current_arithmetic_function(*?Head*)
Successively unifies all arithmetic functions that are visible from the context module with *Head*.

4.28 Built-in list operations

Most list operations are defined in the library `lists` described in section A.11. Some that are implemented with more low-level primitives are built-in and described here.

is_list(+*Term*)
True if *Term* is bound to the empty list (`[]`) or a term with functor '`.`' and arity 2 and the second argument is a list.[42] This predicate acts as if defined by the definition below on *acyclic* terms. The implementation is *fails* safely if *Term* represents a cyclic list.

```
is_list(X) :-
        var(X), !,
        fail.
is_list([]).
is_list([_|T]) :-
        is_list(T).
```

memberchk(*?Elem, +List*)
Equivalent to `member/2`, but leaves no choice point.

length(*?List, ?Int*)
True if *Int* represents the number of elements of list *List*. Can be used to create a list holding only variables.

sort(+*List, -Sorted*)
True if *Sorted* can be unified with a list holding the elements of *List*, sorted to the standard order of terms (see section 4.6). Duplicates are removed. The implementation is in C, using *natural merge sort*.[43] The `sort/2` predicate can sort a cyclic list, returning a non-cyclic version with the same elements.

msort(+*List, -Sorted*)
Equivalent to `sort/2`, but does not remove duplicates. Fails with a `type_error` if *List* is not a list or a cyclic list.

keysort(+*List, -Sorted*)
List is a proper list whose elements are *Key−Value*, that is, terms whose principal functor is (-)/2, whose first argument is the sorting key, and whose second argument is the satellite data to be carried along with the key. `keysort/2` sorts *List* like `msort/2`, but only compares the keys. It is used to sort terms not on standard order, but on any criterion that can be expressed on

[42] In versions before 5.0.1, `is_list/1` just checked for `[]` or `[_|_]` and `proper_list/1` had the role of the current `is_list/1`. The current definition is conform the de-facto standard. Assuming proper coding standards, there should only be very few cases where a quick-and-dirty `is_list/1` is a good choice. Richard O'Keefe pointed at this issue.

[43] Contributed by Richard O'Keefe.

a multi-dimensional scale. Sorting on more than one criterion can be done using terms as keys, putting the first criterion as argument 1, the second as argument 2, etc. The order of multiple elements that have the same *Key* is not changed. The implementation is in C, using *natural merge sort*. Fails with a `type_error` if *List* is not a list or a cyclic list or one of the elements of *List* is not a *pair*.

predsort(+*Pred, +List, -Sorted*)
Sorts similar to `sort/2`, but determines the order of two terms by calling *Pred*(-*Delta*, +*E1*, +*E2*). This call must unify *Delta* with one of <, > or =. If built-in predicate `compare/3` is used, the result is the same as `sort/2`. See also `keysort/2`.[44]

merge(+*List1, +List2, -List3*)
List1 and *List2* are lists, sorted to the standard order of terms (see section 4.6). *List3* will be unified with an ordered list holding both the elements of *List1* and *List2*. Duplicates are **not** removed.

merge_set(+*Set1, +Set2, -Set3*)
Set1 and *Set2* are lists without duplicates, sorted to the standard order of terms. *Set3* is unified with an ordered list without duplicates holding the union of the elements of *Set1* and *Set2*.

4.29 Finding all Solutions to a Goal

findall(+*Template, :Goal, -Bag*) *[ISO]*
Creates a list of the instantiations *Template* gets successively on backtracking over *Goal* and unifies the result with *Bag*. Succeeds with an empty list if *Goal* has no solutions. `findall/3` is equivalent to `bagof/3` with all free variables bound with the existential operator (`^`), except that `bagof/3` fails when goal has no solutions.

findall(+*Template, :Goal, -Bag, +Tail*)
As `findall/3`, but returns the result as the difference-list *Bag-Tail*. The 3-argument version is defined as

```
findall(Templ, Goal, Bag) :-
        findall(Templ, Goal, Bag, [])
```

bagof(+*Template, :Goal, -Bag*) *[ISO]*
Unify *Bag* with the alternatives of *Template*, if *Goal* has free variables besides the one sharing with *Template* bagof will backtrack over the alternatives of these free variables, unifying *Bag* with the corresponding alternatives of *Template*. The construct +*Var*^*Goal* tells bagof not to bind *Var* in *Goal*. `bagof/3` fails if *Goal* has no solutions.

The example below illustrates `bagof/3` and the `^` operator. The variable bindings are printed together on one line to save paper.

```
2 ?- listing(foo).
```

[44] Please note that the semantics have changed between 3.1.1 and 3.1.2

reset_gensym
Reset gensym for all registered keys. This predicate is available for compatibility only. New code is strongly advice to avoid the use of reset_gensym or at least to reset only the keys used by your program to avoid unexpected site-effects on other components.

A.11 lists: List Manipulation

This library provides commonly accepted basic predicates for list manipulation in the Prolog community. Some additional list manipulations are built-in. Their description is in section 4.28.

append(*?List1, ?List2, ?List3*)
Succeeds when *List3* unifies with the concatenation of *List1* and *List2*. The predicate can be used with any instantiation pattern (even three variables).

append(*?ListOfLists, ?List*)
Concatenate a list of lists. Is true if *Lists* is a list of lists, and *List* is the concatenation of these lists. *ListOfLists* must be a list of -possibly- partial lists.

member(*?Elem, ?List*)
Succeeds when *Elem* can be unified with one of the members of *List*. The predicate can be used with any instantiation pattern.

nextto(*?X, ?Y, ?List*)
Succeeds when *Y* immediatly follows *X* in *List*.

delete(+*List1, ?Elem, ?List2*)
Delete all members of *List1* that simultaneously unify with *Elem* and unify the result with *List2*.

select(*?Elem, ?List, ?Rest*)
Select *Elem* from *List* leaving *Rest*. It behaves as `member/2`, returning the remaining elements in *Rest*. Note that besides selecting elements from a list, it can also be used to insert elements.

nth0(*?Index, ?List, ?Elem*)
Succeeds when the *Index*-th element of *List* unifies with *Elem*. Counting starts at 0.

nth1(*?Index, ?List, ?Elem*)
Succeeds when the *Index*-th element of *List* unifies with *Elem*. Counting starts at 1.

last(*?List, ?Elem*)
Succeeds if *Elem* unifies with the last element of *List*. If *List* is a proper list `last/2` is deterministic. If *List* has an unbound tail, backtracking will cause *List* to grow.[2]

reverse(+*List1, -List2*)
Reverse the order of the elements in *List1* and unify the result with the elements of *List2*.

permutation(*?List1, ?List2*)
Permuation is true when *List1* is a permutation of *List2*. The implementation can solve for *List2* given *List1* or *List1* given *List2*, or even enumerate *List1* and *List2* together.

[2] The argument order of this predicate was changed in 5.1.12 for compatibility reasons.

flatten(+*List1, -List2*)
Transform *List1*, possibly holding lists as elements into a 'flat' list by replacing each list with its elements (recursively). Unify the resulting flat list with *List2*. Example:

```
?- flatten([a, [b, [c, d], e]], X).

X = [a, b, c, d, e]
```

sumlist(+*List, -Sum*)
Unify *Sum* to the result of adding all elements in *List*. *List* must be a proper list holding numbers. See `number/1` and `is/2`. for details on arithmetic.

max_list(+*List, -Max*)
True if *Max* is the largest number in *List*. See also the function `max/2`.

min_list(+*List, -Min*)
True if *Min* is the smallest number in *List*. See also the function `min/2`.

numlist(+*Low, +High, -List*)
If *Low* and *High* are integers with $Low \leq High$, unify *List* to a list $[Low, Low + 1, \ldots High]$. See also `between/3`.

A.11.1 Set Manipulation

The set predicates listed in this section work on ordinary unsorted lists. Note that this makes many of the operations order N^2. For larger sets consider the use of ordered sets as implemented by library `ordsets.pl`, running most these operations in order N. See section A.15.

is_set(+*Set*)
Succeeds if *Set* is a list (see `is_list/1`) without duplicates.

list_to_set(+*List, -Set*)
Unifies *Set* with a list holding the same elements as *List* in the same order. If *list* contains duplicates, only the first is retained. See also `sort/2`. Example:

```
?- list_to_set([a,b,a], X)

X = [a,b]
```

intersection(+*Set1, +Set2, -Set3*)
Succeeds if *Set3* unifies with the intersection of *Set1* and *Set2*. *Set1* and *Set2* are lists without duplicates. They need not be ordered.

subtract(+*Set, +Delete, -Result*)
Delete all elements of set 'Delete' from 'Set' and unify the resulting set with 'Result'.

union(+*Set1, +Set2, -Set3*)
Succeeds if *Set3* unifies with the union of *Set1* and *Set2*. *Set1* and *Set2* are lists without duplicates. They need not be ordered.

subset(+*Subset, +Set*)
Succeeds if all elements of *Subset* are elements of *Set* as well.