`misc` — miscellanea

Table of contents

Introduction

The `misc` library contains some miscellaneous utility functions.

The package functions are called using the package name `misc` and the name of the function. E.g., use

```
>> myplus := misc::genassop(_plus, 0)
```

to create an own n-ary plus operator. (This is not really useful, since `_plus` is an n-ary operator anyway.) This mechanism avoids naming conflicts with other library functions. If this is found to be inconvenient, then the routines of the `misc` package may be exported via `export`. E.g., after calling

```
>> export(misc, genassop)
```

the function `combinat::bell` may be called directly:

```
>> myplus := genassop(_plus, 0)
```

All routines of the `misc` package are exported simultaneously by

```
>> export(misc)
```

The functions available in the `misc` library can be listened with:

```
>> info(misc)
```

We would especially like to thank Raymond Manzoni for contributing the function `misc::pslq` .

misc::breakmap – stops the mapping currently done by maprec

misc::breakmap() stops the recursive application of a function to all subexpressions of an expression that misc::maprec is just working on.

Call(s):

- misc::breakmap()

Return Value: misc::breakmap always returns TRUE.

Related Functions: misc::maprec

Details:

- misc::breakmap is useful as a command inside the procedure mapped by misc::maprec in case we know that we are finished with our work and the remaining recursive mapping is not necessary.

Example 1. We want to know whether a given expression contains a particular type t. As soon as we have found the first occurence of t, we can terminate our search.

```
>> myfound := FALSE:
   misc::maprec(hold(((23+5.0)/3+4*I)*PI), {DOM_COMPLEX}=proc() be-
gin \
                 myfound := misc::breakmap(); args() end_proc) :
   myfound; delete myfound :

                                TRUE
```

What did we do? We told misc::maprec just to go down the expression tree and look for subexpressions of type DOM_COMPLEX; and, whenever such subexpression should be found, to apply a certain procedure to it. That procedure stops the recursive mapping, remembers that we have found the type we had searched for, and returns exactly its argument such that the result returned by misc::maprec equals the input. In the example below, we test whether our given expression contains the type DOM_POLY.

```
>> myfound := FALSE:
   misc::maprec(hold(((23+5.0)/3+4*I)*PI), {DOM_POLY}=proc() be-
gin \
                 myfound := misc::breakmap() ; args() end_proc) :
   myfound; delete myfound :

                                FALSE
```

Note that you do not need to use this method when searching for subexpressions of a given type; calling `hastype` is certainly more convenient.

Changes:

- `misc::breakmap` used to be `breakmap`.

`misc::genassop` – generates an n-ary associative operator from a binary one

`misc::genassop(binaryop, zeroelement)` generates an n-ary associative operator from the binary operator `binaryop`, where `zeroelement` is a neutral element for `binaryop`.

Call(s):

- `misc::genassop(binaryop, zeroelement)`

Parameters:

`binaryop` — a function
`zeroelement` — an object

Return Value: `misc::genassop` returns a procedure `f`. That procedure accepts an arbitrary number of arguments of the same kind `binaryop` does; it returns `zeroelement` if it is called without argument, and its only argument if it is called with one argument; its value on n arguments is inductively defined by `f(x1, ..., xn)=f(binaryop(x1,x2), x3,...,xn)`.

Related Functions: `operator`

Details:

- `binaryop` must be a function taking two arguments (no matter of what kind) and returning a valid argument to itself. It must satisfy the associative law `binaryop`(`binaryop`(a, b), c) = `binaryop`(a, `binaryop`(b, c)).
- `zeroelement` is an object such that `binaryop(a, zeroelement) = a` holds for every `a`.
- `misc::genassop` returns a procedure which returns `zeroelement` if it is called without arguments and the argument if it is called with one argument.
- `misc::genassop` doesn't check whether `binaryop` is really associative and whether `zeroelement` is really a neutral element for `binaryop`.

Example 1. We know that `_plus` is an n-ary operator anyway, but let us assume that `_plus` was only a binary operator. We can create an own n-ary addition as follows:

```
>> myplus := misc::genassop(_plus, 0)

                    proc genericAssop() ... end
```

Now we make `myplus` add some values.

```
>> myplus(3, 4, 8), myplus(-5), myplus()

                                15, -5, 0
```

As mentioned in the "Details" section, `myplus` returns the argument if is called with exactly one argument, and it returns the `zeroelement` `0` if it is called without arguments.

Changes:

- The number of arguments has decreased from three to two.

`misc::maprec` – map a function to subexpressions of an expression

`misc::maprec(ex, selector=funci)` maps the function `funci` to all subexpressions of the expression `ex` that satisfy a given criterion (defined by `selector`) and replaces each selected subexpression `s` by `funci(s)`.

Several different functions may be mapped to subexpressions satisfying different selection criteria.

Call(s):

- `misc::maprec(ex, selector=funci)`
- `misc::maprec(ex, selector=funci, PreMap)`
- `misc::maprec(ex, selector=funci, PostMap)`
- `misc::maprec(ex, selector=funci, PreMap, PostMap)`
- `misc::maprec(ex <, selector=funci, ... > <, PreMap> <, PostMap>)`

Parameters:

`ex`	—	any MuPAD object
`selector`	—	any MuPAD object
`funci`	—	any MuPAD object

Options:

- `PreMap` — For each subexpressions `s` of `ex`, the selector is applied to it *after* visiting all of its subexpressions; `s` may have changed at that time due to substitutions in the subexpressions.
- `PostMap` — For each subexpressions `s` of `ex`, the selector is applied to it *before* visiting its subexpressions. If `s` is selected by `selector`, it is replaced by `funci(s)`, and `misc::maprec` is *not* recursively applied to the operands of `funci(s)`; otherwise, `misc::maprec` is recursively applied to the operands of `s`.

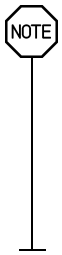
Return Value: `misc::maprec` may return any MuPAD object.

Related Functions: `map`, `mapcoeffs`, `misc::breakmap`

Details:

- `misc::maprec(ex, selector_1=funci_1, ..., selector_n=funci_n)` does two steps: it tests whether `ex` meets a selection criterion defined by some selector `selector_k` (and, if yes, replaces `ex` by `funci_k(ex)`); and it applies itself recursively to all operands of `ex`. The order of these steps is determined by the options `PreMap` and `PostMap`.
- Selectors are applied from left to right; if the expression meets some selection criterion, no further selectors are tried.
- `selector` can have two forms. It can be a set $\{t_1, \ldots, t_n\}$. Here a subexpression `s` of `ex` is selected if type(s1) is one of the types $t_1, \ldots, t_n$. If it is not a set, a subexpression `s` of `ex` is selected if `p`(s) returns `TRUE`. As every MuPAD object may be applied as a function to `s`, `p` may be of any type in the latter case.
- In order not to select a subexpression, the selector need not return `FALSE`; it suffices that it does not return `TRUE`.
- The options `PreMap` and `PostMap` can also be given together; in this case, operands of not selected expressions are visited for a second time.
- If neither the option `PreMap` nor the option `PostMap` is given, then `PreMap` is used.
- Use a `misc::breakmap` command inside `funci` in order to stop the recursive mapping. See the help page of `misc::breakmap` for an example.

- Only subexpressions of domain type `DOM_ARRAY`, `DOM_EXPR`, `DOM_LIST`, `DOM_SET`, and `DOM_TABLE` are mapped recursively. To subexpressions of other types, `selector` is applied, but `misc::maprec` is not mapped to their operands. (This is to avoid unwanted substitutions.) If you want to recurse on them, use a `selector` that selects such subexpressions, and make `funci` initiate another recursive mapping.



- `misc::maprec` is overloadable. If the domain of a subexpression has the method `"maprec"`, then this method is called with the subexpression and the other arguments of the call.

  The subexpression is replaced by the result, but `misc::maprec` is not mapped to its operands; such recursive mapping must be done by the domain method if desired.

- The operators of expressions (`op(expression, 0)`) are also mapped recursively like all the other operands.

Overloadable by: `ex`

Example 1. In the following example every integer of the given expression `a+3+4` is substituted by the value 10. Since `10(n)` returns `10` for every integer `n`, it suffices to write `10` instead of `n -> 10` here.

```
>> misc::maprec(hold(a+3+4), {DOM_INT} = 10)

                                a + 20
```

In the example above, we used `hold` to suppress the evaluation of the expression because otherwise `a+3+4` is evaluated to `a+7` and we get the result:

```
>> misc::maprec(a+3+4, {DOM_INT} = 10)

                                a + 10
```

Example 2. Now we demonstrate the usage of the options `PreMap` and `PostMap`.

```
>> misc::maprec(hold(3+4), {DOM_INT} = 10)

                                  10
```

Here `misc::maprec` was used without an option, this means the default option *`PreMap`* was used. So why did we get a result of 10 instead of the (possibly expected) 20? Because pre-mapping is used by default, `misc::maprec` first applies itself to the integers 3 and 4. They are replaced by the value 10 each such that we get an intermediate result of 20. After that, `misc::maprec` tests the selection criterion for the expression as a whole. This one equals the integer 20 by now, hence it is replaced by 10. Instead, when using the *`PostMap`* option we get:

```
>> misc::maprec(hold(3+4), {DOM_INT} = 10, PostMap )

                                  20
```

Here, the expression `3+4` was tested at first — it is not an integer. Then, `misc::maprec` was applied to the operands, and both were replaced by 10.

Example 3. Now we give an example where the `selector` is a function. We want to eleminate all the prime numbers from an expression.

```
>> misc::maprec(hold(_plus)( i $ i=1..20), isprime= null(), PostMap)

                                  133
```

Here `isprime` returns `TRUE` for every prime number between 1 and 20. Every prime number between 1 and 20 is replaced by null() (since `null()(p)` gives `null()`) which means the call above computes $\sum_{\substack{i=1 \\ i\,composite}}^{20} i$.

Changes:

- `misc::maprec` used to be `maprec`.

`misc::pslq` – heuristic detection of relations between real numbers

`misc::pslq(numberlist, precision)` returns a list of integers `[k1, ..., kn]` such that — denoting the elements of `numberlist` by `a1, ..., an` — the absolute value of $\sum_{i=0}^{n} a_i k_i$ is less than $10^{-precision}$, or `FAIL` if such integers could not be detected.

Call(s):

- `misc::pslq(numberlist, precision)`

Parameters:

`numberlist` — list of real numbers or objects that can be converted to real numbers by the function `float`.

`precision` — positive integer

Return Value: list of integers, or `FAIL`

Side Effects: `misc::pslq` is *not* affected by the current value of `DIGITS`. Numerical computations are carried out with more significant digits such that the output meets the specification given above.

Details:

- This method can be used to get an idea about linear dependencies, before proving them.

Example 1. Does π satisfy a polynomial equation of degree at most 2 ?

```
>> misc::pslq([1, PI, PI^2], 20)

                                FAIL
```

Example 2. Having forgotten the relation between sine and cosine, we can try the heuristic way.

```
>> misc::pslq([1, sin(0.32), sin(0.32)^2, cos(0.32), cos(0.32)^2], 10)

                          +-                  -+
                          | 1, 0, -1, 0, -1 |
                          +-                  -+
```

Background:

- This function has been written by Raymond Manzoni.
- The algorithm has been taken from Bailey and Plouffe, *Recognizing numerical constants*. See also Helaman R.P. Ferguson and David Bailey, *A Polynomial Time, Numerically Stable Integer Relation Algorithm*, RNR Technical Report RNR-92-032.