`listlib` — library for list manipulation

Table of contents

Library `listlib`

This library contains procedures to work with lists.

Related functions of the standard library are `_concat`, `append`, `contains`, `length`, `map`, `nops`, `op`, `poly2list`, `revert`, `sort`, `split`, `subsop` and `text2list`.

`listlib::insert` – insert an element into a list

`listlib::insert(list, element)` inserts `element` into `list`.

Call(s):

- `listlib::insert(list, element <, function>)`

Parameters:

`list`	—	MuPAD list
`element`	—	MuPAD expression to insert
`function`	—	function, that determines the insert position

Return Value: the given list enlarged with the inserted element

Related Functions: `_concat`, `append`, `listlib::insertAt`

Details:

- With the function `listlib::insert` any element can be inserted into any list.
- With the third optional argument a function can be given that compare the elements of the list with the element to insert and therewith determines the position, the element is inserted. The given function is called with two elements and have to return `TRUE`, if the two elements are in the right order, otherwise `FALSE` (see next paragraph).
- The given function is called step by step with an element of the list as first argument and the given element as second argument, until it returns `FALSE`. Then the given element is inserted into the list in *front* of the last proved element (see example 2).
- The list must be ordered with regard to the order function, otherwise the element could be inserted at the wrong place.
- If no third argument is given the function `_less` is used. If no order of the elements with regard to `_less` is defined, a function must be given, otherwise an error appears. The system function `sysorder` always can be used.

Example 1. Insert `3` into the given ordered list:

```
>> listlib::insert([1, 2, 4, 5, 6], 3)

                                     [1, 2, 3, 4, 5, 6]
```

Insert `3` into the given descending ordered list. The insert function represents and preserves the order of the list:

```
>> listlib::insert([6, 5, 4, 2, 1], 3, _not@_less)

                                     [6, 5, 4, 3, 2, 1]
```

Because identifiers cannot be ordered by `_less`, another function must be given, e.g., the function that represents the systems internal order:

```
>> listlib::insert([a, b, d, e, f], c, sysorder)

                                     [a, b, c, d, e, f]
```

Example 2. Because no function is given as third argument, the function `_less` is used. `_less` is called: `_less(1, 3)`, `_less(2, 3)`, `_less(4, 3)` and then `3` is inserted in front of `4`:

```
>> listlib::insert([1, 2, 4], 3)

                                        [1, 2, 3, 4]
```

If the list is not ordered right, then the insert position could be wrong:

```
>> listlib::insert([4, 1, 2], 3)

                                        [3, 4, 1, 2]
```

Example 3. The following example shows, how expressions can be ordered by a user defined priority. This order is given by the function named `priority`, which returns a smaller number, when the expression has a type with higher priority:

```
>> priority := X -> contains(["_power", "_mult", "_plus"], type(X)):
   priority(x^2), priority(x + 2)

                                            1, 3
```

The function `sortfunc` returns `TRUE`, if the both given arguments are in the right order, i.e., the first argument has a higher (or equal) priority than the second argument:

```
>> sortfunc := (X, Y) -> bool(priority(Y) > priority(X)):
   sortfunc(x^2, x + 2), sortfunc(x + 2, x*2)

                                   TRUE, FALSE
```

Now the expression `x*2` is inserted at the "right" place in the list:

```
>> listlib::insert([x^y, x^2, x*y, -y, x + y], x*2, sortfunc)

                        y    2
                      [x , x , 2 x, x y, -y, x + y]
```

Changes:

- `listlib::insert` used to be `insert_ordered`.
- The first both arguments are swapped.

`listlib::insertAt` – insert an element into a list

`listlib::insertAt(list, element, pos)` inserts `element` into `list` at position `pos`.

Call(s):

- `listlib::insertAt(list, element <, pos>)`

Parameters:

`list` — a list
`element` — any MuPAD object
`pos` — any integer

Return Value: the given list enlarged with the inserted element

Related Functions: `listlib::insert`, `append`, `_concat`

Details:

- With the function `listlib::insertAt` any element can be inserted into any list at a specified place.
- The third argument (the "insert index") determines the place to insert the element into the given list.

- If the insert index is less than `1` the element is inserted in front of the list. If the insertion index is greater than `nops(list)` the element is appended to the list. To append an element to a list the kernel function `append` is faster.
- If no third argument is given, the given element is inserted in front of the list.
- If the argument `element` is a list too, the elements of this list will be inserted (or appended) instead of the whole list by preserving the order.

Example 1. Insertion `2` at the third place of the given list:

```
>> listlib::insertAt([1, 1, 1], 2, 3)

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

Insertion of an element in front of a list. The third argument is optional in this case:

```
>> listlib::insertAt([1, 1, 3, 1], 2, 0), listlib::insertAt([1, 1, 3, 1], 2

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

Appending of an element. This could be done faster with `append`:

```
>> listlib::insertAt([1, 2, 3], 4, 1000), append([1, 2, 3], 4)

                [1, 2, 3, 4], [1, 2, 3, 4]
```

Changes:

- `listlib::insertAt` used to be `linsert`.

`listlib::merge` – merging two ordered lists

`listlib::merge(list1, list2)` merges both lists into one list.

Call(s):

- `listlib::merge(list1, list2 <, function>)`

Parameters:

`list1, list2` — a MuPAD list
`function` — a function, that determines the merging order

Return Value: an ordered list that contains the elements of both lists

Related Functions: `listlib::singleMerge`, `listlib::insert`, `_concat`, `zip`

Details:

- With the third optional argument a function can be given that compare the elements of the lists and therewith determines the order of the elements. The given function is called with two elements and have to return `TRUE`, if the two elements are in the right order, otherwise `FALSE` (see next paragraph).
- The given function is called step by step with an element of the first list as first argument and an element of the second list as second argument, until it returns `FALSE`. Then the element of the second list is inserted into the first list in *front* of the last proved element (see example 2).
- The lists must be ordered with regard to the order function, otherwise the elements could be inserted at the wrong place.
- If no third argument is given the function `_less` is used. If no order of the elements with regard to `_less` is defined, a function must be given, otherwise an error appears. The system function `sysorder` always can be used.

Example 1. Merging two ascending ordered lists:

```
>> listlib::merge([1, 3, 5, 7], [2, 4, 6, 8])

                    [1, 2, 3, 4, 5, 6, 7, 8]
```

Merging two descending ordered lists:

```
>> listlib::merge([7, 5, 3, 1], [8, 6, 4, 2], _not@_less)

                    [8, 7, 6, 5, 4, 3, 2, 1]
```

Example 2. The following example shows, how expressions can be ordered by a user defined priority. This order is given by the function named `priority`, which returns a smaller number, when the expression has a type with higher priority:

```
>> priority := X -> contains(["_power", "_mult", "_plus"], type(X)):
   priority(x^2), priority(x + 2)

                              1, 3
```

The function `sortfunc` returns `TRUE`, if the both given arguments are in the right order, i.e., the first argument has a higher (or equal) priority than the second argument:

```
>> sortfunc := (X, Y) -> bool(priority(Y) > priority(X)):
   sortfunc(x^2, x + 2), sortfunc(x + 2, x*2)

                          TRUE, FALSE
```

Now the both lists are merged with regard to the given priority:

```
>> listlib::merge([x^y, x*2, -y], [x^2, x*y, x + y], sortfunc)

                   y   2
                 [x , x , 2 x, -y, x y, x + y]

>> delete priority, sortfunc:
```

Changes:

- `listlib::merge` used to be `listtools::merge`.

`listlib::removeDuplicates` – removes duplicate entries

`listlib::removeDuplicates(list)` removes all duplicate entries of the list `list`.

Call(s):

- `listlib::removeDuplicates(list)`
- `listlib::removeDuplicates(list, KeepOrder)`

Parameters:

`list` — a MuPAD list

Options:

`KeepOrder` — `listlib::removeDuplicates(list, `*`KeepOrder`*`)` returns a list with unique entries in the order of their occurence in `list`.

Return Value: a list that contains each entry only once

Related Functions: `listlib::removeDupSorted`, `DOM_LIST`

Details:

- `listlib::removeDuplicates(list)` removes all duplicates of each entry of the list `list`. The new list is build up from right to left with the order of the *last* occurence of each entry in `list`. Cf. Example 1.
- A faster possibibliy to remove duplicate entries is to convert the list into a `set` and back into a list. You will loose the order of the list entries in this case. Cf. Example 3.

Option <*`KeepOrder`*>:

- `listlib::removeDuplicates(list, `*`KeepOrder`*`)` returns a list of the entries of `list` in the order of their *first* occurence. The list is build up from left to right. Cf. Example 2.

Example 1. Per default `listlib::removeDuplicates` removes duplicate entries in reverse order:

```
>> list:= [1, 1, 1, 3, 1, 5, 5, 1, 3, 3, 1, 7]:
   listlib::removeDuplicates(list)

                              [5, 3, 1, 7]
```

Example 2. With option *`KeepOrder`* entries are selected in the order of their occurence:

```
>> list:= [1, 1, 1, 3, 1, 5, 5, 1, 3, 3, 1, 7]:
   listlib::removeDuplicates(list, KeepOrder)

                              [1, 3, 5, 7]
```

Example 3. If you don't need the order of list entries any more, you may convert the list into a set and back into a list:

```
>> list:= [1, 1, 1, 3, 1, 5, 5, 1, 3, 3, 1, 7]:
   [op({op(list)})]

                              [7, 5, 3, 1]
```

Changes:

- `listlib::removeDuplicates` used to be `listtools::removeDuplicates`.

`listlib::removeDupSorted` – remove duplicates of any element from ordered lists

`listlib::removeDupSorted(list)` removes all duplicates of any element of the ordered list `list`.

Call(s):

- `listlib::removeDupSorted(list)`

Parameters:

`list` — an ordered MuPAD list

Return Value: a list that contains every element only once

Related Functions: `listlib::removeDuplicates`

Details:

- `listlib::removeDupSorted` removes all duplicates of every element of an ordered list.
- `listlib::removeDupSorted` does the same as `listlib::removeDuplicates`, but it assumes that the list is sorted and therefor it is faster. A notable gain will only occur, if there are only few duplicates in a long list.

Example 1. `listlib::removeDupSorted` removes all duplicates from the given list:

```
>> listlib::removeDupSorted([1, 1, 1, 3, 5, 5, 5, 5, 5, 5, 5, 7, 7, 7])

                        [1, 3, 5, 7]
```

If the list is not ordered, `listlib::removeDupSorted` fails:

```
>> listlib::removeDupSorted([1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7])

                [1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
```

Changes:

- `listlib::removeDupSorted` used to be `listtools::removeDupSorted`.

`listlib::setDifference` – remove elements from a list

`listlib::setDifference(list1, list2)` removes all elements from `list1`, that are given by `list2`.

Call(s):

- `listlib::setDifference(list1, list2)`

Parameters:

`list1, list2` — a MuPAD list

Return Value: the first list without all elements of the second list

Related Functions: `_minus`

Details:

- `listlib::setDifference` removes all elements of the list `list1` given by the second list `list2`.
- The order of the list is not preserved. NOTE

Example 1. `listlib::setDifference` removes 2, 4, 6 and 8 from the given list:

```
>> listlib::setDifference([1, 2, 3, 4, 5, 6, 7, 8], [2, 4, 6, 8])

                         [7, 5, 3, 1]
```

Changes:

- `listlib::setDifference` used to be `listtools::setDifference`.

`listlib::singleMerge` – merging of two ordered lists without duplicates

`listlib::singleMerge(list1, list2)` merges two ordered lists without duplicates.

Call(s):

- `listlib::singleMerge(list1, list2 <, function>)`

Parameters:

`list1, list2`	— a MuPAD list
`function`	— a function, that determines the merging order

Return Value: an ordered list that contains the elements of both lists

Related Functions: `listlib::merge`, `listlib::insert`, `_concat`, `zip`

Details:

- `listlib::singleMerge(list1, list2)` merges the both lists into one list. It is assumed that the lists are "disjunct", no element appears in both lists. Otherwise such elements are inserted only once in the result list.
- With the third optional argument a function can be given that compare the elements of the lists and therewith determines the order of the elements. The given function is called with two elements and have to return `TRUE`, if the two elements are in the right order, otherwise `FALSE` (see next paragraph).
- The given function is called step by step with an element of the first list as first argument and an element of the second list as second argument, until it returns `FALSE`. Then the element of the second list is inserted into the first list in *front* of the last proved element (see example 3).
- The lists must be ordered with regard to the order function, otherwise the elements could be inserted at the wrong place.
- If no third argument is given the function `_less` is used. If no order of the elements with regard to `_less` is defined, a function must be given, otherwise an error appears. The system function `sysorder` always can be used.

Example 1. Merging two ascending ordered lists:

```
>> listlib::singleMerge([1, 3, 5, 7], [2, 4, 6, 8])

                    [1, 2, 3, 4, 5, 6, 7, 8]
```

Merging two descending ordered lists:

```
>> listlib::singleMerge([7, 5, 3, 1], [8, 6, 4, 2], _not@_less)

                    [8, 7, 6, 5, 4, 3, 2, 1]
```

Example 2. Merging two ascending ordered lists with duplicates:

```
>> listlib::singleMerge([1, 2, 5, 7], [2, 5, 6, 8])

                      [1, 2, 5, 6, 7, 8]
```

But the following lists does not contain mutual equal elements:

```
>> listlib::singleMerge([1, 1, 3, 3], [2, 2, 4, 4])

                   [1, 1, 2, 2, 3, 3, 4, 4]
```

Example 3. The following example shows, how expressions can be ordered by a user defined priority. This order is given by the function named `priority`, which returns a smaller number, when the expression has a type with higher priority:

```
>> priority := X -> contains(["_power", "_mult", "_plus"], type(X)):
   priority(x^2), priority(x + 2)

                             1, 3
```

The function `sortfunc` returns `TRUE`, if the both given arguments are in the right order, i.e., the first argument has a higher (or equal) priority than the second argument:

```
>> sortfunc := (X, Y) -> bool(priority(Y) > priority(X)):
   sortfunc(x^2, x + 2), sortfunc(x + 2, x*2)

                          TRUE, FALSE
```

Now the both lists are merged with regard to the given priority:

```
>> listlib::singleMerge([x^y, x*2, -y], [x^2, x*y, x + y], sortfunc)

                   y   2
                 [x , x , 2 x, -y, x y, x + y]

>> delete priority, sortfunc:
```

Changes:

- `listlib::singleMerge` used to be `listtools::singleMerge`.

`listlib::sublist` – search sublists

`listlib::sublist(list1, list2)` determines, whether the list `list1` contains another list `list2`.

Call(s):

- `listlib::sublist(list1, list2 <, index> <, option>)`

Parameters:

`list1, list2` — MuPAD list
`index` — integer, that determines the first search position
`option` — option *`Consecutive`*

Options:

`Consecutive` — determines that the sublist `list2` is containing coherent in `list1`

Return Value: the position of the first element of the containing sublist or zero

Related Functions: `contains`, `op`

Details:

- With `listlib::sublist` the position of the first appearance of a list in another list can be determined.
- The position that was found is returned as integer. If the given list does not contain the given `sublist`, the number `0` is returned.
- If an index is given, the search starts at this position. Therewith multiple occurence of a sublist can be determined.
- With the option *`Consecutive`*, the list must contain the sublist in one piece without elements in between.

Example 1. The sublist is a part of the list, but not in one piece:

```
>> listlib::sublist([1, 2, 3, 4, 5, 6, 7, 8], [2, 3, 5, 6])

                                    2

>> listlib::sublist([1, 2, 3, 4, 5, 6, 7, 8], [2, 3, 5, 6], Consecutive)

                                    0
```

The list contains the sublist, coherent and incoherent:

```
>> listlib::sublist([1, 2, 3, 4, 5, 1, 3, 5], [1, 3, 5])

                                    1

>> listlib::sublist([1, 2, 3, 4, 5, 1, 3, 5], [1, 3, 5], Consecutive)

                                    6
```

Example 2. Find the last occurence of the sublist inside of the list:

```
>> POS:= 0:
   while listlib::sublist([1, 2, 3, 1, 3, 1, 2, 3], [1, 2, 3], POS + 1) > 0
     POS:= listlib::sublist([1, 2, 3, 1, 3, 1, 2, 3], [1, 2, 3], POS + 1)
   end_while:
   POS

                                6

>> delete POS:
```

Changes:

- `listlib::sublist` used to be `listtools::sublist`.