
```

```
 testing 'test' 1
   for test(1, 4) = 29
 testing 'test' 2
   for test(3, -2) = 24
 Error in test 'test' 2: test(3, -2) = 24 [<> 23] (the sec-
ond example)
 Info: 2 tests, 1 error,  time:  0.00 / 0.01 s
```

Changes:

- `prog::test` can be used interactively.
- The expected result can be an error number to check the error behavior of a function.
- If the test fails, the returned result will be printed at end of the message.
- An optional third argument can be given to identify tests.

`prog::testexit` – closing tests

`prog::testexit` closes automatic tests from test files.

Call(s):

- `prog::testexit()`

Return Value: `prog::testexit` returns the void object `null()` and closes the last opened protocol file.

Related Functions: `prog::test`, `prog::testinit`, `prog::testfunc`

Details:

- `prog::testexit` closes automatic tests and prints a short statistic about the test (see `prog::test`).
- `prog::testexit` closes the last opened protocol file.
- `prog::testexit` must be called before beginning of a new test with `prog::testinit`.

Changes:

- `prog::testexit` prints a short statistic about the test.

`prog::testfunc` – initialize tests

`prog::testfunc` initializes automatic tests from test files.

Call(s):

- `prog::testfunc(func)`

Parameters:

`func` — any MuPAD function to test

Return Value: `prog::testfunc` returns the void object `null()`.

Related Functions: `prog::test`, `prog::testinit`, `prog::testexit`

Details:

- The function `prog::testfunc` initializes automatic tests.
- The name of `func` will be used for printed messages (see `prog::test`).

Changes:

- No changes.

`prog::testinit` – initialize tests

`prog::testinit` initializes automatic tests from test files.

Call(s):

- `prog::testinit(string)`

Parameters:

`string` — string – the name of the protocol file

Return Value: `prog::testinit` returns the void object `null()` and opens the protocol file with name "`string`.res".

Related Functions: `prog::test`, `prog::testfunc`, `prog::testexit`, TESTPATH

Details:

- The function `prog::testinit` initializes automatic tests (see `prog::test`).
- `prog::testinit` opens the protocol file named *"`string`.res"* (see TESTPATH) and resets the test number.

Changes:

- No changes.

`prog::trace` – observe functions

`prog::trace(obj)` manipulates the MuPAD function `obj` to observe entering and leaving of this function.

Call(s):

- `prog::trace(obj <, option>)`
- `prog::trace(obj_set <, option>)`
- `prog::trace(dom, meth_set <, option>)`

Parameters:

`obj`	—	a MuPAD function, domain or domain method, or function environment to observe
`obj_set`	—	a set of MuPAD functions to observe
`dom`	—	a MuPAD domain to observe
`meth_set`	—	a set of methods of the domain `dom` to observe, given by their name as strings
`option`	—	one of the described options or a set with one or several options

Options:

Backup	— The option *Backup* can be used to trace a function, that is already traced. The function will be restored (by `prog::untrace`) and then traced again. This option can be used, if a function should be traced with another options.
Depth = `level`	— `level` is a positive integer. If this option is given, nested function calls will only be displayed, if the recursion depth is less or equal to `level`.
Force	— The option *Force* can be used to trace a function, that is already traced. The function will traced again and the backup of the original function will be overwritten. This option can be useful, if a function is newly defined and `prog::trace` refuse a newly trace. The option *Backup* would replace the new definition by the old backup (see example 2).
Mem	— The option *Mem* can be used to show the change of memory usage between entering and leaving of an observed function.
NoArgs	— With a view to greater clarity the option *NoArgs* hides the arguments when entering and leaving of an observed function.
Plain	— With the option *Plain* the messages are aligned left and *not* indented to illustrate the dependencies between function calls.

Return Value: `prog::trace` returns the void object `null()`.

Related Functions: `prog::untrace`, `prog::traced`, `setuserinfo`, `debug`, `prog::profile`, `prog::calltree`

Details:

- `prog::trace` is a very helpful function to observe functions to debug or for information. While tracing functions, the inter-relations between calls to these traced functions can be viewed.

- `prog::trace` will be called with the function `obj` to observe. After that, every call of this function `obj` prints a message, when the function is entered and leaved. The arguments and the return value of the function call will also be printed.

  Every message is indented when entering a function and released when leaving the function (to illustrate the dependencies between function calls). The option *Plain* can be used to suppress this behaviour.

- If a set of functions is given, all functions in this set are traced.

- `obj` can be a domain or a function environment, too. Then all methods of the domain or the function environment will be observed.

  If the second argument `meth_set` is a set of method names of the domain or function environment `dom`, then all given methods are traced.

  The method `meth` of the domain or function environment `dom` can also be traced directly by `prog::trace(dom::meth)`.

- The function `prog::untrace` terminates the observation of a function. The function `prog::traced` detects, whether the function is traced.

Example 1. Define a short function, that calls itself recursively, and the calls are observed:

```
>> fib:= proc(n)
        begin
          if n < 2 then
            n
          else
            fib(n - 1) + fib(n - 2)
          end_if
        end_proc:
   prog::trace(fib):
   fib(3)

 enter 'fib'                              with args   : 3
   enter 'fib'                              with args   : 2
     enter 'fib'                              with args   : 1
     leave 'fib'                              with result : 1
     enter 'fib'                              with args   : 0
     leave 'fib'                              with result : 0
   leave 'fib'                              with result : 1
   enter 'fib'                              with args   : 1
   leave 'fib'                              with result : 1
 leave 'fib'                              with result : 2

                                2
```

First restore the function, and then use the option *`Plain`*:

```
>> prog::untrace(fib):
   prog::trace(fib, Plain):
   fib(3)

 enter 'fib'                              with args   : 3
 enter 'fib'                              with args   : 2
 enter 'fib'                              with args   : 1
 leave 'fib'                              with result : 1
```

```
 enter 'fib'                              with args   : 0
 leave 'fib'                              with result : 0
 leave 'fib'                              with result : 1
 enter 'fib'                              with args   : 1
 leave 'fib'                              with result : 1
 leave 'fib'                              with result : 2

                                  2
```

The option *Depth* limits the displaying, *Backup* restores the original code of `fib` before tracing with new options:

```
>> prog::trace(fib, {Depth = 2, Backup}):
   fib(12)

 Warning: backup of object 'fib' will be traced [prog::trace]
 enter 'fib'                              with args   : 12
   enter 'fib'                              with args   : 11
   leave 'fib'                              with result : 89
   enter 'fib'                              with args   : 10
   leave 'fib'                              with result : 55
 leave 'fib'                              with result : 144

                                 144
```

Example 2. Define a short function `f` and observe this function:

```
>> f := x -> if x > 0 then x else -f(-x) end:
   prog::trace(f):
   f(-2)

 enter 'f'                                with args   : -2
   enter 'f'                                with args   : 2
   leave 'f'                                with result : 2
 leave 'f'                                with result : -2

                                 -2
```

Now the function is slightly changed and reassigned to `f`. But the trace mechanism does not know the change of the function `f` and denies the newly observation:

```
>> f := x -> if x > 0 then x else f(-x) end:
   prog::trace(f):

 Warning: object 'f' is already traced [prog::trace]
```

In this situation the option *Force* can be used to force the tracing. The warning means, a possibly existing backup of the function `f` is overwritten:

```
>> prog::trace(f, Force):
   f(-2)

 Warning: backup of object 'f' will be replaced [prog::trace]
 enter 'f'                                with args   : -2
   enter 'f'                                with args   : 2
   leave 'f'                                with result : 2
 leave 'f'                                with result : 2

                                  2
```

Inattentive usage of option *Force* has following results (the function call prints out multiple messages):

```
>> prog::trace(f, Force):
   f(-2)

 Warning: backup of object 'f' will be replaced [prog::trace]
 enter 'f'                                with args   : -2
   enter 'f'                                with args   : -2
     enter 'f'                                with args   : 2
       enter 'f'                                with args   : 2
       leave 'f'                                with result : 2
     leave 'f'                                with result : 2
   leave 'f'                                with result : 2
 leave 'f'                                with result : 2

                                  2
```

Example 3. With the option *Mem* the memory usage is printed:

```
>> prog::trace(sin, Mem):
   sin(x)

 enter 'sin'                              with args   : x
 leave 'sin'                              with result [1327 kB]: sin(x)

                                sin(x)
```

The function `sin` takes such a lot of memory...? This happens, when this function call is the first in the session or after a `reset`, because a lot of libraries are loaded, e.g., `property` to preserve properties of identifiers in `sin`.

Background:

- When calling `prog::trace` with a function `obj` as argument, the function `obj` will be manipulated.

  At beginning of the function a statement will inserted to print a message and the function arguments when entering the function.

  At every location that the function could be leaved a statement will be placed to print a message and the return value of the function.

- The function `prog::untrace` rebuilds the original state of the function. Therefore the function will be saved in an internal table.

Changes:

- `prog::trace` used to be `sharelib::trace`.
- all options are new implemented

`prog::traced` – find traced functions

`prog::traced()` lists all traced functions.

Call(s):

- `prog::traced(<obj>)`

Parameters:

`obj` — a MuPAD function, a function environment or a library

Return Value: `prog::traced` returns the void object `null()`.

Related Functions: `prog::trace`, `prog::untrace`

Details:

- `prog::traced(obj)` detects, whether the function `obj` is traced. If `obj` is a library or a function enviroment, then all methods will be checked. If no argument is given, all traced functions will be displayed.
- If a function is traced, a copy of the original function is saved, and the function is manipulated to display additional information during evaluation.
- `prog::traced` determines whether a copy exists and whether the function has been manipulated the way `prog::trace` does.

- There are two messages that occur:

  `A backup of object 'obj' exists` means, a backup of the original object `obj` exists to restore `obj` with `prog::untrace`.

  `Object 'obj' seems to be traced` says, the object `obj` was analyzed and some points suggests this matter.

Example 1. The function `sin` is traced:

```
>> prog::trace(sin):
   prog::traced(sin)

 A backup of object 'sin' exists.
 Object 'sin' seems to be traced.
```

Changes:

- `prog::traced` is a new function.

`prog::untrace` – terminates observation of functions

`prog::untrace(obj)` undoes the effect of `prog::trace(obj)` and restores the original definition of `obj`.

Call(s):

- `prog::untrace(obj)`
- `prog::untrace()`

Parameters:

`obj` — the MuPAD function that is observed, or a domain or a function environment

Return Value: `prog::untrace` returns the void object `null()`.

Related Functions: `prog::trace`, `setuserinfo`, `debug`, `prog::profile`, `prog::calltree`

Details:

- `prog::untrace(obj)` terminates the observation of the MuPAD function `obj` performed by `prog::trace`.
- `obj` can be a domain or a function environment, too. Then all methods of the domain or function environment will be restored.
- If no argument is given, all observed objects will be restored from observation.

Example 1. The observation of a function will be terminated:

```
>> prog::untrace(sin):

 Error: function was not traced [prog::untrace]

>> prog::trace(sin):
   sin(2)

 enter 'sin'                              with args   : 2
 leave 'sin'                              with result : sin(2)

                               sin(2)

>> prog::untrace(sin):
   sin(2)

                               sin(2)
```

Changes:

- `prog::untrace` used to be `sharelib::untrace`.